Department of Computing

Empirical Investigations Supporting an Extensible, Theoretical Approach to Understanding Software Inspections

David James Arthur Cooper

This thesis is presented for the Degree of
Doctor of Philosophy
of
Curtin University of Technology

June 2010

Declaration

To the best of my knowledge and belief this thesis contains no material previously published by any other person except where due acknowledgment has been made. This thesis contains no material which has been accepted for the award of any other degree or diploma in any university.

David J. A. Cooper   Date

Abstract

Empirical software engineering research has directed substantial effort towards understanding and improving software inspection, a defect detection method much less costly than testing. However, software inspection suffers from a lack of theory governing the process and its outcomes, leading to apparently contradictory experimental outcomes that cannot easily be reconciled. This theoretical uncertainty hinders efforts to effectively address *delocalisation* — the occurrence of related information in different artefacts, or parts of a software system. Delocalisation is a hurdle to software comprehension, an activity fundamental to inspection.

A gap currently exists between the development of inspection strategies and theories of software comprehension, manifested in two ways. First, although some strategies seek to enhance an inspector's understanding of key parts of the software, they generally ignore variability between inspectors. A particular form of guidance or cognitive support given to one inspector may have a different effect when given to another. Second, while models of inspection cost effectiveness exist, they are not expressed in terms of factors that might be manipulated to improve inspection performance. It is not clear how far an inspector should go to address one particular concern in the software, before the benefits of doing so are outweighed by the risk of ignoring other concerns.

This thesis first reports on an industry survey examining the current state of practice with respect to peer reviews. Two more qualitative studies were conducted to explore approaches inspectors might take to the comprehension of artefact interrelationships and the challenges posed by delocalisation. A controlled experiment is then presented to show how active guidance and inspector expertise affect the detection of individual defects.

Using the results of these studies, a theoretical framework and model of inspection cost effectiveness are proposed in which the effects of experience, cognitive support and the reading technique can be used to predict the consequences of a given inspection strategy. A simulation of the model was conducted to compare several new and existing inspection strategies. Thus, the framework and model provide a basis upon which an

appropriate inspection strategy can be developed, selected or refined for a given software project.

The results of these investigations suggest several ways in which inspection practices might be improved, including through the additional use of tool support and selective use of active guidance under specific conditions. By instantiating and using the proposed inspection model, software development organisations can engineer optimally cost effective inspection strategies.

Acknowledgements

In the course of this work, I am fortunate to have been surrounded by colleagues, friends and family who have all, one way or another, helped me through.

I must of course thank my supervisors — Brian, Nihal, Mike and Jim — who have each played a valuable role in steering me towards the finish. Brian initially encouraged me to sign on to the PhD journey. I had no idea what to expect, but the prospect of research held a strange and mysterious appeal. Everyone else thought I was stark raving mad. In large part they were (and are) correct, but then what is life without a bit of raving insanity?

Brian and I had somewhat different modes of operation. My approach to research was to treat the thesis itself not just as an output but as a non-volatile storage device. I would generally articulate my ideas as precisely as I could on paper (or in LaTeX) and then promptly forget either the fine details or the overall rationale. After all, they'd been safely written down rather than left to the whims of my volatile grey matter. My brain was more of an indexing system than a repository of knowledge. Meanwhile, Brian's approach to supervision was to constantly challenge me to defend these ideas. This of course is a perfectly valid and helpful thing for a supervisor to be doing, but was not facilitated by my having forgotten what said ideas actually were.

My family has been quietly patient and supportive throughout, despite some initial worries that I'd be liable for many tens of thousands of dollars in fees (which thankfully was not the case).

My dear Elaine has given me her love and companionship, and has stood by me in the face of delays and uncertainties. She herself has worked exceptionally hard throughout her life, and through inspiration and support she helped me over the last few hurdles.

I greatly value the assistance and friendship of Dave M — the only other strictly software engineering postgraduate student with whom I had regular contact. I couldn't have asked for a more generous, hard working colleague.

That said, I would not diminish the importance of my other fellow postgraduate students and researchers in the Computing and Maths departments. In no particular order, Steve, Saha, Monica, Alex, Graeme, Tiffany, Siewfang and many more all brought humour and humanity to the research environment. I've learnt a great deal from them, in spite of the gap between their research topics and mine. Patrick's informal and unpaid maths seminar series inspired sections of my research.

Finally, I am grateful to all the anonymous participants in my various empirical investigations for taking the time (sometimes for no direct compensation at all) to help me in my efforts. My research would have been exceedingly difficult in the absence of willing volunteers, for whom the principal motivation was the advancement of science and research.

Contents

List of Figures

List of Tables

Glossary

active guidance

Any instance in which the inspector is instructed to find a particular type of information (including a defect), or to search a particular artefact or part thereof. (See Chapter 2, Section 2.2.2.)

artefact

"A physical piece of information that is used or produced by a software development process" (Object Management Group, 2003). These typically include (but are not limited to) requirements documents, design diagrams and source code.

artefact diversity

The number of different types of artefacts used in a given phase. (See Chapter 4, Section 4.2.)

artefact prevalence

The average proportion of a project (by workload) in which a given artefact is used. (See Chapter 4, Section 4.2.)

auxiliary defect

In the checklist experiment, a defect to which the checklist or inspector expertise does not apply, in contrast to a primary defect. (See Chapter 7, Section 7.1.)

Bayesian network (BN)

A graph representing the dependency structure between a series of random variables. (See Chapter 3, Section 3.3.3.)

binding code

A code fragment that helps implement a state transition by directing the flow of control between decision code, mutation code and/or other binding code. (See Chapter 5, Section 5.1.5.)

checklist

Any list of questions directing inspectors to search for particular defect types. (See Chapter 2, Section 2.2.1.)

coding

In the context of empirical data collection, the act of manually categorising free-form text (or any other non-numeric, non-categorical data) according to a particular scheme, to facilitate either qualitative or quantitative analysis. (See Chapter 3, Section 3.2.2.)

coding scheme

The method and categories used to perform coding of empirical data.

cognitive support

Any inspection aid(s) intended to improve the effectiveness and/or efficiency of the comprehension process.

Cohen's kappa

A measure of inter-coder agreement, used to ensure the reliability of the coding process (for empirical data collection).

compact Bayesian network (CBN)

A graph representing a set of possible Bayesian networks having particular categories of random variables, with particular constraints on the dependencies occurring between them. (See Chapter 8, Section 8.2.4.)

comprehension

The act (by a software developer) of acquiring information about the structure and purpose of parts of a software system by reading and traversing the artefacts that describe it.

comprehension dependency

A dependency between two k-instances, such that the comprehension of one improves the probability of comprehending the other. (See Chapter 8, Section 8.1.2.)

comprehension model

Part of the scenario model (within the overall inspection model) describing the comprehension process within an inspection. (See Chapter 8, Section 8.1.2.)

cost effectiveness

Any of several measures of inspection performance derived from both (a) the costs associated with inspection, and (b) hypothetical costs potentially incurred without inspection. (See Chapter 2, Section 2.4.1.)

Cox proportional hazards model

In survival analysis, a model indicating the multiplicative effects of a series of factors on an unspecified hazard function. (See Chapter 3, Section 3.3.2.)

decision code

A code fragment that determines whether a state transition will take place immediately. (See Chapter 5, Section 5.1.5.)

decision dependency

A dependency between a locality and a k-instance, such that comprehension of the k-instance improves the probability of searching the locality. (See Chapter 8, Section 8.1.2.)

decision time

In the scenario study, the time spent contemplating whether a particular line of source code takes part in the use case scenario.

defect

"An instance in which a requirement is not satisfied" (Fagan, 1986). (A "requirement" here can be broadly interpreted as any ultimately-necessary quality or feature of the system or its artefacts, whether explicitly stated or implied. A requirements document itself may contain defects.)

defect type

Any category under which a defect might be classified, under a given classification scheme.

delocalised plan

An instance in which conceptually related information is distributed among multiple artefacts, or different parts thereof. (See Chapter 2, Section 2.3.3.)

entity

A k-instance or locality; one of the units into which the framework breaks down a system. (See Chapter 8, Section 8.1.1.)

entity type

A knowledge type or locality type; any category under which an entity might be classified. (See Chapter 8, Section 8.1.1.)

experience

The length of time spent working as a software developer, and the types of work undertaken.

expertise

The combined knowledge and skill set of a software developer/inspector, acquired through education, training and experience.

failure

An instance in which a given system does not perform as required, due to the presence of one or more defects.

hazard function

In survival analysis, a function $\mu(t)$ of time, proportional to the probability that some event occurs in a small window of time around t, given that it has not occurred before t (Harrell, 2001).

hierarchy

In the theoretical framework, the notion that some entities are subordinate to other entities within the same phase. (See Chapter 8, Section 8.1.5.)

inspection

Any of several structured peer review processes whose principal goal is the detection of defects in software artefacts. (Other terms, particularly "reviews", are also sometimes used in reference to such activities. See Chapter 2, Section 2.1.)

inspection performance

The relative level of success of an inspection, as measured by any of several metrics.

inspection strategy

Any overall strategy for the use of peer reviews in a software project, including the selection of appropriate reading techniques and the broader structure of the peer review process. (See Chapter 8, Section 8.1.6.)

k-instance

A knowledge instance (piece of knowledge) embedded within the artefacts describing a system; possibly a defect. (See Chapter 8, Section 8.1.1.)

knowledge type

Any category under which a k-instance might be classified; possibly a defect type. (See Chapter 8, Section 8.1.1.)

locality

A physical location within the artefacts describing the system; a type of entity. (See Chapter 8, Section 8.1.1.)

locality dependency

A dependency between a k-instance and a locality, such that searching the locality improves the probability of comprehending the k-instance. (See Chapter 8, Section 8.1.2.)

locality type

A type of artefact or structural component thereof; any category under which a locality might be classified. (See Chapter 8, Section 8.1.1.)

log odds

The log of the odds ratio; a way of expressing probability (particularly in logistic models) using the entire set of real numbers, rather than just the range $[0, 1]$. The log odds of some event E is defined as $\text{logit}(E) = \log\left(\dfrac{\mathbb{P}(E)}{1 - \mathbb{P}(E)}\right)$, where $\mathbb{P}(E)$ is the probability of E.

log-linear model

A model in which a series of factors have a multiplicative (rather than additive) effect on the response variable. (See Chapter 3, Section 3.3.1.)

logistic model

A model in which a series of factors determine the log odds (and hence probability) of a binary response variable being 1. (See Chapter 3, Section 3.3.1.)

macro-strategy

The starting point and overall direction of the comprehension process and the nature of the mental model it constructs. (As with *micro-strategy*, this is a broader interpretation of the term than was originally used by Soloway et al. (1988).)

marker

A flag attached to an entity declaring it to have a particular property (such as complexity or importance), independent of its type. (See Chapter 8, Section 8.1.3.)

mental model

The mental representation of a piece of software, constructed internally by an inspector (or maintainer, etc.) when reading the software.

metamodel

Part of the inspection model describing the system structure in terms of entities and dependencies, on which the scenario model is based. (See Chapter 8, Section 8.2.1.)

micro-strategy

The manner by which individual, discrete pieces of knowledge are acquired within the comprehension process. (As with *macro-strategy*, this is a broader interpretation of the term than was originally used by Soloway et al. (1988).)

mutation code

A code fragment that alters the necessary state variables, as required in the transition from one state to another. (See Chapter 5, Section 5.1.5.)

passive guidance

Any form of cognitive support that is purely passive, providing additional information or visualisations of the system, but not specific hints on how to traverse it. (See Chapter 8, Section 8.1.6.)

peer review

Any formal or informal activity in which software artefacts are reviewed by at least one software developer, other than the original author, for the purposes of quality assurance and defect detection.

phase

In the theoretical framework, any of several arbitrary (and possibly uneven) time units into which a project's timeline can be broken down (see Chapter 8, Section 8.1.4).

phase group

In the industry survey, one of six aggregated sets of the various software development phases listed by respondents.

plan

A form of knowledge, acquired by software developers through experience, representing a generic solution to a commonly-occurring situation. (See Chapter 2, Section 2.3.2.)

point of interest

In the scenario study, a point in a participant's working at which the participant makes an error, changes their mind and/or takes longer than normal to make a decision. (See Chapter 6, Section 6.1.4.)

primary defect

In the checklist experiment, a defect to which the checklist or inspector experience directly applies, in contrast to an auxiliary defect. (See Chapter 7, Section 7.1.)

prioritisation

The ordering of parts or aspects of a system by importance (according to some criteria), so that the most important parts/aspects are inspected first. (See Chapter 2, Section 2.2.3.)

propagation

The generation (and multiplication) of entities in one phase from those that existed in the previous phase. (See Chapter 8, Section 8.1.5.)

protocol analysis

A qualitative data collection and analysis technique, whereby subjects verbalise their thoughts (i.e. *think aloud*) while performing a given task. These verbalisations are then transcribed and coded. (See Chapter 3, Section 3.2.1.)

proximity code

In the scenario study, one of a set of codes assigned to verbal references, broadly indicating the proximity of the reference to the current point in the model solution.

reading technique

Any technique intended to assist peer review, usually involving the provision of active guidance (e.g. a scenario), but also (conceivably) passive guidance. (See Chapter 2, Section 2.2.)

reference model

An instantiation of the generic inspection model, designed to be specific to a particular, commonly-occurring software development scenario.

reviews-by-artefact

The proportion of organisations using a given artefact that review it. (See Chapter 4, Section 4.2.)

reviews-by-phase

The proportion of artefacts used in a given phase that are reviewed. (See Chapter 4, Section 4.2.)

scenario

A list of instructions (more detailed than a checklist) specifying how inspectors should traverse the artefacts under inspection and what types of information should be sought out. Scenarios form the basis for a number of reading techniques. (See Chapter 2, Section 2.2.2.)

scenario model

Part of the inspection model concerned with the mechanics of inspection and the propagation of defects, comprising the comprehension and verification models and founded on the metamodel. (See Chapter 8, Section 8.2.)

separation of concerns

The notion of having inspectors focus on different parts or aspects of the software under inspection, with the intention of reducing overlap in the types of defects detected. (See Chapter 2, Section 2.2.2.)

solution index

In the scenario study, an integer index representing a point in the model solution, to which one or more parts of each participant's own solution have been mapped. (See Chapter 6, Section 6.1.4.)

survival analysis

A type of statistical analysis concerned with modelling the time at which some event occurs, given data in which the event is not always observed to occur. (See Chapter 3, Section 3.3.2.)

testing

Any defect detection activity that involves running (or simulating) the software or parts thereof to compare its actual behaviour to the expected behaviour.

verification model

Part of the scenario model (within the overall inspection model) describing the interactions and consequences of various defect detection activities taking place across project phases. (See Chapter 8, Section 8.1.2.)

Chapter 1

Introduction

"Baldrick, that is by far and away without a shadow of doubt the worst and most contemptible plan in the history of the universe."

— Blackadder the Third

The software engineering industry encompasses a diversity of people, methods and tools. This places a responsibility on researchers to consider a broad range of circumstances in which new software development technologies may be used. A given technology can only be successful to the extent that it is applicable to, and compatible with, the processes and environments used by software development organisations.

Software *peer reviews* are a widely used and widely varying defect detection approach employed by organisations with different development methodologies, during different project phases for different artefacts (Brykczynski, 1999, Harjumaa et al., 2005, Hedberg and Iisakka, 2006, Rigby et al., 2008). Software *inspection* is a formalised peer review process applicable to any software artefact, including requirements specifications, design diagrams and source code (Fagan, 1976). Failure to detect defects, and quality deficiencies generally, can be costly. Defects left undetected can be many times more expensive to fix in later stages of a software project (Boehm and Basili, 2001). Defects that survive until release or deployment may result in substantial costs to users, including lost productivity, data loss, security vulnerabilities or even physical harm (Leveson, 1986, National Institute of Standards and Technology, 2002).

Though testing is also essential for defect detection, inspection is typically regarded as the cheaper of the two complementary approaches (Ackerman et al., 1989, Basili, 1997). Early design artefacts containing only an imprecise sketch of the envisioned system cannot be subject to comprehensive testing, if any testing at all, because by

definition the logic has not yet been fully specified. By contrast, inspection can be applied in any phase to any artefact; the system need not be complete or executable.

To inspect a set of artefacts effectively, an inspector must be able to understand them. The difficulty of understanding software depends on the nature of the artefacts, and in particular upon the complexity of the relationships within and between them. *Delocalised plans* occur where related information is dispersed across different artefacts, or across parts of an artefact. To piece together this information, the inspector must actively seek it out from its various sources (Soloway et al., 1988). Approaches exist for addressing delocalisation. For instance, an inspector can be encouraged or prompted to switch between related artefacts (Kim et al., 2000), or might be presented with an elaborated or alternate artefact, possibly tool-generated, that expresses some of the information in a localised form (Storey et al., 1997).

Such approaches are sometimes manifested in inspection reading techniques (Dunsmore et al., 2003). Controlled experiments have demonstrated that different reading techniques can have a significant and substantial effect on inspection efficacy and efficiency (Porter et al., 1995, Thelin et al., 2003). However, the inspection research community currently lacks a cohesive theory with which such techniques may be supported (Porter and Votta, 1997, Jeffery and Scott, 2002, Hannay et al., 2007). As a result, experimentation in this area has been based on isolated, *ad hoc* hypotheses. With the factors influencing them not well articulated, replications of these experiments often fail to reproduce the same results, for unidentifiable reasons (Regnell et al., 2000).

Reading techniques and other aids designed to address delocalised plans may not have a universally positive effect on inspections. In the design or selection of an inspection strategy, the general goal is to maximise the number of defects found and their importance (i.e. the potential cost if left undetected) given limited resources. Therefore, the challenge is not merely one of improving comprehension, but of balancing the benefits of comprehension with the costs of achieving it. However, there is not yet a theoretical basis upon which such tradeoffs can be made, nor are the factors involved well defined. Given an arbitrary set of circumstances, it is not clear how best to adapt an inspection strategy.

1.1 Research Questions

The following overarching research question is posed in response to the lack of consensus opinion on the use of software inspections: *How can the uncertainties of software inspection be resolved, in order to make recommendations of best practice?*

This broad objective is broken down into three sub-questions as follows:

1. What are the prevalent inspection practices, and in what contexts do they occur?

2. What are the challenges inherent in comprehending a system under inspection?

3. To what extent does active guidance support defect detection, and what are the effects on overall cost effectiveness?

1.2 Contribution

This thesis answers the preceding research questions and contributes to software engineering research through four inspection-related empirical studies and a proposed theoretical model of the inspection process.

1.2.1 Identifying Industry Practices

Research question 1: What are the prevalent inspection practices, and in what contexts do they occur?

The first study was a survey of the peer-review practices at 31 software development organisations across Australia. The survey sought to determine the types of artefacts actually used in industry, the extent and circumstances of their use, and the potential for inspection strategies to reduce costs. The survey results indicate that:

- software peer review occurs across a broad range of domains, phases and artefact types;

- most organisations appear to realise the benefits of peer review, though few to none have quantitative data to support this view;

- a substantial proportion of total project workload is typically expended in testing/QA, support and maintenance phases, suggesting that opportunities exist for detecting defects earlier through inspection;

- the simultaneous use of multiple, varying artefact types is common;

- as a project progresses from the requirements phase to development, the number of distinct artefact types in use more than doubles;

- about half of surveyed organisations develop standardised requirements documents;

- checklists and use case scenarios are each used by less than half, but nonetheless a sizeable minority, of surveyed organisations in their peer-review activities; and

- visualisation tools are used by roughly a quarter of surveyed organisations.

The survey identified commonly-used notations that can serve as a focus for future inspection research and data collection.

1.2.2 Comprehension Challenges

Research question 2: *What are the challenges inherent in comprehending a system under inspection?*

To answer the second question, two more studies were undertaken to examine qualitatively the effects and challenges of complex artefact interrelationships. In the *statechart* study, 28 participants were asked to map the transitions of a UML statechart to corresponding segments of source code. Participants reported large numbers of both false positive and false negatives, and incorrectly identified one-to-one mappings.

The *scenario* study explores comprehension challenges encountered in scenario-based reading techniques. Here, ten participants were asked to trace the events depicted in a UML sequence diagram through corresponding Java source code. Tracing use case scenario events through other artefacts is a task essential to Usage-Based Reading (Thelin et al., 2003) and another similar technique (Dunsmore et al., 2003), though the actual instructions and artefacts vary. Participants were also asked to verbalise their thoughts, and their actions were recorded by software. Together, these were used as a fine-grained indicator of the line of code under consideration at any given time. A range of comprehension issues were identified, arising in particular from delocalisation within the source code. Moreover, participants showed awareness of and consideration for markedly different parts of the system, despite the rigid and relatively unambiguous nature of the task.

The results of the statechart and scenario studies illustrate the difficulty of establishing complex artefact interrelationships, even when programmers are explicitly asked to find them. This suggests that cognitive support has a substantial role to play in improving comprehension and thus inspection performance. However, the inspector variability

noted in the scenario study suggests that inspection strategies in general should take account of the expertise of individual inspectors.

1.2.3 Active Guidance Effects

Research question 3: *To what extent does active guidance support defect detection, and what are the effects on overall cost effectiveness?*

To help answer the final question, the *checklist* experiment examined inspection checklists, inspector expertise and their interactions. The experiment involved 42 participants, each of whom were asked to find defects in several code snippets in a two-factor cross-over experimental design. The experiment sought to determine the effects of checklists and prior exposure to specific defect types on the probability and time of individual defect detection. Checklists and prior exposure were designed to assist the detection of some seeded defects but not others. The results show that checklists has a significant positive effect on the probability of defect detection, provided the defect is covered by the checklist. Conversely, there were significant negative effects if the defect was not covered. Prior exposure had suggestive but non-significant effects.

The checklist experimental results indicate that the comparative effectiveness of different reading techniques depends on both the system under inspection and the inspectors themselves. Instructions to the inspector (including checklist questions) can help improve defect detection, but may also hinder it under certain circumstances. That is, the choice of reading technique should be motivated by different factors; there is no singular best choice. Then, by what criteria should a reading technique be selected?

Further, the statechart and scenario studies suggest that cognitive support may improve comprehension in the case of artefact interrelationships. Would this have a net benefit, when combined with checklists or scenario-style reading techniques?

These studies and the questions they raise, along with the second initial research question, motivate the development of inspection theory — a theoretical basis for understanding software inspection, as previously called for in the published literature (Jeffery and Scott, 2002, Hannay et al., 2007). In order to provide a basis for the development, refinement and selection of appropriate reading techniques, inspection theory should incorporate and articulate the following elements:

- the composition of reading techniques, so that different techniques can be distinguished in a non-arbitrary fashion;

- the mechanics of the comprehension process, including the effects of reading technique instructions and cognitive support;

- system composition and variability;

- inspector variability; and

- inspection-related costs, to provide a measure by which all inspection strategies can be compared.

This thesis proposes a theoretical framework and model fulfilling these requirements. A software simulator of the model was constructed and run to compare inspection strategies and explore the model's behaviour with respect to the comprehension process.

The model parameters were based on estimates, but the simulation results broadly agree with the empirical studies. From simulation results, checklist-based reading appears generally more effective than *ad hoc*, but potentially less effective under particular conditions. System size and inspector characteristics did alter the optimal choice of inspection strategy. Finally, the combination of active guidance and cognitive support appeared to be the most resilient of any strategy to a high delocalisation effect (that is, when comprehension and defect detection is highly dependent on understanding interrelationships), though it was not generally the optimal strategy.

The framework and model apply in principle to any software development situation. This flexibility does not arise from the revelation of fundamental laws of software engineering, but from abstraction and descriptive consistency. Nevertheless, the model is both predictive and explanatory, able to compare inspection strategies by modelling their actual effects on inspector behaviour and the wider development process.

1.3   Ethical Research Conduct

The empirical studies discussed in this thesis were approved by the Curtin University Human Research Ethics Committee. All studies had the following characteristics:

- They were minimal risk, in the sense that participation carried no risks greater than those encountered in everyday life.

- Participation was entirely voluntary. Where students were invited to participate, it was explained to them that their choice would have no effect on their ability

Table 1.1: Outline of the remaining chapters.

Chapter	Description	Research question
2	Reviews previous inspection-related research, illustrating the need and basis for the development of inspection theory.	—
3	Reviews some of the empirical and statistical methods used in this thesis.	—
4	Presents the industry survey, to determine how inspections are used in industry.	1 (Prevalent inspection practices)
5	Examines artefact interrelationships via the statechart study.	2 (Comprehension challenges)
6	Examines comprehension issues resulting from use case traversal in the scenario study.	
7	Details the checklist experiment, examining the effects of checklists and expertise on individual defect detection.	3 (Active guidance effects)
8	Articulates the theoretical framework, model and simulation results thus obtained.	
9	Discusses the overall research findings and makes recommendations of best practice.	1–3

to succeed in their courses. It was further explained to all participants that they could withdraw at any time without explanation or adverse consequences.

In two of four empirical investigations, no data identifying the individuals involved was collected or stored. In the two remaining investigations, participants' contact details were collected to facilitate possible follow-up questions. These were kept confidential.

The information sheets given to participants/respondents in the different studies are shown in the appendices related to those studies:

- Appendix A (for the industry survey);
- Appendix B, Section B.1 (for the statechart study);
- Appendix C, Section C.1 (for the scenario study); and
- Appendix D, Section D.1 (for the checklist experiment).

1.4 Outline

The remainder of this thesis is organised as shown in Table 1.1.

Chapters 2 and 3 introduce prior inspection-related research and methodological concepts. Chapters 4–7 discuss the four studies forming the empirical component of this research, while Chapter 8 presents the theoretical contribution — the inspection framework and model. (Chapters 4–8 are each supplemented by an appendix in A–E.)

Chapter 9 concludes by drawing together the findings of previous chapters and making recommendations with respect to the use of visualisation tools, active guidance and the proposed inspection model.

Chapter 2

Software Inspection Background

> *"You've burnt the life's work of England's foremost man of letters?"*
> *"Yup. You did say burn any old rubbish."*

> — *Blackadder the Third*

Software inspection is a formal process for detecting and reworking defects, applicable to any software artefact (Fagan, 1976, 1986). Industrial experience has long suggested that defects occurring early in a software project, if left undetected, can be many times more costly to repair later (Boehm and Basili, 2001). Inspection seeks to detect defects early, when they are more easily rectified.

Inspection-related research has focused on reading techniques as a means of achieving still greater defect detection. Numerous techniques have been suggested for guiding inspectors towards particular types of defects, and/or lending assistance in their detection. However, empirical studies of these techniques generally have not reported consistent results. This suggests that more complex factors govern inspection performance. Given this, by what means can we predict whether and to what extent a particular technique will lead to the successful detection of important defects?

Theories of software comprehension exist that explain qualitatively the process of understanding software. This is necessary for inspection just as it is for software maintenance. Such qualitative theories posit factors that may help or hinder the development of a programmers' mental model of the software, and by implication an inspector's ability to detect defects. With some exceptions, these theories have not been taken into account in the construction or evaluation of reading techniques. Unfortunately, they do not provide a means to predict inspection performance.

This chapter introduces software inspection, discussing developments in reading technique research, outlining concepts in software comprehension and describing the building blocks of software inspection theory.

2.1 Inspection

Software verification relies on two broad, complementary defect detection approaches: peer review and testing. Peer review entails a manual search through an artefact or set of artefacts for defects (or general quality deficiencies). This must be conducted by at least one person not responsible for having created the artefacts.

Though peer review cannot replace testing, it nonetheless has several important advantages:

- Peer review can more easily find defects whose effects are difficult to reproduce in a testing environment (Ackerman et al., 1989). Some defects result in frequent or obvious failures, while others may cause failures only on rare occasions under a complex set of conditions. Peer review seeks out defects directly without relying on observing their resulting failures.

- Peer review can find issues that do not directly result in failure at all, but may present problems for maintenance (Siy and Votta, 2001). (For the purposes of this thesis, such issues are also considered to be defects.)

- Testing finds defects largely on a one-by-one basis (because each test only has two outcomes: pass or fail), whereas any number of defects can be detected by a single peer review (Sommerville, 2001).

- Peer review can help detect defects in untestable artefacts — those that cannot (easily) be executed, such as requirements documents and design diagrams. The early detection of defects is important, because the cost of correcting defective requirements or design can multiply many times if left until the development phase or later (Boehm and Basili, 2001).

Software inspection is a formalised peer review process, proposed by Fagan (1976, 1986). The process relies on a team of inspectors, each participant having a defined role, and comprises six stages:

1. *Planning*, where inspectors, times and meeting places are arranged and the artefacts themselves are confirmed to be ready for inspection;

2. *Overview*, where the inspectors are assigned specific roles and meet to confer on the artefacts to be inspected;

3. *Preparation*, where inspectors separately read and familiarise themselves with the artefacts;

4. *Inspection*, where the inspectors meet as a group to find and record defects;

5. *Rework*, where the recorded defects are resolved; and

6. *Follow-up*, where the rework effort is reviewed, and a decision made as to whether another inspection should be scheduled (based on the amount of rework done).

The overview stage may be omitted if inspectors are already sufficiently well-informed of the artefacts to be inspected.

Fagan advises that most defects are found in the main inspection meeting, which is thus essential to the process. Votta (1993) argues otherwise; that such meetings consume considerable resources, while most defects are actually detected in the preparation phase. Thus, the defects detected in inspection meetings often do not justify the cost of the meetings. Porter and Johnson (1997) find that individual inspection finds substantially more defects than group-based inspection, but also generates more false positives. Laitenberger and DeBaud (2000) conduct a literature survey, reporting a range of opinion weighted towards the position that defect detection is principally an individual activity.

More recent studies have also recommended against inspection meetings, finding that they can actually reduce the number of defects detected (Bianchi et al., 2001, Halling and Biffl, 2002). Although meetings can result in additional defects being detected, they can also result in defects being lost, where the inspection team as a whole rejects defects found by individual inspectors.

As well as non-meeting-based inspections, several other variants of the inspection process have been proposed:

- Active Design Reviews, which seeks to make optimal use of inspector capabilities, distinguishing between the expertise of different inspectors and arranging for them to be more pro-actively involved (Parnas and Weiss, 1985);
- N-Fold Inspections, wherein several inspection teams operate in parallel, in order to improve overall defect detection (Martin and Tsai, 1990); and
- Phased Inspections, which divide an inspection into a sequence of mini-inspections — phases — each of which examines a different aspect of the system (Knight and Myers, 1993).

- Have all materials required for a requirements inspection been received?
- Are all materials in the proper physical format?
- Have all requirements standards been followed?
- Is the requirements document complete, i.e., does it implement all of the known customer needs?
- Does the human interface follow project standards?
- Has all the infrastructure been specified, i.e., backup, recovery, checkpoints, etc.?
- Are the error messages unique and meaningful?
- Have all reliability and performance objectives been listed?
- Have all security considerations been listed?
- Do the requirements consider all existing constraints?
- Do the requirements provide an adequate base for design?
- Are the requirements complete, correct, and unambiguous?

Figure 2.1: A requirements checklist, suggested by Ebenau and Strauss (1994).

2.2   Reading Techniques

The importance of the individual preparation stage of inspection has prompted the development of numerous *reading techniques*, designed to enhance inspectors' ability to detect defects. Reading techniques seek to focus inspectors' attention on specific parts or aspects of the software. To do so, they can employ several mechanisms, and appeal to different rationales. Unaided peer review — the absence of a reading technique — is simply *ad hoc* reading.

No definitive taxonomy of reading techniques exists, and this section does not propose one. Rather, it describes the development of the concepts on which published reading techniques are based.

2.2.1   Checklists

Checklists were proposed by Fagan as part of the original software inspection process. They have been a frequently recommended inspection aid, intended to focus attention on well-defined defect types (Ackerman et al., 1989, Gilb and Graham, 1993, Ebenau and Strauss, 1994). Since the development of other reading techniques, the use of checklists is now referred to as Checklist-Based Reading (CBR).

There are no universal rules on what checklists may contain, but generally they comprise

Table 2.1: Experiments assessing Checklist-Based Reading (CBR).

Experiment	Environment	Subjects	Artefacts	Outperforms *ad hoc*
Porter et al. (1995)	Academic	48	Requirements	No
Porter and Votta (1998)	Industrial	18	Requirements	No
Wohlin et al. (2002)	Industrial	203[a]	Req. & code	Yes
Hatton (2008)	Industrial	238	Code	No
Akinola and Osofisan (2009)	Academic	20	Code	No

[a] Acquired by aggregating previously-collected data; this figure is the sum of the number of inspectors in each data set.

a concise list of relatively straightforward but specific questions. Each question typically identifies a type of defect that may occur in the material under inspection (though some questions may also check that process requirements have been satisfied). As an example, Figure 2.1 shows the requirements document checklist suggested by Ebenau and Strauss (1994).

The frequent use and recommendation of CBR relies on an assumption that it outperforms *ad hoc* reading (with respect to defects detected, for instance). However, as shown in Table 2.1, experimental evidence supporting this assumption is limited.

Nevertheless, the failure of controlled experiments to show support for CBR may result from confounding factors. Anecdotal experience suggests that checklist effectiveness depends on their construction: size, relevance and specificity (Brykczynski, 1999). Checklists should generally not exceed one page, and should not contain overly-vague questions. In particular, checklists should be continually updated to maintain their relevance to the system under development (Gilb and Graham, 1993, Chernak, 1996).

Fagan (2002) no longer recommends the use of checklists, explaining that many checklist items represent relatively straightforward defect types. These can be detected by compilers and other automatic or semi-automatic verification tools (Cooper et al., 2004).

2.2.2 Scenarios

Reading scenarios provide more fine-grained guidance than do checklists (Porter et al., 1995). A scenario in itself is a set of instructions designed to focus the inspector's attention on specific types of information in the artefacts under inspection. This serves two purposes:

A. Data Type Consistency Scenario

1. Identify all data objects mentioned in the overview (e.g., hardware component, application variable, abbreviated term or function):

    (a) Are all data objects mentioned in the overview listed in the external interface section?

2. For each data object appearing in the external interface section determine the following information:
    - Object name:
    - Class: (e.g., input port, output port, application variable, abbreviated term, function)
    - Data type: (e.g., integer, time, Boolean, enumeration)
    - Acceptable values: Are there any constraints, ranges, limits for the values of this object?
    - Failure value: Does the object have a special failure value?
    - Units or rates:
    - Initial value:

    (a) Is the object's specification consistent with its description in the overview?

    (b) If object represents a physical quantity, are its units properly specified?

    (c) If the object's value is computed, can that computation generate a non-acceptable value?

3. For each functional requirement identify all data object references:

    (a) Do all data object references obey formatting conventions?

    (b) Are all data objects referenced in this requirement listed in the input or output sections?

    (c) Can any data object use be inconsistent with the data object's type, acceptable values, failure value, etc.?

    (d) Can any data object definition be inconsistent with the data object's type, acceptable values, failure value, etc.?

Figure 2.2: A reading scenario showing active guidance, developed by Porter et al. (1995). This is one of the three Defect-Based Reading (DBR) scenarios.

- *active guidance*, intended to improve individual inspector performance; and

- *separation of concerns*, intended to reduce overlap between the defects found by different inspectors, thus improving overall inspection team performance.

Active guidance is inherent in the use of reading scenarios, to varying extents. Through instructions to the inspector, it seeks to modify the inspector's approach to the task, by directing him/her to locate particular pieces of information or focus on particular parts of the system. Active guidance is sometimes contrasted against checklists (Denger et al., 2004), though both exist on the same spectrum. Checklists themselves might be considered a weak form of active guidance, while scenarios provide much more comprehensive forms.

Figure 2.2 shows an example of a reading scenario proposed by Porter et al. (1995), which necessarily entails active guidance. As shown, the inspector is given explicit instructions to identify data objects, functional requirements and associated information. The scenario also poses checklist-style questions based on this information.

Separation of concerns requires that different inspectors follow different scenarios. It relies on the existence of at least some minimal level of active guidance. Scenarios typically direct inspectors to examine different parts or aspects of the system. Where this successfully results in inspectors finding different defects, the overall performance of the inspection team will improve.

Techniques that use scenarios are given the umbrella term Scenario-Based Reading (SBR), of which there are several proposed variants. Basili et al. (1996) argue that reading techniques should be tailorable, and should vary based on the context. However, most research has focused on a relatively small set of named techniques.

Defect-Based Reading (DBR) is the original scenario-based technique proposed by Porter et al. (1995), part of which was shown in Figure 2.2. In DBR, scenarios are designed to guide inspectors in the detection of specific defect types, by asking them to locate and list certain pieces of information in which defects may be evident. Separation of concerns is achieved by dividing the defect types between different inspectors.

Perspective-Based Reading (PBR) was proposed by Basili et al. (1996). Here, separation of concerns is achieved by having inspectors examine artefacts from the perspective of different stakeholders; e.g. the user, developer and tester. Inspectors are each asked to create an artefact that would place them in the position of a particular stakeholder. As Basili et al. suggest, the user perspective might entail creation of a user's manual, the developer perspective a high level design, and the tester perspective a set of test cases.

However, Laitenberger and Atkinson (1999) argue that the development of artefacts, particularly those that would normally be developed anyway, is not the domain of software inspection. They suggest that such development occurring within an inspection would be either redundant or an inappropriate usurpation of developers' responsibilities. Instead, Laitenberger and Atkinson propose that the requisite perspective-specific artefacts be created before the inspection, by those who would normally be required to do so.

Most experimental evidence relating to SBR concerns DBR and PBR. However, the evidence does not uniformly support these techniques over CBR or *ad hoc* reading. Tables 2.2 and 2.3 record a number of experiments conducted thus far to compare

Table 2.2: Experiments assessing Defect-Based Reading (DBR).

Experiment	Environment	Subjects	Artefacts	Outperforms	
				Ad hoc	CBR
Porter et al. (1995)	Academic	48	Req.	Yes	Yes
Fusaro et al. (1997)	Academic	30	Req.	No	No
Miller et al. (1998)	Academic	50	Req.	—	Sometimes[a]
Sandahl et al. (1998)	Academic	24	Req.	—	No
Porter and Votta (1998)	Industrial	18	Req.	Yes	Yes

[a] DBR outperformed CBR for one out of two systems inspected.

Table 2.3: Experiments assessing Perspective-Based Reading (PBR).

Experiment	Environment	Subjects	Artefacts	Outperforms	
				Ad hoc	CBR
Basili et al. (1996)	Industrial	23	Req.	Yes	—
Ciolkowski et al. (1997)	Academic	51	Req.	Yes	—
Shull (1998)	Academic	66	Req.	No	—
Lanubile and Visaggio (2000)	Academic	223	Req.	No	No
Laitenberger et al. (2001)	Industrial	60	Code	—	Yes
Sabaliauskaite et al. (2002)	Academic	59	Design	—	No
He and Carver (2006)	Academic	12	Req.	—	Yes
Maldonado et al. (2006)	Academic	18	Req.	—	No

Table 2.4: Experiments assessing miscellaneous scenario-based reading techniques.

Experiment	Environ.	Subjects	Artefacts	Technique	Outperforms	
					Ad hoc	CBR
Cheng and Jeffery (1996)	Acad.	53	Req.	FPS	No	—
Biffl (2000)	Acad.	169	Req.	Hybrid[a]	—	Partially[b]
Halling et al. (2001)	Acad.	346	Req.	Hybrid	—	No
Dunsmore et al. (2003)	Acad.	69	Code	Use case	—	No
McMeekin et al. (2009)	Mixed[c]	62	Code	Use case	—	No

[a] A combination of PBR and OORT.
[b] Outperformed CBR for critical defects only.
[c] Included 36 students and 26 industry professionals.

the performance of DBR and PBR to their simpler alternatives. Overall, the results appear inconclusive. Ciolkowski (2009) conducted a meta-analysis of the PBR studies, finding that PBR is significantly more effective than *ad hoc* reading on requirements documents, but significantly less effective than CBR. For design and code artefacts, PBR appeared to outperform CBR. Ciolkowski expresses concern over researcher bias evident in the results.

Other studies have examined active guidance and separation of concerns in isolation. Sørumgård (1997) modified PBR to include additional active guidance, testing the modified version against the original and finding that it lowered inspection performance. Denger et al. (2004) also examined active guidance in PBR, by implementing separation of concerns in CBR; that is, by assigning different checklists to different inspectors. They tested PBR against their "focused" CBR and found no significant difference, a result supported by replication (Lanubile et al., 2004). Further, Regnell et al. (2000) reported no significant difference in the defects found by the three PBR scenarios, indicating that separation of concerns had no effect. Collectively, these studies would again suggest that PBR is no more effective than CBR. However, since some of the studies listed in Table 2.3 find otherwise, no conclusive statement can be made.

Further proposed scenario-based techniques have not been as widely discussed or evaluated. These include:

- Function Point Scenarios (FPS), wherein the division of inspector responsibilities is broadly based on the five data and transaction functions defined by function point analysis — internal/external files, external inputs, external outputs and external queries (Cheng and Jeffery, 1996);

- the Object Oriented Reading Techniques (OORTs), previously called Traceability-Based Reading (TBR) — a set of seven scenarios designed to find defects by cross referencing object oriented design artefacts, especially UML diagrams (Travassos et al., 1999, 2000);

- a hybrid of PBR and OORT, constructed by Biffl (2000) for the purpose of evaluating scenarios generally; and

- the use case technique developed by Dunsmore et al. (2003), wherein inspectors trace the events of use case scenarios through source code.

The use case technique was not presented by Dunsmore et al. as a type of SBR. Use case scenarios are quite distinct from reading scenarios, and the use case technique does not facilitate separation of concerns. However, the definition of a reading scenario is broad, and might be understood as any detailed set of reading instructions.

Table 2.5: Experiments assessing Usage-Based Reading (UBR).

Experiment	Environment	Subjects	Artefacts	Outperforms CBR	
				For critical defects	For all defects
Thelin et al. (2003)	Academic	23	Design	Yes	No
Thelin et al. (2004)	Academic	62	Design	Yes	Yes
Winkler et al. (2004)	Academic	131	Design	Yes	Yes
Winkler et al. (2005)	Academic	127	Design	Yes	No
McMeekin et al. (2009)	Mixed[a]	62	Code	—	No

[a] Included 36 students and 26 industry professionals.

Table 2.4 shows the studies examining these techniques. The evidence does not generally appear to endorse their effectiveness.

2.2.3   Prioritisation

More recent reading techniques have employed *prioritisation* as a means of improving inspection performance. Thelin et al. (2003) proposes Usage-Based Reading (UBR), similar in principle to the use-case technique independently proposed by Dunsmore et al. (2003). However, UBR requires use-case scenarios to be inspected in order of importance from the user's perspective. The intention is not to find more defects overall, but to find more critical defects.

Experimental results thus far consistently show that UBR outperforms CBR in finding critical defects, as shown in Table 2.5. Some replications have also shown that UBR outperforms CBR for overall defect detection. With the exception of McMeekin et al. (2009), these experiments have also consistently used the same set of artefacts.

Other prioritisation-based techniques have also been proposed, but fewer studies have examined them. These include:

- individual-ranked UBR (UBR-ir) — a variant of UBR in which prioritisation of use case scenarios is conducted by inspectors themselves, rather than by experts prior to the inspection (Winkler et al., 2004);

- tailored checklists (CBR-tc), wherein inspectors prioritise requirements and system functions and track them through the artefacts under inspection (Winkler et al., 2005) (presented as an extension to CBR, but arguably classified as a scenario-based technique, owing to its use of active guidance);

Table 2.6: Experiments assessing miscellaneous prioritisation-based reading techniques. (All experiments were conducted in an academic environment.)

Experiment	Subjects	Artefacts	Technique	Outperforms CBR		Outperforms UBR[a]
				Critical defects	All defects	
Winkler et al. (2004)	131	Design	UBR-ir	Yes	No	No
Winkler et al. (2005)	127	Design	CBR-tc	Yes	Yes	No
Bernárdez et al. (2004)	146	Req.	MBR	—	Yes	—
Lee and Boehm (2005)	28	Req.	VBR	Yes	No	—
Petersen et al. (2008)	23	Design	TC-UBR	—	—	No

[a] Overall and/or for critical defects.

- Metric-Based Reading (MBR), in which certain metrics correlated to defect proneness are used to determine where to focus inspector attention (Bernárdez et al., 2004);

- Value-Based Review (VBR) — another variant of CBR, wherein artefacts are inspected in order of priority, and potential issues within each artefact are examined in order of criticality (Lee and Boehm, 2005); and

- time-controlled UBR (TC-UBR) — another variant of UBR in which inspectors are given a limited time to cover each use case scenario (Petersen et al., 2008).

As illustrated above, prioritisation may be achieved in several ways, and be based on different criteria. UBR uses importance to the user as a criterion for prioritisation, MBR uses defect proneness, and VBR involves negotiation between stakeholders.

Table 2.6 lists studies assessing these other prioritisation-based techniques. The evidence suggests that they also generally outperform CBR, either in the detection of critical defects or overall defect detection. There is currently no evidence that they outperform UBR itself, though MBR, VBR and UBR have not yet been tested against one another.

2.2.4 Abstraction

Few published reading techniques explicitly seek to support the comprehension process itself. Yet, comprehension must (almost self-evidently) precede defect detection.

Table 2.7: Experiments assessing abstraction-based reading techniques.

Experiment	Environment	Subjects	Artefacts	Technique	Outperforms CBR
Dunsmore et al. (2003)	Academic	69	Code	Abstraction	No
Abdelnabi et al. (2004)	Academic	84	Code	Abstraction	Yes
				FBR	Yes[a]

[a] FBR also outperformed the abstraction-driven technique.

Dunsmore et al. (2003) propose an abstraction-driven technique to assist the comprehension process in support of defect detection. This requires the inspector to read artefacts systematically, creating abstract descriptions for methods and classes in the process. Artefact interrelationships are followed as they are encountered.

Functionality-Based Reading (FBR) also seeks to assist comprehension, particularly regarding object-oriented frameworks (Abdelnabi et al., 2004). FBR provides functionality rules containing a high level, abstract description of the system, developed prior to the inspection.

Table 2.7 shows the experimental evidence for the abstraction technique and FBR. Given the inconsistency of experimental outcomes for other reading techniques, such a small collection of results should be treated with caution.

2.3 Software Comprehension

Comprehension is essential to any form of peer review. A substantial proportion of an inspector's time may be devoted to understanding aspects of the system not explicitly documented (Letovsky et al., 1987). Yet, reading technique research has generally paid little attention to the process of understanding software. There is little comprehension-theoretic basis for many reading techniques, especially for checklist questions and active guidance.

Software comprehension research itself has predominantly focused on software maintenance as its principal application. However, comprehension underlies peer review just as it does maintenance (Dunsmore et al., 2000).

A variety of different software comprehension models have been proposed, many with overlapping elements. Soloway et al. (1988) distinguish between two such models by referring to *macro-strategies* and *micro-strategies*. However, the meaning of these two

terms is expanded in this section to capture a broader set of the software comprehension literature. Thus, a macro-strategy represents the starting point and overall direction of the comprehension process, and the nature of the inspector's *mental model* — the mental representation of the system. A micro-strategy represents the manner by which individual, discrete pieces of knowledge are added to the mental model. Macro- and micro-strategies operate in conjunction, and some comprehension models describe both.

2.3.1 Macro-strategies

The comprehension process was described by Brooks (1983) as one of "constructing mappings from a problem domain, possibly through several intermediate domains, into the programming domain". Shneiderman and Mayer (1979) too assert that comprehension lies in building a "multileveled internal semantic structure to represent the program". Inspection should ideally be based on a thorough understanding of the system. In the requirements and design phases this is not necessarily possible, because much of the system does not yet exist. Nevertheless, a thorough inspection requires an understanding of the system at all available levels of abstraction.

Littman et al. (1986) distinguish between *systematic* and *as-needed* strategies. In a systematic strategy, every aspect of the program is understood in a systematic fashion. By contrast, an as-needed strategy requires that judgement be used to determine what parts or aspects of the system are of interest. As-needed strategies are inherently error-prone, since readers construct an incomplete mental model. However, for non-trivial systems a systematic strategy is not generally practical. Hence, these two strategies serve more as an illustration of the difficulties inherent in comprehension rather than as a taxonomy of different approaches.

In other cases, comprehension strategies for understanding multiple levels of abstraction are divided into *bottom-up* and *top-down* varieties, illustrated briefly in Figure 2.3. Shneiderman and Mayer suggest a bottom-up model whereby the reader mentally combines units of semantic structure into larger units, eventually determining the functionality of the whole system. Pennington (1987) proposes a bottom-up model in which two mental representations of the system are constructed. The *program model* is built first and consists of the low and high level structure of a system. The *situation* or *domain model* is built later and captures the system's behaviour and functionality in terms of real-world entities — a level of abstraction above the program model.

Top-down comprehension is advocated by Brooks, who argued that bottom-up models are "only a degenerate special case of a more powerful process." Brooks proposed
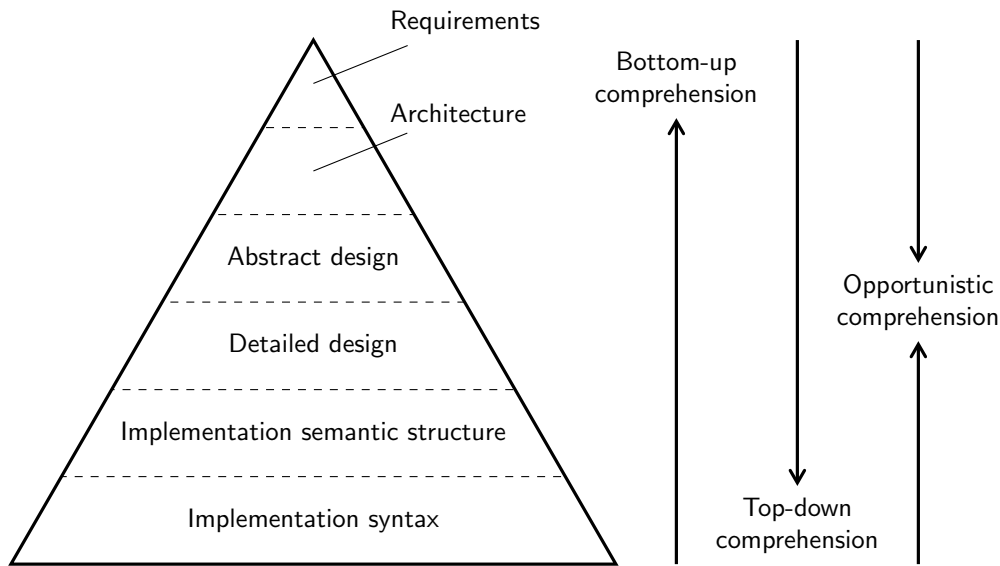
Figure 2.3: An example of the abstraction layers that a comprehension macro-strategy might seek to connect in a mental model.

that comprehension begins with a hypothesis about the system's overall purpose, and proceeds with the reader developing and validating successively more fine-grained hypotheses.

Other researchers have suggested that comprehension is more accurately reflected by an amalgam of top-down and bottom-up models. Linger et al. (1979) argue that "reading can seldom be strictly top down or bottom up. In reading the best-documented programs, one needs an occasional foray into details... And in reading a totally mysterious program, it is useful to back out of details periodically in order to form overall hypotheses or guesses that can help fit the details together more easily." The *integrated metamodel* describes the comprehension process in terms of top-down, program and situation models, all adding to and drawing from a common knowledge base (von Mayrhauser and Vans, 1995).

2.3.2   Micro-strategies

Micro-strategies represent a more fundamental level of the comprehension process.

Miller (1956) introduced the notion of *chunking*, whereby discrete chunks of information are mentally combined to form larger chunks. A very small limit exists on the number of chunks able to be stored simultaneously in short term memory, but the amount of information within each chunk is not limited in the same manner. Shneiderman and

Mayer use chunking to explain the bottom-up comprehension process. They argue that programmers do not read and store the individual symbols that comprise a piece of source code, but rather aggregate statements in the source code to form broader ideas about what the software does.

The top-down model proposed by Brooks (1983) entails the acquisition of knowledge through the iterative refinement of *hypotheses* — assertions regarding the system at some level of abstraction. For each hypothesis, the reader endeavours to find information with which it can be tested, and if confirmed then broken down into more detailed hypotheses. According to Brooks, hypotheses themselves are comparable to *schemas*, a concept taken from cognitive science. The development of hypotheses may be catalysed by *beacons* — parts of the system that are instantly recognisable and generally indicate the use of a particular well-known pattern or algorithm. In particular, beacons may include meaningful identifiers used in source code (Gellenbeck and Cook, 1991). In an experiment, Kim et al. (2000) observed the formation and refinement of hypotheses during comprehension of UML diagrams.

Soloway and Ehrlich (1984) proposed that readers use *plans* and *discourse rules* to understand the system. Plans represent well-known approaches to solving specific types of problems, whether trivial or complex. Both authors and readers of source code (or other artefacts) possess plan knowledge, built up from software development experience, which allows them to write or comprehend a software system. Soloway and Ehrlich also claim a correspondence between plans and schemas, implying a relationship between plans and hypotheses. Plans are used by the original developer to construct the system, while hypotheses arise in attempting to understand the developer's intent. Discourse rules are conventions that, when adhered to by the author, allow the reader to easily recognise plans within the system. As such, they share a similar role to beacons.

Letovsky (1986) constructed a more elaborate model in which readers undergo a series of *inquiry episodes*, consisting of a question, one or more conjectures and a search for evidence, any of which may be omitted. Questions arise due to particular uncertainties on the part of the reader. The reader may seek the mechanism by which the system accomplishes a particular task, the purpose of a particular part of the system, which of two competing explanations is accurate, or a resolution to apparently conflicting information. Conjectures represent plausible answers to the questions readers ask themselves, and are comparable to hypotheses. The reasoning process connecting questions to conjectures may rely on the application of plans, inferences from discourse rules, or a mental simulation of the system's behaviour. The reader assigns each conjecture a degree of certainty, based on the amount of evidence found so far to support it.

Inquiry episodes facilitate both bottom-up and top-down macro-strategies, or rather a combination thereof. Letovsky asserts that "the human understander is best viewed as an opportunistic processor capable of exploiting both bottom-up and top-down cues as they become available." The exploitation of top-down cues is more feasible when a system is well documented, and when the reader's experience encompasses many of the plans used by the author.

2.3.3 Delocalised Plans

Letovsky and Soloway (1986) observed that inquiry episodes occur frequently as a result of *delocalised plans* — plans whose implementation occurs across physically separated parts of a system. A reader must seek out these components in order to understand the plan. However, often the reader does not realise that delocalised plans exist. Chunking too may be affected by delocalised plans; Shneiderman and Mayer assert that chunking is more effective in the absence of `goto` statements.

In the experiment by Kim et al. (2000), participants were presented with one of two sets of diagrams, both with the same informational content. However, one set was designed to allow readers to more easily draw connections between diagrams. The use of this set indeed facilitated more switching between related diagrams, and led to greater numbers of hypotheses being formed and refined. Likewise, Hungerford et al. (2004) found that inspectors who exhibited a rapid switching behaviour between different design diagrams were able to detect more defects than those who concentrated on one artefact at a time.

Rist (1996) argues that plans and objects are orthogonal. That is, in an object-oriented (OO) context, plans are generally distributed among many objects. Thus, OO results in a proliferation of delocalised plans. Delocalised plans in OO were a central motivation behind the development by Dunsmore et al. (2003) of the use-case and abstraction-driven reading techniques. Both of these were designed to encourage inspectors to resolve and understand artefact interrelationships when encountered. Though not presented as such, OORT too might be seen as an attempt to address delocalisation, since it focuses on design artefact interrelationships.

2.3.4 Knowledge and Experience

Software comprehension research generally indicates that the choice of comprehension strategy depends on experience. Wiedenbeck (1986) found that experienced readers were able to recall beacons more reliably than other parts of the program, whereas

inexperienced readers could not. Similarly, Rist (1986) found that the use of plans increases with experience, and decreases with difficulty. Burkhardt et al. (1998) report that experts preferentially employ top-down strategies, while novices work bottom-up. Expertise has been identified as a principal driver of inspection performance (Sauer et al., 2000, McMeekin et al., 2009)

Given such experience-related effects, experienced inspectors would arguably be better able to recognise and understand delocalised plans. The experiment of Hungerford et al. (2004) demonstrates that some inspectors are indeed able to employ better strategies for coping with delocalisation than others.

The level of experience of participants in empirical studies is sometimes raised as an issue affecting their generalisability (Sjøberg et al., 2005). It is assumed that experience is positively correlated to defect detection effectiveness. However, interactions between experience and reading techniques are also of interest. Höst et al. (2005) argue that replications of controlled experiments in software engineering often fail in part because participant experience is not adequately described or controlled. They propose that experimental participants be classified by experience: (a) those with more than two years industrial experience, (b) those with three months to two years, and (for those with less than three months experience) (c) undergraduate students, (d) graduate students and (e) academics.

However, such broad experience categories may not capture the underlying factors driving comprehension. Experience is merely a proxy for expertise, which itself is a complex, multifaceted concept, not a simple categorical or numerical variable (Berlin, 1993). Expertise depends on knowledge acquired through experience, but not all knowledge is equal, nor all experience.

If some inspectors are better able to deal with delocalisation than others, then the pervasive use of OO may widen the gap between expert and non-expert inspectors. This would suggest that different inspection strategies are needed for different levels of expertise. Indeed, reading techniques do not appear equally beneficial to all levels of experience (Shull, 1998, cited in Regnell et al. 2000). However, if expertise is multi-dimensional, then these strategies must be manifold. An inspector may be an expert with respect to certain types of delocalised plans, but a novice with respect to others. Hence, making an informed decision to use one strategy over another must be based on a complex set of factors.

2.3.5   Cognitive Support

Cognitive support promises to help address the effects of delocalisation. Several different approaches have been proposed.

Soloway et al. (1988) suggest the insertion of documentation at strategic points in an artefact, where delocalisation is manifested. At such points, a complete understanding of a given piece of source code would otherwise require the reader to venture into other parts of the code, possibly without any obvious cue.

Other authors argue for tool support, and propose criteria and mechanisms by which this might be achieved. Storey et al. (1997, 1999) suggest a hierarchical set of goals for cognitive support tools. These are centred around systematic support for all comprehension macro-strategies (top-down, bottom-up and opportunistic), and also emphasise support for navigating between artefacts.

Walenstein (2002, 2003) describes three mechanisms by which tools might provide cognitive support:

- *redistribution*, where mental representations of the system are made physical, redistributing the programmer's cognitive load;
- *perceptual substitution*, where one physical representation is replaced by another, informationally-equivalent one that is more easily or quickly digested; and
- *ends-means reification*, where the process of comprehension itself is made physical.

Kim et al. (2000) combined two elements of cognitive support as a treatment in their experiment, resulting in improved comprehension. The first is a form of perceptual substitution — an object message diagram was selected over an informationally-equivalent event-trace diagram (the latter being given to the control group). The object message diagram shared the physical layout of the class diagram, making the two easier to understand in combination. Second, the treatment group was provided with a context diagram, a form of redistribution. This did not add any new information, but provided a representation that participants may otherwise have had to construct mentally.

Cognitive support and reading techniques are directed at very similar problems, though cognitive support is more closely tied to theories of software comprehension. Nonetheless, other alternate annotations or visualisations may directly support reading techniques.

Walkinshaw et al. (2005) use dependence graphs to effectively highlight parts of a sys-

tem related to a particular use case scenario. Similarly, Egyed (2003) uses execution of a system to observe, record and reconstruct the mapping between code and a particular test scenario. Such approaches might be used to assist UBR and its variants, or the tester perspective of PBR. PBR is also supported holistically, across the whole inspection process, through a tool developed by Chan et al. (2005).

Opportunities to support CBR are numerous. Some relatively straightforward defect types can be detected algorithmically (Lu et al., 2005, Moha et al., 2006), and in these specific cases tool support can almost entirely supplant the human comprehension process. Where tools are unable to answer checklist questions themselves, they may still provide metrics or other information with which an answer is more easily reached (Belli and Crişan, 1996, Anderson et al., 2003, Cooper et al., 2006).

2.4 Inspection Theory

Though approaches for addressing delocalised plans are well-established, they do not generally address the multifaceted nature of inspector expertise, or of system complexity. Moreover, they do not consider all the potential impacts on inspection outcomes. The ability to develop, refine or select an appropriate inspection strategy depends on being able to predict resulting inspection performance.

Thus far, no broad consensus has emerged regarding which reading techniques are the most effective. Though a number of ideas (e.g. active guidance, separation of concerns and prioritisation) have been proposed for improving inspection performance, the software engineering research community has not been able to convincingly argue for any one technique. Such ambiguity hinders widespread use of these techniques, or even the ideas on which they are based.

Jeffery and Scott (2002) argue that the apparent contradictions in empirical reading technique research have resulted from a lack of inspection theory. While the rationales for many of the features of reading techniques have been well articulated, their formation has been guided more by intuition rather than a cohesive theory. Hannay et al. (2007) report from a survey of empirical software engineering papers that "theory-driven investigations and theory building are rare in empirical software engineering". Without a common underlying framework, experimental results are often difficult to adequately explain or reconcile with those of other experiments.

Moreover, while qualitative theories of software comprehension are relatively well-established, predictive, quantitative theory is not. Quantitative inspection theory would

allow predictions to be made of inspection performance, and thus would provide a more objective basis for the use of a given inspection strategy.

Such a theory requires a metric — a numerical foundation on which a model can be constructed (for the purpose of comparing inspection strategies). Further, it requires a vocabulary with which to express multidimensional expertise and system complexity. Thus, this section describes some of the concepts that might form the basis of inspection theory.

2.4.1 Metrics

Most experiments assessing scenario-based reading techniques use the following metrics to assess effects on inspection performance:

- *effectiveness* — the number of defects detected (Dunsmore et al., 2003);
- *efficiency* — the number of defects detected per unit of time (Biffl, 2000);
- *detection rate* — the proportion of defects detected (i.e. effectiveness divided by the total number of defects) (Porter et al., 1995);
- *inspection effort* — the time taken to perform the inspection (Biffl, 2000).

Detection rate relies on knowledge of the total number of defects, which can be estimated using a capture-recapture approach (Briand et al., 2000). If the sets of defects found by two different inspectors — A and B — are assumed to be independent, then the proportion of all defects found by A is approximately equal to the proportion of B's defects found by A:

$$\frac{n_{\mathrm{A}}}{n_{\mathrm{T}}} \approx \frac{n_{\mathrm{A} \cup \mathrm{B}}}{n_{\mathrm{B}}}$$
$$\Rightarrow n_{\mathrm{T}} \approx \frac{n_{\mathrm{A}} \cdot n_{\mathrm{B}}}{n_{\mathrm{A} \cup \mathrm{B}}} \tag{2.1}$$

where n_{T} is the total number of defects;
   n_{A} is the number of defects found by inspector A;
   n_{B} is the number of defects found by inspector B; and
   $n_{\mathrm{A} \cup \mathrm{B}}$ is the number of defects found by both inspectors.

Thus, detection rate and similar metrics are easily measured and understood. Effectiveness or detection rate offer the most direct measures of inspection performance. However, their use as such assumes that all defects are equally important. In fact, defects can vary considerably in severity, from those resulting in aesthetic annoyances to those potentially endangering human life.

The prioritisation-based reading techniques discussed in Section 2.2.3 are based on a recognition of varying defect severity. To assess UBR, Thelin et al. (2003) used a different suite of metrics, based on dividing defects into different categories of importance. Here, the use of an alternate performance metric is essential for demonstrating the value of the reading technique. UBR's effect on overall efficacy is ambiguous, but its effect on critical defect detection is clearly positive.

Similarly, Lee and Boehm (2005) used a metric reflecting VBR's dual prioritisation scheme:

$$\text{Impact} = \sum \text{Artefact priority} \times \text{Issue criticality} \tag{2.2}$$

Each detected defect has a value for both artefact priority and issue criticality. In their experiment Lee and Boehm allow the values 1 ("low"), 2 ("medium") or 3 ("high") for each of the two factors, but note that in practice these values would be determined by experts. The metric reflects the cost saved by the inspection, but without a scale. Unlike the critical defect efficacy and efficiency metrics used by Thelin et al., the impact metric does not ignore low or medium priority/criticality defects. However, the cost of the inspection itself is not taken into account.

Ultimately, prioritisation arises from a desire to minimise costs. For this purpose, the most useful measures of inspection performance are those that take into account all inspection-related costs, expressed in units of time, effort or currency. The major (though not sole) economic consequence of defect detection lies in lower costs associated with fixing defects later in the project. However, a single absolute cost value by itself, even if inclusive of all inspection-related costs, is still not a reliable indicator of inspection performance because it cannot be compared to other cost values arising in different circumstances. Measures of inspection performance must also consider hypothetical costs potentially incurred if the inspection had not taken place.

Collofello and Woodfield (1989) propose a *cost effectiveness* metric, based on the real cost of inspection and the hypothetical cost without inspection:

$$\text{CE}_\text{C} = \frac{\text{Future costs avoided}}{\text{Inspection cost incurred}} \tag{2.3}$$

CE_C is similar in principle to the efficiency metric. The number of defects detected is a rough indicator of future costs avoided, and here the latter is used in place of the former. However, approximations notwithstanding, efficiency remains unsuitable as a measure of inspection performance, and CE_C likewise. Both metrics reflect the costs avoided per unit of inspection time, rather than the proportion of the total costs that could have been avoided. They do not penalise inspections for finding fewer defects,

provided they also consume proportionally fewer resources.

For example, an inspection consuming one hour and finding ten defects (or saving ten cost units later in the project) is more efficient than one lasting two hours and finding fifteen. Moreover, efficiency remains constant irrespective of the actual number of defects, whether there be fifteen or a hundred.

Thus, Kusumoto et al. (1991) introduce a different cost effectiveness metric, incorporating the total potential cost incurred without inspection:

$$
\begin{aligned}
\mathrm{CE_K} &= \frac{\text{Net costs avoided}}{\text{Total hypothetical cost without inspection}} \\[2em]
&= \frac{\text{Future (testing) costs avoided} - \text{Inspection cost incurred}}{\text{Future (testing) costs avoided} + \text{Future (testing) costs incurred}} \\[2em]
&= \frac{\text{No. defects detected} \times (\text{Testing cost per defect} - \text{Inspection cost per defect})}{\text{Total no. defects} \times \text{Testing cost per defect}} \\[2em]
&= \text{Detection rate} \times \frac{\text{Testing cost per defect} - \text{Inspection cost per defect}}{\text{Testing cost per defect}} \\[1em]
&= d \cdot \frac{c_\mathrm{T} - c_\mathrm{I}}{c_\mathrm{T}}
\end{aligned}
\tag{2.4}
$$

$\mathrm{CE_K}$ is essentially a scaled version of the detection rate metric d. Whereas d is the proportion of defects found through inspection, $\mathrm{CE_K}$ is the proportion of defect-related costs avoided through inspection. Any positive value would indicate that inspections are reducing costs associated with defects. Values approaching one would indicate that these costs are being almost eliminated. Negative values are also theoretically possible, and would indicate that inspections are actually counterproductive. $\mathrm{CE_K}$ is a measure of inspection performance whose meaning does not vary from inspection to inspection. Its values can be compared across inspections, projects and organisations.

Freimut et al. (2005) introduce a further cost effectiveness metric, keeping the form of $\mathrm{CE_K}$ but refining its definition in several respects. Freimut et al. distinguish between inspection and rework costs (c_I and c_R). They also allow for any number of defect detection activities throughout the project, rather than the fixed set of three that Kusumoto et al. assume (design review, code review and testing).

Importantly, CE_K implicitly assumes that each defect missed by an inspection results in one defect being found in testing. In fact, a single defect missed by inspection may result in multiple defects occurring in the next phase. Some of these may be detected and reworked in subsequent verification activities, or propagate further still.

Thus, Freimut et al. define cost effectiveness for each of multiple verification activities in a project as follows:

$$CE_F(j) = d_j \cdot \frac{c_{Mj} - c_{Rj} - c_{Ij}}{c_{Mj}} \qquad (2.5a)$$

where j identifies a particular verification activity or phase, between 1 and J;

d_j is the detection rate in phase j;

c_{Mj} is the cost of missing a defect in phase j;

c_{Rj} is the cost of reworking a defect in phase j; and

c_{Ij} is the per-defect inspection cost in phase j.

The cost c_{Mj} comprises rework costs incurred later in the project, and can be calculated as follows:

$$c_{Mj} = \sum_{k=j+1}^{J} \left[c_{Rk} \cdot d_k \cdot \prod_{\ell=j+1}^{k} (1 - d_\ell) \cdot g_\ell \right] \qquad (2.5b)$$

where g_ℓ is the propagation factor; the number of defects in phase $\ell+1$ resulting from each unreworked defect in phase ℓ.

Freimut et al. suggest that the terms in equations 2.5a and 2.5b can be obtained using a combination of project data and expert opinion, but caution that for estimated quantities there must be mechanisms in place to identify and mitigate bias.

CE_K and CE_F establish the form of a suitable inspection performance metric, but make several assumptions. First, they assume that inspection costs are incurred on a per-defect basis, manifested in the c_I term. This is not true. Inspection costs are incurred in the search for defects in general, not for each defect individually. Overall, the relationship between the number of defects detected and the effort expended cannot be linear, because there are only a limited number of defects to be found. Actual effort is expended by reading artefacts, forming hypotheses, conducting inquiry episodes, reconstructing plans, etc. as described in Section 2.3.2. A single hypothesis or plan may underlie the detection of multiple defects, or none at all.

Two other process-related assumptions are made: that all detected defects are reworked, and that all costs are borne by the developers themselves (in inspection, test-

ing or rework). However, even after a defect has been found, it may be considered of insufficient importance to warrant rework, given budget or time constraints (Hewett and Kijsanayothin, 2009). Even if rework is attempted, it will not necessarily succeed (Fagan, 1976, Levendel, 1990). Further, some defects are found by users (Jones, 1996) and can result in substantial costs incurred by external parties (National Institute of Standards and Technology, 2002). The UBR and VBR techniques are intended to find the most important defects from the user's perspective. Their rationale would vanish if users did not incur defect-related costs.

Finally, CE_K and CE_F incorporate the detection rate d in their definition, and thus assume that all defects are equal. As a result, they lack the expressiveness needed to model the effects on cost effectiveness of different inspectors, systems and inspection strategies.

2.4.2 Taxonomies

Cost effectiveness can form the basis of a quantitative model of software inspection. However, to predict the cost effectiveness of a given inspection strategy, it must first be possible to express that strategy in the language of the model.

Classifying defects by severity, as done by Biffl (2000) or Thelin et al. (2003), would partially address the lack of expressiveness in the cost effectiveness metrics of Kusumoto et al. and Freimut et al.. That is, the calculation could be broken into components, each dealing with a different severity level. However, this would not necessarily allow for any interesting queries; clearly an inspection strategy favouring the detection of more high-severity defects is preferable. The problem lies in determining how to engineer such a strategy.

Malik et al. (2004) propose a set of criteria intended to allow a theoretical comparison of reading techniques. Their approach categorises techniques by sets of predetermined attributes, describing broad characteristics of reading techniques. This represents an attempt to break down reading techniques into their component parts — a necessary function of inspection theory. However, it does not delve far enough into the processes underlying reading techniques to allow for quantitative predictions. Little is said of the mechanism by which attributes are assigned to techniques, or how the particular set of attributes discussed was determined.

Defect classification schemes are another taxonomic approach; one that describes the beginnings of a vocabulary for inspection theory. They connect reading techniques to

quantitative data that might be used to predict cost effectiveness. Chernak (1996) argues for checklist creation and improvement based on statistical occurrence of defect types. Each checklist item can be mapped to a particular defect type, which had a particular probability of occurrence. Similarly, Sullivan and Chilarege (1991) observe differences in the consequences of different defect types in an operational setting. The generic value-based checklist proposed by Lee and Boehm (2005) (as part of VBR) has its items prioritised by criticality, designed to favour more important defect types. Expert estimation (Freimut et al., 2005) might be used to quantify the costs thereof.

However, this still leaves the comprehension process itself. Ultimately, the cost effectiveness of an inspection strategy cannot usefully be modelled without incorporating elements of software comprehension theories.

2.4.3   Models

The Constructive Cost Model (COCOMO) is a long-standing, quantitative, predictive model of software size, development time and effort (Boehm et al., 2000). Both COCOMO 81 and COCOMO II estimate effort using the following equation:

$$\text{Effort} = \text{Coefficient} \times \left[\prod_i \text{Cost driver } i \right] \times \text{Size}^{\text{Exponent}} \qquad (2.6)$$

The exponent and cost drivers comprise various factors affecting the difficulty of the project. The size is expressed in lines of code, and is itself estimated by means of further equations. The coefficient is present for calibration purposes.

With sufficiently precise calibration, COCOMO may be relatively accurate. However, Equation 2.6 gives little insight into the mechanisms by which development takes place and costs are incurred. The cost drivers are opaque factors, derived from empirical data but not amenable to further theoretical analysis. While it has predictive power in the right circumstances, COCOMO has little explanatory power. Its intended application lies in estimating project characteristics, not in comparing different approaches.

More closely related to inspection theory, Cockram (2001) and Wu et al. (2005) model software inspection itself with Bayesian networks. Figure 2.4 shows part of the model proposed by Wu et al.. Here, inspection effectiveness is determined by system size and complexity, the quality of the process and the remaining defects, and some of these in turn are determined by other variables, and so on.
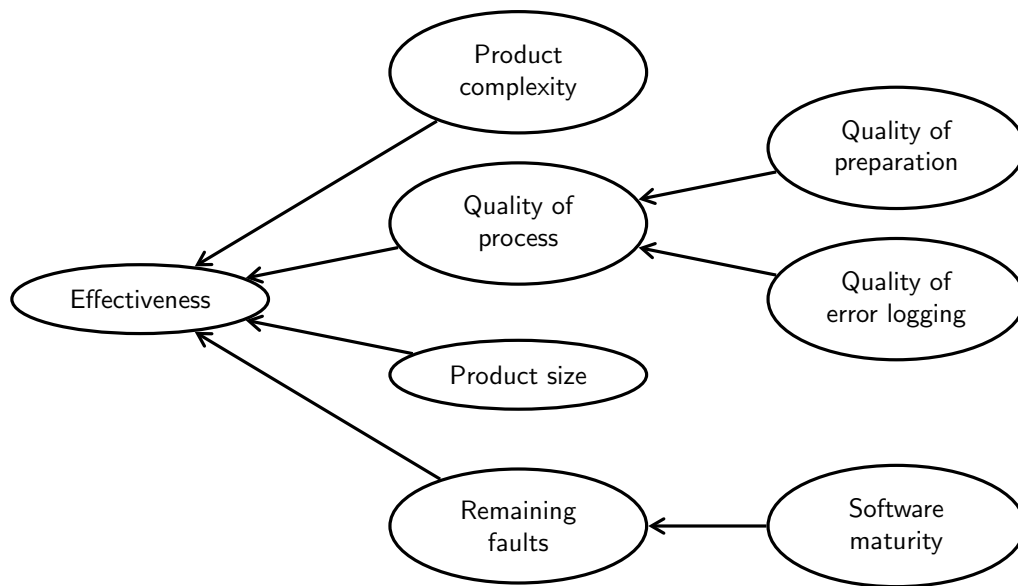
Figure 2.4: Part of the Bayesian inspection model proposed by Wu et al. (2005).

However, like COCOMO, these models do not embody the mechanics of inspection, but are simply an expression of how overall factors affect the overall outcome. Like the notions of experience and expertise described earlier, many of the variables shown in Figure 2.4 are aggregated, simplified forms of complex, multifaceted concepts. As such, they have limited capacity to uncover or explain phenomena that are not already well understood.

Inspection strategies can be complex, comprising many parts that cannot necessarily be represented simply as a list of factors. Even where the inspection strategy is kept constant, contradictory results from previous studies shown in Section 2.2 hint at interaction effects. For instance, active guidance and cognitive support cannot simply be COCOMO-style cost drivers, because they may have different effects depending on when and where they are applied.

Inspection theory must be both predictive and descriptive in order for inspection strategies to be compared. An inspection model must incorporate the mechanisms of peer review — the comprehension process, as influenced by the inspectors, the artefacts and the reading technique — not just the outcomes thereof.

2.4.4   Ethical Application

Inspection theory might be constructed such that the cost effectiveness of an inspection can be accurately and precisely predicted, based on the underlying mechanics of

comprehension and software development.

However, in reducing inspection performance to a single cost effectiveness value, an inspection model might raise ethical issues related to its application. Such a model would assume that all costs potentially incurred are quantifiable and have largely interchangeable units, ultimately expressed in monetary terms.

Some domains involve potential costs to users or other external parties that are not easily reducible to a monetary form. Examples of such costs include:

- physical injury or death, particularly in safety-critical domains (Leveson and Turner, 1993);

- corruption of democratic processes, in situations where software facilitates voting or performs vote counting (Feldman et al., 2007);

- loss of essential services or infrastructure, in situations where software helps coordinate or control such services or infrastructure;

- security leaks, in situations where software handles sensitive government or corporate information;

- loss of freedom or reputation, such as where software is used to identify or track people potentially involved in unlawful activity;

- personal data loss, where photos, videos, documents, etc. have intellectual or sentimental value;

- loss of privacy, in situations where software handles confidential personal information; and

- exposure to inappropriate material in filtering applications (for whatever definition of "inappropriate" the reader may envisage, if any).

The costs listed above are not numerically comparable to each other or to time or monetary costs incurred through inspection, defect correction, etc. For legal purposes, many of the above costs may be assigned monetary values, though ethically they are not exchangeable.

As a matter of principle, software engineers should consider all costs that might arise from software development, monetary or otherwise. A model that reduces inspection performance to a single numerical value cannot directly incorporate non-monetary costs. Therefore, where such costs arise, such a model should not be relied upon to provide a definitive recommendation for the appropriate inspection strategy.

2.5   Summary

Software inspection is an effective means of defect detection. Various reading techniques have been proposed to improve its effectiveness, but the results of controlled experiments comparing them have often been inconsistent. This may result from a lack of consideration for the comprehension process, and for comprehension issues arising from factors associated with the inspectors themselves and the artefacts under inspection. Elements of inspection theory exist, but have not yet been assembled into a working model of software inspection, by which different inspection strategies might be compared theoretically.

In this thesis, Chapter 4 examines prevalent software inspection practices through an industry survey. The statechart and scenario studies (described in Chapters 5 and 6) then seek to uncover and understand specific comprehension issues in an inspection context, particularly those related to delocalised plans. Chapter 7 presents the checklist experiment, which examines active guidance more closely, by controlling inspector- and system-related factors, and examines potential interactions between checklists and inspector expertise.

Based on the findings of these chapters and the discussion in Section 2.4, a theoretical framework and predictive, descriptive model of software inspection are proposed in Chapter 8.

The next chapter undertakes a review of methodological concepts, techniques and issues underlying this empirical and theoretical work.

.

Chapter 3

Methodological Background

> *"We do nothing* until *our heads have actually been cut off."*
> *"And then we spring into action?"*

> — *Blackadder the Third*

This chapter reviews some of the empirical and theoretical methods employed in this thesis. Software engineering research has used both qualitative and quantitative methods in the analysis of empirical data, often adapted from other disciplines.

This chapter does not describe the detailed methodology of the work supporting this thesis. Rather, it introduces the methods in a general sense, in as much detail as necessary, such that their rationales and practicalities can be understood. (The adaptations of these methods to the research supporting this thesis are discussed in subsequent chapters.)

3.1   Subject Experience

In conducting empirical studies of programming or inspection behaviour, the use of human subjects is essential. The use of students as subjects in such studies is commonplace (Sjøberg et al., 2005), but has raised questions regarding experimental *realism* — the extent to which experimental conditions mirror industrial practice (Curtis, 1986, Sjøberg et al., 2002). At stake is whether results from empirical studies involving students can be generalised.

The effects of experience and expertise were discussed in Chapter 2, Section 2.3.4. Ex-

pertise has an important effect on inspection and comprehension performance. However, this difference may be largely quantitative, rather than qualitative. Even though expertise is in essence qualitative, Gugerty and Olson (1986) report that novice programmers do not exhibit qualitatively different comprehension strategies from experts. Rather, they are simply less efficient (as a result of choosing inferior initial hypotheses). Thus, in qualitative investigations that seek to explore the existence of certain cognitive phenomena, the use of students rather than professionals may have relatively minor implications. Caution should be exercised in making generalised conclusions, just as in any qualitative study.

Tichy (2000) further argues that the use of student subjects can be valid if the results show (or fail to show) a trend attributable to a particular factor. The trend for professionals may generally be equal in direction, if not equal in magnitude. In some respects, differences between final-year students and professionals can be small (Höst et al., 2000). However, there is evidence of interactions between expertise and reading techniques (Shull, 1998, cited in Regnell et al. 2000), so some caution must be exercised in drawing conclusions.

Also resulting from the multi-faceted nature of expertise (as discussed in Section 2.3.4), the use of nominally professional subjects is itself no guarantee of generalisability, because expertise has many forms.

3.2 Qualitative Analysis

Qualitative analysis is used to deal with categorical or unstructured data, often for the purpose of theory generation (Seaman, 1999). This section does not undertake a comprehensive treatment of qualitative methods, but describes two techniques used in this thesis: protocol analysis and coding.

3.2.1 Protocol Analysis

Protocol analysis is a qualitative data collection and analysis method, providing insight into subjects' mental processes as they perform a predetermined task (Ericsson and Simon, 1993). A large part of the software comprehension research thus far has relied on protocol analysis; e.g. Letovsky (1986), Littman et al. (1986), Kim et al. (2000), Hungerford et al. (2004) and McMeekin et al. (2008).

To collect data for protocol analysis, subjects are asked to *think aloud* as they undertake the task. They are instructed to say out loud everything that comes to mind. They are helped in doing so in two ways. First, subjects are given prior training tasks to familiarise them with the act of thinking aloud. For instance, they might each be asked to perform a non-trivial multiplication in their head, while voicing their working. Second, an interviewer prompts them to keep talking if they fall silent. As subjects undertake the task assigned, their verbalisations are recorded. These are later transcribed, and the transcription segmented into *utterances*.

Ericsson and Simon describe three levels of verbalisation, where subjects are asked to:

1. vocalise thoughts already mentally encoded in spoken language;
2. vocalise all existing thoughts, translating them to spoken language if necessary; or
3. explicitly describe their own thought processes.

Ericsson and Simon argue that the first two forms do not materially alter subjects' cognitive processes (though the second entails some additional translation overhead). The third form requires subjects to vocalise additional information not normally present. While the extra information sought might be useful, obtaining it causes subjects to alter their thought processes. Thus, this form does not produce an accurate reconstruction of the approach to the task ordinarily taken.

The second form is preferred, because it elicits the most information with minimal interference. However, results obtained from it do not generally represent a complete reconstruction of subjects' thoughts.

3.2.2   Coding

In qualitative data analysis, *coding* is the process of categorising data, whether from subjects' verbal utterances or another source. Coding is done by means of a *coding scheme* — a set of rules for transforming unstructured data supplied by the user into a discrete set of codes, or categories. The coded data may then participate in quantitative analysis Seaman (1999). Often the coding scheme is developed after the data has been informally examined, in order to maximise the amount of useful information captured for qualitative analysis.

Coding involves a degree of interpretation on the part of the coder. In cases of high subjectivity, this may present a threat to validity. To address such threats, the data

may be coded (or partially coded) again by a second, independent coder. Cohen's Kappa statistic (El Emam and Wieczorek, 1998) measures the degree of agreement between coders, and thus provides an indication of the reliability of the coded data. If the two coders assign precisely the same categories to the data, kappa is one. If the level of agreement is that expected purely by chance, kappa is zero. Values between zero and one demonstrate varying levels of agreement beyond chance. (Kappa can be negative if the level of agreement is less than expected by chance; e.g. if the coders were to disagree on everything.)

El Emam and Wieczorek suggest several thresholds as a guide to the interpretation of kappa. In particular, values above 0.78 indicate "excellent" agreement, and values above 0.62 indicate "good" agreement.

3.3 Quantitative Analysis and Modelling

3.3.1 Log-linear and Logistic Models

The construction of a mathematical model may be warranted either for predictive purposes or to test hypotheses. In either case, an equation is obtained that expresses a dependent variable in terms of a series of factors.

A linear model is a relatively simple case with the form:

$$\mathbb{E}(Y \mid \mathbf{X}) = \boldsymbol{\beta} \cdot \mathbf{X} = \beta_0 + \beta_1 X_1 + \ldots + \beta_n X_n \tag{3.1}$$

This expresses the expected value ($\mathbb{E}$) of Y, given a vector of n factors $\mathbf{X}$, in terms of those factors. Here:

- Y is the response (dependent) variable;
- β_0 is the intercept — the expected value of Y when all the factors are zero;
- β_1 to β_n are the regression coefficients — the effect of each factor on Y;
- X_1 to X_n are the factors (independent variables) themselves; and
- the dot product $\boldsymbol{\beta} \cdot \mathbf{X}$ is shorthand for the equation. (The intercept β_0 is multiplied by X_0, but the latter is defined to be 1 and so omitted.)

The factors may be binary, integer or continuous variables. They may not (directly) be categorical variables, whose values have no particular ordering, or ordinal variables, whose values are ordered but not arranged in consistent, definable increments. To

represent the effect of a categorical variable with m possible values, it must be divided into $m - 1$ mutually exclusive binary variables. (One of the m values does not require a binary variable of its own, because it is implied when the other binary variables are all zero.)

Interaction effects can be represented by setting one factor to be the product of two or more others. For instance:

$$\mathbb{E}(Y \mid \mathbf{X}) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2 \tag{3.2}$$

For modelling the outcome of a *repeated-measures* experimental design, an extra *random effect* term can be added:

$$\mathbb{E}(Y_i \mid \mathbf{X}) = \boldsymbol{\beta} \cdot \mathbf{X} + \gamma_i \tag{3.3}$$

Here, γ_i indicates an additional effect associated with the ith group of observations. The actual effect is not interesting in itself, but its inclusion helps minimise experimental error. The response variable Y_i is also given in terms of the observation group i.

Linear models are by their nature additive; each factor adds to (or subtracts from) the response variable. In some cases this is not desirable, in which case a *link* function can be used to alter the relationship between the response variable and the factors:

$$\mathbb{E}(Y \mid \mathbf{X}) = f(\boldsymbol{\beta} \cdot \mathbf{X}) \tag{3.4}$$

Equation 3.4 represents a *generalised linear model*. Here, Y is not (necessarily) a linear function of $\mathbf{X}$ as before, but is nonetheless a linear function of $f(\mathbf{X})$.

A log-linear model is a type of generalised linear model used when the factors must have a multiplicative rather than additive effect:

$$\begin{aligned}
\mathbb{E}(Y \mid \mathbf{X}) &= \exp(\boldsymbol{\beta} \cdot \mathbf{X}) \\
&= \exp(\beta_0) \times \exp(\beta_1 X_1) \times \ldots \times \exp(\beta_n X_n)
\end{aligned} \tag{3.5}$$

Here, Y is prohibited from being negative (unlike in an ordinary linear model).

A logistic model is another type of generalised linear model, used when the response variable is binary. In such situations, the probability that $Y = 1$ is modelled rather than the expected value of Y. Ordinary linear models are generally inappropriate here;

probability is constrained to the range $[0, 1]$ while the response variable of a linear model is not. A linear model may predict values less than 0 or greater than 1 for a given set of inputs.

Logistic models use the logistic function logit^{-1} (the inverse of the logit function) to transform the unconstrained $\boldsymbol{\beta} \cdot \mathbf{X}$, representing the *log odds* of $Y = 1$, into a constrained probability value, guaranteeing valid probabilities for any combination of factors. Probability approaches zero as the log odds approaches $-\infty$, and one as the log odds approaches $+\infty$. A probability of 0.5 is equivalent to a log odds value of zero.

The logistic model is defined as follows:

$$\begin{aligned}
\mathbb{P}(Y = 1 \mid \mathbf{X}) &= \text{logit}^{-1}(\boldsymbol{\beta} \cdot \mathbf{X}) \\
&= \frac{1}{1 + \exp(-\boldsymbol{\beta} \cdot \mathbf{X})}
\end{aligned} \tag{3.6}$$

3.3.2   Survival Analysis

Survival analysis is a statistical technique used to model the time until some particular event occurs, used where the event may or may not actually occur within the window of observation. Where the event is not observed, the time is said to be *censored*, and the end of the period of observation is recorded instead as the *censored time*. Often the event under consideration is the death of a person or other organism, or the failure of a piece of mechanical equipment. A censored observation is therefore one in which the person or organism remains alive at the end of the period of observation, or where the equipment has not yet failed.

Censored observations cannot generally be treated as missing data, because they contain information, particularly if the censored times are large. Censored times represent lower bounds on the actual survival times. A linear model derived only from known survival times may substantially underestimate the actual time. For example, consider a survival time data set in which half the times range from 0 to 100 (in arbitrary units), and the remaining half are censored at time 100. The median of the entire data set will be about 100, but the median of the known survival times may be considerably lower.

In survival models, the response is typically the *hazard function* function $\mu(t)$[1]. This is specific to the data set, and is proportional to the probability of some event occurring in a short window of time around t, given that it has not occurred before t. The hazard

---

[1]Traditionally the symbol λ is used to represent the hazard function, but λ is used for other purposes in this thesis.

function can be derived analytically from the event's cumulative probability function. Each factor in the model influences the hazard function.

Several types of survival models can be constructed, depending on the assumed behaviour of the hazard function. The Cox proportional hazards model is often used where the effects of one or more factors are of interest, but the shape of the hazard function itself is not (Hougaard, 2000, Harrell, 2001). Such models can be constructed without knowing the underlying hazard function, and are therefore expressed in terms of an arbitrary base hazard function $\mu_0(t)$:

$$\mu(t \mid \mathbf{X}) = \mu_0(t) \exp(\boldsymbol{\beta} \cdot \mathbf{X}) \tag{3.7}$$

As in log-linear models, the exponential function in Equation 3.7 gives each factor a multiplicative rather than an additive effect. This is an explicit assumption in the Cox proportional hazards model — the factors must have a proportional (multiplicative) effect on the hazard. However, a Cox proportional hazards model does not require an intercept term, because it expresses a relative effect.

A Cox proportional hazards model can also be *stratified*, wherein the base hazard function is allowed to vary between different groups of observations. A *conditional logistic model*, mathematically related to a stratified Cox model, can also be used to describe a stratified set of observations where the response variable is binary (Collett, 2003, Therneau and Lumley, 2008).

3.3.3 Bayesian Networks

A Bayesian network is an acyclic, directed graph comprising variables connected by arcs representing their dependencies (Koller and Friedman, 2009). Bayesian networks can provide an overview in situations where complex dependency structures arise.

Both random and deterministic variables may appear in a Bayesian network. Deterministic variables are precisely determined by their parents (i.e. nodes connected via incoming edges). Random variables are described by a probability distribution and are merely influenced by their parents. The direction of an edge reflects the direction of a causal relationship between two variables (i.e. which "causes" the other).

Figure 3.1 shows an example of a Bayesian network. Here, several variables serve to determine the extent of plant growth, as shown by the arrows. Rain influences the
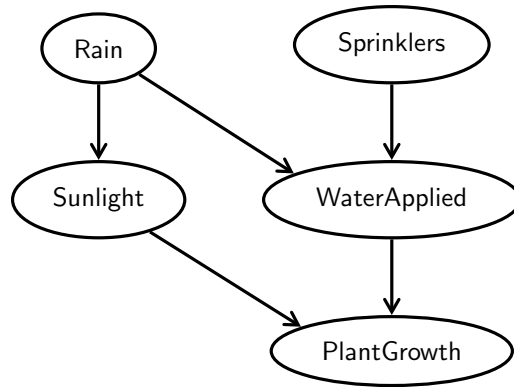
Figure 3.1: An example Bayesian network.

amount of sunlight and water applied. The sprinklers also influence the amount of water applied. Sunlight and water then influence plant growth.

The graph itself only specifies which variables influence one another, not precisely how they do so. For instance, rain diminishes the amount of sunlight, but increases the amount of water, but this cannot be represented in Figure 3.1.

A Bayesian network can be used as a reference in the construction of more precise models for each variable. For instance, a generalised linear model might be used to describe plant growth, as follows:

$$\mathbb{E}(\text{PlantGrowth} \mid \text{Sunlight} = s, \text{WaterApplied} = w) = f(\beta_0 + \beta_1 s + \beta_2 w + \beta_3 sw) \quad (3.8)$$

Equation 3.8 expresses the conditional expected plant growth given the amount of sunlight and water. Given those values, plant growth is independent of rain and sprinklers, and so the model does not need to refer to them. Sunlight and water applied could themselves be modelled in a similar fashion.

3.4   Application

Table 3.1 shows how the issues and methods described in this chapter apply in subsequent chapters.

Coding is used in different forms in all four empirical studies discussed in this thesis (in chapters 4, 5, 6 and 7).

Table 3.1: Applicability of experimental and theoretical issues/methods in this thesis.

	Chapter 4 (industry survey)	Chapter 5 (statechart study)	Chapter 6 (scenario study)	Chapter 7 (checklists experiment)	Chapter 8 (inspection theory)
Student subjects	—	Yes	Partially	Yes	—
Protocol analysis	—	—	Yes	—	—
Coding	Yes	Yes	Yes	Yes	—
Log-linear modelling	—	—	—	—	Yes
Logistic modelling	—	—	—	Yes	Yes
Survival modelling	—	—	—	Yes	—
Bayesian networks	—	—	—	—	Yes

The model proposed in Chapter 8, Section 8.2 is partially described by a Compact Bayesian Network (CBN) notation. CBN itself is a proposed extension to the standard BN notation, and as such is described in Section 8.2.4 rather than in this chapter.

Chapter 4

Prevalent Inspection Practices

"We're for the compulsory serving of asparagus at breakfast, free corsets for the under-fives and the abolition of slavery."

— Blackadder the Third

This chapter describes an industry survey conducted to identify common peer review practices, and the circumstances of their use or potential use (as per the first research question posed in Chapter 1). Results from this survey help frame and inform the approaches taken in subsequent chapters. The survey also finds direct opportunities to improve software peer review practice (and particularly inspection practice) so as to reduce overall costs.

Current inspection literature (as discussed in Chapter 2, sections 2.2 and 2.3.5) discusses numerous techniques for improving software comprehension and defect detection, focusing on more important defects or otherwise managing inspection resources for the benefit of software quality. It is not clear which of these methods, if any, are actually being adopted by industry. Any disparities that do exist may indicate missed opportunities to properly adapt new techniques to real world situations, or to communicate their effectiveness to industry.

Further, feedback from industry sources is essential to maintaining the relevance of software engineering research. For instance, there is comparatively little value in developing a reading technique to detect defects in artefacts that are hardly ever used. The principles behind a technique may be obscured if the context for which it was envisaged does not appear relevant.

To these ends, the survey asked questions relating to the frequency, length and cost

effectiveness of peer review activities, reading techniques used, phases of development and effort expended therein, artefacts used and reviewed, standardisation of those artefacts and the tools used in their creation. Such information broadly illustrates the use of peer reviews, and by implication whether they are being used appropriately. The applicability of reading techniques to industry can thus be inferred based on their applicability to certain types of artefacts and artefact properties. These artefact types, properties and combinations thereof also suggest how software inspection reference models might be instantiated, such that they are widely-representative of real-world software development.

4.1 Survey Process

4.1.1 Online Questionnaire

The survey utilised an online questionnaire. This diminished the potential for data entry errors, but also provided other important advantages:

- Providing a URL to potential respondents is logistically easier than sending them a printed questionnaire, or requiring that they print it themselves, and then collecting it afterwards. This is true regardless of the method of recruitment. Internet access is a virtual certainty for software engineering industry professionals.

- An electronic questionnaire allows for multiple-choice answers to be dynamically generated based on responses to other questions. For example, respondents are asked to indicate which artefacts are used in which phases of development, based on sets of artefacts and phases identified in previous questions. Mapping one to the other is relatively straightforward if the possible answers have been generated automatically, but may be tedious and confusing if done on paper.

- Similarly, a question can be "disabled" if it is irrelevant given previous responses. This helps reduce potential confusion.

The questionnaire was principally multiple-choice, and for many questions respondents were able to select multiple answers. For some questions they were asked to provide (if necessary) a comma-separated list of answers. No question (except the final "general comments" question) asked for free-form discussion. However, respondents were given the opportunity to provide free-form comments through pop-up input fields marked "Comment on this question".

The questionnaire asked for "one person (or more) within your organisation, department or team" to complete the questions, and that this person be "well-informed of the software development activities therein". Further, respondents were encouraged to say "what *actually* happens in your organisation/department/team, not just what is written down".

4.1.2   Preliminary Survey

Several industry professionals were asked to provide preliminary feedback on the initial questionnaire, to assess the relevance of the questions and appropriateness of the provided multiple-choice answers.

In response to this feedback, some additional multiple-choice answers were added to the questionnaire. Also, an allowance was made for optional, open-ended comments to be provided for any given question.

4.1.3   Selection and Recruitment of Respondents

Software engineering organisations were selected from the online membership list of the Australian Information Industry Association (AIIA)[1]. The website of each member organisation was consulted to determine whether they plausibly developed software, in which case their contact details were recorded.

Initially, organisations were contacted by email. An invitation to participate in the survey was sent to 250 email addresses. However, where multiple email addresses were available for the same organisation, sometimes all of them were used. This may have contributed to potential respondents regarding the mail-out as spam. This initial mail-out received 13 responses.

A second mail-out was conducted using organisations' street or postal addresses. This used a list of 217 addresses, produced by adding or removing organisations depending on the availability of their physical addresses compared to email addresses and whether they had already responded. Only one letter was sent to each organisation, and each envelope contained a piece of confectionery intended to compensate respondents for their time. The second mail-out received 18 responses, including two from organisations not directly contacted. (10 envelopes were returned due to the intended recipient having left the address, or for other, unspecified reasons.)

---

[1]http://aiia.com.au

Thus, the survey collected data from 31 organisations.

4.1.4   Classification Scheme

Responses were stored initially in a flat-file format, which included verbatim all input given by respondents. This data was classified in a semi-automatic process that entailed entry into a database. The classification scheme was not a broad specification for classifying arbitrary responses in the vein of traditional coding schemes, but rather a precise mapping of raw input values to classifications. Classification was conducted by a script utilising this mapping.

The development of such a scheme overlapped with the coding process itself, since human judgement was required to update the mapping to accommodate new data. This approach is advantageous for the following reasons:

- two identical responses are automatically given the same classification;
- two equivalent responses will be documented beside each other, and so are unlikely to be classified differently; and
- the classification scheme can be freely modified even after responses have been classified, without needing to manually re-code data.

Many questions on the questionnaire allowed respondents to enter additional answers not accounted for among the multiple-choice options given. For questions where this was not allowed, the mapping was trivial.

In three cases, there was a preliminary, manual component to the coding process. Each of these is discussed below.

1. One respondent did not select an option for the question "On average, how often are inspections, reviews or walkthroughs carried out?" As a result, the questionnaire disallowed answers to all other peer review related questions. After an email exchange, the respondent submitted a second response, containing answers to the peer review questions. These were manually merged into the original response.

2. In response to the question "Which of the artefacts are subject to regular inspection/review/walkthrough?" one respondent disregarded the multiple-choice options and wrote "all" in the free-form comment section. The response file was edited to remove the comment and add a "yes" answer for each artefact.

3. The automated script was unable to process one comment due to formatting

Table 4.1: The exact wording of the options for the inspection techniques question (i.e. "For the artefacts under inspection in a typical inspection/review/walkthrough process, inspectors:").

Technique	Description (Inspectors...)
Checklists	"...consult a checklist of potential defects".
Use case scenarios	"...traverse the artefacts according to use case scenarios".
Verification tool(s)	"...use a software tool to automatically check for certain types of defects".
Artefact cross-referencing	"...frequently reference other documents / diagrams / code not under inspection".
Different perspectives	"...are each asked to examine the artefacts from different perspectives".
Visualisation tool(s)	"...use a software tool to visualise or highlight aspects of the artefacts".
Artefact re-creation	"...actively create other documents / diagrams / code".
Detailed procedure	"...follow a detailed procedure (other than simply a checklist)".
Abstraction	"...actively re-create or reverse engineer abstract descriptions".

limitations. The comment was manually re-formatted.

4.2 Focus of Analysis

The survey encompassed a wide range of software engineering topics. The focus of this chapter is on analysis of the subset of survey questions that help determine how peer reviews are used in industry, and how their use might be improved. Those questions are listed in Figure 4.1.

Answers to questions regarding peer review frequency, length and the number of times an artefact is reviewed help illustrate the extent to which reviews are being conducted. Respondents reported the relative efficacy of peer reviews compared to testing, effectively giving a measure of the importance of peer reviews to surveyed organisations. As shown in Table 4.1, respondents were also asked about the current use of reading techniques and tools in peer reviews. This indicates what peer review mechanisms are being successfully applied, and by implication where further research could be directed.

Questions related to the use of specific programming languages, diagrammatic notations and textual artefacts help determine which of the existing reading techniques, if applied, might be able to improve peer review efficacy across a large range of organisations. Respondents were asked which of these artefacts were subject to review, indicating where opportunities for different/additional peer reviews may lie. Questions concerning the use of development tools — particularly IDEs and CASE tools — help illustrate the environments in which particular peer review mechanisms can operate.

Contextual Questions
- What domain(s) does your organisation / department / team develop software for?
- What type(s) of software are typically produced?
- How is your software typically used or supplied?
- At any one time, how many distinct, significant software projects are typically undertaken by your organisation/department/team?
- How many people are assigned to a typical software project?
- How many people are assigned to maintain a typical software system?

Inspection Characteristics
- On average, how often are inspections, reviews or walkthroughs carried out?
- On average, how many times is an individual artefact subjected to an inspection/review/walkthrough?
- On average, how long does each inspection/review/walkthrough take?
- Compared to testing, how cost-effective are the inspections/reviews/walkthroughs conducted within your organisation/department/team?)
- The answer above is based on: . . . *Respondents select "Informal observation / experience", "Formal quantitative analysis" or "Not applicable".*
- For the artefacts under inspection in a typical inspection/review/walkthrough process, inspectors: . . . *Respondents select one or more review techniques.*
- What development methods(s) are typically used? *Respondents select one or more options, one of which is "Formal Software Inspection (with defined roles and stages)".*

Development Phases
- List each of the distinct phases (if any) that software projects typically go through.
- How much effort does each phase of a typical project demand, as a percentage of total project workload?

Artefacts
- What programming language(s) are regularly used?
- What diagrammatic notation(s) are regularly used to model software?
- What textual artefact(s) are used to describe software?
- Which of the artefacts are subject to regular inspection/review/walkthrough?
- Are specific standards set for each artefact? *For each artefact, respondents select "standard formatting/layout", "standard creation/derivation process", both or neither.*
- Which artefacts are actively developed OR referred to at each phase of a typical software project?

Tool Support
- What tool(s) are typically used to create source code?
- What tool(s) are typically used to create software diagrams?

Figure 4.1: The subset of survey questions relevant to software peer review.

Though the raw tallies of responses for many questions are directly of interest, some additional derived metrics also serve to illustrate how reviews are or could be applied to different phases and different artefacts. Such quantities include:

- the average estimated effort spent in a given phase of a project;
- the average proportion of a project (by workload) in which a given artefact is used — *artefact prevalence*;
- the number of different types of artefacts used in a given phase — *artefact diversity*;
- the most commonly-used combinations of artefact types; and
- the proportion of artefacts used in a given phase that are reviewed — *reviews-by-phase*.

For each phase, respondents indicated one of several set ranges of values (e.g. 20–30%, 40–60%, etc.). To determine average effort for a given phase, each range first had to be resolved into a scalar estimate. For each response, estimates were chosen from within each range such that their sum was 100%. If $\boldsymbol{\ell}_r$ and $\mathbf{h}_r$ are vectors of equal length representing the lower and upper bounds on effort for each phase, for response r, then the vector of effort estimates $\mathbf{f}_r$ can be calculated as follows:

$$m_r = \frac{100 - \sum \boldsymbol{\ell}_r}{\sum \mathbf{h}_r - \sum \boldsymbol{\ell}_r}$$

$$\mathbf{f}_r = \boldsymbol{\ell}_r + m_r \left(\mathbf{h}_r - \boldsymbol{\ell}_r\right) \tag{4.1}$$

The variable m_i represents the proportion of each range that must be added to each lower bound, in order for the sum of the estimates to equal 100%. For example, respondent r might list three phases, with the associated effort ranges 10–20%, 20–30% and 60–80%. Thus, $\boldsymbol{\ell}_r = (10, 20, 60)$ and $\mathbf{u}_r = (20, 30, 80)$. By the above equations, $m_r = 0.25$ and so $\mathbf{f}_r = (12.5, 22.5, 65)$. The effort estimates are therefore 12.5%, 22.5% and 65% of project workload, for the three phases, which add up to 100% and all lie within their respective ranges. Even if the sum of the lower bounds exceeds 100% or the sum of the upper bounds falls short of 100%, estimates can still be derived, outside but in the vicinity of the given ranges.

Since respondents listed their own phases, some of which were more fine-grained than others, these had to be aggregated before any averaging could be done. Six *phase groups* were chosen to represent the most commonly-appearing phases:

1. analysis/requirements (representing any analysis, requirements, quotation and planning phases);
2. architecture/design;
3. development;
4. testing/QA (representing any QA, inspection, test planning, unit testing, integration testing, system testing, acceptance testing or generic testing phases);
5. delivery; and
6. support/maintenance.

Where several of a respondent's listed phases fell under a given heading, the relevant effort estimates were added.

These effort estimates are prone to being slightly inflated. The inflation arises where respondents omit one phase but include its associated activities in another phase. This is mitigated by aggregating phases. However, for instance, it still means that some amount of the average development effort is probably testing effort, because respondents who omit testing as a phase likely include testing in their estimates for development. This amount is not subtracted from average testing effort because the latter is derived only from respondents who do list one or more testing phases. The magnitude of this inflationary effect is discussed in Section 4.4.2.

Artefact prevalence combines the effort estimates with each respondent's indication of which artefacts are used or referred to in each phase. Based on the set of artefacts and phases a respondent listed, the online questionnaire asked respondents to match one to the other. The result is a binary vector $\mathbf{u}_{ra}$ for each respondent r and artefact type a, containing a value for each phase. A value of one indicates that the given artefact is relevant in the given phase, while zero indicates otherwise.

To calculate artefact prevalence, the effort estimates for each phase in which a given artefact is used are added, yielding the proportion of the project during which the artefact is of interest. In the context of the previous example (where respondent r's effort estimates are 12.5%, 22.5% and 65% for three different phases), an artefact used in the first two phases would have a prevalence of $12.5\% + 22.5\% = 35\%$. That is, 35% of project workload is potentially related to that artefact. Another artefact used in the last two phases would have a prevalence of 87.5%.

These values are then averaged across all applicable responses. This is reflected in the following equation, where $n = 31$ is the total number of respondents and n_a is the number of respondents who listed artefact a:

$$\text{Prevalence}(a) = \frac{1}{n_a} \sum_{r=1}^{n} \mathbf{u}_{ra} \cdot \mathbf{f}_r \tag{4.2}$$

No aggregation of phases is necessary for $\mathbf{u}_{ra}$ here, because the phases are removed by the dot product before the results for different responses are averaged. However, aggregation of phases is required for artefact diversity, artefact combinations and reviews-by-phase. Therefore, $\mathbf{u}'_{ra}$ is defined to be a fixed-size version of $\mathbf{u}_{ra}$, in which phases have been aggregated into the six phase groups.

Artefact diversity is obtained by summing values in $\mathbf{u}'_{ra}$, and then averaging over the responses. First, the number of artefacts used in phase j, according to respondent r, is calculated as follows:

$$y_{rj} = \sum_a u_{raj}$$

If n_j represents the number of respondents who listed phase j, then the average number of artefacts used in each phase (artefact diversity) is:

$$\text{Diversity}(j) = \frac{1}{n_j} \sum_{r=1}^{n} y_{rj} \tag{4.3}$$

Artefact diversity and reviews-by-phase must be calculated on a phase-by-phase basis because, even once phases are aggregated, different phase groups vary substantially in the number of responses to which they apply. Dividing the sum by the total number of responses n rather than n_j would produce underestimates for the less common phases.

Artefact combinations are obtained by first taking the list of artefacts used in each phase in each response. For each such list, every possible x-way combination of those artefacts is generated, for $x > 1$. For each response, the union of these sets of combinations is then taken, so that each surveyed organisation has a single list of artefact combinations. Thus, artefact combinations are not merely a statement that particular sets of artefacts are used by the same organisation — artefacts in combination must also be used in the same development phase. The number of responses in which each artefact combination appears is counted. If an x-way combination is a subset of an $(x + 1)$-way combination and both are equally common, the x-way combination is ignored.

Reviews-by-phase relies on whether a surveyed organisation conducts peer reviews for
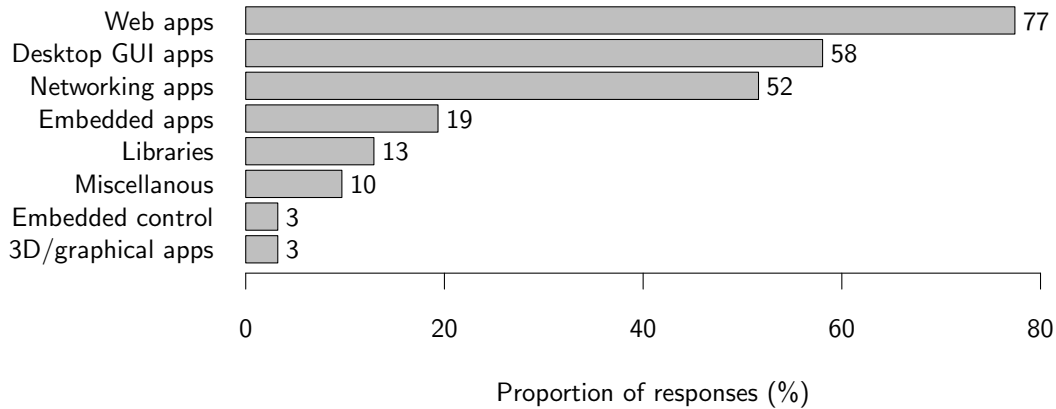
Figure 4.2: Software types.

a given artefact. The latter information (*reviews-by-artefact*) is represented by the binary variable v_{ra} for each respondent. The average proportion of artefacts reviewed in phase j can be calculated as follows:

$$\text{Reviews-by-phase}(j) = \frac{1}{n_j} \sum_{r=1}^{n} \frac{u_{raj} \cdot v_{ra}}{y_{rj}} \tag{4.4}$$

4.3   Surveyed Organisations

As a result of the two mail outs, 31 responses were received from organisations around Australia. These responses were intended to represent a broad and unbiased sample of software organisations. Any substantial biases should be determinable from the responses to the first six questions listed in Figure 4.1. This section describes the overall characteristics of the surveyed organisations, so that the results and discussion that follow can be interpreted accordingly.

26% of respondents said they were not restricted to any particular domain, though some of these did also indicate specific domains. The same number indicated they were involved in developing software for business or a generic business-related concern (payrolls, finance, e-commerce or enterprise). 19% developed software for government or the public sector. Other domains given by multiple respondents included accounting, banking, healthcare, insurance, media and telecommunications. Domains listed by only one respondent included automotive, aviation, distribution, education, environment, gaming, IT, manufacturing, mining, printing, retail, security and transport.

Figure 4.2 shows the proportion of organisations that develop each of a list of software
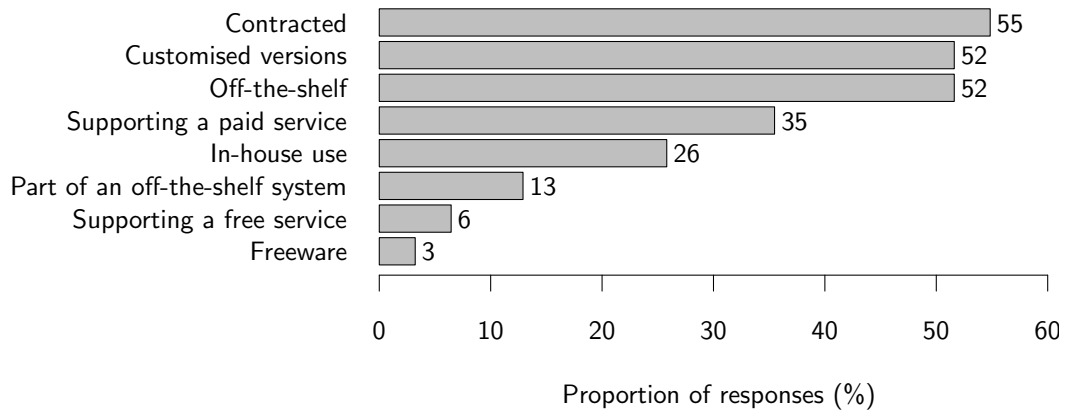
Figure 4.3: Software use/distribution models.
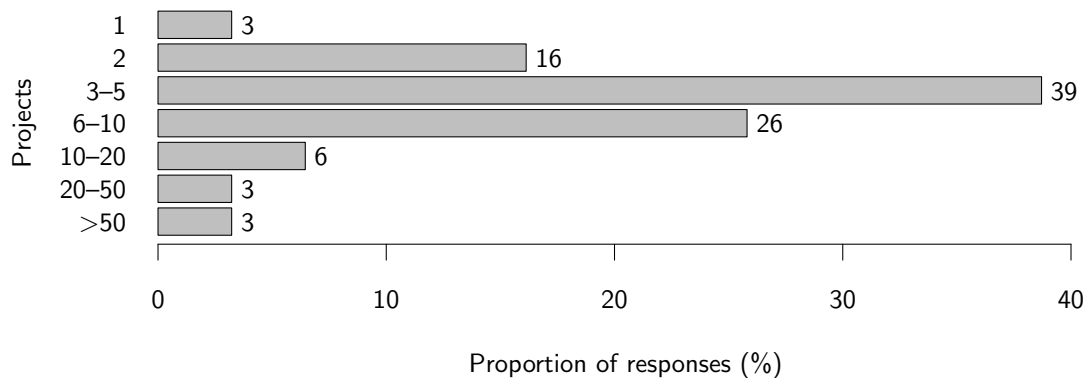
Figure 4.4: Typical number of concurrent software projects.

types. Three quarters of all respondents develop web applications, compared to about half who develop desktop GUI applications and networking applications.

Figure 4.3 shows the relevance of a fixed set of use/distribution models to respondents. For most respondents, two or more options were applicable. About half the surveyed organisations develop software for a given external organisation as per a contract, while half supply customised versions to different organisations, and half develop stand-alone off-the-shelf software packages. Very few respondents indicated distribution free-of-charge to the general public, or in support of a free service.

About a third of respondents said their software supports a paid service. This may include a diverse range of services, from network-based services to support/maintenance contracts.

As Figure 4.4 shows, most respondents reported 2–10 concurrent software projects, though a few had considerably more. As shown in Figure 4.5, most also reported development team sizes of 2–10 people, with the majority reporting 2–5 people. Maintenance
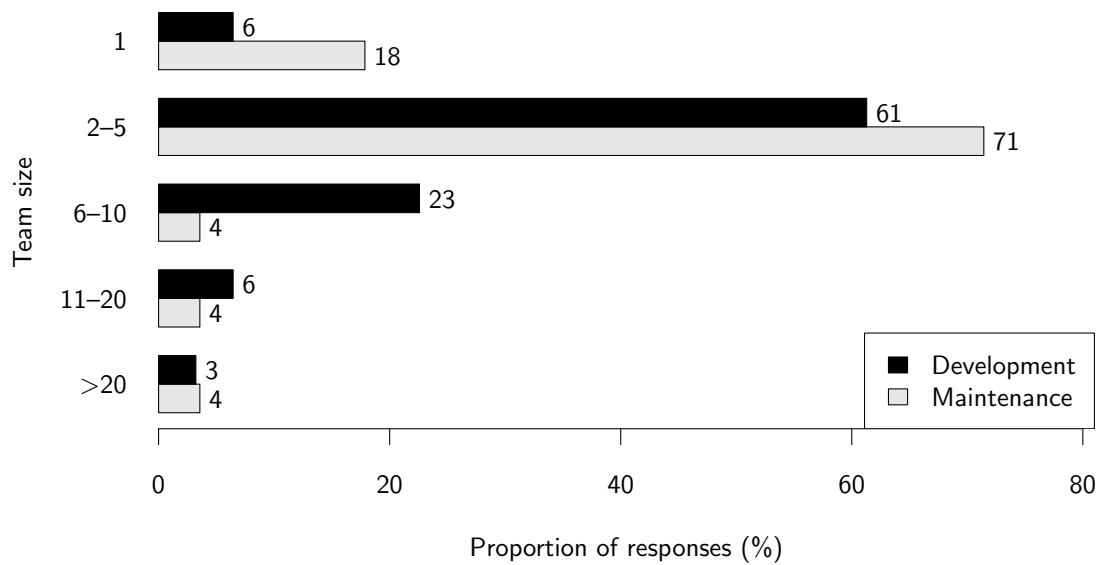
Figure 4.5: Development and maintenance team sizes.

team sizes were mostly 2–5 people, with some consisting of a single person. No data was collected on staff numbers, because this can be ambiguous where organisations are not primarily focused on software development.

No data was collected on the individuals actually filling out the questionnaire (other than their contact details, in confidence).

4.4 Results

Most of the figures referenced in this section show data on the proportion of respondents who gave particular answers. Where aggregate categories are shown (such as "others"), the values are not generally the sum of all aggregated answers, because a given respondent may select multiple answers.

Comments made by respondents are also described here. For each question, only a small minority of respondents (if any) provided a comment. Thus, this is a fundamentally qualitative aspect of the survey, and the number of comments making a given argument or reflecting a given point of view has no particular meaning. These comments provide insight into more subtle aspects of the issues covered by the questionnaire.
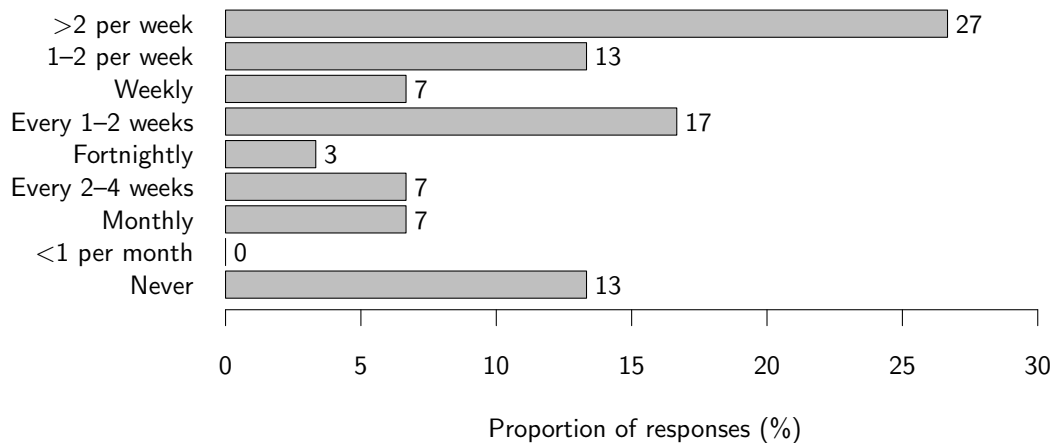
Figure 4.6: Frequency of peer review activities.
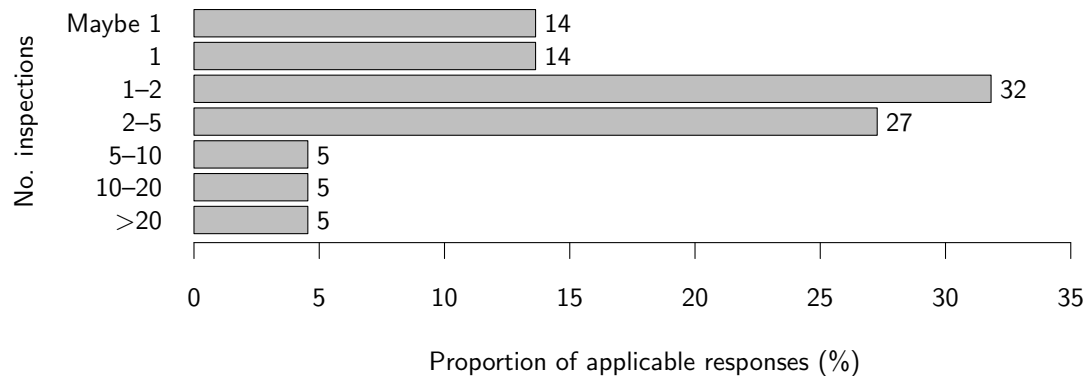
Figure 4.7: Inspections/reviews for a typical artefact.

4.4.1 Overall Peer Review Characteristics

39% of respondents said their organisation conducts formal Software Inspection, with defined roles and stages. However, peer review activities in general are more widespread.

Figure 4.6 shows the frequency of peer review activities. Two thirds of respondents said they conduct inspections, reviews or walkthroughs more than once per fortnight. 40% said these occur more than once per week. Several respondents commented that peer reviews are done after a task has been completed, and so their frequency depends on the project. One remarked that "everyone agrees [peer reviews] should be more regular but deadlines interfere". Answers to this question were used to determine the number of respondents who perform some kind of peer review. For the other questions reported in this subsection, the percentages shown are of respondents who perform peer reviews, not the total number of respondents.

As shown in Figure 4.7, a majority of respondents said that artefacts are typically
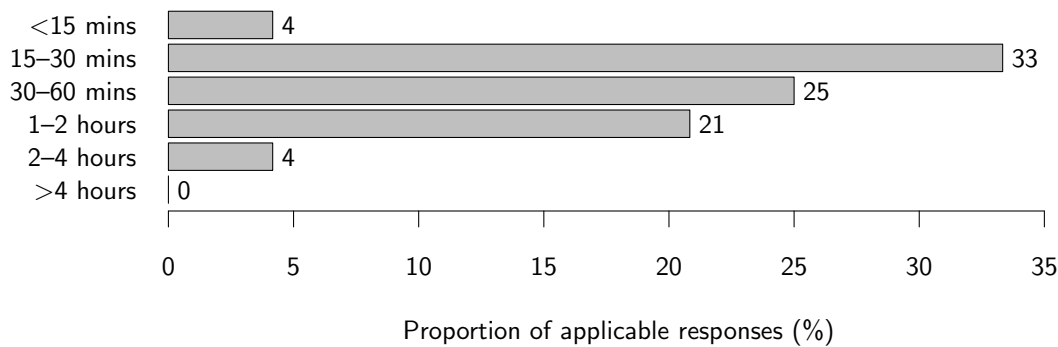
Figure 4.8: Length of peer review activities.
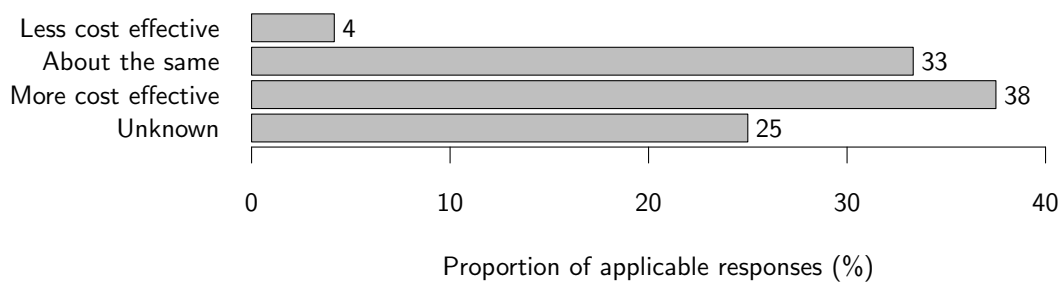
Figure 4.9: Cost effectiveness of peer review activities compared to testing.

inspected 1–5 times. One commented that this depends on the artefact. Another commented that "Once is usually enough, but if more is required then we take stock and ask ourselves why!"

The lengths of peer review activities is shown in Figure 4.8. For most respondents, peer review length lies between 15 minutes and two hours. Two thirds of respondents said that inspections, reviews or walkthroughs lasted less than an hour. Three respondents said that the length depends on the artefact. One commented that inspection consumes about half the original coding time, when all components of the inspection process are included.

Figure 4.9 shows the relative efficacy of peer review activities compared to testing, as reported by respondents. A quarter indicated that it was not known which is more effective. Of the rest, half said inspections were more effective, and half said they were about the same. One indicated they were less effective. When asked whether their responses were based on informal observation/experience or formal quantitative analysis, all respondents indicated the former (where applicable).

One respondent questioned the meaningfulness of a comparison of peer review and testing, given that they are done at different stages of a project. Another pointed out that "Neither inspections nor testing will cover everything, and while often there
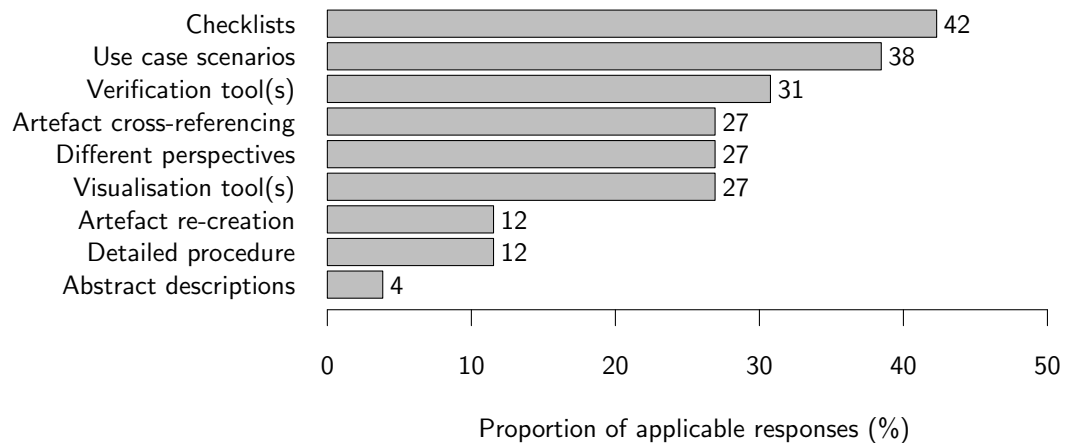
Figure 4.10: Inspection techniques.

is a considerable overlap, both are required." A third commented that testing does not facilitate learning as easily as peer review. Finally, one respondent remarked that high-level walkthroughs done after analysis are essential as a means of agreeing on and checking design, but that code walkthroughs — while a good idea — do not have the same level of importance, and there is generally less pressure from management to conduct them.

Respondents' use of a range of peer review techniques is shown in Figure 4.10. The exact wording of the options as shown to respondents is given in Table 4.1. Most techniques were used by a substantial minority of respondents. Checklists were used by just under half of all those who performed peer reviews, while traversal of use case scenarios was done by almost the same number. One respondent remarked that different strategies are applicable to different artefacts; that in their case a code review done at each check-in takes one person ten minutes while reviews of higher-level documentation are aimed at achieving consensus between different stakeholders.

4.4.2   Development Phases

10% of respondents did not list any specific set of phases through which software projects progress. Those who did listed up to 11 distinct phases. Examples of the phases given, after categorisation but before phase aggregation, are as follows:

- requirements → development (the simplest case);
- analysis → requirements → design → development → testing → delivery; and
- requirements → prototyping → quotation → architecture/design → development → unit testing → QA → documentation → acceptance testing → delivery →
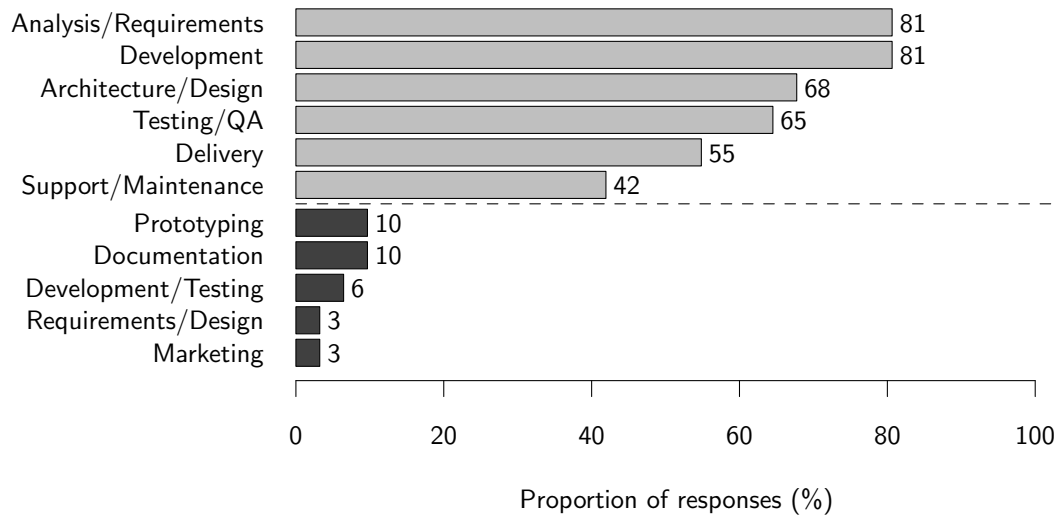
Figure 4.11: The proportion of respondents who listed different phases. The top six bars represent the aggregated phase groups, while the others represent phases not included in any phase group.
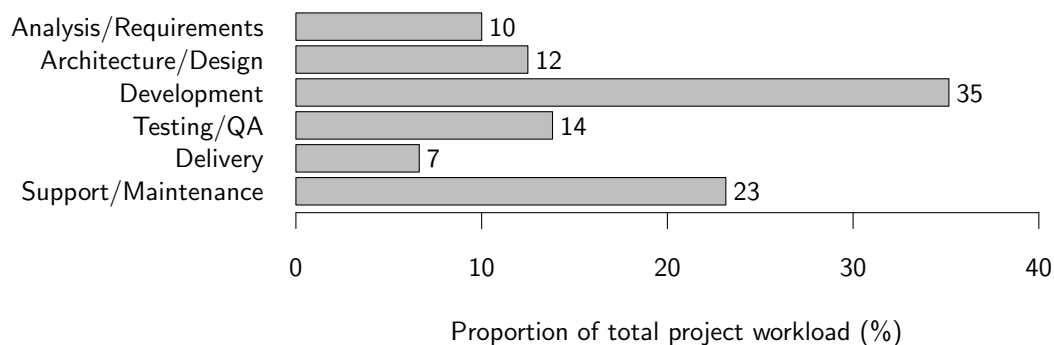
Figure 4.12: Mean derived estimates for effort spent in different phases.

support/maintenance (the most complex case).

The absence of certain phases (such as testing) from a list does not necessarily indicate that activities associated with those phases are not performed. Those activities may instead fall under more broadly-defined phases.

As shown in Figure 4.11, there is a substantial gap between the least common phase group (support/maintenance) and the most common non-included phases (prototyping and documentation). Phase groups cover all phases listed by more than 10% of respondents.

Figure 4.12 shows the mean effort estimates for different phase groups, while Figure 4.13 shows their distribution. The inflationary effect discussed in Section 4.2 can explain why the sum of the mean effort estimates is greater than 100%.
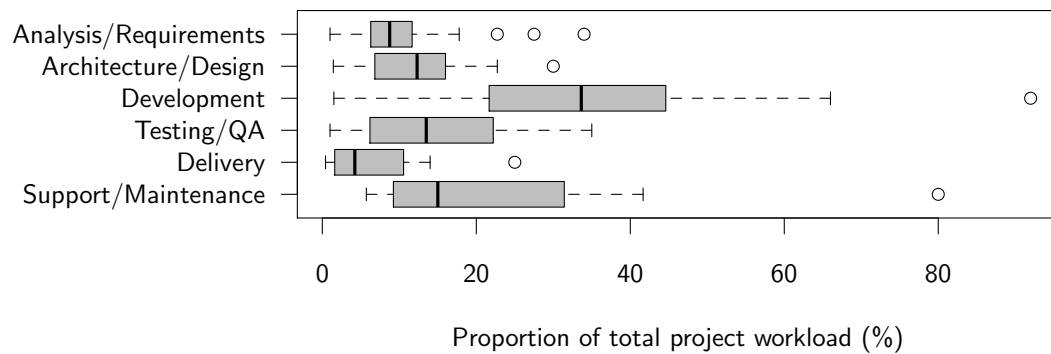
Figure 4.13: Distribution of derived effort estimates for different phases.

To examine the potential size of this inflationary effect, it is assumed that any respondent who listed the development phase but not a testing/QA phase instead included testing effort under development. There were six such respondents — 24% of all those who listed the development phase. If it is further assumed that those surveyed organisations undertake the average level of testing (14% of the project workload), then the development effort shown in Figure 4.12 may be inflated by 3% of the project workload (0.24×0.14). This is not a meaningful figure, given that respondents chose from ranges of values spanning 5%, 10% or 20% of project workload.

The effect is difficult to examine in terms of other phases, because equivalent assumptions cannot necessarily be made. Where the architecture/design phase is omitted, it is not certain whether design is included in the analysis/requirements or development phases, or is not done in any meaningful sense at all. Where the support/maintenance phase is omitted, this may be because it is not considered part of the same project.

Effort expended in development is substantially greater than in other phases, though it does not amount to a majority of total workload. About a quarter of project workload is spent in maintenance, while about half that again is consumed in each of the analysis/requirements, architecture/design and testing/QA phases. Delivery accounts for a relatively small but non-trivial amount of project workload.

4.4.3   Artefacts

The programming languages used by surveyed organisations are shown in Figure 4.14. Of all languages, SQL and JavaScript were used by the highest proportion of respondents, suggesting extensive development of web-based and/or database-driven applications. Java and C# had substantially more widespread use than C and C++, which themselves had only slightly higher use than the set of scripting languages commonly associated with web development on open-source platforms — Perl, PHP, Python and
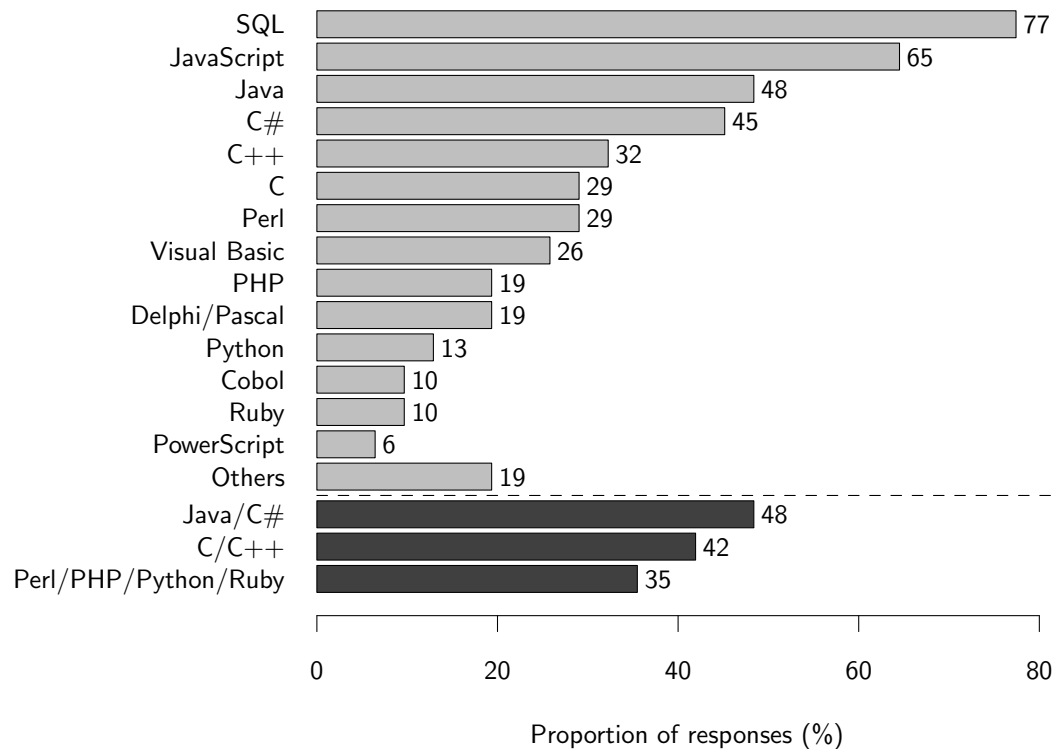
Figure 4.14: Regularly used languages.

Ruby. Languages reported by only one respondent included Common Lisp, Objective C and Progress. One organisation also used a custom-designed language.

Figure 4.15 shows the proportion of surveyed organisations that use different diagrammatic notations. The extensive use of entity relationship diagrams (ERDs) is commensurate with the use of SQL. Flow charts were used by as many surveyed organisations as UML class diagrams (slightly less than half), though conceivably not to represent the software itself. One respondent commented that flow charts are not actually used for development, but for "explaining procedures to users". Use of specific UML notations other than the class diagram was relatively low. Only about a quarter of surveyed organisations used UML use case or sequence diagrams, while other UML diagram types were less prominent still. This compared to the use of data flow diagrams (DFDs) by 35% of surveyed organisations. Two respondents indicated that their organisations were open-minded about the types of diagrams or the level of detail needed.

The use of specific types of textual documents is illustrated in Figure 4.16. About three quarters of surveyed organisations construct a formal specification/requirements document. The majority construct use case scenarios (which are not necessarily separate from the specification).
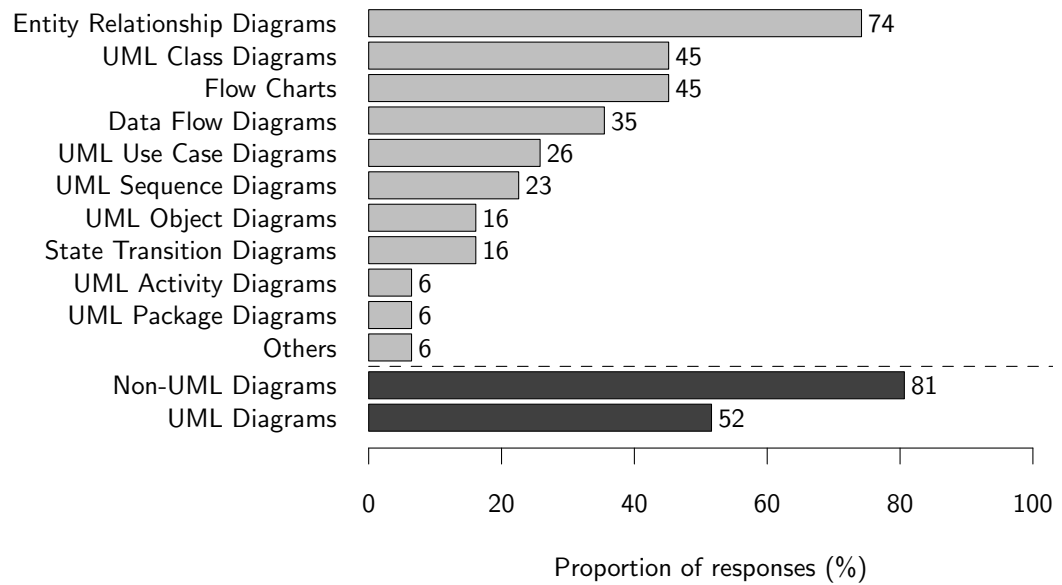
Figure 4.15: Regularly used diagram types.
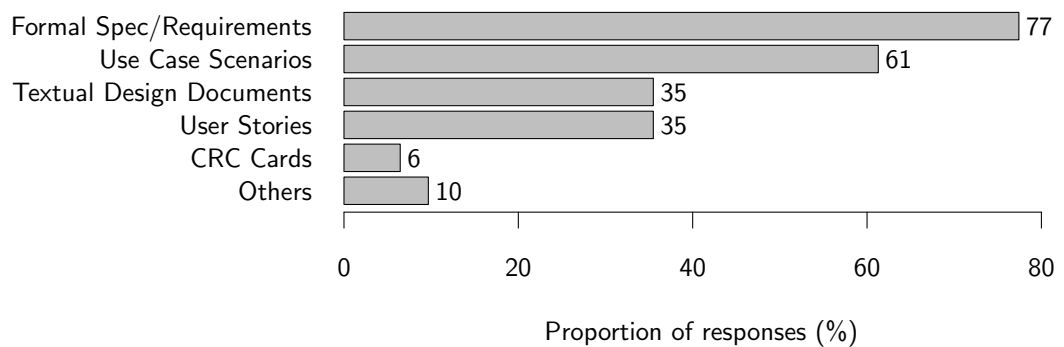
Figure 4.16: Regularly used textual artefacts.

Those languages, diagram notations and other documents used by at least 25% of surveyed organisations will be referred to as *common* artefact types. Metrics associated with the less common varieties are less reliable due to the small sample size.

Standardisation of artefact formatting/layout and creation/derivation is shown in Figure 4.17. For almost all artefact types, formatting/layout standards were more common than creation/derivation standards; in many cases twice as common.

4.4.4 Artefact Usage

Artefact prevalence is shown in Figure 4.18. About half the common artefact types fall within 40–47% of project workload. Artefacts associated with analysis and requirements make up the top three, while C code is the least prevalent type of artefact.

Figure 4.19 shows artefact diversity, which ranges from an average of 2.7 different artefact types in analysis/requirements up to 7.2 in development.

Combinations of artefacts used in some development phase are shown in Figure 4.20. In total there were 4,890 distinct artefact combinations (excluding subsets as discussed in Section 4.2), but this defies meaningful analysis. Therefore, only those combinations reported in at least 25% of responses are shown, and all of these are 2-way, 3-way or 4-way combinations. From the 20 artefact combinations shown, 15 involve the formal specification or requirements document, while 9 involve entity relationship diagrams and 8 involve SQL. All the combinations shown are combinations of these three artefact types plus at most one other.

4.4.5 Peer Review Usage

The level of reviewing undertaken for each type of artefact is shown in Figure 4.21. The level of reviewing varies widely across different programming languages and different diagrammatic notations. Just over half of surveyed organisations that use formal specifications/requirements documents review them.

Figure 4.22 shows the proportion of artefacts reviewed in each phase. The analysis/requirements, architecture/design, development and testing/QA phases are all approximately equal in this respect. The delivery phase has a substantially higher proportion of reviewed artefacts, while a much smaller proportion are reviewed in support/maintenance.
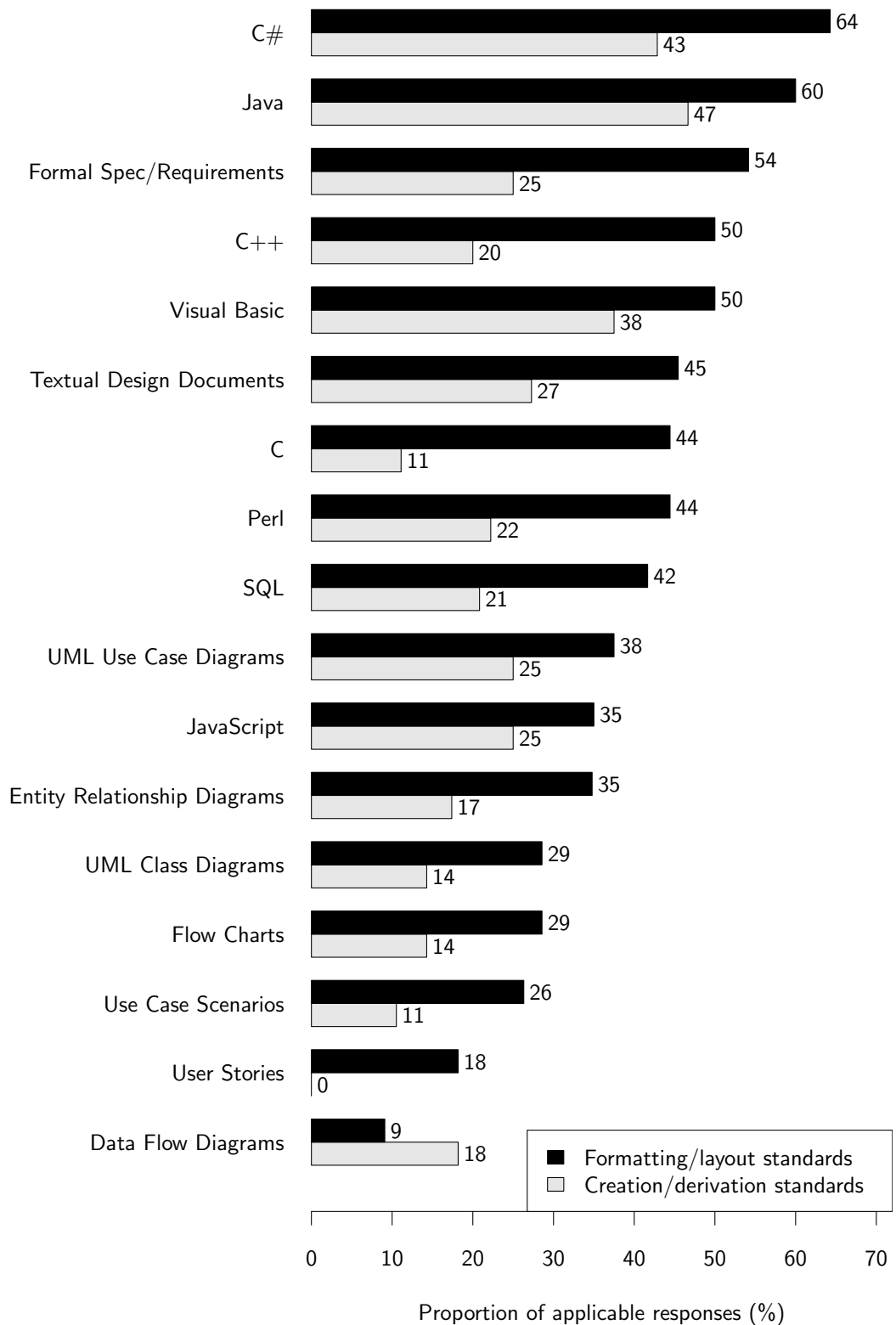
Figure 4.17: The use of formatting/layout and creation/derivation standards for common artefact types.
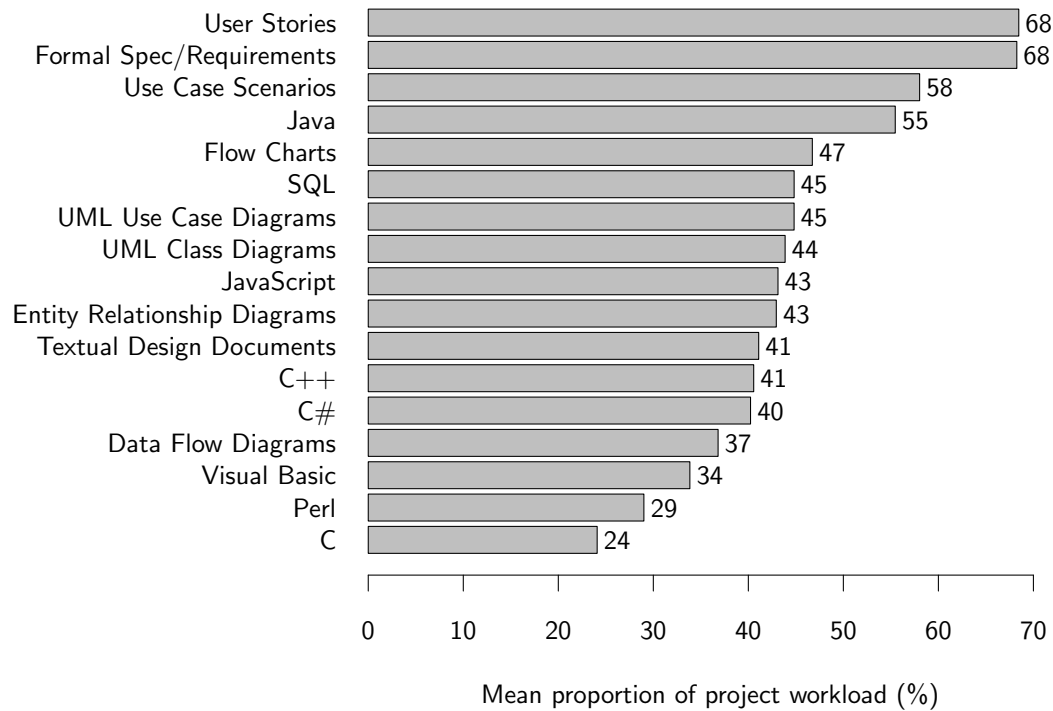
Figure 4.18: Artefact prevalence — the mean proportion of the project (by workload) during which each common type of artefact is used (i.e. developed or referred to).
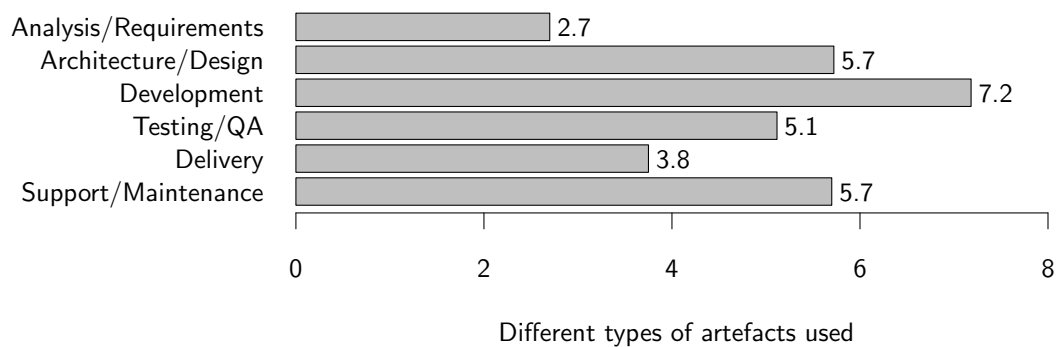
Figure 4.19: Artefact diversity — the mean number of different types of artefacts used in each development phase.
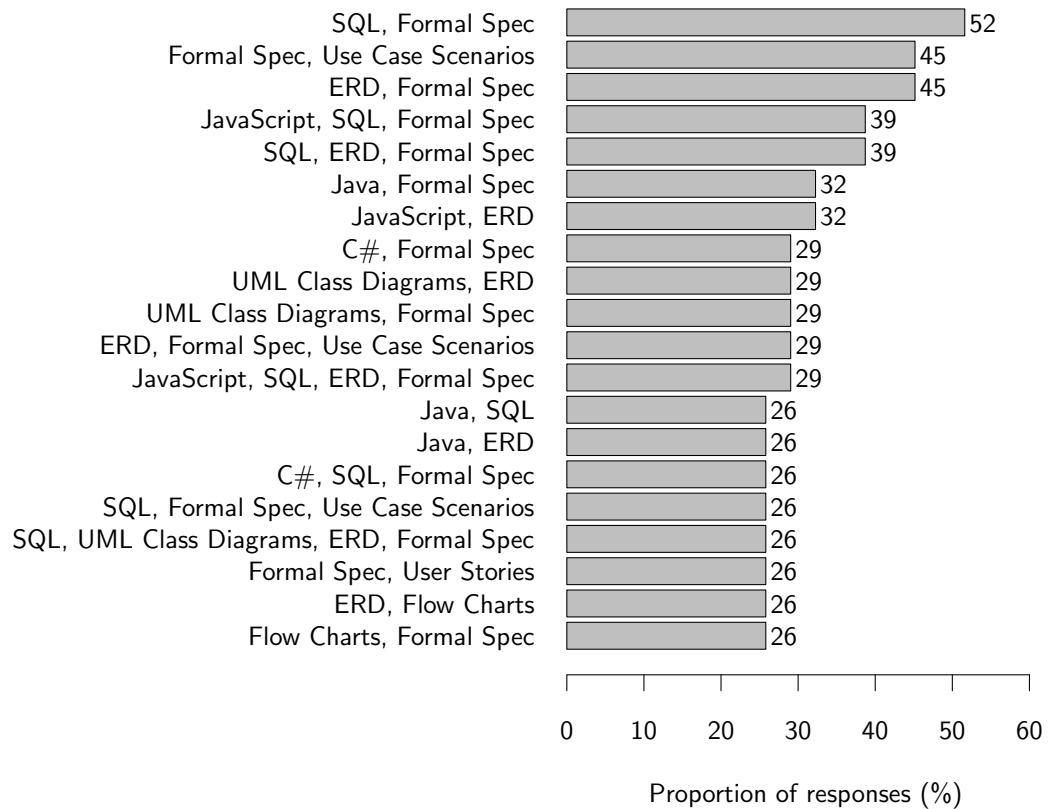
Figure 4.20: The use of several artefacts in combination in some phase. (Here, "Formal Spec" also includes general requirements documents.)
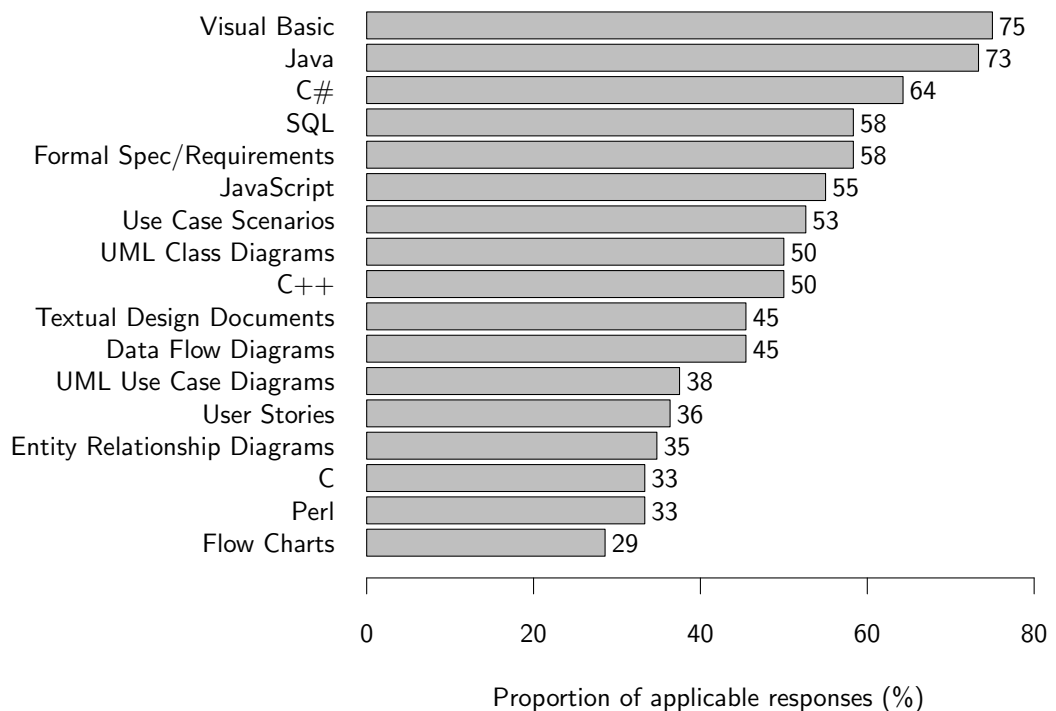
Figure 4.21: Reviews-by-artefact — the number of surveyed organisations that review each common type of artefact, as a proportion of organisations that use them.
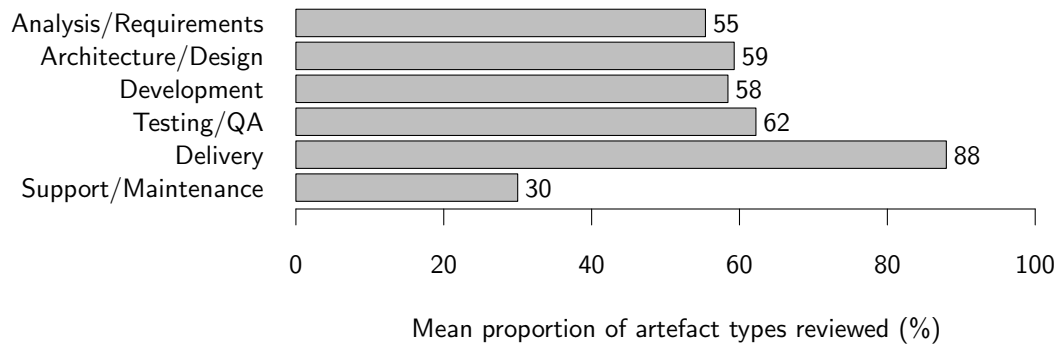
Figure 4.22: Reviews-by-phase — the mean proportion of artefact types reviewed in each development phase.
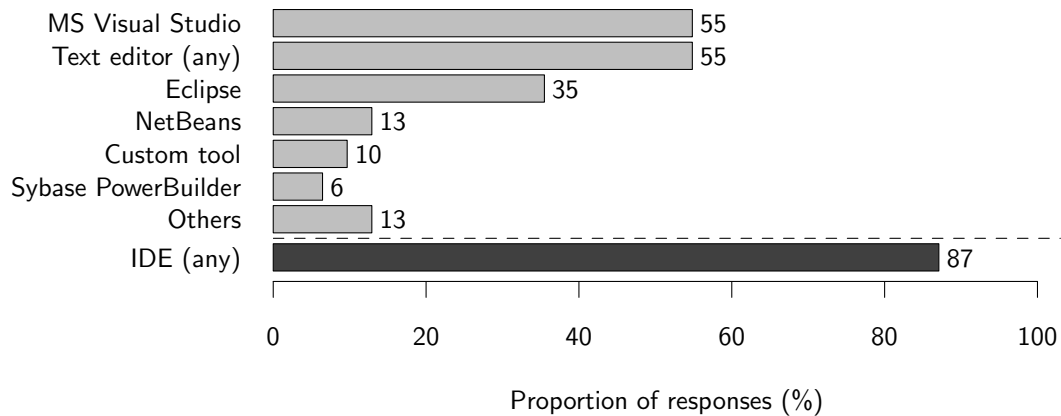
Figure 4.23: Source code creation tools.

4.4.6 Tool Support

Figure 4.23 shows the tools used in the construction of source code. Predominantly these are IDEs and text editors, IDEs being almost ubiquitous. Tools reported by only one respondent included Borland Delphi, IntelliJ IDEA, Microsoft SQL Server Management Studio and xdoclet. A small number of respondents reported using multiple IDEs — three listed both Microsoft Visual Studio and Eclipse, while two listed both NetBeans and Eclipse.

Tools used in the construction of software diagrams are shown in Figure 4.24. Notably, CASE tool usage is much less pervasive than drawing tools and non-computerised diagrams. Here, tools reported by only one respondent included Metastorm Provision, Dia, Microsoft SQL Server Management Studio, Microsoft Word and OpenOffice.
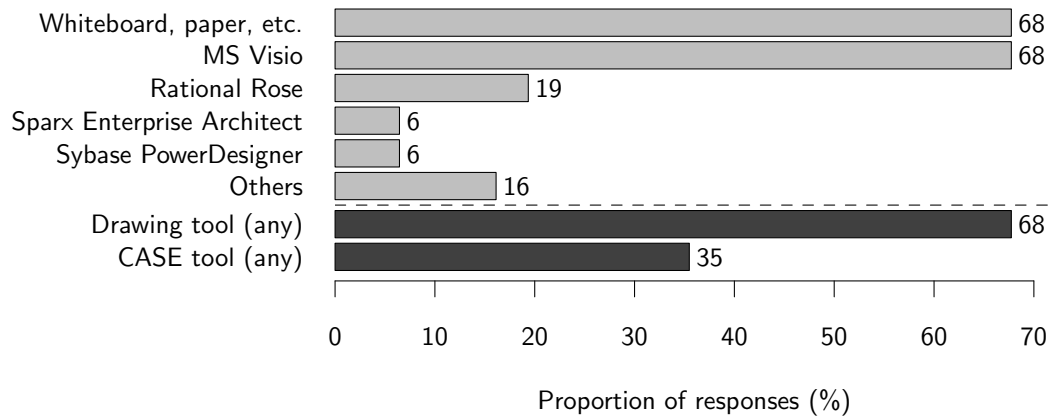
Figure 4.24: Diagram creation tools. (CASE tools included Rational Rose, Sparx Enterprise Architect, Sybase PowerDesigner and Metastorm Provision.)

4.5   Discussion

The organisations participating in this survey included a large number that develop web-based and/or database-driven applications. Correspondingly, the most widely-used languages were JavaScript and SQL, and the most widely used diagrammatic notation was the entity relationship diagram (used by three quarters of respondents). It is not certain that this reflects the broader software development industry. However, absent any identifiable cause for bias in the selection of respondents, the respondents' organisations must at least represent a large segment of the industry.

4.5.1   Overall Peer Review Practice

From the peer review information outlined in Section 4.4.1, most organisations appear to understand the benefits of software peer reviews, even if they do not have quantitative data to support them. Most undertake peer review activities of some kind, and there is evidence in Figure 4.10 that elements of reading techniques proposed in the academic literature (outlined in Chapter 2, Section 2.2) are being implemented.

Most respondents' peer reviews were under the two-hour time limit recommended by Fagan (1976) for formal Software Inspection. Many respondents reported that less than half an hour is spent in peer reviews, raising the question of whether reviews of such short durations can be effective. Almost all respondents were positive or neutral about the effectiveness of peer reviews compared to testing. However, the absence of formal quantitative analysis suggests that little emphasis is being placed on objectively verifying peer review efficacy. There may be little motivation for an organisation to alter the peer review process without an objective evaluation of existing techniques as

used within the organisation. If so, this would represent a barrier to the adoption of new reading techniques.

The results presented in the previous section suggests that there are opportunities to improve peer review practices, with respect to comprehension, reading techniques and the overall peer review strategy. The effort associated with the testing/QA, delivery and support/maintenance phases is about 44% of total project workload, which occurs after the phases in which the software is actually built. Reworking defects found in these phases may contribute substantially to this figure. If those same defects had been discovered nearer the point of origin (i.e. in the analysis/requirements, architecture/design or development phases), much of this rework effort might be saved.

About 60% of artefact types are being reviewed in the analysis/requirements, architecture/design and development phases, much fewer than are reviewed in the delivery phase. A higher review rate in earlier phases is certainly possible, and this would increase the number of defects detected. However, effective peer reviews of a given artefact type depends on the properties of the notation itself, not just on the motivation of the organisation. This is reflected in Figure 4.21, which shows widely differing review rates for different types of artefacts. The availability of an appropriate reading technique is likely a factor influencing the propensity of organisations to review a given type of artefact.

4.5.2   Tools and Techniques

From Figure 4.10, the most prevalent techniques remain checklist-based ones, though they still account for less than half of all respondents. By contrast, traversing an artefact according to use case scenarios (one of the main principles of Usage-Based Reading — UBR) has gained almost the same level of support, despite it being a much more recent proposal in the academic literature. This may simply reflect use cases being a readily used and useful inspection aid. More comprehensive forms of active guidance, such as entailed by variants of Scenario Based Reading (SBR), appear to have very little support, and abstraction-based techniques still less.

Only a quarter of respondents indicated that artefact cross-referencing takes place. With artefact diversity reaching 7.2 distinct artefact types on average in the development phase, this is likely indicative of the difficulty of cross referencing more than the lack of opportunities for it. The same number of respondents indicated use of visualisation tools that might help to facilitate artefact cross referencing and generally help address delocalisation. With a higher utilisation of such tools, it is reasonable

to suppose that artefact cross referencing would also be more prevalent. Certain defects, in the form of inconsistencies between multiple artefacts, are unlikely to be found otherwise.

Similarly under-utilised were verification tools, used in only a third of cases, despite Fagan (2002) recommending against checklists due to the existence of such tools.

4.5.3   Artefact Standardisation

Most reading techniques could be applied to code, given its high level of standardisation. However, code in many languages is already relatively well-reviewed, notable exceptions being C and Perl code. Other artefacts where reviews are lacking include flow charts, entity relationship diagrams, user stories and use case diagrams, and to a somewhat lesser extent data flow diagrams and textual design documents. These include a range of high-level artefact types, most of which are not well-standardised. Techniques most suited to reviewing relatively unstandardised artefacts (such as flow charts, use case scenarios, user stories and data flow diagrams, as shown in Figure 4.17) are likely to include those employing minimal active guidance.

Survey results indicate that about half of software development organisations adhere to standardised requirements documents. However, other than checklists, relatively few organisations utilise active guidance activities. As discussed above, checklists themselves were used by less than half of the organisations surveyed. From this, two inferences are possible:

- that requirements documents are not sufficiently standardised, possibly hindering more extensive use of checklists; and

- that more comprehensive forms of active guidance (for instance, various forms of Scenario Based Reading) are under-utilised, not taking advantage of existing standardisation of requirements documents.

The use of agile methods may limit the level of standardisation that can effectively be implemented. However, many high-level artefacts also tend to be used towards the beginning of a project, where the types of artefacts (and hence types of knowledge) present are relatively constrained. This in itself supports the case for active guidance when inspecting such artefacts.

4.6   Summary

This chapter has discussed the methodology, results and analysis of an industry survey. Software development organisations around Australia provided survey responses indicating the use of different artefacts and practices, particularly in the context of software peer review.

The aggregated results indicate that peer reviews are widely used, but that opportunities exist for improvement, with respect to:

- objective evaluating of peer review using quantitative data;
- reducing costs incurred in later project phases through additional peer review; and
- reviewing additional types of artefacts, particularly early in a project and also where standardisation occurs.

The survey found that checklists and use case traversal were two commonly used inspection techniques. Chapters 6 and 7 discuss empirical studies related to these techniques.

The survey also found that multiple artefact types are generally used in almost every phase of software development. An average of 7.2 distinct documents, notations or languages were being used in the main development phase itself. Such artefact diversity emphasises the importance of understanding artefact interrelationships and thus addressing delocalisation. Relatively limited use is made of visualisation tools, meaning that artefact interrelationships must often be understood by inspectors without support. The next chapter examines artefact interrelationships more closely, to investigate how inspectors identify them.

Chapter 5

Comprehension and Artefact Interrelationships

"There's Grand Duchess Sophia of Turin — we'll never get her to marry him."
"Why not?"
"Because she's met him."

— Blackadder the Third

Delocalisation is a challenge facing software comprehension, and thus inspection. If inspectors do not attempt to understand relationships between different artefacts and parts thereof, then defects pertaining to such relationships will not be detected. However, while some artefact interrelationships are straightforward and unambiguous — especially those between artefacts of the same type — others are not so. Even if the internal semantics of two different artefacts are simple and well-understood, their interrelationships may be complex.

This chapter partially addresses the second research question from Chapter 1: *What are the challenges inherent in comprehending a system under inspection?*

To help provide an answer, this chapter presents a qualitative, empirical analysis of the difficulties encountered in identifying artefact interrelationships. In this case, the artefacts chosen are UML statecharts and Java source code. Participants were asked to identify sections of source code that implemented transitions in the statechart. The study did not involve an inspection *per se*, but rather an exercise in comprehension. Participants were not asked to find defects, and none were seeded. The study sought to observe the identification of artefact interrelationships — an essential component of multiple-artefact inspections.

Figure 5.1: The UML statechart for the Download class.

Participants' efforts were compared against a model solution, derived systematically by expert agreement. Despite the relative simplicity and small scale of the artefacts in question, none of the twenty-eight participants could identify all the code that could reasonably be said to implement each statechart transition.

5.1   Methodology

5.1.1   Participants

Participants in the study included 28 third- and fourth-year students. The student were enrolled in undergraduate programs in software engineering, computer science or information technology at Curtin University of Technology. All had previously completed a UML design subject. All participants were volunteers, and were offered no incentives other than experience gained.

5.1.2   Materials

Participants were given an instruction sheet along with a UML statechart and the Java source code for a single class. They had not previously seen any of these materials. The class under inspection was a simple URL downloader, designed to download a given file in a new thread and respond to requests to stop and restart the download.

Figure 5.1 shows the UML statechart presented to participants. The numbered anno-

tations were used in the study and will be used in this chapter to refer to specific state transitions. An unnumbered, initial transition exists to "AwaitingData" state, but this was not considered in the exercise. The Java source code implementing the statechart is listed in full in Appendix B, Section B.2 and consists of a single class containing seventy-one lines of code (excluding blank lines, comments and brace-only lines).

The instruction sheet (also shown in Appendix B, Section B.1) provided some documentation for the source code, including a description of the algorithm and the purpose of the three most important methods (`run()`, `startDownload()` and `stopDownload()`). This included a brief discussion of threading, as used (minimally) in the system.

5.1.3 Procedure

Three sessions were organised over four weeks. Each participant attended one session only. Besides the descriptions on the instruction sheet, no training was provided for the task.

The instruction sheet (shown in Appendix B, Section B.1) asked participants to "determine which sections of the source code implement each state transition in the statechart", and also provided background information intended to assist in understanding the source code. Some participants had queries about the exact meaning of the word "implement". This issue is discussed in Section 5.1.5. No time limits were imposed on participants.

After matching sections of the code to the state transitions, participants were asked to complete a questionnaire gauging their perceived understanding of the task. All but two also agreed to a recorded interview, which attempted to establish what techniques participants had used.

5.1.4 Coding Scheme

Participants usually chose to mark a source code printout to indicate which parts of the code they thought implemented each state transition. In a few cases participants stated this mapping elsewhere. These responses were classified by the area of code they appeared in or referred to. In total, fourteen classifications were used, as shown in figures 5.2 to 5.9. Only code constituting method definitions was considered. Participants were not asked to consider the initial unnumbered transition, so this has been disregarded.

```
/**
 * Creates a new Download object. The download is automatically started in
 * a new thread.
 */
public Download( URL url, String file ) throws IOException
{
    this.url = url;
    this.file = file;
    startDownload( );
}
```

Figure 5.2: Fragment `A:constructor` — the constructor for the Download class. `A':const-return` denotes the point in the method just prior to the closing brace.

Of the fourteen classifications, `A` to `K` represent non-overlapping fragments of code. The remaining three classifications were used to more precisely account for certain special cases. `A'` and `H'` represent the end points of `A` and `H`, where participants indicated that a state transition would occur immediately before a closing brace. `H1` refers to the first line of `H`, since a number of participants specifically indicated this line rather than the whole of `H`.

Often participants did not indicate the exact extent of code fragments to which they were referring. Some occasionally indicated only part of the code to which a classification was attached. In both these cases (except in the case of `H1`, as explained above), the closest classification was used. A few participants also indicated a fragment encompassing multiple classifications, in which case all relevant classifications were used.

5.1.5   Model Solution

To decide whether a given code fragment actually does implement a state transition, the following principle is used. A code fragment implements part of a transition if, when in the originating state:

1. the transition conceptually occurs at the same time as the fragment's execution;
2. the transition *never* occurs if the fragment is not executed; and
3. either —
    (a) the transition *always* occurs when the fragment is executed, or
    (b) the fragment itself decides whether the transition will occur.

Based on this, a given fragment of source code can play any of three roles in the implementation of a state transition:

```
/** Starts or restarts the download. */
public void startDownload( ) throws IOException
{
    if( stopped )
    {
        URLConnection connection = url.openConnection( );
        connection.connect( );
        inputStream = connection.getInputStream( );
        outputStream = new FileOutputStream( file );

        size = connection.getContentLength( );
        startTime = System.currentTimeMillis( );
        stopped = false;

        // Create a new thread for the download to run in.
        // This will call the run() method.
        new Thread( this ).start( );
    }
}
```

Figure 5.3: Fragment B: startDownload.

```
/** Stops the download, assuming it has been started. */
public void stopDownload( )
{
    stopped = true;
}
```

Figure 5.4: Fragment C: stopDownload.

```java
/**
 * Downloads from the URL supplied to the constructor. The method shouldn't
 * be called directly. It is started indirectly by the startDownload()
 * method.
 */
public void run( )
{
                                ─ D:run-init ─
    byte[] buffer = new byte[ READ_SIZE ];
    boolean timeout = false;

    try
    {
        long waitStartTime = System.currentTimeMillis( );


                                ─ E:loop ─
        while(( downloadedSize < size ) &&
                !timeout &&
                !stopped )
        {

                                ─ F:download ─
            int bytesAvailable = inputStream.available( );

            if( bytesAvailable >= READ_SIZE )
            {
                // We've retrieved enough data to fill the buffer. Write
                // it to disk and reset the timeout counter.
                inputStream.read( buffer );
                outputStream.write( buffer );
                downloadedSize += READ_SIZE;
                waitStartTime = System.currentTimeMillis( );
            }
            else if( bytesAvailable > 0 )
            {
                // Some data was retrieved. Write it to disk and reset
                // the timeout counter.
                inputStream.read(buffer, 0, bytesAvailable);
                outputStream.write(buffer, 0, bytesAvailable);
                downloadedSize += bytesAvailable;
                waitStartTime = System.currentTimeMillis( );
            }
```

Figure 5.5: The first half of the run() method, showing fragments D:run-init, E:loop and F:download.

```
            else
            {
                // No data retrieved. Sleep for a small interval to avoid
                // wasting CPU time. Check for a timeout.
```
┌─ G:timeout ───┐
```
                Thread.sleep( CHECK_INTERVAL );
                if( System.currentTimeMillis( ) >
                    waitStartTime + TIMEOUT )
                {
                    timeout = true;
                }
```
└───┘
```
            }
            calcSpeed( );
        }
```
┌─ H:finish ──┐
```
        stopDownload( );
        inputStream.close( );
        outputStream.close( );
```
└───┘
```
    }
    catch( Exception e )
    {
    }
}
```

Figure 5.6: The second half of the `run()` method, showing fragments `G:timeout` and `H:finish`. `H':run-return` denotes the end point of `H:finish`, just before the closing brace of either the while loop or the whole method. `H1:finish-stop` denotes the first line of `H:finish` — the `stopDownload()` method call.

```
/**
 * Called by the run() method to calculate the mean transfer rate (bytes
 * per second) of the download so far.
 */
private void calcSpeed( )
{
    long time = System.currentTimeMillis() - startTime;
    if( time > 0 )
    {
        speed = (( double )downloadedSize / ( double )time ) * 1000.0;
    }
}


/**
 * Returns the last calculated transfer rate (bytes per second) of the
 * download, or 0 if no calculation has yet been made.
 */
public double getSpeed( )
{
    return speed;
}


/** Returns the download progress as a percentage. */
public double getPercentDone( )
{
    return (( double )downloadedSize / ( double )size ) * 100.0;
}
```

Figure 5.7: Fragment I:accessors — consisting of the calcSpeed(), getSpeed() and getPercentDone() methods.

```
/** Returns true iff the download has finished. */
public boolean hasFinished( )
{
    return downloadedSize >= size;
}
```

Figure 5.8: Fragment J:hasFinished.

```
/** Returns true iff the download is in progress. */
public boolean isDownloading( )
{
    return downloadedSize < size && !stopped;
}
```

Figure 5.9: Fragment K:isDownloading.

Table 5.1: The model solution mapping between state transitions and code fragments, showing the nature of a fragment's involvement in a transition. Fragments that do not implement any transition are omitted. Transitions 6 and 7 are omitted because no fragment implements them.

Fragment	Transition				
	`1:start`	`2:stop`	`3:timeout`	`4:finish`	`5:rcv`
`B:startDownload`	decision/ mutation	—	—	—	—
`C:stopDownload`	—	mutation	mutation	—	—
`E:loop`	—	binding	binding	decision	—
`F:download`	—	—	—	—	decision/ mutation
`G:timeout`	—	—	decision	—	—
`H:finish`	—	mutation	mutation	mutation	—

- *decision code* determines whether a transition will take place immediately (i.e. at the time of execution), having a definite "yes" or "no" outcome;
- *mutation code* establishes the conditions required by the new state; and
- *binding code* directs the flow of control between other code fragments involved in the transition.

None of these components is itself necessarily atomic. Each may be further divided into non-contiguous code fragments depending upon how the source code is structured. In some places, one code fragment may also (at least partially) implement multiple transitions.

Although frameworks for statechart-source code relationships have been proposed, it is not clear that any are widely used or recognised. Neither the Unified Modelling Language itself nor the Java language specification provide any guidance on how this should be done. In sufficiently different situations, such as involving actions, history states or other statechart mechanisms, it would be necessary to introduce different types of relationships. Nevertheless, for situations similar to that in the study, these three roles provide a suitable basis for understanding these artefact interrelationships.

The model solution itself was arrived at by agreement between two experts having independently applied the above taxonomy to the set of code fragments. Table 5.1 lists the code fragments comprising the model solution, and the roles they play in implementing each of the state transitions. Transitions `1:start` and `5:rcv` have a one-to-one mapping with the source code. Transitions `2:stop`, `3:timeout` and `4:finish` involve multiple, overlapping code fragments. Transitions `6:dst` and `7:dst` are not explicitly located in the code.

5.2   Results

5.2.1   Techniques Used

The interview process attempted to gain insight into the techniques used by participants. Some recurring patterns were found.

According to the interviews, many participants almost immediately identified mappings for transitions `1:start` and `2:stop`. This probably occurred because the names given on the statechart matched names occurring in the source code (`startDownload()` and `stopDownload()`). Several carried that idea too far, identifying the fragment `J:hasFinished` for transition `4:finish` and `K:isDownloading` for `5:rcv`.

Another common technique involved finding assignment statements or conditions that participants could see were connected with the statechart logic. Transition `2:stop`, in which the download is halted by a call from another class, was often identified by the statement `stopped = true`. Transition `3:timeout`, in which the download is halted after sixty seconds of not receiving any data, was similarly identified by the statement `timeout = true`. Some participants also indicated that statements incrementing the `downloadSize` variable implemented transition `4:finish`, in which the download finishes.

At least two participants noted that comments were useful for identifying state transitions. Three described how they identified transition `3:timeout` by its use of the constant `TIMEOUT`, whose value matched that on the statechart. One mentioned searching for a loop structure representing transition `5:rcv`, a self-transition in which the "AwaitingData" state is reset through receipt of data.

5.2.2   Solutions

Participants' answers to the exercise, coded as described in Section 5.1.4, are presented in Table 5.2. A transition-centric view of the results is also given in Figure 5.10.

Most participants gave their answers by writing the transition numbers on the source code printout. However, a few did not do this. In particular, participant 15 marked the source code by destination state rather than state transition. Two of the states in the statechart have multiple incoming transitions, so this created ambiguity. Also, participant 20 did not provide any answers at all, evidently due to a misunderstanding
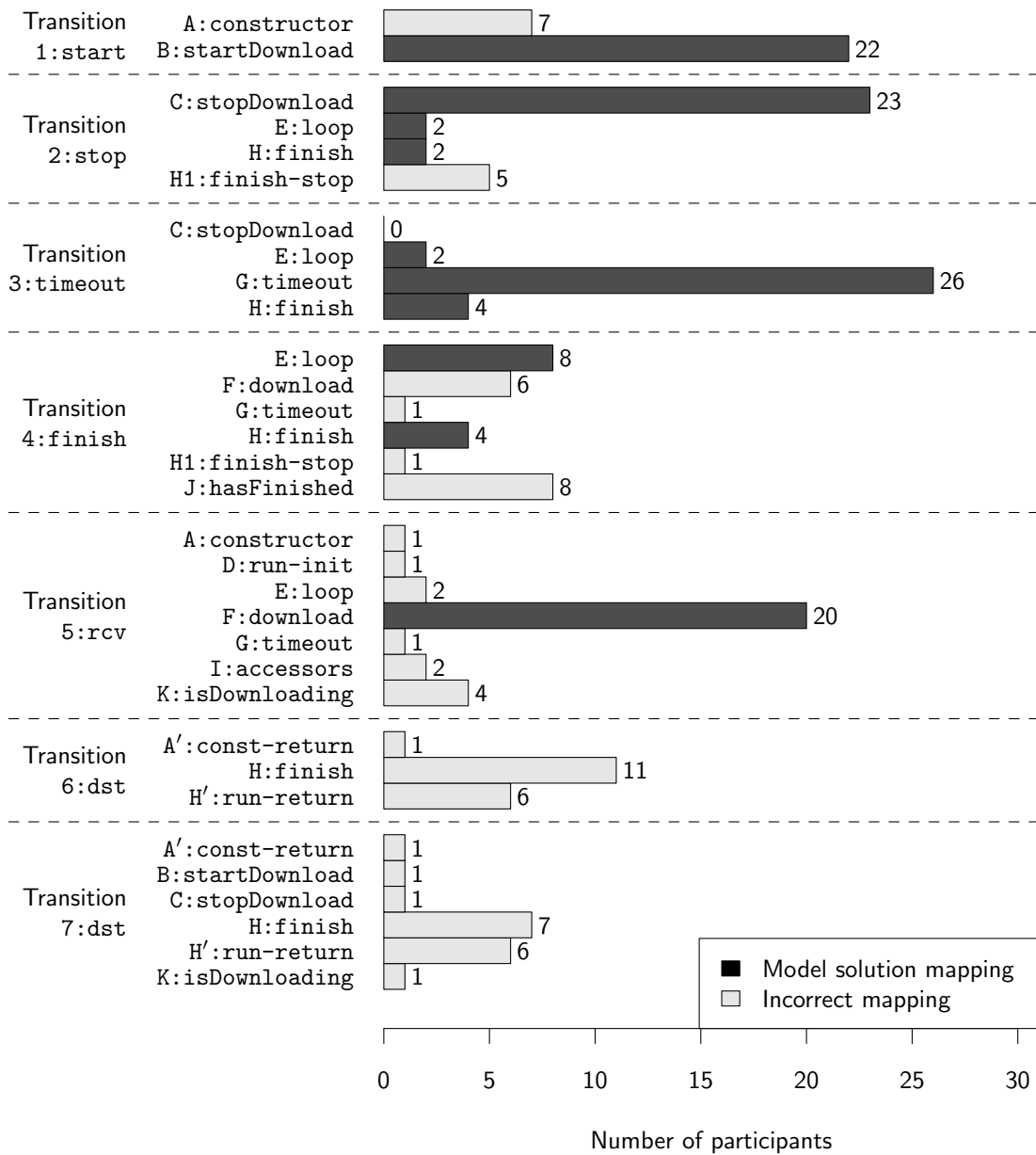
Figure 5.10: The numbers of participants who identified particular fragments for each transition. Dark bars represent transition-fragment mappings that appear in the model solution. Light bars represent mappings that do not appear in the model solution.

Table 5.2: A matrix of possible mappings between fragments (rows) and transitions (columns). Each cell represents a single possible mapping, and contains the proportion of participants who identified that mapping. Colours are used to indicate the extent of agreement/disagreement with the model solution.

Fragment	Transition						
	1:start	2:stop	3:timeout	4:finish	5:rcv	6:dst	7:dst
A:constructor	26	0	0	0	3	0	0
A':const-return	0	0	0	0	0	3	3
B:startDownload	**84**	0	0	0	0	0	3
C:stopDownload	0	**88**	**0**	0	0	0	3
D:run-init	0	0	0	0	3	0	0
E:loop	0	**7**	**7**	**30**	7	0	0
F:download	0	0	0	23	**76**	0	0
G:timeout	0	0	**100**	3	3	0	0
H:finish	0	**7**	**15**	**15**	0	42	26
H':run-return	0	0	0	0	0	23	23
H1:finish-stop	0	19	0	3	0	0	0
I:accessors	0	0	0	0	7	0	0
J:hasFinished	0	0	0	30	0	0	0
K:isDownloading	0	0	0	0	15	0	3

100	Model solution mapping, commonly found.
7	Model solution mapping, not commonly found.
42	Incorrect mapping.

(having indicated that no major difficulties were encountered). These two participants have therefore been excluded from the figures discussed below and the analysis in the following section.

Near-consensus was reached among the twenty-six remaining participants that:

- fragment `B:startDownload` implements transition `1:start`;

- fragment `C:stopDownload` implements transition `2:stop`;

- fragment `F:download` implements transition `5:rcv` — the code for reading and processing downloaded data implements the self-transition to/from the "AwaitingData" state; and

- fragment `G:timeout` implements transition `3:timeout` — the code for checking whether a timeout has occurred implements the timed transition from the "AwaitingData" state to the "Stopped" state.

About half the participants also believed that transitions `6:dst` and `7:dst` to the end state would take place at either H or H′, at or immediately after the clean-up code following the while loop.

There was no strong agreement on the location of transition `4:finish`, when the download finishes. Suggested fragments included the while loop test condition (fragment `E:loop`), the data reading and processing code (fragment `F:download`), the clean-up code (fragment `H:finish`) and the `hasFinished()` method (fragment `J:hasFinished`). Five participants omitted transition `4:finish` altogether, which suggests it was inherently more difficult to identify than the others.

5.3 Analysis

5.3.1 Transition-Fragment Mapping

The techniques used by participants to map state transitions to source code led to solutions that were often readily justifiable, but sometimes questionable.

Suggestive method names allowed participants to quickly identify the `startDownload()` and `stopDownload()` methods and state transitions. These methods must be called as part of those transitions. However, the `hasFinished()` and `isDownloading()` methods (fragments `J:hasFinished` and `K:isDownloading`) are accessors intended solely for external use. They are not used by any other code within the class, even though several participants stated that one was, and two claimed that one or the other should have been used. Transitions `4:finish` and `5:rcv` could therefore take place without them.

Several participants identified call sites for the `startDownload()` and `stopDownload()` methods (fragments `A:constructor` and `H1:finish-stop`) for transition `1:start` and `2:stop`. Although these methods must be called in order for the transitions to take place, the reverse is not necessarily true. The object would not be in the correct state in A or H1 for either transition to be triggered.

Mixed success occurred where transitions were identified by variable assignment. The variable `stopped` indicates whether the system is currently in the "Stopped" state, and so code that assigns it `true` (while in another state) must change the state. The variable `timeout` only temporarily stores state information — the "AwaitingData" state is reflected in the execution of the while loop. Nevertheless, the statement `timeout = true` has the effect of terminating the loop, and so therefore must also change the state. By contrast, the modification of the `downloadSize` variable in fragment `F:download` only sometimes results in the loop being terminated.

For each of the three outward transitions from the "AwaitingData" state (`2:stop`, `3:timeout` and `4:finish`), relatively few participants identified the while loop test condition (fragment `E:loop`) or the clean-up code (fragment `H:finish`). The former decides whether the loop will exit, so it plays a role in all three state transitions. Fragment `H:finish` helps establish the conditions for the new states, so it too is involved in the transition.

Though many participants associated the clean-up code (fragments `H:finish` and `H':run-return`) with the two end state transitions (`6:dst` and `7:dst`), this idea faces two problems. First, at that point the object is only just exiting the "AwaitingData" state, from which there is no transition to the end state. Second, if the end state is regarded as the object's destruction, then in Java no code is required to implement those transitions; at least, not within the class itself. (In a garbage-collected environment, an object's destruction is caused by the removal of external references to it, and it would be unusual for the object in question to trigger this itself. The only conventional contribution a Java object makes to its own destruction is through overriding the `finalize()` method, which has not been done in this case.)

It is not clear if any obvious beacons were involved in the mapping of transition `5:rcv` to fragment `F:download`. Certainly this fragment is responsible for reading and processing downloaded data, and the transition's label is "data received".

5.3.2   The One-To-One Misconception

For the most part, participants identified a one-to-one mapping between the state chart and the source code. They rarely found more than one code fragment for each transition, or more than one transition for a given code fragment. Many-to-many mappings do arise, however, where several transitions leading to the same or similar states share a common set of statements. This would result in a code fragment shared between the transitions, with control constructs separating the remainder of the transitions' code.

For each of transitions `2:stop` and `3:timeout`, one of the correct code fragments was found by most participants (and was easily identifiable using the techniques in Section 5.2.1), while the others went largely unnoticed. No participants identified all the correct fragments for either transition. For transition `4:finish`, only eleven participants found either of the correct code fragments, `C:stopDownload` and `F:download`. Only participant 24 found both.

In total, eighteen participants identified code fragments that they thought implemented

two or more transitions. However, many of these were for transitions `6:dst` and `7:dst`, which share an obvious commonality (both destroy the object and both are triggered in similar circumstances — when no downloading is taking place). Excluding these transitions, the number of participants falls to seven. Only three participants identified one or more of the fragments (`C:stopDownload`, `E:loop` and `H:finish`) that really did implement multiple transitions. Of these, only participant 24 successfully matched all the relevant transitions to any of the fragments.

Participants typically appeared to adhere to the preconception that each state transition is implemented by a single, unique fragment of code. While the *ad hoc* techniques they used are clearly capable of identifying some of the simpler and more obvious relationships between statecharts and source code, they suffer both in that they do not account for many-to-many relationships, and often identify the wrong code.

5.4   Discussion

By providing for different roles for different fragments, the logic underlying the model solution (discussed in Section 5.1.5) makes it clear that multiple fragments might implement a single transition. This breaks the illusion of a one-to-one mapping and thus makes the reverse proposition, that fragments can implement multiple transitions, similarly apparent. Moreover, three specific aspects of the transition will be under scrutiny, rather than just a vague correspondence to source code. Thus, the principles from which the model solution is derived could behave as a checklist for comprehension, similar to the use of active guidance in Scenario-Based Reading (SBR). Though SBR studies overall appear inconclusive (as shown in Chapter 2, Section 2.2.2), this study demonstrates a situation in which guidance is needed.

This particular statechart-source code mapping strategy is not necessarily authoritative. Indeed, the widespread adoption of an authoritative means of mapping one to the other is impractical. However, this method serves as a basis for assessing relationships between the two types of artefacts.

The disparities between participants' answers and the model solution suggests that the identification of artefact interrelationships is a non-trivial exercise and requires a more systematic approach. For software inspectors examining multiple artefacts there is no universal notion of correctness regarding artefact interrelationships. There must nonetheless be a common understanding of these interrelationships, at least within the context of individual organisations and software projects. Were the source code to represent a defective implementation of the statechart, the detection of defects would

necessary entail the identification of mappings from one artefact to the other.

In a real inspection, however, artefact interrelationships need not be explicitly identified one-by-one by inspectors in the manner performed in the study; doing so would likely introduce unnecessary overhead. Nonetheless, inspectors must have the ability to quickly traverse artefact interrelationships where needed. Where the complexity or unfamiliarity of these interrelationships poses a problem, the principles governing them should be used to design reading techniques and forms of cognitive support to assist inspectors. These should allow and encourage inspectors to draw information from multiple artefacts without devoting substantial effort to understanding the interrelationships themselves.

5.5 Summary

The statechart study described in this chapter serves to demonstrate the difficulty of identifying artefact interrelationships, even where the artefacts in question are relatively simple. In this study, participants were observed to adopt an overly-simplified view of artefact interrelationships, as evidenced by common behaviours, including:

- the assumption of a one-to-one mapping between statechart transitions and source code fragments; and
- the reliance on naming conventions to identify mappings.

These might be considered novice mistakes. However, as discussed in Chapter 2, Section 2.3.4, even experts can behave as novices given a sufficiently complex or unfamiliar set of artefacts. Variability among inspectors will mean that, in a given situation, some will generally overcome obstacles posed by delocalisation, while others may not. Providing guidance to inspectors may be a burden when it is unnecessary, but if it can be given only where needed then a more favourable inspection outcome is likely.

The previous studies highlighted in Chapter 2, Section 2.2 support a debate within the research community — first over whether reading techniques are useful, and second over which ones might be the most effective. This study supports the necessity of reading techniques and reading technique research. Being a qualitative study, the results might not be reproduced in quite the same way in different circumstances. Nevertheless, this study illustrates some basic comprehension issues that can arise, and which can be overcome with some form of guidance. Identifying when such guidance should be given, and what form it should take, are examined in subsequent chapters.

Chapter 6

Comprehension and Scenarios

"Doing what we've done eighteen times before is exactly the last thing they'll expect us to do this time."

— Blackadder Goes Forth

Software comprehension has not been widely explored in the presence of a reading technique. While the preceding chapter establishes the need for guidance in some cases, this chapter explores comprehension challenges that arise despite it. This further addresses the second, software comprehension-related research question posed in Chapter 1: *What are the challenges inherent in comprehending a system under inspection?*

The research presented in this chapter has previously been published (by the author of this thesis) as an individual component of the larger work (Cooper et al., 2007).

Usage-Based Reading (UBR), proposed and examined by Thelin et al. (2003), is used here as a representative of active-guidance techniques. The scenario study described in this chapter re-examines qualitatively the central task entailed by UBR, and by the similar use-case technique of Dunsmore et al. (2003): that of tracing the events of a use case scenario through another artefact. The survey discussed in Chapter 4 found that such traversal of use case scenarios is a relatively common practice in industry. UBR is also chosen for its relative prominence in academic literature — the technique is the most recent to have been examined by multiple experimental replications.

UBR has consistently outperformed Checklist-Based Reading (CBR) with respect to critical defect detection. However, with respect to overall defect detection, it exhibits similarly inconsistent performance to active guidance techniques in general. As with such techniques, UBR relies on inspectors following a relatively rigid pattern of inspec-

tion. Such a requirement seems at odds with theories of software comprehension, which suggest that comprehension strategies are employed on an opportunistic basis (Linger et al., 1979, Letovsky, 1986, von Mayrhauser and Vans, 1995). Moreover, although UBR requires inspectors to match inspection artefacts against use case scenarios, little is known of how this matching is or should be done. Theoretical concerns regarding this have been expressed, and solutions including tool support and the insertion of visual cues into artefacts have been proposed (Kim et al., 2000, Walkinshaw et al., 2005). Several variations of UBR have been proposed and tested, but the effects of this form of active guidance on the comprehension process have not been qualitatively examined in detail.

The scenario and statechart studies share a common purpose in their exploration of artefact interrelationships. However, the scenario study seeks to investigate the process of comprehension, rather than just its outcome. Ten participants were asked to follow a use case scenario, represented by a UML sequence diagram, through Java source code. These are not the types of artefacts employed in the original and replicated studies of UBR, nor does this task represent the entire task undertaken in UBR. Nevertheless, this represents an application of a key UBR principle, and serves to isolate part of the technique for observational purposes. Given one use case scenario and a small system of 193 executable lines of code, participants were asked to identify those lines executed in the scenario, and the order thereof. Participants were also asked to think aloud during the exercise, and data collected automatically through an online interface allowed further reconstruction of participants' actions.

Several issues affecting participants' ability to locate the correct lines were identified from data collected. Data generated from the think aloud process indicate digressions from the scenario and large variations in participants' comprehension approaches.

6.1 Methodology

The scenario study examined participants' behaviour as they completed an inspection-related task. Two data collection approaches were employed: (a) the automatic recording of events from an online interface, and (b) thinking aloud. These methods produced distinct but related sets of results.

Data collected through the online interface facilitated analysis of the actual steps participants took towards completing the task, as discussed in Section 6.1.4. The use of protocol analysis described in Section 6.1.5 relied on this data, but also provided insight into how comprehension of the system proceeded.

6.1.1 Participants

Ten participants were involved: five industry professionals, three university-employed graduates and two undergraduate students. The latter two groups had enrolled at or graduated from Curtin University of Technology. Industry experience ranged from zero to five years of professional practice. No incentives were offered to participants other than experience gained.

6.1.2 Materials

The system under inspection was a simple Java-based audio player controlled from a command-line interface. One use case scenario was presented in the form of a UML sequence diagram, shown in Figure 6.1.

Excluding comments, blank lines and brace only lines, the source code consisted of 330 lines of code in seven files. However, to avoid confusion regarding what constituted a line of code, only executable statements and field initialisers were numbered. In this scheme, there were 193 numbered lines of code. The source code and other artefacts presented to participants are shown in Appendix C. Participants had not previously seen any of the artefacts presented.

6.1.3 Procedure

Each participant completed an inspection-related activity individually in the presence of a researcher. Participants were asked to list, in order of execution, each line of code that is or might be executed in the scenario. Such an instruction imposes a rigid ordering on the exercise, because one line's participation in the scenario cannot be determined until all previous lines in the scenario have been found.

The task was undertaken through an online interface. Participants added entries, each consisting of a class name and line number or range, to an input box for lines of code they determined should be executed. The times at which the input box was modified were recorded automatically in a database. No time limits were imposed on participants.

Table 6.1 shows an example of the data collected in this way (the importance of which is discussed further in Section 6.2). Participants were asked to enter lines in a given
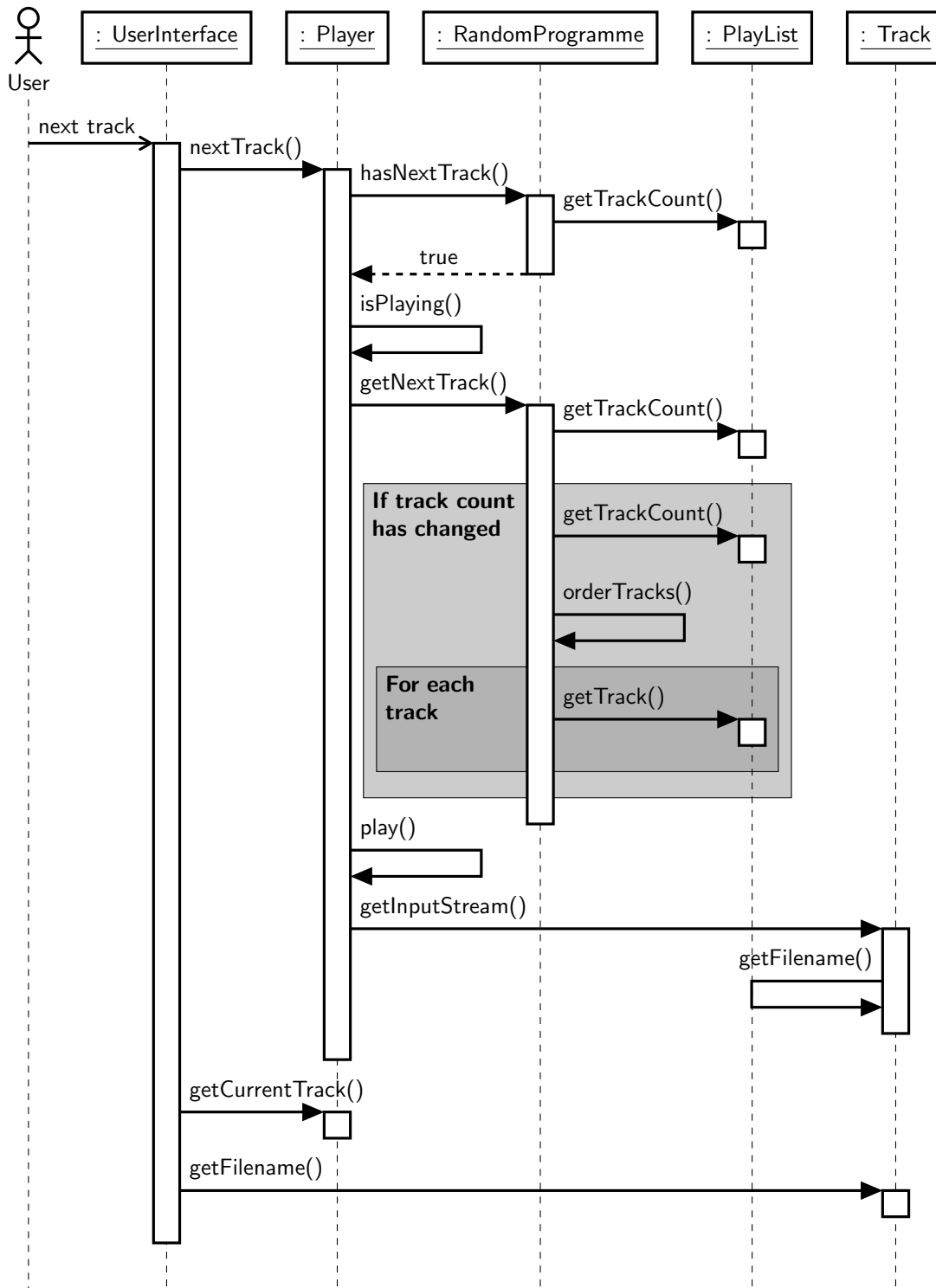
Figure 6.1: The UML sequence diagram shown to participants, representing the scenario in which the *next track* option is chosen while tracks are being played in a random order. The shading emphasises nesting levels.

Table 6.1: An excerpt of the input updates recorded for participant 8. Each row represents one update. Times of the updates are relative to the start of the exercise. For brevity, unchanging parts are marked with ellipses ("...").

Time	Delta time	Full-text input update	Commentary

18:40	—	`UserInterface.java 5-19`
		`UserInterface.java 18-19`
		`UserInterface.java 22,27,31-32`
		`Player.java 27`
		`Programme.java 3`
		`Playlist.java 15`
		`Player.java 28,35-38,28-29,30?,31-32,`	

20:59	2:19	`...`
		`Player.java 28,35-38,28-29,30?,31-32`
		`Programme.java`
			Addition of `Programme.java`.

21:08	0:09	`...`
		`Player.java 28,35-38,28-29,30?,31-32`
		`Programme.java 5,6`
			Addition of line numbers for `Programme.java`.

25:20	4:12	`...`
		`Player.java 28,35-38,28-29,30?,31-32`
		`RandomProgramme.java 5,6`
			Replacement of Programme with RandomProgramme.

25:25	0:05	`...`
		`Player.java 28,35-38,28-29,30?,31-32`
		`RandomProgramme.java`
			Deletion of original line numbers.

25:35	0:10	`...`
		`Player.java 28,35-38,28-29,30?,31-32`
		`RandomProgramme.java 8`
			Addition of new line number.

format, though the format of the actual input data varied. A parser was developed to convert each input update into a list of class/line-number pairs. The parser used a manually-constructed look-up table to (a) resolve misspellings and general variations in the input format, and (b) help it recognise and ignore partially-complete entries. (Participants were also asked to add question marks and asterisks to indicate where execution of a line was conditional or repeated, respectively, though this information was not eventually needed for the analysis described in the next section.)

The use of an online interface conceivably affected participants' performance, by drawing cognitive resources away from the task at hand. This may have lessened the effective expertise participants could bring to bear, but it does not diminish the qualitative results of the investigation.

During the exercise, participants were asked to think aloud and were prompted to "keep talking" if silent for more than 30 seconds. They were each given two short training tasks to familiarise them with this process.

Table 6.2: Example reconstruction of a sequence of four input updates. The final reconstructed input list resembles the last full-text update, but with decision times and line deletions shown.

(a) Full-text updates

Time (secs)	Delta time (secs)	Full input text
3	3	line 3
17	14	line 3 line 5
21	4	line 3 line 4 line 5 line 3
26	5	line 3 line 10 line 5 line 3

(b) Reconstructed inputs

Decision time (secs)	Updated input text
3	line 3
$4 + 5$	line 4 (deleted)
5	line 10
14	line 5
4	line 3

Audio recordings were made independently of the online interface. These were synchronised with the data captured through the online interface by comparing mouse clicks in the former to mouse events recorded in the latter. This synchronisation allowed the protocol analysis process to use contextual information from the input data.

6.1.4   Input Data Analysis

Once translated into a common format, participants' input updates were then used to reconstruct a single solution incorporating all line additions and deletions, in the intended order of execution (using an edit distance algorithm). The reconstructed list also included the *decision time* for each line — the total time spent immediately prior to adding and/or deleting it. The decision time is assumed to be largely spent determining whether the line takes part in the scenario, and thus broadly indicative of the comprehension effort required.

Table 6.2 gives an example of this input reconstruction process. Here, `line 3` was added in the first update, `line 5` in the second, `line 4` and `line 3` in the third and `line 10` in the fourth, in which `line 4` was also deleted. The reconstructed list is built up in the course of examining successive full-text input updates. The arrows in Table 6.2 represent insertions into the reconstructed list, performed as each new line is added. At each stage, the edit distance algorithm compares the incomplete reconstructed list

Table 6.3: Example normalisation of input updates. Here, the model solution consists of "line 3", "line 10" and "line 5", in that order. The participant's reconstructed inputs (from Figure 6.2) are slightly different. The model solution and reconstructed inputs are combined, line by line, to generate the normalised solution.

(a) Model solution

Solution index	Solution text
0	line 3
1	line 10
2	line 5

(b) Reconstructed inputs

Decision time (secs)	Input text
3	line 3
9	line 4 (deleted)
5	line 10
14	line 5
4	line 3

(c) Normalised solution

Solution index	Input solution	Flagged (point of interest)
0	line 3	—
1	line 4 (deleted) line 10	changed
2	line 5	high decision time
3	line 3	erroneous

(the participant's solution so far) with the next full-text input update. From this, the algorithm determines which lines are added or deleted and where they should appear in the list. From the example in Table 6.2, `line 4` is the third line to be added, but appears second in the reconstructed list because it was inserted before `line 5`. Where the same line was listed at multiple points in the solution, the reconstructed list preserves these separate instances in their intended order of execution.

The reconstructed lists were then normalised by comparing them to a model solution, line by line. The model solution itself was constructed by having two experts independently create lists of the lines executed, and then agree on a single solution. Table 6.3 demonstrates the normalisation process. The final, normalised solution reflects the reconstructed input list, but with the addition of *solution indices*, which provide a mapping to the model solution.

In each normalised solution, *points of interest* were designated where lines had one or more of the following characteristics:

- The line deviated from the model solution. Several small deviations were discounted as trivial, where it was plausible that a participant did nonetheless un-

derstand the path of execution. These included some instances of line duplication (as shown at the top of the first row in Table 6.1), two adjacent lines listed in reverse order, or a missing method call before the correctly-listed contents of that method.

- The line was associated with a high decision time, relative to other times for the same participant. High decision times were considered to be those more than 0.5 standard deviations above the mean decision time for a given participant.

- The line had been deleted (and possibly later re-listed).

Table 6.3 gives an example for each of the above. At solution index 1, `line 4` is flagged as having changed, due to being deleted. At solution index 2, `line 5` is flagged as having a high decision time — 14, where the mean is 7 and the standard deviation 2.3 ($14 > 7 + (0.5 \times 2.3)$). At solution index 3, `line 3` is flagged as erroneous because it does not appear in the model solution at that point.

6.1.5   Protocol Analysis

Participants' verbal protocols were used to determine:

- the variability in the comprehension process, as exhibited by different participants; and

- the extent to which participants were able to concentrate on each step of the use case scenario in turn.

Variability was determined by observing the overlap between participants' verbalised thoughts at any given solution index. The extent of digression is determined by observing the proximity of participants' verbalised thoughts to the line additions or deletions at each input update.

Both these indicators rely on the verbal data being coded. Specifically, codes consisted of references to specific points or constructs in the artefacts: methods, method calls, constructors, fields or variables in the source code; and messages, objects and either of the two shaded control-flow constructs in the sequence diagram. Examples of references include:

- `Player.nextTrack()`, indicating the `nextTrack()` method in the `Player` class;

- `Player.nextTrack():programme.hasNextTrack()`, indicating a call site of the method `hasNextTrack()` accessed through the `programme` variable within the `nextTrack()` method; and

- `SD:nextTrack()`, indicating the `nextTrack()` message as shown in the sequence diagram.

The relative simplicity of the coding scheme allowed coding to be done directly from the audio recordings, without the need for transcription or segmentation. However, an excerpt of one transcript is presented in Figure 6.2 for illustration purposes.

To map references to solution indices, the coding process was conducted on a solution index by solution index basis. The period(s) of time spent by each participant at each solution index, as captured by the online interface, were used to determine where to listen in the audio recordings.

Participants often omitted one or more of the first components in each reference, and contextual information had to be relied upon to resolve any ambiguity. Such contextual information included the artefact currently or previously focused on and the method currently being inspected. The former was recorded by the online interface, and the latter inferred from the current or previous solution index.

Although no duplicate references existed at each index, it was possible for some references to be counted for more than one solution index in cases where a single input update added or deleted more than one line. This also occurred where a reference was made at the same time as an input update, and so was counted for both that update and the next one. This duplication is not problematic, however, because no cumulative analysis is done on the references.

A second coding scheme was then superimposed on the first to give an indication of the relevance of each reference to its solution index. A reference to a field, variable, method or message appearing at the current line was labelled `cl`. Failing this, if it appeared on the previous or next line of either the model solution or the source code, it would be labelled `pl`. The label `cm` was otherwise given for references to or within the same method, or `pm` for references to or within the previous or next method in the model solution, on a line-by-line basis. Similarly, `cc` was given for references to or within the current class, or `pc` for those to or within the previous or next class in the model solution. If no other categories were applicable, a reference was considered unrelated and labelled `u`.
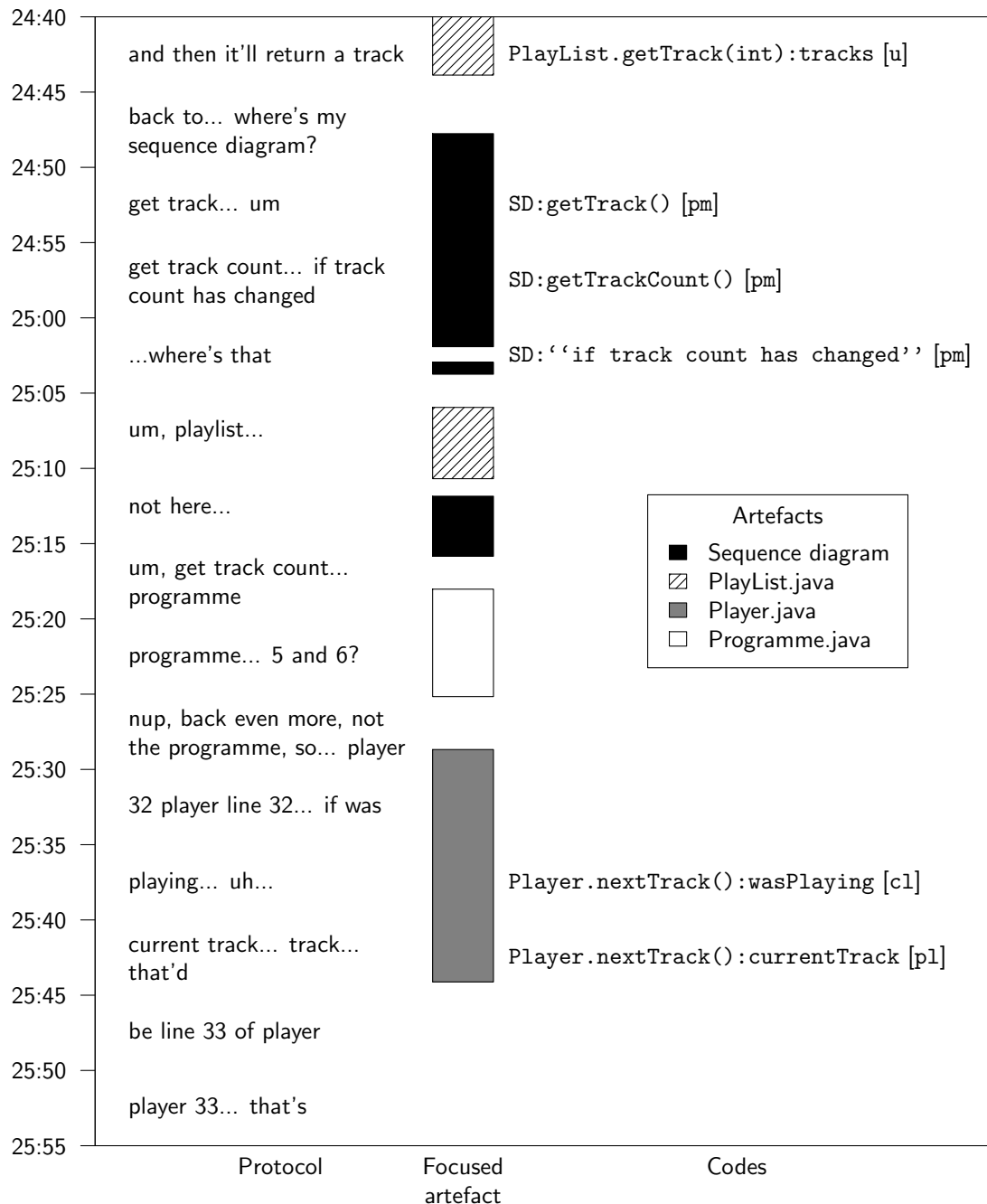
Figure 6.2: An excerpt of a participant's verbal protocol at one solution index, showing the periods of time spent focused on different artefacts and the codes assigned. The gaps between the artefacts indicate where the participant had focused instead on the reporting form.

Table 6.4: Overall characteristics of each participant's solution.

Participant	Total time	Mean decision time	Model solution coverage	Line mismatches
1	54 min	35 sec	42%	74%
2	35 min	28 sec	85%	37%
3	55 min	45 sec	33%	83%
4	35 min	44 sec	63%	25%
5	32 min	43 sec	62%	20%
6	43 min	32 sec	73%	27%
7	42 min	38 sec	67%	55%
8	46 min	37 sec	85%	54%
9	39 min	36 sec	63%	46%
10	28 min	37 sec	40%	64%
mean	41 min	39 sec	61%	48%

Table 6.5: Overall scale and effect of the comprehension issues identified.

Issue	Points of interest	Affected participants (out of 10)	Mean decision time
Scenario start misidentification	13	10	204 sec
Accidental omission	24	9	33 sec
Polymorphism	8	7	120 sec
Method misidentification	8	6	37 sec
Context switching	23	10	98 sec
Condition evaluation	28	9	76 sec

Each reference received one of the above *proximity codes* depending on its proximity to the current solution index line of code. These codes were assigned by comparing each reference to the current solution index in the model solution and selecting the first applicable code in order.

6.2 Results

6.2.1 Input Data

An overview of the characteristics of each participant's solution is given in Table 6.4. Considerable variation exists in both model solution coverage (33–85%), and the proportion of line mismatches (20–83%). Model solution coverage indicates the proportion of the lines of the model solution that also appear in each participant's solution. Line

mismatches indicate the proportion of the lines in each participant's solution absent from the model solution.

On average, 41 minutes were spent by each participant on the task. This is high, given that a real inspection is recommended to last for no more than two hours (Fagan, 1976) and would involve several use case scenarios. However, the requirement for participants to explicitly note each line of code would have added to the time spent.

By comparing points of interest derived from each participant's solution, six common issues were identified. (In some cases, as a result of an earlier omission, participants did not have a chance to confront parts of the source code in which certain issues commonly arose.) These issues are summarised in Table 6.5.

- *Scenario start misidentification.* No participant's solution began at exactly the point defined by the model solution. Four listed lines occurring prior to this point, and six omitted at least some subsequent to it. Anecdotal evidence from one participant suggests that there may have been confusion over the term "command-line based" as given in the instructions. The term referred to the fact that the system instituted its own command-line, not that it operated necessarily from an existing command-line. A consequence of the latter interpretation may have been that the initial construction sequence of the system was assumed to be part of the scenario, as indicated by five participants, whereas in fact the sequence diagram made no mention of it.

- *Accidental omission.* Method omissions, where participants listed method calls but not the corresponding method contents, were a common occurrence, accounting for 14 points of interest. In another instance in the system, the Java `Arrays.sort()` method is called with a customised `Comparator` object, a result of which is that in the scenario the object's `compare()` method is called from a standard Java class. No participants identified this method. Three more points of interest were attributable to the omission of the last one or two statements in a method.

- *Polymorphism.* At the method call `programme.getNextTrack()`, seven participants initially indicated that the `getNextTrack()` method of the `Programme` class was executed. The sequence diagram, however, specifies that the call is actually made to an instance of the subclass `RandomProgramme`. Five participants eventually realised this (perhaps because many of the messages on the sequence diagram were passed only from the subclass's method).

  The challenge posed to participants by polymorphism is evident in the raw input data in Table 6.1. In the third input update (at 21:08), participant 8

has incorrectly listed lines 5 and 6 of `Programme.java`, which represent the `getNextTrack()` method. This begins to change in the fourth update (at 25:20), more than 4 minutes later. After a few more seconds (at 25:35), line 8 of `RandomProgramme.java` is correctly listed. Although the lines ultimately identified by the participant were correct, the high decision time involved and the fact that lines were deleted identified this instance as a point of interest.

- *Method misidentification.* In two cases of method overloading, six points of interest were the result of participants indicating the wrong method. One more point of interest arose from a participant listing the `getNextTrack()` method in place of `hasNextTrack()`. Another occurred where one accessor method was listed in place of another that, as an implementation detail, happened to return the same value.

- *Context switching.* High decision times were recorded in nineteen instances where method calls took place, and in four cases after methods returned. This can be attributed to the time and effort associated with locating a line not adjacent to the previous line, often in another class. Seven of the nineteen high decision times associated with method calls were for the *first* method call.

- *Condition evaluation.* Seven points of interest resulted from participants incorrectly evaluating `if` statement conditions (though four were likely the result of a previous error of the same kind). These were either false negatives or false positives. A further ten high decision times were recorded for the first line inside an `if` statement. Similar evaluation is required to determine whether an exception is thrown at a given point in the scenario. None were included in the model solution, but 11 points of interest resulted from participants listing them. These all reflect the difficulty of manually evaluating dynamic conditions in a static context.

  Determining whether a boolean condition is satisfied or if an exception occurs, in the context of a specific use case scenario, requires the consideration of various constraints imposed by the scenario and the source code. For example, if a message represented on the sequence diagram corresponds to a method call inside an `if` statement, the `if` condition must be true in that scenario. Likewise, if throwing an exception would prevent one or more messages from being passed, it cannot occur within the scenario.

In addition to the above issues, a small number of other points of interest remain unclassified. These include four instances where both branches of an `if` statement were listed, two where no logic appeared to connect one listed line to the previous line, and one where a high decision time was recorded for no apparent reason.
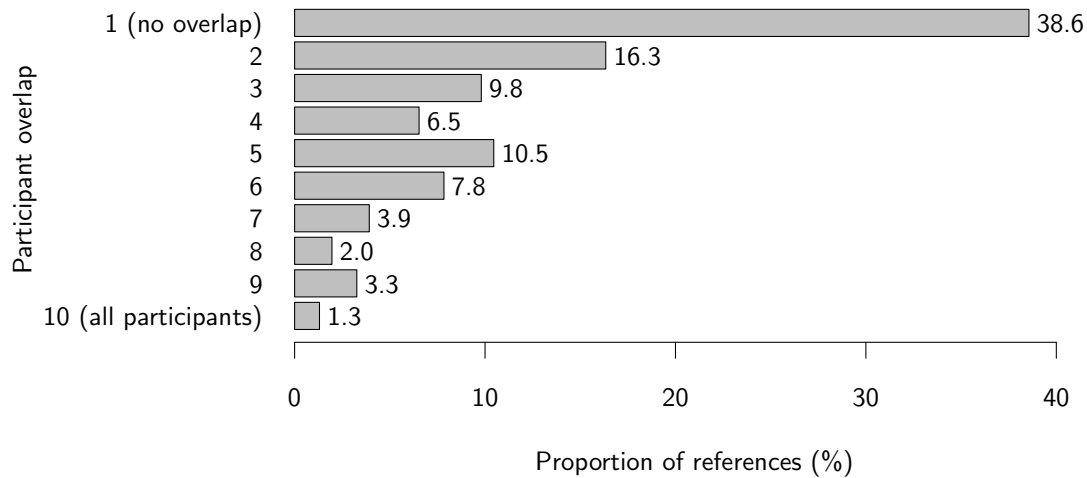
Figure 6.3: Overlap in participants' verbal references. Each bar represents the proportion of all references that were uttered at some point by a particular number of participants. 38.6% of references were uttered by only one participant (any participant), while only 1.3% were made by all participants.

6.2.2   Verbal Data

The two types of coded data described in Section 6.1.5 were first analysed at the solution index level.

For each reference made at each solution index, the number of participants to whom the reference was attributed was counted. Figure 6.3 shows the results. This is an indicator of participant variability. If the participants were identical in their thought processes, then each reference would have been made by all participants. In fact, each distinct reference was made by relatively few participants. Half of all references were made by only one or two participants. About 80% of the references were made by half or fewer participants. This indicates that there is some common ground within subsets of participants, but substantial variation in the thought processes employed overall.

The proximity codes, assigned to each reference as an indication of relevance, were further categorised according to the issues identified in Section 6.2.1. For each issue, the number of references that fell into each proximity category were averaged over all participants for all relevant solution indices. This data is presented in Figure 6.4. References made in the absence of identifiable issues were generally very close to the current line. Unrelated references were proportionally more numerous in cases involving scenario start misidentification, accidental omissions, context switching and condition evaluation. Moreover, the number of references overall was higher where issues were identified.
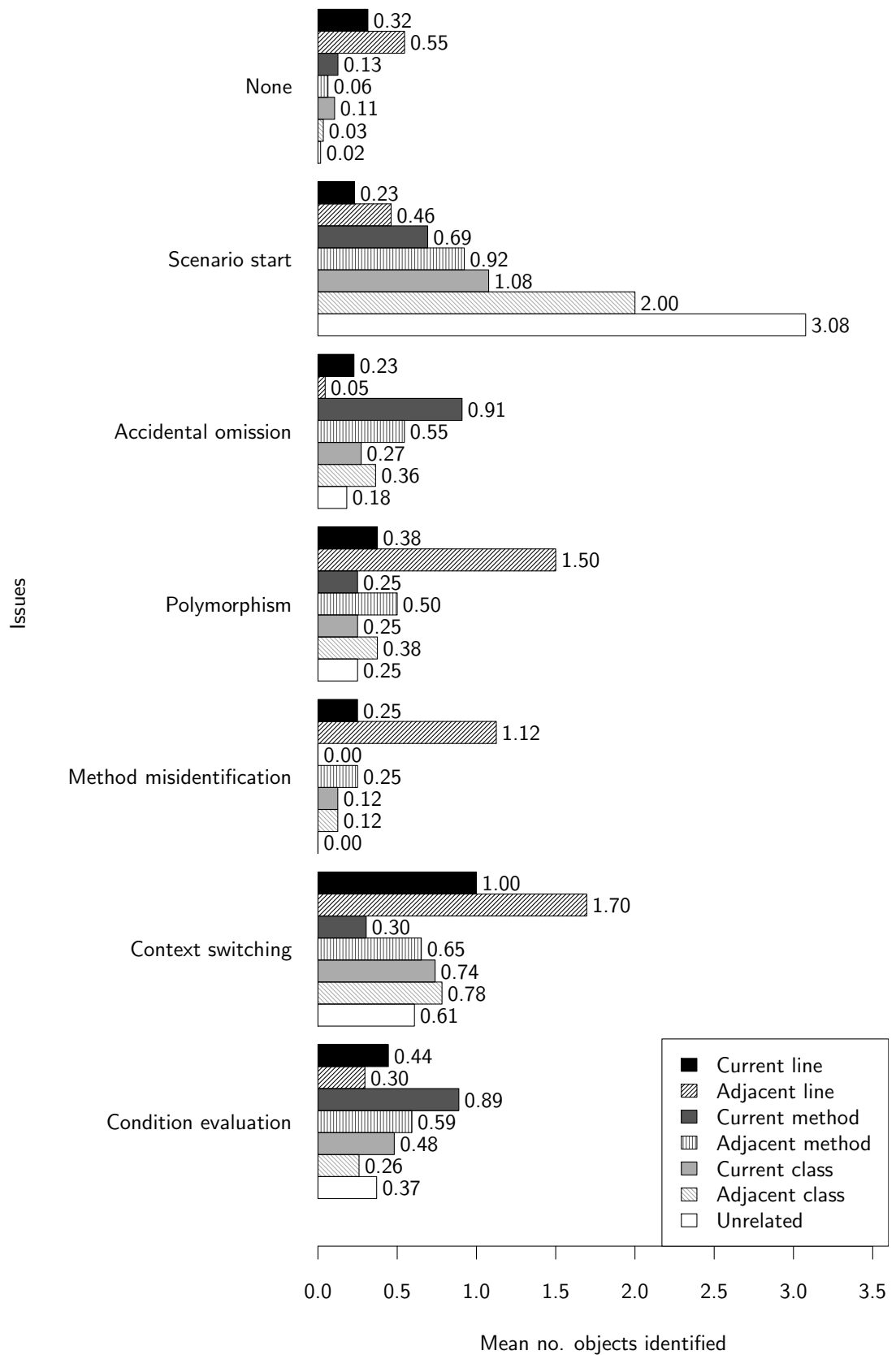
Figure 6.4: The proximity of verbal references to the model solution, for each issue raised (i.e. the mean number of references in each proximity category for each issue).

6.3 Discussion

The model solution coverage for participants in the scenario study, as given in Table 6.4, points to an important problem. Coverage of inspection artefacts — the proportion of an artefact actually subject to human inspection — in UBR is known to be less than 100%, since the available set of use case scenarios is itself unlikely to correspond exactly to all the source code. Dunsmore et al. (2003) cited anecdotal concerns from participants in their study to this effect. However, much of the source code that *is* modelled in a use case scenario is also liable to be effectively left out of a UBR inspection. Participants in the study only identified between 33% and 85% of the relevant code.

The effects of these omissions may be mitigated by involving multiple inspectors, as originally envisaged by Fagan (1976). Although doing so remains a good idea, this is not a perfect solution because the issues encountered by one inspector could apply to others as well.

6.3.1 Misdirection

Many of the issues discovered were not manifested randomly, but occurred at specific points in the use case scenario. For instance, seven out of ten participants in the study encountered difficulty with the same instance of polymorphism. Hence, in UBR inspections polymorphism may well go unnoticed or be misunderstood by several different inspectors, and thus large amounts of relevant code could escape inspection altogether. Such follow-on effects exist for almost all comprehension issues, suggesting the existence of chains of *comprehension dependencies*. Large parts of the overall comprehension effort may fail or be misdirected due to comprehension errors made in a few key places.

The misdirection of the comprehension process is evident in the proportion of line mismatches shown in Table 6.4. Inspectors need not be completely accurate in their analysis of the path of execution; small deviations to examine the effects of exception handling may help rather than hinder inspection performance. However, large deviations will defeat the purpose of UBR, which relies on being able to target specific, high-priority use cases. In the study, on average almost half the lines inspected by participants were deviations from the scenario. The coded verbal data presented in Figure 6.4 does indicate that small digressions from the current point in the scenario occurred even when no identifiable issues arose. When issues did arise, participants digressed further from the scenario, despite the fact that many errors were themselves

in close proximity to the model solution.

Dunsmore et al. express their concern that "more participants using this technique deviated from the recommended application". The risk is that too much time will be spent inspecting non-critical aspects of the software, to the exclusion of its most important functionality. In addition, although switching between artefacts contributes to understanding a software system (Kim et al., 2000, Hungerford et al., 2004), more time spent switching contexts (or resolving other comprehension issues) represents more time likely not spent inspecting anything at all.

6.3.2 Guidance

The challenge for inspection strategies is to identify the specific points at which comprehension issues occur and provide the appropriate level of guidance. They must help ensure that comprehension dependencies are satisfied. In this study, the issues arising may have been mitigated by the presence of additional active guidance or cognitive support targeting problematic constructs.

For instance, such active guidance may include specific instructions to:

- identify method calls, determine whether the method in question is overridden, and thus determine which method implementation is actually executed;

- identify sequence diagram constraints, working backwards from message passing to determine what boolean conditions must be true; and

- verify that all messages encountered on the sequence diagram were also encountered in the source code.

Regarding cognitive support, visual cues could be inserted into artefacts (temporarily, prior to inspection) to highlight the beginning of a scenario, method call sites, called methods themselves, method scopes and where conditional blocks of code (`if`, `switch` and `catch` blocks) are or might be executed. The last may otherwise be inferred anyway where method calls take place. If UBR inspections are to be carried out using a software tool directly, hyperlinks connecting each call site to the set of methods potentially called in the scenario (in polymorphic cases there will be more than one) could be used to further reduce context switching time. These would address the issues described in the previous section.

Deriving the visual cues themselves would require an approach such as that detailed by

Walkinshaw et al. (2005). Their dependence graph technique could largely automate the identification of source code corresponding to a use case scenario. With such support, inspectors would be able to focus their attention to a greater extent on finding defects rather than determining (and often failing to determine) the path of execution. Moreover, inspectors would be free to make digressions as they see fit, while being consciously aware that they are digressions rather than direct examinations of the relevant use case scenario. In combination with the use of visual cues, therefore, such tool support would provide a powerful means to overcome many of the comprehension issues identified in this study. By bridging the gap between source code and use case scenarios, it could also help to ensure that scenarios (in task notation or UML) are kept up to date when the system is modified, and hence could make UBR applicable in situations where it would not otherwise have been.

6.3.3   Cognitive Variation

However, a related problem lies in the provision of too much guidance, particularly active guidance. The pervasive use of active guidance imposes a rigid procedure for traversing artefacts. In light of the variability in participants' verbal references presented in Figure 6.3, it is questionable whether following such a singular, rigid procedure reliably leads to a high level of understanding. Each inspector may have a subtly or markedly different approach to understanding a system, and these different approaches may not always benefit uniformly from the same forms of guidance, especially when a particularly detailed level of guidance is given.

In particular, if the reading technique tries to force the inspector into an unfamiliar approach to understanding the system, then either (a) the inspector will not be as effective as when using a more familiar approach, or (b) the inspector will ignore the reading technique and revert to that preferred approach. Neither outcome is beneficial, and in the second case the technique loses its effectiveness as a means to target specific, high-priority aspects of the system. As well as providing sufficient guidance, reading techniques must refrain from providing extraneous guidance.

6.4   Summary

This chapter has examined comprehension challenges arising in the presence of a reading technique — specifically, use case traversal, as employed in UBR. These challenges included misidentification of the start of the scenario, accidental omission of method

calls, polymorphism, misidentification of called methods, the effort associated with context switching and the difficulty of evaluating dynamic boolean conditions in a static context.

Many of these issues represent unfulfilled comprehension dependencies, where a link in the chain of comprehension is missing, resulting in code being omitted from inspection. Several would also result in additional code being unnecessarily inspected. The error-proneness of identifying the correct method calls and the additional effort associated with context switching highlight the challenges posed by delocalisation. Not only is it difficult to traverse delocalised plans, but even their existence sometimes goes unnoticed. The overall coverage of relevant code by participants was relatively low, in some cases less than 50%. At the same time, on average half the code actually examined was not immediately relevant to the scenario.

These concerns might conceivably be mitigated with an additional level of active guidance, but this would also further constrict the comprehension process. The utilisation of cognitive support may help address such challenges without impinging on inspectors' freedom to apply their own comprehension approaches. This freedom may be an important aspect of an effective inspection, given that inspectors exhibited great variation in their approaches. This variation may reflect a need by different inspectors to have access to different types of information in order to effectively understand the system.

The results shown here should not be seen as a rebuttal to those presented in Chapter 5. Rather, the statechart and scenario studies together suggest that opportunities for improved inspection performance exist in a variety of situations, and that comprehension issues are complex and not generally solvable through the naïve application of a single inspection strategy.

This complexity suggests a need for quantitative investigation, both empirical and theoretical, of the fine-grained effects of guidance and inspector expertise. This is undertaken in subsequent chapters.

Chapter 7

Active Guidance and Defect Detection

"I don't like them doctors. If they start poking around inside me. . . "
"Baldrick, why would anyone wish to poke around inside you?"
"They might find me interesting."

— Blackadder Goes Forth

This chapter examines the fine-grained effects of active guidance and inspector expertise on defect detection. Previous chapters demonstrate that comprehension issues can arise with and without active guidance. However, there does not yet exist a comprehensive explanation of precisely when active guidance should be expected to improve inspection performance, and why it has led to the contradictory outcomes reported in past experiments (as discussed in Chapter 2, Section 2.2).

Thus, this chapter helps to answer the third research question from Chapter 1: *To what extent does active guidance support defect detection, and what are the effects on overall cost effectiveness?*

Porter and Votta (1997) observe that "we have yet to identify the fundamental drivers of inspection costs and benefits", and treating techniques as black boxes will not yield insights into why one does or does not work in particular situations. Many inspection experiments use defect counts — the number of defects detected by a given inspector — as the principal metric by which reading techniques are compared. The characteristics of the defects themselves, and in particular the types of defects detected, are rarely reported. Rather, the defects and inspectors involved are assumed to be representative of a relatively homogeneous population of defects and inspectors seen throughout the

industry. However, such homogeneity is highly doubtful, and representativeness is hard to verify.

Analysis of the overall defect count metric is not generalisable if factors vary between defects, and yet the reading technique itself is such a factor. Reading techniques generally operate by focusing the inspector's attention on defects that are more likely to occur. They do not impart expertise such that any inspector will be equally better able to detect any type of defect. Only a few techniques attempt to assist overall comprehension, and such assistance is not yet convincingly effective (Dunsmore et al., 2003).

Treating experience as a single numerical or categorical indicator suffers the same problem. Plan knowledge, which explains the role of experience in software comprehension (Soloway and Ehrlich, 1984), is specific to patterns and constructs previously encountered by the inspector. Two inspectors with the same number of years of experience may nevertheless not have the same kind of expertise, resulting in different inspection outcomes. The qualitative aspects of experience may be important in determining inspection performance in a given situation.

Neither checklists nor expertise are opaque concepts. Both can be broken down by defect type. A useful checklist does not (and cannot) cover all defect types — to do so would make it either too long, too vague or both. Therefore, it is always possible that defects not covered by the checklist will exist in the system. Expertise is not constrained in the same way, but inspection must nevertheless accommodate the types and levels of expertise at hand.

The checklist experiment described in this chapter examined both checklists and prior exposure to particular defect types on a defect-by-defect basis. The immediate goal of the experiment was to determine the effects of relevant and irrelevant checklist questions and specialised experience on the time and probability of individual defect detection. The experiment is designed to contribute to the construction of inspection theory, and also to provide experience with which future studies might be designed, such that their results can also assist theory generation.

Checklists were chosen as the focus of experimentation because they represent a particularly simple and prevalent (as shown in Chapter 4) form of active guidance.

Understanding factors, like active guidance, that vary between defects would allow software developers to more easily implement effective reading strategies based on the circumstances of a given software project. Different projects and different inspectors would most likely benefit from different reading techniques, or at least variations thereof. For

any given reading technique variant, it will be possible to contrive a situation where it helps detect more defects. Conversely, where the types of defects the technique focuses on are not prevalent in a system, it may actually hinder inspectors.

7.1 Methodology

The experiment was of a two-factor crossover (repeated-measures) design. The independent, within-subject variables were:

- I[1], indicating whether the inspector was previously exposed to relevant defect types;
- A, indicating the presence of a checklist containing a relevant item; and
- H, the code snippet under inspection.

I and A were the treatment variables, each being either 1 or 0 (true or false). Participants were asked to inspect four mini-systems, one to test each treatment combination. H is analogous to a period effect, except categorical rather than ordinal.

The artefacts to be inspected each consisted of a small amount of Java code accompanied by a natural language specification. These snippets were largely developed around the defects to be seeded in them. This allowed for a variety of different defect types, while not depending on large and complex systems to provide opportunities for such defects to arise. Such a dependence would have greatly added to the inspection time required, and so made a repeated-measures experimental design impractical.

Participants were told that a defect represented a deviation of the code from the specification. The code itself contained no syntax errors and was known to compile and run (although additional classes and methods not shown to participants were required for execution).

Participants were made aware that exactly two defects had been seeded in each snippet, and they were instructed to find both. They were not made aware that the two defects had different roles in the experiment: *primary* and *auxiliary*. Primary defects were those designed to be found with the assistance of a checklist and prior exposure to similar defects, where as auxiliary defects were not. Auxiliary defects were not covered by any checklist, nor were participants presented with similar defects beforehand. These defects were intended to capture the effects of checklists and prior exposure on the defect

---

[1]The symbols used in this chapter may appear non-intuitive, but were chosen for consistency with the inspection model introduced in the next chapter.

Table 7.1: A summary of the four snippets inspected by participants.

Snippet	Size [a]	Primary defect	Auxiliary defect
1. SlushFund — determines how to distribute a slush fund.	179 words, 15 LOC	Precision is lost due to integer division.	The output format does not match the specification.
2. TreeNode — outputs a tree structure in XML.	148 words, 12 LOC	Special characters are not escaped in the XML output.	The case insensitivity requirement was not implemented.
3. AddressSearch — searches an address book.	85 words, 14 LOC	A check for `null` fields is missing.	The search algorithm does not match entries properly.
4. WeaponSelector — manages a set of weapons in first-person shooter game.	182 words, 14 LOC	Attempts to add a weapon will overwrite the last weapon.	No bounds checking is done when selecting a weapon.

[a] LOC excludes blank lines, comments and brace-only lines. "Words" refers to the specification of each snippet.

Table 7.2: The procedure for determining the treatment combination (i.e. the values of I and A) at each snippet, based on sequentially-assigned participant IDs.

	Steps	Example (given ID $i = 5$)
1.	The two snippets for which $I = 1$ are chosen from the list $[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]$, based on the result of $(i \operatorname{div} 4) \bmod 6 \in [0, 5]$.	$(5 \operatorname{div} 4) \bmod 6 = 1$, so the chosen pair will be $(1, 3)$ — SlushFund and AddressSearch.
2.	The inverse pair are the snippets for which $I = 0$.	The inverse pair is $(2, 4)$ — TreeNode and WeaponSelector.
3.	The result of $i \bmod 2$ determines which of the two $I = 1$ snippets will have a checklist displayed (i.e. where $A = 1$).	$5 \bmod 2 = 1$, so AddressSearch will have a checklist.
4.	The result of $(i \operatorname{div} 2) \bmod 2$ determines which of the two $I = 0$ snippets will have a checklist.	$(5 \operatorname{div} 2) \bmod 2 = 0$, so TreeNode will have a checklist.

Table 7.3: A summary of the two training snippets used to give participants prior exposure to two particular defect types.

Snippet	Size [a]	Defects [b]
1. Gravity — calculates acceleration due to gravity for a set of physical objects, outputting the results in an HTML table.	175 words, 24–32 LOC	1. Precision is lost in the calculation due to integer division.
		2. Special characters are not escaped in the HTML output.
		3. A check for `null` object labels is missing.
		4. Attempts to add a new physical object actually overwrite the last object.
2. BMI — determines the Body Mass Index of a set of people in a database, given their location.	114 words, 17–19 LOC	1. Converting heights from centimetres to metres uses integer division.
		2. Special characters are not escaped when constructing the database query string.
		3. A check for `null` values returned from the database interface is missing.
		4. Each successive BMI value calculated overwrites the last.

[a] LOC excludes blank lines, comments and brace-only lines. The actual value depends on the variant.
[b] Across all variants. Each variant has two of the four defects.

types they omit. The relative difficulty of finding primary defects compared to auxiliary defects is unimportant, and any difference in difficulty was unintentional. The snippets and the defects therein are summarised in Table 7.1.

The order in which the snippets were encountered by participants was fixed, but different participants received the treatment combinations in different orders. Given four treatment combinations, 24 (4!) orderings were possible. These were assigned to participants based on sequential participant IDs. Thus, the first 24 participants were assigned all 24 permutations, after which the permutations were repeated. This process is illustrated in Table 7.2.

The choice of only one primary and auxiliary defect per snippet was made for practical reasons. The design could have been extended to accommodate more than one of each, but there was no specific need to do so. The repeated measures design already separates the defect effects from the treatment effects.

The experiment measured four dependent variables:

- M_1 and M_2 — binary indicators of whether the primary and auxiliary defects were detected; and
- C_1 and C_2 — the time (cost) taken to detect the primary and auxiliary defects.

The remainder of this section discusses the independent and dependent variables, and the elements of the experiment that support them.

7.1.1   Prior Exposure to a Relevant Defect Type

Participants were given experience by means of two additional training snippets, encountered prior to the four *testing* snippets. The instructions to participants for the training snippets were identical to the testing snippets. The training snippets are summarised in Table 7.3, and were each seeded with analogues of all four primary defects. However, each participant was shown a variant that included only two defects. Six variants of each training snippet were needed to account for the different treatment orderings. Participants received no training for any of the auxiliary defects.

The nature of the training snippets was not made known to participants before or during the exercise. Had they known, they may have made an additional effort (consciously or otherwise) to become familiar with the defect types therein, over and above what would normally occur.

The seeded defects were revealed to participants after completion of each inspection. A short explanation of each defect was given and relevant parts of the code and specification were highlighted. This helped ensure that participants were actually exposed to the intended defect types during training, even if the defects were not found. The defects in each testing snippet were also revealed in this manner, though purely for the benefit of the participants themselves.

7.1.2   Presence of a Checklist

A simple two-item checklist was developed for each of the four testing snippets (shown in Appendix D, figures D.7, D.9, D.11 and D.13). Participants were shown two of the four checklists, in accordance with their assigned treatment permutation. One checklist was shown for a snippet where the participant had received training for the primary defect, and one where they had not.

The first item on each checklist was designed to be relevant for the type of code under inspection, but was not related to any actual defect. The second item related directly to the primary defect. None of the checklist items were repeated between checklists. Some risk existed of participants anticipating the pattern of useful and unuseful checklist items, but their exposure to only two checklists did not provide much opportunity for any potential suspicion to turn into laziness.

Table 7.4: The coding scheme used to categorise defect descriptions.

Code assigned	Meaning
Primary	The description correctly identifies the primary defect.
Auxiliary	The description correctly identifies the auxiliary defect.
Both	The description correctly identifies both primary and auxiliary defects.
False positive	The description does not identify either the primary or auxiliary defects.
Ambiguous	The description plausibly refers to the primary or auxiliary defect, but insufficient information was supplied to convincingly identify it.
Repeated	The description refers to the same defect (primary or auxiliary) as the other description. This is applicable only to the second description supplied by a participant for a given snippet.
Blank	No description was entered.

7.1.3 Detection Probability

For each snippet, participants were provided with two spaces in which to write descriptions of the defects they found. Each defect description was coded according to the scheme outlined in Table 7.4. Codes "primary", "auxiliary" and "both" were applied whenever participants demonstrated some level of understanding of the primary and/or auxiliary defect's faulty logic, even if they misunderstood the actual effect. The "repeated" code was not applied by a human coder, but rather automatically determined by a script based on the initial coding. The "blank" code was likewise automatically assigned. Ultimately, the codes "false positive", "ambiguous" and "repeated" were not treated differently in the analysis.

The defect descriptions for each snippet were presented to the coder in a random order, with no indication of the participant responsible, the associated treatment combination or the other defect description generated in the same inspection.

The dependent variables M_1 and M_2 were either 1 or 0 for each inspection. $M_1 = 1$ if and only if one of the defect descriptions for an inspection was coded "primary" or "both". $M_2 = 1$ if and only if one code was "auxiliary" or "both".

To formally assess the effects of checklists and prior exposure, conditional logistic models were constructed for both primary and auxiliary defect detection. (Logistic regression was discussed in Chapter 3, Section 3.3.) Model simplification (Crawley, 2002) was used to determine the significant terms.

The model terms (before simplification) are as follows:

$$\mathbf{X}_{\mathrm{AIH}} = (A, I, A \times I, H_2, H_3, H_4) \qquad (7.1)$$

These include the checklist (A) and prior exposure (I) factors, a checklist-exposure interaction term ($A \times I$) and three mutually exclusive variables indicating the snippet under inspection (H_2, H_3 and H_4; the first snippet being a reference point for the others). The participant ID (i) identifies a group of observations, to which a random effects term $\zeta_{\delta i}$ can be assigned. Thus, the model is formally expressed as:

$$\mathbb{P}(M_{\delta i} = 1 \mid \mathbf{X}_{\text{AIH}}) = \text{logit}^{-1}(\boldsymbol{\alpha}_\delta \mathbf{X}_{\text{AIH}} + \zeta_{\delta i}) \qquad (7.2)$$

The response term $\mathbb{P}(M_{\delta i} = 1 \mid \mathbf{X}_{\text{AIH}})$ is the probability that participant i detects a primary defect ($\delta = 1$) or an auxiliary defect ($\delta = 2$), given the vector of factors $\mathbf{X}_{\text{AIH}}$. $\boldsymbol{\alpha}_\delta$ is a vector of regression coefficients, estimated using the R statistical package (R Development Core Team, 2009), which indicate the effect of each factor.

7.1.4 Detection Time

The efficient, precise and unobtrusive measurement of detection time for individual defects required that the experiment be conducted through a graphical interface. A web-based system displayed the Java code and specifications, and provided fields into which participants typed the defect descriptions. This experimental infrastructure recorded millisecond timestamps (relative to the beginning of the inspection) for each keystroke entered. These timestamps along with the defect descriptions were stored in a database.

The detection time for a given defect was considered to be the median of the keystroke timestamps. The median was chosen to help guard against keystrokes being entered before a defect was properly identified (e.g. as a test of the interface, or an aborted attempt to describe a defect), or afterwards to tidy up the spelling or grammar. The timestamp of the first or last keystroke, and to a lesser extent the mean timestamp, would have been sensitive to such events.

Alternatively, detection time could have been recorded by the participants themselves, rather than through an online interface. However, prior experience suggested that this may have resulted in substantial amounts of missing data, and a (perhaps subtle) effect on the comprehension process cannot be ruled out.

Survival analysis was used to examine the time required to detect a defect; specifically, Cox proportional-hazards models (discussed in Chapter 3, Section 3.3.2). Where participants did not detect the defect, the required detection time was effectively censored at the time of the last defect description entered. That is, the actual time required to detect the defect was known only to be greater than the length of the inspection.

Effects in a survival model are expressed in terms of the *hazard function*, which is the instantaneous probability of the event (defect detection in this case) occurring immediately after t given that it did not occur before t. An increase in the "hazard" implies a decrease in the required detection time, and vice versa. Cox models make no assumptions about the nature of the hazard function itself, but do assume that factors have a multiplicative effect on it, independent of t (the *proportional-hazards assumption*).

The model is formally expressed as:

$$\mu_{\delta i}(t \mid \mathbf{X}_{\mathrm{AIH}}) = \mu_{\delta i 0}(t) \exp(\boldsymbol{\beta}_d \mathbf{X}_{\mathrm{AIH}}) \tag{7.3}$$

The response term $\mu_{\delta i}(t \mid \mathbf{X}_{\mathrm{AIH}})$ indicates the "hazard" at time t of participant i detecting a primary defect ($\delta = 1$) or an auxiliary defect ($\delta = 2$), given $\mathbf{X}_{\mathrm{AIH}}$ as defined in Equation 7.1. $\mu_{\delta i 0}(t)$ is the *base* hazard function when all independent variables are zero (and is neither assumed nor estimated). $\boldsymbol{\beta}_\delta$ is a vector of regression coefficients.

Linear regression would have been inappropriate here due to the censoring described above. Cases where the defect was not detected could not easily be treated as missing data, because the underlying cause of defect detection is probably closely related to the required detection time.

7.2 Participants

The experiment involved 42 participants, recruited from among students undertaking (or having recently completed) computing-related degrees at Curtin University of Technology. Participation was strictly voluntary, chocolate being an incentive and means of compensation. One participant was excluded from analysis due to technical issues experienced during the exercise. Four (10%) of the included participants were in their first year of study, while 15 (37%) were in their second year, 14 (34%) in their third and the remaining eight (20%) had graduated, some of whom were undertaking further study.

Other background information collected from participants included their prior experience (if any) and their rough level of exposure to software inspections. Of the 41 participants, 36 (88%) indicated that they had not worked before as a software developer. The other five participants (12%) claimed between six months and three years experience.

Table 7.5: P-values testing the proportional hazards assumption for required defect detection time.

	Primary	Auxiliary
Checklist (A)	0.855	0.326
Prior exposure (I)	0.990	0.659
Interaction ($A \times I$)	0.203	0.477
Snippet 2 vs 1 (H_2)	0.849	0.675
Snippet 3 vs 1 (H_3)	0.715	0.598
Snippet 4 vs 1 (H_4)	0.721	0.546
global	0.896	0.835

Six participants (15%) indicated that they had participated in a software inspection or review. Nine (22%) said they had studied software inspections/reviews. Twenty-five (61%) said they had read and understood real-world source code not written by them. These options were not mutually exclusive.

7.3 Threats to Validity

Potential threats to the experiment's internal validity included the correctness of the proportional-hazards assumption and the reliability of the coding scheme. There was no evidence of a violation of the proportional-hazards assumption, p-values for which are shown in Table 7.5.

The coding scheme in Table 7.4 was tested for reliability by having a second expert categorise 11 (approximately one quarter) of the defect descriptions for each snippet, through the same process outlined in Section 7.1.3. The interrater agreement statistic Kappa was calculated to be 0.736. However, since the classifications "false positive" and "ambiguous" are not treated differently in the analysis, disagreements involving these two categories are not of interest. Merging them yielded a Kappa value of 0.784, which is slightly above the threshold for "excellent agreement" (0.78) (El Emam and Wieczorek, 1998).

Inspector variability was not considered a threat to interval validity due to the repeated measures design. In general, differences between inspectors' approaches to the reading task are great, and can overwhelm differences between reading techniques (Uwano et al., 2006). In particular, differences among participants in this experiment was considerable, as shown in the previous section. However, repeated measure designs allow treatment effects to be analysed independently of such participant variability. The participant ID served as a stratification variable in the logistic and Cox proportional-hazards models.

Table 7.6: A broad summary of the significant effects found in the experimental data.

	Detection probability		Detection time	
	Primary defects	Auxiliary defects	Primary defects	Auxiliary defects
Checklist (A)	positive	negative	no	negative
Prior exposure (I)	no	no	no	no
Interaction ($A \times I$)	no	no	no	no
Snippet (H) [a]	yes	yes	yes	yes

[a] Snippet effects are not "positive" or "negative" since there is no meaningful reference point.

Although the use of an electronic interface may impede anyone used to working with paper-based artefacts, this was not considered a problem. Any such effect is merely a component of the random participant effect, dealt with by the repeated-measures design.

Threats to the external validity of the experiment include the size of the snippets under inspection, the number of defects in each snippet, the size of the checklists and the experience of inspectors. Each of these could itself be an important factor in determining inspection performance. The artefacts were all very small (as shown in Table 7.1), there were only two defects in each, there were only two items on each checklist, and the inspectors were all students.

Interactions between such factors and the experimental treatments are not inconceivable. However, there is no obvious reason to expect fundamentally different results had the artefacts been larger, the number of defects higher or the number of checklist items greater. Scope exists for exploring any such effects in future studies. The interaction between the reading technique and inspector experience is in part what this experiment is intended to find, by pairing checklists with prior exposure to specific defect types.

7.4 Results

The results of the experiment are broadly summarised in Table 7.6. The following subsections give descriptive and statistical accounts of the data, and detail participants' own perception of their activities.
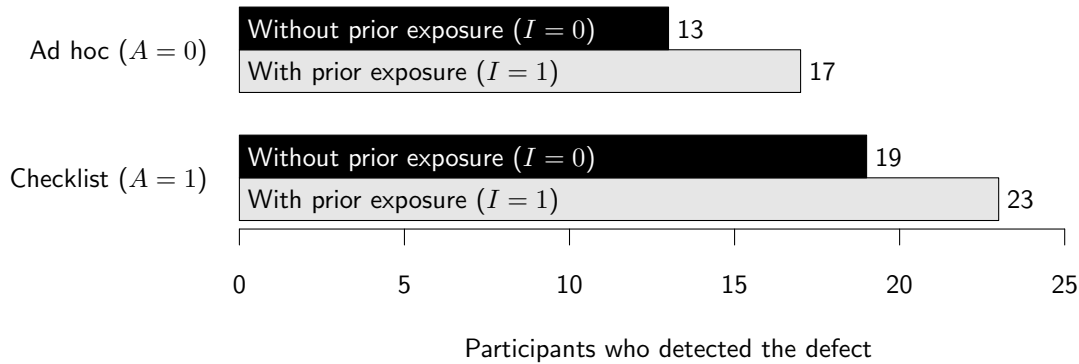
Figure 7.1: Participants who detected the primary defect for each treatment combination.
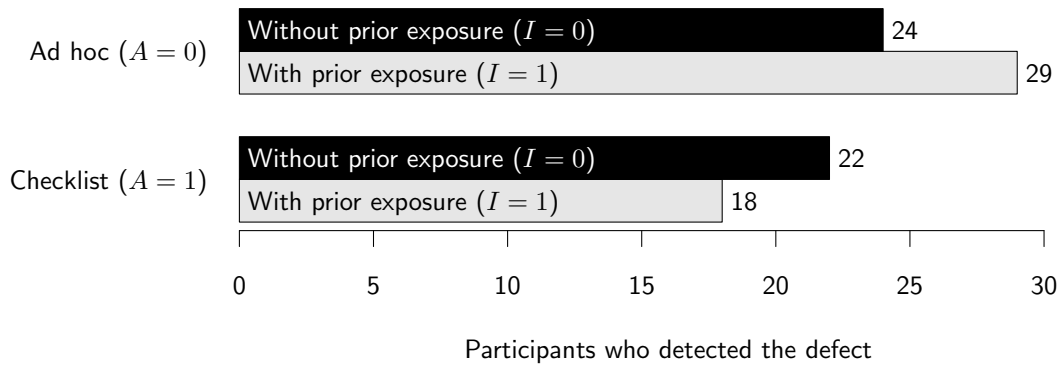
Figure 7.2: Participants who detected the auxiliary defect for each treatment combination.

7.4.1 Detection Probability

On average, participants detected 4.02 out of 8 defects in the four testing snippets, consisting of 1.76 out of 4 primary defects and 2.27 out of 4 auxiliary defects (with rounding).

Figure 7.1 shows how detection of primary defects varied according to the checklist and prior exposure factors. At first glance, there are positive effects for both checklists and prior exposure, with the checklist effect outweighing the prior exposure effect. There is no apparent interaction here between the two factors.

Figure 7.2 shows the detection of auxiliary defects. The smaller disparity between the two black ($I = 0$) bars than between the grey ($I = 1$) bars suggests that here there is interaction between checklists and prior exposure. The main effect of checklists is negative, whereas the effect of prior exposure appears to be masked by the interaction.
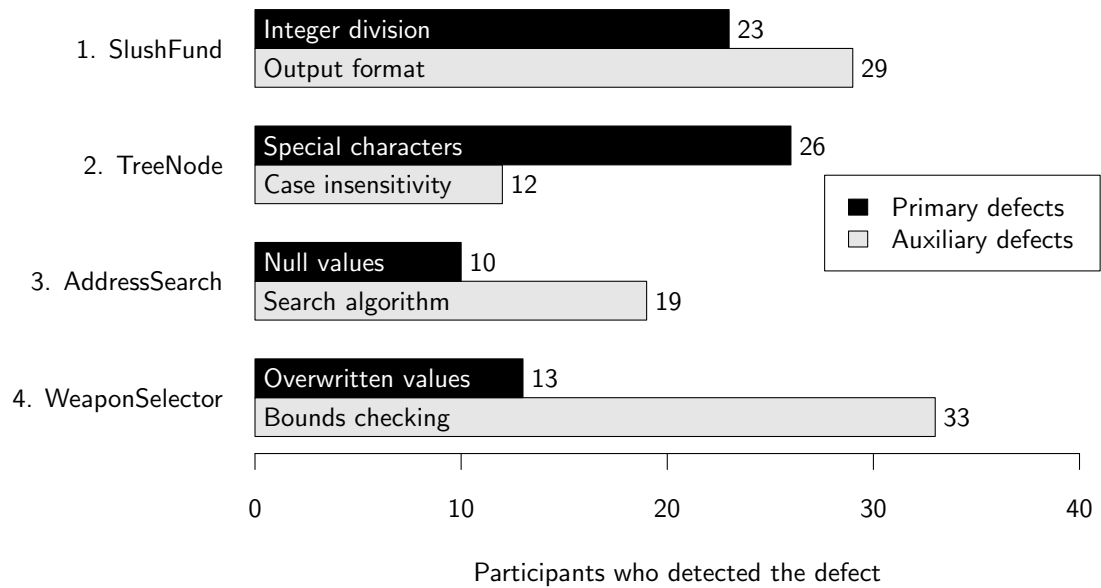
Figure 7.3: Participants who detected each of the eight actual defects.

Table 7.7: P-values for defect detection probability.

	Primary	Auxiliary
Checklist (A)	0.010 *	0.008 *
Prior exposure (I)	0.287	0.544
Interaction ($A \times I$)	0.788	0.176
Snippet (H)	< 0.001 *	< 0.001 *

* Significant effects ($p < 0.05$).

In the presence of a checklist, prior exposure appears to have a negative effect, but in the absence of a checklist the effect seems to be positive.

Figure 7.3 suggests that defect detection is also affected by the defect itself and the snippet under inspection. Later snippets did not exhibit any higher defect detection rates than earlier snippets. The effect of any accumulated experience with the overall process, if it exists, was obscured by the effects of individual snippets and individual defects. Comparing primary and auxiliary defects must be done with caution, since in three quarters of cases the detection of primary defects was assisted by checklists and/or prior exposure.

Table 7.7 gives the significance levels of the A, I and H terms and the $A \times I$ interaction term. At the 5% level the checklist and snippet effects were significant for both primary and auxiliary defects. There were no significant prior exposure or checklist-exposure interaction effects.

Table 7.8: P-values for defect detection time.

	Primary	Auxiliary
Checklist (A)	0.163	0.014 *
Prior exposure (I)	0.180	0.713
Interaction ($A \times I$)	0.490	0.867
Snippet (H)	0.007 *	< 0.001 *

* Significant effects ($p < 0.05$).

The checklist effect on the log odds of primary defect detection lies in the range 1.1593–3.874, with 95% confidence. This effect is multiplicative, meaning that the presence of a relevant checklist item increased the log odds of detection by between 16% and 287%. For auxiliary defects the 95% confidence interval is 0.2314–0.841. Thus, the presence of a checklist without a relevant item reduced the log odds by between 16% and 87%.

7.4.2 Detection Time

Figures 7.4 and 7.5 show the probability according to the data of a defect being detected at each point in time. A separate line is shown for each treatment combination. The detection rates appear to be approximately linear until about six minutes into the inspection, at which point they begin to plateau.

Up until about four minutes into the inspection, there is little difference between the effect of a checklist and the effect of prior exposure on primary defect detection probability. Both effects appear to be positive, and in combination they are no more effective than each by itself. After about four minutes, the checklist effect becomes greater than the prior exposure effect, and eventually the combination of checklist and prior exposure appears to outperform each by itself.

A negative checklist effect on auxiliary detection probability is evident graphically after about three minutes. By contrast, the effect of prior exposure does not appear to manifest itself until after about six minutes. The effect is positive when no checklist is present, in spite of the prior exposure being related to a different kind of defect. However, when a checklist is present the effect is negative.

Table 7.8 gives the significance levels of the terms. At the 5% level the snippet effects were significant for both primary and auxiliary defects, but the checklist effect was significant only for auxiliary defects. There were no significant prior exposure or interaction effects.
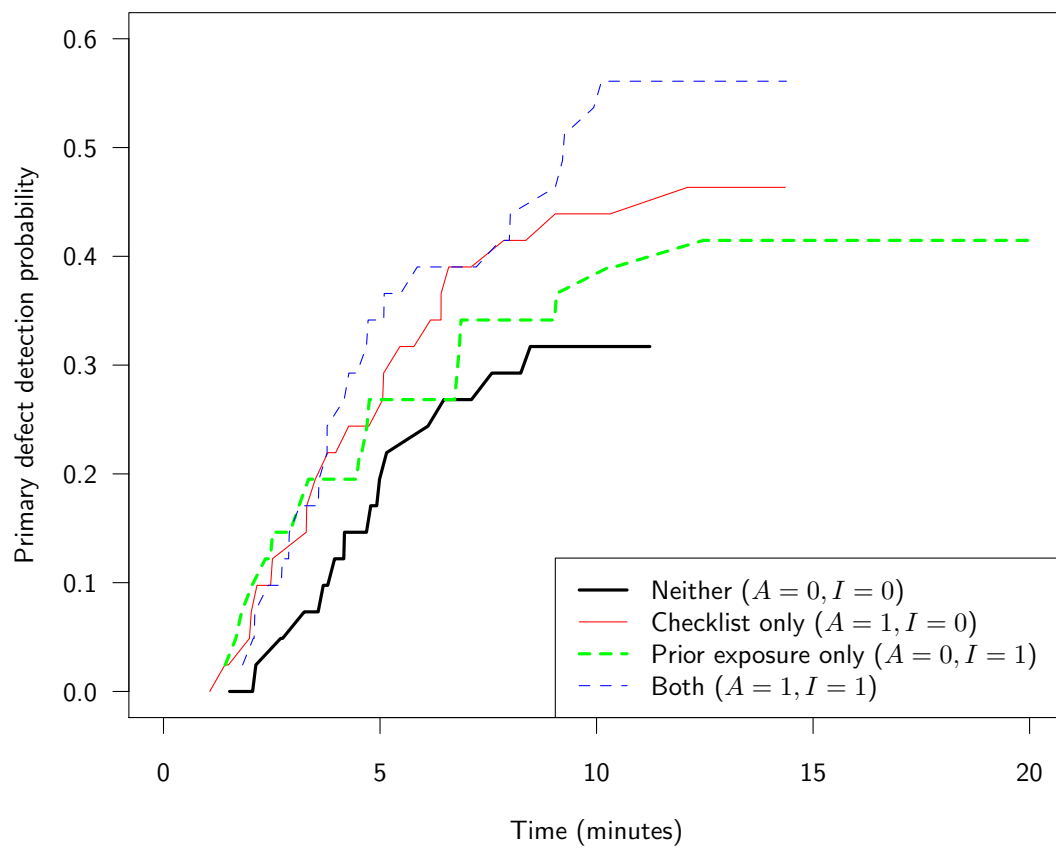
Figure 7.4: The probability of detecting the primary defect within a given time, for each treatment combination.
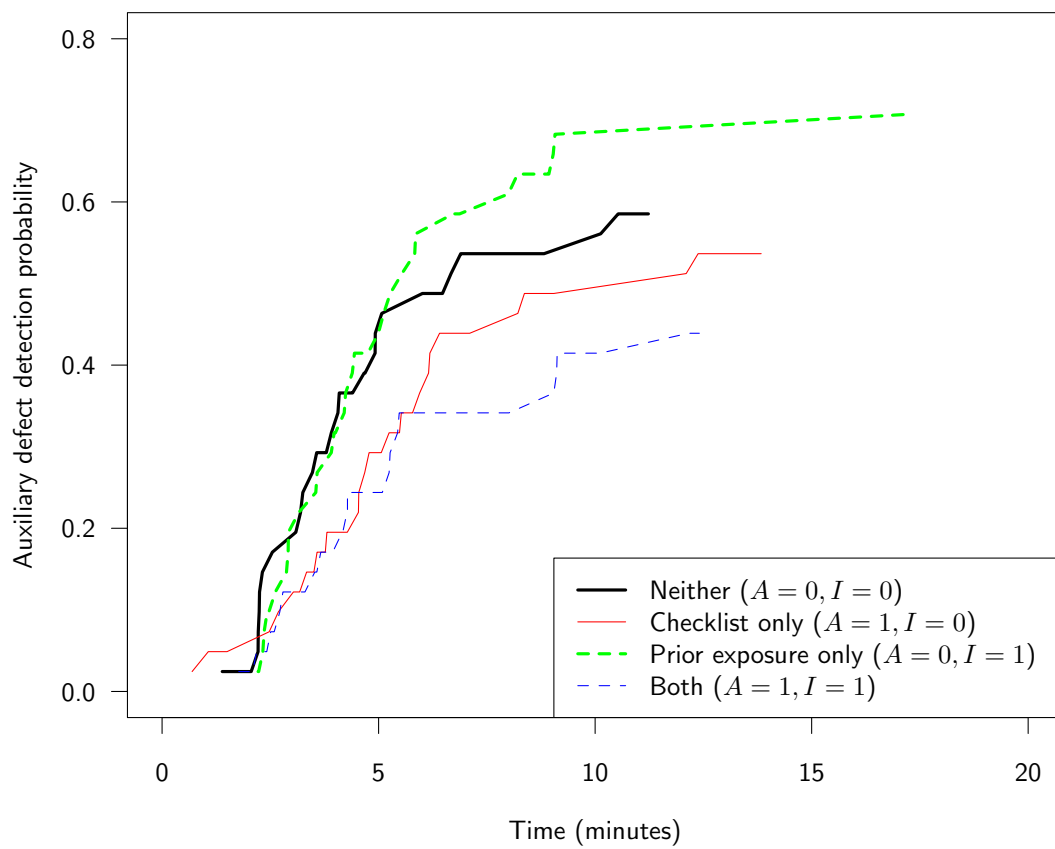
Figure 7.5: The probability of detecting the auxiliary defect within a given time, for each treatment combination.

The checklist effect on the hazard function for auxiliary defect detection time is in the range 0.405–0.916 (multiplicative), with 95% confidence. Thus, checklists reduce the "hazard" and increase the required detection time for auxiliary defects.

7.4.3 Perception

Participants were asked several questions related to the experimental factors, by means of a short questionnaire filled out after completion of the exercise. A small number of responses to these questions were not readily classifiable (e.g. because two or more multiple-choice boxes had been ticked).

Twenty-seven participants (66%) considered the checklists to have assisted them. However, eight (20%) indicated that it made no difference, and five (12%) reported a negative effect. Thirty-five (85%) participants felt that their prior exposure to certain defect types (via the training exercises) helped, while six (15%) said it made no difference. For prior exposure, the prospect of a negative response was not considered plausible, and so no such options were provided on the questionnaire.

Participants reported varying degrees of checklist use. Twenty-three (56%) reported checking each item carefully but not relying exclusively on the checklist, while two more (5%) reported using the checklist exclusively. Eleven (27%) indicated that they at least looked over the checklist, and five (12%) said they ignored it. All of the twenty-seven participants who said the checklist helped used it to some extent. Of the Thirteen who said it did not help, five ignored it, two looked over it, and six checked it carefully.

Only eleven participants (27%) reported that in general they found defects as a result of the checklist. The rest indicated that they found defects before, during or after checklist consultation, but not as a result. This is somewhat at odds with the perceived effect of the checklists. Of the twenty-seven participants who said the checklist helped, only ten reported generally finding defects as a result of consulting it. Another ten said they generally found defects during checklist consultation.

7.5   Discussion

7.5.1   Checklists

The effects of the checklists on defect detection probability can be explained by inspectors' attention being re-focused on particular defect types. Attention is drawn towards defects covered by the checklist, and drawn away from defects not covered. It might be expected that detection time would be similarly affected. However, the checklists did not reduce the time required to detect defects covered by the checklist, even though they increased the time for non-covered defects.

These results suggest that the overall checklist effect examined in many reading technique experiments is really the aggregate effect resulting from the particular defects seeded and the checklist questions chosen. In an industry context, if many of the types of defects likely to arise can be predicted in advance, based on historical data or expert opinion, it should be possible to construct a relatively effective checklist. Conversely, if relatively few defect types can be predicted, a checklist may reduce inspection efficacy. Decisions to use checklists should take into consideration the number of actual defects likely to be covered by the checklist, as far as this can be estimated.

7.5.2   Prior Exposure

Intuitively, prior exposure should have improved the detection time and probability of primary defects. It is not clear what effects, if any, should have been expected for auxiliary defects. Like checklists, prior exposure may also result in a form of cognitive resource allocation, or it may increase the overall cognitive resources available for the task, and thus have a negligible impact on unfamiliar defects.

However, none of the prior exposure effects were significant, perhaps because the two training snippets involved insufficient exposure to similar defects. If the non-significant positive effect on primary defects is real, it can be attributed non-controversially to improved plan knowledge. If the effects observed on auxiliary defects are real, they might require a deeper psychological explanation. It is not obvious why prior exposure to one type of defect should improve the prospects of detecting unrelated defects, or why this effect should only occur in the absence of a checklist. Future research may employ a greater level of training, and thus explore the nature of any such effects.

However, the evidence is suggestive (if not confirmatory) of a much more narrowed focus

resulting from the combination of checklist and prior exposure. This may be a further warning against the use of active guidance for inspectors with matching expertise. That is, any active guidance given should complement the inspector's expertise, not coincide with it.

The lack of any significant main prior exposure effect should not be taken as a sign that experience generally does not matter. Experience in this experiment was attained through short training activities, not through months or years of exposure to certain programming constructs.

7.5.3 Snippets

The snippet had a significant effect in all cases. These effects might result from the defects within each snippet as much as the nature of the code itself, since each snippet had different defects. Nevertheless, the effects found demonstrate that inspection performance depends very much on the artefacts being inspected.

7.5.4 Perception

The perceptions held by a majority of participants concerning the experimental factors were incongruous with the results. Most participants indicated that checklists and prior exposure helped. However, if the results from primary and auxiliary defect detection are combined, both factors had very little net effect. The positive perception of the two factors may arise because defect detection is an entirely conscious activity; by its nature, inspectors see directly the factors leading to it, and are blind to factors inhibiting it.

This illustrates the pitfalls of using anecdotal evidence to support particular inspection strategies. An individual cannot be expected to objectively assess the utility of a checklist (or other inspection strategy) because he/she has no immediate reference point. In any given inspection, an individual cannot know what defects would have been detected if a different strategy had been used.

The results also demonstrate that inspection strategies can have important effects on an inspection that are not reflected in the simple defect count metric.

7.6   Summary

The checklist experiment described in this chapter examined the effects of checklists and specific inspector experience on individual defect detection.

Overall, the results support the utility of checklist questions in some situations, where defect types are easily predictable. By extension, this suggests that active guidance in general should be effective where the underlying concepts and structures in a system are relatively well-known. This is evident in the statechart and scenario studies described in chapters 5 and 6, where unambiguous but relatively complex model solutions could be derived. Conversely, in situations where the defect types are wholly unpredictable, checklists may hinder inspection performance. Similarly, active guidance in general should be avoided where the notations and structure of a system do not follow well-established conventions.

This parallels the argument made in Chapter 6 that active guidance may not be effective if it conflicts with an inspector's preferred comprehension strategy. Underlying both points is the notion that active guidance must be relevant. It must target the links within the comprehension dependency chain — the pieces of knowledge most likely to lead to a level of understanding sufficient for defect detection. Defect types or programmatic constructs that lie outside this chain, either because they do not exist in the system, or because they are not particularly useful for a given inspector, should not be the subject of active guidance. The argument is not merely that such active guidance would be superfluous, but that it would actually be a hindrance to effective comprehension and defect detection.

The results also raise the issues of inspector variability and system variability, the first of which was reported in Chapter 6. This experiment used a crossover design to reduce the influence of individual participants on the results, but artificially re-created variability in inspector expertise in order to examine its consequences. Inspector expertise did not have a statistically significant effect, but the results are nonetheless suggestive of an interaction with the presence of a checklist. The system itself did emerge as a significant factor determining defect detection probability and time.

There are few techniques in software engineering that are unquestionably appropriate in all situations. Rather than providing software engineers with results that suggest simply that a technique can be effective, it would be more valuable to articulate the circumstances in which it is likely to be effective.

It should be possible to appeal to theory to predict inspection efficacy, with some

margin of error, given information on the nature of the system, the probable types of defects therein, the inspectors and the reading technique. The results presented here do not definitively resolve the uncertainty surrounding reading techniques. For instance, they do not explain why Hatton (2008) found no improvement using a checklist, even though all the seeded defects were in some fashion covered by it. Hatton speculates that the wording of the checklist, or other such subtle characteristics, may have an effect. Further research should examine the potential effects of such factors along with checklist size, system size and the number of defects on inspection performance.

The next chapter proposes a theoretical framework and model of the inspection process, taking into account fine-grained effects associated with active guidance, individual inspectors and system composition.

Chapter 8

Inspection Modelling

"Well, my cousin Bert Baldrick... says he heard that all portraits look the same these days, since they are painted to a romantic ideal, rather than as a true depiction of the idiosyncratic facial qualities of the person in question."
"Your cousin Bert obviously has a larger vocabulary than you do, Baldrick."

— Blackadder the Third

The empirical investigations described in the preceding chapters lead to the following observation: that the mechanics of software inspection are as complex and multifaceted as the systems under inspection and the people who inspect them. Given this reality, it is implausible that a single inspection strategy could outperform all others in all circumstances. However, rather than leaving the choice of inspection strategy to chance and intuition, the results presented thus far give some insight into how inspection theory might be developed.

This chapter describes an approach to modelling software inspection. First, a theoretical framework is defined, giving an abstract, qualitative description of the software development process. The framework incorporates a set of concepts applicable across all software development processes and methodologies. Such generality is achieved through abstraction. Based on the framework, a model of software inspection and the broader software verification process is defined. The model's outputs are the costs associated with software inspection strategies (measured in units of time and/or currency) and a normalised, unitless measure of *cost effectiveness*.

The model is implemented using a Monte Carlo simulator, by which selected inspection strategies are compared. A sensitivity analysis is also conducted on selected model parameters to explore the behaviour of the model.

The simulation results principally address the third research question from Chapter 1 concerning active guidance (in combination with the checklist experiment). However, in some respects they also provide further insight into the second question concerning comprehension.

The model is proposed, in part, as a formalised, generalised explanation for the observations noted in preceding chapters. In particular, it seeks to address the problem of reading technique selection. However, the model is also intended to have explanatory power as well as predictive power. Its component parts are intended to reflect the mechanics of software development, not merely to serve as a black box.

This chapter is organised as follows. Section 8.1 presents the theoretical framework on which the model is based. Section 8.2 articulates the logic of the model itself. Section 8.3 describes the implementation of the model and the simulation results obtained. Finally, Section 8.4 discusses these results in the broader context of this thesis and the published literature.

8.1   Framework Concepts

The framework is a set of concepts and structures intended to support a model of inspection cost effectiveness.

8.1.1   Entities

Central to the framework are *entities*, which provide a fine-grained, abstract representation of a software system. Entities come in two broad varieties: *localities* and *k-instances* (knowledge instances). A more detailed taxonomy of entities is shown in Figure 8.1.

Localities are the physical components of the artefacts describing the software, and as such are immediately visible. K-instances form a higher-level semantic representation, requiring some level of comprehension (or at least identification). By contrast, localities are simply *searched* (i.e. scanned, looked over, etc.) rather than understood; they are merely the syntactic components of software artefacts. The framework views defects in a system as being k-instances. Though unintentional, defects are nevertheless distinct aspects of the software that require human cognition to deal with.
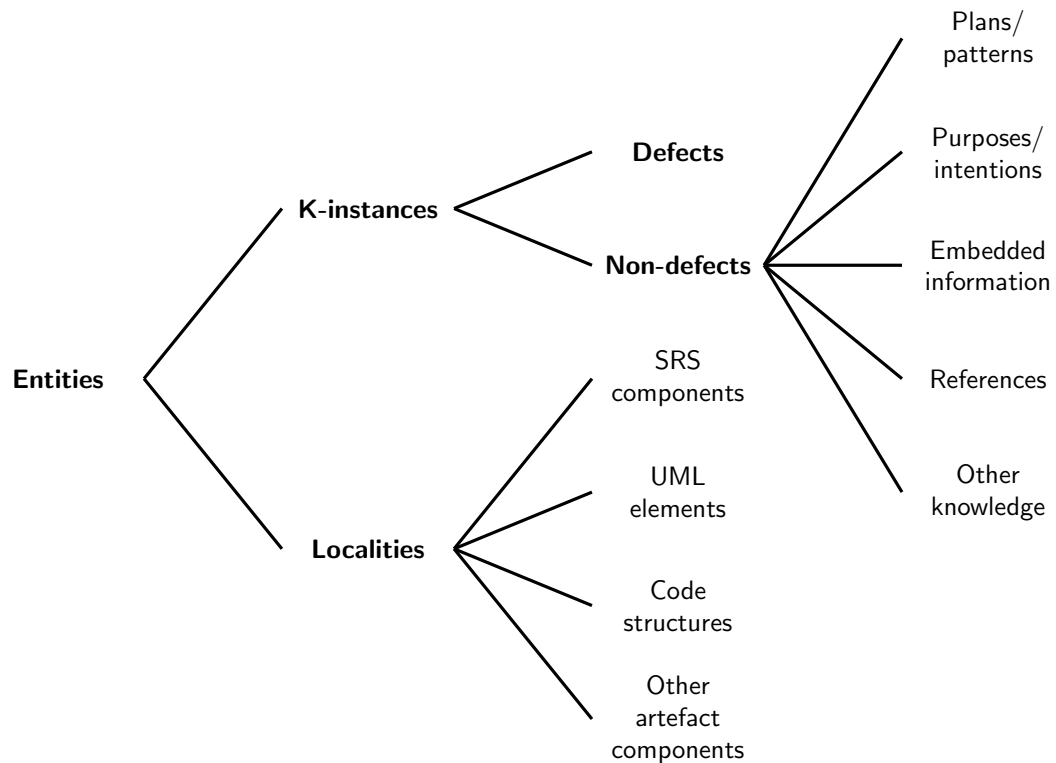
Figure 8.1: Taxonomy of entities. The bold categories are those with special meaning in the model. The "other" categories are included to show the open-ended nature of the taxonomy.

Localities may include sections of requirements documents, elements of UML diagrams, code structures, or any other physical artefact components, or entire artefacts. For example, a sequence diagram, the specification of a functional requirement, and a `while` loop might all be considered localities. They are locations in the system (or rather, its artefacts) open to direct human observation.

K-instances can be further grouped into defects, plans and patterns, purposes and intentions, embedded information and references. Plans (as discussed in Chapter 2, Section 2.3.2) and patterns are abstract knowledge structures, acquired by programmers through experience, that represent re-usable solutions to particular software development problems. Purposes and intentions are the conscious decisions of software developers; i.e. developers' notions of what the subcomponents of the system are supposed to accomplish and why. Embedded information refers to any information expressed literally, such as in comments, annotations and identifiers. References are atomic occurrences of a particular name or identifier.

The above locality and k-instance subcategories are discussed simply to illustrate the nature of localities and k-instances, and may not be exhaustive. The model based on this framework only distinguishes between localities, defects and non-defect k-instances.

However, entities are also subject to a more comprehensive classification scheme, one that the framework itself does not specify. Rather, this is left as a separate task to be undertaken at the time the framework is utilised in a practical setting (e.g. in the development or evaluation of a particular inspection strategy), and depends on the software development environment. However, there are several desirable properties for a classification scheme:

- *Predictive power.* For modelling purposes, the classification scheme should identify entity types that differ substantially with respect to measurable quantities. Such quantities may include, for instance, the time taken to inspect a locality, the probability of finding/understanding a k-instance or the cost of repairing a defect. If a given classification scheme is a good predictor of various quantities that take part in a model, then its use will improve the model's precision.

- *Detectability.* The goal of the framework is to support the development and evaluation of reading techniques. Entity types form a vocabulary for the representation of reading techniques. In essence, a reading technique is a series of instructions telling the inspector where to look and what to look for. However, it is possible to devise entity types that do describe a system (and have predictive power) but which the inspector cannot usefully be instructed to find.

  For example, this occurs in the following naïve instruction:

  > Find defects most likely to result in a high failure rate.

  Such defects are important, and should be found as a matter of priority. The high failure rate characteristic can be a classification criterion. However, a high failure rate is not a basis for defect detection during inspection, because such information is not easily obtained by an inspector. In attempting to follow the instruction, the inspector is unlikely to be any more effective. Since reading techniques cannot usefully incorporate such instructions, entity types within the framework should be detectable.

- *Granularity.* Classification schemes may have varying levels of granularity. A coarse-grained scheme with only a few categories will be simpler. However, certain reading techniques may be effectively unrepresentable if there are insufficient entity types. For instance, the difference between the scenario-based and checklist-based techniques is largely one of detail. Without appropriately fine-grained entity types, these two approaches may become indistinguishable, defeating the purpose of the framework.

The studies described in previous chapters implicitly use entities in their methodology and analysis. These are shown in Table 8.1.

Table 8.1: Entity types used in previous chapters.

Study		Locality types	Knowledge types
Statechart study	(Chapter 5)	• statechart • source code fragment	• state • transition • label • decision • mutation • binding
Scenario study	(Chapter 6)	• sequence diagram • class • method • line	• boolean condition • scenario participation • method reference • method call • constructor reference • field • variable • message • object • sequence diagram flow control
Checklist experiment	(Chapter 7)	• specification • source code	• integer division defect • output format defect • special character defect • case insensitivity defect • null value defect • search algorithm defect • overwritten values defect • bounds checking defect

8.1.2 Dependencies

Entities are related through a network of directional dependencies, as suggested in previous chapters. These are as follows:

- A *comprehension dependency* between two k-instances means that the comprehension of one relies on (or is assisted by) the comprehension of the other. References should have no comprehension dependencies, being atomic and easily identifiable. It is unlikely that any k-instance would depend on a defect, except conceivably another more complex defect. Plans, patterns, purposes and intentions, meanwhile, might depend on each other and have other k-instances depend on them.

- A *locality dependency* exists between a k-instance and a locality, where comprehension of the k-instance depends on (or is assisted by) searching the locality. This would typically mean that the k-instance derives from information physically located within the locality.

- A *decision dependency* occurs where the act of searching a locality is made more likely by having first comprehended a particular k-instance. That is, the k-instance provides some indication of where the inspector's efforts should be directed.

Figure 8.2 shows an example of a dependence structure between entities. Here, the f1 and f2 localities represent the source code for two functions. The k-instances purpose_of_f1 and purpose_of_f2 represent the high-level semantics of these functions. The k-instance call_to_f2 represents the call from f1 to f2. Names are given to these entities only for the purpose of discussion.

The dependencies between these entities can be explained as follows:

- The purpose_of_f1 k-instance has comprehension dependencies on call_to_f2 and purpose_of_f2. Broadly, in order to understand the purpose of function f1, it is necessary to understand its actual contents (in this case, identifying the call to f2) and also the purpose of f2.

- The purpose_of_f1 k-instance also has a locality dependency on f1, meaning that the inspector must observe the source code for f1 in order to understand its purpose. Similarly, purpose_of_f2 has a locality dependency on f2.

- The call_to_f2 k-instance has a locality dependency on f1 because it is physically located in the f1 function.
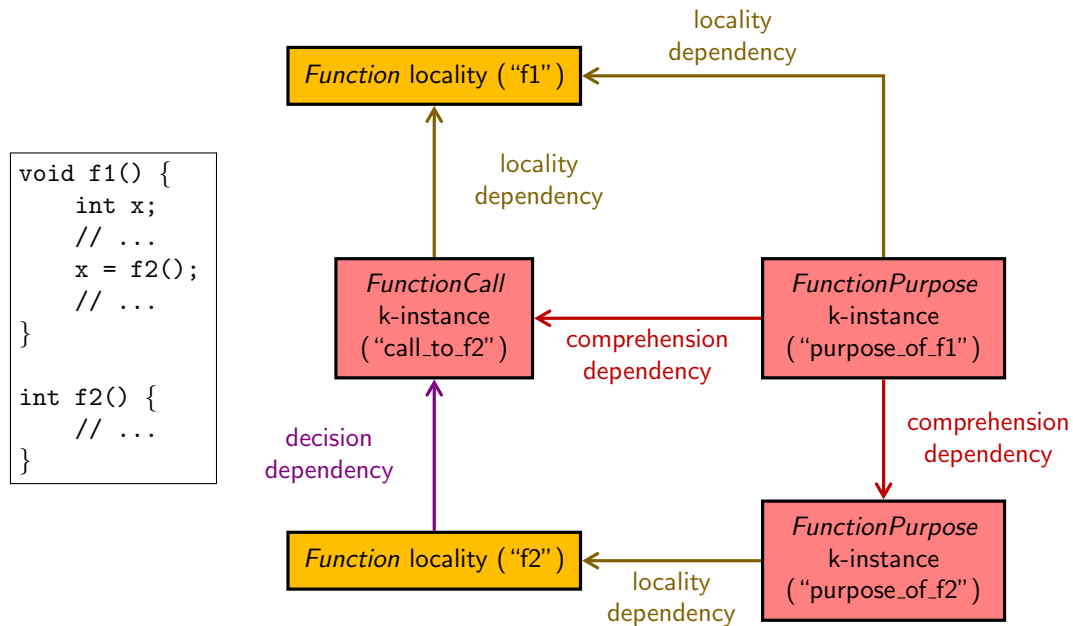
Figure 8.2: An example of a dependence structure representing a snippet of source code, where function `f1` calls function `f2`. The classification scheme comprises the "Function" locality type (yellow/orange) and the "FunctionCall" and "FunctionPurpose" knowledge types (red).

- The f2 function has a decision dependency on call_to_f2. The identification by the inspector of call_to_f2 increases the probability that the inspector will examine the f2 locality.

Some restrictions exist on dependencies between entities. In particular, localities cannot depend directly on other localities, and cycles are prohibited (so as to simplify the model). There are likely to be additional constraints imposed on the dependency structure by the entity types involved. In the example shown in Figure 8.2, a FunctionCall k-instance would probably not depend on a FunctionPurpose k-instance (even if cycles could be resolved); knowing a function's purpose is unlikely to improve the inspector's ability to identify function calls.

The studies presented in previous chapters also imply dependencies between entity types identified therein. For example, in statechart study described in Chapter 5, each state and transition have a locality dependency on the statechart because they are physically located within it. The decision, mutation and binding roles have locality dependencies on source code fragments for the same reason. The decision and mutation roles also have comprehension dependencies on states and transitions; the latter are required to understand the former. Finally, an instance of binding has a comprehension dependency on decisions and mutations (and potentially other bindings as well); binding code by

definition connects other code fragments. Such a formalised dependency structure is not strictly necessary to understand the statechart study itself, but may be useful in illustrating how dependencies arise.

In the scenario study described in Chapter 6, the various types of references each have dependencies on the localities in which they occur. For instance, field references depend on classes, variables depend on methods and messages depend on the sequence diagram. Scenario participation (i.e. whether a line is executed in the given use case scenario) depends on the line in question, any relevant boolean conditions and whether the previous line participated in the scenario. Decision dependencies also exist here; class and method localities depend on method call k-instances, in a similar fashion to the example in Figure 8.2. This may not be an exhaustive list of all the dependencies arising in the study.

Finally, in the checklist experiment described in Chapter 7, a simpler dependency structure arises. The experiment utilised eight defect types, most of which depended on both the specification given and the source code snippet. This simple structure largely reflects the quantitative nature of the experimental analysis rather than the underlying comprehension process. Had more knowledge types been identified and examined within the experiment, a richer variety of dependencies may have been apparent.

8.1.3 Markers

Markers are an extension to entity classification. They are assigned to individual entities to indicate properties that are not part of the classification scheme (particularly due to the detectability requirement). Markers may represent importance, complexity or other visible characteristics that influence activities undertaken in the development process.

As with the classification scheme, the set of possible markers is not supplied by the framework itself. There may be several distinct markers representing different measures of importance, and several indicating different types of complexity. Other, unrelated markers are also possible.

Whereas entity types are exclusive (an entity has exactly one type), markers are independent of one another. Each entity may be assigned any subset of markers, including none.

Localities have markers explicitly assigned to them, whereas k-instances are implicitly
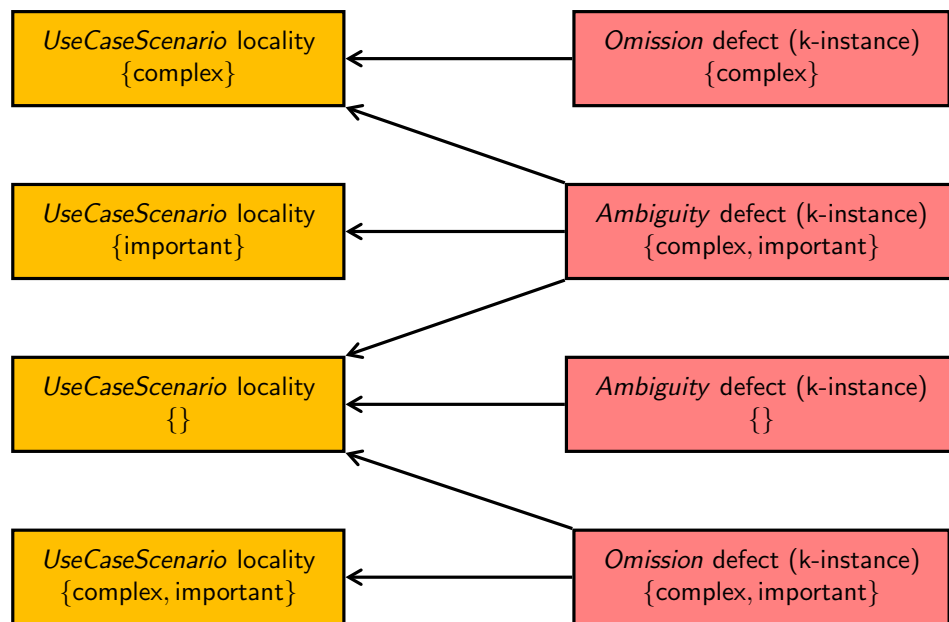
Figure 8.3: An example of marker assignment. Use case scenarios are directly assigned a combination of the markers "complex" and/or "important". Defects, being k-instances, inherit these markers from the relevant use case scenarios.

assigned the markers of their dependencies. Thus, if a class locality has an importance or complexity marker assigned to it, any defects occurring within that class (thereby being dependent on it) will also have the same marker.

Some locality types may have a greater propensity for certain properties than others. For instance, specific functional requirements are more likely to have critical importance or contain complex logic than the overview section of the requirements document.

Figure 8.3 shows an example of marker assignment to use cases scenarios and defects. Here, use case scenario localities are each explicitly assigned a set of markers, which are inherited by the related defect entities. That is, a defect inside a particular use case scenario is complex/important if (and only if) that use case scenario itself is complex/important.

8.1.4 Phase Structure

The framework views a software project as being divided into discrete time units of arbitrary and variable length, which for convenience are called phases. These may or may not map onto the conventional software lifecycle phases, and a given phase may or may not be qualitatively different from the previous phase. In a project undertaken using agile methods, iterations may be considered phases for the purpose of the framework.

Each phase contains a set of entities and dependencies, based on those that existed in the previous phase. Each phase includes a subset of the following activities:

- development (whether requirements gathering, design or coding);
- inspection/review;
- testing;
- operational use; and/or
- defect correction.

By allowing any combination of these activities to take place in any phase, the framework can apply to projects that use either waterfall-style or agile methods. In the latter case, iterations may be considered phases, with testing and operational use conducted virtually throughout the project.

8.1.5 Hierarchy and Propagation

The framework considers two situations in which entities and dependencies can arise.

Hierarchy occurs when one entity is subordinate to and cannot exist without another (within the same phase). For instance, a parameter is subordinate to and cannot exist without a method, and likewise for a method and its containing class. In this case, a tree structure arises in which parameters are children of methods and methods are children of classes. This does not imply any form of causal relationship; merely a structural one. Hierarchy does not necessarily reflect the process of designing or implementing classes and methods (or any other entity types), but simply the outcome of that process.

Propagation occurs when an entity in one phase leads to one or more entities in the next phase. For instance, a functional requirement may result in (i.e. propagate to) a number of different classes, and/or a number of different methods. (Propagation is restricted to one-to-many relationships to simplify the model.)

Hierarchy and propagation links may occur between entities of any type. The resulting structures are distinct from the dependency structure, but impose further constraints on where dependencies may occur. Table 8.2 summarises the types of connections occurring between entities — the three types of dependence along with hierarchy and propagation. Hierarchy and propagation show how a system is structured and how that structure evolves over time, whereas dependencies determine how it is understood.

In general, an entity can depend on any other entity within the hierarchy. However, in

Table 8.2: Summary of connections between entities.

Connection type	Parent[a]	Child[a]	Across phases[b]	Multiple parents[c]	Modelling purpose
Comprehension dependency	k-instance	k-instance	no	yes	inspection
Locality dependency	locality	k-instance	no	yes	inspection
Decision dependency	k-instance	locality	no	yes	inspection
Hierarchy link	entity	entity	no	no	system structure
Propagation link	entity	entity	yes	no	system structure

[a] Dependencies are restricted to particular classes of entity.
[b] Only propagation links may occur between entities in different phases.
[c] Hierarchy and propagation links form tree structures, whereas dependencies do not (necessarily).
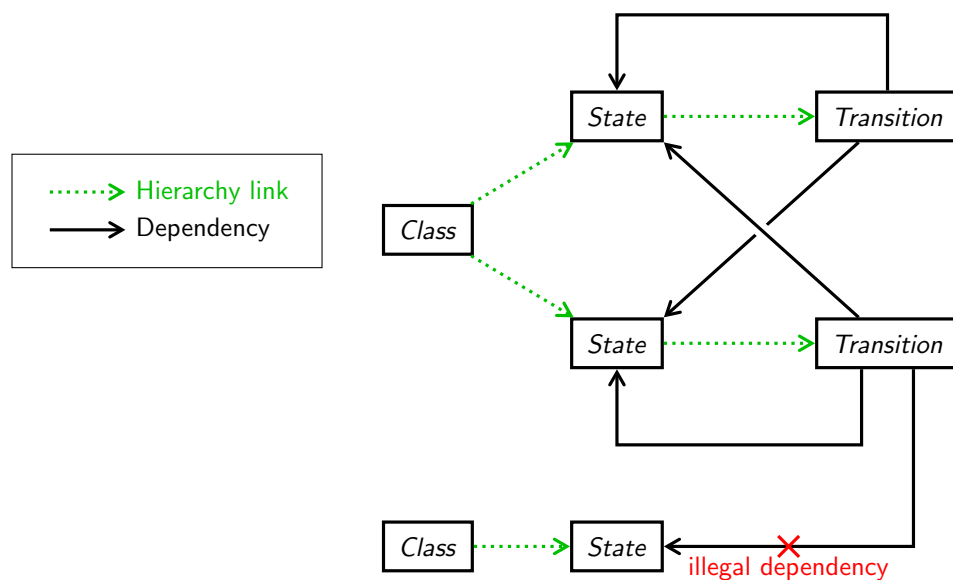
Figure 8.4: An example of an entity hierarchy. In this example, states are children of classes, and transitions are children of states. A transition can depend only on states within the same class.
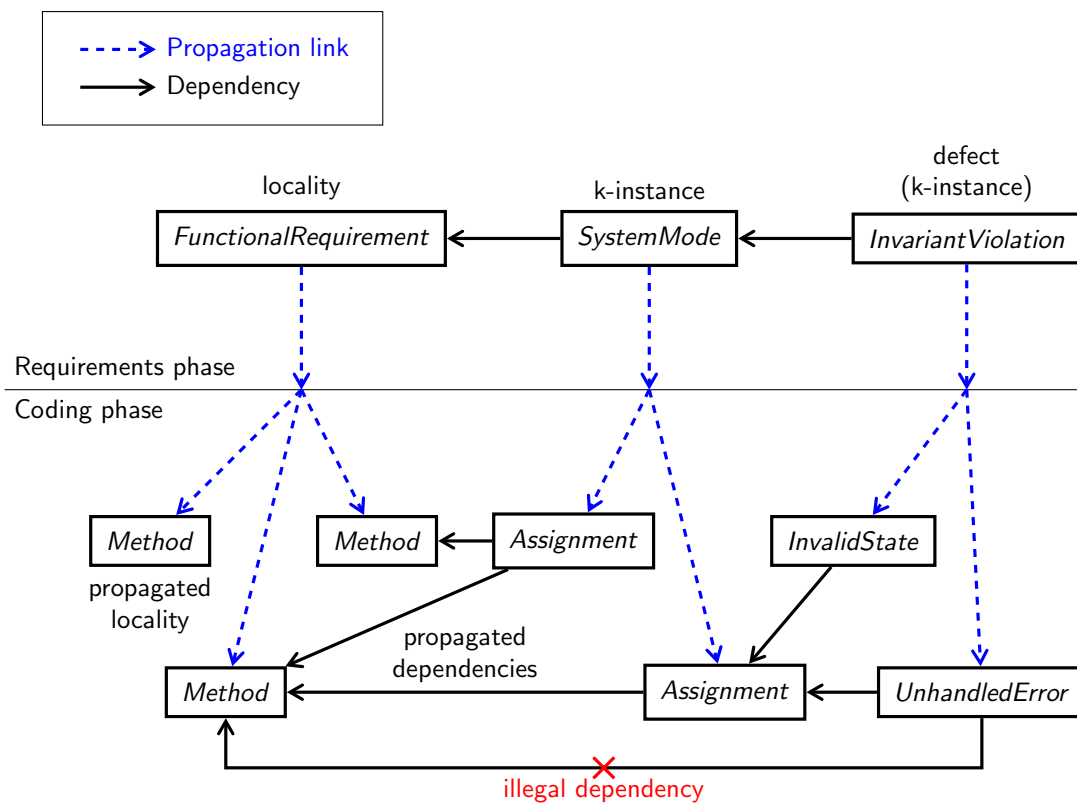
Figure 8.5: An example of entity and dependency propagation. A propagated dependency can only occur between two propagated entities if a corresponding dependency existed in the previous phase.

some situations, dependencies may be restricted to entities within the same subtree. In Figure 8.4, Transition entities have dependencies on State entities, but only on states within the same class (i.e. those descended from the same Class entity).

Figure 8.5 shows an example of propagation. For propagated entities, dependencies can only exist where corresponding dependencies existed in the previous phase. In the example, the UnhandledError defect cannot depend directly on a Method k-instance, because the InvariantViolation defect did not depend on a FunctionalRequirement locality. Even where this condition is satisfied, the framework does not mandate that a dependency must exist.

8.1.6 Inspection Strategies

The framework views reading techniques as being part of a larger *inspection strategy* that encompasses all inspectors across all phases of a software project.

Inspection strategies specify the number of inspectors to use in each phase, and what forms of guidance are provided. In each phase, for each inspector, for each entity, an inspection strategy may provide *active* and/or *passive guidance*. Active guidance entails an instruction asking the inspector to search the locality or find the k-instance in question. Passive guidance occurs when an alternate representation of the entity is provided in a form more easily digested by the inspector.

Passive guidance may imply the use of visualisation tools, or else some preliminary manual effort undertaken to provide inspectors with supporting material. Defects cannot have passive guidance, because that would require that they be known in advance.

8.2 Model

A fine-grained model of inspection cost effectiveness is proposed, based on the framework concepts discussed in the previous section.

Inspection cost effectiveness can be thought of as a comparison between the costs arising in two scenarios: where a particular inspection strategy is used, and where no inspections are used. (This also raises the prospect of *relative cost effectiveness*, which would involve a comparison between two or more different inspection strategies, but this is not explored here.)

Based on the formulae proposed by Kusumoto et al. (1991) and Freimut et al. (2005) (discussed in Chapter 2, Section 2.4.1), the following equation gives the overall notion of cost effectiveness:

$$\text{CE} = \frac{\text{Net costs avoided}}{\text{Cost without inspection}}$$

$$= \frac{\text{Cost without inspection} - \text{Cost with inspection}}{\text{Cost without inspection}}$$

$$= 1 - \frac{\text{Cost with inspection}}{\text{Cost without inspection}}$$

The model described here diverges from those of Kusumoto et al. (1991) and Freimut et al. (2005) in the introduction of a variable representing the inspection strategy (and subcomponents thereof, as described in Section 8.2.5). In a deterministic world, the total cost with/without inspection could be defined as a function of the inspection strategy z, as follows:

$$\text{CE}(z) = 1 - \frac{\text{TC}(z)}{\text{TC}(z_0)} \tag{8.1}$$

where $\text{CE}(z)$ is the cost effectiveness of inspection strategy z;

$\text{TC}(z)$ is the total cost (in units of currency, time or effort) of z; and

$\text{TC}(z_0)$ is the total cost of the *null* inspection strategy z_0, representing the absence of inspections.

In this scheme, z precisely determines the total cost, and hence the cost effectiveness. In reality, cost and cost effectiveness are also affected by random factors beyond the control of the inspection strategy. Thus, instead of representing total cost (TC) as a function of z, it can be represented as a random variable depending on Z. Cost effectiveness (CE) is then also a random variable, and can expressed using conditional probability notation as follows:

$$\mathbb{E}(\text{CE} \mid Z = z) = 1 - \frac{\mathbb{E}(\text{TC} \mid Z = z)}{\mathbb{E}(\text{TC} \mid Z = z_0)} \tag{8.2}$$

where $\mathbb{E}(\text{CE} \mid Z = z)$ is the expected cost effectiveness, given inspection strategy z;

$\mathbb{E}(\text{TC} \mid Z = z)$ is the expected total cost arising from inspection strategy z; and

$\mathbb{E}(\text{TC} \mid Z = z_0)$ is the expected total cost arising from the null inspection strategy z_0.

Assuming that costs are not negative, cost effectiveness theoretically lies in the range $(-\infty, 1]$. However, a value of 1 is unrealistic, because it would indicate that the inspection strategy completely eliminates all defect-related costs with no effort expended. Negative cost effectiveness is possible, and would mean that the inspection strategy generates greater costs than it saves. The cost effectiveness of the null strategy is zero, by definition.

Total cost is the sum of the various inspection-related costs resulting from:

- searching a locality (essentially the cost of performing the inspection itself);
- providing passive guidance;
- operational failures of the software;
- investigating failures to determine the underlying defect; and
- reworking a defect.

These cost terms are formalised as a set of random variables, which in turn depend on other variables, detailed further in sections 8.2.5 and 8.2.6. However, these variables exist in the context of a system of entities and dependencies, which must themselves be established beforehand. Thus, the model has two components:

- The *metamodel* describes the system of entities and dependencies, and their propagation across all project phases. For each phase, it also establishes the set of potential inspectors and the system's operational runtime.

- Each of two *scenarios* describe the consequences of an inspection strategy, ultimately on the various costs, within the context of the metamodel. Both inspection strategies are therefore evaluated using the same defects, k-instances, localities, inspectors and other facets of the software process.

  For convenience, the scenario model is further divided into two parts, one dealing with inspection itself and the second modelling the wider software verification process.

In principle, if the two inspection strategies both happen to result in the same defect being detected, the consequences following from this (including whether the defect is

reworked, and the number of other defects that result from it in the next phase) should be the same.

8.2.1 Metamodel Entities

Based on the entity classification scheme, the model introduces $\mathcal{E}$ — the set of all entity types. This has subsets $\mathcal{E}_\mathrm{L}$, $\mathcal{E}_\mathrm{K}$ and $\mathcal{E}_\mathrm{D}$, representing the set of all locality, k-instance and defect types. Entity types are considered to be atomic values.

An entity itself is defined mathematically as a tuple of four values, using the notation $\varepsilon^{(e,\eta,h,\varepsilon')}$. Without parentheses, ε may represent any arbitrary entity. The parenthesised values serve to identify the entity, as follows:

- e (also written $\tau(\varepsilon)$) is the entity type of ε;

- ε' (also $\pi(\varepsilon)$) is the originating or superordinate entity by which the existence of ε is implied (through hierarchy or propagation), or 0 if ε is an initial entity;

- $h \in \{0, \mathtt{G}, \mathtt{H}\}$ (also $H(\varepsilon)$) indicates that ε has been propagated from a previous phase ($\mathtt{G}$), is implied as part of the entity hierarchy ($\mathtt{H}$) or neither (0); and

- η is an arbitrary index.

These details do not encode all the real-world characteristics of an entity, but are sufficient to define entities as mathematical objects. Other entity characteristics are represented by a variety of random variables.

Each entity can imply the existence of some number of other entities of each type, both in the current phase (through hierarchy) and in the subsequent phase (through propagation). $Q_{\mathrm{G}j}(\varepsilon)$ is the set of entities propagated directly from entity ε to phase j. $Q_{\mathrm{H}j}(\varepsilon)$ is the set of entities directly subordinate to ε within phase j. These are each defined by discrete random variables, giving the number of entities of each type implied by an entity of type $\tau(\varepsilon)$. The set of initial entities (Q_0) is also defined by a discrete random variable for each entity type. All other entities in the system are implied (directly or indirectly) by those in Q_0, through recursive applications of Q_G and Q_H. This is defined formally in Appendix E, Section E.1.1, and the random variables defined in Table E.3.

The symbol ε is used frequently in the metamodel because there is no need to distinguish between k-instances (except defects) and localities. In the scenario model, defects, k-
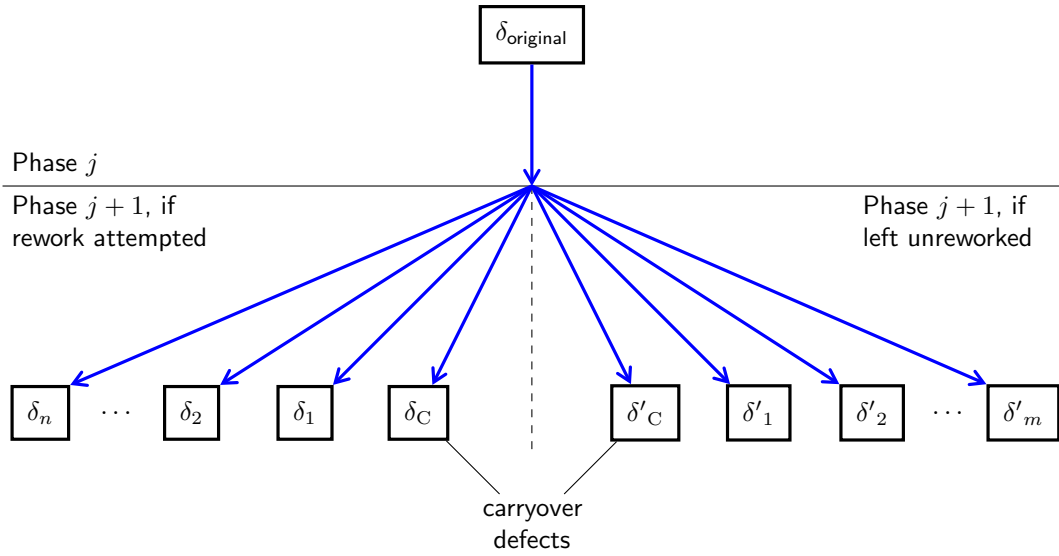
Figure 8.6: Defect-to-defect propagation. A defect in phase j results in two alternate sets of defects in phase $j+1$. Both sets form part of the defect pool.

instances more broadly and localities each have particular roles and accordingly are given separate symbols: δ for defects, κ for k-instances and λ for localities.

However, the metamodel makes a special concession to defect propagation. As a result, defects require further parameters. The construction $\delta^{(e,\eta,h,\varepsilon',w,y)}$ represents a defect δ with additional characteristics as follows:

- $w \in \{0,1\}$ (also $W(\delta)$) indicates whether δ is the result of an attempt to rework a defect in the previous phase; and

- $y \in \{0,1\}$ (also $Y(\delta)$) indicates whether δ has been carried over from the previous phase (i.e. rework was either not attempted, or not successful).

If the defect was not propagated (i.e. $h \neq \mathtt{G}$), both w and y are 0.

For each phase, the metamodel considers a pool of potential defects, only some of which exist in a given scenario. By contrast, all other entities are common to both scenarios. Two types of propagation are considered: non-defect-to-entity, and defect-to-defect. Non-defect entities can propagate to any type of entity, including defects. Defects themselves can only propagate to other defects. This restriction ensures that an entity common to both scenarios cannot arise from a defect specific to only one scenario.

Figure 8.6 briefly illustrates defect propagation. Propagated defects may result from either rework effort, or the absence of rework effort. At the same time, they may also be carryover defects from the previous phase, or new defects introduced as a result of

the original. A carryover defect is considered a distinct object from the original defect for modelling purposes, even though it represents the same defect.

For each defect in the preceding phase, the defect pool in the current phase consists of:

- exactly one carryover defect (i.e. the original defect itself) resulting from not attempting rework;
- zero or one carryover defects resulting from a failed rework attempt; and
- any number of other defects (not the original), resulting from either reworking or not reworking the original.

Each scenario considers only some of these propagated defects, depending on which were reworked in the previous phase. However, defects propagated from non-defect entities, defects derived hierarchically and initial defects are present in all scenarios.

The metamodel (rather than each scenario model) determines whether a hypothetical rework attempt will succeed. The inspection strategy is assumed to have no effect on this. Non-defect entities may also be carried over from one phase to the next, though the model has no need to distinguish between these and new, introduced entities.

8.2.2   Metamodel Dependencies

Having established (probabilistically) the set of entities in a given phase, the metamodel then specifies what dependencies may exist between them. To avoid introducing dependency cycles, each entity is assigned a *dependence complexity* number. Dependence complexity has no scale, but broadly represents the potential for an entity to depend on other entities. Each entity may only depend on other entities with lower dependence complexity. Such values are given by the continuous random variable Θ_e, each entity type e having a different distribution.

Just as entities themselves occur in three ways — initially, hierarchically or via propagation — so do dependencies. (Dependency construction is formally defined in Appendix E, Section E.1.2.)

Initial dependencies occur between initial entities; each entity depending on a random number of other entities of each type. $P_{00}(\varepsilon)$ is the set of initial dependencies. Here, a matrix of random variables specifies the number of dependencies each entity type will have on each other entity type. The actual dependencies are also chosen at random, with uniform probability.

Dependencies can also occur within branches of the entity hierarchy. A context entity defines the branch of the hierarchy in which such a dependency occurs. Thus, the array of random variables here has three dimensions: the dependent entity type, the target entity type, and the context entity type. The dependent and target entities must occur within a single branch defined by the context entity. However, there may be several such context entities, each with its own enclosed set of dependencies. $P_{\mathrm{H}j}(\varepsilon)$ gives the total set of hierarchical dependencies for ε.

Finally, dependencies can also be propagated. A propagated dependency may only occur (with some non-zero probability) between two entities if a corresponding dependency existed between the two original entities in the previous phase. The model uses a matrix of binary random variables, each indicating whether or not a propagated dependency exists between one entity type and another, where applicable. $P_{\mathrm{G}j}(\varepsilon)$ gives the total set of propagated dependencies for ε.

The set of all entities upon which ε depends is $P_j(\varepsilon)$, taken from $P_{00}(\varepsilon)$, $P_{\mathrm{H}j}(\varepsilon)$ and/or $P_{\mathrm{G}j}(\varepsilon)$ as appropriate. This is disaggregated into subsets representing the three types of dependence:

- $P_{\mathrm{M}j}(\kappa)$ represents the comprehension dependencies of k-instance κ;
- $P_{\mathrm{L}j}(\kappa)$ represents the locality dependencies of k-instance κ; and
- $P_{\mathrm{D}j}(\lambda)$ represents the decision dependencies of locality λ.

8.2.3 Metamodel Markers

Markers are represented by the symbol ψ, and the set of all markers by Ψ. The assignment of a marker ψ to entity ε in phase j is represented by the binary variable $X_{\mathrm{K}j\varepsilon\psi}$. The means by which markers are assigned depends on the entity.

The logic of marker assignment has several cases, illustrated in Figure 8.7. Markers are assigned probabilistically to initial or hierarchically-occurring localities, based on a binary random variable for each combination of locality type and marker. By contrast, propagated localities have the same markers as their counterparts in the previous phase. A k-instance is assigned a marker if the same marker is assigned to any of its dependencies, or to the entity from which it propagated (if applicable). (The formal definition of marker assignment occurs in Appendix E, Section E.1.3).

Almost all of the variables in the scenario model depend on $X_{\mathrm{K}j\varepsilon\psi}$. Thus, markers have the capacity to influence almost any aspect of the scenario.
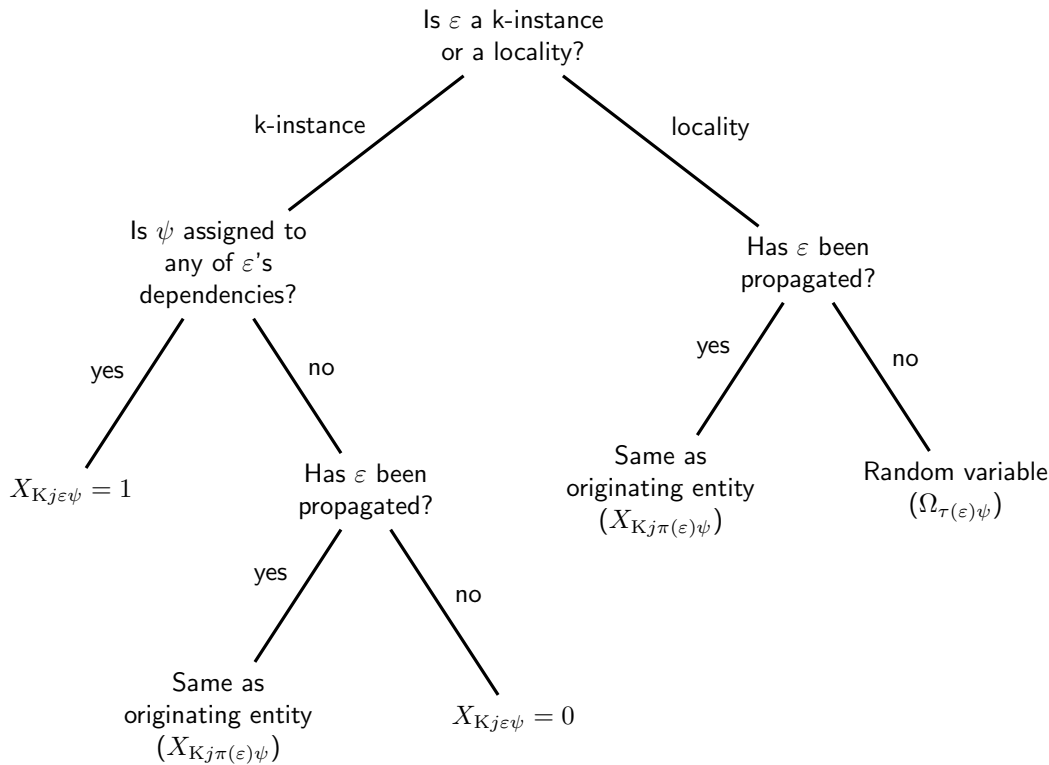
Figure 8.7: Decision tree for determining whether marker ψ is assigned to entity ε (i.e. the value of $X_{\mathrm{K}j\varepsilon\psi}$).

8.2.4   Compact Bayesian Network Notation

The metamodel and scenario models differ in the notations needed to describe them. The metamodel concerns sets of entities, whereas the scenario model concerns events and activities associated with those entities. The latter is more amenable to being described by a Bayesian Network (BN), as explained in Chapter 3, Section 3.3.3.

However, the actual number of random variables in the scenario is determined probabilistically by the metamodel, and some of their relationships are similarly random. This arrangement is not easily represented in a conventional BN, with one node for each variable. However, it can be represented in a compact form, the notation for which is described here.

Compact Bayesian Network (CBN) notation is a means of generalising a set of BNs. Each CBN represents multiple possible BNs in a compact form. In principle, with additional information, any given CBN can be redrawn as an ordinary BN (though in practice the resulting network might be unreadable). CBN notation is similar in principle to *plate* notation.

In a CBN, variables with similar meaning and behaviour are grouped into arrays called *variable sets*. Each node in the graph represents a variable set. Each variable set is indexed by zero or more subscripts, different variable sets potentially having different sets of indices. Member variables are identified by their subscript values.

A variable set is distinct from a vector- or matrix-valued variable, because each of its member variables may have different dependency relationships. However, the existence of these relationships must follow a definable rule.

An arc between two variable sets represents a set of statistical dependencies between the member variables. The individual relationships thus implied are as follows:

1. If two connected variable sets have no common subscripts, then statistical dependencies occur between all pairings of their member variables (i.e. the Cartesian product).

2. If two connected variable sets share one or more common subscripts, statistical dependencies only occur between pairings where the common subscripts match.

3. Notwithstanding the above, two member variables may need to satisfy additional arbitrary predicates before one can depend on the other. These predicates are shown explicitly along the arc.

Compact Bayesian networks can contain cycles, as long as there are predicates in place to prevent cycles among the member variables.

The following sections demonstrate the use of CBN notation.

8.2.5 Comprehension Modelling

The scenario model considers the inspection process to consist of a series of searching and comprehension events. For a given phase j and a given inspector i, each of the localities may or may not be searched, and each of the k-instances may or may not be comprehended. This is expressed by binary random variable sets $S_{ji\lambda}$ and $M_{ji\kappa}$. Comprehension, locality and decision dependencies manifest as relationships between these variables.

Searching and comprehension are affected by guidance, and by an *active guidance level*, which is defined as the overall number of k-instances or localities for which active

guidance is given (for an individual inspector). Both forms of guidance are presumed to have a positive effect on the log odds of searching and comprehension. Active guidance level is presumed to have a small negative effect, but one that can mount up as the level increases. The intention here is twofold: to model inspectors shifting their attention away from those entities not covered by active guidance, and to capture overhead associated with following instructions rather than intuition. Thus, it is not a foregone conclusion that more active guidance yields lower costs. Meanwhile, passive guidance has a more direct counterbalancing mechanism. The use of passive guidance for a given entity (for at least one inspector) incurs a cost associated with the necessary preparation effort.

Dependencies also play a role in determining searching and comprehension log odds. The log odds of searching a locality are increased if any decision dependency k-instances have been comprehended. By contrast, the log odds of comprehending a k-instance are reduced if any comprehension and locality dependencies have not been found. Such inverse effects are used in the latter case because comprehension and locality dependencies reflect requirements that should be met, rather than opportunities that may arise.

The log odds of searching are also affected by markers, with each marker having a different (either positive or negative) effect on the log odds. Searching a particular locality is a deliberate action of the inspector, potentially motivated by certain properties of that locality, represented by markers. Comprehension is not similarly affected because it is essentially involuntary, or at least not amenable to such precise decision making.

Formally, $S_{ji\lambda}$ and $M_{ji\kappa}$ are defined using logistic model equations, shown in Appendix E, sections E.2.3 and E.2.4. (Logistic model equations in general are discussed in Chapter 3, Section 3.3.1.) These begin with a term for the baseline log odds of searching or comprehension, and contain additional effect coefficients for active guidance, passive guidance, active guidance level, dependencies and markers as described above. The baseline log odds and all of the effect coefficients are selected from random variables. Their values are fixed for a given inspector and vary between inspectors. This helps to model inspector variability.

A compact Bayesian network for comprehension is shown in Figure 8.8. The guidance variables influencing $S_{ji\lambda}$ and $M_{ji\kappa}$ are ultimately determined by the inspection strategy. The inspection strategy itself (Z) formally comprises three components:

- the number of inspectors to be used in each phase (the vector $\mathbf{Z}_I$, or Z_{Ij} for a single phase j);
- the provision of comprehension guidance (the function Z_M); and
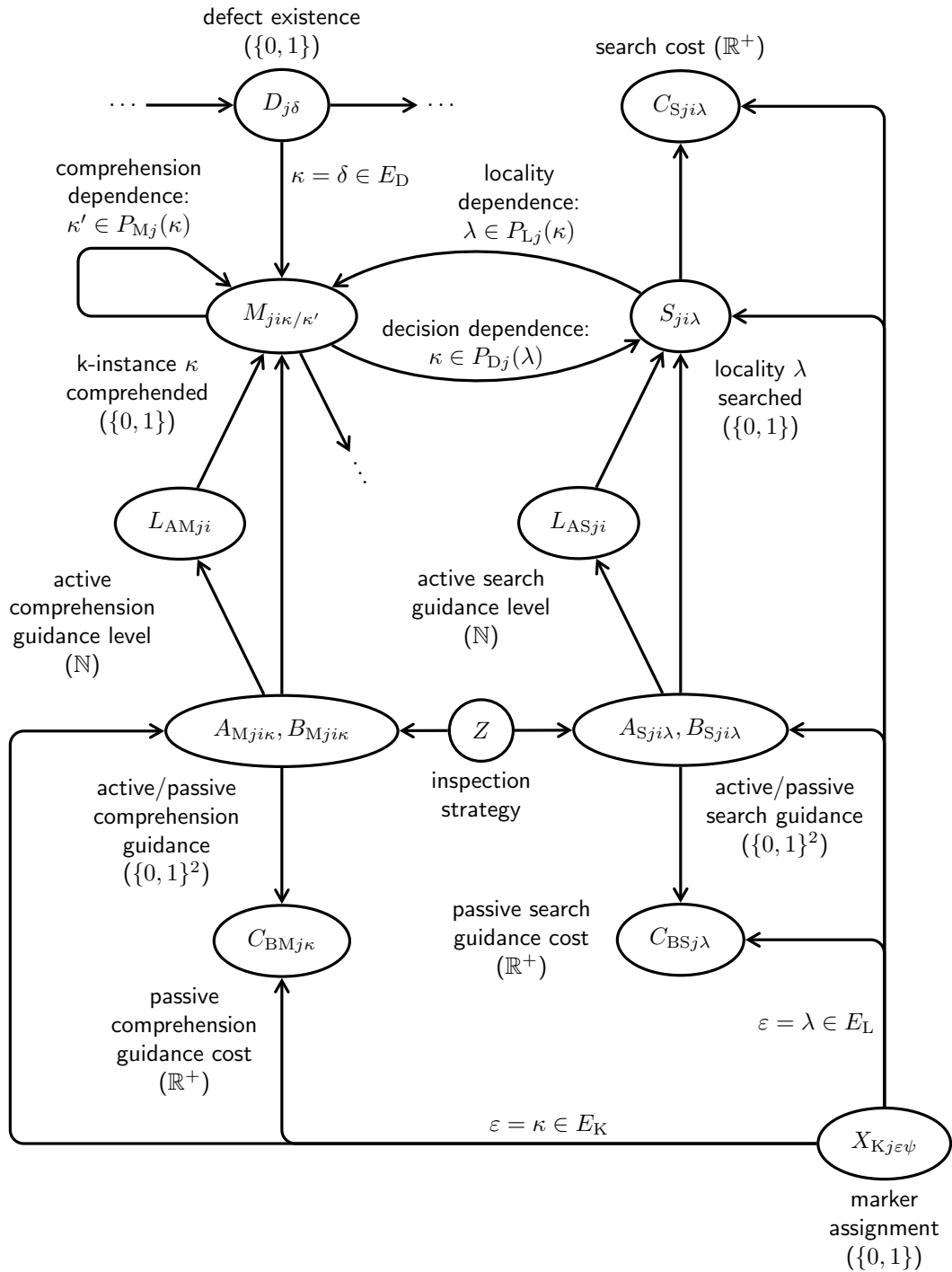
Figure 8.8: The inspection process, represented as a compact Bayesian network. Each variable set is annotated with a brief description and the set of possible values. Ellipses indicate where this diagram overlaps with and joins onto Figure 8.9. The $D_{j\delta}$ and $M_{ji\kappa}$ variable sets form part of the overlap.

- the provision of search guidance (function Z_S).

Thus, $Z = (\mathbf{Z}_I, Z_M, Z_S)$. The functions Z_M and Z_S map phase j, inspector i, entity type $\tau(\varepsilon)$ and the set of applicable markers represented as binary vector $\mathbf{X}_{Kj\varepsilon}$ to indicators of active and passive guidance:

$$(A_{Mji\kappa}, B_{Mji\kappa}) = Z_M\left[j, i, \tau(\kappa), \mathbf{X}_{Kj\kappa}\right] \tag{8.3a}$$

$$(A_{Sji\lambda}, B_{Sji\lambda}) = Z_S\left[j, i, \tau(\lambda), \mathbf{X}_{Kj\lambda}\right] \tag{8.3b}$$

Thus, inspection strategy and marker assignment (X_K) jointly determine active/passive comprehension guidance (A_M and B_M) and active/passive search guidance (A_S and B_S). In turn, the active guidance variables directly determine the active guidance levels (L_{AM} and L_{BM}). The Z, A, B and L variables are not random but deterministic.

The passive guidance variables affect the passive guidance provision costs (C_{BM} and C_{BS}). These costs along with the cost of searching (C_S) are determined by log-linear equations, where the baseline/intercept term is a random variable. These are shown in Appendix E, sections E.2.10, E.2.11 and E.2.12. Log-linear (rather than linear) equations are used here because the factors affecting costs — the effects of any markers assigned to the entity in question — are more likely to be multiplicative than additive; i.e. a percentage increase/decrease rather than an absolute increase/decrease.

8.2.6 Verification Process Modelling

The wider verification process concerns the detection and rework of defects, leaving aside non-defect entities. Three activities/events may lead to the discovery of a defect: inspection, test failure and operational failure. All three means of detection lead to the possibility of reworking the defect. For test and operational failure, the failure must be investigated before any corrective action can be taken. Whether the defect is reworked determines its propagation (or lack thereof) to the next phase, where there are further opportunities for inspection, test failure and operational failure, and so on. Figure 8.9 shows a compact Bayesian network representation of the verification process, as it exists in the model.

The detection of defect δ through inspection is represented by binary random variable $M_{ji\delta}$, as described in the previous section. Thus, detecting a defect is a specific case of comprehending a k-instance. The same defect may be detected by any number of the inspectors involved in phase j; having multiple inspectors detect a defect makes no
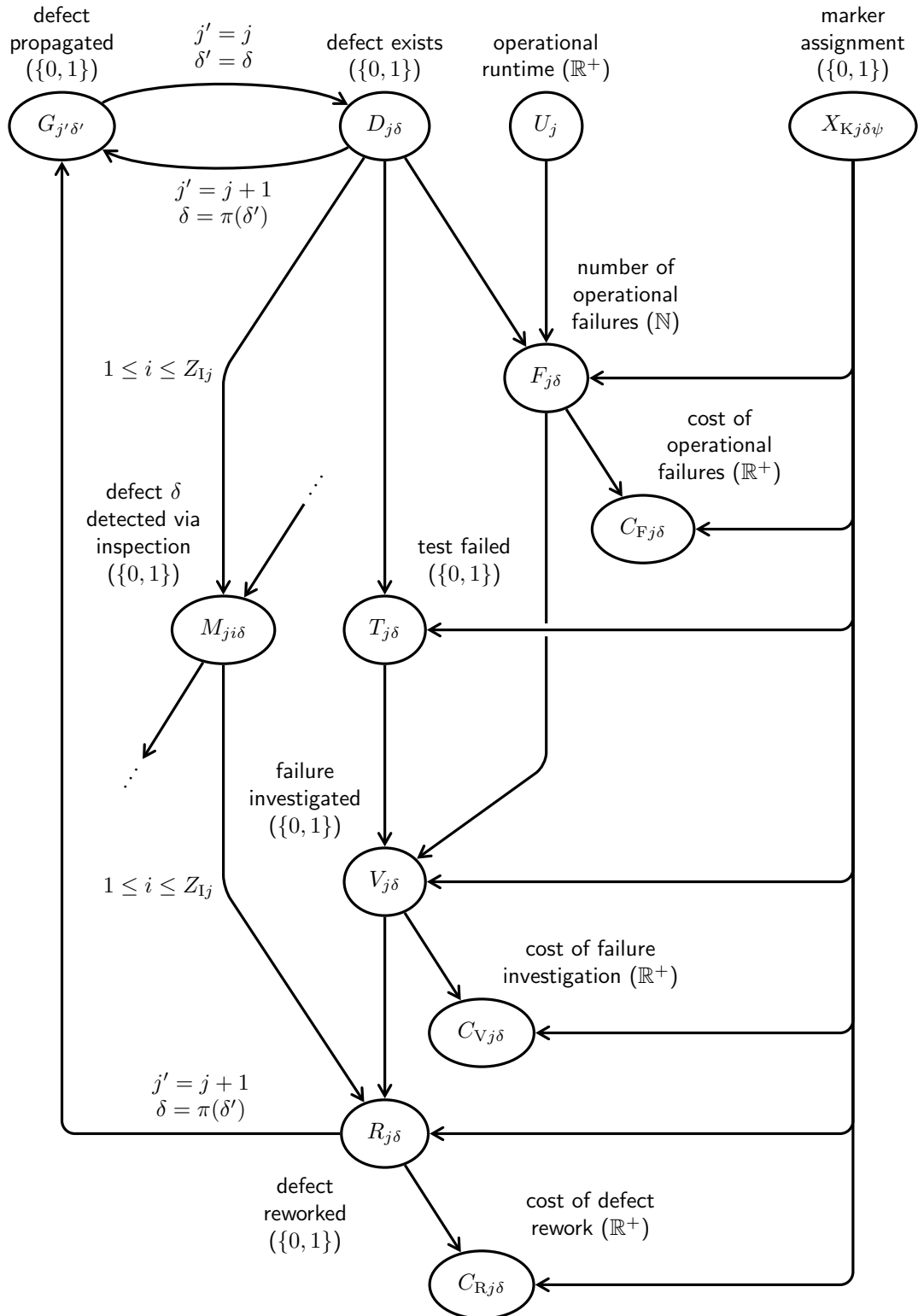
Figure 8.9: The software verification process, represented as a compact Bayesian network. Each variable set is annotated with its function and range of values. Ellipses indicate where this diagram overlaps with and joins onto Figure 8.8. The $D_{j\delta}$ and $M_{ji\kappa}$ variable sets form part of the overlap.

difference to the log odds of rework. In some phases (and in the scenario for the null strategy), there may be no inspectors at all, in which case there are no $M_{ji\delta}$ variables present.

Defects may also be detected through failures of the system, either in controlled conditions or in operational use. Each operational failure incurs a direct cost, whereas test failures do not. The model assumes that the same testing cost would be incurred irrespective of the defects in the system, making testing cost irrelevant in a comparison of inspection strategies. Hence, the model merely considers whether a test failure occurs ($T_{j\delta} \in \{0, 1\}$), while considering the number of operational failures ($F_{j\delta} \in \mathbb{N}$). The cost of operational failures for defect δ is $C_{\mathrm{F}j\delta}$.

Further, if either $T_{j\delta}$ or $F_{j\delta}$ is non-zero, an investigation into the failure(s) may proceed. $V_{j\delta}$ records whether an investigation takes place. If so, an investigation cost ($C_{\mathrm{V}j\delta}$) is incurred. The model assumes that any investigation effort successfully locates the defect.

Operational failures can only occur on carry-over defects. The model assumes that the version of the system being put into operational use does include new defects introduced in the current development phase, but only those present in the previous phase. That is, the sequence of events unfolds as follows:

- defects found in the previous phase are reworked (or rather rework is attempted);
- the system is released for operational use; and then
- further development is conducted in the current phase, which may lead to defects not present in the released version.

Further, test failures, operational failures and resulting investigations are not considered in the initial phase. Ultimately, the purpose of the model is to assess the effects of different inspection strategies, and at this point the inspection strategies have not yet had any effect.

Finally, the binary variable $R_{j\delta}$ records whether an attempt is made to rework defect δ. A rework attempt can only be made if the defect has been detected, either through inspection or failure/investigation. The success of any rework attempt is determined by the metamodel. Regardless of its success, a rework attempt incurs a rework cost ($C_{\mathrm{R}j\delta}$).

The variables discussed above all depend on defect δ actually existing within the scenario in question. This is indicated by the binary variable $D_{j\delta}$. $D_{j\delta} = 1$ in two cases:

- defect δ is an initial defect, or arose in the current phase from the entity hierarchy (i.e. $H(\delta) \in \{0, \mathtt{H}\}$); or

- defect δ propagated from one ($\pi(\delta)$) that also existed in the current scenario (i.e. $D_{(j-1)\pi(\delta)} = 1$), and either:
    - a rework attempt was made on $\pi(\delta)$ and δ is the result of rework, or
    - $\pi(\delta)$ was not reworked and δ is not the result of rework.

  (This second condition is represented by the binary variable $G_{j\delta}$.)

If $D_{j\delta} = 0$, then by definition $M_{ji\delta} = T_{j\delta} = F_{j\delta} = V_{j\delta} = R_{j\delta} = 0$, and also $D_{(j+1)\delta'} = 0$ (for any defect δ' propagated from δ).

Test failure ($T_{j\delta}$), investigation ($V_{j\delta}$) and rework ($R_{j\delta}$) are conditionally described by logistic equations (shown in Appendix E, sections E.2.7, E.2.8 and E.2.9). Logistic equations are used because these variables are binary. By contrast, the number of operational failures ($F_{j\delta}$) along with the costs of operational failure ($C_{\mathrm{F}j\delta}$), investigation ($C_{\mathrm{V}j\delta}$) and rework ($C_{\mathrm{R}j\delta}$) are described by log-linear equations (shown in Appendix E, sections E.2.6, E.2.13, E.2.14 and E.2.15).

To calculate the total cost for a scenario, the various cost terms are summed across all phases, all inspectors and all applicable entities:

$$
\mathrm{TC} = \sum_{j=0}^{J-1} \left[\sum_{\delta \in E_{\mathrm{D}}} \left(C_{\mathrm{F}j\delta} + C_{\mathrm{V}j\delta} + C_{\mathrm{R}j\delta} \right) + \right.
$$
$$
\left. \sum_{i=0}^{Z_{\mathrm{I}j}-1} \left(\sum_{\kappa \in E_{\mathrm{K}}} C_{\mathrm{BM}ji\kappa} + \sum_{\lambda \in E_{\mathrm{L}}} \left(C_{\mathrm{BS}ji\lambda} + C_{\mathrm{S}ji\lambda} \right) \right) \right] \quad (8.4)
$$

This would allow a cost effectiveness value to be calculated as per Equation 8.1. However, Equation 8.2 requires a value for $\mathbb{E}(\mathrm{TC} \mid Z)$, based on the probability distribution of TC given Z.

8.3 Simulation

The model was implemented in software as a Monte Carlo simulator (Rubinstein, 1981). That is, random samples are taken for each random variable, and the resulting cost effectiveness is calculated. This is repeated many times to find an approximation for the cost effectiveness.

As implied by Equation 8.2, cost effectiveness requires the total inspection-related costs to be calculated twice; once for the strategy in question and once for the null strategy. However, it would make little sense for the entire set of variables to be sampled twice, because the metamodel variables are unaffected by the inspection strategy. Conceptually, the two strategies must be applied to the same system in the same circumstances, or else the resulting value of cost effectiveness would be meaningless. A valid approximation could still be achieved, but would require many more simulation runs.

The same principle applies when multiple inspection strategies are to be compared in terms of cost effectiveness. To this end, each run of the simulation produces one metamodel and multiple scenario models, one for each of several inspection strategies to be compared (including the null strategy). Multiple cost effectiveness values are computed, one for each non-null strategy in each run.

8.3.1   Analytical Intractability

In principle, the cost effectiveness distribution for a given inspection strategy could instead be derived analytically from Bayesian methods. However, in doing so, every combination of values for the various random variables must be considered. There is little scope for simplifying the calculation without approximation, and the number of random variables is more than sufficient to render this approach intractable.

To illustrate the problem informally, consider a single hypothetical development phase in which there are 20 defects, 30 other k-instances, 10 localities and 3 inspectors. The model contains binary random variables representing the comprehension of each k-instance by each inspector, and the searching of each locality by each inspector. There are 50×3 comprehension variables alone, and 10×3 searching variables. This produces 2^{150+30} combinations of values, with each combination contributing to a different outcome. Taking into account several other types of binary variables — test failure, investigation and rework (one of each for each defect) — the number of combinations expands to $2^{150+30+20+20+20}$. Further, consider that the number of entities in each phase is not actually fixed at all, but rather is itself random. Each possible entity tally yields an entirely new set of comprehension, search, test failure, investigation and rework variables. Finally, consider that this merely represents a subset of variables in a single phase, while a comparable (or even larger) number of variables may exist in other phases.

There is little scope for considering variables in isolation (i.e. taking common factors out of the joint probability distribution) because most variables share common de-

Table 8.3: Inspection strategies used for model evaluation.

Inspection strategy	Inspectors	Description	Guidance (type and scope)
Null	0	No inspections	n/a
Ad hoc	1	No reading technique or cognitive support	None
Checklist	1	Checklist-Based Reading (CBR)	Active, for defects only
Scenario	1	Scenario-Based Reading (SBR)	Active, for all entities
Mixed	1	SBR in the initial phase, CBR in subsequent phases	Active, for all entities in the initial phase, but for defects only in subsequent phases
Passive-guided	1	Cognitive support with no reading technique	Passive, for all entities except defects
Maximal	1	Cognitive support mixed with SBR	Active for defects, passive for other k-instances, both active and passive for localities
Focused	1	Prioritisation-based reading technique	Active, for defects and localities marked "important" only
Dual *ad hoc*	2	Two-inspector *ad hoc*	None
Dual passive-guided	2	Two-inspector passive-guided	Passive, for all entities except defects
Divided responsibilities	2	Two-inspector CBR, but with separation of concerns based on defect type	Active, for defects only; different defect types for each inspector

pendencies. In particular, many variables depend on defect existence (D) and marker assignment (X_{K}).

A Monte Carlo simulator was developed in response to the intractability of an analytic solution.

8.3.2 Evaluation Methodology

Evaluation of the simulation (and hence the model), involves comparing inspection strategies and performing a sensitivity analysis. The inspection strategies chosen are listed in Table 8.3, along with their definition within the model. The goal is to examine the model's behaviour and assess the plausibility of the results.

The simulation was run using synthetic data. Model inputs — consisting principally of parameters for the various probability distributions — were conceived artificially, so as to represent a small spectrum of plausible software projects. The results necessarily reflect the choices made in the construction of these inputs. Thus, caution must be exercised when interpreting simulation output.
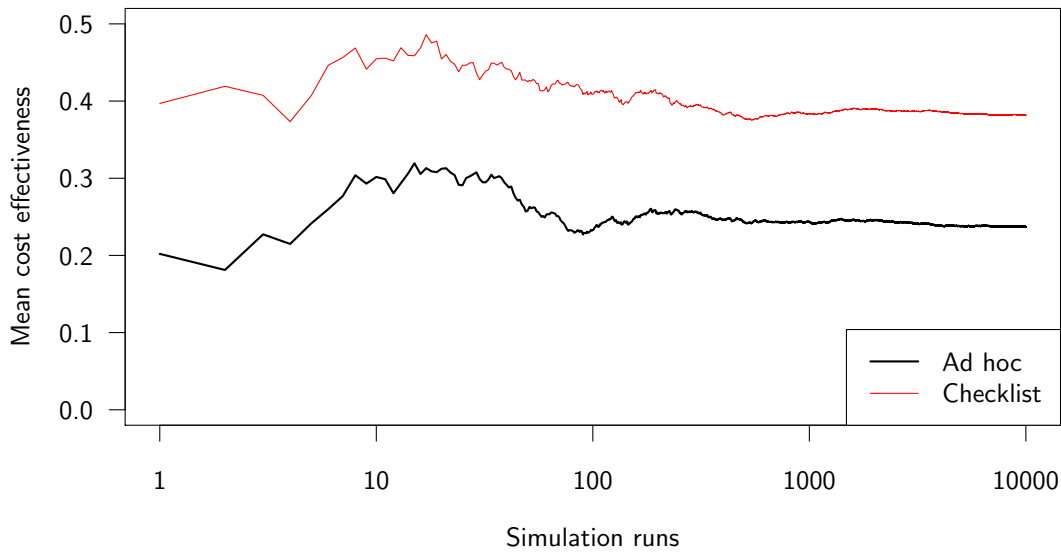
Figure 8.10: Convergence of simulated cost effectiveness, for the *ad hoc* and checklist strategies. The number of simulation runs is represented on a logarithmic scale.

Several datasets were generated to explore different aspects of the model's behaviour. One main dataset was used to compare inspection strategies and examine the probability distribution of cost effectiveness. The main dataset was derived from ten thousand simulation runs. Figure 8.10 shows how simulated cost effectiveness converges as the number of simulation runs increases. Other specialised datasets, each based on one thousand simulation runs, were used to perform sensitivity analysis; to examine the effects of selected model parameters, and the interaction of these parameters with the inspection strategy.

Analysis of simulation outputs was conducted using the R statistical package (R Development Core Team, 2009).

8.3.3   Cost Effectiveness Distribution

Figures 8.11 and 8.12 show probability density functions for *ad hoc* and checklist cost effectiveness. Figure 8.11 presents the marginal distributions, while Figure 8.12 plots the joint distribution. The latter is made possible by the simulator's simultaneous evaluation of different inspection strategies in the same context. The axes represent cost effectiveness achieved under two alternate, parallel scenarios, while the shading and contour lines indicate the height of the probability density curve.

The marginal probably density functions appear almost triangular, with sharp peaks.
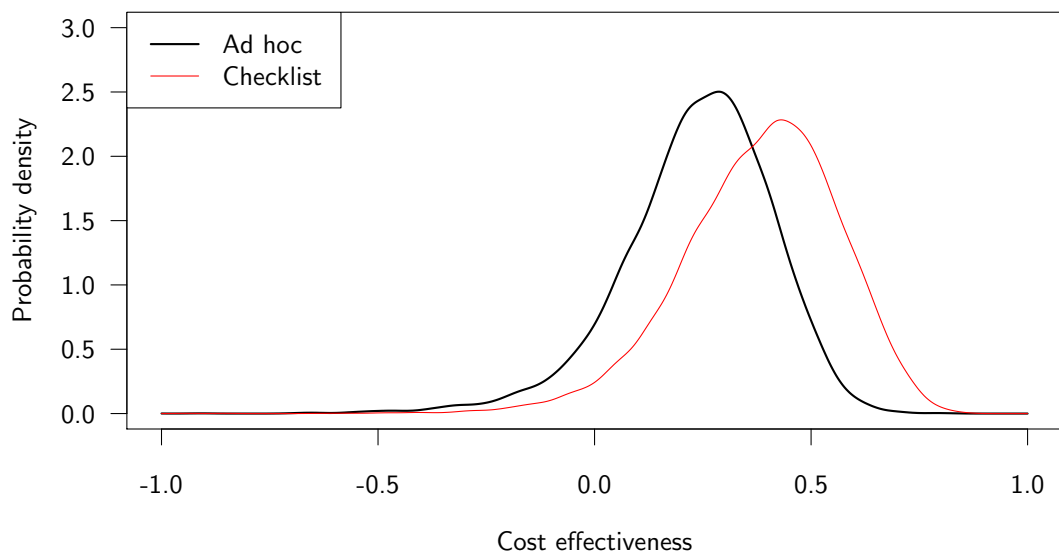
Figure 8.11: Distribution of *ad hoc* and checklist cost effectiveness (based on 10,000 simulation runs). The curves were derived from kernel density estimation using the R `density` function (where the bandwidth was calculated using the `dpik` function).
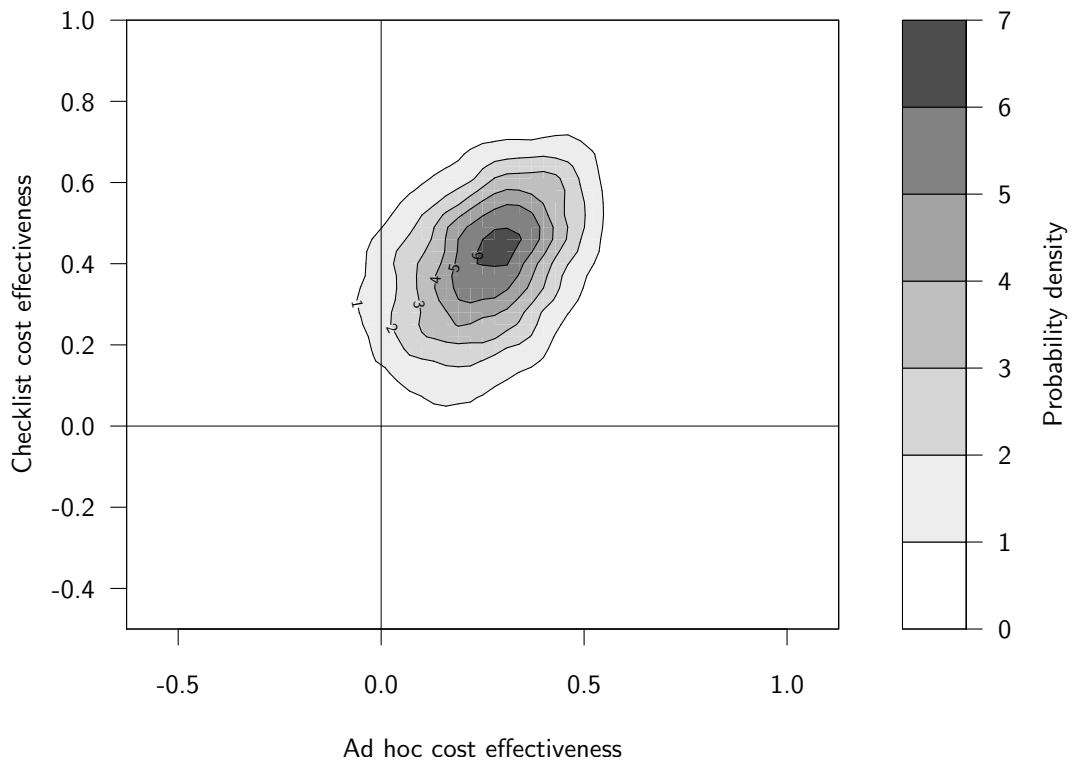
Figure 8.12: Joint probability distribution for *ad hoc* and checklist cost effectiveness (based on 10,000 simulation runs). This was derived from kernel density estimation using the R `bkde2D` function (where the bandwidth was calculated using the `dpik` function).
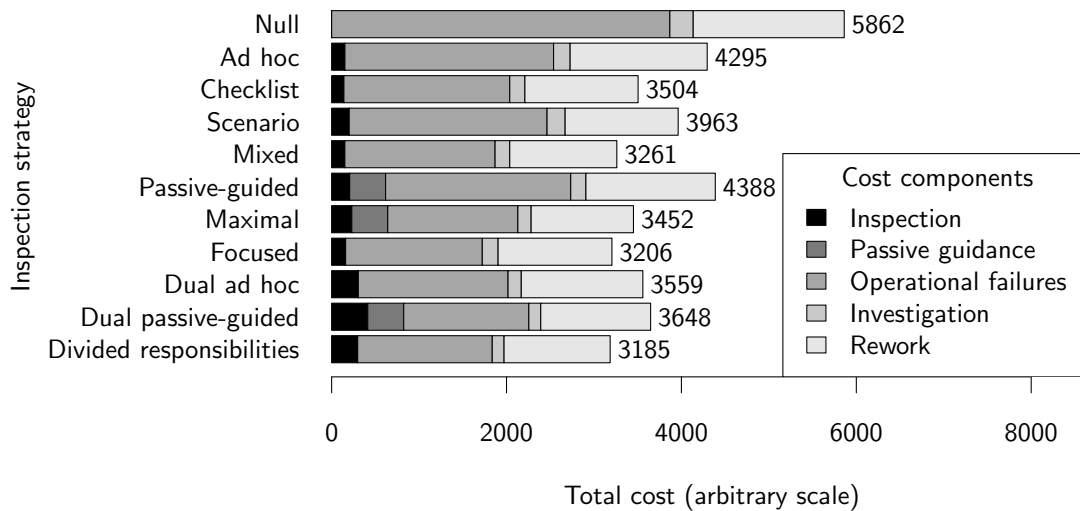
Figure 8.13: Simulated cost breakdown for different inspection strategies (based on 10,000 simulation runs).

The checklist distribution has a higher mean and somewhat higher variance than *ad hoc*. A relatively high overlap exists between the two distributions, reflecting the uncertainty in the model parameters.

The contour graph's elongation along the diagonal shows that *ad hoc* and checklist cost effectiveness are correlated, but not perfectly. This is expected; each pair of cost effectiveness values is derived from the same metamodel (i.e. the same software system) but a different scenario.

8.3.4 Inspection Strategy Performance

The inspection strategies can be compared statistically and graphically.

First, a repeated-measures ANOVA reveals that costs among different strategies do vary ($p = 2.2 \times 10^{-16}$). Tukey's Honest Significant Difference (R Development Core Team, 2009) is then used to compare costs for each pair of strategies. There are 55 pairwise comparisons in total. All but one of these were statistically significant, each with $p < 0.001$. No significant difference was found between the focused and divided responsibilities strategies, where $p = 0.817$.

Figures 8.13, 8.14 and 8.15 compare the various inspection strategies graphically using three different measures of performance.
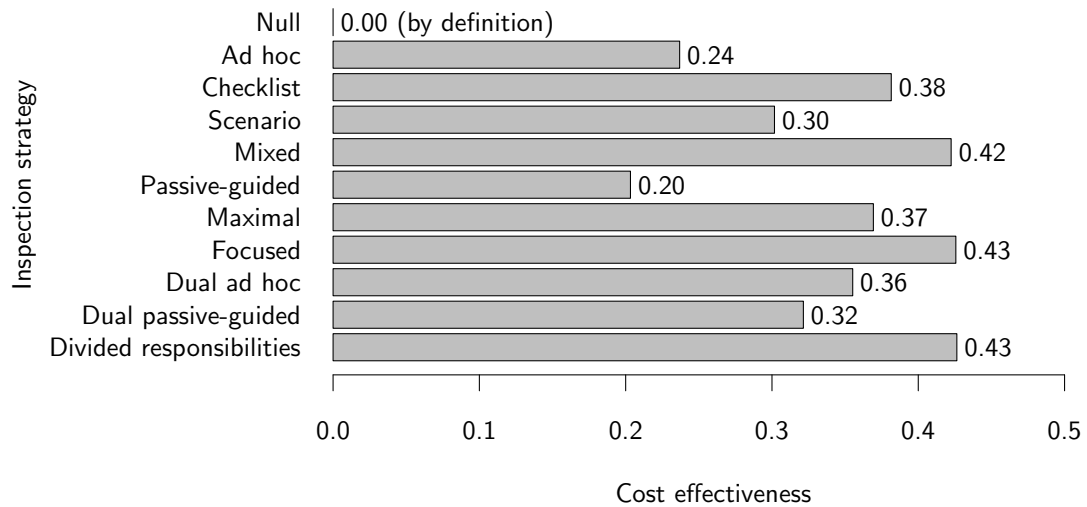
Figure 8.14: Simulated cost effectiveness for different inspection strategies (based on 10,000 simulation runs).

Figure 8.13 shows the total costs for the different strategies, broken down by source. The two major cost sources are operational failure and defect rework. All strategies show marked improvement over the absence of inspections, owing principally to a drop in operational failure cost, though investigation and rework costs are also reduced.

Figure 8.14 presents the strategies' overall cost effectiveness. This is shown to demonstrate the cost effectiveness metric, and is simply a normalised, inverted form of the cost data in Figure 8.13. By definition, those strategies with the lowest costs in Figure 8.13 have the highest cost effectiveness in Figure 8.14, and vice versa. Also by definition, the null strategy has zero cost effectiveness. In this case, all other strategies show positive cost effectiveness, meaning improvement over the null strategy. Unlike raw cost, cost effectiveness cannot be broken down into components.

Figure 8.15 shows the proportion of defects detected using each inspection strategy, in each phase. This hides the flow-on effects of defect detection from one phase to the next, thus allowing comparisons between strategies for specific phases.

Scenarios are shown to have lower overall cost effectiveness than checklists. Both checklists and scenarios have relatively high detection rates in the initial phase. However, in later phases, scenarios suffer from the worst performance of any nontrivial strategy. Checklists themselves also perform no better, or even worse than *ad hoc* in the latter phases.

The passive-guided strategy fails even to outperform *ad hoc*, in both the one- and two-inspector cases. In both cases, an improvement in operational failure cost is balanced by
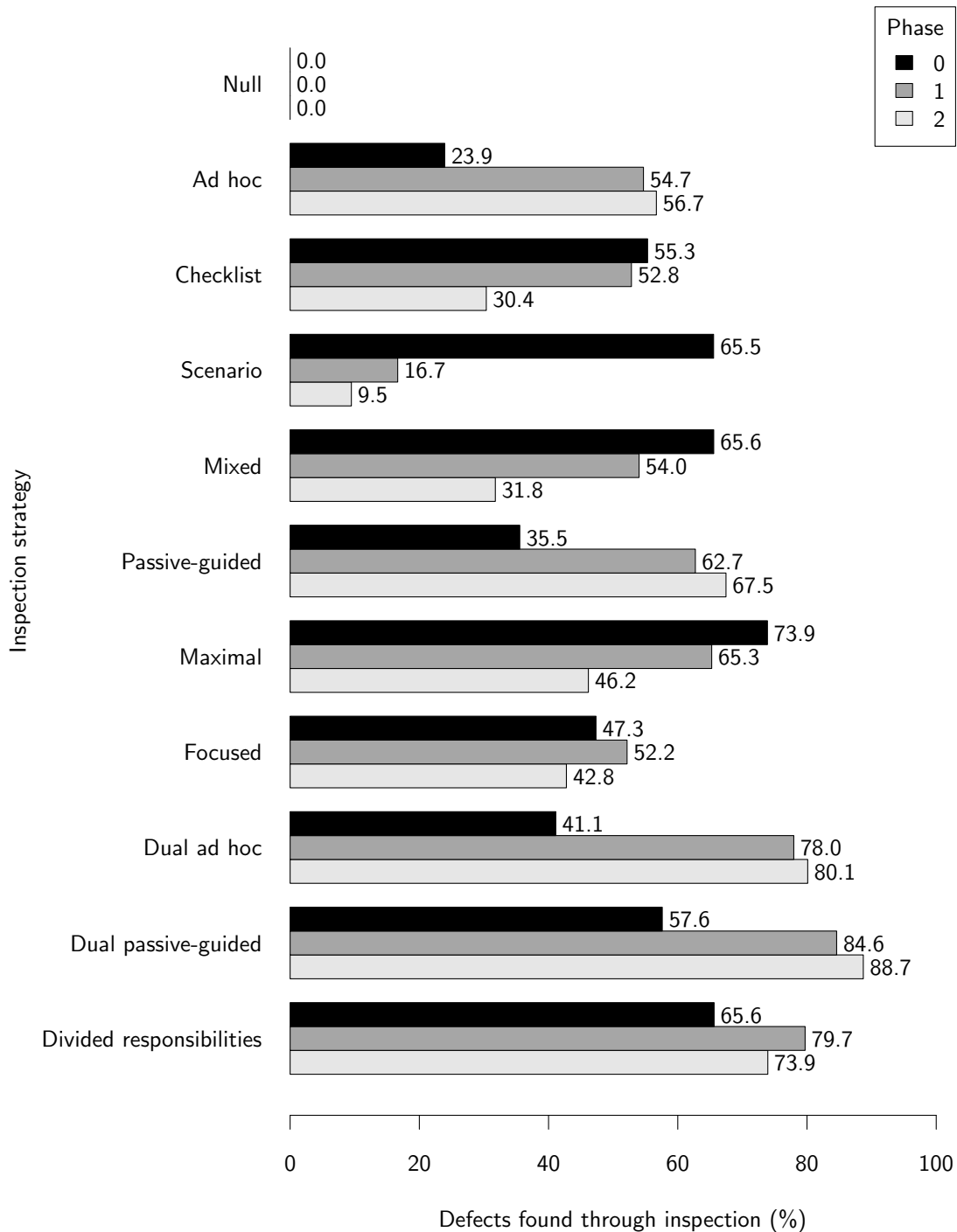
Figure 8.15: Percentage of defects found through inspection in each phase, using each inspection strategy (based on 10,000 simulation runs).

the cost of providing passive guidance. (Some of these provision costs may be mitigated through the use of automated visualisation tools, in which case the passive-guided and maximal strategies would be more cost-effective.)

The mixed and maximal strategies demonstrate that combinations of different inspection techniques may outperform each technique by itself. The mixed strategy combines scenarios and checklists in different phases, and is more cost effective than either by itself. Maximal combines scenarios and passive guidance, and again outperforms both. The focused strategy is another refinement of scenarios, prioritising entities in the system by importance and thus also improving performance.

The two-inspector strategies each show somewhat improved performance over their one-inspector counterparts, mitigated by the increased cost of inspection. The dual passive guidance strategy improves on the single-inspector version, in part because passive guidance is effectively cost free for the second inspector. By contrast, the divided responsibilities strategy makes only marginal improvements over a single-inspector checklist.

8.3.5 Sensitivity Analysis

Sensitivity analysis is conducted by controlling selected model parameters, particularly those that may interact with the inspection strategy. Most such parameters deal with aspects of comprehension. Each parameter is examined separately, in turn. For each parameter, several values are chosen and the simulation run one thousand times for each value.

(The number of inspectors is an exception to this. This parameter is part of the inspection strategy. By assigning each value to a different scenario, the simulator can consider multiple values in a single run.)

These parameters were each assigned probability distributions in order to generate the main dataset. In this sensitivity analysis, the chosen values for a given parameter replace its probability distribution. Thus, the parameter ceases to be a random, uncontrolled variable and instead becomes an independent variable. In each case, all parameters not under analysis retain their original distributions.

In effect, this analysis examines how the construction of the model inputs affect the inspection strategy comparison. Thus, a broader picture of the model's behaviour is presented.
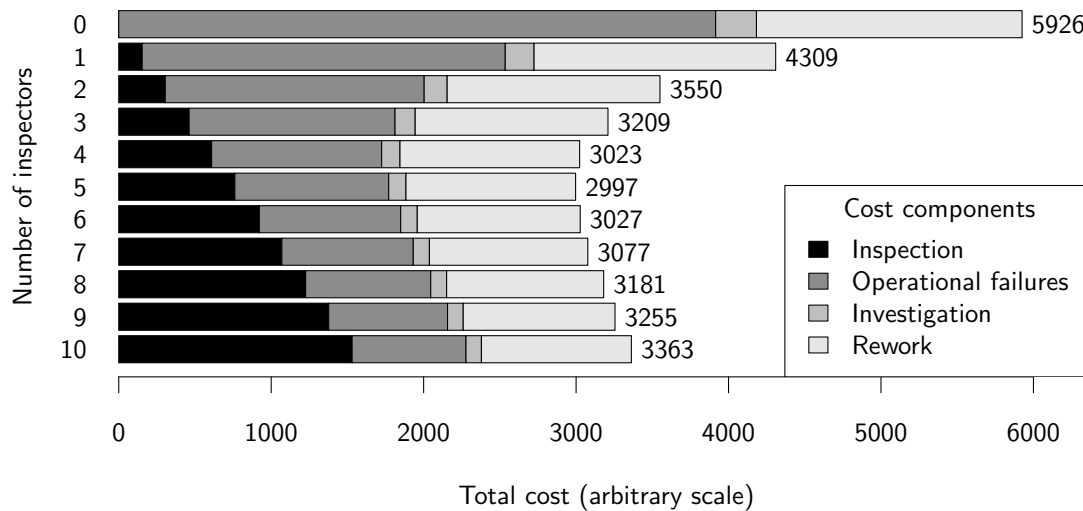
Figure 8.16: Simulated costs for different numbers of inspectors, using the *ad hoc* strategy (based on 1,000 simulation runs).

Figure 8.16 shows the total cost and cost breakdown using the *ad hoc* strategy with different numbers of inspectors. The cost of operational failure, investigation and rework all diminish as more inspectors are added, but at a decreasing rate. Meanwhile, the inspection cost increases linearly with the number of inspectors. Initially, the added inspection cost is small compared to the cost savings from other sources, but eventually the cost of adding an inspector outweighs the benefits. The optimal number of inspectors is highly dependent on the search costs for the various locality types. The chosen model parameters result in the optimal number of inspectors being five, though this is only marginally better than four.

Figure 8.17 shows the effects of system size on cost effectiveness. Size is represented by a coefficient that determines the number of initial entities (while keeping fixed the ratio of entities of different types). A size-1 system is defined to contain, initially:

- two data objects;
- three functional requirements; and
- one external interface.

A size-2 system contains twice this many initial entities in the same proportions, a size-3 system three times as many, and so on. Only the initial entities are explicitly specified; entities in subsequent phases are determined probabilistically. This scheme is used in lieu of traditional size metrics (e.g. lines of code or function points), which cannot easily be derived from the model inputs chosen.

Generally, inspection appears to be less cost effective on small systems. Here, active
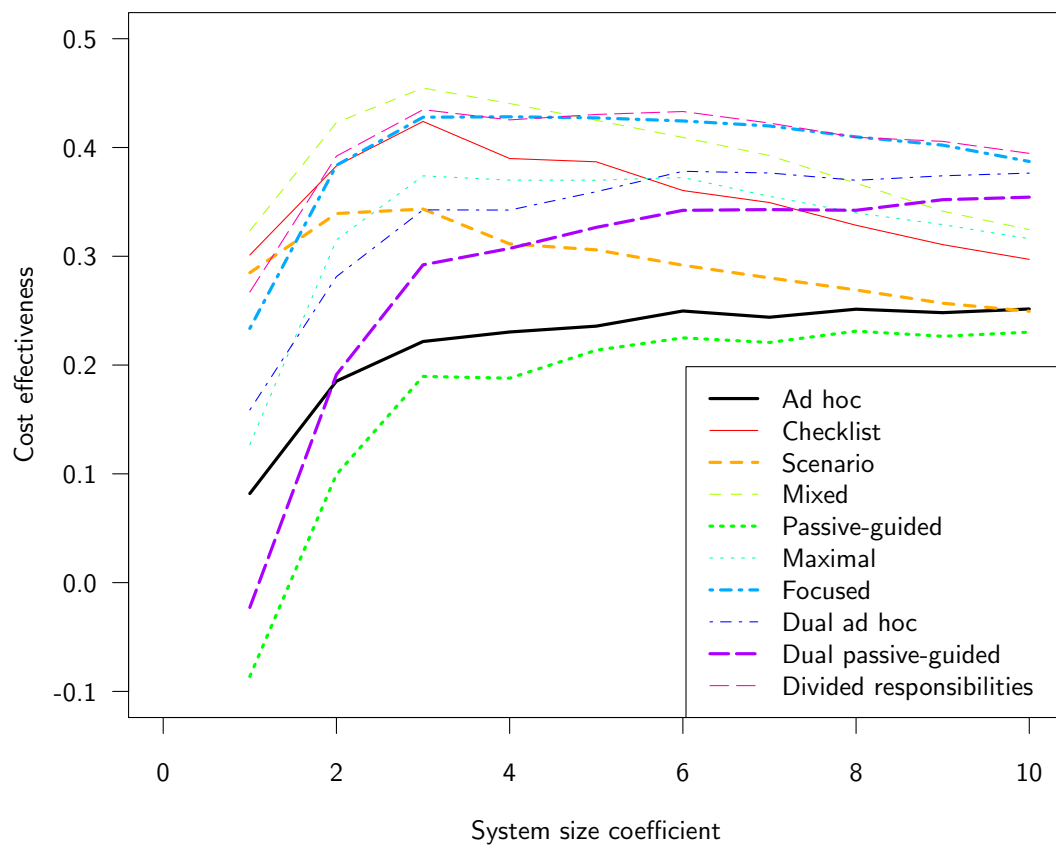
Figure 8.17: Effects of system size on cost effectiveness, for different inspection strategies (based on 1,000 simulation runs for each size coefficient).
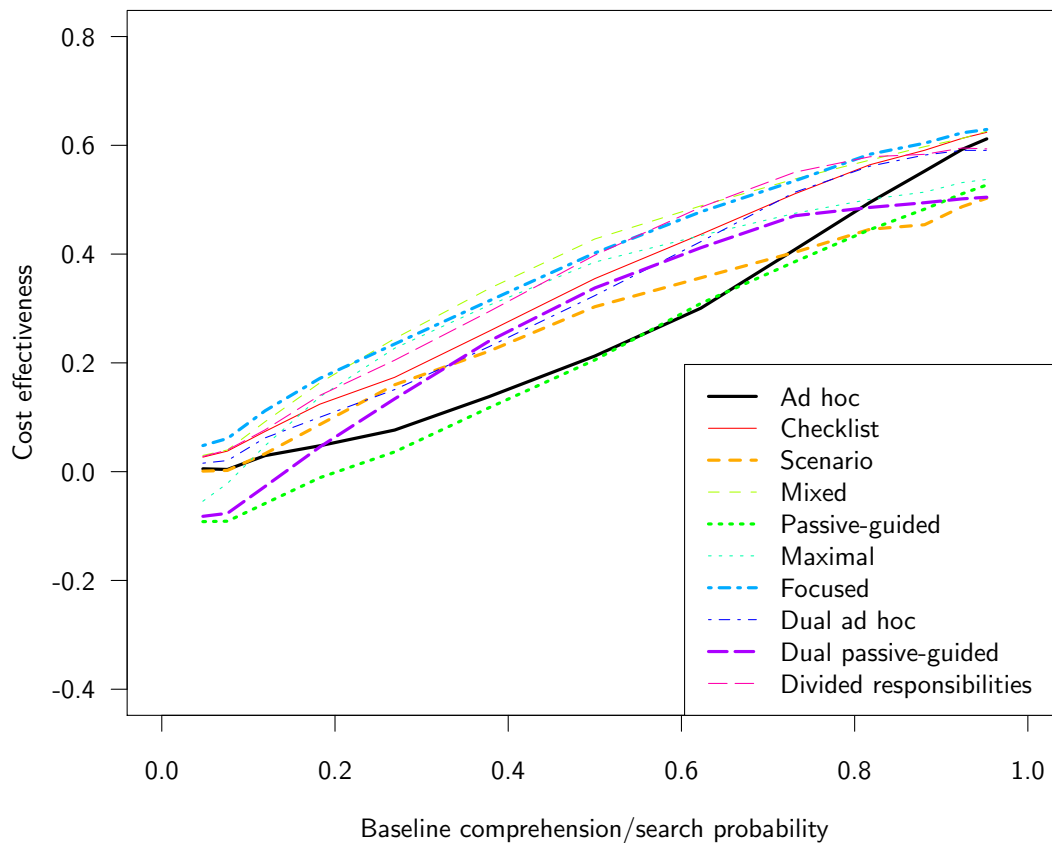
Figure 8.18: Effects of varying the baseline log odds of comprehension and searching, for different inspection strategies (based on 1,000 simulation runs for each odds value between -3 and 3, in increments of 0.5). For clarity, the log odds are transformed here to probabilities.

guidance strategies mostly outperform the others. For a size coefficient of one, the purely passive-guided strategies are even counterproductive.

As the system size increases, cost effectiveness increases for all strategies, up to a point (where the size coefficient is three). After that point, strategies that rely on active guidance begin to see a steady decrease in cost effectiveness, whereas other strategies merely plateau.

For a size coefficient of ten, the checklist, scenario and mixed strategies perform no better, or even worse, than at one. The scenario strategy loses its entire advantage over *ad hoc*. Here, the focused and divided responsibilities strategies still outperform all others, despite slowly becoming less effective.

Figure 8.18 shows the effects of varying the baseline log odds of comprehension and searching. In effect, these parameters are a measure of inspector experience. Higher log odds would naturally represent a higher experience level, and for the purposes of

this discussion, log odds and experience are used interchangeably.

All strategies exhibit an upwards trend in performance across experience levels, but their relative performance also changes. Some strategies see a slight upwards or downwards bulge in their cost effectiveness.

The *ad hoc*, dual *ad hoc* and some active guidance strategies (checklist, focused, mixed and divided) exhibit similar performance at both low and high experience levels, but vary more widely for intermediate experience. Mixed, focused and divided responsibilities are generally the best performing strategies across all experience levels.

The maximal strategy is relatively high-performing between probabilities of 0.2 and 0.6, but begins lower and plateaus at higher probabilities. The dual passive-guided strategy has a similar curve, bulging upwards, a behaviour not seen in either of two related strategies — dual *ad hoc* and single-inspector passive-guided. The latter is generally the worst performing strategy, only briefly surpassing *ad hoc* and scenario.

The scenario strategy begins level with *ad hoc*, and is more effective up until probabilities around 0.7. However, it never ranks better than sixth among all ten chosen strategies, for any level of experience. At high probabilities, scenarios become an encumbrance, and the least effective strategy. This contrasts with the mixed strategy, which uses scenarios in the first phase and is consistently the best or second best performing strategy.

Figures 8.19 and 8.20 show the impacts of varying the active guidance effect and active guidance level effect. Predictably, varying either effect greatly alters the performance of strategies using active guidance, while having no impact on other strategies. The performance of the active guidance strategies relative to each other does not change substantially. With the active guidance effect at zero, the level effect ensures that the active guidance strategies generally perform much worse than *ad hoc* (though they are not actually counterproductive). When set to 4, the level effect is drowned out and these strategies instead outperform all others.

Of note is the divided responsibilities strategy (i.e. two inspectors using different checklists). It outperforms other active guidance strategies when the active guidance effect is zero because it uses two inspectors. However, when the effect is positive, the additional inspector has no noticeable benefit.

Without the level effect (i.e. when set to zero), the scenario and mixed strategies are the equal best-performing. With the level effect set to -0.1, these same strategies are the equal worst, even marginally counterproductive. However, mixed outperforms
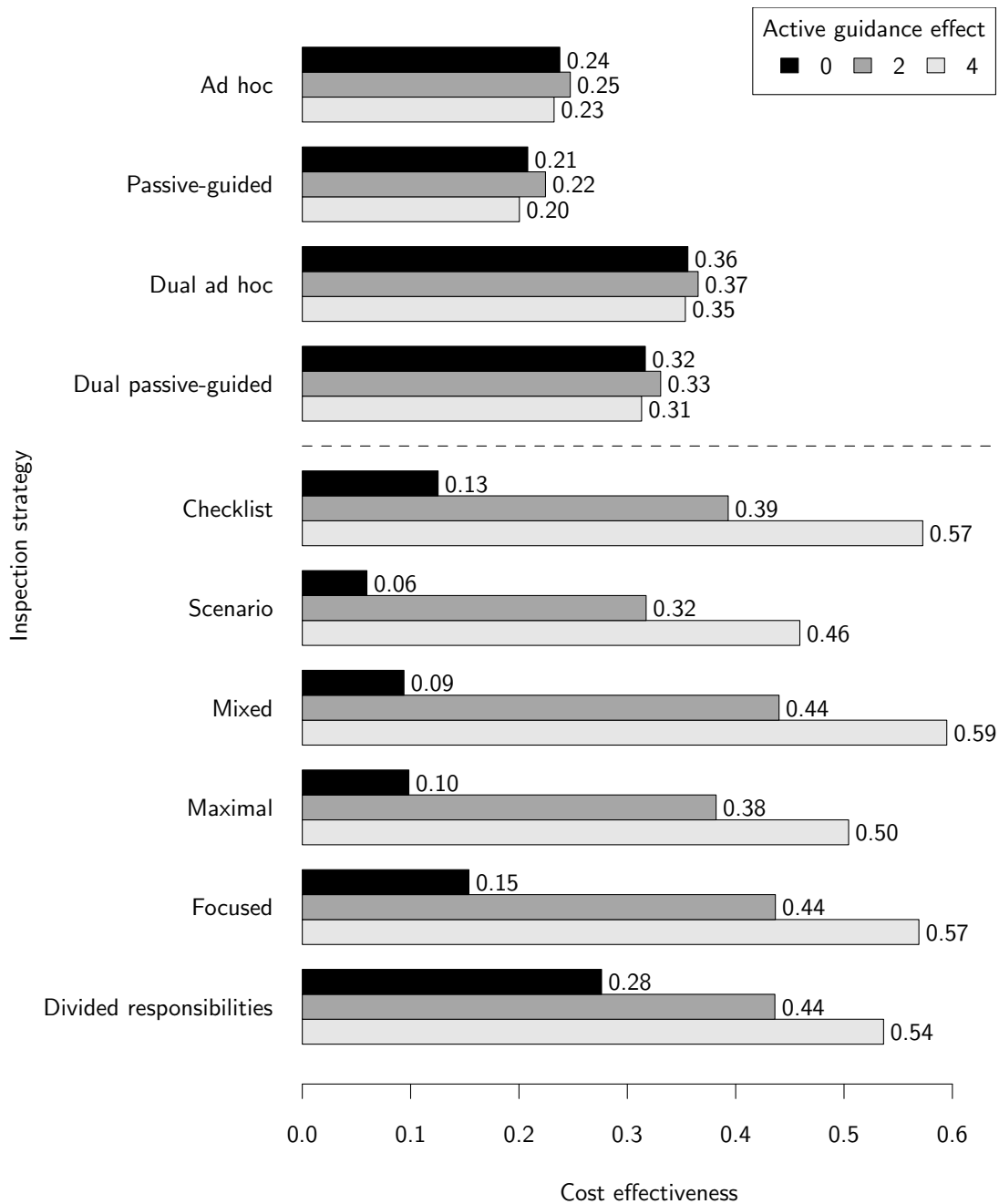
Figure 8.19: Results of varying the active guidance effect, for different inspection strategies (based on 1,000 simulation runs for each effect size). The strategies are grouped into non-active guided (the top four) and active guided (the bottom six).
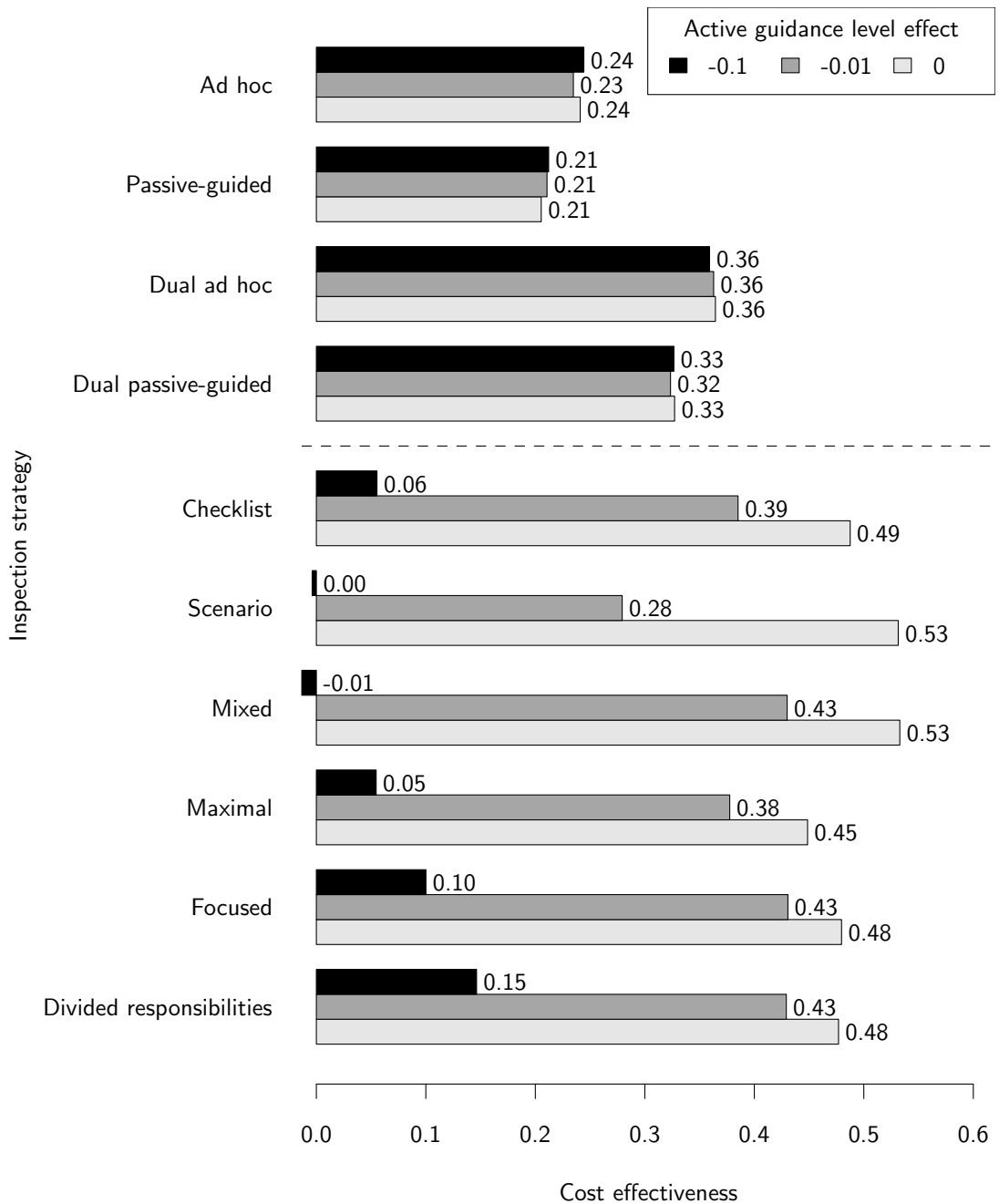
Figure 8.20: Results of varying the active guidance level effect, for different inspection strategies (based on 1,000 simulation runs for each effect size). The strategies are grouped into non-active guided (the top four) and active guided (the bottom six).

scenario when the level effect is -0.01. The focused and divided responsibilities strategies employ relatively little active guidance and outperform the other active guidance strategies under a highly negative level effect. However, their performance relative to *ad hoc* under these conditions is still poor.

Figure 8.21 shows the impact of varying the inverse dependency effects. These are the effects on the comprehension log odds when a comprehension or locality dependency has not been fulfilled. A highly-negative effect means that dependencies are especially important, while a zero effect would make them irrelevant. (A positive effect would defy basic assumptions about cognition, implying that ignorance assists comprehension.)

This effect is a rough measure of the importance of delocalisation. Strategies that maintain their performance despite a highly-negative effect are those that successfully address delocalisation, by helping inspectors to search the right locations and acquire prerequisite knowledge. Strategies involving passive guidance (including passive-guided, maximal and dual passive-guided) are more successful in this respect than other strategies, suffering only a minor drop in cost effectiveness. However, despite the passive-guided strategy's relatively stable cost effectiveness, it still fails to outperform *ad hoc*.

Of the remaining strategies, scenarios appear to be the least affected. By contrast, checklist-based strategies (checklist, focused and divided responsibilities) appear most vulnerable to highly negative dependency effects, with their cost effectiveness reduced by more than half.

8.4   Discussion

8.4.1   Inspection Strategy Comparison

The preceding section reports on differences in simulated cost effectiveness of different inspection strategies. Comparing these results to those obtained empirically presents a challenge, because:

- cost effectiveness has not often been used as a means of comparing reading techniques or broader inspection strategies; and

- from empirical data, there is no clear consensus on which strategies are more effective.
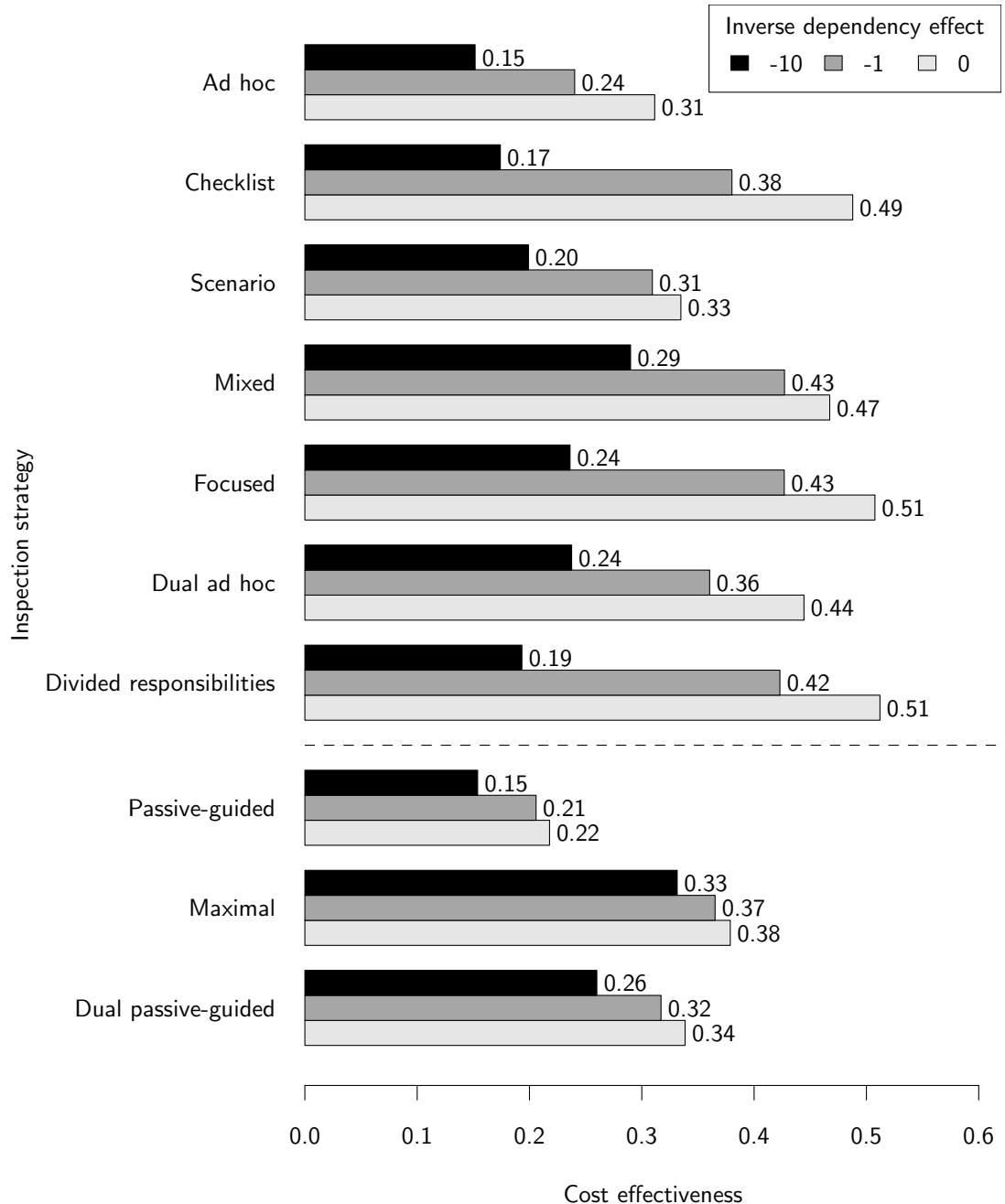
Figure 8.21: Results of varying the inverse dependency effect; the effect of not fulfilling a given comprehension or locality dependency on the odds of comprehension (based on 1,000 simulation runs for each effect size). The strategies are grouped into non-passive guided (the top seven) and passive guided (the bottom three).

Many of the strategies examined in Section 8.3 have not been empirically investigated, though the results for the *ad hoc*, checklist, scenario and focused strategies have some basis for comparison.

The simulation results generally show checklists to be more cost effective than *ad hoc*. Checklists are supported by a large body of anecdotal evidence (Brykczynski, 1999), but the simulation does not specifically support the findings of past controlled experiments, which have shown that checklists have little or no effect (Porter et al., 1995, Porter and Votta, 1998, Hatton, 2008, Akinola and Osofisan, 2009). However, the results in Section 8.3.5 show that checklists can also be less cost effective than *ad hoc* (or only marginally more cost effective) under a variety of conditions. This is in line with results from the checklist experiment discussed in Chapter 7, where checklists were shown to have varying effects on defect detection.

With respect to the scenario strategy, the simulation results can be compared to the meta-analysis conducted by Ciolkowski (2009). Based on a range of prior controlled experiments, Ciolkowski reported that perspective-based reading (a scenario technique) was significantly more effective than *ad hoc*, but significantly less effective than checklists in requirements inspections. The simulator can reproduce a similar result in terms of overall cost effectiveness, for the chosen model parameters. For defect detection in the requirements phase, the simulation data shows that scenarios help detect more defects than either *ad hoc* and checklists. Several controlled experiments have also produced this result, but others have not.

Simulation results for the focused strategy show it to be among the most cost effective. This strategy most closely reflects reading techniques that involve prioritisation, including usage-based reading (Thelin et al., 2003), metric-based reading (Bernárdez et al., 2004) and value-based review (Lee and Boehm, 2005). All three techniques have been found to outperform checklists, for various measures of performance — overall defect detection, critical defect detection or the sum of detected defect importance.

Theoretically, the model suggests that the success of active guidance strategies should depend on there being a relatively constrained set of defect types and other knowledge present in the artefacts under inspection. The model posits that the negative active guidance level effect occurs across the entire comprehension process, but is overridden by a stronger positive effect in instances where specific guidance applies. As more guidance is given, the applicability of the positive effect increases, but so does the magnitude of the negative effect. Where the types of knowledge are limited, a small set of guiding instructions can be applicable to a large proportion of the knowledge contained in the system.

8.4.2 Delocalisation

Much of the discussion of delocalised plans in the academic literature has focused on cognitive support (passive guidance) as a means to address the resulting comprehension challenges, as discussed in Chapter 2, Section 2.3.5. However, the inspection techniques examined by Dunsmore et al. (2003) represent an attempt to apply active guidance to the problem as well.

From the simulation results, the maximal strategy (a combination of active and passive guidance) appears to outperform the checklist, scenario and passive-guided strategies, and is resilient to delocalisation. As Figure 8.21 shows, maximal is the best-performing strategy under a highly negative dependency effect. Such effects make plain the case for guidance, whether active or passive, as noted in Chapter 5. In particular, this strengthens the argument made in Chapter 6 that both active and passive guidance (or cognitive support) should be considered when developing a reading technique.

The passive-guided, maximal and dual passive-guided strategies are handicapped in the simulation by their reliance on the manual provision of passive guidance. With tool support, these costs may be substantially reduced and the cost effectiveness of these strategies correspondingly increased.

8.4.3 Team and System Size

As shown in Figure 8.16 in Section 8.3.5, the lowest costs (and thus the highest cost effectiveness) are achieved for an inspection team containing four or five inspectors, with the fifth inspector having marginal net benefit. This compares to the findings of Weller (1993), who reported that four-person teams exhibit substantially improved performance over three-person teams. Ebenau and Strauss (1994) recommend between three and seven inspectors, preferring lower numbers. Fagan (1976) stated that the inspection team should generally not exceed four inspectors without good cause.

The model does not directly consider the size of the subset of material actually inspected in any single inspection. However, the trend in cost effectiveness over systems of increasing size can be compared to trends reported in existing literature. Porter et al. (1998) model a positive but diminishing trend in defect detection as the system size increases, which they model logarithmically, and explain to be a result of there being more defects in larger systems. Raz and Yaung (1997) also report a positive, logarithmic trend, but rather in the probability of missing a defect during inspection as system size increases.

Therefore, if both the defects detected and defects missed increase logarithmically, the cost effectiveness should remain roughly stationary. Figure 8.17 generally supports this, though it also shows an initial rapid rise in cost effectiveness. The *ad hoc* or passive-guided strategies do appear to converge at fixed cost effectiveness values. The active-guidance strategies (e.g. checklist, scenarios, etc.) exhibit a decline in cost effectiveness that may be attributed to an increasing active guidance level effect.

8.4.4   Interactions

The decline in the cost effectiveness of the active guidance strategies for larger systems represents one of the interaction effects apparent from simulation results. Figure 8.18 in Section 8.3.5 also shows that inspector experience interacts with the inspection strategy. Thus, the optimal choice of strategy depends on the context.

This is demonstrated somewhat in the results presented in Chapter 7, which show a potential (though non-significant) interaction effect between experience and checklists on the probability of auxiliary defect detection. There, the addition of experience had opposing effects depending on the presence of a checklist. Generic experience, even if unrelated to a given defect, appeared to improve the log odds of defect detection, but only in the absence of a checklist. In the presence of a checklist, unrelated experience appeared to hinder defect detection. No interaction was observed with respect to primary defects — those matching items on the checklist, or previous experience — but primary and auxiliary defects must be taken in combination. The magnitude of the interaction effect may depend on the relative number of primary and auxiliary defects in the system. The findings of Biffl (2000) also generally support this for the case of scenarios and checklists, where the effect of inspector capability depends on the reading technique.

Such interactions are further implicitly supported by the failure generally of controlled experiments and replications to consistently reproduce the same results, as shown in Chapter 2, Section 2.2. Moreover, experiments involving industry professionals show somewhat different results from those conducted in an academic environment. Of the four controlled industrial experiments listed in Section 2.2, three found that scenario-based techniques outperform checklists or *ad hoc*. Only four of fourteen academic experiments yielded similar results. If the experimental context (industrial vs academic) is treated as an indicator of experience, then empirical data weakly suggests that the performance of scenarios relative to other strategies is greater for high experience levels.

Unfortunately, the simulation results across different experience levels do not at any

point show scenarios outperforming checklists. However, scenario performance relative to *ad hoc* does improve with experience, up to a baseline comprehension/search probability of 50%. It is conceivable, though purely speculative, that a "high level" of experience corresponds to only a 50% probability of comprehending any given k-instance. (If so, inspection strategy comparisons at higher probability levels would be relevant only for unusually-high levels of experience.)

The bulge observed in Figure 8.18 warrants a tentative explanation. Several inspection strategies exhibit very similar cost effectiveness at both low and high probabilities, but vary more widely for probabilities around 50%. This might be explained in terms of guidance opportunity. At the extreme ends of the experience spectrum, the question of whether the inspector will find a given k-instance is virtually decided even before the inspection strategy is contemplated. A baseline probability of 50% represents the point of least certainty regarding the outcome of comprehension, and therefore the point at which guidance will be most influential.

Figure 8.15 in Section 8.3.4 also illustrates the interaction between strategy and phase. Several strategies have widely-varying defect detection counts across different phases. Some perform better in the requirements phase, while others perform better in later phases. The dual passive-guided strategy consistently outperforms dual *ad hoc* in terms of defects detected, but nevertheless has a lower cost effectiveness. Similarly, maximal detects a greater proportion of defects than mixed in all phases, and yet it too is actually less cost effective.

8.4.5 Defect Detection Dependence

The simulation results also provide an opportunity to examine the independence assumption in capture-recapture approaches to defect estimation, discussed in Chapter 2, Section 2.4.1. The total number of defects present in a system can be estimated from the overlap between defects detected by multiple inspectors, assuming that the detection of defects occurs independently.

In general, two inspectors will not detect the same number of defects. However, the chosen model inputs characterise all inspectors using the same set of probability distributions. Hence, across many simulation runs, the first and second inspectors will have nearly identical defect detection counts for any given scenario.

The actual overlap and expected overlap given the independence assumption can both be calculated from the simulation results shown in Table 8.4; in particular, from the

Table 8.4: Overlap between defects detected by two inspectors, as used in the capture-recapture defect estimation technique.

Phase	Detection proportion		Overlap proportion		
	Ad hoc	Dual *ad hoc*	Actual	Expected	Difference
0	0.239	0.411	0.067	0.057	+0.010
1	0.547	0.780	0.314	0.299	+0.015
2	0.567	0.801	0.333	0.321	+0.011

proportion of defects detected by *ad hoc* and the proportion detected by dual *ad hoc*. Thus:

$$\text{Actual overlap proportion} = (2 \times Ad\ hoc\ \text{proportion}) - \text{Dual } ad\ hoc\ \text{proportion}$$
$$\text{Expected overlap proportion} = (Ad\ hoc\ \text{proportion})^2$$

The results of applying this formula are summarised in Table 8.4. The actual overlap is consistently greater than the expected overlap. As a result, applying the independence assumption will underestimate the total number of defects, a finding previously reported by Briand et al. (2000).

8.5 Summary

This chapter proposes a theoretical framework and model of the software inspection process — a theoretical basis upon which reading techniques, and inspection strategies more widely, may be chosen, developed and refined. It incorporates, formalises and extends the notion of comprehension dependencies, initially discussed in Chapter 6. It includes fine-grained, extensible mechanisms to represent the focus of reading techniques. It also includes an active guidance level effect, designed to model the negative effects on comprehension of misdirected active guidance, noted in Chapter 7.

Based on its inputs, the model reports projected costs and the normalised cost effectiveness for any set of inspection strategies under examination.

Simulation results support the following analysis from empirical studies described in previous chapters:

- that checklists can be either more or less effective than *ad hoc*, as discussed in Chapter 7;

- that a combination of active and passive guidance is most effective at satisfying entity dependencies and countering delocalisation (as proposed in Chapter 6); and

- that the system and the inspector both interact with the inspection strategy, as broadly indicated by the checklist experiment discussed in Chapter 7.

The model illustrates how apparently-contradictory results from studies such as those summarised in Chapter 2, Section 2.2 can be reconciled. Though probably too late to gather additional, fine-grained information on the expertise of subjects from past studies, future studies should take account of possible interactions between the system, inspector expertise and inspection outcomes. Moreover, to build confidence in the meaningfulness of their results, they should express any inspection performance effects in terms of cost effectiveness rather than defect counts.

Chapter 9

Conclusion

"If I have two beans and then I add two more beans, what do I have?"
"Umm... a very small casserole?"

— Blackadder II

The findings presented in this thesis impact in several ways upon our understanding of the mechanics of software inspection. The points raised illustrate hitherto unaddressed problems regarding delocalisation, inspector and system variability and the use of active guidance.

The introduction of an extensible, theoretical approach to understanding software inspection should lead to reduced inspection- and defect-related costs. By instantiating organisation-specific inspection models, or by using or refining reference models, organisations will be able to develop, select or refine more appropriate, and thus more cost effective, inspection strategies. These would make better use of inspectors' expertise and better reflect development practices in use.

This chapter details the findings of the research discussed in preceding chapters, makes recommendations based on these findings and discusses possible extensions to the inspection model.

9.1   Findings

The research presented in this thesis has sought to answer three research questions, listed in Chapter 1, Section 1.1. Those questions are answered here.

9.1.1 Current Industry Practice

Research question 1*: What are the prevalent inspection practices, and in what contexts do they occur?*

The industry survey discussed in Chapter 4 found a diversity of practices in real-world software development, particularly with respect to:

- the domains in which organisations operate;
- the types of development phases employed in a project;
- the proportion of project workload spent in each phase; and
- the types of artefacts in use.

The use of peer review in general is widespread, though Section 4.5.1 discusses broad opportunities for improvements. On average, 44% of total project effort is expended in testing or maintenance phases, coming after the software is actually built. This is suggestive of the presence of defects having a high rework cost, having escaped detection early in a project. Substantial numbers of respondents did not review requirements specifications or other related artefacts used early in a project. More frequent, longer and/or more formal inspections, along with quantitative analysis of their performance, may help reduce workload later in a project.

Survey results identified both checklists and use case traversal as relatively common inspection techniques (though neither was used by a majority of organisations). Other techniques enjoyed less support. (Subsequent chapters — specifically chapters 6 and 7 — discussed the empirical investigation of use case traversal and checklists. Hence, the results of these studies have relatively wide applicability.)

9.1.2 Comprehension and Delocalisation

Research question 2*: What are the challenges inherent in comprehending a system under inspection?*

The statechart study presented in Chapter 5 shows that inspectors can easily miss or over-simplify complex artefact interrelationships; concentrating on obvious connections while neglecting more subtle ones. Participants demonstrated logical but often shallow approaches, including the use of simple names as a basis for mapping one artefact to the other. Many participants identified a one-to-one mapping, without apparently considering plausible many-to-many relationships. This can be contrasted against the

model solution, based on a deep and systematic understanding of the artefact inter-relationships. Despite the relatively small scale of the system, it is apparent that participants had difficulty in identifying these interrelationships. Thus, the statechart study demonstrates why guidance is sometimes needed in inspection.

The scenario study discussed in Chapter 6 adds to the list of comprehension issues, identifying several distinct challenges for inspectors attempting to trace a use case scenario through source code. Some of these include context switching between artefacts, missing method calls and misidentifying method calls due to polymorphism. The effects of such issues can include omission of large amounts of relevant source code from inspection. Overall, coverage of the relevant source code can be low.

Further, individual inspectors' approaches to the task presented were highly varied, despite the exercise requiring all participants to examine and mark the same code in the same way. This raises the question of whether this procedural approach can work equally well for all inspectors. A single fixed set of instructions to inspectors may not uniformly benefit all inspectors.

As indicated by survey results, the simultaneous use of multiple notations in software projects (*artefact diversity*) is also commonplace. This is most prevalent in the main development phase. The proportion of total project workload expended in testing, quality assurance and maintenance suggests that scope exists for improving comprehension and hence inspection performance earlier in a project.

Cognitive support has long been suggested in response to difficulties arising from de-localised plans, which themselves are made more prevalent by the use of object orientation. Simulation results show that inspection strategies using cognitive support are more resistant to the effects of delocalisation than other strategies. Survey results also show that visualisation tools are used by roughly a quarter of software development organisations, leaving inspectors in most organisations without this form of cognitive support.

9.1.3 Active Guidance Effects

Research question 3: *To what extent does active guidance support defect detection, and what are the effects on overall cost effectiveness?*

In contrast to several prior experiments discussed in Chapter 2, Section 2.2.1, the checklist experiment presented in Chapter 7 shows that checklists can improve defect

detection over *ad hoc* reading.

This successful empirical demonstration of checklist utility does not directly contradict previous results. Rather, it comes as a result of measuring checklist effects at a more fine-grained scale — that of individual defects. At this level, both positive and negative effects were observed, depending on whether the defect in question was covered by the checklist. Extrapolating to overall inspection performance, this means that checklists can either help or hinder inspection performance depending on the prescience of their construction — their coverage of actual defects in the software.

This conclusion can be generalised if two tenets of the theoretical framework are held to be true:

- that defect detection is a special case of comprehension; and
- that checklist questions are a special case of active guidance.

Thus, the success of any active guidance-based inspection strategy is closely tied to the ability to predict the types of defects and other knowledge likely to occur in a software system. The simulation results illustrate how active guidance strategies can be less effective in some situations.

For defects specifically, active guidance may be a victim of its own success. Checklists are (or at least should be) based on historical defect data, under the assumption that present defects are likely to be representative of past defects. The extent to which this assumption holds determines, in large part, the extent to which checklists are a useful defect detection tool. However, in the broader context of software development, having defect types reoccur is fundamentally undesirable. Inspector expertise and development practices may change (intentionally or not) in response to past defects, particularly those of greatest concern, specifically in order to avoid them reoccurring. This will alter the types of defects likely to be introduced in the future, and thus undermine — to some extent — the case for checklists.

For non-defect knowledge, the model implies that active guidance is more successful where there are fewer knowledge types, where a set of instructions given to inspectors can cover a greater proportion of the knowledge in the artefacts under inspection. The survey presented in Chapter 4 shows that fewer artefact types are used towards the beginning of a project, with the number of distinct notations more than doubling as the project progresses from analysis/requirements to development. Reflecting this, simulation results from Chapter 8 show that active guidance strategies are generally more effective earlier in a project.

Standardisation also contributes to the predictability of defects and other knowledge occurring in a system. Thus, where artefacts are standardised, active guidance is likely to be more applicable. In the case of formal requirements specifications, based on survey results from Chapter 4, there is room for both:

- increased use of active guidance, taking advantage of existing standardisation; and
- increased standardisation, leading to further opportunities for the use of active guidance.

The experiment discussed in Chapter 7 also appeared to show that the negative effect on defect detection (but not the positive effect) was amplified by experience. This interaction effect was suggestive but not statistically significant. If real, it indicates that the risks of misusing active guidance increase across experience levels. A similar effect was observed in simulation results (Chapter 8, Section 8.3.5), where the heavy use of active guidance led to poor inspection results for experienced inspectors (compared to *ad hoc* reading).

Simulation results too show that active guidance becomes less effective as system size increases. Experimental results show that the qualitative nature of the system is also an important determining factor in inspection performance.

The prioritisation concept initially discussed in Chapter 2, Section 2.2.3 is supported by simulation results, which show that the prioritised ("focused") strategy is consistently one of the most cost effective. This comes despite having lower defect detection rates than other strategies.

9.1.4   Resolving Uncertainties

The overarching research question framing this thesis has been:

> *How can the uncertainties of software inspection be resolved, in order to make recommendations of best practice?*

Such uncertainties have concerned existing real-world inspection practices, the comprehension challenges arising from inspection and the relative merits of different reading techniques. In particular, as discussed in Chapter 2, Section 2.2, experimental replications of different reading techniques have produced inconsistent results that have not

been easily reconcilable.

The purpose of inspection theory, and in particular the model presented in Chapter 8, is to address these uncertainties and reconcile the inconsistencies. The model seeks to describe the effects and interactions within inspection in the language of formal mathematics, so as to encapsulate our understanding of inspection and explore its consequences.

The simulation results thus obtained are broadly consistent with findings previously published in the academic literature. The model, as instantiated, indicates that:

- the optimal inspection team size is about four or five inspectors (Fagan, 1976, Weller, 1993, Ebenau and Strauss, 1994);
- for non-active guidance strategies, cost effectiveness converges as system size increases, with the increased detection rate (Porter et al., 1998) being counterbalanced by an increased miss rate (Raz and Yaung, 1997);
- prioritisation-based techniques are generally more cost effective (Thelin et al., 2003, 2004, Winkler et al., 2004, 2005, Bernárdez et al., 2004, Lee and Boehm, 2005); and
- in capture-recapture approaches to defect estimation, the independence assumption results in an underestimate of the number of defects left undetected (Briand et al., 2000).

Where inconsistent results appear in the published literature, particularly with respect to reading techniques, the model broadly reproduces these different outcomes under certain conditions. As discussed in the previous section, active guidance can either help or hinder inspection performance, depending on the extent of its provision, the predictability of defects and other knowledge in the system, the project phase and inspector experience.

Thus, this research has sufficiently resolved inspection uncertainties to allow a number of statements of best practice to be made.

9.2   Recommendations

In helping to resolve inspection uncertainty, the findings discussed in the previous section suggest a number of beneficial inspection practices. Table 9.1 indicates the discussions in preceding chapters that support these recommendations.

Table 9.1: Recommendations and their most direct supporting discussion/analysis in this thesis.

Recommendation	Supporting discussion (section numbers)
1. Use active guidance only where it can be made materially relevant.	5.4, 6.3.3, 7.5.1, 8.4.4
1a. Use active guidance in early phases.	4.5.3, 8.3.4
1b. Use active guidance for standardised artefacts.	4.5.3, 7.5.1
1c. Standardise artefacts, where possible.	4.5.3
2. Use active guidance complementing inspector expertise.	6.3.3, 7.5.2, 8.4.4
3. Use prioritisation.	8.4.1
4. Use visualisation tools.	4.5.2, 6.3.2, 8.4.2
5. Combine the use of active and passive guidance.	6.3.2, 8.4.2
6. Instantiate and use the model.	Chapter 8
6a. Collect inspection-related data, where possible.	4.5.1, 7.5.3, 8.4.4
6b. Distrust anecdotal accounts of inspection performance.	7.5.4

1. *Use active guidance only where it can be made materially relevant.* Active guidance must reflect the actual composition, characteristics and quality issues in the artefacts under inspection. If such information cannot be predicted or obtained with reasonable accuracy, then the provision of active guidance is likely to be counterproductive. However, where properly used, active guidance does improve inspection performance. This leads to the following sub-recommendations:

   (a) *Use active guidance in early phases.* Active guidance can more easily be targeted at specific artefact characteristics and quality issues when there is less overall complexity; e.g. when there are fewer artefact types. This is the case in early stages of a project, as shown in chapters 4 and 8.

   (b) *Use active guidance for standardised artefacts.* Active guidance can also be more successfully employed in artefacts adhering to well-defined standards.

   (c) *Standardise artefacts, where possible.* If there is further scope for standardising some types of artefacts without sacrificing flexibility, doing so would improve the applicability and effectiveness of checklists or other active guidance techniques.

2. *Use active guidance complementing inspector expertise.* Active guidance should not coincide with an inspector's expertise, but rather should reflect concepts and issues that may help to extend it. Results from Chapter 7 suggest that active guidance coinciding with expertise narrows the inspector's focus. Simulation results from Chapter 8 show that active guidance is generally less effective for more highly-experienced inspectors.

3. *Use prioritisation.* The simulation results showed that the prioritised (focused) inspection strategy, using active guidance to target high-cost defects, was more

cost effective than non-prioritised variants. This suggests that, where active guidance can be used, a prioritisation scheme of some sort should be implemented.

4. *Use visualisation tools.* The simulation results further showed that passive-guided strategies (implementing cognitive support) were more resilient to delocalisation than other strategies. Therefore, theoretically, an increased use of appropriate software visualisation tools should help address some of the comprehension issues arising from delocalisation, including the issues identified in the scenario study in Chapter 6.

5. *Combine the use of active and passive guidance.* Respecting the above recommendations and following from the simulation results discussed in Chapter 8, the simultaneous use of both active guidance and visualisation tools may lead to a better inspection outcome than either approach by itself.

6. *Instantiate and use the model.* The above recommendations notwithstanding, inspection strategies are best compared within an organisational context. By using historical data and/or expert estimation to instantiate or contextualise the model, project leaders can evaluate inspection strategies based on the unique characteristics of the people, software and development practices involved. This leads to two further sub-recommendations:

    (a) *Collect inspection-related data, where possible.* The existence of historical data underlies effective decision making with respect to inspection strategies, particularly given the extent to which inspectors, systems and organisations differ.

    (b) *Distrust anecdotal accounts of inspection performance.* Questionnaire data from Chapter 7 suggest that inspectors are not generally in a position to objectively evaluate inspection performance based solely on their experience. An essential component of cost effectiveness is the hypothetical cost incurred without inspection; this cannot be known without some level of quantitative analysis.

Software development organisations might be convinced to begin collecting more quantitative data if such collection ultimately leads to reduced costs. With data to populate the model, accurate predictions can be made of inspection cost effectiveness. This is comparable to recalibration in COCOMO II (Boehm et al., 2000), wherein organisations adapt the model to their own historical data and expert estimates. Here, the generic inspection model itself provides a template for such data collection.

A series of reference models might also be developed to fulfil the same function, but outside of any one organisation. These reference models would also be instantia-

tions of the generic inspection model. Rather than being organisation-specific, they would be developed to model commonly-occurring software development scenarios. Organisations lacking the resources to collect historical data or expert estimates could use the closest-matching reference model instead, possibly refining it with whatever organisation-specific data may be available.

9.3 Extensions

The framework, model and simulation provide a theory of software inspection, to characterise and predict outcomes of particular approaches. A number of opportunities exist for further refinement and investigation of the model, as discussed in this section. The suggested modifications would help better reflect aspects of the software development process, leading to a better understanding of that process and a more accurate model.

9.3.1 Data Collection

One of the model's functions is as a reference for the types of empirical data that might be collected in future work, so as to better predict inspection outcomes. Though many model inputs are demonstrably system-, person- or organisation-specific, it is conceivable that some effects are consistent across a broad range of software development practices. Future research should seek to identify any such constants; this would lessen the data collection task faced by each individual organisation.

The model also currently includes more inputs than the principle of parsimony (or Occam's Razor) might suggest. It is possible that some of the aforementioned constants might be zero. That is, they may be dispensed with entirely if future experiments fail to find statistically significant effects. For the time being, these effects are postulated anyway so as to explore the scope of what might significantly affect inspection cost-effectiveness.

The repeated instantiation of the model in a research setting would help to explore and refine fundamental effects and relationships in a way not easily achieved by individual organisations.

9.3.2   Hierarchy and Propagation

The model regards hierarchy and propagation as one-to-many relationships. That is, a given entity may owe its existence to at most one other entity, through one of these two mechanisms. However, software development reality is somewhat more complex. A class may not result from any single functional requirement or data object in the requirements document, but rather from a combination of requirements entities.

An extension to the model might therefore introduce many-to-many hierarchy and propagation relationships, where a given entity owes its existence to a group of entities. Moreover, this group of entities might occur in both the current and previous phases; that is, a single entity may arise via a combination of hierarchy and propagation.

9.3.3   Markers

In the current model, marker assignment (X_K) is a binary variable. Each marker is either present or absent. However, more information than this is often available. For instance, Thelin et al. (2003) and Lee and Boehm (2005) each use three different levels of criticality to classify potential or actual defects. Complexity might be measured on an even finer scale, using metrics like cyclomatic complexity (McCabe, 1976).

The model might be augmented to allow for such multi-valued marker assignment. Unfortunately, multi-valued markers break orthogonality. In the current model, an entity can be important, complex, neither or both (or some similar combination of other markers). However, an entity cannot simultaneously be both somewhat and very important. The levels of a multi-valued marker would be mutually exclusive, and thus not independent.

This creates a problem in combination with the many-to-many hierarchy/propagation relationships discussed above. Markers propagate along with entities. If markers originate from multiple sources, then some may conflict. For example, if entity A is very important, entity B is somewhat important, and entity C results from both A and B, then what level of importance should be assigned to C?

Such questions do not arise in the current model (due to binary markers and one-to-many propagation), but may arise in future extensions of it.

9.3.4   Comprehension

Several extensions to the model's view of comprehension are possible.

The framework excludes locality-to-locality dependencies, but this restriction is arguably artificial. Such *structural* dependencies may represent the occurrence of one locality within another, such as a method within a class. If a class is searched, the log odds of searching any given method within that class increase. Structural dependencies would exist alongside comprehension, locality and decision dependencies.

As specified in the model, the passive guidance concept may be an oversimplification of a more complex set of cognitive support mechanisms. For any given entity, there may actually be several conceivable forms of passive guidance, some being more effective and/or costly than others. For instance, Walenstein (2002) describes three general forms of cognitive support. Some of these may be more naturally modelled by special entities representing the annotations or alternate visualisations constructed.

The active guidance level effect may also be more complex than specified in the model. Conceptually, the effect includes both the cognitive overhead of following instructions and the withdrawal of attention from those entities not mentioned. This may vary between entities. Moreover, for defects, the effect may relate more to the number of defect types receiving active guidance (i.e. the number of checklist questions) than the number of actual defects present.

A more extensive change to the model might consider the passing of time within an inspection, to more naturally represent the inspection process. The current model considers time only in the sense of development phases. In a temporal model of the inspection process, localities would be searched over distinct periods of time, and k-instances would be comprehended at specific points in time. These events would be influenced only by what has happened previously. Inspector fatigue could then be modelled as a factor determining the effectiveness of comprehension.

9.3.5   Verification

The model is principally concerned with evaluating the inspection process. As a result, other activities like development, testing and operational use are not modelled in depth. However, all play a role in determining inspection cost effectiveness, and so future extensions to the model might take a more detailed approach.

In particular, the following should be considered for inclusion:

- general quality issues that impact on development and maintenance costs (e.g. the use of anti-patterns or poorly formatted code), not just defects causing operational failures;

- modelling of operational failures, taking into account steps taken by users to work around defects and/or mitigate their effects;

- modelling of the formal inspection process, including the effects of meetings and the potential for re-inspection;

- modelling of defect detection during development, not just in designated verification activities; and

- modelling of the costs and probabilities of release delays and project abandonment due to defects and quality issues.

9.3.6 Incomparable Costs

As discussed in Section 2.4.4, some of the costs arising from software defects may not be expressible in units of time, effort or currency. Such costs (such as human injury, loss of privacy, etc.) are not currently taken into consideration by the model, which assumes that all costs are expressed in interchangeable units.

Nevertheless, these costs can in principle be modelled, as long as no attempt is made to exchange one type of cost for another. The current model predicts a single scalar cost value for each inspection strategy. An extended model might instead consider a *cost vector* for each strategy, where each vector element represents a qualitatively different type of cost. For example, a cost vector might contain financial and human injury costs. The extended model could provide separate estimates of these costs.

For a single inspection strategy, this information should be used to inform stakeholders of any risks posed by the software. Judgements as to the acceptability of these risks should be left to the relevant stakeholders.

In some cases, even when dealing with incomparable costs, such a model may still be able to deliver an unambiguous judgement as to which of two inspection strategies is preferable (i.e. less costly). When comparing two strategies, each pair of corresponding costs would be examined. If all costs for one strategy were lower than those for the other

strategy, the former could be chosen without posing any ethical dilemma (assuming the absolute costs are acceptable, as above).

Otherwise, a human decision would be required (e.g. if one strategy achieved lower financial costs while the other achieved lower human injury costs). In this case, the projected costs should inform a consensus decision involving all affected stakeholders, who must agree on an acceptable trade-off.

9.4   Summary

This chapter has tied together the discussion of four empirical studies and a theoretical inspection model. In answering the research questions posed in Chapter 1, several recommendations of best practice have been made. These are centred around the use of active guidance and cognitive support, and provide more fine-grained, nuanced advice on the use of reading techniques than previously available. Opportunities exist to refine the inspection model and thus extend our understanding of software inspection.

Ultimately, this thesis demonstrates that there is no single best inspection strategy. Nonetheless, there is a logic underlying the relative performance of different strategies. The research presented here represents an argument for inspection theory and an articulation of its underlying logic, from the details of comprehension to the scope of an entire software project. The model is predictive and explanatory, yet through abstraction can apply to a broad range of artefact types and development methodologies. By instantiating and using the proposed model, software engineers will be able to develop, refine or select an appropriate inspection strategy based on the circumstances of a project.

Appendix A

Industry Survey — Materials

This appendix contains information supplementing Chapter 4.

Figures A.1 and A.2 show the background and introductory information presented to potential industry survey respondents.

This survey is being conducted by two PhD students within the Department of Computing at
Curtin University of Technology:

- David McMeekin is examining how inspections affect program comprehension within the
  software development lifecycle.

- David Cooper is investigating how knowledge of software artefact interrelationships can
  assist the software inspection process.

Both are working under the supervision of Dr Brian von Konsky.

Approval has been granted for this research by the Curtin University Human Research Ethics
Committee. The approval numbers are 'PHD JG 002/2006' and 'PHD JG 005/2006'. If
needed, verification of approval can be obtained either by writing to the Curtin University
Human Research Ethics Committee, c/- Office of Research and Development, Curtin University
of Technology, GPO Box U1987, Perth, 6845 or by telephoning 9266 2784.

Figure A.1: Background information provided to potential industry survey respondents.

The purposes of this survey are:

- to determine the prevalence in the software engineering industry of a range of notations,
  methods, techniques and tools, and

- to understand the factors that influence their use.

The researchers ask that one person (or more) within your organisation, department or team
complete this questionnaire. This person should be well-informed of the software development
activities therein. All responses will be greatly appreciated. If you need clarification on some
questions, please email David Cooper at `david.cooper@postgrad.curtin.edu.au`.

Survey results will be published in conference paper(s) or journal article(s). However, no
information identifying you or your organisation, department or team will be published.

Your organisation's name:
(and department/team name if applicable)

_____

A contact email address:
(You may be contacted to clarify your responses if needed.)

_____

Please note:

- We're interested in what *actually* happens in your organisation/department/team, not
  just what is written down (unless otherwise stated).

- In some cases none of the provided checkboxes may be relevant, in which case you should
  simply leave them unselected.

- Many questions have an "Other(s)" option, and all have a "Comment on this question"
  option. Please make use of these where appropriate.

Figure A.2: Introductory information provided to potential industry survey respondents.

Appendix B

Statechart Study — Materials and Raw Results

The material shown here complements the description and analysis given in Chapter 5.

B.1   Forms and Sheets

Figures B.1, B.2, B.3 and B.4 show the information sheet, consent form, instructions and questionnaire given to participants.

B.2   Source Code

The following is the complete Java source code for the Download class used in the statechart study. (Most of this code is also shown in figures 5.2 to 5.9 in Chapter 5.)

```java
import java.io.*;
import java.net.*;

public class Download implements Runnable
{
    private static final int READ_SIZE = 1024; // bytes
    private static final long TIMEOUT = 60000; // milliseconds
    private static final long CHECK_INTERVAL = 250; // milliseconds

    private URL url;
    private String file;
```

Consent Form

Study: Identifying interrelationships between state charts and source code

By signing your name on this form, you agree to the following:

- "I have been informed of and understand the purposes of the study."
- "I have been given an opportunity to ask questions."
- "I understand I can withdraw at any time without prejudice."
- "I understand the study is not connected with any of my university assessments."
- "Any information which might potentially identify me will not be used in published material."
- "I agree to participate in the study as outlined to me."

Name of participant: _____

Signature: _____ Date: _____

Figure B.1: The consent form signed by participants in the statechart study.

```
private InputStream inputStream;
private OutputStream outputStream;

private long startTime;
private long downloadedSize = 0;
private long size;
private boolean stopped = true;

private double speed = 0.0;

/**
 * Creates a new Download object. The download is automatically started in
 * a new thread.
 */
public Download( URL url, String file ) throws IOException
{
    this.url = url;
    this.file = file;
    startDownload( );
}

/** Starts or restarts the download. */
private void startDownload( ) throws IOException
```

Information Sheet

Study: Identifying interrelationships between state charts and source code

1. Aims

The purpose of this study is to gain insight into how relationships between software artefacts can be tracked. At present, this is largely unknown. Specifically, the study will look at the use of UML state charts and Java source code. The researchers intend to use the results to better understand and further develop techniques for cross-referencing software artefacts.

2. Participation

Participation in this study is entirely voluntary. Participants may withdraw at any time – no reason needs to be given. Nobody will be penalised in any way for non-participation or withdrawal.

This is not a test. This study is entirely separate from all university assessments, and will have no bearing on your ability to succeed in your course.

The study will take about half an hour for each participant. It will involve reading a UML state chart and Java source code, and deciding upon particular interrelationships. Following this there will be a short questionnaire as well as a short, voluntary interview, which will be recorded.

There are no risks to participants, and participants may benefit by gaining a deeper understanding of UML and/or Java.

3. Publication of results

No personal information will be collected, except for the express purpose of obtaining written consent to participate. Consent forms will not be linked to the collected data. Participants will not be identifiable from any material published as a result of this study.

The format of any published data will depend on the results of the study.

4. Contacts

If you want further information or have any queries regarding the study, feel free to contact the researchers:
• David Cooper (PhD student) – cooperdj@cs.curtin.edu.au
• Brian von Konsky (supervisor) – bvk@cs.curtin.edu.au
• Mike Robey (supervisor) – mike@cs.curtin.edu.au

This study has been approved by the Curtin University Human Research Ethics Committee. If needed, verification of approval can be obtained either by writing to the Curtin University Human Research Ethics Committee, c/- Office of Research and Development, Curtin University of Technology, GPO Box U1987, Perth, 6845 or by telephoning 9266 2784.

Please keep this sheet for your own reference.

Figure B.2: The overview/information sheet given to participants in the statechart study.

Instruction Sheet

Study: Identifying interrelationships between state charts and source code

Instructions

You have been given a UML state chart and the Java source code for the same class. You have up to 20 minutes to complete the following task:

Determine which sections of the source code implement each state transition in the state chart. This can be indicated on the source code printout. The following sections provide some background information on how the Java code works.

Once you've completed the task, answer the questions on the questionnaire.

Class overview

The Download class is designed to download files. An instance of the class is created to download a single file from a given URL and write it to disk. It does this in a separate thread of execution, so the file can be downloaded while other things, such as user interaction, are happening.

Threading

In the source code, the statement "`new Thread( this ).start( );`" causes Java to execute the `run()` method in a new thread. The thread will stop when the `run()` method finishes. The `run()` method itself is responsible for the actual downloading. Executing it in a separate thread means that downloading can proceed 'in the background'.

Starting, stopping and finishing

The `startDownload()` method establishes a connection to the URL and starts a new thread in which the actual downloading takes place. It is called by the constructor, but can also be called to restart a stopped download.

A download is stopped when either the `stopDownload()` method is called, or a timeout occurs, in which no data is received for 60 seconds.

Stopping a download is different from finishing it. A download is finished when all bytes have been received.

Mechanics of downloading

The URL is downloaded in chunks. Each time a chunk of data is received, it is written to file. If a chunk larger than a kilobyte is received, it will be dealt with one kilobyte at a time.

Each time a chunk is received, the current time is noted. Thus, timeouts can be detected.

Figure B.3: The instructions given to participants in the statechart study.

Questionnaire

Study: Identifying interrelationships between state charts and source code

1. Rate your confidence in your understanding of the **UML state chart** from 1 (low confidence) to 4 (high confidence):

   1   2   3   4   (circle one)

2. Rate your confidence in your understanding of the **Java source code** from 1 (low confidence) to 4 (high confidence):

   1   2   3   4   (circle one)

3. Any specific comments concerning the state chart or source code:

4. Any comments concerning the activity itself:

Figure B.4: The questionnaire completed by participants in the statechart study, in addition to the main task.

```
{
    if( stopped )
    {
        URLConnection connection = url.openConnection( );
        connection.connect( );
        inputStream = connection.getInputStream( );
        outputStream = new FileOutputStream( file );

        size = connection.getContentLength( );
        startTime = System.currentTimeMillis( );

        // Create a new thread for the download to run in. This will call
        // the run() method.
        new Thread( this ).start( );
    }
}


/** Stops the download, assuming it has been started. */
private void stopDownload( )
{
    stopped = true;
}


/**
 * Downloads from the URL supplied to the constructor. The method
 * shouldn't be called directly. It is started indirectly by the
 * startDownload() method.
 */
public void run( )
{
    byte[] buffer = new byte[ READ_SIZE ];
    boolean timeout = false;

    stopped = false;

    try
    {
        while(( downloadedSize < size ) &&
                !timeout &&
                !stopped )
        {
            int bytesAvailable = inputStream.available( );

            if( bytesAvailable >= READ_SIZE )
            {
                // We've retrieved enough data to fill the buffer. Write
```

```
                    // it to disk and reset the timeout counter.
                    inputStream.read( buffer );
                    outputStream.write( buffer );
                    downloadedSize += READ_SIZE;
                    waitStartTime = System.currentTimeMillis( );
                }
                else if( bytesAvailable > 0 )
                {
                    // Some data was retrieved. Write it to disk and reset the
                    // timeout counter.
                    inputStream.read( buffer, 0, bytesAvailable );
                    outputStream.write( buffer, 0, bytesAvailable );
                    downloadedSize += bytesAvailable;
                    waitStartTime = System.currentTimeMillis( );
                }
                else
                {
                    // No data retrieved. Sleep for a small interval to avoid
                    // wasting CPU time. Check for a timeout.
                    sleep( CHECK_INTERVAL );
                    if( System.currentTimeMillis > waitStartTime + TIMEOUT )
                    {
                        timeout = true;
                    }
                }
                calcSpeed( );
            }
        }
        catch( IOException e )
        {
        }

        stopped = true;
        inputStream.close( );
        outputStream.close( );
    }

    /**
     * Called by the run() method to calculate the mean transfer rate (bytes
     * per second) of the download so far.
     */
    private void calcSpeed( )
    {
        long time = System.currentTimeMillis() - startTime;
        if( time > 0 )
        {
```

```
            speed = (( double )downloadedSize / ( double )time ) * 1000.0;
        }
    }


    /**
     * Returns the last calculated transfer rate (bytes per second) of the
     * download, or 0 if no calculation has yet been made.
     */
    public double getSpeed( )
    {
        return speed;
    }


    /** Returns the download progress as a percentage. */
    private double getPercentDone( )
    {
        return (( double )downloadedSize / ( double )size ) * 100.0;
    }


    /** Returns true iff the download has finished. */
    public boolean hasFinished( )
    {
        return downloadedSize >= size;
    }


    /** Returns true iff the download is in progress. */
    public boolean isDownloading( )
    {
        return downloadedSize < size && !stopped;
    }
}
```

B.3   Raw Results

Table B.1 contains the complete set of coded mappings between the statechart and
source code, for each participant.

Table B.1: Participants' coded responses. State transition markings in the source code were classified according to the fourteen categories, shown in columns. Question marks indicate ambiguity.

ID	A	A′	B	C	D	E	F	G	H	H′	H1	I	J	K
1			1	2			5	3	6, 7				4	
2	1		1	2		5	5	3, 5	6, 7					
3			1	2		4	5	3		6, 7				
4	1, 5			2				3	6		2	5	4	7
5			1	2		4	4	3						
6			1	2			4	3	6, 7					
7			1	2			5	3	4					
8			1	2		4	4, 5	3, 4	6, 7					
9			1	2		4	5	3		6, 7				
10			1	2	5		5	3				5	4	5
11	1		7	2			5	3	6		4			
12			1	2			5	3	6				4	
13	1		1	2				3	6, 7					5
14	1						5	3				2		
~~15~~	1?		1?	2, 3?			5?	3?	2, 3?				4	5?
16	1		1	2				3				2	4	5
17			1	2		4	5	3		6, 7	2			
18			1	2			5	3	6, 7					
19	1		1	2, 7			5	3	4, 6					
~~20~~														
21			1	2		4	5	3		6, 7				
22			1	2			4, 5	3	2					
23			1	2		3	5	3		6, 7			4	
24						2, 3, 4	5	3	2, 3, 4	6, 7				
25			1	2		4	4, 5	3	3				4	
26			1	2			5	3	3			2	4	5
27			1	2		5	4	3	3, 4, 6, 7					
28		6, 7	1			2	5	3						

Appendix C

Scenario Study — Materials

This appendix complements the description and analysis given in Chapter 6.

C.1 Forms and Sheets

Figures C.1 and C.2 show the consent form and information sheet given to participants. Figures C.3 and C.4 show offline reproductions of the two web-based questionnaires completed by participants.

C.2 Source Code

This section lists the source code used in the scenarios study. Some minor spacing and line break modifications have been made.

C.2.1 AudioPlayer.java

`AudioPlayer.java` was used to test the system under inspection, but was not shown to participants in the study.

```
package audioplayer;

public class AudioPlayer
{
    public static void main( String[] args )
```

Consent Form

Identifying and understanding software development artefact interrelationships

By signing this form, you agree to participate in the study described on the Information Sheet, and to the statements below. Make sure you've read and understood the Information Sheet before signing.

- "I am 18 years of age or over."
- "I have been informed of and understand the purposes of this study."
- "I have been given an opportunity to ask questions."
- "I understand that participation is entirely voluntary."
- "I understand that I can withdraw at any time without prejudice."
- "Any information which might potentially identify me will not be used in published material."

Name: _____

Signature: _____ Date: _____

Witness

Name: _____

Signature: _____ Date: _____

Figure C.1: The consent form signed by participants in the scenario study.

Information Sheet

Curtin ❖
University of Technology

Identifying and understanding software development artefact interrelationships

1 Aims

The purpose of this research is to gain insight into relationships between software development artefacts. In this context, such artefacts are documents containing textual and/or diagrammatic information about a software system.

Data collected will enable the researchers to examine and draw conclusions regarding:

- the nature of artefact relationships,

- patterns of artefact usage,

- artefact comprehension techniques, and

- means for locating defects in and inconsistencies amongst artefacts.

This supports the goal of improving software quality by proposing methods by which software engineers can more effectively and efficiently locate defects.

2 Participation

Participation in this study is entirely voluntary. You may withdraw at any time without prejudice — no reason needs to be given.

Students: this is not a test. This study is not related to any university assessment, and will have no effect on your ability to succeed in your course.

The study will take about 50 minutes of your time. It will involve reading and understanding several software development artefacts, potentially including diagrams and source code. A short questionnaire beforehand will be used to gauge your prior knowledge and experience, and another afterwards will ask your opinions of the exercise. During the exercise, you may be asked to *think aloud* — to put your thoughts into words. This, and the ways in which you make use of the artefacts, will be monitored.

There are no risks to you (beyond those encountered in everyday life), and you may benefit by gaining experience in reading Java source code and UML sequence diagrams, and a deeper understanding of how the two are related.

3 Data Storage and Publication

Only the principal researcher will be authorised to directly access data collected in this study.

You will not be identifiable from any material published as a result of this study. Your name and contact details will remain confidential. They will be collected only for purposes of verifying your consent to participate, contacting you if the researchers have any follow-up questions, and inviting you to participate in two related studies in the future.

The format of any published data will depend on the results of the study.

4 Contacts

If you have any queries regarding the study, please contact the researchers:

- David Cooper (PhD student) — cooperdj@cs.curtin.edu.au

- Dr Brian von Konsky (supervisor) — bvk@cs.curtin.edu.au

- Dr Mike Robey (supervisor) — mike@cs.curtin.edu.au

This study has been approved by the Curtin University Human Research Ethics Committee. The approval number is 'PHD JG 002/2006'. If needed, verification of approval can be obtained either by writing to the Curtin University Human Research Ethics Committee, c/- Office of Research and Development, Curtin University of Technology, GPO Box U1987, Perth, 6845 or by telephoning 9266 2784.

Figure C.2: The information sheet given to participants in the scenario study.

1. **Do you have, or are you currently studying for, a computing degree from Curtin University?**
   ☐ yes, currently enrolled   ☐ yes, graduated   ☐ no

   Degree:
   ☐ Computer Science   ☐ Information Technology   ☐ Software Engineering
   ☐ other: _____

   Year of commencement: _____

   Current year level:
   ☐ 1st year   ☐ 2nd year   ☐ 3rd year   ☐ honours or 4th year

2. **Do you have, or are you currently studying for, a computing degree from another institution?**
   ☐ yes, currently enrolled   ☐ yes, graduated   ☐ no

   Institution: _____

   Has your course of study covered Java?
   ☐ yes, in depth   ☐ yes, somewhat   ☐ not at all, or not to any significant extent

   Has your course of study covered UML?
   ☐ yes, in depth   ☐ yes, somewhat   ☐ not at all, or not to any significant extent

3. **Have you worked in the software industry?**
   ☐ yes, currently employed   ☐ yes, previously employed   ☐ no

   Years of experience: _____

Figure C.3: The first, demographic questionnaire filled out by participants in the scenario study. (The actual questionnaire used a web-based interface. This is a reproduction of the questions therein.)

1. **Rate your understanding:**

	very good	fairly good	fairly low	very low
(a) instructions	☐	☐	☐	☐
(b) sequence diagram	☐	☐	☐	☐
(c) source code	☐	☐	☐	☐

2. **Did you have difficulty keeping track of all the different classes?**
   ☐ it was reasonably easy   ☐ it was reasonably difficult   ☐ I was completely lost

3. **If the task had been done on paper rather than electronically, would you have found it easier or harder?**
   ☐ much easier   ☐ somewhat easier   ☐ about the same   ☐ somewhat harder
   ☐ much harder
   Why? _____

4. **Were you comfortable with the think aloud process?**
   ☐ yes   ☐ somewhat   ☐ not at all

5. **If you had been asked to type your thoughts at the keyboard rather than speak them out loud, would you have found it easier or harder?**
   ☐ much easier   ☐ somewhat easier   ☐ about the same   ☐ somewhat harder
   ☐ much harder

6. **Any comments on the think aloud process?**

   _____

7. **Was it easy or hard to locate an area of source code that roughly corresponded to the sequence diagram?**
   ☐ easy   ☐ moderately difficult   ☐ very difficult
   Why? _____

8. **Was it easy or hard to determine whether a given line of code should be marked?**
   ☐ easy   ☐ moderately difficult   ☐ very difficult
   Why? _____

9. **Was it easy or hard to determine the order in which lines of code were executed?**
   ☐ easy   ☐ moderately difficult   ☐ very difficult
   Why? _____

10. **Apart from anything already mentioned, did you encounter any ambiguities anywhere in the task?**

    _____

11. **Do you believe your participation has benefitted you?**
    ☐ not really   ☐ yes
    In what way(s)? _____

12. **General comments:** _____

Figure C.4: The second, opinion questionnaire filled out by participants in the scenario study. (The actual questionnaire used a web-based interface, like the first. This is a reproduction of the questions therein.)

```java
    {
        new UserInterface( ).menu( );
    }
}
```

C.2.2   PlayList.java

```java
package audioplayer;

import java.io.*;
import java.util.*;

public class PlayList
{
    private List tracks;

    public PlayList( )
    {
        tracks = new ArrayList( );
    }

    public PlayList( String filename ) throws IOException
    {
        BufferedReader reader =
            new BufferedReader( new FileReader( filename ));

        tracks = new ArrayList( );

        String trackFile = reader.readLine( );
        while( trackFile != null )
        {
            try
            {
                tracks.add( Track.loadTrack( trackFile ));
            }
            catch( TrackException e )
            { // Track couldn't be loaded - ignore it.
            }
            trackFile = reader.readLine( );
        }
    }

    public void save( String filename ) throws IOException
    {
```

```java
    PrintWriter writer = new PrintWriter( new FileWriter( filename ));
    int size = tracks.size( );
    for( int i = 0; i < size; i++ )
    {
        writer.println( (( Track )tracks.get( i )).getFilename( ) );
    }
    writer.close( );
}


public int getTrack( Track track )
{
    return tracks.indexOf( track );
}


public Track getTrack( int trackNum ) throws IndexOutOfBoundsException
{
    return ( Track )tracks.get( trackNum );
}


public int getTrackCount( )
{
    return tracks.size( );
}


public void addTrack( Track newTrack )
{
    if( !tracks.contains( newTrack ))
    {
        tracks.add( newTrack );
    }
}


public void removeTrack( Track track ) throws PlayListException
{
    if( !tracks.remove( track ))
    {
        throw new PlayListException( "No such track: \"" + track + "\"" );
    }
}


public void removeTrack( int track ) throws IndexOutOfBoundsException
{
    tracks.remove( track );
}
}
```

```java
class PlayListException extends Exception
{
    public PlayListException( String msg )
    {
        super( msg );
    }
}
```

C.2.3   Player.java

```java
package audioplayer;

import java.io.*;
import java.util.*;
import javax.sound.sampled.*;

public class Player
{
    private Programme programme = null;
    private Track currentTrack = null;
    private Clip clip = null;

    public void setProgramme( Programme programme )
    {
        this.programme = programme;
    }

    public void play( ) throws PlayerException, TrackException
    {
        if ( currentTrack == null )
        {
            nextTrack( );
        }

        play( currentTrack );
    }

    public void play( Track track ) throws PlayerException, TrackException
    {
        if( clip == null )
        {
            try
            {
                AudioInputStream input = track.getInputStream( );
```

```
                DataLine.Info info =
                    new DataLine.Info( Clip.class, input.getFormat( ));
                clip = ( Clip )AudioSystem.getLine( info );
                clip.open( input );
            }
            catch( Exception e )
            {
                throw new PlayerException( "Could not access sound device" );
            }
        }

        clip.start( );
    }

    public void pause( )
    {
        if( clip != null )
        {
            clip.stop( );
        }
    }

    public void rewind( )
    {
        if( clip != null )
        {
            clip.setFramePosition( 0 );
        }
    }

    public void previousTrack( ) throws PlayerException, TrackException
    {
        if( programme.hasPreviousTrack( ))
        {
            boolean wasPlaying = isPlaying( );
            if( wasPlaying )
            {
                clip.stop( );
            }

            clip = null;
            currentTrack = programme.getPreviousTrack( );

            if( wasPlaying )
            {
                play( );
```

```
                    }
                }
            }

    public void nextTrack( ) throws PlayerException, TrackException
    {
        if( programme.hasNextTrack( ))
        {
            boolean wasPlaying = isPlaying( );
            if( wasPlaying )
            {
                clip.stop( );
            }

            clip = null;
            currentTrack = programme.getNextTrack( );

            if( wasPlaying )
            {
                play( );
            }
        }
    }

    public boolean isPlaying( )
    {
        boolean active = false;
        if( clip != null )
        {
            active = clip.isActive( );
        }
        return active;
    }

    public Track getCurrentTrack( )
    {
        return currentTrack;
    }
}

class PlayerException extends Exception
{
    public PlayerException( String msg )
    {
        super( msg );
    }
```

```
}
```

C.2.4 Programme.java

```java
package audioplayer;

public class Programme
{
    protected PlayList playList;
    protected int currentTrack;

    public Programme( PlayList playList )
    {
        this.playList = playList;
        currentTrack = 0;
    }

    public boolean hasNextTrack( )
    {
        return currentTrack < playList.getTrackCount( ) - 1;
    }

    public boolean hasPreviousTrack( )
    {
        return currentTrack > 0;
    }

    public Track getNextTrack( )
    {
        currentTrack++;
        return playList.getTrack( currentTrack - 1 );
    }

    public Track getPreviousTrack( )
    {
        currentTrack--;
        return playList.getTrack( currentTrack );
    }
}
```

C.2.5   RandomProgramme.java

```java
package audioplayer;

import java.util.*;

public class RandomProgramme extends Programme
{
    private Track[] tracks;

    public RandomProgramme( PlayList playList )
    {
        super( playList );
        orderTracks( );
    }

    private void orderTracks( )
    {
        currentTrack = 0;
        tracks = new Track[ playList.getTrackCount( )];
        for( int t = 0; t < tracks.length; t++ )
        {
            tracks[t] = playList.getTrack( t );
        }
        Arrays.sort( tracks, new Randomiser( ));
    }

    public Track getNextTrack( )
    {
        if( tracks.length != playList.getTrackCount( ))
        {
            orderTracks( );
        }

        currentTrack++;
        return tracks[ currentTrack - 1 ];
    }

    public Track getPreviousTrack( )
    {
        if( tracks.length != playList.getTrackCount( ))
        {
            orderTracks( );
        }
```

```
            currentTrack--;
            return tracks[ currentTrack ];
        }
}


class Randomiser implements Comparator
{
    private Random randomNumGen = new Random( );

    public int compare( Object o1, Object o2 )
    {
        return randomNumGen.nextInt( 3 ) - 1;
    }
}
```

C.2.6   Track.java

```
package audioplayer;

import java.lang.reflect.*;
import java.util.*;
import javax.sound.sampled.*;

/**
 * Represents an abstract audio track. Subclasses of this class implement
 * functionality for reading and decoding audio files of a specific format.
 */
public abstract class Track
{
    /**
     * Instantiates the appropriate subclass (i.e. WAVTrack, MP3Track, etc.)
     * for the specified audio file.
     */
    public static Track loadTrack( String filename ) throws TrackException
    {
        int extensionIndex = filename.lastIndexOf( '.' );
        if( extensionIndex == -1 )
        {
            throw new TrackException( "Unknown format" );
        }

        String format =
            filename.substring( extensionIndex + 1 ).toUpperCase( );
        Track newTrack;
```

```
        if( format.equals( "wav" ))
        {
            newTrack = new WAVTrack( filename );
        }
        else
        {
            throw new TrackException( "Unsupported format" );
        }

        return newTrack;
    }

    private String filename;

    public Track( String filename )
    {
        this.filename = filename;
    }

    public String getFilename( )
    {
        return filename;
    }

    /** Two tracks are equal if their filenames match. */
    public boolean equals( Object obj )
    {
        boolean result = false;
        if( obj instanceof Track )
        {
            result = (( Track )obj ).filename.equals( filename );
        }
        return result;
    }

    public String toString( )
    {
        return filename;
    }

    /** Returns a stream from which uncompressed audio data can be read. */
    public abstract AudioInputStream getInputStream( ) throws TrackException;
}

class TrackException extends Exception
```

```
{
    public TrackException( String msg )
    {
        super( msg );
    }
}
```

C.2.7   UserInterface.java

```
package audioplayer;

import java.io.*;

public class UserInterface
{
    private static final String PLAY = "y";
    private static final String PAUSE = "p";
    private static final String STOP = "s";
    private static final String NEXT_TRACK = "n";
    private static final String PREVIOUS_TRACK = "r";
    private static final String RANDOM_ORDER = "a";
    private static final String NEW_PLAYLIST = "np";
    private static final String LOAD_PLAYLIST = "lp";
    private static final String SAVE_PLAYLIST = "sp";
    private static final String DISPLAY_PLAYLIST = "d";
    private static final String ADD_TO_PLAYLIST = "ap";
    private static final String REMOVE_FROM_PLAYLIST = "rp";
    private static final String EXIT = "x";

    private Player player;
    private PlayList playList;
    private Programme programme;

    public UserInterface( )
    {
        playList = new PlayList( );
        programme = new Programme( playList );
        player = new Player( );
        player.setProgramme( programme );
    }

    public void menu( )
    {
        BufferedReader console =
```

```
        new BufferedReader( new InputStreamReader( System.in ));
boolean done = false;

while( !done )
{
    String command;

    System.out.println( "Choose an option:" );
    System.out.println( "(" +
        PLAY + ") play, (" +
        PAUSE + ") pause, (" +
        STOP + ") stop" );
    System.out.println( "(" +
        NEXT_TRACK + ") next track, (" +
        PREVIOUS_TRACK + ") previous track, (" +
        RANDOM_ORDER + ") toggle random track order" );
    System.out.println( "(" +
        NEW_PLAYLIST + ") new playlist, (" +
        LOAD_PLAYLIST + ") load playlist, (" +
        SAVE_PLAYLIST + ") save playlist");
    System.out.println( "(" +
        ADD_TO_PLAYLIST + ") add to playlist, (" +
        REMOVE_FROM_PLAYLIST + ") delete from playlist, (" +
        DISPLAY_PLAYLIST + ") display playlist" );
    System.out.println( "(" + EXIT + ") exit" );
    System.out.print( ">> " );

    try
    {
        command = console.readLine( );

        if( command == null )
        {
            done = true;
        }
        else
        {
            command = command.toLowerCase( );
            if( command.equals( PLAY ))
            {
                player.play( );
                System.out.println( "Playing " +
                    player.getCurrentTrack( ).getFilename( ));
            }
            else if( command.equals( PAUSE ))
            {
```

```java
            if( player.isPlaying( ))
            {
                System.out.println( "Paused" );
                player.pause( );
            }
            else
            {
                player.play( );
            }
        }
        else if( command.equals( STOP ))
        {
            System.out.println( "Stopped" );
            player.pause( );
            player.rewind( );
        }
        else if( command.equals( NEXT_TRACK ))
        {
            player.nextTrack( );
            System.out.println( "Playing " +
                player.getCurrentTrack( ).getFilename( ));
        }
        else if( command.equals( PREVIOUS_TRACK ))
        {
            player.previousTrack( );
            System.out.println( "Playing " +
                player.getCurrentTrack( ).getFilename( ));
        }
        else if( command.equals( RANDOM_ORDER ))
        {
            toggleRandomOrder( );
        }
        else if( command.equals( NEW_PLAYLIST ))
        {
            newPlayList( null );
        }
        else if( command.equals( LOAD_PLAYLIST ))
        {
            newPlayList( getFilename( console ));
            displayPlayList( );
        }
        else if( command.equals( SAVE_PLAYLIST ))
        {
            savePlayList( getFilename( console ));
        }
        else if( command.equals( ADD_TO_PLAYLIST ))
```

```java
            {
                playList.addTrack(
                    Track.loadTrack( getFilename( console )));
            }
            else if( command.equals( REMOVE_FROM_PLAYLIST ))
            {
                displayPlayList( );
                removeFromPlayList( console );
            }
            else if( command.equals( DISPLAY_PLAYLIST ))
            {
                displayPlayList( );
            }
            else if( command.equals( EXIT ))
            {
                done = true;
            }
            else
            {
                System.out.println( "Unknown command" );
            }
        }
    }
    catch( IOException e )
    {
        done = true;
    }
    catch( PlayerException e )
    {
        System.out.println( e.getMessage( ));
    }
    catch( TrackException e )
    {
        System.out.println( e.getMessage( ));
    }

    System.out.println( );
    }
}

private void toggleRandomOrder( )
{
    if( programme instanceof RandomProgramme )
    {
        System.out.println( "Using sequential programme" );
        programme = new Programme( playList );
```

```
        }
        else
        {
            System.out.println( "Using random programme" );
            programme = new RandomProgramme( playList );
        }
        player.setProgramme( programme );
    }


    private void displayPlayList( )
    {
        int size = playList.getTrackCount( );
        System.out.println( "Play list:" );
        for( int i = 0; i < size; i++ )
        {
            System.out.println(i + ": " + playList.getTrack(i).getFilename());
        }
    }


    private void removeFromPlayList(BufferedReader console) throws IOException
    {
        System.out.println( );
        System.out.print( "Enter number of track to remove: " );

        try
        {
            playList.removeTrack( Integer.parseInt( console.readLine( )));
        }
        catch( NumberFormatException e )
        {
            System.out.println( "Not a valid track number" );
        }
        catch( IndexOutOfBoundsException e )
        {
            System.out.println( "No such track" );
        }
    }


    private void newPlayList( String filename )
    {
        player.pause( );
        try
        {
            if( filename == null )
            {
                playList = new PlayList( );
```

```
            }
            else
            {
                playList = new PlayList( filename );
            }

            programme = new Programme( playList );
            player.setProgramme( programme );
        }
        catch( IOException e )
        {
            System.out.println("Could not load play list: " + e.getMessage());
        }
    }


    private void savePlayList( String filename )
    {
        try
        {
            playList.save( filename );
        }
        catch( IOException e )
        {
            System.out.println("Could not save play list: " + e.getMessage());
        }
    }

    private String getFilename( BufferedReader console ) throws IOException
    {
        System.out.println( );
        System.out.print( "Enter filename: " );
        return console.readLine( );
    }
}
```

C.2.8   WAVTrack.java

```
package audioplayer;

import java.io.*;
import javax.sound.sampled.*;

/**
 * An audio track in WAVE format.
```

```
 */
public class WAVTrack extends Track
{
    public WAVTrack( String filename )
    {
        super( filename );
    }


    /** Returns a stream from which uncompressed audio data can be read. */
    public AudioInputStream getInputStream( ) throws TrackException
    {
        String filename = getFilename( );
        try
        {
            return AudioSystem.getAudioInputStream( new File( filename ));
        }
        catch( UnsupportedAudioFileException e )
        {
            throw new TrackException( "Unsupported format" );
        }
        catch( IOException e )
        {
            throw new TrackException( "Could not read " + filename );
        }
    }
}
```

Appendix D

Checklist Experiment — Materials

This appendix complements the description and analysis given in Chapter 7.

D.1 Forms and Sheets

Figure D.1 shows the consent form signed by participants. Figure D.2 shows the information/instruction sheet presented to participants before the exercise. Figure D.3 shows the questionnaire filled out by particiants afterwards.

D.2 Training Snippets

This section contains the code and associated materials used in training. Each of the two training systems has four potential defects, of which only two were seen by each participant. Thus, both the defective and corrected code are shown.

The highlighting reproduced here was shown to participants upon completion of each inspection activity.

Consent Form (Inspection Experiment)

By signing this form, you consent to participate in this study and agree that:

- You are 18 years or over.
- You have been informed of and understand the purposes of this study.
- You have been given an opportunity to ask questions.

Participation is entirely voluntary, and you can withdraw at any time.

Name: _____

Signature: _____ Date: _____

Figure D.1: The consent form signed by participants in the checklist experiment.

Instructions to participants

Note: you must read and sign the Consent Form before you participate in this study.

- You will be presented with six mini-systems, one after another, each with its own specification. Each system is written in Java and contains **exactly two** defects. (For our purposes, a defect is any deviation from the specification.)

- Find both defects in each mini-system and concisely describe what is wrong. You don't need to fix them.

- Whether you need to read all the material is up to you.

- You may be provided with a defect checklist for some or all of the mini-systems. If so, make use of it.

- Don't worry if you can't find one or both defects. Just move on to the next system.

Figure D.2: The information/instruction sheet shown to participants in the checklist experiment (via a web-based interface).

Prior Knowledge and Experience

Please indicate the computing-related degree you're studying for, or already have.
☐ Computer Science   ☐ Information Technology   ☐ Software Engineering   ☐ None

☐ Other computing-related degree: _____

At what stage of the degree are you currently at?
☐ 1st year   ☐ 2nd year   ☐ 3rd year   ☐ Honours, 4th year or graduated

Have you worked as a software developer before, and if so for how long?
☐ no
☐ yes, for _____ years

Which of the following have you done before? (tick as many as appropriate)
☐ Participated in a software inspection/review
☐ Studied software inspections/reviews
☐ Read and understood real-world source code not written by you

Your Opinion

Did the checklist generally make the task easier or harder?
☐ Much harder
☐ Somewhat harder
☐ No difference
☐ Somewhat easier
☐ Much easier

How did you use the checklist?
☐ I basically ignored it
☐ I looked over it
☐ I checked each item carefully, but didn't rely on it to find all the defects
☐ I used it exclusively

You generally found defects...
☐ Before consulting the checklist
☐ As a result of consulting the checklist
☐ While consulting the checklist (but not as a result)
☐ After consulting the checklist (but not as a result)

You may have noticed that some defects in different snippets were very similar. Did your familiarity with those types of defects make the task any easier?
☐ No   ☐ Somewhat easier   ☐ Much easier

Figure D.3: The questionnaire filled out by participants in the checklist experiment.

This is the specification for a Java class.

The class must calculate acceleration due to the Earth's gravity. The acceleration ("a") of an object when falling towards the Earth (ignoring friction) depends on three values:

- the object's distance from the centre of the Earth ("r"),
- the Earth's mass ("M"), and
- the gravitational constant ("G").

To calculate acceleration, the formula is: $a = GM/r^2$. Since M and G do not change, they can be dealt with together as a single value: $398600441800000\mathrm{m}^3\mathrm{s}^{-2}$.

The class must have methods to (a) store the distances of several objects above the Earth's surface, and (b) return an HTML table containing the results of the calculations. For each object, the table returned should contain a row showing the name of each object (or an empty cell if the name is null), its distance and its precise acceleration due to the Earth's gravity. If the characters "&", "<" and/or ">" occur in an object's name, they must be removed or replaced (or else the HTML string would be invalid).

Figure D.4: The specification for the Gravity training snippet.

Table D.1: The Gravity defect descriptions (two of which were shown to each participant after inspection).

Label	Description
Integer division defect	In the method `calculateAccel()`, `EARTH_MASS_TIMES_G` and `distance` are both integers. The division is therefore integer division, and so the resulting value is imprecise.
Special characters defect	In the method `getHtml()`, no attempt is made to replace the characters &, < and > in `labels[i]` with their proper HTML representations: `&`, `<` and `>`. If any of the labels contain such characters, the HTML returned will be invalid.
Null values defect	In the method `getHtml()`, `labels.get(i)` might be null, in which case errors arise. When added to a string with the + operator, `null` is converted to the string "null", not the empty string "". If used to call a method, it results in a `NullPointerException`.
Overwritten values defect	In the method `addReading()`, the supplied values incorrectly overwrite the previous values, rather than being appended to the end of the three `Vectors`.

D.2.1 Gravity

Figure D.4 shows the specification for the Gravity training system, and Table D.1 shows the defect descriptions. The source code is as follows:

```java
import java.util.*;

public class Gravity
{
```

— Integer division defect —
```java
    public static final long EARTH_MASS_TIMES_G = 3986004418000001l;
        // metres^3 seconds^-2
```

— Corrected code —
```java
    public static final double EARTH_MASS_TIMES_G = 398600441800000.0;
        // metres^3 seconds^-2
```

```java
    public static final long EARTH_RADIUS = 6372797; // metres

    private Vector labels = new Vector();
    private Vector distances = new Vector();
    private Vector accelerations = new Vector();

public Gravity() {}

    public void addReading( String label, long distance )
    {
```

— Overwritten values defect —
```java
        int index = this.labels.size() - 1;
        labels.set( index, label );

        // Distances and accelerations must be wrapped inside a 'Long'
        // object, because primitive types cannot be stored in a Vector.
        distances.set( index, new Long( distance ) );
        accelerations.set( index, new Double( calculateAccel( distance ) ) );
```

— Corrected code —
```java
        labels.add( label );

        // Distances and accelerations must be wrapped inside a "Long"
        // object, because primitive types cannot be stored in a Vector.
        distances.add( new Long(distance) );
        accelerations.add( new Double( calculateAccel( distance ) ) );
```

```java
    }

    /**
```

```
 * Calculates and returns an object's acceleration due to gravity, given
 * its distance above the Earth's surface.
 */
private static double calculateAccel( highlightif(d1, long) distance )
{
    distance += EARTH_RADIUS;
```

Integer division defect
```
    return EARTH_MASS_TIMES_G / ( distance * distance );
```

Corrected code
```
    return EARTH_MASS_TIMES_G / (double)( distance * distance );
```

```
}


public String getHtml()
{
    // HTML for the table headings.
    String html =
        "<table><tr><th></th><th>Distance (m)</th>" +
        "<th>Acceleration (m/s^2)</th></tr>";

    for( int i = 0; i < labels.size(); i++ )
    {
```

Null values defect
```
    (Missing code.)
```

Corrected code
```
        String label = (String)labels.get(i);
        if( label == null )
        {
            label = "";
        }
```

Special characters defect
```
    (Missing code.)
```

Corrected code
```
        // Replace all special HTML characters (&, < and >) in the label.
        label = label
            .replaceAll( "&", "&amp;" )
            .replaceAll( "<", "&lt;" )
            .replaceAll( ">", "&gt;" );
```

```
        // One table row for each object.
        html += "<tr><td>" +
            label + "</td><td>" +
            distances.get(i) + " m</td><td>" +
            accelerations.get(i) + "</td></tr>";
    }
```

This is the specification for a single Java method.

The method must calculate Body Mass Index (BMI), correct to <mark>at least one decimal place</mark>, for people in a specified country. BMI can be determined from a person's weight ("w") in kilograms and height ("h") in metres, according to the following formula: $bmi = w/h^2$.

Weight and height records must be retrieved from a database using SQL. Heights are stored in centimetres, and so must be converted. <mark>For security reasons any data supplied in the SQL query should be escaped.</mark>

<mark>The calculated list of values must be returned.</mark> <mark>If no records are available for the specified country, an empty list should be returned.</mark>

Figure D.5: The specification for the BMI training snippet.

Table D.2: The BMI defect descriptions (two of which were shown to each participant after inspection).

Label	Description
Integer division defect	To convert height from centimetres to metres, it is divided by 100. Unfortunately this is an integer division, where (for example) 180 / 100 == 1. As a result the calculation will assume that most people are only a metre tall and will thus grossly overestimate their BMI. The value is converted to a `double`, but only after the error has occurred.
Special characters defect	When `country` is embedded in the SQL query, no special characters are escaped. A malicious user could construct a `country` string containing such characters, allowing them to execute any SQL commands and possibly destroy the entire database ("SQL injection").
Null values defect	The `Database.query()` method will return `null` if no records were found. However, the `calculateBmi()` method does not check for such values. If `query()` returned `null`, `calculateBmi()` would throw a `NullPointerException`.
Overwritten values defect	Each successive BMI value calculated overwrites the last value in `bmiVector` rather than being added to the end. This will produce an exception for the first value every time, since `bmiVector` is initially empty.

```
        return html + "</table>";
    }
}
```

D.2.2 BMI

Figure D.5 shows the specification for the BMI training system, and Table D.2 shows the defect descriptions. The source code is as follows:

```
public Vector calculateBmi( Database db, String country )
{
    String[][] records;
```

```
Vector bmiVector = new Vector();

// A null value for country
if( country == null )
{
    country = "Australia";
}

// Retrieve all relevant records from the database.
records = db.query(
    "SELECT weight, height FROM people WHERE country = '" +
```

────── Special characters defect ──────
```
    country + "'" );
```

────── Corrected code ──────
```
    db.escape( country ) + "'" );
```

────── Null values defect ──────
```
 (Missing code.)
```

────── Corrected code ──────
```
 if( records == null )
 {
     records = new String[0][];
 }
```

```
for( int i = 0; i < records.length; i++ )
{
    // Each record consists of two integers (represented as Strings) for
    // weight (in kg) and height (in cm). Height must be converted to
    // metres, as a double. BMI can then be calculated.
    int weight = Integer.parseInt( records[[i][0]] );
```

────── Integer division defect ──────
```
    double height = Integer.parseInt( records[i][1] ) / 100;
```

────── Corrected code ──────
```
    double height = (double)Integer.parseInt( records[i][1] ) / 100.0;
```

```
    double bmi = (double)weight / ( height * height );

    // Add the BMI value to the return vector.
```

────── Overwritten values defect ──────
```
    bmiVector.set( bmiVector.size() - 1, new Double( bmi ));
```

────── Corrected code ──────
```
    bmiVector.add( new Double( bmi ));
```

```
}
```

Table D.3: The SlushFund defect descriptions (shown to participants after inspection).

Label	Description
Integer division defect	The calculation of the distribution amount for each person involves an integer division rather than floating-point division. As a result, most people are likely to be underpaid by varying amounts.
Output format defect	Although the specification requires the output to be of the form "[name]: [amount]", the code actually outputs "[amount]: [name]".

```
    return bmiVector;
}


// ---

abstract class Database
{
    /**
     * Perform an SQL query on the database. If no records are available as a
     * result of the query, this method will return null. Otherwise, it will
     * return a matrix of Strings. Each row in the matrix will be a record,
     * and each column will be a field.
     */
    public abstract String[][] query( String sql );

    /**
     * Returns an escaped version of a string, which can be safely embedded
     * in an SQL query.
     */
    public abstract String escape( String data );
}
```

D.3   Test Snippets

This section contains the code and associated materials used to generate experimental data. Here, the same defects are seen by all participants.

D.3.1   SlushFund

Figure D.6 shows the specification for the SlushFund testing system, Figure D.7 shows the defect checklist, and Table D.3 shows the defect descriptions. The source code is as follows:

This is the specification for a single Java method.

The method must determine how to distribute a slush fund among a group of people of varying importance. Each person must receive money in proportion to their importance (e.g. if person A is twice as important as person B, person A should receive twice the money). All the money must be used up.

Therefore, the relative proportion of the money each person gets (between 0 and 1) is equal to their importance divided by the sum of the importance values for all the people. Multiplying the result by the total amount in the fund yields the actual amount of money the person receives.

The amount of money in the fund and the list of people are both pre-initialised by another method (which is outside the scope of this specification).

The method must output the results to a given file, with one line for each person. Each line should be in the form "name: amount", where name is the person's name and amount is precise amount of money that person receives.

Figure D.6: The specification for the SlushFund test snippet.

- When a file is written to, is it closed afterwards?
- For division where precision is required, are integers first type-cast to floats or doubles?

Figure D.7: The SlushFund defect checklist, shown to half the participants.

```java
private int amount;
private Person[] people;


public void outputSlushFund( String filename ) throws IOException
{
    PrintWriter writer = new PrintWriter( new FileWriter( filename ));

    // Calculate the total 'importance'
    int totalImportance = 0;
    for( int i = 0; i < people.length; i++ )
    {
        totalImportance += people[i].getImportance();
    }

    // Each person's share of the money is calculated in accordance with
    // their importance.
    for( int i = 0; i < people.length; i++ )
    {
```

Integer division defect
```java
        double distribution =
            people[i].getImportance() / totalImportance * amount;
```

Output format defect
```java
        writer.println( distribution + ": " + people[i].getName() );
```

```java

    }
```

This is the specification for an additional feature of an existing class: TreeNode. TreeNode represents one node of a generic n-ary tree structure. It contains a label (a string), a value (an arbitrary object) and an array of child nodes.

It is now required that TreeNode be able to return an XML representation of itself. This should be of the form "`<node value="`*value*`">`*child nodes*`</node>`". *Value* is the string representation of the node's value object, and *child nodes* are the XML representations of the node's children. ==Any special characters in *value* (including quotes, ampersands and greater-than/less-than characters) must be replaced by their XML representations.==

However, the output must be conditional upon the node's label ==matching a given string, case insensitive==. If it does not, the empty string must be returned instead. Likewise, if any child node's label does not match, that child node must be omitted from the output.

Figure D.8: The specification for the TreeNode test snippet.

- Where recursion occurs, is there a mechanism to avoid infinite recursion?
- When generating strings in a particular language, are all special characters (e.g. quotes, depending on the language) removed or escaped?

Figure D.9: The TreeNode defect checklist, shown to half the participants.

```
    writer.close();
}


// ---


abstract class Person
{
    public abstract String getName();
    public abstract int getImportance();
}
```

Table D.4: The TreeNode defect descriptions (shown to participants after inspection).

Label	Description
Special characters defect	Where `value.toString()` is embedded in the XML string, the code does not replace any special characters (&, < or >) that may happen to be in the value's string representation. Consequently, if these characters are present they will render the XML invalid.
Case insensitivity defect	The specification requires that string matching be case-insensitive, but this requirement is not implemented by the method. For example, a node with the label "Apple" will incorrectly be omitted when the method is called with the label "apple".

D.3.2   TreeNode

Figure D.8 shows the specification for the TreeNode testing system, Figure D.9 shows the defect checklist, and Table D.4 shows the defect descriptions. The source code is as follows:

```
public class TreeNode
{
    private String label;
    private Object value;
    private TreeNode[] children;

    // ...

    public String toXml( String label )
    {
        String xml = "";

        // Only generate XML if this node's label matches the specified one.
```

———————— Case insensitivity defect ————————
```
        if( this.label.equals( label ) )
        {
```

———————— Special characters defect ————————
```
            xml = "<node value=\"" + value.toString() + "\">\n";
```

```
            for( int i = 0; i < children.length; i++ )
            {
                // Recurse to generate XML for child nodes.
                xml += children[i].toXml( label );
            }
            xml += "</node>\n";
        }

        return xml;
    }
}
```

D.3.3   AddressSearch

Figure D.10 shows the specification for the AddressSearch testing system, Figure D.11 shows the checklist, and Table D.5 shows the defect descriptions. The source code is as follows:

> This is the specification for a single Java method.
>
> The method must implement a search feature for an address book application. The address book's design allows a flexible number of fields in each entry. Each field has a name (e.g. "name", "phone", "email", etc.) and a value, both strings.
>
> The method will be supplied with a search string and a list of fields. It must find and return all entries that <mark>contain at least one of the specified fields</mark> that exactly matches the search string.

Figure D.10: The specification for the AddressSearch test snippet.

> - Are the return types for methods appropriate given the requirements?
> - Where `null` is an acceptable (useful/meaningful) value for a variable, is there a check to ensure that `variable.method()` (or similar) is *not* called when the variable is `null`?

Figure D.11: The AddressSearch defect checklist, shown to half the participants.

```
private Vector addressBook;


/**
 * Searches the address book for a given search string and returns a Vector
 * of all matching entries.
 */
private Vector search( String searchString, String[] fieldNames )
{
    Vector result = new Vector();

    for( int entryNum = 0; entryNum < addressBook.size(); entryNum++ )
    {
        Entry entry = (Entry)( addressBook.get( entryNum ));
```
———————————————— Search algorithm defect ————————————————
```
        boolean matched = true;


        // For each entry in the address book, test each field to see if it
        // matches the search string.
        for( int f = 0; f < fieldNames.length; f++ )
        {
```
———————————————— Null values defect ————————————————
```
            if( !entry.getField( fieldNames[f] ).equals( searchString ))
```

Table D.5: The AddressSearch defect descriptions (shown to participants after inspection).

Label	Description
Null values defect	No check is made to ensure a field is non-`null` before trying to compare it to the search string. The `search()` method will throw a `NullPointerException` if any entry in the address book does not contain one of the specified fields.
Search algorithm defect	The algorithm requires every field in an entry to match the search string, if that entry is to be included in the results. This violates the specification, which states that at least one field must match.

This is the specification for two methods in a Java class.

The class is part of a 3D, first-person shooter game, in which the player takes the role of a hero who must complete a mission with a variety of hand-held weaponry. The class itself handles the acquisition and selection of the player's weapons. The player can acquire new weapons within the game, and can have one weapon "selected" (ready for use) at any given time. The class must contain methods to allow a weapon to be acquired, and to allow the next weapon to be selected.

A weapon can only be acquired if the player does not already have it. When any new weapon is acquired, it must be automatically selected.

A player's available weapons should be arranged in a list. The "select next weapon" option must move the selection to the next weapon in the list, or to the first weapon if the last is selected. However, weapons with no ammunition left must be automatically skipped. If no weapons are left with ammunition, the option should not have any effect.

Figure D.12: The specification for the WeaponSelector test snippet.

```
——————————— Search algorithm defect ———————————
            {
                matched = false;
            }
```

```
        }


        // If the field matches, add it to the results list.
        if( matched )
        {
            result.add( entry );
        }
    }

    return result;
}


// ---

abstract class Entry
{
    /**
     * Retrieves a specified field from the address book entry. If the entry
     * does not contain the specified field, null is returned instead.
     */
    public abstract String getField( String fieldName );
}
```

> • For loops, is the appropriate construct (i.e. `for`, `while` or `do...while`) used?
> • When a data structure (e.g. a `Vector`) is being manipulated, is the appropriate operation being performed?

Figure D.13: The WeaponSelector defect checklist, shown to half the participants.

Table D.6: The WeaponSelector defect descriptions (shown to participants after inspection).

Label	Description
Overwritten values defect	The `addWeapon()` method overwrites the last element in `weapons` with `newWeapon`, rather than appending it to the end.
Bounds checking defect	The `selectNextWeapon()` method allows `currentWeapon` to run off the end of the `weapons` list, causing an exception to be thrown.

D.3.4 WeaponSelector

Figure D.12 shows the specification for the WeaponSelector testing system, Figure D.13 shows the checklist, and Table D.6 shows the defect descriptions. The source code is as follows:

```java
import java.util.*;

public class WeaponSelector
{
    private Vector weapons = new Vector();
    private int currentWeapon = 0;

    // ...

    /** Adds a new weapon and selects it. */
    public void addWeapon( Weapon newWeapon )
    {
        if( !weapons.contains( newWeapon ))
        {
```
─────────────── Overwritten values defect ───────────────
```java
            weapons.set( weapons.size() - 1, newWeapon );


            currentWeapon = weapons.size() - 1;
        }
    }

    /** Selects the next available weapon that has ammunition. */
    public void selectNextWeapon()
    {
```
─────────────── Bounds checking defect ───────────────
```java
        do
        {
```

```
                    currentWeapon++;
            }
        while( !((Weapon) weapons.get( currentWeapon )).hasAmmunition() );
```

```
    }
}


// ---

abstract class Weapon
{
    public abstract boolean hasAmmunition();
}
```

Appendix E

Inspection Modelling — Equations and Inputs

This appendix contains the formal set of equations comprising the inspection model described in Chapter 8. The set of model inputs used in evaluating the model is also given.

E.1   Metamodel

E.1.1   Entities

$$Q_0 = \left\{ \varepsilon^{(e,\eta,0,0)} : e \in \mathcal{E}, 1 \leq \eta \leq \gamma_{0e} \right\} \tag{E.1}$$

$$Q_{\mathrm{H}j}(\varepsilon) = \begin{cases} \emptyset & \text{if } \tau(\varepsilon) \in \mathcal{E}_{\mathrm{D}} \\ \left\{ \varepsilon'^{(e,\eta,\mathbf{H},\varepsilon)} : e \in \mathcal{E}, 1 \leq \eta \leq \gamma_{\mathrm{H}je\tau(\varepsilon)}^{(\varepsilon)} \right\} & \text{otherwise} \end{cases} \tag{E.2}$$

$$Q_{\mathrm{G}j}(\varepsilon) = \begin{cases} \emptyset & \text{if } j = 0 \\ Q_{\mathrm{GE}j}(\varepsilon) & \text{if } j \geq 1, \tau(\varepsilon) \in \mathcal{E}_{\mathrm{D}} \\ Q_{\mathrm{GD}j}(\varepsilon) & \text{if } j \geq 1, \tau(\varepsilon) \notin \mathcal{E}_{\mathrm{D}} \end{cases} \tag{E.3}$$

Table E.1: Symbols for entities, entity types and their sets and identifying characteristics.

Symbol	Description
e	An entity type.
ε	An entity.
λ	A locality.
κ	A k-instance.
δ	A defect.
ψ	A marker.
j	A development phase ($\mathbb{N}$).
$\mathcal{E}$	The set of all entity types.
$\mathcal{E}_{\mathrm{D}}$, $\mathcal{E}_{\mathrm{K}}$, $\mathcal{E}_{\mathrm{L}}$	The set of defect types, k-instance types and locality types.
Ψ	The set of markers.
J	The number of development phases.
$\varepsilon^{(e,\eta,h,\varepsilon')}$	An entity construction; the η'th entity of type e derived via method h from entity ε'.
$\delta^{(e,\eta,h,\varepsilon',w,y)}$	A defect construction; the η'th defect of type e derived via method h from entity ε' having rework status w and carryover status y.
η	An arbitrary index
$e = \tau(\varepsilon)$	The type of entity ε.
$e' = \pi(\varepsilon)$	The originating entity from which ε was derived, or 0 if ε is an initial entity.
$h = H(\varepsilon) \in \{0,\mathtt{G},\mathtt{H}\}$	The method by which ε was derived; 0 indicating an initial entity, $\mathtt{G}$ indicating propagation and $\mathtt{H}$ indicating hierarchy.
$w = W(\delta)$	The rework status of defect δ; 1 iff δ resulted from rework.
$y = Y(\delta)$	The carryover status of defect δ; 1 iff δ is a carryover defect.
E_j	The set of entities in phase j.
$E_{\mathrm{L}j}$	The set of localities in phase j.
$E_{\mathrm{K}j}$	The set of k-instances in phase j.
$E_{\mathrm{D}j}$	The set of (potential) defects in phase j.
$P_j(\varepsilon)$	The dependencies of entity ε.
$P_{\mathrm{M}j}(\kappa)$	The comprehension dependencies of k-instance κ.
$P_{\mathrm{L}j}(\kappa)$	The locality dependencies of k-instance κ.
$P_{\mathrm{D}j}(\lambda)$	The decision dependencies of locality λ.
$X_{\mathrm{K}j\varepsilon\psi}$	Indicates whether marker ψ is assigned to entity ε in phase j.

Table E.2: Intermediate entity sets used to build the metamodel structure.

Symbol	Description
Q_0	The set of initial entities.
$Q_{\mathrm{H}j}(\varepsilon)$	The set of entities in phase j hierarchically derived from ε, directly.
$Q_{\mathrm{GE}j}(\varepsilon)$	The set of entities in phase j propagated from non-defect entity ε.
$Q_{\mathrm{GD}j}(\varepsilon)$	The set of entities in phase j propagated from defect δ.
$Q_{\mathrm{G}j}(\varepsilon)$	The total set of entities in phase j propagated from ε.
$N_{\mathrm{H}j}(\varepsilon)$	The set of entities in phase j hierarchically derived from ε, directly or indirectly.
$N_{\mathrm{G}j}$	The set of entities in phase j resulting from propagation, or the initial entities.
$P_0(\varepsilon)$	The initial dependencies of ε, if applicable.
$P_{\mathrm{G}j}(\varepsilon)$	The dependencies of ε resulting from propagation.
$P_{\mathrm{H}j}(\varepsilon)$	The number of initial and hierarchically-derived dependencies of ε.

Table E.3: Metamodel variates.

Variate	Variable	Description
γ_{0e}	Γ_{0e}	The number of initial entities of type e.
$\gamma^{(\varepsilon)}_{\mathrm{H}je\tau(\varepsilon)}$	$\Gamma_{\mathrm{H}je\tau(\varepsilon)}$	The number of entities of type e hierarchically derived from ε in phase j.
$\gamma^{(\varepsilon)}_{\mathrm{GE}je\tau(\varepsilon)}$	$\Gamma_{\mathrm{GE}je\tau(\varepsilon)}$	The number of entities of type e in phase j propagating from non-defect entity ε.
$\gamma^{(\delta)}_{\mathrm{GD}je\tau(\delta)w}$	$\Gamma_{\mathrm{GD}je\tau(\delta)w}$	The number of defects of type e in phase j with rework status w propagating from defect δ.
$\sigma^{(\delta)}_{j\tau(\delta)}$	$\Sigma_{j\tau(\delta)}$	Indicates whether an attempt to rework δ in phase j will fail.
$\xi^{(\varepsilon)}_{0\tau(\varepsilon)e}$	$\Xi_{0\tau(\varepsilon)e}$	The number of initial dependencies ε has on type-e entities.
$\xi^{(\varepsilon)}_{\mathrm{G}j\tau(\varepsilon)\tau(\varepsilon')}$	$\Xi_{\mathrm{G}j\tau(\varepsilon)\tau(\varepsilon')}$	Indicates whether ε has a propagated dependency on ε', given that one can exist.
$\xi^{(\varepsilon,\varepsilon'')}_{\mathrm{H}\tau(\varepsilon)e\tau(\varepsilon'')}$	$\Xi_{\mathrm{H}\tau(\varepsilon)e\tau(\varepsilon'')}$	The number of localised/hierarchical dependencies ε has on type-e entities within the context of ε''.
$\mathrm{select}[n,S]^{(\varepsilon)}$	$\mathrm{select}[n,S]$	A randomly-chosen size-n subset of the set S associated with entity ε, chosen with uniform probability.
$\theta^{(\varepsilon)}_{\tau(\varepsilon)}$	$\Theta_{\tau(\varepsilon)}$	The dependence complexity of entity ε.
$\omega^{(\lambda)}_{\tau(\lambda)\psi}$	$\Omega_{\tau(\lambda)\psi}$	Indicates whether non-propagated locality λ is assigned marker ψ.

$$Q_{\mathrm{GE}j}(\varepsilon) = \left\{ \varepsilon'^{(e,\eta,\mathtt{G},\varepsilon)} : e \in \mathcal{E}, 1 \leq \eta \leq \gamma^{(\varepsilon)}_{\mathrm{GE}je\tau(\varepsilon)} \right\} \qquad j \geq 1, \tau(\varepsilon) \notin \mathcal{E}_{\mathrm{D}}$$
(E.4)

$$Q_{\mathrm{GD}j}(\delta) = \left\{ \delta'^{(e,\eta,\mathtt{G},\delta,w,0)} : e \in \mathcal{E}_{\mathrm{D}}, w \in \{0,1\}, 1 \leq \eta \leq \gamma^{(\delta)}_{\mathrm{GD}je\tau(\delta)w} \right\}$$
$$\cup \left\{ \delta'^{(\tau(\delta),0,\mathtt{G},\delta,w,1)} : w \leq \sigma^{(\delta)}_{j\tau(\delta)} \right\} \qquad j \geq 1, \tau(\delta) \in \mathcal{E}_{\mathrm{D}}$$
(E.5)

$$N_{\mathrm{H}j}(\varepsilon) = \{\varepsilon\} \cup \bigcup_{\varepsilon' \in Q_{\mathrm{H}j}(\varepsilon)} N_{\mathrm{H}j}(\varepsilon')$$
(E.6)

$$N_{\mathrm{G}j} = \begin{cases} Q_0 & \text{if } j = 0 \\ \displaystyle\bigcup_{\varepsilon \in E_{j-1}} Q_{\mathrm{G}(j-1)}(\varepsilon) & \text{if } j \geq 1 \end{cases}$$
(E.7)

$$E_j = \bigcup_{\varepsilon \in N_{\mathrm{G}j}} N_{\mathrm{H}j}(\varepsilon)$$
(E.8)

$$E_{\mathrm{D}j} = \{\delta \in E_j : \tau(\delta) \in \mathcal{E}_{\mathrm{D}}\} \tag{E.9}$$

$$E_{\mathrm{K}j} = \{\kappa \in E_j : \tau(\kappa) \in \mathcal{E}_{\mathrm{K}}\} \tag{E.10}$$

$$E_{\mathrm{L}j} = \{\lambda \in E_j : \tau(\lambda) \in \mathcal{E}_{\mathrm{L}}\} \tag{E.11}$$

E.1.2 Dependencies

$$P_{00}(\varepsilon) = \bigcup_{e \in \mathcal{E}} \mathrm{select}\left[\xi_{0\tau(\varepsilon)e}^{(\varepsilon)}, \left\{\varepsilon' : \tau(\varepsilon') = e,\ \theta_{\tau(\varepsilon)}^{(\varepsilon)} > \theta_{\tau(\varepsilon')}^{(\varepsilon')}\right\}\right]^{(\varepsilon)} \tag{E.12}$$

$$P_{\mathrm{G}j}(\varepsilon) = \left\{\varepsilon' : \pi(\varepsilon') \in P_{j-1}\left[\pi(\varepsilon)\right], \xi_{\mathrm{G}j\tau(\varepsilon)\tau(\varepsilon')}^{(\varepsilon)} = 1\right\} \tag{E.13}$$

$$P_{\mathrm{H}j}(\varepsilon) = \bigcup_{\{\varepsilon'' : \varepsilon \in N_{\mathrm{H}j}(\varepsilon'')\}}\ \bigcup_{e \in \mathcal{E}} \tag{E.14}$$

$$\mathrm{select}\left[\xi_{\mathrm{H}\tau(\varepsilon)e\tau(\varepsilon'')}^{(\varepsilon,\varepsilon'')}, \left\{\varepsilon' \in N_{\mathrm{H}j}(\varepsilon'') : \tau(\varepsilon') = e,\ \theta_{\tau(\varepsilon)}^{(\varepsilon)} > \theta_{\tau(\varepsilon')}^{(\varepsilon')}\right\}\right]^{(\varepsilon)} \tag{E.15}$$

$$P_j(\varepsilon) = \begin{cases} P_0(\varepsilon) \cup P_{\mathrm{H}0}(\varepsilon) & \text{if } H(\varepsilon) = 0 \\ P_{\mathrm{G}j}(\varepsilon) & \text{if } H(\varepsilon) = \mathtt{G} \\ P_{\mathrm{H}j}(\varepsilon) & \text{if } H(\varepsilon) = \mathtt{H} \end{cases} \tag{E.16}$$

$$P_{\mathrm{M}j}(\kappa) = \left\{\kappa' \in P_j(\kappa) : \tau(\kappa') \in \mathcal{E}_{\mathrm{K}}\right\} \tag{E.17}$$

$$P_{\mathrm{L}j}(\kappa) = \left\{\lambda \in P_j(\kappa) : \tau(\lambda) \in \mathcal{E}_{\mathrm{L}}\right\} \tag{E.18}$$

$$P_{\mathrm{D}j}(\lambda) = \left\{\kappa \in P_j(\lambda) : \tau(\kappa) \in \mathcal{E}_{\mathrm{K}}\right\} \tag{E.19}$$

E.1.3 Markers

$$X_{\mathrm{K}j\lambda\psi} = \begin{cases} X_{\mathrm{K}(j-1)\varepsilon\psi} & \text{if } \exists\varepsilon : \lambda \in Q_{\mathrm{G}(j-1)}(\varepsilon) \\ \omega_{\tau(\lambda)\psi}^{(\lambda)} & \text{otherwise} \end{cases} \tag{E.20}$$

$$X_{\mathrm{K}j\kappa\psi} = \begin{cases} 1 & \text{if } \exists\varepsilon : X_{\mathrm{K}j\varepsilon\psi} = 1 \wedge \left[\varepsilon \in P_j(\kappa) \vee \kappa \in Q_{\mathrm{G}(j-1)}(\varepsilon)\right] \\ 0 & \text{otherwise} \end{cases} \tag{E.21}$$

Table E.4: Principal variables used in the scenario model, as shown in figures 8.8 and 8.9.

Symbol	Description
i	An inspector index ($\mathbb{N}$).
$D_{j\delta}$	Indicates whether defect δ exists in phase j in this scenario.
$M_{ji\kappa}$	Indicates whether inspector i comprehends k-instance κ in the phase j inspection.
$S_{ji\lambda}$	Indicates whether inspector i searches locality λ in the phase j inspection.
$A_{\mathrm{M}ji\kappa}$	Indicates whether active guidance is provided to inspector i for k-instance κ in phase j.
$B_{\mathrm{M}ji\kappa}$	Indicates whether passive guidance is provided to inspector i for k-instance κ in phase j.
$L_{\mathrm{M}ji}$	The comprehension active guidance level for inspector i in phase j.
$A_{\mathrm{M}ji\lambda}$	Indicates whether active guidance is provided to inspector i for locality λ in phase j.
$B_{\mathrm{M}ji\lambda}$	Indicates whether passive guidance is provided to inspector i for locality λ in phase j.
$L_{\mathrm{M}ji}$	The search active guidance level for inspector i in phase j.
Z	The inspection strategy.
$G_{j\delta}$	Indicates whether defect δ propagated from the previous phase, in this scenario.
$F_{j\delta}$	The number of operational failures in phase j resulting from defect δ.
$T_{j\delta}$	Indicates whether a test failure occurs in phase j as a result of defect δ.
$V_{j\delta}$	Indicates whether an investigation into defect δ is conducted in phase j.
$R_{j\delta}$	Indicates whether defect δ is reworked in phase j.
$C_{\mathrm{S}ji\lambda}$	The cost of inspector i searching locality λ in phase j.
$C_{\mathrm{BM}j\kappa}$	The cost of providing passive guidance for k-instance κ in phase j.
$C_{\mathrm{BS}j\lambda}$	The cost of providing passive guidance for locality λ in phase j.
$C_{\mathrm{F}j\delta}$	The cost of operational failures resulting from defect δ in phase j.
$C_{\mathrm{V}j\delta}$	The cost of investigating defect δ in phase j.
$C_{\mathrm{R}j\delta}$	The cost of reworking defect δ in phase j.

E.2   Scenario Model

E.2.1   Defect propagation — $G_{j\delta}$

$$G_{j\delta} = \begin{cases} 1 & \text{if } D_{j\pi(\delta)} = 1 \text{ and } R_{j\pi(\delta)} = W(\delta) \\ 0 & \text{otherwise} \end{cases} \tag{E.22}$$

E.2.2   Defect existence — $D_{j\delta}$

$$D_{j\delta} = \begin{cases} 1 & \text{if } G_{j\delta} = 1 \text{ or } H(\delta) \in \{0, \mathtt{H}\} \\ 0 & \text{otherwise} \end{cases} \tag{E.23}$$

Table E.5: Random variates and parameters used in comprehension modelling.

Variate	Variable	Description
		Comprehension
$\phi_{\mathrm{M}\tau(\kappa)}^{(i)}$	$\Phi_{\mathrm{M}\tau(\kappa)}$	Baseline odds of inspector i comprehensing k-instance κ.
$\phi_{\mathrm{MA}}^{(i)}$	Φ_{MA}	Comprehension active guidance effect for inspector i.
$\phi_{\mathrm{MB}}^{(i)}$	Φ_{MB}	Comprehension passive guidance effect for inspector i.
$\phi_{\mathrm{ML}}^{(i)}$	Φ_{ML}	Comprehension active guidance level effect for inspector i.
$\phi_{\mathrm{MM}\tau(\kappa)\tau(\kappa')}^{(i)}$	$\Phi_{\mathrm{MM}\tau(\kappa)\tau(\kappa')}$	Inverse comprehension dependency effect for inspector i, for where κ depends on κ'.
$\phi_{\mathrm{MS}\tau(\kappa)\tau(\lambda)}^{(i)}$	$\Phi_{\mathrm{MS}\tau(\kappa)\tau(\lambda)}$	Inverse locality dependency effect for inspector i, for where κ depends on λ.
		Search
$\phi_{\mathrm{S}\tau(\lambda)}^{(i)}$	$\Phi_{\mathrm{S}\tau(\lambda)}$	Baseline odds of inspector i searching locality λ.
$\phi_{\mathrm{SA}}^{(i)}$	Φ_{SA}	Search active guidance effect for inspector i.
$\phi_{\mathrm{SB}}^{(i)}$	Φ_{SB}	Search passive guidance effect for inspector i.
$\phi_{\mathrm{SL}}^{(i)}$	Φ_{SL}	Search active guidance level effect for inspector i.
$\phi_{\mathrm{SM}\tau(\lambda)\tau(\kappa)}^{(i)}$	$\Phi_{\mathrm{MM}\tau(\lambda)\tau(\kappa)}$	Decision dependency effect for inspector i, for where λ depends on κ.
$k_{\mathrm{S}\psi}^{(i)}$	$K_{\mathrm{S}\psi}$	Effect of marker ψ on search odds for inspector i.
		Costs
$\phi_{\mathrm{CS}\tau(\lambda)}^{(\lambda)}$	$\Phi_{\mathrm{CS}\tau(\lambda)}$	Log-linear baseline cost of searching locality λ.
$\phi_{\mathrm{CSI}\tau(\lambda)}^{(i)}$	$\Phi_{\mathrm{CSI}\tau(\lambda)}$	Log-linear effect of inspector i on locality λ search costs.
$\phi_{\mathrm{CBM}\tau(\kappa)}^{(\kappa)}$	$\Phi_{\mathrm{CBM}j\tau(\kappa)}$	Log-linear baseline cost of passive guidance provision for k-instance κ.
$\phi_{\mathrm{CBS}\tau(\lambda)}^{(\lambda)}$	$\Phi_{\mathrm{CBS}j\tau(\lambda)}$	Log-linear baseline cost of passive guidance provision for locality λ.
$k_{\mathrm{CS}\psi}$	—	Log-linear effect of marker ψ on search costs.
$k_{\mathrm{CBM}\psi}$	—	Log-linear effect of marker ψ on comprehension passive guidance provision costs.
$k_{\mathrm{CBS}\psi}$	—	Log-linear effect of marker ψ on search passive guidance provision costs.

Table E.6: Random variates and parameters used in verification process modelling.

Variate	Variable	Description
		Process
u_j	U_j	The total operational runtime in phase j.
$\phi_{\mathrm{F}j\tau(\delta)}^{(\delta)}$	$\Phi_{\mathrm{F}j\tau(\delta)}$	Log-linear baseline of operational failures resulting from defect δ.
$\phi_{\mathrm{T}j\tau(\delta)}$	—	Baseline test failure odds for defect δ.
$\phi_{\mathrm{V}j\tau(\delta)}$	—	Baseline odds of investigating defect δ.
$\phi_{\mathrm{R}j\tau(\delta)}$	—	Baseline odds of reworking defect δ.
ϕ_{VF}	—	Effect of the number of operational failures on investigation odds.
$k_{\mathrm{F}\psi}$	—	Log-linear effect of marker ψ on operational failures.
$k_{\mathrm{T}\psi}$	—	Effect of marker ψ on test failure odds.
$k_{\mathrm{V}\psi}$	—	Effect of marker ψ on investigation odds.
$k_{\mathrm{R}\psi}$	—	Effect of marker ψ on rework odds.
		Costs
$\phi_{\mathrm{CF}\tau(\delta)}^{(\delta)}$	$\Phi_{\mathrm{CF}\tau(\delta)}$	Log-linear baseline cost of operational failures resulting from defect δ.
$\phi_{\mathrm{CV}\tau(\delta)}^{(\delta)}$	$\Phi_{\mathrm{CV}\tau(\delta)}$	Log-linear baseline cost of investigating defect δ.
$\phi_{\mathrm{CR}\tau(\delta)}^{(\delta)}$	$\Phi_{\mathrm{CR}\tau(\delta)}$	Log-linear baseline cost of attempting to rework defect δ.
$k_{\mathrm{CF}\psi}$	—	Log-linear effect of marker ψ on operational failure cost.
$k_{\mathrm{CV}\psi}$	—	Log-linear effect of marker ψ on investigation cost.
$k_{\mathrm{CR}\psi}$	—	Log-linear effect of marker ψ on rework cost.

E.2.3 K-instance comprehension (inc. defect detection) — $M_{ji\kappa}$

$$\mathbb{P}(M_{ji\kappa} = 1 | A_{\mathrm{M}ji\kappa}, B_{\mathrm{M}ji\kappa}, L_{\mathrm{M}ji}, \{M_{ji\kappa'}\}_{\kappa' \in P_{\mathrm{M}j}(\kappa)}, \{S_{ji\lambda}\}_{\lambda \in P_{\mathrm{L}j}(\kappa)}) =$$

$$\mathrm{logit}^{-1}\Bigg[\phi_{\mathrm{M}\tau(\kappa)}^{(i)} + \phi_{\mathrm{MA}}^{(i)} \cdot A_{\mathrm{M}ji\kappa} + \phi_{\mathrm{MB}}^{(i)} \cdot B_{\mathrm{M}ji\kappa} + \phi_{\mathrm{ML}}^{(i)} \cdot L_{\mathrm{M}ji} +$$

$$+ \sum_{\kappa' \in P_{\mathrm{M}j}(\kappa)} \phi_{\mathrm{MM}\tau(\kappa)\tau(\kappa')}^{(i)} \cdot (1 - M_{ji\kappa'}) + \sum_{\lambda \in P_{\mathrm{L}j}(\kappa)} \phi_{\mathrm{MS}\tau(\kappa)\tau(\lambda)}^{(i)} \cdot (1 - S_{ji\lambda})\Bigg] \quad \text{(E.24)}$$

E.2.4 Locality searching — $S_{ji\lambda}$

$$\mathbb{P}(S_{ji\lambda} = 1 | A_{\mathrm{S}ji\lambda}, B_{\mathrm{S}ji\lambda}, L_{\mathrm{S}ji}, \{M_{ji\kappa}\}_{\kappa \in P_{\mathrm{D}j}(\lambda)}, \mathbf{X}_{\mathrm{K}j\lambda}) = \mathrm{logit}^{-1}\Bigg[\phi_{\mathrm{S}\tau(\lambda)}^{(i)} +$$

$$+ \phi_{\mathrm{SA}}^{(i)} \cdot A_{\mathrm{M}ji\kappa} + \phi_{\mathrm{SB}}^{(i)} \cdot B_{\mathrm{M}ji\kappa} + \phi_{\mathrm{SL}}^{(i)} \cdot L_{\mathrm{M}ji} + \sum_{\kappa \in P_{\mathrm{D}j}(\lambda)} \phi_{\mathrm{SM}\tau(\lambda)\tau(\kappa)}^{(i)} \cdot M_{ji\kappa} + \sum_{\psi \in \Psi} k_{\mathrm{S}\psi}^{(i)} \cdot X_{\mathrm{K}j\lambda\psi}\Bigg]$$

$$\text{(E.25)}$$

where $\mathbf{X}_{\mathrm{K}j\lambda} \equiv \{X_{\mathrm{K}j\lambda\psi}\}_{\psi \in \Psi}$.

E.2.5   Active and passive guidance — $A_{\mathrm{M}ji\kappa}, B_{\mathrm{M}ji\kappa}, A_{\mathrm{S}ji\lambda}, B_{\mathrm{S}ji\lambda}$

$$(A_{\mathrm{M}ji\kappa}, B_{\mathrm{M}ji\kappa}) = Z_{\mathrm{M}}\left[j, i, \tau(\kappa), \{\psi \in \Psi : X_{\mathrm{K}j\kappa\psi} = 1\}\right] \tag{E.26}$$

$$(A_{\mathrm{S}ji\lambda}, B_{\mathrm{S}ji\lambda}) = Z_{\mathrm{S}}\left[j, i, \tau(\lambda), \{\psi \in \Psi : X_{\mathrm{K}j\lambda\psi} = 1\}\right] \tag{E.27}$$

E.2.6   Operational failures — $F_{j\delta}$

$$F_{j\delta} = \begin{cases} 0 & \text{if } D_{j\delta} = 0 \text{ or } Y(\delta) = 0 \\ \mathrm{round}\left[U_j \cdot \exp\left(\phi_{\mathrm{F}j\tau(\delta)}^{(\delta)} + \sum_{\psi \in \Psi} k_{\mathrm{F}\psi} \cdot X_{\mathrm{K}j\delta\psi}\right)\right] & \text{otherwise} \end{cases} \tag{E.28}$$

E.2.7   Test failure — $T_{j\delta}$

$$\mathbb{P}(T_{j\delta} = 1 | D_{j\delta}, \mathbf{X}_{\mathrm{K}j\delta}) = \begin{cases} 0 & \text{if } D_{j\delta} = 0 \\ \mathrm{logit}^{-1}\left(\phi_{\mathrm{T}j\tau(\delta)} + \sum_{\psi \in \Psi} k_{\mathrm{T}\psi} \cdot X_{\mathrm{K}j\delta\psi}\right) & \text{otherwise} \end{cases} \tag{E.29}$$

E.2.8   Failure investigation — $V_{j\delta}$

$$\mathbb{P}(V_{j\delta} = 1 | F_{j\delta}, T_{j\delta}, \mathbf{X}_{\mathrm{K}j\delta}) =$$
$$\begin{cases} 0 & \text{if } F_{j\delta} + T_{j\delta} = 0 \\ \mathrm{logit}^{-1}\left(\phi_{\mathrm{V}j\tau(\delta)} + \phi_{\mathrm{VF}} \cdot F_{j\delta} + \sum_{\psi \in \Psi} k_{\mathrm{V}\psi} \cdot X_{\mathrm{K}j\delta\psi}\right) & \text{otherwise} \end{cases} \tag{E.30}$$

E.2.9 Defect rework — $R_{j\delta}$

$$\mathbb{P}(R_{j\delta}|\{M_{ji\delta}\}_{1\leq i\leq Z_{Ij}}, V_{j\delta}, \mathbf{X}_{Kj\delta}) =$$

$$\begin{cases} 0 & \text{if } V_{j\delta} + \sum_{i=1}^{Z_{Ij}} M_{ji\delta} = 0 \\ \text{logit}^{-1}\left(\phi_{Rj\tau(\delta)} + \sum_{\psi\in\Psi} k_{R\psi} \cdot X_{Kj\delta\psi}\right) & \text{otherwise} \end{cases} \quad \text{(E.31)}$$

E.2.10 Cost of searching — $C_{Sji\lambda}$

$$C_{Sji\lambda} = \begin{cases} 0 & \text{if } S_{ji\lambda} = 0 \\ \exp\left(\phi_{CS\tau(\lambda)}^{(\lambda)} + \phi_{CSI\tau(\lambda)}^{(i)} + \sum_{\psi\in\Psi} k_{CS\psi} \cdot X_{Kj\lambda\psi}\right) & \text{otherwise} \end{cases} \quad \text{(E.32)}$$

E.2.11 Cost of providing passive comprehension guidance — $C_{BMj\kappa}$

$$C_{BMj\kappa} = \begin{cases} 0 & \text{if } \sum_{i=1}^{Z_{Ij}} B_{Mji\kappa} = 0 \\ \exp\left(\phi_{CBM\tau(\kappa)}^{(\kappa)} + \sum_{\psi\in\Psi} k_{CBM\psi} \cdot X_{Kj\kappa\psi}\right) & \text{otherwise} \end{cases} \quad \text{(E.33)}$$

E.2.12 Cost of providing passive search guidance — $C_{BSj\lambda}$

$$C_{BSj\lambda} = \begin{cases} 0 & \text{if } \sum_{i=1}^{Z_{Ij}} B_{Sji\lambda} = 0 \\ \exp\left(\phi_{CBS\tau(\lambda)}^{(\lambda)} + \sum_{\psi\in\Psi} k_{CMS\psi} \cdot X_{Kj\lambda\psi}\right) & \text{otherwise} \end{cases} \quad \text{(E.34)}$$

E.2.13 Cost of operational failure — $C_{Fj\delta}$

$$C_{Fj\delta} = \sum_{\eta=1}^{F_{j\delta}} \exp\left[\phi_{CFj\tau(\delta)}^{(\eta)} + \sum_{\psi\in\Psi} k_{CF\psi} \cdot X_{Kj\delta\psi}\right] \quad \text{(E.35)}$$

where $\phi_{\mathrm{CF}j\tau(\delta)}^{(\eta)} \sim \Phi_{\mathrm{CF}j\tau(\delta)}, 1 \leq \eta \leq F_{j\delta}$.

E.2.14 Cost of failure investigation — $C_{\mathrm{V}j\delta}$

$$
C_{\mathrm{V}j\delta} = \begin{cases} 0 & \text{if } V_{j\delta} = 0 \\ \exp\left(\phi_{\mathrm{CV}j\tau(\delta)}^{(\delta)} + \sum_{\psi \in \Psi} k_{\mathrm{CV}\psi} \cdot X_{\mathrm{K}j\delta\psi}\right) & \text{otherwise} \end{cases} \tag{E.36}
$$

E.2.15 Cost of defect rework — $C_{\mathrm{R}j\delta}$

$$
C_{\mathrm{R}j\delta} = \begin{cases} 0 & \text{if } R_{j\delta} = 0 \\ \exp\left(\phi_{\mathrm{CR}j\tau(\delta)}^{(\delta)} + \sum_{\psi \in \Psi} k_{\mathrm{CR}\psi} \cdot X_{\mathrm{K}j\delta\psi}\right) & \text{otherwise} \end{cases} \tag{E.37}
$$

E.3 Simulation Inputs

This section contains the *model context* file `synthetic.py` used to obtain simulation results. Tables E.7, E.8, E.9 and E.10 briefly describe the simulation inputs and show their correspondence to symbols used in the model.

```
# -*- coding: iso-8859-1 -*-

dtypes = ['req_omission', 'req_commission', 'spec_mismatch', 'incorrect_logic']
ktypes = ['data_obj', 'func_goal', 'method_logic', 'state']
ltypes = ['func_req', 'ext_interface', 'model', 'view', 'controller']
markers = ['important', 'complex']
phases = 3  # Requirements, coding and release

dep_complexity_pdf = { # [ETYPE], PDF
    'req_omission':    Gaussian(100.0, 25.0),
    'req_commission':  Gaussian(100.0, 25.0),
    'spec_mismatch':   Gaussian(100.0, 25.0),
    'incorrect_logic': Gaussian(100.0, 25.0),

    'data_obj':        Gaussian(40.0, 5.0),
    'func_goal':       Gaussian(60.0, 5.0),
    'method_logic':    Gaussian(50.0, 20.0),
```

Table E.7: Metamodel inputs.

Input	Description (including simulation input name)
$\mathcal{E}_{\mathrm{D}}$	**dtypes**: The set of defect types.
$\mathcal{E}_{\mathrm{K}} - \mathcal{E}_{\mathrm{D}}$	**ktypes**: The set of non-defect k-instance types.
$\mathcal{E}_{\mathrm{L}}$	**ltypes**: The set of locality types.
Ψ	**markers**: The set of markers.
J	**phases**: The number of development phases.
Γ_{0e}	**base_entity_count_pmf**: Distribution of the number of initial type-e entities.
$\Xi_{0ee'}$	**base_dep_pmf**: Distribution of the number of initial dependencies that type-e entities have on type-e' entities.
Θ_e	**dep_complexity_pdf**: Distribution of dependence complexity for type-e entities.
$\Omega_{e\psi}$	**marker_prob**: Probability of non-propagated type-e localities having marker ψ.
$\Gamma_{\mathrm{H}jee'}$	**entity_generation_pmf**: Distribution of the number of type e entities hierarchically derived from type-e' entities in phase j.
$\Xi_{\mathrm{H}ee'e''}$	**dep_generation_pmf**: Distribution of the number of localised dependencies that type-e entities have on type-e' entities in the context of a type-e'' entity.
$\Gamma_{\mathrm{GD}jee'0}$	**unreworked_defect_propagation_pmf**: Distribution of the number of type-e defects in phase j propagating from a type-e' defect when the latter is left unreworked.
$\Gamma_{\mathrm{GD}jee'1}$	**reworked_defect_propagation_pmf**: Distribution of the number of type-e defects in phase j propagating from a type-e' defect when rework has been attempted.
Σ_{je}	**rework_failure_prob**: Probability of failure in attempting to rework a type-e defect in phase j.
$\Gamma_{\mathrm{GE}jee'}$	**entity_propagation_pmf**: Distribution of the number of type-e entities in phase j propagating from a non-defect, type-e' entity.
$\Xi_{\mathrm{G}jee'}$	**dep_propagation_prob**: Probability of a type-e entity having a propagated dependency on a type-e' entity, given that one is possible.

```
    'state':             Gaussian(70.0, 10.0),


    'func_req':          Gaussian(20.0, 5.0),
    'ext_interface':     Gaussian(20.0, 5.0),
    'model':             None, # Propagated rather than generated
    'view':              None,
    'controller':        None,
}


base_entity_count_pmf = { # [ETYPE], PMF
    'data_obj':          PoissonPMF(10),
    'func_req':          PoissonPMF(15),
    'ext_interface':     PoissonPMF(5),
}



base_dep_pmf = { # [ETYPE, ETYPE], PMF, zeroPMF
    ('data_obj',         'func_req'):        PoissonPMF(5),
    ('data_obj',         'ext_interface'):   PoissonPMF(2),
    ('req_omission',     'data_obj'):        PoissonPMF(1),
    ('req_commission',   'data_obj'):        PoissonPMF(1),
```

Table E.8: Comprehension modelling inputs.

Input	Description (including simulation input name)
Φ_{Me}	**comprh_intercept_pdf**: Distribution of inspectors' baseline comprehension odds for type-e k-instances.
Φ_{MA}	**comprh_aguidance_effect_pdf**: Distribution of inspectors' comprehension active guidance effects.
Φ_{MB}	**comprh_pguidance_effect_pdf**: Distribution of inspectors' comprehension passive guidance effects.
Φ_{ML}	**comprh_aguidance_level_effect_pdf**: Distribution of inspectors' comprehension active guidance level effects.
$\Phi_{MMee'}$	**comprh_deps_inv_effect_pdf**: Distribution of inspectors' inverse comprehension dependency effects, for where type-e k-instances depend on type-e' k-instances.
$\Phi_{MSee'}$	**locality_deps_inv_effect_pdf**: Distribution of inspectors' inverse locality dependency effects, for where type-e k-instances depend on type-e' localities.
Φ_{Se}	**search_intercept_pdf**: Distribution of inspectors' baseline search odds for type-e localities.
Φ_{SA}	**search_aguidance_effect_pdf**: Distribution of inspectors' search active guidance effects.
Φ_{SB}	**search_pguidance_effect_pdf**: Distribution of inspectors' search passive guidance effects.
Φ_{SL}	**search_aguidance_level_effect_pdf**: Distribution of inspectors' search active guidance level effects.
$\Phi_{MMee'}$	**decision_deps_effect_pdf**: Distribution of inspectors' decision dependency effects, for where type-e localities depend on type-e' k-instances.
$K_{S\psi}$	**search_marker_effect_pdf**: Distribution of inspectors' marker ψ effects on search odds.

Table E.9: Verification process modelling inputs.

Input	Description (including simulation input name)
U_j	**runtime_pdf**: Distribution of operational runtime in phase j.
Φ_{Fje}	**op_failure_intercept_pdf**: Distribution of log-linear baseline operational failures resulting from type-e defects in phase j.
ϕ_{Tje}	**test_failure_intercept**: Baseline odds of test failure for type-e defects in phase j.
ϕ_{Vje}	**investigation_intercept**: Baseline odds of investigating type-e defects in phase j.
ϕ_{Rje}	**rework_intercept**: Baseline odds of reworking type-e defects in phase j.
ϕ_{VF}	**investigation_op_failure_effect**: Effect of the number of operational failures on investigation odds.
$k_{F\psi}$	**op_failure_marker_effect**: Log-linear effect of marker ψ on operational failures.
$k_{T\psi}$	**test_failure_marker_effect**: Effect of marker ψ on test failure odds.
$k_{V\psi}$	**investigation_marker_effect**: Effect of marker ψ on investigation odds.
$k_{R\psi}$	**rework_marker_effect**: Effect of marker ψ on rework odds.

Table E.10: Scenario cost inputs.

Input	Description (including simulation input name)
$\Phi_{\mathrm{CS}e}$	**search_cost_intercept_pdf**: Distribution of baseline search costs for type-e localities.
$\Phi_{\mathrm{CSI}e}$	**search_cost_inspector_effect_pdf**: Distribution of inspector effects on search costs for type-e localities.
$\Phi_{\mathrm{CBM}je}$	**comprh_pguidance_cost_intercept_pdf**: Distribution of baseline comprehension passive guidance provision costs for type-e k-instances.
$\Phi_{\mathrm{CBS}je}$	**search_pguidance_cost_intercept_pdf**: Distribution of baseline search passive guidance provision costs for type-e localities.
$k_{\mathrm{CS}\psi}$	**search_cost_marker_effect**: Log-linear effect of marker ψ on search costs.
$k_{\mathrm{CBM}\psi}$	**comprh_pguidance_cost_marker_effect**: Log-linear effect of marker ψ on comprehension passive guidance provision costs.
$k_{\mathrm{CBS}\psi}$	**search_pguidance_cost_marker_effect**: Log-linear effect of marker ψ on search passive guidance provision costs.
$\Phi_{\mathrm{CF}e}$	**op_failure_cost_intercept_pdf**: Distribution of log-linear baseline operational failure costs resulting from type-e defects.
$\Phi_{\mathrm{CV}e}$	**investigation_cost_intercept_pdf**: Distribution of log-linear baseline investigation costs for type-e defects.
$\Phi_{\mathrm{CR}e}$	**rework_cost_intercept_pdf**: Distribution of log-linear baseline rework costs for type-e defects.
$k_{\mathrm{CF}\psi}$	**op_failure_cost_marker_effect**: Log-linear effect of marker ψ on operational failure cost.
$k_{\mathrm{CV}\psi}$	**investigation_cost_marker_effect**: Log-linear effect of marker ψ on investigation cost.
$k_{\mathrm{CR}\psi}$	**rework_cost_marker_effect**: Log-linear effect of marker ψ on rework cost.

```
    ('req_omission',   'func_goal'):       PoissonPMF(1),
    ('req_commission', 'func_goal'):       PoissonPMF(1),
}


marker_prob = { # [LTYPE, MARKER], BinaryPMF, zeroPMF
    ('func_req',       'important'):   BinaryPMF(0.3),
    ('func_req',       'complex'):     BinaryPMF(0.2),
    ('ext_interface',  'complex'):     BinaryPMF(0.2),
}


# Entity generation & localised dependency insertion
entity_generation_pmf = { # [PHASE, ETYPE, ETYPE], PMF, zeroPMF
    (0, 'func_goal',       'func_req'):        DeltaPMF(1),
    (0, 'req_omission',    'func_req'):        PoissonPMF(0.5),
    (0, 'req_commission',  'func_req'):        PoissonPMF(0.5),
    (0, 'req_omission',    'ext_interface'):   PoissonPMF(0.25),
    (0, 'req_commission',  'ext_interface'):   PoissonPMF(0.25),

    # Phase 1 (coding) generation
    (1, 'method_logic',    'model'):           PoissonPMF(10),
    (1, 'state',           'model'):           PoissonPMF(2),
    (1, 'method_logic',    'view'):            PoissonPMF(15),
```

```
    (1, 'state',            'view'):              PoissonPMF(3),
    (1, 'method_logic',     'controller'):        PoissonPMF(5),
    (1, 'state',            'controller'):        PoissonPMF(1),
    (1, 'spec_mismatch',    'method_logic'):      PoissonPMF(0.1),
    (1, 'spec_mismatch',    'state'):             PoissonPMF(0.2),
    (1, 'incorrect_logic',  'method_logic'):      PoissonPMF(0.2),
    (1, 'incorrect_logic',  'state'):             PoissonPMF(0.3),

    # No generation occurs in the release phase; all entities are propagated
    # from coding.
}


dep_generation_pmf = { # [PHASE, ETYPE, ETYPE, ETYPE], PMF, zeroPMF
    (0, 'func_goal',        'func_req',           'func_req'):      DeltaPMF(1),
    (0, 'req_omission',     'func_req',           'func_req'):      DeltaPMF(1),
    (0, 'req_commission',   'func_req',           'func_req'):      DeltaPMF(1),
    (0, 'req_omission',     'ext_interface',      'ext_interface'): DeltaPMF(1),
    (0, 'req_commission',   'ext_interface',      'ext_interface'): DeltaPMF(1),

    # Phase 1 (coding) generation
    (1, 'method_logic',     'model',              'model'):         DeltaPMF(1),
    (1, 'method_logic',     'view',               'view'):          DeltaPMF(1),
    (1, 'method_logic',     'controller',         'controller'):    DeltaPMF(1),
    (1, 'state',            'model',              'model'):         DeltaPMF(1),
    (1, 'state',            'view',               'view'):          DeltaPMF(1),
    (1, 'state',            'controller',         'controller'):    DeltaPMF(1),
    (1, 'state',            'method_logic',       'model'):         PoissonPMF(3),
    (1, 'state',            'method_logic',       'view'):          PoissonPMF(3),
    (1, 'state',            'method_logic',       'controller'):    PoissonPMF(3),
    (1, 'spec_mismatch',    'method_logic',       'method_logic'):  DeltaPMF(1),
    (1, 'spec_mismatch',    'state',              'state'):         DeltaPMF(1),
    (1, 'incorrect_logic',  'method_logic',       'method_logic'):  DeltaPMF(1),
    (1, 'incorrect_logic',  'state',              'state'):         DeltaPMF(1),

    # No generation occurs in the release phase; all entities are propagated
    # from coding.
}


# Entity propagation
reworked_defect_propagation_pmf = { # [PHASE1, DTYPE, DTYPE], PMF, zeroPMF
    (1, 'incorrect_logic', 'req_omission'):      PoissonPMF(0.5),
    (1, 'incorrect_logic', 'req_commission'):    PoissonPMF(0.5),
    (2, 'incorrect_logic', 'req_omission'):      PoissonPMF(1),
    (2, 'incorrect_logic', 'req_commission'):    PoissonPMF(1),
    (2, 'incorrect_logic', 'spec_mismatch'):     PoissonPMF(0.5),
    (2, 'incorrect_logic', 'incorrect_logic'):   PoissonPMF(0.5),
```

```
}


unreworked_defect_propagation_pmf = { # [PHASE1, DTYPE, DTYPE], PMF, zeroPMF
    (1, 'incorrect_logic', 'req_omission'):    PoissonPMF(2),
    (1, 'incorrect_logic', 'req_commission'):  PoissonPMF(4),
    (2, 'incorrect_logic', 'req_omission'):    PoissonPMF(2),
    (2, 'incorrect_logic', 'req_commission'):  PoissonPMF(4),
    (2, 'incorrect_logic', 'spec_mismatch'):   PoissonPMF(4),
    (2, 'incorrect_logic', 'incorrect_logic'): PoissonPMF(2),
}


rework_failure_prob = { # [PHASE, DTYPE], BinaryPMF)
    # The spec_mismatch and incorrect_logic defects cannot occur in the
    # requirements phase.
    (1, 'req_omission'):   BinaryPMF(0.4),
    (1, 'req_commission'): BinaryPMF(0.2),
    (1, 'spec_mismatch'):  None,
    (1, 'incorrect_logic'): None,

    (2, 'req_omission'):   BinaryPMF(0.4),
    (2, 'req_commission'): BinaryPMF(0.2),
    (2, 'spec_mismatch'):  BinaryPMF(0.2),
    (2, 'incorrect_logic'): BinaryPMF(0.2),
}


entity_propagation_pmf = { # [PHASE1, ETYPE, ETYPE], PMF, zeroPMF
    (1, 'model',        'data_obj'):       PoissonPMF(1),
    (1, 'view',         'ext_interface'):  PoissonPMF(1),
    (1, 'controller',   'func_req'):       PoissonPMF(1),
    (1, 'method_logic', 'func_req'):       PoissonPMF(5),
    (2, 'model',        'model'):          ListPMF([0.0, 0.8, 0.2]),
    (2, 'view',         'view'):           ListPMF([0.0, 0.6, 0.4]),
    (2, 'controller',   'controller'):     ListPMF([0.0, 0.8, 0.2]),
    (2, 'method_logic', 'method_logic'):   ListPMF([0.0, 0.8, 0.2]),
    (2, 'state',        'state'):          ListPMF([0.0, 0.8, 0.2]),
}


dep_propagation_prob = { # [PHASE1, ETYPE, ETYPE], BinaryPMF, zeroPMF
    (1, 'model',         'method_logic'):   BinaryPMF(0.5),
    (2, 'model',         'method_logic'):   BinaryPMF(1.0),

    (1, 'method_logic',  'model'):          BinaryPMF(0.5),
    (1, 'method_logic',  'view'):           BinaryPMF(0.5),
    (1, 'method_logic',  'controller'):     BinaryPMF(0.5),
    (1, 'state',         'model'):          BinaryPMF(0.5),
    (1, 'state',         'view'):           BinaryPMF(0.5),
```

```
    (1, 'state',             'controller'):      BinaryPMF(0.5),
    (1, 'state',             'method_logic'):    BinaryPMF(0.5),
    (1, 'spec_mismatch',     'method_logic'):    BinaryPMF(0.5),
    (1, 'spec_mismatch',     'state'):           BinaryPMF(0.5),
    (1, 'incorrect_logic',   'method_logic'):    BinaryPMF(0.5),
    (1, 'incorrect_logic',   'state'):           BinaryPMF(0.5),
    (1, 'req_omission',      'method_logic'):    BinaryPMF(0.5),
    (1, 'req_omission',      'controller'):      BinaryPMF(0.5),
    (1, 'req_omission',      'view'):            BinaryPMF(0.5),
    (1, 'req_commission',    'method_logic'):    BinaryPMF(0.5),
    (1, 'req_commission',    'controller'):      BinaryPMF(0.5),
    (1, 'req_commission',    'view'):            BinaryPMF(0.5),

    (2, 'method_logic',      'model'):           BinaryPMF(1.0),
    (2, 'method_logic',      'view'):            BinaryPMF(1.0),
    (2, 'method_logic',      'controller'):      BinaryPMF(1.0),
    (2, 'state',             'model'):           BinaryPMF(1.0),
    (2, 'state',             'view'):            BinaryPMF(1.0),
    (2, 'state',             'controller'):      BinaryPMF(1.0),
    (2, 'state',             'method_logic'):    BinaryPMF(1.0),
    (2, 'spec_mismatch',     'method_logic'):    BinaryPMF(1.0),
    (2, 'spec_mismatch',     'state'):           BinaryPMF(1.0),
    (2, 'incorrect_logic',   'method_logic'):    BinaryPMF(1.0),
    (2, 'incorrect_logic',   'state'):           BinaryPMF(1.0),
    (2, 'req_omission',      'method_logic'):    BinaryPMF(0.5),
    (2, 'req_omission',      'controller'):      BinaryPMF(0.5),
    (2, 'req_omission',      'view'):            BinaryPMF(0.5),
    (2, 'req_commission',    'method_logic'):    BinaryPMF(0.5),
    (2, 'req_commission',    'controller'):      BinaryPMF(0.5),
    (2, 'req_commission',    'view'):            BinaryPMF(0.5),
}


comprh_intercept_pdf = { # [KTYPE], PDF
    'req_omission':     Gaussian(-1.0, 0.5), # P ~= 0.3
    'req_commission':   Gaussian(0.0, 0.5),  # P ~= 0.5
    'spec_mismatch':    Gaussian(0.0, 0.5),  # P ~= 0.5
    'incorrect_logic': Gaussian(1.0, 0.5),  # P ~= 0.7
    'data_obj':         Gaussian(1.5, 0.5),  # P ~= 0.8
    'func_goal':        Gaussian(1.5, 0.5),  # P ~= 0.8
    'method_logic':     Gaussian(1.0, 0.5),  # P ~= 0.7
    'state':            Gaussian(0.0, 0.5),  # P ~= 0.5
}


comprh_aguidance_effect_pdf = Gaussian(2.0, 0.5) # PDF
comprh_pguidance_effect_pdf = Gaussian(3.0, 1.0) # PDF
comprh_aguidance_level_effect_pdf = Gaussian(-0.01, 0.005) # PDF
```

```
search_intercept_pdf = { # [LTYPE], PDF
    'func_req':         Gaussian(0.5, 0.5),
    'ext_interface':    Gaussian(0.5, 0.5),
    'model':            Gaussian(0.0, 0.5),
    'view':             Gaussian(-0.5, 0.5),
    'controller':       Gaussian(0.5, 0.5),
}


search_aguidance_effect_pdf = Gaussian(2.0, 0.5) # PDF
search_pguidance_effect_pdf = Gaussian(1.5, 0.3) # PDF
search_aguidance_level_effect_pdf = Gaussian(-0.01, 0.005) # PDF


search_marker_effect_pdf = { # [MARKER], PDF, 0.0
    'important':    Gaussian(0.5, 0.1),
    'complex':      Gaussian(-0.5, 0.1),
}


search_cost_intercept_pdf = { # [LTYPE], PDF
    'func_req':         Triangular(0.5, 4.0, 1.5, log),
    'ext_interface':    Triangular(0.5, 3.0, 1.5, log),
    'model':            Triangular(0.5, 2.0, 1.0, log),
    'view':             Triangular(1.0, 4.0, 1.5, log),
    'controller':       Triangular(1.0, 5.0, 2.5, log),
}


search_cost_inspector_effect_pdf = { # [LTYPE], PDF
    'func_req':         Gaussian(1.0, 0.5, log),
    'ext_interface':    Gaussian(1.0, 0.5, log),
    'model':            Gaussian(1.0, 0.2, log),
    'view':             Gaussian(1.0, 0.5, log),
    'controller':       Gaussian(1.0, 1.0, log),
}


search_cost_marker_effect = { # [MARKER], float, 0.0
    'important':    log(1.0),
    'complex':      log(10.0),
}


comprh_deps_inv_effect_pdf = { # [KTYPE, KTYPE], PDF, zeroPDF
    ('req_omission',    'data_obj'):        Gaussian(-1.0, 0.2),
    ('req_omission',    'func_goal'):       Gaussian(-1.0, 0.2),
    ('req_commission',  'data_obj'):        Gaussian(-1.0, 0.2),
    ('req_commission',  'func_goal'):       Gaussian(-1.0, 0.2),
    ('spec_mismatch',   'method_logic'):    Gaussian(-1.0, 0.2),
    ('spec_mismatch',   'state'):           Gaussian(-1.0, 0.2),
```

```
    ('incorrect_logic', 'method_logic'):    Gaussian(-1.0, 0.2),
    ('incorrect_logic', 'state'):           Gaussian(-1.0, 0.2),
    ('data_obj',        'func_goal'):        Gaussian(-1.0, 0.2),
    ('state',           'method_logic'):     Gaussian(-1.0, 0.2),
}


locality_deps_inv_effect_pdf = { # [KTYPE, LTYPE], PDF, zeroPDF
    ('req_omission',    'func_req'):         Gaussian(-1.0, 0.2),
    ('req_omission',    'ext_interface'):    Gaussian(-1.0, 0.2),
    ('req_commission', 'func_req'):          Gaussian(-1.0, 0.2),
    ('req_commission', 'ext_interface'):     Gaussian(-1.0, 0.2),
    #('spec_mismatch',   ''):
    #('incorrect_logic', ''):
    ('data_obj',        'ext_interface'):    Gaussian(-1.0, 0.2),
    ('func_goal',       'func_req'):         Gaussian(-1.0, 0.2),
    ('method_logic',    'model'):            Gaussian(-1.0, 0.2),
    ('method_logic',    'view'):             Gaussian(-1.0, 0.2),
    ('method_logic',    'controller'):       Gaussian(-1.0, 0.2),
    ('state',           'model'):            Gaussian(-1.0, 0.2),
    ('state',           'view'):             Gaussian(-1.0, 0.2),
    ('state',           'controller'):       Gaussian(-1.0, 0.2),
}

decision_deps_effect_pdf = { # [LTYPE, KTYPE], PDF, zeroPDF
    #'func_req'
    #'ext_interface'
    ('model', 'method_logic'):               Gaussian(0.5, 0.1),
    #'view'
    #'controller'
}

test_failure_intercept = { # [PHASE1, DTYPE], float
    (1, 'req_omission'):    -4.0,
    (1, 'req_commission'): -4.0,
    (1, 'spec_mismatch'):   -2.0,
    (1, 'incorrect_logic'): 1.0,
    (2, 'req_omission'):    -2.0,
    (2, 'req_commission'): -2.0,
    (2, 'spec_mismatch'):   -1.0,
    (2, 'incorrect_logic'): 1.0,
}

test_failure_marker_effect = { # [MARKER], float, 0.0
    'important':    1.0,
    'complex':      -0.5,
```

```
}

op_failure_intercept_pdf = { # [PHASE1, DTYPE], PDF
    (1, 'req_omission'):    Triangular(0.001,  0.1, 0.025, log),
    (1, 'req_commission'):  Triangular(0.001,  0.1, 0.025, log),
    (1, 'spec_mismatch'):   None,
    (1, 'incorrect_logic'): None,
    (2, 'req_omission'):    Triangular(0.001,  0.1, 0.025, log),
    (2, 'req_commission'):  Triangular(0.001,  0.1, 0.025, log),
    (2, 'spec_mismatch'):   Triangular(0.0002, 0.05, 0.010, log),
    (2, 'incorrect_logic'): Triangular(0.0001, 0.1, 0.010, log),
}


op_failure_marker_effect = { # [MARKER], float, 0.0
    'important':    2.0,
    'complex':     -0.5,
}

op_failure_cost_intercept_pdf = { # [PHASE1, DTYPE], PDF
    (1, 'req_omission'):    Triangular(0.05,  1.5, 0.30, log),
    (1, 'req_commission'):  Triangular(0.05,  1.5, 0.30, log),
    (1, 'spec_mismatch'):   None,
    (1, 'incorrect_logic'): None,
    (2, 'req_omission'):    Triangular(0.05,  1.5, 0.30, log),
    (2, 'req_commission'):  Triangular(0.05,  1.5, 0.30, log),
    (2, 'spec_mismatch'):   Triangular(0.01,  1.0, 0.20, log),
    (2, 'incorrect_logic'): Triangular(0.01,  1.0, 0.10, log),
}

runtime_pdf = { # [PHASE1], PDF, 0.0
    1: DeltaPDF(0.0),
    2: Triangular(100.0, 1000.0, 300.0),
}

op_failure_cost_marker_effect = { # [MARKER], float, 0.0
    'important':    3.0,
    'complex':      0.5,
}

investigation_intercept = { # [PHASE1, DTYPE], float
    (1, 'req_omission'):    2.0,
    (1, 'req_commission'):  2.0,
    (1, 'spec_mismatch'):   2.0,
    (1, 'incorrect_logic'): 1.0,
    (2, 'req_omission'):    2.0,
```

```
    (2, 'req_commission'):  2.0,
    (2, 'spec_mismatch'):   2.0,
    (2, 'incorrect_logic'): 1.0,
}


investigation_marker_effect = { # [MARKER], float, 0.0
    'important':    1.0,
    'complex':      -1.0,
}


investigation_op_failure_effect = 0.05 # float

investigation_cost_intercept_pdf = { # [PHASE1, DTYPE], PDF
    (1, 'req_omission'):    Triangular(0.05,  5.0, 1.00, log),
    (1, 'req_commission'):  Triangular(0.05,  5.0, 1.00, log),
    (1, 'spec_mismatch'):   Triangular(0.50, 10.0, 2.00, log),
    (1, 'incorrect_logic'): Triangular(0.05, 25.0, 2.00, log),
    (2, 'req_omission'):    Triangular(0.05,  5.0, 1.00, log),
    (2, 'req_commission'):  Triangular(0.05,  5.0, 1.00, log),
    (2, 'spec_mismatch'):   Triangular(0.50, 10.0, 2.00, log),
    (2, 'incorrect_logic'): Triangular(0.05, 25.0, 2.00, log),
}


investigation_cost_marker_effect = { # [MARKER], float, 0.0
    'important':    log(1.0),
    'complex':      log(0.5),
}


rework_intercept = { # [PHASE, DTYPE], float
    (0, 'req_omission'):    4.0,
    (0, 'req_commission'):  4.0,
    (0, 'spec_mismatch'):   None,
    (0, 'incorrect_logic'): None,
    (1, 'req_omission'):    3.0,
    (1, 'req_commission'):  3.0,
    (1, 'spec_mismatch'):   3.0,
    (1, 'incorrect_logic'): 2.0,
    (2, 'req_omission'):    1.0,
    (2, 'req_commission'):  1.0,
    (2, 'spec_mismatch'):   2.0,
    (2, 'incorrect_logic'): 2.0,
}


rework_marker_effect = { # [MARKER], float, 0.0
    'important':    2.0,
    'complex':      -1.0,
```

```
}


rework_cost_intercept_pdf = { # [PHASE, DTYPE], PDF
    (0, 'req_omission'):   Triangular(0.1,  10.0,  2.0, log),
    (0, 'req_commission'): Triangular(0.1,  10.0,  2.0, log),
    (0, 'spec_mismatch'):  None,
    (0, 'incorrect_logic'): None,
    (1, 'req_omission'):   Triangular(1.0, 100.0, 20.0, log),
    (1, 'req_commission'): Triangular(1.0, 100.0, 20.0, log),
    (1, 'spec_mismatch'):  Triangular(0.5,  50.0, 10.0, log),
    (1, 'incorrect_logic'): Triangular(0.1,  10.0,  2.0, log),
    (2, 'req_omission'):   Triangular(2.0, 200.0, 40.0, log),
    (2, 'req_commission'): Triangular(2.0, 200.0, 40.0, log),
    (2, 'spec_mismatch'):  Triangular(1.0, 100.0, 20.0, log),
    (2, 'incorrect_logic'): Triangular(0.2,  20.0,  4.0, log),
}


rework_cost_marker_effect = { # [MARKER], float, 0.0
    'important':    log(2.0),
    'complex':      log(2.0),
}


comprh_pguidance_cost_intercept_pdf = { # [KTYPE], PDF
    'data_obj':         Triangular(0.05, 1.0, 0.2, log),
    'func_goal':        Triangular(0.05, 2.5, 0.5, log),
    'method_logic':     Triangular(0.05, 2.5, 0.5, log),
    'state':            Triangular(0.05, 2.5, 0.5, log),
}


comprh_pguidance_cost_marker_effect = { # [MARKER], float, 0.0
    'important':    log(1.0),
    'complex':      log(5.0),
}


search_pguidance_cost_intercept_pdf = { # [LTYPE], PDF
    'func_req':         Triangular(0.4,  4.0, 1.0, log),
    'ext_interface':    Triangular(0.4,  4.0, 1.0, log),
    'model':            Triangular(0.1,  0.5, 0.2, log),
    'view':             Triangular(0.2,  1.0, 0.4, log),
    'controller':       Triangular(0.2,  1.0, 0.4, log),
}


search_pguidance_cost_marker_effect = { # [MARKER], float, 0.0
    'important':    log(1.0),
    'complex':      log(5.0),
}
```

Bibliography

Abdelnabi, Z., G. Cantone, M. Ciolkowski, and D. Rombach (2004, Aug). Comparing code reading techniques applied to object-oriented software frameworks with regard to effectiveness and defect detection rate. In *Proc. of the 3rd International Symposium on Empirical Software Engineering (ISESE'04)*, pp. 239–248.

Ackerman, A. F., L. S. Buchwald, and F. H. Lewski (1989). Software inspections: An effective verification process. *IEEE Software 6*(3), 31–36.

Akinola, O. S. and A. O. Osofisan (2009). An empirical comparative study of checklist-based and ad hoc code reading techniques in a distributed groupware environment. *5*(1), 25–35.

Anderson, P., T. Reps, and T. Teitelbaum (2003, Aug). Design and implementation of a fine-grained software inspection tool. *IEEE Transactions on Software Engineering 29*(8), 721–733.

Basili, V. R. (1997). Evolving and packaging reading technologies. *Journal of Systems and Software 38*(1), 3–12.

Basili, V. R., S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M. V. Zelkowitz (1996). The empirical investigation of perspective-based reading. *Empirical Software Engineering 1*(2), 133–164.

Belli, F. and R. Crişan (1996). Towards automation of checklist-based code-reviews. In *Proc. of the 7th International Symposium on Software Reliability Engineering (ISSRE'96)*, pp. 24–33.

Berlin, L. M. (1993). Beyond program understanding: A look at programming expertise in industry. In *Proc. of the 5th Workshop on Empirical Studies of Programmers*, pp. 6–25.

Bernárdez, B., M. Genero, A. Durán, and M. Toro (2004, Sep). A controlled experiment for evaluating a metric-based reading technique for requirements inspection. In *Proc. of the 10th International Symposium on Software Metrics (METRICS'04)*, pp. 257–268.

Bianchi, A., F. Lanubile, and G. Visaggio (2001, Apr). A controlled experiment to assess the effectiveness of inspection meetings. In *Proc. of the 7th International Symposium on Software Metrics (METRICS 2001)*, pp. 42–50.

Biffl, S. (2000, Dec). Analysis of the impact of reading technique and inspector capability on individual inspection performance. In *Proc. of the 7th Asia-Pacific Software Engineering Conference (APSEC 2000)*, pp. 136–145.

Boehm, B. and V. R. Basili (2001). Software defect reduction top 10 list. *IEEE Computer 34*(1), 135–137.

Boehm, B. W., C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece (2000). *Software Cost Estimation With CO-COMO II*. Prentice Hall.

Briand, L. C., K. El Emam, B. G. Freimut, and O. Laitenberger (2000, Jun). A comprehensive evaluation of capture-recapture models for estimating software defect content. *IEEE Transactions on Software Engineering 26*(6), 518–540.

Brooks, R. E. (1983, Jun). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies 18*(6), 543–554.

Brykczynski, B. (1999, Jan). A survey of software inspection checklists. *ACM SIGSOFT Software Engineering Notes* (1), 82–89.

Burkhardt, J.-M., F. Détinne, and S. Wiedenbeck (1998, Jun). The effect of object-oriented programming expertise in several dimensions of comprehension strategies. In *Proc. of the 6th International Workshop on Program Comprehension (IWPC'98)*, pp. 82–89.

Chan, L., K. Jiang, and S. Karunasekera (2005). A tool to support perspective based approach to software code inspection. In *Proc. of the 2005 Australian Software Engineering Conference (ASWEC'05)*, pp. 110–117.

Cheng, B. and R. Jeffery (1996). Comparing inspection strategies for software requirements specifications. In *Proc. of the 1996 Australian Software Engineering Conference (ASWEC'96)*, pp. 203–211.

Chernak, Y. (1996, Dec). A statistical approach to the inspection checklist formal synthesis and improvement. *IEEE Transactions on Software Engineering 22*(12).

Ciolkowski, M. (2009). What do we know about perspective-based reading? An approach for quantitative aggregation in software engineering. In *Proc. of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM'09)*, Washington, DC, USA, pp. 133–144. IEEE Computer Society.

Ciolkowski, M., C. Differding, O. Laitenberger, and J. Münch (1997). Empirical investigation of perspective-based reading: A replicated experiment. Technical Report 97-13, International Software Engineering Research Network (ISERN).

Cockram, T. (2001, Jan). Gaining confidence in software inspection using a Bayesian belief model. *Software Quality Journal 9*(1), 31–42.

Collett, D. (2003). *Modelling Binary Data* (Second ed.). Texts in Statistical Science. Chapman & Hall/CRC.

Collofello, J. S. and S. N. Woodfield (1989, Mar). Evaluating the effectiveness of reliability-assurance techniques. *Journal of Systems and Software 9*(3), 191–195.

Cooper, D. J. A., M. W. Chan, M. Harding, G. Mehra, P. Woodward, B. R. von Konsky, and M. C. Robey (2006, Apr). Using dependence graphs to assist manual and automated object oriented software inspections. In *Proc. of the 2006 Australian Software Engineering Conference (ASWEC'06)*, pp. 262–269.

Cooper, D. J. A., B. Khoo, B. R. von Konsky, and M. C. Robey (2004, Jan). Java implementation verification using reverse engineering. In *Proc. of the 27th Australasian Computer Science Conference (ACSC'04)*, pp. 203–211.

Cooper, D. J. A., B. R. von Konsky, M. C. Robey, and D. A. McMeekin (2007, Apr). Obstacles to comprehension in usage based reading. In *Proc. of the 2007 Australian Software Engineering Conference (ASWEC'07)*, pp. 233–244.

Crawley, M. J. (2002). *Statistical Computing: An Introduction to Data Analysis using S-Plus*. Wiley Publishing.

Curtis, B. (1986). By the way, did anyone study any real programmers? In *Proc. of the 1st Workshop on Empirical Studies of Programmers*, pp. 256–262.

Denger, C., M. Ciolkowski, and F. Lanubile (2004, Aug). Investigating the active guidance factor in reading techniques for defect detection. In *Proc. of the 3rd International Symposium on Empirical Software Engineering (ISESE'04)*, pp. 219–228.

Dunsmore, A., M. Roper, and M. Wood (2000). Object-oriented inspection in the face of delocalisation. In *Proc. of the 22nd International Conference on Software Engineering (ICSE 2000)*, pp. 467–476. ACM Press.

Dunsmore, A., M. Roper, and M. Wood (2003, Sep). The development and evaluation of three diverse techniques for object-oriented code inspection. *IEEE Transactions on Software Engineering 29*(8), 677–686.

Ebenau, R. G. and S. H. Strauss (1994). *Software Inspection Process*. Systems Design & Implementation Series. McGraw-Hill.

Egyed, A. (2003, Feb). A scenario-driven approach to trace dependency analysis. *IEEE Transactions on Software Engineering 29*(2), 116–132.

El Emam, K. and I. Wieczorek (1998, Nov). The repeatability of code defect classifications. In *Proc. of the 9th International Symposium on Software Reliability Engineering (ISSRE'98)*, pp. 322–333.

Ericsson, K. A. and H. A. Simon (1993). *Protocol Analysis: Verbal Reports as Data* (Revised ed.). Massachusetts Institute of Technology.

Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal 15*(1), 182–211.

Fagan, M. E. (1986, Jul). Advances in software inspections. *IEEE Transactions on Software Engineering 12*(7), 744–751.

Fagan, M. E. (2002). *Software Pioneers: Contributions to Software Engineering*, Chapter A History of Software Inspections, pp. 563–573. Springer-Verlag.

Feldman, A. J., J. A. Halderman, and E. W. Felten (2007). Security analysis of the Diebold AccuVote-TS voting machine. In *Proc. of the 2nd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT'07)*.

Freimut, B., L. C. Briand, and F. Vollei (2005, Dec). Determining inspection cost-effectiveness by combining project data and expert opinion. *IEEE Transactions on Software Engineering 31*(12), 1074–1092.

Fusaro, P., F. Lanubile, and G. Visaggio (1997). A replicated experiment to assess requirements inspection techniques. *Empirical Software Engineering 2*(1), 39–57.

Gellenbeck, E. M. and C. R. Cook (1991). An investigation of procedure and variable names as beacons during program comprehension. In *Proc. of the 4th Workshop on Empirical Studies of Programmers*, pp. 65–81.

Gilb, T. and D. Graham (1993). *Software Inspection*. Addison-Wesley.

Gugerty, L. and G. M. Olson (1986). Comprehension differences in debugging by skilled and novice programmers. In *Proc. of the 1st Workshop on Empirical Studies of Programmers*, pp. 13–27.

Halling, M. and S. Biffl (2002, Oct). Investigating the influence of software inspection process parameters on inspection meeting performance. In *IEE Proceedings — Software*, Volume 149, pp. 115–121.

Halling, M., S. Biffl, T. Grechenig, and M. Köhle (2001, Sep). Using reading techniques to focus inspection performance. In *Proc. of the 27th EUROMICRO Conference (EUROMICRO'01)*, pp. 248–257.

Hannay, J. E., D. I. K. Sjøberg, and T. Dybå (2007, Feb). A systematic review of theory use in software engineering experiments. *IEEE Transactions on Software Engineering 33*(2), 87–107.

Harjumaa, L., I. Tervonen, and A. Huttunen (2005, Sep). Peer reviews in real life — motivators and demotivators. In *Proc. of the 5th International Conference on Quality Software (QSIC'05)*, pp. 29–36.

Harrell, Jr., F. E. (2001). *Regression Modeling Strategies: With Applications to Linear Models, Logistic Regression, and Survival Analysis*. Springer-Verlag.

Hatton, L. (2008, Jul-Aug). Testing the value of checklists in code inspections. *IEEE Software 25*(4), 82–88.

He, L. and J. Carver (2006, Sep). PBR vs. checklist: A replication in the n-fold inspection context. In *Proc. of the 5th International Symposium on Empirical Software Engineering (ISESE'06)*, pp. 95–104.

Hedberg, H. and J. Iisakka (2006, Oct). Technical reviews in agile development: Case Mobile-D$^{\text{TM}}$. In *Proc. of the 6th International Conference on Quality Software (QSIC'06)*, pp. 347–353.

Hewett, R. and P. Kijsanayothin (2009, Apr). On modeling software defect repair time. *Empirical Software Engineering 14*, 165–186.

Höst, M., B. Regnell, and C. Wohlin (2000, Nov). Using students as subjects — a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering 5*(3), 201–214.

Höst, M., C. Wohlin, and T. Thelin (2005). Experimental context classification: Incentives and experience of subjects. In *Proc. of the 27th International Conference on Software Engineering (ICSE'05)*, pp. 470–478.

Hougaard, P. (2000). *Analysis of Multivariate Survival Data*. Statistics for Biology and Health. Springer-Verlag.

Hungerford, B. C., A. R. Hevner, and R. W. Collins (2004, Feb). Reviewing software diagrams: A cognitive study. *IEEE Transactions on Software Engineering 30*(2), 82–96.

Jeffery, R. and L. Scott (2002). Has twenty-five years of empirical software engineering made a difference? In *Proc. of the 9th Asia-Pacific Software Engineering Conference (APSEC'02)*, pp. 539–546.

Jones, C. (1996, Apr). Software defect-removal efficiency. *IEEE Computer 29*(4), 94–95.

Kim, J., J. Hahn, and H. Hahn (2000, Sep). How do we understand a system with (so) many diagrams? Cognitive integration processes in diagrammatic reasoning. *Information Systems Research 11*(3), 284–303.

Knight, J. C. and E. A. Myers (1993, Nov). An improved inspection technique. *Communications of the ACM 36*(11), 51–61.

Koller, D. and N. Friedman (2009). *Probabilistic Graphical Models: Principles and Techniques*. Massachusetts Institute of Technology.

Kusumoto, S., K. ichi Matsumoto, T. Kikuno, and K. Torii (1991, Sep). Experimental evaluation of the cost effectiveness of software reviews. In *Proc. of the 15th International Computer Software and Applications Conference (COMPSAC'91)*, pp. 424–429.

Laitenberger, O. and C. Atkinson (1999, May). Generalizing perspective-based inspection to handle object-oriented development artifacts. In *Proc. of the 21st International Conference on Software Engineering (ICSE'99)*, pp. 494–503.

Laitenberger, O. and J.-M. DeBaud (2000, Jan). An encompassing life cycle centric survey of software inspection. *50*(1), 5–31.

Laitenberger, O., K. El Emam, and T. G. Harbich (2001, May). An internally replicated quasi-experimental comparison of checklist and perspective-based reading of code documents. *IEEE Transactions on Software Engineering 27*(5).

Lanubile, F., T. Mallardo, F. Calefato, C. Denger, and M. Ciolkowski (2004, Sep). Assessing the impact of active guidance for defect detection: A replicated experiment. In *Proc. of the 10th International Symposium on Software Metrics (METRICS'04)*, pp. 269–278.

Lanubile, F. and G. Visaggio (2000). Evaluating defect detection techniques for software requirements inspections. Technical Report 00-08, International Software Engineering Research Network (ISERN).

Lee, K. and B. Boehm (2005, Nov). Empirical results from an experiment on value-based review (VBR). In *Proc. of the 4th International Symposium on Empirical Software Engineering (ISESE'05)*, pp. 3–12.

Letovsky, S. (1986). Cognitive processes in program comprehension. In *Proc. of the 1st Workshop on Empirical Studies of Programmers*, pp. 58–79.

Letovsky, S., J. Pinto, R. Lampert, and E. Soloway (1987). A cognitive analysis of a code inspection. In *Proc. of the 2nd Workshop on Empirical Studies of Programmers*, pp. 231–247.

Letovsky, S. and E. Soloway (1986, May). Delocalized plans and program comprehension. *IEEE Software 3*(3), 41–49.

Levendel, Y. (1990, Feb). Reliability analysis of large software systems: Defect data modeling. *IEEE Transactions on Software Engineering 16*(2), 141–152.

Leveson, N. G. (1986). Software safety: why, what, and how. *ACM Computing Surveys 18*(2), 125–163.

Leveson, N. G. and C. S. Turner (1993). An investigation of the Therac-25 accidents. *IEEE Computer 26*(7), 18–41.

Linger, R. C., H. D. Mills, and B. I. Witt (1979). *Structured Programming: Theory and Practice.* The Systems Programming Series. Addison-Wesley.

Littman, D. C., J. Pinto, S. Letovsky, and E. Soloway (1986). Mental models and software maintenance. In *Proc. of the 1st Workshop on Empirical Studies of Programmers*, pp. 80–98.

Lu, S., Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou (2005, Jun). BugBench: A benchmark for evaluating bug detection tools. In *Proc. of the 2005 Workshop on the Evaluation of Software Defect Detection Tools.*

Maldonado, J. C., J. Carver, F. Shull, S. Fabbri, E. Dória, L. Martimiano, M. Mendonça, and V. Basili (2006, Mar). Perspective-based reading: A replicated experiment focused on individual reviewer effectiveness. *Empirical Software Engineering 11*(1), 119–142.

Malik, M. M., M. I. Ullah, M. Jaffar-Ur-Rehman, and H. B. Asghar (2004, Dec). An attribute-based comparison of software design inspection techniques. In *Proc. of the 8th International Multitopic Conference (INMIC 2006)*, pp. 409–416.

Martin, J. and W. T. Tsai (1990, Feb). N-fold inspection: A requirements analysis technique. *Communications of the ACM 33*(2), 225–232.

McCabe, T. J. (1976, Dec). A complexity measure. *IEEE Transactions on Software Engineering 2*(4), 308–320.

McMeekin, D. A., B. R. von Konsky, E. Chang, and D. J. A. Cooper (2008). Checklist inspections and modifications: Applying bloom's taxonomy to categorise developer comprehension. In *Proc. of the 16th International Conference on Program Comprehension (ICPC'08)*, pp. 224–229.

McMeekin, D. A., B. R. von Konsky, M. C. Robey, and D. J. A. Cooper (2009, Apr). The significance of participant experience when evaluating software inspection techniques. In *Proc. of the 20th Australian Software Engineering Conference (ASWEC 2009)*, pp. 200–209.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review 61*, 81–97.

Miller, J., M. Wood, and M. Roper (1998). Further experiences with scenarios and checklists. *Empirical Software Engineering 3*(1), 37–64.

Moha, N., Y.-G. Guéhéneuc, and P. Leduc (2006, Sep). Automatic generation of detection algorithms for design defects. In *Proc. of the 21st International Conference on Automated Software Engineering (ASE'06)*, pp. 297–300.

National Institute of Standards and Technology (2002, May). *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Planning Report 02-3.

Object Management Group (2003, Mar). *Unified Modelling Language, v1.5*. Object Management Group.

Parnas, D. L. and D. M. Weiss (1985). Active design reviews: Principles and practices. In *Proc. of the 8th International Conference on Software Engineering (ICSE'85)*, pp. 132–136.

Pennington, N. (1987). Comprehension strategies in programming. In *Proc. of the 2nd Workshop on Empirical Studies of Programmers*, pp. 100–113.

Petersen, K., K. Rönkkö, and C. Wohlin (2008, Oct). The impact of time controlled reading on software inspection effectiveness and efficiency. In *Proc. of the 2nd International Symposium on Empirical Software Engineering and Measurement (ESEM'08)*, pp. 139–148.

Porter, A., H. Siy, A. Mockus, and L. Votta (1998, Jan). Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology 7*(1), 41–79.

Porter, A. A. and P. M. Johnson (1997, Mar). Assessing software review meetings: Results of a comparative analysis of two experimental studies. *IEEE Transactions on Software Engineering 23*(3), 129–145.

Porter, A. A. and L. G. Votta (1997, Nov-Dec). What makes inspections work? *IEEE Software 14*(6), 99–102.

Porter, A. A. and L. G. Votta (1998, Dec). Comparing detection methods for software requirements inspections: A replication using professional subjects. *Empirical Software Engineering 3*(4).

Porter, A. A., L. G. Votta, and V. R. Basili (1995, Jun). Comparing detection methods for software requirements inspections: A replicated experiment. *IEEE Transactions on Software Engineering 21*(6), 563–575.

R Development Core Team (2009). *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. ISBN 3-900051-07-0.

Raz, T. and A. T. Yaung (1997, Apr). Factors affecting design inspection effectiveness in software development. *Journal of Information and Software Technology 29*(4), 297–305.

Regnell, B., P. Runeson, and T. Thelin (2000, Dec). Are the perspectives really different? — Further experimentation on scenario-based reading of requirements. *Empirical Software Engineering 5*(4), 331–356.

Rigby, P. C., D. M. German, and M.-A. D. Storey (2008, May). Open source software peer review practices: A case study of the apache server. In *Proc. of the 30th International Conference on Software Engineering (ICSE'08)*, pp. 541–550.

Rist, R. S. (1986). Plans in programming: Definition, demonstration and development. In *Proc. of the 1st Workshop on Empirical Studies of Programmers*, pp. 28–47.

Rist, R. S. (1996). System structure and design. In *Proc. of the 6th Workshop on Empirical Studies of Programmers*, pp. 163–194.

Rubinstein, R. Y. (1981). *Simulation and The Monte Carlo Method*. John Wiley & Sons.

Sabaliauskaite, G., F. Matsukawa, S. Kusumoto, and K. Inoue (2002). An experimental comparison of checklist-based reading and perspective-based reading for uml design document inspection. In *Proc. of the 1st International Symposium on Empirical Software Engineering (ISESE'02)*, pp. 148–157.

Sandahl, K., O. Blomkvist, J. Karlsson, C. Krysander, M. Lindvall, and N. Ohlsson (1998). An extended replication of an experiment for assessing methods for software requirements inspections. *Empirical Software Engineering 3*(4), 327–354.

Sauer, C., R. Jeffery, L. Land, and P. Yetton (2000, Jan). The effectiveness of software development technical reviews: A behaviorally motivated program of research. *IEEE Transactions on Software Engineering 26*(1), 1–14.

Seaman, C. B. (1999, Jul-Aug). Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering 25*(4), 557–572.

Shneiderman, B. and R. Mayer (1979, Jun). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information Sciences 8*(3), 219–238.

Shull, F. (1998). *Developing Techniques for Using Software Documents: A Series of Empirical Studies.* Ph. D. thesis, Computer Science Department, University of Maryland, USA.

Siy, H. and L. Votta (2001). Does the modern code inspection have value? In *Proc. of the 17th International Conference on Software Maintenance (ICSM 2001)*, pp. 281–289.

Sjøberg, D. I. K., B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanovic, E. F. Koren, and M. Vokác (2002). Conducting realistic experiments in software engineering. In *Proc. of the 1st International Symposium on Empirical Software Engineering (ISESE'02)*, pp. 17–26.

Sjøberg, D. I. K., J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanović, N.-K. Liborg, and A. C. Rekdal (2005, Sep). A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering 31*(9), 733–753.

Soloway, E. and K. Ehrlich (1984, Sep). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering 10*(5), 595–609.

Soloway, E., J. Pinto, S. Letovsky, D. Littman, and R. Lampert (1988, Nov). Designing documentation to compensate for delocalized plans. *Communications of the ACM 31*(11), 1259–1267.

Sommerville, I. (2001). *Software Engineering* (Sixth ed.). Addison-Wesley.

Sørumgård, S. (1997, Feb). *Verification of Process Conformance in Empirical Studies of Software Development.* Ph. D. thesis, Department of Computer and Information Science, Norwegian University of Science and Technology.

Storey, M.-A. D., F. D. Fracchia, and H. A. Müller (1997, Mar). Cognitive design elements to support the construction of a mental model during software visualization. In *Proc. of the 5th International Workshop on Program Comprehension (IWPC'97)*, pp. 17–28.

Storey, M.-A. D., F. D. Fracchia, and H. A. Müller (1999). Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software 44*(3), 171–185.

Sullivan, M. and R. Chilarege (1991, Jun). Software defects and their impact on system availability — a study of field failures in operating systems. In *Proc. of the 21st(FTCS-21)*, pp. 2–9.

Thelin, T., C. Andersson, P. Runeson, and N. Dzamashvili-Fogelström (2004, Sep). A replicated experiment of usage-based and checklist-based reading. In *Proc. of the 10th International Symposium on Software Metrics (METRICS'04)*, pp. 246–256.

Thelin, T., P. Runeson, and C. Wohlin (2003, Sep). An experimental comparison of usage-based and checklist-based reading. *IEEE Transactions on Software Engineering 29*(8), 687–704.

Therneau, T. and T. Lumley (2008). *survival: Survival analysis, including penalised likelihood.* R package version 2.34-1.

Tichy, W. F. (2000, Dec). Hints for reviewing empirical work in software engineering. *Empirical Software Engineering 5*(4), 309–312.

Travassos, G. H., F. Shull, and J. Carver (2000). A family of reading techniques for OO design inspections. In *Proc. of the 7th Software Quality Workshop (WQS'2000), 14th Brazilian Symposium on Software Engineering (SBES'2000)*, pp. 225–237.

Travassos, G. H., F. Shull, M. Fredericks, and V. R. Basili (1999, Oct). Detecting defects in object oriented designs: Using reading techniques to increase software quality. In *Proc. of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pp. 47–56.

Uwano, H., M. Nakamura, A. Monden, and K. Matsumoto (2006, Mar). Analyzing individual performance of source code review using reviewers' eye movement. In *Proc. of the 2006 Symposium on Eye Tracking Research and Applications (ETRA'06)*, pp. 133–140.

von Mayrhauser, A. and A. M. Vans (1995, Aug). Program comprehension during software maintenance and evolution. *IEEE Computer 28*(8), 44–55.

Votta, L. G. (1993). Does every inspection need a meeting? In *Proc. of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'93)*, pp. 107–114.

Walenstein, A. (2002, Jun). Theory-based analysis of cognitive support in software comprehension tools. In *Proc. of the 10th International Workshop on Program Comprehension (IWPC'02)*, pp. 75–84.

Walenstein, A. (2003, May). Observing and measuring cognitive support: Steps toward systematic tool evaluation and engineering. In *Proc. of the 11th International Workshop on Program Comprehension (IWPC'03)*, pp. 185–194.

Walkinshaw, N., M. Roper, and M. Wood (2005, May). Understanding object-oriented source code from the behavioural perspective. In *Proc. of the 13th International Workshop on Program Comprehension (IWPC'05)*, pp. 215–224.

Weller, E. F. (1993, Sep). Lessons from three years of inspection data. *IEEE Software 10*(5), 38–45.

Wiedenbeck, S. (1986, Dec). Beacons in computer program comprehension. *International Journal of Man-Machine Studies 25* (6), 697–709.

Winkler, D., S. Biffl, and B. Thurnher (2005). Investigating the impact of active guidance on design inspection. In *Proc. of the 6th(PROFES 2005)*, pp. 458–473.

Winkler, D., M. Halling, and S. Biffl (2004, Sep). Investigating the effect of expert ranking of use cases for design inspection. In *Proc. of the 30th EUROMICRO Conference (EUROMICRO'04)*, pp. 362–371.

Wohlin, C., A. Aurum, H. Petersson, F. Shull, and M. Ciolkowski (2002, Aug). Software inspection benchmarking — a qualitative and quantitative comparative opportunity. In *Proc. of the 8th International Symposium on Software Metrics (METRICS'02)*, pp. 118–127.

Wu, Y. P., Q. P. Hu, K. L. Poh, S. H. Ng, and M. Xie (2005, Dec). Bayesian networks modeling for software inspection effectiveness. In *Proc. of the 11th Pacific Rim International Symposium on Dependable Computing (PDRC 2005)*.

Copyright Material