Proceedings of the IASTED International Conference
COMPUTER SCIENCE AND TECHNOLOGY
May 19-21, 2003, Cancun, Mexico

A RE-CONFIGURABLE COMPONENT-BASED SOFTWARE FRAMEWORK

Alex Talevski[1], Elizabeth Chang[2], Tharam S. Dillon[1]

[1] Dept. of Computer Science and Computer Engineering,
La Trobe University, Bundoora, VIC, Australia 3083
Email: {alex, tharam}@cs.latrobe.edu.au

[2] School of Information Systems,
Cutrin University, WA, Australia 6845
Email: change@cbs.curtin.edu.au

ABSTRACT

Component-based software engineering is a way of raising the level of abstraction for software development so that software can be built out of existing context-independent software components that can be widely reused. Research has shown component-based software engineering leads to software that is of higher quality when a component is reused between multiple projects, shorter time-to-market due to the reduction of written source code, and therefore lowers cost [1].

The ever-increasing changes in user requirements and diverse user-bases are the reasons for developing software that can be easily customised after it has been deployed by the user at runtime as opposed to the developer. Using current methods it is difficult to add, remove and/or modify components and their interconnections within an application without changing and recompiling the application source code. Customisation and evolution of an application is difficult to perform.

In this paper we propose a plug and play dynamically reconfigurable component-based framework and we present a prototype implementation. Component-based plug and play software will allow the dynamic addition, removal and modification of a components and reconfiguration at runtime. The framework proposed will allow for evolution and customisation of software dynamically at run-time..

KEY WORDS

Software evolution, Software plug and play, Component-based Software, Re-configurable Software.

1. INTRODUCTION

Considerable progress has been made in recent times in the techniques available for the development of software systems. A variety of approaches have appeared which include object-oriented and component-based techniques. These techniques by and large emphasize important principles of software development such as encapsulation, low coupling and reuse. The object-oriented approach has been utilized in several different forms:

- Writing code from scratch using object-oriented languages such as C++ and Java.
- Use of applications such as InfoPower and VisualAge, which constitute elements of complete applications that can be customised to meet your needs, such as control the flow of interaction.
- The use of class libraries containing classes that can create objects such as buttons, menus or sockets for use with an application.

The important contribution of the object-oriented paradigm is that it structures the system according to the items that exist in the problem domain. This makes a more natural solution to the problem. These items are normally very stable and changes in the environment where they occur can be easily identified with software constructs. The changes occurring normally affect only one or a few such items, which means that the changes made are usually localised. From this perspective, object-oriented development promotes understanding and maintainability of the system to be developed. However, by reusing existing classes at a higher level, less code is written and productivity is improved but the extent of customisation possible is limited. Current work on components has been largely limited to the creation and static assembly of components manually as the software system is being constructed. Once the system has been developed any further re-configuration, addition or replacement of components requires re-coding of parts of the system (Fig.1).

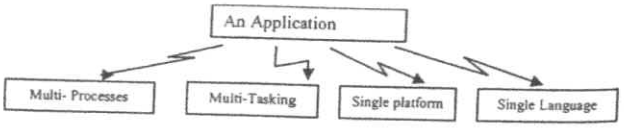

Fig. 1 Application category in mid 1990s. [2]

In order to satisfactorily develop and deploy software that must evolve and adapt to varying user requirements we must look at developing cooperating applications that have multiple processes, are multi-tasking, work across platforms and are able to be implemented using multiple languages. The schematic structure of such a system is illustrated in (Fig.2).

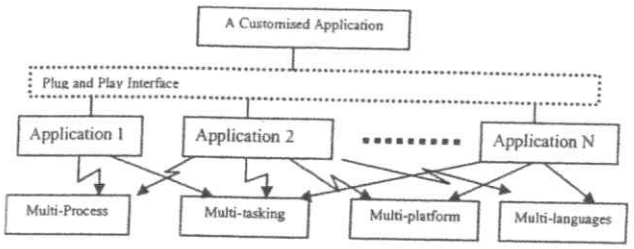

Fig. 2 Modern application category. [2]

Whilst several industrial applications of the category of system shown in (Fig. 1) have been previously described in the literature very few if any of those shown in (Fig. 2) have been.

The IT industry must look at current technologies and produce a hyper-linking model of computing where an organization' s collection of networks, data, applications, clients, and servers can be dynamically connected and disconnected in response to varying user requirements.

In this paper we propose a framework that forms the basis for the development of more complex systems, as shown in (Fig. 2). In the following section we define objects and components. In sections three and four we discuss our conceptual and architectural framework. Section five demonstrates a sample implementation that has been performed using the framework proposed. Section six concludes the paper.

2. OBJECTS AND COMPONENTS

The terms "component" and "object" are sometimes used interchangeably however there are important differences.

Objects. The notions of instantiation, identity, and encapsulation lead to the notion of objects. In contrast to the properties characterizing components, the characteristic properties of objects are:

- An object is a unit of instantiation; it has a unique identity.
- An object has state; this state can be a persistent state.
- An object encapsulates its state and behaviour.

Because an object is a unit of instantiation, objects cannot be partially instantiated. Since an object has individual state, it also has a unique identifier that suffices to identify the object despite state changes for its entire lifetime.

As objects get instantiated, there needs to be a construction plan that describes the state space, initial state, and behavior of a new object. This plan exists before the instantiation of the object. This plan is usually explicitly available and is then called a class. Alternatively, it may be implicitly available in the form of an object that already exists, that is sufficiently close to the object to be created, and that can be cloned. Whether using classes or prototype objects, the newly instantiated object needs to be set to an initial state. The code required to initialise the object can be a static procedure and is usually called a constructor. Alternatively, it can be an object of its own, which is called an object factory.

Components. A component is a binary unit of composition. It is possible to compose a component with other components and integrate this composite component with disparate systems through its interfaces. Composite component architectures are formed form layered components where components at the lower layers provide services to the components above them through middleware. The main notion of component-based development is to provide the ability to inherently reuse previously designed, implemented, composed and refined business services.

The characteristic properties of components are:

- A component exposes its underlying functionality through its exposed well-defined interfaces. Through interfaces the client is completely isolated from the server implementation.
- A component is a unit of independent production, acquisition, and deployment
- A component is a unit of third-party composition.
- A component has no persistent state.
- A component will never be deployed partially. However, the possibility exists of utilising a proportion of the services of a fully deployed component.

Interfaces. An interface is defined as a collection of operations that are used to specify a service of an object or a component. Through its interfaces a component exposes both its provided and required services. It is possible to replace or modify a component' s internals without changing the interface or component interconnections as long as the implementation adheres to the interface contract. An interface contract specifies the pre-conditions that must be met by a client prior to invoking a server operation and the post-conditions that it will receive as a return from the server. A component interface should be standardised so that a component can be widely reused.

3. CONCEPTUAL FRAMEWORK

A component meta-data model as illustrated in (Fig. 3) is used to describe a component, its interfaces, state, post/pre conditions and parameters. Here, for clarity, we have used the Unified Modeling Language (UML) to illustrate our framework. In future it may be necessary to define a different notation system. The component META data model is an explicit representation of a software component, its interdependency, and its environmental assumptions that forms the component specification. The component specification identifies the component interfaces or interface and pre and post conditions of invoking the component. A component specification is realized as a component implementation.

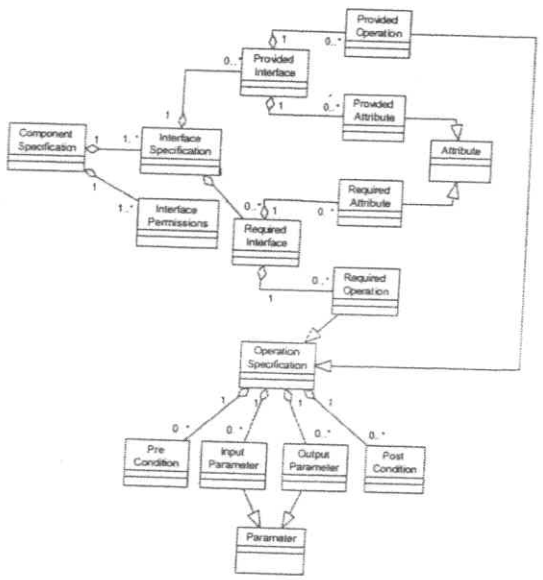

Fig 3. UML Meta model for component specifications

Interfaces. Components expose their functionality through their interfaces. The interface provides access to the exposed underlying properties and methods of the component. The interfaces are seen as a provider and a consumer [3, 4, 5]. In other words, the provider must satisfy the requirements of the consumer. The concept of required interfaces is essential to enabling software plug and play. Each component will expose one or more provided interfaces and zero or more required interfaces.

Methods and attributes are either provided or required by a component.

- **Attributes.** An attribute is a named property value that describes the characteristics of a component. Given permission to do so, an attribute may be requested from any other component.
- **Operation.** An operation is the implementation of a service of a component and represents the dynamic behaviour of that component. Given permission to do so, an operation can be requested from any other component in order to perform some task.

A component' s provided and required interfaces can b obtained by querying the component.

- **Provided Interfaces.** The underlying provided methods and properties that are exposed through a server component' s interfaces represent the services that a server component provides. Exposed interfaces may be either methods or attributes and can be accessed through the provided interface by the client. The results of a call to a provided operation should be documented as post-conditions of that operation.
- **Required Interfaces.** A component may also request a list of services that it requires in order to perform. The services that a component requests through its interfaces may be either methods or attributes. The requirements of a component whether attributes or methods should be documented in the components pre-conditions.

Configuration – Attribute glue

A component or composition may expose a set of required properties that need to be satisfied. Therefore, each component and composition may be customised. Configurations may be preformed either dynamically or statically. With dynamic mapping a required attribute is glued to another component in the application. Two dynamically mapped possibilities exist, either gluing a required attribute to a provided attribute or gluing a required attribute to an operation return. A statically mapped attribute is linked to a static value provided by the user.

Attribute adaptors are used to glue required and provided interfaces together [6]. Once glued, a required attribute always requests the value it requires from the provided attribute that it is glued to.

An adaptor can also convert a provided attribute that is of the wrong type so that it satisfies the requirements of a required attribute.

Composition - Operation glue

A collaboration defines elements that work together to provide some cooperative behavior. Therefore, collaborations have structural as well as behavioral dimensions. A given class might participate in several collaborations. Thus, compositions are groups of interconnected components or other compositions that are composed together using operation glue in order to perform a particular task.

Identifying the clear decompositions in your architecture is the key to developing a hierarchical architecture where an application is built from context-independent building blocks. Once the proper decomposition is performed,

components may change independently, without affecting the components that they are connected to, as long as both components conform to the contract specified by their interfaces. In a similar fashion layered components and compositions can be added, removed and replaced. Like configurations, compositions may be preformed either dynamically or statically. With dynamic mapping a required operation is glued to another component in the application. It is possible to either glue a required operation to a provided operation or to statically map an attribute to a static value provided by the user.

Operation adaptors are used to glue required and provided interfaces together [6]. Once glued, a required operation always calls the provided operation that it is glued to when required. Operation adaptors may be used when a provided operation has an incompatible return type or parameters and needs conversion. If required an attribute adaptor may be used to convert both the return type and operation parameters to what is required. Once the interface requirements of the required operation are satisfied an operation adaptor is used to glue the interfaces.

Component/Composition Interaction

The interaction between client and server must be predefined as an interaction protocol.

Two forms of communication are possible: direct and indirect (Fig. 4).

- **Direct communication (Synchronous)** occurs when the methods and/or properties of a component are glued to another. Operation calls and information is passed directly through these interfaces. This represents component message passing.
- **Indirect communication (Asynchronous)** occurs when a component broadcasts a message on the whiteboard and other components intercept, parse and react to these messages.

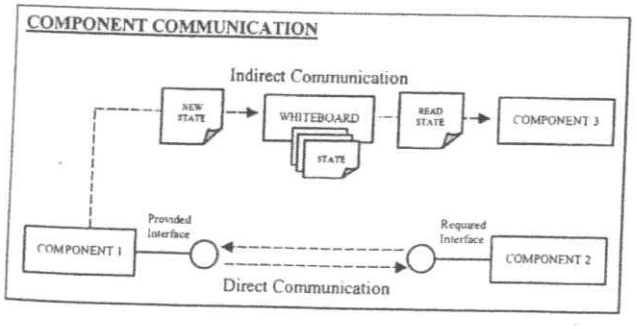

Fig 4. Component Communication

When a component invokes the methods of another through its interface it can generally pass data in the standard fashion using parameters of the standard types. However in order to maintain system independence, complex data types must be sent in the form of a structured document such as an XML file with a XML schema.

4. ARCHITECTURAL FRAMEWORK

The architecture outlined below realizes the abstract conceptual plug and play model. The architecture consists of three major components.

The application generator is used to construct and tailor component-based plug and play applications. The application generator holds a repository of components that have been discovered and are available to the plug and play application building process. From the component list the user is able to create, modify, save, load and use configured compositions, components, modules and solutions and assign security permissions.

The communications server or whiteboard is used by components to post messages about changes in their internal state that may be of concern to others that are not directly connected to them. Messages are in the form of structured XML documents with an XML schema so that they can be easily parsed for relevant information by other components. The message board buffers all messages that have been posted as a sequence of events since its instantiation so that components entering the application after this time are able to synchronise with the current application state.

The solution server holds pointers to both the application and solution databases.

Application database holds the typical underlying database schema and data using a Relational Database Management System (RDBMS). The database is wrapped using the framework so that it also acts as a component. The reason for using the wrapping the application database is so that a more structured and restricted form of access to the database is enforced and a more diverse set of plug-in components can be used. Interface methods are exposed for:

- **Modifying the schema** such as describe the current schema, create a database, create a table and alter a table.
- **Interacting with the database** such as querying the database, inserting records, updating records and deleting records.

Solution database holds currently available configurations and compositions of components, modules and solutions and their user privileges.

The plug and play framework is broken down into four major constructs: solutions, modules, compositions and components.

- **Application solutions** represent completely composed and configured plug and play applications that can be used by a number of users. Application solutions consist of many modules, compositions and components that are configured and composed completely (Fig. 5).

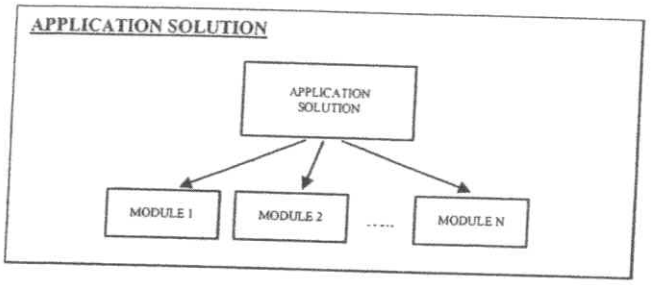

Fig. 5 Application solution with multiple modules

- **Modules** represent groups of components and compositions that usually perform a single business process. A module represents an independent entity that is able to run with no further composition and configuration. A module typically (but not necessarily) has a user interface and several interconnected components and compositions (Fig. 6). Modules can be removed and added at anytime because they do not impact other modules, compositions or components they simply add and remove independent functions to the application.

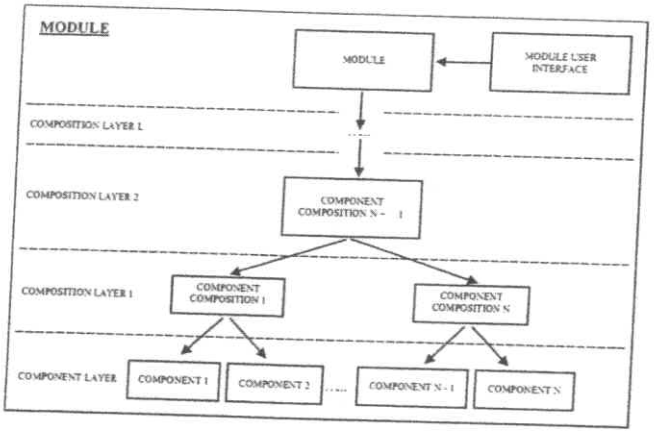

Fig 6. Module with multiple component compositions

Components. A component exposes selected underlying parameters and methods though provided interfaces. Other components can utilise these provided parameters and methods to satisfy their own requirements. In this way it is possible to add and remove components from an application dynamically at run-time (Fig. 7).

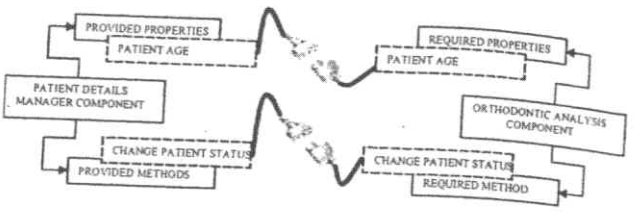

Fig 7. Plug-in a component

Components expose information such as:

- **Class identifier** (CLSID – A unique string consisting of numbers and letters that distinguishes one component from the rest). The CLSID is used to invoke components wherever they reside, whether on a local computer or distributed over the network.
- **The component name** represents the name that the user will see to identify the component.
- **The component type** is used so that components can be found easily and grouped with others of a common type and functionality.
- **The component description** is used to outline the major functions of a component and any other information that the component programmer would like to provide.
- **A QueryInterface** gives detailed information about the required and provided interfaces, the graphical user interface (GUI) and how they are used.
- **A GUIInterface** points to the displayable graphic user interface (GUI) that can be instantiated within a separate window or as a part of a composition.
- **A pre-condition** specifies a constraint that must hold before the invocation of an operation. A pre-condition is the condition under which the operation guarantees that the post-condition will be true.
- **A post-condition** specifies a constraint that must hold after the invocation of an operation.

We have classified components into four different types:

Active Components are those that trigger actions or events to occur without being glued to another action or event. An alarm clock can be considered to be an Active Component where it triggers the ringing action after continually polling the time-checker component for a pre-specified time to occur.

Reactive Components are ones that perform a task only after a particular action or event message has been received from the main application or another component. A stop-alarm component would be considered a reactive component. The stop-alarm component will stop the ringing sound only when the user or the main system has triggered the stop action.

Active and Reactive Components are components that perform both as active and reactive types of components as described above.

Passive Components are considered neither active nor reactive. They can be considered to be tools of the application that can be used by other components to perform their tasks. A time-checker can be considered a

passive component. A time-checker will return the time when asked to by another component.

5. DESIGN AND IMPLEMENTATION

Our previous work concentrated on the addition and removal of module plug-ins into an application to add/remove functionality and was heavily based on the Microsoft Component Object Model (COM) [7, 8]. This work is aimed at providing a more robust and powerful plug and play framework that supports the notion of collaborative components that are hierarchically composed. The plug and play framework is also general enough to be able to be programmed in any of the major component technologies.

Implementation

The current software prototype is implemented using Microsoft' s Component Object Model (COM). The orthodontic application has been broken down into components and the plug and play framework has been implemented with the features mentioned above.

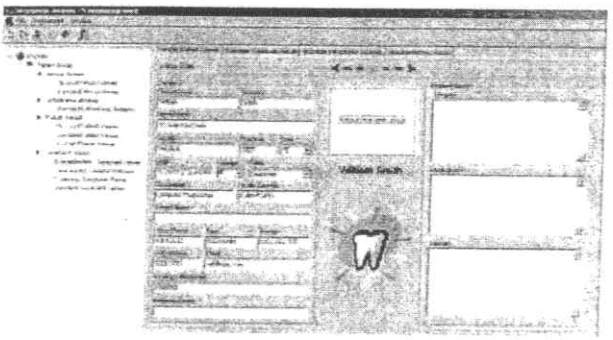

Fig 8. Orthodontic Analysis Module Plug-in

Container. Active document containment is used to implement the application generator. As with OLE documents, active document containment uses a container that provides the display space for Active documents, and servers that provide the user interface and manipulation capabilities for the active documents themselves [4]. Once embedded within a container the module is initialised and is free to access the database and interact with other modules, components and compositions.

Plug-ins. Each plug-in in the orthodontic application is a COM Component that resides in an OleContainer. The application generator dynamically maintains each OleContainer. Each plug-in component can access the central database and communicate with other plug-ins that are connected to it.

Database. The database is a Microsoft Access database wrapped as a COM automation server that exposes one interface per database table. Each interface exposes properties and methods that can be used to modify the schema, query, insert and update the database.

6. CONCLUSION

Benefits such as reduced time-to-market, increased reusability, increased quality and reduced cost can be experienced by reusing components [1]. However, improperly defined component interfaces and incorrect component decomposition results in components that are difficult to add to diverse application environments. It is also difficult to rearrange, add and remove components dynamically at runtime. This paper presents an overview of a plug and play dynamically re-configurable component based framework. This framework defines a component communication protocol and encourages proper component decomposition, reuse and composition. It also makes it possible to rearrange, reconfigure, add and remove components dynamically at runtime.

The framework has been utilized to construct an orthodontic application to demonstrate the feasibility of the approach.

REFERENCES

[1] M. Aoyama, New Age of Software Development: How Component-Based Software Engineering Changes the Way of Software Development, *Proceedings of the International Workshop on Component-Based Engineering, Kyoto*, Japan, April 1998

[2] E. Chang, D. Annal, F. Grunta, Dynamic Plug and Play GUI Architecture, *Proceedings of the 18th IASTED International Conference on Applied Informatics*, AI2000, Innsbruck, Austria, February 2000

[3] I. Holland, Specifying Reusable Components Using Contracts, *Proceedings of ECOOP*, March 1993

[4] N. Medvidovic, R. Taylor, Separating Fact from Fiction in Software Architecture, *Proceedings of the International Software Architecture Workshop*, Orlando, Florida, USA, November 1-2 1998

[5] D. Linthicum, *Enterprise Application Integration* (Addison Wesley Longman, Inc., 1999).

[6] D. Rine, N. Nada, K. Jaber, Using Adapters to Reduce Interaction Complexity in Reusable Component-Based Software Development, *Proceedings of the Symposium on Software Reusability*, Los Angeles, CA, USA, May 1999

[7] A. Talevski, E. Chang, A dynamically re-configurable component-based architecture, *International Journal of Engineering and Intelligent Systems, 10*(1), 2002

[8] A. Talevski, E. Chang, A dynamically re-configurable GUI architecture, World Multiconference on Systemics, Cybernetics and Informatics, Orlando, Florida, USA, July 2001