Composition Without Composite Components

A Major Qualifying Project Report
Submitted to the Faculty of
Worcester Polytechnic Institute

In partial fulfillment of the requirements
For the Degree of Bachelor of Science

Submitted by:
Yi Wang

Submitted To:
Project Advisor, Professor George T. Heineman

Date: December 21st, 2009
ABSTRACT

Component-based architecture designs are aimed to provide high extensibility, maintainability and reusability. More specifically, such systems typically allow new components to be added with no or very little impact on existing system. Maintenance of a system requires only maintenance of its problematic components. And components in existing systems can be directly reused with other components to assemble brand new systems. However, in many cases, direct assembly, maintenance and reuse of primitive components\(^1\) can be repetitive, time consuming and error prone, as such procedures are usually performed on an entire collection of primitive components that are combined together as a single unit. It has become clear that component-composition\(^2\) is not only useful but also essential for a well-designed component-based system. In this paper we propose a new approach to component-composition, which does not require special definition for composite components\(^3\). In this design, all components are treated the same, in order to greatly simplify the system design and implementation while still proving the same level of extensibility, maintainability and reusability.

---

\(^1\) Primitive components are the most basic type of components that are considered as an indivisible atomic unit of state and logic.

\(^2\) Component composition is the process of composing multiple primitive and/or composite components together to form a single unit of composite component.

\(^3\) Composite components are complex component units that are formed by a collection of primitive or other composite components. A composite component is still considered as a single unit rather than a plain collection.
Acknowledgement

The author would like to recognize Professor George T. Heineman for his help and assistance in the completion of the project.
Dedicated to my beloved family members and friends who have supported me throughout my academic studies.
# Table of Contents

ABSTRACT .................................................................................................................................................. 2

ACKNOWLEDGEMENT .......................................................................................................................... 3

DEDICATION ............................................................................................................................................. 4

TABLE OF CONTENTS ........................................................................................................................... 5

TABLE OF FIGURES ............................................................................................................................... 7

1. INTRODUCTION ............................................................................................................................... 8

2. COMPUNIT MODEL[1] ......................................................................................................................... 10
  2.1 COMPONENT MODEL .................................................................................................................. 10
  2.2 CONNECTION MODEL ................................................................................................................. 12
  2.3 COMPOSITION MODEL ............................................................................................................... 12
  2.4 APPLICATION MODEL .............................................................................................................. 14

3. DESIGN ................................................................................................................................................ 15
  3.1 COMPONENT DEFINITION ........................................................................................................... 15
    3.1.1 Activation Tag ....................................................................................................................... 16
    3.1.2 Mappings Tag ....................................................................................................................... 16
    3.1.3 Internal Binding Tag ............................................................................................................. 17
  3.2 COMPONENT AS APPLICATION .................................................................................................... 17
  3.3 RESOURCE HANDLING ............................................................................................................... 18
  3.4 APPLICATION CONFIGURATION ................................................................................................. 19
  3.5 FOUNDATION DEVELOPMENT SUITE [1] .................................................................................. 20

4. SAMPLE APPLICATION ....................................................................................................................... 21
  4.1 APPLICATION OVERVIEW .......................................................................................................... 21
  4.2 PRIMITIVE COMPONENTS ......................................................................................................... 21
    4.2.1 Timer Component .................................................................................................................. 21
    4.2.2 Renderer .............................................................................................................................. 23
    4.2.3 Updater and Driver ............................................................................................................... 23
  4.3 COMPOSITION ............................................................................................................................... 24
    4.3.1 Naming ............................................................................................................................... 24
    4.3.2 Primitive Component Addition ............................................................................................ 25
    4.3.3 Explicit Internal-binding ...................................................................................................... 25
    4.3.4 Activation Setting ............................................................................................................... 26
    4.3.4 Export Composition ............................................................................................................ 27
  4.4 LAUNCHING COMPONENT AS APPLICATION ........................................................................... 27

5. FUTURE WORK .................................................................................................................................... 28

6. RELATED WORK .................................................................................................................................. 30
  6.1 ENTERPRISE JAVA BEAN[4, 5] .................................................................................................. 30
  6.2 FRACTAL JULIA[6, 7] .................................................................................................................. 31
7. EVALUATION ........................................................................................................33
REFERENCES ...........................................................................................................34
Table of Figures

Figure 1 Component model ............................................................... 10
Figure 2 Component lifetime phases ................................................. 11
Figure 3 Connection model .............................................................. 12
Figure 4 Composite Component Model .............................................. 16
Figure 5 TexturedCube application running ...................................... 28
1. INTRODUCTION

Component composition has been a somewhat troubling matter for almost all component-based architectural designs. In order to still provide the same level of extensibility, maintainability and reusability, while keeping the system design as simple as possible, we decided to take a rather low-level approach. As the title of this paper suggests, we propose a design that allows component composition without the use of explicit composite components.

In traditional component architecture design, there are typically five separate distribution files for a component-based application, component description file, application description file, component class archive file, component external resources archive file and application configuration file. The component description file provides the definition of all the components utilized in the application, which may include both primitive and composite components. Application description file contains the structure of the application in terms of the available components provided by the component definition files. Component class archive file holds the compiled class data of all components specified by the component definition files. The external resources archive file contains the resources that are utilized by the components at application runtime. And finally the application configuration file specifies the configuration data for the components in a particular application instance.

In our architecture design, we decided to take a four-type approach, in which we kept all file types except the application description file. Instead of explicitly form an application using an external file, we took the approach to treat an application simply as a composite component. Along with modifications to the component description file structure, our approach makes a simple and clean, yet still powerful architecture for component-based application development.

In section 2, we describe the architectural design of our component system, with each key elements explained in full detail. In section 3, we further explain our composition design with a sample application of a 3D rotating textured cube. This application is a simple 3D graphics program in which a rotating textured cube is displayed on the screen. Through
this example, we show the full testing of primitive component packaging, component composition, the concept of composite component as application, external resources handling and application configuration. In section 4 we examine some of the related component architecture designs and discuss the similarities and differences between those designs and ours. And finally in section 5, we state the future work that are to be implemented for our component architecture.
2. **CompUnit MODEL**\(^1\)

The component architecture design model we developed is based on the CompUnit component architecture. We reused the designs that are appropriate and efficient and added and modified the others to provide a better architecture model for component-based application development. In this section, we discuss the architecture model used in CompUnit platform.

2.1 **Component Model**

In CompUnit architecture model, a component is defined as a single atomic unit that contains a set of providing and requiring connection points.

![Component model](image)

Figure 1 Component model

A providing connection point defines a feature or functionality provided to the outside environment, allowing external units to access the functionality implemented by the component unit. In the above figure, the right circle point represents the providing connection point.

A requiring connection point defines an input connection where external units can be injected into the component to allow the component to utilize the injected unit to perform its internal operations. In the above figure, the left arch point represents the providing connection point.

A component under the CompUnit platform has five distinct lifetime phases including instantiation, customization, connection, activation and deactivation.
The instantiation phase allows the runtime environment to instantiate an instance of the component implementation. During this phase, the constructors of the component implementations are directly invoked to perform the instantiation logic.

Customization phase allows components to initialize their internal data fields to specific values defined in the component assembly file. However, this design requires a completely new assembly description file just for a set of new configuration data even though the application assembly process is identical.

Connection phase is where the runtime environment connects multiple component instances together according to the component assembly file to form the internal network structure of the application components. During this phase, component instances instantiated in the previous phase are injected into each other to form the connections.

The activation phase allows components to activate explicitly to perform operations such as database connection establishment or thread starting. This separation between activation and initialization (instantiation and customization phases) allows components to internally perform late binding on external contexts.

Finally the deactivation phase allows components to properly dispose allocated resources, such as graphics contexts, threads and network connections.
2.2 Connection Model

Under CompUnit’s component connection model, components are connected with each other to form a collection of components that forms a new feature of functionality. This connection model is also used for our design.

Component connections are formed based on matching of provided and required connection points of various components being connected.

![Connection model diagram]

Figure 3 Connection model

As the above figure shows, component A is being injected into component B via the matching between the providing connection point of component A and the requiring connection point of component B.

2.3 Composition Model

CompUnit does not support a full composition model. However, developers may be able to write stub-wrapper classes that manually form the composition of components. The code segment below shows how this is achieved in CompUnit environment.

```java
@Provide({ICommandLine.class})
public class Wrapper implements IComponent, ICommandLine {
    Viewer v;
    BestQuotes bq;
    @Override
    public boolean activate(IResourceRetriever handler) throws Exception {
        // manually instantiate V and Q
        v = new Viewer();
        bq = new BestQuotes();
        v.connect(
```
In this example, the Wrapper component internally performs the phases of its internal components. It first instantiates the Viewer and BestQuotes components and manually connects them together to form the internal binding. Then during the activation phase of the Wrapper component, it manually activates the internal components in the order that is hard-coded into the component implementation.

This may seem to work on a small scale, however, it can quickly break down for large-scale compositions. The manual instantiation, connection and activation processes are all hard-coded into the Wrapper component. This means that this Wrapper component can never be directly reused without modification to existing code. And once modifications occur, the rest of the system that may depend on the Wrapper implementation can fail. The flexibility, reusability and maintainability of this manual wrapping composition process are incredibly low. Also it is impossible to perform the customization phase on the internal components, as the resource retriever injected into the Wrapper component is not capable of locating resources for the internal components. It is only aware of the resources that are used by the Wrapper component. Further wrapping of external
resources may solve the problem, however, the same issues of low flexibility, reusability and maintainability applies to this resource wrapping design as well.

2.4 Application Model
The application definition in CompUnit is defined using an explicit application component assembly file that specifies both the component connection to form the application and also the application configurations. We determined this as being wasteful since an application running with a different configuration should not require the system to re-define the application itself. We discuss our design in detail in the next section.
3. DESIGN

The fundamental principle of our component composition design is to treat both primitive and composite components as a single indivisible unit in the system. From a design perspective, a composite component is composed of various primitive components and/or other composite components. However, once a composite component is successfully constructed, it should be recognized as an atomic unit just like a typical primitive component from a compilation perspective. More specifically, both primitive and composite components are defined in roughly the same way in which they are defined by their provided and require interfaces, their corresponding implementations and the activation of the component. The differences being, a composite component includes additional binding definitions. Based on this observation, instead of defining components at a high up design level, components in our design are defined at the low code and implementation level. Down to the bottom, any component, primitive or composite is simply an atomic unit that may provide or require a set of interfaces backed up by their corresponding implementations. From this low code perspective, both primitive and composite components are almost identical with the exception of internal binding definitions added to a composite component. With this understanding in mind, the composition of components becomes trivial. In fact, the composition process is almost identical to that of the packaging process of primitive components.

3.1 Component Definition

Component definition is exported automatically using our foundation suite as a cx file. It consists of four basic tags, activation, required-mapping, provided-mapping and internal-binding tags. At the very minimum, a component must provide a single activation tag, and the internal-binding tag is only utilized by composite components.
3.1.1 Activation Tag
Activation tag contains one or more activation unit tags that specify the concrete component implementation class that needs to be instantiated, customized and activated by the system. Each activation unit holds a fully qualified class name of the component implementation and an optional resource field that specifies the external resources archive file of the component. A single component can only have a single activation tag. Activation tag also specifies the component implementation activation ordering. Since, each activation unit tag directly corresponds to a single component implementation, the order in which the activation unit tags appear in the parent activation tag defines the order of component implementation activation. The system goes through all the activation unit tags and performs the instantiation, customization and activation process for each of the component implementation defined by that unit tag.

3.1.2 Mappings Tag
A component mapping tag defines an external connection point, either input or output, for the component unit. There are two types of mapping tags, required and provided, which correlates to the input and output connection point.

Required-mapping tag specifies a single required interface of the component and contains one or more class tags that represent the implementations, which actually utilize the interface internally. A required-mapping is considered as an external input point that allows the system to inject appropriate providers instances into the corresponding
required implementations within the component. A single component unit may have multiple required-mapping tags to specify multiple required interfaces.

Provided-mapping tag defines a single provided interface of the component and contains one or more class tags that represent the implementations, which implements the interface within the component unit. It can be considered as an external output point that allows the system to inject the component unit as a provider to the corresponding requiring components with matching interface. A single component unit may have multiple provided-mapping tags to specify multiple provided interfaces.

### 3.1.3 Internal Binding Tag

Internal-binding tag is an optional tag that is only utilized by composite components. It defines a pair of provider and consumer concrete implementations that are bound together based on their matching required and provided interfaces. The system at launch time injects an instance of the provider class into an instance of the consumer class to form the binding connection.

The launching system guarantees that there is only a single instance of each concrete implementation class created for the entire system. This means that if multiple bindings specify the same provider or consumer class, there is actually only a single instance of that class being utilized for all the binding connections. For instance, if a composite component is composed of a pair of matching primitive components, the requiring and providing concrete implementations of the primitive components may be bound together using an internal-binding tag. A single composite component may contain as many internal-binding tags as needed.

### 3.2 Component as Application

Given the component definition structure above, it has become obvious to us that a single component can be directly treated as a complete application unit without the need for an external application definition file.
An application in a traditional component-based architecture is defined by an external definition file, which provides the specifications for component activation and binding procedures. However, such a design introduces duplication of information since composite components also require activation and binding specifications, in which case that portion of information must be duplicated at two separate places. As general software engineering principles state, such duplication of information can greatly reduce the flexibility and maintainability of the system, and also making the design error prone.

In our architecture, an application is simply described by a component, whether being primitive or composite. Activation and internal-binding information is only placed once in the component definition and reused directly for application launching purposes. Not only does this design improve the flexibility and maintainability of the system and making the system much more error resistant, it also helps the system to simplify the application configuration process. This aspect is described in the following section.

### 3.3 Resource Handling

Any non-trivial applications must contain its collection of external resources, such as images, data files and etc. The same concept also applies to a lot of service provider components. During the component composition process, the definition of a pre-composed component is destroyed and replaced with its bare-bone interfaces, implementations, interface-mapping definitions and internal-binding definitions. In order to preserve the resource associations of pre-composed components to allow such its implementations still be able to locate and utilize its resources, a resource locator is created and assigned to the implementations based on the resource field in the activation tag.

The composition process preserves the external resources archive files of each individual pre-composed component. During the activation process, the system constructs a resource retriever instance for each component implementation in the activation tag that has a non-null resource field. As described in the previous section, the resource field in the activation tag specifies the external resources archive file thus allowing the resource retriever to locate the file along with the archived resources data.
An alternative approach would be to collapse the resources archive files of all pre-composed components into a single archive file and create a hierarchy file directory structure within the archive file. This approach would indeed reduce the number of files need to be deployed, however, it requires the system to implicitly form a hierarchical directory structure that is invisible to the developer. Such implicit implementation requirements can be very difficult to enforce and problem detecting may also be complicated due to the implicit nature. For instance, if the resource archive file required by a particular component implementation is corrupted, with our explicit approach, a simple I/O handling exception might be thrown to inform the developer that the component is missing its external resources. This allows the developer to easily find and resolve the problem and may even ignore the problem but still allowing the rest of the components to operate properly. However, with an implicit approach such as the one described above, a failure of the archive file will cause all components to fail thus making it impossible for the application to be functional. This is very much similar to a distributed design allowing us to avoid a single point of failure thus making the system more error prone and robust.

3.4 Application Configuration

As mentioned before, components can be customized with a set of properties that sets the initial values of various data fields within a component. And since in our design, an application is simply a component unit, we support application configuration using component customization while allowing the application definition to be un-changed for multiple configurations.

The developers can export a XML based application configuration file that contains the properties data for all components within the application. At application launch time, this configuration file can be passed into the system launcher to allow customizable components to be customized based on the enclosed properties data. Since the application in our design is simply a primitive component or composite component, this configuration file becomes independent of the actual application definition (component definition), thus allowing a single application definition to be directly reused with
multiple configuration files without the need for any changes to the application (component) definition at all.

Alternatively, if we separated the component definition and application definition, in which case the application definition is responsible for specifying the connection and activation process of components, then for each configured application instance with the same components connected and activated in the same way, we would have to duplicate the application definition data as many times as the number of configurations. Evidently, our current design approach is much more flexible and reusable since no data or definitions are duplicated. Due to this advantage, we chose our current configuration handling design.

3.5 Foundation Development Suite [1]

Along with our architecture design, we have also developed a foundation development suite in Java that allows the developers to automate the primitive component packaging, composite component composition, application configuration export and component-based application launching processes. More specifically, Packager provides primitive component packaging functionality as well as external resources packaging for the components, Composer provides the functionality of composition of primitive components, ConfigurationExporter allows developers to export application configuration files in the correct format and finally the Launcher automates the launching process of component-based application instances. The sample application we present below fully utilizes this development suite for the automated development processes.
4. SAMPLE APPLICATION

In this section, we present a potentially real-world application developed using our component-based architecture, a 3D graphics rendering application. Through this sample application, we attempt to demonstrate all aspects of our design.

4.1 Application Overview

The sample application we present here is a 3D graphics rendering application of a cube that is textured using an external image resource and rotated based on the rendering speed and the external configuration. The application also periodically reports the current rendering frame-rate to the standard output.

The application is presented in the form of a composite component composed form four distinct primitive components, Timer, Updater, Renderer and Driver. Each primitive component is designed to only encapsulate a single feature or functionality of the application, thus allowing them to be reused directly if necessary. However, in this paper, we mainly focus on the composition process of these four primitive components into the application composite component.

4.2 Primitive Components

In this section, we present the individual primitive components that form the application composite component as well as their packaging processes using the Foundation Development Suite, Packager utility unit.

4.2.1 Timer Component

The Timer component defines the utility component that provides the functionality to monitor elapsed time between tick invocations. And based on this interpolation time, it also provides the functionality to calculate a tick update rate.
Based on its provided functionality, the application uses the timer to report the current rendering frame rate. And perhaps more importantly, the application uses the between-frame interpolation time value to control the rotating speed of the rendered cube along with the speed factor specified by the external application configuration file.

The following code segment presents the entirety of the packaging process of this timer component.

```java
packager.setName("Timer");
packager.setPrimary(NanoTimer.class);
packager.addClass(AbstractTimer.class);
packager.seal(this.directory);
```

The first line sets the name of the component. This name value is also used directly as the name of the exported component definition cx file, class archive file and the prefix for the resources file if there are any resources to be packaged. In this particular case, since there are no external resources specified for export, the packager automatically exports Timer.cx and Timer.jar files as the component definition and class archive file correspondingly.

The second line tells the packager of the component implementation class, NanoTimer. The packager internally examines the required and provided annotations to properly fill-in the information of required and provided mappings. The component implementation is automatically set as the only activation unit in the component definition, since a primitive component should only contain single component implementation to be activated.

The third line adds an additional class implementation that is internally utilized by the component implementation to the packager so it exports the class along with the component implementation to the class archive file. In this particular case, AbstractTimer is the super class of the component implementation, NanoTimer.

Finally the fourth line invokes the packager to seal the package and export all the necessary files to the given directory. The directory given must be a valid URL to a directory location.
4.2.2 Renderer

*Renderer* is the component that contains the actual rendering logic of the cube. In this particular case, an OpenGL\(^2\) *VertexBufferObject*\(^3\) is utilized to render the cube spatial. It is also responsible for reading in an external image resource and uses it as the texture for the rendered cube. This allows us to demonstrate the packaging of external resources.

```java
1. packager.setName("Renderer");
2. packager.setPrimary(Renderer.class);
3. packager.addClass(Vector3f.class);
4. packager.addClass(Box.class);
5. packager.addResource(resource);
6. packager.seal(this.directory);
```

Line 1 through 4 and 6 have the same meaning as explained before in the previous section. Line 5 introduces the way to add an external resource file to the packager to be packaged into an external resources archive file that can be used by component at launch time. This informs the packager to properly write in the resource field in the activation unit tag, thus allowing the *Launcher* to properly construct a resource retriever for the component.

4.2.3 Updater and Driver

*Updater* and *Driver* are packaged in very much the same way as *Timer*, so we simply provide the code segments in this section.

*Updater* provides the logic control to the rotation of the cube. More specifically, it allows the cube to be rotated according to the rendering speed as well as the speed factor. *Updater* is in fact a customizable component, allowing the *Launcher* to customize the component with an application configuration file. In this particular case, the speed factor is set using the property value in the configuration file.
Driver is the main rendering loop control unit. It is responsible for maintaining the rendering loop and invoking the other three components to perform the rendering operations of the application. It initializes the OpenGL context and various other settings. And finally it provides the feature that prints out the current rendering frame-rate to the standard console output periodically.

### 4.3 Composition

Once all primitive components are packaged with their cx component definition files, class archive files and optional external resources archive files, the developers can use the Composer utility unit provided by Foundation Development Suite to compose all components together into a single composite component that can either be reused in other applications as an atomic component unit or used as a stand-alone application. In this section we present the composition process of our sample rendering application.

#### 4.3.1 Naming

As with using the Packager, the first thing we need to inform the Composer is the name of the result composite component as the following code segment shows.

```java
composer.setName("TexturedCube");
```

This invocation informs the Composer that composed composite component should be named as TexturedCube. This name value is reflected in the exported composite component definition file, class archive file and the component name field inside the definition file.
4.3.2 Primitive Component Addition

Next we simply add in the primitive components to the Composer unit thus allowing it to extract the necessary information associated with the primitive components to form the composite one. The following code segment shows the adding process.

```java
composer.addComponent(
    this.directory, "Timer");
composer.addComponent(
    this.directory, "Updater");
composer.addComponent(
    this.directory, "Renderer");
composer.addComponent(
    this.directory, "Driver");
```

In order to add a primitive component, a URL to the directory that contains all the necessary files defining the component and the name of the component must be provided to the Composer. The Composer then can automatically locate the files within the directory and extract component information accordingly.

Since a composite component is a distinct component on its own, none of the mappings are carried over from the primitive component definitions. However, all necessary activation information and all class data within the class archive files of the components are automatically carried over as well as binding information if the composition process involves composite components.

4.3.3 Explicit Internal-binding

With primitive component bindings, explicit specification of the binding connections between them must be supplied to with the Composer unit, as the following code segment shows.

```java
final CxBinding td = new CxBinding();
td.setProvider(NanoTimer.class.getName());
td.setConsumer(Driver.class.getName());
composer.addBinding(td);```

25
Internal-bindings are represented by the CxBinding data structure that contains the fully qualified class name of a single provider and a single consumer. For the sample application, there are three bindings need to be formed. From line 1 through 4, an internal-binding between NanoTimer and Driver implementations is created and added to the Composer. From line 5 through 8, an internal-binding between Updater and Driver is created and added to the Composer. From line 9 through 12, an internal-binding between Renderer and Driver is created and added to the Composer. These binding data structures allow the Composer to properly form the composite component internal-binding definitions.

4.3.4 Activation Setting

Unlike primitive component packaging process, since there are multiple components, both primitive and composite are being packaged into a single unit of component; the activation must be explicitly set rather than carrying over the activation settings from the candidate components. This allows the developers to control what component implementations are activated when the result composite component is activated as well as their activation ordering. The following code segment shows the activation setting procedure for our sample application.

```java
composer.setActivation(new ICxUnit[]{
    new CxUnit(NanoTimer.class.getName()),
    new CxUnit(Updater.class.getName()),
    new CxUnit(Renderer.class.getName(), "Renderer"),
    new CxUnit(Driver.class.getName())
});
```
The *Composer* utility unit takes in an array of *ICxUnit* data structures as the activation setting argument. Each *ICxUnit* element in the array specifies a single activation unit in the result composite component. It must contain a fully qualified class name of the component implementation and may contain an optional external resources field that specifies the prefix of the external resources archive file used by the particular component implementation. The order in which the *ICxUnit* appears in the supplied array dictates their activation ordering.

In the above code segment, we specify that Nano*Timer*, *Updater*, *Renderer*, and *Driver* should be activated when the result composite component is activated and in that specific ordering. The *Renderer* component implementation also must be given a resource retriever linked to the external resources archive file that is prefixed with “Renderer” string.

### 4.3.4 Export Composition

After completing the above steps, the composite component is ready for export. Similar to the primitive component packaging process, we supply an URL to a directory that we wish all the exported files should be put into to the *Composer* unit when invoking the `compose` method.

```java
composer.compose(this.directory);
```

The *Composer* unit automatically exports the composite component *cx* definition file and class archive file directly to the given directory. This marks the completion of the component composition process.

### 4.4 Launching Component as Application

As stated in previous section, our design allows a component unit either being primitive or composite to be launched as an application. In other words, an application is simply a component unit. In this section we show how our composite *TexturedCube* component can be launched as a stand-alone application.
To launch a component as an application, Foundation Development Suite provides the utility unit, *Launcher* that provides such functionality. The following code segment shows the launching of our sample application.

```java
1 launcher.launch(directory, "TexturedCube", false, configfile);
```

The *Launcher* requires at the minimum three arguments to be supplied to launch a component. An URL to the directory where the component definition, class data archive and external resources archive files are located. A second argument specifying the name of the component to be launched and a third argument indicating if all the data in the external resources archive file should be cached in memory for fast access. An optional argument of an URL to an application configuration file may be supplied to customize the component and its internal component implementations if the component is a composite one. If the configuration file URL is invalid, the customizable component implementations will be launched with default configuration data.

*Launcher* also provides the functionality to shutdown the launched component application via simple invocation of *shutdown*. It will automatically invoke the *deactivate* method of all the activated component implementations.

![Figure 5 TexturedCube application running](image)
5. FUTURE WORK

The current Foundation Development Suite implementation is completely code-based. In order to improve productivity and ease of use, a graphical user interface will be implemented for the system in the future.

Currently, the system does not support running multiple versions of the same component within a single application. It may be useful in some cases to support such functionality. Therefore, further investigation, design and implementation is planned for supporting multiple versions of the same component within an application concurrently.
6. RELATED WORK

In order to design and develop a fully featured component-based architecture platform, we have investigated into two similar platforms in addition to the root CompUnit platform, Enterprise Java Bean (EJB) and Fractal Julia. In this section, we discuss the properties of these two platforms.

6.1 Enterprise Java Bean

Enterprise Java Bean or EJB is initially developed by IBM and later adopted by Sun Microsystems as part of the Java Enterprise Edition platform. It is designed to provide a standard development framework for the server-side business logic of web applications. It provides a modular design pattern to allow developers to reuse existing modules (components) in new application development, instead of re-inventing the same thing over and over again. From a design perspective, EJB aims to provide the same set of properties as our platform aims to do. This includes the reusability, flexibility and maintainability of existing units of functionality for application development. However, there are quite some differences between how EJB achieves this and our design.

EJB allows developers to declare a reusable component with two interfaces and an implementation. The Home interface and the Component interface along with the implementation that implements these two interfaces. The EJB implementation is then deployed into an EJB container within the application server for execution.

The Home interface defines the set of methods that are not tied to a particular instance of the implementation. These methods are typically factory and utility methods that are used to create and EJB implementation instance or to find an existing instance.

The Component interface defines the feature or function specific methods that are tied to a particular instance of the EJB implementation. The clients invoke these methods to perform the business logic on the application server. More specifically, clients invoke the proxies generated by the EJB container, which in turn places the method arguments into a

---

4 Fractal is the component-based architecture design. Fractal Julia is a Java implementation based on the Fractal design.
message and sends it to the EJB server. Upon receiving a message, the EJB server
degrees the invocation to a particular instance of the EJB implementation to perform the
actual logic processing based on the arguments wrapped inside the message.

In the sense that both EJB and our design are aimed to provide modular component-based
architecture for reusability, flexibility and maintainability, these two designs are similar.
However, in terms of design specifics, the two designs are only similar in the way that
both utilize a providing connection point to allow external invocations. EJB unlike our
design does not provide any means to connect multiple EJB instances together through
provided and required interface matching to form a bigger component. It does not
explicitly address the issue of external resources handling, component customization nor
component composition. The two designs are only similar in their goals, but far different
from the way they achieve the goals and the functionalities supported.

6.2 Fractal Julia[6, 7]

Fractal Component Model is developed by the OW2 Consortium and released under
LGPL license. It is a fully featured component-based architectural model that is language
agnostic. There are several implementations of the Fractal Component Model, and we
investigated with the Fractal Julia Java implementation in detail. Overall, the Fractal
Component Model is quite similar to the CompUnit component model in the sense that
they both support a similar set of features. However, Fractal is slightly more powerful in
the component composition perspective. When compared with our composition design,
Fractal falls behind in terms of flexibility and simplicity.

Fractal Julia defines components in the same way CompUnit and our design do.
Components are defined by a definition file along with a set of provided and required
interfaces with a single implementation that provides the actual logic instructions. The
provided interfaces define the set of functionalities the component provides. And the
required interfaces define the set of injection points where other component instances can
be injected to allow the implementing component to perform operations using the
.injected components.
Similar to our design, Fractal supports component binding to allow multiple components to be connected together to form a collection of components that provides a more sophisticated feature set. Fractal also allows components to be customized via a component membrane, which is somewhat similar to our design in the sense that we also support component customization. Though we implement the customization process using an explicit configuration file to provide a higher level of flexibility.

Though, unlike our design, Fractal handles component composition by the mean of nested, automated wrapping. The platform implementation itself provides the means to automatically generate wrapper classes for multiple components to form a composite component. This is quite inefficient and complex when compared with our design. Fractal Component Model or Fractal Julia does not provide any explicit external resource handling mechanisms, where our design does using the resource retriever and resource field of the component definition.

However, Fractal does provide a powerful feature called instance sharing, where a single component instance can be shared among multiple component instances in the system. This is particularly useful for components like memory manager or device drivers.
7. EVALUATION

<table>
<thead>
<tr>
<th>Feature</th>
<th>EJB</th>
<th>Fractal Julia</th>
<th>CompUnit</th>
<th>Our Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provided</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Required</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Resources</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Composition</td>
<td>✗</td>
<td>Automated Wrapping</td>
<td>Manual Wrapping</td>
<td>✓</td>
</tr>
<tr>
<td>Customization</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Instance Sharing</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Table 1 Feature Comparison

The above table summarizes a comparison between our design and various other component architecture designs including Enterprise Java Bean, Fractal Julia and our parent design CompUnit. As shown in table, our design supports all the identified features except for instance sharing for component-based application development. More importantly, our design supports full component composition, which is a big improvement on top of our parent system CompUnit, which only supports manual wrapping-based composition at the time of writing.
REFERENCES


