

Customizable Software Component Run-time Model

Thanh V Lam, *I.B.M. eServer Group*, Poughkeepsie, New York, USA, thanhlam@us.ibm.com

Lixin Tao, *Computer Science Dept., Pace University*, Pleasantville, New York, USA, ltao@pace.edu

Abstract--Software components are specially built software units that can be reused most efficiently and independently from their development environments. Although specifications are mandatory, user of components cannot depend on how the components are developed because they cannot be tested or debugged like they are supposed to be in development environment. Improving development processes and software engineering are crucial but there are big gaps between development environment and run-time environments. This paper studies the inter-component interfaces and the capabilities of assembling components into component software on-the-fly at run-time. First, programmer will have to shift his attentions to run-time environments in order to minimize risks of application failures and to maximize programming efficiency. The Application Run-time Life Cycle Model captures and provides application contexts, which are combinations of component contexts. The model also has the ability of adapting to different run-time environments by packing necessary information into a Component Run-time Integration Box. The benefits are that, component reuse and the efficiency of component software are emphasized.

Index terms-- Application Runtime Driver, Application Runtime Life Cycle, Component reuse, Component Runtime Integration Box, Local Environments

I. INTRODUCTION

From the testing perspective, software applications have to be tested out in environments as close to end user runtime environments as possible. In other words, local environments are where software testing shows most effective and valuable. This paper proposes a software component model that is designed for local environments experiments. The ability of dynamic loading of pre-built components at runtime opens up new possibility for the user in testing those pre-built components while integrating them. The usefulness of this software component model can be extended for testing

those components in new environments without major rewriting of the test driver.

From the application perspective, extraneous code checking for runtime environments is necessary to ensure code execution. Some parts of the code cannot even run in a wrong environment. The concerns are especially on software that will be parts of the pervasive computing environments. It is difficult to decide on how much specific environment variables should be checked. This decision is in turn reflected in code complexity. In the model of application runtime lifecycle, some components now have a role in obtaining local environments at application configuration stage. Other components will just assume that the application has the local environment information available. The definitions of their role enable both types of components to be reused with other components pertaining similar roles. The main idea is that code which checks runtime environments and configurations can be separated from application code.

From the software component perspective, component software is made up of pre-built components that had gone through rigorous unit and function testing in development environment where they should have been integrated with some other components. That is called component integration test. This research suggests that a similar integration test is done in a runtime environment where could be the first time these components are put together, because development cannot possibly do integration test in all runtime environments. One concern would

be that the person who initiates this test may or may not be a professional software test engineer. She could mostly be the user who runs the application. Further more, the original developer(s) and component source code are presumably not available immediately. All the required validations should be “built-in” to the components or depending upon the user’s knowledge of the application.

The validations start from building up confidence integrating the components. Thus, the most commonly used components are more reliable. This research calls the application runtime lifecycle the most common component that no application can run without. It is also the simplest application in terms of features. Such component should be kept reused and only the application components are exchanged for different features. From the testing perspective, it is like a test driver. For the purposes of the application runtime lifecycle model, it also serves as a frame of references for the application components to “plug in”. Rather than depending merely on the component application programming interfaces, components now have to find ways of addressing their context dependencies. A simple example is that, a component that is loaded in a later stage of the application will require some resources or services obtained by other components in the configuration stage. Hence, the component integration test proceeds from most simple to more complex.

It cannot be over stressed the importance of communications and interfaces between components within the application. This research introduces another common component that will facilitate inter-component communications. It also keeps track of local environments and the stages of the application as integration going on. This is the part of the test driver that can be carried across different environments or “life cycle” of the application. It serves both the application components and the human user when there is need to log or trace back application information for the purposes of validating, testing, or debugging application.

In reality, software makers’ dilemma is that of generalization and localization. Although they would welcome the choice of making the application small and run in one particular environment locally, extra coding have to be added to make the application run in more than just one environment. That is the cost of generalization. All the related decisions have to be made before or during the time the application is developed. The application runtime lifecycle model presents the arguments and methodologies shifting those decisions into runtime when different components can be loaded based on the local environments.

A Simple Example: The Clock

When a particular application is designed or even just formed in the developer’s mind, it is usually not associated with any particular environments. In general, an application is a tool that mimics or reflexes a tool in real life. A clock application is obviously a counter part of the real clock that we use in every day life. We use clocks to tell time. The inventor who invented the original clock did not have any ideas that the world is divided into different time zones. The fact that a clock in New York tells a different time than one in London is not part of the clock’s functionality. And, the inventor was right! One could guess if the clock works correctly in one time zone then it will work similarly in another time zone. But one needs to bring the clock to the particular time zone to prove it. The clock is a classic case of an application that separates its functionality from the environments it is in. It is obvious that such small invention has lasted and is going to last beyond time and space.

This paper discusses related work in component reuse and its complexity. It also studies some recent work in component architecture, component adaptation and adaptive software architecture. The core contribution of this paper is the model of component software that can be used in testing the integration of pre-built components. Basic components of the model are described and the ways they work are explained with a simple clock application in the Component Software Integration section. In the Prototype section, details of how to implement the model and testing it is presented with a case study of some Queue and Stack components. This is the groundwork for future extensions in self-efficient component software assembled for adapting to runtime environments.

II. RELATED WORK

It sounds strange how the software world is divided and then the pieces are glued back together. But that seems to be the current state of software development. It is divided so that each software maker owns the little pieces for easy of developing and fast turn-around time to market. However, the finer the components are divided, the higher complexity when integrating them into full-blown application.

A. Component Reuse and The Complexity

Software component concepts promise that independently developed units of software can be reused at different situations of deployment. Binary code reuse has been around for long time but the complexity of the technology is hard to master [1]. Solving the general problem of component reuse presents challenges in searching for appropriate architectures and frameworks. Nevertheless, the book “Component Software, Beyond Object-Oriented Programming” [2] gives convincing arguments of the “inevitability of components”.

Taking a different approach, this research suggests a bottom-up solution: the user plays important roles in choosing components and integrating them at runtime. When the user’s choices are emphasized, software maker needs to shift focus to making the application run in different local environments without imposing extraneously complicated code. This fits the definition and purposes of software components. Using components, a complex problem can be broken up to smaller pieces for simpler solutions. The following graph is based on the software component spectrum from the above book [2] with respect to localization versus generalization.

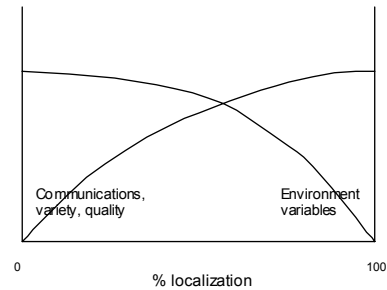


Figure 1 The spectrum of localization

If a component can assume that all of the *environment variables* it needs are either available or not available locally, its logic in searching and configuring those environments is reduced to minimum. However, a component usually does not run by itself. It needs ways of *communications* to other components what it has to offer in its interfaces and what it requires or depends on. The goal is to address environment variables and communications so that component variety and quality will increase. A component integration test driver is a good starting point.

B. Enterprise JavaBeans

Enterprise JavaBeans™ (EJB) [3] is one of the widely accepted architectures and standards for the development and deployment of component-based distributed business applications. It takes advantages of technologies such as Java language platform and the Java 2 Enterprise Edition. They present tremendous efforts in providing uniform frameworks and programming interfaces for server side applications.

The EJB complete solutions cover software development life cycle from application assembling to deployment. EJB components are contracted to communicate with the application server and bean container through programming interfaces. In general, the beans are assumed to run in some virtual environments. Therefore, component maker can concentrate on writing business logic or business processing type of components.

Although the user may not be aware of which system his or her application runs on, the loading

and executing of the beans occur on the server. The deploying environments have tight relationship with the server environments. Because servers have requirements for runtime resources, they depend on special kind of applications called clients. On the client side, the runtime environments are highly diverse and dynamic. It becomes difficult or impractical to depend upon the server to resolve runtime environments due to the spectrum of localization discussed above.

Further more, pervasive and personalized applications will be driving and changing their runtime environments. There will always be opportunities in introducing new types of clients or runtime environments. This research aims at solving the local runtime environment problem from the application and testing perspective. It takes advantage of Java and similar concepts such as EJB. However, the model starts from a bottom-up approach. The component software application is assembled from simple forms of components in a package. The application is not predefined as server or client. It does not require a server to run, although the server can be a source of delivering component packages.

C. *Binary Component Adaptation*

Problems in component communications are partly due to the facts that although many components are written with similar or identical functions in mind, there is no guarantee that their interfaces are compatible. Because these are supposed to be the component's own functionalities, no standards can force strict rules on writing these interfaces.

Based on the arguments that independently developed software components often cannot be used as is, without modifying source code, Binary Component Adaptation (BCA) [4] offers to solve the problems in component integration and interface evolution by modifying these interfaces in order to make them compatible.

Equipped with both off-line and on-line adaptation approaches, BCA can modify component binary code before it is loaded into memory (or the Virtual Machine in case of Java)

or when it resides in storage. Although the source file for the component being modified is not required, this is essentially for programmers who know what to do in changing those interfaces to make them compatible. A delta file has to be prepared and compiled. The good thing about the delta file is that it can be made as general or as specific it can be by the provided BCA-aware Java virtual machine. This is rather low-level software methodology.

Whenever changes are introduced into code, no matter how small they are, there is always a danger in breaking the code. It's inevitable to require some sorts of debugging tools to help with the accidental incorrect modification. These tools may require highly specialized programmers to trouble shoot at the programming interface level. This is more likely not the same user who runs the application.

It's difficult for user to deal with the programming interface aspects of the components. However, if these interfaces can be translated to factors in local environments, a user will understand better what's going on. Actually no one understands local environments better than the user. The research in this paper approaches the integration problems with user assisted debugging in mind.

D. *Adaptable and Adaptive Architecture*

While BCA deals with component code and programming interface level, this Adaptable and Adaptive Architecture is at a high level of software model called Viable Software Approach (VSA) [5]. It goes to the highest points of software concepts to include an Intelligent Control Paradigm. From there, meta-constructs can be added to describe templates of domain specific architectures. The viable component framework is a top-down view of components. The Viable Software Architecture could be most general unification in the software world.

More then ever, in these types of unification world, if possible, there are high demands for localization of many core elements in computing. The Smart Environments and Smart Camera Controller described are two cases in point. As

stated, “the goal is to explore and develop...” local environments are mandatory for these types of applications.

Smart room, smart spaces, and smart furniture in the SmartComponent WorkBench™ are object models of real world objects. These object models can be parts of the software components. However, many of the software components are not visible and cannot be dragged and dropped.

This research is another experiment and exploration into the abstractions of application workflows in light of software components and local environments. It is a bottom-up approach starting from the most common parts of software: the application.

E. Localization of Components

From the perspectives of assembling components into a running application, communications has to happen inside the application first. From the integration test perspectives, it is more than just the programming interfaces. Following are important factors for integration test:

- Pinpointing out the component under test
- Replacing a component without modifying either the integration code or the component code

When components are coming from different sources (independent software component makers), getting stack traces at error time may not be helpful because the source code is not available. Similarly, while the application is suspended for unknown reasons, attaching a debugger at runtime also requires the availability of the source code of the application.

Here is where the approach of loading components dynamically at runtime has advantages. Every time a component is loaded or a method is called for execution, a record can be kept, in the Component Runtime Integration Box (CRIB). These records make up the history of component loading and method calling within the application. However, the CRIB is independent of the loaded components or the called methods. These records are valuable for testing and debugging.

An example of global knowledge in software programming is the internationalization ability in some encoding standards such as Unicode that can basically display most popular languages of the world. However, an internationalized application rarely needs to display more than one language at the same time. Therefore, a localized solution is to figure out the “local” language at runtime. There are existing mechanisms in solving this problem such as providing each language in separate library and message catalog. Then, at runtime, a variable is set to point to a particular library and catalog. The component approach in this paper uses similar concept. However, instead of a form of library, the languages come in the form of components.

Based on the component reuse premise and methodologies, this paper suggests that software maker does not have to make all the decisions at development time. They can shift the decision logic to runtime while the user is having controls of the application. Then, decisions are made based on the user’s choices that can be ensured by on-the-fly integration testing. It is the best for both worlds where software maker decisions meet user choices.

This research uses localized environments to reduce complexity. One application needs not have the knowledge of all environments in the software world to run in a local environment. One person who is most knowledgeable in local environments is, not surprisingly, the user—not the programmer writing code in his own development environment. Hence, making good use of user know-how and local situations can reduce the complexity in component reuse and increase the ability in runtime problem determination.

III. COMPONENT SOFTWARE INTEGRATION

A. A Simple Example

Assuming that a clock application is to be delivered to three different systems. Each system requires a different language for usability purposes. The three different languages are: English, Chinese, and Italian.

One solution would be to write a single clock application with menus so that at runtime the user can choose the preferred language. Notice that these menus do not have anything to do with the

clock's functionality. They are just some means for a user to change the display context of the clock. In this case, the context is a certain language. Adding these menus no doubt increases complexity and leads to some drawbacks. If later, a forth system will need a different language, for instance, Vietnamese, the clock application needs to be modified. Code modification or white box code reuse is part of the component reuse problems. Quality testing gets the worst disadvantages because there are no ways to test a new language unless development has to add that new language into the menus.

This research experiments with how the application can be broken down to components from an application perspective. It is observed that every application has some common parts that the programmer follows when writing code without consciously knowing it such as the four parts of a program: initialization, configuration, operating, and finalization. These four parts, which have already been implemented in software platform such as the Java 2 Enterprise Edition, can be "globalized" in each application. The main idea is to match these stages with the practice of writing the "driver" program in testing. On the same lines of thought, components are like "localized" parts plugging into this "driver". This research discusses the issues of interfaces and context dependency between components.

B. *Application Runtime Life Cycle: An Analogy*

In programming, each programmer is the creator and programs are masterpieces. It implies that the programmer has total controls on what and how to make programs. However, that is probably only true in development. In real life, applications run in the absent of its creator: the programmer. Hence, the purpose of borrowing the runtime life cycle metaphor is to emphasize the application perspectives. In other words, applications are not broken down to components the ways the programmers want to. The runtime life cycle serves as the frame of references for components to be broken up and be plugged into the application.

The term life cycle has been used in software development very often though, to describe the software development and maintenance processes. It is actually the software maintenance churn cycles or software engineering processes. There have been big push in component-based software engineering (CBSE) which is believed to be "the way to produce software fast, with less effort, of high quality – not just the first time a product is released but for its entire life." [6]. Software engineering is in general a persistent

process among software makers. This research emphasizes the life cycle of the application runtime as appose to the life cycle of software development.

Application runtime life only comes to existence in memory for a period of time. When it ends, it completes one life cycle. It is expected that an application can run more than one times. For each run, it follows similar stages of operations but with different contexts such as data input or output. Therefore, like a human being who can learn from previous experiences, application can be recovered and restarted over after failures or termination. This concept suggests that application activities in the last run can be applied or corrected in the next run. Combining with the capability of component integrating at runtime and dynamic loading of components, there is the possibility of autonomic software or hardware in the future.

C. *The Four Runtime Stages*

The idea is, fundamentally, for the application to be assembled on-the-fly at runtime. It has to start from a simple stage and proceed to more complex stages. Following is a short description of the four basic stages of the life of a running application.

1. *Initialization or Arrival Stage*

At the beginning of this stage, the application comes into existence in the system memory. This is the simplest form of the application. However, it tells a lot about the application, for instance:

- ❑ Does this part of the code exist where it should?
- ❑ Is the code executable?
- ❑ How about code compatibility?

An answer NO to any of the above questions results in failure to start the application.

In the simple clock example, this is even before the clock starts. From the test perspectives, there has got to be some means of recording this situation

2. *Configuration or Growing Stage*

Like a baby quickly growing up to teenager or adult, the application in this stage loads components that do necessary configurations, such as:

- Opening input/output streams
- Connecting to networks
- Initializing graphic context

Note that the above items will be used later by other components in the application. There must be some kinds of communications for doing this. Also note that the items are all local to where the application is running.

In the simple clock example, either a standard output stream descriptor or a graphic context is initiated and stored in some place in memory so that the later components loaded in the operating stage can access to them.

3. *Operating or Matured Stage*

At this stage, the application loads the main feature components. This is the “main body” in programming terms. What happens in this stage can be very vague but this research is focusing on integration test. Here are the basic operations for testing the integration of components:

- Loading of a component
- Running a method provided by the component
- Repeat invoking methods of the component with different arguments

In the simple clock example, the local Unicode is passed into the clock component so it can display the right language without making any conditional testing. Also, as mentioned in the Configuration Stage, the clock component needs a graphic context or a standard output stream descriptor to display on. These two things are the clock component’s context dependency. It may also provide an interface method called “start” for example.

4. *Finalization or Departure Stage*

The end of this stage marks the termination of the application. It is another simple stage. However, like the Initialization stage, this is invaluable for testing and debugging the integration of components. Information about this life cycle that is about to end can be saved for the next life cycle use. This information can be:

- Did the application fail or succeed?
- What stage was prior to this Finalization stage?

- What components had been loaded?

These are test treasures, which a programmer never wants to deal with them unless the application fails, but then it would be too late.

In the simple clock example, it could be that the clock component failed to load the Unicode language. This answers the test question why did it fail.

IV. METHODOLOGY FOR TESTING

In Extreme Programming [7], the test cases are written even before the application code. Obviously, those test cases will fail first and then come back to successfully passing when application code is added. Most of these tests are functional or unit testing when or where the programmer is highly knowledgeable of the code he writes. This is not the case in component integration testing when the components are already pre-built and no modification to source code is allowed. What available at this point are the component interfaces and explicit context dependencies [1]. Figure 2 is an overview of the proposed runtime integration testing for software components.

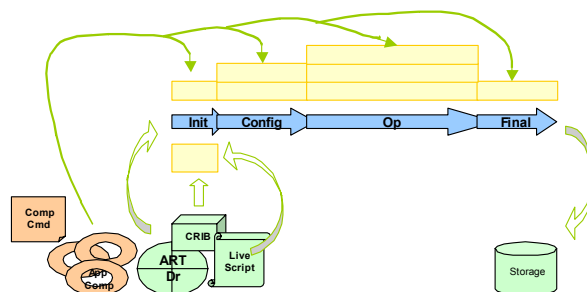


Figure 2 Overview of ARD and CRIB

A. *A Package Deal*

The deal of this package of software component integration testing framework is to include all the necessary components, and some optional components, for starting and testing the components within the application in the local environments. Two types of components compose the package: common components and application components.

- a) Common components are components that always come with the package. Except for the variety of inputs and the historical information that is previously saved, these components are the

same for any package. Two main components are:

- Application Runtime Driver (ARD)
- Component Runtime Integration Box (CRIB)

b) Application components are specific to the application. Some of these are minimum requirements for starting the application. Changing these components will change the application functionalities. Depending on how these components are written to communicate with each other, they can be thought of as:

- Tightly coupled (or closely interfaced)
- Open interfaced

The common components are the main features introduced in this paper. Hence they are described in details next. The application components are suggesting ideas. Component makers can write them according to suggestions based on the common components. The Prototype section includes some samples of these application components.

B. *Application Runtime Driver (ARD)*

An application is a unit of software that can provide services to the users, in general. Although there may never be a general or universal application, most applications have similar structures or organizations. A test driver represents such common patterns because it is in general written to test some parts of the application or functionalities of code.

1. *A Test driver*

A programmer writes a test driver to test his or her code in unit testing. The test driver usually starts from a very simple template that has been used for many times. Test cases are then gradually added when more code is done. At the end of the application development, the test driver becomes a full-blown test suite with all the test cases written for this application only. When the application is released to public, this test driver is abandoned or kept for some regression test purposes. Even with the same programmer, for the next application, the test driver will be started from the simplest template again.

With software components, the test driver can be written for reuse on testing any other code. Using similar ideas, a test driver has a simple code skeleton – a component that can be reused. As discussed in Section III, the test driver has four stages so that it can grow from the simplest to more complex test cases.

The flows of the test driver is automated as in a state machine which reads in inputs for

instructions on what components should be loaded and which operations should be executed.

2. *A frame of references*

Beside the benefits of reusability, the test driver also plays important roles in how the components can be held together. It serves two purposes:

- a) A time line of components executing: the components interface with each other through the time line, thus leading to component isolation. Sometimes when a component interfaces with another component, it is hard to pinpoint which end of the interface is at fault. The test driver is like a third party reference for tracing back to what happened.
- b) A relative location reference: where and what components should be plugged into the structures of the test application. It suggests that every software component written with a purpose in mind. In other words, a component is designed to work with one of the four stages. For example, an operating component will not work if plugged into the configuration stage.

3. *How it works*

The test driver starts at the Initialization stage by reading inputs telling it the name of the application. It may also reads in default local system information.

At the end of the Initialization stage, the test driver transfers to the Configuration stage. In this stage, those configuration components are loaded and registered.

Then, when all the configuration components are loaded and registered, the test driver transfers to Operating stage and reads in inputs. Components in this stage are loaded with contexts or services requirements that can come from the previously loaded configuration components.

Finally, when all components are loaded and the services are completed, the test driver transfers to the Finalization stage. At this stage, pre-built components can be loaded to do the clean-up and finalizing operations. Most of the time, the log of execution stages, which essentially contained in the memory (See descriptions of the CRIB in

Section IV.C) of the application, is written out to persistent storage for use in the next run. As part of the inputs to the Initialization stage, this persistent storage can be read in and analyzed.

Also note that, if the test driver ended abnormally in any of the previous stages, it always transfers to the Finalization stage to save stage or debugging information.

C. *Component Runtime Integration Box (CRIB)*

If the ARD is the skeleton, the CRIB is like the brain of the application.

1. *It is a concrete component*

The CRIB is a component containing the application contexts – things that help the application resolve different contexts at runtime. Following are the core things contained in the CRIB. However, at the moment, there are no limits on what entities can be in the CRIB.

- name: name of the application
- livescript: a list of components to be loaded according to stage
- register: for services etc.
- default setup

All those things are predefined or pre-built when the application is packaged, in the Live Script that the ARD will read in at beginning of the Initialization stage. The CRIB is a common component that is not changed. Instead, changing the Live Script changes its local environments or contexts.

The Service Register is part of the CRIB to provide a place for the Configuration components to communicate with Operating components. It is a table containing a list of services. As discussed in section IV.A, components can have closed interfaces or open interfaces. Open interface components use Service Register. Closed interface components may not need Service Register although it is available.

2. *With some persistent capability*

As described in Section B.3, at the Finalization stage of the test driver, application execution information is written into persistent storage. Since the CRIB contains part of the crucial

information that will be needed in the next run of the application, it can be saved to persistent storage in the Finalization stage.

3. *How it works*

When the ARD first starts with the Initialization component, it attempts to read the CRIB from persistent storage. If the CRIB cannot be found there, then this is the first time this application brought up in this environment. If a CRIB exists and is read in from persistent storage, it is compared with the CRIB that comes with the package. This is pre-application execution analysis. Different types of analysis components can be provided as Initialization components. At the end of this analysis, one CRIB with preferred environment information is loaded into memory. This is the application's initial CRIB.

For closed interface components, the package integrator has to ensure that the required components are loaded before their methods are called from other components. Configuration components are always loaded before Operating components. Even with closed interface, using of the components can be very flexible when combined with inputs and local environment variables.

For open interface components, the CRIB provides a set of application programming interface to facilitate the exchange or publication of one component's interfaces or other component's context dependencies.

V. IMPLEMENTATION

A. *Strategy*

Applications are valued and made good use of by their main functionalities. One application may include at least one function but usually would comprise of many functions. Notice that the term function is used in generic sense, not to mean the function as in a programming language. For an end user, functions are more or less units of deployment. Therefore, making and selling components to provide certain functionalities make more sense.

Three levels of skill are involved in making this approach work: component maker, package integrator, and component software evaluator.

For a simple starting point, those three types of work can be carried out by human. Then, the work can be gradually automated through tools. In either way, the critical mass of software components will increase because components can be made and deployed independently. Component quality will be raised. Testing in development environments, which is no longer sufficient because the running environments are no longer under the software or component designer's control [8], will be enforced. This posts challenges. But in return, end users will be happy because the application is highly customizable to one's preferences that play the critical roles in pervasive, mobile, or personal computing. The Application Runtime Lifecycle Model enables final “contextual design” [9] by users.

1. *Software Component Maker (Manufacture)*

The goal of the component makers is to produce as many components as they can. They make components for different platforms and different business functionalities. What they have to have in mind is that every component they make will be used in one of the four stages of the ARD described in section IV.B. They provide necessary information about their components that will be used by the component integrator.

2. *Component Integrator (Packaging)*

The goal of the component integrators is to take components produced by the software component makers and put into packages. To do that, they have to understand the two core components, ARD and CRIB, ins and outs. What the integrators do is to choose the appropriate components for their packages and create the Live Script for each package.

3. *Component Software Evaluator (Assembling)*

The goal of the component evaluators is to bring up the component software to their preferences using the components provided in the package, in a local environment. By customizing the inputs and optional components, an evaluator can also create simulated environments for integration testing purposes.

B. *Prototype*

The following groundwork is a very limited implementation of the concepts presented in this paper. However, it is a good start in a sense that such flexible model can be implemented from very simple parts or components. Some advanced concepts such as the CRIB's application programming interface require further research and much more resources for implementation. The Java programming language is used in this implementation for its rich collections of classes and available utilities. Java's ability of loading classes dynamically at runtime is also the first requirement for implementing this software component model.

1. *An Example of Stack and Queue*

Stack and Queue are two common data structures that are taught in basic programming classes. They are basically collection of elements. But each of them provides different ways of inserting and removing elements. Using the Java LinkedList class from the java.util package, a stack can be implemented as a class with two simple methods as following:

```
import java.io.*;
import java.util.LinkedList;

public class Stack extends LinkedList {
    public Stack() {
        super();
    }
    public void push(Object elem) {
        addLast(elem);
    }
    public Object pop() {
        removeLast();
    }
    public String toString() {
        return "Stack";
    }
}
```

The Queue can also be implemented similarly but with the method enqueue(Object elem) in place of push(Object elem) and dequeue() in place of pop():

```
public void enqueue(Object elem) {
    addLast(elem);
}
public Object dequeue() {
    elem = removeFirst();
    return elem;
}
```

To test the Stack class above, a test driver is written as following:

```
import java.io.*;
import java.util.ListIterator;

public class TestStack {
    Stack tst;
    public TestStack() {
        tst = new Stack();

        tst.push(new String("100"));
    }
}
```

```

tst.push(new String("200"));
tst.pop();
tst.push(new String("300"));

ListIterator stack = tst.listIterator(0);
while (stack.hasNext()) {
    System.out.println(stack.next());
}
}
static public void main( String[] argv ) {
    new TestStack();
}
}

```

Similarly, to test the Queue class, the exact same test driver can be reused. However, everything that is called Stack has to be changed to Queue. The two methods push and pop have to be changed to enqueue and dequeue. Furthermore, the test driver needs to be recompiled. The problem is that the data structure used in the test driver is hard coded. Although this can be said of some form of code reuse, it is not as simple with a huge test driver that loads many data structures and calls many methods.

An alternative implementation would be using Polymorphism or Interface in object oriented programming. Both Stack and Queue will be based on an abstract class or interface. A drawback from this implementation is that to introduce a third class, chances are the abstract class or the interface will have to be changed. This is not simple to do with a huge test driver either.

2. *Use of closed interface components*

The problems of testing Queue and Stack can be solved with the software component model in this paper. Since this is a very small example, the addition of extra components seems to be overkill. However, the additional components are common parts of and can be reused for any application. As discussed throughout this paper, this implementation emphasizes in separation of component's real functions from its environments or context dependency.

a) **The application**

Just as a quick introduction, the test driver, knew as TestStack or TestQueue above, now has become the following:

```

import java.io.*;

public class Application {
    private AppObserver appObs;
    private StateManger stageMgr;

    public Application() {
        appObs = new AppObserver();
    }
}

```

```

stateMgr = new StateManger(appObs);
stateMgr.activateInit();

while (!stateMgr.stopped()) {
    stateMgr.execute();
}
}
}

```

Notice that there is no mention about Stack or Queue. Instead, two new entities are introduced: AppObserver and StateManger. AppObserver is the code name for the CRIB and StateManger is the code name for the ARD.

b) **The ARD**

The ARD is implemented following the State pattern [10]. Each of the four states is implemented as a Java class. Notice that each state matches with the stage of application runtime life cycle. All four states extend the State abstract class. The StateManger class acts as a mediator that has access to all four states. Hence, StateManger can activate a state, transfer to next state, or execute a component in a state.

Activate a state: current state is set to this state.

Transfer to next state: the next state method of the current state is called so that current state will be transferred to the next state.

Each state class overloads these four methods in the abstract State class:

```

public abstract class RTState {
    public abstract int execute();
    public void tryNext() {}
    public void nextState() {}
    public void gotError() {}
}

```

The method execute() is the most important. It loads the component class file for this stage of the life cycle and execute the appropriate methods via this code segment:

```

Class classObj = Class.forName(appObs.appInput());
Constructor appInput = classObj.getConstructor(sign);
Object obj = appInput.newInstance(param);
Method filename = classObj.getMethod("filename", null);
Method reader = classObj.getMethod("reader", null);
inputReader = (BufferedReader)reader.invoke(obj, null);

```

For clarity and demonstration purposes, it is shown that the name of the component is read from the CRIB by the method appInput(). The code segment also shows that the method names are "filename" and "reader". In reality, these name should be read in from the Live Script as described in the next section.

c) Live Script

For ease of implementation, the Live Script is in the form of a text file with contents as following for this Stack and Queue prototype. The package integrator prepares this file.

```
Queue
ReadFile
IterateCmd
IterateOutput
```

The first line is the application name. In this case it is also the name of the component to be loaded in the Initialization stage. Therefore, the ARD will look for the Queue class and load it.

The second line is the name of the component to be loaded in the Configuration stage. In this case, the configuration service is to read in a file. It's known from specifications that the Queue component requires read in a file. The file name is given from the Queue object as seen in the code segment earlier. The component software evaluator prepares this file to test out the functions of the Queue component.

Local data I/O is one of the types of context dependency this research emphasizes. This is just to use closed interface between components. For open interface, line two has to specify the service that the configuration component provides, which is reading a file, and the filename it will read. For instance, the line can be like this:

```
ReadFile service:list
```

where `list` is the name of another file listing the public methods offered by the `ReadFile` component. In that list, each service, reading in this case, is associated with a filename.

The third line is the name of the component to be loaded in the application Operating stage. In this case, it is a command iterator that will access the service provided by the configuration component above. Since it is a closed interface, this component knows that every time it does a read, it gets a command with appropriate parameters. All it needs to do is to execute the command.

For open interface, the Operating component needs to specify its requirements in order to do its operations. In this

case, it needs to read in one command at a time. It keeps reading the next command until no more command available. The line in the Live Script can be something like this:

```
IterateCmd context:read
```

where `read` is one of the services provided by the Configuration components.

The forth line is also the name of a component to be loaded in the application Operating stage. Therefore, in this case, two Operating components are loaded and they are independent of each other. However, they use or require the same service provided by the Configuration component: `ReadFile`. This one is an output iterator that will print out every element in the data structure. Because this is a closed interface, it's assumed that the output iterator knows that some sort of print method is provided by the Configuration component.

For open interface, this Operating component needs to specify its requirements in order to do its operations. In this case, it needs to print out elements in the data structure. Line four can be something like this:

```
IterateOutput context:printall
```

where `printall` is a method provided by the Queue or Stack component. Hence, there may be another line in the Configuration stage like this:

```
Queue service:list
```

To sum up, to use open interfaces between Configuration and Operating components, the Live Script may look like this:

```
INIT: Queue
CONF: Queue service:list1
CONF: ReadFile service:list2
OP: IterateCmd context:read
OP: IterateOutput context: printall
FINAL:
```

In this case, there is no Finalization component to be loaded.

C. Test Scenario

Assuming a scenario in the Java Data Structure classroom, the instructor gives out two assignments:

1. Write a Stack class using `LinkedList` and test it with some input

2. Write a Queue class using LinkedList and test it with some input

For assignment 1, students write the Stack class and then write a TestStack class to test the Stack with some input. The instructor will run the TestStack application to see if the output is correct. If the output is incorrect, the instructor can return it to the student for a redo. But that doesn't help. The instructor may have to look into the TestStack program to see if any bugs there. If not, he has to look into the Stack class to see if any bugs there.

Similar activities happen with assignment 2.

With the software component model, the student is the package integrator. Although, he is also a component maker because according to the assignments, two components have to be made. Now, there are different level of student skills that can be considered for the assignments:

As a package integrator, students have all the components available to them. They will just have to look into the component specifications and pick the appropriate components. Then, they write the Life Script for the package and submit it to the instructor.

As a novice component maker, students write the application components such as Stack or Queue for the package integrator students to use in the package.

As an advanced component maker, students understand the common parts of the application such as the ARD and CRIB. They might even want to write their own.

The instructor is the component evaluator who doesn't have to look into the students' programs to determine if they are correct. She just runs the package submitted by the students. Or the instructor may want to play the role of the package integrator to grade the application components written by students. The assignments may become:

1. Write a Stack component for Package A
2. Write a Queue component for Package A or B

3. Write an application that incorporates the Stack or Queue component

The idea is that, students do not have to write an entire big application and the instructor can easily evaluate students' work. This is in academic world but the idea can be applied in real world as well.

VI. FUTURE WORK

A. *The CRIB With Open Interfaces*

As discussed in the prototype Section V.B, we have just touched the iceberg of the fully open interfaces between the Configuration components and Operating components. These interfaces will allow independently developed components to communicate via the CRIB. It is a two-way communications between two types of components that have clearly defined roles. This will help reduce the complexity of interfaces if components have to communicate with any other components. The CRIB serves as the mediator that can do additional verifications to assure that the interfaces are in good forms. Following are some suggestions:

- Verify the method signatures
- Validate input syntax
- Log of interface accessed or serviced
- Pinpoint the local component in case of a failure in an interface occurs

B. *Automated Observing and Controlling Tools*

The advantages of separating the common parts of an application into the ARD and CRIB is that now automated tools can be written to improve or enhance these parts without imposing upon the diverse application components. The software component makers are freed from constrains of these tools and can concentrate on making more components. These tools are not for component development although they can be used there as well. These tools are used at application integration time or runtime. Following are some suggestions:

- Tool to help the component evaluator generate the Live Script at the beginning of a run. This tool can also change the Live Script while the application is running by triggering

the ARD to reread the Live Script and do reset or reload of the components

- Tool to access information in the ARD and CRIB. Since the ARD is implemented as different states, this tool can determine the current state of the running application at any time during the application execution. Changing the information in the ARD and CRIB will indirectly change the execution flows of the application

C. Application Domains

The prototype of Stack and Queue in this paper is just one very simple example of how this software component model can be used. Based on the two factors in software component integration: local environment services and context dependency, applications in other domains can be integrated similarly. This will most likely be used in mobile and pervasive computing where client applications need to dynamically determine their local environments. There are also on-going efforts in the areas of autonomic computing. This software component model of ARD and CRIB could be a small contribution to the global schemes of self-efficient applications.

VII. REFERENCE

- [1] C. Szyperski, "Component Software and the Way Ahead," in *Foundations of Component-based Systems*, G. T. Leavens and M. Sitaraman, Eds. Cambridge: Cambridge University Press, 2000, pp. 1 - 20.
- [2] C. Szyperski, *Component software, beyond object-oriented programming*. New York; Harlow England ; Reading Mass.: ACM Press Addison-Wesley, 1997.
- [3] L. G.DeMichiel, L. U. Yalcinalp, and S. Krishnan, "Enterprise Java Beans (TM) Specification," Proposed Final Draft. Sun Microsystems Inc, 2000.
- [4] R. Keller and U. Holzle, "Binary Component Adaptation," *ECOOP'98 Proceeding, Springer Verlag Lecture Notes on Computer Science*, Department of Computer Science, University of California, Santa Barbara, 1998, 17 pages.
- [5] C. E. Herring Jr, *Viable Software, The Intelligent Control Paradigm For Adaptable and Adaptive Architecture*, Ph.D. Thesis Paper. Department of Computer Science and Electrical Engineering. Brisbane, Australia: The University of Queensland, 2001, 345 pages.
- [6] G. T. Heineman and W. T. Councill, *Component-Based Software Engineering, Putting the Pieces Together*: Addison-Wesley, 2001.
- [7] Extreme. Programming, "Extreme Programming A Gentle Introduction," <http://www.extremeprogramming.org/>, J. Donovan Wells, 2001.
- [8] K. C. Wallnau, S. A. Hissam, and R. C. Seacord, *Building Systems from Commercial Components*: Addison-Wesley, 2002.
- [9] H. Beyer and K. Holtzblatt, *Contextual Design: A Customer-Centered Approach to System Designs*: Morgan Kaufmann, 1997.
- [10] J. W. Cooper, *Java Design Patterns, A Tutorial*. Reading, Massachusetts: Addison-Wesley, 2000.