Visual Assembly of Software Components

Wm Leler
Victoria University of Wellington
wm@leler.com

Abstract

Many recent solutions to the problems of building large software applications are based on components and text-based Architectural Description Languages. These solutions are very promising, but they have been slow to attain widespread use. Visual programming techniques can not only make these solutions much easier to use, they provide significant advantages such as live editing. We have built a visual application assembly environment called AppComposer, which has been used to assemble large applications in such diverse domains as animation, multimedia, distributed applications, and Web apps.

Introduction

There has been considerable progress made recently on solving the problems of building large software applications, including easing integration issues and promoting reuse. Most of these approaches involve the use of an architectural description language (ADL) to specify an application as a set of **software components** and a set of **connections** between these components [ADL93]. Typical ADLs include Koala [Koala00], Jiazzi [Jiazzi01], and ArchJava [ArchJava02].

ADLs are currently text-based languages, layered on top of an existing programming language such as Java or C++. But it can be difficult to visualize interactions between components using a text-based programming language, so it is no surprise that papers on ADLs invariably use lots of diagrams to describe component interactions, and then translate these diagrams into text form. This begs the question: can we use diagrams directly instead of a text-based ADL?

In order to build a visual programming system for use instead of an ADL, we need to understand what ADLs do. ADLs extend current OOP systems by providing the "three C's" of component-based software:

1. **connectors** (*glue code*) so that components whose interfaces do not match perfectly can be connected without modifying the components,
2. **caller interfaces** (also called *call-out* or *requires* interfaces) so that the interfaces from a component out to other components are well defined, and
3. **compound components** (also called *hierarchical composition*) so that new components can be built hierarchically out of other components.

We set out to provide these same facilities visually, in an interactive environment called AppComposer.

We made an early decision to build AppComposer in Java, but within the Java world we wanted to be inclusive in what can be used as a component. AppComposer is able to use just about anything as a component, including JavaBeans, Enterprise JavaBeans (EJB), databases, Java Server Pages (JSP), Web services, and even regular Java classes. Note that the same techniques we used for Java could as easily be used with C# and COM/DCOM components.

Connectors and Behaviors

AppComposer needed a powerful visual programming paradigm, and it also needed to provide connectors (glue code) for components. We satisfied both requirements by using an actor-based visual programming paradigm (also called behavior-based programming) [Actor82]. Actor-based visual programming has been previously used in such systems as the Artificial Reality Kit [ARK86] and the commercial multimedia authoring tool mTropolis [mTropolis00].

In actor-based programming, you apply **behaviors** to **actors**. Figure 1 shows a simple animation created in AppComposer. The rabbit and happy actors are both animations that loop through a set of images (so they appear to be running). Notice the two behaviors called Jump and Drag, which have been applied to the rabbit.

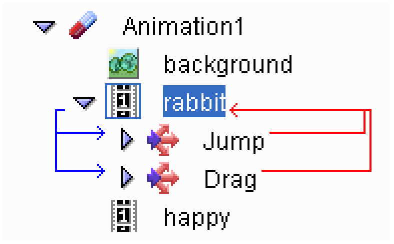


Figure 1: Animation with 3 actors and 2 behaviors

Another view of the animation is shown in Figure 2. The behaviors enable the user to drag the rabbit around the scene with the mouse, or cause it to jump in the air by clicking on it. The same behaviors can be applied to the other actors (the happy character, or even the background image) simply by moving (or copying) the behaviors. We can also use the same behaviors in other applications.

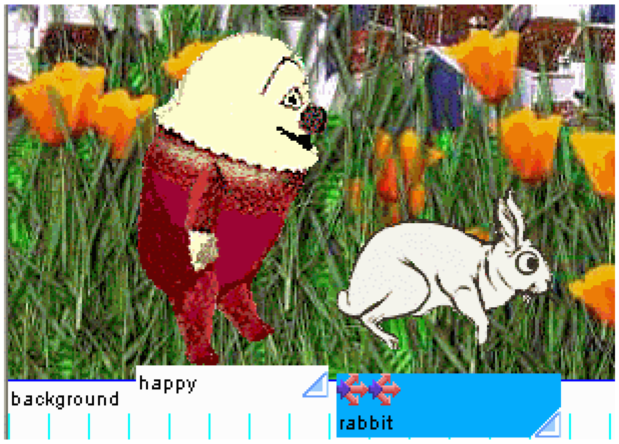


Figure 2: AppComposer visual editor

A key innovation of AppComposer is that it uses behaviors as connectors between components (the first thing we need for our visual ADL). This is more clearly shown in Figure 3, where the behavior named "start" connects the StartButton actor to the Juggler actor so that when the StartButton is pressed the Juggler starts juggling.

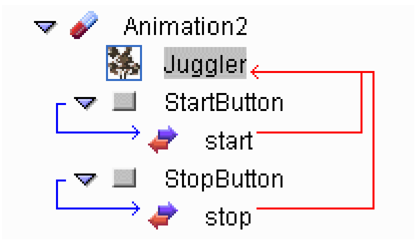


Figure 3: Behaviors as connections

Figure 4 shows the resulting animating juggler, captured in mid-juggle.

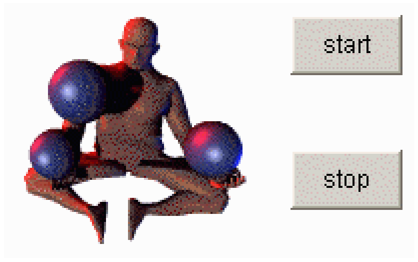
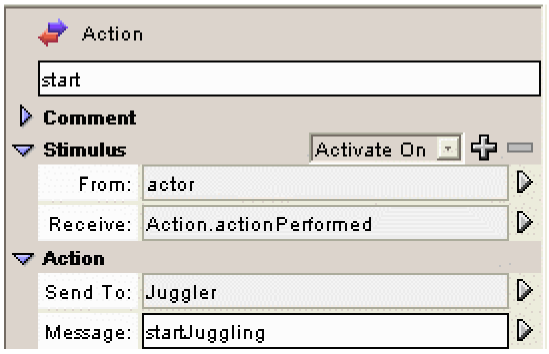


Figure 4: Juggler and buttons

AppComposer allows any component to be used as an actor, and (as mentioned above) AppComposer allows pretty much anything to be used as a component. In Figure 3, the StartButton and StopButton actors are regular Java AWT buttons. Nothing needed to be done to them to allow them to be used in AppComposer, even though they are not actually JavaBeans (or any other kind of standard component).

A behavior always has a **stimulus** and a **response**. The stimulus is an event, such as a mouse click or a button press. In Figure 3, AppComposer highlights the stimulus of a behavior using blue arrows, and the response using red arrows.

AppComposer includes seven primitive behaviors, and the user can define new reusable behaviors (in particular, the Drag and Jump behaviors from Figure 1 are user-defined). Primitive behaviors, like the "start" behavior from Figure 3, are defined using a dialog, as shown in Figure 5.



Action
start
Comment
Stimulus Activate On
From: actor
Receive: Action.actionPerformed
Action
Send To: Juggler
Message: startJuggling

Figure 5: The start behavior

For the "start" behavior, the stimulus is an event from the StartButton that says the button has been pressed, and the response is a method call (startJuggling) on the Juggler.

The response of a behavior can be a single method call, as in Figure 5, or it can be more complicated. The response of the Jump behavior (from Figure 1) is a sequence of method calls over time to move the position of the actor (rabbit) up and down. Behaviors execute asynchronously in parallel.

A behavior *always* represents a connection, even when it is connecting an actor to itself. The Jump behavior (in Figure 1) is connecting the rabbit actor to itself by wiring an output on the rabbit (mouse clicked) to an input (position).

AppComposer also allows the stimulus of a behavior to be a Java exception, which AppComposer treats like an event. This provides a convenient way to visually deal with program exceptions using the same behavior mechanism.

Caller Interfaces and Events

The second requirement of our visual ADL is the ability to define **caller interfaces**. Object-oriented programming requires the specification of an interface into an object (method interfaces), but allows implicit dependencies between objects through calls buried inside of objects to other objects, which severely limits reuse. ADLs solve this problem by requiring the specification of interfaces out of a component. Different ADLs use different techniques to implement caller interfaces, typically by adding new language constructs.

AppComposer uses the standard JavaBeans event mechanism for caller interfaces. This fits in naturally with visual programming, and has the added benefit of not requiring any new language constructs. Caller interfaces are not actually required in order to use a component in AppComposer, since there are many components out there that do not provide them. Of course, components that do not provide caller interfaces may be more difficult to reuse.

Compound Components and Capsules

In AppComposer, actors (components) and behaviors (connections) are assembled inside of **capsules**. For example, in Figure 3 the capsule is called "Animation2". Capsules themselves are also components, and can be used as actors in other capsules. This allows applications to be built up hierarchically, satisfying the third and final requirement of our visual ADL. The name "capsule" comes from the fact that capsules provide encapsulation (data hiding). Internal details of the capsule are hidden, and can only be accessed through the capsule's interface.

A capsule has a **type**. In the previous examples, the capsules have all been of type Application, but a capsule can be of a number of types, including a servlet, applet, JavaBean, or even an EJB.

A **servlet capsule** is a Java servlet that can receive HTTP requests and send back a response. Figure 6 shows a servlet capsule for a Web application named TimeServlet that receives an HTTP GET request and responds by sending back the current time as an HTML page.

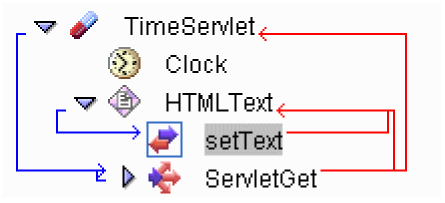


Figure 6: Servlet capsule (Web application)

The ServletGet behavior is activated when it receives a doGet event from the servlet capsule (indicating that a GET request has been received). It responds by outputting the value of its actor (HTMLText). The setText behavior is activated when its actor (HTMLText) is about to be output, and sets the text to be the current time.

AppComposer contains a built-in Web server and servlet container for building and debugging servlets. The servlet defined in Figure 6 can be accessed from any Web browser using the URL http://localhost/servlet/TimeServlet. You can even access the servlet from another machine.

Live Editing and Dynamic Compilation

A powerful feature of many visual environments is the ability to edit an application even when it is running. We wanted to provide the same **live editing** ability in AppComposer, but we realized this would be difficult since Java is a compiled, statically typed language. For application assembly, we absolutely require the speed and type safety of compilation, so we didn't want to resort to interpretive techniques. Our solution is **dynamic compilation**.

When part of an application has been changed, AppComposer saves the execution state of that part. This is done using serialization (although AppComposer does not require components to implement the Java serialization interface, since we want to be able to use just about anything as a component). The changed part is then recompiled, dynamically linked back in, and deserialized. AppComposer uses lazy compilation so that a changed part is only recompiled just before it is executed.

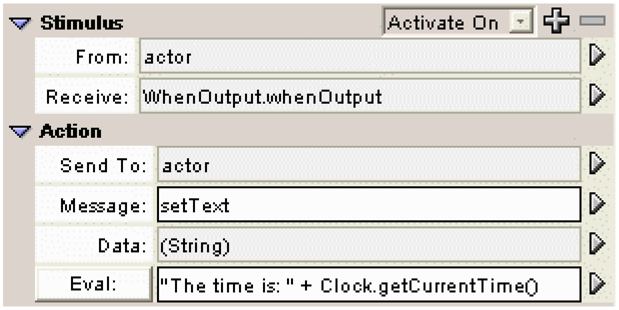
Live editing is one of the most powerful and popular features of AppComposer. Dynamic compilation provides the advantages of compiled code while also providing all the flexibility and responsiveness of an interpreted system. And because the output is 100% Java, no runtime system is required (beyond the normal Java runtime).

If there is a programming error, the error message is displayed along with the object that caused the error so it can be fixed. Compiled languages allow many errors to be caught and fixed at compile time (including type mismatches).

Java as a Scripting Language

Most visual environments provide a scripting language for those tasks that are more easily specified textually than visually. Usually such environments create a new (proprietary) scripting language, such as the Lingo scripting language, used by Macromedia Director.

AppComposer already implements dynamic compilation, so we were able to use regular Java as its scripting language. A huge benefit of this is that in any place where a value needs to be specified, an arbitrary Java expression can be used instead. For example, Figure 7 contains the definition of the setText behavior from Figure 6.



From: actor
Receive: WhenOutput.whenOutput
Send To: actor
Message: setText
Data: (String)
Eval: "The time is: " + Clock.getCurrentTime()

Figure 7: Java expressions in behaviors

This behavior sets the value of the HTMLText actor when it is about to be output (as part of a Web page). The value of the text is set to an evaluated expression – a constant string concatenated with the value of the current time returned by the Clock actor.

Note that while this example uses a snippet of Java code, the same result could have been obtained visually by using two HTMLText actors (one containing the constant string and the other set to the current time), but it was easier to just use a little Java code.

In addition to using Java expressions in action behaviors, AppComposer provides a **script behavior**, which makes it easier to include extended pieces of Java code, if desired. Note that using Java as a scripting language has nothing to do with the JavaScript language.

Scripting Support

In order to use make Java easier to use as a scripting language, AppComposer does two things:

1. Automatic boxing and unboxing of primitives. In AppComposer, primitive Java types (e.g., int) can be used interchangeably with their object equivalents (e.g., Integer). AppComposer inserts code to do any necessary conversions. This ability was subsequently added to the Java language.
2. Automatic method and class scaffolding. In Java, code must be defined inside a method, and methods must be defined as part of a class. Java expressions used in behaviors are not part of any method or class, so AppComposer wraps scaffolding around them so they can be compiled and executed, and obey scoping rules.

These features make Java much easier to use as a scripting language (apparently also as a programming language – we received an email from a teacher who told us he was using AppComposer for his introductory programming class).

Distributed Applications

The advantages of component-based design are especially apparent in distributed applications – applications that involve multiple processes that do not share memory. AppComposer supports the construction of several different kinds of distributed applications:

- Applications that communicate with a database manager, using SQL.
- Applications that use Enterprise JavaBeans (EJB).
- Applications that communicate with services, such as Web services using SOAP protocols.

A common design pattern in distributed applications is the use of proxies (or stubs) to communicate with remote components. Whether the database, EJB, or Web service is running on a computer half way around the world or just in a separate process on the same computer, the technique is the same. To communicate with the remote component, a proxy component is created. The proxy component handles all communication with the remote component, as shown in Figure 8.

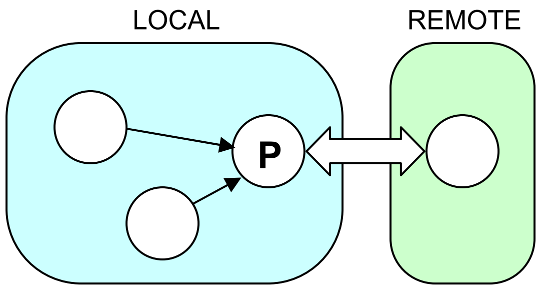


Figure 8: Proxy Component

AppComposer creates and manages proxy components automatically:

- For databases, AppComposer includes a set of components to talk to database managers. These components handle the connection to the database, and even generate SQL automatically so that most database applications can be written without any knowledge of SQL.
- For EJBs, AppComposer creates a proxy component that manages both the home and the remote interface to the EJB.
- For Web services, AppComposer creates a proxy component (using a standard WSDL description file) to communicate with the Web service.

AppComposer comes with a built-in database manger (HSQL) and an EJB container (the JBoss application server), which makes it much easier to create and debug applications using databases and EJBs. The resulting distributed applications will run on any Java application server and any SQL database manager. Because there is no standard execution environment for Web services (there is no way to even tell what language a Web service is written in), so you must either access an existing Web service, or set up the service manually, to use it in AppComposer.

AppComposer comes with a number of distributed applications, including a full e-commerce Web application for renting DVDs that uses EJBs, a Web application for buying and selling mutual funds that uses a Web service for accessing the current price of the fund, and a Web application for an online bug tracking system that accesses a database.

Web Applications

Web Applications are typically built using a multi-tiered approach that separates the presentation layer from the business logic layer. In the Java world, the business logic layer is implemented using components (typically EJBs), and the presentation layer is implemented using Java Server Pages (JSP), which are HTML pages containing embedded Java code [J2EE02]. Typically, the business logic layer is built by programmers using traditional programming tools, and the presentation layer is built by Web designers using HTML layout tools.

As with any application that is separated into components, glue code is required. Usually, one of two things happens:

1. In theory, the business logic components should be independent of the presentation, so the glue code should go in the presentation layer (i.e., the JSPs). The big problem with this is that it requires the designers to write glue code (even though they are not programmers). Even worse, they are writing code using a web page layout tool, not a programming tool.
2. In practice, the programmers end up writing the glue code and put it in their business logic layer. This makes the business logic totally dependent on the presentation. Even simple changes in the presentation require the programmers to modify their components. The resulting components are not reusable.

Glue code must be separate both from the presentation layer and the business logic layer. AppComposer does this by treating JSP files as components, and uses behaviors to implement the glue code between the presentation layer and the business logic layer, as shown in Figure 9.

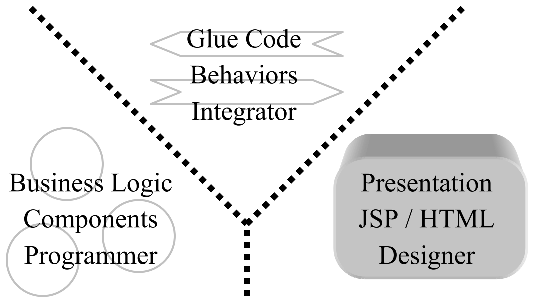


Figure 9: 3-tier Business Application

In this approach, the business logic layer is programmed using normal components, which are written to be independent of the presentation layer (and thus are reusable). The presentation layer is made up of JSP files containing only simple method calls. These calls are easy to create using standard HTML layout tools. AppComposer comes with a JSP tag library to make this even easier and with a set of extensions to Macromedia Dreamweaver to automate access to this tag library, so the designer never actually has to write any Java code at all. The glue code is written using AppComposer, and can be created either by the programmers, the designers, or by separate application assemblers.

The AppComposer approach to multi-tier applications is similar to the Model-View-Controller (MVC) pattern. The business logic layer is the model, the presentation layer is the view, and the AppComposer behaviors are the controller.

Reuse

Our experiences with AppComposer taught us several things about software reuse. First, it is much easier to reuse services and processes, rather than objects. For example, it is easier to reuse a chargeCreditCard process or a calculateShippingCharge process, than a CreditCard or Shipper object. This is because objects are domain specific, and domains change from one application to another. Despite all the claims and attempts, objects are just not very reusable. It is no surprise that Service Oriented Architectures (SOA) are more successful at promoting reuse.

Traditional attempts at reuse concentrated on the components to be reused, but that is only half the story. Attempts to use commercial, off-the-shelf (COTS) components have largely failed because of integration problems.

Buying or reusing a component can decrease the time spent building application functionality, but it increases the integration time, as shown by the second bar in Figure 10. The resulting integration problems can easily cancel out any gains from reuse. And since integration is the riskiest part of building large applications, it can make a large project more likely to fail.

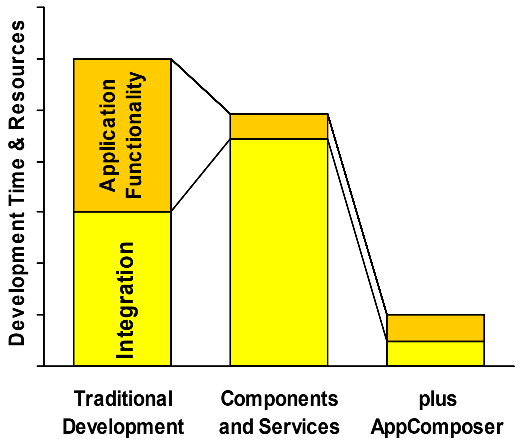


Figure 10: CBD and Integration

New technologies, such as text-based ADLs and visual application assembly tools like AppComposer can greatly reduce the time and risk of application integration, as shown by the third bar in Figure 10. Given the right tools and methodologies, software reuse is not as hard as people have been making it out to be.

Results

AppComposer has been used in several dozen enterprise software projects, including a major Web application built by several governmental bodies, a Web application that uses a spatial database to modify results based on the user's location, and an online bug tracking system.

Our experience over a range of projects is that the right tools and methodologies for performing component integration, along with reusable components, typically decreases application development time by a factor of four to ten times. The time savings tends to go up over time, as programmers get better at building reusable components, and build up libraries of good components. We also found that the resulting applications are much easier to maintain and modify. Unfortunately, we were only able to compare AppComposer against traditional development techniques, since we did not have any data for text-based ADLs.

Our experience confirms that application assembly benefits greatly from visual programming techniques. Application assembly tasks can be tedious and error prone, and are easier to perform visually than using a text-based solution. We also found that the ability to show the customer what the application will look like during development – and make changes based on their feedback – greatly increased the chances that the final application would meet their expectations.

We originally targeted AppComposer at non-programmers, such as artists and web designers. We were surprised to find that programmers like to use it just as much, if not more. In a way, we think of application integration as similar to building a financial model using a spreadsheet – you could do it using a text-based programming tool, but even programmers prefer using a visual tool. Not only is application integration a primarily visual task, but there are also benefits from being able to see the result of changes immediately.

AppComposer helps storyboard applications, and gives project teams that are often composed of people with diverse skills a common language to talk to each other. For example, since graphic designers and programmers can both use AppComposer, they use it as a way to toss ideas back and forth. AppComposer works especially well in projects that use agile (extreme) programming techniques.

The original idea for AppComposer was conceived at a workshop at the Banff Centre for the Arts in the mid-90s, and we have always been interested in the use of AppComposer by artists. We also found that artists have a keen interest in building Web applications. They have learned how to build static web pages using HTML, and now want an equally easy way to build things that are dynamic or interactive. AppComposer has been used in a wide variety of such projects, including generating graphics for a major motion picture ("All the Rage").

And lastly, AppComposer is lots of fun to use!

References

[Actor82] Roy J. Byrd, Stephen E. Smith, S. Peter deJong, "An actor-based programming system," *ACM SIGOA Newsletter*, June, 1982.

[ADL93] David Garlan, Mary Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering, I*, Ambriola V, Tortora G, (eds.), World Scientific Publishing Company, 1993

[ArchJava02] Jonathan Aldrich, Craig Chambers, David Notkin, "ArchJava: connecting software architecture to implementation," *Proceedings of the 24th International Conference on Software Engineering*, May, 2002.

[ARK86] Smith, R.B., "The Alternate Reality Kit: an animated environment for creating interactive simulations," Proc. 1986 IEEE Comp. Soc. Workshop on Visual Languages, Dallas, Texas, June 1986, CS-IEEE, Los Alamitos, Calif., pp 99-106. Also reprinted in Computers and Learning, Boyd-Barret, E. Scanlon, (eds.), Addison Wesley.

[J2EE02] *Designing Enterprise Applications with the J2EE™ Platform, Second Edition*, Sun Microsystems, 2002. (http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/)

[Jiazzi01] Sean McDirmid, Matthew Flatt, Wilson C. Hsieh, "Jiazzi: new-age components for old-fashioned Java," *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, October 2001.

[Koala00] Rob van Ommering, Frank van der Linden, Jeff Kramer, Jeff Magee, "The Koala Component Model for Consumer Electronics Software," *IEEE Computer*, March 2000.

[mTropolis00] Glen Hunter, "mTropolis, the Greatest CBD Tool that Never Was," Mojo Productions, Inc., 2000. (http://www.cbd-hq.com/articles/2000/000501gh_mtropolis.asp)