Dynamic Architectures

André Augusto Cesta
Department of Computer Science
University of Auckland
aacesta@cs.auckland.ac.nz

John Hamer
Department of Computer Science
University of Auckland
J.Hamer@cs.auckland.ac.nz

Abstract

Software frameworks that incorporate a dynamic component have become increasingly widespread in recent years, in response to demand for more flexible and extensible systems.

In this paper, we survey the main alternative approaches to the construction of dynamic architectures, and propose a new variant called "code incorporation". The architectures presented rely on extension languages, active object models, and other approaches.

1. Introduction

Many are the forces that foster the development of software towards flexibility. This paper addresses the question "How can a system be made flexible?" by looking at techniques for building systems that change through some form of programming or dynamic extension. Such systems implement features borrowed from compilers, interpreters and programming environments. We survey these dynamic software architectures (*3. Survey*), and propose a new one (*4. Code incorporation*).

A system that provides a built in programming language has many capabilities. Such a system can change itself during runtime, thereby becoming more adaptable. This helps in the construction and use of frameworks, by allowing the system to be extended, configured and programmed. Prototyping is facilitated, as is the generation of new or customised systems. Further, testing and debugging facilities can be provided in a systematic way.

A programmable component provides for direct communication between separate modules in a package by allowing arbitrarily complex objects within the framework to be exposed and manipulated. This can be applied to bridge other software interfaces, and to extend the set of libraries available to the system.

The needs of advanced users can be satisfied through a programming interface, in the form of text scripts. More complex visual programming tools (wizards, macro programming tools, etc.,) can be built on top, using the programming interface as a target layer.

There is a natural fit between systems that provide a built in programming language and *frameworks*. A framework is a system that is designed to be changed. It can be adapted to work in many instances of the particular domain it was created for, thus allowing reuse of analysis, design and code.

When a framework evolves, its parts tend to form a domain specific programming language. At this stage the programmer is not building many new methods, but *combining old ones*. Programming now takes place at a higher level of abstraction built over the programming language used in the system. The syntax used may be the one provided by the underlying programming language, but the elements operated over are the ones created or imported into the system. At this point, the creation of an explicit extension language is indicated.

An extension language is a language used to extend a particular piece of software. The extension language is like a programming shell, and indeed many shell like or interpreted languages are used for this task. The term "glue" is sometimes used to describe such languages, as they serve to connect software components.

Well known examples include AutoCAD, a computer aided design tool that provides an extension language called Autolisp. Many AutoCAD actions can be executed either through mouse movements or through Autolisp commands. New modules for AutoCAD are written by third-party developers using this language as the sole interface. The software used to write this paper, GNU-Emacs [16], is a programmable text editor that has an extension language named "elisp" [17]. The case tool used to build the UML [15] diagrams in this work, ObjectDomain, uses Python [18] [19] as an extension language.

The techniques used to create extension languages are similar to those of programming language construction. However, since we are changing a working program, many of the approaches will rely on the theory of structure and representation of computer programs. The architecture of the extended program and the architecture of its execution environment will play an important role in the construction of this language.

2. Issues

When studying a dynamic architecture many issues must be considered. Most of them are related to how the dynamic component works, rather than the form of the dynamic component. The form can range from extension languages and embedded interpreters, through to integrated software components or even single modules.

System visibility How much of the system can the dynamic component see?
Binding time Does this dynamic extension happens at compile time, runtime or both?
Dynamic component visibility Is the extended framework able to access the dynamic component or not? How?
Flexibility How flexible is the dynamic part? Is it a late bound component that modifies the system? Is it a full programming language or a restricted language?
Knowledge How much knowledge about the framework is built into the dynamic component? This is different from visibility.
Introspection How much can the dynamic component learn about the system and vice versa.
Adaptability How does the dynamic component adapt to changes in the framework?
Efficiency Is the dynamic component efficient on its creation and execution?
Communication Is the communication between component and application efficient? Is marshaling (converting data formats between disparate systems) an issue? What mappings are necessary to communicate component and extended system?
Security What security issues must be addressed?
Applicability What are the different ways of deploying a particular approach? What is the leverage and scalability achieved?

3. Survey

In this survey section we discuss previous results in making frameworks more flexible through the use of programming languages and other dynamic components. First, we introduce architectural variations. Next, we provide some background on the Java platform that is used in the examples. Last, we provide a set of sub-sections, each one describing a particular architecture variation in terms of the issues presented.

3.1. Architecture variations

Some languages, such as Lisp [1] and SmallTalk [20], can be extended directly. In other cases, the extension language may be different from the language used for the main system. For example, the ObjectDomain case tool is built in Java, but its extension language is Python.

The extension language may be a pre-existing, standardised language (such as Guile [21]), a language dialect (such as elisp), or a totally new language specialised for the system.

Besides all these design possibilities, there is also a range of different environments for program execution. An extension language may be compiled or interpreted, or a hybrid (exemplified by Java, that is half compiled and half interpreted).

Other approaches that do not even rely on extension languages can be used to make a system flexible. There are particular object models that allow the creation of dynamic components, such as UDP [11]. We discuss this approach later.

3.2. The Java language

Java [9], the language chosen to demonstrate most surveyed approaches, is not a typical programming language. In order to obtain portability with efficiency a Java program is compiled into an instruction set known as byte-code which is then interpreted by the Java Virtual Machine [12]. The Java Virtual Machine is not an usual interpreter: it provides a full operating system independent environment for programs to be represented and executed.

A Java program is not a monolithic block of executable code. It is composed by many compiled class files that are read and interpreted by the Java runtime or Java Virtual Machine(JVM) when it is started. The Java class loader is the part of the JVM responsible for adding each class in a Java program.

The Java Virtual Machine provides code security verifications, late binding of classes, garbage collection and computational reflection. All these make Java an interesting language to study dynamic architectures.

Late binding of classes enable us to add new classes to the program namespace during runtime. Late binding allows use of synthesized code (program generated code) and incorporation of this code into the system during runtime.

ExampleConsider a class named `AvailClass` that is already available in the program namespace. The normal way of binding to this code is shown in figure 1.

```
import APackage.AvailClass;
...
AvailClass vEC = new AvailClass();
System.out.println(vEC.toString());
```

Figure 1. Normal Binding of a Java method

Late binding occurs when the class is not known to the compiler. The Java code to achieve this is shown in figure 2.

```
1 Class vNC = Class.forName("NewClass");
2 Object vLtBind = vNC.newInstance();
3 System.out.println(vLtBind.toString());
```

Figure 2. Late Binding of a Java method

Line 1 declares a `Class` object to store a late bound class named `NewClass`. The `forName` call locates the class in the file system and loads it, binding the class into the program namespace. It also returns its representation in the form of a `Class` reflective object.

Line 2 creates the `vLtBind` object to store an instance of the class obtained through a call to the `newInstance` in the object representing the bound class. Now, `vLtBind` could be typecast to one of its base types if they were known, or the `Object` reference used to introspect its methods and invoke them. Line 3 in fact uses the bound object directly, through the `toString` method provided by the `Object` class.

Late binding of classes poses security threats to systems. Late bound code can come from untrusted sources: the extension language, the users, or the network. This is where Java broad variety of security mechanisms and APIs comes to rescue [8] [6] [7]. There are many resources that may require protection from undesired access.

System resources System resources are files, network ports, printers. Access to these resources is easily restricted by means of security policies configured on a per class basis using the Java security API.
Classes directly known Classes directly known are classes for which the script has a Java reference. Access to these classes is restricted by enforcing data hiding and carefully choosing which references are passed to the script and which other references can be indirectly obtained through the ones passed.
Classes imported Classes imported are those for which the system can directly create an instance. Late bound components can be created with import restrictions.
Classes loadable Classes loadable are classes for which the script can perform a late binding and instantiation. This is restricted through the `classpath` variable, and through restriction to access of directories.

By splitting the compiler into two parts, Java gains flexibility and portability. But it also loses some performance and information.

Every time code is compiled, a lot of important information is thrown away. This information is often the kind of meta-information that could be useful in dynamic models. This is where the Java reflection API and the relatively high level of the program representation provided by the Java virtual machine can help.

The introspection mechanisms implemented by Java enable one class to query other class for its method signatures and fields during runtime. This is important when you modify the structure of a system and its objects need to be recoupled, or when you have to interface different namespaces.

The reflection API works by providing information about the running code in the form of data. Java provides classes for: `Class`, `Object`, `Constructor`, `Method`, `Field`, `Modifier`. The method call in figure 3 exemplifies how Java introspection works. Line 1 creates a `Method` object named `m`. It receives the result of an introspection for a method. The method that performs the introspection is called `getMethod`. `getMethod` takes two arguments that are sufficient to define an unique method signature: `mnm`, the method name, and `types`, an array of types the introspected method should expect as parameters. `getMethod` introspection operation is called over `vLtBind`, which is an instance of the `Class` class obtained previously to represent a particular object class.

```
Method m = vLtBind.getMethod(mnm, types);
retValue =  m.invoke(obj, args);
```

Figure 3. Reflective method call in Java

Line 2 performs the `invoke` method call over `m`, passing as arguments `obj` that is of class `Object` and an array of `Object` arguments, `args`. `returnValue` is of `Object` class, because the return type varies from method to method and only the `Object` class is able to receive instances of any type.

The degree of reflection provided by the language will be of importance when coupling a dynamic component. In Java, one can introspect objects to find out their classes and introspect classes to find out their fields, constructors, methods, modifiers. It is also possible to verify if a class implements one interface or inherits from a particular superclass. But introspection of method bodies is not possible, so we cannot ask: "What methods does this method call?", or "What is the source code for this method?"

3.3. Interpreters and BeanShell: a reflective approach

Extending frameworks to support a domain specific language might be as simple as implementing a configuration file parser or as complex as the construction of a new programming language, with its grammar and interpreter. An interpreter can be run by the framework, reading the instructions in the domain specific programming language, providing an intermediate representation for them, interpreting them and generating actions over the program representation. If the interpreter is built into a system for its extension, then an interface is usually available to allow communication with the system.

The BeanShell [13] is an example of the Interpreter approach.

> "BeanShell is a small, free, embeddable, Java source interpreter with object scripting language features, written in Java. BeanShell executes standard Java statements and expressions, in addition to obvious scripting commands and syntax. You can also script objects as simple method closures like those in Perl and JavaScript(tm)" [13].

The BeanShell does not interprets class declarations, but can use pre-compiled Java classes. It allows dynamic creation of arbitrary methods and scripts. The interpreter acts like a single extendible class that can gain methods, attributes and execute arbitrary code. It has extended the Java syntax, providing a very loose typing scheme when constructing and using BeanShell methods and objects.

```
1 bsh % myObj = object();
2 bsh % myObj.foo = 42;
3 bsh % myObj.bar = "bla";
4 bsh % print(myObj.bar);
5 bla
```

Figure 4. BeanShell script

The script in figure 4 shows how loosely typed became this Java language variation implemented by the BeanShell. BeanShell code usually looks like regular Java code. `bsh %` is the BeanShell prompt. Line 1 creates a BeanShell object. At this time the field `foo` does not exist, but line 2 declares and assigns a value to it. The same is done for an attribute of other type in line 3. Line 4 uses a BeanShell method called `print`.

The BeanShell is interesting because it uses Java to create a dynamic Java interpreter. This has a number of advantages, principally the easy sharing of a name space with the system, and avoiding bridging two different languages.

The Java grammar used is a scripting adaptation of the Java language grammar provided by Sun Microsystems. The BeanShell makes use of computational reflection to integrate the interpreted code with the compiled code of extended systems.

The example in figure 5 shows the declaration and instantiation of a Java object within the BeanShell.

```
bsh % String aStr = new String("BShell");
```

Figure 5. Creating a Java object in the BeanShell

When the BeanShell parses this construction it creates an intermediate program representation where the top node is an assignment and its two child nodes are: a declaration and an allocation. In these nodes it stores the names for the type declared and allocated, together with other information such as the constructor arguments. The BeanShell never generates byte-codes; it always represents the parsed script as an intermediate program representation.

When the BeanShell is asked to evaluate this program representation, it knows that `String` is a compiled type, with a representation available in byte-code. It obtains a `Class` object for the `String` class and uses introspection to retrieve the desired `Constructor` class from this object. Then it invokes the introspected constructor, passing the argument references that were earlier evaluated and converted from intermediate program representation to Java references. The result of the allocation is stored in the current interpreter namespace under the name “aStr”. If the `String` type before `aStr` was omitted, the BeanShell would infer the type from the right hand side of the assignment construction.

3.4. Kawa

Kawa [2] is a full Scheme [14] implementation, completely written in Java. Scheme functions and files are automatically compiled into Java byte-codes [3]. Kawa compiler optimizations achieve performance levels between interpreted and compiled Scheme.

Recall that the BeanShell would never generate byte-code. Kawa takes the approach of an interpreter, but to obtain more efficiency and portability it compiles into byte-code.

Kawa does not generate Java code and then compile it into byte-code. Nor does it map Lisp to Java and then Java to byte-code. Rather, it generates byte-code directly. Nevertheless, some of the Lisp primitives (such as the list and the pair abstractions) are programmed in Java and are always loaded into the virtual machine before the Kawa starts running.

Kawa proposes some different ways of structuring applications. In its server mode, Scheme expressions come through the network for evaluation. Kawa can also deploy Scheme scripts as stand-alone, portable, byte-code applications.

Kawa is a successful compiler of a particular language to byte-code. This compilation might not be so easy for some other languages because the byte-code was designed specially for Java language.

Kawa opens the door to integrate two languages, Java and Scheme. This integration can happen not only during compilation, but during runtime, at software component level.

3.5. Extension languages

There are many attractions to using a tested interpreter for a widely used language like Perl [22] and bridging it to the system. Now, the work is concentrated not in implementing the new language, but in bridging it. This can be done using dynamic libraries.

The most important step of this approach is defining the interface between system and the extension language. This interface should not change frequently, it should hide details of the system implementation, and provide a security barrier.

Languages usually employed as extension languages are: Python, Perl, Tcl [23], Guile. Security is not a big issue, because the extension language accesses the system through a narrow interface, defined by the developer. However, as the language and framework will almost certainly adopt different datatypes and representations, marshaling and unmarshaling become an issue. While communicating primitive datatypes is usually straightforward, it may be more difficult working with complex objects.

A lot of knowledge has to be built in the extension language to interface with the system. The extension language cannot obtain more information about the system, and the system usually cannot call back to the extension language or obtain information about the dynamic component unless these lacking features are explicitly included in the interface.

3.6. RADs

RAD stands for Rapid Application Development. RAD tools typically provide a graphical application development environment with compilers, editors, debuggers and GUI construction facilities for the programmer.

These tools are mentioned here because most of them generate code for graphical user interfaces designed visually, they can compile and run this code (as a separate process). However, they are not as dynamic as other architectures discussed, since code is only added to the system during compile time.

RADs do more than generating code; they manage the generated code, keeping it synchronized with its graphical representation. Some of them even allow the programmer to edit the source code manually.

3.7. BeanBox

The BeanBox [10], provided by Sun Microsystems, is an example of a Java tool that makes use of a compiler during runtime. It can have Java classes generated, compiled and loaded during runtime.

It is a small demo program, created for helping the development and configuration of Java Beans, the default software component model for Java. It allows users to customize and deploy Java beans and small programs involving them, accessing an extensible library of software components. The BeanBox is able to introspect, configure and combine these components. To combine two or more components through their interfaces, the BeanBox is able to generate an "adaptor" class, compile, test, and deploy the components combined through the generated adaptor.

The BeanBox uses computational reflection techniques to configure software components built in the Java Beans standard. For instance, if you add to this tool library an animation component, the tool can find out what configurations it accepts and allow the user to configure its properties graphically. Obviously the component has to be build using some techniques and conventions that will allow it to be introspected, it must provide much meta-information about its properties and behavior.

It is much harder to build a Java Bean that has properties of complex datatypes, because for the configuration, these datatypes must be constructed. When you have such complex properties in your component, you have to provide an editor that enables the configurator to graphically generate instances of the complex datatype. These generated instances are then used for configuration.

Suppose you add to the Beanbox an animation component. The tool might allow you to configure the component in terms of the image sequence it animates, the frames per second rate used, etc. After that the configured component can be serialized and stored for future use in a particular software, or combined with other components to enhance its features.

The BeanBox uses synthesized code and computational reflection to wire components together using the Java event model. Let's suppose one wants to build a computer animation that can be paused and restarted by pressing correspondent buttons in the animation GUI. The animation component available in the BeanBox demo is able to be started and stopped, but does not provide the buttons to do so in its GUI. One can add these buttons in the form of components linked to the animation. The BeanBox tool allows the user to drag and drop the components visual representation in the screen and connect the button pressed events introspected with the `start` and `stop` introspected methods in the animation. This is done by means of an adaptor that listens to button pressed events and fires the corresponding introspected method in the animation component once the event is received. The BeanBox also allows you to link introspected events to introspected methods that take a single argument that is of that event type. Or to introspected methods that take no arguments.

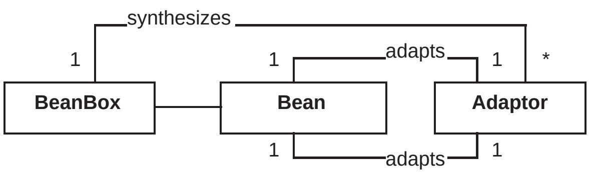

Figure 6. Reduced class diagram for BeanBox

The diagram in figure 6 shows how the Adaptor design pattern is applied by the BeanBox. The adaptor intermediates the association of two Beans.

The BeanBox code does not change during runtime. Rather, its library of software components changes during runtime by gaining new elements. The code added by new components is reused in new component combinations that can be deployed and tested by the BeanBox. The BeanBox might call component methods to test and exhibit them, but they do not act upon the BeanBox.

3.8. UDP—User defined product framework

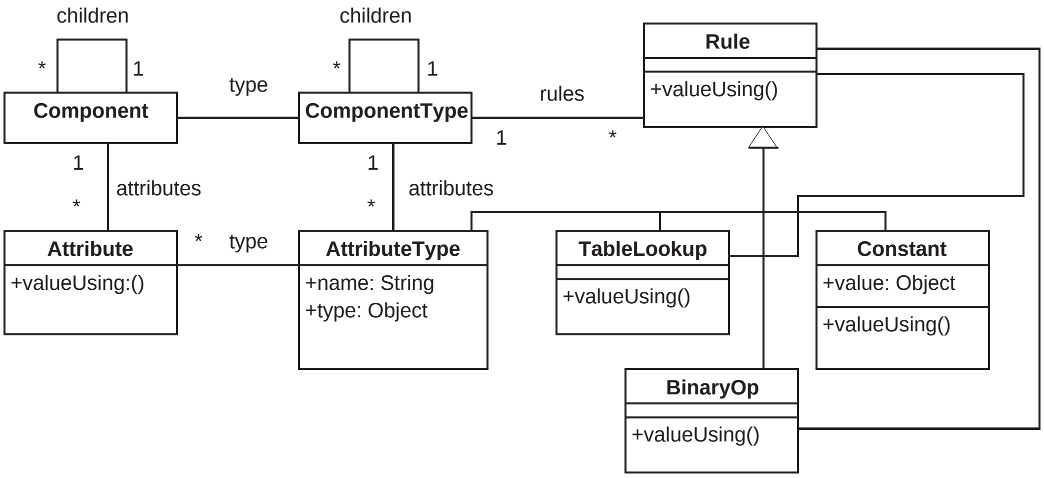

Figure 7. Class diagram for UDP

All of the architectures proposed are flexible because they generate, process or incorporate code. The UDP [11] is an example of an architecture that is flexible because it can change its program representation in memory, without the use of a compiler or interpreter module.

The kind of flexibility needed by the UDP user is being able to define new datatypes for commercial products during runtime. One might think about using the previous compilation and interpretation approaches to follow on creating new datatypes while the program is running. The UDP does it differently, it uses a flexible program representation.

The UDP key concept is making types of products and their attributes become data instead of classes. This framework has separate classes for representing: product types, product instances, product properties, property types. This avoids having to write new classes on the event of a new product creation or recompiling a class when it gains one attribute.

The "User Defined Product" or UDP uses metadata and design patterns such as the Type Object, Interpreter and Property [5], to create an active object model.

> "An object model is an abstract representation of a particular domain, using objects as the description. An active object-model is an object model whose object representation is interpreted or generated at run-time and can be changed with immediate (but controlled) effect on the system interpreting and running it" [4].

The UDP interprets its own dynamic object model which represents not only objects and classes, but also rules (a data representation for simple methods) that can be changed during runtime.

An important issue when building a framework similar to "The User Defined Product" is to have the object model data stored and interpreted in a way that it can be easily changed. On our implementation of the UDP framework we used Java serialization mechanism for that. This way the user can build layer upon layer with the UDP. The products can be reused and also transfered between systems.

The UDP class diagram is not a regular object model for representing products, because you do not see classes with product part names such as: panel, nuts, wheels, etc. Is a meta-object model. An object model that is able to generate some other object models.

On the left side of the diagram you have four classes that model the concept of object and object type, their attributes and the attribute types. On the right side you have an hierarchy of classes that will enable you to construct small intermediate program representations for operations over the products defined on the left. These operations might involve constants, table lookups and binary operators only.

The the `Component` and `Attribute` classes implement a design pattern known as properties. The `Component` class stores a particular product part instance. The product attributes are not represented as fields in the `Component` object, but as multiple instances of the `Attribute` class. `Components` might also contain sub-components (children) association.

With only `Component` and `Attribute` we can build arbitrary products, but their hierarchies and instances are type-less. For creating a type for these, we create parallel classes to store their types and aggregation structures. The hierarchies constructed (instance and type) can be superposed and replicated (each instance over its type, or each instance with a reference to its type). This is an application of the type-object design pattern. The `Rule` inheritance hierarchy is simply for constructing simple interpreters that operate over the product representations.

4. Code incorporation

This paper proposes a different approach to the creation of extension languages and active object models, we called this new approach: "Code Incorporation Approach". They key concept of the "Code Incorporation Approach" is: instead of building a domain specific language with its grammar and interpreter inside the framework, make the framework generate the code that will modify itself and then, during runtime, add this code to the framework.

There is a difference between changing program code and changing program representation. Every program changes its representation during runtime. In usual programs and machines, the program counter is constantly changing, the data-structures in the heap are modified, deleted, etc. The stack segment can grow and shrink. But these usual programs do not learn about new data-structures, types and algorithms, they simply do what they where programmed to do and change in the way they were programmed to change. They are changing representation and not code.

It is rare to have a program change its code. Some environments do not allow programs to change because of a design choice or of security restrictions. Nonetheless, programs that change their code have been around for decades and have been used in many areas. Early assembly languages and operating systems could allow a program to write in its text segment residing in memory. Some microprocessor can have their microcode reprogrammed for better performance or even fully downloaded from the web. Some interpreted languages can evaluate and execute incorporated expressions and code. It is possible to write self modifiable programs in Lisp [1] and in many other interpreted languages.

Code incorporation imposes requisites on the programming environment architecture and on the architecture of the system being built. The Java language allows incorporation of code because it provides late binding of classes.

The code incorporation approach uses more of the theory of structure and representation of programs than of the theory of programming languages construction. It uses an active object model to represent the program and manipulates this model in many ways, but mostly through code incorporation, sometimes through design patterns such as the interpreter pattern.

If the program is written in an interpreted language, then once the code is generated, it is ready to be incorporated. If you are using a language like Java (half interpreted and half compiled) then you must pass through a compilation step first. The diagram in figure 8 shows how it is done in Java.

In figure 8, the Java virtual machine (JVM) is interpreting an object model. One of the classes in this model is able to generate or obtain Java code to be incorporated into the system. The object model performs a system call (SC) to compile this source code into byte-codes and requests to the Java class loader (CL) to load it. This introduces a dynamic component in the system.

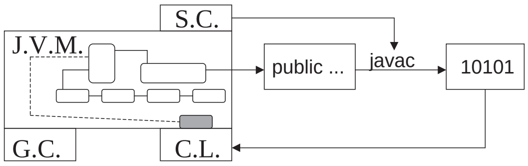

Figure 8. Simplified diagram for code incorporation

The Java language is appropriate for studying code incorporation approach because Java Virtual Machine provides class loading policies and security verifications on code, even during runtime. It also imposes access restrictions to system resources. These policies and restrictions will be specially useful in this work, because incorporated code brings extra power to your program and this power comes at the cost of increased risk. The only way to balance this tradeoff between power and risk is to tame the incorporated code. This can be done only if the execution environment allows code loading policies and also system resources access restrictions (Java fortunately allows both).

The next section: Advantages and Disadvantages will help you make a comparative analysis.

4.1. Advantages

- The proposed approach does not exclude any of the others mentioned.
- Does not require writing and interpreter, nor marshaling.
- The dynamic part can receive much knowledge about about the system.
- The only restrictions applied to the communication between the parts are the ones related to security.
- It can be used for building extension languages.
- If used for extension languages, the programmer is already familiar with the syntax of the language and the objects he will have to deal.
- As the system programming interface evolves, you do not need to reprogram the extension language that accesses it, since it directly accesses the system and keeps pace with its growth.
- Easy to upgrade to new releases of the programming language, just change the interpreter.
- Saves time because does not require building a grammar and an interpreter and testing them.
- Does not require dealing with a second programming language and its consequences.
- Provides scalability and leverage. The created classes and scripts can be stored and reused in many ways because they are byte-code representing classes.
- Modifications can happen statically or dynamically (during runtime). That is why it is important to have a tool for organizing the generated code.
- Making the framework generate the code does implies that an advanced user will never be able to write lines of code himself.
- The interpreter used is an already tested, optimized, portable and fully developed interpreter. The same happens to the language used which is powerful and suitable for the application.

4.2. Disadvantages

- It brings a lot of risk. The code incorporation approach introduces a tradeoff between power and risk.
- Letting the end user program (even indirectly) can be risky and the developer must spend considerable time building the restrictions delimiting what the user can or cannot do. Refer to security issues by the end of *3.2. The Java language.*
- The user can make the framework stop working if he receives enough power in the extension language. We solve this enabling users to undo some operations and reset the framework state.
- Just in the case of compiled languages like Java, the user has to have a compiler in his system, even if he is not a programmer.
- Using a compiler can be expensive, but time spent compiling is usually more than paid in subsequent uses of the compiled class.

• Calling an external process might limit the degree of interaction obtained with the language. This is a temporary limitation because we already have the choice of using the compiler within the program in the form of a Java class.

4.3. Code incorporation approach architecture

Following is part of a minimalist Java program that was based on the code incorporation approach. This is useful only to demonstrate the technique. Many important parts have been removed to make this program minimal (security verifications, error checking, exception handling and imports).

Minimalist Code Incorporation Program:

```
 1 //component to be dynamically changed
 2 Runnable vHotSpot = new OriginalClass();
 3 vHotSpot.run();
 4
 5 //"Synthesizes code" in a file
 6
 7 //Compile this file
 8 Runtime vRTm = Runtime.getRuntime();
 9 Process vPCompilation =
10     vRTm.exec("javac NewClass.java");
11 vPCompilation.waitFor();
12
13 Class vNC = Class.forName("NewClass");
14
15 //Run Loaded Class Method
16 vHotSpot = (Runnable)vNC.newInstance();
17 vHotSpot.run();
```

`OriginalClass` is instantiated and called in lines 1-3. This is a compiled class, it will be dynamically substituted by incorporating program generated code.

Line 5 indicates that you must generate code for the class to be incorporated. Do it in a file called `NewClass.java`.

Lines 7–11 call the compiler process on the generated class. `Runtime` is a class that provides facilities for executing processes. Notice that this code is waiting for the compilation to complete before loading the class. We omitted the test for successful compilation, but it must be performed in non minimalist programs.

On line 13 we perform the late binding of the generated class. We bind it to a Java class named: `Class`. `Class` can contain any other class in the system. After this we must cast the loaded class to an usable type, something we can call methods on or access attributes. In this example we cast it to the `Runnable` interface.

If the loaded class is a totally new class and we know nothing about it and cannot cast it to a known superclass or interface, we must introspect its methods using computational reflection facilities of the language and then call these methods. The same is valid for its properties.

On line 15 we change the object pointed by the `vHotSpot` method variable to point to an instance of the loaded class. Since we do not know the loaded class type, we cannot call `new TypeName()` to instantiate it, we request its instantiation to the `Class` class representing it. Line 17 calls a method in the loaded class.

The diagram below exemplifies how the example framework we constructed is structured in terms of packages and what are the dependencies among them.

Package diagrams are useful to structure large systems [15], usually systems where the class diagram is too big. The diagram encompasses mainly packages and their dependencies.

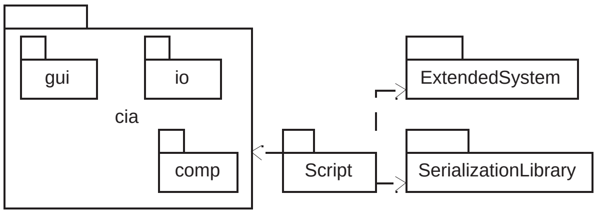

Figure 9. Package diagram for code incorporation example

`cia` is the main package for our example. It stands for code incorporation approach. It coordinates the interactions of its subpackages using the Model View Controller architecture. The `cia` is the Controller while the default view is implemented by the `gui` package. We say default view, because the code incorporation approach needs to be a working example, so it needs a gui, but any user may not be happy with this gui and build his own.

`comp` is a more important package than `gui`, while `gui` can be substituted, `comp` cannot. `comp` implements the steps of code generation, compilation and execution of a script in a multithreaded way that must be controlled by `cia` and externalized in `gui`.

The controlling performed by `cia` is mostly related to events triggered in the `gui`, but there is another flow between these packages that is composed of compilation and execution messages that must be output to the user or logged. This flow of data is handled by the `io` package. The `io` package is responsible for implementing abstractions such as standard error, standard output, standard input and spooling for the compilation and execution of the extension language. These outputs can be redirected and changed. One can change the standard output from a character stream that flows into the user interface to another that is output in the process shell.

Interestingly, the script generated to be incorporated into the system does not make part of the `cia` package. It is created in a separate package called `Script`. Depending on the security restrictions configured, the `Script` package might access parts of the `cia` package and an arbitrary number of libraries (right side of diagram).

The `ExtendedSystem` package represents the system that is extended using the code incorporation approach. In our example, this system is an implementation of the UDP framework that encompasses tens of classes. The `Script` does not see all the classes of the system, but a facade for it. This is merely a design choice. `Script` and `ExtendedSystem` could be made very dependant, by making them share many associations.

The extended system facade is usually the implementation of a domain specific language for interfacing the system. Besides this interface, others can be imported into the script to make it more powerful. In the diagram we picture a `SerializationLibrary` that might provide additional features not encountered in the extended system facade and might be interesting for a user willing to script some persistency into the system.

The framework is extensible through the incorporation of new libraries. We encourage that these libraries be designed using the Facade [5] design pattern.

4.3.1. Interaction

Once we are performing late binding of classes, designing their interaction will be an interesting task. The computational reflection facilities provided by the language might be used to let one class become aware of the other class features, even when they did not know about each other existence during compilation time. In Java this is done through the `java.lang.reflect` package.

Therefore, the main use of computational reflection in our system is in the coupling between the script and `cia` package and between the script and libraries it accesses.

In order to simplify the explanation of the possible interactions between these classes when the late binding occurs, we will provide an interaction diagram. Interaction diagrams usually picture the interaction among classes, but in this case we want to be generic and think about the interaction among the packages:

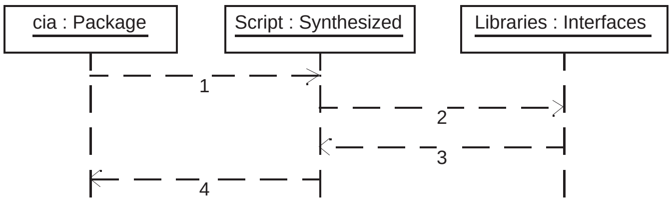

Figure 10. Interaction diagram for code incorporation example

From the package diagram presented before it can be inferred that the script class depends on the `cia` package and on the libraries. This dependency or relationship is now evidenced in the interaction diagram too.

If the script is programmed by the user, then he knows what classes he has to deal with and there is no need for computational reflection. In this case, the user will declare variables of these types and follow on using them without performing many type casts.

If the script is sinthesized, for instance by a visual programming tool that is able to import arbitrary libraries then it might not know much about the current libraries (extended system is seen as a library by the script).

References

1. Harold Abelson, Gerald J. Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1996.
2. Per Bothner. "Compiling Scheme to Java". *Lisp Users Conference*, November 1998.
3. Per Bothner. "Kawa - Compiling Dynamic Languages to the Java VM". *Usenix Conference in New Orleans*, June 1998.
4. Brian Foote and Joseph Yoder. "Metadata and Active Object-Models". At *PloP'98, Allerton Park*, August 1998. Also `http://www-cat.ncsa.uiuc.edu/~yoder/papers/patterns/Metadata/`, 1998.
5. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, 1995.
6. Li Gong. Going Beyond the Sandbox: An Overview of the new Security Architecture in the Java Development Kit 1.2. *Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California*, December 1997.
7. Li Gong. New Security Architectural Directions for Java (extended abstract). *Proceedings of IEEE COMPCON, San Jose, California*, pages pp.97–102, February 1997.
8. Li Gong. Secure Java Class Loading. *IEEE Internet Computing*, 2(6):56–61, November/December 1998.
9. Guy Steele James Gosling, Bill Joy. *The Java Language Specification*. Addison Wesley, Menlo Park, Calif., 1996.
10. SUN Microsystems Javasoft. *Beanbox*. 1997.
11. Ralph Johnson and Jeff Oakes. The User-Defined Product Framework. ***http://st-www.cs.uiuc.edu/users/johnson/DOM.html***, 1998.
12. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, Menlo Park, Calif., 1996.
13. Pat Niemeyer. *The Beanshell User Manual*. `http://www.ooi.com/beanshell/manual/`, 1999.
14. William Clinger Richard Kelsey and Jonathan Rees (editors). *Revised Report on the Algorithmic Language Scheme*. 1998.
15. Martin Fowler with Kendall Scott. *UML Distilled*. Addison Wesley, Reading, Massachusetts, 1997.

16. Richard Stallman. GNU Emacs Manual. *Free Software Foundation*, Cambridge, MA, tenth edition, July 1994. ISBN 1-882114-03-5.
17. Bob Glickstein. *Writing GNU Emacs Extensions.* O'Reilly & Associates, Inc., Sebastopol, California, 1997.
18. Aaron Watters, Guido van Rossum, James C. Ahlstrom. *Internet Programming with Python.* M&T Books, 1996.
19. Guido van Rossum. *Python Reference Manual.* Dept. AA., CWI, P.O. Box 94079, 1090 GB Amsterdam, NL, 1995. Also: (`http://www.python.org`)
20. Adele Goldberg and David Robson, *Smalltalk 80: The language and its implementation*, Addison Wesley, Reading, MA, 1983.
21. GUILE (*GNU's ubiquitous intelligent language for extension*). `http://www.gnu.ai.mit.edu/software/guile/`, August 1997. This documentation is under development.
22. Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl.* O'Reilly and Associates, Inc, 1996.
23. John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, Massachusetts, 1994.