JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6, No. 6, July-August 2007

Extending eclipse RCP with dynamic update of active plug-ins

Allan Raundahl Gregersen, University of Southern Denmark, Denmark
Bo Nørregaard Jørgensen, University of Southern Denmark, Denmark

Abstract

While the dynamic linking mechanism of modern programming languages, such as Java, allows loading of classes dynamically, it does not allow class reloading. Hence, dynamic linking facilitates development of component platforms, such as eclipse RCP, which supports dynamic loading but not dynamic updates of components, since this requires reloading. This paper presents an approach that enhances eclipse RCP with dynamic updating capability. It overcomes the version barrier imposed by Java's dynamic linking, while maintaining the security and type safety of Java. The feasibility of the approach validates through a modified implementation of the eclipse RCP run-time system. Analysis indicates that our approach imposes a moderate performance penalty relative to the unmodified platform.

1 INTRODUCTION

Applications that suffer from the predominant halt, redeploy and restart update scheme can benefit from dynamic adaptation. Dynamic adaptation refers to changing an application's behavior without stopping and restarting it. While most modern component platforms, such as eclipse, partially support dynamic adaptation through dynamic addition of new plug-ins, they lack support for dynamic update of plug-ins already running. The reason behind this is the missing support for dynamic update caused by restrictions on the dynamic linking scheme of modern object-oriented programming languages. Where the scheme, in for instance Java, supports dynamic loading and linking of classes, it does not support dynamic reloading of classes already linked. This restriction implies that Java considers objects of any class loaded twice as distinct types. Type correspondence between different loaded versions of the same class is therefore missing. [Sato04] discusses the implications of this problem, also known as the version barrier, in greater details.

Due to the missing support for reloading classes, modern component systems built on top of Java, such as eclipse [RCP06], must provide the necessary mechanisms for allowing dynamic updates to take place. Despite the version barrier recent advances in the architecture of eclipse [Eclipse06], allow addition of functionality in a more dynamic way than previous versions (versions 2.x). The capabilities of the Eclipse Update Manager, [EUM06] now include adding plug-ins dynamically without restarting the application thanks to the implementation of the OSGi Service Platform, [OSGi06]. However, when it comes to performing updates of already active plug-ins, that is plug-ins that have been assigned a class loader, the update manager comes out short, suggesting a restart (not of the VM but the workbench held by the eclipse runtime) before the changes are effectuated. Dynamically adding of new plug-ins in eclipse is made possible by organizing the class loading scheme in such a way that it compensates for the implications of the version barrier. It associates a class loader with every plug-in, which is solely responsible for loading all classes defined by a plug-in. Consequently, class loaders for plug-ins using classes in other plug-ins must delegate class loading to the class loader of the plug-in defining the class. This loading scheme ensures that all classes are loaded by one loader only, thus having only one type of each class. While this class loading scheme allows for new components to be added dynamically, due to the support for sharing class definitions, it cannot support dynamic update of running components because of the missing support for reloading classes in java. The only way to simulate the reloading capability is to reload the complete plug-in by associating it with a new class loader, which results in the version barrier problem.

This paper presents a new approach capable of updating active plug-ins. The approach assumes no support from the language runtime thus allowing execution on a standard Java Virtual Machine. In fact, in eclipse RCP the approach is implemented by modifying the class loading scheme and the update manger. The approach solves the problem of the version barrier by introducing a level of indirection in a technique we call *In-place proxification*. The term is used to describe the process of making objects of a former class version behave like a proxy, [Gamma95a]. This technique proxificates all objects of the previous active plug-in at update time, leaving execution of application code only in the latest version of the object's class. All invocations done by existing clients knowing only former versions (now proxies), forwards method invocations to the current version, thus allowing multiple version of a class to co-exist for the same "real" object. The level of indirection solves the common problem of *dynamic component re-wiring* mentioned by [Kniesel99]. The approach handles correspondence between different versions of objects in terms of *correspondence mappings*, which section 2 explains further. To ensure consistency in updated plug-ins only instances of classes in the most recent version hold state. In essence, objects go transparently back and forth through the version barrier making clients unaware of the update.

The approach prepares the plug-ins in the original application for dynamic updates through a sequence of compile time transformations. These transformations consist of adding small pieces of code in classes reachable from the plug-in's API and by performing some specified checks of version compatibility and plug-in dependencies. The classes reachable from a plug-in API are the set of classes that belong to the exported packages stated in the manifest file of the plug-in. Furthermore, it is subclasses of classes obtainable from return values of the API, which in general are all classes reachable directly or indirectly from other plug-ins. The term *update-enabled* refers to plug-ins having gone through the transformation process. The overall characteristics of the approach are:

Continuous execution. The update process only halts the execution of plug-ins affected by the update. Any user or system interaction involving non-affected plug-ins continues executing.

Behavioral preserving. An update-enabled plug-in will behave as the original. Furthermore, the behavior of an application after a dynamic update is equivalent to that of the same application built from the updated set of plug-ins.

Standard JVM. The approach does not require a modified virtual machine. It runs on top of any standard virtual machine, [Lindholm99].

Ease of use. The programmer only needs minimal knowledge of the approach. The transformation of the plug-ins is automatic and the update manager only needs access to the new versions of the plug-ins for the update. The programmer's insight is only required in case of new fields in updated classes. Section 2 returns to this issue.

Flexibility. The approach provides the ability for a class to implement new interfaces, thus making future clients able to use the new inheritance hierarchy. Likewise, the programmer can add methods in existing classes and add any number of new classes. The API of a plug-in however cannot change in a way that breaks backwards compatibility. This is not overly restrictive, considering good practice for normal object oriented development.

Updatable updates. A plug-in is equally updateable regardless of the number of previously performed updates on that particular plug-in.

There have been quite a few proposals to dynamic software updating [Bialek04; Duggan01; Gupta96; Gustavson05; Hicks01; Malabarba00; Orso02; Redmond02; Vandewoude05a]. Some of them require a modified virtual machine and some lack the support of generic Java applications or target other languages. To the best of our knowledge, the approach described in this paper is the only pure software based approach with capabilities of adding new methods and new subclassing relationships in a new version. This section provided a brief overview of the main difficulties in dynamic software updating. For an overview of general pitfalls in dynamic updating see [Ebraert05] and for interested readers [Eclipse06] outlines more on the architecture of Eclipse.

The contributions of this paper are twofold. Primarily to present the design of a new approach for dynamic update of Java applications. Secondly to provide implementation details for applying the approach to the eclipse rich client platform.

Having introduced the problem domain and the main characteristics of our solution section 2 now turns to an in-depth description of the approach. Section 3 gives an overview of specific implementation details and section 4 discusses related work. Section 5 gives an outline of status and future work, and finally section 6 concludes the paper.

2 DYNAMIC UPDATING

This section presents the approach for dynamic updating of Java applications. First, it explains the essence of the approach and the problems it implies. Next, it discusses solutions to specific design problems and finally gives assumptions under which the approach will be applicable. Throughout the rest of this paper, the following terminology applies:

***Component**. This paper uses the term component to describe a module in a running application built on top of a component system. A component consists of a number of classes, resources and file descriptions. These descriptions hold the unique id of the component typically split in a symbolic name and version identifier along with information about component dependencies. Furthermore, it states which packages of the component should be visible to other components. These packages constitute the component API. While eclipse plug-ins also include file declarations of extensions and extension points, this definition states the minimum information that the approach expects to find in a component. Furthermore, it is easier to generalize the principles of the approach given this subset.*

***$CompX_n$** defines component X of version n. When the subscript n does not appear, component X is in a context not specific to a particular version.*

***O_BX_n** defines an object B that has dynamic type corresponding to a class loaded by the class loader associated with $CompX_n$. In this context, $CompX_n$ declares the type or class of B.*

***Cl_AX_n** defines a class A declared in $CompX_n$.*

Dynamic updating in a component system is in essence the ability to replace one or more of the components while the system continues to execute. The main goal of the approach is to provide mechanisms that are purely software based and run on a standard virtual machine to enable such updating. It primarily deals with updating existing components and not the ability to add new ones as this support typically already exits in modern component system such as the eclipse RCP. It does not directly address problems involving updating distributed systems as such updates require a coordination mechanism. However, the nature of the approach presented is in fact suitable for enhancements of this kind in future studies.

A dynamic update approach must face a number of issues in a statically typed language like Java. The root problem is the version barrier as stated in section 1. The fact that objects of the same class are considered to be of distinct types when loaded by different class loaders results in a number of language technical issues which the approach must face. These issues are briefly described below, whereas discussions and solutions are given in separate subsections.

Dynamic component re-wiring. In order for new code to be reachable from existing clients a re-wiring mechanism needs to be present.

Object correspondence handling. In section 1 it was stated that the approach allows multiple versions of the same class to co-exist. Without proper handling of correspondence of such versions, an application using the approach is likely to end up in an inconsistent state.

Type compatibility in state migration. In order for an updated component to continue consistent execution with the previous version, state has to be migrated. When allowing multiple versions of classes to co-exist, the state of a former version of a component is not likely to be compatible with the new class definitions due to the version barrier.

Component dependency handling. The code managing dynamic updates (i.e., the update manager) must possess knowledge of dependencies between components. This is particularly relevant when two or more components interchange types of other components.

Type compatibility in method invocation. Types of parameters and return values interchanged across components in method invocations suffer from the same incompatibility as field values do in state migration. However, in order to understand the workflow of method invocation in the approach a separate subsection is used.

Fragile base class problem. Any component system must face the difficulties of decentralised development. Of particular interest in the sense of updating components is *the fragile base class problem*. In general, the problem states that changes to superclasses might break compatibility with unknown subclasses.

The following subsections discuss the above problems in relation to the approach. Throughout the subsections a running example of different update scenarios is used.

Dynamic component re-wiring

In order to present how the approach handles re-wiring of components to execute the latest version of code this subsection gives a high-level view of component relations and the impact of dynamic updates.

Consider the situation in figure 1 containing two components CompA and CompB in a running application.

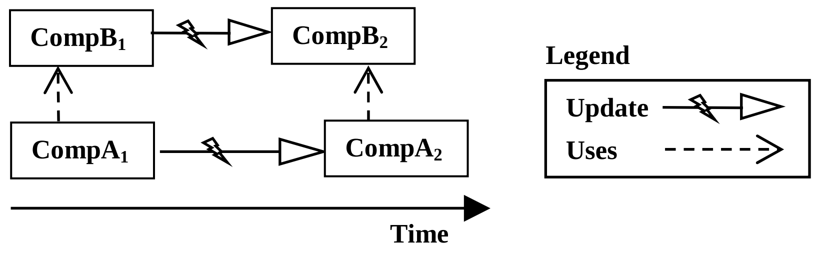

Figure 1: Two components in a running application. The figure shows that CompA$_1$ depends on CompB$_1$ and at a given time an update of CompB$_1$ to CompB$_2$ occurs and later CompA$_1$ to CompA$_2$, which now depends on CompB$_2$.

One point of concern is how CompA$_1$ in the time before the update to CompA$_2$ uses the code in CompB$_2$. The idea is to make CompB$_1$ a proxy of CompB$_2$ so that CompB$_1$ redirects any incoming method call to CompB$_2$. To support this kind of restructuring of a component any method reachable from the API of CompB$_1$ contains code to check if it should act as a proxy and some additional code to invoke the same method in a corresponding object (or class in case of a static method invocation) in CompB$_2$. The code for invoking corresponding methods uses the Java Reflection API, [Sun06d]. This supports bug fixing in the methods already declared in CompB$_1$ but lacks support of calling new methods in the API of CompB$_2$. The solution to this issue lies in the class loading scheme of the approach, as every component has an associated loader. When CompA$_1$ becomes CompA$_2$, the class loader associated with CompA$_2$ will contain references to the loaders of the latest versions of the required components. This implies that class lookup will search only the latest versions installed in the system and thereby locating classes in CompB$_2$ instead of CompB$_1$. The result of this is that code in CompA$_2$ can invoke newly added methods and treat classes in CompB$_2$ as new types if they either extend or implement them in the new version.

The above descriptions show how the approach addresses the problem of redirecting invocations to the latest version of a given component. As far as we know our approach is the first to actually use the original objects as proxies instead of wrapping them in a separate class, which unlike our approach does not allow adding new methods and interfaces to updated classes.

Object correspondence handling

To understand what happens in the object level of a system after applying updates, figure 2 illustrates different scenarios.

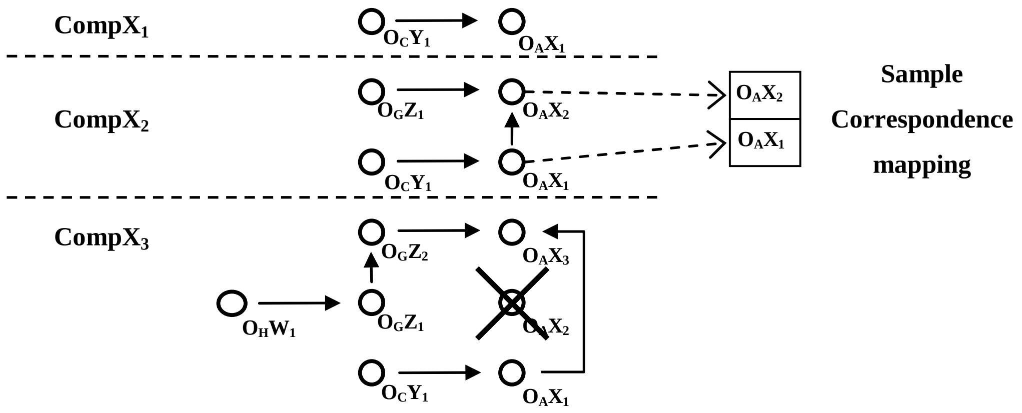

Figure 2: Object behavior in the approach.

The above figure illustrates what happens to object references after two updates of CompX. Each dashed line separates different versions of CompX. The top of the figure illustrates a situation before any dynamic updates have been performed. An object (O$_C$) in a client component (CompY$_1$) of CompX$_1$ holds a reference to an object in CompX$_1$ (O$_A$X$_1$). In the second scenario between the dashed lines CompX$_1$ has been updated to

$CompX_2$. Now when O_CY_1 invokes a method in O_AX_1 it acts as a proxy forwarding the incoming call to O_AX_2, thus ensuring that only the latest version of the application code is in use. The figure shows a *correspondence mapping* from O_AX_1 to O_AX_2, which states that the two objects really represent the same instance in the system. As an example of the correspondence mappings workflow, consider if O_CY_1 invokes a method in O_AX_1 and this method takes a parameter of a type declared in $CompX_1$. In such situations, a lookup in the correspondence map holding mappings of objects from $CompX_1$ to $CompX_2$ has to be performed to find the object representing the corresponding object in $CompX_2$. When the correct object is found, the method with the same signature in O_AX_2 is invoked using this object as parameter. The approach ensures that the correspondence map holds the corresponding object at any time by moving the state of $CompX_1$ to $CompX_2$ at update time mapping corresponding objects. Furthermore, any constructor in $CompX_1$ called after the update results in calling the same constructor in $CompX_2$ and mapping the two objects instantiated. When the method returns, the return value can be a type of a class in $CompX_2$, which is incompatible with the return value of the method called in O_AX_1. To address this problem the approach uses the reverse workflow doing a lookup in the correspondence map from $CompX_2$ to $CompX_1$. This mapping is not shown in the figure and is only used to decrease lookup time by indexing objects of $CompX_2$ instead of $CompX_1$. In this reverse lookup, it cannot be assured that mappings are available for all objects in $CompX_2$ to objects in $CompX_1$. For instance, if the return value is a new object created after the update, either internally by $CompX_2$ or externally by another client. If the corresponding object in $CompX_1$ is not found for the object in $CompX_2$ a new proxy object, which forwards incoming calls to the object in $CompX_2$ is instantiated and returned to the caller. The proxy and the "real" object are then mapped to allow fast lookups in future invocations. The middle section of the figure also shows that if a new client component ($CompZ_1$) of $CompX_1$ is added after the update to $CompX_2$, then objects of $CompZ_1$ bypasses the proxy component of $CompX_1$ making direct references to objects in $CompX_2$. The bottom of the figure represents the scenario of an update of $CompX_2$. First of all, it is obvious that the proxy object O_AX_1 now forwards calls to O_AX_3 instead of O_AX_2. In the figure, a later update of $CompZ_1$ to $CompZ_2$ is illustrated to see what happens when a proxy component has no clients. The result of updating $CompZ_1$ is that $CompZ_1$ now forwards method calls to $CompZ_2$. This implies that $CompX_2$ is never used by $CompZ_1$ as no original application code is executed. The net effect of this is that all instances of $CompX_2$ are garbage collected. Using correspondence mappings and in-place proxification proves to solve the common self-problem mentioned in [Lieberman86], as it guarantees the use of the correct object understood by clients using it. If for instance a new version of a class defines a method that returns `this,` the object returned to the client calling it would in fact be a reference to the proxy itself forwarding to the "real" return value.

This high-level view of the approach shows how the version barrier can be effectively circumvented to allow objects of different versions of the same class to co-exist at runtime.

Type compatibility in state migration

The approach needs to face a number of compatibility issues when allowing distinct objects disguised as the same real object to cross the version barrier. Whenever a field, parameter or return type has a static type corresponding to a class in an update-enabled component there are possible conflicts that need handling when updating the system. To see such conflicts in a more illustrative way the next subsection turns to the example in figure 1.

Consider what happens at update time when migrating state from $CompA_1$ to $CompA_2$. In general, field values represent the state of a running program. In presence of the version barrier, the approach cannot simply make a shallow copy of the field values in a given instance in a component that is about to be updated. The update manager must include runtime type checks to ensure that all field values are compatible with the ones seen by the updated component. The possible scenarios of static type of a given field that influence these checks are divided in two. Either the declaration of the type of a particular field lies in an update-enabled component or it does not. In case of the former, the location of the component declaring the static type of the field is found. Based on this knowledge the dynamic type of the field value must comply (that is being either that particular type or any subtype of it) with the most recent version declaring the particular static type due to the class loader scheme.

In short, when an update occurs the approach (1) extracts the state from $CompA_1$, (2) converts it to be compatible with $CompA_2$ and (3) deploys the transformed state in $CompA_2$. The main difficulties lie in converting the field values to let the new component understand them while mapping the correspondence between objects of the two versions. In order to explain these difficulties the following description uses examples starting with the simple update scheme of figure 1. Suppose that an instance in $CompA_1$ (O_EA_1) needs updating to $CompA_2$, which implies deep-copying the fields of O_EA_1. For every field the update manager checks to see if the static type is dynamically enabled. If that is the case, then it locates the components declaring both the static and the dynamic type of the field. Then it checks for equality of the two components found and in this case it requests an instance of the most recent version of that component. A situation like this occurs for two reasons in figure 1: Either a field has both a static and a dynamic type of class/classes declared in $CompA_1$, or in $CompB_1$. In the first situation, a request for a compatible instance of the dynamic type in $CompA_2$ is made. This request must ensure that the value of the original field acts as a proxy for the new instance created, in the future. Furthermore, mapping of the two objects is kept, so when they later figure as parameters or return values in method invocations, the caller or callee recognizes them as a type known to them. Given that both types are declared in $CompB_{1,}$ a request for a new instance of the class corresponding to the dynamic type from $CompB_2$ is made and the same proxyfication and mapping takes place as before. Consider what would have happened if $CompB_1$ was not updated in time of the update of $CompA_1$ as shown in figure 3. Then fields in $CompA_1$ having types declared in $CompB_1$ would be compatible to $CompA_2$ and a request for an updated instance would not be necessary.

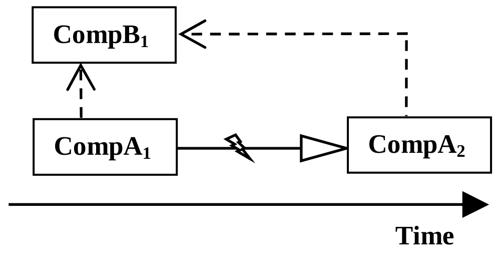

Figure 3: Simple update from $CompA_1$to $CompA_2$

Component dependency handling

The complexity of the state migration process grows when the components declaring the static and dynamic type of field values differ. A situation like this occurs when a client of a component implements or extends a type of that component. Consider the scenario illustrated in figure 4 when updating an instance of a class in component $CompA_1$.

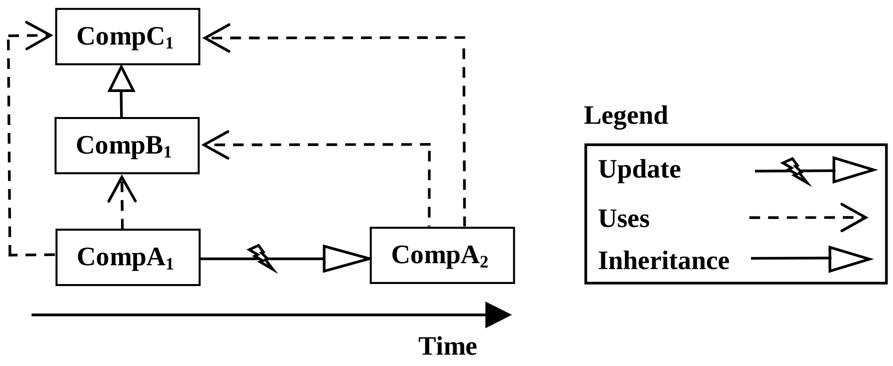

Figure 4: Component scheme as in figure 3, but now $CompA_n$ depends on $CompC_1$ as well. The inheritance relation between $CompB_1$ and $CompC_1$ means that $CompB_1$ defines at least one subclass of a class in $CompC_1$.

Suppose the update manager at some point locates a field with static type declared in $CompC_1$ and dynamic type in $CompB_1$. To handle such situations it searches for the most recent component of the static type as this will be the one known by the updated component. After this, it checks if the latest version of the component containing the dynamic type requires the component found for the static type. In such cases, represented by the scenario in figure 4 , it simply copies the field if the types are already of the latest version. On the other hand, in case of further updating of the components declaring the static and dynamic types, the update manager requests a new instance as seen before. In rare situations, however, the most recent component declaring the dynamic type of the field does not require the latest version of the component with the static type. Consider the example in figure 5.

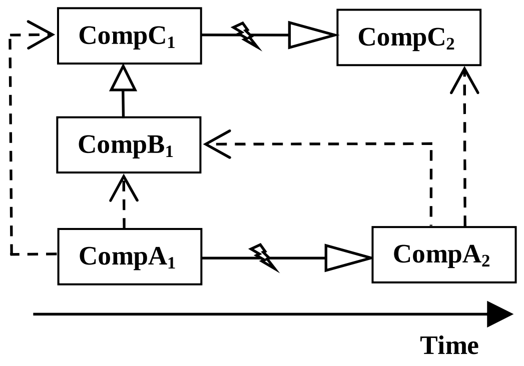

Figure 5: The problem of two components depending on different versions of a third.

The problem at hand is that component $CompA_2$ depends on the updated version of CompC ($CompC_2$) and of $CompB_1$, which on the other hand depends on $CompC_1$. In this paper, we refer to this particular problem as *the conflicting dependencies problem*. A similar situation occurs when a field in $CompA_1$ has a static type declared in $CompA_1$ and a dynamic type declared in a component using $CompA_1$. In that case the dynamic type will not comply with the updated version of the static type. To handle these situations the approach performs an analysis of component dependencies and inheritance across components. It consists of (1) analyzing component dependencies and inheritance relations across components before the start of the update, (2) *Class loader migration of* any component ($CompB_1$) that might cause this particular conflict, (3) update $CompA_1$ so that it would depend on $CompB_2$ and $CompC_2$ while $CompB_2$ also depends on $CompC_2$. *Class load migration* happens through moving the code and state of a component to a new namespace by loading classes with a new class loader. The difference between this and a normal update is that the same code base is used; only the types changes to comply with the new namespace. In this case, the component dependencies end up as illustrated in figure 6.

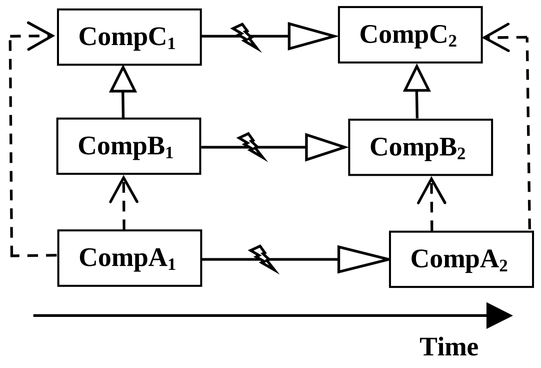

Figure 6: Solving the conflicting dependencies problem. In the updated system the two components $CompA_2$ and $CompB_2$ will both depend on $CompC_2$, thus preventing the conflict to occur.

This approach in itself be relative time consuming in worst-case scenarios in which every dynamic-enabled component needs updating to fulfil the component dependencies mentioned. However, in many cases the proposed restrictions are not necessary. Remember that a prerequisite of the conflicting dependency problem is that a component declares subtypes of classes in components other than itself. To capture those rare situations that require updates of other components, a mechanism to discover potential conflicts needs to be present. The approach handles the conflicting dependency problem by performing load time checks of relevant dependency and subtype relationships for every component. For a given component, CompX the approach must search for types in the component that might cause a conflict when updating CompX or any component depending on CompX. It then writes the information gathered to the file system so that the update process can use it in later updates. In relation to figure 5, the approach finds a cross component subtype in $CompB_1$ when it is installed and it determines that $CompB_1$ depends on $CompC_1$. This information is used when updating $CompA_1$ to $CompA_2$ to ensure that $CompB_1$ updates before the update of $CompA_1$.

The example above shows how the conflicting dependency problem is solved in relation to state migration. However the conflict manifests itself differently at runtime. Suppose the dependencies of figure 5 holds but no inheritance relation is present between CompB and CompC. Then the approach allows the update to $CompA_2$ leaving possible runtime conflicts in cross component method invocations where types of CompC are used. Suppose $CompB_1$ defines a method in its API returning a type from $CompC_1$. When $CompA_2$ invokes the method it expects a type from $CompC_2$. The approach solves this issue by a combination of pre-compile modifications and load time wrapping. At pre-compile time checks of component API identifying conflicts are made by determining the components declaring every return or parameter type. If a dynamic enabled type declared in a component other than the one defining the method in which it belongs is found, then this type is changed to the interface that every dynamically enabled type must implement (this is actually code generated by the pre-compiler, leaving the programmer unaware). Information about the substitution must be kept by the dynamic update manager for later use when updating components using the involved component. Every method call done by components using a method that returns a conflicting type is wrapped (at load time) in a method defined by the update manager that always returns the expected type. In case of conflicting parameter types, the converting is done by load time generated code in the start of the method.

Previously described solutions of state transfer ensures that migration of fields does not end up in runtime errors, because it is guaranteed that any dynamic type either directly copied or requested as a new version complies with the static type of the field. However, there are still cases where the types reachable from the new state instances can cause errors. Consider what happens when a field has a type belonging to the "*Java Collection Framework*", [Sun05a] or any internal declared data structure defined by $CompB_1$ that could hold instances of types not directly known to $CompB_1$. If such data structures references any type declared in $CompA_{1,}$ then these types would not be compatible to $CompA_2$. To capture these cases the update manager checks if the field value is member of the collection framework and converts the instances (if necessary) held by the collection to types known by the new version. To ensure compatibility of any instance reachable from the updated component, it simply recursively traverses and convert all fields that are dynamically enabled.

Type compatibility in method invocation

Problems related to method invocation when using the approach has its origin in the version barrier. Consider the following pseudo code in a given method in the API of an update-enabled component (the code for checking the proxy field and invoking the corresponding method in the new version are code generated).

```
public ATypeInThisComp getTheType(String name,
AnotherTypeInThisComp theClass) {
   if proxy then // Execute the same method in updated version
        return dynRef.invoke("getTheType", …;
   else … // Execute normal method body
}
```

An obvious problem in this case is the return type when invoking the corresponding method in the dynamic reference to the latest version. Because of the version barrier, the return type in the latest version needs converting in order to be usable as return type. This process is similar to the one seen in transferring state by using static and dynamic type information to retrieve an instance of correct version. Another issue is the parameter types which is essentially the same problem you have with return types. Thus, to ensure correct method invocation all methods in the API of a component need to comply with the following template generated by the pre-processor. Of course, lookups in the appropriate correspondence mappings are performed before any converting.

```
public TypeX methodName(TypeY param1, TypeZ param2 …) {
   if proxy then // Execute the same method in updated version
        // Convert parameters to types
        // understood by latest version
        Object newParam1 = convertType(param1, newVersion …)
        // Get the return value from latest version
        Object res = dynRef.invoke(("getTheType",newParams …);
        // Convert it to type understood by this comp
        res = convertType(res, thisVersion …);
        return res;
   else … //Execute normal method body
```

This ensures that the caller or callee understands any return and parameter type. In the description of state transfer, the problem of having type belonging to the collection framework or other such data structures was mentioned. The same problem arises in method invocation across components, as the return type could be a collection containing types declared in a previous version of a component using the particular method. Suppose this was the case, then a situation in which one component functions as a keeper of instances of another component occurs. A possible scenario exemplifying this could be a bank where different departments define a specialized class of an account superclass. The system could consist of a common model component holding every account and a number of department components that register their instances of accounts by casting them to the common superclass. Imagine that a department retrieves an instance of an account known to that particular department (this would require that the common model component deliver a key to inserted instances) and casting it to the particular subtype of that department. Consider now if the department is updated and all its instances are updated (all old instances are now proxies of the new ones). Subsequently, when retrieving that same instance as before it would not be compatible with the expected type when casting it to the subtype. The needed instance is in fact the one referenced by the proxy returned by the common model. The approach addresses the problem by replacing all casts to types that are dynamically enabled with a cast-method that returns the expected instance in runtime. Another issue arises when the department tries to check if a given instance is in a given collection retrieved from the common model. This check along with the remove functionality provided by collections will always fail after an update. To address this issue the approach makes sure that every comparison of different representations of the same object version always returns true. It accomplishes this by overriding the equals and hashcode methods.

Fragile base class problem

The presence of inheritance relations across components reveals another problem known as *the fragile base class problem* described in [Szyperski98], which states that if a class in a component evolves it might break either syntax or semantics of subclasses implemented in client components.

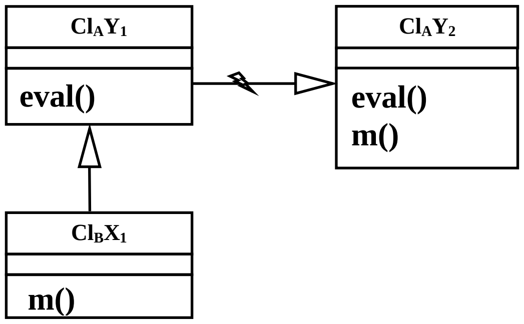

Figure 7: The fragile base class problem. The method m() is not originally implemented in the superclass, but added dynamically. The two methods may be semantically different.

Figure 7 illustrates how easily decentralized development can introduce the fragile base class problem in the presence of inheritance relations across components. Consider what happens when CompX declares a subclass of ClAY1 which defines a method m() that is not present in the superclass. Suppose now that the developers of CompY defines a method also called m() in ClAY2. In a previous subsection it was explained how the approach in such cases would trigger class loader migration of CompX to make the subclass comply with the correct super type. A dispatch issue arises from this as the approach cannot determine which method to execute when method m() is invoked. While "Old" clients expect the semantics of the method in the subclass, which is chosen by default in Java, new clients aware of the updated superclass would expect the new method definition in the superclass. One solution would be to implement an advanced stack trace to see from where the method was invoked, however, this solution brings too much of an overhead to be feasible in the approach of this paper. In fact the fragile base class problem poses the same difficulties if the system is updated through the restart scenario. Hence, to ensure system integrity the approach rejects all updates of a component declaring a superclass to a class declared in another component, if the update results in modifications to the superclass, causing either syntactic or semantic incompatibilities.

This concludes the description of the main issues faced by the approach. Some of the technical implementation details, of which a description will follow in section 3, are left out.

Assumptions

The approach works for arbitrary java applications under some assumptions, presented in this subsection.

Public and protected fields. To maintain the level of indirection of the approach, it only allows access to public or protected fields across components through appropriate accessor methods. This restriction complies with information hiding, first described by [Parnas72], which is a common requirement in object-oriented programming. Retrieving field values internally in components is not subordinated to this restriction.

Reflection. Values of fields retrieved by the reflection API can be inconsistent with the right value held by the latest version of the instance. For instance, if reflection is used to capture the values of particular fields of a class whose state has been transferred to the latest version, it would not be semantics preserving. Furthermore, malicious programmers can exploit all fields, methods and the initializing constructor generated by the approach posing a security risk. For this scenario to happen a programmer has to know of the approach.

API evolution. The approach preserves type safety by defining restrictions on component API. All classes in the API must be backwards compatible in the sense that they should at least provide the same set of public methods and constructors as the previous version. Classes in the API can add any number of methods, fields and constructors in a new version, which makes the approach very flexible. In short, a class in the API can evolve following the normal API design guidelines [McManus05]. You are, however, free to add, remove or substitute internal classes not included (directly or indirectly) in the API, as only types reachable from the API are affected by indirection.

Timing of updates. The approach does not support updating of components containing active methods at the time of the update request. If one or more methods belonging to a component that needs updating are executing, then it has to wait for them to stop. This may imply that some updates never takes place.

Native methods. The approach assumes that classes reachable from the API of a component do not contain native methods. Native methods are methods implemented in another language that can be used in Java through the Java Native Interface, [JNI06].

Field values. The approach cannot assign values to fields in a new version of a class, if the fields were not present in the preceding version. In case of additional fields in a new version of a class, the approach assumes that the programmer implements default values at the class level. Moreover, the programmer should take these values into account when using the fields, as instantiation of new objects which is part of moving state, is done by a code generated constructor which do not initialize fields.

3 IMPLEMENTATION

This section explains how the approach handles the problems stated from a technical point of view.

A system implementing the approach consists of two main components; a pre-processor and the *dynamic update manager* (DUM). The main purpose of the pre-processor is to prepare components for dynamic updates. The job of the DUM is to perform the updates by migrating state and maintaining correspondence mappings of corresponding objects in the system. Furthermore, it handles the runtime conversions of return and parameter types. As previously mentioned, any class reachable from the API of a component needs pre-processing. In such classes the pre-processor add fields as follows to let instances of the classes behave like a proxy.

- A static proxy field that determines the behavior of the class.
- A static reference to the class representing the latest version if it is proxy.
- A reference to the current version of the instance if it is proxy.
- A static field that references the component instance in which the class is declared.
- A field containing the value returned by the hashcode method.

Redirecting static method invocations happens by the use of the static field that represents the latest version of the class whereas redirecting instance methods uses the object reference to the corresponding instance in the latest version. In order to redirect methods the DUM declares two general methods for method invocation from a proxy, one for static and one for instance invocations.

The component class (singleton pattern, [Gamma95b]) which is code generated holds values for determining the unique id of each component, typically divided in a symbolic name and a version identifier. Furthermore, it holds a list of all instances currently active in the component. This enables the DUM to retrieve instances of a component that needs updating one at a time. To ensure that no memory leaks appear in this case, weak references in java.lang.ref package, [Sun05b] is used in order for the garbage collector to reclaim objects that is unreachable from the main program.

To see what goes on behind the scenes of the DUM this subsection exemplifies what happens when updating a component and more importantly how the approach handles updates of previously updated components. On time of an update request caused by either user interaction or a push-through strategy from a web-server, of which the specific technique is out of the scope of this paper, the DUM first checks if additional updates of other components should take place as described in section 2. It then starts the proxification and state transfer process by traversing the list of live instances in the component. For every instance the proxy field is set to true and the newly created object from the new version is associated with the proxy instance. Furthermore, it maps the two corresponding objects in two different maps indexed by proxy objects in one and real objects in the other to keep lookup time minimal when using these maps in parameter and return value determination. It also makes sure to set the hashcode field to the value returned by invoking hashcode() on the proxificated object. This ensures correct comparison between objects representing the same instance. The creation of new objects happens by the use of a code generated constructor that actually does nothing. The approach cannot use a copy constructor like the technique in [Orso02] that takes the state of the former version because the encoding of state needs to create new instances. Instead it adds any new instance to the correspondence mapping so they can be used when transferring the fields of the instance. When transferring state of a specific field is completed the state held by the old field is cleared to free up memory. This leaves the memory print of a proxy relatively small. After the update any constructor called in a former version of a component invokes the corresponding constructor in the current version and associating itself with the object returned. Furthermore, the created instance of the current version is added to the active objects list in the appropriate component instance.

When a component eventually needs updates a second or third time, special care must be taken not to introduce more than one level of indirection from former to latest version. The approach needs to update the field representing the real objects and classes in any previous version of live instances. This is handled in parallel with state migration by looking up objects in former versions that correspond to the one being proxificated. In some cases a given object does not have a representation in any former versions if it has been created at a point after an update and has never been returned to clients using particular versions of the updated component. In such cases the approach simply updates proxy instances of the versions currently present in the maps. As a result of this scheme a lot of objects in versions between the first and the current can be garbage collected, thus minimizing memory overhead. Actually, the only live objects of a particular version after an update are the ones being directly referenced by clients using this version.

An issue arises after an update in handling references in the correspondence mappings. It has been described that use of weak references solves the problem of reclaiming objects that are unreachable from application code. Consider what would happen using this type of references when an instance of the latest version is created, and at some later point figures as a return value (directly or indirectly in collections or likewise) in a method called by a client using a former version. In this case a proxy instance of the corresponding class in the particular version is requested and put into a correspondence map. If weak references are chosen, the proxy instance would not be reachable from application code and thus reclaimed in the first coming garbage collection leaving the DUM of doing the same request every time this specific instance takes place in method invocations (assuming that the clients receiving the object not caching it permanently). In correspondence mappings from a new version to a former java.lang.ref.SoftReference, [Sun06c] is used instead. In this way the garbage collector reclaims objects referenced in this way only when needing more memory.

Performance

This section briefly discusses performance issues related to the approach. The migration of state is of concern along with converting parameters and return types in method invocation.

As stated before the migration of state happens through converting old types to new ones which implies executing one constructor for every live instance in a component. This approach is relatively time consuming and very dependant of component size. The fact that the approach updates whole components regardless of the impact of the change in the updated version could seem odd. However, what is lost in one place is gained in another. By accepting longer update time, the actual runtime performance increases significantly in the sense that internal classes run as normal java classes. A system where every class is a component like in [Orso02], makes the update of finer granularity and thus requires less update time in average per component. This approach, however, implies that every class is penalized by a proxy or wrapper making the conversion between types much more frequent in runtime. Our approach ensures that classes which are tightly connected belong to the same namespace and are thus compatible when communicating. Assuming that dependencies between components are kept as low as possible, it is concluded that the vast majority of interaction happens inside components making the approach very efficient indeed. As an extreme example in opposition to defining every class as a component, an entire application could be one huge component. In this case making a dynamic update corresponds to the normal stopping and restarting scenario. In our opinion we have found a golden middle way between these extremes. We stress that the purpose of the approach is to support normal desktop applications such as java IDE's in which users are tired of restarting when updating. Another issue that benefits the approach of updating entire components at a time is that an update typically consists of more than just one component. In real life an entire feature containing several components is typically the normal updating granularity. In this case a lot of bindings between previous versions get converted to new ones thus not needing ever to go through a proxy except for components in the very bottom controlling the event flow triggered by user interactions.

4 RELATED WORK

This section focuses on work that relates to the approach described in this paper in the sense that it possesses one or more of the capabilities needed to do a dynamic update of a running system. First, it looks into work contributing to working around the version barrier held by Java, which has shown to be a major problem in dynamic updating systems as anticipated in [Ebraert05]. Secondly, it focuses on existing approaches for supporting multiple representations of the same objects. Finally, we turn to the subject of transferring state between versions.

Different approaches for handling the version barrier have been discussed in the literature and can generally be divided into the following two subcategories:

- Running a modified version of a standard VM to allow object compatibility of multiple versions of the same class.
- Introducing method indirection and state transfer between versions.

Although not directly intended for dynamic software updating, [Sato05] presents an approach based on modifying a VM that relaxes the version barrier under certain circumstances. They propose a terminology called "*Sister namespaces*" that extends the normal class-loader scheme with the ability to associate a sister class-loader. This means that multiple versions of the same class loaded by sister class-loaders can effectively be assigned to each other under the assumption that they are version compatible. They give a definition of version compatibility that builds on binary compatible changes in the Java language specification [Gosling00]. Other approaches that modify a standard virtual machine and thus not directly comparable to our work are [Malabarba00; Redmond02; Gustavson05]. Among techniques not requiring a modified virtual machine, [Orso02] presents DUSC which is purely software-based like the approach presented here. It addresses the version barrier problem by statically modifying a Java application to be dynamically enabled by wrapping all classes in the system, thus introducing indirection of all public method calls in every class. One advantage of the technique is that it does not heavily exploit the reflection API, as interfaces of the wrapper classes are frozen and all references to types in the original application classes interchanged with wrapper types. Unlike our approach, one disadvantage of that approach is the fact that you cannot add new methods to existing classes without making a completely new wrapper and thus loosing any state held by the previous version. Another drawback of the approach is that a class cannot implement or extend any new class in a new version, which makes a lot of normal refactoring impossible. [Bialek04] presents a similar approach, which uses wrappers and renaming of classes like [Orso02] to support hot swapping. In [Bialek04], however, it is possible to alter the signature of existing methods by using interface adaptors.

Different approaches for supporting multiple representations of the same object, by dynamically changing the class of the object, have been discussed in [Drossopoulou01; Serrano99; Malabarba00]. The approach in [Serrano99] called "*Wide classes*" proposes to let an instance of a class be reclassified as a subtype and later to be shrunk back to the original. Unlike our approach the different views cannot exist simultaneously which is necessary in order to let old clients see old types and new ones new types. This kind of support is proposed in [Bertino95], which introduced the idea that an object can have multiple most specific classes. Class selection is then based on the static type of the object reference through which the object is accessed. This allows for multiple simultaneous representations of the same object. However, their approach cannot be used for classes belonging to disjoint class hierarchies, which is the case when loading classes with different class loaders, because it requires the set of most specific classes to have a common superclass. Another approach called "*Aliased Multi-Object Type Widening*" presented in [Joergensen04] defines wrappers for extending both state and behaviour of objects. Clients have a specific view of the object like in our approach (a component using another is a client of that component). However, the approach is not applicable to the problem addressed in this paper, because an object has differing state and behaviour depending on the particular view in which it operates. The approach presented here preserves the semantics of corresponding objects regardless of the specific clients using them.

When it comes to transferring state, several approaches have been suggested. The technique most similar to ours is the one in [Orso02] where copy-constructors are used. They simply encode the values of the fields of the former version of an instance and apply it to a code-generated copy-constructor in the new version. This approach cannot be directly transferred to our approach because, as stated in section 2, the update manager needs to keep track of both the old and new versions of the instance while migrating state. In [Vandewoude05a] an approach for dynamically updating component systems is presented. The work mainly focuses on developing a tool that can assist a programmer in transferring state in an intelligent way. This implies finding correspondence between fields in two versions of a class in which either name or type (or both) of the fields have changed. The tool they have developed called DEEPCOMPARE is presented in [Vandewoude05b]. In our work, we focus on the underlying ideas of the approach and as a result of that, we have chosen a simple and to some extend primitive way of migrating state. In future studies, however, a tool such as DEEPCOMPARE would be interesting to integrate in the approach as it would provide the programmer with a more flexible approach to refactoring. We point out that the main ideas behind the approach do not rely on specific state transferring logic.

The work of Gupta et al. [Gupta96] also includes dynamic software updating although it mostly focuses on validating updates. A method that likewise falls into the domain of dynamic updating is the one presented in [Hicks01]. This method, however, targets a C-like language and does not deal with the problems related to object-oriented languages.

[Duggan01] presents a proposal of a new language that relaxes the requirement that all distinct types should be converted when interacting with clients depending on different versions. While the research presented is not related to providing suitable solutions to applications written in existing languages such as Java, it contributes to the area of defining a language that supports dynamic adaptation.

5 FUTURE WORK

One area of interest for future work is the investigation of performance penalty when component size changes. Using an automatic profiling tool to analyze runtime performance of method calls across components could lead to a better configuration of component size. The update manager should handle merge or split operations needed to optimize performance along with additional updates of newly configured components.

Furthermore, we plan to extend the approach with the ability to perform compile-time checks of version compatibility to ensure that no runtime exceptions such as NoSuchMethodException occur.

As stated in section 1, the approach does not handle updates of distributed applications. This support could be added by implementing a transaction manager that controls the coordination of the updates.

6 CONCLUSION

This paper presented the design and implementation of a dynamic update approach for extending eclipse RCP with dynamic update. Using in-place proxification as the necessary indirection mechanism, allows the approach to execute updated code. It thereby introduces an object correspondence issue, which appropriate correspondence mappings handle, thus maintaining object identity. Hence, the approach provides a technique for objects to go back and forth through the version barrier with as little overhead as a hash table lookup. It balances update time with runtime performance by making components the granularity of updates. Experiments show that the approach runs with a moderate performance overhead that would be acceptable for most applications.

REFERENCES

[Bertino95] Bertino E., Guerrini G.: Objects with Multiple Most Specific Classes. In: proceedings of ECOOP'95. LNCS, Vol. 952. Springer-Verlag, (1995) pp. 102-126

[Bialek04] Bialek R., Jul E.: A Framework for Evolutionary, Dynamically Updatable, Component-based Systems. In: ICDCS'04 Workshops, (2004) pp. 326-331

[Chiba00] Chiba S.: Load-time Structural Reflection in Java. In: proceedings of ECOOP'00. LNCS, Vol. 1850. Springer-Verlag, (2000) pp. 313-336

[Drossopoulou01] Drossopoulou S., Damiani F., Dezani-Ciancaglini M.: *Fickle:* Dynamic Object Re-classification. In: proceedings of ECOOP'01. LNCS, Vol. 2072. Springer-Verlag, (2001) pp. 130-149

[Duggan01] Duggan D.: Type-based hot swapping of running modules. In: proceedings of ICFP'01, ACM Press, (2001) pp. 62-73

[Ebraert05] Ebraert P., Vandewoude Y., D'Hondt T., Berbers Y.: Pitfalls in unanticipated dynamic software evolution. In: proceedings of RAM-SE'05 – ECOOP'05 Workshop on Reflection, AOP, and Meta-Data for Software Evolution, (2005) pp. 41-49.

[Eclipse06] Eclipse Foundation, Inc.: Eclipse Platform Technical Overview. http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf

[EUM06] Eclipse Foundation, Inc.: Eclipse Update Manger. http://dev.eclipse.org/viewcvs/index.cgi/%7Echeckout%7E/platform-update-home/main.html

[Gamma95a] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, (1995) pp. 207-218

[Gamma95b] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, (1995) pp. 127-134

[Gosling00] Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification Third Edition. Addison-Wesley (2005), ISBN 0-321-24678-0

[Gupta96] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. IEEE Transactions on Software Engineering, 22(2), (1996) pp. 120–131

[Gustavson05] Gustavson J,: Dynamic Updating of Computer Programs - Proposed Improvements to the JDrums Updating System, Rise publications, 2005.

[Hicks01] Hicks M. W, Moore J. T., Nettles S.: Dynamic Software Updating. In: proceedings of PLDI'01, ACM Press, (2001) pp. 13-23

[JNI06] Sun Microsystems, inc.: Java Native Interface. http://java.sun.com/j2se/1.5.0/docs/guide/jni/

[Joergensen04] Jørgensen B. N.: "Integration of Independently Developed Components through Aliased Multi-Object Type Widening", in Journal of Object Technology, vol. 3, no. 11, December 2004, Special issue: OOPS track at SAC 2004, Nicosia/Cyprus, pp. 55-76.

[Kniesel99] Kniesel G.: Type-Safe Delegation for Run-Time Component Adaptation. In: proceedings of ECOOP'99. LNCS, Vol. 1628. Springer-Verlag, (1999) pp. 351-366

[Lindholm99] Lindholm T, Yellin F.: The Java(TM) Virtual Machine Specification 2nd Edition. Addison-Wesley (1999), ISBN 0-201-43294-3

[Lieberman86] Lieberman H.: Using prototypical objects to implement shared behavior in object-oriented systems. In: proceedings of OOPLSA'86, ACM Press, (1986) pp. 214-223

[Malabarba00] Malabarba S., Pandey R., Gragg J., Barr E., and Barnes F.: Runtime Support for Type-Safe Dynamic Java Classes. In: proceedings of ECOOP'00. LNCS, Vol. 1850. Springer-Verlag, (2000) pp. 337-361

[McManus05] McManus E.: Java API Design Guidelines. Dr. Dichotomy's Development Diary, December 28, 2005. http://weblogs.java.net/blog/emcmanus/archive/2005/12/java_api_design.html

[Orso02] Orso, A., Rao,A., Harrold M.J.: A Technique for Dynamic Updating of Java Software. In: proceedings of ICSM'02. IEEE Press, (2002) pp. 649-658

[OSGi06] OSGi Alliance.: OSGi Service Platform, Release 4 CORE. http://www.osgi.org

[Parnas72] Parnas D. L.: Information distribution aspects of design approach. Tech. Rept., Depart. Computer Science, Carnegie-Mellon U., Pittsburgh, Pa., 1971. Also presented at the 1FIP Congress 1971, Ljubljana, Yugoslavia.

[RCP06] Eclipse.org: "Rich Client Platform" http://wiki.eclipse.org/index.php/Rich_Client_Platform

[Redmond02] Redmond B., Cahill V.: Supporting Unanticipated Dynamic Adaption of Application Behavior. In: proceedings of ECOOP'02. LNCS, vol. 2374. Springer-Verlag, (2002) pp. 205-230

[Sato04] Sato Y., Chiba S.: Negligent class loaders for software evolution. In: proceedings of RAM-SE'04 – ECOOP'04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution. (2004) pp. 53-58

[Sato05] Sato Y., Chiba S.: Loosely-separated "Sister" Namespaces in Java. In: proceedings of ECOOP'05. LNCS, Vol. 3586. Springer-Verlag, (2005) pp. 49-70

[Serrano99] Serrano M.: Wide Classes. In: proceedings of ECOOP'99. LNCS, Vol. 1628. Springer-Verlag, (1999) pp. 391-415

[Sun06a] Sun microsystems, inc.: Java Collections Framework. http://java.sun.com/j2se/1.5.0/docs/guide/collections/

[Sun06b] Sun microsystems, inc.: http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ref/WeakReference.html

[Sun06c] Sun microsystems, inc.: http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ref/SoftReference.html

[Sun06d] Sun Microsystems,inc.: http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/package-summary.html

[Szyperski98] Szyperski E.: Component Software, Beyond Object-Oriented Programming. Addison-Wesley (1998), ISBN 0-201-17888-5

[Vandewoude05a] Vandewoude Y., Berbers Y.: Component state mapping for runtime evolution, In: proceedings of PLC'05. World Academy of Science, (2005) pp. 230-236

[Vandewoude05b] Vandewoude Y., Berbers Y.: DeepCompare: static analysis for runtime software evolution, Department of Computer Science, K.U.Leuven, Report CW 405, Leuven, Belgium, February, 2005

About the authors

Allan Gregersen is a PhD student in computer systems engineering at the Maersk Mc-Kinney Moller Institute, University of Southern Denmark. Main interests are object technologies, meta-programming and reflection. Email Allan at allang@mmmi.sdu.dk

Bo Nørregaard Jørgensen is associate professor in Software Engineering at the Maersk Mc-Kinney Moller Institute, University of Southern Denmark. His current research areas include reflective middleware systems, component-based software development and the design of programming language technologies for dynamic adaptation. Bo can be reached at bnj@mmmi.sdu.dk.