A Classification of Unanticipated Runtime Software Changes in Java

Jens Gustavsson
Linköpings universitet
jengu@ida.liu.se

Abstract

For some software systems with high availability requirements, it is not acceptable to have the system shut down when a new version of it is to be deployed. An alternative is to use unanticipated runtime software evolution, which means making changes to the software system while it is executing. We propose a classification of unanticipated runtime software changes. Our classification consists of a code change aspect, a state change aspect and an activity aspect. The purpose of the classification is to get a greater understanding of the nature of such changes, and to facilitate an abstract view of them. We also present results from a case study, where historical changes to an existing software system have been categorized according to the classification. The data from the case study gives an indication that the Java Platform Debugger Architecture, a standard mechanism in Java virtual machines, is a viable technical foundation for runtime software evolution systems.

1. Introduction

It is well known that software systems evolve over time. New versions are developed in order to remove errors and to change or extend the functionality of the system. Traditionally, a new version of a program is deployed by shutting down the executing version and then starting the new one, perhaps taking some actions to make the state of the new version match the state of the old one. For some systems, this approach is not acceptable, since the interruption disturbs whatever the system is doing. For example, a telephone switching system may be required to have a downtime of less than two hours in 40 years [1]. An alternative is to use unanticipated runtime software evolution, which means making changes to the software system while it is executing. Typically, a program is exchanged for a newer version where bugs have been removed or where the functionality has changed. Runtime software evolution systems are systems that make unanticipated runtime software evolution possible, by providing relevant change mechanisms to the administrator of the system.

The major goal of the work presented in this paper, is to gain a greater understanding of the nature of runtime software changes, which will lead to a better understanding of the requirements on runtime software evolution systems. We do this by proposing a classification of unanticipated runtime software changes, which allows us to group changes into change classes and thereby facilitating an abstract view of them. The classification has been used in a case study where historical changes made to a software system have been categorized according to it. The result is quantitative data on how common different classes of changes are in the investigated software system. We have also investigated how many of the changes can be performed using standard mechanisms in Java virtual machines. This gives an indication of the viability of those mechanisms as a technical foundation for runtime software evolution systems.

By "unanticipated" we mean that the changes are not anticipated by the software developer before the deployment of the system. An example of *anticipated* changes is the plug in architecture of web browsers, where the browser developers has prepared for components to be added at runtime.

The reminder of this paper is organized as follows. Section 2 describes the kind of runtime evolution systems addressed in this paper, which is needed since there are many different ideas about what it is. Section 3 contains our classification of runtime changes. Section 4 describes the case study, including quantitative results and a discussion of it. In section 5, the results from the case study are related to the abilities of standard mechanisms in Java virtual machines. Finally, section 6 contains conclusions and ideas for future research.

2. Runtime evolution system model

There are many different ideas about what runtime software evolution is and how to achieve it. In this section, we describe the kind of runtime evolution systems we address in this paper.

We are interested in having mechanisms in the runtime system that makes it possible to change the running program. When it is time to apply a runtime change, the runtime system must be provided with a new version of the program and a specification on how to transfer the state of the running version of the program to the new version of it. The runtime system must also be provided

Proceedings of the International Conference on Software Maintenance (ICSM'03)
1063-6773/03 $17.00 © 2003 IEEE

with information about timing restrictions on when the switch to the new version may be performed.

Several runtime evolution systems put extra requirements on the application to be updated. For example, Erlang [2] requires the developer to decide which functions will be changeable, and Podus [3] requires the program to be structured in a certain way. We argue that this is a problem for two reasons. First, it complicates the software development, which is already a complex activity, by putting extra requirements on the systems. Second, it makes runtime evolution impossible for systems of which we do not have control of the development, for example when using third-party software. In this paper, we look at unanticipated runtime software changes where the programs that are to evolve are not written with runtime evolution in mind. The runtime environment provides the means for runtime evolution. We have chosen to limit our investigations to the Java programming language and its runtime system.

Runtime software changes can be applied at different granularity levels, i.e. the sizes of the items changed between versions are different. In this paper, the focus is on evolution consisting of changing medium grained entities, such as classes, objects and methods.

When performing runtime software evolution, one option is to let different versions of the changed entities be executed at once. This is used in some runtime software evolution systems, e.g. in Dymos [4]. Disallowing this is called sequential evolution [5]. Since letting several versions of the same code segment to be executed at the same time introduces extra complexity, the work presented in this paper is limited to sequential evolution. We also restrict software evolution to be performed from one complete and consistent version of the software system to another complete and consistent version of it. For example, if the old version defines a method that is removed in the new version, no call to the method may be done within the new version of the program.

3. The classification

Creating a relevant classification of unanticipated runtime software changes is an important step in gaining a better understanding of the nature of the changes, which is one step towards an understanding of the requirements of runtime software evolution systems. Hence, in this section we develop a classification of unanticipated runtime software changes. A preliminary version of the classification has been presented as work-in-progress earlier [6].

The classification is made up of different aspects of the change. The c*ode change aspect* describes the changes made to the program code. The s*tate change aspect* describes what technical mechanisms needed when transforming the state of the running program to a corresponding state of the new version of it. The *activity aspect* describes restrictions on when the update may take place, due to the fact that methods currently executing must be replaced. Each aspect is described in a subsection below. Note that our classification is not exhaustive and it is not the only possible classification. Several other aspects could have been chosen, but we argue that the aspects included are central for describing unanticipated runtime software changes. Every class in the classification has an id number for easy reference.

3.1. Code change aspect

The code change classification describes the changes made to the program code between two versions. The classification consists of a set of classes, listed in Table 1. A change to a software system typically includes several of the code change classes, for example that a method is added and a call to that method is introduced in another method. We are aware of the fact that the classes in the code change classification are ambiguous, i.e. some changes can correctly be categorized in several ways. This is intentional, since we argue that it makes sense to differentiate things like renaming a method, from removing a method followed by adding a new method with a different name but the same method body.

Table 1. Code change classification

Id	Class
1	Comments changed
2	Physical source layout changed
3	Name of formal parameters changed
4	Modifier public/final/static changed for field in interface
5	Modifier abstract/public changed for method in interface
6	Class added
7	Class removed
8	Class renamed
9	Class moved to another package
10	Modifier public added to top-level class
11	Modifier public removed from top-level class
12	Inner class made more accessible
13	Inner class made less accessible
14	Modifier final added to class
15	Modifier final removed from class
16	Modifier abstract added to class
17	Modifier abstract removed from class
18	Modifier static added to inner-class
19	Modifier strictfp added to class
20	Modifier strictfp removed from class
21	Modifier static removed from inner-class

22	Super class of class changed
23	Interface added to implementation list of class
24	Interface removed from implementation list of class
25	Constructor added to class
26	Constructor removed from class
27	Constructor parameter list changed in class
28	Constructor made more accessible in class
29	Constructor made less accessible in class
30	Constructor implementation changed in class
31	Throws class added to constructor in class
32	Throws class removed from constructor in class
33	Instance method added to class
34	Instance method removed from class
35	Instance method renamed in class
36	Instance method parameter list changed in class
37	Instance method return type changed in class
38	Instance method implementation changed in class
39	Static method added to class
40	Static method removed from class
41	Static method renamed in class
42	Static method parameter list changed in class
43	Static method return type changed in class
44	Static method implementation changed in class
45	Method made more accessible in class
46	Method made less accessible in class
47	Modifier final added to instance method in class
48	Modifier final removed from instance method in class
49	Modifier final added to static method in class
50	Modifier final removed from static method in class
51	Modifier abstract added to method in class
52	Modifier abstract removed from method in class
53	Modifier native added to method in class
54	Modifier native removed from method in class
55	Modifier static added to private method in class
56	Modifier static removed from private method in class
57	Modifier static added to non-private method in class
58	Modifier static removed from non-private method in class
59	Modifier synchronized added to method in class
60	Modifier synchronized removed from method in class
61	Modifier strictfp added to method
62	Modifier strictfp removed from method
63	Throws class added to method in class
64	Throws class removed from method in class
65	Static initializer implementation changed in class
66	Static initializer added to class
67	Static initializer removed from class
68	Instance field added to class
69	Instance field removed from class
70	Instance field type changed in class
71	Static field added to class
72	Static field removed from class
73	Static field type changed in class
74	Field made more accessible in class
75	Field made less accessible in class
76	Modifier static added to field in class
77	Modifier static removed from field in class
78	Modifier final added to field in class
79	Modifier final removed from field in class
80	Modifier transient added to field in class
81	Modifier transient removed from field in class
82	Modifier volatile added to field in class
83	Modifier volatile removed from field in class
84	Interface added
85	Interface removed
86	Interface renamed
87	Interface moved to another package
88	Modifier public added to top-level interface
89	Modifier public removed from top-level interface
90	Modifier strictfp added to method
91	Modifier strictfp removed from method
92	Inner interface made more accessible
93	Inner interface made less accessible
94	Super interface added to interface
95	Super interface removed from interface
96	Method added to interface
97	Method removed from interface
98	Method modifiers changed in interface
99	Throws class added to method in interface
100	Throws class removed from method in interface
101	Field added to interface
102	Field removed from interface
103	Field type changed in interface
104	Field initializer changed in interface

Class 8, *Class renamed*, means that a class is given a new name. This change includes necessary changes of constructor signatures to make the constructor names match the new class name.

Class 25, *Constructor added to class*, and class 26, *Constructor removed from class*, might refer to the addition or removal of default constructors. In Java, a default constructor is added to every class that does not have any constructors explicitly defined. This means that adding a constructor that takes some parameters to a class that did not have any explicit constructors earlier, removes

a default constructor. Hence, this would be a code change of both class 25 and 26.

Classes 12, 28, 45, 74, and 92 are all about making items more accessible. This means changing access modifier of the item to a less restrictive one, i.e. moving to the right in the following enumeration of access modifiers: *private*, *default* (i.e. no accessibility modifier specified), *protected*, *public*.

Classes 13, 29, 46, 75, and 93 are all about making items less accessible. This means changing access modifier of the item to a more restrictive one, i.e. moving to the right in the following enumeration of access modifiers: *public*, *protected*, *default* (i.e. no accessibility modifier specified), *private*.

There are no classes in the classification for handling changes in *import* statements. Changing the *import* statements alters what classes and interfaces that are actually referred to in all source code in the same file as the *import* statements. This can have several different kinds of implications, for example that the type of a field is changed or that an inheritance chain is changed. All the implications a change in import statements can have is handled by the existing code change classes and therefore there is no need for explicit change classes for import statements. Import statements in Java, is in fact a compile time issue. In a compiled program, the import statements are turned into the desired use of classes and interfaces in the rest of the code.

3.2. State change aspect

When performing a runtime program change, the new version of the program must continue working where the old version left off. Therefore, it is typically not feasible to start the new version from its initial state. Instead, the state of the old version must be transformed to a corresponding state of the new version. The aim of the state change classification is to describe the state change in terms of what technical mechanisms are needed to transfer the state of the old version to the new one.

We define the state of the running program to include the running threads, data on the method call stacks (e.g. local variables and program counters) and all the objects on the heap. We do not consider a change to the program code in classes to be a state change; rather it is a code change. Similarly, adding or removing a class or an interface is not considered a state change. Runtime changes might require transformation of the state of the environment outside the program itself. Examples of this include transformation of files in a file system and database contents. We consider such state changes to be part of the state change aspect of the classification, although they have all been grouped together in one single class (117). The classes in the state change classification are listed in Table 2.

Table 2. State change classification

Id	Class
108	Object 1 to n mapping
109	Object n to m mapping
110	Change program counter for thread
111	Advanced call stack change
112	Thread created
113	Thread killed
114	Static field value changed
115	Non-static field value changed
116	Local variable value change
117	Environment state change
118	ClassLoader change for class objects
119	Unavailable information needed

Class 108, *Object 1 to n mapping*, means that all objects of a certain type on the heap must be transformed to one or more objects in the new version. This is typically done because of a change in the objects' class. During the transformation of one object, new objects of arbitrary classes may be created. References to these new objects may be stored in fields of the new version of the original object. This makes it possible to replace each object in the old version with an object structure consisting of several objects in the new version. However, there must be one such structure created for each object in the old version, hence the name 1 to n mapping.

Class 109, *Object n to m mapping*, means that a more complex restructuring of the objects of a certain type on the heap, than the one described by class 108, is needed. Different object patterns may have to be transferred to other arbitrary object patterns. The only restriction on the object patterns is that they can be detected automatically using available information from the virtual machine state.

Class 110, *Change program counter for thread*, means that the execution point of one or more threads must be moved. For example, if the execution point is in a specific method it must be moved back to restart the execution of the method.

Class 111, *Advanced call stack change*, means that the call stack of one or more threads must be modified (in a way different from what is described by classes 110, 113 and 116). This includes adding, removing and restructuring stack frames. For example, if the old version of the application calls method M_1 under certain circumstances, and the new version calls method M_2 instead. It might be necessary to replace stack frames that represent calls to M_1 with stack frames that represent calls to M_2.

Class 112, *Thread created*, means that one or more new threads must be created. This includes the creation of object of the Java class *Thread* to control them. Initialization of the states of new threads is also included.

Class 113, *Thread killed*, means that one or more existing threads must be removed.

Class 114, *Static field value changed*, means that the value of one or more static fields must be redefined.

Class 115, *Non-static field value changed*, means that one or more non-static fields in one or more objects must have their value altered. This only includes the values of the fields. If new fields are needed or fields must be removed, classes 108 or 109 are needed.

Class 116, *Local variable value change*, means that values of one or more local variables of one or more method calls on some call stacks must be altered. This does not include the possibility to add or remove local variables.

Class 117, *Environment state change*, means that the state of something outside the virtual machine must be changed. This includes any kind of state change, including but not limited to, file system, other programs, database contents and user behavior.

Class 118, *ClassLoader change for class objects*, regards a rather Java specific issue. It means that the ClassLoader for a Java class must be changed. Each Java class is loaded by a specific ClassLoader, which is an entity that handles part of the dynamic linking of Java classes. Java implies security restrictions on classes depending on which ClassLoader loaded them.

Class 119, *Unavailable information needed*, means that the state transformation requires information that cannot be derived from the state of the program. It also means that the information is unknown or difficult to specify when creating the update specification. An example of this is given in section 4.4.

3.3. Activity aspect

A method is defined as *active* if, and only if, it is on at least one method call stack. That means it has been called but it has not finished executing. A class is defined as active if it has at least one active method or if any of its subclasses are active. The concept of activity is relevant since it can be troublesome to change the implementation of active methods. The problem lies in knowing what to do with executing methods. There are several options, e.g. let the old version continue executing in parallel with the new version, kill the threads containing active methods, or wait until the active methods are not active anymore. Since it is troublesome to change the implementation of active methods it is interesting to know the activity patterns of the classes to be updated. The activity aspect classification gives some information of the activity patterns. There are only three classes in the classification, and they are shown in Table 3.

Table 3. Activity classification

Id	Class
105	Changed classes are often inactive
106	Changed classes are sometimes inactive
107	Changed classes are never inactive

The activity is largely dependent on how the software system is used and is therefore only relevant together with a specific usage profile. This, together with the fact that the activity characteristics are very different from system to system, we have intentionally made the activity aspect classes quite broad. The intention is to allow for a distinction between systems where activity is probable to be a problem when making runtime changes, and the systems where such problems are unlikely. However, the classification does not give a very detailed picture of the activity pattern.

Class 105, *Changed classes are often inactive*, means that using the specific user profile, the classes affected by an update are at least inactive several times a minute.

Class 106, *Changed classes are sometimes inactive*, means that the activity depends a lot on what the application is doing, but sometimes during the usage profile they become inactive. An example is that high workload might make things more active. Another example is that some classes might be very active when a batch job is processed, but very inactive otherwise.

Class 107, *Changed classes are never inactive*, means that the classes affected by the update are always active. In those cases, waiting for them to be inactive is not a viable solution to the activity problem since the wait would be infinite.

Other timing restrictions than the code activity may limit when changes can be applied. For example, changes might not be possible when connected to a database or some code may rely on the fact that a method always returns the same value or the fact that a global variable is constant.

4. Case study

We have performed a case study, where historical changes to an existing software system have been investigated. The changes were made by a third-party without runtime software changes in mind. In the case study these changes has been categorized according to our proposed classification. The purpose of the case study was to get a better understanding of the nature of real world changes. The case study is described in greater detail in a licentiate thesis [15].

4.1. Jetty

The investigated software system is an HTTP and servlet server called Jetty [7]. It is written entirely in Java and contains everything needed to host web sites consisting of both static web pages and dynamic contents created with Java Servlets [8] or Java Server Pages (JSP) [9]. Jetty consists of approximately 40 000 lines of source code and was first released in 1997. It is an open source system, developed by consultants from Mort Bay Consulting [7] with contributions from the open source community. The case study covers changes made from version 3.1.0 to version 3.1.5 of Jetty. This includes almost 3 months of evolution, consisting of almost 100 changes. The source code for these and other versions is publicly available at the Source Forge open source community web site [10]. We argue that Jetty is a system for which runtime software evolution is relevant. Today, introducing new versions with bug fixes and feature enhancements of the Jetty server requires the server to be stopped and restarted. With this follows that the web application running within it can be interrupted. If Jetty was executed in a Java virtual machine with runtime evolution enabled, this interruption could be avoided.

4.2. Methodology

In the case study, we classified changes made between different versions of the Jetty server. Even though those changes were made without runtime software evolution in mind, we argue that they are representative of what changes would be useful at runtime. We do not only classify the actual changes that are made statically, but we also analyze what would be needed to perform the changes at runtime, for example what state transfer would be needed.

We consider an *improvement* to be what has been checked in into the Jetty version control system, CVS, at once or what is described as an improvement in the Jetty release documentation. If these two things do not have the same size (i.e. a described improvement is implemented with several check-ins or several described improvements are checked in at once), the smallest one has been considered as an improvement. There are a few exceptions where several check-ins are part of the same improvement. This is done when we found the check-ins to be so much related that it does not make sense to introduce the changes in one of the check-ins and not the other. This definition of an improvement allows a very different amount of changes made in an improvement. A small improvement might only be a minor change to a single class, while a large improvement might affect large portions of code in several classes. We have chosen not to take notice of changes made to test code that has the sole purpose of being a tool during development of the application, such as test code. We have also chosen not to take notice of changes to comments made by the version control system, CVS, which changes version numbers and dates in source code comments automatically.

In order to find out the activity level of the individual classes in Jetty, a usage profile must be specified. We choose a very simple profile where static web pages, JSP-pages and pages generated by Servlets were requested periodically. 30 different clients sends a get-request each, every 10 seconds. After a while, the load is lowered to nothing in order to examine how this affects the activity of the classes in the system. This usage profile does not include all tasks Jetty is capable of but the most central tasks are covered.

4.3. Quantitative Results

We found that 92 improvements were made to the Jetty system in the time interval selected in the case study. Table 4 shows in how many improvements a certain code change class was used. It also shows what percentage of the 92 improvements this corresponds to. The code change classes that are not in the table were not used in any improvement.

Table 4. Usage of code change classes

Id	Class	Total	%
38	Instance method implementation changed in class	81	88%
1	Comments changed	35	38%
33	Instance method added to class	35	38%
68	Instance field added to class	21	23%
30	Constructor implementation changed in class	20	22%
34	Instance method removed from class	14	15%
6	Class added	13	14%
71	Static field added to class	12	13%
44	Static method implementation changed in class	11	12%
69	Instance field removed from class	10	11%
36	Instance method parameter list changed in class	6	7%
2	Physical source layout changed	5	5%
70	Instance field type changed in class	5	5%
96	Method added to interface	5	5%
22	Super class of class changed	4	4%
39	Static method added to class	4	4%
40	Static method removed from class	4	4%
97	Method removed from interface	4	4%
7	Class removed	3	3%

24	Interface removed from implementation list of class	3	3%
35	Instance method renamed in class	3	3%
37	Instance method return type changed in class	3	3%
63	Throws class added to method in class	3	3%
66	Static initializer added to class	3	3%
23	Interface added to implementation list of class	2	2%
27	Constructor parameter list changed in class	2	2%
29	Constructor made less accessible in class	2	2%
59	Modifier synchronized added to method in class	2	2%
64	Throws class removed from method in class	2	2%
65	Static initializer implementation changed in class	2	2%
72	Static field removed from class	2	2%
84	Interface added	2	2%
8	Class renamed	1	1%
25	Constructor added to class	1	1%
26	Constructor removed from class	1	1%
31	Throws class added to constructor in class	1	1%
45	Method made more accessible in class	1	1%
46	Method made less accessible in class	1	1%
60	Modifier synchronized removed from method in class	1	1%

Table 5 shows in how many improvements a certain state change class was used. It also shows what percentage of the 92 improvements this corresponds to.

Table 5. Usage of state change classes

Id	Class	Total	%
108	Object 1 to n mapping	21	23%
114	Static field value changed	4	4%
109	Object n to m mapping	3	3%
115	Non-static field value changed	2	2%
117	Environment state change	2	2%
118	ClassLoader change for class objects	1	1%
119	Unavailable information needed	1	1%

Table 6 shows in how many improvements a certain activity class was used. It also shows what percentage of the 92 improvements this corresponds to.

Table 6. Usage of activity classes

Id	Class	Total	%
105	Changed classes are often inactive	68	74%
106	Changed classes are sometimes inactive	15	16%
107	Changed classes are never inactive	9	10%

We do not make any statistical analysis of the quantitative results, since we do not intend to make any general claims. What we do claim is that the results reflect the changes made to the investigated versions of Jetty.

4.4. Discussion of case study

Different improvements are of different sizes, ranging from some minor change to a single class to major restructuring of several classes and their relation to each other. In the case study, the improvements have not been weighted against any size measure, since we have not found any size measure that seems relevant. This means that although a large number of improvements are relatively simple to introduce at runtime, we have not shown that they represent a large part of the total evolution. It could be that the small improvements are simple while the large ones are more difficult.

A general impression we got when we did the actual classifications is that state transformation is very complex for nontrivial scenarios. We believe this is a major issue for the person that is to specify the transformation. This means future research should not only look into how to implement mechanisms for state transformation in the runtime systems, but also how to create tools and methods that makes this task easy enough to be useful.

We argue that it is impossible to create a runtime evolution system that can handle all desirable changes, since there is always a possibility that information that can only be collected at a certain time is not stored and therefore unavailable when it later is needed for performing a runtime change. A solution is to let the update specification provide this information, but this is not always viable. We found the following example in the case study. In the old version, a class contains a status field that can have any of the three values *none*, *integral* and *confidential*. When new objects of the class are created, the status field is set to *none*. In the new version, a new possible value *undefined* is introduced and this value is given to all new instances. When transforming the state of the old objects, it is not known whether the status

fields with the value *none* in the old version corresponds to the value *undefined* or to the value *none* according to the semantics of the new version.

Limited scope of the runtime system can also be a limit of what changes that can be performed. For example, in the case study we only considered the Java system on the server side as part of the runtime system. Changes that require change of state in the web browsers used as clients are therefore impossible. One example from the case study is that changing URL parameter format, will fail if done only on the server side. The address field of the web browsers will still contain parameters in the old format. Another example from the case study is that the server program changed the way it saved some data to file. Making this change as a runtime change would require a mechanism to modify the contents of the existing file system. In order to change user experience, the user must be considered as part of the runtime system, i.e. it must be possible to make the users change their behavior at the correct time.

5. Java Platform Debugger Architecture

During the case study, we also investigated how many of the changes can be performed using standard mechanisms in Java virtual machines. This gives an indication of the viability of them as a technical foundation for runtime software evolution systems.

The Java Platform Debugger Architecture (JPDA) is the standardized debugging mechanism in the Java 2 Platform [11]. It provides an infrastructure for building end-user debugger applications. JPDA contains interfaces that supports everything that is typically included in debugger applications, such as possibility to inspect and modify values of variables, inspect contents of call stacks and to suspend and resume the execution of threads. Dmitriev has proposed extensions to the JPDA interfaces to make them support runtime changes of the running program [12]. The proposal contains descriptions of four different stages of the functionality for runtime program change:

- Stage 1: Change of method bodies only.
- Stage 2: Binary compatible changes, except changes to instance formats.
- Stage 3: Arbitrary changes, except changes to instance formats.
- Stage 4: Arbitrary changes.

Binary compatible changes are changes that cannot make the running program become inconsistent. For example, adding a method is a binary compatible change, while removing a method is not binary compatible, since there might be calls to that method in the running code [13]. Stage 4 allows changes to instance formats, i.e. fields in classes may be added, removed and changed. This means that the size of objects might change. Stage 1 is implemented in the Sun JDK 1.4 virtual machine and the following stages might be made available in coming versions of it. There exist prototype implementations of stages 2 and 3, even though they have not been made publicly available [12].

The JPDA is a natural choice as a technical foundation for a runtime software evolution system for Java since it is standardized by Sun, who is leading the development of Java. The JPDA is not a runtime software evolution system in itself since it lacks things like definitions of how state mappings should be specified and when updates should be performed. Neither are there are any tools to facilitate runtime evolution.

Table 7 contains information on what stages of the JPDA is required to perform changes in the case study at runtime. Using only mechanisms from JPDA stage 1, 37% of the changes can be performed, using mechanisms from JPDA stage 2, additionally 9% of the changes can be performed, and so on. There are a significant number of changes, 14% that cannot be performed using only the mechanisms provided by JPDA stage 4. Examples of reasons for this are active code, a need to change things outside the virtual machine, and need to access unavailable information.

Table 7. Required JPDA stages for changes

Required JPDA Stage	Changes
Stage 1	37%
Stage 2	9%
Stage 3	7%
Stage 4	33%
Stage 4 not sufficient	14%

These results give an indication that JPDA is a viable technical foundation for creating runtime software evolution systems. The current implementation (Stage 1) seems to be viable to a substantial amount of changes. Stage 4 will contain mechanisms that are sufficient for most changes. Stages 2 and 3 does not seem to make that much of a difference. The reason for this can be that most small changes only require changing method implementation, which is possible with stage 1, and most large changes require fields to be added or removed, which requires stage 4.

6. Conclusions and future work

In this paper, we have proposed a classification of unanticipated runtime software changes. Using the classification in a case study, we have collected data about

how frequent changes of the different classes was used in a real world software system. This is a step in gaining a better understanding of the nature of changes, which is one step towards an understanding of the requirements of runtime software evolution systems.

Since runtime software evolution is not widely adopted, finding the reasons why this is the case is interesting. The work presented in this paper addresses the flexibility in runtime software evolution systems, i.e. what kinds of changes systems should be able to perform, given correct specifications of the changes. In this area, more work on gaining statistical data would give us a better understanding of the nature of changes that are desired in practice. For this it could be worthwhile to refine our classification, for example by adding new aspects, such as advanced timing restrictions. However, maximizing the benefit of an updating system means trading off various criteria [14], which means that flexibility is not the only thing to strive for. Another important criterion is usability. We are interested in investigating how to help developers handling the complexity involved in specifying runtime changes by researching how tools and methods that help the developers in this respect should be designed to give a high level of correctness. One idea for such a method is runtime refactorings [15].

We are in the process of developing a runtime software evolution system, based on the Java Platform Debugger Architecture (JPDA). The aim of this system is to give us a research platform for investigating the issues described above. Our investigation of JPDA indicates that JPDA is viable as a technical foundation for such a system.

7. Acknowledgements

This work has been supported by Vinnova 2GAP (Second Generation Application Provisioning) and EU EASYCOMP (Easy Composition in Future Generation Component Systems).

8. References

[1] M. E. Segal and O. Frieder, "On-the-fly program modification: Systems for dynamic updating," *IEEE Software*, vol. 10, pp. 53 - 65, 1993.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in Erlang*, second ed: Prentice Hall, 1996.

[3] O. Frieder and M. E. Segal, "On Dynamically Updating a Computer Program: From Concept to Prototype," *Journal of Systems and Software*, pp. 111-128, 1991.

[4] I. Lee, "DYMOS: A Dynamic Modification System," University of Wisconsin, 1983.

[5] G. Kniesel, J. Noppen, T. Mens, and J. Buckley, "Unanticipated Software Evolution," in *Object-Oriented Technology, ECOOP 2002 Workshop Reader*, J. Hernández and A. Moreira, Eds.: Springer-Verlag, 2002, pp. 92-106.

[6] J. Gustavsson and U. Assmann, "A Classification of Runtime Software Changes," presented at First International Workshop on Unanticipated Software Evolution, Malaga, Spain, 2002.

[7] "Mort Bay Consulting web site," vol. 2002, 2002. http://jetty.mortbay.org/jetty/index.html

[8] D. Coward, "Java Servlet Specification, Version 2.3," Sun Microsystems, 2001. http://www.jcp.org/aboutJava/communityprocess/final/jsr053/index.html

[9] E. Pelegrí-Llopart, "JavaServer Pages Specification, Version 1.2," Sun Microsystems, 2001. http://www.jcp.org/aboutJava/communityprocess/final/jsr053/index.html

[10] "Source Forge web site," 2002. http://sourceforge.net

[11] "Java Platform Debugger Architecture," vol. 2002: Sun Microsystems, 2002. http://java.sun.com/products/jpda/

[12] M. Dmitriev, "Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications," presented at Workshop on Engineering Complex Object-Oriented Systems for Evolution, Tampa Bay, Florida, USA, 2001.

[13] B. Joy, G. Steele, J. Gosling, and G. Bracha, *The Java(tm) Language Specification*, Second ed: Addison-Wesley, 2000.

[14] M. Hicks, "Dynamic Software Updating," in *Department of Computer and Information Science*: University of Pennsylvania, 2001.

[15] J. Gustavsson, "Towards, Unanticipated Runtime Software Evolution," in *Department of Computer and Information Science*. Linköpings universitet, 2003. ISBN 91-7373-630-9.