UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Computer Science Curriculum

Taivo Käsper

Runtime UI Reloading for Java

Master's Thesis (30 ECTS)

Supervisor: Allan Raundahl Gregersen, PhD

Supervisor: Luciano García-Bañuelos, PhD

Tartu 2015

Runtime UI Reloading for Java

Abstract:
In order to test if an edit works as expected, developers have to go through many routine tasks. Depending on the application given it may span from multiple seconds to minutes per change. Developer tests a change on average three to four times per hour - it is obvious that the process is excessively time-consuming. To minimize the amount of time, dynamic class updating software, like JRebel, can be used.

Sadly, up to the present moment, none of the available products support reloading the UI of a running desktop application. For that reason, the aim of this thesis is to bring out the possible approaches that would enable runtime UI reloading for Swing programs. Accordingly a plugin UiReload for JRebel is implemented which can detect what was changed and therefore apply the modifications to running UI components whilst losing none of the existing state.

In order to reach the expected result some major challenges had to be overcome. For a start, it was necessary to determine what was changed and which objects were affected by those editions. Secondly, to make sure how to propagate those changes to a running application. Thirdly, to ascertain how to create new instances of components and finally, how to replace an old instance of a component with new, in a way that the fresh version maintains the state of the previous one. For finding the state an automatic approach was devised.

As a result, with UiReload, developers can test the changes in milliseconds. This implies to the fact that there is no more need to restart the application, navigate to the changed view nor fill in forms on the way. Overall, testing changes reaches the maximum speed and simplicity.

Keywords: Runtime, user interface, reloading, dynamic updating, JRebel, Java, Swing

Käitusaegne kasutajaliidese uuendamine Java programmidele

Lühikokkuvõte:
Tarkvaraarendajad peavad tegema hulga rutiinseid töid, et testida kirjutatud koodi muudatuse tulemust. Olenevalt rakenduse keerukusest võib neil selleks minna paarist sekundist kuni paari minutini. Oma töö tulemust soovitakse katsetada tunni aja jooksul keskmiselt kolm kuni neli korda, mis tähendab märkimisväärset ajakulu teiste täitmist vajavate tööülesannete arvelt. Seda probleemi saab aga vältida, kui kasutada mõnda dünaamilisel viisil klasse taaslaadivat tarkvara, nagu seda on näiteks JRebel.

Kahjuks tänasel päeval ei toeta mitte ükski olemasolev toode kasutajaliidese uuendamist töölaua programmide jaoks, nende töötamise ajal. Käesolevas lõputöös otsitakse lahendusi, et tuua käitusaegne kasutajaliidese uuendamine Swing teegiga kirjutatud programmidele. Selleks loodi pistikprogramm, mis täiendab JRebeli olemasolevat funktsionaalsust ning suudab tuvastada arendaja tehtud muudatused ja kanda need üle töötavasse rakendusse, ilma et seal juba olevad andmed kaduma läheksid.

Soovitud tulemuse saavutamiseks oli vajalik lahendada järgmised probleemid: kuidas leida, mida arendaja täpselt muutis, kuidas uuendused saaksid rakenduse töötamise ajal jõustuda, kuidas luua uus isend ükskõik millisest kasutajaliidese komponendist ning seejärel kindlaks teha, kuidas vanalt isendilt uuele andmed üle kanda, kui pole teada, mis täpselt üleviimist vajab.

Selle töö tulemusena saavad arendajad kasutada pistikprogrammi nimega UiReload, mis koostöös JRebeliga võimaldab neil näha enda töö tulemusi vaid millisekundite jooksul. Seesugune kiire tulemus oli saavutatav ainult lahendusega, mis välistaks tarkvara taaskäivitamise, õigesse kohta navigeerimise ja vormide täitmise vajaduse.

Võtmesõnad: käitusaegne, kasutajaliides, taaslaadimine, dünaamiline uuendamine, JRebel, Java, Swing

Contents

1. Introduction

1.1 Context

Java is a language with one of the richest ecosystems. This fact and the high number of developers on the market is often the number one argument when choosing a language for the next project. According to TIOBE Index[1] from February 2015, Java is the number two language in popularity (just after the C).

It is often visible - especially on popular developer forums - that Java for desktop applications is considered dead [2], that Swing is slow and its application programming interface (API) is overly complicated. It is also noted in the forums that Swing is in the maintenance mode and will only receive bug fixes, consequently getting ready to be left behind by Oracle [3]. It is true that Oracle has released a second version of Java FX and by that hints not to use Swing for new projects. Then again, no public statement has been made.

Regardless the technology, there is some support for the claim that Java is still used for developing desktop applications. By looking at the open source repositories on GitHub, the author can conclude that every year at least 10% (10.31% in 2014 and even more in previous years) of the Java repositories created are desktop applications. These numbers are visualized in Figure 1.1. It is important to note that the numbers might be skewed due to comparing just public repositories and the fact that Java libraries are more likely to be open source than desktop applications.

[1] The ratings are calculated by counting hits of the most popular search engines [1].

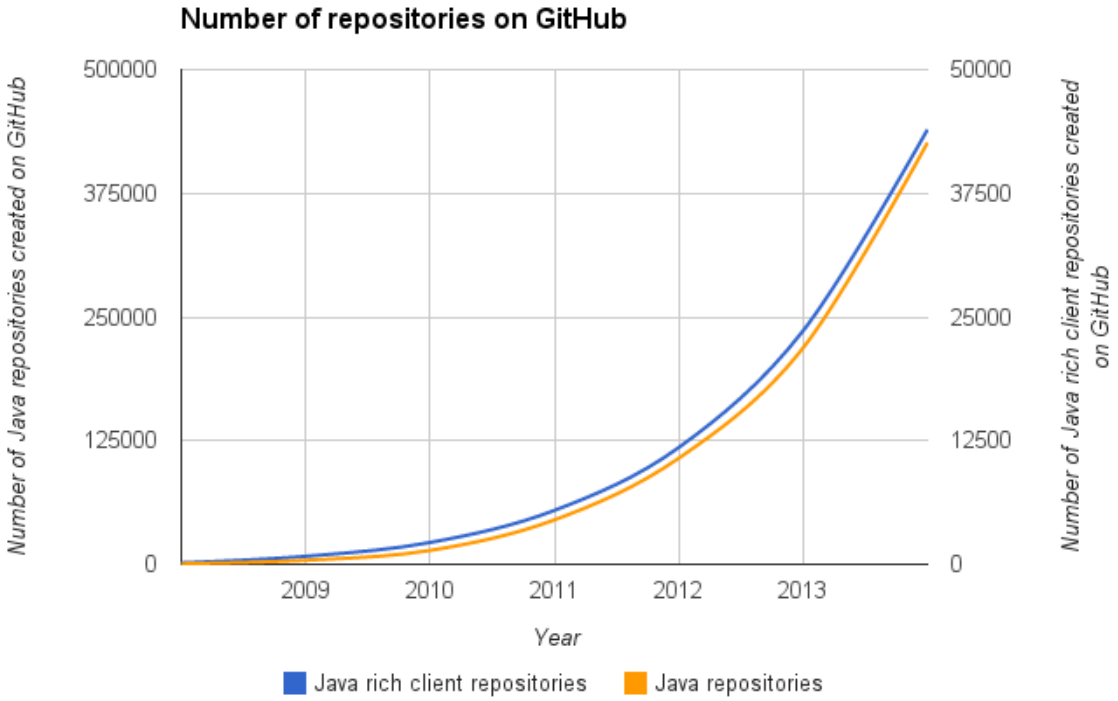

Figure 1.1: Number of Java repositories on GitHub and Java rich client applications. Data for each year is found with the following search term: "swing OR java fx OR desktop OR RCP created:2014-01-01..2014-12-31 language:Java"

The idea of using Java in an upcoming project is abandoned worryingly more often due to the difficulty exposed by recompiling, restarting and navigating after every small change in the code base - just to see the modifications in action. When using an interpreted language like JavaScript, this is interchanged with the lack of performance and compile time validation. When looking at web based applications - all the programmer has to do to immediately check the changes is to press the refresh button in the browser. During this era, when software is far more expensive than hardware and developer time costs exorbitant amounts of money, the decision is clear. That is why we lately hear so much about the birth and rise of different interpreted languages.

The Java Enterprise Edition (Java EE) standard was developed without keeping in mind the turnaround time[2] from making a change in source code, building an archive (package of Java bytecode), deploying it to an application server, waiting for it to get to a state where it can respond to requests, navigating through the application to the place where the code change was made and validating whether the same process needs to be repeated. According to a survey conducted by RebelLabs technology blog by ZeroTurnaround, the average turnaround time across all application servers is approximately 2.5 minutes [1]. This has been reformed by a

[2]Turnaround is the time it takes for the changes in code to propagate to the running application. It includes build time, deploy time, startup time, navigation to the view in the application and filling in forms on the way [4].

product called JRebel, that can propagate the changes done by developer into a running web application, without the need to do any of the described steps. What is left to do for the developer is to request the updated page by refreshing.

The World Wide Web (WWW) has been conceived to be stateless from the very beginning. The notion of state was introduced later to provide user specific content, in form of authentication, to tie separate requests to a particular user. Thereof the parallel multi-user environment started supporting single user activities. Most of web application frameworks have implemented a session storage, where developers can save and read the state of a user. This means that storing and retrieving any such data has to be specifically handled by the developer. Usually it is small in size, does not change in structure and because of how HTTP servers are engineered, using URL endpoints, does not contain any navigation information and other such metadata for displaying correct content.

Desktop applications have from the very beginning been used by one user at a time. This means that the development practices to not follow the same patterns as web applications do. There is no one specific place where the state is stored, but instead it is scattered across different objects used to represent the UI components. This means that the developed solution must first collect the information and provide means to update the changed subset of UI components so that the collected data is carried from reload to reload and updated if needed. Therefore the approaches used by JRebel for web applications do not suffice.

1.2 Motivation and problem description

The current workflow of a developer consists of multiple tools that each have a task to perform for deploying and starting a new version of an application. First an integrated development environment (IDE) is used, to edit the code. Then build tools like Ant, Maven or Gradle, for compiling and building an archive of the project, with all the dependencies. That is followed by deployment that is done using a terminal. Sometimes these steps are performed through IDE, that has the capabilities, but still the same tasks have to be done. Next the developer uses the created application to navigate to the changed view, by providing input where required, and checks if the edit was successful. If not, then the process starts from the beginning.

Overall when following the preceding workflow, the turnaround from edition to validation can, depending on the complexity of the application, take from tens of seconds to multiple minutes. Decreasing the time close to zero in all cases is believed to significantly increase the developer productivity. This can only be done by introducing a new workflow.

It is wanted that the IDE detects the changed classes and compiles them. Then the class change is propagated to a running application and the state created by the previous version of this class is processed as if it was made by the new version. Only then it is possible for the changes to affect a desktop application.

It becomes increasingly clear from the workflow description that a lot of development time is spent on doing some of the most time consuming tasks over and over again. This is were the reader is introduced to skipping some of the steps, which is an introduction to runtime UI reloading, allowing the developer to edit the source code and see the results immediately.

1.3 Objectives

1.3.1 Full state preservation

State - State is at any given time the information stored in memory. In object oriented programming pieces of data stored in fields are encapsulated into objects and all the objects in memory form the program state. Sometimes term important is used to refer to a subset of information that is essential and can not be derived from other pieces of data.

This project promises to remove the necessity to create the same state after every program restart. It is done by removing the need for restarts, by adopting advanced state transfer from class version to version. When an element on the UI is replaced, then the newly created object is processed so that it holds the previous object's state. This results in reloads where components on the UI can be switched but the important state is preserved.

The data conversion from one structure to another, that is done with a custom converter, cannot be automated. A simple yet complex to solve example would be a drop-down menu, replaced with a checkbox. It is out of the scope of the thesis to deal with the issues that include for example automatic yes/no string to boolean true/false conversion. Therefore the state to be transferred has to to suit what the recipient expects.

1.3.2 Transparent reloading

From the adaption point of view it is most beneficial to create a tool that can work transparently. This allows to demo automatic UI element reloading with any software and makes introducing the runtime[3] UI reloading to an existing project as simple and fast as possible - no need for any code changes, dependencies, rules or learning a new framework.

It gives a feeling to developers that they are in charge of the tools which they adopt. Forcing them to use a specific library in the code would mean that choosing any alternative or abandoning the use of this project in the future would not be feasible due to the amount of changes needed to make in the codebase.

[3]Runtime is a state of the program where it has already been executed

1.3.3 Decrease turnaround time

It is aimed to decrease the turnaround time to milliseconds. This is not achievable by making any step described in Section 1.2 faster but rather skipping most of them - no need to recompile the whole application, package, restart and recreate the state.

1.4 Scope and limitations

An existing dynamic class updater is extended by creating a plugin called UiReload. For this extension JRebel is chosen. The plugin analyses what is necessary to make the edits visible, fully functional and executes the strategy that fits best. Thus it attempts to address all the set objectives.

It is understood that programming languages are versatile. Thereof being fully transparent while understanding the indent of the developer is not possible. For this a minimal set of constructs are provided that help to understand the structure of the code - it is a balance between flexibility and transparency.

1.5 Thesis organization

This thesis is organised as follows:

Introduction - introduces the problem that is the motivation of this thesis. Describes the objectives that are to be achieved and adds context to the issue.

State of the art - gives background into this domain. Introduces related work and places this project into the field. Explains how already existing solutions to similar problems are used.

Swing and runtime reloading concepts - introduces Swing which is followed by a conceptual view of the problem at hand. Then real world examples of the desired results are given. The examples follow throughout the thesis and act as the test cases for the created plugin. Finally the major challenges to be surpassed are described and solutions provided.

Towards a working prototype - describes how the chosen concepts are implemented. Additionally points out challenges that were detected during the implementation. Finally gives detailed information on how the ideas are integrated with JRebel without accessing the internals of the product.

Case study and project evaluation - evaluates the success of the UiReload plugin. Discusses on what objectives were reached and what could be improved.

Conclusion - concludes the work by making a summary on how the objectives were reached. Then discusses what features are currently missing and how to improve it to a quality product.

2. Swing and runtime reloading concepts

This chapters serves the following purpose:

- It introduces Swing and its internal structures for holding visual components as data.
- It presents an example application.
- It provide detailed examples for what is expected from the UiReload plugin.
- It points out the challenges and enumerates alternative solutions.

2.1 Swing framework

Before giving an explanation of Swing, the Abstract Window Toolkit (AWT) has to be introduced. AWT is a platform independent Java toolkit that provides an interface between the JVM and the underlying operating system in order to create native graphical elements. With this method, Java developers are able to create applications that operate identically on any platform. The applications also feature the platform-specific appearance, which is sometimes desired – sometimes not.

Swing is a successor to AWT. It mostly uses plain Java to draw its components with the local graphics subsystem (Java 2D). This gives the components a distinct look and feel when the painting process is not exploited to a great extent for custom appearance. The Swing UI consists of top-level containers, intermediate containers and atomic components. Combined, these components form one containment hierarchy for every top-level container. An umbrella term **component** is used to refer to all three types.

Top-level container - The main purpose of the top-level container is to provide a content panel (content pane) for the children to paint themselves onto. The only way to display any non top-level component is by adding it to a containment hierarchy. These commonly used Java types are:

- javax.swing.JFrame
- javax.swing.JDialog

- javax.swing.JApplet

They are the subtypes of java.awt.Container and not javax.swing.JComponent that is otherwise extended by every Swing component. With support from the operating system, top-level containers display a window specific to the platform as shown in Figure 2.1.

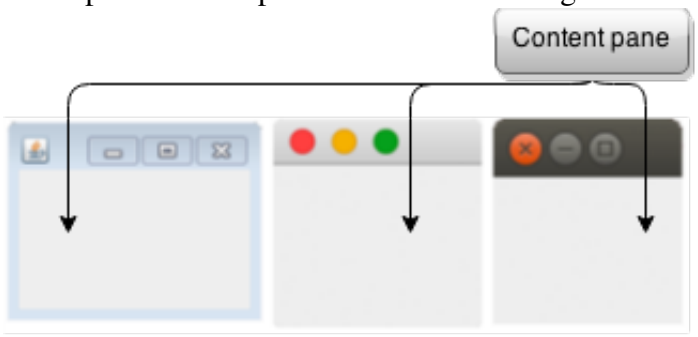

Figure 2.1: JFrame top-level container on different operation systems. From left to right: Windows 7, Mac OS X Yosemite and Ubuntu 14.4.

Intermediate container - JPanel is the most frequently used intermediate container. It is often used to group related components together or assist with their layouting. They can also be used to create reusable building blocks of component groups which act as a whole and form one domain specific section. Intermediate containers use Java 2D for painting and therefore have a consistent look across all platforms. Common examples are:

- javax.swing.JPanel
- javax.swing.ScrollPane
- javax.swing.TabbedPane

Atomic component - Components that enable the communication between the application and its end user by presenting data and/or accepting data. An atomic component is a single item that inherits from the `javax.swing.JComponent` class and has a consistent appearance on all platforms. Some examples of these components are:

- javax.swing.JButton
- javax.swing.JCheckBox
- javax.swing.JTextField
- javax.swing.JLabel
- javax.swing.JTable

Every Swing application is built using one top-level container that holds intermediate containers. These in turn hold atomic components and in this way the containment hierarchy is formed. Often times, action listeners are added to components. The action listeners track user interactions and are a place holders for code to improve the user experience - e.g. calculations that change the UI structure on completion.

From now on, package names from types are excluded. The types mentioned most often are either from package java.awt or javax.swing.

2.2 Example application

To showcase the use of components and reloading mechanisms, a small application called **Account Registration** is developed and used as an example throughout the document. The application allows the user to fill in personal details, including contact information, address and other preferences to register a fictional account. It is used to mimic the activities done by the developers, provide support for some use cases, find limitations and test the prototype of the UiReload plugin. The application could be anything using Swing that wants to take advantage of the power of the plugin.

For testing purposes, the UI is refined by adding, removing and changing the elements and their properties, as well as adding events and basic data validation. All these developments are performed with JRebel and the UiReload plugin enabled. The difference is in the power of performing the changes without restarting the application.

A snapshot of our test application is shown in Figure 2.2. It consists of four reusable building blocks called panels (instances of JPanels), which are then linked together to create the full contextual experience. Initially, it is assumed that panels and the application cannot validate data and can only reset all the fields to empty values - or print out the data. Such application would usually have required fields like name, email validation against a pattern and - depending on the domain - some business logic checks (for instance when a newsletter is ordered, filling the email field is mandatory). The speech bubbles on the UI are used to display how components link to tree nodes in the containment hierarchy shown in Figure 2.4.

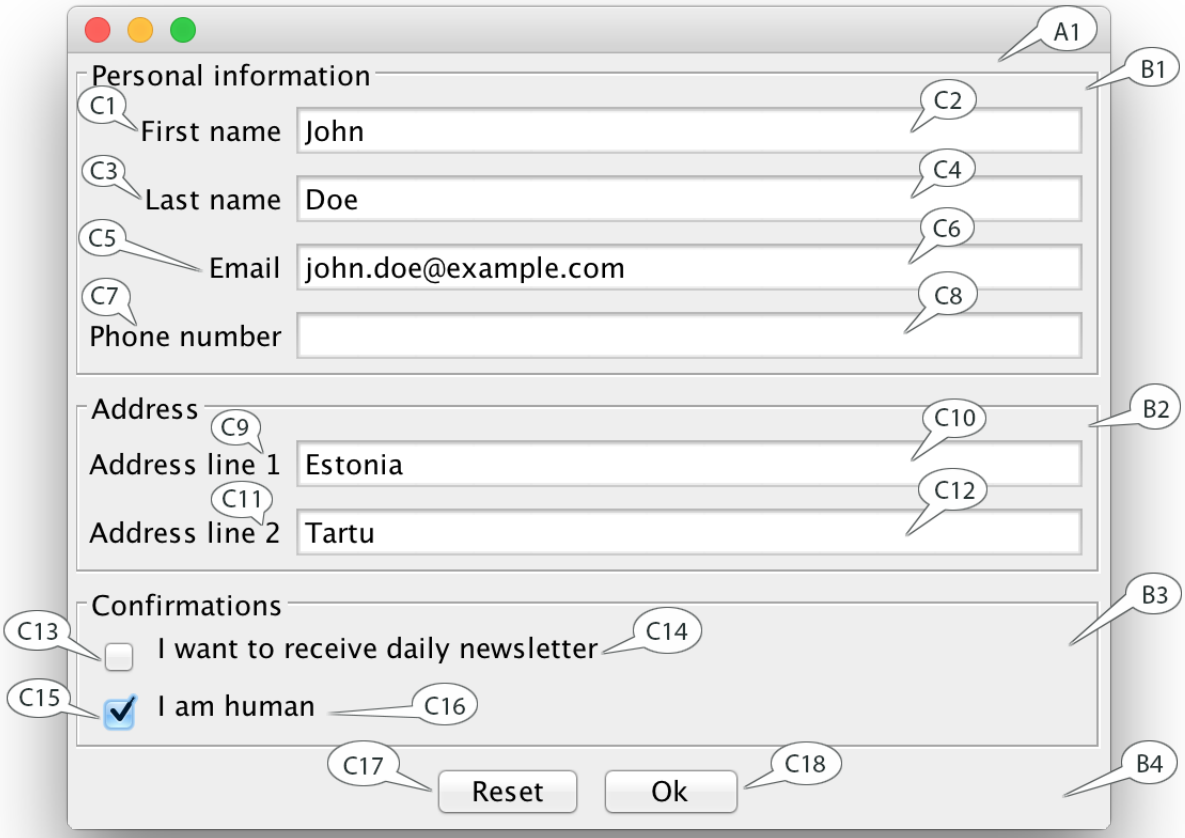

Figure 2.2: Initial example application with added references to Figure 2.4 in speech bubbles.

2.3 The source code changes

The developer wants to press the OK button shown in Figure 2.2 after every edit and validate whether the correct output is displayed for validation errors. Should the developer modify the elements on the UI, for example from Figure 2.2 to Figure 2.3, the developer would want to see the new elements on the UI while coding.

Lets assume the developer wants to add a validator for the "email" field. To this end he/she fills in the corresponding field with a value. When the developer adds the code for validating the "email", it is expected that the UiReload plugin would keep the pre-filled values and that the validation is done against that value.

When writing code, components and their behaviors are added iteratively and tested as the software advances, making the turnaround time grow more and more as restrictions need to be met for every iteration. This means that to test the entered email against a pattern, the developer

first has to fill out all required fields. When testing validation for the newsletter checkbox, the previously filled fields plus the correct email are needed - yet after every change, followed by a restart, the fields are emptied.

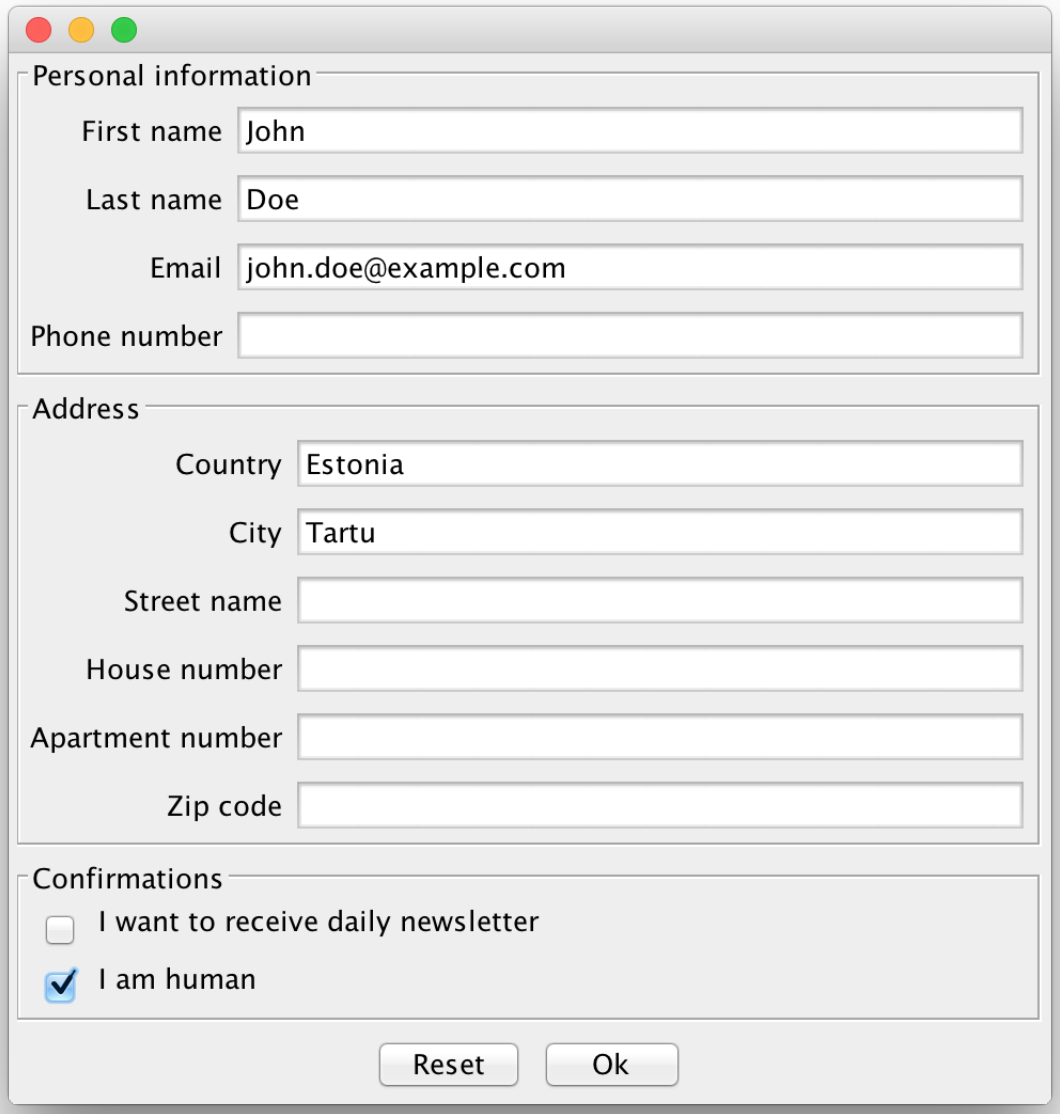

Figure 2.3: Example application after adding fields with runtime reloading.

A common development pattern occurs when the developer adds a component and then checks whether it is correctly aligned with existing components. If not, it is fixed and checked again. Secondly, he/she ties the component to a model so that the data is accessible when submitted. This is then verified. Third, the developer adds validation to direct the user through the application flow with as little mistakes as possible. Again, this is then verified. Repeatedly

going through the application flow, getting stuck at one point, fixing the problem, starting from the beginning until getting stuck in the next step and so on - until the feature is complete - generates the smallest amount of value by the developer. It is desirable that the developer can fix a problem and continue from the step he/she got stuck in. However, this is not possible without JRebel and the UiReload plugin. Currently, this tedious process is considered the norm.

When editing the UI, the developer changes the source code that represents it. This means arranging the components in the desired manner, configuring them and performing other structural changes. The changes do not happen directly to the UI, but rather to the containment hierarchy structure that in turn becomes the UI. This means that the UiReload plugin has to be able to reconfigure the structure and then display the updated version of the UI.

2.4 UI structure and changes

The state of the UI elements can be represented as a N-ary tree, where every node can have n number of children, with n being theoretically limited only by the capacity of Integer. The leaves (atomic components) and internal nodes (intermediate containers) in the tree must be of the type *JComponent* that inherit *Container* from AWT. The reloading effects by the UiReload plugin have to manipulate the nodes of the tree: without breaking its structure, creating cycles or referencing objects not accessible through the root of the tree (top-level container).

The Window displayed in Figure 2.2 is translated to a containment hierarchy shown in Figure 2.4. Every component of the UI needs to have a two-way path to the top-level container, otherwise displaying it is impossible. It is also worth mentioning that the developer can accidentally create circular dependencies between child-parent relationships, but Swing cannot handle those.

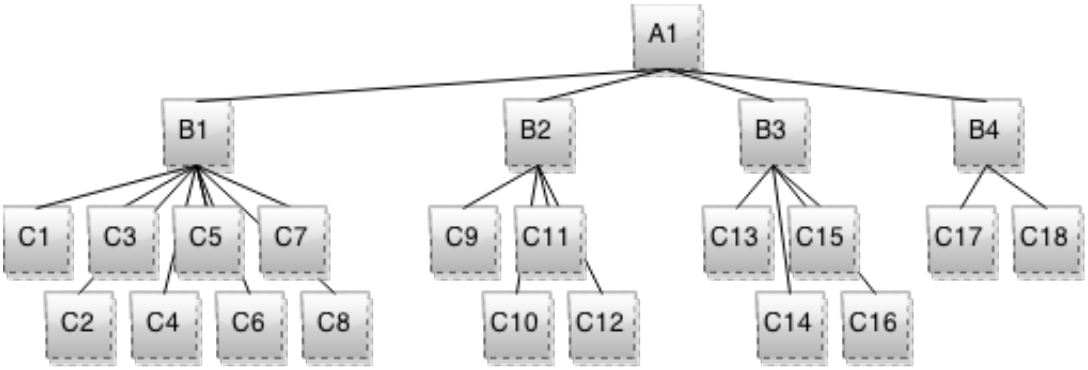

Figure 2.4: Containment hierarchy of the UI shown in Figure 2.2.

The containment hierarchy in Figure 2.4, starting from the top, associates directly to the UI shown in Figure 2.2. The levels of the tree represent the items that are within each other and follow the parent-child relationship. Items of the same depth are displayed side by side. The root node A1 is the whole window that corresponds to the top-level component with the *JFrame* type. Its children B1-B4 are within the window, next to each other, representing the panels **Personal**

information, **Address**, **Confirmations** and the **Action** panel at the bottom. For example, the **Personal information** panel, node B1 at the top, contains 8 components from C1-C8. The odd numbered children are the labels in front of the inputs and even numbers are the inputs corresponding to Java type `JTextField`. It can also be that the tree contains hidden UI nodes, which are only shown conditionally. The current example does not contain them.

The following notion is defined here and used from later on.

Updated method - A public, no arguments, void method with name `javeleon$$updated` shown in Listing 2.1. It is used by JRebel to notify that the class of the object has been changed and successfully reloaded. This is intended for implementing some custom reloading approach that is not properly handled by JRebel or for debugging. In the domain this is sometimes called a state transfer function.

```
1 public class MyClass {
2     public void javeleon$$updated() {
3
4     }
5 }
```

Listing 2.1: Updated method example

When a developer modifies the UI for example the node B2 represented by **Address** panel in the way hinted by the snapshot Figure 2.2 and Figure 2.3 then the hierarchy has to be reconfigured starting from the corresponding node's branch. To this end, a helper class can be used that is aware of the containment hierarchy. This helper class is notified by JRebel by calling the `javeleon$$updated` method, indicating that the class has been updated. The method executes code to replace any component as required. By using bytecode rewriting the code in this helper class can be entirely injected into the target class, removing the need for separate class. As this notification approach is chosen by JRebel, we adopt its convention. The method is showcased in Listing 2.1 and called **Updated method**.

Back to the example, lets assume that the developer changed two of the label texts (C9 and C11), left two inputs as they were (C10 and C12) and added 4 new labels with corresponding to inputs C21-C28 as shown in Figure 2.5. When an algorithm detects from the source code that the text was changed for 2 labels, it calls the `setText` method for only these two instances to change the text. The two can be found by analsing the source code and matching it to the existing instances of nodes, added to container B2. By executing only the lines added, it creates the new instances of labels and inputs shown in Figure 2.5. Because these are new elements, it knows

that they have to be added to the container B2. Thus, it analyses the source code that initialized the others in the container and from there detects in which order and between which existing nodes to insert the new components. In the current example, it detects the need to append at the end of the container. All that is left to do is to paint the intermediate container B2 to display the changed structure.

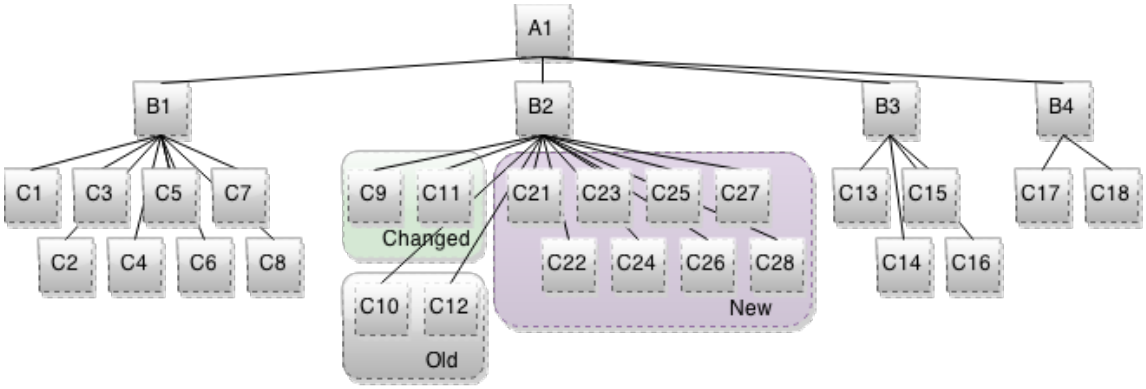

Figure 2.5: Containment hierarchy of the reloaded UI shown in Figure 2.3

Unfortunately, the desired behavior with the provided description in previous paragraphs is impossible because this expects the following:

- The UiReload plugin to have an intelligence comparable to humans.
- That there is source code which can be analysed in needed detail. There is actually only the Java bytecode and making static code analysis on it to receive the needed level of detail is not possible.
- That it knows what methods to call when something as simple as a text label has been changed.
- That the initialization source code can be compared to an instance and then detected that the concrete instance is the one created by those lines of code. It can happen that these lines are called multiple times and multiple contextually equal instances are created. There is no way to match the lines of code to instances and/or it is virtually impossible.

2.5 Challenges, ideas and solutions

In the following, the main challenges and some alternative solutions are discussed.

2.5.1 Detecting changes

To propagate the changes to the running UI, it is necessary to find the precise items that have been changed and their details. An ideal solution for the algorithm would be to return a set of changes, where a change is represented with an object that has the following:

- Metadata about the change type. Whether it is an **add**, **delete** or **change**.
- References to the changed objects.

2.5.1.1 Sufficient changeset data

The changeset refers to the data structure that has to be actionable. This means that it has to have enough detail. What does this mean? The most precise change description could be given by providing the fields of an object that have a different, changed value. This could mean a map of maps that has the structure shown in Listing 2.2. In such a way, a map of all affected objects is created and the fields that were meant to change can be changed to their new value. This helps to keep track on changes that have to be installed when refreshing a running application.

```
{
    changedObject1: {
        field1: newValue1,
        field2: newValue2,
        ...,
        fieldN: newValueN,
    },
    ...
}
```

Listing 2.2: Details of a change. Keys with prefix "field" are field names and values objects.

2.5.1.2 Collecting the changeset

Now that it is known in what form the output data has to be, it is necessary to collect the data by analysing what the developer did. The options for this can be divided to two major categories: static analysis and dynamic analysis. These methods are popular for finding bugs and security breaches in software, but can also be used for different reasons.

Static analysis - Analysis performed without executing the program code [5]. In this context, it means interpreting the Java bytecode accessible whenever a new version of a class is loaded at runtime, comparing the version of bytecode before and after the change and picking out the difference to corresponding objects in memory.

Dynamic analysis - Analysis performed by executing the program code and looking at what it does [6]. In this context, it means re-executing parts of the code after the change and comparing the previous output and side effects with the current. This analysis allows to identify the change-set.

When comparing Java bytecode changes using static analysis, it could easily be found whether some string or constant value passed to some Swing method is changed. An additional problem is to identify the subset of objects and fields to update.

There exist two main solutions to this problem:

Code analysis approach

During runtime, the UiReload plugin could access the stack of the JVM and by analysing it can find, where the program flow came from and what where the bytecode changes done to the classes in the flow. By going backwards, it could leverage the power of static and dynamic analysis and propagate the affecting changes. It might happen that the modifications break the flow and the UiReload plugin has to figure out on a broader level what was done by the developer - so that the reconfiguration of the instance is not possible because it should no longer exist.

Replacement approach

At runtime, the plugin knows that the UI components have a containment hierarchy. From there, it can expect with false positives that a parent object knows how to create its children. It is then possible to leverage the knowledge and recreate the child with the modified configuration - the configuration being done by the parent. This means that collecting the changes is no longer needed. When we can replace the changed child - though the state created by the user that was held in it, is lost.

The second option, **Replacement approach**, is selected because it has the most uniform solution without any exceptional cases. This is described later on: how to find and transfer the state from the replaced component to the replacement component.

2.5.2 Changeset propagation

If the first solution, **Code analysis approach**, was selected, the collected changeset would have to be applied to an existing object. With the selected solution this is not necessary. However, a similar approach is used to transfer the state from one object to another - therefore a description is given.

JavaBean approach

A JavaBean convention could be used. This means that for each field a setter method is provided.

The drawback of this approach is that a setter method can perform additional tasks such as validation, that could interfere with runtime replacement. For instance sometimes the behaviour is wanted like for example setting a text to a `JTextField`, that afterwards notifies Swing, that the value has been changed. Additionally, this does not enable to set

private fields that are deeper in the hierarchy, which are not meant to be changed and which match with fields in subclasses. To summarize, this approach is not uniform, leading to undesirable side effects.

Reflective approach

Alternatively, **Reflection** (it comes with Java API) could be used to overwrite all the fields from the changeset in the object hierarchy. It enables to maintain control over what is done, but falls short on domain specific actions - when for example a listener has to be fired to notify that a value has been changed. Fortunately, a repaint of the UI usually fixes such problems.

For this problem, the alternative **Reflective approach** is selected because it provides the greatest amount of control.

Reflection - *Reflection is the integral ability for a program to observe or change its own code as well as all aspects of its programming language (syntax, semantics or implementation) - even at runtime. A programming language is said to be reflective when it provides its programs with (full) reflection.*[7].

2.5.3 Automatic state detection

The plugin does not fulfill the expectations when it replaces the changed elements in the container and loses the state of the previous element. Defining the state fields for every Swing component is not possible - because of custom code written by the developers and a great number of extension libraries. This means that the plugin has to have a mechanism to detect fields that have to be transferred over from the replaced component to the new.

Almost every field in the object hierarchy of the component can be considered a state holder. Some filtering can be done to exclude certain false positives - including static and transient fields, fields with certain types like *javax.swing.plaf.ComponentUI* and manually found field names like "isAddNotifyComplete", "isFocusTraversableOverridden", "highlighter" and "acc". The list of excluding conditions is not final and can grow for optimization reasons. This is not necessary for a prototype.

To find important data created by the user, we can detect what is changed in the object during the time the user interacts with the UI. Some solutions are provided for this scenario.

Tracking changes

The most obvious solution is to track the fields that are changed. For this, the initial value - when a component is added to a container - is saved upon the change event. This can

be done by intercepting Swing methods to save the argument value of a method, if it is called. This does create a lot of overhead, because there is no way to know which methods are actually important to track and which are not. Matching method calls to fields is not possible because of inconsistent naming. It is also too complex when customized UI elements are added by developers - because no structure is known.

Tracking changes with accessor methods

Instead of intercepting all methods to save the last argument that they were called with, it is possible to modify the bytecode of the program at runtime to direct the field access through accessor methods. The setter could be used to register the last value of the element which could then be compared to the value that was registered when the component was added to a container. This approach has a shortcoming when the setter method already exists - it is not known what else the method might do in addition to setting the value. This also involves a lot of class bytecode instrumentation, because all the classes have to be analysed and processed to use accessor methods instead of fields.

Comparing copies

When a component is added to the container (**UI component**), its state can be preserved by creating a deep copy of the object (**Original component**). When a reload is triggered by changing some UI component's configuration, a new version of it is instantiated (**New component**). A deep clone is then created and the difference of the Original component and UI component are transferred to the New component by using reflection to inject the values for the fields in the change set. This approach is shown in Figure 2.6. Afterwards, New component becomes the UI component and the Original new component the Original component in the drawing. In reality only two versions of the same component are held - the component on the UI (in the containment hierarchy) and its initial copy.

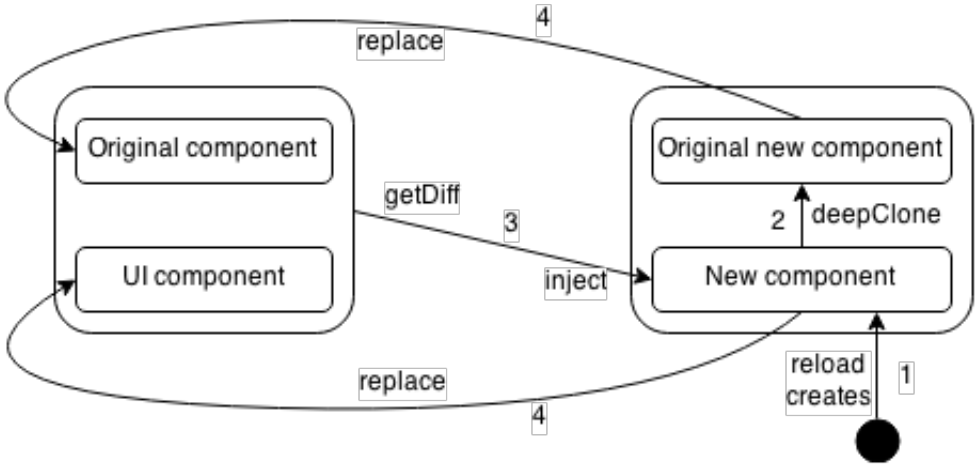

Figure 2.6: Automatic state detection concept with an execution order.

The **Comparing copies** approach is chosen to provide the state for newly created components

because it has the least amount of probability for errors. This is due to it avoiding error prone bytecode instrumentations, and not having any side effects. Its behavior is also predictable.

2.5.4 Creating a new instance of a component

All the selected approaches are based on creating a new instance and replacing it in the UI instead of reconfiguring an existing object. There is no uniform way to find the code that instantiates and configures a component and to just re-execute those lines. This has a crucial part in the UiReload plugin - to be able to provide input for the other solutions selected so far. Next, the solutions are provided.

Containment hierarchy followed with class architecture

This approach expects the parent to know how to instantiate its children. This means that the instantiation and configuration code for a child must be in the class that represents the parent. This assumption requires the developers to write more reusable code which in turn is beneficial for them and also offers some structure of knowledge for the UiReload plugin.

Individual methods with naming conventions or annotations

If the plugin knows in which class to look for the code by using the **Containment hierarchy followed with class architecture**, it only has to filter out the lines interested in instantiating and configuring a specific component. One possible solution for filtering out the lines is in providing a naming convention - where every child must have a method that handles the creating and configuring. A suggested naming convention could be including the order number in the method name: e.g. "child0", "child1" etc. This allows to easily re-execute lines of code and in conjunction, create a new component with updated configurations. Alternatively, annotations could be used to provide information on which method deals with which child.

This solution has some major drawbacks. It expects the developer to divide code to small functions - one for each. Also, the methods have to be called so that the program works when the UiReload plugin is not used. This complicates situations where a configuration would depend on another component because the code is separated into methods. Finally, it could with almost complete certainty break references, because all the existing components that were not re-instantiated can have a reference to a component that was replaced, thus being removed from the containment hierarchy and not shown anymore.

Following GUI builder patterns

There are some frequently followed coding patterns. Some of them were born because of drag-and-drop GUI builders that were used that generated source code which has a certain structure. These patterns can be exploited and used for understanding the structure of the code without analysing it.

GUI builders often (but not always) create a class for every item in the containment hierarchy. The same was already described in **Containment hierarchy followed with class architecture**, thus it is used but extended. Two popular builders: Swing GUI Builder and WindowBuilder Pro both create a separate method **initComponents** and **initialize** correspondingly. This consists of all the code that creates, configures and adds the children to itself (from now on this approach is referred to **initComponents method**). Meaning that whenever any child has been changed, all the children must be recreated.

To proceed the following definition has to be introduced. **Following GUI builder patterns**.

InitComponents method - Every container class that has children is expected to have a recallable method which creates the instances of the children and adds them to the enclosing container. This method can be called multiple times during the development life cycle. It cannot have more side effects than the creation and adding of children.

The **InitComponents method** is showcased in Listing 2.3. It is important that no child is created in a constructor of the class, but in this method. Some additional configuration can be done elsewhere, like shown in the example with the `changeInputTextColorTo` method. When the input text colors where changed and the reload created all new children, the color changes would not be lost but they would be discovered as a state of the application with subsection 2.5.3 **Comparing copies** and transferring to the new objects.

```
1  public class MyFrame extends JFrame {
2      private JTextField firstInput;
3      private JTextField secondInput;
4
5      public MyFrame() {
6          setLayout(new FlowLayout(FlowLayout.LEFT));
7          initComponents();
8      }
9
10     private void initComponents() {
11         firstInput = new JTextField(30);
12         add(firstInput);
13
14         secondInput = new JTextField(30);
15         add(secondInput);
16     }
17
```

```
18      public void changeInputTextColorTo(Color color) {
19          firstInput.setForeground(color);
20          secondInput.setForeground(color);
21      }
22  }
```

Listing 2.3: InitComponents method convetion example

Analysing the stack and bytecode

In Java, the runtime stack of the program can be analysed and information retrieved in a manner similar to how the stacktrace is printed on exceptions. This means that it is possible to iterate backwards in the program flow and find methods that created a new item.

From backtracking the stack, we can find the location where an object we are interested in was created, what methods were called on it etc. The information is not simply available, but it can be collected from the stack of the program. This allows finding the methods that need to be re-executed with the same parameters as previously, to create and configure a new component.

Unfortunately, when method calls on an object are added to source code, they are not present in the stack. Making the added lines of code to be findable and executable in a reliable manner close to impossible.

For recreating changed components, the solution **Following GUI builder patterns** is chosen. This offers the greatest amount of reliability when the developer follows some patterns, but breaks the total transparency statement. In the first iterations of the plugin, the transparency is very important (but not a requirement).

2.6 Conclusion

The following has been achieved in this chapter:

- The foremost purpose was achieved and reloading of an example Swing application introduced. We described the visual change that the project tries to achieve, what structural changes take place in Swing and how are the components of the UI stored in the computer's memory.
- The main use cases were introduced that are going to be supported.
- Description of the major challenges that the project faces were given. What can be done to overcome or postpone those complexities was discussed and the drawbacks of each analysed. Best solutions for following work were selected.

3. State of the art

The chapter carries the following purpose:

- It discusses relevant related work.
- It describes industrial solutions supporting runtime reloading of Java applications and their suitability as workbench for this work.

3.1 Related work

The demand for modifying the behavior of a running application dates back to early computers and thus does not involve only Java but other procedural and class-based object oriented programming languages. Back in 1998, G. Hjálmtÿsson and Robert Gray investigated how to introduce dynamic C++ classes as a lightweight mechanism to update code at runtime [8]. Their approach was not in any way transparent and limited the use of constructs available in the chosen language. As a future work they also planned to provide dynamic class updating for Java.

For Java it has been a long awaited feature - to dynamically update already loaded classes. In 2002, Java 1.4 JVM version introduced an experimental feature titled HotSwap in the Debugger API. In Java 5 this feature was incorporated and made directly available to Java applications through the Instrumentation API. It allowed to redefine only a method body but was a good basis for the subject.

Project UpgradeJ did research in 2008 on how to enable class versioning at runtime, in order to add new versions of a class. A major restriction is that the changes cannot change the field and method signatures nor add or remove them. [9]. This made the project good for applying small hot patches and bug fixes in method bodies. However, this approach is not sufficient for continuous software evolution.

A. R. Gregersen, D. Simon and B. N. Jørgensen argue that simple extensions to an existing JVM can bring full flexibility and transparency to dynamic updates in Java back in 2009 [10]. It was said that for the best results in the dynamic updating, a separate VM is required. In light of the module system about to be introduced in Java 7, it is a better suited platform for increasing the update granularity. Some examples are provided, where support from the runtime would greatly improve the dynamic updating. Like updating Swing class JFrame, that is not final and has native methods and/or fields. Intercepting native methods has no effect, when the VM sets the

field directly, thus the flow cannot be proxied to the most up-to-date instance and field listeners are used instead.

In separate work [11], the same authors analyse the tradeoffs between flexibility and transparency. The main concern is how to provide the most transparent class evolution framework while supporting the largest possible set of dynamic updates. Whilst exclusively being close to transparent there are situations where loss of state happens, although others provide manual converters for state and thus do not have to tackle the problem.

In runtime UI reloading for Java, balancing between the two has been done, even though full transparency is wanted. This enables to reload the Swing UI with the updated classes so that no specific help from the VM and dynamic class updating framework is required. Relatively high transparency is achieved by having only a few guidelines that need to be followed.

It is also acknowledged that the development of a dynamic software updating system for statically-typed object-oriented programming languages has turned out to be a challenging task [12]. This applies to this thesis, because there can currently be support only for updating **Intermediate container**s and **Atomic components**, which fortunately covers most of the use cases.

Others have approached the problem from a different perspective by suggesting implementing a separate update-enabled VM. This includes **Dynamic Code Evolution Virtual Machine**, JVolve and research by Malarbarba. All except for Dynamic Code Evolution Virtual Machine are quite restrictive: Malarbarba suggests allowing redefinition at a single class level, ruling out any greater change, and JVolve cannot change class hierarchies plus is relatively slow in performance-wise.

3.2 Discussion

The main reason that the dynamic software updating has not become popular until the recent years has been the lack of approaches that support the workflow of a common developer. So far the paths taken to boost the developer productivity and experience have been associated with the ability to reload changes to classes. But the research has fallen short with respect to processing the runtime state of the objects already in memory to mirror the done changes. This means that the domain has a gap that is not been considered so far, though needs to be filled to allow to UI reloading at runtime.

The example given in Listing 3.1 shows why class updating is not enough. If the program is running, the constructor has already called the **initComponents** method that has added two components to the JFrame container. Now, if a developer modifies the class and thus existing objects, so that the method added three components, all new instances of MyFrame class would do that but the existing instances have two items in the container. Therefore the update has to be processed to mirror the desired changes to the existing objects for every `MyFrame` instance to contain three items.

```
1  public class MyFrame extends JFrame {
2      public MyFrame() {
3          initComponents();
4      }
5
6      private void initComponents() {
7          add(new JTextField());
8          add(new JTextField());
9      }
10 }
```

Listing 3.1: Example of why state processing is needed.

All the research so far is biased toward web applications, where an incoming request executes most of the code written by developers. In the Listing 3.1 example this would mean that the MyFrame class would be instantiated by every incoming request. In desktop applications has no notion of a request and the existing components are created and painted onto a canvas just once.

This project intends to fill the described gap. For this it still requires the ability to update classes but additionally, it needs to mirror the changes from the source code to the already existing application state. For these reasons it extends one of the dynamic class updaters by creating an application level plugin.

Each of the products listed in Section 3.3 could be selected as the extension point. JRebel is chosen to conduct the experiments with because the author had the opportunity to work closely with the developers of ZeroTurnaround, the owners of the product.

3.3 Solutions for dynamic class updating

A subset of technologies enabling dynamic class updating, to increased developer productivity, are selected. It is not a finite set but it definitely contains the most mature and state-of-the-art products available on the market.

3.3.1 Dynamic Code Evolution Virtual Machine

The Dynamic Code Evolution Virtual Machine (DCE VM) is a modification of an existing high-performance virtual machine that allows arbitrary changes to the definition of loaded classes [13]. Because of the access to the low level resources the implementation has no negative performance impact on normal program execution.

It supports adding, removing fields and methods as well as changes to the super types of a class [14].

3.3.2 Spring loaded

Spring Loaded is an application level dynamic code updater. It uses a JVM agent to inject its logic to the application classes during the time they are loaded by classloaders to enable updating. This means that it has to use Java to enable new functionality in the JVM. For example: Grails 2 has integrated the project to its internals to allow the developers to evolve the software at runtime.

It supports adding and removing fields, methods and constructors. The annotations on types, methods, fields, constructors and values in enum types. It is not able to change type hierarchies. [15]

3.3.3 JRebel

JRebel is the only commercial product in the listing. It works in a similar fashion to **Spring loaded**, by using a JVM agent to transform the application classes at load time and to enable dynamic class updating. It is the research and work by Gregersen that has introduced the ability to change class hierarchies to the project. [16]

JRebel can do all that **Spring loaded** and **Dynamic Code Evolution Virtual Machine** can. Additionally, it adds the ability to update the configuration for most of the web frameworks that would otherwise be loaded only during the startup, rebuild caches and rewire components.

3.4 Conclusion

It is seen that there has been interest into the topic for quite some time and it is not only the problem of Java programming language. So far, all solutions are provided for web applications that do not share the same merits with desktop applications that are for reasons set aside.

In this chapter the following is achieved:

- Existing research is introduced and its maturity discussed.
- Products solving the needed dynamic runtime class updating are discussed and compared. A suitable product is selected.
- It is discussed why updating classes is not enough in some cases and how the current project links to the previous research.
- The uniqueness of the approach and solution is discussed.

It is believed that the domain has become aware and strong enough to allow to build another product upon. Therefore without the listed projects and research the current would be impossible.

4. Towards a working prototype

This chapter has the following purpose:

- Describe how the given concepts in **Swing and runtime reloading concepts** are implemented.
- Point out implementation challenges, proposes solutions and selects one.

After achieving a basic understanding of what is needed to do to reload the changes, a solution based on the choices performed in **Swing and runtime reloading concepts** is implemented. A modular design with an open/closed principle is kept in mind, to keep the application open for extension, but closed for modification [17]. This allows to experiment faster by only writing new code to add functionality, meaning no time is spent on modifying the old to include the changes.

4.1 Container oriented

In Section 2.5.1 (**Detecting changes**) it has been argued that it is not attainable to collect the exact changes done to a component. The approach was not suitable because it required to describe the logic for reloading and finding state for too many classes - meaning the software could never be used for a project that uses custom libraries.

Therefore a container oriented method is used instead. This means that instead of describing logic for every individual component, a method is described to update the content of a container with new elements. An easy state ignoring example is provided in Listing 4.1. A completely separate logic, described in Section 2.5.3 (**Automatic state detection**), is used to preserve the state of the components even when they are replaced with a new instance of the element.

```
1 public class ReloadableContainer extends JPanel {
2     private static final Logger LOG =
3             LoggerFactory.getLogger(ReloadableContainer.class);
4
5     public ReloadableContainer() {
6         initComponents();
7     }
8
```

```
9       public void initComponents() {
10          add(new JTextField("Default value"));
11      }
12
13      public void javeleon$$updated() {
14          removeAll();
15          initComponents();
16          revalidate();
17          repaint();
18          LOG.debug("Container content reloaded");
19      }
20  }
```

Listing 4.1: Replacing container content on reload

Swing relies heavily on containers to display components. This can be seen right from the beginning, when a developer has to use the top-level container `JFrame` to create a visual frame to display its components. `JFrame` extends the `Container` class that has a number of methods from which we are interested in add, remove, removeAll as the basic functions to manipulate the content of the container. The container class is inherited by a great number of commonly used classes like JFrame, JPanel, JDialog, JApplet, so making bytecode manipulations to inject the logic of this project is the only logical approach.

4.1.1 Intercepting container methods

The starting point for this project is JRebel's **Updated method** (see Figure 2.4), that will trigger the entire process. This method has no input on what has been changed, which can be both fortunate and unfortunate. Fortunately, this makes the initialization process much easier and not dependent on JRebel. The UiReload plugin could easily be moved to use any other dynamic code reloading framework. Unfortunately, this also means that the project has to collect all the necessary data by analysing inputs, outputs and side effects of the algorithm, creating the component instances and building the UI. In general, the data about the edits of the source code is collected by intercepting the methods that pass the results to the container and comparing them.

To collect data, the add methods should be intercepted in a way shown in Listing 4.2[1]. This means that the actual functionality of the method is not changed but rather extra purpose is added. It might be confusing why the extra data holder abstraction is needed when the elements could be retrieved from the `Container` class itself by calling the `getComponents` method. The thorough explanation will be provided in sections later in the thesis. However, at this point the

[1]The modifications to existing classes are done with overriding at first. This means that the subclasses must be used instead.

explanation could be that there is no other way to retrieve the constraint object used with the component when placing it in a container.

```
public class ContainerWithInterceptedAdd extends Container
             implements Interceptor {
    private DataHolder dataHolder;

    @Override
    public Component add(Component comp) {
        add(comp, null);
        return comp;
    }

    @Override
    public void add(Component comp, Object constraints) {
        dataHolder.add(comp, constraints);
        super.add(comp, constraints);
    }
}
```

Listing 4.2: Container data collecting interceptor example

4.2 Re-executing code

The greatest obstacle to overcome is the need to re-execute parts of code during the reload. This raises concerns because due to the need to be transparent, it cannot be expected that programmers have followed some guidelines. Nor can they be forced to use any helper libraries or structures so that it could be more easily understood which parts of the code can be re-executed and where it is located.

By analysing the possible solutions in Section 2.5.4 (**Creating a new instance of a component**), it is clear that it is needed to find a balance between flexibility and transparency and therefore the most reliable solution, **Following GUI builder patterns**, is selected. This means that the developers are expected to follow guidelines. These are not new, which means that there are already projects following these guidelines. This also makes it immediately usable for some.

Reflection is used to find the common methods used by GUI builders like **initComponents** and **initialize**. Next, using bytecode rewriting, the container add methods are intercepted, as described in Section 4.1.1 (**Intercepting container methods**), and the found method is executed. This way it is possible to collect the changes without influencing the UI where not needed.

If none of the methods are found, the developer is notified that for the best results he/she should follow the GUI builder pattern.

4.2.0.1 Timely behavior of interceptors

Container add methods are intercepted for analysing the contents of the **InitComponents method** method, without affecting the existing UI.

The action of an interceptor depends on the state of execution of the underlying application. Three different possible states for reloading are identified:

Method pass-through interceptor

Acts as if the method was not intercepted by passing the arguments forward to the original method and returning the result if there is one.

This interception type is used to put the reloaded components back on the UI without triggering the reload and the state transfer logic again.

Regular interceptor

This interceptor is used to collect all the data that is put in the container during the regular behavior of the application.

One sample usage is to collect the components that are put in a container during the application startup period - so that the data could be used to find what the UI had before the reload was triggered.

On update interceptor

The interceptor is used to catch the components that are created and put in the container during the code re-execution phase (while initComponents is executing). The data is then compared to the previous information, collected by **Regular interceptor** and processed, before actually put back in the container.

Once again, the open/closed principle is used to not lock down what the interceptors do. Instead, the right logic is provided every time an intercepted method is triggered - this keeps the system open for extension and closed for modification.

4.2.0.2 Automatic intercepting

The following notion is described here and used later on.

Javassist - Is a Java Programming Assistant that simplifies Java bytecode manipulation. It is a library for editing Java bytecode during the time when a class is first loaded by a classloader. It allows to add new classes at runtime, modify every aspect of a class during its first load and does not require the programmer to know the bytecode specification. [18]

UiReload requires lots of bytecode rewriting. To this end UiReload uses **Javassist**. A major drawback of Javassist is that it requires as input the Java code in form of String. This is error-prone because the validation happens at runtime - when the transformed class is first loaded by a classloader. A solution is created to provide compile time validation and comfort.

To intercept a method A it is needed to move the original body to a separate method B and forward every call of A to a matching method C in the interceptor object. This process could be easily automated.

While loading the class, an algorithm is given an interface and an implementing object. The interface lists all methods that are to be fully intercepted, by delegating the flow to the implementing object. The object could then do whatever needed and even call the original method body.

In Section 4.2.0.1 (**Timely behavior of interceptors**), it is described that depending on the reload state, different interceptors are needed. This can be done quite easily by providing different objects implementing the interface at the correct time.

This solution allows making easy further improvements to this project, without the need to manipulate the bytecode. The easier it is to make complex things, the more drastic ones are done - this could have both a positive and a negative result.

4.3 Automatic state detection

It was described in Section 2.5.3 (**Automatic state detection**) how the state is transferred from object to object during a reload. It was also explained that it is not so simple to define what a state is, because it can be virtually any field in an object hierarchy.

There is no fully working and attainable manner to manually declare instance fields that hold state versus those that - regardless of their value - should not be regarded as state. This presents an introduction to automatic state detection, which is used instead to carry on the state fields throughout the reloads from object to object.

4.3.1 Difference as a state declaration

To transfer the important application state from a replaced component to a new one, an automatic approach is taken. For this, a multiple version path is taken, as described in Section 2.5.3 **Comparing copies**.

The approach relies heavily on deep cloning, but in Java the object is cloneable only if it is marked with an interface Cloneable, otherwise an exception is thrown. It is needed to clone Swing instances which are not intended to be used in such a way and even contain some native fields and methods.

Next, some solutions are provided on how to overcome this complication and to still create deep clones or, if not possible, at least partial. Often a full deep clone is not needed because the important state is stored in fields, which can be copied easily.

4.4 Cloning component hierarchies

To freeze the component values it has during the time it is placed on the UI, deep cloning is used. This means that the **Original component** and **UI component** (see Section 2.5.3) have no reference to the same object. When a user interacts with the component: clicks, hovers, resizes or provides any input, only the **UI component** is affected. This helps to easily find the changes at any given time by comparing the two.

Deep cloning is an activity where a copy of the cloneable object is created. Then, a copy of all the objects referenced by it. This object tree or a graph grows exponentially in size, meaning that when deep cloning is used, care has to be taken that not too many objects are cloned. This yields another problem where a solution needs to be found - to filter out the objects that with a high probability do not hold an important state, but because the developer can do virtually anything he or she wants, there is no way to be sure when something important is not filtered out.

Multiple cloning methods are implemented which can be easily interchanged in the application logic. The best of these methods is picked based on both performance and reliability.

4.4.1 Serialization cloner

What is eventually desired is to clone the state and for this Java has a good mechanism with a marker interface `Serializable`. This can be used for that very purpose - to create clones. Using serialization and deserialization in conjunction, a first prototype is created of the idea described in this chapter.

Firstly, serialization is usually applied for data objects and the developer has to explicitly implement the Serializable interface. Because this project is all about transparency, this is the last thing that goes through a programmer's head to do - to think about making something that for the application purposes is not needed to be serializable, but for reloading the UI components has to be. This means that the developer has to make a lot of changes to the code in order to use this reloading mechanism.

Sadly, most of the Swing classes in the Java API or external UI libraries are not serializable because they are not meant to be serialized. A preferred use would be when some value of a field in the object tree is not cloneable, then this branch is ignored and the value of the field set to null. This way it can still fulfill the reload request and with a high probability the branch does not have an important state in it.

4.4.2 Objenesis cloner with reflection

Creating a clone of an object in Java can be more complex than for example some interpreted languages. Java is strict with access rights and enforces classes to be initialized with a correct state through constructors. This means that every class initialization must go through a constructor. When a programmer does not specify his/her own no arguments constructor, the compiler will automatically create one and, as a first thing in the constructor, call a super constructor. This makes sure that by the time constructor finishes, its super class is initialized coherently, then the current class is initialized and the final fields are initialized with a value.

When cloning an object, an algorithm that would create a new instance by calling any constructor cannot be used. The reader might think that no arguments constructor would be perfect. The field values from the original object will be copied to the new object by using reflection, but this can have interesting side effects. The no arguments constructor might have some content defined by the programmer that was not meant to be called without consent from the developer. Yet, Java enforces a developer to only create instances through constructors.

There is a way to create an instance of a class without calling any constructors, in return leaving all the fields - including final fields - uninitialized throughout the class's inheritance. This means that it is the developer's responsibility to make sure that the object will be in a correct state before using it. This solution has another problem - the process can be different across different JDK implementations and platforms. To avoid doing the implementation for every platform, a library called Objenesis is used. This works on a variety of platforms and JDK implementations and fulfills a single purpose - creating instances of classes.

In this cloner implementation, an open source library by Kostas Kougios that internally uses Objenesis and reflection to recursively clone objects is used [2]. The library required heavy extension before it could be used for cloning Swing components - a custom filtering for fields and values to ignore and set to null etc.

4.5 Comparing and finding component matches

During the reload a changed code is re-executed that creates the components for the UI. Next, an algorithm has to match previous components to the new components, but the task at hand is not that straightforward. The components may not have any connection to each other. To match components that did not change and those that were edited and then transfer the state over, multiple approaches are implemented which are discussed in more detail in the following subsections.

[2] Library source available from https://code.google.com/p/cloning

4.5.1 Match by type

The most straightforward solution is to match the components in the order that they were added to the UI by comparing the type. This covers most of the cases were components were added, removed or modified, which suits most of the reloads. This way the algorithm knows that for example the component of type X was the second of its type added to the container before the reload. Therefore it must be the second of that type added to the container during the reload.

This method has a potential shortcoming regarding the type - it will not match subtypes nor supertypes. If the number of components, with the same type, does not match, then it can also happen, that because of this, contextually invalid components are matched and thus state injected to the wrong component. It would be better if the matcher understood the context and knew what was removed or added and where.

4.6 Collecting components and additional data

The approach described would delay adding the components to the UI because it has to collect the components, analyse, find state, transfer state and then add the new element to the UI. This raises another problem when mutable objects are used in conjunction with the component.

The `Container` class has several add methods that have additional arguments for the component. The methods are listed in an interface form in Listing 4.3. The problem arises with methods that take component and its constraints for the layout (Object constraints). A common practice is that any constraint object, for example `GridBagConstraints`, is mutated after every add and then reused for multiple components. This works, because the constraints object is consumed immediately during the add and not used afterwards. With the current approach all the objects passed to add methods are collected and then consumed. This means that only the last mutation of the `GridBagConstraints` is used and all the components are therefore placed onto each other.

```
1 public interface Container {
2     Component add(Component comp);
3     Component add(Component comp, int index);
4     void add(Component comp, Object constraints);
5     void add(Component comp, Object constraints, int index);
6     Component add(String name, Component comp);
7 }
```

Listing 4.3: Container add methods in interface form

To overcome the above issue, a simple approach of creating a clone of the constraint object is taken. For this, the same cloner that was used to clone Swing components can be used, but this step is not always necessary. Some of the constraint objects are cloneable and implement the

Cloneable marker interface, therefore at first a check is made whether the constraints object is cloneable and whether it contains the clone method. When suitable, it will be used instead.

4.7 Integration with dynamic class updater

So far, the experiments were conducted by providing a set of subclasses that have some methods overridden and the **Updated method**, described in Figure 2.4, added. While this allows to test out different approaches faster, it is not a solution that simulates a real world usage very well. Then again, it helps to develop a method that is not specific to any dynamic class updater.

For this project, a dynamic class updating software called JRebel was selected. It has a lot of plugins that help to update web applications, therefore the plugin system is well developed and documented. It has a software development kit (SDK), that allows anyone to create their own plugin just by implementing a `org.zeroturnaround.javarebel.Plugin` interface and doing the bytecode modifications where needed [19].

The `Plugin` interface has a number of metadata methods that return the author, the name, the version of the project etc. A more interesting method is the preinit method where the integration takes place. It allows to list classes, the bytecode of which needs to be manipulated during the loadtime with a class bytecode processor (CBP) that uses **Javassist** to change the classes behavior, fields and a lot more. Secondly, it is possible to add class reload listeners that will be triggered when a class has been successfully updated. Previously, this functionality was achieved by adding a **Updated method** method to a class.

The plugin registers the `Container` class with a CBP that injects the **Automatic intercepting** code to intercept the add methods of the `Container` class. Next, it registers a class reload listener that executes the plugin logic only when a `Container` type is changed and replaces the content with new elements.

4.8 Conclusion

This chapter was about implementing the abstract concepts described in chapter 2. We learned that not all complications can be foreseen, like for example deep cloning Swing objects that needed multiple different tries to work and then to work well. The technical details faced were highly educational and taught us to think about a program source code as data which was never experienced before.

The results of the approaches working in collaboration, to enable runtime UI reloading, is further discussed in the following Chapter. There the created software is analysed and a verdict is rendered on how well it managed to fulfill its purpose.

5. Case study and project evaluation

In this chapter a running example is described to showcase the achievements of UiReload plugin.

5.1 UI modification and state preservation

The plugin has to be able to support the basic containment hierarchy functions like adding, removing and modifying nodes. Next, it has to enable to add action listeners to newly created components and propagate any reconfiguration without losing the state. This is validated in the following sections by conducting tests with the **Example application**. All of the tests are completed sequentially, meaning no restart of the application is done when evaluating the process.

5.1.1 Component insertion

One activity often conducted by a developer is adding new elements to a running application. A component can be added virtually anywhere, in respect to others within a container. In the following the success of adding new items onto the UI is discussed.

A developer has started up the application at hand and wants to modify the UI by adding a ZIP code field with a label. The initial state of the application is shown in Figure 5.1. By adding the line of code marked with green in Listing 5.1, one should expect to have the UI as presented in Figure 5.2.

For this, the `AddressPanel` type containing **Intermediate containers** and **Atomic components** with type `JLabel` and `JTextField` is modified. Following the reload, the data in the fields is not lost and the application is still functioning as expected.

```
1 public class AddressPanel extends FormPanel {
2     public AddressPanel() {
3         super(new GridBagLayout());
4         setBorder(new TitledBorder("Address"));
5     }
6
7     @Override
8     public void initComponents() {
9         for (int i = 1; i <= 2; i++) {
```

```
10              addLabeledComponent("Address line " + i,
11                      new JTextField(30));
12          }
13          addLabeledComponent("ZIP code", new JTextField());
14      }
15
16      protected void addLabeledComponent(String label,
17              JComponent component) {
18          add(new JLabel(label), newLeftConstraint());
19          add(component, newRightConstraint());
20      }
21  }
```

Listing 5.1: Address panel with added "ZIP code" field.

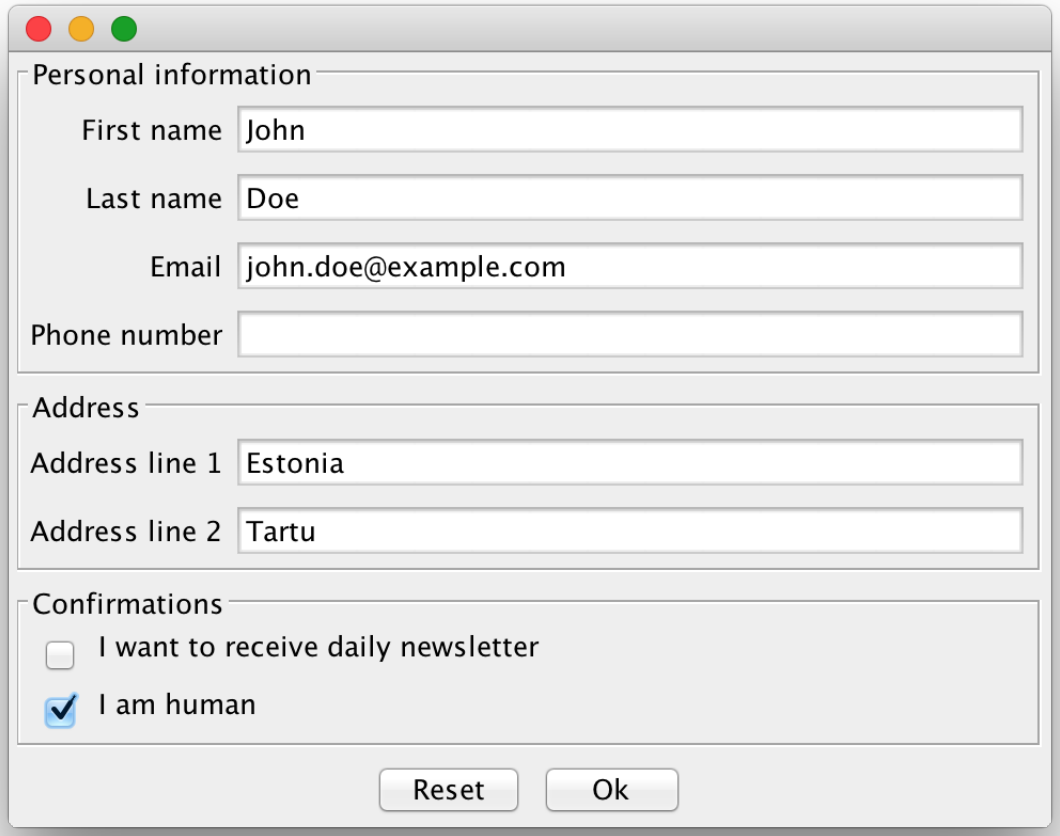

Figure 5.1: Initial GUI of the application.

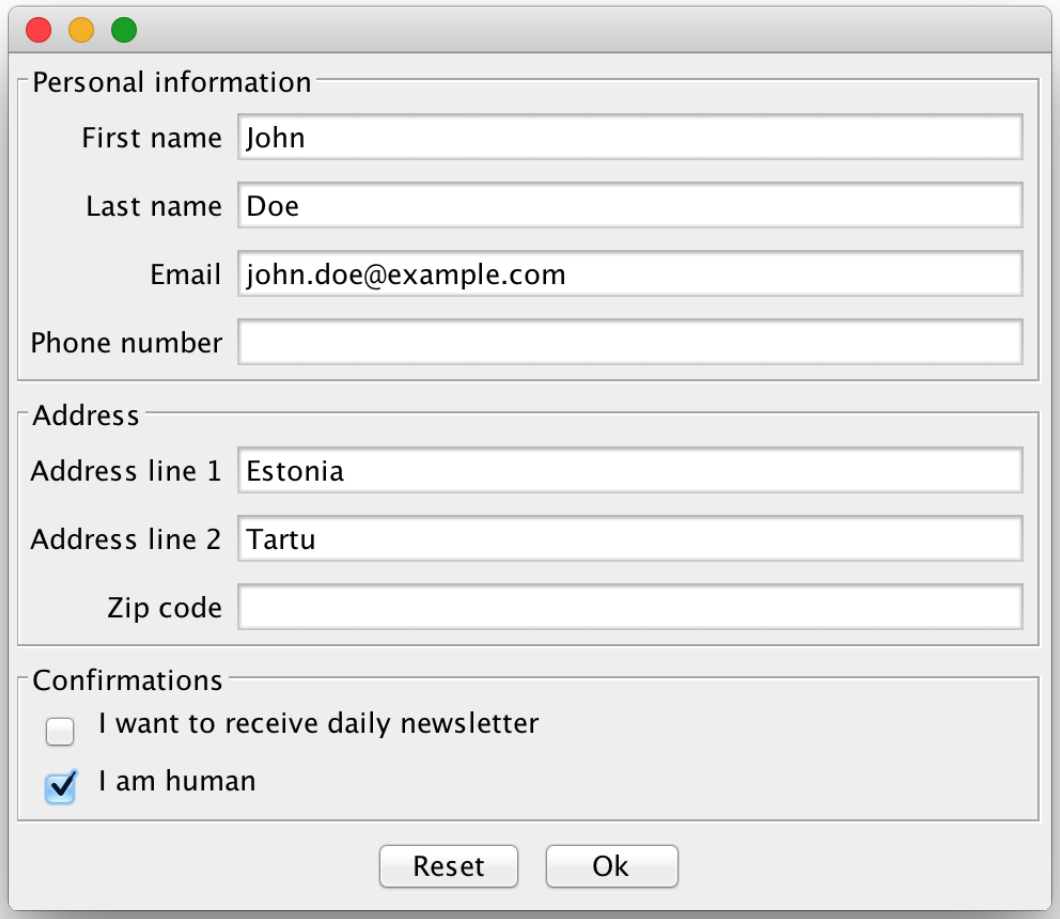

Figure 5.2: The example address panel with "ZIP code" field added during the runtime.

5.1.1.1 Problems

The **Match by type** (see subsection 4.5.1) approach resulted to be insufficient when looking for existing component matches. It misses the mark when a container holds multiple atomic components with the same type X and another component with this type is added in the middle of the existing items. Therefore, the state is transferred to contextually incorrect fields. It would not happen if the added components were with a different type than the existing children.

Figure 5.3 illustrates the effects of this problem. There it can be seen that the value "Tartu" is put into a "Zip code" field which was not the original intention of the developer.

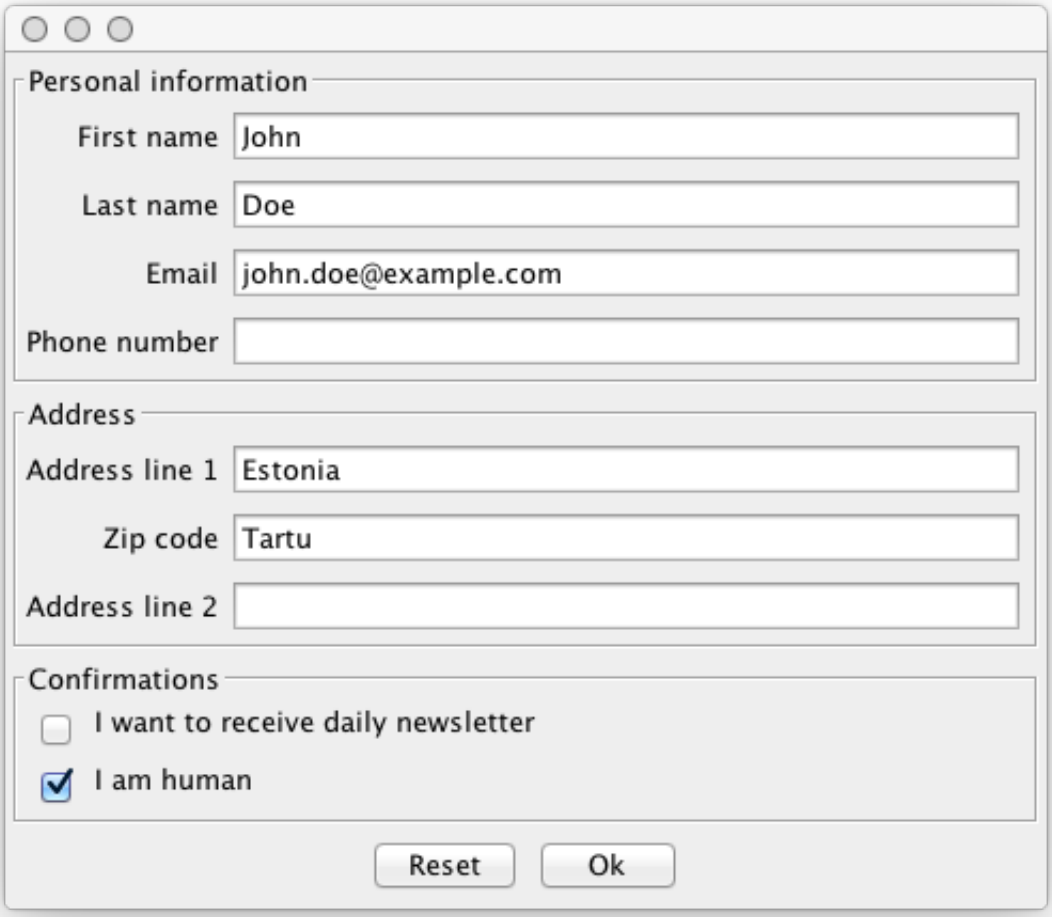

Figure 5.3: Example with contextually invalid state transfer.

5.1.1.2 Discussion

It can be seen from the previous examples that adding components to a running application with the selected approaches works in most cases. Meaning the Section 2.5.4 **InitComponents method** is called by the plugin, the "add" methods are intercepted by the correct interceptors at the correct time and the automatic state detection through deep cloning of Swing components - fulfilling the requirements for this example. What needs improving is the matching of components, where the state candidates are wrongly selected.

5.1.2 Component deletion

Evolving software does not mean only adding new features and components but also removing them. Dismissing can be considered a simpler task to complete at runtime, because it does not

involve state transfer. In reality - with the chosen approaches - there does not exist such a notion as removing a component. It is further discussed on how well the functional requirement is accomplished without directly approaching it.

In Section2.5.1.2 it was described that the **Replacement approach** (see Section 2.5.1.2) can be used instead of finding the exact changes and propagating them. This means that a container is first emptied and the components remaining in source code remaining.

5.1.2.1 Problems

As for component insertion, the **Match by type** also causes problems when removing components. For example: when "Address line 1" was removed from Figure 5.2, the matching algorithm would not know how to match the state to fields. This results in every following field having a contextually invalid state as shown in Figure 5.4. This would not occur if the objects holding the state would be of a different type.

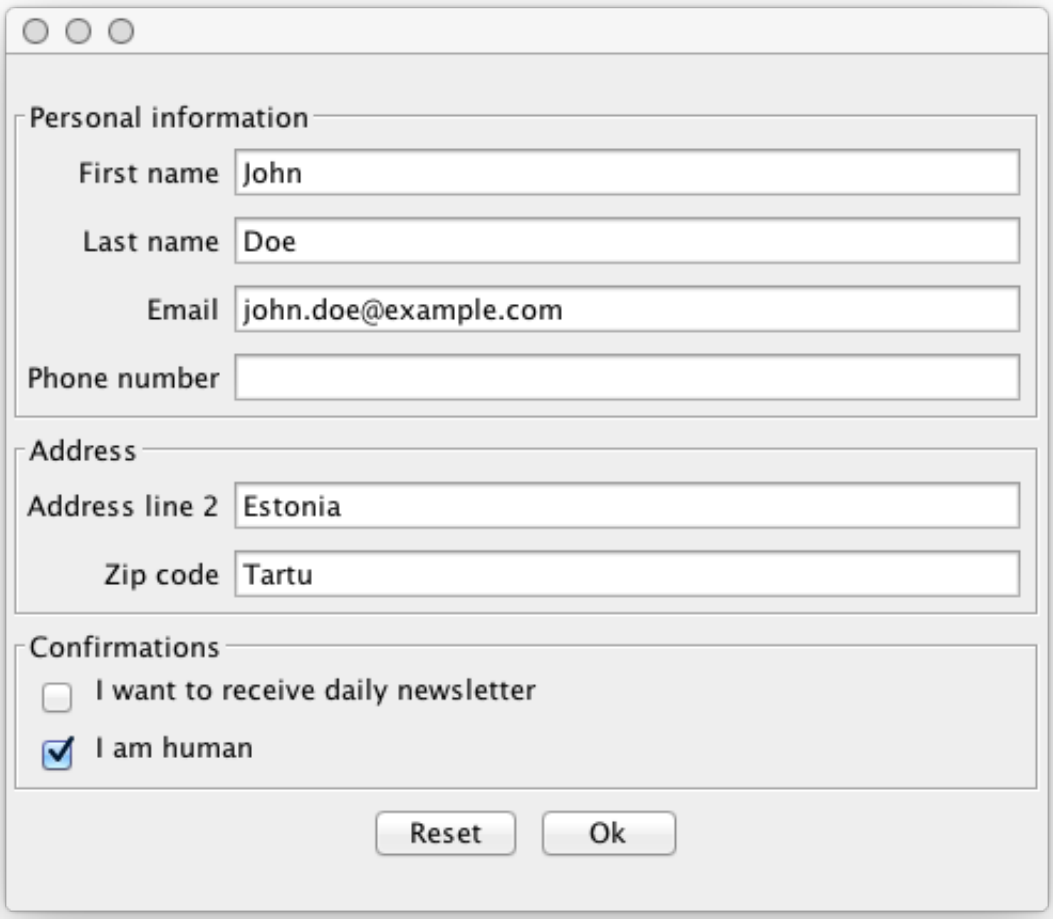

Figure 5.4: Example of removing a field that incorrectly transfers a state.

5.1.2.2 Discussion

Removing a component from the containment hierarchy is a function that comes for free by selecting the **Replacement approach**. While less often required than adding components, deleting of components remains a key activity for runtime software evolution.

However, this task shares the same problems observed, when adding components.

5.1.3 Component modification

Conceptually component modification is equivalent to replacing and existing component and then adding a new one. If the types match, the state is transferred over. If any property of a component is changed, the type stays the same and to a developer the process seems as if it were reconfigured.

A simple example of changing a component is adding a change listener to a text field, that validates the data every time it changes. If the text field contains invalid data, then the background color is set to red, otherwise reset to default. The source code for this is shown in Listing 5.2, the marked lines were added to enable the functionality shown in Figure 5.5.

```
1  public class PersonPanel extends FormPanel {
2      // Beginning of added code
3      private static final Pattern emailPattern = Pattern.compile(
4              "^[_A-Za-z0-9-\\+]+(\\.[_A-Za-z0-9-]+)*@" +
5              "[A-Za-z0-9-]+(\\.[A-Za-z0-9]+)*(\\.[A-Za-z]{2,})$");
6      // Ending of added code
7
8      public PersonPanel() {
9          super(new GridBagLayout());
10         setBorder(new TitledBorder("Personal information"));
11     }
12
13     @Override
14     public void initComponents() {
15         addLabeledComponent("First name", new JTextField(30));
16         addLabeledComponent("Last name", new JTextField(30));
17
18         final JTextField emailField = new JTextField(30);
19
20         // Beginning of added code
21         emailField.getDocument().addDocumentListener(
22                 new DocumentListener() {
23             /--/
```

```
24              @Override
25              public void changedUpdate(DocumentEvent e) {
26                  if ("".equals(emailField.getText()) ||
27                          emailPattern.matcher(
28                              emailField.getText()
29                          ).matches()) {
30                      emailField.setBackground(Color.WHITE);
31                  } else {
32                      emailField.setBackground(Color.RED);
33                  }
34              }
35          });
36          // Ending of added code
37
38          addLabeledComponent("Email", emailField);
39          addLabeledComponent("Phone number", new JTextField(30));
40      }
41  }
```

Listing 5.2: Example of component with email pattern validation.

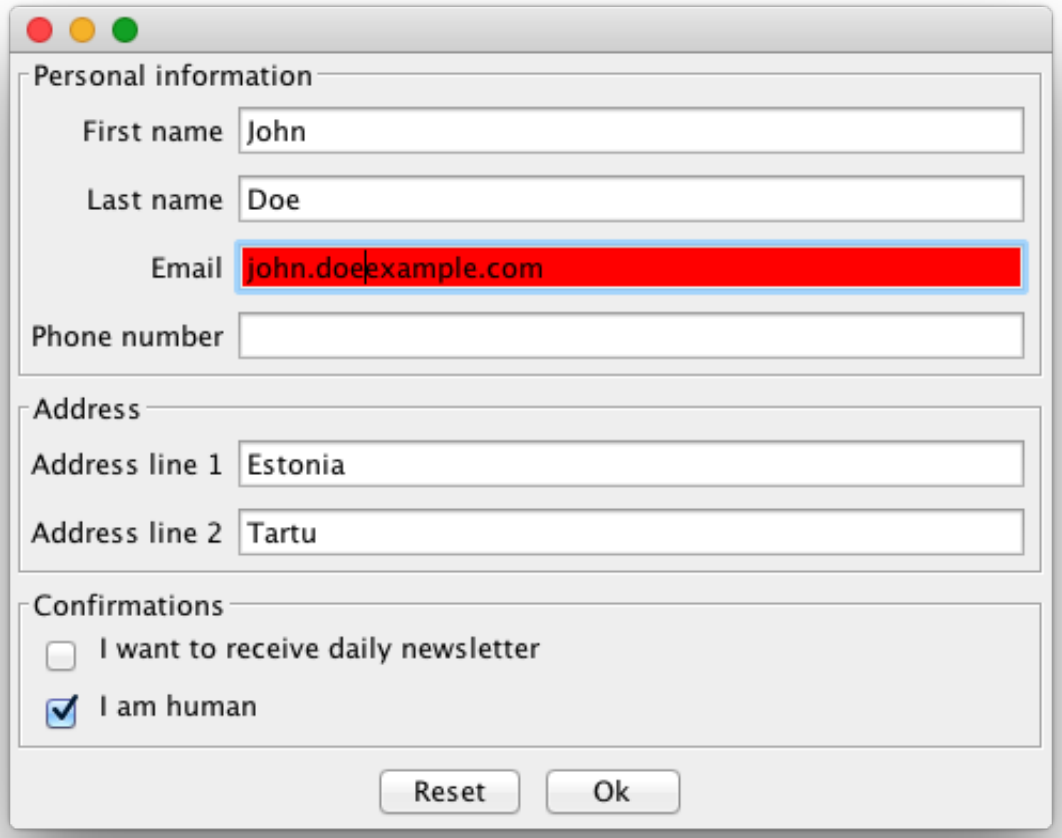

Figure 5.5: Email field with pattern validation.

As can be seen from the example, making any changes to a component is supported. This enables a developer to add components to the UI and afterwards add any additional behavior. This can be considered the change done most often, thus a common development flow is successfully supported.

5.1.3.1 Problems

It was stated that modifying is actually a remove and add working together - to trick the developer into thinking that an existing instance is changed. This indicates that the activity has the same shortcomings regarding the **Match by type** approach. In the real world, this problem does not manifest very often, because the type is usually not changed when modifying a component.

5.1.3.2 Discussion

Modifying an existing component on the UI has proven to be the most stable change. While the same complications can happen, they are usually not that common to occur. This is considered good, because making small changes to existing components is probably the most common action taken by developers.

5.2 Transparency

Although the total transparency requirement was re-thought at the beginning of the project, there are still approaches provided that could achieve it. The current solution provides a relatively high level of transparency by requiring the developer to instantiate and add components to a container in the initComponents method. This is currently considered as the only strict requirement that does not force the developer to know too much about the internals of the project to reap its benefits.

5.3 Reloading time

It is known that restarting applications takes time - usually a few seconds. But for some applications this takes minutes – let's say a constant R units of time. Waiting for the input from the user is the slowest and can be divided into navigation and input filling. In the example application there is no navigation, but in real world it could take longer – let's say N units of time. Filling data to fields can be the slowest because it involves typing or selecting values from drop-down menus – let's say F units of time per required input. It can easily be seen that the necessary activities add up to excessive amount of time.

The test application is small - its startup time takes on average 853ms. There is no navigation, therefore the time is 0 seconds. There are 6 required fields (First name, Last name, Email, Address line 1, Address line 2 and I am human checkbox). This means that the turnaround time is around 0.857 + 0 + 6 * 3 = 18.857 seconds - when using simple restarting of the application.

The validation edit shown in Figure 5.5 took the UiReload plugin 138ms to reload. It is not necessary to navigate even if the example featured navigation. The data in the fields is preserved. This means that the turnaround time using the UiReload plugin is 0.138 + 0 + 6 * 0 = 0.138 seconds.

It is clear that the measurements of this example are not representative of a real world situation. The restart time for the application is small, mainly because the application has no navigation elements and user flow. While it is all true, it can be seen from the mathematical formula that when using the plugin, the only thing contributing to the turnaround time is the reload time - whereas simple restarting accumulates restart time with navigation and data input. This means that the UiReload plugin will always be much faster.

5.4 Evaluation summary

The objectives set for the UiReload plugin were partially achieved. While there is room for improvement to increase the flexibility, it is already seen that the project promises to increase developer performance and convenience. It was seen in the evaluation that the current solution had one major flexibility issue - matching components by type did not prove to be a stable solution. Secondly, the total transparency was not achieved because the developer has to follow the **Following GUI builder patterns**. Both of these deficiencies can be improved by first trying alternative approaches. Still, the current solution has already proven to be good enough to be useful during daily development.

The process of making different types of changes to a running application is shown in the following demo video `http://tiny.cc/ui-reloading-demo`. In the demo the **Example application** is evolved and functionality with the new UI components is added. It demonstrates that the developer can do any changes with no disturbance in the process.

6. Conclusion

Using Java desktop application frameworks should not be as complicated and time consuming as it currently is. In this thesis, I explored how to propagate source code changes to a running Swing application. People learn by exploring and testing, therefore making it possible to see the results of an experiment immediately. This increases developer productivity and experience.

The objective was to decrease the turnaround time and preserve application states across reloads, while providing transparent approach to reloading. Although full transparency was not achieved, because of set rules for the source code, the goal is still considered as reached because only one strict guideline has to be followed. The state is retained in nearly all cases, which in conjunction means that the turnaround time is decreased - there is only one contributing variable - the reload time, compared to restart, navigation and filling out of fields all put together in simple restarting.

The ideas developed in this work, were implemented in UiReload, that is extension plugin for the popular dynamic class updating software named JRebel. Anyone using the plugin can see the impact of their changes in the source code immediately within the UI.

The work was divided into multiple parts that all needed to be solved. It was discovered that it is difficult to collect a changeset of a reload for a stable propagation, thus an alternative approach was used by recreating and replacing component instances. There was no transparent method found that would allow to filter out lines of code that create a specific object in the containment hierarchy. Instead, a popular solution proposed by GUI builders was selected. This decision had important implications on the programming conventions to be imposed to developers. However, there exists a number of projects that already follow them.

Because a component on the UI was replaced with a new, it loses all of its state created by a user. For this, a method was developed for finding this automatically and injecting it into the newly created instance. The approach raised a challenge on how to match a list of original items to a list of newly created instances to receive the state. It was solved by matching the components with the same type. For example text fields are a match to text fields and checkboxes to checkboxes in the order they are placed on the UI. This proved to cause contextually incorrect state transfers when the number of types in the two lists is not equal. A more stable and controlled solution is left for future improvements that understands the context and considers it for matching.

So far, the research in this field has been focused on getting similar results by enabling

dynamic class updating. This is not sufficient for desktop applications which have thus been set aside and the scope has been limited to web applications. This project promises to be the first to bring reloading to Swing applications.

6.1 Future work

The main focus of this project has been to show whether enabling runtime UI reloading for Java desktop applications is feasible. For this, a prototype has been created that supports this claim.

Within the evaluation of the goals, some limitations of the selected approaches are observed. Most importantly, the UiReload plugin should be able to transfer the state to correct components in all cases. To achieve this, an improved version of a matching algorithm should be created that considers not only the order of the items but also the context they are placed into.

Secondly, the transparency of the plugin should be increased. This can be done by providing and analysing more sophisticated approaches for filtering out code that initializes and adds the components to a container.

Thirdly, collect metrics to assess the gains in productivity due to the use of UiReload.

The software created in this thesis does not provide applicable use for a developer unless it is made into a product and accessible to public. It can be improved even further by collecting feedback and data from the testers. It is the hope of the author that all this happens in future work.

Bibliography

[1] TIOBE Software. Tiobe programming community index definition. http://www.tiobe.com/index.php/content/paperinfo/tpci/programminglanguages_definition.html. Accessed February 19, 2015.

[2] Is java (on the desktop) dead? or is the desktop itself dying (except for programmers)? https://weblogs.java.net/blog/editor/archive/2010/09/07/java-desktop-dead-or-desktop-itself-dying-except-programmers. Accessed February 19, 2015.

[3] What's after swing? http://www.dreamincode.net/forums/topic/339323-whats-after-swing-desktop/page__view__findpost__p__1965942. Accessed February 26, 2015.

[4] Jevgeni Kabanov. Zero turnaround in java development. http://jaoo.dk/aarhus-2008/presentation/Zero+Turnaround+in+Java+Development. Accessed February 19, 2015.

[5] Atollic Inc. Improving software quality with static source code analysis. http://www.atollic.com/download/Atollic_Static_Code_Analysis_whitepaper.pdf. Accessed May 7, 2015.

[6] Eric Bodden, Andreas Follner, and Siegfried Rasthofer. Challenges in defining a programming language for provably correct dynamic analyses. In *Proceedings of the 5th International Conference on Leveraging Applications of Formal Methods, Verification and Validation: Technologies for Mastering Change - Volume Part I*, ISoLA'12, pages 4–18, Berlin, Heidelberg, 2012. Springer-Verlag.

[7] M. Jacques J. Malenfant and F.-N. Demers. A tutorial on behavioral reflection and it's implementation. http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/malenfant/malenfant.pdf. Accessed May 8, 2015.

[8] Gísli Hjálmtýsson and Robert Gray. Dynamic c++ classes – a lightweight mechanism to update code in a running program. In *IN PROCEEDINGS OF THE USENIX 1998 ANNUAL TECHNICAL CONFERENCE*, pages 65–76. USENIX Association, 1998.

[9] Gavin M. Bierman, Matthew J. Parkinson, and James Noble. UpgradeJ: Incremental typechecking for class upgrades. Technical Report UCAM-CL-TR-716, University of Cambridge, Computer Laboratory, April 2008.

[10] Allan Raundahl Gregersen, Douglas Simon, and Bo Nørregaard Jørgensen. Towards a dynamic-update-enabled jvm. In *Proceedings of the Workshop on AOP and Meta-Data for Software Evolution*, RAM-SE '09, pages 2:1–2:7, New York, NY, USA, 2009. ACM.

[11] Allan Raundahl Gregersen and Bo Nørregaard Jørgensen. Dynamic update of java applications—balancing change flexibility vs programming transparency. *J. Softw. Maint. Evol.*, 21(2):81–112, March 2009.

[12] Allan Raundahl Gregersen and Bo Nørregaard Jørgensen. Run-time phenomena in dynamic software updating: Causes and effects. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 6–15, New York, NY, USA, 2011. ACM.

[13] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Unrestricted and safe dynamic code evolution for java. *Sci. Comput. Program.*, 78(5):481–498, May 2013.

[14] About page. `http://ssw.jku.at/dcevm`. Accessed February 19, 2015.

[15] Andy Clement. What is spring loaded? `https://github.com/spring-projects/spring-loaded`. Accessed February 19, 2015.

[16] ZeroTurnaround AS. How does it work? `http://zeroturnaround.com/software/jrebel`. Accessed February 19, 2015.

[17] Uncle Bob. The open closed principle. `http://blog.8thlight.com/uncle-bob/2014/05/12/TheOpenClosedPrinciple.html`. Accessed March 17, 2015, Created May 12, 2014.

[18] Shigeru Chiba. Javassist. `http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist`. Accessed May 14, 2015.

[19] ZeroTurnaround. Jrebel plugins. `http://zeroturnaround.com/software/jrebel/learn/jrebel-plugins`. Accessed May 17, 2015.

Non-exclusive licence to reproduce thesis and make thesis public

I, Taivo Käsper (date of birth: 17th of January 1991),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

   Runtime UI Reloading for Java
   supervised by Allan Raundahl Gregersen and Luciano García-Bañuelos

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 21.05.2015