Master’s Thesis

Modularization and Distribution of an Existing Java GUI Application with R-OSGi

by

Ivan Krivulev

17.07.2009

Advisor:
Dr. Oriana Riva
Prof. Gustavo Alonso

Department of Computer Science
Swiss Federal Institute of Technology (ETH), Zurich, Switzerland
ETH-Zentrum, IFW
CH-8092 Zürich
Abstract

Smart phones have been improving constantly over the last decade, and are nowadays becoming universal access point for various kinds of digital infrastructures. The growing popularity, wide user base, sophisticated hardware and rich user experience have resulted in a boom of applications, some of which push the performance of these devices to the limit. These trends combined with the still limited capabilities of smart phones when it comes to sheer computational power and battery life, call for an alternative approach to building dedicated software for these platforms[2]. This approach allows mobile devices to balance the load they are exposed to and enables them to tap into the practically unlimited computational resources behind cloud infrastructures. This work presents a prototype implementation of this approach based on the R-OSGi/AlfredO stack. In this work, an existing monolithic Java Application is modularized and distributed, in order to enable it to flexibly shift the load between the mobile device and the surrounding infrastructure.
Preface

This work was completed in the Spring Semester of 2009 as a Master Thesis prerequisite at the Department of Computer Science (D-INFK) at ETH Zürich.

I would like to express my gratitude to Prof. Dr. Gustavo Alonso for giving me the opportunity to work on this project and to Dr. Oriana Riva for her expert guidance and inexhaustible patience. I would also like to thank Ioana, Jan, Dejan, Ionut and Gabriel for their timely help in solving various practical problems.

In addition, I would like to thank my family and friends for their unwavering support, without which I would have not been able to complete this project.

Thanks a lot!
## Contents

Preface

1 **Introduction**
   1.1 Motivation .................................................. 1
   1.2 Goals and Contribution .................................. 3
   1.3 Outline of the Thesis .................................. 3

2 **Background**
   2.1 Technologies ................................................. 5
      2.1.1 Open Service Gateway Initiative (OSGi) ............. 5
      2.1.2 Equinox, Concierge .................................. 7
      2.1.3 R-OSGi .................................................. 7
      2.1.4 Java Serialization ..................................... 8
   2.2 Design Patterns .............................................. 9
      2.2.1 Observer ................................................ 9
      2.2.2 Model View Controller ............................. 10
      2.2.3 Whiteboard ............................................ 10

3 **Use Case Application**
   3.1 Introduction ................................................. 11
   3.2 User Interface ............................................... 12
   3.3 Software Architecture ..................................... 12
   3.4 Source Code Organization .................................. 13
   3.5 Simplifications ............................................... 14

4 **Approach** ...................................................... 15

5 **Modularization** ................................................ 17
   5.1 Modularization Process Description ..................... 17
   5.2 Process Execution .......................................... 18
      5.2.1 Functionality Identification .......................... 18
      5.2.2 Interface Creation .................................... 20
      5.2.3 Refactoring and Decomposition ..................... 20
   5.3 Modular Architecture ....................................... 20
      5.3.1 Model Manager ......................................... 21
      5.3.2 Application Services ................................ 21
      5.3.3 Visual Components .................................... 22
      5.3.4 Workbench .............................................. 23
# Contents

6 Distribution ............................................. 25
   6.1 Challenges ........................................... 25
   6.2 RemoteObserver Implementation ................. 27
      6.2.1 Bootstrap ......................................... 27
      6.2.2 Remote Observer Pattern ....................... 27
      6.2.3 Distributed Application Services .............. 30
   6.3 Whiteboard Implementation ....................... 30
      6.3.1 Observer Pattern Problems ...................... 30
      6.3.2 Whiteboard Pattern Benefits .................... 31
      6.3.3 Implementation .................................. 31

7 Evaluation .............................................. 35
   7.1 Functional Evaluation ............................... 35
      7.1.1 Mobile Device Client ............................ 35
      7.1.2 SOA Modularized Application Functionality ...... 36
   7.2 Performance Evaluation ............................. 36
      7.2.1 Experimental Setup ............................... 37
      7.2.2 Experimental Results ............................ 37
      7.2.3 Experiment Evaluation ......................... 38

8 Conclusions ............................................. 39
Chapter 1

Introduction

1.1 Motivation

Smart phones, devices that resulted from merging cell phones with personal digital assistants, have been constantly improving over the last decade and recently we have come to think of them as portable "mini" computers. The drivers behind this progress are found in all layers of the technology stack. With semiconductor devices getting smaller and faster, the formerly limited CPU and memory power of the mobile devices have expanded tremendously. Wi-Fi network connectivity and Internet access have become possible, and support for advanced user input/output devices and sensors has been added. On top of the improved hardware, desktop grade operating systems capable of downloading, installing and running productivity applications and managing advanced hardware components provide an excellent platform for developing application software. The above mentioned improvements provide great productivity and a rich user experience, combined with the wide user base, make smart phones very attractive target platform for the latest software applications. Myriads of applications specifically written for them have become available: personal assistants, browsers, games, map applications, etc. that make it possible for smart phones to provide capabilities previously found only on personal computers, such as the ability to send and receive e-mail, edit office documents, listen to music, record videos, buy parking tickets, browse information online, etc.

Figure 1.1: "The latest generation of smart phones"
Despite the big progress, however, the nature of the smart phones’ hardware still limits them when it comes to sheer computational power and prevents them from being able to run resource intensive applications. These limitations are further aggravated by the limited energy reserves of the smart phone (i.e. the power stored in the battery), which means that the more work the phone has to do; the quicker its battery life gets eaten up [1]. This limits the utility of the user and poses serious obstacles for the development of computationally-intensive applications for the mobile device platform.

A possible solution to this problem lies in the network and Internet connectivity of mobile devices, and the wide availability of cloud infrastructures and 3G mobile networks. By reusing the server client paradigm, mobile devices can offload some of their processing tasks to a dedicated server residing in the wide spread network of Wi-Fi capable devices (the cloud) and turn the phone into an interface to the services provided by the cloud. [2, 3].

Other attempts to solve this problem have gone a long way in implementing the technologies required to realize the vision. The main issue to be solved is that majority of solutions are hindered by lack of flexibility in deployment between the client and the server. The server client applications either run entirely on the cloud exposing only a simple thin graphical user interface client (typically incurring large data transfer, high latency and not optimal UI) or on the other extreme, the applications run almost entirely on the mobile device and thus run into the above mentioned hardware limitations, updates. [2] Therefore a new approach is needed that allows dynamically adjusting the load on the server and client side, i.e. flexibility in distributing the components that build the whole application. Although, this solution may face some challenges of its own including network latency and bandwidth limitations, as well as variable data connection speed, it is a promising approach that will be investigated in the rest of the work in the context of a real-world application use case.
1.2 Goals and Contribution

The goal of this master thesis is to enable the flexible distribution and execution of an existing Java GUI application by analyzing, modularizing and finally transparently distributing it. The main focus is on creating a service oriented modular architecture that will enable distribution while preserving interactivity of the application for the user.

The two main goals of this work are the following:

- **Service Oriented Modular Architecture**: The existing monolithic code base will be divided into coarse grained logical units that represent separation of concerns. The new software architecture will simplify development and improve maintainability by enforcing logical module boundaries. Furthermore, it will enable distribution, and provide configurability, extensibility, separation of concerns, division of labor and reuse.

- **Application Distribution**: The distribution should happen in such a manner that flexible transparent and dynamic reconfiguration of the physical locations of the different modules is possible. This will lead to the possibility to support flexible load sharing between the mobile device client application and the server cloud application. Distribution will also bring along its traditional benefits such as availability, fault tolerance, etc.

A secondary, but also important goal, will be the formulation of a general process for modularizing and distributing GUI applications. Different approaches will be explored and the main lessons learned during the project will be summarized in a stepwise approach that could help developers distribute other desktop applications and make them accessible on resource constrained mobile devices.

1.3 Outline of the Thesis

**Chapter 1 Introduction**: presents the motivation behind this work and briefly outlines the main problems.

**Chapter 2 Background**: provides background on the main technology stack used in the proposed solution.

**Chapter 3 Use Case**: describes the monolithic application that will serve as a use case to test the approach and concepts of this work. Both the functionality and the architecture of the application are discussed.

**Chapter 4 Approach**: defines the vision of the solution and the approach used to construct it.

**Chapter 5 Modularization**: discusses the problems that occurred during the modularization phase of the project.

**Chapter 6 Distribution**: discusses the problems that had to be solved during the distribution phase of the project.
Chapter 7 Evaluation: evaluates the solution according to its comparative performance to the original application.

Chapter 8 Conclusion: this final chapter sums up the main points and insights of this thesis and lines out possible future improvements and developments.
Chapter 2

Background

2.1 Technologies

This sections describes the technologies used to modularize and distribute the use case application.

2.1.1 Open Service Gateway Initiative (OSGi)

OSGi is a standard for a module managements system for Java applications. It builds on top of an already existing mechanism, namely jar files and their manifest file, to provide a flexible framework for creating modular software. The central idea behind OSGi is very simple: the source of many of the problems in traditional Java is the global flat class path:

- Conflicting classes
- Lack of explicit dependencies
- Lack of information hiding
- Lack of versioning

That is why OSGi takes a completely different approach to locating byte code at runtime: each module has its own classpath, separate from the classpath of all other modules and provides mechanism in which modules can interact and share classes.

Figure 2.1: "The OSGi Stack"
OSGi is actually a specification for a platform running on top of the Java Virtual Machine, which consists of two parts: the OSGi framework and OSGi standard services. The framework is the core of the OSGi platform; it is the runtime that implements and provides the basic OSGi functionality (modularity, life cycle management, SOA). The standard services define reusable APIs for common tasks such as Logging, Event Admin and Preferences.

The OSGi framework is comprised of three layers, with each layer dependent upon the layer beneath it. In the following paragraphs each of the three conceptual layers is discussed:

**Modularity Layer:** provides packaging and code sharing. It defines the OSGi bundle concept (module), which is simply a JAR file, containing class files and related resources, with some extra meta data in the manifest file. Bundles are typically logical modules that combine to comprise a given application. The extra meta data allows bundles to explicitly declare which contained packages are externally visible and which external packages the bundles depend on, thus extending the normal Java access modifiers (i.e., public, private, and protected). The OSGi framework can thus manage and verify the consistency of bundle dependencies automatically. This process called bundle resolution, ensures consistency among bundles with respect to classes, class version and other constraints.

**Lifecycle Layer:** provides dynamic run time module management and access to the underlying OSGi framework. External to the application, the lifecycle layer defines the bundle lifecycle operations (e.g., install, update, start, stop, and uninstall), which allow the developer to dynamically manage the application (modules can be safely added and removed without restarting the application). Internal to the application, the lifecycle layer provides your bundles with access to their execution context, which provides run time interact with the OSGi framework its facilities. This overall approach to the lifecycle layer is powerful since it allows you to create externally (and remotely) managed applications or completely self-managed applications (or any combination of the two).

**Service Layer:** Finally, the service layer supports a flexible application programming model based on service-oriented computing. The main concepts revolve around the service-oriented publish, find, and bind interaction pattern: service providers publish their services into a service registry, while service clients search the registry to find available services to use.
2.1 Technologies

Nowadays, SOA is largely associated with web services, but OSGi services are local to a single VM, which is why it sometimes referred to as "SOA in a VM"[2]. The OSGi service layer promotes an interface-based development approach, i.e. the separation of interface and implementation. OSGi services are Java interfaces that represent a contract between service providers and service clients. Additionally, the service layer expands the bundle-based dynamism of the lifecycle layer with service-based dynamism, i.e., services can appear or disappear at any time and special tools (Service Trackers, Event Handlers) are provided to deal with this dynamically occurring situations. The result is a very modular and flexible programming model [4].

2.1.2 Equinox, Concierge

Two specific implementations of the OSGi standard used in this project are Equinox and Concierge:

**Equinox:** Equinox is an implementation of the OSGi R4 core framework specification, a set of bundles that implement various optional OSGi services and other infrastructure for running OSGi-based systems. Equinox is supported by the Eclipse Foundation and serves as foundation for the Eclipse framework, which in the end is just a collection of bundles running on top of the OSGi framework.

**Concierge:** is an OSGi R3 framework implementation intended for resource-constrained devices like mobile and embedded systems. It features the framework itself and low level services like start level service and package admin. The overall footprint of Concierge is about 80 kBytes and its architecture is optimized for resource-constrained environments [8].

2.1.3 R-OSGi

Remoting OSGi (R-OSGi) is a distributed middleware platform, which takes service oriented OSGi application across VM boundaries by extending the centralized OSGi standard to support distributed module management. [5]
R-OSGi middleware runs as a bundle inside the OSGi platform. It achieves transparent distribution at service boundaries by dynamic proxy generation. Proxies are created by the local R-OSGi bundle and are generated on the fly using byte code supplied by the service provider. They redirect all method calls to the remote service object and from the client perspective they are indistinguishable from local services.[10]

To achieve transparent distribution a distributed registry is need. Since distributing the service registry is impossible without changing the underlying implementation, R-OSGi provides a complementary service discovery protocol, and builds proxies for remote services which then register with the local registry. Since this a scalability hazard, R-OSGi makes service discovery reactive: bundles register a discovery listener service with a filter string which is notified when a service of the requested type appears. The local R-OSGi bundle then notifies peers about its listener and when some remote peer sees a service of the requested kind, it notifies the listener. [10]

A further pillar of transparent distribution in R-OSGi is the smart way in which the platform exploits the already present paradigm of OSGi programs handling dynamic bundle behavior to map network and remote failures to local hot plug bundle events.[10]

Finally, all code imports from other bundles have to be explicitly declared in the bundle manifest. A possible problem then arises: the interface of an acquired service might contain nonstandard types, which not present in the client’s environment. Type injection is therefore used to resolve possible type inconsistencies and make service proxies self contained.[10]

### 2.1.4 Java Serialization

Serialization is the conversion of an object into a sequence of bits so that it can be shipped to a different VM across the network. When the resulting series of bits is re-read according to the serialization format, it can be used to create an identical clone of the original object.[7]

Java provides automatic serialization which requires that the object be identified as serializable by implementing the java.io.Serializable marking interface. Implementing the interface marks the class as "OK to serialize," and Java then handles serialization internally [6]. For many complex objects (especially object that use a lot of references), however, Serialization is not straightforward. The standard encoding method uses a simple translation of the fields into a byte stream. Primitives as well as each non-transient, non-static object that is referenced by the serialized object and not marked as transient must be also serialized.

In the context of this work, serialization is very important since in distributing an application a major concern is minimizing interaction latency and communication overhead when executing remote services. Therefore, the reference graphs of the objects exchanged between the service provider and the service consumer
have to be carefully trimmed using the available tools: e.g. proper use of transient keyword and java.io.Externalizable interface. These two methods allow for customizing the object state that will be serialized and instantiated between the different application instances.

While Swing components do implement the Serializable interface, they are not portable between different versions of the Java Virtual Machine. As such, a Swing component, or any component which inherits it, may be serialized to an array of bytes, but it is not guaranteed that this storage will be readable on another machine.

2.2 Design Patterns

This section describes several software design patterns that serve as foundation for the existing monolithic application and the modularized distributed version of this work.

2.2.1 Observer

The observer pattern defines a one-to-many dependency between objects so that when one object changes its state, all its dependents are notified. This pattern is at the foundation of the event handling mechanism for GUI events in Java. The Observer pattern also provides the foundation for Java’s component model, the Java Bean. The pattern is based on three simple concepts: events, event producer, and event listener.

![Observer Design Pattern Diagram](image)

Figure 2.4: "The Observer design pattern"

Events are messages that are sent from one object to another. The object that forwards the event "fires" the event, while the object that receives the event "handles" it. An event producer is the class that fires events. It also has a private listener registry that allows it to add and delete event listeners (object that receive events). Event listeners, also known as event consumers, "listen" for an event.[13]
2.2.2 Model View Controller

Model-View-Controller (MVC) is an architectural design pattern, which splits the domain model, presentation and user actions into three separate concerns. In this way business logic is isolated from user interface considerations, which simplifies future modification of any of the entities. The model manages the data of the application, responds to state queries (usually from the view), and executes instructions to modify its state (usually from the controller). The view manages the display of information and user interface elements, while the controller interprets user input and modifies the state according to the application business rules.

![The MVC design pattern](image)

The MVC pattern is also considered a composite design pattern and can be partitioned into its constituent design patterns, namely the Observer, Composite and Strategy patterns.\[12\]

2.2.3 Whiteboard

The Whiteboard design pattern provides the same functionality as the Observer pattern, but goes a long way towards solving several problems, discussed in later chapters, that are particularly relevant in the OSGi context and are .

![The Whiteboard design pattern](image)

In the Whiteboard pattern, a centralized public Observer/Listener registry is used instead of each source implementing a private registry. Listeners then register themselves with this centralized registry, and when an event becomes available, event sources can find them. Once having acquired the reference to all listeners interested in their events, event source can propagate. This pattern simplifies both the source and its listener by delegating the responsibility for tracking to the centralized registry.\[11\]
Chapter 3

Use Case Application

3.1 Introduction

Sweet Home 3D is an open source home interior design application, aimed at users who want to quickly prototype home interior designs. It allows the user to create a two dimensional plan and place floors, walls and furniture items on it. The application provides a 3D rendering of the home and allows for virtual walk inside the newly designed home from both a bird’s eye view and human visitor perspective. Different visual guides help the user in drawing the plan and laying out the furniture.

Walls and room are drawn upon the image of an existing plan. Furniture can then be dragged and dropped onto the plan from a furniture catalog. Each change in the 2D plan is simultaneously updated in the 3D view, to display a realistic rendering of the layout. Furthermore all home modifications done in the home plan are undoable and home elements can be transferred between different home frames. [14]
3.2 User Interface

Figure 3.2: "interaction pattern"

Each Sweet Home 3D frame edits the interior design of a single home and is divided in four resizable panes, with a tool bar at the top, as shown in 3.2:

1. Furniture catalog: This catalog, organized by categories, contains all the furniture and objects that can be added to the home design. The furniture in a certain category can be displayed by clicking on the triangle beside its name. Each furniture item provides a tool-tip with a 2D icon and description.

2. Home furniture table: This list contains the furniture inside a home, where the name, size, positional coordinates and other characteristics are displayed. It may be sorted by clicking on each column title. Furthermore, subsets of the furniture items can be grouped and aligned.

3. Home plan (2D Plan): This pane displays the home as seen from top, upon a grid and surrounded by dimension rulers. Walls, rooms and dimension lines of the home are drawn with the mouse and their colors and textures can be edited. Furniture layout is put upon the plan by dragging and dropping items.

4. Home 3D view: This pane displays your home in 3 dimensions. The home can be seen in this pane either from the top, or from a virtual visitor point of view.

3.3 Software Architecture

The Sweet Home 3D application architecture is based on the MVC pattern discussed earlier. The home model is represented by the Home class, which is an aggregation of other domain items such as furniture pieces, rooms, walls, dimension and labels. All the home items are Java Beans with PropertyChangeSupport installed that fires PropertyChangeEvent every time a property changes.
Above each Home model, the MVC pattern is applied recursively and views and controllers form a tree using the Composite design pattern, with finer grained components branching out from coarser grained components. At the top of the tree is the home frame that is the gui component finally presented to the user. It contains the application toolbar and menu bar and the HomePane. The HomePane is a four way split pane that aggregates the views from the modules described in the previous section. The controllers form a similar tree, in which the HomeController aggregates the functionality of all the user interface modules and additional basic services. A further point worth mentioning is that controllers are instantiated recursively with the root initialized by the main application class and each controller in turn initializes its child controllers in the tree. Views on the other hand are instantiated by the controllers they belong to.

Figure 3.3: "MVC tree of Sweet Home 3D"

As already mentioned above, the model, view and controller entities, are instantiated by the main application class, which also serves as model manager and is responsible for instantiating basic infrastructure shared by the GUI modules: undoable edit support, user preferences, content manager, and home recorder. Further functionality of the main application class is insuring a single instance of the application is running on the system at each moment and support for running the application as a browser applet and loading the application remotely with the Java Network Loading Protocol.

The infrastructure layer of the application is heavily based on standard Java classes and the precise functionality is discussed in later chapters.

3.4 Source Code Organization

The source code of the application is organized in several packages and the package dependency graph looks like this:

![Diagram showing package dependency graph]
3.5 Simplifications

Since an alternative approach to remote execution of the application is attempted in this work, JNLP and Applet execution are left out from further consideration. Furthermore, the focus of this work is enabling distribution of GUI components and therefore the modularization and distribution of other parts of the application are described in less detail in the following chapters.
Chapter 4

Approach

The work presented in this report, is based on a service oriented modularization and distribution model. Before delving into the implementation details we need to consider the different implications of this model that go beyond the traditional meanings of modularity and distribution. First of all it means that the existing use case application has to be made available in the form of flexible service items. All its functionalities should be decomposed and packaged as modular services that can be invoked and distributed using the service oriented architectural approach of R-OSGi. The granularity of these services should be coarse and they should encapsulate whole functional units. The dependencies between services should be semantic in nature and at the application level.

In the envisioned distributed application, the mobile device acquires on-the-fly a standard (Java OSGi) service interface and is able to acquire variable portions of the application depending on its capabilities with the rest of the application running in the cloud infrastructure. This allows the device to avoid the limitations of pre-installing and maintaining software (except for Java runtime and portable OSGi framework) and also to optimize the application performance according to its capabilities. Furthermore the device can interact with any counterpart that speaks the same network protocol and exposes recognizable services.

To achieve the above described vision a twofold approach will be applied in which the application will be first modularized and then distributed.

- **Modularization:** Redesign the existing application by breaking it down into service interfaces (i.e., normal interface-based programming) and clients of those interfaces. The main difference to the existing application will be how to locate interfaces and their implementations. Instead of the traditional Java way of creating objects by passing references as parameters, in the OSGi world, the services will publish themselves in the service registry and the clients will look up available services in the registry. Once the modules exposing the services are installed and started, the application will start and execute like normal, but with a significant advantages, underneath the application, the OSGi framework is providing rigid modularity and consistency checking and dynamic nature give a lot of new opportunities. The exact process and the resulting modular architecture
are described in chapter 5.

- **Distribution:** In the distribution part, the modularized application will be taken across process space boundaries with the help of the distributed OSGi platform. Different challenges related to the remote execution of parts of the application will be solved. These challenges and their solutions are described in Chapter 6.
Chapter 5

Modularization

In chapter 4 the concept of service oriented modularization was discussed. In this chapter the first step of the approach, namely modularization of the use case application, is explained in detail. A design consideration worth mentioning before setting off on the description is that when an existing application is being modularized it is important that minimal changes to the existing code base are done in order to preserve as much as possible the original functionality and only rearrange the macro architecture and the infrastructure connecting the components. The modularization of Sweet Home 3D follows this requirement.

In the rest of this chapter, the process used to modularize the use case application and create the SOA is presented. The process is generic in the sense that it can be applied to other GUI applications and the same concepts can be easily mapped to other GUI applications. After the process is briefly described, the most important steps from applying the process on the use case are discussed and the resulting modules are presented.

Once the existing application was modularized, few changes were made in some modules and a new module was added to accommodate distribution. These modifications are discussed in the next chapter.

5.1 Modularization Process Description

Process:

1. Discover functionalities that will be exposed as services
2. Create service interfaces for the identified functionalities
   (a) Identify objects providing the functionalities
   (b) Group related functionalities into interfaces
   (c) Organize the created interfaces through TODO
3. Identify dependencies graph for all service interfaces
   (a) Refactor again if necessary to avoid cyclic dependencies
(b) Produce Service dependency graph

4. Refactor the original packages
   (a) Separate View and Controller packages into corresponding visual module packages and separate each module’s packages into a separate bundle.
   (b) Create component packages

5. Decompose the existing application
   (a) Remove the connecting infrastructure
   (b) Refactor objects aggregating unrelated functionalities
   (c) Create new bundles for functionalities that have been removed

6. Plug in the OSGi infrastructure:
   (a) Place service initialization from refactored classes into bundle activators
   (b) Expose service objects to the OSGi service registry in the bundle constructors
   (c) Replace interface calls and constructor dependencies with service registry trackers.

5.2 Process Execution

A brief discussion of the most important steps of the process above follows:

5.2.1 Functionality Identification

This step was performed by looking at the features provided by the application at the user level. The functionalities identified include:

- Model Manager functionalities
  - Adding/removing a new home model to the application
  - Saving and deleting models
  - Exporting model as a 3d object

- Furniture Catalogue
  - Exploring furniture catalog
  - Importing new furniture libraries
  - Furniture item preview
  - Adding and deleting items to existing libraries

- Furniture table overview of all pieces of furniture currently in the home:
  - adding and deleting items to catalogue
  - adding items to home model
5.2 Process Execution

- displaying configurable set of properties for each furniture item
- sorting furniture items on any of their properties
- collective editing of a group of furniture items
- furniture alignment

- Editing models in a 2 dimensional plan:
  - Creating walls
  - Creating rooms
  - Creating dimensions
  - Creating labels
  - Editing item textures
  - Inserting furniture from catalog

- 3D View functionalities
  - 3D rendering
  - Change textures of sky and ground
  - Visitor camera view
  - Top camera view
  - Changing different 3D scene properties

After the high level functionalities were identified, the architecture of the application was further examined and several system level functionalities were identified that provide building blocks for the above mentioned functionalities:

- Content Manager: used as interface to the file system and cache file system items such as icons and textures.
- Home recorder: used to record object streams to the local file system.
- Workbench: the main frame where the application user interface is displayed
- Common functionalities
- Editing user preferences
- Undoable edit support
- Extensible help
5.2.2 Interface Creation

After all features were identified, related ones were grouped and the class entities providing these features were figured out by going down through the aggregation parts of the MVC tree structure and finding the classes where the functionalities were implemented. User level functionalities were thus mapped to the objects that provide them. It was then easy to create interfaces for these objects and decouple them from their clients. Some of the objects had existing interfaces that had to be modified slightly to fit the new setup. All the interfaces were then gathered and put into a new bundle that would serve as interface library to all the client modules and the source was modified to use the interfaces instead of the original objects.

5.2.3 Refactoring and Decomposition

Once insertion of the interfaces was completed, refactoring and decomposition could begin. As, described in the use case chapter, the original application was organized according to the MVC architecture. The main class in the application served as both GUI builder and model manager. It was also responsible for assuring a single instance of the application was running and providing the Java Network Loading and applet capability. It further instantiated all the basic application services and supplied them to the visual component. In order to make the application modular and configurable this main class was removed and its functionality (model manager, GUI builder, basic application services) was broken down into separate components. The responsibility of initializing the objects and exposing their services was handed over to the bundle activators of the newly created bundles.

The next step was then to look at the View and Controller packages from the original application. The Home Controller and Home Pane (View) were two entities that aggregated the functionalities and visual components of the furniture catalogue, furniture table, plan, 3D and basic application services modules and exposing them to the main class. These objects were removed from the application and their services were distributed to the actual providers. The View and Controller packages were thus split into four bundles for the four visual components. Visual component construction and discovery of the services that served as constructor parameters were then placed in the bundle activators. Each of the resulting modules then exposed a View and Action Contributor services.

Other bundles that were created include the help component that provides the application help menu and the workbench.

5.3 Modular Architecture

Completion of the steps above produced a new modular service oriented architecture for the application illustrated in the figure below. Descriptions of the more interesting components follow.
5.3 Modular Architecture

5.3.1 Model Manager

The Home Application class serves as model manager for the original application. To fit the new distributed setup it had to be decorated as event publisher and listener. It publishes a separate IHome service for each new home that is added to the application by the user and respectively brings the service down when the home is closed. This class has further and event publisher and event listener. The Event Publisher is connects to every home in the model manager. It listens for property change and collection events from all the items using the already built in Java Beans property change support. The publisher builds remote event by trimming the received event from its source reference and old value and replacing it with source object id. Thus, in the case of property change events only property name, home id, object id and new value are transferred, whereas in collection event, only the home id and newly added home item or the removed object id are transferred. The Event Listener and executor part exposes an interface that allows external entities to publish event on the homes currently in store. When the listener/executor receives an event it looks up the id of the home contained in the event. Once the target home and object are identified and found in the model manager, the event is executed. Distribution, synchronization, object ids and event generation are described in detail in the next chapter.

5.3.2 Application Services

The common application functionalities were collected together for ease of management and start-up. They reside together in application services bundle and are instantiated and exposed before all other components in the applications services bundle.

- Undoable edit support: this is the Java Undoable Edit support in which every modification to the model is posted by the component that modifies. Additional functionality added to the wrapper of this service is the user interface controls for performing the undo and redo actions, and also enabling and disabling these actions. The Action Contributor of this service provides buttons for the application level menu bar, so edits are tracked across all homes, but this service could also be exposed for each separate home.

- Cut, copy, paste, this service builds on top of the Swing components Transfer Handler functionality. Every component that wants to support these actions must provide a Transfer Handler in its Action Contributor that will decide which objects can be transferred and handle the import and export of these objects. This service then only provides the Action Contributor controls that the components can find and use to provide cut copy paste support and controls the enabled state of these controls. The Action Contributor is also used in the application edit menu.

- User preferences: this service supports internationalization by allowing for localized properties look up.
22 Modularization

- Content manager: is the interface to the local file system and provides file browser component as a service to the other modules.

- Home Recorder: is the service that records Home object streams to disk. It provides an Action Contributor with the "Save", "Save as" and "Open" actions for the application file menu.

5.3.3 Visual Components

As it was already mentioned, the original View and Controller packages were split into four modules with each separate visual component into its own module: Furniture Catalogue, Furniture Table, 2D Plan and 3D module. A few reusable controls shared between the components (texture chooser) had to be either displayed as separate service or kept in the original bundles and used as a component library. For simplicity and in order to preserve the granularity of the modularization the second approach was chosen.

The visual components are the ones that contribute to the application graphical user interface. Each of the modules is organized in the following form: a controller responsible for executing user modifications on the model and a View and an Action Contributor part that define the visual module display component and the GUI controls offered to the user. Each bundle activator then observes the addition of new homes and creates an instance of its controller and publishes it as a service. The service gives access to the View and the Action Contributor to interested modules. One such interested module is the Workbench, described below, which aggregate all View and Action Contributor services in the OSGi registry and builds the user interface.

The View is basically a Swing component, which displays useful information to the user. As in the classical Model View Controller architecture the View listens locally for home modification events and updates itself accordingly. The Action Contributor, as its name suggests, provides "actions": these are swing controls that can be placed at different places in the application. It encapsulates all the actions provided by the controller along with the possible ways in which they can be displayed: popup menu items, menu items, toolbar items, transfer handlers and the necessary resources: button icons, button names etc. Another functionality provided by the Action Contributor is the Transfer Handler used by the cut-copy-paste support(CCP Support).

A further point worth mentioning is that both the View and the Action Contributor were designed in order to be able to operate remotely from the controller. Therefore, they are both serializable and provide the capability of setting the controller and Home service after acquisition, although this functionality is not used at this stage.

To connect to the OSGi framework and expose its services, each of the modules is now responsible for acquiring the basic services at start up and spawning a new copy of its controller each time a new home model appears. The building of the screen is therefore no longer part of the modules and had is outsourced to the workbench, discussed later.
5.3.4 Workbench

The workbench bundle is a new module that was created in order to automatically build the GUI of the application by tracking modules that provide GUI elements and allowing for easily plugging these elements in and out. It consists of a Swing frame, with a four way split tabbed pane.

The bundle uses a service tracker, in order to get notified for new model services appearing. When a new one appears a new tab is inserted in the frame. Service trackers are also used to track modules providing GUI services. These services are either the View provider or the Action Contributor provider. It also publishes event for change of focus between homes and inside the tabbed pane.
The screen is actually built by the work bench component which is tracking for graphical user contributions all the time using Service Trackers. This happens every time a new home gets spawned: new model service is exposed by the model manager, then the active graphical component spawns a controller instance which exposes its services, then the workbench is notified about the GUI contributions and incorporates them.
Chapter 6

Distribution

Distributing an application means enabling parts of it to execute remotely in a separate address space. In this work, the objective behind distribution is to allow resource constrained devices to dynamically reconfigure the application at runtime by choosing the components that will be executed locally and have these components (especially the GUI ones) run transparently, unaware of remote application execution.

In particular, the Java GUI application should be able to run on two computers simultaneously with heavy weight modules running on the server machine and a light version of the application must be enabled to run on a mobile device. Furthermore, as with most distributed applications, consistency of the state of the application (model) must be preserved between the distributed instances. Finally, minimization of communication and data transfer overhead between the distributed components is a must in order not to impair performance.

In the rest of this chapter the general idea behind the solution to the distribution problem will be presented and then two particular implementations that build on top of each other will be discussed.

6.1 Challenges

GUI application can be characterized by the close relationships between the different actors in the MVC design pattern, already discussed in previous chapters. Therefore, approaches to distribute GUI applications along the lines of the MVC pattern are bound to meet tough challenges related to data access and manipulation, especially components interacting with modifiable object acquired through a service interface. An example of the latter, would be a component acquiring a reference to the collection of walls inside a home through the Home service and modifying them.

After initial attempts to enable this kind of distribution, a realization was reached that a break down of the applications along the Composite pattern aspect of the MVC pattern would be much more suitable. The solution described below, therefore tries to enable the remote execution of the independent
GUI modules by solving the data access problems.

After approaches for partial data model distribution were tried, complete duplication of the application's model at every partial instance was found most beneficial. This approach would allow application entities running at the same instance to share event notification and would further require no changes in the code of the existing components that resulted from modularization.

Complete application state replication implies that home modification events get propagated to all instances of application. This replication overhead, however, is compensated by the fact that all the components in the same address can share the duplicated model. This minimizes the communication requirements between the client and the server instance as repeated data access calls are served locally, and objects below the service interface can be locally accessed and modified. This approach further solves the problem with application views and controllers based on the observer patterns which access the data model directly and eliminates the need for modifying the existing code. All modifying calls to both the service interface and objects accessible through it are propagated to the other instance.

Another problem that had to be solved was the fact that the form of Home event used in the modularized version of the application is not suitable for remote propagation since they contain references to objects in the address space of the event originator in order to identified the objects which have been modified. If this event was to be shipped remotely over the network, the whole object would be serialized along with any other objects that it references. One towards the solution to this problem is carefully trimming the object by using the transient keyword or by explicitly implementing its serialization with the externalizable interface.

However, this still doesn't uniquely identify the object in the remote address space. Therefore if only the identity of the modified object is of interest, its reference in the event can be replaced with a unique object id. For this purpose a unique object identifier field can be placed inside each object that will be the same on the server and client address spaces. A simple implementation of this idea is a special String id field derived from the objects hash code that is inserted in all objects at instantiation time. When an object gets duplicated in the remote address space its id is preserved and thus the object can be identified, or even better if only the identity is of interest, just the id is shipped instead of the whole object. To support object identification, the application model also had to be slightly modified, by enabling object access and modification using these identifiers and forcing some of its service interface to use the id instead of the reference. The event communication overhead also benefits from this as a string is smaller than the serialized object.
6.2 RemoteObserver Implementation

After a conceptual solution was perceived and all kinds of possible issues were considered, a practical implementation was realized.

6.2.1 Bootstrap

During bootstrap, the client connects to the server by finding the server’s RemoteApplication service in the OSGi registry and notifies it of its existence. The server then also looks up the client’s RemoteApplication service. In this way a kind of two way hand shake protocol is implemented. The client then proceeds by downloading all home currently present in the servers model manager and publishing them as services in its own OSGi framework. This triggers the component initialization events described in the previous chapter.

Now, the client is ready to initiate another two way handshake interaction by exposing its RemoteEventPublisher service remotely and registering its RemoteEventListener with the server’s RemoteEventPublisher (the remote event publisher and handlers are discussed in the next subsection). This forces the server to look up the client’s remote event publisher service and registers its event publisher with them. It is important that the client starts this process after downloading all the homes and instantiating the dependent modules, so that a situation does not occur in which a Home event is published whose Home object is not yet present in the client model manager.

6.2.2 Remote Observer Pattern

The RemoteEventPublisher and RemoteEventListener entities mentioned above are just an implementation of the Observer pattern that uses R-OSGi in order to access remote services.

The RemoteEventPublisher, keeps a reference to all the Home services in the application and registers listeners to all the events that the homes produce. When a home event occurs, the publisher trims it to minimize its size, wraps it in a remote event wrapper and puts in some additional data. The publisher is now ready to ship the event and sends it to the remote observer. When the RemoteListener receives an event, it finds the home that the event is intended for and applies it.

In the event propagation diagram there is no mechanism to prevent from an event propagating infinitely between the two instances as illustrated in the diagram below. Different solutions were considered with event hashing and counting, but the problem could still not be resolved easily. Then the following assumption was made. The home design application is only used by one person at a time and he possibly switches between the two instances if the server application provides GUI. Then only one of the two listener/executors pairs need be active at any time. To use this in the application we have to be aware when the focus of the user switches between the two environments. Fortunately the design of the workbench provides a service to notify component when focus be-
Thus now both the server and the client have an instances of the RemoteEventPublisher and RemoteEventListener pair. As already discussed, only one publisher/listener pair has to be active at any time. To this end, a publisher token exchange mechanism is implemented, in which the workbench module is also involved. When the user becomes active on the client side, the workbench publishes a focus change event. When the client is notified about this event, it steals the token from the remote server application. This triggers the following sequence of events: on the client side, the client can be sure that the server will not be publishing anything, and therefore it enables its remote event publisher and disables its remote event listener. On the server side, the server does not have the publisher token anymore and is therefore forced to disable its publisher and enable its listener.

Java Beans: All model objects are java beans with property change support enabled. When a property change event is fired, it is captured locally by the local event publisher and the local property listeners: view and controllers. Collection events are tracked by custom collection events that are fired by the model when a new room is added to the collection of rooms in a home. The Property Change events from java contain the source of the event (the object that got modified), the name and the new value and the old value of the property. Three type of remote events have been defined, one for each type of application event that occurs: collection, property and selection. Since, remote events have to be sent over the network it is crucial that they contain only the necessary data items and simple objects that have no external references that will cause whole reference graphs to be transferred. To this end, the application model had to be changed, so that each object in the model is identifiable by a String id. Thus, when an event occurs concerning a specific object in the model, only the object can be shipped instead of a reference to the modified objects. This mechanism of course does not function when an object needs to be added to the model. In this case the whole object is shipped.
Figure 6.2: "The RemoteObserver implementation"
A brief description of each type of event follows:

- **Remote Collection Event**: collection events notify the listeners about the addition or deletion of home items to a home model. The contain the home name, the type of the event (ADD or DELETE) and the object to be added or the id of the object to be deleted. This type of event is the largest when it comes to size of the event on the network, as it possibly contains a reference to a home item that will be serialized.

- **Remote Selection Event**: selection events notify listeners of change in the item selection inside a home. It contains the home name and a list of String ids which represent the object currently selected.

- **Remote Property Change Event**: notify the listeners about a JavaBean property change for an item inside a home. They contain the home name, modified object id, property name and the new value of the property.

### 6.2.3 Distributed Application Services

Once, the model manager and graphical components were taken care of the rest of the application had to be distributed as well: Cut Copy Paste support, Undo Support, Content Manager, User Preferences, etc. Some of these are inherently not distributable. For example the content manager is the interface to the filesystem and can therefore not be distributed unless a distributed filesystem was available between the client and the server. Some of the other services, such as cut copy and paste between could be enabled in distributed way, but for simplicity was duplicated and only operates locally.

### 6.3 Whiteboard Implementation

#### 6.3.1 Observer Pattern Problems

The observer pattern is very popular, but it has several problems. Most of these are quite general but they are particularly bad in the OSGi context. The first problem is keeping track of dynamic observable objects. The source of events is a service that has a dynamic life-cycle appear and disappear many times during the life of the application. When a new event source service appears, no listeners are registered with it which forces all listener to open ServiceTracker so they can be notified when an event source service appears.

A second problem is that, just as the event producers can come and go, so can the listeners. Each observable therefore needs to manage a changing set of listeners, and make sure it does not attempt to deliver events to listeners that have disappeared. Since the registering and unregistering of listeners can occur in a different thread from the firing of an event, proper synchronization must be used to avoid concurrency bugs.

This is poses also a memory management problem: the observer pattern is one of the principal causes of memory leaks in Java applications. When a listener is registered with an event source, the internal registry implementation of the observable will typically add that listener to collection field. But now there is a
strong reference to the listener object merely because it exists inside the collection, preventing it from being cleaned up by the garbage collector. Even if the listener is not useful any more in its original context, it will live on in memory until the observer dies. Therefore it is very important to clean up listeners when they are no longer required.

### 6.3.2 Whiteboard Pattern Benefits

As it was already mentioned, the Whiteboard pattern utilizes the OSGi framework’s service registry instead of implementing a private registry as in the listener pattern. In the Whiteboard pattern, event listeners register themselves as a service with the OSGi framework and when the event source has an event object to publish, the event source calls all event listeners in the service registry.

The event source is not registered with the framework as a service. This makes bundles using the Whiteboard pattern significantly smaller and easier to implement. The inter-bundle dependency between the event source and the event listener is handled by the framework and requires almost no code in the event source and event listener bundles.

### 6.3.3 Implementation

In order to implement the whiteboard pattern the EventAdmin service from the standard OSGi services was utilized. This service is further augmented by the R-OSGi bundle with remote event propagation, by the R-OSGi bundle which spreads events occurring on one OSGi platform to peer platform connected to it.

Going from the implementation of the previous section to the whiteboard implementation is very straightforward. The RemoteEventListener and RemoteEventPublisher from the are replaced with the WhiteboardListener and WhiteboardPublisher respectively. The WhiteboardListener’s registered themselves as services in the OSGi registry and await for OSGi Event notifications. The WhiteboardPublisher entity then listens for events on Home models and posts them to the EventAdmin service, which takes care of propagating them to all interested services. The resulting architecture looks like 6.3

The OSGi Event differs slightly from the RemoteEvent of the previous chapter. They are immutable objects with a topic and a properties map containing fields of interest to the consumer. The topic is the logical type of the event, and its primary purpose is to act as a filter to determine which handlers should receive which events. The topic is built of a sequence of tokens separated by slashes, which form a hierarchy. This mechanism allows consumers to filter out at any level of the hierarchy. For this implementation the following topics were utilized: HOME EVENT COLLECTION, HOME EVENT PROPERTY, HOME EVENT SELECTION. The WhiteboardListener actually registers for all three topics simultaneously by using the following shortcut HOME EVENT *

Data describing the actual event appears in the properties, which is a simple
Dictionary object. The properties should in general contain only String and primitive Java types, but in the case of the collection event this rule is violated so that the newly inserted Home items can be passed to the remote model. The properties of the different types of events are similar to the fields of the remote events of the previous chapter, so further explanations are not necessary. The only new thing is the SOURCE property, which can be either "CLIENT" or "SERVER". This field was added so that a distinction could be made between events occurring locally and event occurring remotely. At present, it is not used as in the current version of the application, the EventAdmin is only used to propagate events between the remote model managers, and the communication between the model and the view and controllers is based on the standard Observer pattern.

A final note concerning the Whiteboard pattern implementation concern the way in which events are delivered to consumers. The EventAdmin provides a much more sophisticated event delivery mechanism than the one implemented in the previous chapter. The most significant difference between the two being the possibility for asynchronous event notification provided by the EventAdmin. In the RemotePublisher implementation each listener is called in turn from the event source thread, and therefore the event source cannot continue any other processing until all of the listeners have finished processing the event, one by one.

EventAdmin supports synchronous processing, however it also supports asynchronous processing, which means that the event publishing call will return immediately, allowing the event source to continue with other processing. The events will be delivered to the listeners in one or more threads created by EventAdmin for this purpose. To use asynchronous processing we simply call the postEvent method of EventAdmin rather than sendEvent.
Figure 6.3: "The Whiteboard implementation"
Chapter 7

Evaluation

In this chapter the modularized distributed application will be evaluated both functionally and performancewise. The purpose of the functional evaluation is to make sure that the goal of seamlessly distributing the application while preserving the interactivity for the user has been achieved, whereas the goal of the performance evaluation is to get a feeling for the kind of performance tradeoffs and overhead that have been made by modifying the application architecture and taking it across device boundaries.

7.1 Functional Evaluation

The functional evaluation of the application included testing different runtime configurations. The main criteria for determining whether the distribution process was successful is the ability to reconfigure the load on the client side and preserve the interactivity of the application.

7.1.1 Mobile Device Client

In the beginning of this work the goal was set to be the refactoring of original application for running on top of a mobile device. A known limitation of mobile devices is the restricted Java virtual machine that does not support the full array of Java technologies (i.e. Swing). That is why custom AWT presentation interface had to be developed for the application so that it can be tested on the Nokia N810.
Figure 7.1: "The Nokia N810 custom interface"

AWT interface was developed for the Furniture catalog component and the furniture table. The application was then run with these two modules enabled on the client side and its functionality was tested. The functional tests included normal operation as well as stress scenario in which the application was intentionally stressed. During normal operation the application was found to be responsive. The only time when the responsiveness was compromised was during continued mouse events on the server side during which time a large number of small size property change events were generated. This slowed down the client side a bit. Although not fatal, this problem could be tackled by aggregating mouse drag and drop actions and only publish position updates at a given time interval.

7.1.2 SOA Modularized Application Functionality

Because of the implementation of Concierge used above complied with the R3 OSGi specification, the customized mobile device client was abandoned and the efforts were concentrated on modularization and distribution of the existing code rather than writing new one. Thus the Nokia N810 was replaced with a laptop as a testing device.

Different scenarios were tried in which the application ran with different number of components available to the user. The functionality testing found that the different GUI component configurations had no effect on the application latency and that Swing event queue component update for some of the modules (3D View) was visibly slower than event propagation to the remote application instance. The problem with the drag and drop mouse events generating too many events was also experienced although its effect on performance was as serious as on the mobile device because of the more advanced capabilities of the laptop.

7.2 Performance Evaluation

The performance evaluation aims at determining the overhead of distribution with regard to application event latency. To this purpose an experiment was conducted in which the event delivery latency for the largest home events - the collection event, was measured.
7.2 Performance Evaluation

7.2.1 Experimental Setup

The experimental setup consists of two laptops connected over a Wi-Fi router. This setting is a realistic approximation of the kind of environments in which the distributed application could be used.

In this setup, the client side repeats the same event for a number of times and the propagation time is measured. Because the client and the server run on top of different hardware with different clocks measuring the exact propagation time could be tricky. Therefore, the normal operation of the distributed application was modified and once an event was sent from the client and executed on the server it was propagated back to the client where it was not executed but its arrival time was checked to obtain an estimation the round trip time for the event delivery. This kind of ping type of experimental setup helped gather the data presented below.

7.2.2 Experimental Results

Before presenting the results of the experiment we have to make sure that we know the network latency of the network in which the distributed setup was run. That is why a number of ping tests were executed and the average round trip time for ICMP packets varied between 30 and 55 ms.

Two different implementations of the distributed application were tested: the RemoteObserver implementation with synchronous message delivery and the Whiteboard pattern implementation with asynchronous event delivery. The following data was obtained:
### 7.2.3 Experiment Evaluation

The average round trip time measurements for both implementations are very similar and comparable to the ping rtt. This is not very surprising since in both implementations the messages are shipped remotely by R-OSGi and this probably is the main contribution to the RTT. Event execution and local propagation happen instantaneously and were below the measurement threshold of the standard Java tools.

It is also important to compare this event latency to other distributed GUI applications such as PC games for example. The specifications for running such applications usually state that event propagation should take no more 200ms (Quake). The results obtained above are clearly in this range and therefore the application can be considered interactive over the network.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>asynchr</td>
<td>synchr</td>
</tr>
<tr>
<td>ev1</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>ev2</td>
<td>31</td>
<td>47</td>
</tr>
<tr>
<td>ev3</td>
<td>62</td>
<td>31</td>
</tr>
<tr>
<td>ev4</td>
<td>32</td>
<td>78</td>
</tr>
<tr>
<td>ev5</td>
<td>31</td>
<td>47</td>
</tr>
<tr>
<td>ev6</td>
<td>109</td>
<td>79</td>
</tr>
<tr>
<td>ev7</td>
<td>94</td>
<td>188</td>
</tr>
<tr>
<td>ev8</td>
<td>31</td>
<td>62</td>
</tr>
<tr>
<td>ev9</td>
<td>188</td>
<td>63</td>
</tr>
<tr>
<td>ev10</td>
<td>31</td>
<td>47</td>
</tr>
<tr>
<td>ev11</td>
<td>31</td>
<td>46</td>
</tr>
<tr>
<td>ev12</td>
<td>31</td>
<td>125</td>
</tr>
<tr>
<td>ev13</td>
<td>15</td>
<td>47</td>
</tr>
<tr>
<td>ev14</td>
<td>31</td>
<td>47</td>
</tr>
<tr>
<td>ev15</td>
<td>47</td>
<td>125</td>
</tr>
<tr>
<td>ev16</td>
<td>31</td>
<td>47</td>
</tr>
<tr>
<td>ev17</td>
<td>16</td>
<td>47</td>
</tr>
<tr>
<td>ev18</td>
<td>62</td>
<td>47</td>
</tr>
<tr>
<td>ev19</td>
<td>31</td>
<td>203</td>
</tr>
<tr>
<td>ev20</td>
<td>31</td>
<td>16</td>
</tr>
<tr>
<td>ev21</td>
<td>266</td>
<td>16</td>
</tr>
<tr>
<td>ev22</td>
<td>32</td>
<td>125</td>
</tr>
<tr>
<td>ev23</td>
<td>234</td>
<td>47</td>
</tr>
<tr>
<td>ev24</td>
<td>46</td>
<td>78</td>
</tr>
<tr>
<td>ev25</td>
<td>31</td>
<td>125</td>
</tr>
<tr>
<td>ev26</td>
<td>188</td>
<td>47</td>
</tr>
<tr>
<td>ev27</td>
<td>62</td>
<td>31</td>
</tr>
<tr>
<td>ev28</td>
<td>31</td>
<td>47</td>
</tr>
<tr>
<td>ev29</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>avg</td>
<td>66</td>
<td>68.51724</td>
</tr>
<tr>
<td>stdev</td>
<td>67</td>
<td>46.75668</td>
</tr>
</tbody>
</table>

Table 7.1: "RTT event delivery results"
Chapter 8

Conclusions

In this thesis an existing Java GUI application was modularized and distributed. The use case application demonstrated that the approach of using R-OSGi to make resource intensive applications available on the mobile device platform can be applied to productive real life applications. The modularization and distribution were achieved by a streamlined process of refactoring the existing code base, partitioning it into modules and restructuring the functionalities in a way that allows different modules to run remotely from each other. Furthermore, the latency of event delivery for the distributed components was evaluated and shown to be within the allowable limits.

The most serious problem faced during the work was allowing distributed copies to access modifiable objects exposed by remote service interfaces. The solution to this problem was found in complete application state replication, which bring a number of additional benefits to the application, such as minimized latency for local data access, preserving the most of the original application by allowing local listeners and sharing of data updates between components. A number of other problems were encountered such as the inflexibility of the Swing GUI frameworks, which make it difficult to quickly plug components in and out and consume a lion’s share of the development time.
Bibliography


[12] G. Krassner, S. Pope, A cookbook for using the model-view controller user interface paradigm in Smalltalk-80
