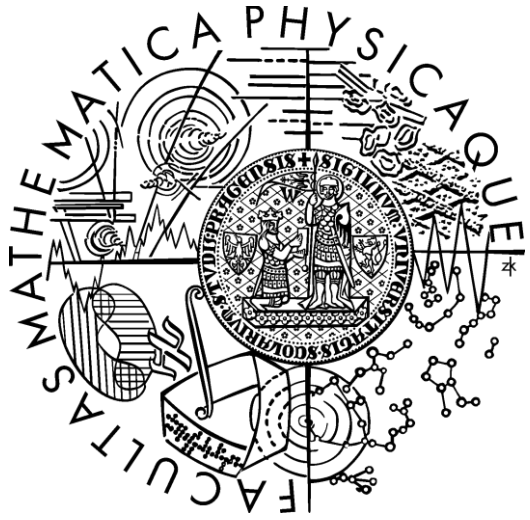
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jaroslav Pastorek

Deployment of SOFA 2 applications for embedded environment

Department of Software Engineering
Supervisor: RNDr. Petr Hnětynka, Ph.D.
Study program: Informatics, Software systems

I would like to thank to my supervisor, RNDr. Petr Hnětynka Ph.D., for his advices and the time he spent during our consultations. I would also like to thank my family and girlfriend for their support not only during writing this thesis.

I proclaim that I have written this thesis all on my own and used exclusively the cited resources. I agree with public availability of this thesis.

In Prague on August 5, 2010 Jaroslav Pastorek

Název práce: Nasazení SOFA 2 aplikací v prostředí vestavěných systémů
Autor: Jaroslav Pastorek
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Petr Hnětynka, Ph.D.
e-mail vedoucího: hnetynka@d3s.mff.cuni.cz
Abstrakt:

Vývoj aplikácií pre vstavané zariadenia je náročnou úlohou hlavne vďaka rôznorodosti použitého hardwaru. Technológie ako Java ME sa snažia poskytnúť jednotný programový model v duchu hesla „write once – run anywhere“, avšak špecifiká jednotlivých platforiem naďalej pretrvávajú. Aplikácie by preto mohli profitovať z využitia komponentovo orientovaného vývoja kedy by platformovo špecifické časti boli oddelené do dobre definovaných a ľahko vymeniteľných komponent.

Cieľom práce je analyzovať aktuálny proces nasadzovania komponentových aplikácií napísaných pomocou komponentového systému SOFA 2 a navrhnúť také zmeny, ktoré by umožnili nasadiť komponentové aplikácie v prostredí Java ME, konkrétne s konfiguráciou CLDC a profilom MIDP.

Navrhnuté riešenie je založené na transformácii SOFA 2 komponentovej aplikácie na MIDlet. Táto transformácia zahŕňa predgenerovanie kódu na statickú inštanciáciu komponent, ktorá je normálne vykonávaná dynamicky podľa popisu jednotlivých komponent. Výsledkom transformácie je samostatný MIDlet package, ktorý obsahuje všetok potrebný kód – to znamená prispôsobený runtime pre komponenty a komponenty samotné. Proces vývoja SOFA aplikácií je modifikovaný tak, aby podporoval nový proces nasadzovania.

Klíčová slova: komponentové systémy, Java ME, vestavěné systémy, nasazení

Title: Deployment of SOFA 2 applications for embedded environment
Author: Jaroslav Pastorek
Department: Department of Software Engineering
Supervisor: RNDr. Petr Hnětynka, Ph.D.
Supervisor's e-mail: hnetynka@d3s.mff.cuni.cz
Abstract:

Development of applications for embedded devices is a daunting task particularly due to the diversity of used hardware. Technologies like Java ME attempt to provide unified programming model in the spirit of slogan "write once – run anywhere"; however the platform specifics still linger. Applications for embedded devices could therefore benefit from the use of component – based development where platform – specific parts can be separated into well – defined easily replaceable components.

The goal of this thesis is to analyze the current deployment process for the component applications written using SOFA 2 component system and propose changes that would allow such applications to be deployed in Java ME environment, particularly CLDC configuration with MIDP profile.

The proposed solution is based on transformation of SOFA 2 component application into MIDlet application. This transformation includes pregeneration of code for static instantiation of components which is normally done dynamically by interpreting component descriptions. The result of the transformation is standalone MIDlet package that contains all necessary code - this includes adjusted component runtime and components themselves. The development process of SOFA 2 applications is also adjusted to support new deployment process.

Keywords: component systems, Java ME, embedded systems, deployment

Table of contents

List of figures

1. Introduction

Recent years have witnessed increasing number of devices driven by some kind of processing unit – smart phones, PDAs, set top boxes, modems or other embedded systems. These devices differ vastly in the terms of their purpose, processing power and other capabilities and the development of applications for this universe of devices became a daunting task. Especially poor portability of applications written in native languages like C/C++, low-level programming and lack of abstraction have raised the need for common programming model which would hide the platform specifics and yet be powerful and extensible enough to allow implementing various kinds of applications for embedded systems. However full-featured multi-platform solutions like Java Platform, Standard Edition - Java SE [**1**] bring too much overhead and the limited configuration of embedded devices is not capable of running such runtimes. Therefore attempts to solve this issue resulted in development of scaled-down versions of afore mentioned technology, particularly Java Platform, Micro Edition (Java ME) [**2**].

Java ME found its place especially in many mobile phones and it has become the platform of choice for developing mobile applications. This success can be credited to the versatility it provides – the runtime is divided into several modules (so-called configurations and profiles) and the mobile phone vendor can choose which module he will implement and provide in his device. In mobile phones the most common configuration is Connected Limited Device Configuration (CLDC) [**3**] and profile Mobile Information Device Profile (MIDP) [**4**].

Although technologies like Java ME are a great step forward in the process of development for embedded devices, applications built using them are monolithic and it is hard to overcome platform-specific differences that still linger even when using afore mentioned frameworks. In monolithic applications the need for platform-specific adjustment may introduce new bugs into the system, especially in the case when the application lacks proper architectural design, the platform-specific code is not properly encapsulated and is scattered over the whole application.

The applications for embedded devices could therefore benefit from the use of component technology, where the platform-specific parts can be moved into well-defined, easily replaceable components. The final assembling of the application for specific device would then require just defining which components implement particular functionality of the application and no (or minor) changes in the code implementing each component would be needed.

Component systems paradigm exploits the idea of software built from building blocks – components – similar to building machinery or electronics from prefabricated parts. There are many possible characterizations of components. According to [**5**], component can be viewed as a black-box entity with well-defined interfaces and behavior. Interfaces are endpoints for communication between multiple components and define the functionality the component provides or requires. The black-box nature of the component ensures the component can be used by third-parties when building larger applications by composing multiple components without any knowledge of its internal structure and implementation. This grants the

component systems one the most important features - the reusability of components. However, during the development of the component itself the internal structure of component is a subject of design, especially in component systems that allow hierarchical nesting of the components. Component also serves as an encapsulation of related functionality and enforces proper separation of concerns. During deployment of the application, the component can be also viewed as a unit of deployment.

Component system is actually an implementation of component model. Component model is a set of abstractions that defines semantics and rules that apply when the components are composed into larger ensembles. Component models can be flat, where all components are primitive, or hierarchical, in which nesting of components is allowed and components can be built by composing their subcomponents. Some component models also allow dynamic reconfiguration of components and their bindings.

As a result, component systems enforce proper architecture design, help to raise the level of abstraction when designing complex systems and make the development of complex systems easier.

Nowadays several component models and component systems exist, both academic and industrial. Industrial component systems usually provide greater stability but also offer less sophisticated features - most of them implement just flat component models. The examples of industrial component systems are JavaBeans [**6**] designed for easier design of user interface, COM/DCOM [**7**] used for ensuring interoperability between different languages, Enterprise Java Beans Technology [**8**] or CCM [**9**].

Academic component systems provide complex features, on the other hand their runtime implementation is experimental or they provide just component model without any reification at all. Academic component systems include for example Fractal [**10**], SOFA [**11**] or SOFA 2 [**12**]. SOFA 2 provides advanced component features like hierarchical components, dynamic reconfiguration of components, aspects that can be applied to components when the application is composed or behavioral checking. Currently the implementations for Java and C++ are available.

1.1. Goal of the thesis

Goal of this thesis is to modify the Java implementation of SOFA 2 component model in a way it will allow the SOFA 2 applications to be deployed in Java ME environment, particularly mobile phones.

This includes identifying features of SOFA 2 component system that cannot be implemented in Java ME and proposing restrictions on these features, identification of Java ME specifics that have to be accounted when developing SOFA 2 application for Java ME environment, proposing any needed changes to SOFA 2 component model, creating the Java ME runtime for SOFA 2 components and modifying the development and deployment process of SOFA 2 applications. Also tooling support has to be included.

1.2. Structure of the text

Chapter one - Introduction - contains basic description of component systems and technologies for embedded systems. It also proposes the goals of the thesis. Chapter two - Background - contains basic description of principles of SOFA 2 component system and Java ME as these are the foundations of following work. It also analyses the requirements and restrictions posed by Java ME that applies to the SOFA 2 applications. In chapter three - Java ME deployment for SOFA 2 applications – the foundations described in chapter two are further elaborated and the solution for the process of deploying the SOFA 2 applications in Java ME environment is proposed. Chapter four - Sample application – describes the design and implementation of sample component application for Java ME environment. The sample application is Worm game. Chapter five - Related work - describes other related component technologies that are focused on embedded devices and provides comparison with results of this thesis. In chapter six - Conclusion – the goals of the thesis are revised and the results of this work are summarized.

2. Background

2.1. Java Platform, Micro Edition (Java ME)

Java Platform, Micro Edition [**2**], or Java ME, is scaled-down edition of common Java Platform designed especially for embedded devices with restricted computing power. First introduced in 1999 nowadays it is widely spread over many different kinds of devices, in particular mobile phones. It is developed under Java Community Process [**13**] to ensure consensus among all community members. To cover as much different platforms as possible its specification is divided into several parts that define the capabilities of underlying hardware, specify the basic common runtime, device-specific features and other optional functionality. This structure is primarily represented by configurations, profiles and optional packages.

2.1.1. Configurations

The main purpose of configuration is to divide different classes of embedded devices into categories on the basis of their capabilities and purpose. On the other hand it also unifies the diversity of devices in the same class and defines common runtime environment. Configuration defines subset of Java language that device has to support, describes the capabilities of Java virtual machine, defines the security model and basic libraries and APIs that will be available to the applications. The basic minimum hardware specification, such as minimum memory the device has to provide, is also the part of the configuration specification.

Currently two configurations are available:

Connected Limited Device Configuration (CLDC) [**3**] – used in small devices with very low processor speed and memory and simple or no user interface at all. It can be found in small, usually mobile devices like mobile phones, pagers or PDAs.

Connected Device Configuration (CDC) [**14**] – used in more capable devices like televisions, set-top boxes or communicators. In comparison to CLDC it provides richer API and more features.

Both CLDC and CDC configurations provide subsets of Java SE libraries and both define their own specific APIs. CLDC is upwards compatible with CDC.

For the purpose of this thesis, the CLDC profile has been chosen as the target platform as it is widely spread among mobile phones. Therefore detailed description of CLDC can be found in chapter 2.1.4.

2.1.2. Profiles

Profiles are defined on top of configurations and they define additional sets of libraries and features that are specific for particular device category, the market segment the device targets or industry branch. Profiles provide some kind of categorization orthogonal to the one of configurations. Profiles help to distinguish between different usages of the various devices. For example the washing machine and mobile phone can share the same configuration (probably CLDC), but there is no reason the washing machine should be capable of running games as is possible on mobile phone. The profile always targets specific configuration, either CLDC or CDC.

There are several profiles available:

Mobile Information Device Profile (MIDP) [**4**] – based on top of CLDC, used in devices with wireless network connection, small displays and limited user input. Therefore it is used particularly in mobile phones and pagers.

Information Module Profile (IMP) [**15**] – also based on top of CLDC, used in networked embedded devices with very limited or no user interface. It can be found for example in modems, parking machines or alarm systems.

Foundation Profile [**16**] – based on CDC, extends the CDC with additional classes from Java SE. It does not provide any user interface support.

Personal Basis Profile [**17**] – based on CDC, adds components for building lightweight user interface.

Personal Profile [**18**] – based on CDC, extends Personal Basis Profile with heavyweight user interface components and support for applets.

For the purpose of this thesis the MIDP has been chosen as the target implementation profile for the same reasons as the target configuration – it is widely spread among the mobile phones. Therefore detailed description of MIDP can be found in chapter 2.1.5.

2.1.3. Optional APIs

Configurations together with profiles form basic runtime for Java ME applications. However the device may provide specific features that are not covered by the API of the particular configuration or profile. Therefore several optional APIs have been designed to address such cases, for example:

JAVA APIs for Bluetooth [**19**] – enables Java applications to access Bluetooth interface on the device

Mobile 3D Graphics [**20**] – adds basic support for 3D graphics

Wireless Messaging API [**21**] – introduces API for handling Short Message Services (SMS)

In addition to these APIs the device vendor may provide additional libraries, however these are specific and may be unsupported on devices of other vendors, which decreases the portability of the application.

All mentioned parts - configurations, profiles and optional APIs – together form stack on top of which the Java ME application resides. Example of one possible stack in depicted on Figure 1. The stack on figure is based on CLDC configuration and MIDP profile. For purpose of this thesis it is assumed that target device will provide stack similar to the one on figure.

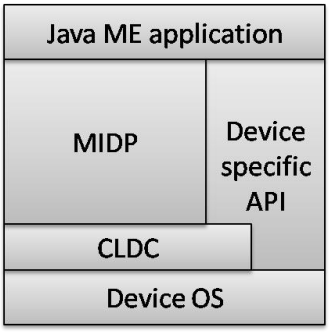


Figure 1: Example of Java ME stack with CLDC configuration and MIDP profile.

2.1.4. Connected, Limited Device Configuration

As mentioned in previous chapter, in this thesis we assume that the target device provides CLDC [**3**] configuration. This configuration must be capable of running applications written in Java language version 1.3 [**22**]. This means that many convenient features from higher versions cannot be used; this includes mainly generics, annotations, for-each loops, autoboxing/unboxing, enumerations and exception chaining.

The virtual machine for CLDC configuration has to capable of loading classes with version numbers 45.* -48.* (this means classes produced by JDK 1.1 – JDK 1.4). There is no possibility to create user defined class loader, runtime provides just bootstrap class loader and this cannot be overridden. This is mainly due to security reasons – CLDC has to implement sandbox security model – each Java application runs in its own environment and it cannot escape it or interfere with other applications. The application has to use the APIs provided by the Java ME stack (configuration, profile and optional APIs) and it cannot load its own. In comparison to Java SE the CLDC virtual machine lacks object finalization, thread groups and daemon threads.

The class verification process requires too much memory and is too complex to be done completely when loading class into the virtual machine. Therefore preverification has to be done before deploying the application to the target device. The preverification tool generates the preverified classes and the verification process on the device itself can be therefore much simpler.

The classes and API the CLDC provides are either derived from Java SE version 1.4 or CLDC specific. In comparison to Java SE, CLDC lacks support most notably for Remote

Method Invocation (RMI) [**23**] and reflection. It also does not have as rich exception set as Java SE.

There are two version of CLDC – 1.0 and 1.1. Version 1.1 adds support for floating point, thread names, weak references and also requires more memory. For the purpose of this thesis the CLDC version 1.1 has been chosen as the primary target configuration because it presumes more powerful hardware.

2.1.5. Mobile Information Device Profile

MIDP profile [**4**] is based on the CLDC configuration. It defines application model and set of additional API related mainly to displaying user interface, handling user input, security, network connection and application packaging and deployment to target device. MIDP also specifies additional requirements on underlying hardware.

MIDP applications are called MIDlets. Entry point to MIDlet application is class derived from class `javax.microedtion.midlet.MIDlet` – it will be called main MIDlet class in the following text. This base class is used to retrieve representation of user screen for displaying user interface and provides methods for checking MIDlet's permissions, properties, launching web browser but most notably for managing MIDlet lifecycle. As the MIDlet is supposed to run on mobile phones it is important that its execution can be suspended in case of some outer event occurs – for example phone call or SMS is incoming. The execution of the MIDlet should not prevent the user from being able to respond to these events. Therefore the MIDlet application passes through several states during its execution. The states transitions are managed by special software on the target device called Application Management Software (AMS). This AMS determines in which of the following states the application should be in particular point in time:

Stopped – application is not running. Application resides in this state right after the creation of the MIDlet. In the constructor of the main MIDlet class just the basic initialization should be done and all expensive operations should be postponed. Application can also enter this state when AMS decides that application should be paused (due to some external events – for example incoming call, the need to free resources), in this case the AMS calls method `pauseApp()` of the main MIDlet class. In this method the MIDlet should free any resources it owns (such as threads, network connections) and after it completes the MIDlet is moved to "Stopped" state.

Active – application is currently running. The MIDlet enters this state when the AMS calls method `startApp()` of the main MIDlet class of stopped MIDlet. At this point the application should fully initialize itself, however it can tell the AMS that it is not ready to start yet by throwing `MIDletStateChangeException`. In this case, the AMS moves the application back to the "Stopped" state and the `startApp()` method will be called later.

Destroyed – application is ready to be disposed. This state is entered when the AMS determines the application should terminate. This transition can be initiated from states "Stopped" or "Active" and it is the final state. The AMS moves the application to this state right after the calls to method `destroyApp(boolean unconditional)` successfully

returns. The parameter of the method indicates whether the MIDlet application can refuse to destroy itself. If its value is equal to false, the application may throw `MIDletStateChangeException` from the method and the AMS then can (but does not have to) postpone the destroying process and call the method again later. The transition to this state is also fired when any of the calls to `startApp()` or `pauseApp()` throws `RuntimeException` (or its subclass). When `RuntimeException` occurs in call to `destroyApp(boolean unconditional)` the state is entered immediately.

The application can also tell the AMS that it is ready to be moved to another state. This is done by calling methods `notifyPaused()` for requesting pausing the application and `notifyDestroyed()` for requesting destroying the application. When AMS changes the state of the MIDlet after one of these methods has been called, the methods `pauseApp()` and `destroyApp(boolean unconditional)` are not called. The application can also request to start itself if it is currently in "Stopped" state. This can be done by calling method `resumeRequest()` on the main MIDlet class from some kind of callback.

The state transitions are summarized in Figure 2. The methods called by AMS are prefixed with "AMS::" and those called by application are prefixed by "APP::".

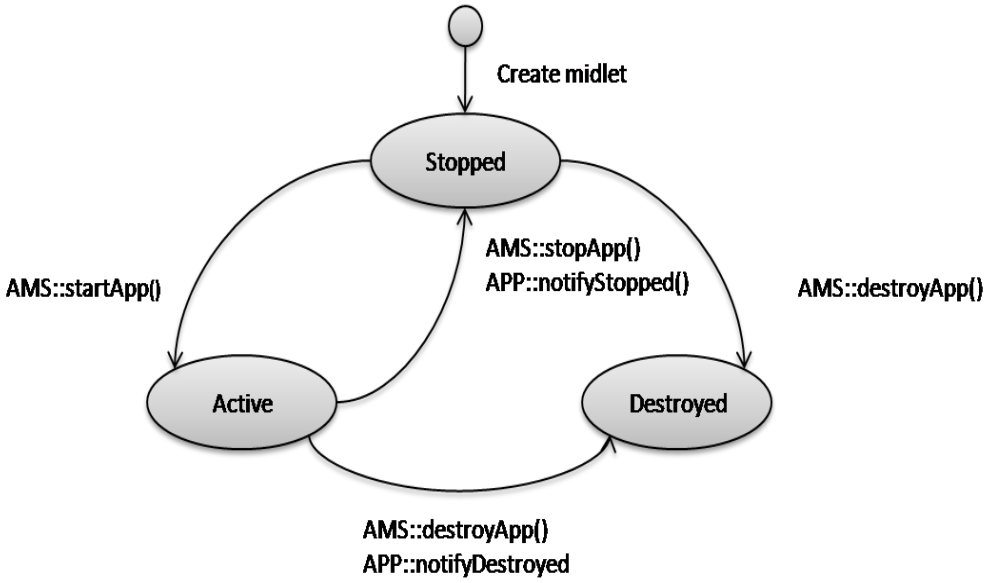


Figure 2: MIDlet lifecycle and state transitions

The MIDP profile also provides security model based on protection domains. These protection domains contain sets of permissions the application has granted. Permissions are related to the API functions and their names are derived from package, class and method name. The application can check whether it has been granted the particular permission at runtime. The application can also specify the sets of required and optional permissions in its descriptor file.

One MIDlet application should be deployed as single Java archive (jar) file. However this jar file may contain multiple MIDlet applications, in this case the whole deployment unit (one jar file) is called MIDlet suite. Each jar file with MIDlets may be accompanied by descriptor file with extension *.jad (jad file). The descriptor file contains attributes of the

MIDlet suite – for example the required configuration and profile, required permissions and others. These attributes may be duplicated in manifest file in the jar archive. When deploying the application over the network just the descriptor file can be downloaded prior to downloading the whole jar to inspect whether the device is capable of running the MIDlet suite and save the network bandwidth.

At this time there are four versions of MIDP profile available – 1.0, 2.0, 2.1 and 3.0. The version 3.0 currently provides just specification and reference implementation and no development tools. Also the range of devices supporting MIDP 3.0 is very limited. For the purpose of this thesis the version 2.0 has been chosen, because it usually accompanies the chosen configuration CLDC 1.1 and offers more features than 1.0 and is spread more widely than the version 3.0.

2.2. SOFA 2

SOFA 2 is a component system developed at Department of Distributed and Dependable Systems, Charles University, Prague. It defines its own component model and also runtime support for components. SOFA 2 provides some advanced component features like hierarchical nesting of components, dynamic reconfiguration of components, extensible architecture of component control and management mechanism and multiple communication styles.

2.2.1. Component model

Component model for SOFA 2 is defined as EMF [**24**] metamodel instead of ADL[1] as is common in other component models (for example SOFA). However, additional ADL has been designed to simplify development of component applications so the developer does not have to describe the architecture of component application using XMI [**25**]. ADL is automatically transformed into XMI during development process. The EMF-based approach brings several benefits related mainly to the automatic generation of repositories and editors for models. SOFA 2 uses EMF repository to store models of component applications (instances of the EMF metamodel). To provide identification to entities stored in repository, the base entity *NamedEntity* has been designed. It provides name for any entity that inherits from it. The repository can also store several versions of the same entity. All entities for which this feature should be enabled have to be derived from entity *VersionedEntity*.

In component model the black-box view of SOFA 2 component is represented by *Frame*. The frame defines the collections of interfaces it requires and provides. The interface type is described using entity *InterfaceType* and it is usually bound to an actual interface in particular programming language. InterfaceType is wrapped in entity *Interface* which describes some additional features of the communication endpoint (for example communication style - more on this can be found later in this chapter). In the end the frame contains collections of instances of interface entities which describe its provided and required

[1] Architecture definition language

interfaces. In addition to this, the frame can also contain behavior specification that can be used during behavior checking.

The design time view of component is represented by *Architecture*. Architecture can implement several frames and the single frame can be implemented by multiple architectures. The architecture gives the developer the possibility of defining nested components, because it can contain subcomponents definitions. The subcomponent is described using the frame and implementing architecture. Developer also has the specify connections between interfaces of particular subcomponents. These connections can be of three types – *connection, delegation* and *subsumption.* Connection connects required and provided interfaces of the components on the same level of hierarchy, delegation connects provided interface of the component to the provided interface of the subcomponent, subsumption connects required interface of the subcomponent to required interface of the parent component. If the architecture does not contain any subcomponents, it has to contain code that provides its functionality. These architectures are called *primitive architectures*; the former ones are called *composite architectures*. The example of whole situation is depicted on Figure 3. In this picture the top level component is defined by frame A and it provides interface depicted by the black box and requires interface represented by the white box. The Frame is implemented by architecture A, which is composed of subcomponents defined by frames B and C and respective architectures. Architectures B and C are primitive architectures and they provide code that implements their functionality. Calls to interfaces the component provides are propagated to innermost component where implementation code deals with them. On the other hand the calls from the implementation code are propagated out of the component and directed to site that services the call (can be another component or third-party site).

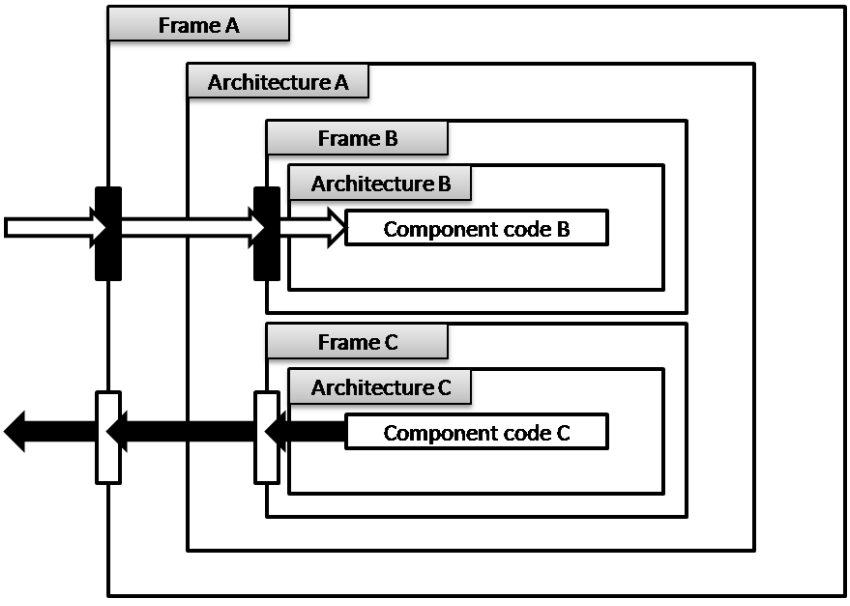


Figure 3: Example of nested component architecture

Frames and architectures can be associated with entities *Property* and *PropertySet* (where PropertySet is set or Property entities). These represent the properties of components and can be used to parameterize the components as the values can be set during the application deployment.

As mentioned before, the developer can interconnect the subcomponents in the architecture. These connections are realized through connectors which exist both at design time and run time. Connectors provide multiple communication styles – *method invocation*, *streaming* and *messaging*. The communication style of particular connector has to be inferred from communication style of the interfaces it connects.

SOFA 2 also supports dynamic reconfiguration of architecture of component application. To avoid erosion of architecture that can be caused by uncontrolled reconfiguration three patterns were designed: *nested factory pattern, component removal pattern* and *utility interface pattern*. Detailed description of these patterns can be found in [**26**].

Apart from the required and provided interfaces of the component (these are called *business interfaces*) the component can also provide special *control* interfaces that can be used to administrate the component. To make the control part of the component extensible *aspects* and *microcomponents* have been designed. Microcomponents incorporate simple flat component model and form the functional portion of the control part of the component. Microcomponents may also provide and require interfaces, but unlike frames and architectures these interfaces are represented using special entity *MicroInterfaceType*. Aspects define which control interfaces should be available for the component and also specify which microcomponents should be connected to these control interfaces and form the control part of the component. Microcomponents can also be used for intercepting calls to business and control interfaces. Multiple microcomponents can interrupt same call and these microcomponents are organized in *delegation chains*. The range of components and business interfaces to which the particular aspect is applied can be restricted by *component select* and *interface select.* The component select determines the type of the component to which the aspect can be applied – *all, primitive* or *composite*. The exact name of the component can also be defined or wildcard "*" can be used. The interface select defines the name (again, the wildcard "*" can be used) and type of the interface which the microcomponent intercepts – can be one of *all, control, provided, required, business, collection* or *factory*. There are three basic aspects – bootstrap aspects – *lifecycle, component* and *incomponent*. Lifecycle aspect is used to manage component lifecycle (starting and stopping the whole component), component aspect provides access to basic information about the component itself – provided and required interfaces, defining frame and implementing architecture of the component – and incomponent aspect enables the business code to access control interface of component aspect.

2.2.2. Component runtime

In addition to the component model the SOFA 2 also provides the runtime for running the component applications defined in SOFA 2 component model. The environment in which the SOFA 2 application lives is called SOFANode. SOFANode is distributed environment capable of running on separate physical machines which consists of *deployment docks, deployment dock registry, repository* and *global connector manager*. Deployment docks serve as execution environments for running components of particular applications and each dock is capable of running several components from several applications. Deployment dock registry

is used to lookup particular deployment dock during application launch. Repository stores the information about the component application definitions (frames, architectures etc.) and code for the primitive architectures in so-called *code bundles*. Finally the global connector manager takes care of connecting connectors among components deployed in different deployment docks. The communication between deployment docks, deployment dock registry and global connector manager is based on RMI. On Figure 4 the SOFANode with one deployment dock is pictured. In the deployment dock two components are currently running (component A and component B).

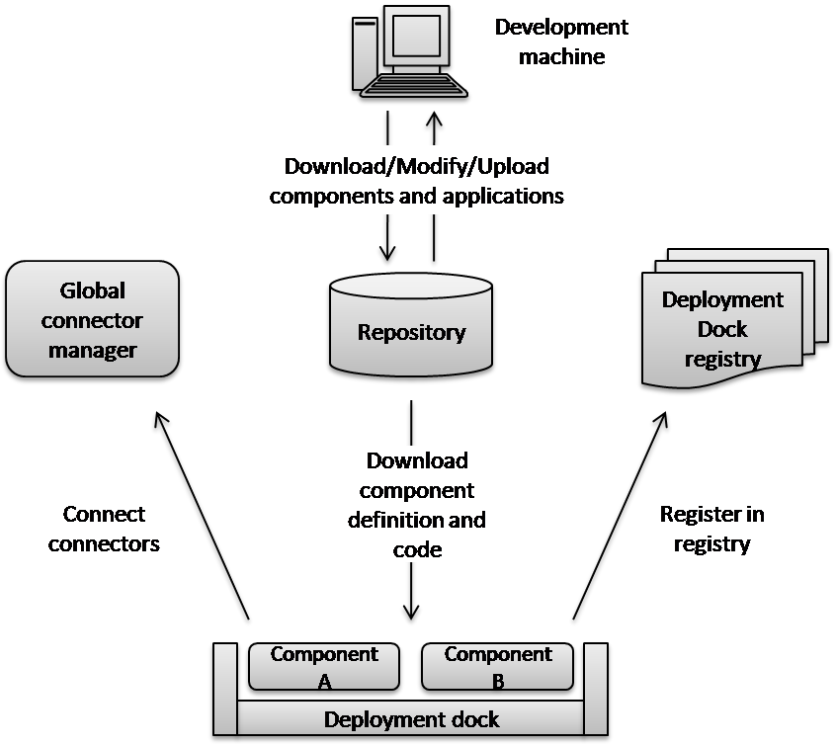


Figure 4: SOFANode with one deployment dock and development machine

Before component application can be started in SOFANode, the developer has to provide some information about the structure of the application. The *assembly descriptor* needs to be created which describes the structure of the application from top-level architecture down to innermost primitive architecture. Architectures for all subcomponents have to be filled. Then the *deployment plan* can be created from this assembly descriptor. In the deployment plan the developer has to specify which component should run on which particular deployment dock and he can also specify the properties of components (as mentioned in previous chapter). Then the deployment plan can be deployed. During this process the connectors are generated according to the architecture and placement of the components. For example when two connected components communicate via method invocation and should run in different docks the RMI is used in connectors, when they run in the same dock local methods can be called. The connector for communication style messaging exploit JMS [**27**] and those for streaming are base on standard Java sockets. Then the application can be launched.

Before launching SOFA 2 application the whole SOFANode – repository, deployment dock registry, proper deployment docks and global connector manager - has to be started.

When deployment dock starts it has to register itself in the deployment dock registry. Launch process starts on dock where the top-level component has to be deployed and then is recursively called on subcomponents and respective deployment docks. During the launch process the runtime representation of components is created. The code for primitive components is loaded using custom classloader which is capable of searching code bundles in the remote repository for needed classes. Because several components can run on the same deployment dock there is possibility that two components are implemented by the class with same name, but with different versions (for example application provides older version of some component for backward compatibility but also provides new version). Therefore renamed classes are loaded into Java virtual machine. The classes are however renamed during the development process already as is described in chapter 2.2.3. When the application is successfully deployed on target deployment docks the components can be started.

The SOFA 2 defines special component execution lifecycle which is managed through lifecycle aspect. The component can reside in one of the following states: *starting*, *started*, *stopping* and *stopped*. When component is instantiated, it is in the stopped state. The launch process can then start the component application. During the start process the component is in starting state and after the starting process finishes it is moved to the state started. The component can be also stopped; this process has two steps – first lifecycle aspect notifies the component that the stopping process started and the component enters state stopping. This is non-blocking process. Then the lifecycle aspect notifies the component it has to stop and this operation blocks until the component is stopped. The component will then be in the state stopped. The component can then be exited or started again. The code that implements primitive components is notified about the lifecycle changes through lifecycle related interfaces `SOFARunnable`, `SOFAStoppable` and `SOFALifecycle` which it has to implement if it wants to react to the lifecycle state changes.

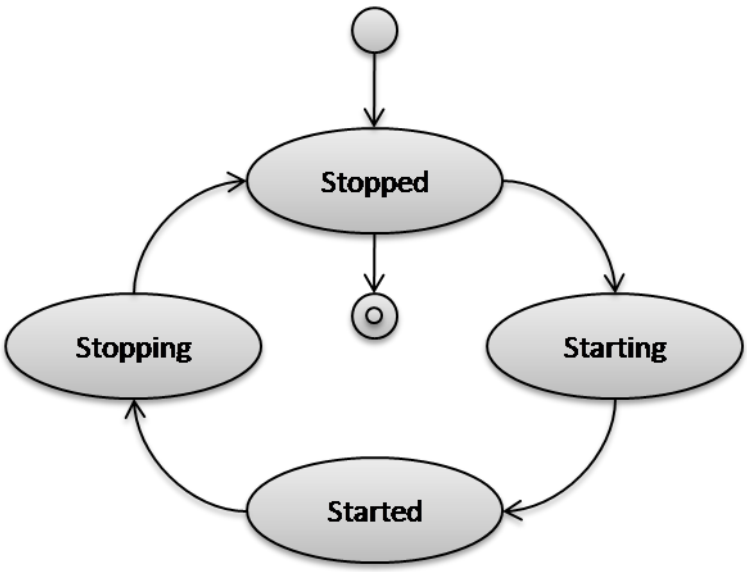


Figure 5: Lifecycle of SOFA 2 component during execution

If the component wants to be able to shut down the whole application, it has to implement interface `SOFASelfShutting` through which it obtains the reference to object of type `SOFAShutdownContext` which can be used to shut down whole application. The

lifecycle aspect also interrupts the calls to the provided interfaces and determines if the calling thread should continue or wait. Generally all calls to component in state other than started are blocked with exception of thread reentering the component it has already entered. The child threads of such a thread are also permitted to enter the component's code. This thread reentrance functionality is implemented using thread context which is stored in thread local storage. Thread context of child thread is derived from the context of parent thread therefore the lifecycle implementation can easily determine whether the thread can enter the component. The lifecycle of component is depicted on Figure 5.

The thread local storage is also used in implementation of incomponent aspect. This aspect also interrupts calls to business interfaces of components and sets the thread local variable which holds the reference to the incomponent microcomponent of the current component.

The intercepting functionality used in lifecycle and incomponent aspects is implemented by interceptors which are generated for each intercepted interface using ASM framework [**28**] during the launch process. The interceptors are generated in a way they cannot collide with the methods in business interfaces. The interceptors are usually bound to other microcomponents that are notified when the call to the interfaces is made and when the call returns.

2.2.3. SOFA 2 development process

On Figure 4 there is one part that is not mentioned in chapter 2.2.2. It is the development machine. The SOFA 2 also provides the development tools to make the development of component applications easier. These include command line tool called cushion [**29**] and plugin SOFA IDE [**30**] for Eclipse [**31**]. Both these tools provide access to the repository from which the files with ADL of the entities can be downloaded, than edited and later uploaded back to the repository. The entities in repository are store using XMI therefore in download/upload process the two-way transformation from XMI to ADL takes place. These tools also provide functionality to compile and upload code for the primitive components. When the code bundles with code for these primitive components are uploaded the renaming mentioned in chapter 2.2.2 takes place. The names of all classes from the code bundle are augmented with the version of the code bundle and all references to these renamed classes are updated. As the code bundles can depend on each other, the renaming process has to take this into consideration.

2.3. Restrictions and requirements on deployment process and component application

In this section all restrictions and requirements on component application and the deployment process are analyzed.

2.3.1. Restrictions posed by Java ME

First step is to identify the restrictions and requirements the Java ME environment poses. As mentioned in chapters 2.1.4 and 2.1.5 the target platform is composed of CLDC 1.1 configuration and MIDP 2.0 profile. Following text refers to this set up as "target device" or "target platform". The most significant restrictions posed by this decision are:

a) **No support for RMI** – CLDC configuration lacks support for RMI mainly due to the security reasons.

b) **Absence of JMS** – MIDP profile lacks the JMS provider; connection to the JMS provider is possible only through some kind of gateway solution.

c) **The Generic Connection Framework** – the networking in MIDP profile is different than in Java SE, only implementation of HTTP protocol is mandatory.

d) **No custom classloaders** – CLDC configuration does not allow defining custom classloaders.

e) **Sandbox security model** – the application cannot escape its sandbox and can use just classes provided by the runtime or the application itself. Downloading of additional jar files is prohibited.

f) **Restricted environment** – target devices on which the Java ME environment is running are limited in terms of available memory and processing power, therefore these should be used wisely.

Since the communication in current implementation of SOFANode relies heavily on RMI, point a) implies that connecting to deployment dock registry, global connector manager or contacting other deployment docks from Java ME environment is not possible. Also method invocation style connectors could not be used if components were distributed to different deployment docks because the generated connectors are currently based on RMI. Local method invocation should not be affected. Point b) disallows the use of connectors with communication style messaging because the generated connectors currently rely on JMS. The use of streaming connectors is complicated by the fact that the MIDP implementation of networking has different structure than the one in Java SE thus the generated connectors are currently not compatible and adjustments needed to achieve compatibility would require significant changes in connector generator. In addition to this the absence of protocols other than HTTP is very limiting. Point d) together with e) makes the implementation of custom classloader and downloading classes on demand from repository impossible. Therefore downloading code for components from repository and instantiating it in deployment dock is not possible in the target Java ME environment. Last restriction does not have any direct effect but has to be taken into consideration.

2.3.2. Requirements posed by Java ME

In addition to the above described restrictions the target platform also poses following requirements:

a) The source code has to comply with Java language version 1.3.

b) The application code needs to be compiled against bootstrap files of target platform.

c) The versions of the compiled classes has to be in range 45.* - 48.* (see section 2.1.4).

d) The compiled classes have to be preverified before deploying the application to the target device.

e) Jad file needs to be generated.

These requirements imply that the code of primitive components, business interfaces and microcomponents has to be compiled against different bootstrap files than common SOFA 2 application and the code itself must not use any features from higher versions of Java language. The current compilation process also generates classes that cannot be loaded into Java ME virtual machine because they do not have proper version. In addition to this the class files needs to be preverified and the jad file has to be generated before the application can be deployed to the target device. Neither of the last two steps is supported in current implementation of SOFA 2.

3. Java ME deployment for SOFA 2 applications

As mentioned in chapter 1.1 the primary goal of the thesis is to provide mechanisms for deploying the component applications developed in SOFA 2 in Java ME environment. The general idea is to keep as much as possible from the original development and deployment process of SOFA 2 application and to introduce only necessary changes implied by the Java ME environment.

3.1. Analysis of Java ME restrictions and requirements

Following conclusions can derived from the restrictions posed in chapter 2.3.1:

a) Target device is not capable of running common SOFA 2 deployment dock and cannot be a part of the SOFANode infrastructure.

b) Target mobile device is not capable of downloading components and their code on-demand.

c) Component application running on the target device can communicate only using local method invocation.

These conclusions imply that deployment process of SOFA 2 application has to be changed significantly in order to allow applications to be deployed in Java ME environment. The whole component application has to be prepared off the target device and then uploaded to it as any other mobile application. Therefore the distributed, dynamically instantiated component application has to be transformed into application that looks monolithic to the target device. In addition to this, the component application has to communicate using only local method invocation. The development process has to be adjusted - it has to incorporate compilation against different runtime, class file preverification and MIDlet packaging.

3.1.1. Dynamic reconfiguration of architecture

As mentioned in chapter 2.2.1 SOFA 2 allows the architecture to evolve in time using predefined patterns. However these patterns are not fully supported in current Java SE implementation of SOFA 2 runtime. Also dynamic reconfiguration would bring additional overhead to the restricted environment of the target device. Because of these reasons the deployment process proposed in this thesis will not support dynamic reconfiguration of architecture of component applications.

3.1.2. Summary of requirements and restrictions

The conclusion from previous section is: the SOFA 2 component application that is going to be deployed in Java ME environment cannot be distributed, has to communicate using local method invocation, its implementation code has to conform to the Java language version 1.3 and its architecture has to be static. Before deploying to target device, the

application has to be transformed into single runnable package, preverified and the jad file has to be generated.

3.2. Incorporating Java ME deployment into SOFA 2 development process

In Figure 6 the development process of SOFA 2 component application is depicted. On the left side there is the process for developing common SOFA Java SE applications. It starts with designing the architecture of the application (defining frames, architectures etc.), which has to be uploaded into the repository. Then the code for primitive components has to be implemented and also uploaded into the repository. The deployment of the application encompasses creating assembly descriptor, deployment plan and deploying the deployment plan. Then the deployment plan can be launched in SOFANode environment. However the development process for the Java ME component applications needs to be slightly different. The first difference is that the developer has to keep in mind that he cannot use the Java SE to implement code of the primitive components, business interfaces or microcomponents. However if the components does not use any platform – specific features the code of these components may be shared between the Java SE and Java ME implementation. Next the code needs to be compiled against Java ME-specific bootstrap jar files.

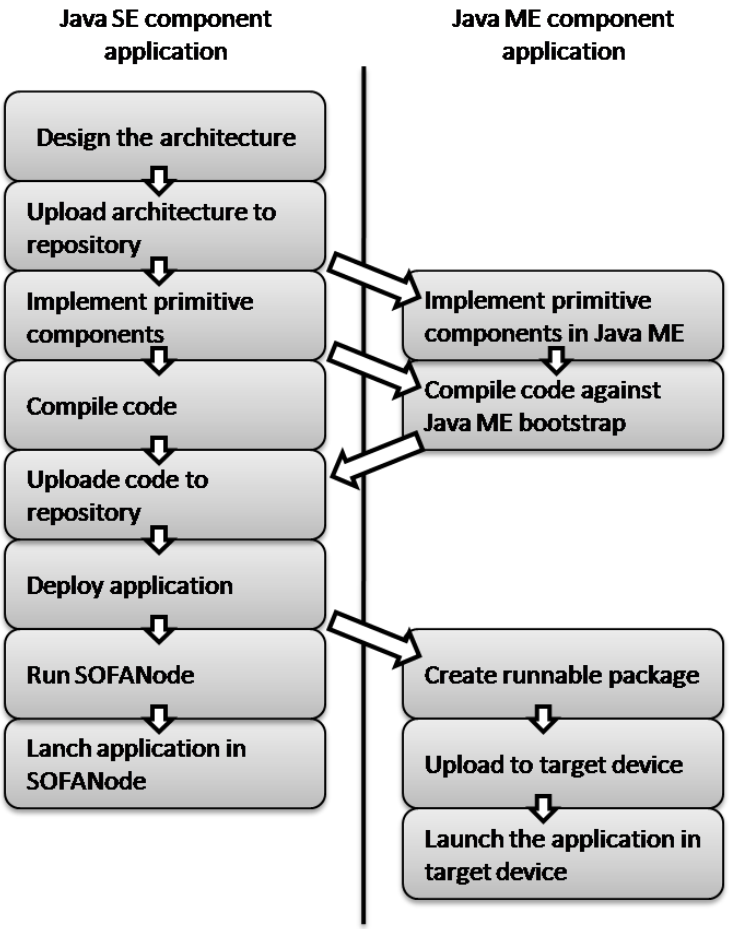


Figure 6: Development process of component application

Then the application has to be deployed as in standard development process; however the deployment plan cannot be distributed. From this point the process differs significantly. For Java ME deployment the runnable package has to be created from the component application, then it has to be uploaded to target device and launched.

In following chapters the analysis and design of Java ME specific parts of the development process is described. The description starts with the creating of runnable package because that is the most significant part of whole process.

3.3. Runnable package for component application

The target platform implies that the runnable package has to be MIDlet application (SOFA MIDlet). This MIDlet has to provide mechanisms for instantiation of microcomponents, components, connecting components using generated connectors and provide functionality the deployment dock provides to the running application. Therefore it has to contain the code for primitive components, microcomponents, connectors, generated interceptors and some kind of a minimal infrastructure that manages all these parts.

3.3.1. Architecture of SOFA MIDlet

The main question when designing the MIDlet for SOFA 2 is how to instantiate the components. One possible approach is to take inspiration from instantiating components in common deployment dock where the component instance is represented by one general class that holds information about the component – its business and control interfaces, microcomponents and connections. This class is parameterized according to component's description in defining frame, architecture and deployment plan. This approach is however too complex for target platform – representation of deployment plan and component application architecture would be needed to be passed to the target device which might consume too much memory and the processing would require additional system resources. Therefore the component instantiation should be prepared during the creation of SOFA MIDlet.

The proposed solution is based on generating a particular Java class for each component of application. These classes should initialize themselves properly – initialize business and control interfaces, set up microcomponents, instantiate implementing classes for primitive components – but these actions should be hard coded in the class and not performed dynamically as in case of interpreting the deployment plan, frames, architectures etc. For example during the common SOFA 2 instantiation process the frame has to be searched for business interfaces in a loop and the proper delegation chains are set up to form component instance. However the generated class that should represent the component instance in SOFA MIDlet has this initialization hardcoded in one of its methods. This way each component has its own specific method that initializes exactly those business interfaces that are needed for the particular component instance. This principle applies not just to business interfaces but also to control interfaces, microcomponents and implementation code of primitive

components. This makes the instantiating process simpler and the target device will not be overwhelmed by it.

As mentioned in chapter 2.1.5 the MIDlet application needs its entry class to be derived from MIDlet class therefore the SOFA MIDlet application also has to have one. One possible solution is to force the developer to implement it in one the primitive components. However this component would be nothing more than a controller that manages whole application and developer would have to initiate instantiation of components from his code. It is obvious that this functionality can be generalized and provided by the SOFA MIDlet infrastructure therefore this approach has been chosen. The infrastructure therefore contains class derived from MIDlet class that manages whole application.

At runtime the deployment dock provides components with infrastructure for managing connectors and microcomponents. This infrastructure also needs to be the part of SOFA MIDlet. However it has to be slightly redesigned to fit into the target device.

Previous paragraphs imply that the SOFA MIDlet is composed of 5 basic parts:

a) **Infrastructure** – MIDlet class to manage whole application, management of microcomponents and connectors, interfaces for component API.

b) **Code of microcomponents**

c) **Code of generated interceptors**

d) **Code of connectors**

e) **Generated classes for components** – for each component in the application separate class will be generated

The MIDlet suite with the SOFA MIDlet should contain just one MIDlet – the one for the component application.

3.3.2. Full - featured development of SOFA 2 applications for Java ME

Until now the analysis was focused on making the Java ME deployment of SOFA 2 applications possible. Now it is clear that each component application is a MIDlet but the SOFA 2 applications are not capable of exploiting this fact. Some functionality is simply not available to the components because the current runtime is not able to provide it to the implementing code. This decreases the potential of such applications (for example they are not capable of displaying anything on graphic user interface). For development of full – featured MIDlet applications requirements summarized in following sections need to be met and taken into consideration when designing the MIDlet's infrastructure code:

a) **MIDlet and component lifecycle management** - the components should be notified about state changes of SOFA MIDlet and be able to react appropriately. This means acquire/free resources, start or stop threads or even reject the state change as is usual in common MIDlet applications. This may require some adjustments in current design and implementation of lifecycle of components.

b) **Lifecycle self management** - common MIDlet application is able to manage its lifecycle by notifying the AMS (Application Management Software, see chapter 2.1.5) that it can be stopped; destroyed or that it wants to run. SOFA 2 however provides just the possibility to shutdown the whole application.
c) **Access to MIDlet's user interface** - in order to display anything on graphical user interface the access to instance of class `javax.microedition.midlet.lcdui.Display` is needed. To obtain this instance however the reference to MIDlet object is needed. Therefore some mechanism of passing instance of `Display` class has to be designed.
d) **Checking the application permissions** - MIDlet applications are able to check if particular permission has been granted to the application or not. This functionality should be enabled also for the code implementing primitive components.
e) **Setting the jad file attributes** - jad file contains attributes describing the application and requirements the MIDlet poses on the target device. The developer can fill the attributes on his own however some attributes could be defined during the development of SOFA MIDlet and later just summarized in the jad file.
f) **Reading the values of attributes from jad file** - the common MIDlet application is able to read values of attributes from the jad file.
g) **Launching web browser** - SOFA MIDlet should be able to launch web browser. In common MIDlet application this is done using method of main MIDlet class. Since the main MIDlet class is a part of infrastructure and not the implementation code of components, this functionality has to be accessed in other way.
h) **Using optional packages** - code of primitive components should be able to exploit APIs provided by optional packages. Therefore jar files of these optional packages need to be added to classpath when compiling code of the components.

3.4. Migrating SOFA 2 to MIDlet application

This chapter describes all particular changes needed in SOFA 2 component model and runtime in order to deploy SOFA 2 applications as MIDlets. All changes are based on requirements and restrictions described in previous chapters.

3.4.1. Component API

SOFA 2 provides set of interfaces the component developer can implement in code for primitive architectures so the runtime can communicate with the components. Other possible way is to use Java annotations in the code and the runtime will interpret them using reflection. However annotations are not supported in Java language version 1.3 therefore the usage of annotations in code developed for SOFA MIDlets is prohibited.

The list of interfaces that forms component API and differences when developing SOFA MIDlets:

- SOFAClient – implemented in components with required interfaces. Interface can be used in SOFA 2 MIDlet applications as in usual SOFA 2 application.

- SOFALifecycle – implemented by components that need to be notified about changes in component's lifecycle. This interface is in fact union of the interfaces `SOFARunnable` and `SOFAStoppable`. It can be used in SOFA MIDlet applications as in usual SOFA 2 application.

- SOFAParametrized – implemented by components that need to be parameterized by values specified in deployment plan. Since it uses class `java.util.Properties` which is not available for MIDlet applications its use is prohibited. Interface `Sofa2MeParametrized` provides similar functionality for SOFA MIDlet applications.

- SOFARunnable – implemented by components that need to be notified that the component has been started. Interface can be used as in common SOFA 2 applications.

- SOFASelfShutting – implemented by components that want to be able to shut down the whole component application. This interface is used to pass reference to instance of `SOFAShutdownContext` to the component. It can be used also in SOFA MIDlet applications.

- SOFAServer – implemented by components with provided interfaces. Interface can be used as in common SOFA 2 applications.

- SOFAShutdownContext – grants the component the possibility of shutting down the whole application. Interface can be used as in common SOFA 2 applications.

- SOFAStoppable - implemented by components that need to be notified that the component is going to be stopped or should be stopped. This interface can be used as in common SOFA 2 applications.

- SOFAUpdatable – used for updating business code. Since the dynamic reconfiguration is not allowed for SOFA 2 MIDlets, the usage of this interface is prohibited.

Following interfaces have been added to the component API:

- Sofa2MeLifeCycle – derives from interface `SOFALifecycle` and adds MIDlet-specific features. The interface is described in detail in section 3.4.6.

- SOFASelfPausing – enables the SOFA MIDlet to pause itself. Description of this interface can be found in section 3.4.6.

- Sofa2MeParametrized – replaces the interface SOFAParametrized. Detailed description can be found in section 3.4.2.

3.4.2. Parameterization of SOFA MIDlet

SOFA 2 components are parameterized using properties defined in frames and architectures (as mentioned in chapter 2.2.1). Values for these properties are specified in the deployment plan. These properties are passed to components through interface `SOFAParametrized` where instance of `java.util.Properties` class is passed to component. However this class is not available on the target platform, therefore new class `Sofa2MeParametrized` has been introduced. This class implements basic methods from common `Properties` class excluding those related to loading, saving and listing properties. If component wants to obtain its properties it has to implement interface `Sofa2MeParametrized` through which it can obtain the instance of class `Sofa2MeProperties`. Both are showed in Figure 7.

```
public interface Sofa2MeParametrized {
    public void setProperties(Sofa2MeProperties props);
}
public class Sofa2MeProperties extends Hashtable {
    public String getProperty(String key){ ... }
    public String getProperty(String key, String defaultValue){ ... }
    public Enumeration propertyNames() { ... }
    public Object setProperty(String key, String value){ ... }
}
```

Figure 7: Interface Sofa2Parametrized and class Sofa2MeProperties

According to section 3.3.2, point f) the common MIDlet is able to read values of its attributes from jad file. However this way of parameterization is not supported in SOFA MIDlets as this functionality is already provided through parameters of frames/architectures.

3.4.2.1. Specifying the MIDlet's attributes

In section 3.3.2 in point e) the need for more convenient way of defining attributes for jad file was mentioned. The developer can fill the attributes by hand; however these manual adjustments can be avoided (or at least minimalized). Designing special mechanisms for defining attributes and their values is needless because the current property model can be exploited for this purpose. In component model the Property entity has two attributes – "name" and "type" and this are used for specifying the MIDlet attribute. The property that should be used as MIDlet attribute has to have name "sofa2Me.midlet.attribute" and in the type there should be the name of attribute and its value separated by "|" (vertical bar) character. For example when the architecture wants to define the required size of persistent storage for its data, the type of the property should contain string "MIDlet-Data-Size|10", where "MIDlet-Data-Size" is name of the attribute and the "10" is its value. When multiple frames/architectures specify the value for the same attribute these values are aggregated in a way suitable for particular attribute. For the common attributes the aggregation rules are defined (in case of the given example with required size of persistent storage the result is sum of all values), the custom attributes are simply concatenated using " " (space) character as a separator. During the launch of common SOFA 2 application the properties defined in frames and architectures are checked whether the value for them has been defined in deployment plan

and if not, the warning is issued. This is not case when the property is supposed to be a MIDlet attribute.

Some MIDlet attributes are specific for the current deployment or distribution, not the particular frame of architecture. Currently the SOFA 2 component model allows defining the values of properties for particular component instances (using collections of *PropertyValue* entities), but not for the deployment plan in general. Therefore the component model has been extended and the *DeploymentPlan* entity can now directly contain collection of PropertyValue entities. The definition of ADL has also been extended to allow developer to specify the attributes when designing the deployment plan. This can be done using xml tags `deplplan-prop` as is shown in Figure 8.

```
<depl-plan name="sofaworm.deplplan.WormGame" node="nodeA">

    <deplplan-prop name="sofa2Me.midlet.attribute">MIDlet-
Name|SofaWorm</deplplan-prop>

</depl-plan>
```

Figure 8: Property for deployment plan that will be used as attributes in jad file

Some attributes are mandatory and they must be present in the jad file otherwise the application will not be able to run. These attributes should be specified by the developer; however the default value has been defined for each of these attributes so the developer does not need to bother with these issues when he does not want to. These attributes include:

- MIDlet-Name – name of the whole MIDlet suite, default is "SOFA MIDlet application".
- MIDlet-Version – version of the whole MIDlet suite, default "1.0.0".
- MIDlet-Vendor – vendor of the whole MIDlet suite, default is "org.objectweb.dsrg".
- MIDlet-Jar-URL – URL from which the MIDlet jar file can be downloaded, default is "Sofa2MeApp.jar".
- MIDlet-Jar-Size – size of the jar package that contains the SOFA MIDlet application, dynamically computed according to the real size.
- MicroEdition-Profile – profile required by the MIDlet suite, default "MIDP-2.0".
- MicroEdition-Configuration – configuration required by the MIDlet suite, default "CLDC-1.1".

Since the jar file can in general contain several MIDlet applications additional attribute describing the particular MIDlet is required. This attribute has name MIDlet-n where n is replace by the number of the MIDlet (in case of SOFA MIDlet always 1) and it contains the name of the particular MIDlet, path to the icon of the MIDlet and the name of the main class

of the MIDlet. The name of the class is predefined; however the remaining two attributes can be specified using special attributes "MIDlet-Midlet-Name" and "MIDlet-Midlet-Icon". The values of these attributes are filled into the value of the attribute "MIDlet-1" and will not appear in jad file under the names under which they have been specified. The default value for MIDlet-Midlet-Name is the same as for MIDlet-Name and for MIDlet-Midlet-Icon the default is empty string.

The last unresolved issue concerns the name of the jar and jad file. This can be inferred from the URL of the jar file. When no URL is specified or the file name cannot be inferred, the default value "Sofa2MeApp" is used.

3.4.3. Microcomponent architecture

Implementation of SOFA 2 runtime for Java SE provides set of classes and interfaces for managing and developing microcomponents. The implementation for SOFA MIDlet application should provide the same functionality and programming model to the developer therefore porting the model from Java SE runtime to Java ME has been chosen as the best option. Ideally whole microcomponent infrastructure could be reused; however this is not always the case. In this chapter all required changes to microcomponent architecture are described.

3.4.3.1. Microcomponent infrastructure interfaces

In microcomponent architecture each component instance is represented by the instance of class that implements interface `ComponentDescription`. This interface however provides access to component's defining frame and architecture which are not available on target device and use generic collections in some of the defined methods. Therefore new interface `Sofa2MeComponentDescription` has been designed that fixes these problems. This interface is shown on Figure 9.

```
public interface Sofa2MeComponentDescription {
    public Object getComponentContent();
    public Vector getRequiredInterfacesNames();
    public Sofa2MeDelegationChain getRequiredInterfaceChain(String name)
        throws InterfaceNotFoundException;
    public Vector getProvidedInterfacesNames();
    public Sofa2MeDelegationChain getProvidedInterfaceChain(String name)
        throws InterfaceNotFoundException;
    public Vector getControlInterfacesNames();
    public Sofa2MeDelegationChain getControlInterfaceChain(String name)
        throws InterfaceNotFoundException;
}
```

Figure 9: Sofa2MeComponendDescription interface

In SOFA 2 runtime for Java SE the implementation of particular microcomponent has to provide class that implements interface `SOFAMicroComponent`. However this interface provides method `init(...)` which takes parameters of types `ComponentDescription` and generic `Map`. Therefore it cannot be used and new interface `Sofa2MeMicroComponent` has been introduced. This interface is shown in Figure 10.

```
public interface Sofa2MeMicroComponent {
    void init (ComponentDescription component,
            Map<String, String> parameters) throws SOFAException;
    Object getProvided (String name) throws SOFAException;
    void setRequired( String name, Object ref ) throws SOFAException;
}
```

Figure 10: Sofa2MeMicrocomponent interface

As mentioned in chapter 2.2.1 the microcomponents are composed into delegation chains. These are at runtime represented by objects of type `DelegationChainInstance` (derived from abstract base class `DelegationChain`) which takes care of instantiating microcomponents and connecting them. However these classes are supposed to work with microcomponents implementing interface `SOFAMicroComponent` and during instantiation of microcomponents they download the code for microcomponents from the repository. Therefore the delegation chains have been reimplemented in classes Sofa2Me`DelegationChainInstance` and Sofa2Me`DelegationChain`. These classes are able to work with classes that implement interface `Sofa2MeMicroComponent` and the microcomponents have to already be instantiated before they are inserted into delegation chain.

SOFA 2 also offers the possibility to generate components at runtime. This functionality is used particularly for generating microcomponents that should intercept calls to business interfaces. However for the purpose of SOFA MIDlet applications the generating of microcomponents at runtime would cause significant overhead that can be avoided. For example the current implementation of interceptor generator uses ASM framework to perform the generation and it is not convenient to require the SOFA MIDlet to contain the whole ASM framework. In addition to this the generator may even not be able to run on target platform. Therefore the generated microcomponents are pregenerated during the preparation of SOFA MIDlet and packed into the MIDlet jar file.

In Java SE implementation the microcomponents that generate other microcomponents have to implement interface `SOFAMicroGenerator` which provides method that returns class that should be instantiated as microcomponent; however in addition to this the generator can generate many helper classes. This causes problems when the generated microcomponents should be packed into the MIDlet's jar because all classes that form the microcomponent have to be identified. There are several possible solutions for this problem – a tool for class dependency analysis can be used, the generator has to supply the list of generated class or the generator can store the generated classes in class files in some specified location. The class dependency analyzers often need class files in order to discover the class dependencies and these dependencies would need additional revision in order to determine which classes have been really generated and which are from Java ME or SOFA 2 runtime. If the generator would provide the list of classes that form the microcomponent it would be possible to store just the classes that were specified. However during the generation process the generator often keeps its internal representation of generated class that is more suitable for storing than the class type. Therefore the third solution has been chosen and saving the generated classes is generator's responsibility. In order to define the location where the

generated classes should be stored the new interface `Sofa2MeMicroGenerator` has been designed. The interface is shown in Figure 11. The parameter `outputDir` denotes the location where the files with generated classes should be stored.

```
public interface Sofa2MeMicroGenerator {
    Class<?> generate(Class<?> iface,
        Map<String, String> parameters, File outputDir)
           throws SOFAException, SecurityException,
               NoSuchFieldException, Exception;
}
```

Figure 11: Sofa2MeMicroGenerator interface

3.4.3.2. Aspects

As mentioned in chapter 2.2.1 aspects define which control interfaces and microcomponents should be instantiated in which components and for which interfaces. The functionality of aspects and component/interface selects remains the same for SOFA MIDlets with one adjustment. The entity *Aspect* in SOFA 2 component model has been extended by new attribute "platform" of type string. The attribute can have values "J2ME" or "J2SE". This attribute determines whether the particular aspect should be applied when the component is going to be deployed. If the component is going to be deployed as a part of common SOFA 2 application only aspects with value of platform equal to "J2SE" should be applied. On the other hand if the component is going to be deployed in SOFA 2 MIDlet only aspects with platform equal to "J2ME" should be applied. Reasons for this solution are described in chapter 3.5.2.

3.4.4. Bootstrap aspects

Bootstrap aspects add three control interfaces to each component – Component, InComponent and Lifecycle – and microcomponents that implement functionality behind these control interfaces. Since the microcomponent architecture has slightly changed the bootstrap control interfaces and microcomponents had to be reimplemented in new, Java ME specific aspects. In addition to these changes some other changes had to be made.

3.4.4.1. Component aspect

Component aspect adds Component control interface to each component. The interface

```
public interface Sofa2MeMIComponent {
    Sofa2MeDelegationChain getDelegationChain(int type, String name)
        throws SOFAException;
    Vector getDelegationChains ();
    Vector getControlInterfaces ();
    Object getControlInterface (String name);
    Object getContent ();
    void setContent (Object o);
}
```

Figure 12: Sofa2MeMIComponent interface

type of this control interface is `MIComponent`. This interface however provides methods with return types that are not available on the target platform; therefore the new interface `Sofa2MeMIComponent` has been designed. This interface is shown in Figure 12.

3.4.4.2. InComponent aspect

InComponent aspect adds InComponent control interface which is used to provide the business code with access to Component control interface. Because the Component control interface type has changed the interface type for InComponent control interface also had to be adjusted. Therefore interface `MIInComponent` has been replaced by interface `Sofa2MeMIInComponent` (shown in Figure 12) in InComponent aspect for SOFA MIDlet applications.

```
public interface Sofa2MeMIInComponent {
    Sofa2MeMIComponent getComponent ();
}
```

Figure 13: Sofa2MeMIInComponent interface

Implementation of microcomponent that provides implementation for the InComponent control interface had also to be changed. The implementation in SOFA 2 Java SE runtime is based on interceptors that intercept calls to the business interfaces of the component and store the reference to InComponent of the currently entering component in variable that is local for each thread using methods of class `SOFAThreadHelper`. However the target platform lacks the built-in support for variables local for particular thread therefore new threading model had to be designed. This is discussed in chapter 3.4.5.

3.4.4.3. Lifecycle aspect

Lifecycle aspect adds Lifecycle control interface which is used to control the lifecycle of the component at runtime. The interface type for the control interface had to be changed from `MILifecycle` to `Sofa2MeMILifecycle` because of the changes made in the lifecycle

```
public interface Sofa2MeMILifecycle {
    boolean canStart();
    void preStart();
    void start ();
    void stop ();
    boolean exit (boolean unconditional);
    void waitStopped();
    int getState();
}
```

Figure 14: Sofa2MeMILifecycle interface

of the components (discussed in chapter 3.4.6). The implementation of lifecycle aspect exploits the interceptors on business interfaces to decide whether the calling code is allowed to enter the component according to the state of the component and call context of the calling thread (as described in chapter 2.2.2). However as in case of implementation of InComponent

control interface the lack of thread local storage raises the need for different implementation which is described in chapter 3.4.5.

3.4.5. Threading model

In chapters 3.4.4.2 and 3.4.4.3 it was mentioned that implementation of InComponent and Lifecycle aspects rely on storing references to InComponent interfaces and thread context in variables local for each thread. Implementation of this functionality in class `SOFAThreadHelper` exploits the class `InheritableThreadLocal` which is used to declare variables that is local to each thread and the value of the local variable of child thread can be derived from the value of thread local variable in parent thread. However the class `InheritableThreadLocal` is not available on target platform therefore the current implementation cannot be simply reused for SOFA MIDlet applications. The idea of using thread local variables elegantly fulfills requirements posed by the desired functionality of InComponent and Lifecycle aspects therefore it is exploited also in the implementation for SOFA MIDlets just it had to be implemented manually.

For the purpose of managing thread local variables the new class `Sofa2MeThreadHelper` (depicted on Figure 15) has been designed.

```
public class Sofa2MeThreadHelper {
    public static  void setInComponent(
        Sofa2MeMIInComponent inComponent) { ... }
    public static Sofa2MeMIInComponent getInComponent() { ... }
    public static Thread createThread(Runnable r) { ... }
    public static String getCallContext() { ... }
}
```

Figure 15: Sofa2MeThreadHelper class (public methods)

This class implements thread local storage by using mappings in which the thread serves as a key for which the particular value is retrieved. These mappings may grow as the threads are created and destroyed therefore the mapping structure has to be checked whether it contains any dead threads and mapping for these dead threads has to be removed in order to allow the threads to be disposed. Other problem is that the values of the thread local variables should be inherited between parent/child threads. This is done automatically when using `InheritableThreadLocal`; however the `Sofa2MeThreadHelper` has to implement this manually. The relationship between threads cannot be easily determined at runtime therefore the `Sofa2MeThreadHelper` has to be somehow notified of relationship between the threads. Therefore the `Sofa2MeThreadHelper` provides factory method `createThread(…)` which should be used for creating new thread instead of common usage of constructor. In this method the values of the thread local variables for the newly created thread are derived from the values local for the current thread that issued the call to the factory method. Other possible solution would be to use AspectJ [**32**] and define advice for the constructor of the thread class that would do the same as the factory method. However AspectJ is not officially supported on the target platform and it would require additional AspectJ runtime to be packed with the application. Therefore the factory method has been chosen as the solution for the problem.

3.4.6. Lifecycle of SOFA MIDlet

The SOFA 2 defines states in which the component can reside at runtime. On the other hand the Java ME provides its own lifecycle model for MIDlets therefore for the purpose of SOFA MIDlets new lifecycle model should be designed. This new model should not bring any overhead to components that want to follow the common SOFA lifecycle model and simultaneously provide the possibility to react to state changes of MIDlets to those components that want to be able to react to the lifecycle related changes. Main differences in lifecycle models are:

a) SOFA lifecycle has no destroyed state and there is no possibility to notify the component that the whole application is about to exit.

b) SOFA lifecycle does not allow components to refuse the state transition as is possible in MIDlet applications.

In SOFA Java SE implementation the component enters stopped state when it should exit therefore there is no need to define new SOFA lifecycle state. When component enters state stopping in fact it is still active and when the component is in state starting it is not really running yet. Therefore mapping of SOFA lifecycle states to MIDlet states depicted on Figure 16 can be defined.

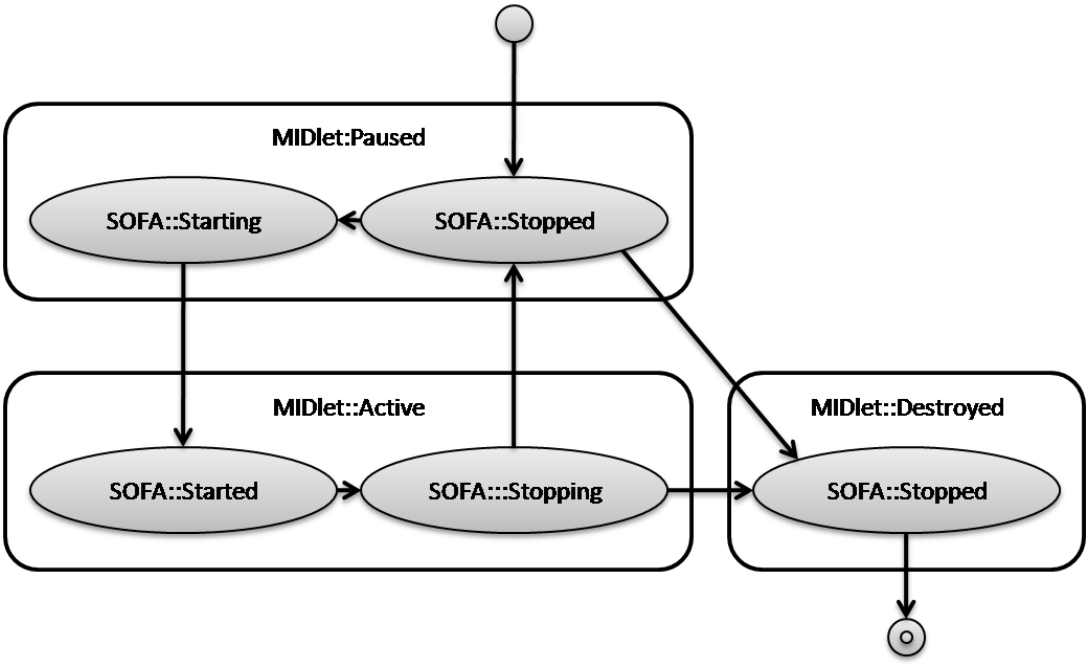


Figure 16: Mapping of SOFA lifecycle states to MIDlet states

When the SOFA MIDlet application is going to be destroyed the components have to be notified that they should exit not just stop and they also have to be able to determine if the destroy process can be rejected. The component also has to be able to tell the runtime if it rejects to exit. For this purpose the method `exit(…)` (see Figure 14) of Lifecycle control interface has been modified – it takes boolean parameter that determines whether the component can refuse to exit and returns boolean value that indicates whether the component rejects to exit or not. Also the semantics of the method slightly changed – originally the

method should not block and additional call to `waitStopped` was needed to complete the exiting process. In implementation for SOFA MIDlet this method incorporates the waiting for the component to stop therefore it can block. When the component refuses to exit, it can continue doing its work; however it cannot rely on other components to be working. This is because the behavior of AMS in case the MIDlet refuses to destroy itself is defined vaguely. The AMS may or may not postpone the destroying process and the components cannot be sure whether they can accept more business calls because the destroy process will be postponed or they have to do any necessary cleanup and quit.

Therefore when the component application is going to be destroyed all components should exit and if some of them require any additional time to complete their work they cannot do calls to any other components because they may be exited. For the purpose of notifying the business code of the components about the exiting process the interface `SOFALifecycle` has been extended into new interface `Sofa2MeLifecycle` that provides complete lifecycle-related notifications for the business code. This interface is shown in Figure 17. The business code is notified through method `destroy(…)`.

```
public interface Sofa2MeLifecycle extends SOFALifecycle {
    boolean canStart();
    boolean destroy(boolean unconditional);
}
```

Figure 17: Sofa2MeLifecycle interface

The MIDlet application is able to postpone its start therefore the SOFA MIDlet should also be able to do the same. Therefore new method `canStart()` has been added both to the Lifecycle control interface and `Sofa2MeLifecycle` interface. Prior to starting whole SOFA MIDlet these methods are called on each component and the component returns boolean value indicating whether it is ready to start or not. If any of the components is not ready to start, the starting process is aborted and the AMS can initiate the starting process later. In comparison to common SOFA 2 application the starting process itself is divided into two parts – first all components are moved to starting state and then the components are started. In common SOFA 2 application the component is moved to starting state just prior to starting it, therefore some components can be in stopped state when other components have been started already. This is undesirable (details will be described later in this chapter) and the separation of the starting process in two parts ensures that all components are at least in starting state when particular component is moved to started state. For this purpose the Lifecycle control interface defines new method `preStart()`.

Process of stopping components remains the same as in Java SE implementation. The components are first moved to stopping state and then to stopped.

One problem is hidden in the current implementation of component stopping/exiting process in SOFA 2 Java SE runtime. The lifecycle aspect defines interceptor that intercepts all calls to business interfaces and depending on the state of the component the call is blocked or let in. In current implementation however the calling thread can stick in this approval process waiting for the component to stop or it can just ignore that the component is exiting and

continue into component's code. Both situations are undesirable in Java ME environment. First the calling thread cannot be blocked on call to business interface because during stopping process all resources (including threads) should be released and second if the thread continues into the exited component the component may not be properly initialized because it has already released all acquired resources. Therefore the calling thread has to be notified immediately that it cannot continue into the component's code. This is done by throwing newly introduced runtime exception `SOFAInterruptedException` which is thrown when the called component is stopped or exited. The caller has to handle this exception – usually it indicates that the caller itself will be stopped soon. This is the reason why the starting process has to be separated in two steps – inconsistency in component states would cause needless exceptions indicating that the component is stopped, using the starting state this can be avoided.

The MIDlet application is also able to shutdown or pause itself. The possibility of shutting down the whole application is already present in SOFA using interfaces SOFASelfShutting and SOFAShutdownContext. The implementation of the shutdown functionality has been extended with notification for the AMS that the application can be destroyed. In order to maintain consistency the similar approach has been chosen for the self pausing functionality. New interfaces SOFASelfPausing and SOFAPauseContext have been designed. Both are depicted in Figure 18.

```
public interface SOFASelfPausing {
    public void setPausingContext(SOFAPauseContext context);
}
public interface SOFAPauseContext {
    public void pauseApp();
}
```

Figure 18: Interfaces SOFASelfPausing and SOFAPauseContext

The last part of self-managing of application lifecycle is the request to start the application when it is in stopped state. This can be done from some callbacks or by specifying some push registration to react to network connections. In the first case the method `resumeRequest()` of main MIDlet class is used. However this cannot be called directly, therefore helper class `Sofa2MeMidletHelper` has been designed which provides wrapper for the call of this method. This class is described in chapter 3.4.8. In the second case the push registration is done in the jad file and the name of the main MIDlet class is needed therefore this class has always the same, fixed name:

`org.objectweb.dsrg.sofa.microedition.midlet.Sofa2MeApplication`

This can be used in all cases when the name of main MIDlet class is needed.

Features described in this chapter cover the requirements a) and b) proposed in chapter 3.3.2.

3.4.7. Connector management

The SOFA MIDlet package has to contain the code of the connectors and also the infrastructure for managing (instantiating and connecting) them. This infrastructure is in fact defined by the architecture of connectors and connector generation tool which is described in [**33**]. Therefore changes in the infrastructure would pose changes in the generation tool which is out of scope of this thesis. The whole connector management infrastructure has been therefore rewritten to be able to run on target platform. This includes mainly removing any RMI – related functionality and features from higher versions of Java language (particularly generics). Most significant change is the way the connector units are instantiated in dock connector manager. Java SE implementation uses reflection to create connector unit object which is not available on target platform. Therefore factory class `ConnectorInstanceProvider` (shown in Figure 19) has been designed.

```
public class ConnectorInstanceProvider {
    public static Object getInstanceForArtifact(
          String artifactLocation,
            ConnectorUnit parentUnit,
            boolean isTopLevel) throws ElementLinkException  { ... }
}
```

Figure 19: ConnectorInstanceProvider class

Method getInstanceForArtifact(…) takes the name of the connector unit implementation class as a parameter and then instantiates the particular class. The content of this class is generated during the creation of SOFA MIDlet package (see chapter 3.5.1).

During the porting process of connector infrastructure one problem has risen. The generated connectors use for - each loops and class `StringBuilder` which were introduced in Java 1.5 and therefore they are not available on target platform. Therefore templates for generated connectors had to be adjusted. The for - each loops has been replaced by indexed loops and the usage of `StringBuilder` class has been replaced by `StringBuffer` class.

3.4.8. Access to methods and features of main MIDlet class

Main class of the MIDlet applications provides several useful methods and features that should be accessible through the whole SOFA MIDlet application. However the components do not have direct access to the main MIDlet class therefore they cannot exploit these methods. One possible solution is to pass reference to the MIDlet class to each component, or make it publicly available in some static field. This approach would however allow the business code to directly call methods that it is not supposed to call (for example methods related to lifecycle state changes). Because of this the different approach has been chosen. The methods provided by MIDlet class can be used across whole application therefore reference passing similar to the passing of shut down context is unwanted overhead. Therefore all useful methods of MIDlet class are available through the static wrapper class `Sofa2MeMidletHelper` which is shown in Figure 20.

```
public class Sofa2MeMidletHelper {
    public static void init(Sofa2MeMidlet midlet){ ... }
    public static int checkPermission(String permission){ ... }
    public static boolean platformRequest(String URL)
        throws SOFAException{ ... }
    public static Object getDisplay(){ ... }
    public static void resumeRequest(){ ... }
}
```

Figure 20: Sofa2MeMidletHelper class

Method checkPermission(…) allows checking whether the application has been assigned the particular permission which fulfills requirement d) from chapter 3.3.2. Method platformRequest(…) launches web browser which fulfills requirement g) from the same chapter and method getDisplay() returns instance of `Display` class needed to draw the graphical user interface which fulfills requirement from point c). MIDlet class also provides methods for reading the values of attributes from jad file. However this way of MIDlet parameterization is not supported, the developer should use SOFA parameterization instead (described in chapter 3.4.2).

3.5. Creating the SOFA MIDlet package

In chapter 3.3 the architecture of SOFA MIDlet has been proposed. This chapter describes the process of creating the SOFA MIDlet package and related issues.

3.5.1. Generating the component classes

In chapter 3.3.1 the approach to process of instantiating components has been proposed. For each component separate class should be generated that takes care of setting up the whole component. There are two possible solutions to code generation for these classes, either to generate directly bytecode and store it in class files or to generate the Java source code and then compile it into the class files. For the purpose of generating bytecode the ASM framework can be facilitated; generating the Java source code can be done using some kind of templating engine. In comparison to source code approach the bytecode approach does not need the compilation step and the generated classes can be used right away; however it is not easily human – readable. The ability to inspect the generated code is helpful during the development and debugging process therefore the templating approach has been chosen. The Velocity [**34**] has been chosen as the templating engine for generating the Java source code because it has already been used by SOFA 2 implementation.

The data for the template are gathered in process similar to instantiating the component in deployment dock because the generated class should have merely the same functionality as the component instance in deployment dock. The deployment plan is traversed and the data are extracted from the defining frames and architectures of the components.

The architecture of the generated classes is similar to the architecture of class `ComponentInstance` from SOFA 2 Java implementation. The class has to initialize its

business interfaces and corresponding delegation chains, instantiate implementing classes (in case of primitive components), resolve the applied aspects and initialize the control interfaces and microcomponents, bind the microcomponents in delegation chains and components to each other. Therefore frames are searched for business interfaces and the generated class is filled with information which interfaces should be initialized, the aspects are inspected to generate the control interfaces initialization, component and interface selects are evaluated in order to gather information about which microcomponents should be instantiated, the bindings among the microcomponents are resolved and appropriate code is filled in the generated class. Also the properties for the components are retrieved and inserted into the generated code. The name of the generated class is derived from the name of the architecture and component's hierarchical name which is concatenation of component names of all parent components. All generated component classes inherit from the base class `Sofa2MeComponentBase` which implements `Sofa2MeInternalComponentDescription` interface derived from interface `Sofa2MeComponentDescription`. This class contains functionality common to all components, particularly methods for managing component lifecycle. These lifecycle - related methods take care of lifecycle state change handling of the component itself (through its Lifecycle control interface) and also of all subcomponents. The main MIDlet class has to keep the reference to the root component and call appropriate methods every time the lifecycle state change is needed.

In addition to the classes representing components two more classes are generated; the main MIDlet class `Sofa2MeApplication` and the `ConnectorInstanceProvider` class. The main MIDlet class has to know the name of the class generated for the root component and the `ConnectorInstanceProvider` has to be filled with names of the classes that need to be instantiated in order to build connectors.

During the generation process all entities for which the implementation in form of code bundles is needed are identified. These code bundles need some additional processing which is described in chapter 3.5.2.

3.5.2. Code bundle management

The MIDlet package has to contain the code for implementation of primitive components, connectors and microcomponents. This code is stored in code bundles in the repository and particular code bundles that have to be packed into the MIDlet package are identified during the generation of component classes. Ideally the class files would be extracted from code bundle's jar file and packed in the MIDlet package. However this is not possible in case of connectors because their class files have version 50 which cannot be loaded in Java ME virtual machine. In addition to this it would be useful if some of the components from SOFA 2 Java SE could be reused in SOFA MIDlets. Code of such components also may not have the proper version (or it can be incompatible with target platform in case the developer specified wrong implementation). This could be solved by allowing entities to be associated with multiple code bundles. However this would pose significant changes to SOFA 2 component model and Java SE runtime which is out of scope of this thesis. The code bundles contain not only the compiled class files but also the source

code. This can be exploited to solve the problem with incorrect classes because the new, proper class can be compiled. Therefore during the creation of MIDlet package the source code is extracted from code bundles and compiled. However the compiled code does not reflect the version of code bundle and multiple classes with same name but different version are not able to coexist in one MIDlet. Therefore the renaming process has to be performed as in SOFA 2 Java SE implementation.

The possibility of reusing the components and microcomponents is the reason why the aspect entity is decorated with platform attribute. The generator of component classes has to determine whether the aspect should be applied. This cannot be done using the language attribute of corresponding code bundles, because by default the attempt to recompile the code bundle is done and it can fail because the applied aspect is not supposed to be applied on SOFA MIDlet application.

Generally, the code of microcomponents, connectors, business interfaces and primitive component implementation has to be recompiled. The only exceptions are microcomponents that generate other microcomponents. The generation process takes place during the MIDlet package assembling and these microcomponents are not recompiled; only the generated microcomponents are packed into the MIDlet package. There is one limitation for the current interceptor generators – the interface for which the interceptors should be generated should not contain any Java ME specific classes as the generator uses reflection to traverse all methods of the interface and therefore it needs to load the class into virtual machine which fails if the interface references any Java ME specifics.

3.5.3. Assembling of the MIDlet package

To assemble the MIDlet package first the component classes have to be generated. Then these classes have to be compiled together with SOFA MIDlet infrastructure code (which includes classes for connector management, component API etc.) and code of connectors, microcomponents and primitive components. The source code has to be compiled against Java bootstrap jar files. The resulting class files are renamed. Then the preverification has to take place. The preverified class files are packed into the jar file and the jad file is generated. The SOFA MIDlet is now ready to be deployed to target device.

3.6. Development tools

The development process of SOFA 2 application is supported by the command line tool called cushion and plugin for Eclipse IDE called SOFA IDE. These tools allow developer to create new entities (architectures, frames etc.) in repository, download existing entities, made local changes in entities and later upload modified entities back to repository. As mentioned in chapter 2.2.3 the process of developing SOFA MIDlet applications has been changed slightly therefore the development tools also need to be adjusted. For the purpose of this thesis only the command line tools should be modified, adding SOFA MIDlet support to SOFA IDE is out of scope of this work. However this is not a complex task because the implementation of all the features provided by the development tools is shared by both

cushion and SOFA IDE and it is called through common API; just the frontend is different. The implementation of Java ME – related functionality has also been incorporated into this common API therefore in order to add support for SOFA MIDlets into SOFA IDE the user interface for the already implemented actions has to be created.

3.6.1. Cushion

The cushion is command line tool used to develop SOFA applications. It gives the developer the possibility to create his own working copy of entities in one of his local folders. This local copy is some kind of workspace which contains the entities on which the developer currently works and the configuration file for cushion. Cushion provides set of actions the developer can perform on entities in his workspace or the repository. The general pattern of calling cushion is depicted on Figure 21.

```
cushion <action> [params…]
```

Figure 21: General usage of cushion

The action denotes the action that should be performed, for example "compile" to compile the code of entities, "new" to create new entity or "commit" to upload modified entity to repository. These actions may take optional parameters, for example name, version or tag of the entity on which the action should be performed. The configuration file contains additional general parameters used across all calls to cushion actions.

To allow the developer to develop the SOFA MIDlet applications the cushion has to provide:

a) The ability to compile the code for entities against Java bootstrap jar files.

b) The ability to use optional packages during compilation (as discussed in chapter 3.3.2 point h)).

c) The ability to generate MIDlet package for SOFA MIDlet application.

The point a) implies that the cushion has to be able to determine against which bootstrap jar files it should compile the code of entities. This is already present in cushion because the configuration file contains the language attribute. This attribute needs to be filled with proper value during the initialization of the work space. The first point also implies that the cushion has to know the path to the bootstrap jar files. The point b) implies that during the compilation the jar files with optional packages has to be added to classpath and therefore the cushion has to be aware of location of these jar files. Neither of the last two mentioned functionalities is currently supported in cushion. During the generation of SOFA MIDlet package the generation process has to preverify the class files therefore also the path to preverification tool has to be set. Neither this is currently possible in cushion. Setting up these parameters for each call of the action that needs the paths is tedious because the paths can be long. Therefore they should be stored in the configuration file and this configuration file needs to be properly initialized. The initialization of the configuration file (and the whole workspace) is done using action `init`. This action has been extended with parameters for defining the paths to

bootstraps jar files; optional packages jar files and preverification tool. The usage of this action is depicted on Figure 22.

```
cushion init [-l <lang> [-bootclasspath <bootclasspath jars>]
    [-preverifytool <preverify tool dest>]
    [-classpath <classpath dest>] ] [<dest>]
```

Figure 22: Usage of cushion init action

This action can be called without any additional parameters. The option "-l" denotes the implementation language of components. To initialize the workspace for the development of SOFA MIDlet applications the value of this option should be "j2me". If the value of "-l" option is "j2me" the values for options "-bootclasspath" and "-preverifytool" have to be specified. Value for option "-bootclasspath" should be filled with classpath containing the bootstrap jar files and the value for option "-preverifytool" should contain the path to the preverification tool executable. The next option "-classpath" is not mandatory, it should be used to specify paths to jar files of optional packages. Neither of these paths should contain whitespaces. Last parameter is the directory where the workspace should be created.

The point c) requires functionality that is not currently present in cushion and it is not related to any existing actions. Therefore the best solution is to define new action. This new action is called "midlet" and its usage is depicted on Figure 23.

```
cushion midlet <deplplan-name> [--|<tag>|-v <version>]
```

Figure 23: Usage of cushion midlet action

This action takes name of the deployment plan as mandatory parameter and two optional parameters version and tag as is common in other actions that work over particular entities. Calling this action requires running repository because the deployment plan has to be retrieved from it. When the action is called the SOFA MIDlet package consisting of jar and jad file is created in the workspace. This package can be then deployed into the target device however this functionality is not covered by cushion because it depends on the chosen deployment method and target device.

There may be some problems when this action is triggered in a workspace located in a deeply nested directory. This is because the names of the generated interceptors together with the long path to the workspace directory exceed the maximal path length on the development machine's operating system and preverification tool is not able to load them.

4. Sample application

In this chapter the design and development process of real – life SOFA MIDlet application is described in order to demonstrate how the component - based development can be incorporated in the process of developing applications for Java ME and how this can be done using SOFA 2 component system.

4.1. SofaWorm

For the purpose of demonstrating basic features of SOFA MIDlet application the implementation of game present on almost all mobile phones has been chosen. It is the game Worm and since it is developed using SOFA the application is called SofaWorm. In this game the player takes control over small worm which is constantly moving forward in bordered game area where food for worm is randomly placed. The player navigates the worm so it would find the food and simultaneously would not crash into the borders of the game area. The game provides the possibility to restart, pause, resume or exit the application.

The game demonstrates some of the features of SOFA MIDlet applications – new threading model, usage of user interface and setting up attributes for jad file.

4.2. Application architecture

The architecture follows simplified common design pattern Model - View – Controller (MVC) [**35**]. In this design pattern the view represents the user interface through which the user can interact with application. All actions the user performs using the user interface are forwarded to the controller part which implements the business logic. When needed the controller modifies the data of the application represented by the model part. When change is issued on model, the view is notified so it can update what the user can see according to the data of the model.

In SofaWorm the view takes care of displaying the game area and food, score, menu for pausing, resuming restarting and exiting the game. It also takes care of handling user input – changes of direction the worm is heading and picking various actions using menu. The controller is notified if the user changes the direction or picks some action from the menu. It also issues periodic updates of the data of the worm in model which causes the worm to move on the screen. The model holds the data of the game – the position of the worm and food. When the data changes, the view is notified it should update itself.

In the terminology of component applications the particular parts that form the MVC pattern can be viewed as components. It is therefore appropriate to design the components that fit the particular parts of the MVC pattern. The whole application is represented by the frame WormGame and architecture WormGame. This top – level architecture contains primitive components that implement the particular parts of MVC pattern – view part is defined by frame WormView and architecture WormView, controller is defined by frame WormController and architecture WormController and finally frame WormModel and

architecture WormModel define the model part of the MVC pattern. The whole situation is depicted on Figure 24.

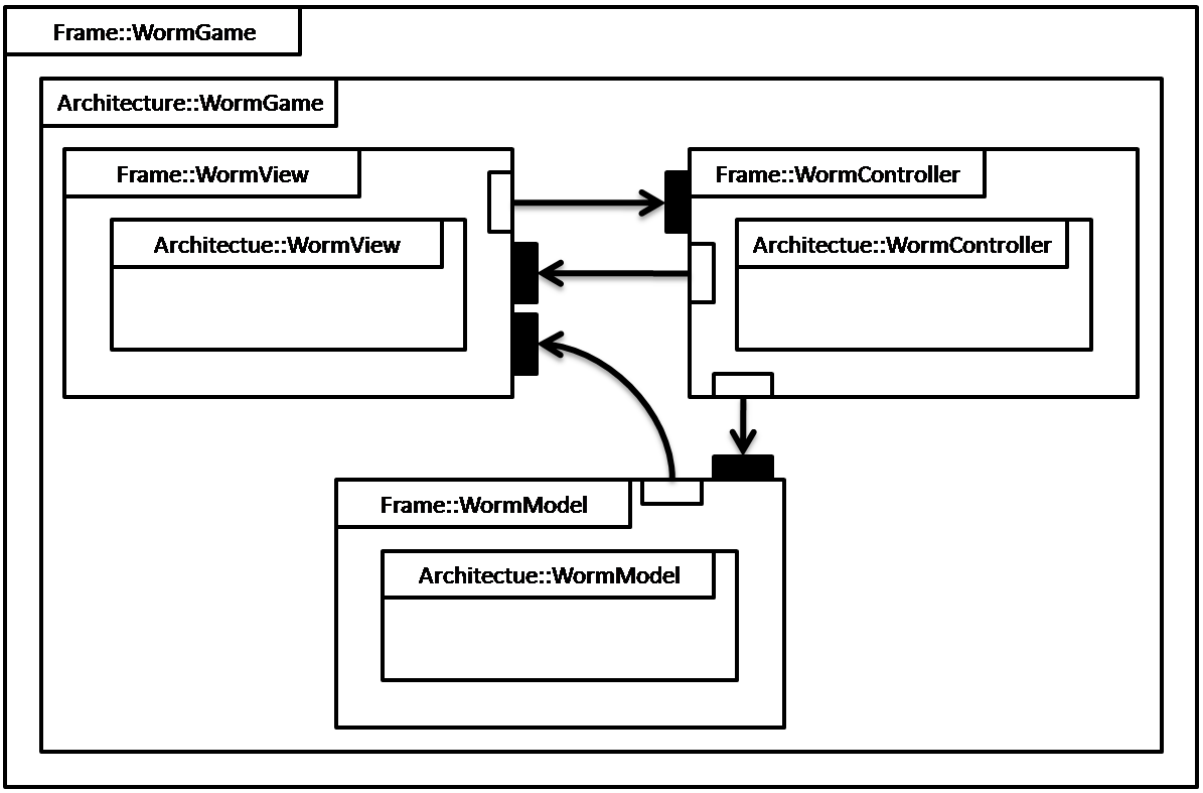


Figure 24: Architecture of SofaWorm game

The component communication scheme also follows the MVC pattern. The WormController provides interface that the WormView can use to notify the controller about direction changes and requires interfaces for updating the model and notifying the view of changes caused by user input. The WormView frame requires the interface through which it can notify the controller about direction changes and other user actions and provides interfaces though which it can be notified of state changes and changes in the position of the worm and food. The WormModel frame requires the interface for notifying the view of updates of application data and provides interface through which the controller can update the worm's position.

4.3. Implementation of SofaWorm

The application architecture has been designed and the next step is to transform the model into SOFA 2 frames and architectures. For this purpose the command line tool cushion is used. To develop SofaWorm application the SOFA 2 and cushion have to be installed properly.

The following text does not describe complete process of developing SofaWorm game nor contains all the ADL and source code. It rather pin – points the most significant parts of implementation with emphasis on Java ME specifics. The step - by - step walkthrough is presented in user's guide which can be found on accompanied CD together with complete source code of the application.

4.3.1. Development prerequisites

In order to develop the SofaWorm game the workspace has to be initialized for Java ME development.

This is done by calling `init` action of the cushion tool. The usage of the init action is:

```
cushion init -l j2me -bootclasspath <pt1> -preverifytool <pt2> <pt3>
```

The value <pt1> should contain the classpath that contains the locations of Java bootstrap jar files. Particular jar files have to listed, not just directory. The classpath must not contain whitespace characters. The bootclasspath can for example look like this (on Windows systems):

```
c:\Java_ME_platform_SDK_3.0\lib\cldc_1.1.jar;c:\Java_ME_platform_SDK
_3.0\lib\midp_2.1.jar
```

The value <pt2> should contain the path to the preverification tool executable. Again, the path must not contain whitespace characters. The path can for example look like this (on Windows systems):

```
c:\Java_ME_platform_SDK_3.0\bin\preverify.exe
```

The value <pt3> denotes the local folder in which the workspace should be created.

This initialization is vital to the SOFA MIDlet development because the source code has to be cross – compiled against different bootstrap jar files and preverified. When the language is set to "j2me" cushion compiles all source code of entities against bootstrap jars specified as a parameter.

The development process aided by the cushion also needs running repository. This is the same as when developing common SOFA 2 applications.

4.3.2. Interfaces

The interfaces through which the components should communicate have to be defined.

4.3.2.1. WormView interface

The interface WormView contains two methods the model component WormModel should use to notify the view about the changes in its data. Method `updatePlayground(…)` notifies the view that the position of worm and food has changed and the method `gameOver()` tells the view that the worm crashed into borders and the game is over. The interface should be provided by the WormView component.

```
public interface WormView {
    void updatePlayground(Worm worm,Food food,int score);
    void gameOver();
}
```

Figure 25: WormView interface provided by WormView component

4.3.2.2. WormViewUtility

This interface contains method that the controller can use to tell the view that it should notify the user that the game is paused. It should be provided by the WormView component.

```
public interface WormViewUtility {
      void gamePaused();
}
```

Figure 26: WormViewUtility interface provided by WormView component

4.3.2.3. WormController interface

This interface provides methods the WormView component should call to notify the controller of user actions. The methods `pausedGame()`, `resumeGame()`, `newGame()` and `exitGame()` are used to notify the controller that the user picked some action from the menu. The methods `gameHidden()` and `gameShown()` notify the controller that the worm should stop moving as the game area was hidden because the menu is shown. The method `init(…)` is used to initialize the whole game – the view can determine the size of the screen and compute the size of game area and this information has to be passed to the model so it can be properly initialized. Last method `changeDirection(…)` is used when user issues the change of direction. This interface should be provided by WormController component.

```
public interface WormController {
    void pauseGame();
    void resumeGame();
    void newGame();
    void exitGame();
    void gameHidden();
    void gameShown();
    void init(int playgroundWidth,int playgroundHeight);
    void changeDirection(int direction);
}
```

Figure 27: WormController interface provided by WormController component

4.3.2.4. WormModel interface

This interface provides functionality for initializing the game area, restarting the whole game and moving worm in given direction. It should be implemented by WormModel component.

```
public interface WormModel {
    void initWorm(int playgroundWidth,int playgroundHeight);
    void restart();
    void moveWorm(int direction);
}
```

Figure 28: WormModel interface provided by WormModel component

4.3.3. Frames

SofaWorm game is composed of four components – the top – level component for whole game and three primitive components. Therefore the definition of application consists of four frames:

- WormView – defines the view component, provides the interfaces "view" of type `WormView` and "viewUtility" of type `WormViewUtility`, requires interface "controller" of type `WormController`.
- WormController – defines the controller component, provides interface "controller" of type WormController and requires interfaces "model" of type `WormModel` and "viewUtility" of type `WormViewUtility`.
- WormModel - defines the model component, provides interface "model" of type `WormModel` and requires interface "view" of type `WormView`.
- WormGame – top – level frame that defines whole application.

4.3.4. Architectures

Each of the four frames has to be implemented by particular architecture.

4.3.4.1. WormView architecture

Primitive architecture for the view component. It defines the property "wormColor" that should be filled later in the deployment plan in order to demonstrate parameterization of components. The implementing class `WormViewImpl` implements interfaces `WormView` and `WormViewUtility` in order to provide the implementation for provided business interfaces of the view component. It also implements the interface `SOFAClient` so it is able to obtain references to required interfaces and `Sofa2MeParametrized` so the wormColor parameter can be passed into the component. It implements interface `SOFARunnable` because it has to launch thread that initializes the whole application. For the purpose of creating the thread the factory method of class `Sofa2MeThreadHelper` is used. Since the view has to draw the user interface it needs access to the instance of `Display` class. This is obtained using `Sofa2MeMidletHelper` class.

4.3.4.2. WormController architecture

Primitive architecture for the controller component. The class `WormControllerImpl` provides implementation for this primitive architecture. The class implements interface WormController to implement functionality for provided interface of the component, interface SOFAClient to obtain references to required interfaces, interface SOFASelfShutting so the controller can exit the whole application and finally the Sofa2MeLifecycle interface in order to be able to respond to state changes in MIDlet's lifecycle. The component is driven by a background thread that periodically updates the model to move the worm. When the whole application is paused or exited this thread has to be stopped and disposed. This demonstrates the adaptation of SOFA 2 lifecycle to MIDlet lifecycle.

4.3.4.3. WormModel architecture

Primitive architecture for the controller component. The implementing class `WormModelImpl` implements interface `WormModel` to implement functionality for the component's provided interface and interface `SOFAClient` in order to obtain reference to required interface.

4.3.4.4. WormGame architecture

Composed architecture that consists of view, model and controller components defined by architectures WormView, WormModel and WormController. It defines connections between the business interfaces of the components according to Figure 24.

4.3.5. Deployment plan

All the components have to be assigned to deployment docks. When developing the SOFA MIDlet applications the deployment plan has to be local – all components have to be assigned to the same deployment dock. Also the values of properties for components and deployment plan properties that should be passed to jad file as attributes should be specified. The example of such deployment plan is on Figure 29. In this deployment plan the value for property wormColor is set to red (FF0000) and various attributes for jad file are set. For example the name of the MIDlet suite is SofaWorm, the minimal configuration has to be CLDC 1.1 and required profile is MIDP 2.0.

```
<?xml version="1.0" encoding="UTF-8"?>
<depl-plan name="sofaworm.deplplan.WormGame" node="nodeA">
  <deplplan-prop name="sofa2Me.midlet.attribute">
    MIDlet-Name|SofaWorm</deplplan-prop>
  <deplplan-prop name="sofa2Me.midlet.attribute">
    MIDlet-Vendor|Jaroslav Pastorek</deplplan-prop>
 <deplplan-prop name="sofa2Me.midlet.attribute">
    MIDlet-Icon|org/sofa-icon.png</deplplan-prop>
 <deplplan-prop name="sofa2Me.midlet.attribute">MIDlet-Version|1.0.0</deplplan-prop>
 <deplplan-prop name="sofa2Me.midlet.attribute">
    MicroEdition-Configuration|CLDC-1.1</deplplan-prop>
 <deplplan-prop name="sofa2Me.midlet.attribute">
    MicroEdition-Profile|MIDP-2.0</deplplan-prop>
 <deplplan-prop name="sofa2Me.midlet.attribute">
    MIDlet-Jar-URL|SofaWorm.jar</deplplan-prop>
 <deplplan-prop name="sofa2Me.midlet.attribute">
    MIDlet-Midlet-Icon|org/sofa-icon.png</deplplan-prop>
 <depl-subc name="view" node="nodeA">
  <depl-prop-value name="wormColor">FF0000</depl-prop-value>
 </depl-subc>
 <depl-subc name="model" node="nodeA" />
 <depl-subc name="controller" node="nodeA" />
</depl-plan>
```

Figure 29: Deployment plan for SofaWorm game

4.3.6. Creating SofaWorm MIDlet package

Finally the MIDlet package can be created for the SofaWorm game. For this purpose the cushion's `midlet` action has to be performed:

```
cushion midlet sofaworm.deplplan.WormGame
```

This creates files SofaWorm.jar and SofaWorm.jad which can be immediately deployed to target device.

When the SofaWorm game is launched on the target device should look similar to the Figure 30.

Figure 30: SofaWorm game deployed on mobile phone

4.4. Summary of developing SofaWorm

The development process of SofaWorm game proved to be similar to process of developing common SOFA 2 applications however there are some differences. The first difference is the mandatory initialization step at the beginning and the creating of the MIDlet package at the very end. However the developer has to be aware of differences between the Java SE and Java ME and to fully exploit the features of Java ME also the SOFA MIDlet specifics have to be used.

5. Related work

The idea of exploiting component based development in the field of embedded applications is not new. Several attempts were made either to use component based approach during the development process of embedded application or to create component systems that provide mechanisms for creating embedded component software. The example of the first approach is the component based Java virtual machine for embedded device written in C language proposed in [**36**]; second approach is realized in MIND [**37**] which a pure C implementation of Fractal component model (which is discussed later in this chapter) for embedded devices. In following text attempts to use component based development in Java ME environment are described and compared with SOFA 2 implementation.

In [**38**] the component based framework for creating GUI is presented. This framework provides higher level API for creating GUI components in Java ME, particularly MIDP profile. The UML Components [**39**] model is used to design the architecture of the framework; however implementation does not incorporate any component runtime.

MUSIC project [**40**] is a middleware for development and execution of self-adaptive mobile applications. The components can form nested hierarchies and are viewed as units of adaptation to changing external conditions. These conditions are described using so called *context parameters* and the changes of these parameters are detected by *sensors*. Each change of context parameter issues the adaptation of particular component to current conditions. Component model therefore presumes that each component has multiple implementations each of which fulfils different needs. The components communicate through *ports* decorated by *parameters* that define the conditions for which the component has been designed. During the adaptation process the decision can be made according to these properties. Although MUSIC primarily targets mobile devices the minimal requirements of reference implementation include virtual machine compatible with Java 1.4. Nevertheless port to CDC configuration has been developed.

Draco [**41**] is component runtime implementing SEESCOA [**42**] component model. It provides flat components with provided and required *interfaces* that are associated with *ports*. These ports are connected using *connectors* and the communication is message - based. Draco itself provides infrastructure for instantiating components and ports, message ordering and delivery and connecting the components using connectors. The runtime is extensible so additional functionality can be added. One of the extensions provides the dynamic update of the components when component is replaced by another component with the same interfaces. Port to Java ME has been developed for the purpose of comparison of technologies for embedded devices [**43**]. However due to extensive use of reflection in messaging communication the target platform for the Java ME port is CDC configuration with Personal Profile.

Fractal is a language independent component model. It allows creating hierarchical components which consist of *membrane* and *component content*. Membranes allow accessing the component management functionality through set of *controllers* which provide several levels of component introspection. The components define their client and server *interfaces*

which are connected using *bindings*. These bindings are in fact special – purpose components. Fractal is highly modular and extensible and multiple levels of conformance with the full specification are defined. Julia [**44**] is reference implementation of Fractal in Java. It supports developing applications for CLDC configuration however with some restrictions. Since Julia generates code for controllers and interceptors at runtime these have to be pregenerated in less constrained environment. In addition to this the runtime of Julia itself has to be transformed into CLDC compliant form because it is not completely compatible. All these steps can be however performed automatically. AOKell [**45**] is Java implementation of Fractal that satisfies highest conformance level and its main goal is to bring component based approach to defining controllers. AOKell can be compiled for Java SE and Java ME, both CDC and CLDC configurations but no particular profile is specified. For the purpose of creating CLDC application the AOKell generates all needed infrastructure. The final application consists of AOKell infrastructure itself (which was compiled for Java ME), generated infrastructure code and business code.

Both the component based Java virtual machine and the MIND project mentioned at beginning of this chapter are written in C language and do not have the same restrictions and requirements as the Java ME environment and the comparison with SOFA MIDlet would be more the summary of differences between native C and Java ME than comparison of component systems therefore just the attempts based on Java ME are evaluated.

It is not possible to compare the library proposed in [**38**] with SOFA MIDlet because the library does not exploit any component runtime; however it is a good example of applying component principles in development for Java ME. Since both MUSIC and Draco target different platform then SOFA MIDlet it is hard to compare them. Dynamic reconfiguration is substantial for MUSIC and achieving this in CLDC configuration is not possible without extending the target platform. Draco also provides dynamic reconfiguration though it is not as important as in MUSIC. This and the fact that reflection is widely used in the implementation of the component runtime make the CDC more suitable platform for Draco.

Fractal provides component model with features similar to SOFA 2 and Fractal applications developed using Julia or AOKell are able to run under CLDC configuration which makes them the closest competitors to SOFA MIDlet. All three runtimes take similar approach to the problem of deploying component application in restricted environment – they all rely on pregeneration of all needed code on a device that is more capable than the target platform. They also share the common restrictions related to dynamic reconfiguration and component communication which are caused by CLDC configuration. However both AOKell and Julia are based on CLDC and they do not consider any particular Java ME profile. This makes them more flexible as the choice of profile is left to the developer however also the incorporation of particular profile into the component development is left up to him which may cause significant overhead. In comparison to this the SOFA MIDlet is constrained to develop just MIDlet applications on the other hand it provides the developer with complete development process for creating full – featured applications on top of MIDP profile.

6. Conclusion

In this thesis the possibilities of deploying the component applications developed in SOFA 2 component system in restricted environment have been explored. The possible target platforms have been evaluated and the Java ME, particularly Connected Limited Device Configuration with Mobile Information Device Profile, has been chosen. The restrictions posed by target platform have been recognized and analyzed in order to identify features of SOFA 2 applications that are not feasible on target platform. Therefore definition of restricted SOFA 2 application suitable for Java ME has been proposed. This definitions requires component application to be non – distributed with static architecture, method invocation – based communication and implementation conforming to Java language version 1.3 in order to be deployable on target platform. According to the characteristics of the target profile the MIDlet model has been chosen for runtime representation of component application. Based on chosen application model the further requirements for full – featured development of MIDlet applications using SOFA 2 components have been identified. Automatic transformation of component application description stored in repository to standalone MIDlet application has been chosen as the solution for the problem of deployment in restricted environment. The component runtime and component API have been adapted in order to obey the restrictions on version of Java language and to reflect the restrictions of target platform. New features have been added to component API to allow development of full – fledged MIDlets.

Small changes have been introduced into SOFA 2 component model in order to distinguish between different platforms where the component application can be deployed and to allow parameterization of particular deployment packages.

The development process of SOFA 2 applications has been augmented with steps required for the development of SOFA MIDlet applications, particularly cross - compilation of code of entities for Java ME and process of transformation of component application into MIDlet. The cushion development tool has been adjusted to cover the changes in the development process.

The demonstrative application SofaWorm has been developed as a proof of concept for chosen approach. Being a common game it presents some specific features of component development for Java ME.

The current implementation however leaves some place for further improvements. The restriction on application to be non – distributed is very limiting; however designing the deployment and communication strategy for components located on different mobile devices is a complex task. Another possible improvement is the support for dynamic reconfiguration using predefined patterns, especially factory pattern. And at last the reimplementation of interceptor generator that would allow defining business interfaces containing Java ME specifics should be also considered.

7. References

[1] Java Platform, Standard Edition, http://java.sun.com/javase/.

[2] Java Platform, Micro Edition, http://java.sun.com/javame/index.jsp.

[3] Sun Microsystems,Inc., Connected Limited Device Configuration (CLDC) Specification Version 1.1, March 2003, http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html.

[4] Sun Microsystems, Inc., Motorola Inc., Mobile Information Device Profile for JavaTM 2 Micro Edition Version 2.1, May 2006, http://jcp.org/aboutJava/communityprocess/mrel/jsr118/index.html.

[5] T. Bures, P. Hnetynka, and F. Plasil, "SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model," in *Proceedings of SERA 2006*, Seattle, 2006, pp. 40-48.

[6] Sun Microsystems, Inc., JavaBeans (TM) Specification, August 1997, http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp.

[7] COM: Component object model technologies, http://www.microsoft.com/com/default.mspx.

[8] Enterprise Java Beans Technology, http://java.sun.com/products/ejb/.

[9] Object Management Group, CORBA Component Model Specification, v4.0, 2006 April, http://www.omg.org/technology/documents/formal/components.htm.

[10] The Fractal Project, http://fractal.ow2.org/.

[11] SOFA, http://sofa.ow2.org/sofa1/index.html.

[12] SOFA 2.0, http://sofa.ow2.org/.

[13] Java Community Process, http://jcp.org/en/home/index.

[14] Java ME Technology - CDC, http://java.sun.com/javame/technology/cdc/index.jsp.

[15] Siemens Mobile, Nokia, Sun Microsystems, Inc., Motorola, Inc., Information Module Profile (JSR-195), 2003, http://jcp.org/aboutJava/communityprocess/final/jsr195/index.html.

[16] Foundation Profile, http://java.sun.com/products/foundation/.

[17] Personal Basis Profile, http://java.sun.com/products/personalbasis/index.jsp.

[18] Personal Profile, http://java.sun.com/products/personalprofile/index.jsp.

[19] Sun Microsystems, Inc., JavaTM APIs for BluetoothTM Wireless Technology (JSR-82), April 2002, http://jcp.org/aboutJava/communityprocess/final/jsr082/index.html.

[20] Nokia Corporation, Mobile 3D Graphics API Technical Specification, 2005 June, http://www.jcp.org/en/jsr/detail?id=184.

[21] Siemens AG, Wireless Messaging API (WMA) for JavaTM 2 Micro Edition Version 1.1, March 2003, http://jcp.org/aboutJava/communityprocess/final/jsr120/index2.html.

[22] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The JavaTM Language Specification*, Second Edition ed.: Addison-Wesley, 2000.

[23] Remote Method Invocation (RMI), http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp.

[24] Eclipse Modeling Framework Project (EMF), http://www.eclipse.org/modeling/emf/.

[25] Object Management Group, MOF 2.0 / XMI Mapping Specification, Version 2.1.1, December 2007, http://www.omg.org/spec/XMI/2.1/PDF/.

[26] P. Hnetynka and F. Plasil, "Dynamic Reconfiguration and Access to Services in Hierarchical Component Models," *Proceedings of CBSE 2006*, pp. 352-359, June 2006.

[27] Java Message Service (JMS), http://java.sun.com/products/jms/.

[28] ASM, http://asm.ow2.org/.

[29] Cushion for SOFA 2, http://sofa.ow2.org/docs/cushion.html.

[30] SOFA IDE, http://sofa.ow2.org/docs/index.html.

[31] Eclipse, http://www.eclipse.org/.

[32] The AspectJ Project, http://www.eclipse.org/aspectj/.

[33] Tomas Bures, "Generating Connectors for Homogeneous and Heterogeneous Deployment," Charles University in Prague, Faculty of Mathematics and Physics, Prague, PhD thesis 2006.

[34] The Apache Velocity Project, http://velocity.apache.org/.

[35] S. Burbeck, "Applications Programming in Smalltalk-80: How to use Model–View–Controller," 1992.

[36] Hiroo Ishikawa and Tatsuo Nakajima, "EarlGray: A Component-Based Java Virtual Machine for Embedded Systems," *Eighth IEEE International Symposium on Object-Oriented Real-Time Distrubuted Computing*, pp. 403-409, 2005.

[37] MIND, http://mind.ow2.org/index.html.

[38] L. M. Nascimento, E. S. Almeida, and S. R. Meira, "Component-Based Development in J2ME: A Framework for Graphical Interface Development in Mobile Devices," *6th Workshop on*

Component Based Development (WDBC'2006), pp. 88-95, December 2006.

[39] J. Cheesman and J. Daniels, *UML Components: A Simple Process for Specifying Component-Based Software*.: Addison-Wesley, 2001.

[40] MUSIC Project, http://www.ist-music.eu/.

[41] Yves Vandewoude et al., "Draco: An adaptive runtime environment for components," CS Dep.KULeuven, Technical Report CW372.

[42] SEESCOA, http://distrinet.cs.kuleuven.be/projects/SEESCOA/.

[43] Koen Victor, Yves Vandewoude, and Yolande Berbers, "Application Platforms for Embedded Systems: Suitability of J2ME and.NET Compact Framework," *Proceedings of the International Conference on Software Engineering Research and Practice*, pp. 367-374, 2006.

[44] Julia, http://fractal.ow2.org/julia/index.html.

[45] AOKell, http://fractal.ow2.org/tutorials/aokell/index.html.

Appendix A: Contents of the accompanied CD

This thesis is accompanied with CD with source code, binary distribution and electronic version of this thesis.

The contents of the CD:

- ***bin*** folder – contains files ***cushion.zip*** and ***sofa2.zip*** which are binary distributions of SOFA MIDlet project
- ***doc*** folder – contains user's guide and programmer's guide
- ***src*** folder – contains source code of the SOFA MIDlet project
    - ***build*** folder – contains general build definitions for the whole project
    - ***congen-core*** folder – project for connector generator
    - ***cushion*** folder – implementation of cushion development tool, in subdirectory ***dist*** the distribution of cushion is created
    - ***demo*** folder – contains source code for demo application SofaWorm and generated SOFA MIDlet package (files ***SofaWorm.jad*** and ***SofaWorm.jar***) that can be deployed directly to the target device
    - ***jdoc*** folder – contains documentation generated from source code for whole SOFA MIDlet project.
    - ***sofa*** folder – implementation of general SOFA 2 functionality
    - ***sofa-j*** – implementation of SOFA 2 for Java SE and Java SE
    - ***build.xml*** file – build file with definitions for building and cleaning the projects, generating documentation and generating the packages for distribution
- ***README.txt*** – contains information about the contents of the CD
- ***thesis.pdf*** – electronic version of this text

Appendix B: Summary of modifications caused by SOFA MIDlet

The implementation of Java ME support for SOFA 2 is spread among several SOFA 2 metaprojects and projects. In SOFA 2 the metaproject does not contain any source code; it is used to encapsulate several common projects.

The implementation of MIDlet generator, MIDlet component runtime and bootstrap aspects is a part of sofa-j metaproject and is divided into following projects:

a) sofa-microj-deployment – contains implementation of MIDlet generator.

b) sofa-microj-bootstrap – contains implementation of bootstrap aspects for Java ME environment.

c) sofa-microj-runtime – contains the component runtime, for example the infrastructure for connectors.

d) sofa-microj-api – contains interfaces for development of components and microcomponents for SOFA MIDlet applications.

Several other projects have been modified in order to allow Java ME deployment:

a) congen-core – project from metaproject congen. Modifications have been done to generate connectors compliant to Java language version 1.3.

b) cushion – development tool cushion has been modified to add development tool support for SOFA MIDlet applications.

c) sofa-j-boostrap – project from metaproject sofa-j, contains new generator of interceptor microcomponents for Java ME and upload of Java ME bootstrap aspects has been added.

d) sofa-j-dock – project from metaproject sofa-j, the deployment process has been modified to ignore aspects specific to Java ME.

e) sofa-repository – project from metaproject sofa, the component model has been adjusted to distinguish between the aspects for Java ME and Java SE applications and to allow parameterization of the deployment plan. The renaming process has also been changed.

f) sofa-tools-api – project from metaproject sofa, contains new api for generating the MIDlet. Also the compilation process has been modified.