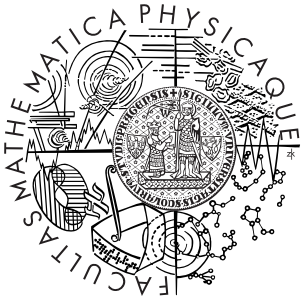
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Peter Hladký

CoCoME in SOFA

Katedra softwarového inženýrství

Vedoucí bakalářské práce: Doc. Ing. Petr Tůma, Dr.
Studijní program: Informatika

2008

Ďakujem vedúcemu bakalárskej práce Doc. Ing. Petrovi Tůmovi, Dr. za poskytnutú pomoc a cenné konzultácie počas vypracovávania zadania.

Ďalej ďakujem RNDr. Petrovi Hnětynkovi, Ph.D. a Mgr. Michalovi Malohlavovi za pomoc pri riešení problémov súvisiacich s komponentovým systémom SOFA.

Taktiež ďakujem rodičom a bratovi za ich podporu, rady a trpezlivosť.

Prehlasujem, že som svoju bakalársku prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním práce a jej uverejňovaním.

V Praze dne 27.5. 2008 Peter Hladký

Contents

Název práce: CoCoME in SOFA
Autor: Peter Hladký
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: Doc. Ing. Petr Tůma, Dr.
e-mail vedoucího: Petr.Tuma@dsrg.mff.cuni.cz

Abstrakt: Využitie hardwarových zdrojov softwarom je možné modelovať a tak predpovedať zmeny vo využití zdrojov, ktoré spôsobia zmeny v implementácii softwaru. CoCoME modeluje reálny obchodný systém. Referenčná implementácia CoCoME slúži ako benchmark pre modelovacie softwarové technológie a zahrňuje modelovanie výkonu aplikácie. Cieľom práce je vytvoriť verziu CoCoME vhodnú na modelovanie výkonu a vyťaženosť hardwarových zdrojov jednotlivými časťami aplikácie. Práca popisuje použité technológie a postupy pri implementácii CoCoME v komponentovom modeli SOFA s komponentami reflektujúcimi existujúcu SOFA CoCoME architektúru.

Klíčová slova: komponentový model, využitie hardware zdrojov, middleware

Title: CoCoME in SOFA
Author: Peter Hladký
Department: Department of Software Engineering
Supervisor: Doc. Ing. Petr Tůma, Dr.
Supervisor's e-mail address: Petr.Tuma@dsrg.mff.cuni.cz

Abstract: Hardware resource usage of a software can be modeled and therefore it is possible to predict changes in resource usage, which will be caused by changes in implementation of the software. CoCoME models a real trading system. The reference implementation of CoCoME serves as a benchmark for software modeling technologies, including performance modeling. The goal of the thesis is to create a version of CoCoME suitable for performance modeling with hardware resource usage by individual parts of the application. The thesis describes technologies and procedures used to implement CoCoME in SOFA component model with components reflecting the existing SOFA CoCoME architecture.

Keywords: component model, hardware resource usage, middleware

Chapter 1

Introduction

It is difficult to predict how will changes made in source code affect performance of a software system. With a small software system it is possible to launch and profile it or test it to inspect changes in performance. For a large software system the whole runtime environment would have to be replicated in order to inspect changes in performance. This would be very expensive as a large system usually consists of many smaller parts running on multiple machines.

Software modeling tools and methodologies evolved to provide procedures how to assess changes in performance when changing parts of a software system. This approach does not require any changes in the inspected software and it does not require to fully replicate runtime environment, therefore it is much cheaper and viable.

In some cases software modeling tools do not provide realistic results, especially when modeling a system running on several machines by a monolithic model running on one machine. Hardware effects such as caching, swapping, network latency are difficult to model while parts of the application running on same machine are affecting each other.

To better inspect performance of a monolithic application we can divide the application into smaller parts called components. There are tools allowing us to build an application consisting of multiple components and run it. With components we are able to isolate, study and measure only parts of the application we are interested in.

The goal of our work is to create a reasonably complex implementation of a component application, which would be used in the process of developing new application performance models that would take into account the impact

of complexities related to resource sharing.

Before describing in detail how CoCoME was ported to SOFA, the thesis continues with describing the architecture of CoCoME and SOFA. CoCoME servers as the model of a monolithic application and SOFA [4, 3, 5] as the component model used to implement CoCoME as a component application.

Chapter 2

CoCoME

2.1 Introduction and System Overview

This section provides brief description of CoCoME - The Common Component Modeling Example as mentioned in [7]. CoCoME models a real *Trading System* which can be observed in a regular supermarket. It is fully defined in [7], where individual parts are defined using UML diagrams, use cases and properties of the trading system (number of stores, cash desks per store, etc.). The trading system consists of the following parts:

Cash Desk (figure 2.1) operated by a cashier. The sale is started and finished at each Cash Desk. It is possible to switch it into express checkout mode, where each customer buys only a few goods and pays by cash to speed up the payment. The cashier uses *Bar Code Scanner* to scan items to be purchased by a customer. *Cash Box* displays the final price and handles payments. A customer can pay either by a credit card or by cash. The credit card payment is handled by *Card Reader*. Final receipt is produced by *Printer* and express mode is displayed by *Light Display*. All the parts are connected to *Cash Desk PC* which communicates with *Bank*.

Store (figure 2.2) consists of *Cash Desk Line* which models multiple cash desks connected to *Store Server*. Store Server is connected to *Store Client* which can be used by a store manager to view reports related to store operation or change prices of the goods. The *Store Server* includes *Inventory* of all goods of the corresponding Store.

Enterprise (figure 2.3) is a set of Stores where *Enterprise Server* exists and all Stores are connected to it. *Enterprise Client* can be used by a manager to produce different reports related to stores in Enterprise.

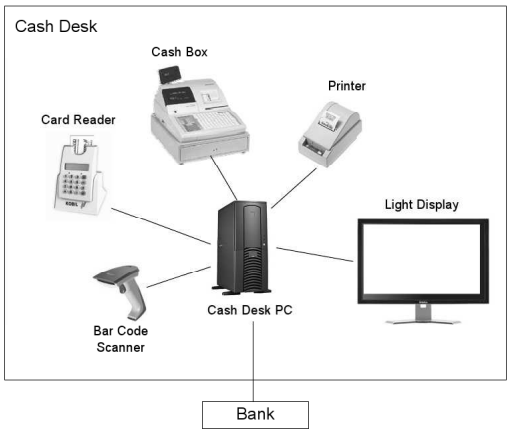


Figure 2.1: The hardware components of a single Cash Desk [7].

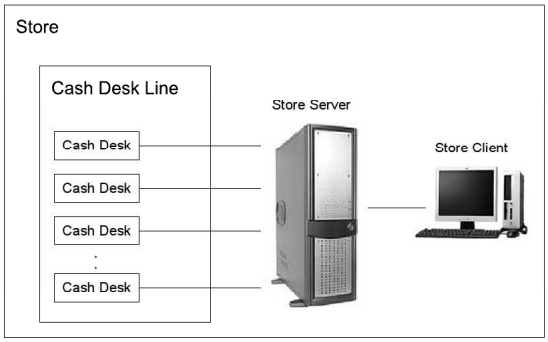


Figure 2.2: An overview of entities in a store which are relevant for the Trading System [7].

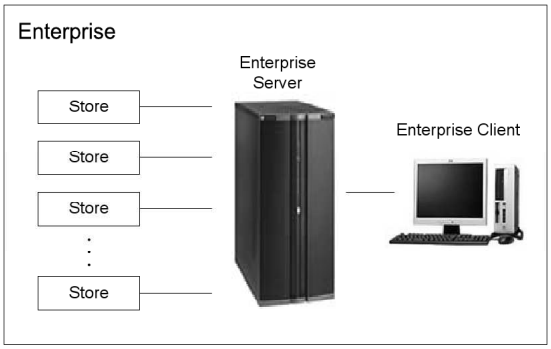


Figure 2.3: The enterprise consists of several stores, an enterprise server and an enterprise client [7].

2.2 Reference Implementation

The CoCoME specification in [7] with UML diagrams, use cases and properties of the trading system was used to create a reference implementation of CoCoME. The implementation is monolithic application using Hibernate [10] middleware and Derby database [6]. Hibernate is providing a middle layer between a database which stores data and an application which communicates with the database. As the implementation is monolithic it does not provide necessary granularity to measure hardware usage and performance of individual parts of the application. The application needed to be divided into smaller parts called components and implemented in a component model. The following section 2.3 describes implementation in Fractal component model.

2.3 Implementation in Fractal

CoCoME in Fractal [1] is implementation of CoCoME in Fractal component model. Several changes to the reference CoCoME implementation were made to fit the Fractal component model. Only part of the reference implementation of CoCoME was implemented in Fractal, omitting graphical user interfaces and parts of the implementation using Hibernate [10] to communicate with database. The implementation in Fractal and changes to the reference CoCoME implementation are described in detail in [1]. The diagram of the implementation of CoCoME in Fractal can be observed on figure 2.4.

We started with implementation of CoCoME in Fractal as it was already logically divided into components and changes were made to the reference implementation to fit the hierarchical model. However, we faced several difficulties as some of the implementation details were related to the Fractal component model. It is further discussed in the chapter 5 and section 5.2.

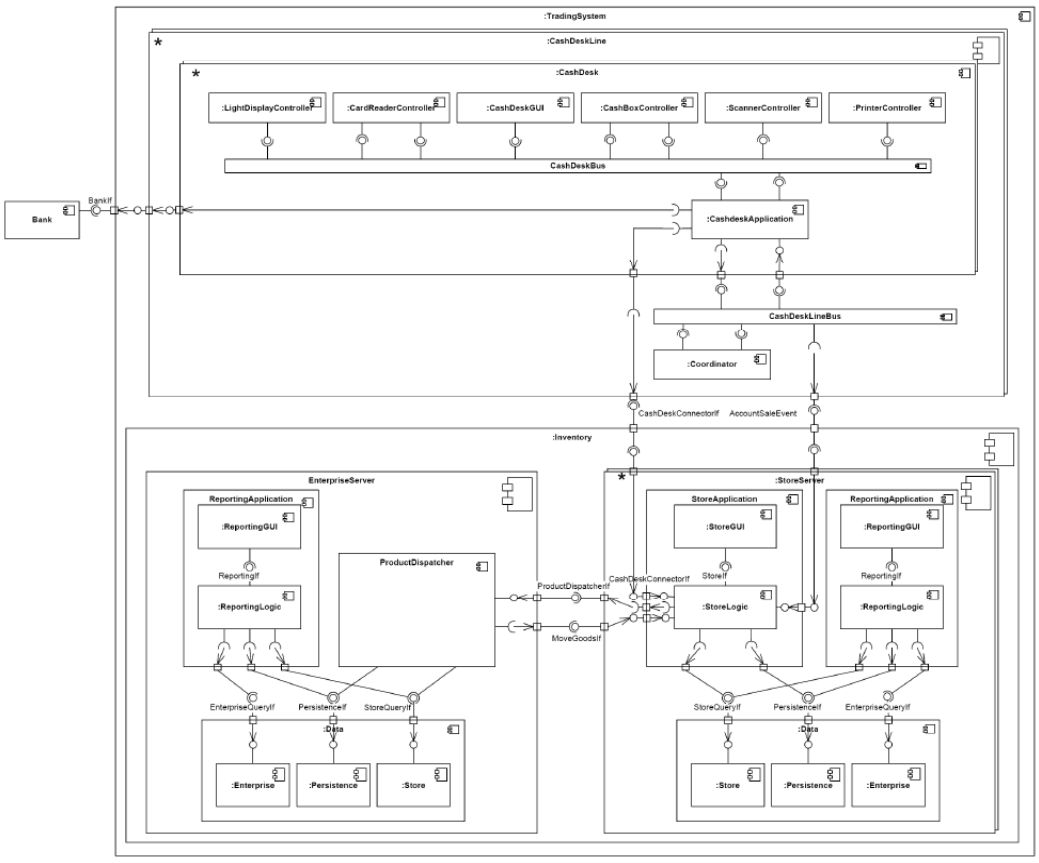


Figure 2.4: CoCoME architecture as it is used in the Fractal component model [1].

Chapter 3

SOFA

3.1 Introduction

This chapter talks about main ideas behind component based development and SOFA component model as published in [4, 3, 5]. Component based development is a way of developing applications consisting of smaller parts called components which are connected together by interfaces. A component is usually viewed as a black-box software entity with well defined interface and behavior. A component model is specified by the component rules and features for component lifecycle, composition, etc.

There is a flat component model and a hierarchical component model. The flat component model allows to build only applications composed from components that are on the same level of hierarchy. This property is very limiting as it does not allow to build a large component by nesting smaller components inside. The hierarchical component model allows multiple levels of hierarchy which means it is possible to build *composite* components which are created by nesting other components inside.

SOFA - SOFtware Appliances project was developed mainly by Distributed Systems Research Group at Charles University in Prague, current version is SOFA 2 (further referenced as SOFA). SOFA is a hierarchical component model and it is better described in the following sections.

3.2 SOFA Component Model

SOFA provides an infrastructure to develop and run applications built from components. Each component communicates with other components via *required interfaces* and *provided interfaces* which are specified by each component.

Required interfaces define methods that are called by a component, but they are implemented by different components. *Provided interfaces* implement methods of a component, but they are usually called by different components. *Component frame* serves as a description of component's required and provided interfaces. *Component architecture* implements component's frame and provided interfaces.

SOFA is hierarchical component model which implies two types of architecture - *primitive* and *composite*. *Primitive architecture* is implementation of component's provided interfaces and includes the code of the implementation. *Composite architecture* is a composition of sub-components and it does not include any implementation, its functionality is derived from its sub-components.

A component is a set of classes and interfaces. In the case of primitive component it includes at least one class with implementation. To describe a component, ADL - Architecture Description Language is used and the description is stored in an XML file. There are separate ADL descriptions for interface, architecture and frame of a component.

Interface ADL description includes interface type name and signature of a class with interface definition. Signature is fully qualified name of the class with interface definition in Java namespace.

Architecture ADL description of a primitive component includes architecture name, frame type name associated with the architecture and fully qualified name of the class with implementation in Java namespace. In the case of a composite component, the ADL description includes architecture name, frame type name associated with the architecture, list of subcomponents and connections between subcomponents. Each subcomponent item includes subcomponent name, frame type name and architecture type name. Each connection item lists names of two subcomponents and name of one interface through which they communicate.

Frame ADL description includes frame name and a list of provided and required interfaces by a component. Each provided and required interface item includes interface name and interface type name.

3.3 Component Lifecycle

Lifecycle of a SOFA component involves following stages: (i) component development, (ii) application assembly, (iii) application deployment and execution.

Component development involves writing description of a component architecture and frame, writing code for provided interface and its implementation in the case of a primitive component.

Application assembly is done using frame ADL descriptions, where each subcomponent in an architecture described by a frame is assigned to particular architecture. The process starts with a top-level architecture and continues recursively until primitive architectures are found. The result is an assembly ADL description stored in an XML file, which specifies how are components assembled into final result.

Deployment and execution is the last step of the SOFA component lifecycle. During the deployment it is specified, where will be particular component of the application executed. Connectors between communicating components are generated. The result of the deployment stage is a deployment plan that serves for execution of an application.

3.4 Runtime Environment

SOFA component model is implemented in Java. The runtime environment consists of a single *repository* and several *deployment docks*. The SOFA runtime environment is called *SOFAnode* (figure 3.1).

3.4.1 Deployment Docks and Deployment

A *deployment dock* is used for launching components and it provides necessary infrastructure for starting, stopping and updating components. *Deployment* is phase of an application development where a developer assigns particular deployment dock in SOFAnode to each component. The assignment is stored in a deployment plan and it is used when launching the application, where each deployment dock is contacted and instructed to launch components assigned to the dock. Code of the components is automatically obtained by deployment docks from the SOFA repository.

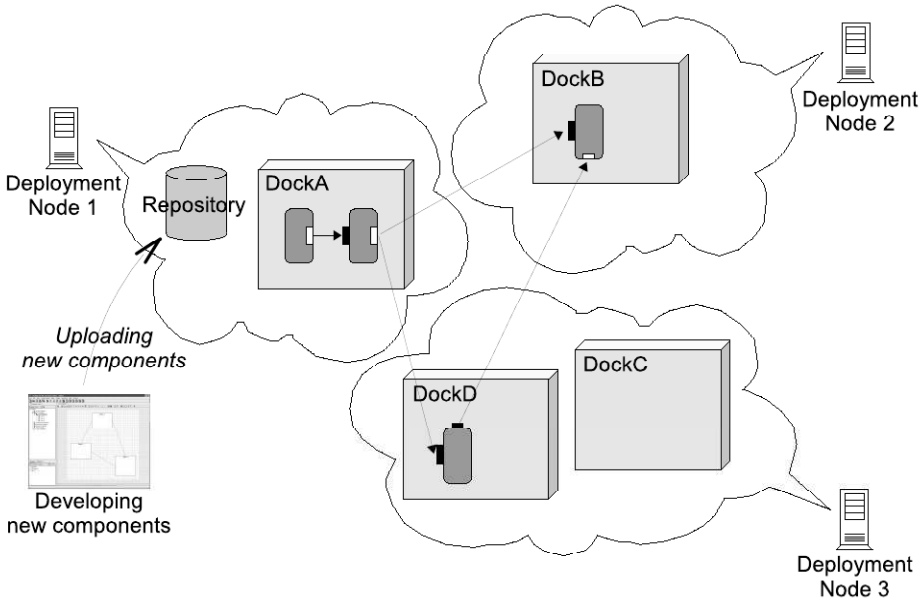


Figure 3.1: SOFAnode example as illustrated in [3].

3.4.2 Repository

The *repository* stores *component meta-data* and *component implementations*. It is accessed during development to store created components and during runtime to load the components. All entities in the repository are versioned. This allows to use different versions of the same component within one application. With different versions of the same component the problem of *class name clashes* can arise as described in detail in [8, 9].

Class name clashes [8, 9] occur when two different classes with the same name have to be loaded into Java Virtual Machine. In Java, two classes with the same name cannot coexist in Java Virtual Machine unless they are loaded by different classloaders.

SOFA solves this problem by bytecode manipulation of compiled Java classes. During the process of uploading classes to the SOFA repository, each class is renamed to have unique class name and references to this class are changed in the bytecode accordingly. A unique class name is created from the original class name and version of the component, which is appended to the class name. This mechanism allows to use different versions of a component within one application.

3.5 SOFA Development Tools

When using the SOFA development tools, every interface, architecture and frame resides in a separate directory. Each directory includes ADL description stored in **adl.xml** file. Java source code is stored in the **code** directory in case of interface and primitive architecture directories.

To develop an application in SOFA component model a developer needs to run **sofa-node.sh** script to start the repository and other important parts. Then the developer defines interfaces, frames and architectures by **cushion.sh** tool which creates the basic directory structure and ADL files. The developer then fills the ADL description of interfaces, frames and architectures and creates implementation of each component which is stored in the **code** directory. After all the components are described by ADL and implemented, it is then compiled by the **cushion.sh** tool and uploaded to the SOFA repository.

The **cushion.sh** tool is then used to create an assembly description and deployment plan to deploy the application. The assembly description provides information about how is the final application composed from different components. The deployment plan needs to be filled in manually by a developer to assign a deployment dock to each component, as described in earlier section 3.4.1 and figure 3.1. After filling in the deployment plan the application is ready to be deployed. During the process of deployment, connectors are automatically generated between connected components.

The application is then ready to be launched. For every dock specified in the deployment plan a deployment dock with the same name needs to be started, this is done by **sofa-dock.sh** script. The application is then launched by **sofa-launch.sh** script and stopped by **sofa-shut.sh** script, to view all running applications **sofa-ps.sh** script is used.

The whole SOFA development lifecycle and SOFA development tools are described in detail at the SOFA website [11].

Chapter 4

Hibernate

4.1 Introduction

This chapter gives a basic overview of the Hibernate system as discussed in the Hibernate reference documentation [10]. Hibernate is object/relational mapping tool. It is used to map Java objects to database tables. It uses its own SQL-like language to query a database. One of the advantages is that Hibernate is not tied to one particular database system. It is configured within application, which database system will be used to store data. This advantage gives an option to developer to change the underlying database without changing the implementation.

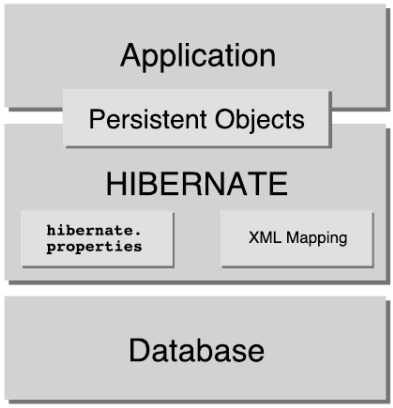


Figure 4.1: A high-level view of the Hibernate architecture as illustrated in Hibernate reference documentation [10].

CoCoME reference implementation uses Hibernate to create persistent objects and to access *Inventory* in the *Store Server* part of the application. A persistent object is an object which outlives the execution of the program that created it. Persistent objects are used to store and work with data queried from database. A persistent object represents database table and instance of the object represents one row of the table.

The whole integration of the Hibernate system into the SOFA framework caused a number of issues, which were problematic to resolve. The following sections describe different parts of the Hibernate system that are necessary for the implementation of CoCoME in SOFA. Details about individual issues related to Hibernate and their solutions are discussed in the implementation chapter 5.

4.2 Entity Manager

As described in the Hibernate documentation for EntityManager [10], the EJB3 specification standardizes the basic APIs and the metadata needed for any object/relational persistence mechanism. Hibernate EntityManager implements the programming interfaces and lifecycle rules as defined by the EJB3 persistence specification.

The EntityManager API is used to access a database in a particular unit of work. It is used to create and remove persistent entity instances, to find entities by their primary key identity, and to query over all entities.

4.3 Annotations

As described in the Hibernate documentation for Annotations [10], annotations are used as meta-data to mark Java objects as persistent objects. Meta-data are compiled into bytecode and read during runtime.

The **@Entity** annotation marks an object as a persistent object. The **@Id** annotation marks object's identification property and it serves as a primary key to database table. The identification can be set by application or can be generated by Hibernate using the **@GeneratedValue** annotation. Other annotations can be used to define table specific properties with **@Table** annotation, mapping specific properties with **@ManyToOne** and **@OneToMany** annotations, etc.

4.4 Configuration

As described in the Hibernate documentation for EntityManager [10], the configuration of entity manager reside in the **persistence.xml** file. There can be more than one entity managers specified. Each entity manager (persistence unit) specifies its name, properties and classes defining persistent objects.

A name is used to identify particular entity manager in the application. Properties are used to specify which database system will be used, access to the database and URL where the database listens for queries. List of persistent objects is specified by a list of classes defining these objects.

The **persistence.xml** file needs to be located in the **META-INF** directory in the Java class path. It is then packaged into persistence archive and the information stored in **persistence.xml** is used during runtime. All properly annotated classes included in the archive will be added to the configuration of the entity manager.

The **Persistence.createEntityManagerFactory()** method is used to create entity manager factory with given name of the entity manager. The class path will be searched for **META-INF/persistence.xml** using the **ClassLoader.getResource("META-INF/persistence.xml")** method. The **Persistence** class will examine all the persistence providers available in the class path and find the one which is responsible for creation of the entity manager with given name. Persistence provider will then find the entity manager that matches the name specified in the source command line with the name specified in **persistence.xml** file. If no **persistence.xml** file with a correct name of entity manager is found, **PersistenceException** is raised.

Chapter 5

Implementation

5.1 Motivation

As described in chapter 2, CoCoME is model of a trading system. A trading system usually consists of several parts of software and hardware. These parts are connected together and they communicate with each other.

The reference implementation of CoCoME is monolithic application, so every part of the modeled trading system runs on the same machine. There is no possibility to run logical parts separately, except the database which can run on a separate machine. The implementation therefore does not give precise view of a real trading system.

In addition, the monolithic implementation of CoCoME does not provide necessary granularity to measure resource usage of individual parts of the application. Every logical part of the implementation is influenced by every other logical part as they all share the same hardware resources and interact with each other. Therefore results of resource usage measurements are difficult to clearly interpret.

The SOFA framework gives us an option to divide the monolithic implementation of CoCoME into several components. Components then represent different parts of a trading system as it is seen in reality and can be run within different deployment docks as described in chapter 3. Connection between components is provided by the SOFA framework, which uses Java RMI - Remote Method Invocation as it provides the necessary communication mechanisms.

The SOFA framework gives us an option to measure resource usage of a single component. Deployment docks can be distributed and run on several

physical machines separating components which would otherwise influence each other as it is in the monolithic implementation of CoCoME. This approach can give us a better understanding of resource usage and mutual interactions of different parts of the original application.

The process of implementation of CoCoME in SOFA will consist of the following steps. We need to define interfaces, architecture and frames, then upload them to the SOFA repository. The next step is to implement all the primitive components and upload them to the SOFA repository. To accomplish these steps, we need to solve technical issues introduced by functionality which is difficult to express as components, namely object persistence.

We had several options how to implement the CoCoME model in the SOFA component model. We could use the CoCoME specification, UML diagrams, use cases and properties of the trading system defined in [7] and implement CoCoME in SOFA from scratch. This would mean to write parts of the application that were already written in the reference implementation of CoCoME. The other option was to use the reference implementation of CoCoME, divide source code into logical parts and adjust it to fit the SOFA component model. We were also provided with limited implementation of CoCoME in the Fractal component model. We chose to use the CoCoME in Fractal [1] implementation to start with and the reasons are explained in the following section 5.2.

5.2 Conversion from Fractal

The CoCoME in Fractal [1] implementation was chosen as the implementation to start with for several good reasons. Fractal is hierarchical component model as SOFA and both component models are similar. CoCoME implementation in Fractal was already logically divided into components and described by Fractal ADL. It was also modified compared to the original implementation to better fit the component model, see [1] for details. However only cutout of the original CoCoME application was implemented in the Fractal component model as described in chapter 2.3.

A basic conversion script from Fractal implementation of CoCoME to SOFA framework was provided. As the conversion process is not fully automated, we had to do the following steps manually: fill the created directory structure with additional classes and resolve class dependencies, create component bindings, resolve issues related to properties of the Fractal component model. All these steps are discussed in detail in the following subsections.

5.2.1 Basic Structure

To create components of the CoCoME implementation in SOFA framework, we needed to create the basic directory structure for every interface, architecture and frame. We used the conversion script that was part of the CoCoME in Fractal implementation. The conversion script created the directory structure, filled it with ADL description files and basic Java source code with interface definitions and architecture implementations. Java namespace was changed in classes and directory structure holding source code was created automatically according to the new namespace.

Functionality of the conversion script is very limited as it does not handle dependencies between Java classes defining types and interface definitions and architecture implementations. We had to resolve the dependencies manually by checking all the interfaces and architectures as it is described in the following subsection 5.2.2.

5.2.2 Classes and Dependencies in SOFA

For every interface, architecture and frame there needs to be a separate directory. Interface directories and primitive architecture directories also include **code** directory, which holds the Java source code. To compile the application in the SOFA framework, **cushion.sh** script is provided. The **cushion.sh** script compiles all interfaces at first and then it compiles primitive architectures. Classes from provided and required interfaces specified in ADL description are found in **code** directories of the interfaces and used during architecture compilation. So the classes used by interfaces in the architecture do not need to be in the architecture **code** directory.

We resolved the class dependencies that were not handled during conversion process by the following approach. We tried to compile an interface which resulted in a number of Java compiler errors saying which classes were missing. We found all the needed classes in the implementation of CoCoME in Fractal, changed namespaces and copied the classes into **code** directory of the interface. After resolving dependency issues of all interfaces, most of the dependency issues of architectures were also resolved. After the previous process was finished, we just needed to find classes used only by the individual architectures.

The Eclipse IDE provides refactoring tools, which we could use. We would have to find the missing classes as described in the previous paragraph. We could then copy the missing classes to the right **code** directory. And finally

rename the classes to match the right namespace. However, the refactoring tools in Eclipse would not help much as it can be used only to work with the whole source code package or a single class and not with a selection of multiple classes from different source code packages.

The following subsection 5.2.3 suggests, how could be the conversion script further developed to automate the process of conversion.

5.2.3 Conversion Automation

The conversion script could be further developed to automate the process of finding all the class dependencies, changing namespaces of classes and copying them to right **code** directories. However, it would have to be quite sophisticated as dependencies of an architecture are resolved by an ADL file where a list of provided and required interfaces is stored. The **code** directories of the interfaces are inspected during compilation and needed classes are used to resolve dependencies.

To automate the conversion, the script could try to compile every interface by itself. Then it could process the Java compiler error output and create a list of missing symbols. Then it could try to find a class for every symbol and use it for implementation of CoCoME in SOFA. However, this approach cannot be used in the case of architectures.

In order to resolve dependencies of an architecture, the coversion script would have to call **cushion.sh** tool for compilation. The reason is that **cushion.sh** inspects the architecture ADL file and uses classes of required and provided interfaces during the architecture compilation. If the conversion script would just use a Java compiler as in the previous paragraph, it would copy classes that are already used by architecture interfaces to **code** directory of the architecture. This would cause class duplicities in the SOFA framework, which are discovered and reported by **cushion.sh** when uploading the compiled components to the SOFA repository.

After we resolved all the class dependency issues, we needed to create bindings between components so they can communicate. The process is discussed in the following subsection 5.2.4.

5.2.4 SOFA Component Bindings

In order to create bindings between application components so they are able to call interface methods of other components, the component architecture

has to implement the **setRequired()** method of the **SOFAClient** interface. The **setRequired()** method is called during initialization of the component and assigns required interfaces to the local interface variables in the architecture implementation. These variables are then used to call methods on required interfaces, which are provided by a different application component.

We needed to manually define all the **setRequired()** methods in all components that were using interfaces of other application components. To accomplish this task, every component was checked for a list of required interfaces. The list is stored in the ADL file of the component frame. Then for every required interface a variable was defined and assigned in the **setRequired()** method. The code was then checked for method calls of required interfaces in order to see if every required interface is properly assigned.

In the implementation of CoCoME in Fractal, some required interface variables were assigned to references to object instances rather to provided interfaces of the object. This is not possible in SOFA framework unless all the Java source code related to the object is within the **code** directory of the component using its interface. In such a case, there is no way, how to separate two components where one providing the interface is part of the one requiring the interface. This is not wanted as objects providing the interface are logically different components, therefore we changed all the references from object instances to interface references.

The described process in this section is very error prone as every component needs to be checked for required interfaces, proper variables need to be created and assigned in the **setRequired()** method and the component architecture implementation needs to be checked for every method call of every required interface. Therefore we suggest partial generation of SOFA bindings in the following subsection 5.2.5.

5.2.5 SOFA Component Bindings Generation

To further automate the process of porting CoCoME to the SOFA framework, we suggest partial generation of SOFA component bindings. The generation tool would check every component frame for a list of required interfaces and location of the architecture implementation. In the architecture implementation, the **setRequired()** method would be partially defined listing all the required interfaces.

The generation tool could then give a choice to developer whether to automatically generate variables for every required interface and assign them

properly in the **setRequired()** method or let the developer do it manually. The rest of the Java source code could be automatically checked for references of object instances which implement the required interfaces and changed to variables with references to required interfaces assigned in the **setRequired()** method.

The whole process would require developer's cooperation, but we believe it would speed up the process of porting and reduce number of errors produced when going through the same process manually.

5.3 Shared Application Parts

The Fractal component model allows to define a shared component, which can be shared by multiple components. In the implementation of CoCoME in Fractal, the **Simulator** component is an example of shared component. SOFA component model does not support this feature, so we needed to resolve this issue in the implementation of CoCoME in SOFA.

The **Simulator** component simulates the CoCoME Use Case 1 - Process Sale of CoCoME specification [7] and it does not specify an interface. Instances of other components register at the **Simulator** by calling its register methods. Every reference to an instance is then added to an array of the registered objects. The **Simulator** then works with the arrays to choose instances and calls their methods.

This is not possible in SOFA for a number of reasons. Firstly, the class and all related classes of the object **Simulator** would have to reside in every component which uses its register methods to register itself within the **Simulator**, because the **Simulator** does not specify interface for these register methods. That would result in copying major part of the application in every component **code** directory as every component which Simulator works with, would have to be present in the particular component **code** directory. This is unwanted state, because it would result in most of the application source code being part of every component.

We needed to adjust the **Simulator** component in a way such that it does not have to be part of other primitive components. It is discussed in detail in the following section 5.4.

5.4 Simulator Component

To resolve the problem of shared **Simulator** component (further referenced as simulator) in the SOFA framework, we extracted the component and used it as a separate component. We defined and implemented interface, architecture and frame of the component. In the interface ADL, we specified the provided interface which consisted of the register methods for components to register their interfaces within the simulator. These register methods were implemented in the architecture and after a register method was called by other component, reference to the component's interface was added to an array of the simulator. The simulator interface **code** directory needed to include classes of all the interfaces that simulator worked with in order to correctly compile. It is because SOFA resolves interface dependencies only for architectures, where the list of required interfaces is provided in the frame ADL.

Even if the compilation was successful, there was a problem which was disclosed during upload to the SOFA repository. When **cushion.sh** script performs upload, jar code bundles are created from the compiled classes of a component and uploaded to the SOFA repository. During the process of adding classes of a component to the jar code bundle, it is checked for duplicate classes. This checking caused a problem when we tried to upload components which communicated with the simulator. The reason was the simulator used variables to hold references to interfaces of components that registered within the simulator. Classes defining these interfaces needed to be present in the **code** directory of simulator, so it could be compiled. When a component that required the simulator interface to register was being uploaded, classes in the simulator interface **code** directory were used. And because class with provided interface of the component was also in the simulator **code** directory a duplicity was detected and exception was raised by the **cushion.sh** tool.

In order to resolve this problem, we needed to adjust the **Simulator** component again. The component in the final version does not provide any interfaces. In the frame of the component, we specified to require all the interfaces it was working with. The binding between the simulator and other components was created by defining the **setRequired()** method which added reference to every required interface into the simulator arrays and therefore the simulator was able to call methods of the needed components. To connect the simulator with other components we needed to specify the bindings

also in the top-level architecture of the **CashDesk** component.

After all these changes made to the **Simulator** component, we were able to compile and upload most of the components which were using it to the SOFA repository. We experienced similar issues related to duplicate classes with object types shared by multiple interfaces. We address the problem and possible solutions in the following section 5.5.

5.5 Shared Types in SOFA

It is common for an application to share types that are used within different classes of the application. For example there are two interfaces that use the same common type. Imagine a part of source code that implements both interfaces. It becomes a problem when this part of source code forms a component and this component is about to be uploaded to the SOFA repository. The class duplicity is caused by the common type, because it is being added to the code bundle multiple times, as more than one interface is using it. Figure 5.1 illustrates the described situation.

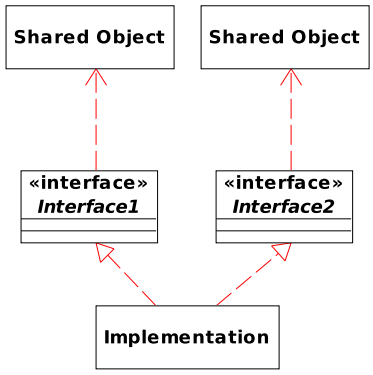


Figure 5.1: Component implementing two interfaces using shared object.

It is partly a design question, whether to have a component implementing multiple interfaces or multiple components each implementing one interface. The **CashDeskBus** component in the implementation of CoCoME in Fractal is good example of a component implementing multiple interfaces. It provides communication bus between smaller back-end components and **CashDeskApplication** component as it can be observed in figure 2.4. If there are multiple interfaces using same types, classes defining these types are also included in code directories of these interfaces, otherwise it would

not be possible to compile them. If these interfaces are implemented by the same component the upload to the SOFA repository fails. It is because during the code bundle creation, multiple classes with the same name are detected and class duplicity exception is raised. We suggest several solutions in the following subsections.

5.5.1 Component Division

The problem with duplicate classes can be resolved by multiple approaches. The component implementing multiple interfaces can be divided into smaller components each implementing one interface. However this would result in enormous fragmentation of the application and in many cases this is not necessary. Other disadvantage by this approach is that it leaves no choice for a developer as everything would have to be divided in small components each implementing just one interface. Figure 5.2 illustrates the described approach.

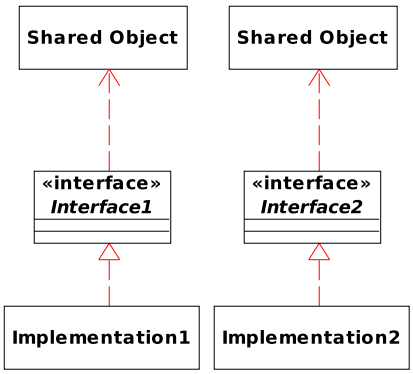


Figure 5.2: Component division.

5.5.2 Merging Interfaces

Another approach resolving the situation would be merging multiple interfaces into one large interface, which would then be implemented by the same component. This approach could result in duplicities of code if different components would implement different parts of the large interface. The reason is that besides the large interface other parts would be defined as interfaces with different names and then implemented. And therefore same parts of

code would be in different **code** directories. This would mean correcting the same code in multiple places if errors are found. Figure 5.3 illustrates the described approach.

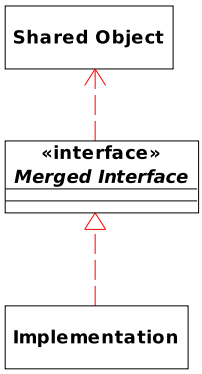


Figure 5.3: Merging interfaces.

5.5.3 Class Skipping or Renaming

SOFA could resolve the problem by skipping the duplicate classes during the code bundle creation. It would have to be checked whether the two duplicate classes are the same or they only share same name. In the latter case, this approach would probably result in an error that would be difficult to find as different class would be used as originally intended.

Renaming one of the duplicate class would also resolve the problem, so the classes would be the same, but with different names. The name would also have to be changed in the code as all the object types are named after the name of the class. The renaming can be done in Eclipse easily. However it would again mean duplicate code with different class name and it would be difficult to fix errors in the application.

5.5.4 Separating Common Java Types

Probably the best approach is to have a possibility to specify where common Java types can be stored and let the SOFA know and use it. The present version of SOFA does not support this approach directly. It can be achieved indirectly by editing SOFA development tools.

We used the last approach to solve the problem temporarily, until this issue is resolved in the SOFA framework. The following section 5.6 fully describes how issues with class path are resolved.

5.6 SOFA Class Path

Many applications use methods implemented by external libraries. SOFA does not directly give an option to developer to specify paths where additional libraries can be found. There are again several approaches how to solve this issue.

The additional libraries can be copied to directory for Java Runtime Environment Extensions, which is **/usr/lib/jvm/java-6-sun/jre/lib/ext** in our case. The libraries are then visible to Java Virtual Machine and therefore to every running application. This was our initial approach. However, the content of the extension directory affects all other Java applications running on the same machine. If there are multiple applications developed using different libraries or different versions of the same libraries it is better to tie the libraries with the particular application.

We tried to change the environment variable **CLASSPATH** which specifies the default user class path. So we set **CLASSPATH** variable to directory with all the needed libraries. This did not have any effect, because as we later found out the SOFA development tools set their own class path variables. It is possible to adjust the SOFA development tools to use additional class paths to find all the necessary libraries and classes. We needed to edit several scripts which are used in order to run the development tools. After these adjustments, we were able to move all the necessary libraries associated with the application to directory which is then inspected by the SOFA development tools. It would be helpful if SOFA could provide some way of environment configuration, so developers can easily set all the class paths they need for extra libraries and code either for the whole application or individual components.

5.7 Assembly and Deployment

We adjusted the class path of SOFA development tools and moved all the external libraries and classes of shared types outside the directory structure of components. We were then able to successfully assemble the application

from components and create a deployment plan. We specified name of a deployment dock for each component and successfully deployed the application. During the deployment process, connectors were automatically generated and stored in the SOFA repository. Connectors are used to link all component interfaces and provide communication between them.

When we finally tried to launch the application it resulted in **ClassCircularityError** exception which occurs in case the default **ClassLoader** is overridden. The exception is related to SOFA inner structures, we reported the issue to SOFA framework developers and it was later resolved.

Meanwhile, we decided to implement **Store** and **StoreTester** components. The **Store** component communicates with database using the Hibernate system [10]. The implementation of CoCoME in Fractal omitted components involving communication with database. So the following sections describe in detail how to integrate Hibernate [10] into the SOFA framework.

5.8 Store and StoreTester Components

We came across various SOFA limitations during the effort of implementation CoCoME in SOFA. We were not able to resolve all the SOFA limitations, so we were not able to continue in the implementation process. Therefore we decided to try to implement smaller part of the original CoCoME implementation which communicates with database. This part was not implemented in the Fractal version of CoCoME. We think it is beneficial for future SOFA application developers as we provide detailed approach how to integrate Hibernate [10] into the SOFA framework.

We used structure prepared by the conversion script from implementation of CoCoME in Fractal. The implementation of **Store** was used from the reference CoCoME implementation. It needed to be adjusted so it can run independently from the rest of the application. We created two components, *Store* and *StoreTester*. The *Store* component implements interface to query a database. And the *StoreTester* component implements methods to call the *Store* interface.

Hibernate middleware is used by the components to communicate with an underlying database, which is the Derby database [6] in our case. Therefore we needed to integrate Hibernate configuration with our components and SOFA framework. We needed to resolve several issues, which are described in the next section 5.9.

5.9 Hibernate and SOFA

The reference implementation of CoCoME uses Hibernate [10] in order to communicate with a database. The Hibernate documentation [10] instructs, there needs to be a **META-INF** directory with **persistence.xml** file in the class path of the application so it can be added to the jar code bundle and loaded during runtime. SOFA implements its own **ClassLoader** and we needed to find a way how to make SOFA framework use the **persistence.xml** file.

5.9.1 Hibernate persistence.xml

We tried number of approaches, starting with adding the **META-INF** folder to every **code** directory of the SOFA application. Cushion upload then created code bundles including the **META-INF** folder. However, launching the application revealed that the **persistence.xml** could not be accessed by SOFA tools used to launch the application. So we added the **META-INF** into the SOFA class path by editing SOFA development scripts. This did not resolve the problem either, so we decided to better locate the problem. We added debug messages to source code of **ejb3-persistence**, the EJB3 Persistence Library to better understand the problem.

After some time trying all sorts of other approaches, we found out the right way by recompiling Hibernate Entity Manager. We put the **META-INF** directory with **persistence.xml** file into the Hibernate Entity Manager **build/classes** directory. This way the **META-INF** directory became part of the Hibernate Entity Manager jar. The jar was then moved to the class path of the SOFA application and the **persistence.xml** file became accessible.

5.9.2 Hibernate Libraries

Hibernate is depending on a number of other libraries, which either have to be included in a Java Runtime Environment Extensions directory (in our case **/usr/lib/jvm/java-6-sun/jre/lib/ext**) or in a class path of a SOFA application. So we moved all the needed libraries to class path of our SOFA application, which included **cglib**, the Code Generation Library and **asm**, the Java Bytecode Manipulation Library. The problem is that the set of libraries used by SOFA itself include **asm** library of different version.

Hibernate uses the older version of **asm** library, where SOFA uses the newer version of the **asm** library. When we tried to run our SOFA application it resulted in a number of compiler errors regarding the **asm** library. We tried to resolve the issue by using only the newer version of the **asm** library within Hibernate. However, the newer version of the library did not include all the classes of the older version. So when Hibernate was trying to use some of the missing classes, it resulted in a number of compiler errors. We later found out the **cglib** library was distributed also as **cglib-nodep** library with the older **asm** library included within. When we used the **cglib-nodep** library, the problem was resolved as no dependency issues arose.

Our application uses the Derby database [6] as a back-end database with cooperation with Hibernate. In order to use Derby database, we had to copy all the database libraries into the class path of our SOFA application.

5.9.3 Persistent Classes and SOFA Renaming

To use Hibernate, **persistence.xml** file also includes a list of persistent classes used by an application. So we listed all the persistent classes in **persistence.xml** file. We then compiled the application and uploaded the generated code bundles with the **cushion.sh** tool to the SOFA repository. When we were trying to launch the application we came across several exceptions telling us the persistent classes could not be found. This situation was caused by the SOFA class renaming mechanism to avoid class name clashes as described in section 3.4.2. All the references to classes were changed in the bytecode including persistent classes. We ended up with bytecode with references to the renamed classes and **persistence.xml** file with original names of the classes.

We needed to be able to access the renamed classes. We could change the **persistence.xml** and use generated names of the persistent classes by SOFA development tools. The generated names are found in the internal structures of the SOFA repository in **renamed.jar** of a component. This is not very useful solution as the **persistence.xml** would have to be edited and entity manager recompiled every time version and name of a persistent class is regenerated.

To circumvent this issue, we used **addAnnotatedClass()** method to add all persistent classes to the entity manager configuration directly in the source code. This way the class names were automatically changed in the bytecode to the right names and references that pointed to these classes.

As we were trying to launch the application, we found out that class name used as a type for declaration of a generic type, like **Collection** is not covered by SOFA class renaming mechanisms. The place in the bytecode with class reference is not changed and it uses the original class name.

So the only solution, when persistent classes are not renamed, but can be accessed during compilation and runtime of the application was to locate and move all persistent classes outside the SOFA application directory structure. Class path to directory with persistent classes was set in the class path of SOFA development tools. This way we were able to compile the application and upload the generated code bundles to the SOFA repository. The persistent classes were not renamed as they were outside the component directories.

We were able to successfully assemble, deploy and launch the **Store** and **StoreTester** components. The **Store** component was able to use Hibernate [10] and successfully communicate with the Derby database [6]. We were able to perform several tests with this implementation regarding resource usage.

Chapter 6

Measurements

6.1 Resource Usage

A componentized application gives us more ways to study its performance and hardware resource usage than a monolithic application. We have a possibility to limit our interest only to several components of the application. SOFA component model gives us these possibilities, but it is not trivial to find out the right way how to study and observe a component. With runtime of componentized applications we have to face the overhead created by middleware, which provides necessary communication layer for the components. We need to address a number of questions. How much does the middleware influence the measurements and how can we measure the effects of middleware?

We have to set a base for measurements. We can do so by measuring how are hardware resources consumed by middleware before the component starts to execute anything. In the case we are not able to produce precise results, we can at least observe the correlations. Using correlations, we can predict how will behavior of a particular part of application change when number of method invocations on that part increase or decrease.

6.2 Launching Componentized Application

SOFA framework can launch componentized application in different ways. All the components can either share and run in the same dock or different docks can be used for different components. Then all the docks can run on

the same physical machine or different docks can run on multiple machines. When trying to measure hardware usage of a component, all these possibilities should be taken into account as all of them imply different results.

Components running in one dock do not need to use Java Remote Method Invocation mechanism and therefore less resources are consumed. However as the components run in one dock, they run on one physical machine. This means each component influences every other component in the dock as they share same hardware resources.

If components are run in different docks they need to use Java Remote Method Invocation to call methods on each other's provided interfaces. The advantage is that a single component can be running in one dock and measured in isolation. This can be taken even further and run component that is subject to study on a dedicated machine. This way we have confidence that behavior of such a component is not influenced by consumption of hardware resources by other components. The disadvantage is the middleware layer, which is used to provide communication between components and therefore creating overhead.

6.3 Middleware

It is necessary to identify whether middleware creates a bottleneck when running componentized application. If the component runs on an isolated machine, networking also influences hardware usage and therefore measurement results. Because hardware usage is many times related to number of parallel threads running within the application. The number of parallel threads is determined by number of parallel components communicating with the component or parallel invocations of provided interface. If networking is slowing the number of events that cause method invocations, less resources will be used and the measurements will be also biased by this fact.

6.4 Tests

We were able to execute tests measuring memory usage by the **Store** and **StoreTester** components. We were interested in memory usage related to number of running threads. We needed to implement a thread that would collect statistics of used memory and a thread that would query database.

We implemented **UsedMemoryThread** class to measure memory usage during runtime of the application and take samples every second. It uses **Runtime** Java class, which provides **totalMemory()** and **freeMemory()** methods. The **QueryThread** uses **queryProducts()** method to query database in a loop for products of certain *Store* of the *Trading System*.

The **startTest()** method starts the **UsedMemoryThread** thread and specified number of **QueryThread** threads. Every test runs for a minute and then the number of threads increases.

We believe the memory used by the application will increase in a linear trend related to a number of threads executed, if sufficient amount of memory is provided. It is because every thread allocates the same amount of memory for products retrieved from database by the **queryProducts()** method.

In the case the amount of memory becomes insufficient, the linear trend changes slowly approaching the upper bound. It is because as soon as the amount of used memory crosses certain boundary, the Java garbage collector is invoked more often to lower the amount of used memory under the boundary. Because of higher number of threads running, more memory is consumed and as soon as the garbage collector frees portion of the memory, it is used by a running thread. Following section 6.5 includes the test results with diagrams.

6.5 Test Results

The tests were run with default Java settings, which means 64 MB of memory was dedicated to our application in our case.

Test in figure 6.1 started with 10 threads, increasing the number of threads by 10 every minute and taking samples of the amount of used memory every second.

Test in figure 6.2 started with 50 threads, increasing the number of threads by 50 every minute and taking samples of the amount of used memory every second.

We repeated the test twice measuring how many times is the Java garbage collector invoked in order to prove our results are stable. These tests should be viewed as a proof of concept, how a component resource usage can be measured when running in componentized application. These tests should be further developed to provide more accurate and more detailed results.

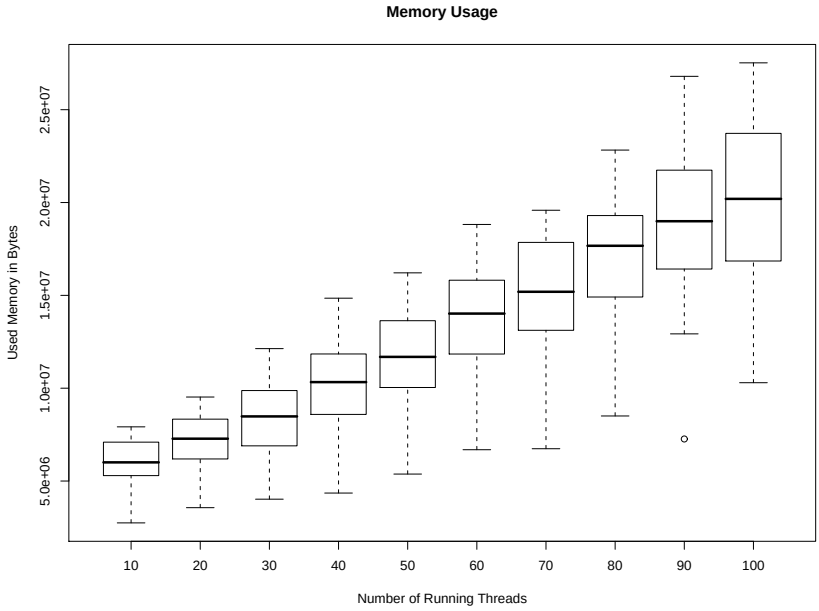


Figure 6.1: Memory usage test increasing number of threads by 10.

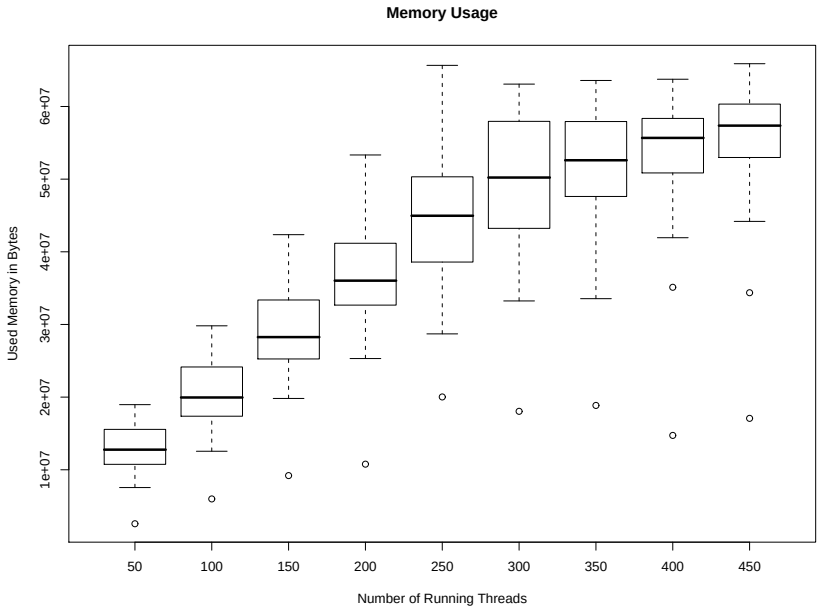


Figure 6.2: Memory usage test increasing number of threads by 50.

6.6 Behavioral Protocols and Tests

In the implementation of CoCoME in Fractal, every component comes with behavior specification, which is described by a behavior protocol as described in [1]. A behavior protocol specifies the behavior and properties of a component. It includes specification how are individual methods of the component invoked (e.g. order of method invocation). The behavior specification can be then verified against component code.

The behavior specification is included in the ADL file of every component frame. It can be used in the future work to design more advanced tests as it provides enough information how are methods of components invoked. Individual test can be prepared for every component from its behavior specification and then it can be launched to measure resource usage of the component. This approach is more straight forward as the tests would otherwise have to be designed from scratch.

Chapter 7

Conclusion

7.1 Implementation of CoCoME in SOFA

Implementation of CoCoME in SOFA is the largest application that was tried to be run in the SOFA framework. Our work shows it is not trivial to convert a monolithic application into a componentized application, as it is not clear which parts should form a component. It is difficult to divide a monolithic application into smaller parts so that they are easily connected together.

We were finally able to launch the implementation of CoCoME in SOFA after the last issue related to inner parts of SOFA framework was resolved by SOFA developers. Because we were dealing with number of technical issues regarding the SOFA framework, we were not able to implement parts of the application that were cutout from the CoCoME in Fractal implementation. However, we provide solutions to number of technical issues discovered during the implementation process, which would remain otherwise hidden. Most of the issues were discovered by the needs of the CoCoME application and it was not related to the way CoCoME was implemented in SOFA. We believe these issues would be discovered by any other application of comparable size to CoCoME application.

CoCoME in SOFA implementation is built from about 30 components, which means about 90 directories with definitions and implementations of interfaces, frames and architectures. All of these need to be configured and manually checked by a developer as there is no support for the application development within SOFA framework apart from the **cushion.sh** tool.

The **cushion.sh** tool helps a lot with creation of the basic directory

structure and ADL files. However, it does not provide any way to check if the ADL files are correctly filled in. This could be done automatically by going through all the ADL files and checking if all paths to classes with implementation are correct and if source code is present. If there is an error found, the developer can be informed by a message indicating which component needs to be checked.

7.2 What Was Done

The implementation of CoCoME in SOFA is not fully finished due to various technical issues. But we believe we resolved most of the issues during our effort and it will be valuable for any future application developments in the SOFA framework. We provide a list of what was done:

- Complete CoCoME architecture definition in the SOFA format.
- Skeleton implementation of all the CoCoME components.
- Simulator component creation and implementation.
- Store and StoreTester components implementation.
- Investigation of class sharing issues and temporary resolution.
- Investigation of libraries integration issues and temporary resolution.
- Hibernate system integration with SOFA framework.
- Investigation of persistence classes and class renaming issue and temporary resolution.
- Launching the implementation of CoCoME in SOFA.
- Launching basic memory usage test with Store and StoreTester components using Hibernate and Derby database.
- Suggestions how to solve remaining technical issues.

Chapter 8

CD Content

CD with CoCoME in SOFA implementation and CoCoME Store in SOFA implementation is included in the end of the thesis. It also includes cited articles and documentation used during implementation and writing of the thesis.

The articles and documentation are included in the **Articles and Documentation** directory. The CoCoME in SOFA implementation is included in **cocome.original** directory. The CoCoME Store in SOFA implementation is included in **cocome.store** directory. Each of these directories also include **sofa** directory with the SOFA repository and **doc** directory with documentation. The **cocome** directory includes **cocome-impl** directory with the original monolithic implementation of CoCoME, **fractal** directory with the implementation of CoCoME in Fractal, **sofa** directory including the conversion script used to convert CoCoME in Fractal to SOFA.

The manual with instruction how to launch both implementations is included in the **readme.txt** file. The thesis is included in the **CoCoME in SOFA.pdf** file.

Appendix A

List of Components

A.1 Interfaces

sofa2.BankIf
tradingsystem.AccountSaleEventHandlerIf
tradingsystem.CashDeskConnectorIf
cashdesk.CardReaderControllerIf
cashdesk.CardReaderEventDispatcherIf
cashdesk.CardReaderEventHandlerIf
cashdesk.CashBoxControllerIf
cashdesk.CashBoxEventDispatcherIf
cashdesk.CashBoxEventHandlerIf
cashdesk.CashDeskAppEventDispatcherIf
cashdesk.CashDeskAppEventHandlerIf
cashdesk.CashDeskApplicationIf
cashdesk.CashDeskGUIIf
cashdeskline.CashDeskEventDispatcherIf
cashdesk.GUIEventHandlerIf
cashdesk.LightDisplayEventHandlerIf
cashdesk.PrinterEventHandlerIf
cashdesk.ScannerControllerIf
cashdesk.ScannerEventDispatcherIf
cashdeskline.CoordinatorEventDispatcherIf
cashdeskline.CoordinatorEventHandlerIf
inventory.MoveGoodsIf
inventory.ProductDispatcherIf

shared_data.EnterpriseQueryIf
shared_data.PersistenceIf
shared_data.StoreQueryIf
reportingapplication.ReportingIf
storeapplication.StoreIf

A.2 Frames

Application.BankFrame
sofa2.ApplicationFrame
Application.SimulatorFrame
CashDeskLine.CashDeskLineBusFrame
CashDeskLine.CoordinatorFrame
TradingSystem.CashDeskLineFrame
Application.TradingSystemFrame
Inventory.EnterpriseServerFrame
EnterpriseServer.ProductDispatcherFrame
TradingSystem.InventoryFrame
Inventory.StoreServerFrame
StoreServer.StoreApplicationFrame
StoreApplication.StoreGUIFrame
StoreApplication.StoreLogicFrame
CashDesk.CardReaderControllerFrame
CashDesk.CashBoxControllerFrame
CashDesk.CashDeskApplicationFrame
CashDesk.CashDeskBusFrame
CashDesk.CashDeskGUIFrame
cashdesk.CashDeskFrame
CashDesk.LightDisplayControllerFrame
CashDesk.PrinterControllerFrame
CashDesk.ScannerControllerFrame
Data.EnterpriseFrame
shared_data.DataFrame
Data.PersistenceFrame
Data.StoreFrame
shared_reportingapplication.ReportingApplicationFrame
ReportingApplication.ReportingGUIFrame
ReportingApplication.ReportingLogicFrame

A.3 Architectures

sofa2.ApplicationArch
Application.BankArch
Application.SimulatorArch
Application.TradingSystemArch
TradingSystem.CashDeskLineArch
CashDeskLine.CashDeskLineBusArch
CashDeskLine.CoordinatorArch
TradingSystem.InventoryArch
Inventory.EnterpriseServerArch
EnterpriseServer.ProductDispatcherArch
Inventory.StoreServerArch
StoreServer.StoreApplicationArch
StoreApplication.StoreGUIArch
StoreApplication.StoreLogicArch
cashdesk.CashDeskArch
CashDesk.CardReaderControllerArch
CashDesk.CashBoxControllerArch
CashDesk.CashDeskApplicationArch
CashDesk.CashDeskBusArch
CashDesk.CashDeskGUIArch
CashDesk.LightDisplayControllerArch
CashDesk.PrinterControllerArch
CashDesk.ScannerControllerArch
shared_data.DataArch
Data.EnterpriseArch
Data.PersistenceArch
Data.StoreArch
shared_reportingapplication.ReportingApplicationArch
ReportingApplication.ReportingGUIArch
ReportingApplication.ReportingLogicArch

Appendix B

Project Timeline

- **5.3. - 6.3.**
  Studying articles.
- **11.3. - 13.3.**
  Analyzing CoCoME.
- **17.3. - 19.3.**
  Studying SOFA framework.
- **25.3. - 26.3.**
  Converting the CoCoME in Fractal implementation to SOFA framework.
- **31.3. - 2.4.**
  Proceeding in the conversion process and implementing setRequired() methods.
- **7.4. - 9.4.**
  Proceeding in the conversion process solving issues with Simulator part of the application.
- **14.4. - 16.4**
  Duplicate classes issue in SOFA framework reported, it is not possible to launch the CoCoME in SOFA implementation until the issue is fixed in SOFA framework. Starting to work on Store and StoreTester implementation.

- **21.4. - 23.4.**
  Proceeding with the implementation of Store and StoreTester, solving issues with Hibernate libraries while integrating it into SOFA framework.

- **28.4. - 30.4**
  Proceeding with the implementation of Store and StoreTester, solving issues with Hibernate persistence.xml while integrating it into SOFA framework.

- **5.5 - 6.5.**
  Proceeding with the implementation of Store and StoreTester, solving issues with Hibernate persistent classes caused by renaming process of SOFA framework. Duplicate classes issue circumvented. Connector generation issue in SOFA framework reported, not able to launch CoCoME in SOFA implementation until the issue is fixed.

- **7.5.**
  Writing bachelor thesis.

- **12.5. - 14.5.**
  Writing bachelor thesis.

- **23.5. - 24.5.**
  Implementing Store and StoreTester memory usage test and conducting measurements. Connector generation issue in SOFA framework fixed and implementation of CoCoME in SOFA successfully launched.

- **25.5 - 26.5.**
  Finishing bachelor thesis and preparing for presentation of the results to the Distributed Systems Research Group.

Bibliography

[1] Bulej, L., Bures, T., Thierry Coupaye, Decky, M., Jezek, P., Parizek, P., Plasil, F., Poch, T., Nicolas Rivierre, Sery, O., Tuma, P.: *CoCoME in Fractal*, Chapter in The Common Component Modeling Example: Comparing Software Component Models, Springer-Verlag, LNCS, Apr 2008

[2] Bures, T., Decky, M., Hnetynka, P., Kofron, J., Parizek, P., Plasil, F., Poch, T., Sery, O., Tuma, P.: *CoCoME in SOFA*, Chapter in The Common Component Modeling Example: Comparing Software Component Models, Springer-Verlag, LNCS, Apr 2008

[3] Bures, T., Hnetynka, P., Plasil, F.: *Runtime Concepts of Hierarchical Software Components*, In International Journal of Computer & Information Science, Vol. 8, No. S, ISSN 1525-9293, pp. 454-463, Sep 2007

[4] Bures, T., Hnetynka, P., Plasil, F.: *SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model*, Proceedings of SERA 2006, Seattle, USA, IEEE CS, ISBN 0-7695-2656-X, pp. 40-48, Aug 2006

[5] Bures, T., Hnetynka, P., Plasil, F., Klesnil, J., Kmoch, O., Kohan, T., Kotrc, P.: *Runtime Support for Advanced Component Concepts*, Proceedings of SERA 2007, Busan, Korea, IEEE CS, ISBN 0-7695-2867-8, pp. 337-345, Aug 2007

[6] Derby Database (http://db.apache.org/derby/)

[7] Sebastian Herold, Holger Klus, Yannick Welsch, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, Christian Pfaller: *CoCoME - The Common Component Modeling Example*

[8] Hnetynka, P., Tuma, P.: *Fighting Class Name Clashes in Java Component Systems*, Proceedings of JMLC 2003, Klagenfurt, Austria, Copyright (C) Springer-Verlag, Berlin, LNCS2789, ISSN-0302-9743, pp. 106-109, Aug 2003

[9] Hnetynka, P., Tuma, P.: *Managing Class Names in Java Component Systems with Dynamic Update*, Tech. Report No. 2003/2, Dep. of SW Engineering, Charles University, Prague, Feb 2003

[10] JBoss (Red Hat Middleware): Hibernate. (http://www.hibernate.org)

[11] Objectweb: SOFA (http://sofa.objectweb.org)