ECONET Project

TEST1 BENCHMARK - ANNOTING A JAVA PROGRAM FROM UML MODELS

Version 1.0

Pascal ANDRE[1]

September 29, 2008

supported by

[1]COLOSS - LINA - FRE CNRS 2729 - 2, rue de la Houssinière, B.P.92208, F-44322 Nantes Cedex 3, France

Abstract

This document presents the `Test1` benchmark for the ECONET project. It is defined as a small subset the CoCoME benchmark. This document include the component description, the Java code implementation and the annotation discovery and insertion.

Chapter 1

Introduction

In the context of the Econet project[1], we decided during the workshop of Prague [ACPR07] to use a common component application benchmark. DSRG proposed the COCoME contest. DSRG already participated to the contest by providing studies on SOFA and Fractal based solutions. The base information is available at
http://agrausch.informatik.uni-kl.de/CoCoME
and the results of the CoCoME contest will be published soon [RRMP08].

The whole benchmark is too big to serve as support for the experimentations. During the workshop of Nantes [ACPR08] we restricted the experimentation field according to the following he constraints:

- The selected subset must be large enough to include representative examples for each subproject (concepts and constraints for the metamodel, primitive component for the behaviour abstraction, primitive and also composite components for the structural abstraction.
- The selected subset must be as small as possible to avoid time consuming instanciations.
- The slice is vertical (UML model and Java code).

We retain two included subsets related to two deadlines:

- *Test1*: The `CashDesk` composite component for the structural abstraction. We retain two included subsets:
  - The `CashDesk` composite component for the structural abstraction.
  - The `CashDeskApplication` primitive component, which is a component of the `CashDesk` composite component that helds a dynamic behaviour.
- *Test2*: The `CashDeskLine` composite component, which is the front-end subsystem of the application.

This document summarises an experimentation of the `Test1` benchmark for the ECONET project, which is a subset of the CoCoME case study. The starting point include a (UML) component model, a Java code and annotations. The experimentation goal are (1) to study the link between the component level and the implementation level (how the implementation is close or far from the component model), (2) to investigate the discovery of annotations from UML descriptions and Java programs (how can we use manually or systematically UML informations to find the annotations, what to look for in the Java programs). Writing and exploiting the annotation is outside the scope of this study.

The document is structured as follow. In section 3 we overview the component model informations and the way to use them. In section 4 we just overview the (java) implementation model. Last in section 5 we provide annotations to link both models. But first lets relate a previous annotation experimentation.

[1] http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:start

Chapter 2

Previous Experimentation

Experimentation tests were led with a small subset of the CoCoME case study by a group of students in march 2008. The project is summarised in chapter 3 of [ACPR08] and in a project report. The tests were based on the three following components present the CoCoME case study: the `:CashBoxController`, `:PrinterController` and `:ScannerController` components. These three components are contained in the component `:CashDesk` (Fig. 2.1).

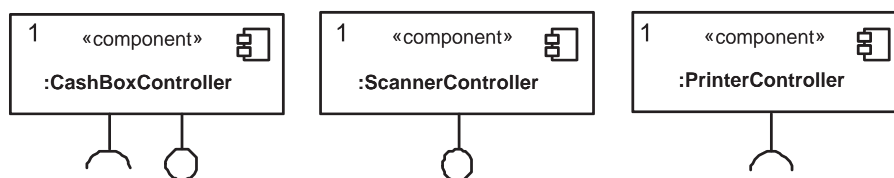

Figure 2.1: Master Project: CoCoME subset

Each component has a package name beginning with `org.cocome.tradingsystem`. The package name is hierarchised according to the composition of the name of the composite components which contain the component. There are a Java interface for the component and also a folder named `impl` which contains the java classes that implement the interface. (Fig. 2.2).

The test proceeded as follow:
(1) a single class was tested that contained all the annotation. This class allowed to generate the structure that was required for the generation of the model.
(2) this structure was then used to instantiate the metamodel.
(3) the structure was exported to another structure to the part that wrote the annotations.
(4) the writing annotation processor inserted the annotation corresponding to the intanciated model in Java classes that are not annotated.
(5) the students checked that the automatic annotated classes are exacly the same that the classes that we annotated manually.

The processor was implemented using the APT tool. The above experimentation was driven by the first version of the Java annotations even if the multiple annotation sources was introduced in several annotation definitions.

```
@InComponent(annotation_scr = {"Manual"}, component_name = "CashDesk")
public class CashBoxControllerEventHandlerImpl implements MessageListener,


        CashBoxControllerEventHandlerIf {

    final String CHANNEL_CONNECTION_FACTORY = "ChannelConnectionFactory";

    private String topicName;

    private Context jndiContext;

    private TopicPublisher cashBoxPublisher;

    private TopicSession topicSession;

    private Logger log = Logger
            .getLogger(CashBoxControllerEventHandlerImpl.class);

    @Businessattribute(annotation_scr = {"Manual"})
    private CashBox cashbox;

    @Initmethod(annotation_scr = {"Manual"}, name_of_the_component = "CashBox")
    protected CashBoxControllerEventHandlerImpl(CashBox cashbox,
            String eventchannel) {
        try {
            this.cashbox = cashbox;

            topicName = eventchannel;

            jndiContext = new InitialContext();
```

Figure 2.2: Master Project: One class of CoCoME annoted

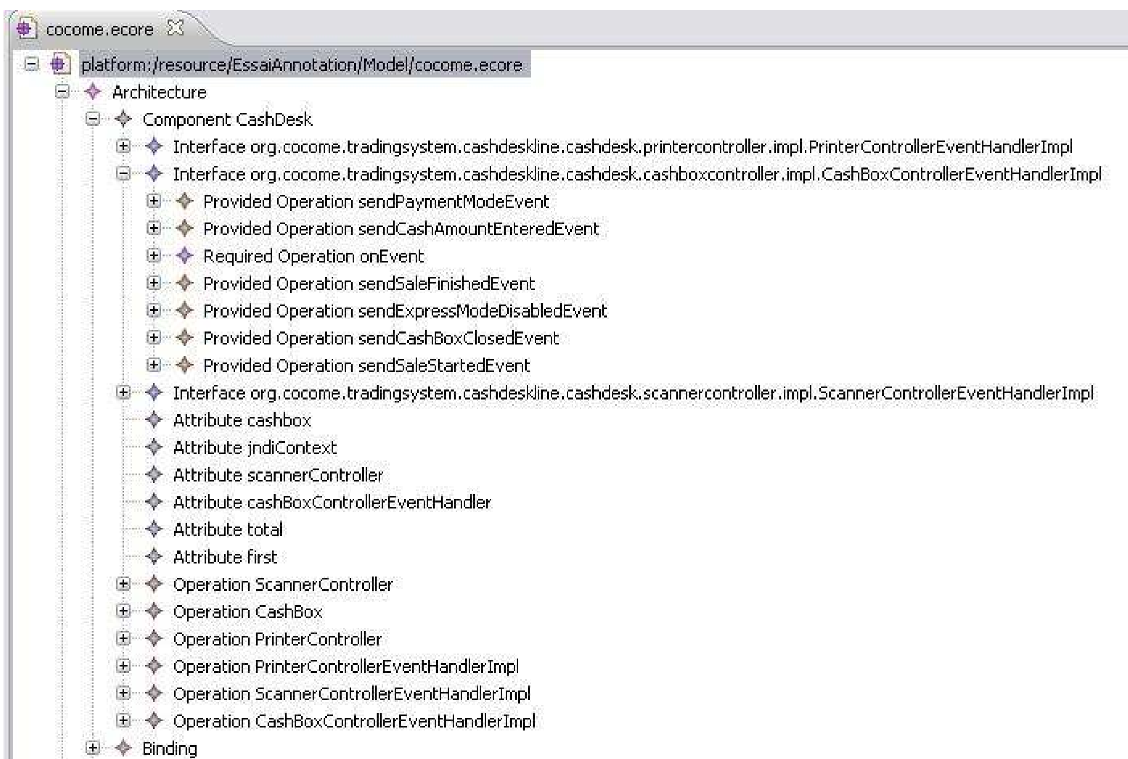

Figure 2.3: Master Project:Extract of the Annotated Class and the CoCoME generated model

Chapter 3

Component Model

We do realy have a full component model as input: we have a structural model provided as UML component diagrams and also a behavioural model provided as a collection of UML sequence diagrams.

3.1 Structural Component Model

The CoCoME structural reference model in defined by a UML component diagram [1]. This is just a MS Visio drawing and not a rich one generated from a UML CaseTool [2]. Therefore if we want to reverse engineer the Java code using UML the UML informations, this model has to be drawn again.

Fig. 3.1 illustrates the structural view of the components for the subset we kept in test1.

The structural UML models contain the component instances, the composite relations, the ports and interfaces. Only component type names, stereotype and number of occurrence are provided. Ports support interfaces which are named and decribed as Java interfaces. The UML notes (comments) show the datatypes (*entity* classes) wihch are relevant for the corresponding interfaces. This is true for the `Inventory` related components. There are no such entities for the `CashDesk` related components. But additional informations (these rectangle boxes are a new UML notation ?) is used to denote events associated to interfaces. These events coming from uses case definitions and sequence diagrams. Curiously the send events are associated to provided interfaces and the "*The semicircles indicate events the component can handle while the circles indicate events which are sent by the component. For example, the controller* `CardReaderController` *handles the event* `ExpressModeEnabledEvent` *while sending the events* `CreditCardScannedEvent` *and* `PINEnteredEvent`."

The internal specification of components seems to be an implementation issue (or a refinement if available).

[1] see chapter 3 - CoCoME - The common Component Modelling Example available in http://cocome.org/

[2] We mean the one we could process to get useful informations for the abstraction process A and B.

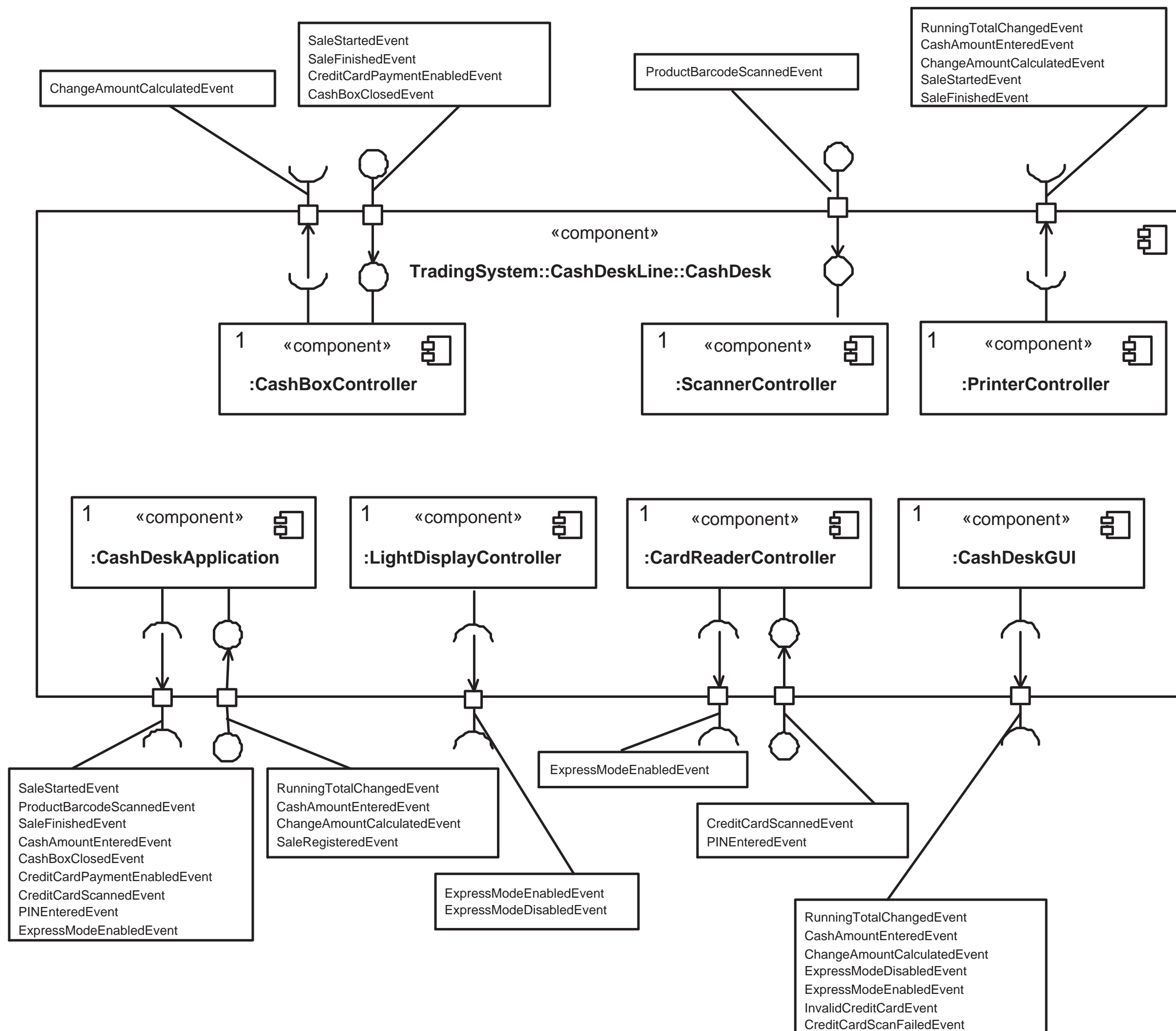

Figure 3.1: CoCoME subset 1: the `CashDesk` component

3.2 Behavioural Component Model

The behavioural model is not related to components (staemachines could play that role) but is defined informally with use case descriptions and UML sequence diagrams. The events attached to the interfaces in the structural model are message send in the sequence diagrams. The elements related to `Test1` are those of the `ProcessSale` use case extract (Fig. 3.2).

The `ProcessSale` use case is illustrated by one main scenario (Fig. 3.3) and two referenced scenarii (Fig. 3.4 and 3.5). These are described in [Reu06].
UC `ManageExpressCheckout` is an extension of of the `ProcessSale` use case.Its role is to hold a statistic about sales. It is not of interest for `Test1`.

The sale process starts when the `Cashier` presses the button *Start Sale* at his *Cash Box*. Then the corresponding software component `CashBox` calls a method at the component `CashBoxController` which publishes the *SaleStartedEvent* using the *cashDeskChannel*. The three components `CashDeskApplication`, `PrinterController` and `CashDeskGUI` react to events of the kind *SaleStartedEvent*. In order to receive these they have to register themselves at the channel *cashDeskChannel* for these events and implement the according event handlers. At the *Cash Desk* the *Printer* starts printing the header of the receipt initiated by the component `PrinterController` and initiated by the component `CashDeskGUI` a text at the *Cash Desk* indicates the start of a new sale.

Some components connected with the channel *cashDeskChannel* implement a finite state machine, like `CashDeskApplicati`

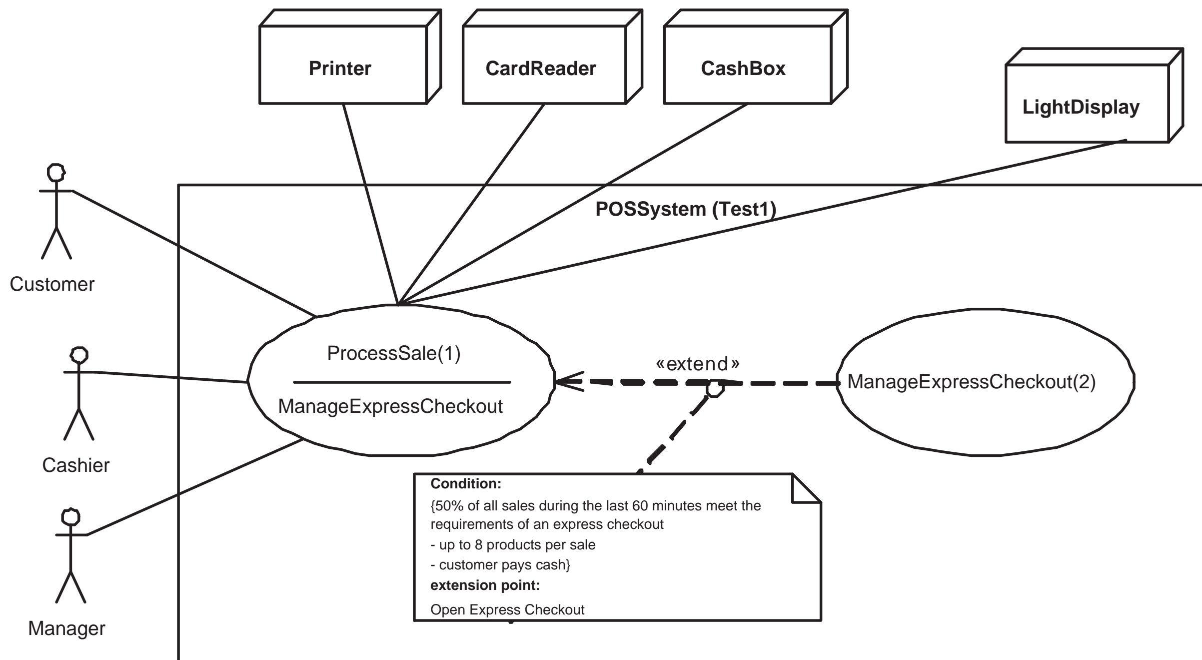

Figure 3.2: CoCoME subset 1: the `ProcessSale` UC

or `PrinterController` in order to react appropriately on further incoming events. In the next phase of the selling process the desired products are identified using the *Bar Code Scanner* which submitts the data to the corresponding controller `ScannerController` which in turn publishes the event *ProductBarCodeScannedEvent*. The component `CashDeskApplication` gets the product description from the `Inventory` and calculates the running total and announces it on the channel. After finishing the scanning process, the *Cashier* presses the button *Sale Finished* at the *Cash Box*. Now the *Cashier* can choose the payment method based on the decision of the costumer by pressing the button *Cash Payment* or *Card Payment* at his *Cash Desk*.

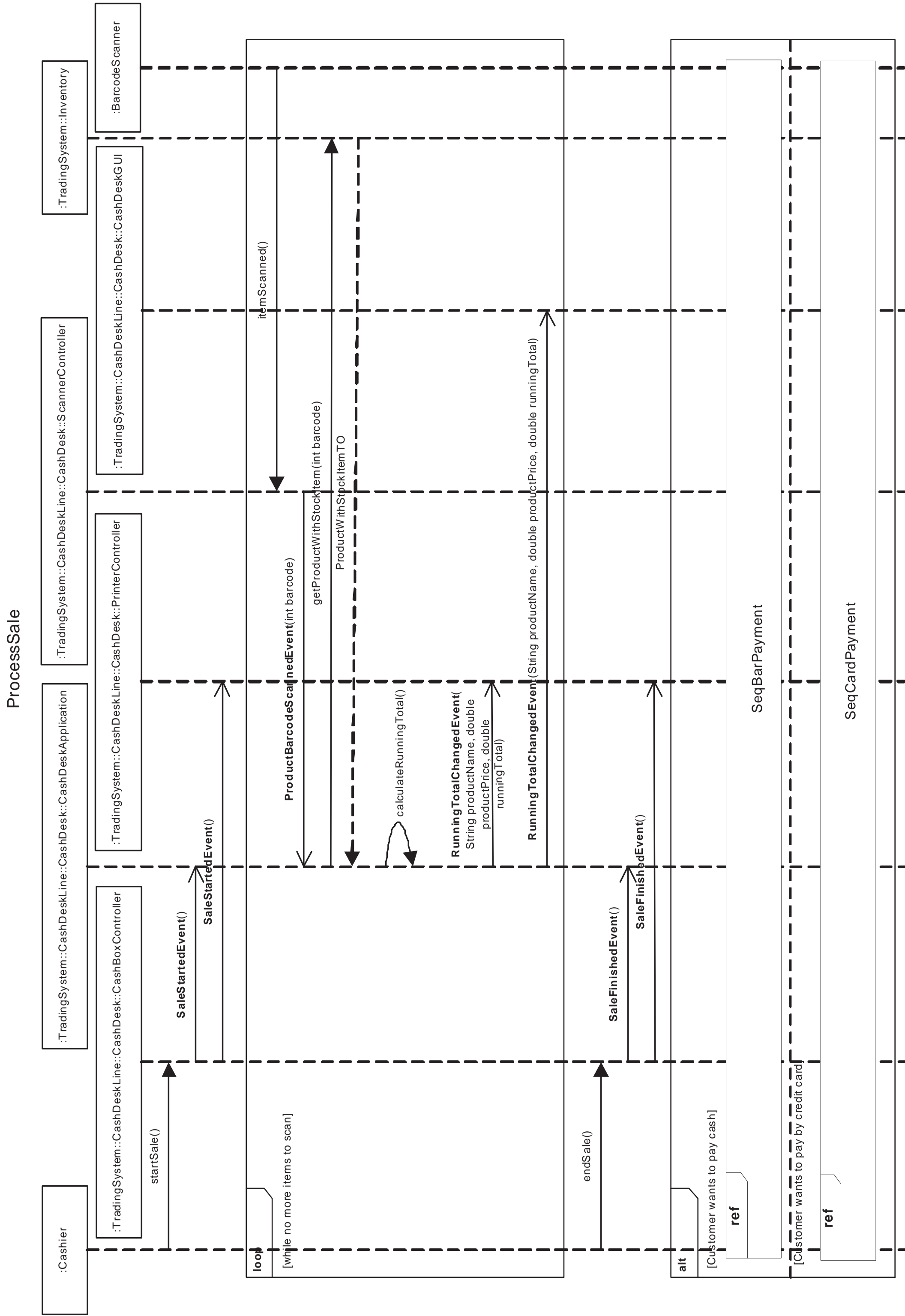

Figure 3.3: CoCoME subset 1: the `ProcessSale` interactions

Fig. 3.4 and 3.5 illustrate the sequences for each payment method which shall not described in detail here.

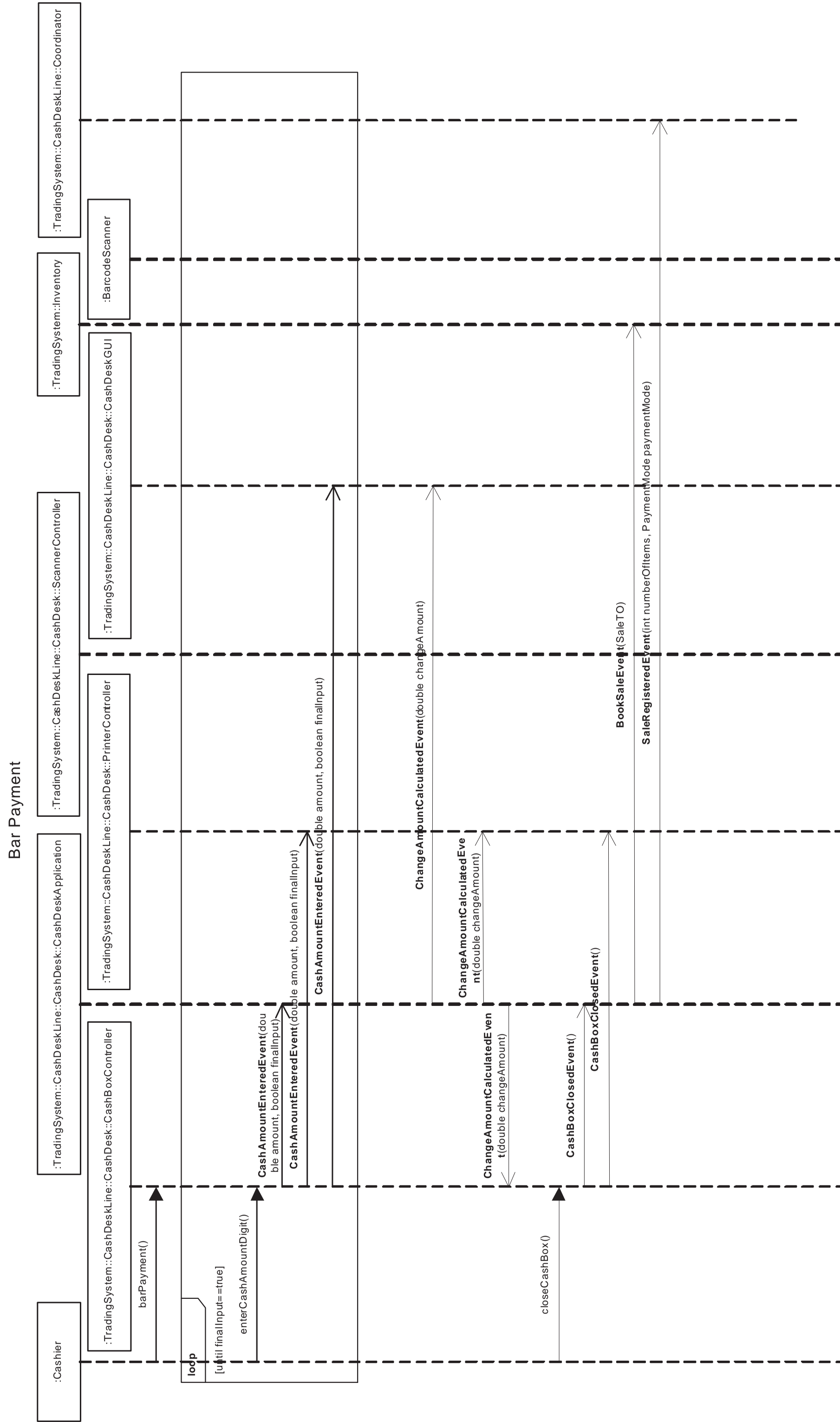

Figure 3.4: CoCoME subset 1: the ProcessSale sub-interactions 1/2

contd.

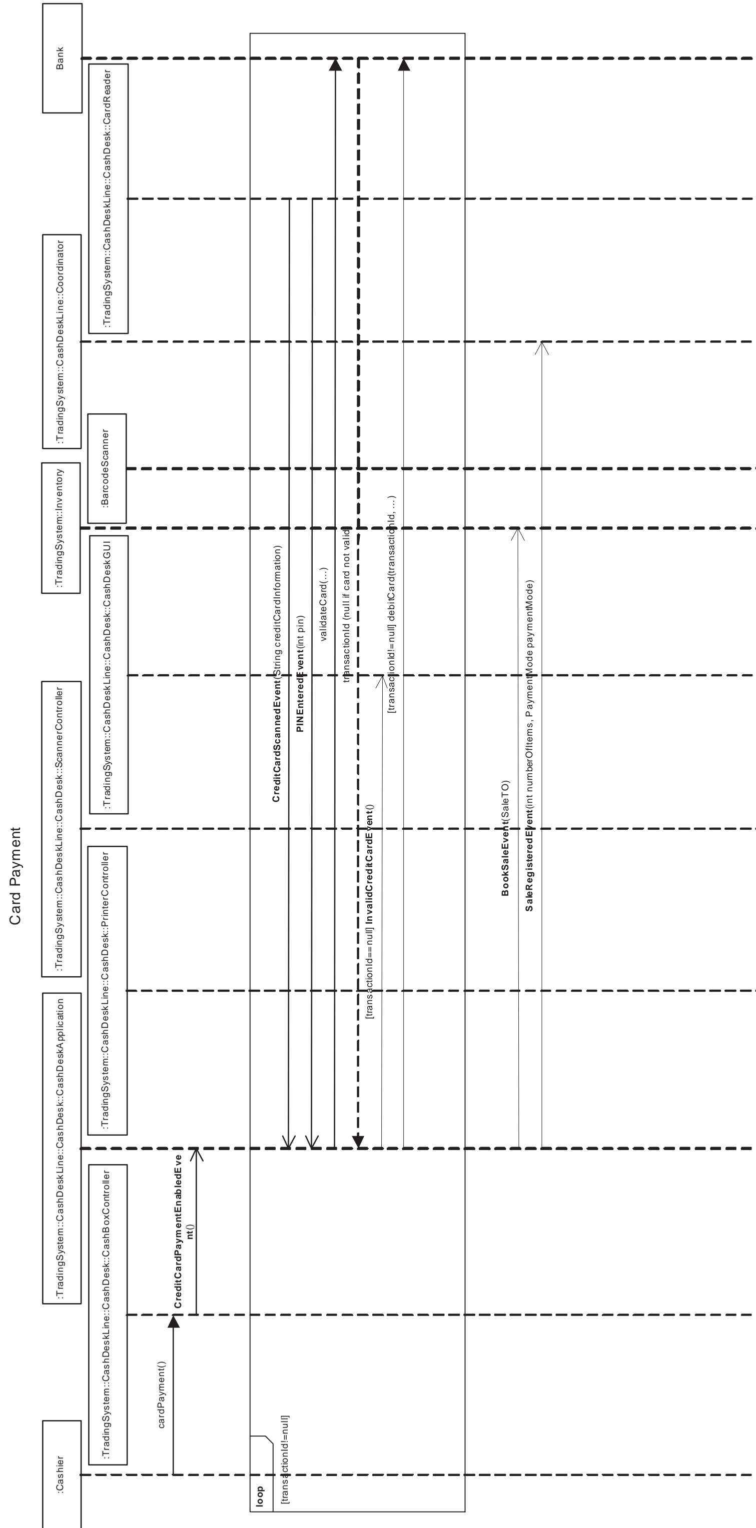

Figure 3.5: CoCoME subset 1: the `ProcessSale` sub-interactions 2/2

contd.

Chapter 4

Implementation Model

Figure 4.1 shows a UML representation of the Java implementation of the `CashBoxController` component model.

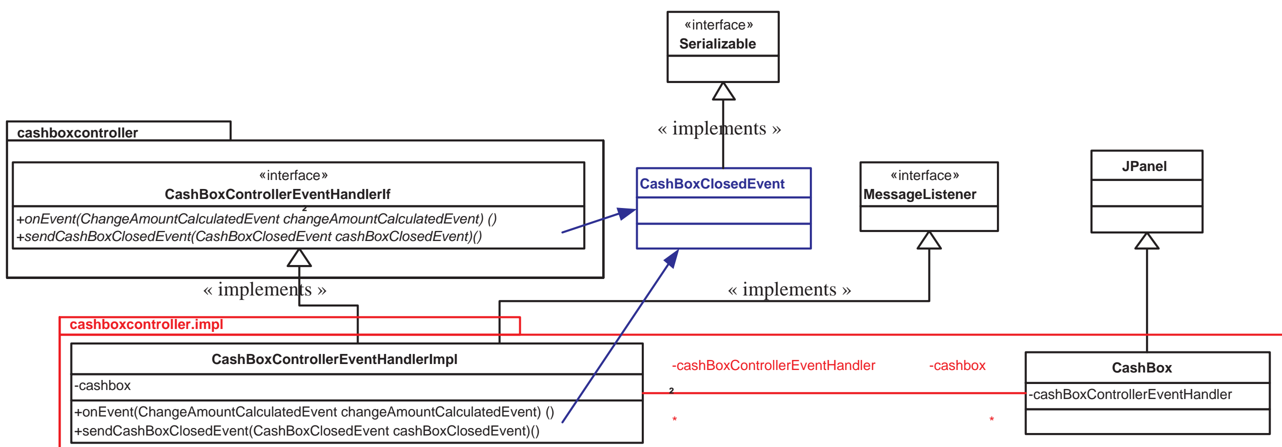

Figure 4.1: CoCoME component: the `CashBoxController` implementation

Fig. 4.2 illustrates the implementation for the subset we kept. It is a set of Java classes.

The main design decisions focus on the implementation of the `:EventBus` component which is based on the JMS API. The Java Message Service (JMS) API is a messaging standard that allows application components based on the Java 2 Platform, Enterprise Edition (J2EE) to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous. A JMS application is composed of the following parts:

- JMS Clients - These are the Java language programs that send and receive messages.
- Non-JMS Clients - These are clients that use a message systemŠs native client API instead of JMS. If the application predated the availability of JMS it is likely that it will include both JMS and non-JMS clients.
- Messages - Each application defines a set of messages that are used to communicate information between its clients.
- JMS Provider - This is a messaging system that implements JMS in addition to the other administrative and control functionality required of a fullfeatured messaging product.
- Administered Objects - Administered objects are preconfigured JMS objects created by an administrator for the use of clients.

The main interfaces of the API are: `Connection`, `Session`, `Message`, `MessageProducer`, `MessageListener`.

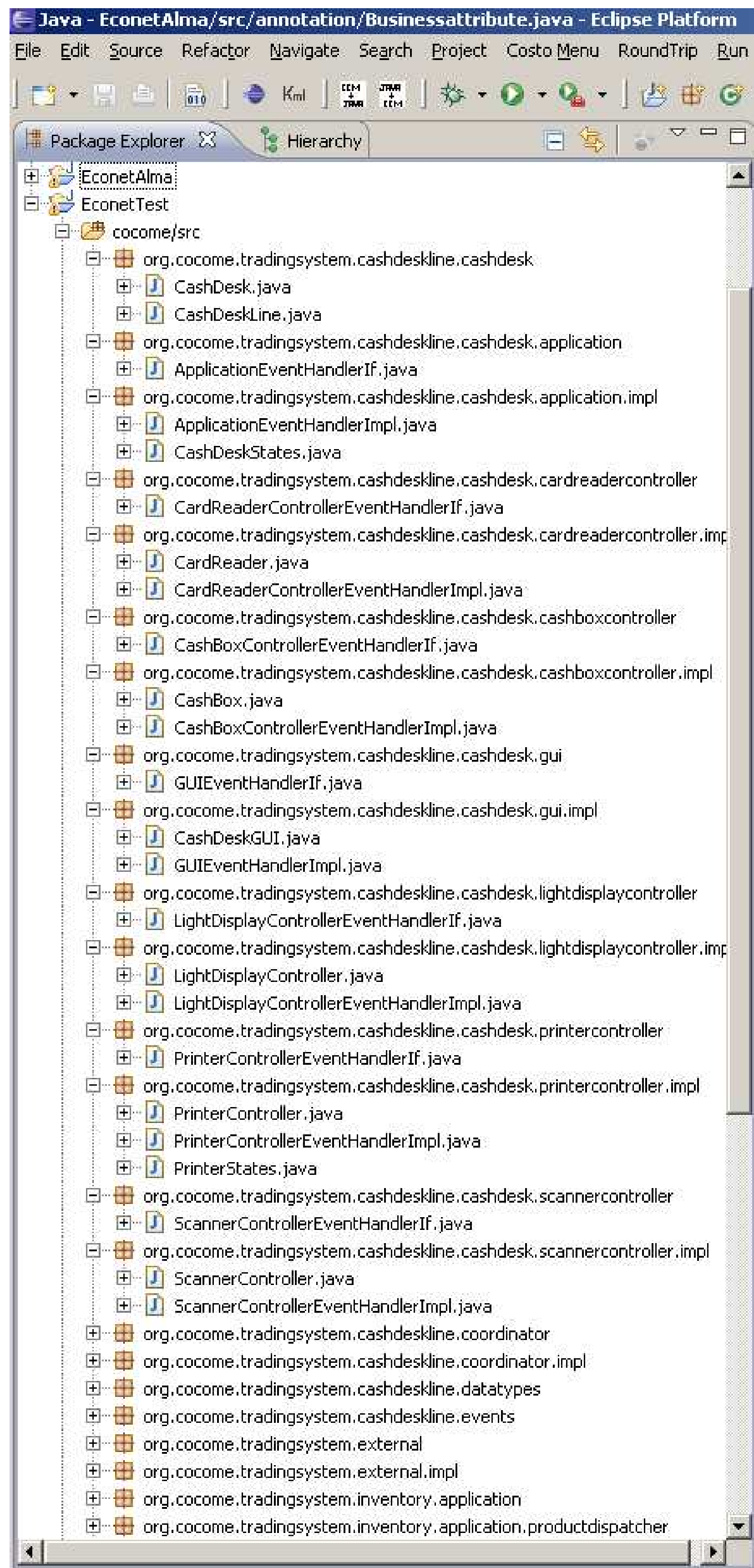

Figure 4.2: CoCoME subset 1: the `CashDesk` java implementation

Chapter 5

Finding and Writing the Annotations

In this section we discuss about the process of annoting a Java code from component information. The annotating process is composed of two processes : finding the annotation (*Annotation Discovery*) and writing the annotations. *We only deal with the first one.* The second one will be done manually.

5.1 Assumptions

At this point we assume that there is some component model description (here in UML2) and a Java implementation of it. We also assume that the Java code is not a component model transformation (no code generation) and it does not include component annotations. This is the case in the CoCoME example.

The annotations must conform to the current Java annotation definition for Econet. These are detailed in appendix A.

5.2 Annotation Discovery

Assuming a UML component model and a Java implementation, the main goal of the annotation process is to map concepts between the models.

5.2.1 Annotation Templates

An annotation is a link between a model element and an implementation element. This is some kind of the one we discussed (Fig. 5.1).

The useful UML (component or not) concepts are: component, composition, class (e.g. parameters or parts of components), operations with signatures, types, interfaces, ports, connectors, stereotypes, instances (objects can be interpreted component instances), messages with parameters, ...

5.2.2 Mapping the concepts

Since the implementation is more concrete and more detailed than the abstract model and the process must start from the abstract component model. This not an easy process.

There are no direct transformation (in the sense of MDA) from UML (abstract) component model to Implementation models. So it is not possible to *trace*[1] the model elements.

Moreover (abstract) component model are usually definitely not UML component diagrams but rather a collection of UML diagrams. The CoCoME example can be a good example of legacy component systems: the collection of UML diagrams is a documentation that helps to uinderstand the Java (component) implementation but it is not its abstract description. The main difficulty to set the annotations is that there are no complete component reference model to define what are business features and implementation features. The full model is provided by UML documentation but also the system Java implementation. The UML models are only an overview with many holes. We have to look at the implementation source to define some of the component model features.

[1] Traceability is a mapping that fits to our needs.

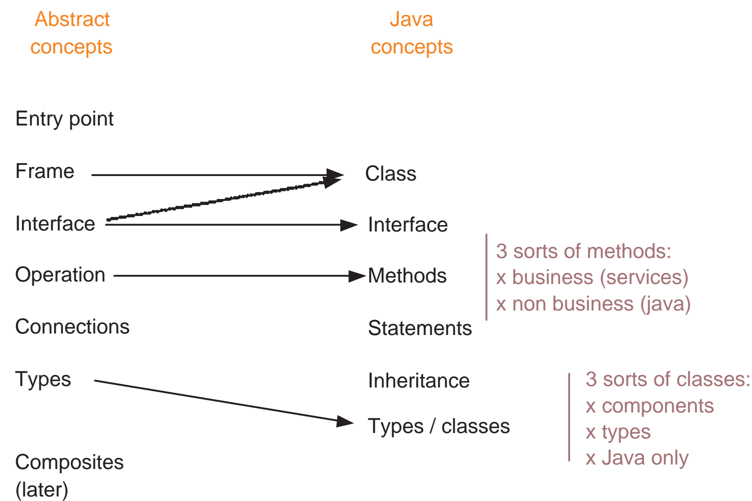

Figure 5.1: Mapping concepts

Indeed, missing Component information can be infered from various source code elements. But some component model elements names are changed during the manual implementation: for example the `CashBoxController` does not exist as so in the Java implementation.

5.2.3 Implementation Patterns

An interesting solution is to draw some transformation patterns in ordre to set the mappings. We will see examples in section 5.3.

5.2.4 Automatic Annotation Inference vs Manual Inference

Automatic annotation inference is based on concept names and string processing. Thus it requires a component model textual description provided by some software engineering casetool (XMI or text format). In our case we only have Visio diagrams. So we will lead a manual inference. But the way is still open for later study because we try to find some implementation patterns.

5.3 Annotating the CoCoME

In the `Test1` benchmark, the annotations are put manually in the Java code. Only the classes related to the `Test1` subset are annotated.

5.3.1 Finding Mappings

First we look at the structural (component diagram) and after to the dynamic view. In each case we try to find a correspondence.

5.3.2 Structural Component Model Analysis

The static model includes anonymous component instances, interfaces, ports and connection information, component compositions. The connection information are not formalised but we find events (used in the behavioural model) or data (data model). Component types are not really specified but the structure is the same at the instance level and the type level.

Mapping UML Components to Java

The components and interfaces are linked to packages and classes. The name are not always identical but manual inference is quite easy. We try to find patterns for the mappings.

As an example, let's take the `CashBoxController` component of Figure 3.1. The component provides an anymous provided interface and an anymous required interface. Events (types) are related to these interfaces.

Figure 5.2 shows the mapping between the component model (extracted from Figure 3.1) and the implementation model (Figure 4.1).

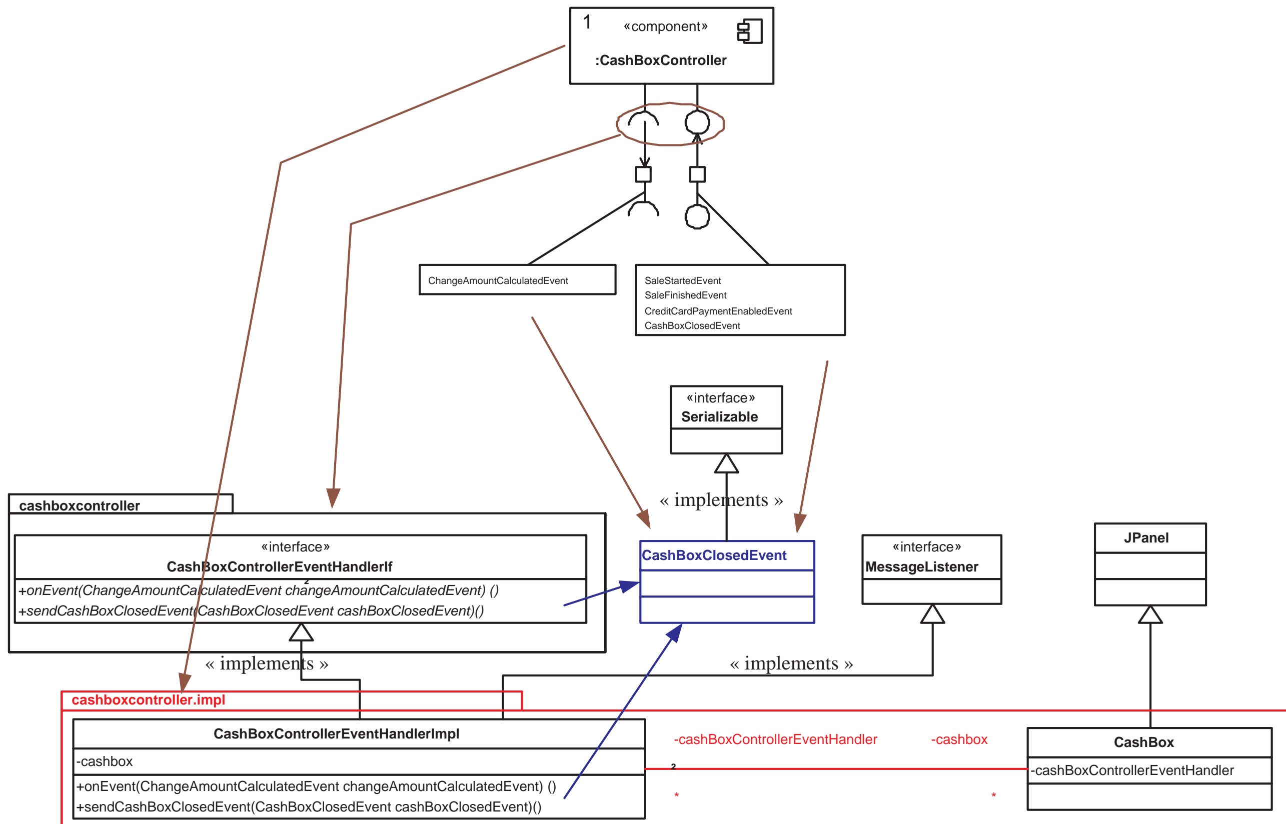

Figure 5.2: CoCoME component: the `CashBoxController` implementation mapping

Component Pattern A component `C1` is implemented by a package `C1` including

- the `C1`*EventHandlerIf* interface
- the `C1`.*impl* package including
  - the `C1`*EventHandlerImpl* class that implements the `C1`*EventHandlerIf* interface,
  - the `C1'` class that implements a GUI part.

These classes includes corresponding attributes that can be represented by a UML bidirectional association.

Interface Pattern Provided and required interfaces are merged and the method name is an indication of whether events are sent (provided interface) or received (required interface).

The anonymous interfaces of component `C1` are implemented by the `C1`*EventHandlerIf* interface where

- a required event `REvt` (operation ? service ?) is defined by a `onEvent(REvt rEvt);` method. Example: `void onEvent(ChangeAmountCalculatedEvent changeAmountCalculatedEvent);`

- a provided event `PEvt` (operation ? service ?) is defined by a `sendPEvt(PEvt pEvt);` method. Example: `void sendCashBoxClosedEvent(CashBoxClosedEvent cashBoxClosedEvent);`

The `C1`*EventHandlerIf* interface is then implemented by the `C1`*EventHandlerImpl* class.

Service/operation Pattern The implicit convention is to interpret events as services (operations) such that emitting events is providing services and receiving events is requiring services.

But this rule is not followed systematically. New events appear that issued from the behavioural component model *e.g.* `sendPaymentModeEvent`, `sendExpressModeDisabledEvent`. Some events are not implementd as so *e.g.* `CreditCardPaymentEnabledEvent`.

Events are specified by classes in the `cashdeskline.events` package.

Composition Patterns From a scope (naming/lexical) point of view, component packages are included in the composite package but except to this there are no true representation of composition:

- ports are not explicitely represented (no promotion: the subcomponent are directly connected) because
- interfaces are shared by the component and its composite,
- there are no object composition *e.g.* by instance variable declaration.

In the `CashDeskLine` example, the `CashDeskLine` class is grouped with the `CashDesk` class in the the `cashdesk` package. The `EventBus` is not implemented as so but rather via the Java GUI. So there are no direct mapping.

Looking outside the scope

The above mapping seems to be convenient for `CashDeskLine` subcomponents even if name inference is quite difficult because the rules are evolving. For example the component `CashDeskApplication` is implemented by the package `application` including the `ApplicationEventHandlerIf` interface and the `application.impl` package `C1` including the `ApplicationEventHandlerImpl` and `CashDeskStates` classes.

Moreover the above components are somewhat related to dynamic aspects of the model. In the case of `Inventory` components (Figure B.1 of appendix B.1) *e.g.* `Application` or `Store`, the rules look like different.

- Each interface is implemented by a Java interface. A required interface is in fact an exact matching of a provided one in another component (*it refers to a provided one*). The package importations solve the interface linking (this is an explicit promotion/delegation that do not respect composition encapsulation).
- A (primitive ?) component `C1` is implemented by a class `C1`*Impl* class of the `C1`.*impl* package, such that
  - `C1`*Impl* implements the (Java) provided interfaces,
  - `C1`*Impl* declares an instance variable (attribute) for each required interface (it is initialised using a factory).
- Here events are replaced by datatype (implemented by Java classes).
- An application factory design pattern is used.
- The component model includes "business" data types modeled by classes, implemented in the same package as the component.

5.3.3 Behavioural Component Model Analysis

It is important to note that message names are built using some conventions. In the `CashDeskLine` they all end by `Event` and some start with `send`

Message Patterns

The idea is to project message send end receptions on each lifeline of the sequence diagrams according to the naming convention given in the above sections.

We find the same mismatches.

Unfortunately the sequence diagrams are not numerous to imagine a systematic and automatic discovery process?

5.3.4 Entry Point Patterns

Statically, one can have a look the constructors or the `main` methods. Dynamically one can have a look at the top level sequence diagrams.

5.3.5 Writing the annotations

The annotations are put manually. We take the symmetric point of view of the above mapping. We look for elements in the source code corresponding to

- *InComponent* the class belongs to a component
- *InitClass* the class is a "main" part of the component
- *InitMethod* the method belongs to the main operations of the component
- *Provided* the field links to a provided interface
- *ProvidedIf* the Java interface refers to a provided interface
- *ProvidedMethod* a method implemnts a provided operation
- *Required* the field links to a required interface
- *BusinessType* the java type implements a component basic type
- *BusinessField* the field refers to a component basic type
- *BusinessParameter* the parameter refers to a component basic type

We illustrate the process on the `CashBoxController` component of the `CashDesk` composition.

a) Exploring the Java interfaces

Intuitively each Java interface should map to a component provided interface but actually Java interfaces are used twofold

- As a provided interface, it is then implemented by some class.
- As a required interface, it is then referenced in "provided" fields.

Moreover the Java interface gathers incoming and outgoing events (push/pull modes) so that it is not clear what is provided or required inside. Nevertheless we will follow the Java structure where there are no distinction between required and provided interface.

There no annotations envisaged for Java interface *e.g* to indicate which is the owner component, whether it is provided or required. Indeed, a required element (only interface are envisaged here) is attached to a class field and a provided element is attached to a class via the *ProvidedIf*. The missing link should be deduce later when exploring all required fields to get implementors.

Reverse Rule 5.3.1 (Java Interface) *Java interfaces are not annoted.*

Reverse Rule 5.3.2 (Provided Java Interface) *When a Java interface is implemented by a class which is InComponent a component `C1` then it is a provided interface of `C1`. There are no special annotation for that because it can be deduced in the class declarations via the ProvidedIf annotation.*

Reverse Rule 5.3.3 (Required Java Interface) *A Java interface is a required interface of a component* `C1` *if it is referenced in a Required field of a class which is InComponent* `C1`*. There are no special annotation for that because there can be many classes (and components) requiring this interface.*

Reverse Rule 5.3.4 (Java Interface Qualification) *Java interfaces can be qualified as both provided or required.*

```
package org.cocome.tradingsystem.cashdeskline.cashdesk.cashboxcontroller;

import org.cocome.tradingsystem.cashdeskline.events.CashAmountEnteredEvent;
import org.cocome.tradingsystem.cashdeskline.events.CashBoxClosedEvent;
import org.cocome.tradingsystem.cashdeskline.events.ChangeAmountCalculatedEvent;
import org.cocome.tradingsystem.cashdeskline.events.ExpressModeDisabledEvent;
import org.cocome.tradingsystem.cashdeskline.events.PaymentModeEvent;
import org.cocome.tradingsystem.cashdeskline.events.SaleFinishedEvent;
import org.cocome.tradingsystem.cashdeskline.events.SaleStartedEvent;

public interface CashBoxControllerEventHandlerIf {
  void onEvent(ChangeAmountCalculatedEvent changeAmountCalculatedEvent);
  void sendSaleStartedEvent(SaleStartedEvent saleStartedEvent);
  void sendSaleFinishedEvent(SaleFinishedEvent saleFinishedEvent);
  void sendPaymentModeEvent(PaymentModeEvent paymentModeEvent);
  void sendCashAmountEnteredEvent(
     CashAmountEnteredEvent cashAmountEnteredEvent);
  void sendCashBoxClosedEvent(CashBoxClosedEvent cashBoxClosedEvent);
  void sendExpressModeDisabledEvent(
     ExpressModeDisabledEvent expressModeDisabledEvent);
}
```

b) Exploring the Java classes declarations

For each class that have a correspondence to the component model we apply the following rules.

Reverse Rule 5.3.5 (Business Class) *In the class declaration we add the annotation (@InComponent) that link the class to the component.*

Reverse Rule 5.3.6 (Business Class Interface) *If the class implements a "business" interface we add the annotation (@ProvideIf) that link the class to the component interface and the Java interface.*

Reverse Rule 5.3.7 (Business Main Class) *If the class is the main entry point a component we add the annotation (@InitClass).*

Reverse Rule 5.3.8 (Anonymous model interface) *We assumed that every component should have only named interfaces. By default a component unnamed interface will be named as* `<ComponentName>If`.

```
package org.cocome.tradingsystem.cashdeskline.cashdesk.cashboxcontroller.impl;

import info.clearthought.layout.TableLayout;

import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JLabel;
```

```java
import javax.swing.JPanel;

import org.cocome.tradingsystem.cashdeskline.cashdesk.cashboxcontroller.
  CashBoxControllerEventHandlerIf;
import org.cocome.tradingsystem.cashdeskline.datatypes.KeyStroke;
import org.cocome.tradingsystem.cashdeskline.datatypes.PaymentMode;
import org.cocome.tradingsystem.cashdeskline.events.CashAmountEnteredEvent;
import org.cocome.tradingsystem.cashdeskline.events.CashBoxClosedEvent;
import org.cocome.tradingsystem.cashdeskline.events.ExpressModeDisabledEvent;
import org.cocome.tradingsystem.cashdeskline.events.PaymentModeEvent;
import org.cocome.tradingsystem.cashdeskline.events.SaleFinishedEvent;
import org.cocome.tradingsystem.cashdeskline.events.SaleStartedEvent;

import econet.annotations.*;

/**
 * GUI for the CashBox component
 * @author Yannick Welsch
 */
@SuppressWarnings("serial")
@InComponent(annotationSrc = {"Pascal"}, componentName = {"CashBoxController"})
@InitClass(annotationSrc = {"Pascal"}, componentName = {"CashBoxController"})
public class CashBox extends JPanel {...
}

package org.cocome.tradingsystem.cashdeskline.cashdesk.cashboxcontroller.impl;

import java.io.Serializable;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.ObjectMessage;
import javax.jms.Session;
import javax.jms.TopicConnection;
import javax.jms.TopicPublisher;
import javax.jms.TopicSession;
import javax.jms.TopicSubscriber;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import org.apache.log4j.Logger;
import org.cocome.tradingsystem.cashdeskline.cashdesk.cashboxcontroller.
  CashBoxControllerEventHandlerIf;
import org.cocome.tradingsystem.cashdeskline.events.CashAmountEnteredEvent;
import org.cocome.tradingsystem.cashdeskline.events.CashBoxClosedEvent;
import org.cocome.tradingsystem.cashdeskline.events.ChangeAmountCalculatedEvent;
import org.cocome.tradingsystem.cashdeskline.events.ExpressModeDisabledEvent;
import org.cocome.tradingsystem.cashdeskline.events.PaymentModeEvent;
import org.cocome.tradingsystem.cashdeskline.events.SaleFinishedEvent;
import org.cocome.tradingsystem.cashdeskline.events.SaleStartedEvent;

import econet.annotations.*;

/**
 * Implementation for the CashBox component
 *
 * @author Yannick Welsch
 */
@InComponent(annotationSrc = {"Pascal"}, componentName = {"CashBoxController"})
@ProvidedIf(annotationSrc={"Pascal"}, modelIfaceName={"CashBoxControllerIf"},
  javaIfaceName={"CashBoxControllerEventHandlerIf"})
public class CashBoxControllerEventHandlerImpl implements MessageListener,
  CashBoxControllerEventHandlerIf {...
}
```

Reverse Rule 5.3.9 (Business Type) *Some classes correspond to business datatypes and not to components. We found them in the data description.*

Examples In cashbox example, all events and datatypes are defined by classes related to business datatypes.

```
package org.cocome.tradingsystem.cashdeskline.events;

import java.io.Serializable;
import org.cocome.tradingsystem.cashdeskline.datatypes.KeyStroke;
import econet.annotations.BusinessType;

/**
 * This event signals the entering of a cash amount at the cash box keyboard
 * after taking cash from the customer. It is raised by the cash box controller
 * component after EVERY key stroke, <code>isFinalInput()
 * </code> is true if
 * the final input is entered.
 */

@BusinessType(annotationSrc={"Pascal"})
public class CashAmountEnteredEvent implements Serializable {
 private static final long serialVersionUID = -5441935251526952790L;
 private KeyStroke keystroke;
 public CashAmountEnteredEvent(KeyStroke keystroke) {
  this.keystroke = keystroke;
 }
 public KeyStroke getKeyStroke() {
  return keystroke;
 }
}
```

c) Exploring the Java classes structure

This section applys for the class that are *InComponent* classes only.

The instance variables (fields) can implement a link to a required interface, a business field, or an internal coupling (for example the `CashBoxController` component is implemented by the (main) class `CashBox` and the `CashBoxController` where each class declares a field to the other class).

Reverse Rule 5.3.10 (Required field) *Based on the types, one can find the field that correspond to a required interface (even if it was not declared as so in the UML component model).*

Reverse Rule 5.3.11 (Business field) *Based on business types, one can find the business field an annotate them.*

Reverse Rule 5.3.12 (Internal coupling) *Internal coupling is represented as a special* `@Required` *annotation with an interface name set by a reserved keyword* `internal`*. Its information would be useful for further investigations. This is not a business field.*

Reverse Rule 5.3.13 (Implementation Required field) *Sometimes the requirements refer to some implementation rather than the component concepts. To keep that information we propose to define a special keyword* `implementation` *to denote that the required interface is not present at the implementation level but obtained from various sources. If possible we also provide another a source and interface entry to the component model.*

Examples Nothing in the `CashBox` class, there's only an internal coupling toward `CashBoxController EventHandlerImpl`.

In the `CashBoxControllerEventHandlerImpl` class, there's an internal coupling toward `CashBox` and also implementation required fields related to the implementation of the `:EventBus`. The difficulty here is that component concepts disappear at the implementation level we note it using the special keyword `implementation`.

```
public class CashBoxControllerEventHandlerImpl implements MessageListener,
  CashBoxControllerEventHandlerIf {
 final String CHANNEL_CONNECTION_FACTORY = "ChannelConnectionFactory";
 private String topicName;
 //implementation references
```

```
@Required(annotationSrc = {"Pascal","Model"}, modelIfaceName = { "implementation" , "EventBusIf"} )
private Context jndiContext;
@Required(annotationSrc = {"Pascal","Model"}, modelIfaceName = { "implementation" , "EventBusIf"} )
private TopicPublisher cashBoxPublisher;
@Required(annotationSrc = {"Pascal","Model"}, modelIfaceName = { "implementation" , "EventBusIf"} )
private TopicSession topicSession;
@Required(annotationSrc = {"Pascal","Model"}, modelIfaceName = { "implementation" , "EventBusIf"} )
private Logger log = Logger
  .getLogger(CashBoxControllerEventHandlerImpl.class);
//internal references
@Required(annotationSrc = {"Pascal"}, modelIfaceName = { "internal" } )
private CashBox cashbox;
```

d) Exploring the Java class behaviour

This section applys for the class that are *in component* classes only. The main goal is to find so-called business methods and main (init) methods.

The methods refer to service (component operation, business method) specification. We manually decide whether a method is a service (component operation, business method) or not.

Reverse Rule 5.3.14 (InitMethod) *The InitMethod is chosen among the constructor or initialization methods.*

Reverse Rule 5.3.15 (ProvidedMethod) *The "business" methods signature refer to service or operations. The business qualification is decided manually.*

Examples The component service description is quite absent of the UML model. Only the sequence charts provide some valuable but information.

The *InitMethod* were not present in the component model.

```
@InitMethod(annotationSrc = {"Pascal"}, componentName = {"CashBoxController"})
 public CashBox(String eventchannel) {
  super();
 ...}
```

The `Service` were partially present in the component model. Their shape changed during the implementation pattern of JMS.

```
//should be a required method according to the UML model
@ProvidedMethod(annotationSrc = {"Manual"}, modelIfaceName = {"CashBoxControllerIf"})
public void onEvent(ChangeAmountCalculatedEvent changeAmountCalculatedEvent) {
 log.info("ChangeAmountCalculatedEvent received");
 cashbox.openCashBox();
}

@ProvidedMethod(annotationSrc = {"Manual"}, modelIfaceName = {"CashBoxControllerIf"})
public void sendSaleStartedEvent(SaleStartedEvent saleStartedEvent) {
 try {
  cashBoxPublisher.publish(topicSession
    .createObjectMessage(saleStartedEvent));
 } catch (JMSException e) {
  log.error(e);
  e.printStackTrace();
 }
}
```

e) Exploring the Java methods

Reverse Rule 5.3.16 (BusinessParameter) *The business parameters are found in the "business" methods signature. Among the method signature some refer to service (operations) parameters others are implementation ones. The business qualification is decided manually.*

```
//should be e required method according to the UML model
@ProvidedMethod(annotationSrc = {"Manual"}, modelIfaceName = {"CashBoxControllerIf"})
public void onEvent(
  @BusinessParameter(annotationSrc = {"Pascal"}) ChangeAmountCalculatedEvent changeAmountCalculatedEvent) {
 log.info("ChangeAmountCalculatedEvent received");
 cashbox.openCashBox();
}

@ProvidedMethod(annotationSrc = {"Manual"}, modelIfaceName = {"CashBoxControllerIf"})
public void sendSaleStartedEvent(
  @BusinessParameter(annotationSrc = {"Pascal"}) SaleStartedEvent saleStartedEvent) {
 try {
  cashBoxPublisher.publish(topicSession
    .createObjectMessage(saleStartedEvent));
 } catch (JMSException e) {
  log.error(e);
  e.printStackTrace();
 }
}
```

e) Full Example

The (main) class `CashBox` of the `CashBoxController` component.

```
package org.cocome.tradingsystem.cashdeskline.cashdesk.cashboxcontroller.impl;

import info.clearthought.layout.TableLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import org.cocome.tradingsystem.cashdeskline.cashdesk.
  cashboxcontroller.CashBoxControllerEventHandlerIf;
import org.cocome.tradingsystem.cashdeskline.datatypes.KeyStroke;
import org.cocome.tradingsystem.cashdeskline.datatypes.PaymentMode;
import org.cocome.tradingsystem.cashdeskline.events.CashAmountEnteredEvent;
import org.cocome.tradingsystem.cashdeskline.events.CashBoxClosedEvent;
import org.cocome.tradingsystem.cashdeskline.events.ExpressModeDisabledEvent;
import org.cocome.tradingsystem.cashdeskline.events.PaymentModeEvent;
import org.cocome.tradingsystem.cashdeskline.events.SaleFinishedEvent;
import org.cocome.tradingsystem.cashdeskline.events.SaleStartedEvent;

import econet.annotations.*;

/**
 * GUI for the CashBox component
 * @author Yannick Welsch
 */
@SuppressWarnings("serial")
@InComponent(annotationSrc = {"Pascal"}, componentName = {"CashBoxController"})
@InitClass(annotationSrc = {"Pascal"}, componentName = {"CashBoxController"})
public class CashBox extends JPanel {

 //error : @BusinessField(annotationSrc = "Manual")
 @Required(annotationSrc = {"Pascal"}, modelIfaceName = { "internal" } )
 private CashBoxControllerEventHandlerIf cashBoxControllerEventHandler;
 private JLabel cashbox;

 @InitMethod(annotationSrc = {"Pascal"}, componentName = {"CashBoxController"})
 public CashBox(String eventchannel) {
 ...
 }

 private void addListenerToButton(final JButton b) {
 ...
 }
```

```
  //found in sequence chart diagrams of the UML model — default interface
  @ProvidedMethod(annotationSrc = {"Pascal"}, modelIfaceName = {"CashBoxControllerIf"})
  public void openCashBox() {
   cashbox.setForeground(Color.PINK);
   cashbox.setText("cashbox open");
  }

  //found in sequence chart diagrams of the UML model — default interface
  @ProvidedMethod(annotationSrc = {"Pascal"}, modelIfaceName = {"CashBoxControllerIf"})
  public void closeCashBox() {
   cashbox.setForeground(Color.BLACK);
   cashbox.setText("cashbox closed");
  }
}
```

5.3.6 Annoted Components

Here is the list of annoted primitive components of the `Test1` benchmark. Java interfaces are not annotated but appear in the *ProvidedIf* annotation.

- Component `CashBoxController` with its implicit interface `CashBoxControllerIf`.
  Classes `CashBox` and `CashBoxControllerEventHandlerImpl`.
- Component `ScannerController` with its implicit interface `ScannerControllerIf`.
  Classes `ScannerController` and `ScannerControllerEventHandlerImpl`.
  The `ScannerController` creates a `ScannerControllerEventHandlerImpl` but there is a unidirectional internal link from the scanner to the controller.
- Component `PrinterController` with its implicit interface `PrinterControllerIf`.
  Classes `PrinterController` and `PrinterControllerEventHandlerImpl`.
  The printer state is internal to the component here and we assume it could be considered as a business type.
  The `PrinterController` creates a `PrinterControllerEventHandlerImpl` but there is a unidirectional internal link from controller to printer.
- Component `LightDisplayController` with its implicit interface `LightDisplayControllerIf`.
  Classes `LightDisplayController` and `ControllerEventHandlerImpl`.
  The `LightDisplayController` creates a `LightDisplayControllerEventHandlerImpl` but there is a unidirectional internal link from controller to display.
- Component `CardReaderController` with its implicit interface `CardReaderControllerIf`.
  Classes `CardReader` and `CardReaderControllerEventHandlerImpl`.
  The `CardReader` creates a `CardReaderControllerEventHandlerImpl` but there is a unidirectional internal link from reader to the controller.
- Component `CashDeskGUI` with its implicit interface `CashDeskGUIIf`.
  Classes `CashDeskGUI` and `GUIEventHandlerImpl`.
  The `CashDeskGUI` creates a `GUIEventHandlerImpl` but there is a unidirectional internal link from controller to gui.
- Component `CashDeskApplication` with its implicit interface `CashDeskApplicationIf` is not really implemented as usual.
  Classe `ApplicationEventHandlerImpl` represent the controller but part of the applicat belong to the composite `CashDesk` or `CashDeskLine`.
  The cash desk state `CashDeskStates` is internal to the component here and we assume it could be considered as a business type.
  The composite `CashDesk` (or `CashDeskLine`) creates a `ApplicationEventHandlerImpl` but there is no link from controller to application, they may communicate via the buses.

5.3.7 Composition

No annotations are defined for the composition. Moreover, encapsulation and promotion is not preserved in Java except on the package naming.

The distinction between `CashDesk` and `CashDeskLine` is not clear in the Java code.

We only annotated the `CashDesk`. There are two *InitMethod*s: a constructor and a main.

This is not clear what should be all the interfaces because there are no encapsulation, it is directly handled by (sub) components.

5.3.8 Conclusion

Finding business elements in the Java code is mainly an intellectual process in the case of CoCoME. Some guidelines or templates can apply but there are many exceptions.

- Mapping models = trace the concepts and decisions
- Reverse engineering should work on patterns
- Manual implementation lead to exceptions
- Incomplete models prevent nouns comparison
- Syntactic is not sufficient
- Problem of inheritance

Chapter 6

UML/Java

Draft version resulting from Cluj's workshop

6.1 Introduction

In this chapter we investigate the field of reverse engineering Java code against UML. In this context, there are several approaches

1. compose two transformations Java to UML $\circ$ UML to components
   + get a more abstract object oriented representation
   + reuse existing attemps
   + useful for data types modelling
   - loose pertinent information ? behaviour
   - similar heuristic problems on the "business" part
   - still a problem to get a component model
2. compare an existing UML component model with Java to set annotations (e.g. the Test1 experimentation)
   + components are known
   + identify similarities is easier and more sure than finding from scratch
   + useful for data types modelling
   - partial component model informations
   - defining what is a UML component model from UML diagrams
   - noun comparisons is still difficult
   - instanciate an XML ou XMI model (API)
3. Find a mapping Component UML - Java
   + simple, applicable to plain Java and annotations
   + quite close to the CCMM
   + define reverse patterns
   - code information is still needed for behaviour
   - strict code arrangement
   - no tools (?)

Solution 1 is not yet feasible without powerfull UML RE tools including statecharts.
Solution 2 was experienced manually, implementation requires 3 tool (UML models, Java code, a bridge)
Solution 3 may run on a limited set of programs. It implies a set of recognition patterns.

6.2 Mapping UML Components to Java Classes

We consider a UML Component Model made of a set of *consistent* UML diagrams.

Some assumptions

- Components are distingued from classes (even if the metamodel sets that it it a class)
- UML interface are restricted to Java interface
- Ports and port connections are ignored but not binding of interfaces.
- Connectors are simply bindings.
- Protocol state machines are associated to components, ports and interfaces.
- Lightened UML model (events, actions...)
- Properties and constraints

Unser these assumptions we try to identify some translation patterns.

6.2.1 Basic Component Pattern

- component $Co \rightarrow$ a class Cl of a package Co
  - provided interface $pi \rightarrow$ inherited interface pi of a package Co
  - requided interface $ri \rightarrow$ field of type an interface ri of a package Co (may vary here)
  - ports are omitted $\rightarrow$ traceable comments to the interfaces
- Features $\rightarrow$ methods
  - Attributes $\rightarrow$ fields to Datatype classes
  - Operations $\rightarrow$ methods
- Dynamic Behaviour (protocol) $\rightarrow$ implementation pattern
  - communications $\rightarrow$ message send or some communication support
  - state/transitions $\rightarrow$ some automaton pattern

6.2.2 Composition Component Pattern

- architecture $A \rightarrow$ a class A of a package A
  - components and interfaces (as above) component packages can be included in package A (but it can also be classify in some "reusable" library of types.
  - terface $rconnectorsrightarrow$ fieither an exact matching or inheritance of interface (somewhat disturbing to imply the same name)
  - ports are omitted $\rightarrow$ traceable comments to the interfaces
- Composite (UML composite structure, UML composition relation)
  - Logicalfifield in the composite class of type the component class (according to multiplicity) + some marking or annotation
  - Name: package inclusion (disturbing for reusing the types)
- Connections
  - type level = interface link
  - instance level = object value with a consistent type

6.3 Mapping Java Classes UML Components

The idea is to reverse the mapping patterns.
to continue

Chapter 7

Conclusion

Since the mapping from the UML component model to the Java code is not a model transformation (defined as a set of rules) there are no true correspondence between the concrete (java) model and the abstract (UML component) model. Moreover the component is specification includes many holes that are fullfilled by the code "interpretation" Therefore reversing the mapping is not easy to draw: the user must identified elements that may correspond to the abstract model, find an abstract element if it exists, find a rule if it can be ruled, or create a component model element.

We also met the problem of inheriting annotations *e.g.* interface/class.

Last, we surely had better to change the reference model instead of UML to get a "more component" description. On one hand we could compare the implementation with a FRACTAL or SOFA component specification (or later create a CCMM one). On the other hand we will abstract to CCMM models. This can be led in an ongoing experimentation.

All the content informations of this experimentation is on the SVN repository in the `casestudy` directory.

Ongoing Work include

- Define a CCMM-UML mapping (or transformation)
- Explore further the UML-Java (engineering, reverse-engineering)
- Classify patterns
- Rule based-system investigation
- ... open issue

Bibliography

[ACPR07] Pascal André, Dan Chiorean, Frantisek Plasil, and Jean-Claude Royer. ECONET Project - Prague Workshop Report, September 2007.

[ACPR08] Pascal André, Dan Chiorean, Frantisek Plasil, and Jean-Claude Royer. ECONET Project - Nantes Workshop Report, June 2008.

[Reu06] Ralf et al. Reussner. CoCoME - The Common Component Modeling Example, 2006. GI-Dagstuhl Research Seminar.

[RRMP08] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *LNCS*. Springer, Heidelberg, 2008.

Appendix A

Annotations

In this appendix we provide the Java definition of the annotations.

Component - Class Relation

Listing A.1: InComponent.java

```
package econet.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

/**
 * One or more Java classes can be assigned to a single component.
 * Such an assignment is specified by this annotation.
 */
@Target(ElementType.TYPE)
// Should be just a class
public @interface InComponent {
 /**
  * @return the array of sources for this annotation
  */
 String[] annotationSrc();

 /**
  * @return the array (one entry per annotation source) containing component
  *  Names which the annotated class is assigned to. If a single
  *  source declares the class to participate in several components,
  *  its entry should be a comma-separated list of component name
  */
 String[] componentName();
}
```

Entry points

Listing A.2: InitClass.java

```
package econet.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

/**
 * This class is the first instantiated and is responsible
 * (its constructor) for the instantiation and initialization
 * of the component's content.
 */
@Target(ElementType.TYPE)
```

```
// Should be just a class
public @interface InitClass {
 /**
  * @return the array of sources for this annotation
  */
 String[] annotationSrc();

 /**
  * @return the array (one entry per annotation source) containing component
  *  Names for which the annotated class provides the initialization.
  *  If a single source declares the class to participate in several
  *  components, its entry should be a comma-separated list of
  *  component name
  */

 String[] componentName();
}
```

Listing A.3: InitMethod.java

```
package econet.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

/**
 * The component content is instantiated and initialized by a method
 * ( it can be a constructor, a static method or an initialization method
 * to be called after the default constructor).
 */
@Target( { ElementType.CONSTRUCTOR, ElementType.METHOD })
public @interface InitMethod {
 /**
  * @return the array of sources for this annotation
  */
 String[] annotationSrc();

 /**
  * @return the array (one entry per annotation source) containing component
  *  Names for which the annotated method provides the initialization.
  */
 String[] componentName();
}
```

Interfaces

Provided

Listing A.4: Provided.java

```
package econet.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

/**
 * In Java sources, a provided interface might be in a form of a class
 * attribute. The attribute stores a reference to a class implementing
 * the provided interface.
 *
 * This type is missing in the ECONET proposal
 */
@Target(ElementType.FIELD)
public @interface Provided {
 /**
  * @return the array of sources for this annotation
```

```
 */
String[] annotationSrc();

/**
 * @return the array (one entry per annotation source) containing
 *  the name of the interface represented by this field
 */
String[] modelIfaceName();
}
```

Listing A.5: ProvidedIf.java

```
package econet.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

/**
 * All methods of the specified Java interface (which the annotated
 * class has to implement) are marked as a part of the provided
 * interface of the component
 */
@Target(ElementType.TYPE)
public @interface ProvidedIf {
 /**
  * @return the array of sources for this annotation
  */
 String[] annotationSrc();

 /**
  * @return the array (one entry per annotation source) containing the
  *  name of the component interface represented by this type
  */

 String[] modelIfaceName();

 /**
  * @return the array (one entry per annotation source) containing the name
  *  of the java interface which is defining one component Interface
  *  If a single source declares to participate in several components,
  *  its entry should be a comma-separated list of java interface
  *  names (for instance {"ActionListener,WindowListener"}
  */

 String[] javaIfaceName() default { "" };
}
```

Listing A.6: ProvidedMethod.java

```
package econet.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

/**
 * The method is a part of the provided interface of the component
 */
@Target(ElementType.METHOD)
public @interface ProvidedMethod {
 /**
  * @return the array of sources for this annotation
  */
 String[] annotationSrc();

 /**
  * @return the array (one entry per annotation source) containing the
  *  name of the component interface which the annotated method is
  *  part of. If a single source declares to the method participate
  *  in several interfaces, its entry should be a comma-separated
```

```
 *  list of interface names
 */
String[] modelIfaceName();
}
```

Required

Listing A.7: Required.java

```
package econet.annotations;
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

/**
 * In Java sources, a required interface is present in a form of a
 * class attribute. The attribute stores a reference to another
 * component, whose provided interface is bound to this required
 * interfaces.  Therefore, the target of the annotation for required
 * interface is an attribute of a Java class.
 */
@Target(ElementType.FIELD)
public @interface Required {
 /**
  * @return the array of sources for this annotation
  */
 String[] annotationSrc();

 /**
  * @return the array (one entry per annotation source) containing
  *  the name of the interface represented by this field
  */
 String[] modelIfaceName();
}
```

Business elements

Listing A.8: BusinessType.java

```
package econet.annotations;
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

/**
 * all their instances of such type are important for a component
 * behaviour.
 */
@Target(ElementType.TYPE)
public @interface BusinessType {
 /**
  * @return the array of sources for this annotation
  */
 String[] annotationSrc();
}
```

Listing A.9: BusinessField.java

```
package econet.annotations;
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

/**
 * Marks particular Java class attributes as important for business logic.
 */

@Target(ElementType.FIELD)
```

```java
public @interface BusinessField {

  /**
   * @return the array of sources for this annotation
   */
  String[] annotationSrc();
}
```

Listing A.10: BusinessParameter.java

```java
package econet.annotations;
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

/**
 * Marks particular method parameter as important for business
 * logic.
 */
@Target(ElementType.PARAMETER)
public @interface BusinessParameter {
  /**
   * @return the array of sources for this annotation
   */
  String[] annotationSrc();
}
```

Appendix B

Component Model Examples

These examples are outside the scope of Test1 but they are used to explain some mapping rules of chapter 5.

B.1 Structure: the `TradingSystem::Inventory::Application` component

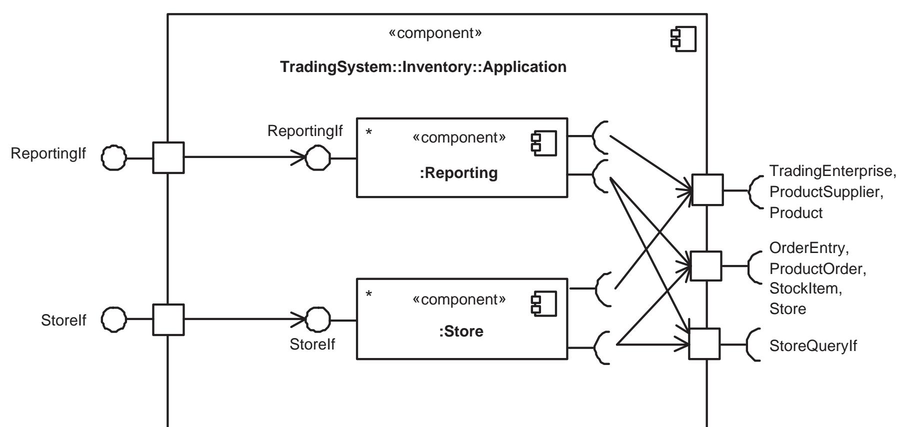

Figure B.1: CoCoME component: the `TradingSystem::Inventory::Application` component