# Test1 Case Study Analysis

## UML Components vs Java code

P. André

COLOSS-LINA

september, 21-24 2008

## Introduction

The Test1 example is a subset of the CoCoME case study.

`CoCoME` case study

- pertinent application (45 packages, 121 classes or interfaces)
- model + implementation
- component model and UML diagrams
- same example support for the whole project
- allow comparison with other works

`Test1` benchmark

- one of the two selected subset of CoCoME (Nantes workshop)
- small but representative (same example support for the whole project experimentations)
- vertical slice (model, code)

`Test1` ⊂ `Test2` ⊂ `CoCoME`

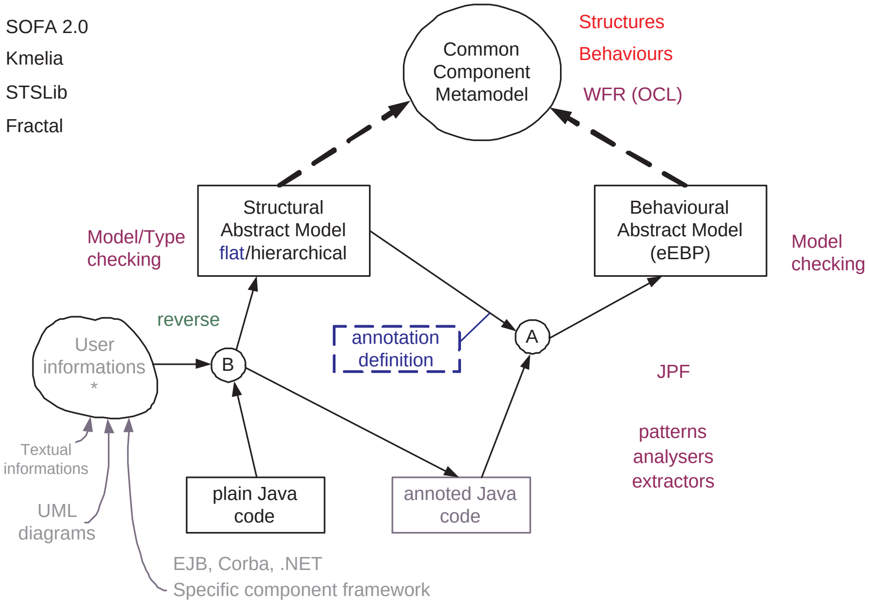
## Where in the ECONET Project

SOFA 2.0

Kmelia

STSLib

Fractal



## Where in the process B

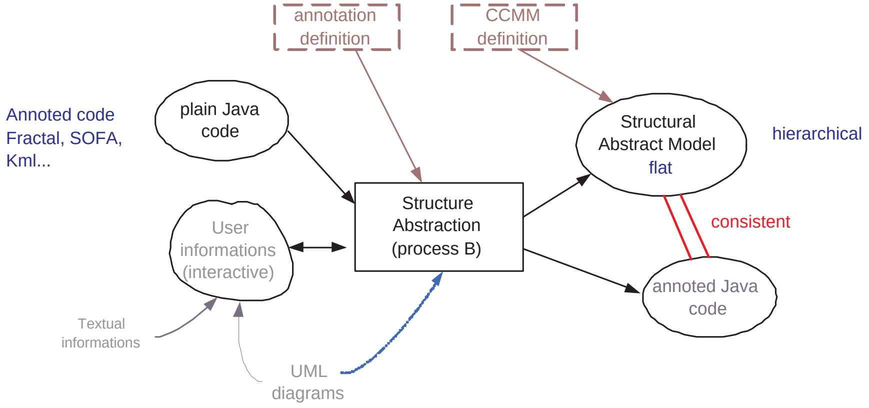


Figure: A general view of the process B

p. 40 of the report of Nantes' workshop

## This Talk

Pretext for various discussions and investigations.

- select model and code subset
- explain it for those who did not have a look to it
- (naively) investigate the relation between model and code (in practice)
  - how did the implementors proceed ?
  - can we trace teh decisions ?
- investigate the reverse relation between model and code (with UML)
  - what to look for ?
  - how to abstract ?
- put the annotations
  - where ?
  - how ?
- propose a benchmark for CCMM API testing

## Outline of the Talk

## Outline

## Previous Experimentation:MP

Process B (details in chapter 3 of Nantes' workshop report)

- Model management (CMM 1.0)
  - ATL / EMF
  - Instanciation
- Annotations (v1.0, strings instead of arrays)
  - reading (APT)
  - writing from Model
- Eclipse Plugin
- CoCoME experimentation

## Previous Experimentation:Model

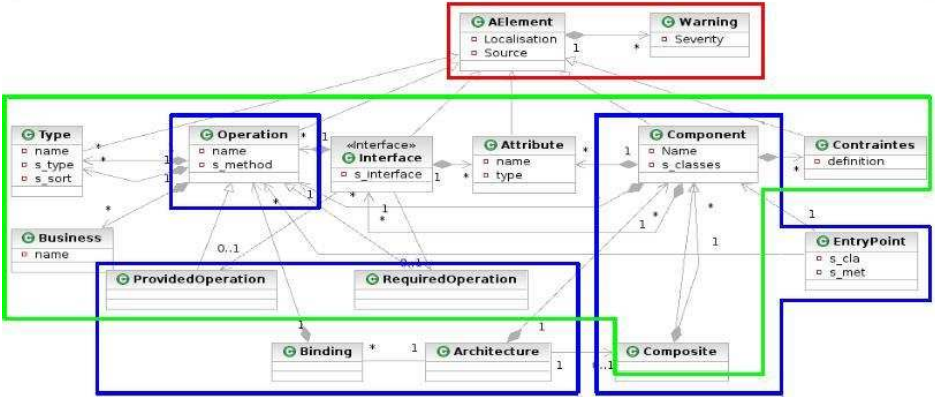


Figure: Master Project: CCM

## Previous Experimentation:CoCoME

CoCoME subset

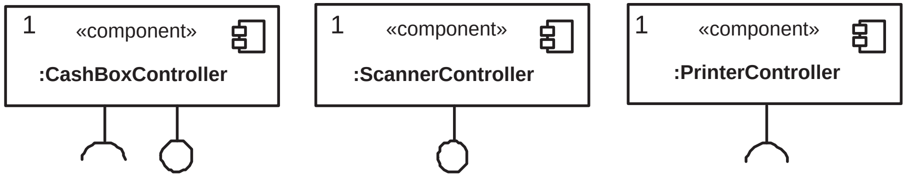
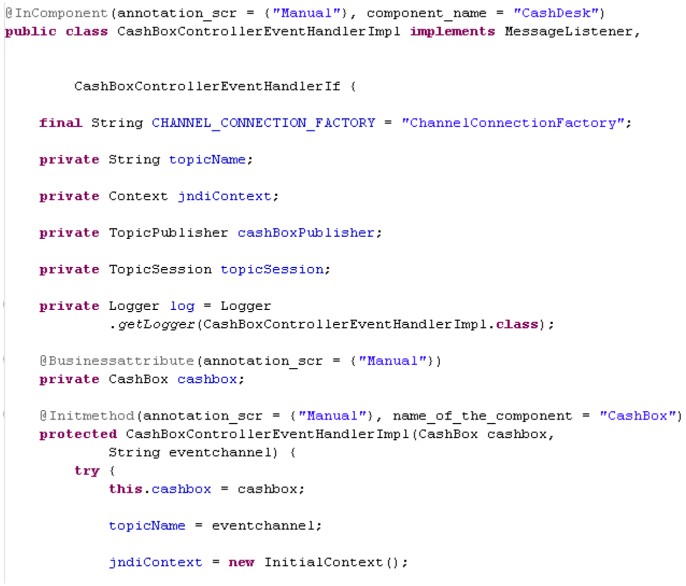


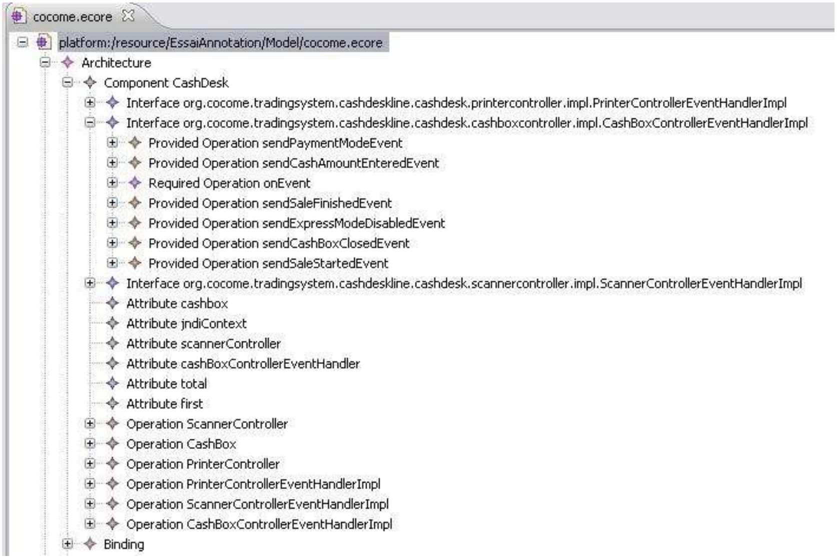
Figure: Master Project: CoCoME subset

## Previous Experimentation:annotation



```
@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();
```

## Previous Experimentation:Model instanciation


cocome.ecore

```
platform:/resource/EssaiAnnotation/Model/cocome.ecore
  Architecture
    Component CashDesk
      Interface org.cocome.tradingsystem.cashdeskline.cashdesk.printercontroller.impl.PrinterControllerEventHandlerImpl
      Interface org.cocome.tradingsystem.cashdeskline.cashdesk.cashboxcontroller.impl.CashBoxControllerEventHandlerImpl
        Provided Operation sendPaymentModeEvent
        Provided Operation sendCashAmountEnteredEvent
        Required Operation onEvent
        Provided Operation sendSaleFinishedEvent
        Provided Operation sendExpressModeDisabledEvent
        Provided Operation sendCashBoxClosedEvent
        Provided Operation sendSaleStartedEvent
      Interface org.cocome.tradingsystem.cashdeskline.cashdesk.scannercontroller.impl.ScannerControllerEventHandlerImpl
      Attribute cashbox
      Attribute jndiContext
      Attribute scannerController
      Attribute cashBoxControllerEventHandler
      Attribute total
      Attribute first
      Operation ScannerController
      Operation CashBox
      Operation PrinterController
      Operation PrinterControllerEventHandlerImpl
      Operation ScannerControllerEventHandlerImpl
      Operation CashBoxControllerEventHandlerImpl
    Binding
```

Figure: Master Project: CoCoME subset

## Outline

## CoCoME (Component) Model

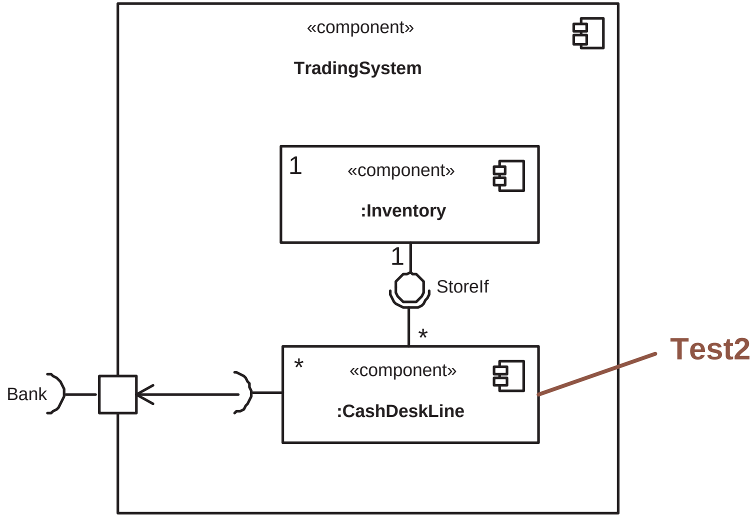
UML2 model

- structure
  - component diagram (few details)
    - component instances
    - interfaces, ports, connections
  - class diagram
    - class, operations, interfaces
    - relations
- behaviour
  - sequence diagrams (partial view, instance level)
  - no statecharts
  - writing from Model
- functional : USE cases (processes) + text
- non functional

Implementation frameworks (JMS, Hibernate...)

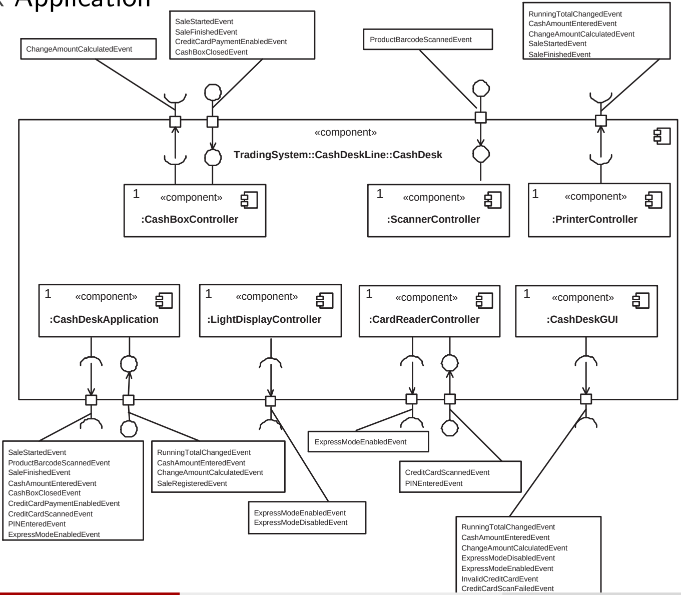
## CoCoME Component Model:structure

Data Management + Bus + Application



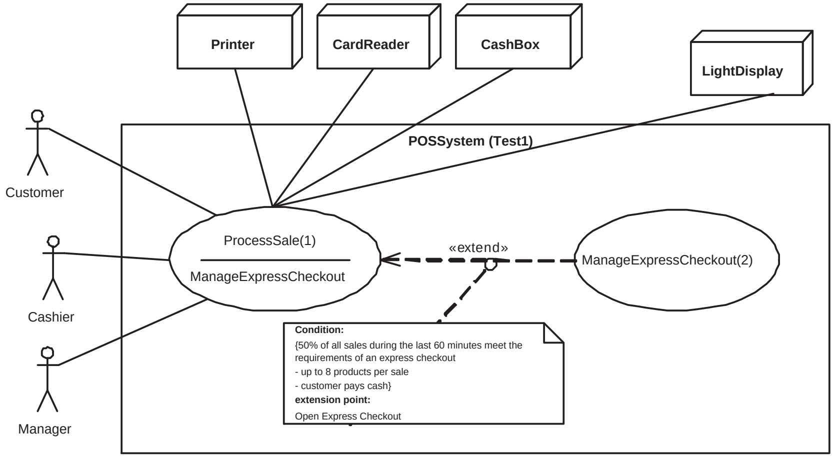
## CoCoME Component Model:structure:test1

### CashDesk Application



## CoCoME Component Model:behaviour:fct

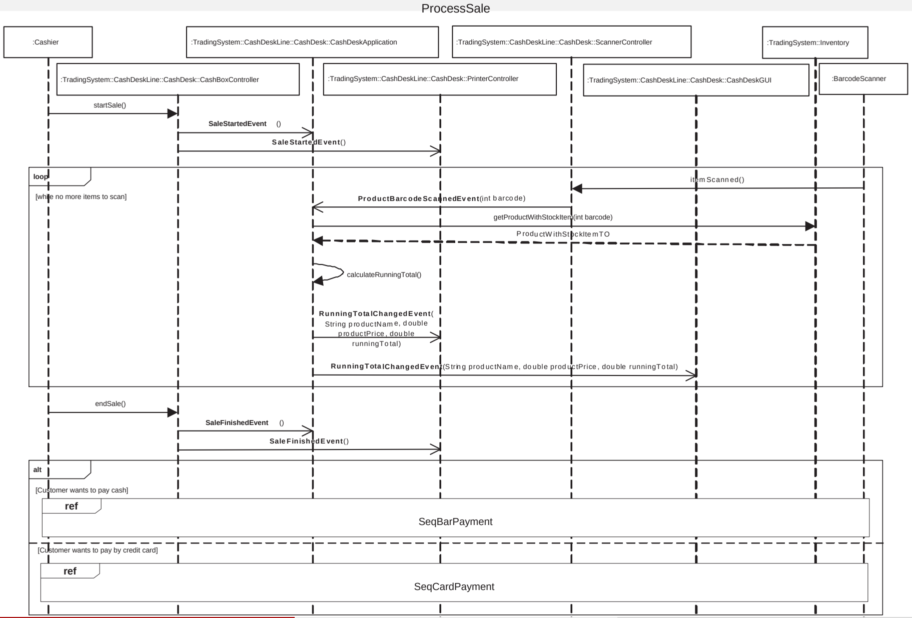
CashDesk Application



Condition:
{50% of all sales during the last 60 minutes meet the requirements of an express checkout
- up to 8 products per sale
- customer pays cash}
extension point:
Open Express Checkout

Figure: cocome:cashdeskuc

## CoCoME Component Model:behaviour:seq


ProcessSale
:Cashier
:TradingSystem::CashDeskLine::CashDesk::CashDeskApplication
:TradingSystem::CashDeskLine::CashDesk::ScannerController
:TradingSystem::Inventory
:TradingSystem::CashDeskLine::CashDesk::CashBoxController
:TradingSystem::CashDeskLine::CashDesk::PrinterController
:TradingSystem::CashDeskLine::CashDesk::CashDeskGUI
:BarcodeScanner
startSale()
SaleStartedEvent ()
SaleStartedEvent()
loop
[while no more items to scan]
itemScanned()
ProductBarcodeScannedEvent(int barcode)
getProductWithStockItem(int barcode)
ProductWithStockItemTO
calculateRunningTotal()
RunningTotalChangedEvent( String productName, double productPrice, double runningTotal)
RunningTotalChangedEvent(String productName, double productPrice, double runningTotal)
endSale()
SaleFinishedEvent ()
SaleFinishedEvent()
alt
[Customer wants to pay cash]
ref
SeqBarPayment
[Customer wants to pay by credit card]
ref
SeqCardPayment

## CoCoME Component Model:behaviour:seq

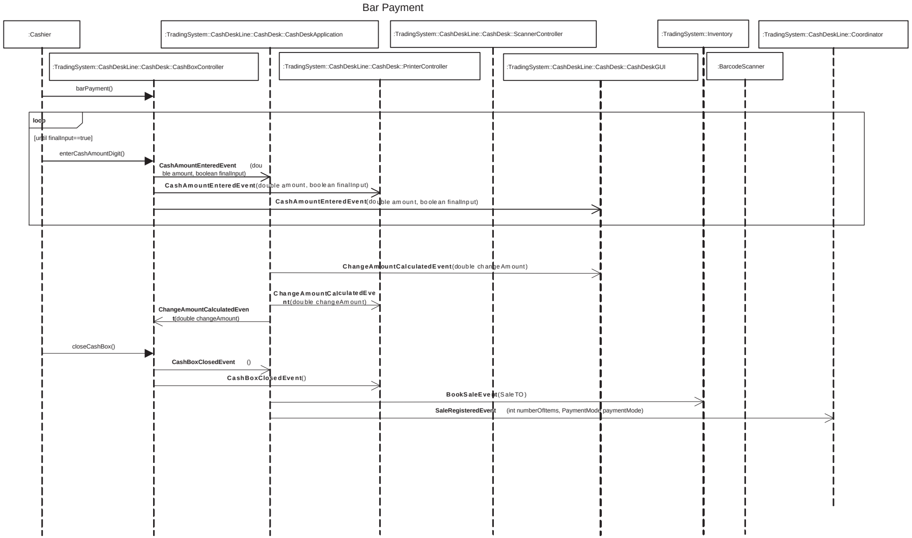

Bar Payment
:Cashier
:TradingSystem::CashDeskLine::CashDesk::CashDeskApplication
:TradingSystem::CashDeskLine::CashDesk::ScannerController
:TradingSystem::Inventory
:TradingSystem::CashDeskLine::Coordinator
:TradingSystem::CashDeskLine::CashDesk::CashBoxController
:TradingSystem::CashDeskLine::CashDesk::PrinterController
:TradingSystem::CashDeskLine::CashDesk::CashDeskGUI
:BarcodeScanner
barPayment()
loop
[until finalInput==true]
enterCashAmountDigit()
CashAmountEnteredEvent (double amount, boolean finalInput)
CashAmountEnteredEvent(double amount, boolean finalInput)
CashAmountEnteredEvent(double amount, boolean finalInput)
ChangeAmountCalculatedEvent(double changeAmount)
ChangeAmountCalculatedEvent(double changeAmount)
ChangeAmountCalculatedEvent(double changeAmount)
closeCashBox()
CashBoxClosedEvent ()
CashBoxClosedEvent()
BookSaleEvent(SaleTO)
SaleRegisteredEvent (int numberOfItems, PaymentMode paymentMode)

Figure: cocome:processSaleSCPDSp

## CoCoME Component Model:behaviour:seq

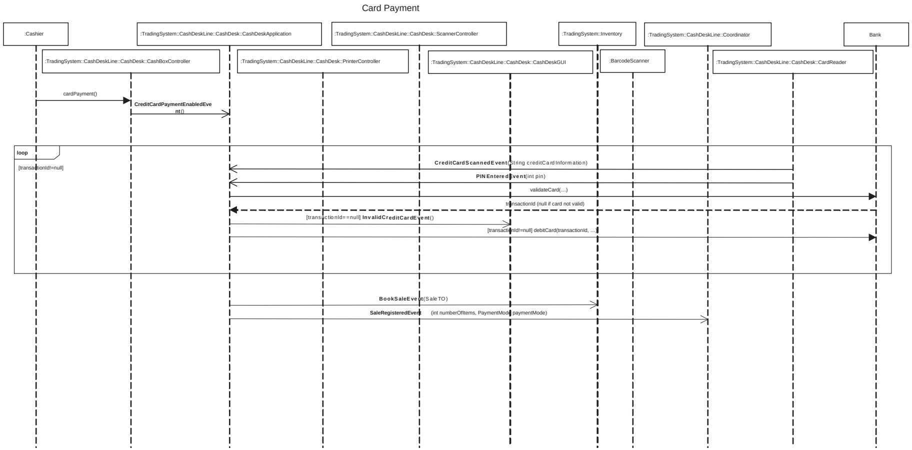

Card Payment
:Cashier
:TradingSystem::CashDeskLine::CashDesk::CashDeskApplication
:TradingSystem::CashDeskLine::CashDesk::ScannerController
:TradingSystem::Inventory
:TradingSystem::CashDeskLine::Coordinator
Bank
:TradingSystem::CashDeskLine::CashDesk::CashBoxController
:TradingSystem::CashDeskLine::CashDesk::PrinterController
:TradingSystem::CashDeskLine::CashDesk::CashDeskGUI
:BarcodeScanner
:TradingSystem::CashDeskLine::CashDesk::CardReader
cardPayment()
CreditCardPaymentEnabledEvent()
loop
[transactionId!=null]
CreditCardScannedEvent(String creditCardInformation)
PINEnteredEvent(int pin)
validateCard(...)
transactionId (null if card not valid)
[transactionId==null] InvalidCreditCardEvent()
[transactionId!=null] debitCard(transactionId, ...)
BookSaleEvent(SaleTO)
SaleRegisteredEvent (int numberOfItems, PaymentMode paymentMode)

Figure: cocome:processSaleSCPDSp

## Outline

## CoCoME implementation:java

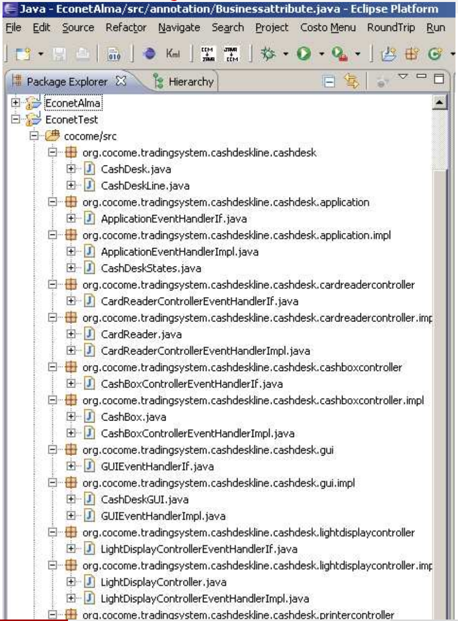
- whole application
  - 45 packages
  - 121 classes or interfaces

  organised similarly according to the model components
  - components
  - interfaces
  - data classes
- Test1 (application oriented)
  - 14 packages
  - 24 classes or interfaces

  organised around the Swing GUI and JMS middleware
  - user interaction events
  - software events (messages between components)

Implementation frameworks (JMS, Hibernate, JDBC...)

## CoCoME implementation:java


Java - EconetAlma/src/annotation/Businessattribute.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Costo Menu RoundTrip Run
Package Explorer
Hierarchy

```
EconetAlma
EconetTest
  cocome/src
    org.cocome.tradingsystem.cashdeskline.cashdesk
      CashDesk.java
      CashDeskLine.java
    org.cocome.tradingsystem.cashdeskline.cashdesk.application
      ApplicationEventHandlerIf.java
    org.cocome.tradingsystem.cashdeskline.cashdesk.application.impl
      ApplicationEventHandlerImpl.java
      CashDeskStates.java
    org.cocome.tradingsystem.cashdeskline.cashdesk.cardreadercontroller
      CardReaderControllerEventHandlerIf.java
    org.cocome.tradingsystem.cashdeskline.cashdesk.cardreadercontroller.imp
      CardReader.java
      CardReaderControllerEventHandlerImpl.java
    org.cocome.tradingsystem.cashdeskline.cashdesk.cashboxcontroller
      CashBoxControllerEventHandlerIf.java
    org.cocome.tradingsystem.cashdeskline.cashdesk.cashboxcontroller.impl
      CashBox.java
      CashBoxControllerEventHandlerImpl.java
    org.cocome.tradingsystem.cashdeskline.cashdesk.gui
      GUIEventHandlerIf.java
    org.cocome.tradingsystem.cashdeskline.cashdesk.gui.impl
      CashDeskGUI.java
      GUIEventHandlerImpl.java
    org.cocome.tradingsystem.cashdeskline.cashdesk.lightdisplaycontroller
      LightDisplayControllerEventHandlerIf.java
    org.cocome.tradingsystem.cashdeskline.cashdesk.lightdisplaycontroller.imp
      LightDisplayController.java
      LightDisplayControllerEventHandlerImpl.java
    org.cocome.tradingsystem.cashdeskline.cashdesk.printercontroller
```

## CoCoME implementation:decisions

Comparing design and implementation

- model elements disappear
  - The main design decisions focus on the implementation of the `:EventBus` composite which is based on the JMS API.
  - interfaces are merged, port do not exist
  - components
- restructurations
  - Cash desk application is merged with cash desk and cash deskline
  - delegation is flatten through imports
- new features
  - GUI framework classes
  - JMS framework classes
  - Hibernate/JDBC framework classes
- non component model (data classes, sequences...)

The question is how far is the code from the model

## CoCoME implementation:patterns

Tracing the decisions

- can we find patterns ?
  - manual or not
  - systematic or not
  - syntactic vs semantics
- restructurations
  - Cash desk application is merged with cash desk and cash deskline
  - delegation is flatten through imports
- new features
  - GUI framework classes
  - JMS framework classes
  - Hibernate/JDBC framework classes
- non component model (data classes, sequences...)

The question is how far is the code from the model

# CoCoME implementation:patterns

## An example

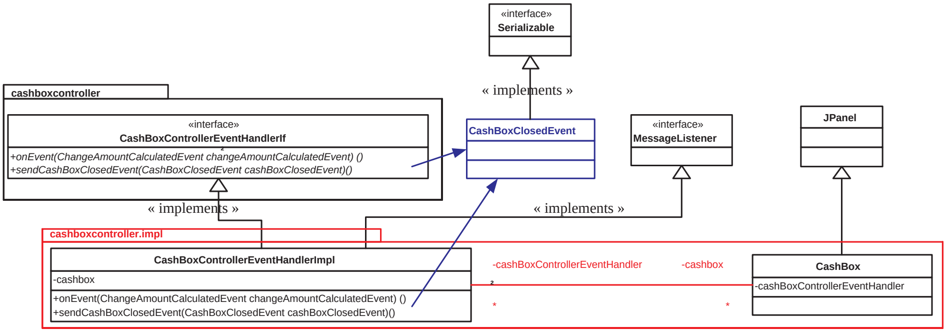

«interface»
Serializable
« implements »
cashboxcontroller
«interface»
CashBoxControllerEventHandlerIf
+onEvent(ChangeAmountCalculatedEvent changeAmountCalculatedEvent) ()
+sendCashBoxClosedEvent(CashBoxClosedEvent cashBoxClosedEvent)()
CashBoxClosedEvent
«interface»
MessageListener
JPanel
« implements »
« implements »
cashboxcontroller.impl
CashBoxControllerEventHandlerImpl
-cashbox
+onEvent(ChangeAmountCalculatedEvent changeAmountCalculatedEvent) ()
+sendCashBoxClosedEvent(CashBoxClosedEvent cashBoxClosedEvent)()
-cashBoxControllerEventHandler
-cashbox
*
*
CashBox
-cashBoxControllerEventHandler

Figure: CoCoME component: the `CashBoxController` implementation

## Outline

## Finding and Writing the Annotations

- Annotations
  - are a link between code elements and model elements (trace)
  - must conform the Econet (java) definitions
  - cannot be seen individually (consistent set/pattern of annotations)
- Finding Annotations
  1. study engineering $\Longrightarrow$ patterns
  2. study reverse engineering $\Longrightarrow$ patterns
  3. define Rule based system
- Writing annotations
  - where
    - In the code
    - In the model
  - how
    - manually
    - automatic tool support
  - why : solve conflicts (existing, previous)

## Annotation Mappings

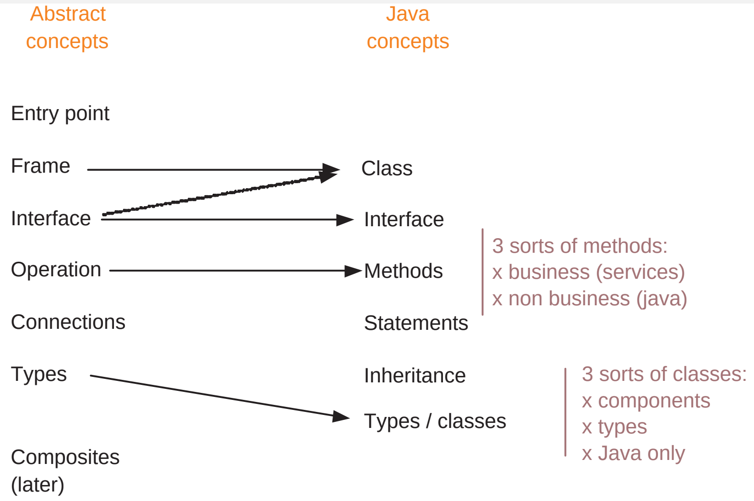


Figure: Mapping concepts

## Finding and Writing the Annotations: Obstacles

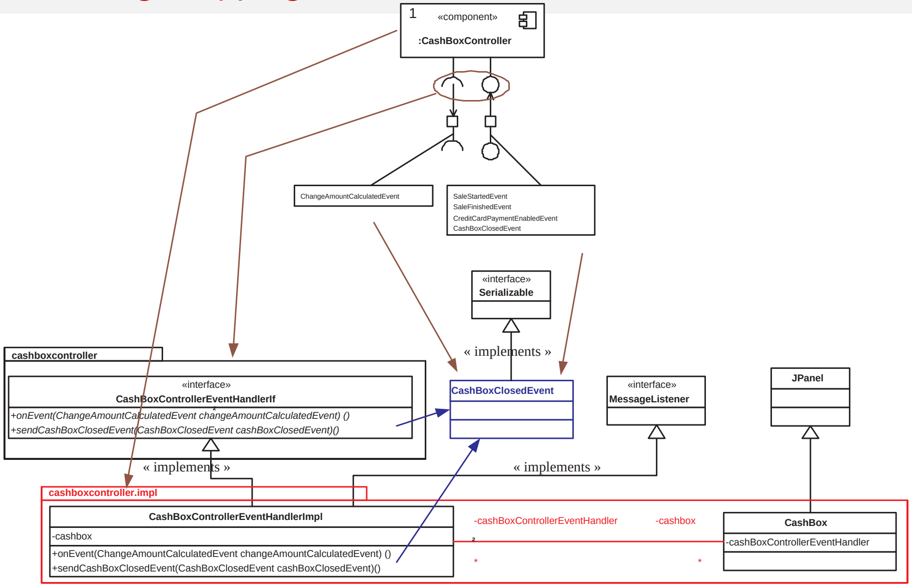
- Incompleteness: the model can be incomplete
  the CoCoME UML model is ”some” view of the implementation, the full model is the couple UML-Java
- Holes: a model element may have no correspondence
  composite components, sequences have not correspondence
- Inconsistency: the model may be inconsistent
  the proposed diagrams are clearly not defining an integrated component model, but it is a collection of UML diagrams with an intuitive semantics.
- Regularity: a transformation pattern can be sytematic or not
  data/control patterns for components, application/controllers
- Traceability: informations on the design and implementation process
  no direct (model) transformation, no reference are given to the abstract model
- Noise: informations of implementation model
  how to distinguish technical packages from business packages without new information ?
  how to consider a Java class is a business type ? additionnal information (fields, parameters) is to be removed before comparing with abstract concepts

## Annotating Process

1. Select the concepts in the model
   - individual concepts
   - related concepts

   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, ...
2. Finding Mappings
   1. structural
   2. behavioural
3. Exploring the code
   1. define reverse rules
   2. special cases
4. Writing annotations

## 2- Finding Mappings


1
«component»
:CashBoxController
ChangeAmountCalculatedEvent
SaleStartedEvent
SaleFinishedEvent
CreditCardPaymentEnabledEvent
CashBoxClosedEvent
«interface»
Serializable
« implements »
cashboxcontroller
«interface»
CashBoxControllerEventHandlerIf
+onEvent(ChangeAmountCalculatedEvent changeAmountCalculatedEvent) ()
+sendCashBoxClosedEvent(CashBoxClosedEvent cashBoxClosedEvent)()
CashBoxClosedEvent
«interface»
MessageListener
JPanel
« implements »
« implements »
cashboxcontroller.impl
CashBoxControllerEventHandlerImpl
-cashbox
+onEvent(ChangeAmountCalculatedEvent changeAmountCalculatedEvent) ()
+sendCashBoxClosedEvent(CashBoxClosedEvent cashBoxClosedEvent)()
-cashBoxControllerEventHandler
-cashbox
*
*
CashBox
-cashBoxControllerEventHandler

## 2- Finding Structural Mappings

### 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.

## 2- Finding Structural Mappings

### 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.

## 2- Finding Structural Mappings

### 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.

## 2- Finding Structural Mappings

### 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.

## 2- Finding Structural Mappings

Rule Set ?

- The above mapping seems to be convenient for `CashDeskLine` subcomponents.
- These components are somewhat related to dynamic aspects of the model.
- But 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.

## 2- Finding Structural Mappings

In the case of `Inventory` components, 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.

## TradingSystem::Inventory::Application component

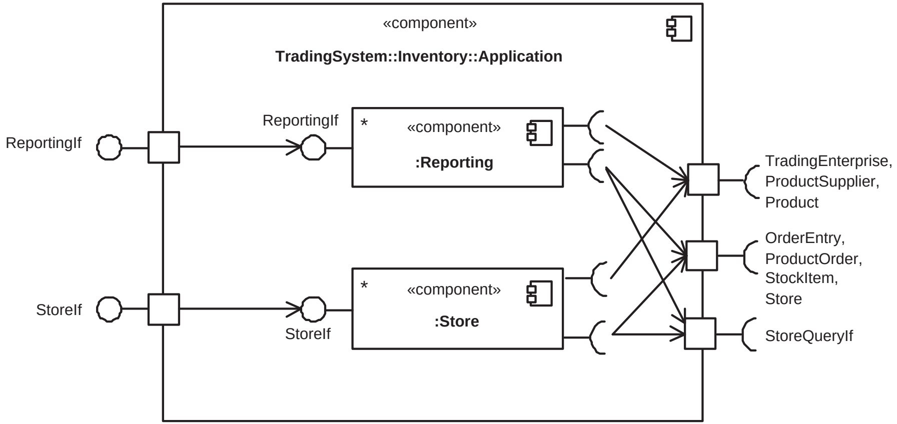


Figure: CoCoME component: the `TradingSystem::Inventory::Application` component

## 2- Finding Behavioural Mappings

### Message Patterns

- Message names are built using some conventions. In the `CashDeskLine` they all end by `Event` and some start with `send`
- 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?
- This applies for process oriented code, what about data ?
- Nothing about dynamic behaviour, just basic (static) one.

### Entry Points

- no explicit in UML
- look for constructors and main methods

## 3- Exploring the code

### Annotations to put

- *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

## 3- Exploring the code

### a) 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. 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.

## 3- Exploring the code

### a) Java interfaces (Reverse Rules)

**RR-1 (Java Interface)**
*Java interfaces are not annoted.*

**RR-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.*

**RR-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.*

**RR-4 (Java Interface Qualification)**
*Java interfaces can be qualified as both provided or required.*

## 3- Exploring the code

### a) Java interfaces (Example)

```
package org.cocome.tradingsystem.cashdeskline.cashdesk.cashboxcontrol

import org.cocome.tradingsystem.cashdeskline.events.CashAmountEntered
import org.cocome.tradingsystem.cashdeskline.events.CashBoxClosedEven
import org.cocome.tradingsystem.cashdeskline.events.ChangeAmountCalcu
import org.cocome.tradingsystem.cashdeskline.events.ExpressModeDisabl
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);
}
```

## 3- Exploring the code

### b) Class declaration (Reverse Rules)

**RR-5 (Business Class)**
*In the class declaration we add the annotation (@InComponent) that link the class to the component.*

**RR-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.*

**RR-7 (Business Main Class)**
*If the class is the main entry point a component we add the annotation (@InitClass).*

## 3- Exploring the code

### b) Class declaration (Reverse Rules)

**RR-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`.*

**RR-9 (Business Type)**
*Some classes correspond to business datatypes and not components. We found them in the data description.*

## 3- Exploring the code

### b) Class declaration (Example 1)

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

import ...
import econet.annotations.*;

@SuppressWarnings("serial")
@InComponent(annotationSrc = {"Pascal"},
componentName = {"CashBoxController"})
@InitClass(annotationSrc = {"Pascal"},
componentName = {"CashBoxController"})
public class CashBox extends JPanel {...
}
```

## 3- Exploring the code

### b) Class declaration (Example 2)

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

import ...

@InComponent(annotationSrc = {"Pascal"},
componentName = {"CashBoxController"})
@ProvidedIf(annotationSrc={"Pascal"},
modelIfaceName={"CashBoxControllerIf"},
javaIfaceName={"CashBoxControllerEventHandlerIf"})
public class CashBoxControllerEventHandlerImpl
implements MessageListener,
CashBoxControllerEventHandlerIf {...
}
```

## 3- Exploring the code

### b) Class declaration (Example 3 Type)

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;

@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;
}
}
```

## 3- Exploring the code

### c) Class 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).

## 3- Exploring the code

### c) Class structure (Reverse Rules)

**RR-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).*

**RR-11 (Business field)**
*Based on business types, one can find the business field an annotate them.*

**RR-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.*

## 3- Exploring the code

### c) Class structure (Reverse Rules)

**RR-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.*

## 3- Exploring the code

### c) Class structure (Example 1)

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`.

## 3- Exploring the code

### c) Class structure (Example 1 (contd.))

```
public class CashBoxControllerEventHandlerImpl implements MessageListe
CashBoxControllerEventHandlerIf {
final String CHANNEL_CONNECTION_FACTORY = "ChannelConnectionFactory";
private String topicName;
/implementation references
@Required(annotationSrc = {"Pascal","Model"}, modelIfaceName = { "impl
private Context jndiContext;
@Required(annotationSrc = {"Pascal","Model"}, modelIfaceName = { "impl
private TopicPublisher cashBoxPublisher;
@Required(annotationSrc = {"Pascal","Model"}, modelIfaceName = { "impl
private TopicSession topicSession;
@Required(annotationSrc = {"Pascal","Model"}, modelIfaceName = { "impl
private Logger log = Logger
.getLogger(CashBoxControllerEventHandlerImpl.class);
/internal references
@Required(annotationSrc = {"Pascal"}, modelIfaceName = { "internal" }
private CashBox cashbox;
```

## 3- Exploring the code

### d) 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.

## 3- Exploring the code

d) Class behaviour (Reverse Rules)

**RR-14 (InitMethod)**
*The InitMethod is chosen among the constructor or initialization methods.*

**RR-15 (ProvidedMethod)**
*The "business" methods signature refer to service or operations. The business qualification is decided manually.*

## 3- Exploring the code

### d) Class behaviour (Example 1)

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 = {"CashBoxCont
public CashBox(String eventchannel) {
super();
...}
```

## 3- Exploring the code

### d) Class behaviour (Example 2)

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 = {"CashBo
public void onEvent(ChangeAmountCalculatedEvent changeAmountCalculate
log.info("ChangeAmountCalculatedEvent received");
cashbox.openCashBox();
}

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

## 3- Exploring the code

### e) Methods (Reverse Rules)

**RR-15 (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.*

## 3- Exploring the code

### e) Methods (Example)

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

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

## 3- Exploring the code

### f) Composition (composite)

- 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.

## 4- Writing annotations: Annotated classes

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 `LightDisplayControllerEvent HandlerImpl` but there is a unidirectional internal link from controller to display.

## 4- Writing annotations: Annotated classes (contd.)

- 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.

## Outline

## UML/Java:Introduction

In the context of reverse engineering Java code, there are several approaches

1. compose two transformations Java to UML ∘ 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)
3. Find a mapping Component UML - Java

## UML/Java:Introduction

In the context of reverse engineering Java code, there are several approaches

1. compose two transformations Java to UML ∘ UML to components
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

## UML/Java:Introduction

In the context of reverse engineering Java code, there are several approaches

1. compose two transformations Java to UML $\circ$ UML to components
2. compare an existing UML component model with Java to set annotations (e.g. the Test1 experimentation)
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 (?)

## UML/Java:Introduction

In the context of reverse engineering Java code, there are several approaches

1. compose two transformations Java to UML $\circ$ UML to components
2. compare an existing UML component model with Java to set annotations (e.g. the Test1 experimentation)
3. Find a mapping Component UML - Java

- 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.
  $\Longrightarrow$ set of recognition patterns.

## Mapping UML Components to (Java) Classes

### 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

## Mapping UML Components to (Java) Classes

### Basic Component Pattern

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

## Mapping UML Components to (Java) Classes

### 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

reverse = find the pattern

## Outline of the Talk

## 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

## Perspectives

### Ongoing Work

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