The rCOS Method and Modeler

Zhenbang Chen, Zhiming Liu, and *Volker Stolz*
vs@iist.unu.edu

http://rcos.iist.unu.edu

United Nations
University

UNU-IIST
International Institute for
Software Technology

rCOS Overview

- Motivation: Use case-driven component based S.E.
- rCOS Models and their refinement and composition
- Component based model driven design

The rCOS Modeler

- Why UML?
- The rCOS UML profile
- The rCOS Modeler

Aims

Deal with two challenges in software engineering

1. inherent complexity of Software Projects
   - multiple aspects: structural, functionality, interaction, security, timing, distribution, mobility and general QoS
   - most aspect are interrelated and assurance consistency is hard
2. ensure correctness of software systems
   - formal modelling, design, verification and validation

State of Practical SE

Deal with complexity of *component-based and model driven* development through

- UML-like multi-view modelling of different aspects
- Separation of design and validation of different concerns by design patterns, object-oriented and component-based designs techniques

No rigorous theories and tools for specification, verification and validation

Objectives of rCOS

1. Incorporating formal methods and tools of modelling, design and VV into model-driven development process:
   - model different views and analyse correctness of different concerned with different VV techniques and tools
   - automate verified design patterns and strategies to reduce the burden on (automated) verification
2. Provide a semantic foundation for relating the methods and the integrating of tools of VV with those of design

Putting theories, methods and tools consistently together in design processes

Strands of Research on rCOS

1. Theory: a modelling language, its semantics, refinement calculus, analysis and verification of models
2. Tool support: an integrated tool suite to support model construction, model transformation and model verification
3. Applications: develop a set of verified case studies, trading system, e-government, etc
4. Knowledge and technology transfer: teach a coherent and comprehensive methodology that begins with design for verification and validation and integrates verification into development process

rCOS in a Nutshell

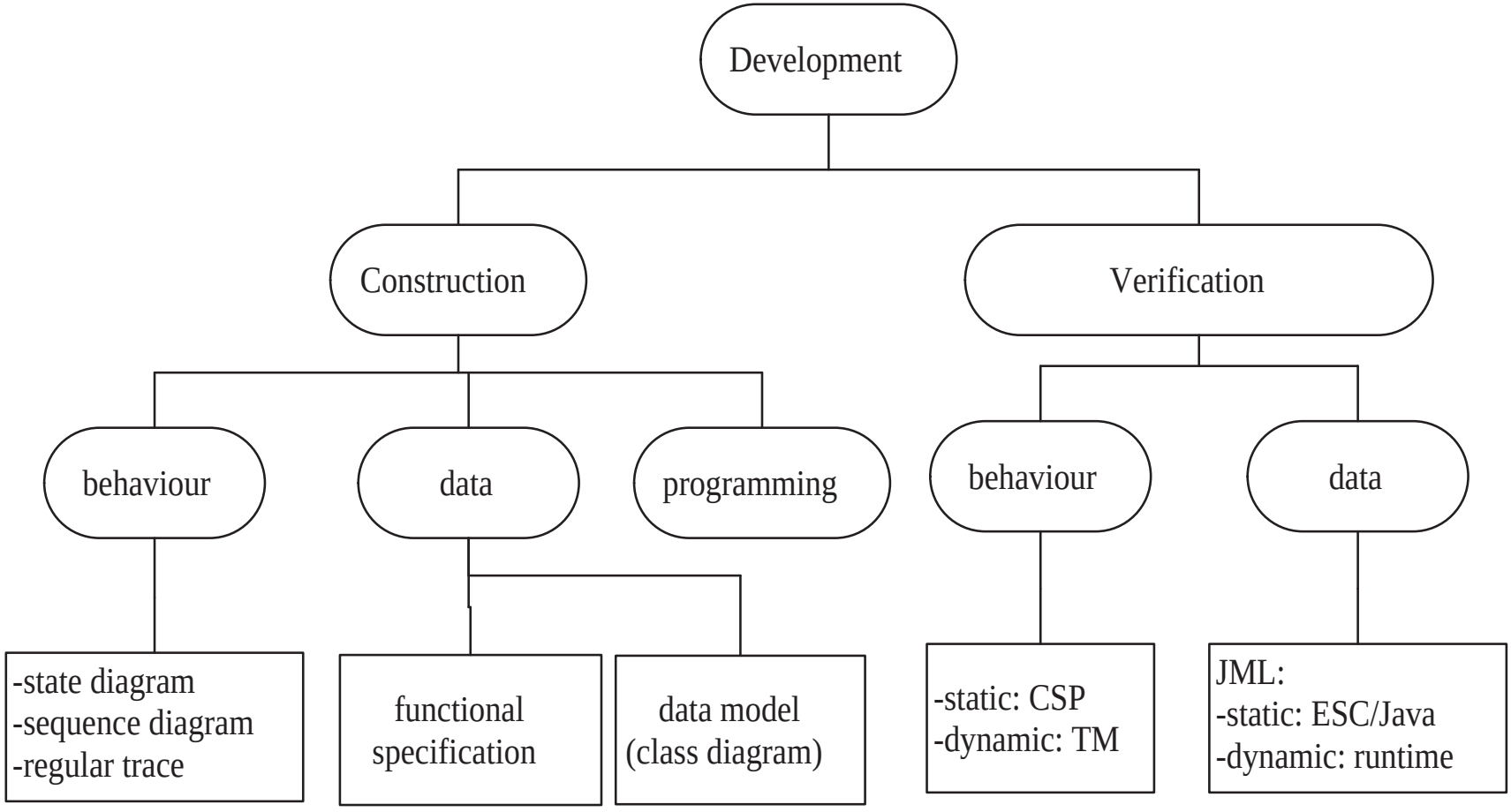
rCOS is a multi-view and multi-notational modelling framework supporting the two key techniques of trustworthy software development:

- Separation of concerns and aspects
- Formal modelling of requirements, design and analysis

rCOS specifies and analyzes models of

- application requirements,
- object-oriented designs and refinement,
- component-based architecture,
- component interfaces, and their contracts,
- processes for glue and application programs.

Overview


Development
Construction
Verification
behaviour
data
programming
behaviour
data
-state diagram
-sequence diagram
-regular trace
functional specification
data model (class diagram)
-static: CSP
-dynamic: TM
JML:
-static: ESC/Java
-dynamic: runtime

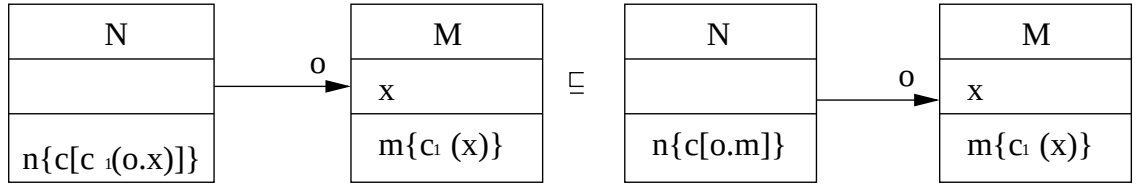
Object-Orientation in rCOS [TCS06]

An OOP: $P = Classes \bullet Main$

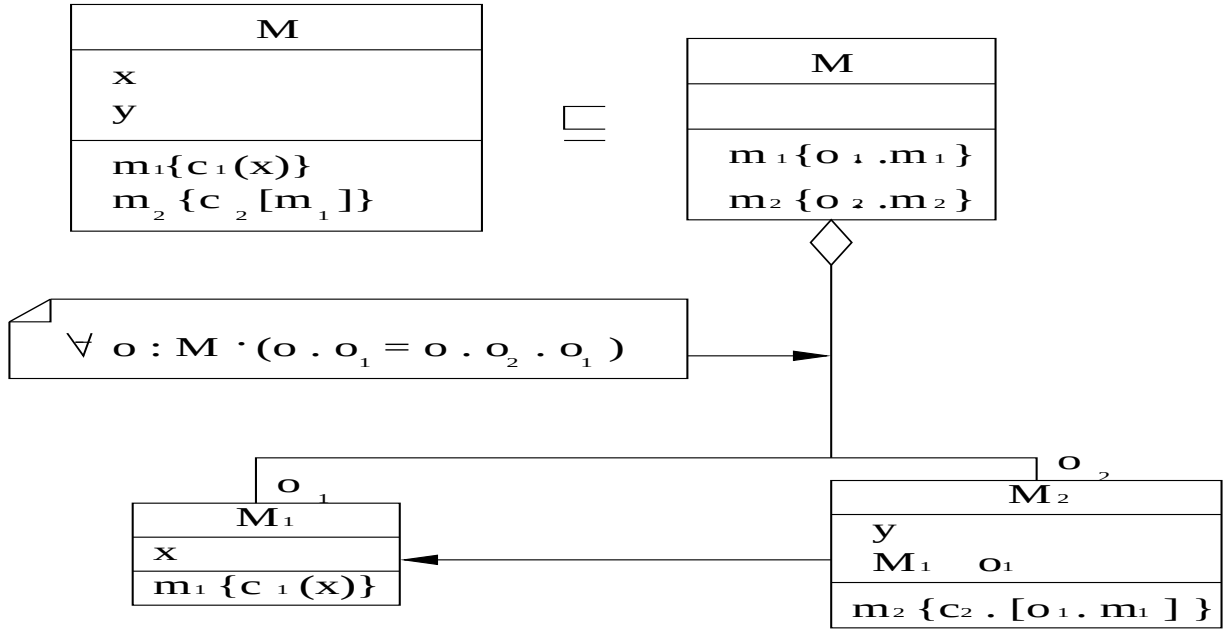
- *Classes*: an list of *class declarations*
- *Main*: *main class* with a *main* method
- *Classes*: represented by a (or a set of) UML class diagram(s)
- *Main*: program of use cases (use case controller classes)
- *Classes*: represented by a directed and labeled graph
- A state of P: a rooted, directed and labelled graph/UML Object Graph [ICFEM03,REFINE06]
- A relational semantics based on UTP [TCS06]
- A refinement calculus for both functional refinement and structure refinement [TCS06]
- Support analysis, design, refactoring and code generation [ICTAC05]

Functional Refinement [TCS06]

Functional Decomposition

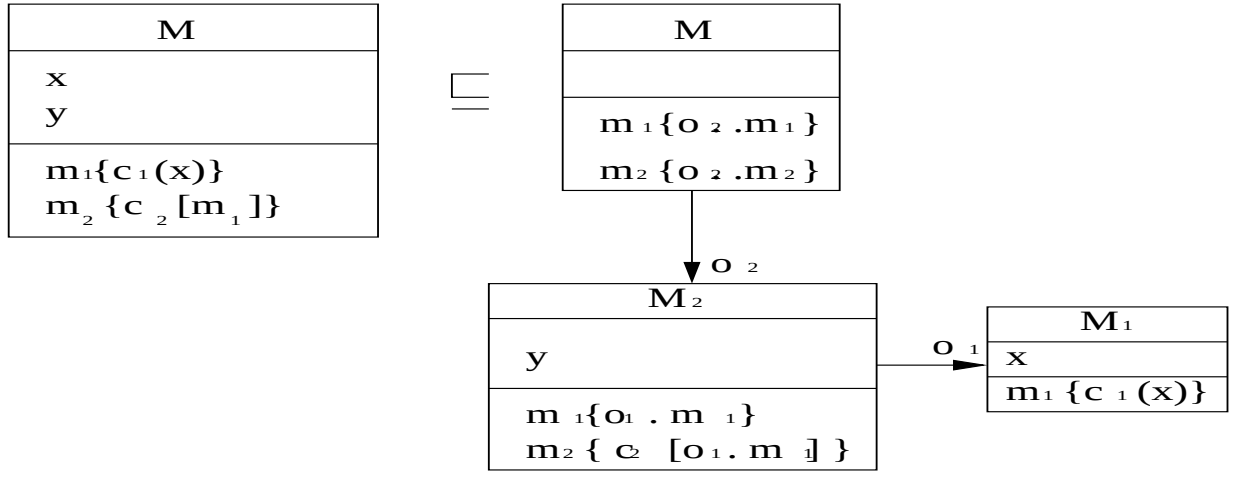


Class Decomposition 1

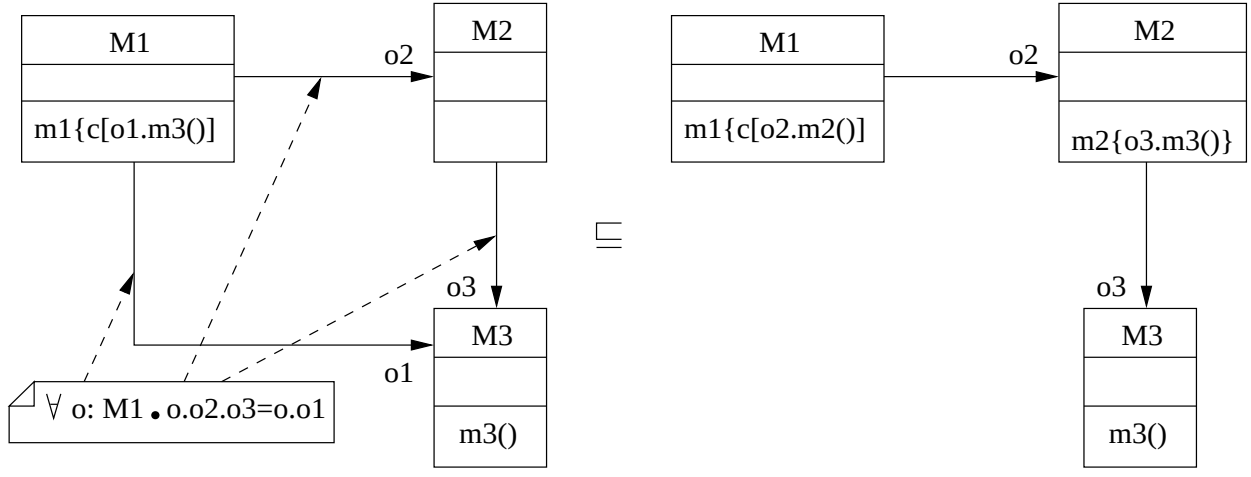


More Refinement

Class Decomposition 2



Low Coupling

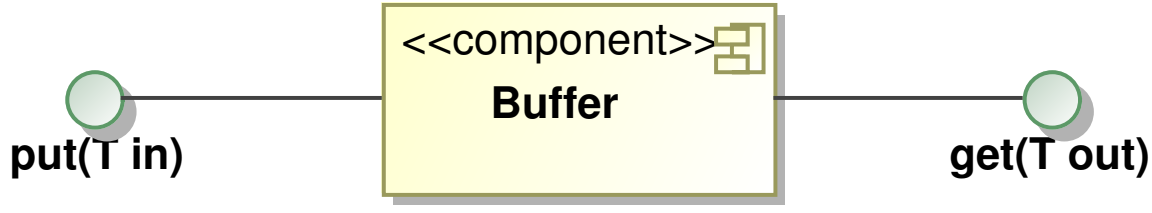


rCOS Model of Components [FACS05, FSEN07]

- Interfaces: operation signatures for syntactic compositionality
- Contracts: interface specifications including static & dynamic behavior, interaction protocol, timing ...; refinement
- Components: Provided and required interface + code
- Object rCOS: provides a common semantics for different PLs to implement components (interoperability)
- Semantics of Components: relation between components and contracts (correctness), substitutability
- Composition Operations: simple connectors
- Component-Based Programming: glue, application processes

Interfaces and their Contracts [ICTAC05]

- An *interface* of a component is a description of what is needed for the component to be *used* in building and maintaining software without the need to know the code of the component.
- Interface determines external features of component and allows component to be used as a black box.
- Interfaces determine substitutability of components



Contracts

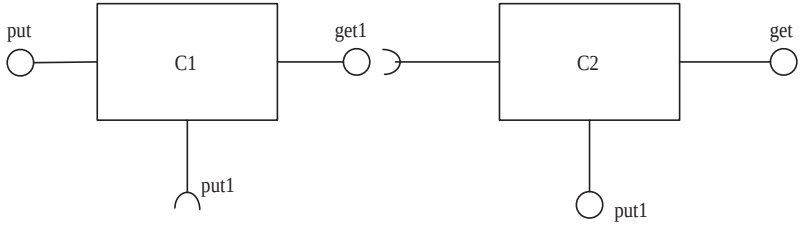
A *contract* is a tuple $Ctr = (I, Init, MSpec, Prot)$

- *MSpec* assigns each operation to a guarded design $g \& D$.
- *Prot* is called the *protocol* and is a set of sequences of call events; each is of the form $?op_1(x_1), \ldots, ?op_k(x_k)$
- A contract *Ctr* is *consistent*, if it will never enter a deadlock state if its environment interacts with it according to the protocol:
  For all $\langle ?op_1(x_1), \ldots, ?op_k(x_k) \rangle \in Prot$,

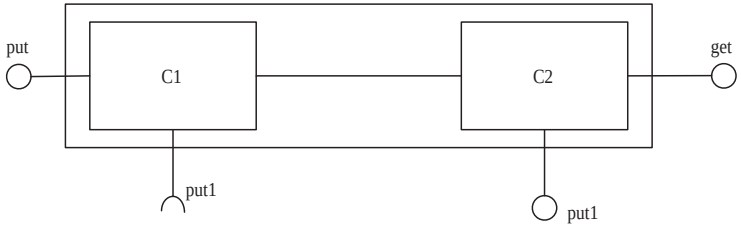
$$\mathbf{wp}\left(\begin{array}{l} Init; g_1 \& D_1[x_1/in_1]; \ldots; g_k \& D_k[x_k/in_k], \\ \neg wait \wedge \exists op \in MDec \bullet g(op) \end{array}\right) = true$$

Component Composition

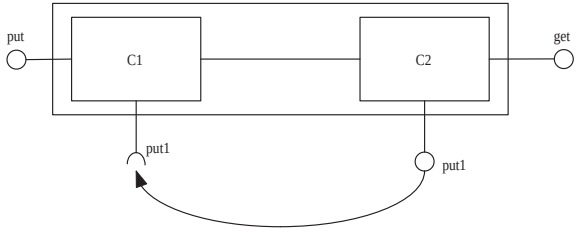
Chaining



Hiding after Chaining

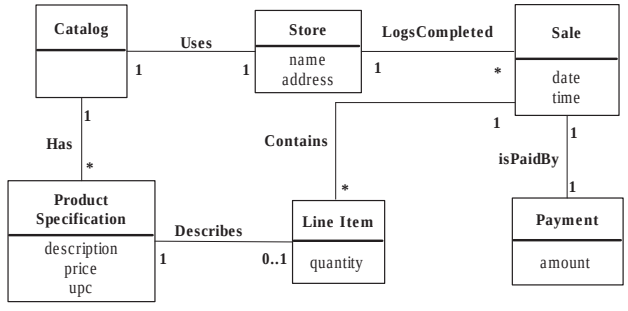


Feedback

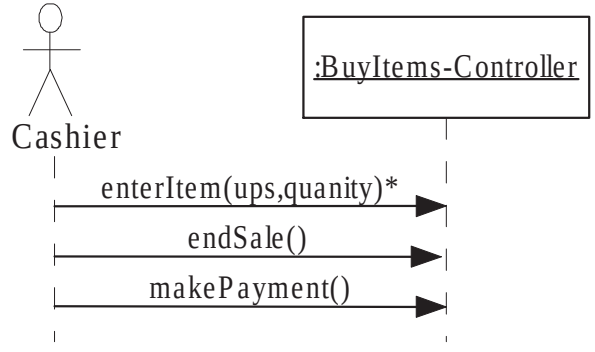


Models of Application Requirements

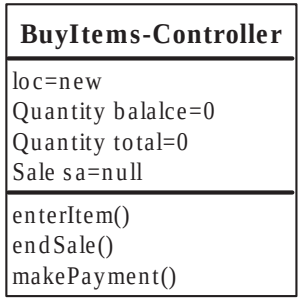
- Conceptual class model
- use-case model – static functionality + interaction and dynamic behavior
- consistency and integration defined in rCOS [SOFSEM 2007]



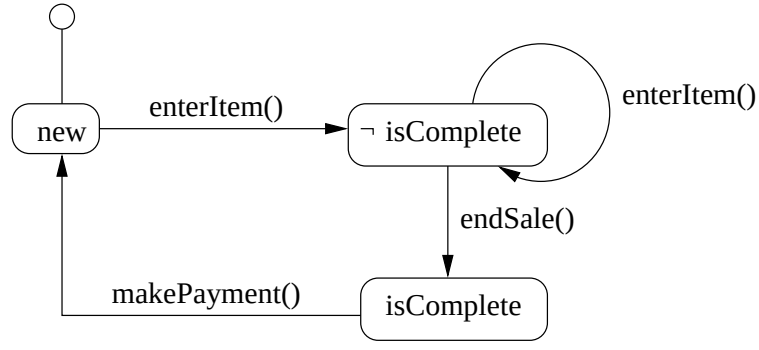
(a)



(b)



(c)

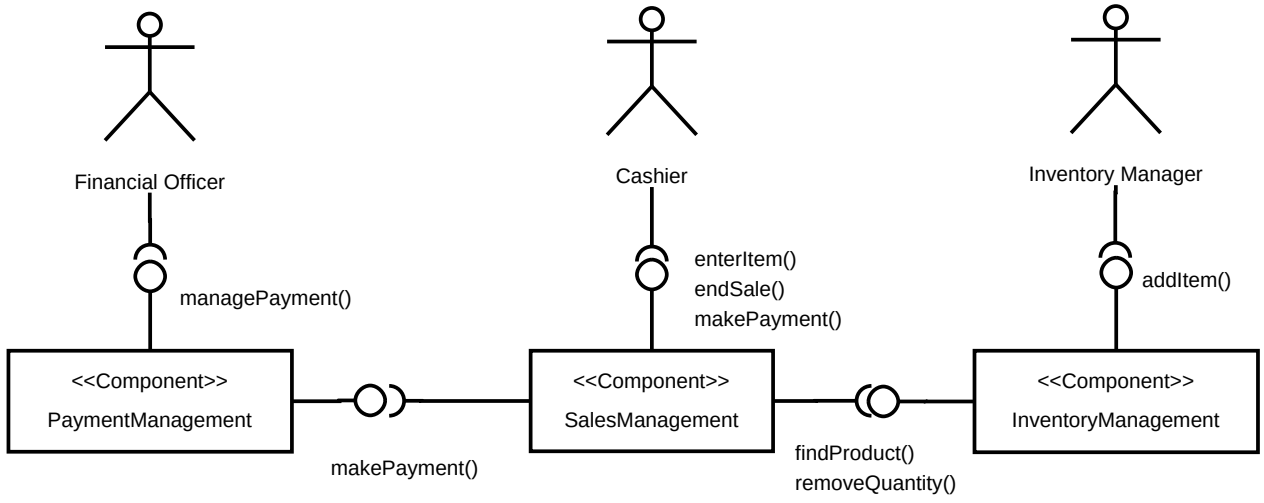


(d)

Requirements Analysis

- Static consistency (think "*compiler*"):
  - all types and methods are defined in class diagram
  - type checking of signatures and statements
- Dynamic consistency of SeqD, StD and ClassD
  - $[\![SeD]\!] \parallel [\![StD]\!]$ deadlock free
  - application dependent properties: properties of the StD
  - Dynamic checking: e.g. through FDR
- Automatic Prototype Generation

Component Architecture Design



- Assign classes and associations to components according to the use cases – partition the state space.
- Decomposing use case sequence diagrams into component sequence diagrams – define interaction protocols of components.
- Verifying the decomposition against the application requirements model.

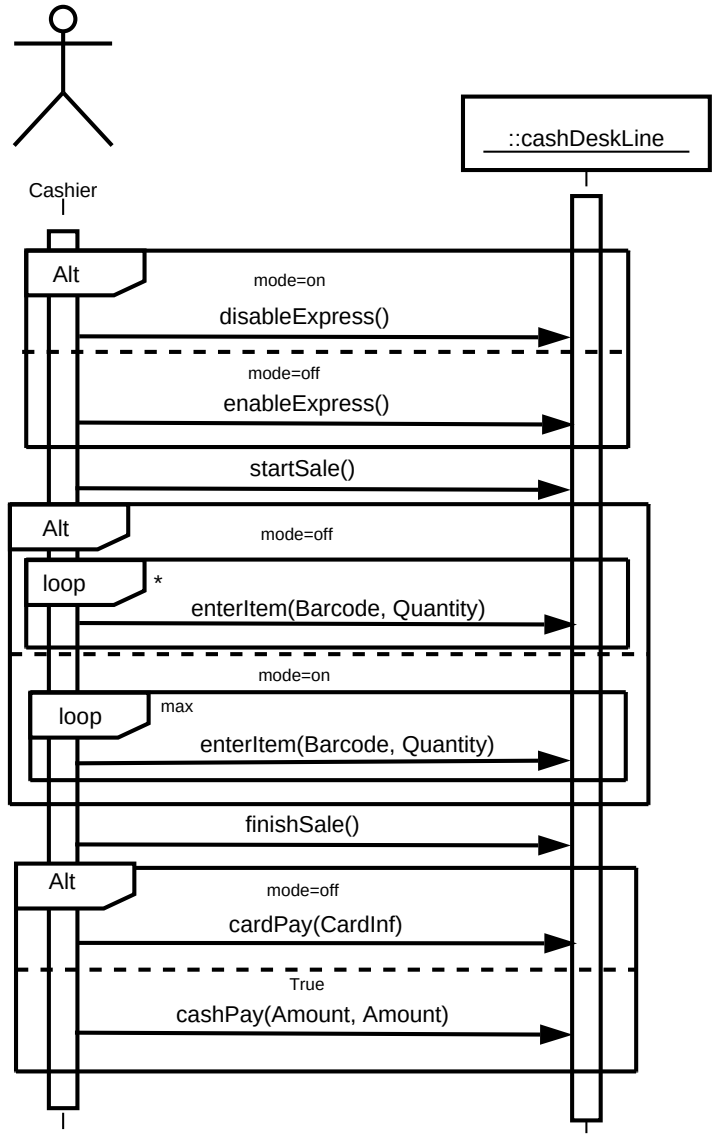
Applied to CoCoME

- informal common problem description
- component-based
- various aspects to formally model and analyse
- generate code

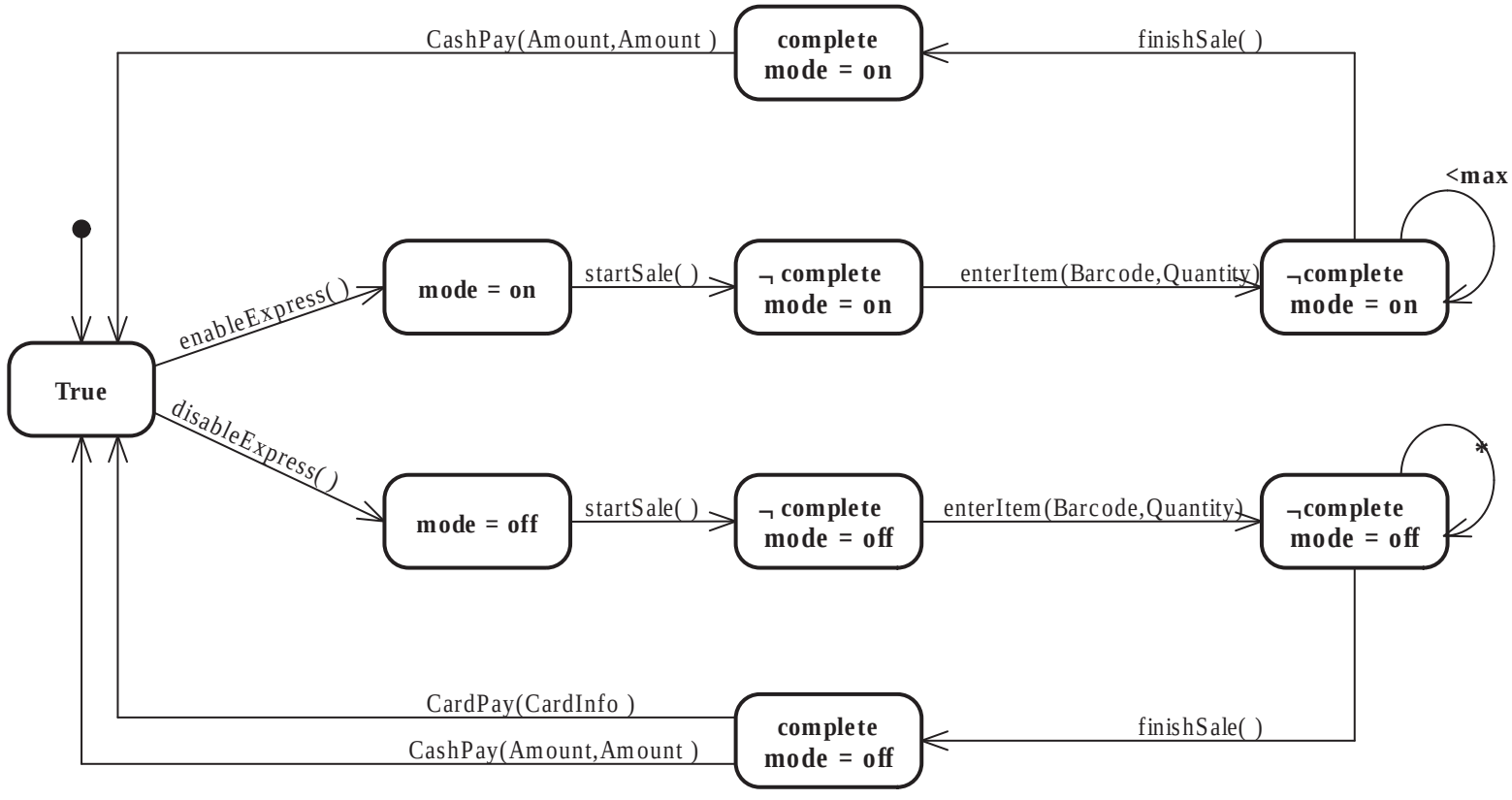
Case study:

- trading System: cashdesks with GUI and peripherals
- connected to store-server using a message bus
- enterprise-server connected to various stores via RMI
- 8 use cases of (inter)actions between entities
- GUI, hardware controllers (embedded systems design)
- formal verification/analysis

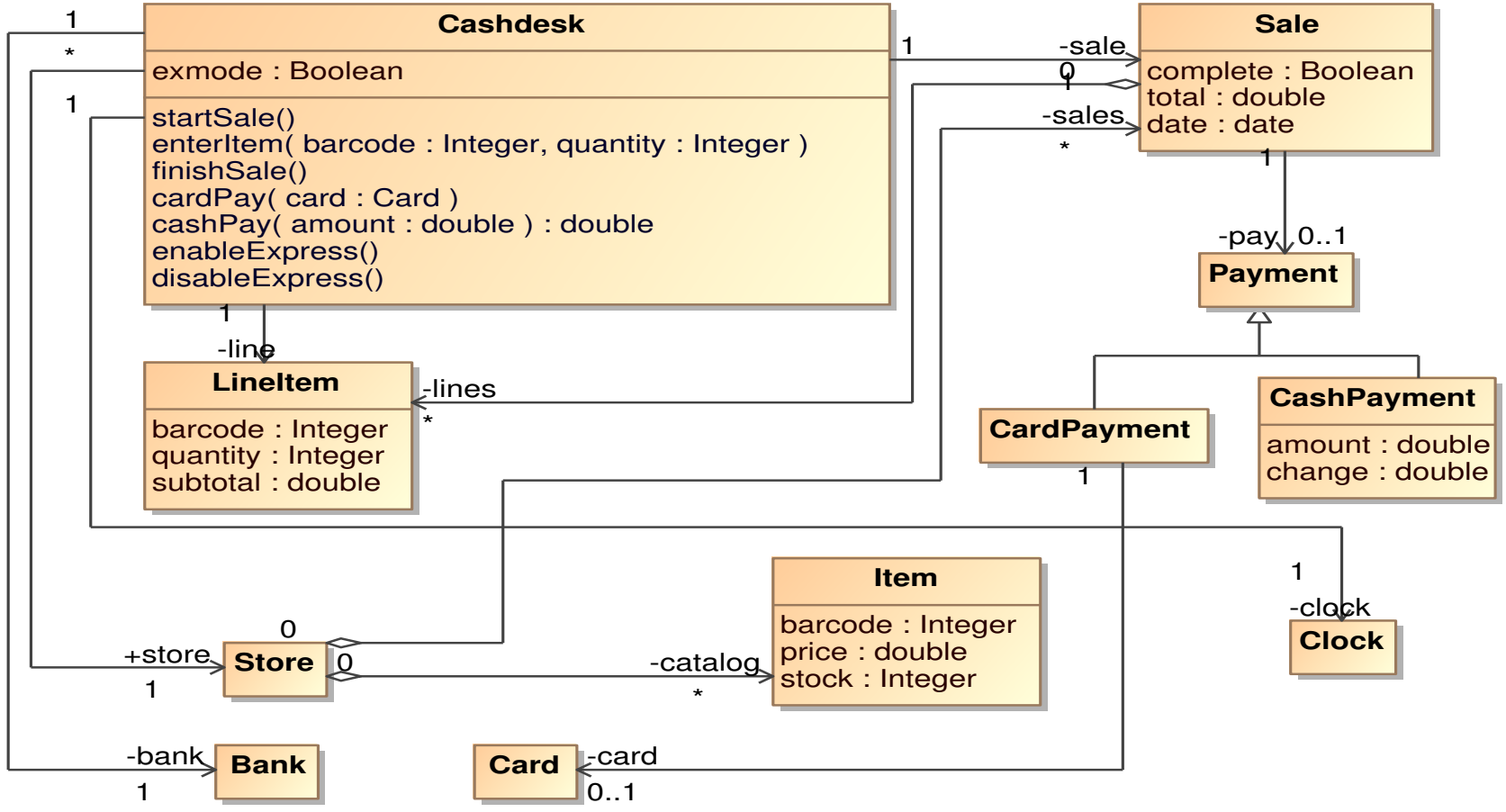
Interface Sequence Diagram



Interface State Diagram



Class Diagram



Functionality Specification of Method

Updating actions:

- object creation: $\ell' = C.New(e)$
- attribute or variable modification
  - $\ell' = \overline{a}.x$ – association link established
  - $\ell' = e(\overline{a}_1.d_1, ...\overline{a}_k.d_k)$ – data modified
- actions on sets: $s.find(ID\ id)$, $s.add(T\ a)$, $s.delete(T\ a)$

Conditions: $\overline{a}.x = null \mid c(\overline{a}_1.d_1, \ldots, \overline{a}_k.d_k)$

Quantification: $\forall T\ x : S(x)\ \exists T\ x : R(x)$

Specification:: $S ::= f : [p, R] \mid S; S \mid S \sqcap S \mid S \lhd b \rhd S \mid b * S$

$m(T\ x; U\ y)\{S\}$

Functionality Specification

Use Case **UC 1: Process Sale**

class *Cashdesk*{

method *enterItem(Barcode c, int q)* {

pre: /* there exists a product with the input barcode c */

store.catalog.find(c) $\neq$ *null*,

post: /* a new *line* is created with its *barcode c* and *quantity q*, and then */

line' $=$ *LineItem.New(c,q)*

/* the *subtotal* of the *line item* is set, and then */

$\wedge$ *line.subtotal'* $=$ *store.catalog*.find(c).*price* $\times$ q

/* add *line* to the current *sale* */

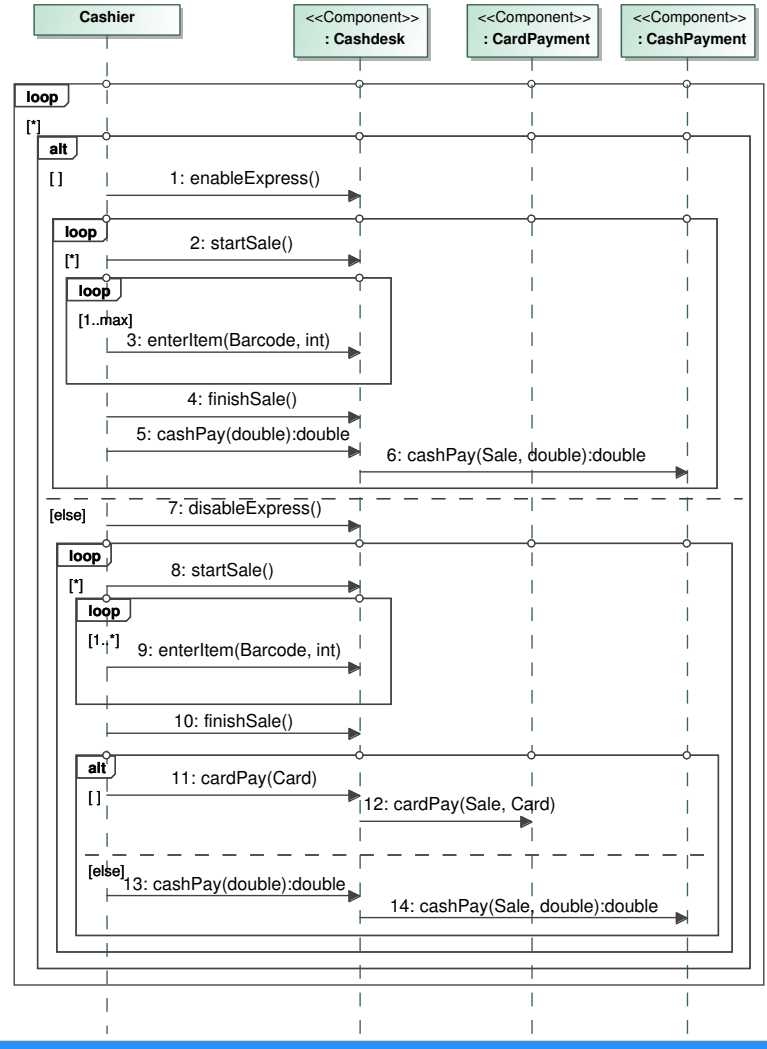
$\wedge$ *sale.lines'* $=$ *sale.lines* $\cup$ {*line*}

invariant *store* $\neq$ *null* $\wedge$ *store.catalog* $\neq$ *null* $\wedge$ *sale* $\neq$ *null*

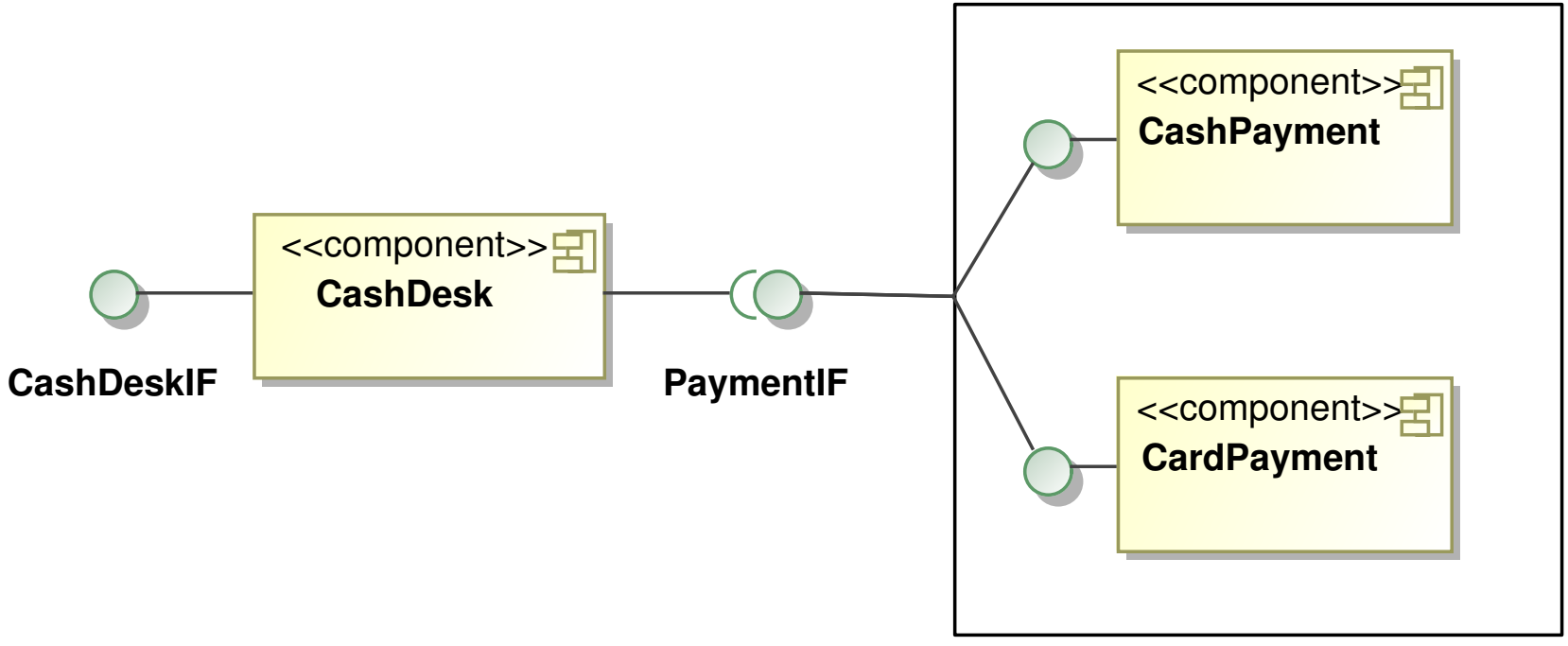
}

Use Case Decomposition and Composition

$$ProcessSale \mathrel{\widehat{=}} CashDesk \ll (CashPay|||CardPay)$$

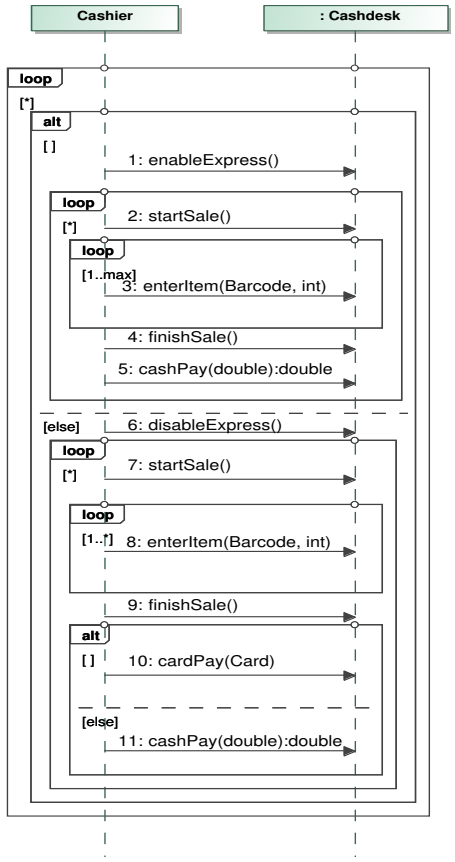


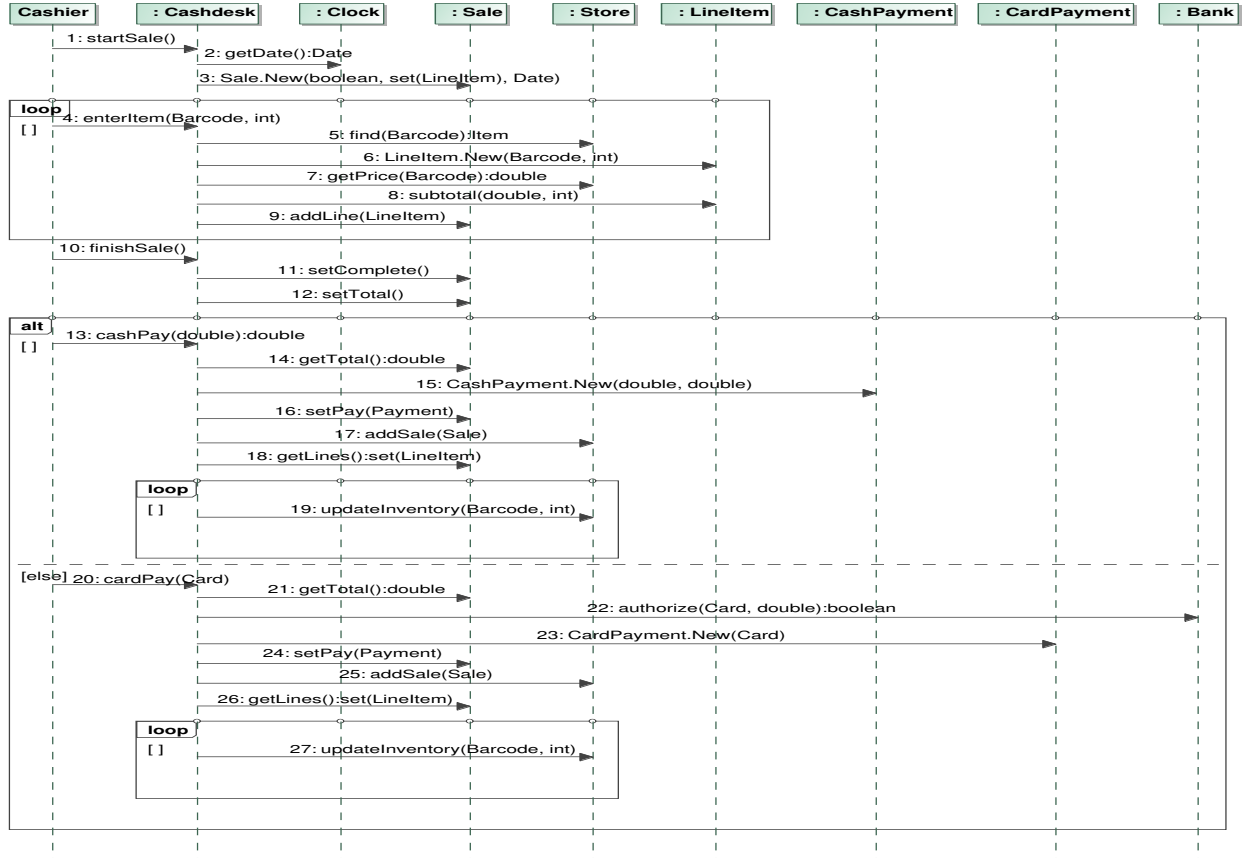
Component Diagram



Object-Oriented Refinement

Expert pattern for functional decomposition:





- Automate the expert pattern
- Design by drawing using provably correct rules/design patterns

Quantified Specification to Code

`updateInventory()`

Class *Cashdesk*::

$\forall l \in sale.lines,\ p \in store.catalog \bullet$ (**if** p.*barcode*=l.*barcode* **then**

p.*amount*' = p.*amount* -l.*quantity*)

yields almost executable Java with assertions:

`class` *Product*:: *update(int qty)* { *amount* := *amount-qty* }

`class` *set(Product)*:: *update(Barcode code, int qty)* {

Iterator i := *iterator()*;

while (*i.hasNext()*) {

Product p := *i.next()*;

if *p.barcode=code* **then** *p.update(qty)*; }

`class` *Store*:: *update(Barcode code, int qty)* { *catalog.update(code,qty)* }

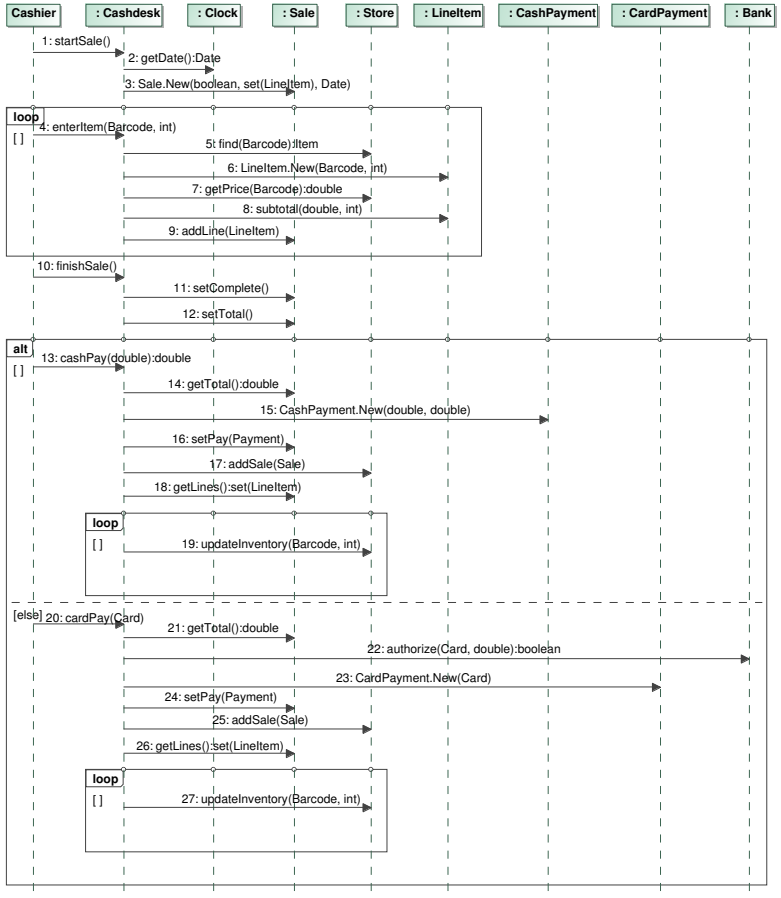
$\exists T\ o \in s : p(o) \wedge statement(o) \sqsubseteq \sqcap_{o \in s} p(o) \wedge statement(o)$

$\mathbf{boolean}\ b := true$; *Iterator* $i := s.iterator()$;

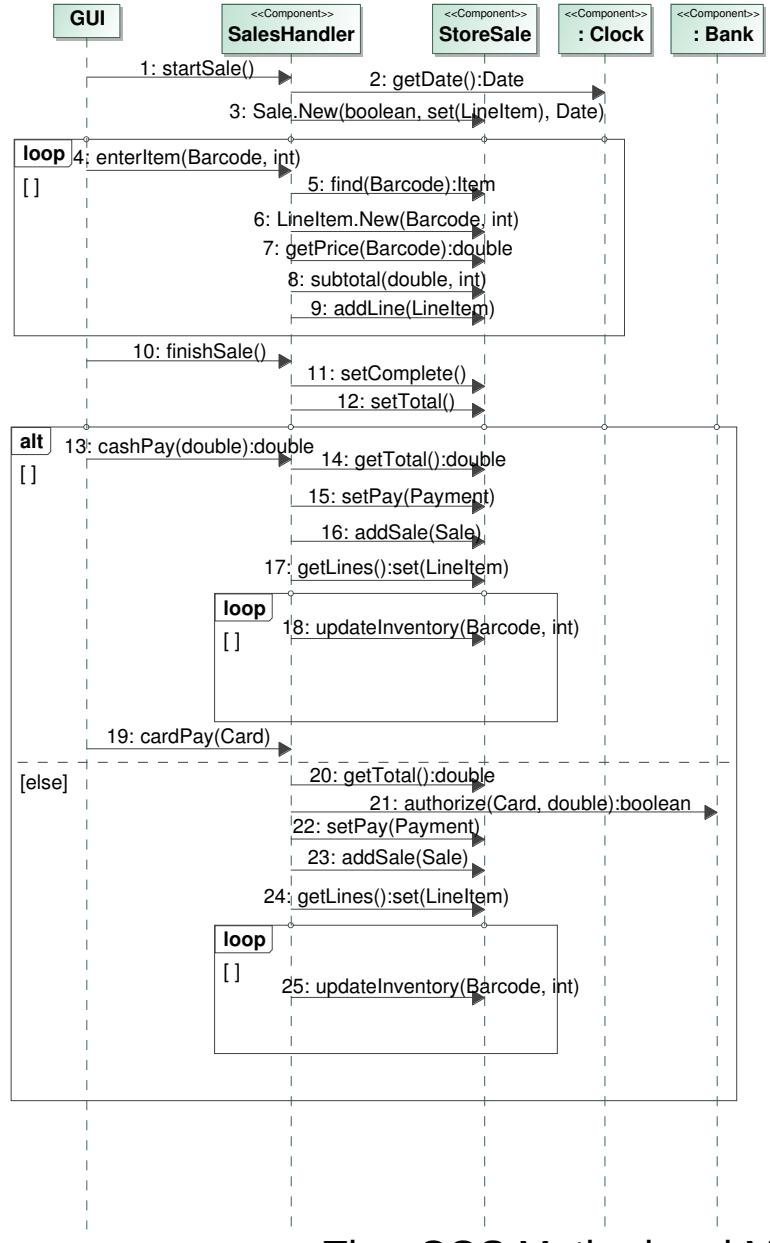
$\mathbf{while}\ i.hasNext() \wedge b \{T\ o := i.next(); \mathbf{if}\ p(o)\ \mathbf{then}\ \{b := \neg b; statement(o)\}\}$

FM should take advantages of high level PL!

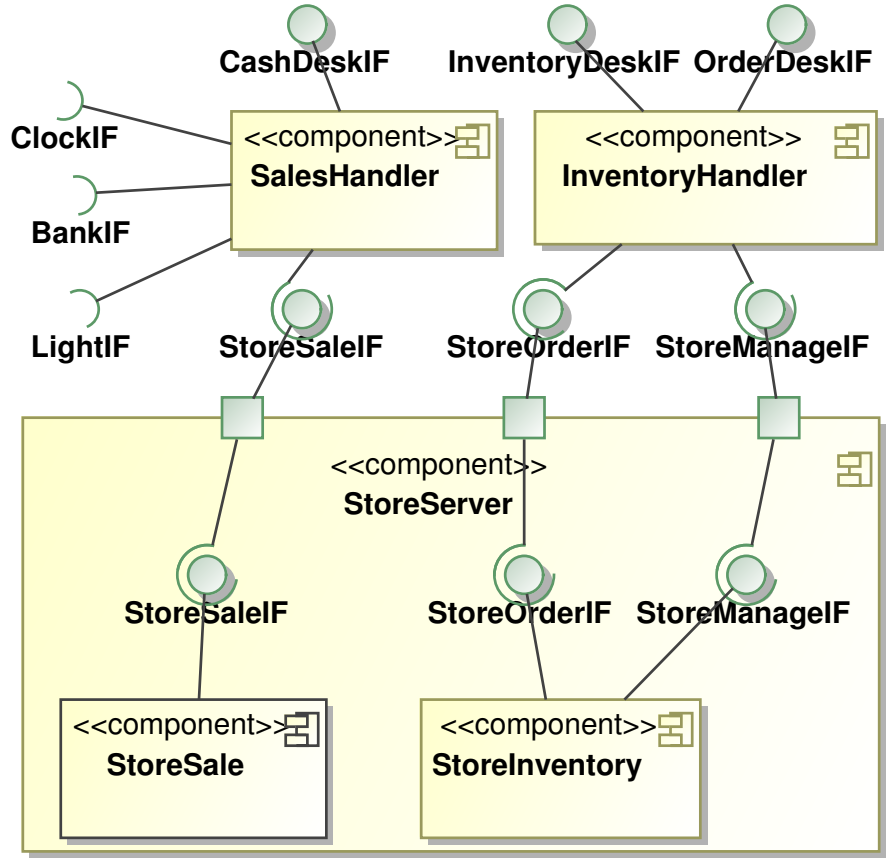
Component-Based Logical Design



$$ProcessSale \mathrel{\widehat{=}} SalesHandler \ll (StoreSale \parallel Clock \parallel Light)$$



From OOA to CBA



- identify components and intercomponent interfaces
- choose middle ware

The rCOS Modeler

rCOS Modeler

- “Guided tour” through use-case driven development process
- Uses UML models
- Diagramming support built on top of Eclipse/TOPCASED
  `http://www.topcased.org`
- Implemented features:
  - CSP generation from state- and sequence diagrams
  - Expert Pattern transformation
  - Modelling the Modeler in the Modeler!

Design Decision

- Advantages of using UML
  - many artifacts overlap: (classes/methods/associations, state machines, sequence diagrams)
  - existing tool support for UML modeling
  - easy to store additional data
- Disadvantages of using UML
  - allows incomplete specifications
  - not every UML model an rCOS model
  - subtle differences may confuse the casual user
  - some datastructures convoluted (e.g. in state machines)

UML Stereotypes/Profiles

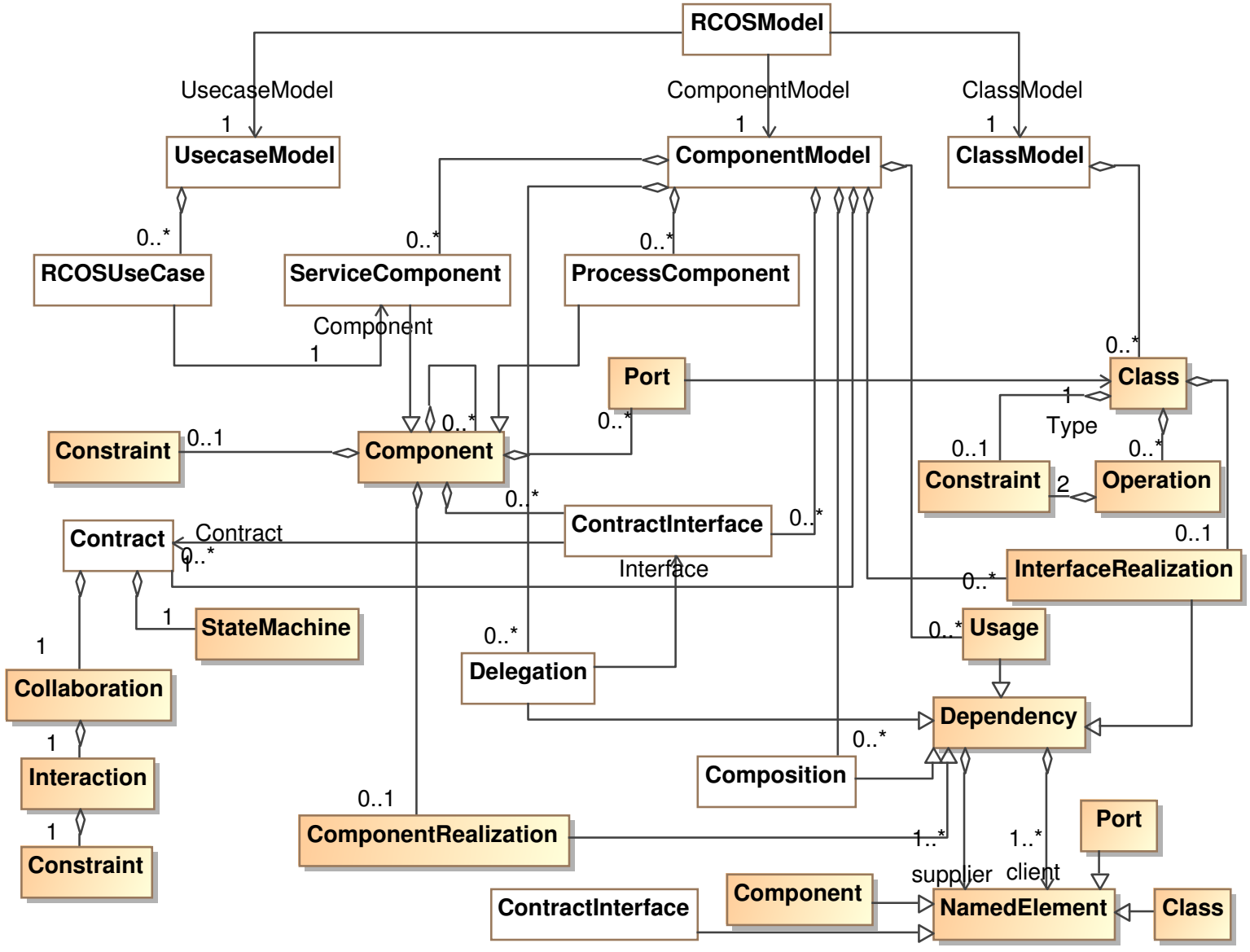
Stereotypes. . .

- allow extension of UML through *specialization*
- *tagged values* introduce attributes
  (very much like OO modeling)
- may use *constraints* in OCL
- have user-defined semantics
- may be processed by other tools

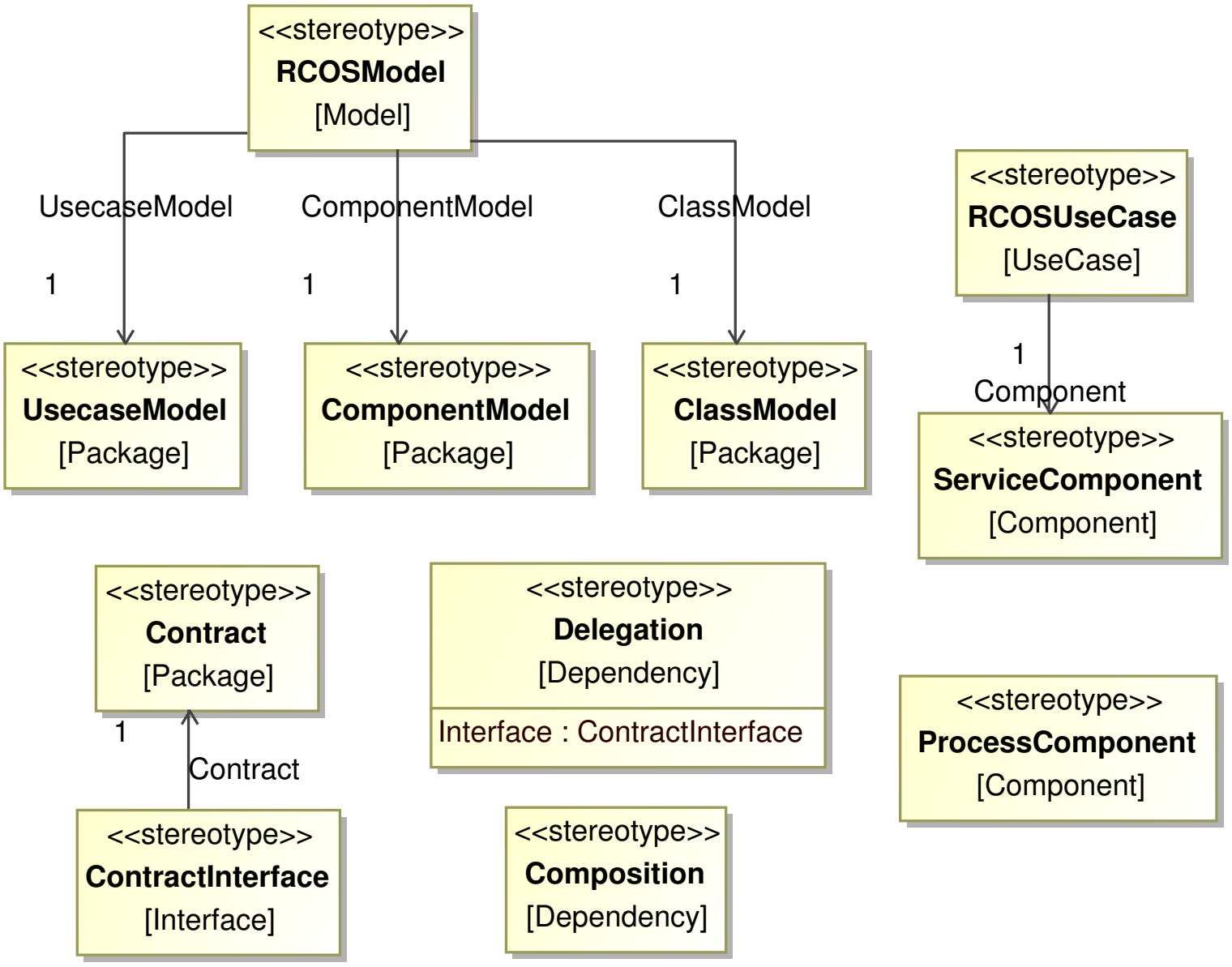
Profiles. . .

- define set of stereotypes and tagged values
- existing, standardized profiles for many purposes
- graphically specified similar to class diagrams
- can be complex, and require documentation

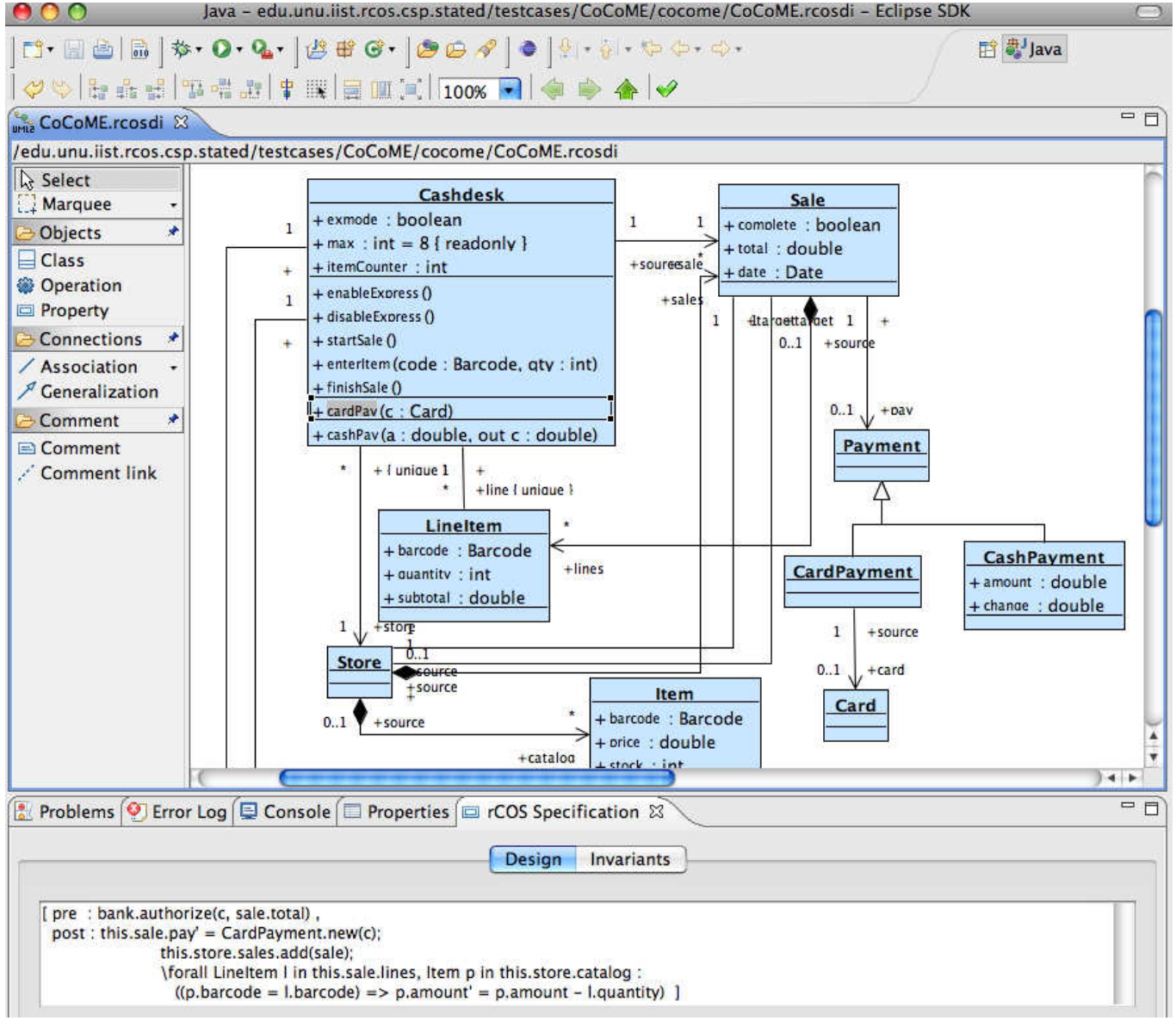
rCOS Data Model



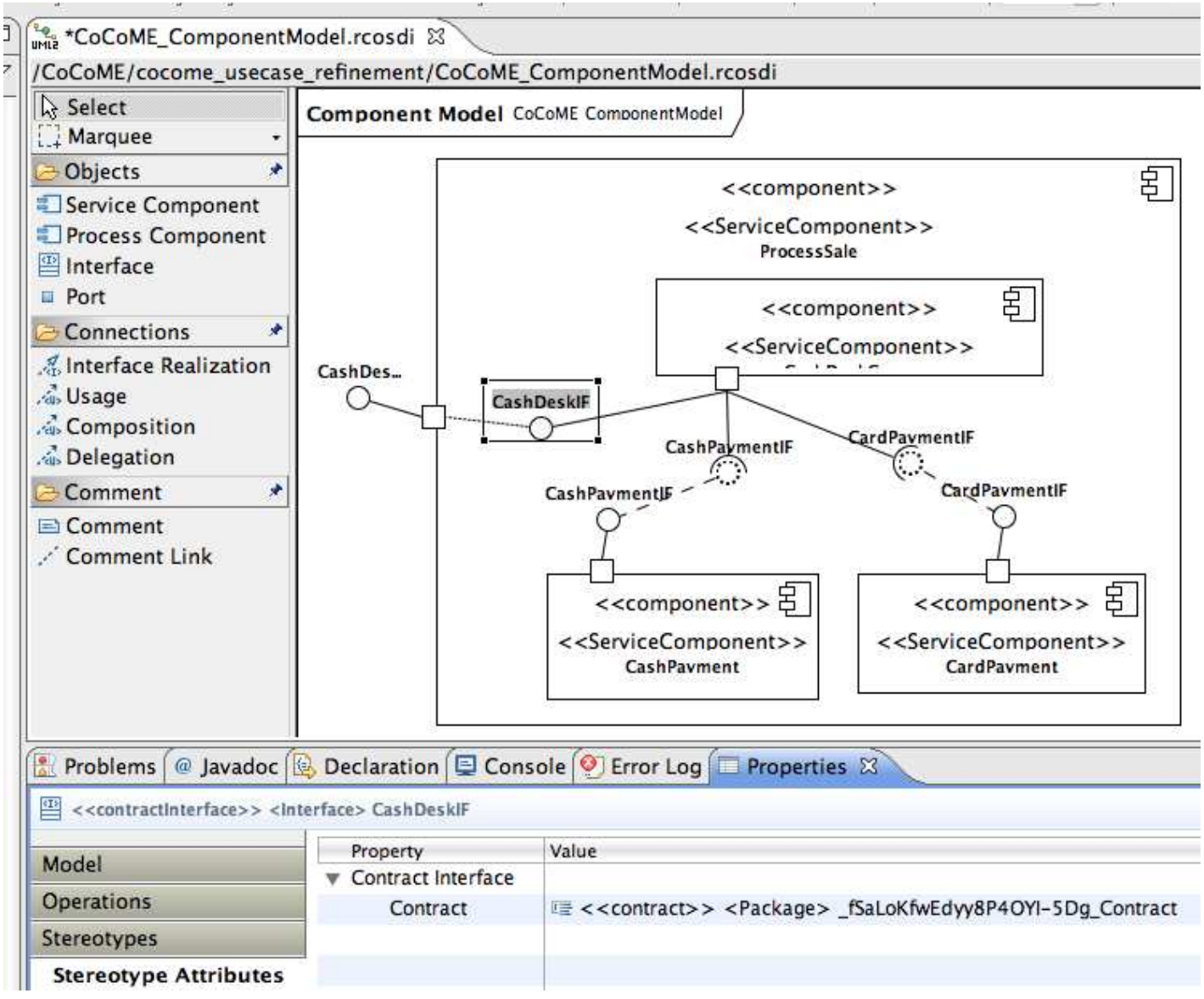
. . . as a Profile Diagram:



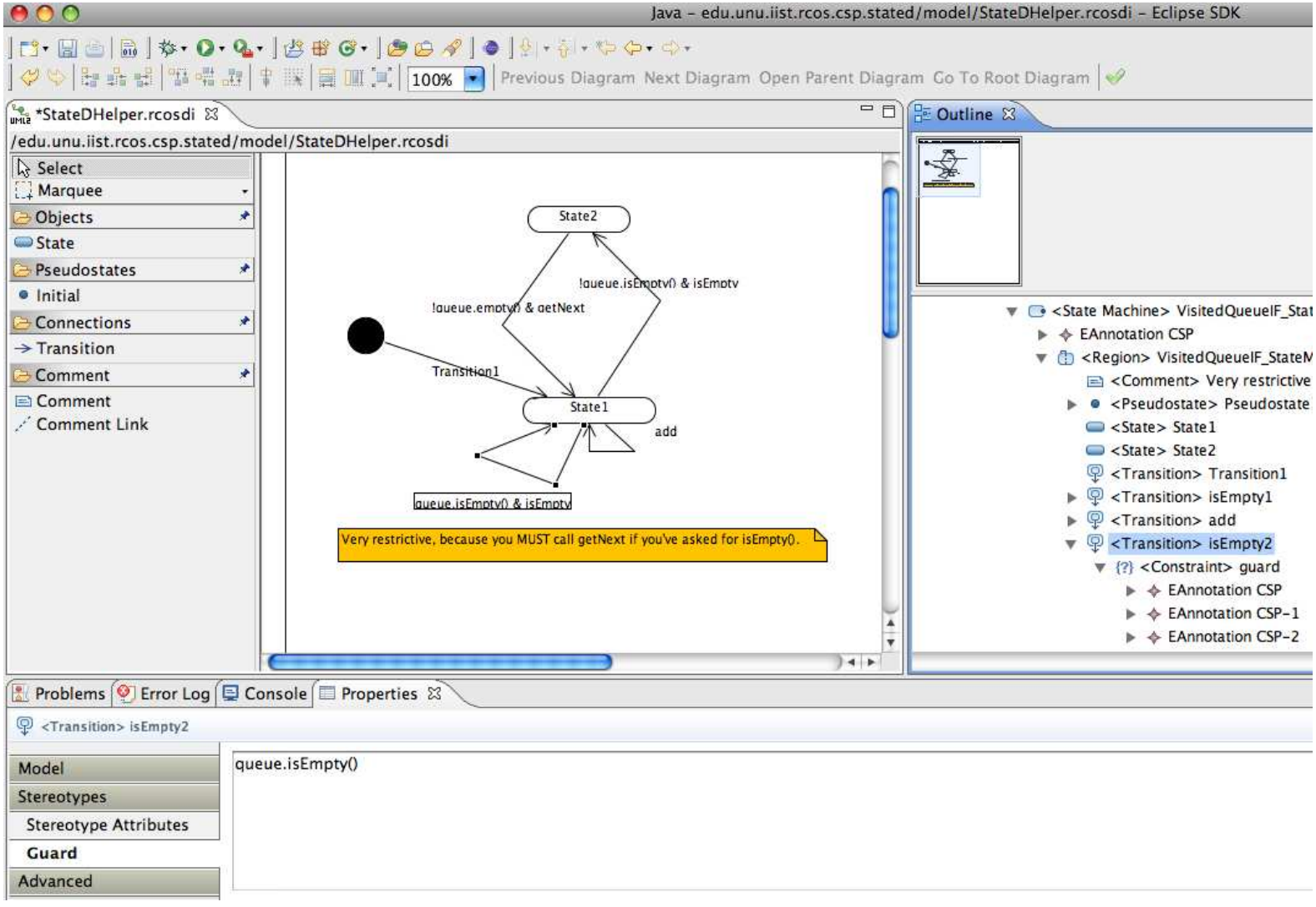
Class Diagrams



Component View



Modeling the Modeler



CSP Generation

- State machines with guarded transitions and link to operations (method bodies)
- Pre/post conditions have to be abstracted to CSP manually (data!)
- Model can contain various abstractions

- FDR2 must still be run interactively
- Component composition still to implement

Refinement

- Expert Pattern:
  refinement of navigation path to setters/getters
- modifies class structure and method bodies

- does not update diagrams
- versioning

More technical stuff

- Make use of EMF
  implicit: Eclipse UML explicit: class skeleton generation
- TOPCASED: easy start, but inherit all their “issues”
- UML support at the moment Eclipse specific
- Other input support: MagicDraw (NetBeans)
  (no diagramming info, only model)
- OCL preserved from modeling to generated code

Summary

- rCOS: Refinement of Component and Object Systems
- Formal method
- Accompanying methodology for component-based modeling
- Consistent multi-view modeling (use case, data, behaviour)
- Compatibility check of component composition through CSP
- UML modeling tool

Future Work

1. Theoretical Aspects: Extensions for Real-Time, QoS, Web-Services; Formal Syntax, Type system, Operations Semantics; Specification and Verification; Link to and Compare with JML, Alloy ....
2. Tool Development: Development of Correctness Preserving Transformations, Bring in Model Checking, Theorem Proving in rCOS, Model and Code Generation, Roles/Activities [ISoLA06]
3. Applications: Case Studies – POS, Production Cell, CORBA, Mondex, Drive By Wire, The space-flight file-store (POSIX)....