MODELS OF COMPONENTS

- Components
- Contracts
- Publications
- Compositions
- Refinement, correctness of component w.r.t. contract, faithfulness of publication

Components

A component is characterized by

- a provided interface *K.pIF*, possibly an internal interface *K.iIF* and a required interface *K.rIF*
  - an interface $I = (V, M)$ declares a set of typed fields and a set of methods
- a code *K.code*(m) for each $m() \in$ *K.pIF* $\cup$ *K.iIF*
- *K.rIF* contains the methods called in K, but defined inside K

Components in sequential programming

- each method is given in a sequential programming language
- it is executed in sequential environment in which it does not have control to the accesses to its services

component Buffer;
provided interface Buffer.IF;
seq(int) q = $\langle\,\rangle$;
public put(int in), get(;int out);
put(int in)$\{q := \langle in \rangle\}$; get(;out)$\{(out := head(q); q := \langle\,\rangle\}$

add empty(;bool a) $\{a := q = \langle\,\rangle\}$....

Same interface but different behavior

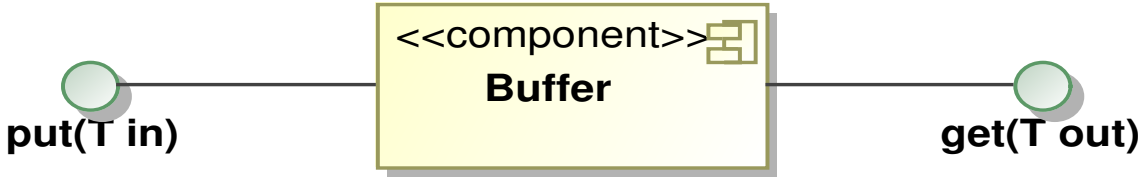
$$put(intin)\{q := \langle in \rangle \cdot q\};$$
$$get(;out)\{(out := head(q); q := tail(q)\}$$

We assume functions $head$ and $tail$ are defined, but

- we do not assume they are total functions, and therefore
- we do not claim the methods programmed correctly here

Reactive components

- used in a concurrent/distributed environment
- controls the accesses to its services/resources
- written in a concurrent langauge



component Buffer;
provided interface Buffer.IF;
seq(int) $q =< >$;
public put(int in), get(;int r);
put(in;)$\{q =< > \&q :=< in >\}$;
get(;r)$\{q \neq < > \&(r := hr(q); q :=< >)\}$

Semantics of components

- When K is a closed component, for $m \in K.\mathit{plF}$

$$[\![K.\mathit{code}(m)]\!]$$

So $[\![K]\!] : K.\mathit{plF}.\mathit{methods} \to \mathcal{D}$

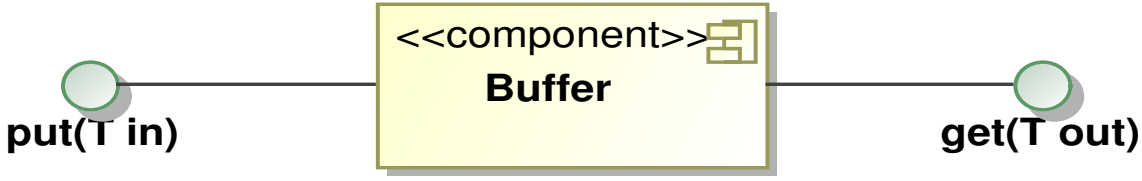
$$[\![K]\!](m) \mathrel{\widehat{=}} [\![K.\mathit{code}(m)]\!]$$

$$m(y)\{x := x + y\}, \quad [\![K.code(m)]\!] = true \vdash x' = x + y$$

- When K is open: $[\![K]\!] \mathrel{\widehat{=}} \lambda C_{\mathit{rlF}} \cdot \mathit{spc}.K$, s.t. for $C_{\mathit{rlF}} : K.\mathit{rlF}.\mathit{methods} \to \mathcal{D}$ and $m \in K.\mathit{plF}$,

Contracts: incremental modeling of interfaces

Black-box specification of what is needed for a component to be designed



- data functionality: $m()\{p \vdash R\}$
- reactivity: guards of methods $m()\{g\&(p \vdash R)\}$
- interaction protocol: traces $\subseteq \{put, get\}*$
- timing $[l, b]$ or $[a, b, l, u]$
- location, address, resources, and general QoS

Contract

$C = (I, s_0, F)$

- I is an interface
- s_0 is an initial condition
- F is a specification function $F(m)$ is a reactive design

In the top level contract specification, $F(m)$ is a guarded design

$g \& (p \vdash R)$

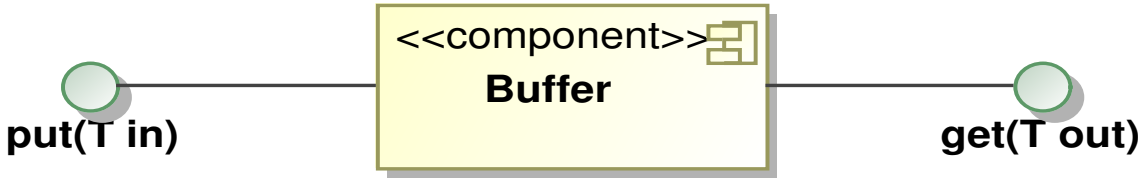
Dynamic behavior of a contract

Taking interface method invocations as the alphabet

- $\mathcal{D}(C) \ni tr \cdot s$ if $ok' = true$ for $s_0; tr_1; ...; tr_f$
- $\mathcal{F}(C) \ni \langle tr, X \rangle$ if
  - $tr \in \mathcal{D}(C)$,
  - $s_0; tr_1; \ldots; tr_f$ terminates in *wait* or falsifies the guards of all $m \in X$; notice tr cam be empty.
- Failure-divergence model: $(\mathcal{F}(C), \mathcal{D}(C))$
- Acceptable Traces:

$$\mathcal{T}(C) \quad \widehat{=} \quad \{tr \mid \exists X.(tr, X) \in \mathcal{F}(C)$$
$$\wedge(\forall s \preceq tr, (s, X) \in \mathcal{F}(C) \Rightarrow X \neq C.IF.methods)$$

Example



component Buffer;
provided interface Buffer.IF;
seq(int) $q =< >$;
public put(int in), get(;int out);
put(int in;)$\{q =< > \&q :=< in >\}$;
get(;int r)$\{q \neq< > \&(r := h(q); q :=< >)\}$

$[\![put(in;)]\!] = q =< > \&q' = q \cdot < in >$
$[\![get(;r)]\!] = q \neq< > \&q' =< > \wedge r' = h(q)$
$\mathcal{T}(Buffer) = (put()get())^* + (put()get())^* put()$

Failure-divergence refinement

$C_1 \sqsubseteq C_2$, if

$C_1.IF = C_2.IF$, $\mathcal{D}(C_1) \supseteq \mathcal{D}(C_2)$, and $\mathcal{F}(C_1) \supseteq \mathcal{F}(C_2)$

Refinement by Simulation

$C_1 \sqsubseteq C_2$ if

1. $C_2.init \Rightarrow (C_1.init)$
2. $C_1.guard(m) = C_2.guard(m)$ for all m
3. $C_1.spec(m) \sqsubseteq C_2.spec(m)$ for all m

Data refinement is defined

Components and their contracts

- For a component K and contract C_r, *spc*.$K(C_r)$ is a contract
- If $C_r^1 \sqsubseteq C_r^2$, *spc*.$K(C_r^1) \sqsubseteq$ *spc*.$K(C_r^2)$
- Failure-divergence refinement between components: $K_1 \sqsubseteq K_2$ if for all C_r, *spc*.$K_1(C_r) \sqsubseteq$ *spc*.$K_2(C_r)$
- K implements contract C, if $C \sqsubseteq spc.K(C_r)$ for some C_r
- Contract of open component K: $C = (P, R)$ s.t. $P \sqsubseteq spc.K(R)$

Theoretical work to underpin the models, their relations and manipulations

Publication

- A publication of an interface $P = (I, s_0, S, Prot)$
  - I is an interface
  - S is a data functionality specification: $S(m) = pre \vdash Post$
  - $Prot \subseteq (IF.methods)^*$ is a set of traces
- A publication P can be transformed to a contract $\mathcal{C}(P)$
- A faithful publication of a component K: $U = (G, A)$ s.t.
  - G is a publication of $K.pIF$ and A is a publication of $K.rIF$, and
  - $\mathcal{C}(G) \sqsubseteq spc.K(\mathcal{C}(A))$
- For a contract C, $\mathcal{P}(C) = (I, s_0, F^{-guard}, \mathcal{T})$ is a publication
- $(\mathcal{C}, \mathcal{P})$ is a Galois connection
- If (P, R) is a contract of K, $(\mathcal{P}(P), \mathcal{P}(R))$ is a faithful publication of K

Theorem

For a contract $C = (P, R)$ of a component K,

1. $(\mathcal{C}(P), \mathcal{C}(R))$ is a faithful publication
2. if $U_1 = (G_1, A_1)$ is a faithful publication of K, $U_2 = (G_2, A_2)$ is a faithful publication of K, provided
   - $G_1.IF = G_2.IF$, and $A_1.IF = A_2.IF$
   - $G_1.spec \sqsupseteq G_2.spec$, and $A_1.spec \sqsubseteq A_2.spec$
   - $G_1.prot \supseteq G_2.prot$, and $A_1.prot \subseteq A_2.prot$

Buffer Example

$spec(put(in;)) = q' = q \cdot < in >$
$spec(get(;r)) = q' =< > \wedge r' = h(q)$
$Prot \subseteq (put()get())^* + (put()get())^* put()$

Compositions

1. hiding provided services
2. parallel composition (disjoint union)
3. renaming interface methods
4. plugging
5. internalization of provided services

black-box composition for contracts and publications

Compositions as connectors and coordinators

- simple connectors component implement 1), 2), 3) and 4)
- process components implement internalization
- general synchronization/coordination defined by these operators
- compositions preserve refinement
- Data refinement is implemented by data converters

Example

Define component C_1:

```
component C1 {
 provided interface C1.pIF {
    public put(int in); public get1(;int out);
    class B implements C1IF {
       protected Seq(int) x1;
       public put(int in){x1:=<in> ◁ x1=<> ◁ put1(head(x1))};x1:=<>};
       public get1(,int out){x1!=<>& out:=head(x1);x1=<>};
    }
 required interface R1IF { public put1(int in) }
```

Define C2

```
component C2 {
 provided interface C2IF {
    public put1(int in); public get(;int out);
    class B2 implements C2IF {
       protected Seq(int) x2;
       public put1(int in){x2=<>& x2:=<in>};
       public get1(;int out){get1(in)◁ x2=<> ▷ out:=head(x2);x2:=<>};
    }
 required interface R2IF { public gwt1(;int out)
}
component C= C1 >> C2
```

Exercise: Calculate the contract of C.

Examples/Exercises

1. One place buffers: B_1 provides $put()$, $get()$ and requires $foward()$; B_2 provides $put()$, $get()$ and requires $ask()$.

$$B_1[get/ask] >> B_2[put/forward]$$

2. Given an one place buffer B that provides $put()$, $get()$, and *isEmpty*$?()$, build a two place buffer B_2 by using two copies of B_1 and a connector.
3. Given the one place buffer B that provides $put()$ and $get()$, build a two place buffer with a connector and an internalization process.

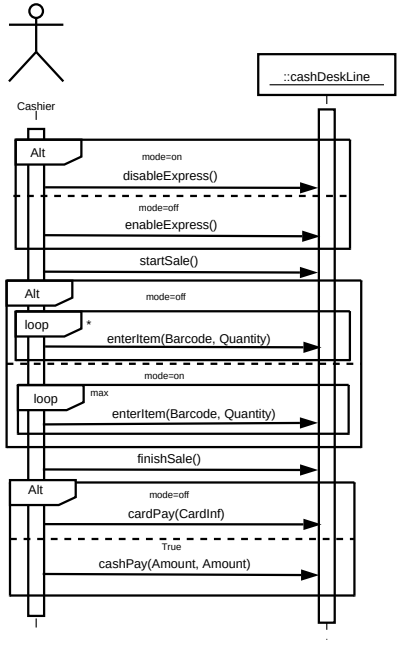
Development process

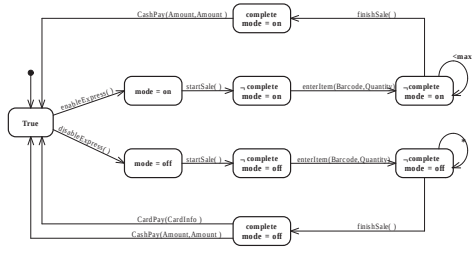
Use case driven and incremental

- RA: use case specified as component contract
- OO design of component: design patterns (PIM)
- Transform OO design model to component based design model (PIM)
- PIM to PSM transformation
- PSM to code transformation
- Verification and analysis before and after transformations

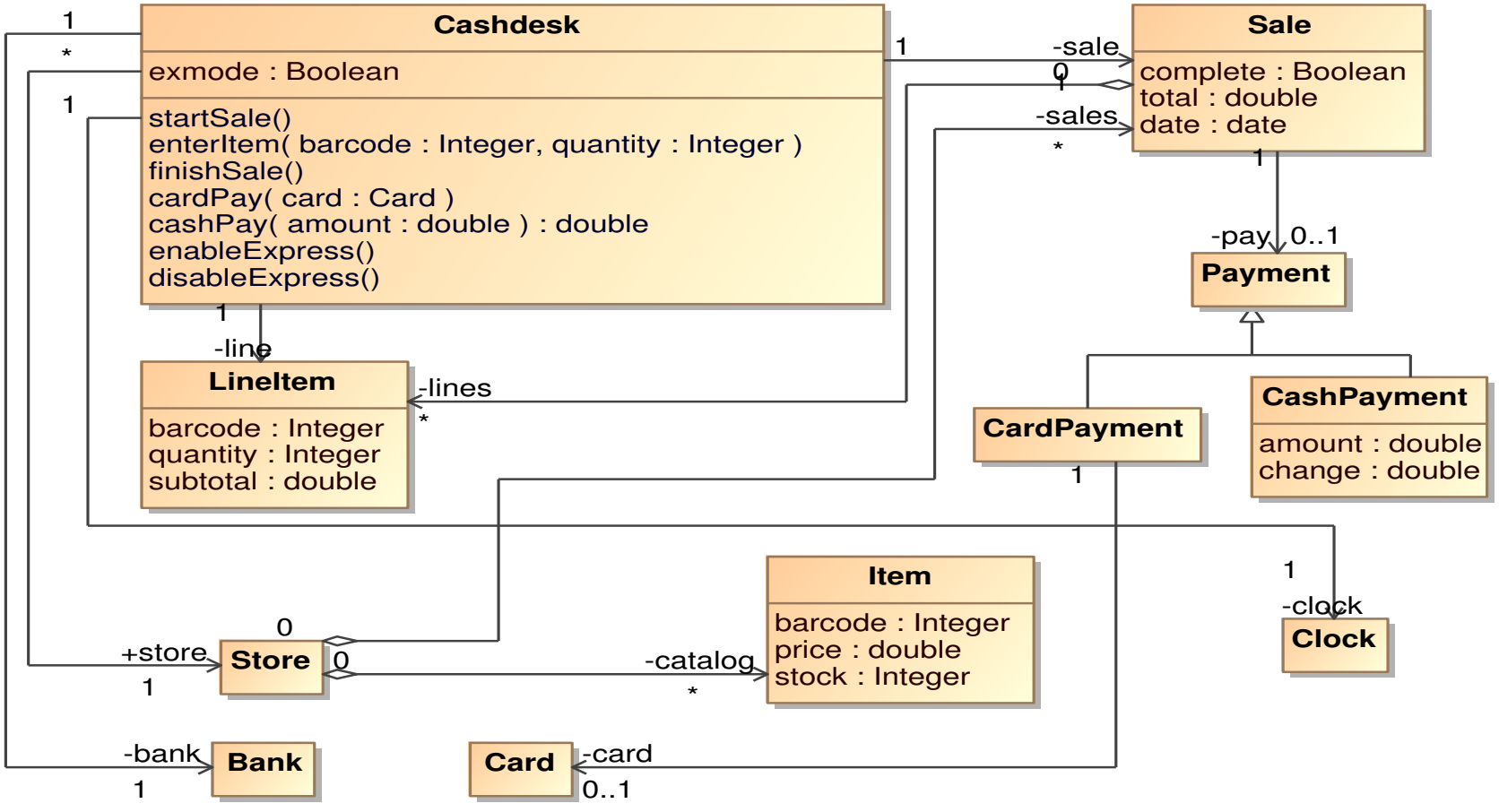
RA: Use-Case as contracts of component

- sequence diagram (actor's behavior)
- state diagram (component's reactive behavior)
- class diagrams (data and object structure)
- CSP processes *Actor* || *Comp*
- functionality actions
  $m()\{f : p \vdash R\}$
- *Ctr* $= (I, Spec, Prot)$
- UML profile for rCOS

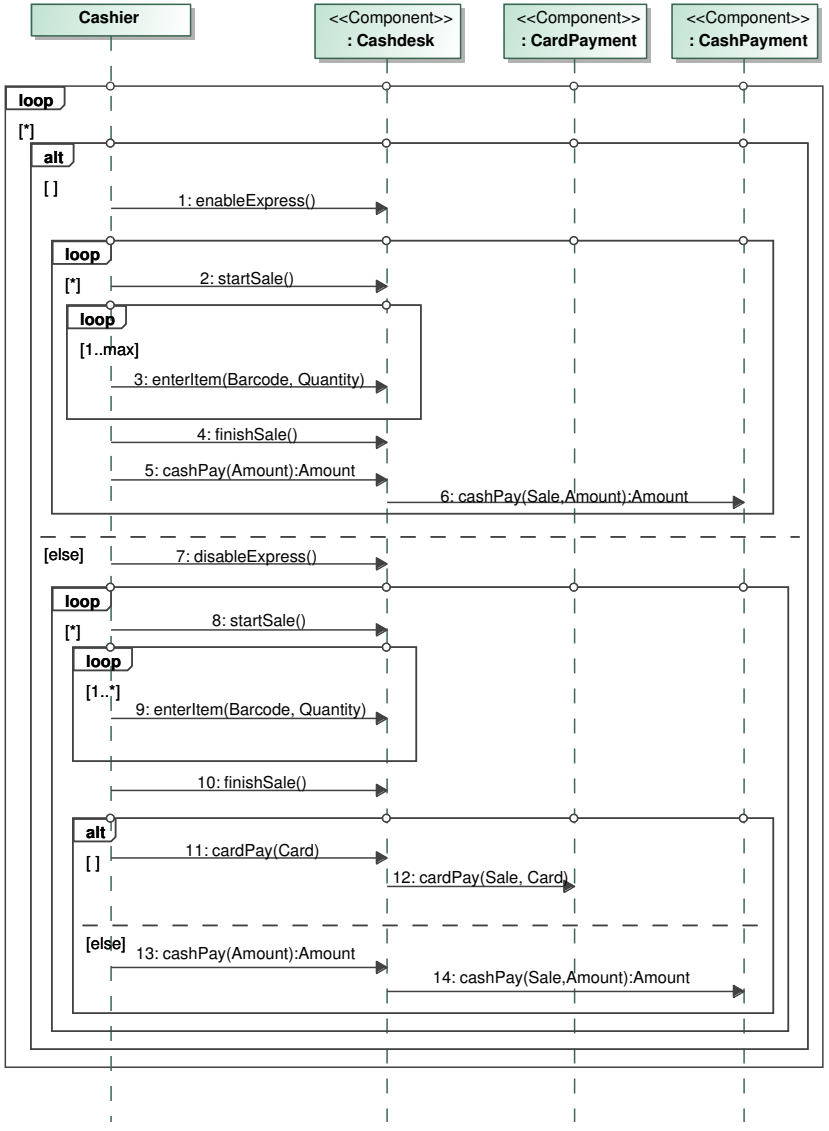


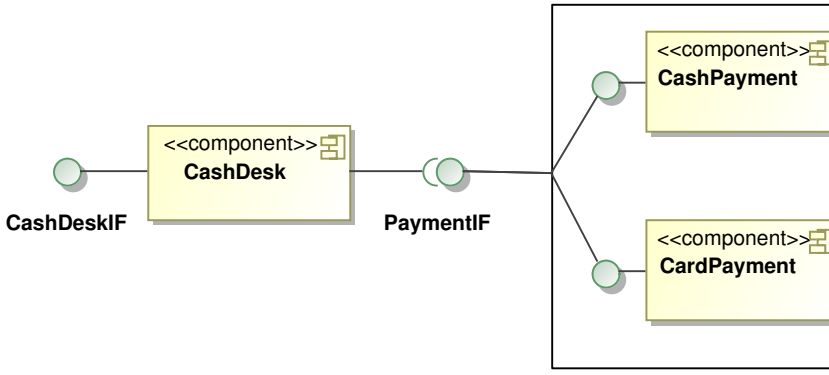


Class diagram



Use cases = composition of components





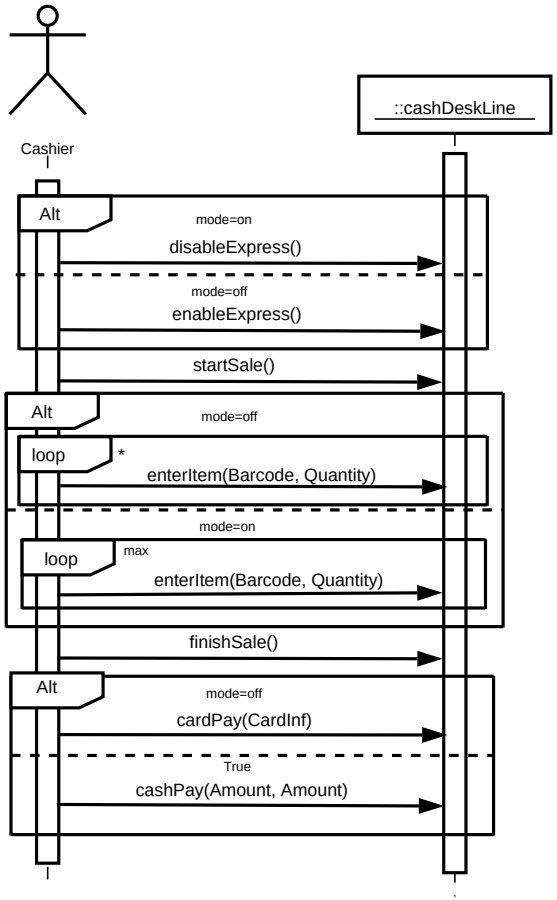
CashDesk≪(CashPay||CardPa

Validation of consistency

Verification of properties

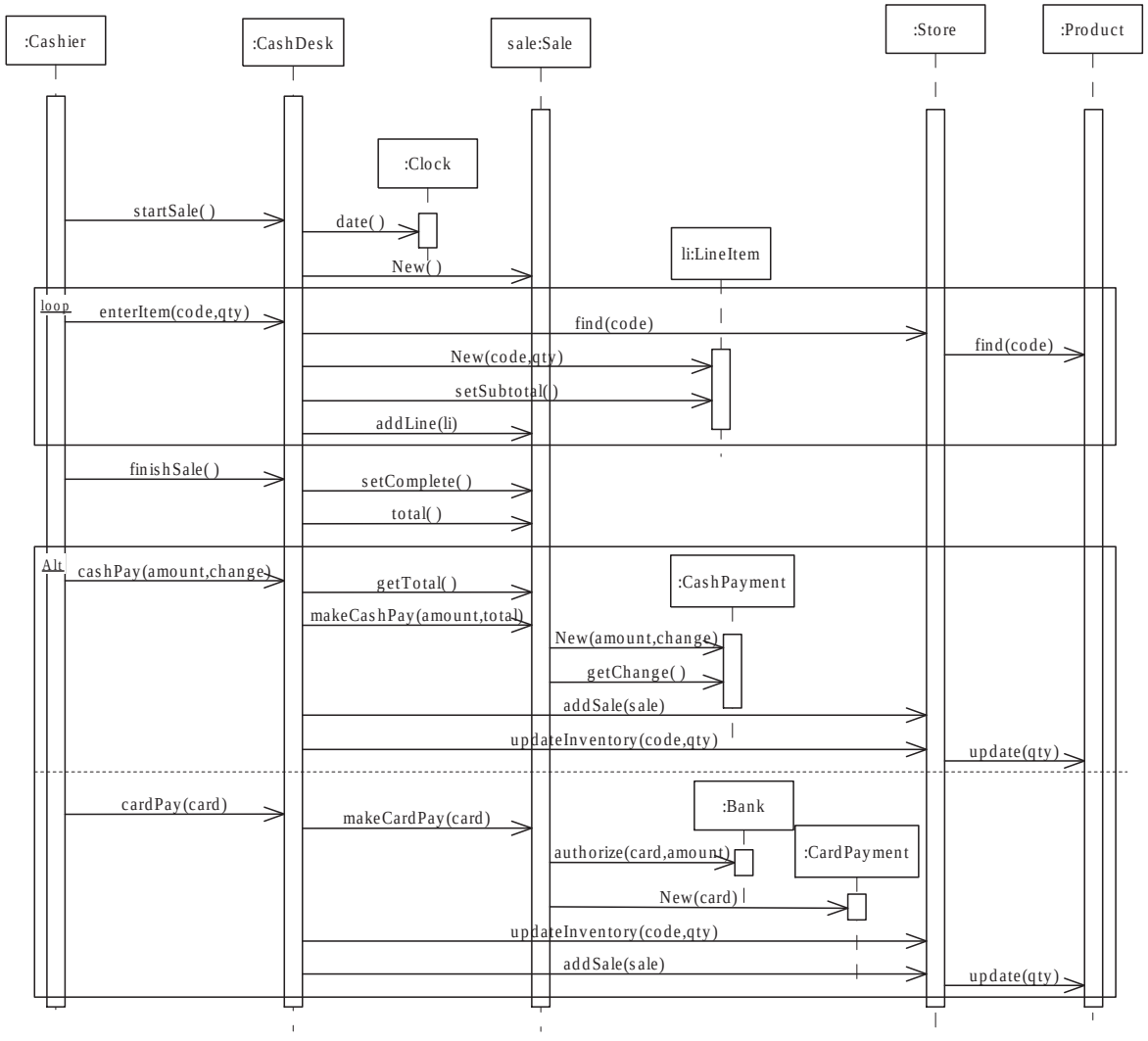
Design: OO refinement & Expert Patterns

Start with Contract of Use Case

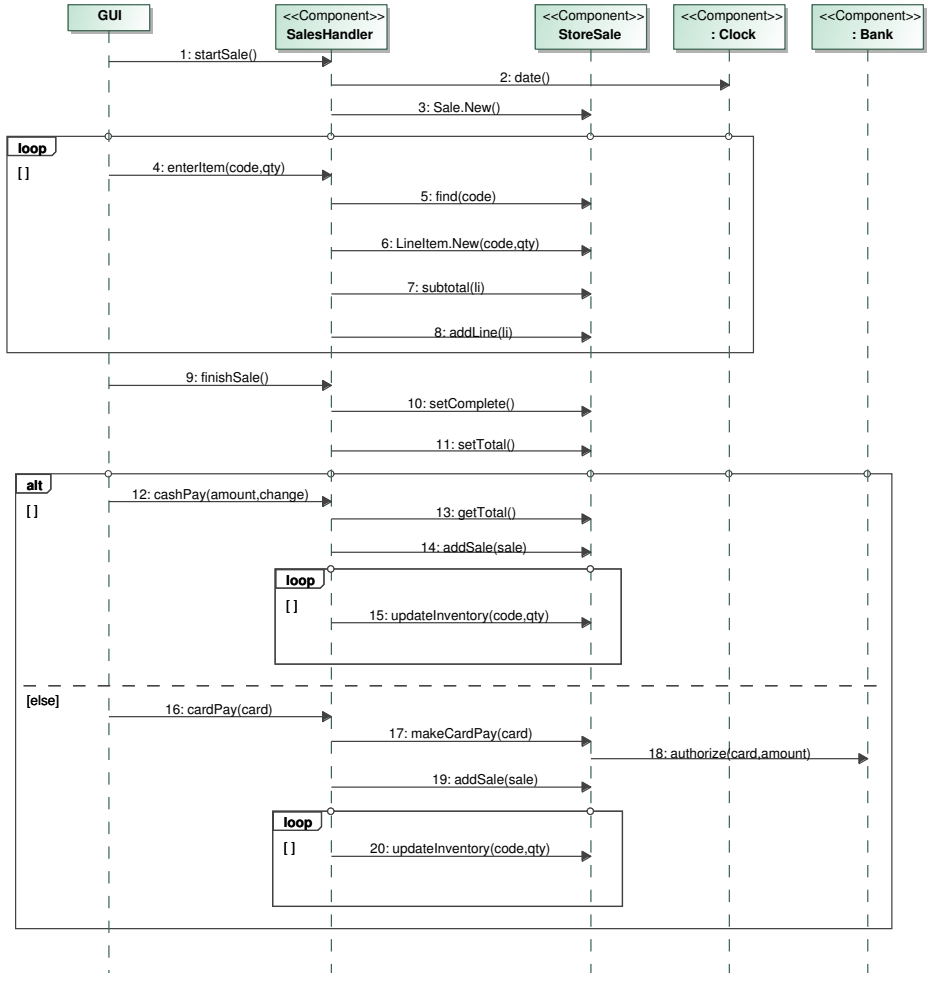


Apply automated design patterns

Produce model of OO design

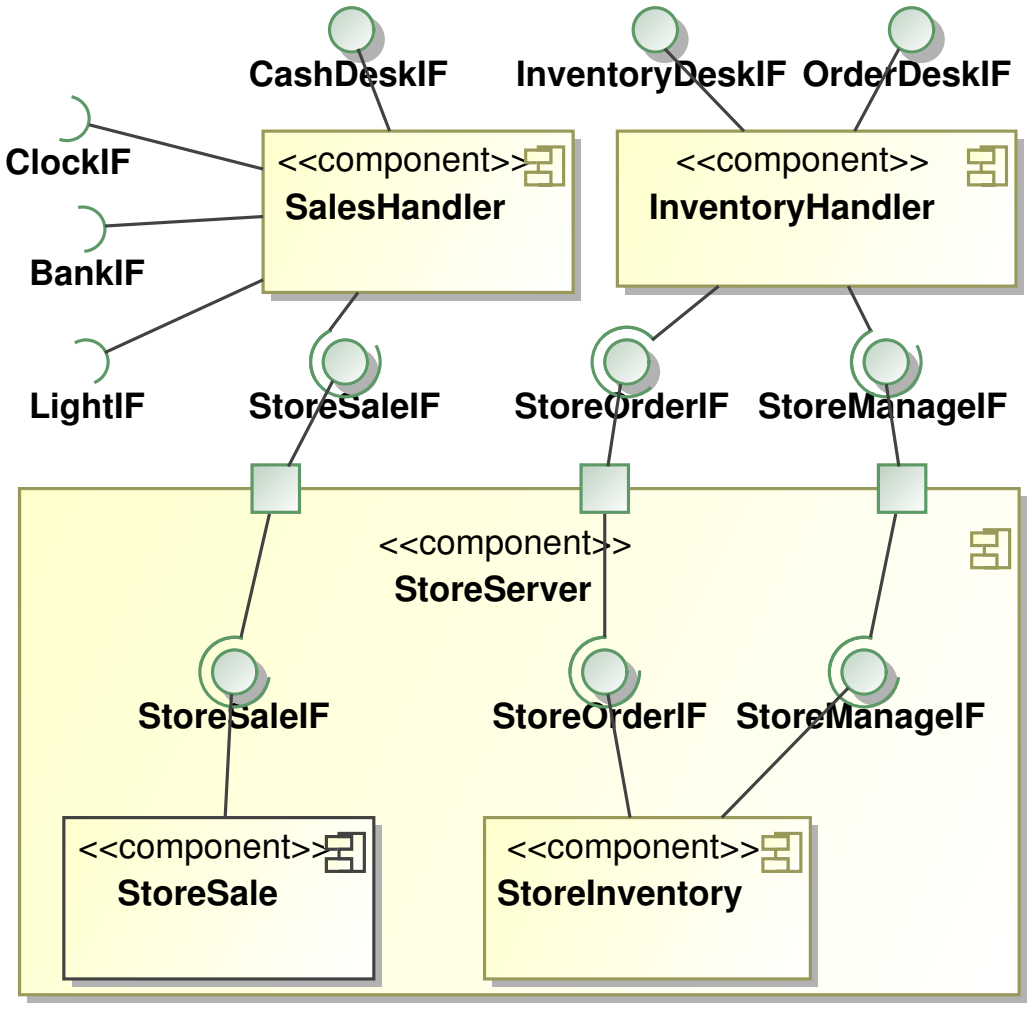


Making OO design component-based

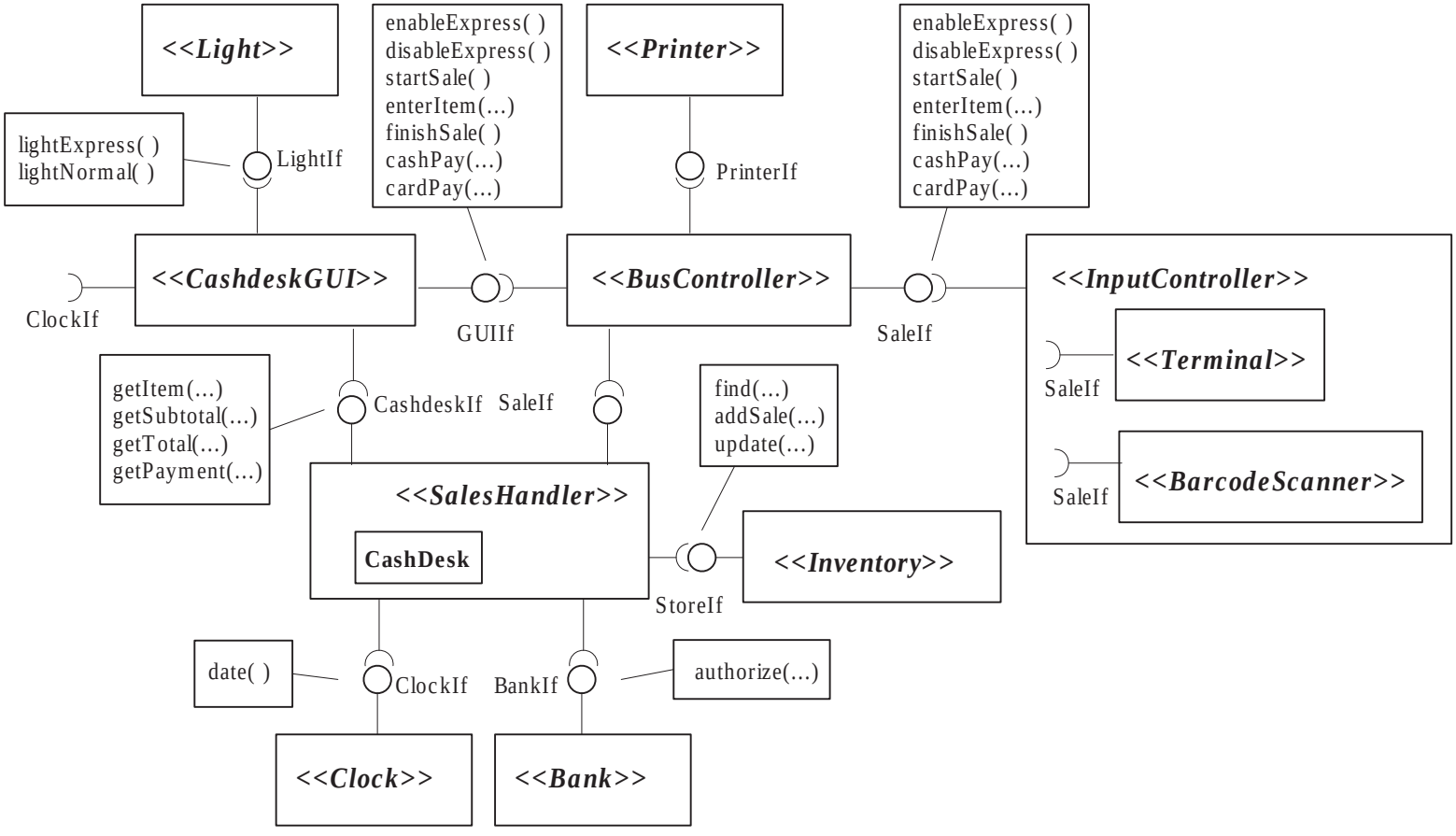


$ProcessSale \mathrel{\widehat{=}} SalesHandler \ll (StoreSale \parallel Clock \parallel Light)$

Logical CBA


CashDeskIF
InventoryDeskIF
OrderDeskIF
ClockIF
<<component>>
SalesHandler
<<component>>
InventoryHandler
BankIF
LightIF
StoreSaleIF
StoreOrderIF
StoreManageIF
<<component>>
StoreServer
StoreSaleIF
StoreOrderIF
StoreManageIF
<<component>>
StoreSale
<<component>>
StoreInventory

Integration



- GUI objects, controllers of hardware and application components interact
- Add middlewares and change OO interface to concrete interaction mechanisms

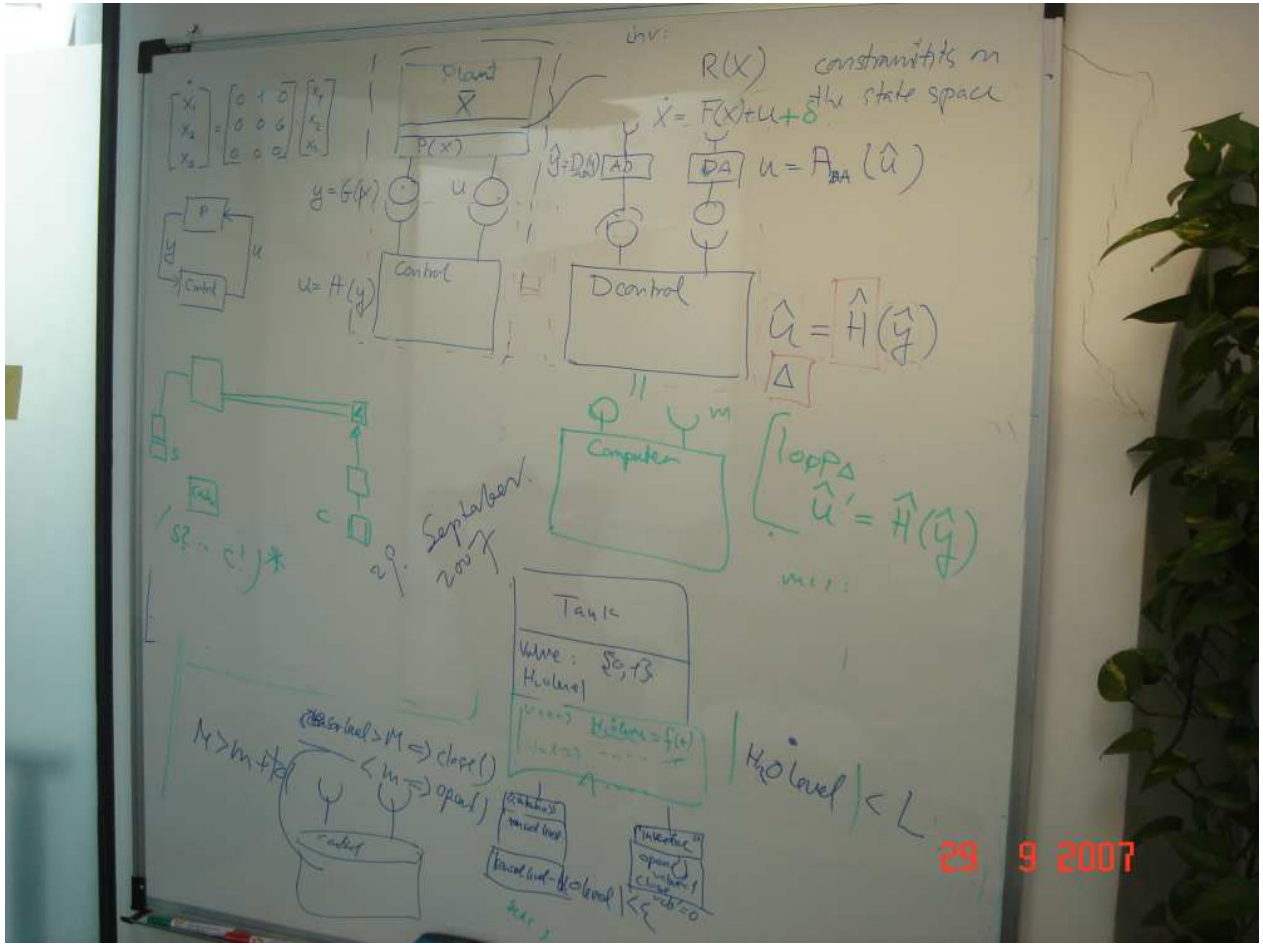
Conclusion

- Software complexity and correctness are handled by
  - factoring a model of a component into models of different views of the data computation, interaction protocol, the reactive behavior,
  - techniques for model analysis, verification and correct by construction through model transformations,
  - tool support for model construction, model verification and model transformations.
- Ambition: developing and teaching a coherent and comprehensive methodology that integrates theories and techniques of verification into development.

To do

- tool implementation and integration
- Theory
  1. graph-based operational semantics for OOP
  2. graph-based Hoare-Logic for OOP
  3. semantics and analysis of multi-thread programs
  4. further work on CB refinement
  5. real-time rCOS
  6. service-oriented programming
  7. fault-tolerance
  8. applications

Systems of Hybrid Components??



References: http://rcos.iist.unu.edu

- For OO refinement and Design Patterns: [TCS365(1)-06, JFAC21(1)-09, ISoLA08]
- For model of components, contracts, composition and refinement: [FSEN07,UTP08]
- For development process and case study: [CoCoME, LNCS08,JSCP 74(4) 2009]
- For Model consistency and integration: [ICFEM03,SOFSEM08]
- Automata Model of rCOS-ADL [UNU-IIST Tech. Report 438]
- Tool support: http://rcos.iist.unu.edu