



Component-Based Development With Catalysis

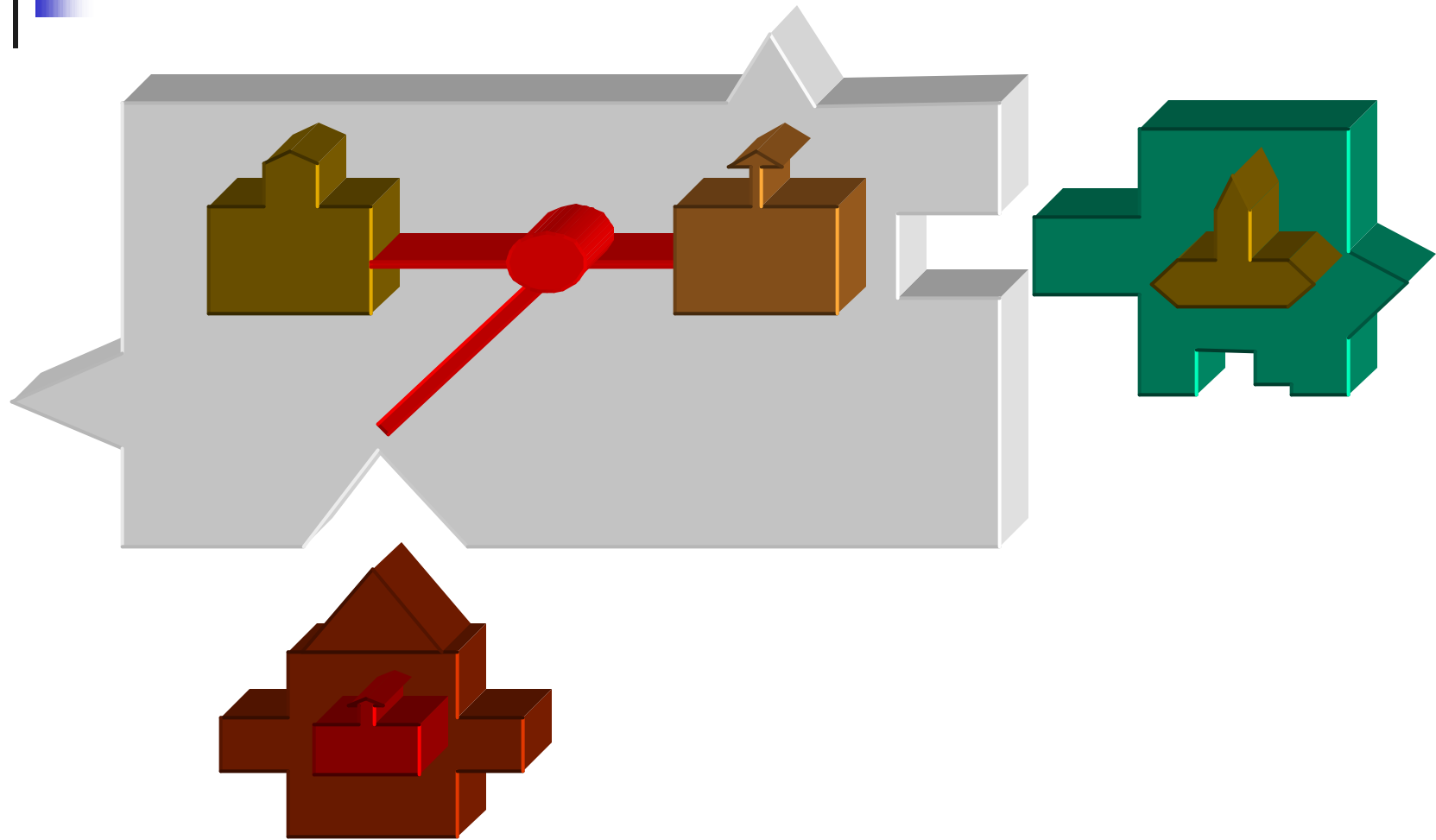
Daryl Winters
Component Architects
July 19, 2001



Why Components?

- A practical way to organize and package systems
- Develop, market, and maintain on a component basis
- Support capabilities that are impractical for “small” objects
 - Interfaces through different programming languages
 - Components interact transparently across networks
 - More cost-effective to maintain since they do more than “small” objects, and less than “monolithic” programs
- Each component could itself be a candidate “product”
- Component partitioning enables parallel development

The Holy Grail of Software



- Assemble with components at **any** level!



Outline

- **Method Overview**
- Types
- Collaborations
- Refinements
- Frameworks
- Process

What is Catalysis™?

Precise models and systematic process

UML partner, OMG standards, TI/MS standards

Dynamic Architectures

A next-generation standards-aligned method
For open distributed object systems
from components and frameworks
that reflect and support an adaptive enterprise

*From business
to code*

*Compose pre-built interfaces,
models, specs, implementations...*

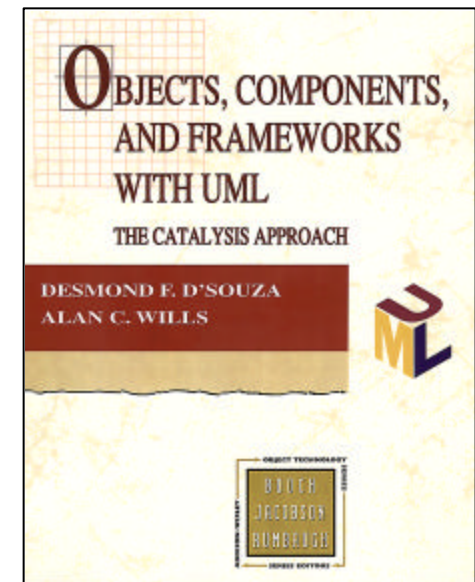
...all built for extensibility

Catalysis has been in development and use since 1992.

Authors: Desmond D'Souza and Alan Wills

Text: Addison Wesley, "*Objects, Components, and Frameworks with UML: The Catalysis Approach*".

OMG Model Driven Architecture uses Catalysis.



Three Modeling Scopes or Levels

Level/Scope

Goal

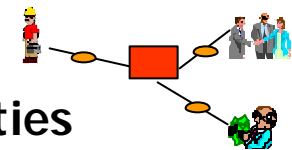
Domain/Business

Identify Problem: "Outside"
establish problem domain terminology
understand business process, roles, collaborations
build *as-is* and *to-be* models



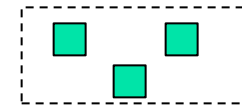
Component Spec

Specify Solution: "Boundary"
scope and define component **responsibilities**
define component/system **interface**
specify desired component operations

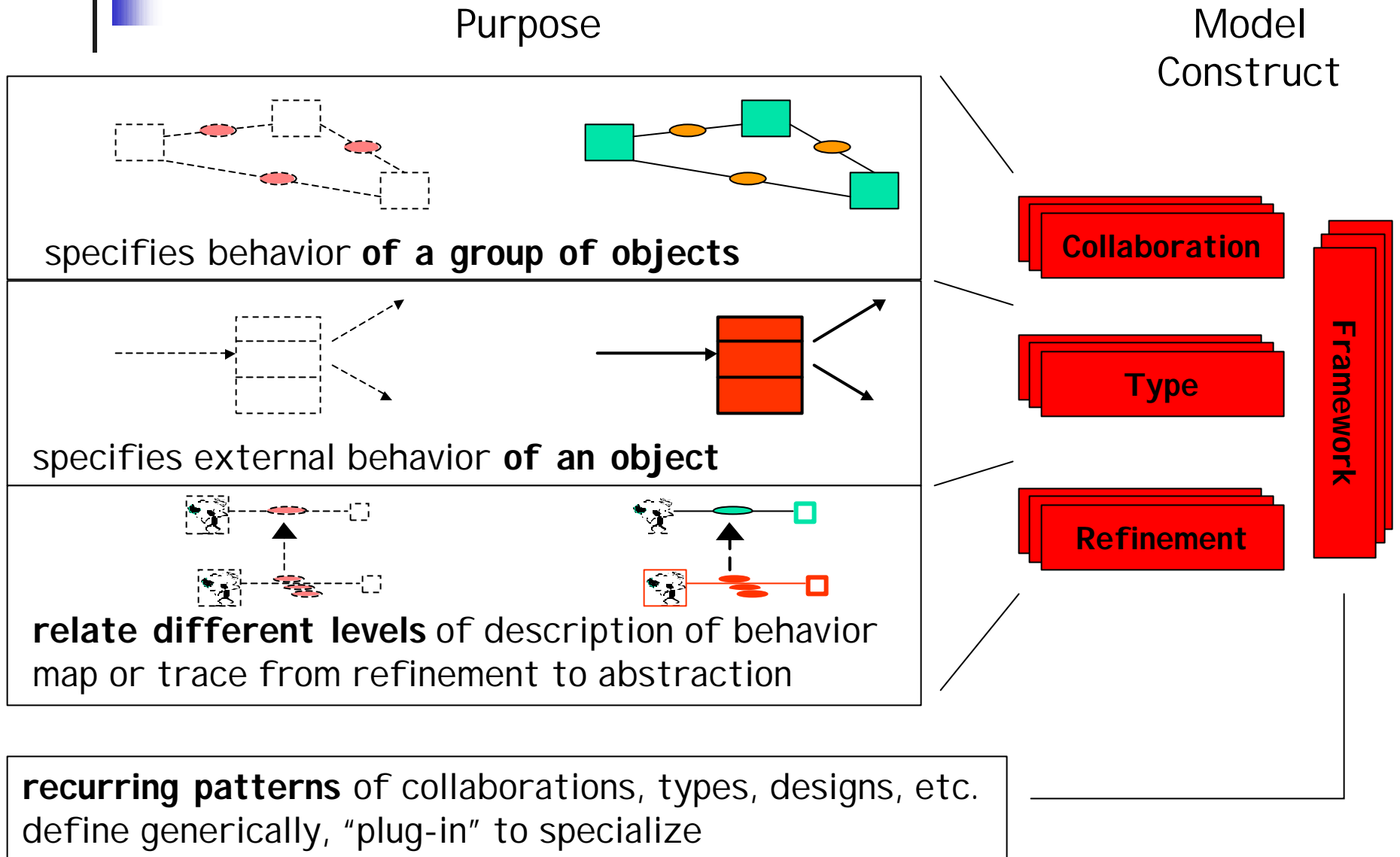


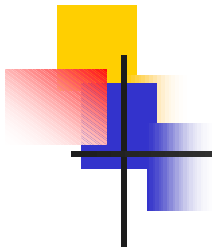
Internal Design

Implement the Spec: "Inside"
define **internal architecture**
define internal components and collaborations
design each component's internals



Three Modeling Constructs





Three Principles

Principle

Intent

Abstraction

focus on **essential aspects**, deferring others
uncluttered description of requirements and architecture

Precision

expose gaps and inconsistencies early
make **abstract models accurate**, not fuzzy

Pluggable Parts

all work done by **adapting and composing parts**
models, architectures, and designs are assets

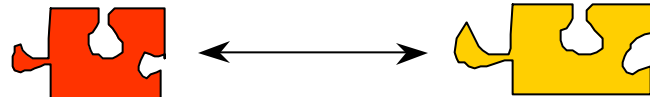


Outline

- Method Overview
- **Type Models**
- Collaboration Models
- Refinement Models
- Framework Models
- Process

Components

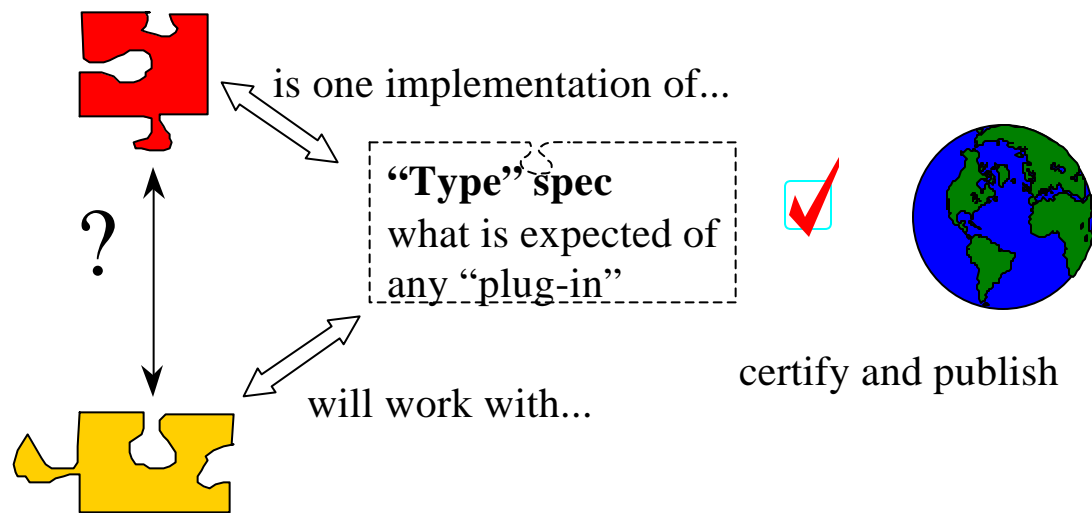
- There are many definitions of *component*
 - A user-interface widget that can be assembled on an interface builder
 - An executable module with a specified interface
 - A large-grained object with encapsulated persistent state and an interface
- All have the goal of *software assembly*



- An independently deliverable unit that encapsulates services behind a published interface and that can be composed with other components

Components Depend On Interfaces

- To build systems by “plugging” together components



- Two choices: “plug-n-pray”, or use better specifications that are:
 - **abstract**: apply to any implementation
 - **precise**: accurately cover all necessary expectations

Component Interfaces

Thingami

```
frob (Thing, int)  
Thing fritz(int)
```

Editor

```
spellCheck()  
layout()  
addElement(...)  
delElement(...)
```

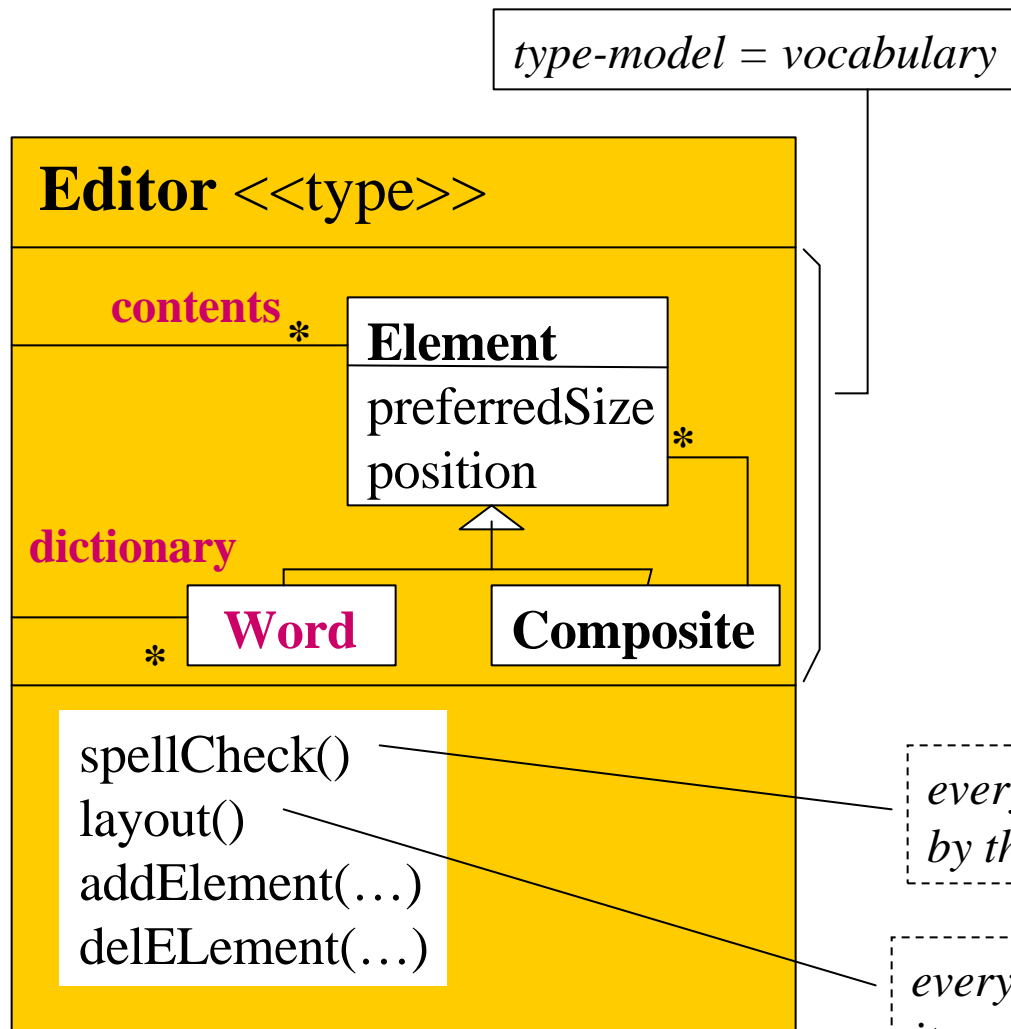


NuclearReactorCore

```
add (ControlRod, int)  
ControlRod remove(int)
```

- Signatures are not enough to define widely-used components

Model-Based Type Specification



- Type described by list of operations
- All operations specified in terms of a **type model**
- The Type Model defines **specification terms**

*every **word** in **contents** is correct by the **dictionary***

*every **element** has been **positioned** so that its **preferred size** can be accommodated*

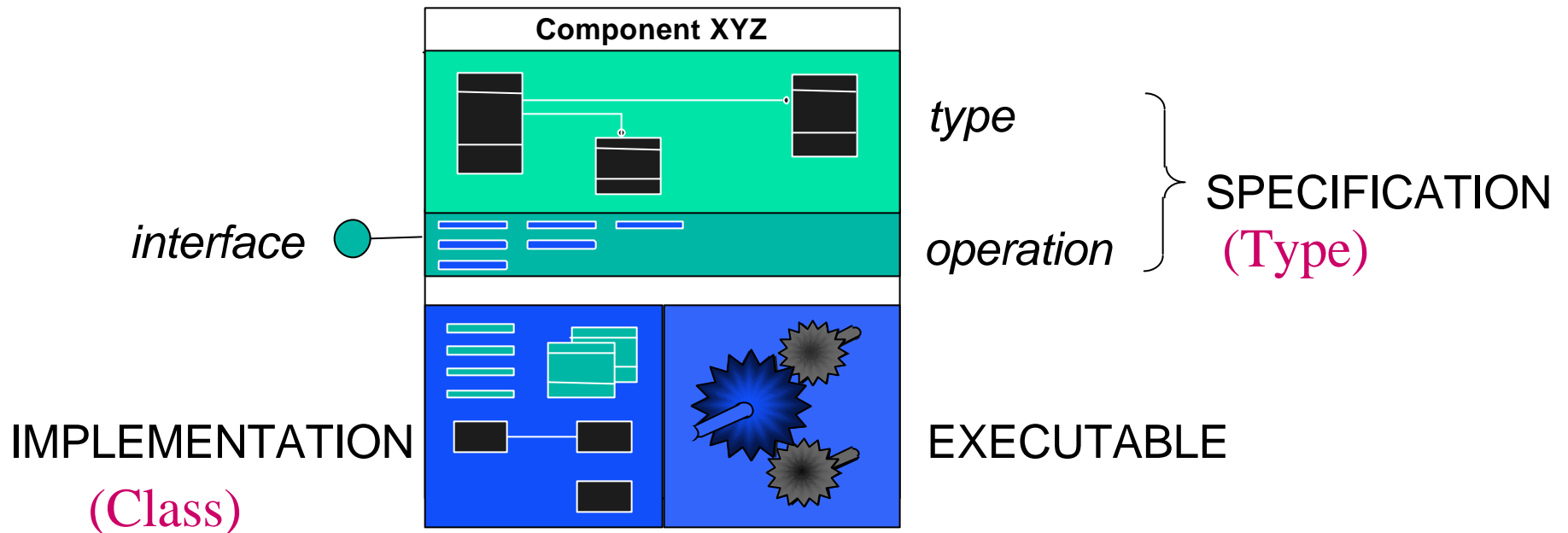


Formal Operation Specifications

- Editor:: spellCheck ()
- **post**
- -- every word in contents
- contents->**forAll** (w: Word |
- -- has a matching entry in the dictionary
- dictionary->**exist** (dw: Word | dw.matches(w)))

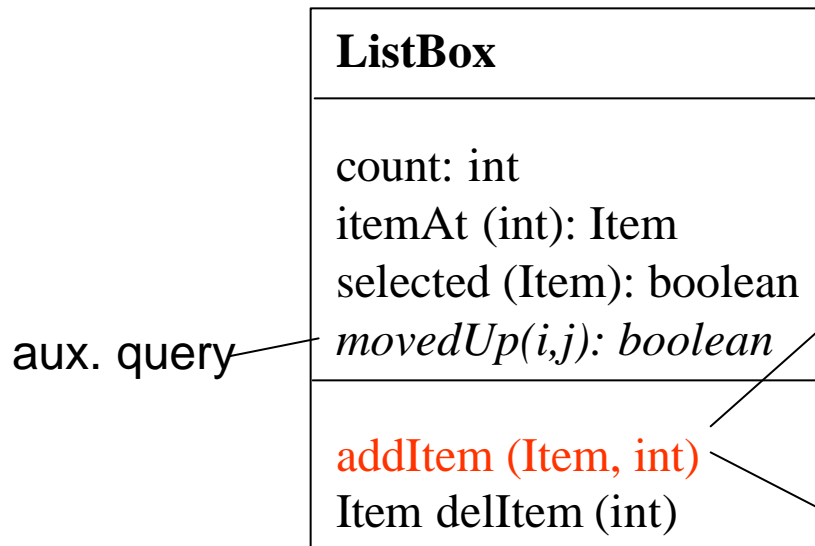
- Operation specification can formalized
 - *OCL* (Object Constraint Language) used as needed
 - Checked, used for refinements, testing, change propagation, ...
- Enables checkable/testable design-by-contract

Component Models



- Every component description has at least 2 parts:
 - The specification of the interfaces via types.
 - The implementation (designed classes, modules).

Behaviors Specified Using Models



Behavior "contract"
you can rely on *post-conditions* provided *pre-conditions* are met

```
addItem (Item item, int pos)
pre (0 < pos <= count+1)
post item = itemAt(pos)
      and count = count@pre+1
      and movedUp(pos, count@pre)
      and selected(item)
```

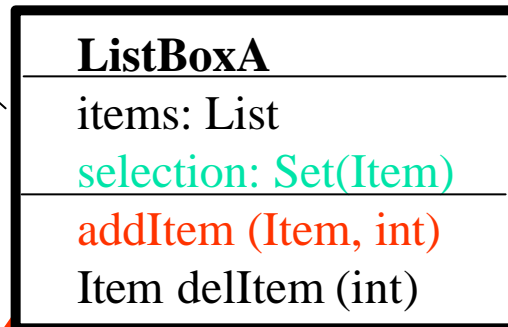
prior value

```
interface ListBox {
  model { .... }
  void addItem (Item item, int pos)
  pre 0 < pos and pos <= count+1 // provided position is in range
  post item = itemAt(pos) and selected(item) // inserted and selected
      and count = count@pre + 1 // count increased from before
      and movedUp (pos, count@pre) // items moved up
```

Valid Implementations of a Type

- Any valid implementation class
 - Selects data members
 - Implements member functions
 - Can implement (*retrieve*) the model queries
 - Guarantees specified behavior

class



```
addItem (Item item, int pos) {  
    items.insertAt(item,pos);  
    selection.add(item);  
}
```

<<implements>>

awt.ListBox <<type>>

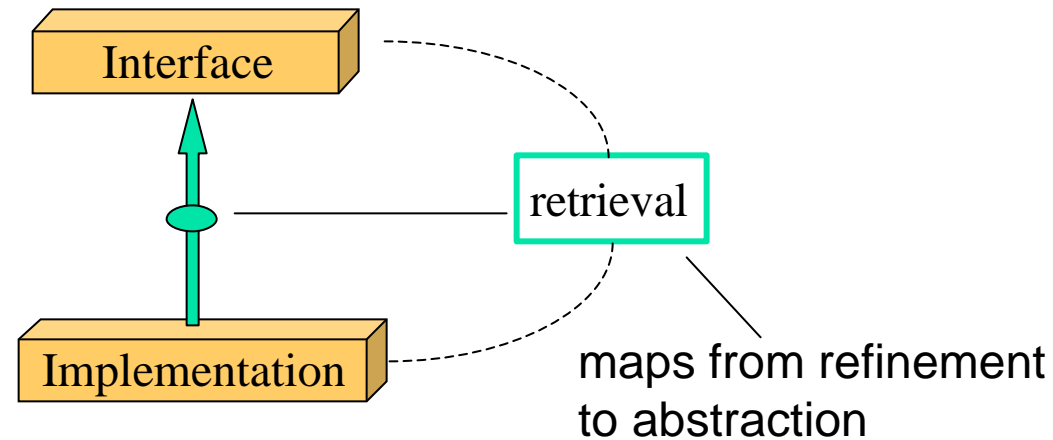
count: int
itemAt (int): Item
selected (Item): boolean
movedUp(i,j): boolean

addItem (Item, int)
Item delItem (int)

retrieval:

```
boolean selected(i) { return (selection.has(i));}  
int count() { return (items.length()); } ....  
Item itemAt(pos) { return items[pos]; }
```

Conformance and Retrieval



- A class implements an interface
 - The interface defines its own model and behavior specification
 - The class selects its own data and code implementation
- The class is a *refinement* of the type I.e. *conforms* to it
 - It claims to meet the behavior guarantees of the type for any client
 - A *retrieval* (informal or formal) can support the claim
 - Implemented abstract queries (formal) can be used for **testing**



Conformance/Refinement

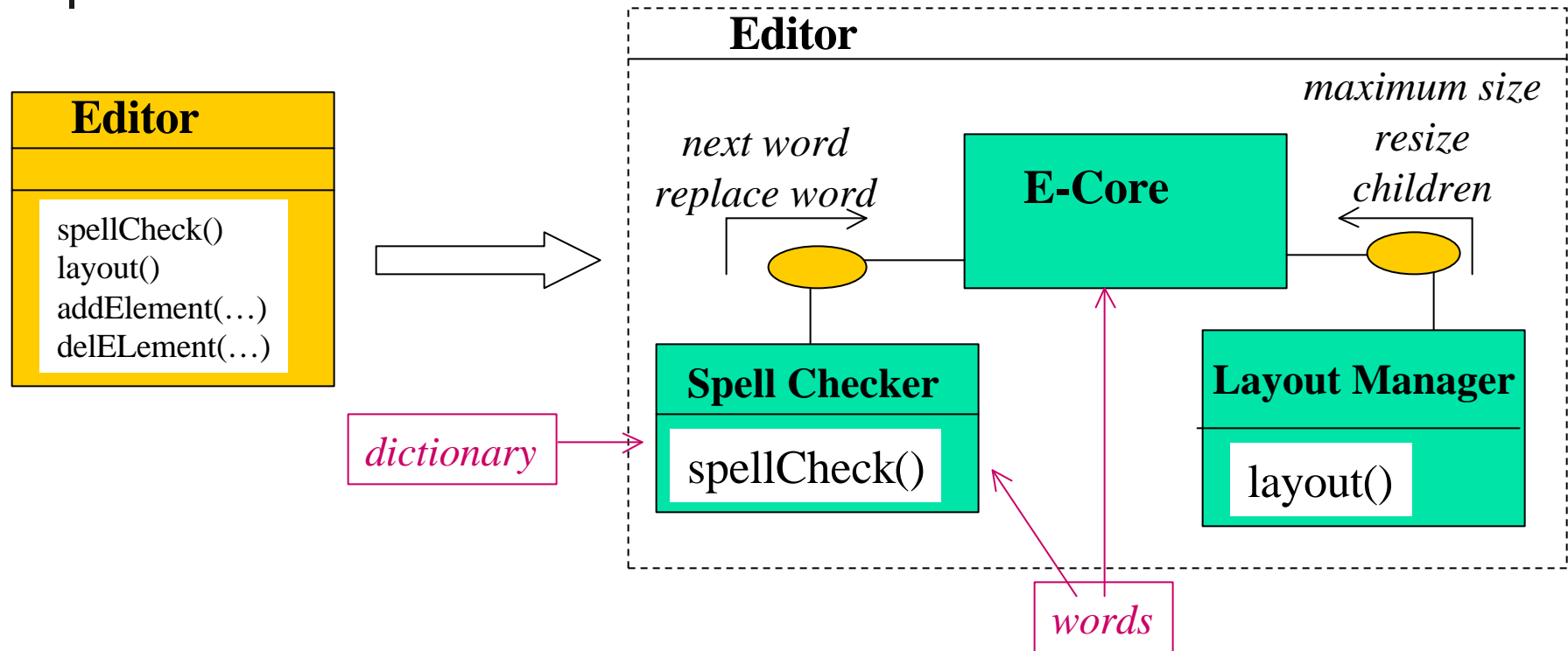
- The notion of conformance and retrieval is very useful
 - It permits flexible *mapping* between from refinement to abstraction
 - It solves a very real problem:
 - *"I have just made some change to my code. Do I have to update my design models? Do I have to update my analysis models?"*
- Pick abstract model to conveniently express client spec
 - Implementation model must have correct *mapping* to the abstract
 - Encapsulates implementation without hiding specified behavior
- Even more powerful with *temporal refinement*
 - The abstract level describes an abstract action
 - The concrete level details an interaction sequence
 - The retrieval establishes the mapping between the two



Outline

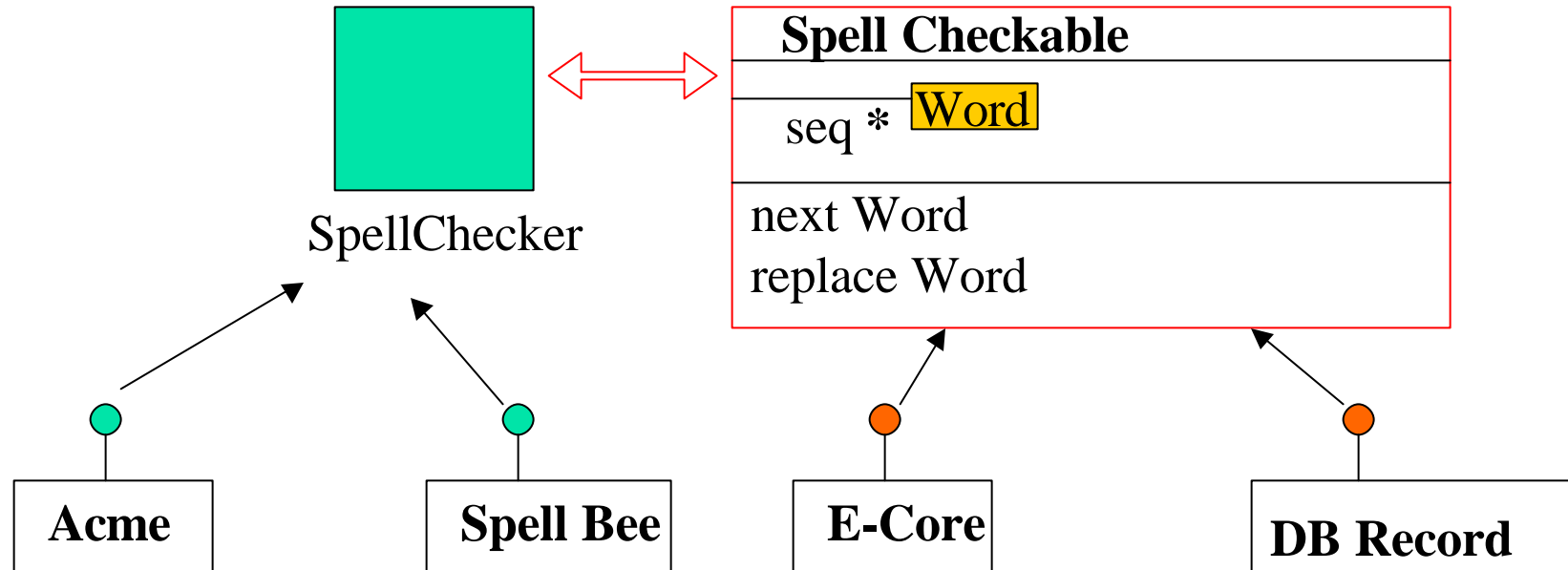
- Method Overview
- Type Models
- **Collaboration Models**
- Refinement Models
- Framework Models
- Process

Component-Based Design



- Large component is a **type** for its external clients
- Implement it as **collaboration** of other components
- Specify these other components as **types**
- The child component models must map to the original model

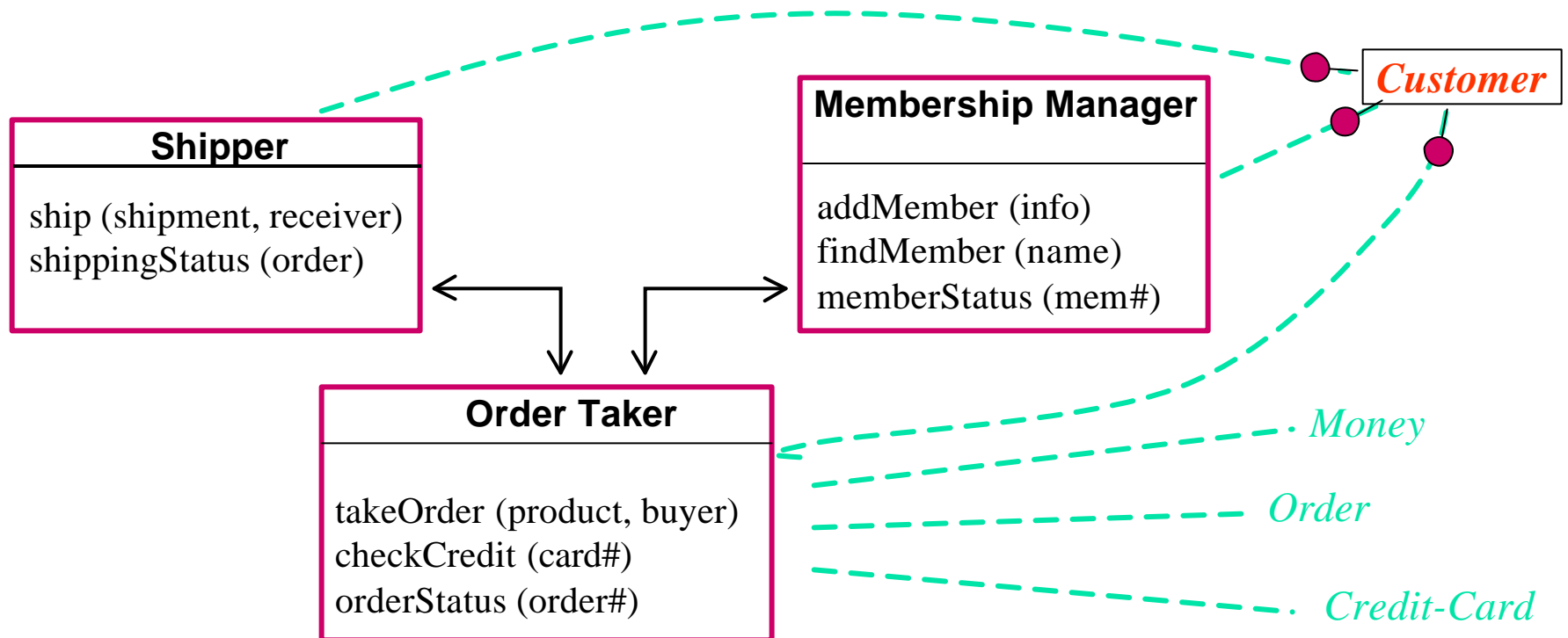
Type-Based Components



- Any component that provides the correct interface (operations and apparent type model) can plug-into another component

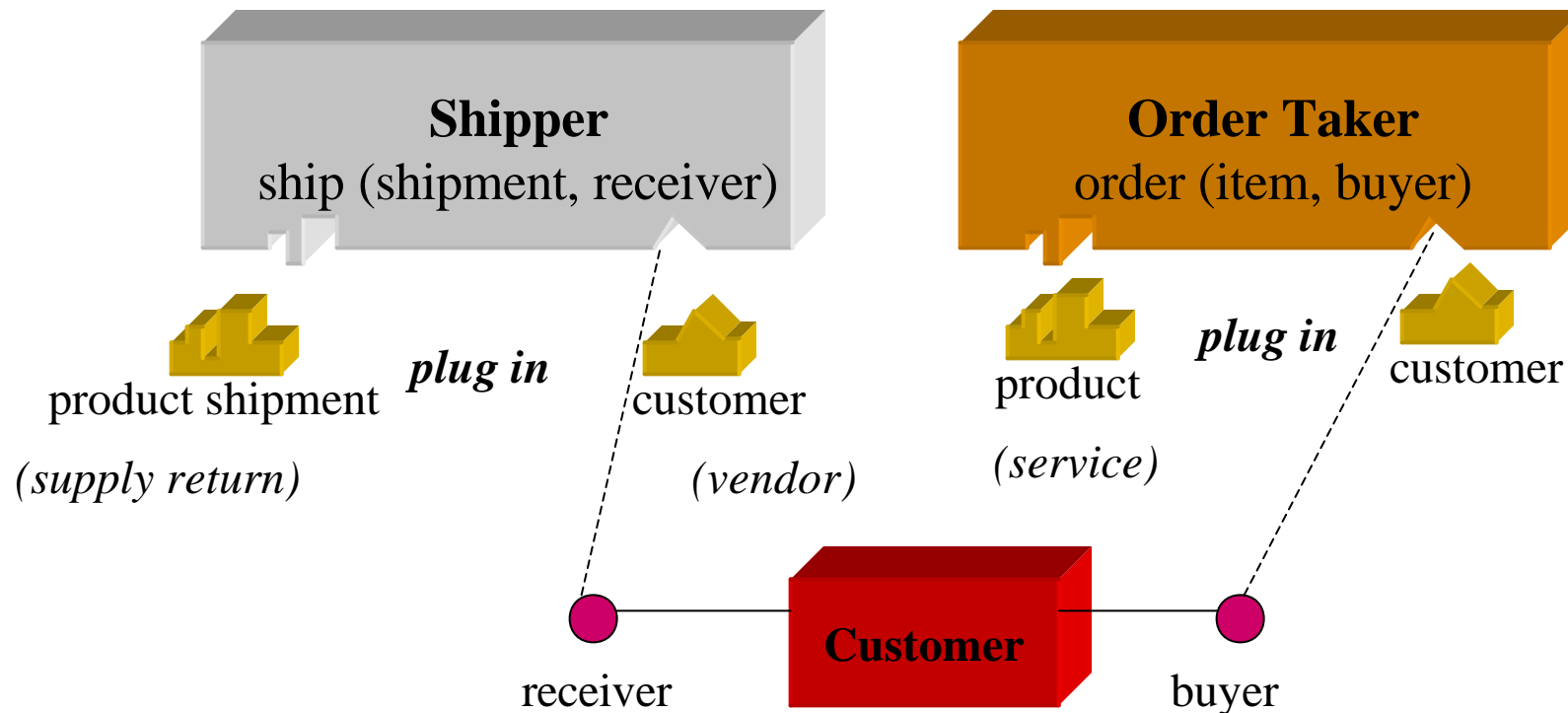
Large-Grain Business Components

- “Executable” component = large-grained object
- Components configurations tend to be more static
- One component may be built from several classes
- Underlying objects implement several types e.g. *Customer*



"Frameworks" as Components

- A large-grain component designed with "plug-points"
- Application will "plug" domain objects into plug-points
- "Plug-in" based on interface, sub-class, delegation, etc.





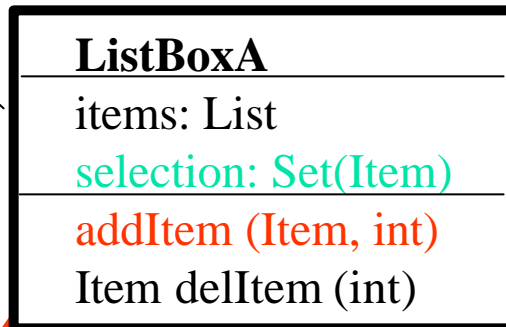
Outline

- Method Overview
- Type Models
- Collaboration Models
- **Refinement Models**
- Framework Models
- Process

Class Vs. Type -- A Basic Refinement

- Any valid implementation class
 - Selects data members
 - Implements member functions
 - Can implement (*retrieve*) the model queries
 - Guarantees specified behavior

class



```
addItem (Item item, int pos) {  
    items.insertAt(item,pos);  
    selection.add(item);  
}
```

<<implements>>

ListBox <<type>>

count: int
itemAt (int): Item
selected (Item): boolean
movedUp(i,j): boolean

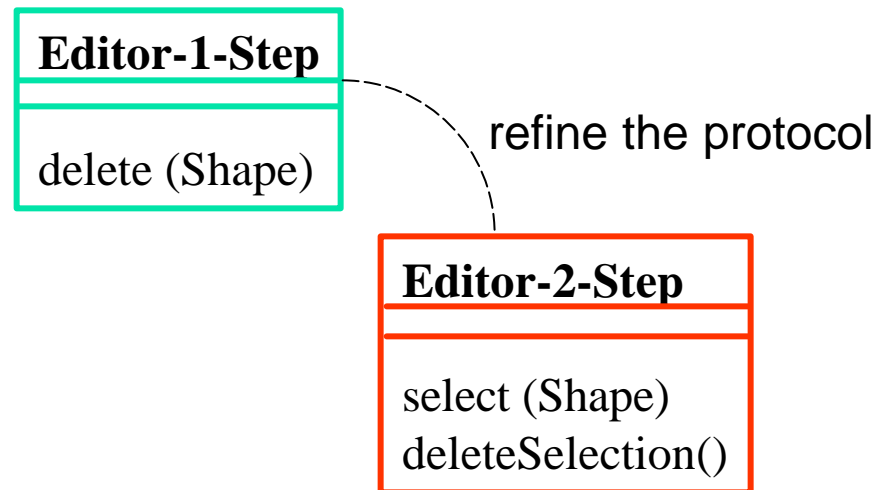
addItem (Item, int)
Item delItem (int)

retrieval:

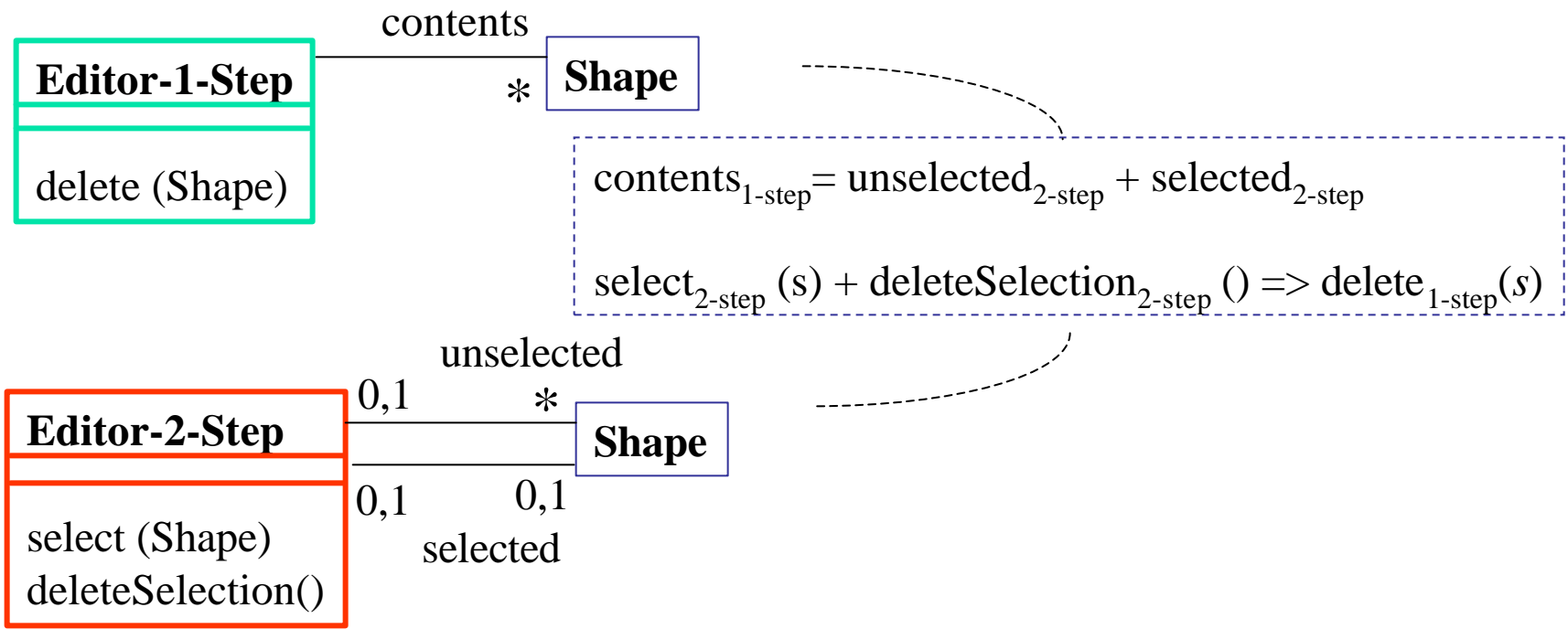
```
boolean selected(i) { return (selection.has(i));}  
int count() { return (items.length()); } ....  
Item itemAt(pos) { return items[pos]; }
```

Subtypes and Refinements

- Sub-types refine (and retain) guarantees made by super-types
 - The concrete implements, and can *retrieve* to, the abstract
 - Any sub-type implementation meets all super-type behavior guarantees
 - *Clients remain blissfully **unaware** of any change*
- Here is a common refinement: is it a sub-type?

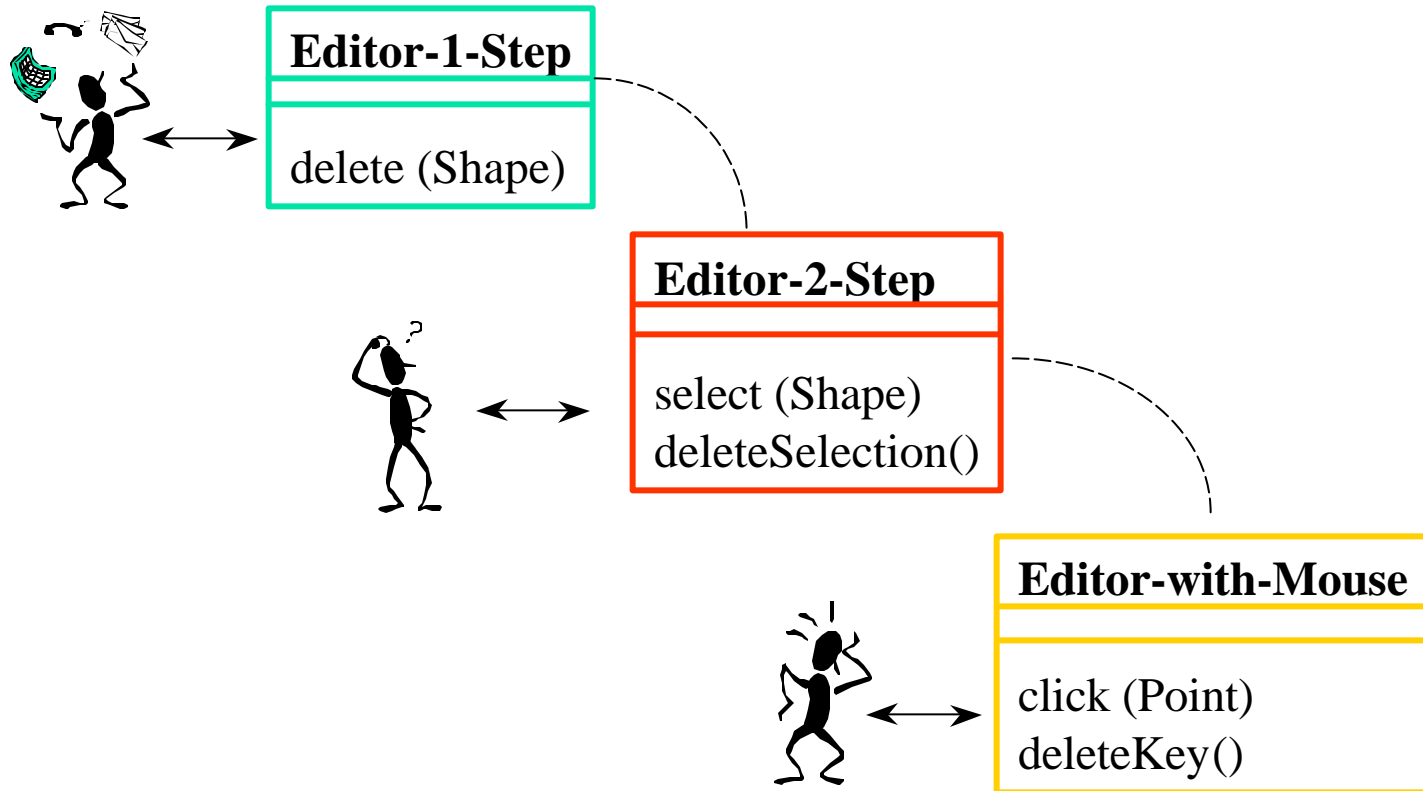


Refined Models and "Retrieval"



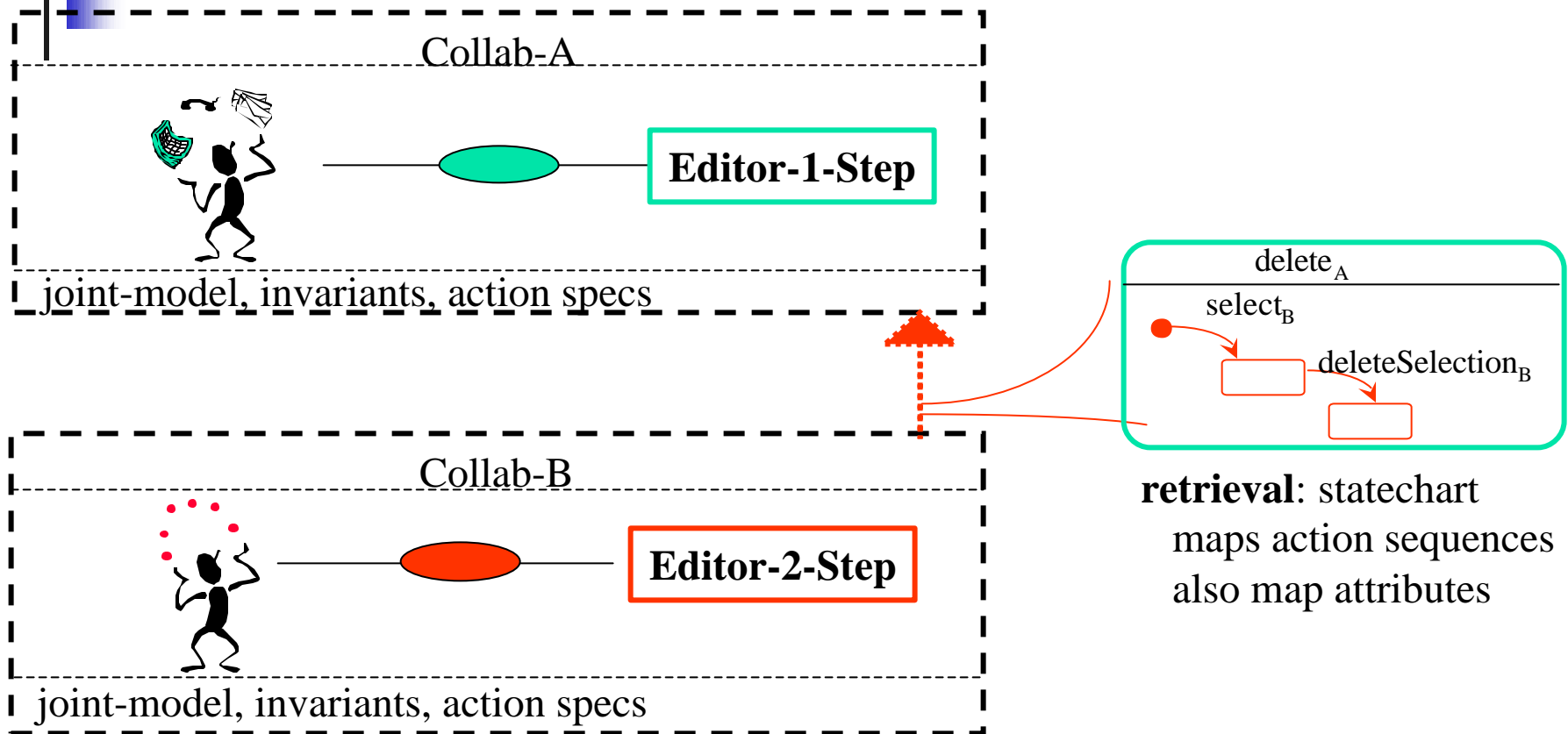
- Finer grained interactions induce a finer grained model
- Retrieve: Define abstract query in terms of refined model
 - Define refined sequences that achieve each abstract action
- Still, this is *not* a sub-type

Varieties of Refinements



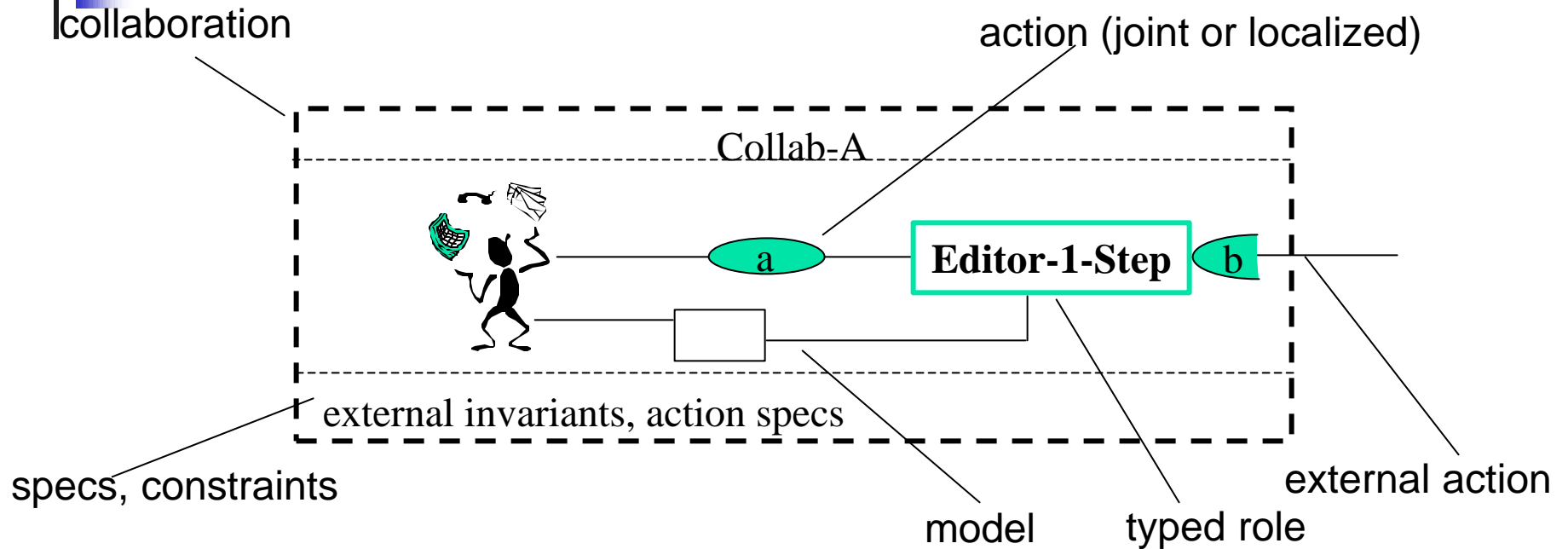
- Many common refinements do not create sub-types
 - Time granularity, signature, helper objects, ...

Joint Refinement of Collaborations



- We refined the *joint interaction protocol*
- Both sides are affected by the refinement

Collaboration

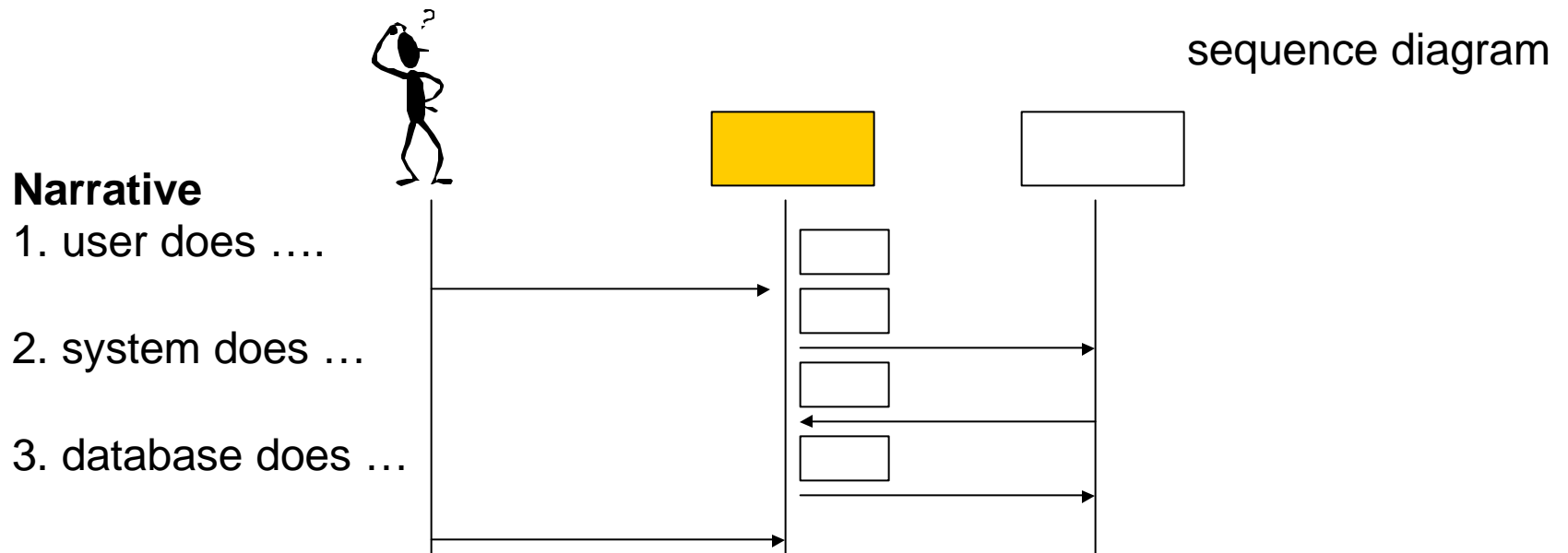


Collaboration:

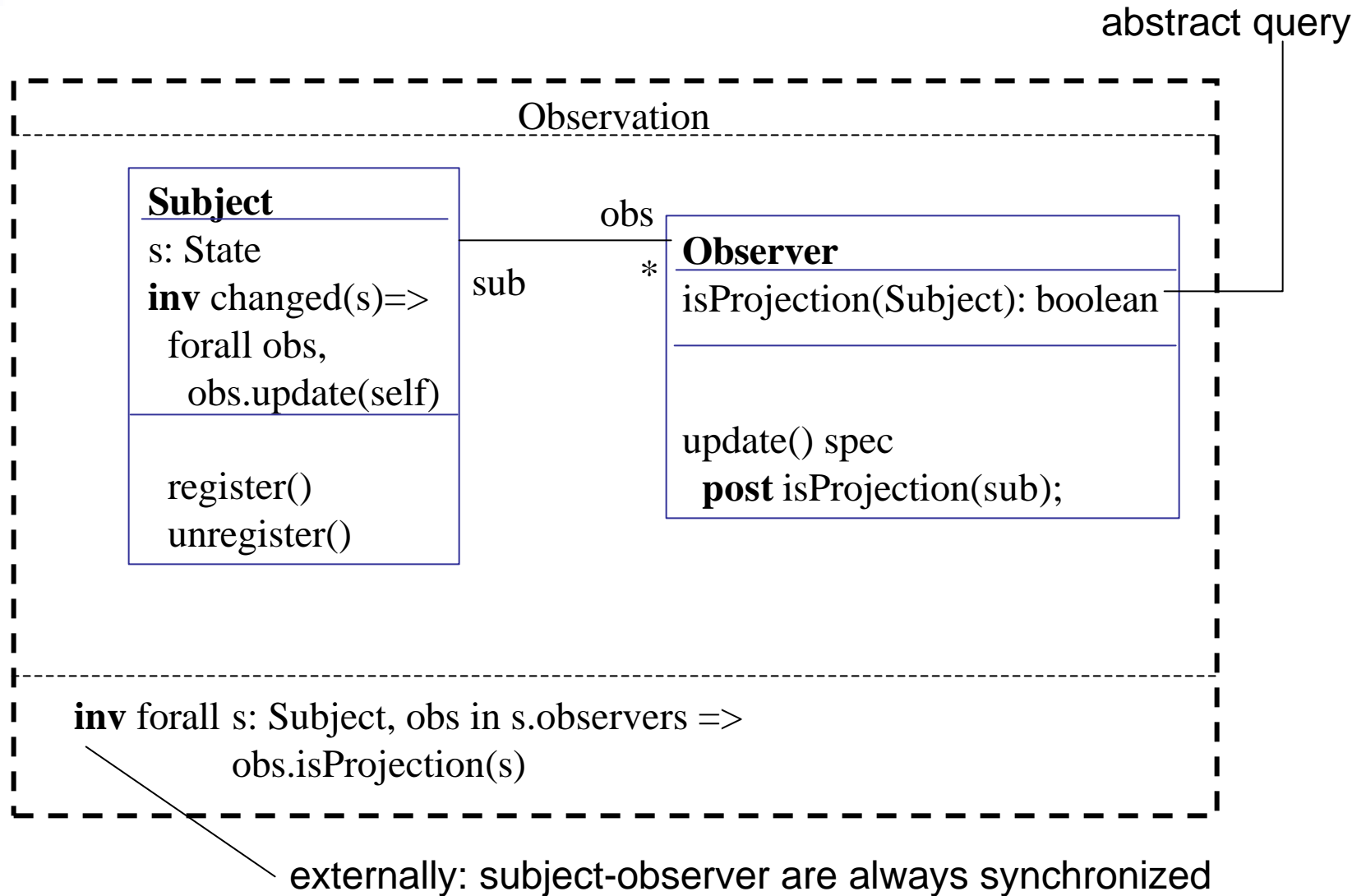
A set of actions between typed objects playing certain roles, specified in terms of a common model. The actions themselves may be *joint* (not assigned to a particular type) or *localized* (responsibility assigned to a particular type), and may be *external* (not between collaborators, must maintain invariants) or *internal* (between collaborators, does not have to maintain invariants).

Scenario and Sequence Diagram

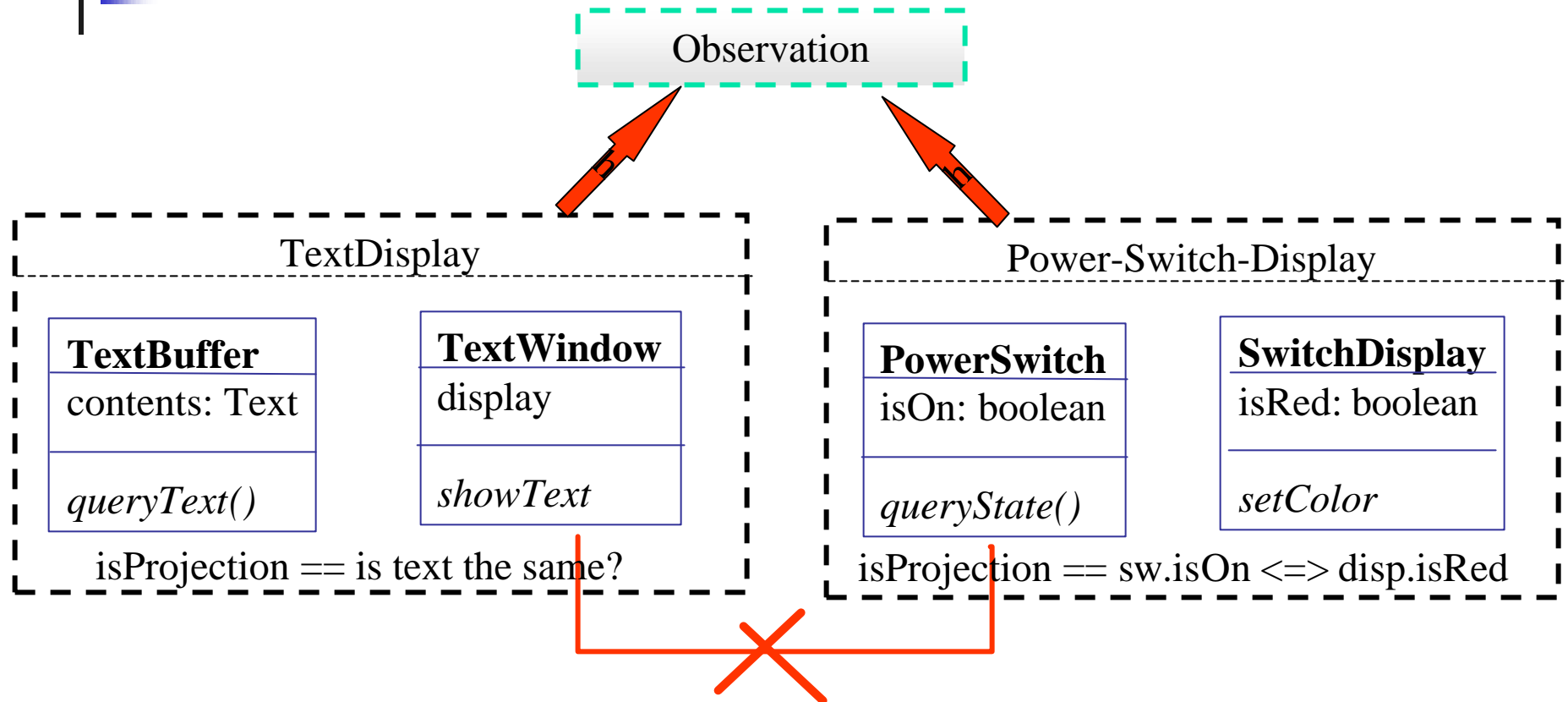
- A scenario is a trace through a collaboration
- Action pre/post attribute values are “snapshots”
- Snapshots conform/define type model
- Snapshot-pairs conform/define action specs
- Scenarios conform/define collaboration specs



Subject-Observer Collaboration

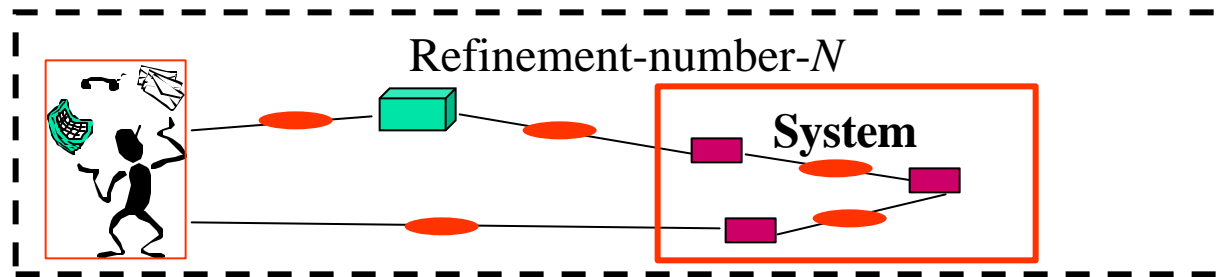
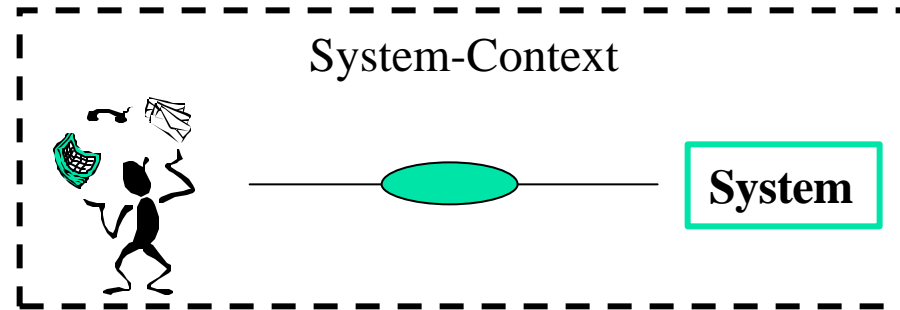


"Specializing" Subject-Observer



- Do not confuse this with subtype or subclass
 - The entire family of related types (playing roles) is being specialized
 - Will be addressed by "frameworks"

From Use-cases to Code (and Back)



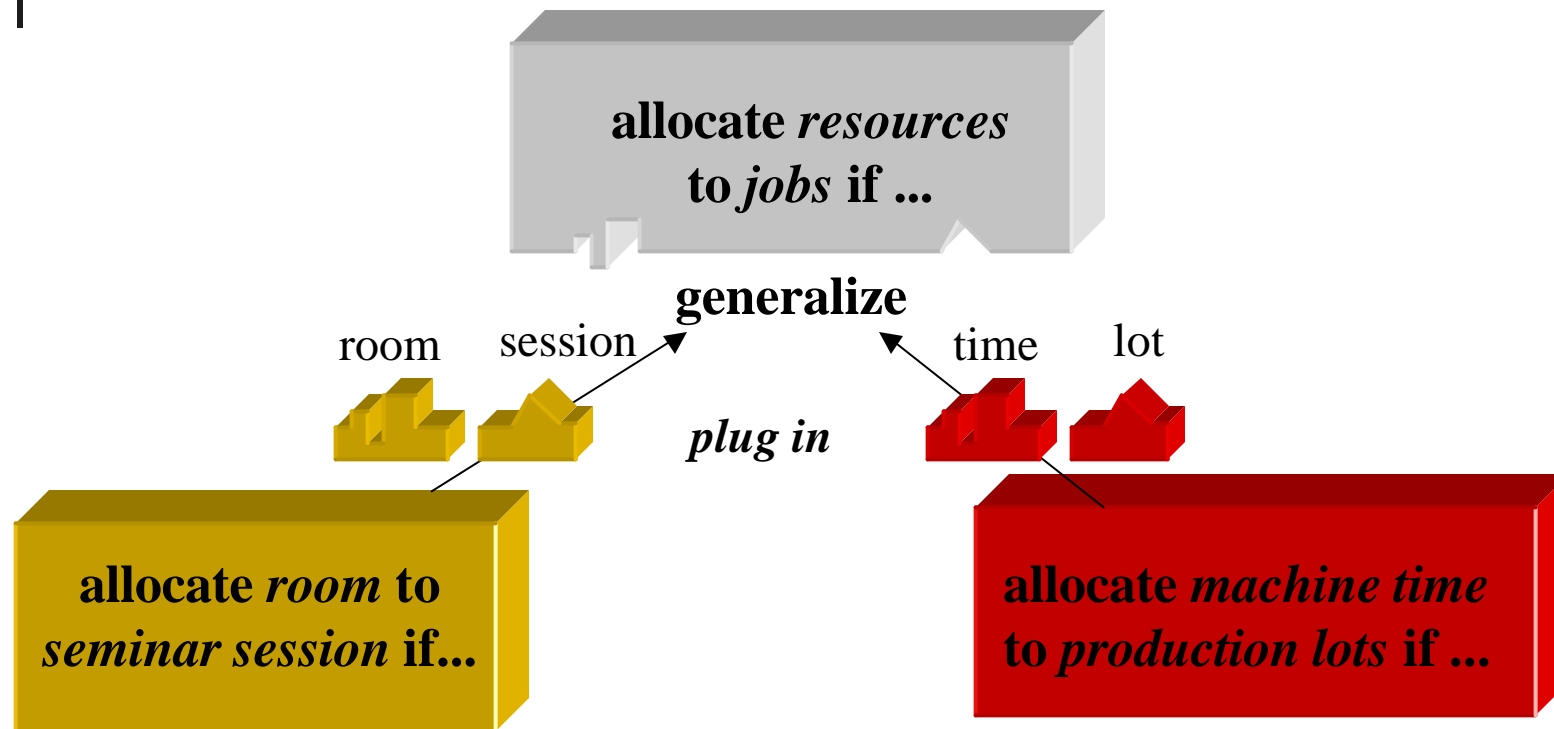
- Collaborations and refinement provide full traceability
 - Use-cases at the level of system and user-tasks
 - Refinement of interaction granularity, external and internal roles
- Clear semantics for use-case development, business to code



Outline

- Method Overview
- Type Models
- Collaboration Models
- Refinement Models
- **Framework Models**
- Process

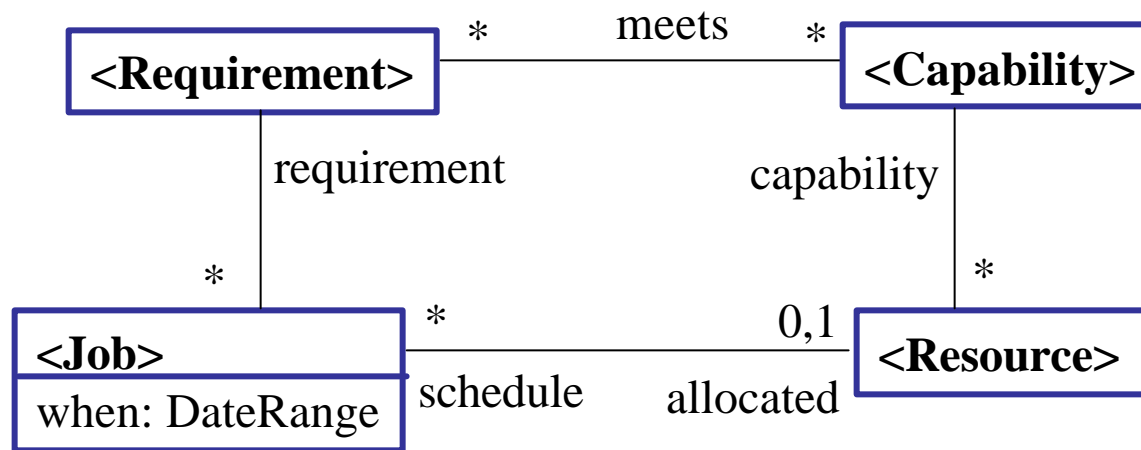
Frameworks - *Generic Components*



- A generic model / design / implementation component that
 - Defines the broad generic structure and behavior
 - Provides *plug-points* for₃₇ adaptation

Resource Allocation Framework

ResourceAllocation



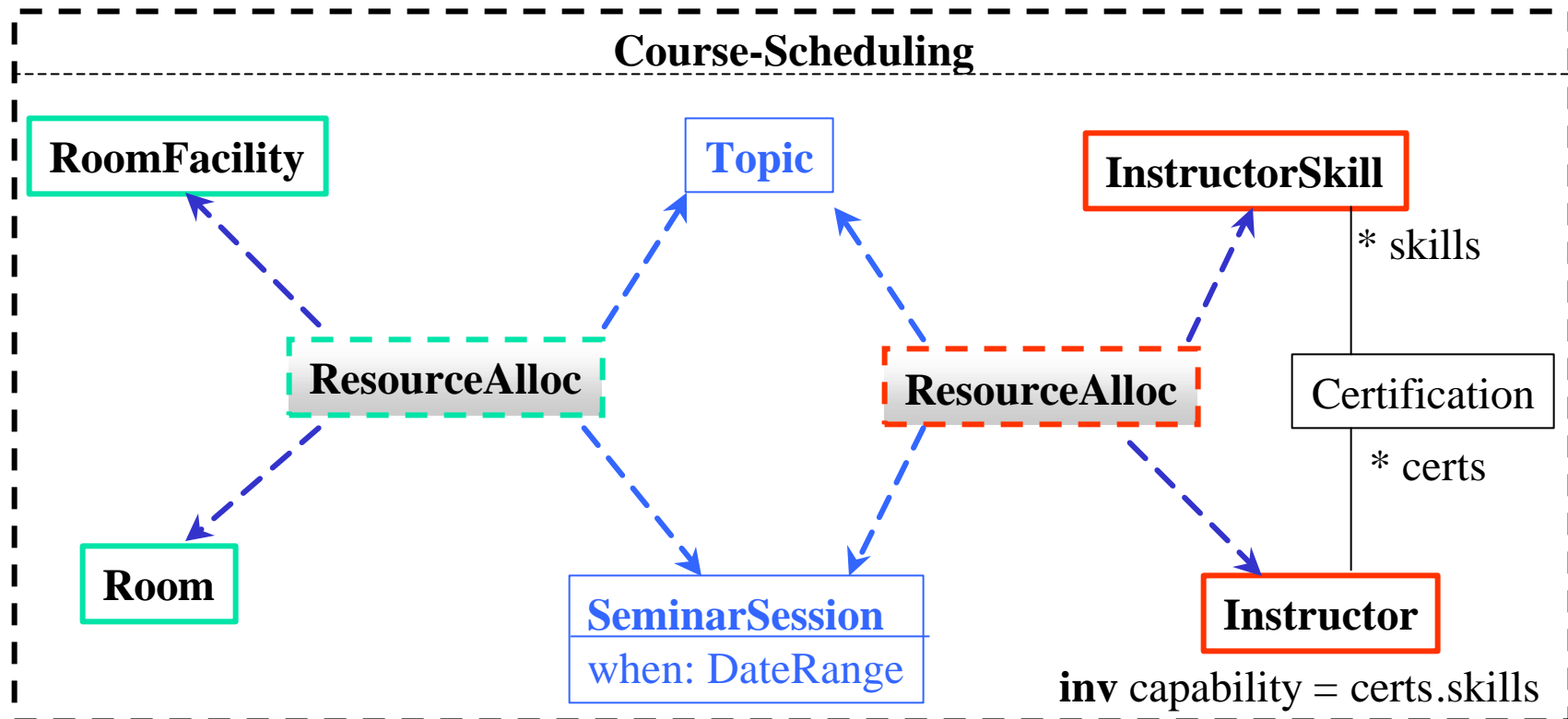
invariants

Job:: //only allocate resource whose capability matches requirements
allocated <> nil **implies** allocated.capability.meets #includes (self.requirement)

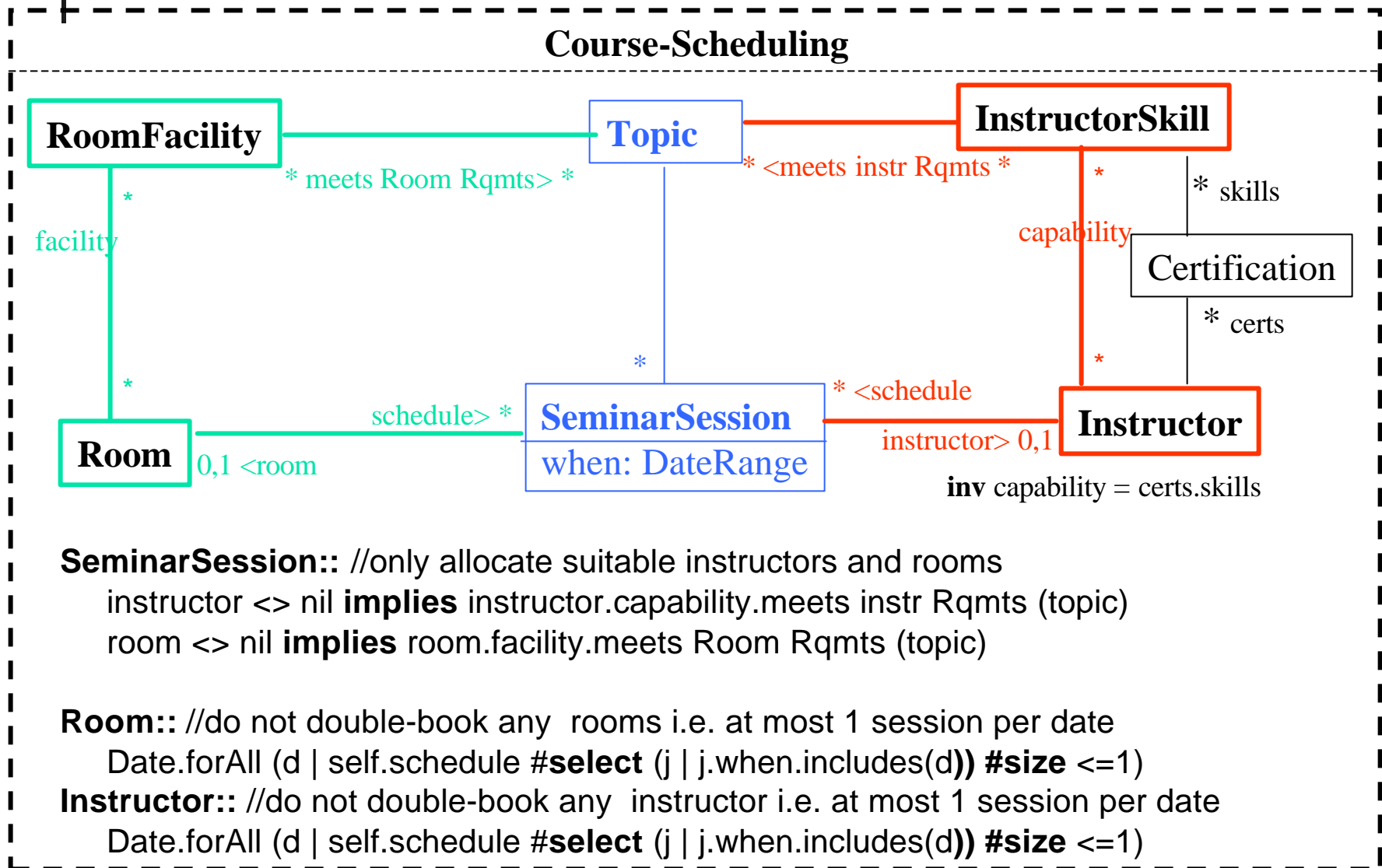
Resource:: //do not double-book any resource i.e. at most 1 job per date
Date.forAll (d | self.schedule #select (j | j.when.includes(d)) #size <=1)

"Applying" a Modeling Framework

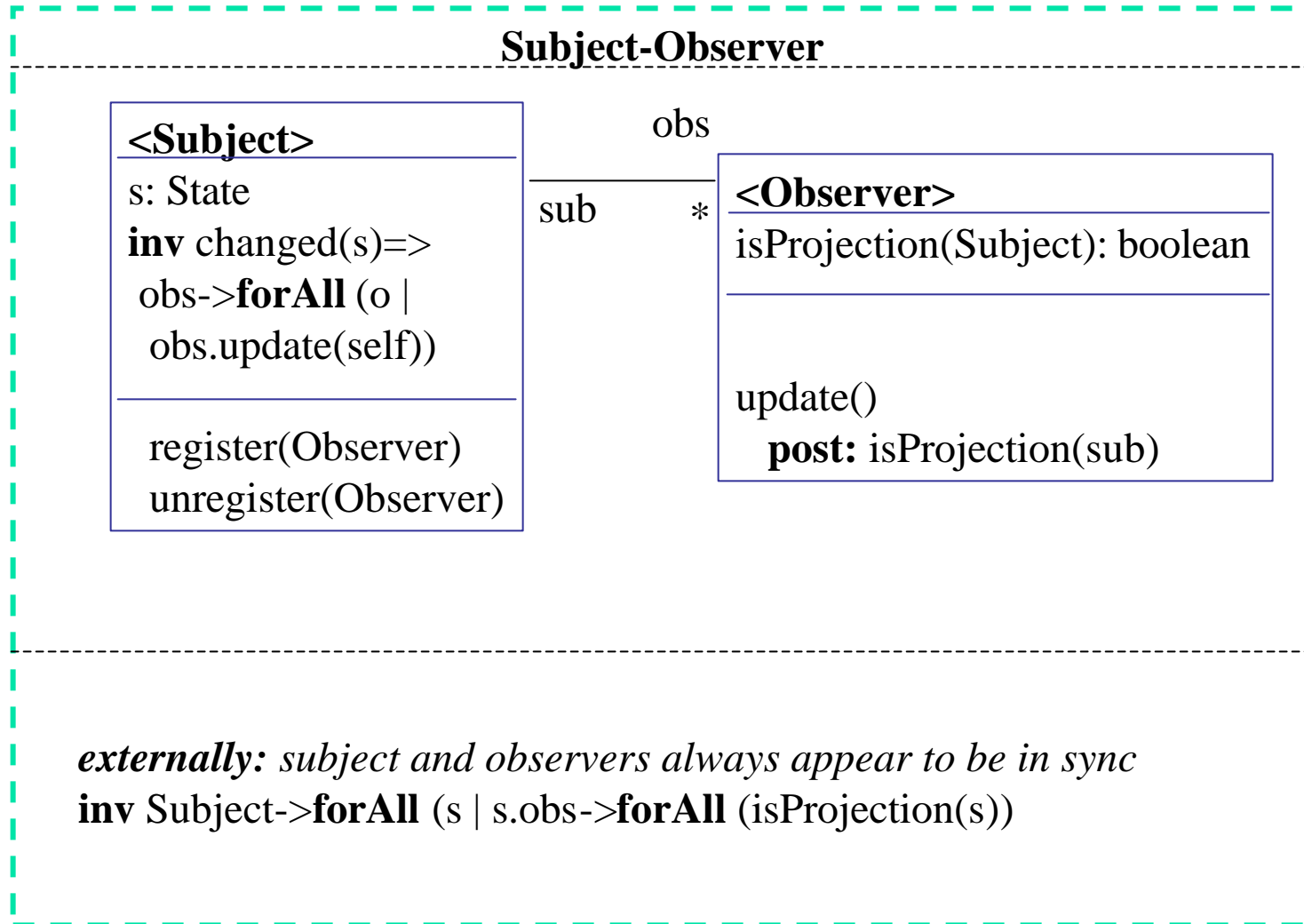
- **"Apply"** resource-allocation **twice** to course scheduling
 - Each application substitutes different *resource, capability, etc.*
 - Both apply to the same *job*: Seminar Session



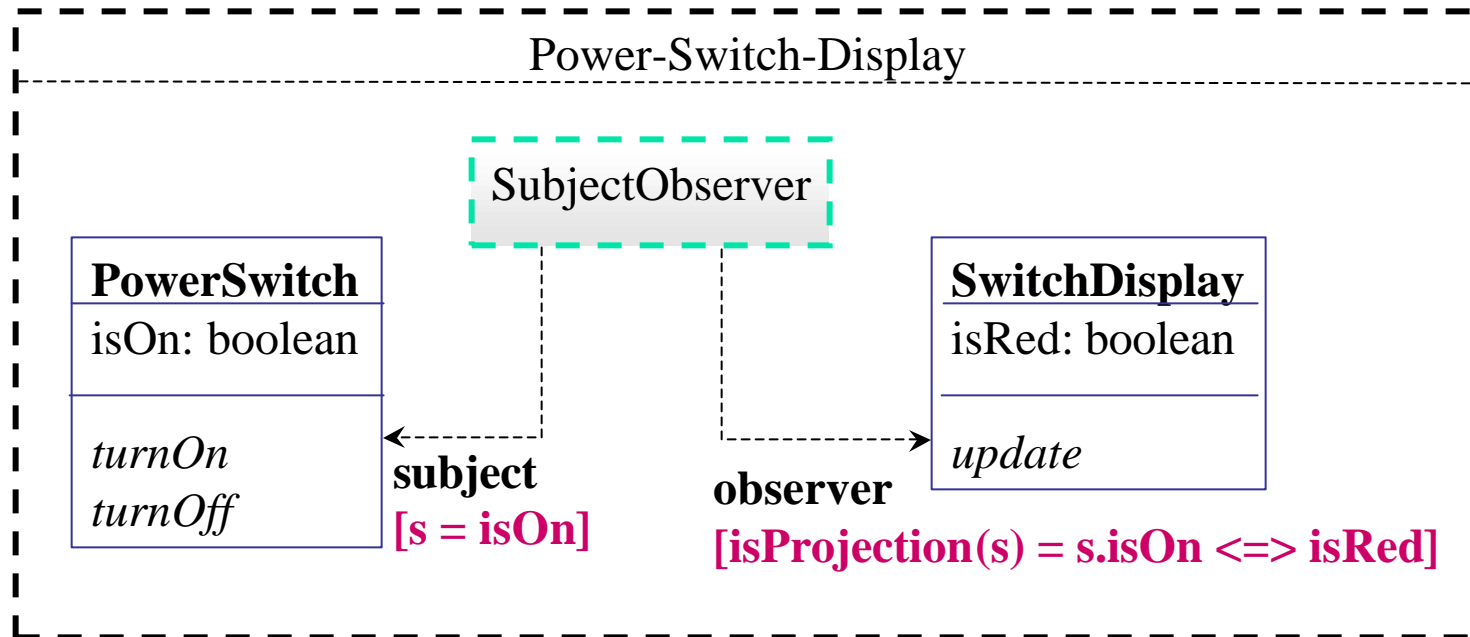
The "Model" is Generated



Design Patterns as Frameworks



Applying Design Patterns



- The instantiation defines mappings of types, queries, actions
 - Needed to generate the instantiation, and for the “retrieval”

Example Frameworks at All Levels

Business Models

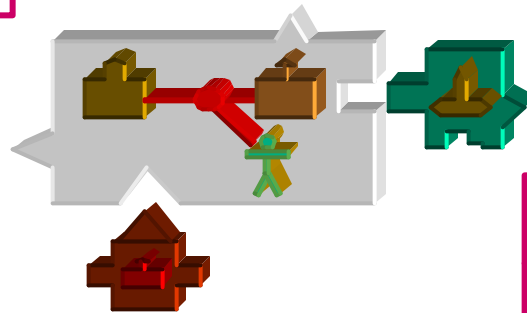
Barter
Trader
Authorizer

Domain Models

Resource Allocation
Account Settlement
User-Interface Patterns

Design Patterns

Subject-Observer
2-Way Link
Cache
Moving Window
Data Normalization



Fundamentals

Total Ordering
Groups
Range
Descriptors

- **Constructive approach to modeling and design with full traceability**
- **Libraries and commerce of frameworks of models, designs, and code**

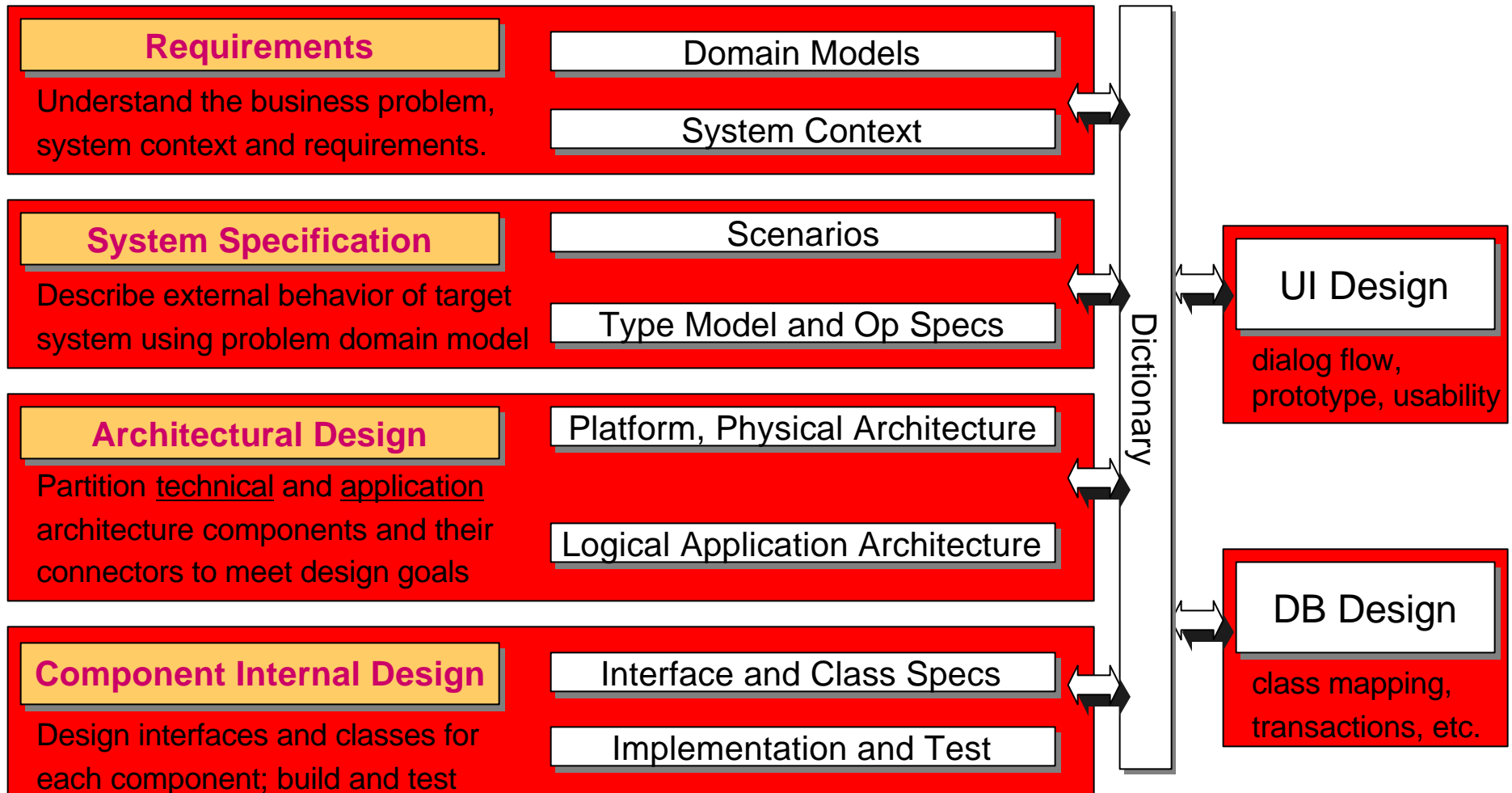


Outline

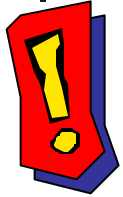
- Method Overview
- Type Models
- Collaboration Models
- Refinement Models
- Framework Models
- **Process**

Catalysis Development Process

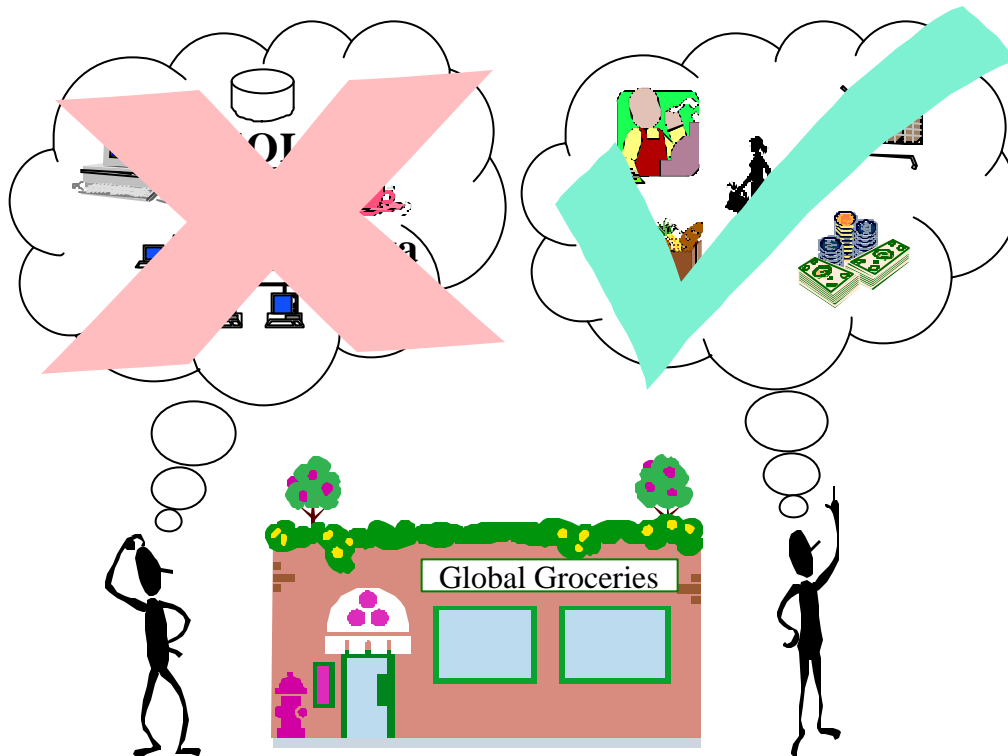
UML = Unified Modeling Language, standard notation for OO design



Focus on the Problem Domain



External models should reflect the customer's view of the problem domain, *not* the programmer's view.

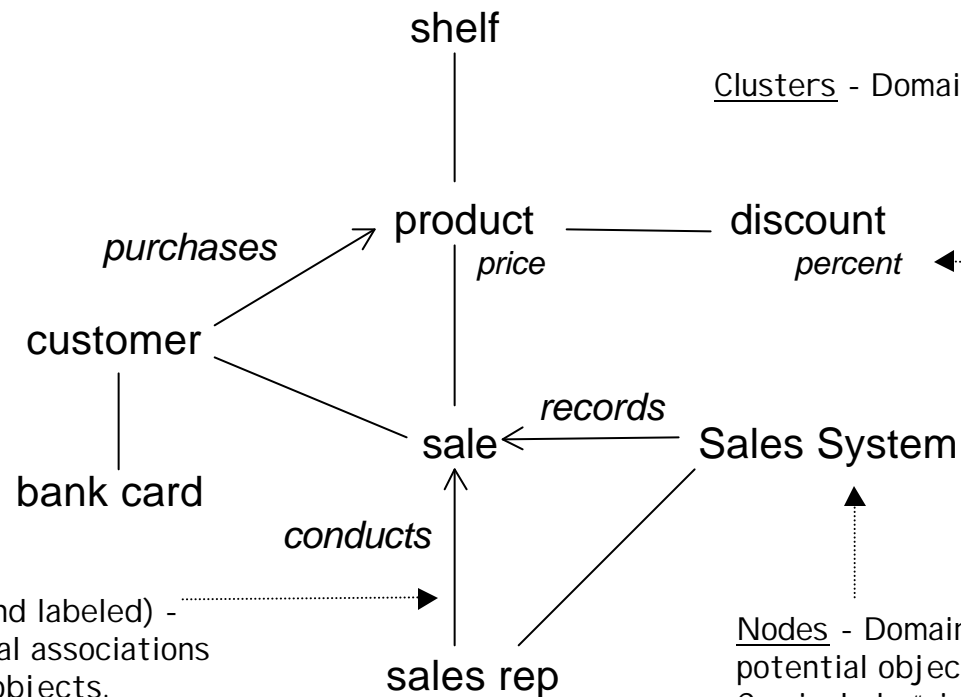
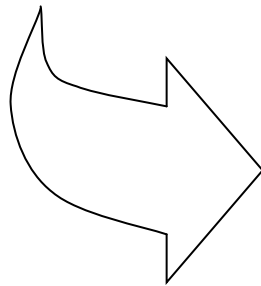
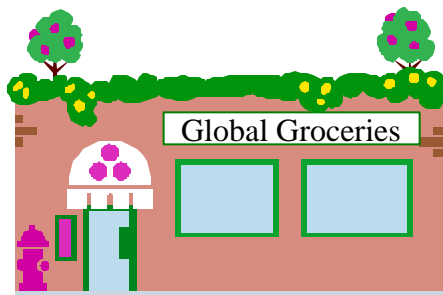


A problem domain focus helps to ensure continuity between the software and the real world.

- Continuity makes it easier to
- verify model with customer
 - train new developers
 - estimate maintenance effort
 - identify sources of defects

Informal Domain Model

Concept Map - An informal structured representation of a problem domain
Not a stored data model!!



Clusters - Domain terms representing potential attributes.

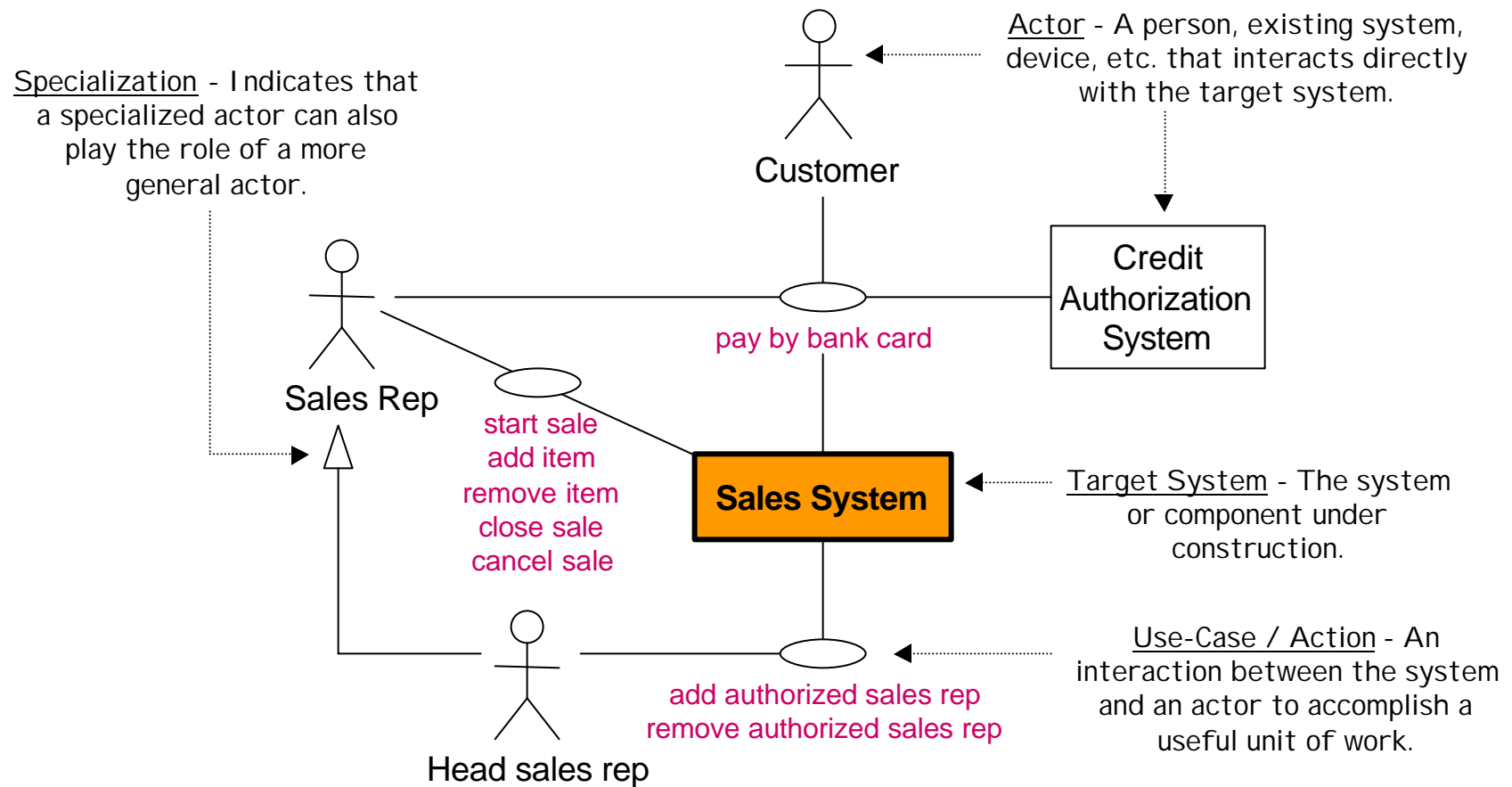
Arcs (optionally directed and labeled) - Representations of potential associations or collaborations between objects.

Nodes - Domain terms representing potential objects, types, or actors. Can include "rich-pictures" as drawings of the problem domain

Can be formalized if appropriate

System Context -- a Collaboration

System Context Model - A structured representation of the collaborations that take place between a system and objects in its surrounding environment (context).



Scenario of Use

Context: A customer approaches a sales rep with the intention of purchasing three watzits using her bank card. There are sufficient funds in her account to pay for the purchase. The sales rep has completed his last sale so there is currently no sale in progress.

Actors identified from System Context.

Narrative:

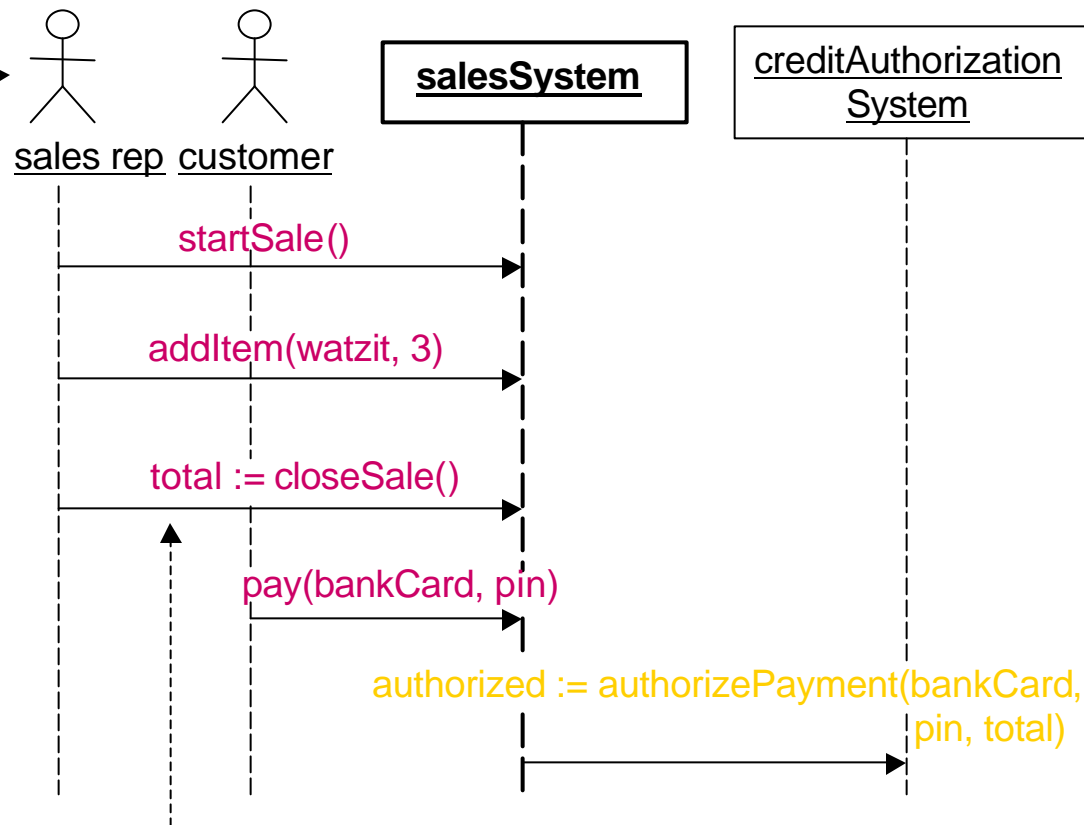
The sales rep starts a new sale. The total for the sale is \$0.00.

The sales rep adds the three watzits to the current sale.

The sales rep closes the sale. The sales system returns the total due.

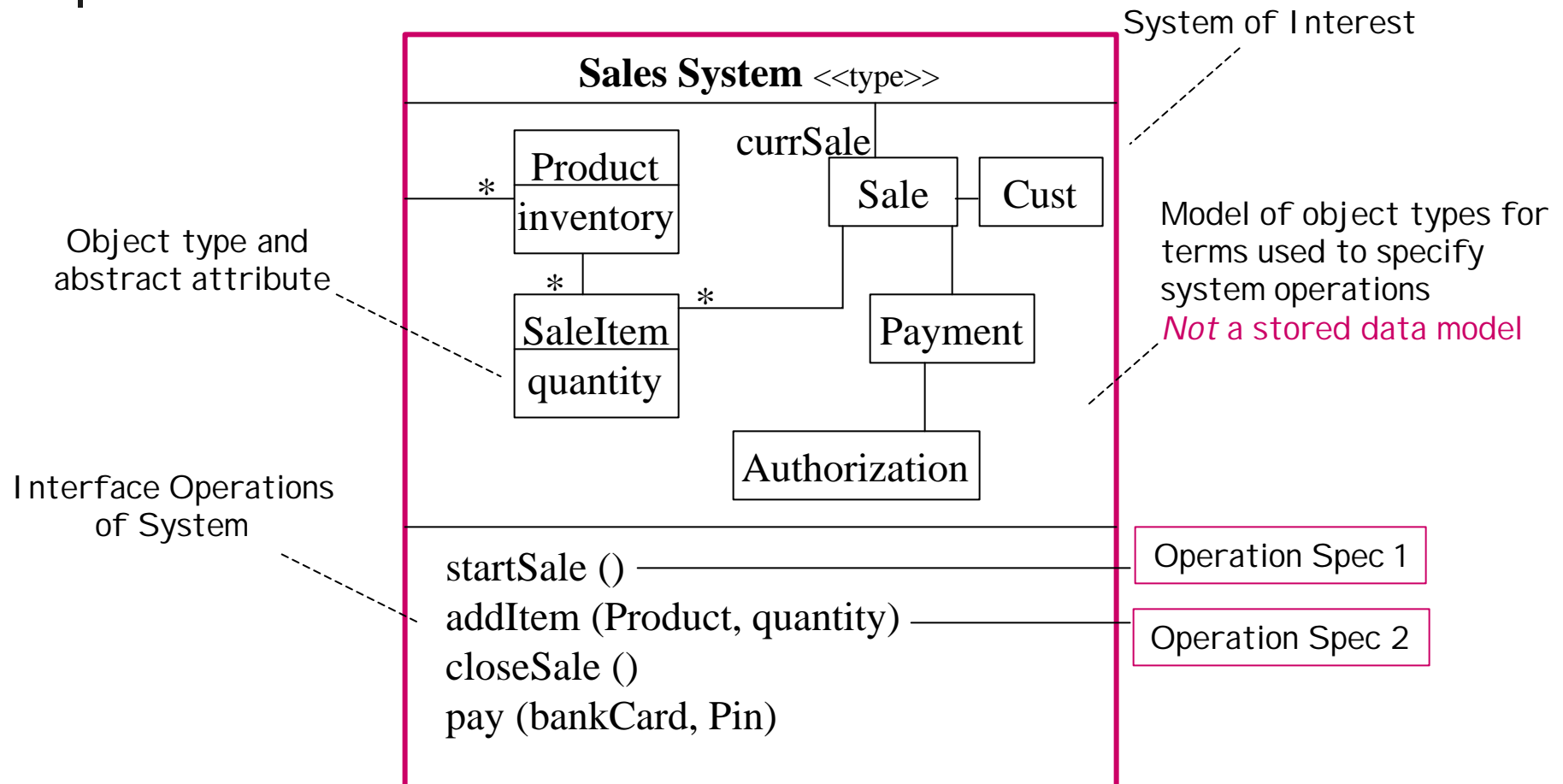
The customer inserts her bank card into the reader and enters her pin number

The Sales System requests payment authorization from the credit authorization system. The system authorizes payment.



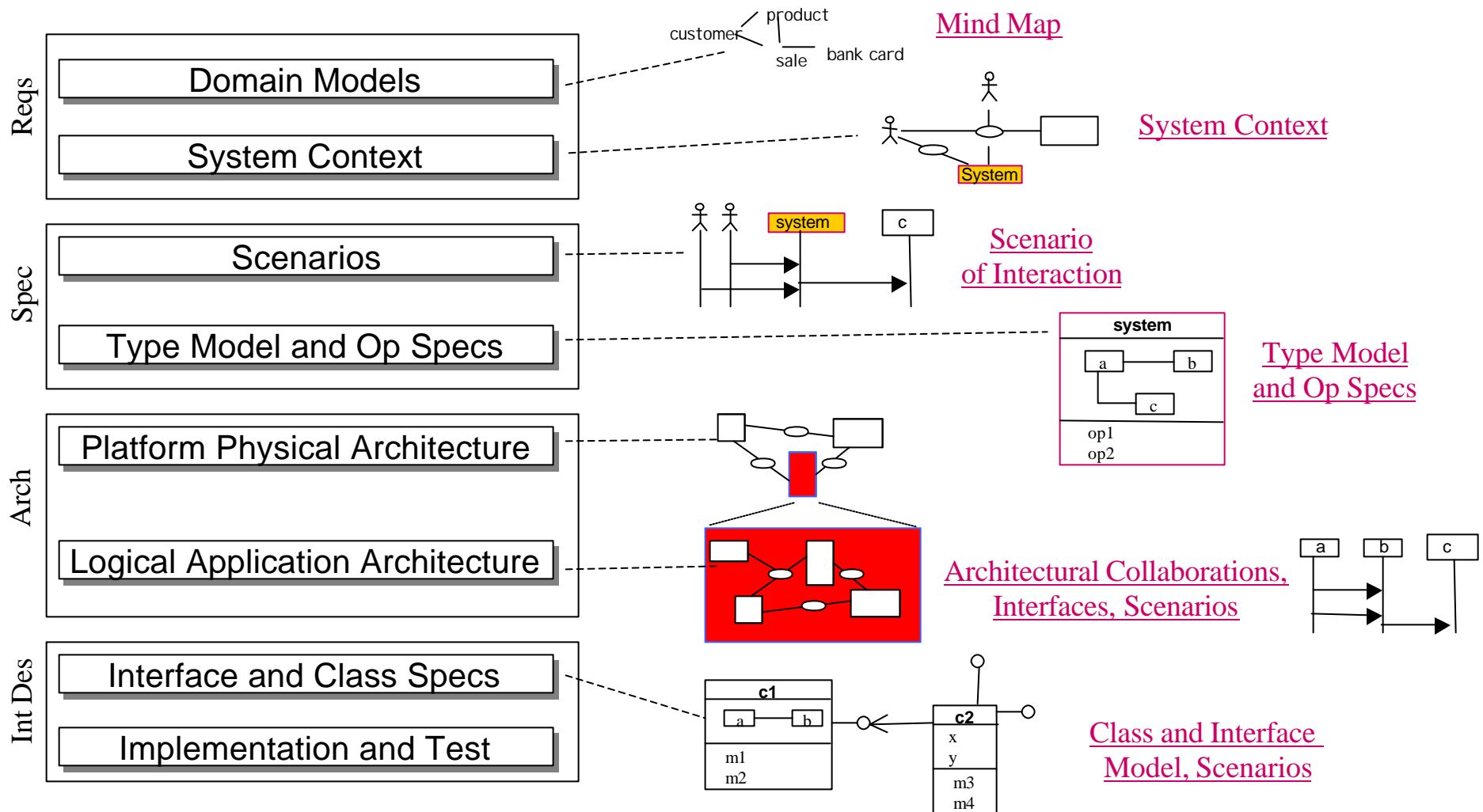
Interactions from system-context actions

Type Model and Operation Specs



Note: Does not as yet commit to operations on individual classes inside system
Internal component partitioning and class design not decided yet.

A Recursive Process: Domain to Code





CBD Bibliography

- “Objects, Components, and Frameworks with UML: The Catalysis Approach”, Desmond D’Souza & Alan Wills. ISBN: 0201310120.
- “UML Components: A Simple Process for Specifying Component-Based Software”, John Cheesman & John Daniels. ISBN: 0201708515.
- “Large Scale Component Based Development”, Alan Brown. ISBN: 013088720X.
- “Component Software: Beyond Object-Oriented Programming”, Clemens Szyperski. ISBN: 0201178885.
- “Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise”, Herzum & Sims. ISBN: 0471327603.