Precise Component Architectures

with UML and CATALYSIS

Desmond D'Souza and Ian Maung

Presented at OOPSLA 1999

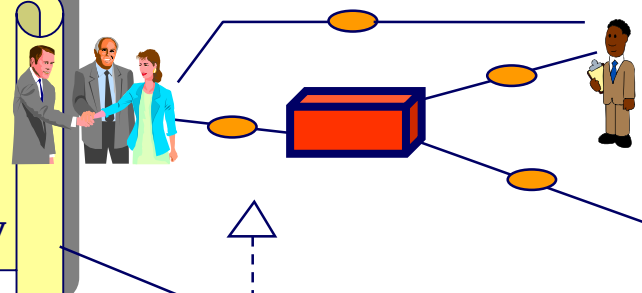
About the Speakers

Desmond D'Souza is senior vice president of component-based development at Computer Associates, and leads its Catalysis/CBD Technology Center where he works on methods, tools, and architectures for effective component-based software engineering. He is co-author and developer of the *CATALYSIS* method, published by Addison Wesley in 1998. Mr. D'Souza is a sought after authority and speaker at companies and conferences internationally and may be contacted at dsouzad@acm.org

Ian Maung is a Senior Architect at Computer Associates and a member of its Catalysis/CBD Technology Center, working on methods, architectures, and tools for effective component-based software engineering. He was the co-developer of the CBD/Catalysis process guide available from CA, and has extensive background in object modeling, semantics, and formal methods. Dr. Maung may be contacted at ian.maung@platinum.com

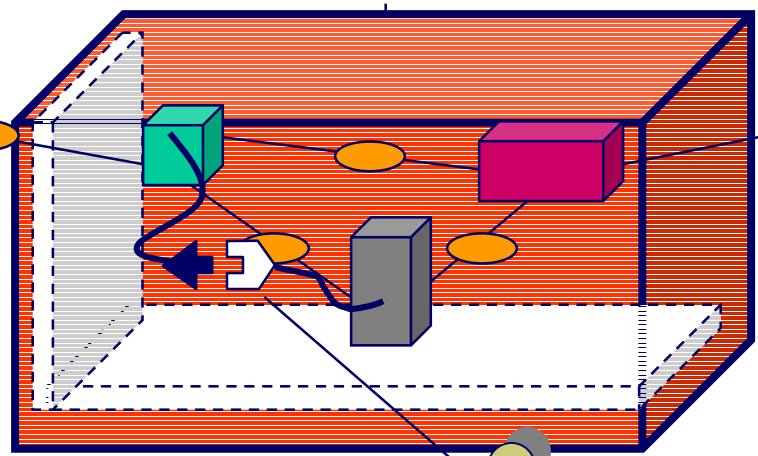
Success with Components

2. Traceability
bridge **Business** and IT
precise shared vocabulary
critical business questions **early**



1. Business Driven
understand **business first**
relate disparate views

4. Architecture
infrastructure
interfaces, connectors
plug **standards**
… interoperability



5. Reuse Process
develop **for** reuse
develop **with** reuse
first reuse **problem mod**
… then **architectures**
… then interface,code

= Integrated Business and Systems Engineering

3. Pluggable Components
precise interface-centric design
traceable business components
assemble from coherent **component kit**

What is *Catalysis*™?

UML partner, OMG standards, TI/MS standards

Precise models and systematic process

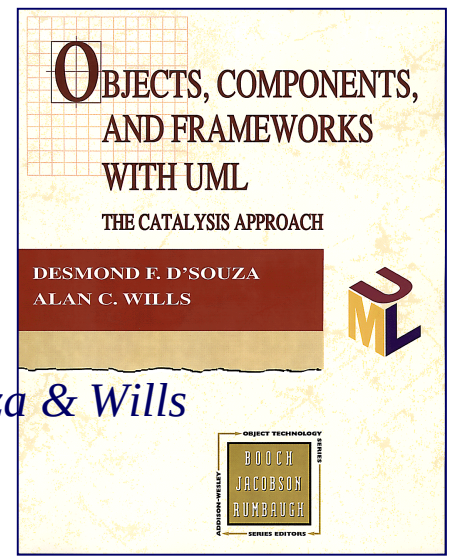
Dynamic non "stovepipe" systems

- A *next-generation standards-aligned* method
  - ✓ For *open distributed component 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
Supports components, OO, legacy, heterogenous systems
Addison Wesley, *"Objects, Components, Frameworks…" 1998, D'Souza & Wills*

The foundation of CA's CBD and methods offering

Outline

Note: some new slides today;
Latest version at catalysis.org

- **Precise component specifications in UML**
  - ✓ **Interfaces**
- Refinement
  - ✓ Component implementation refines spec
- What is Software Architecture?
  - ✓ Common definitions and examples
  - ✓ Catalysis definition of architecture
  - ✓ Specifying architectural styles using OCL
  - ✓ Generating architectural styles using Frameworks
  - ✓ Specifying architectural styles with Stereotypes
- Specifying component architectural styles
  - ✓ Components, ports, connectors and assemblies
  - ✓ Static assemblies
  - ✓ Dynamic assemblies
- Realizing component architectural styles
  - ✓ The CORBA component model
- Conclusion

Type of an Object: Multiple (Inter)Faces

"Plug" into new service

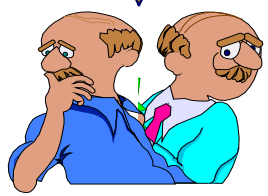
What Type?

What Type?

Airline

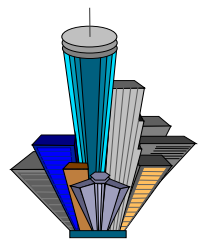
Passenger

check baggage
board
deplane



Guest

do check in
order room service
do check out

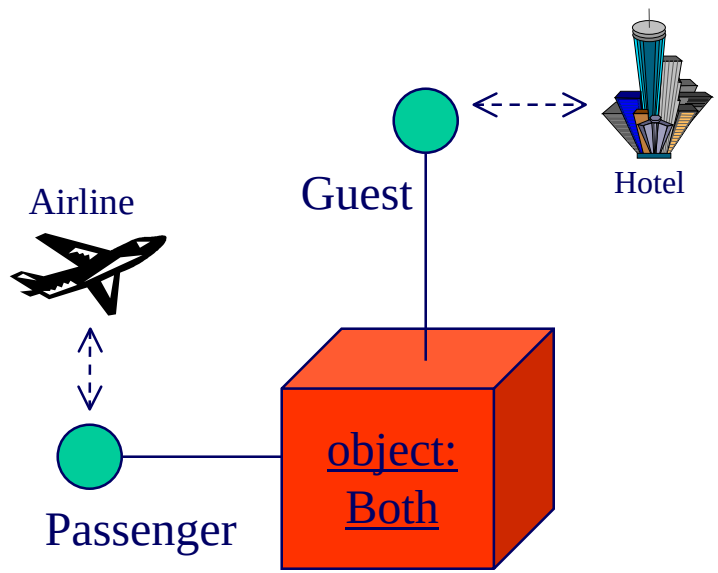


Hotel

- ✓ Object or component plays different roles
- ✓ It offers multiple interfaces, and is seen as different types
- ✓ Benefits to the *airline*, *hotel*: less coupling, more "pluggability"
- ✓ Benefits to *person*: easier to adapt to "plug" into a bigger system

Types: JavaBeans, COM+, Corba, ...

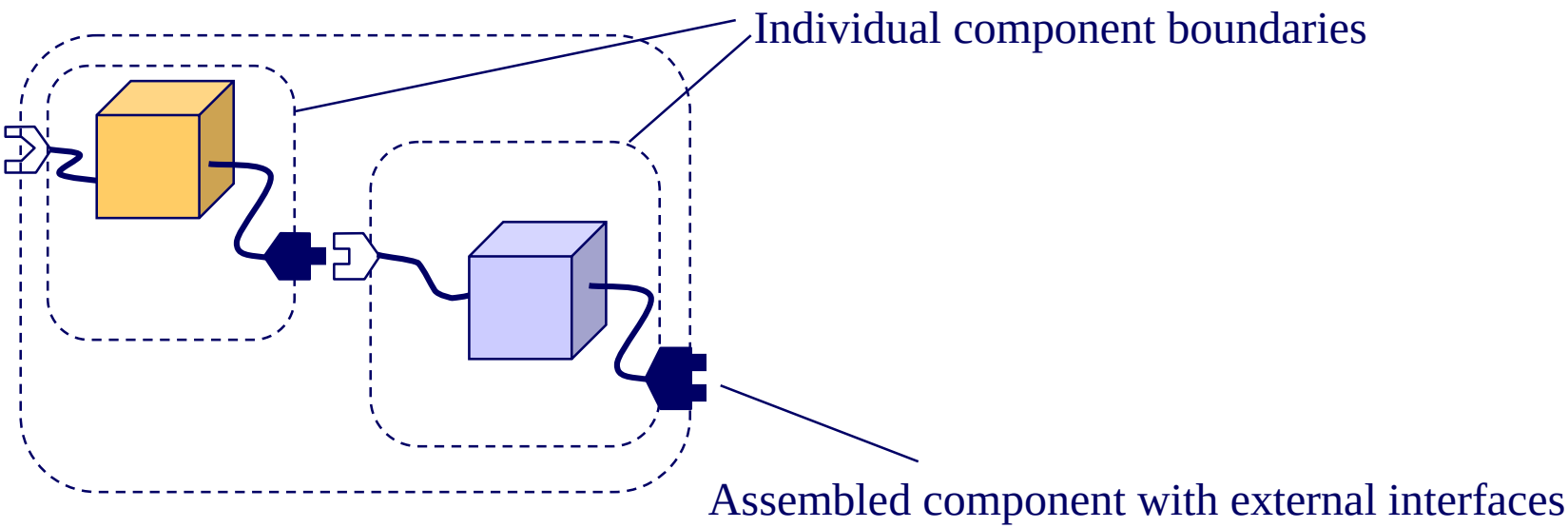
```
interface Guest {
    doCheckIn ();
    orderRoomService();
    doCheckOut ();
}

interface Passenger {
    checkBaggage ();
    board ();
    deplane ();
}
```




```
class Both
   implements Guest, Passenger {

      // implementation ….
}
```

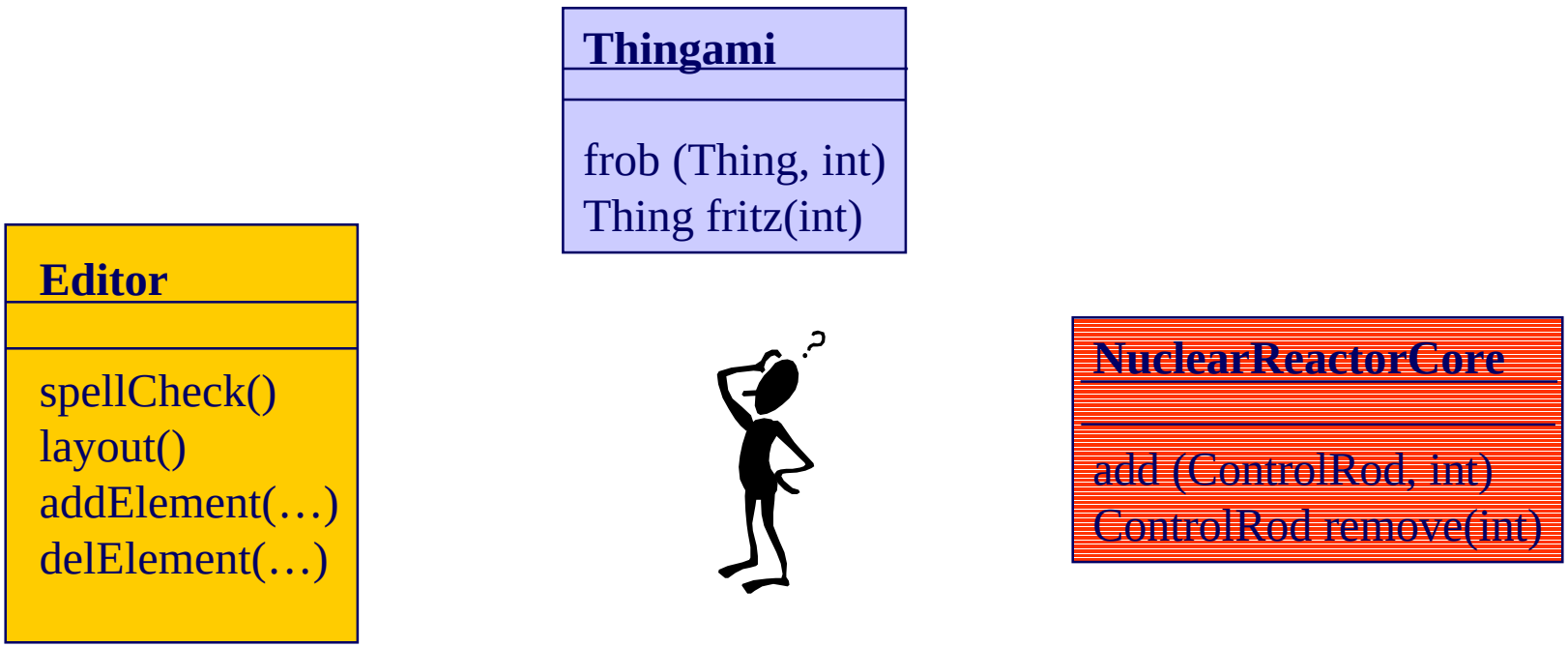
What is a Component?

A package of software that is independently developed, and that defines interfaces for services it provides and for services it requires.



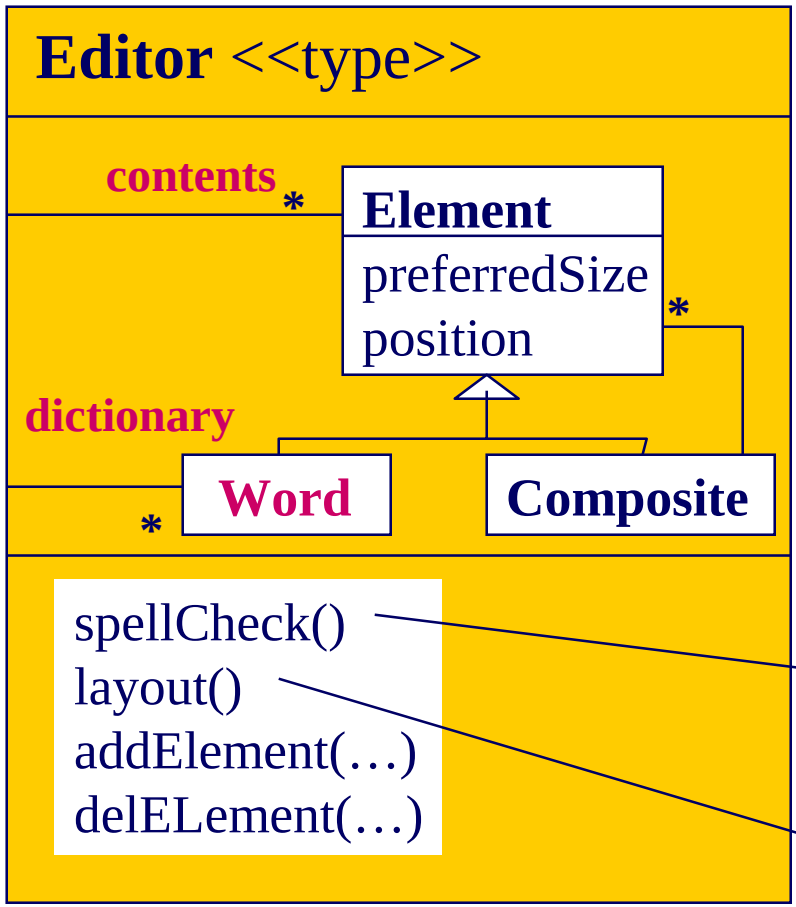
- ✓ External interface totally separated from internal design and implementation
- ✓ Implementation written in terms of interfaces of other components
- ✓ Package can include executable, interface, specs, models, tests, docs, ...

The “Black-Box” Component



- Signatures are not enough to define widely-used components
- Two choices: either “**plug-n-pray**”, or …
- Use better specifications that are both:
  - ✓ **abstract**: apply to any implementation
  - ✓ **precise**: accurately cover all necessary expectations

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** (*vocabulary*) that must *somehow* be known to any implementation

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

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

But, *What about Encapsulation?*

- Encapsulation is about hiding *implementation decisions*
- It does not make sense to hide *interfaces*
- And interfaces always imply *expected behavior* as well
  - ✓ This is what *polymorphism* is all about
- Hence, specify a ***minimal model*** of any implementation

Formalizing Operation Specs

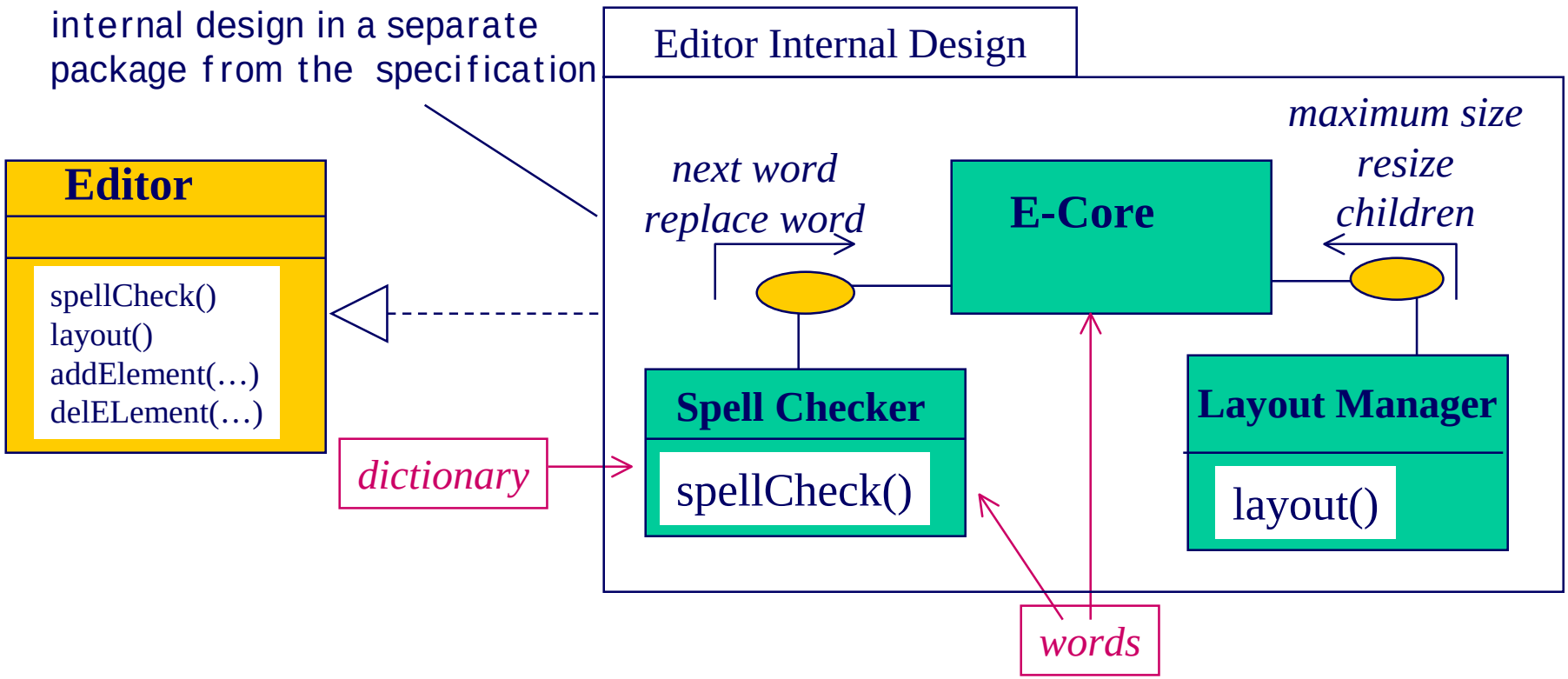
```
Editor:: spellCheck ()
post // every word in contents
   contents->forAll (w: Word |
        // has a matching entry in the dictionary
        dictionary->exists (dw: Word | dw.matches(w)))
```

- Operation specification can be formalized
  - ✓ *OCL* (Object Constraint Language)
  - ✓ Checked, used for refinements, testing, change propagation, …

Outline

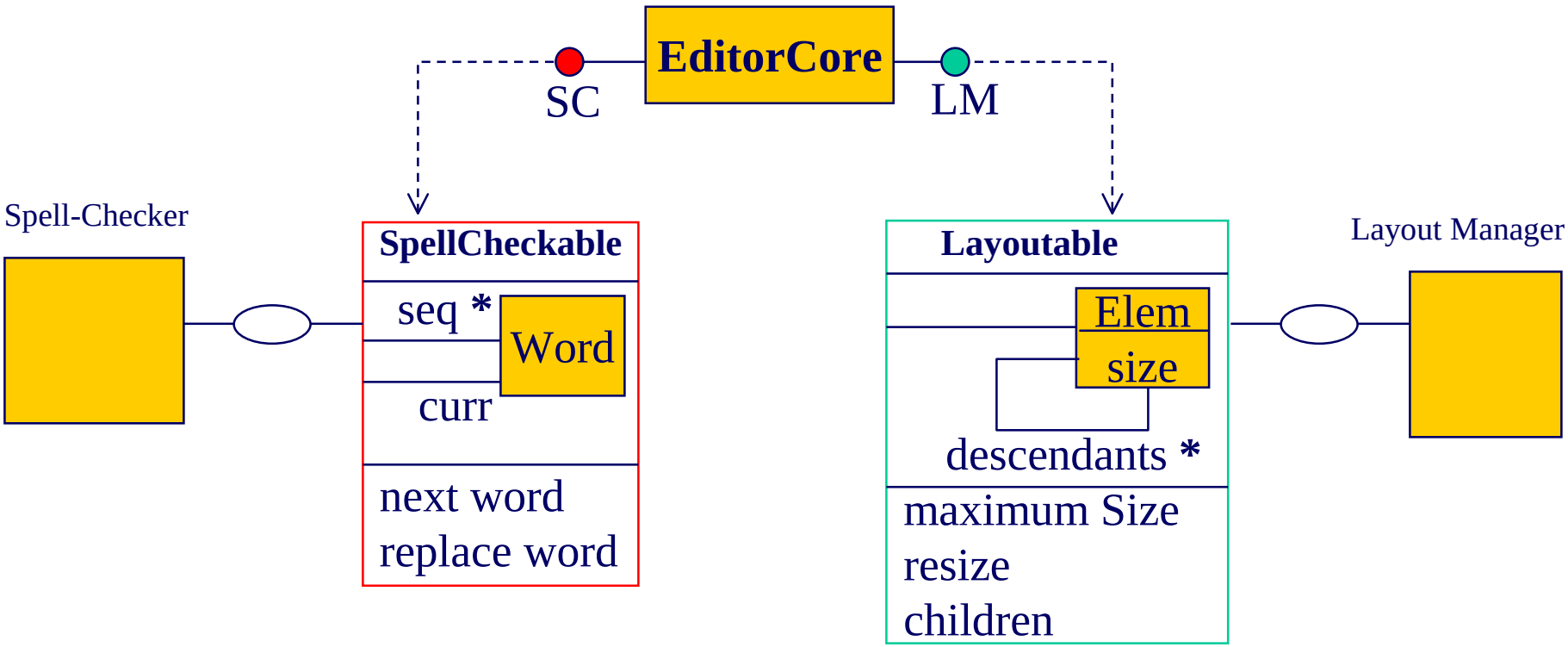
- Precise component specifications in UML
  - ✓ Interfaces
- **Refinement**
  - ✓ **Component implementation refines spec**
- What is Software Architecture?
  - ✓ Common definitions and examples
  - ✓ Catalysis definition of architecture
  - ✓ Specifying architectural styles using OCL
  - ✓ Generating architectural styles using Frameworks
  - ✓ Specifying architectural styles with Stereotypes
- Specifying component architectural styles
  - ✓ Components, ports, connectors and assemblies
  - ✓ Static assemblies
  - ✓ Dynamic assemblies
- Realizing component architectural styles
  - ✓ The CORBA component model
- Conclusion

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

Each Component implements many Types

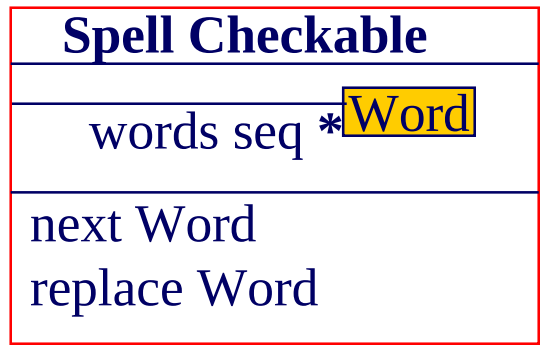


- ✓Components offer different interfaces to each other
- ✓Each interface has a different supporting model
- ✓The implementation **refines** each interface spec

Component Implementation

```
class EditorCoreClass implements SpellCheckable, Layoutable {

        // store a tree of all elements — graphics, words, characters, tables, etc.
        private ElementTree contents;

        // this iterator traverses all elements in the contents tree in depth-first order
        private DepthFirstIterator elementIterator = contents.root();

        // this iterator only visits Word elements, ignoring others
        private WordIterator spellPosition = new WordFilter (elementIterator);

        // return the "next" word in the tree
        public Word nextWord () {
                Word w = spellPosition.word();
                spellPosition.next();
                return w;
        }

        // replace the last visited word with the replacement
        public void replaceWord (Word replacement) {
                spellPostion.replaceBefore (replacement);
        }
}
```

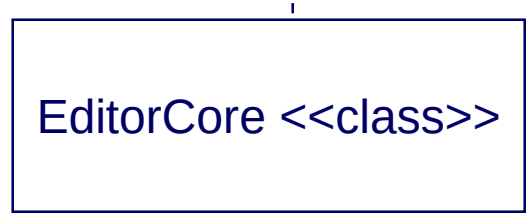
Class *refines* Interface

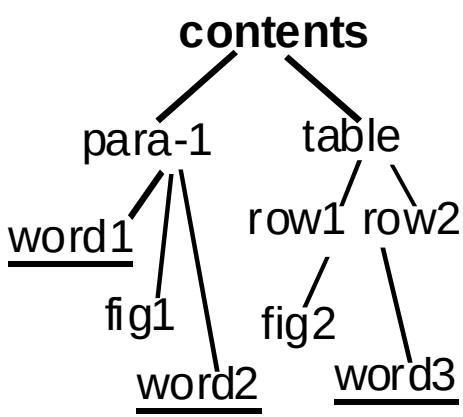


"abstraction function" or "retrieval"

```
words =
  contents.asDepthFirstSequence -> select (e | e.isKindOf (Word))
```

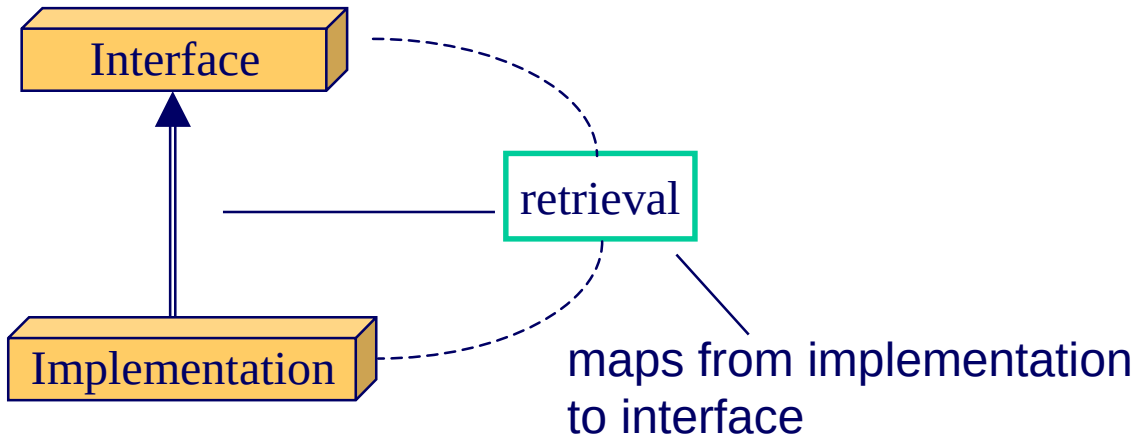
refinement





- Abstraction function "retrieves" abstract model attributes

Implementation: Refinement + Retrieval



- A class implements an interface
  - ✓ The interface defines its own model and behavior spec
  - ✓ The class selects its own data and code implementation
- The class is a *refinement* of the type
  - ✓ 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**
- One abstraction function for each specification attribute
  - ✓ Specification types must be implemented as adapter for this test harness
  - ✓ The pre/post conditions can then be executed as part of the code

The Power of Refinement

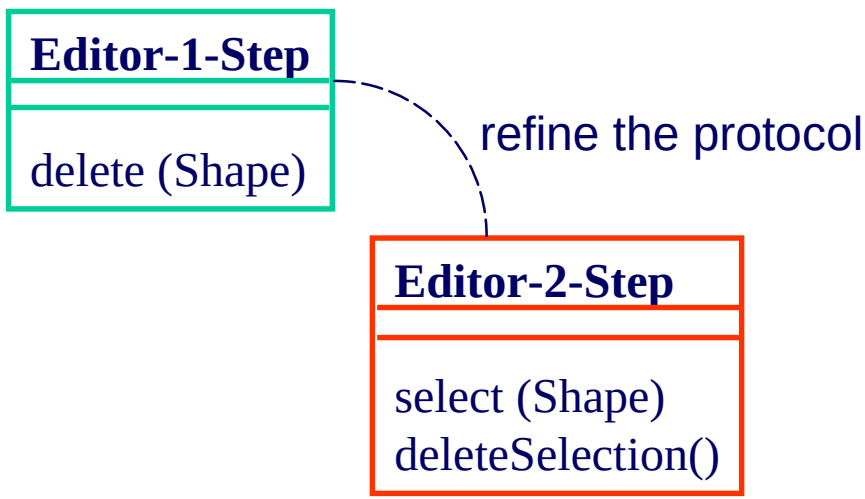
- The notion of refinement and retrieval is very useful
  - ✓ It permits flexible *mapping* between from realization to specification
  - ✓ It solves a very real problem:
    - ✓ *"I have just made some change to my code. Do I have to update my design models? Do I have to update my analysis models?"*

- Pick abstract model to conveniently express client spec
  - ✓ Implementation model must have correct *mapping* to the abstract
  - ✓ Encapsulates implementation without hiding specified behavior

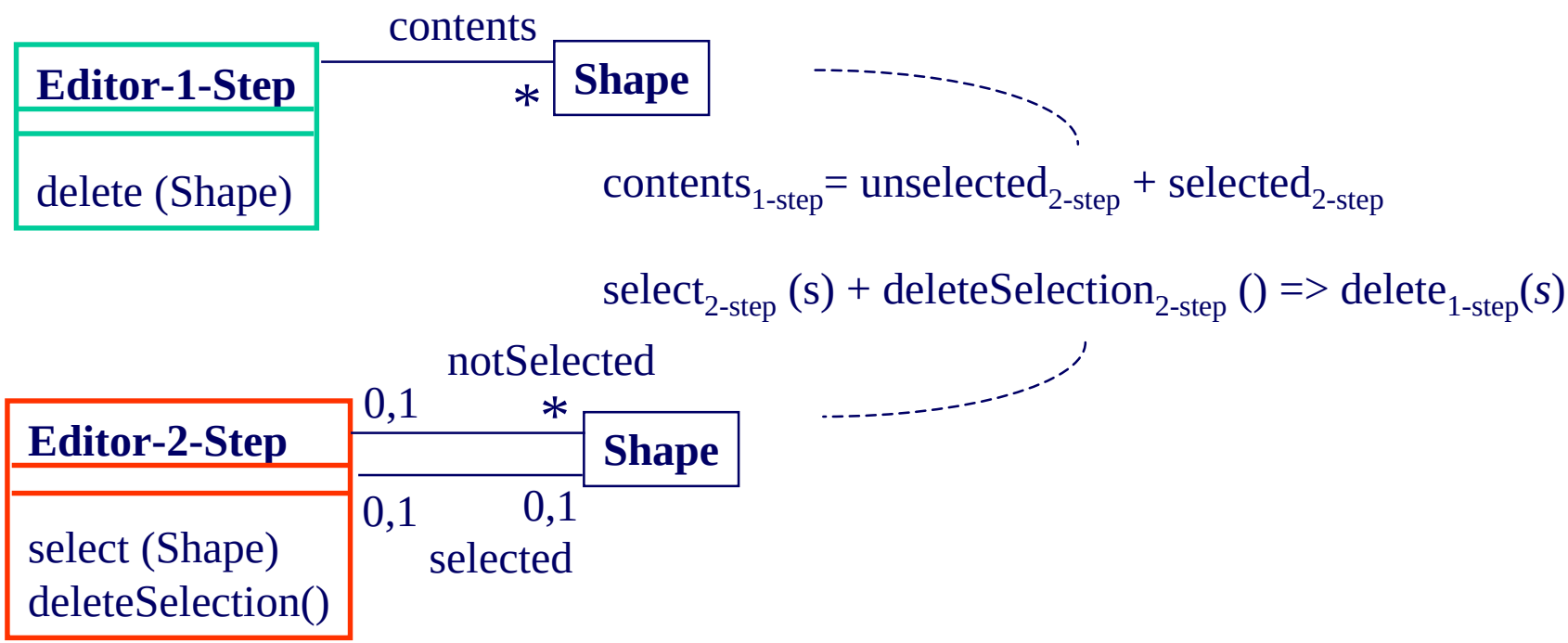
- Even more powerful with *temporal (action) refinement*
  - ✓ The abstract level describes an abstract action
  - ✓ The concrete level details an interaction sequence
  - ✓ The retrieval establishes the mapping between the two

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 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

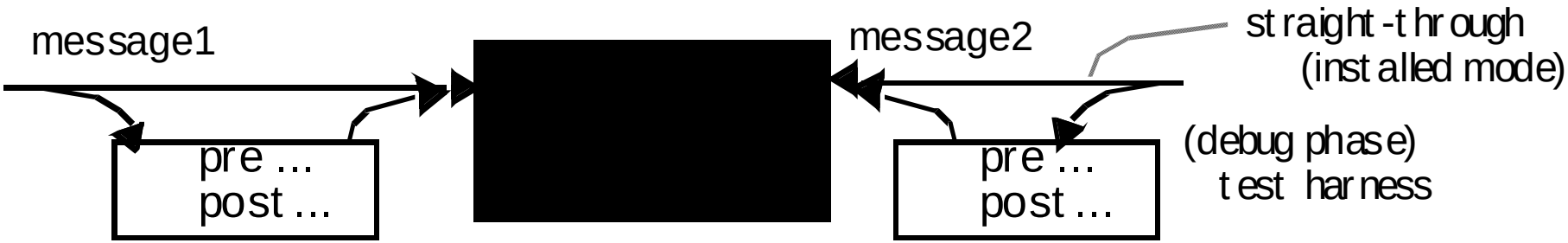
Validity Rules

- The existence of a retrieval function or a state model does not guarantee a valid refinement (although it is a great start).The following conditions must also be checked:

- Subtype
  - ✓ whenever an abstract operation is applicable, so is its realization
  - ✓ for any initial state, the retrieval of the final realization state after the realization operation occurs satisfies the postcondition of the specification

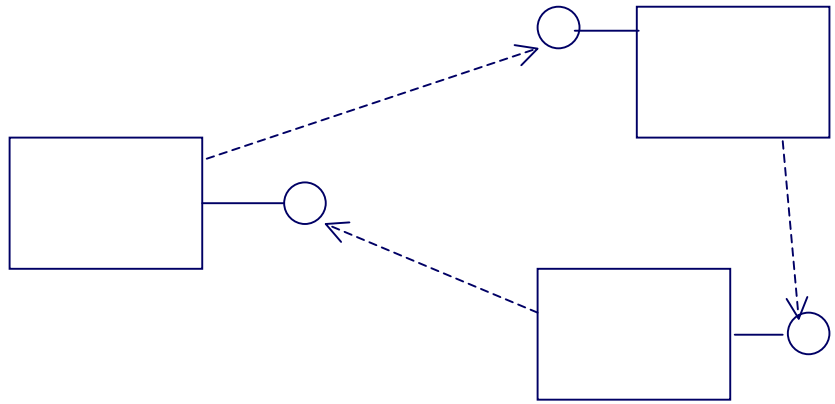
- Collaboration Refinement
  - ✓ whenever the abstract action is applicable, there is a path through the state model (sequence of realization actions) from the initial state to the final state such that the effect of the realization action sequence satisfies the specification action postcondition

Refinement kind determines Test harness



- Different test strategies for different refinement kinds
  - ✓ pre/post instrumentation
  - ✓ sequence of actions, covering success and exceptions paths
  - ✓ hardware test structures with mapping to deployment hardware
- Link to systematic test generation

Section Summary - Components

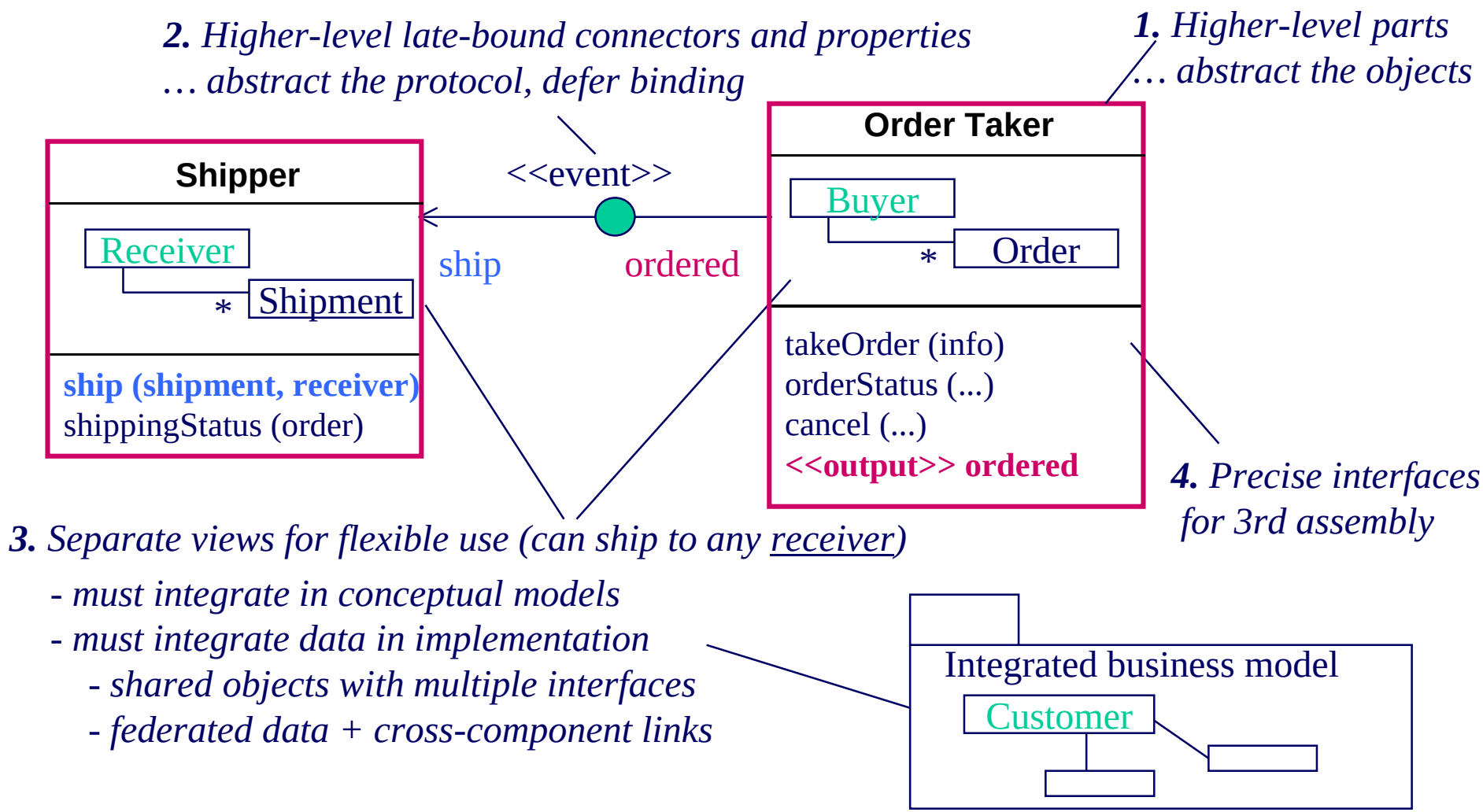


- Interface centric, collaboration patterns
- Symmetrical, precise, black box views
- Refinement - separate interface from implementation

Outline

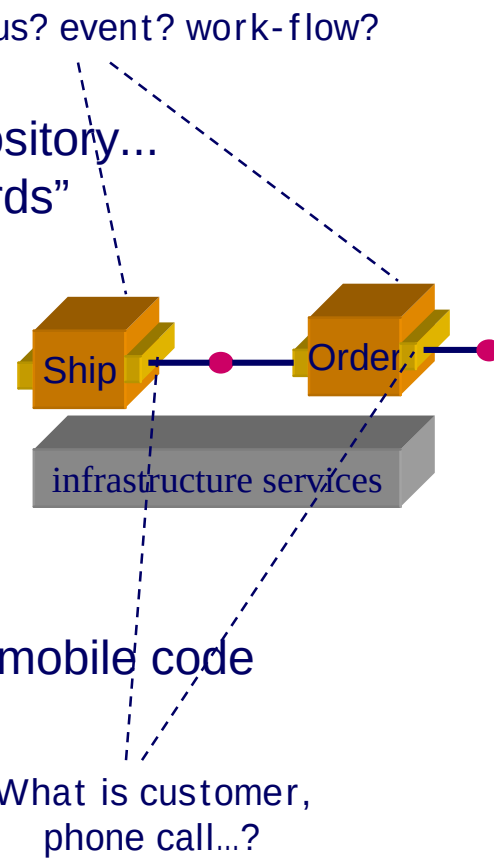
- Precise component specifications in UML
  - ✓ Interfaces
- Refinement
  - ✓ Component implementation refines spec
- **What is Software Architecture?**
  - ✓ **Common definitions and examples**
  - ✓ Catalysis definition of architecture
  - ✓ Specifying architectural styles using OCL
  - ✓ Generating architectural styles using Frameworks
  - ✓ Specifying architectural styles with Stereotypes
- Specifying component architectural styles
  - ✓ Components, ports, connectors and assemblies
  - ✓ Static assemblies
  - ✓ Dynamic assemblies
- Realizing component architectural styles
  - ✓ The CORBA component model
- Conclusion

Business Components: Modeling Issues

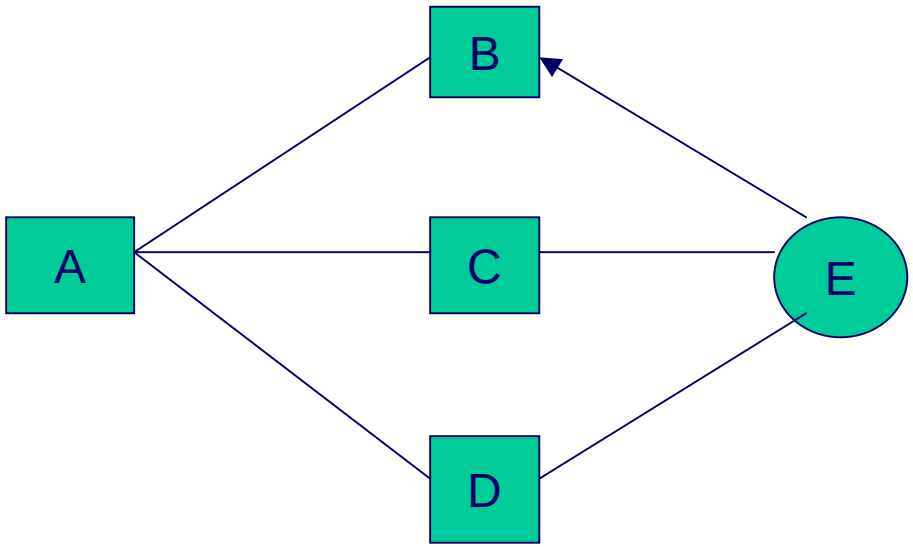


Components demand Architecture Standards

- For separately built components to work together they must share…
- Standard “horizontal” infrastructure services and interfaces
  - ✓ transactions, security, errors, directory, interface repository...
  - ✓ OMG, Microsoft rapidly defining many global “standards”
- Standard “vertical” models of domain concepts
  - ✓ What is a “Customer”, “Phone Call”, “Order”, etc.
    - ✓ components must talk the same “domain language”
  - ✓ OMG defines “Vertical” architectures standards as well
- Standard “connector” mechanisms between components
  - ✓ Synchronous / asynchronous message, event, workflow, mobile code
  - ✓ Location transparency: CORBA, DCOM



What Architecture Isn't



A neat drawing of boxes, circles, lines etc. laid out nicely in PowerPoint is NOT an architecture

Are A-E objects, modules, libraries or processes?
Are A-D similar, E different?
What do lines between blocks mean: interprocess communication, data flow?

How would you determine if an implemented system conformed to this architecture?

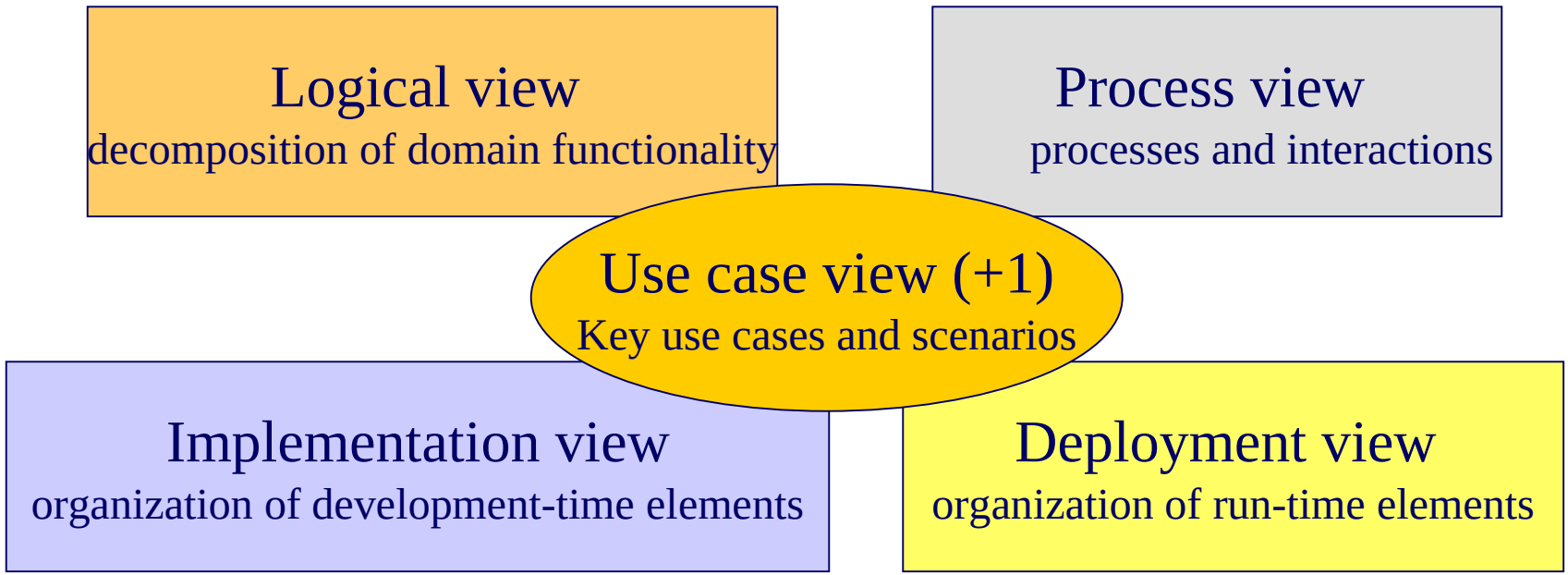
A Conventional Definition of Architecture

- The architecture of a system consists of
  - ✓ the structure of its parts
    - ✓ (includes design-time and run-time, hardware and software)
  - ✓ the nature and relevant externally visible properties of those parts
    - ✓ (modules with interfaces, hardware units, objects)
  - ✓ the relationships and constraints between them

Ivar Jacobsen (Component Strategies 1999)

"Architecture is about everything, but it isn't everything."

4+1 Views of Architecture (Kruchten 1996)



IEEE Architecture Definition (1999) is a generalization

above views can be further subdivided e.g. calls, uses, physical etc.

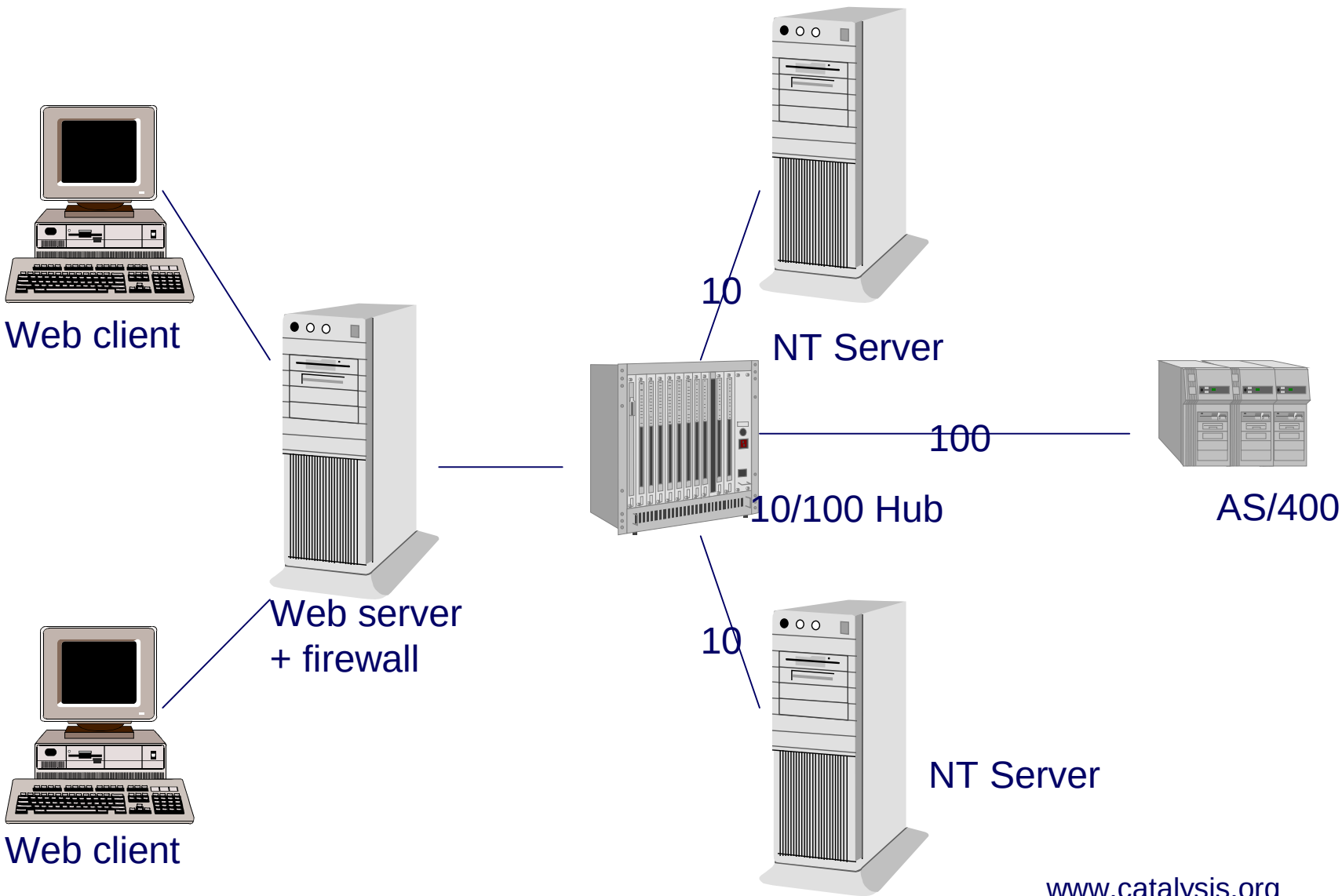
Examples of “Architecture”

- Coding rules
- Hardware (physical) architecture
- UI style
  - ✓ e.g. navigate a 1-* association as a master-slave list box
- Java Beans
- Design patterns
  - ✓ e.g. all object creation via factories
- Persistence approach

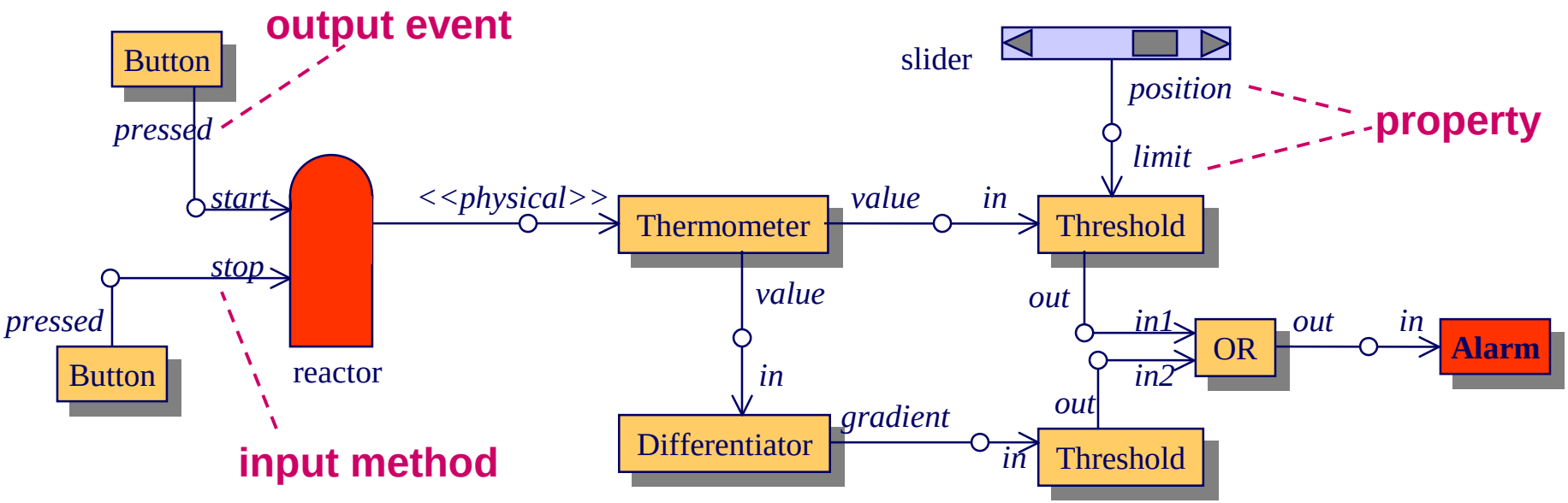
Coding rules

- e.g. canonical form in C++
  - ✓ every class must define
    - ✓ a parameter-less constructor
    - ✓ a copy constructor
    - ✓ an assignment operator
    - ✓ a (virtual) destructor
- ban gotos

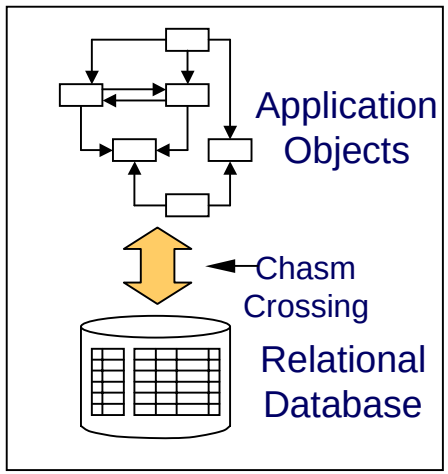
Hardware architecture



Java Beans example



A static assembly of beans



RDBMS Mapping Outline

- OOD class
- Object
- Object-ID
- *Primitive* attribute
- Object attribute (single-valued)
- Set-valued attribute
- Many-to-many association
- Polymorphism
- Subtype

- Table
- Row in table
- Primary key
- Column in row
- Foreign key
- Reverse foreign key + join
- Associative table (normalize)
- No counterpart
- Many options
  - ✓ Collapse to 1 table
  - ✓ Multiple tables + joins

Section Summary - Architecture

- … limiting needless creativity
- Wide range:

  0% (cowboy)……. 100% (compiler)
- Architecture defines / uses specific constructs, language, patterns, rules
- Architecture definition shared across projects

Outline

- Precise component specifications in UML
  - ✓ Interfaces
- Refinement
  - ✓ Component implementation refines spec
- **What is Software Architecture?**
  - ✓ Common definitions and examples
  - ✓ **Catalysis definition of architecture**
  - ✓ Specifying architectural styles using OCL
  - ✓ Generating architectural styles using Frameworks
  - ✓ Specifying architectural styles with Stereotypes
- Specifying component architectural styles
  - ✓ Components, ports, connectors and assemblies
  - ✓ Static assemblies
  - ✓ Dynamic assemblies
- Realizing component architectural styles
  - ✓ The CORBA component model
- Conclusion

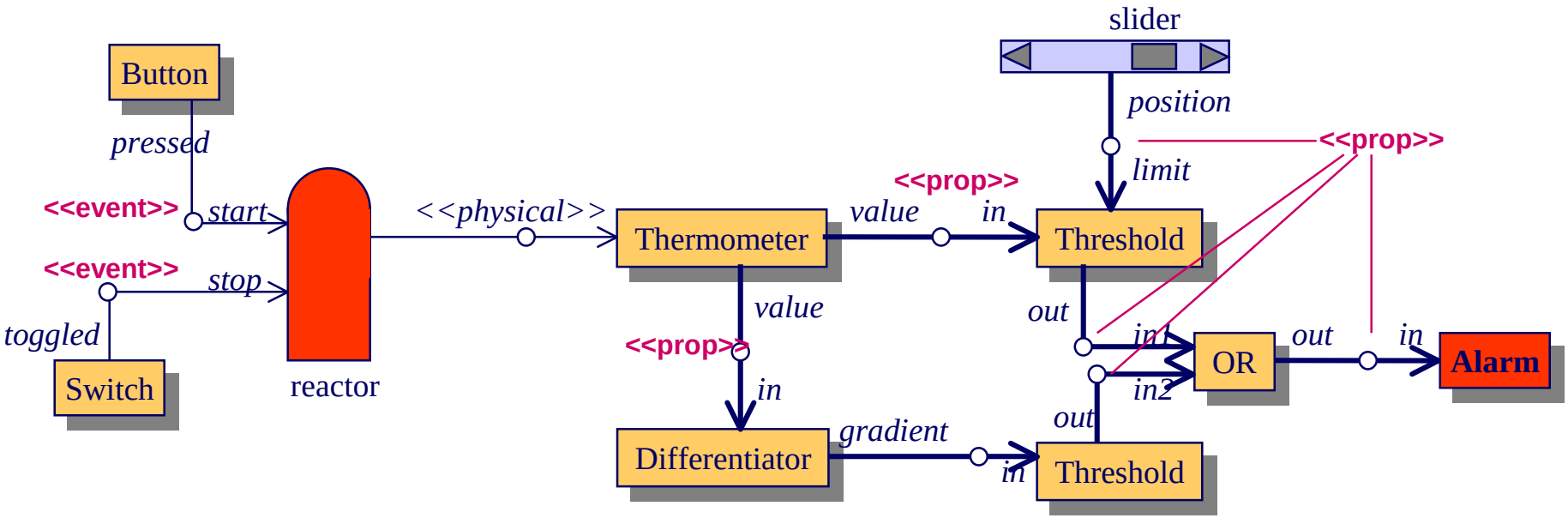
What is the intent of “Architecture”

The set of design decisions, rules, or patterns about any system that keep its designers from exercising **needless creativity**

- Desmond D’Souza

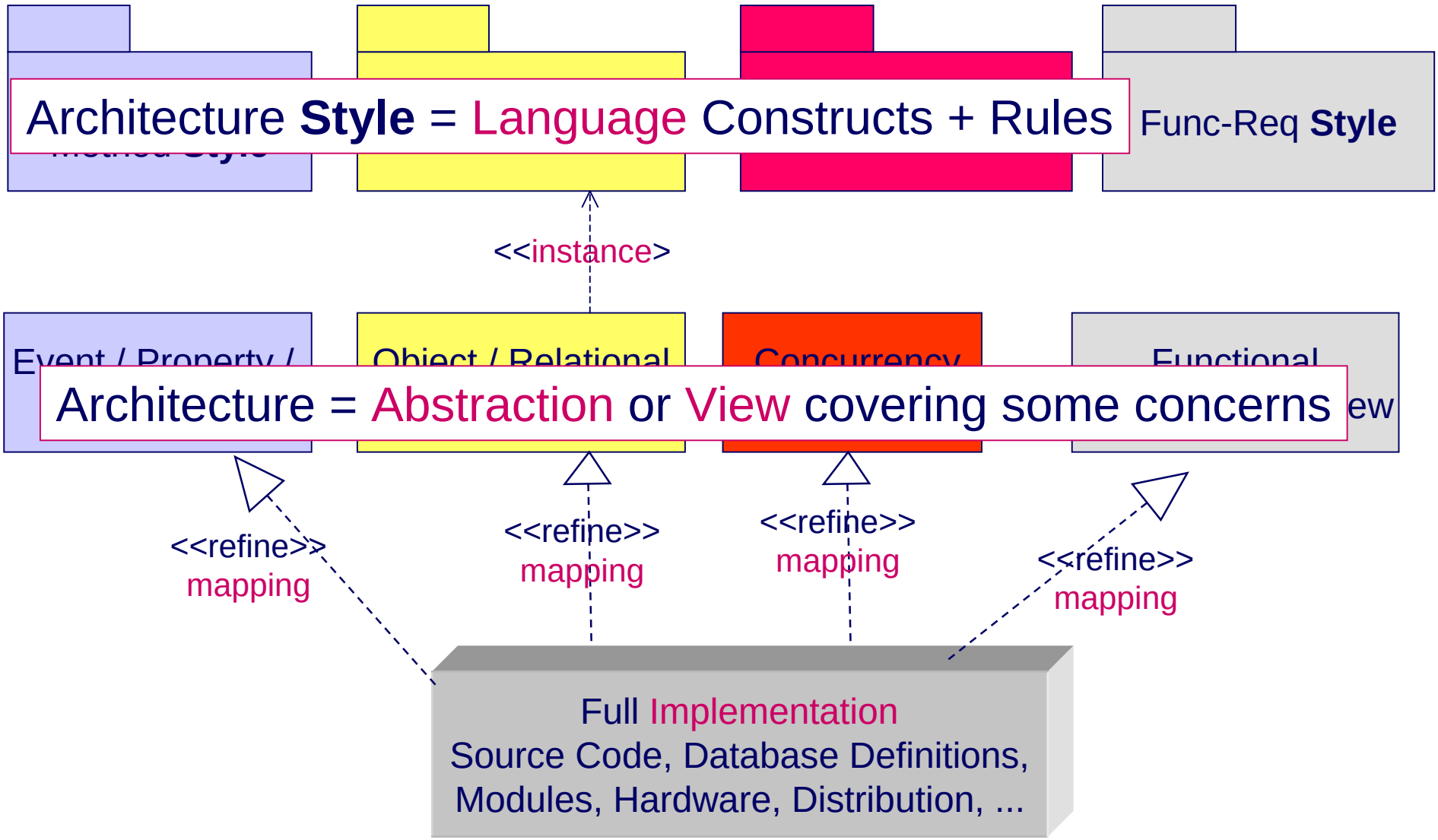
- It is not **just** about any specific size, scale, domain, or infrastructure
- Can range from “*3-tier C/S*” to “*use Corba OTS*” to “*get/set method names*”
- Includes ways of defining, structuring, packaging, and sharing services
- Should often be made precise using an appropriate formalism

Is This an “Architecture”?

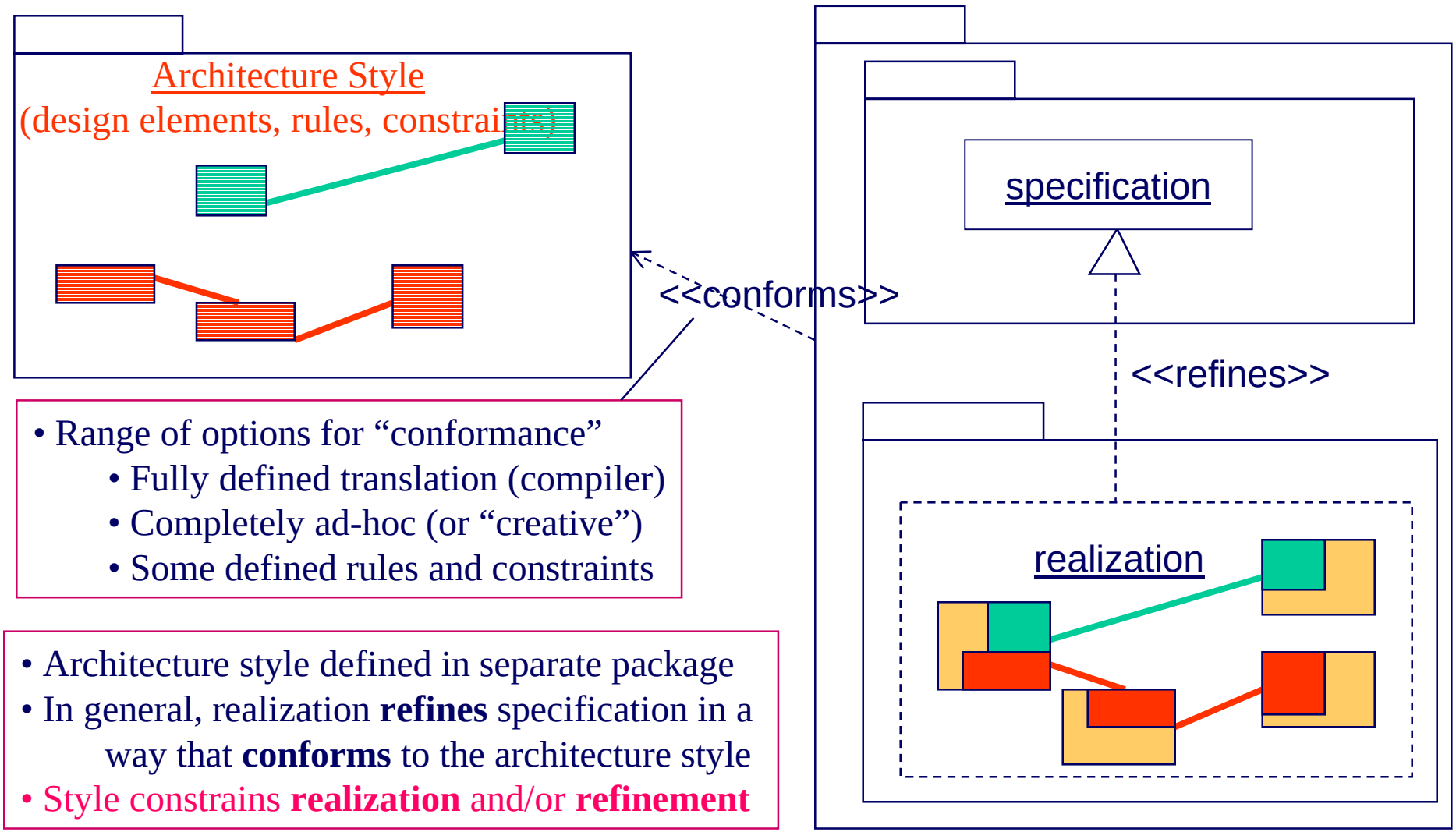


- This is an abstract view of the implementation
  - ✓ it uses the language of properties, events, methods
    - ✓ … and of connectors between these “connection points”
  - ✓ it maps to corresponding patterns in the Java code
- This is an instance of the Java Beans style: design + code
- This cannot be done (effectively, cleanly, precisely, …) in UML

Architecture as **View** based on **Style**



Architecture and “Style” in Catalysis



- Range of options for “conformance”
  - Fully defined translation (compiler)
  - Completely ad-hoc (or “creative”)
  - Some defined rules and constraints

- Architecture style defined in separate package
- In general, realization **refines** specification in a way that **conforms** to the architecture style
- Style constrains **realization** and/or **refinement**

“Realization” Architectural Style

- A **realization architectural style** defines a set of realizations
  - ✓ Just as an object type defines a set of objects
  - ✓ Think of it as **architectural type**, similar to an object type

- The aspects of a realization that are relevant to deciding whether or not it conforms to a style is called its **architecture**

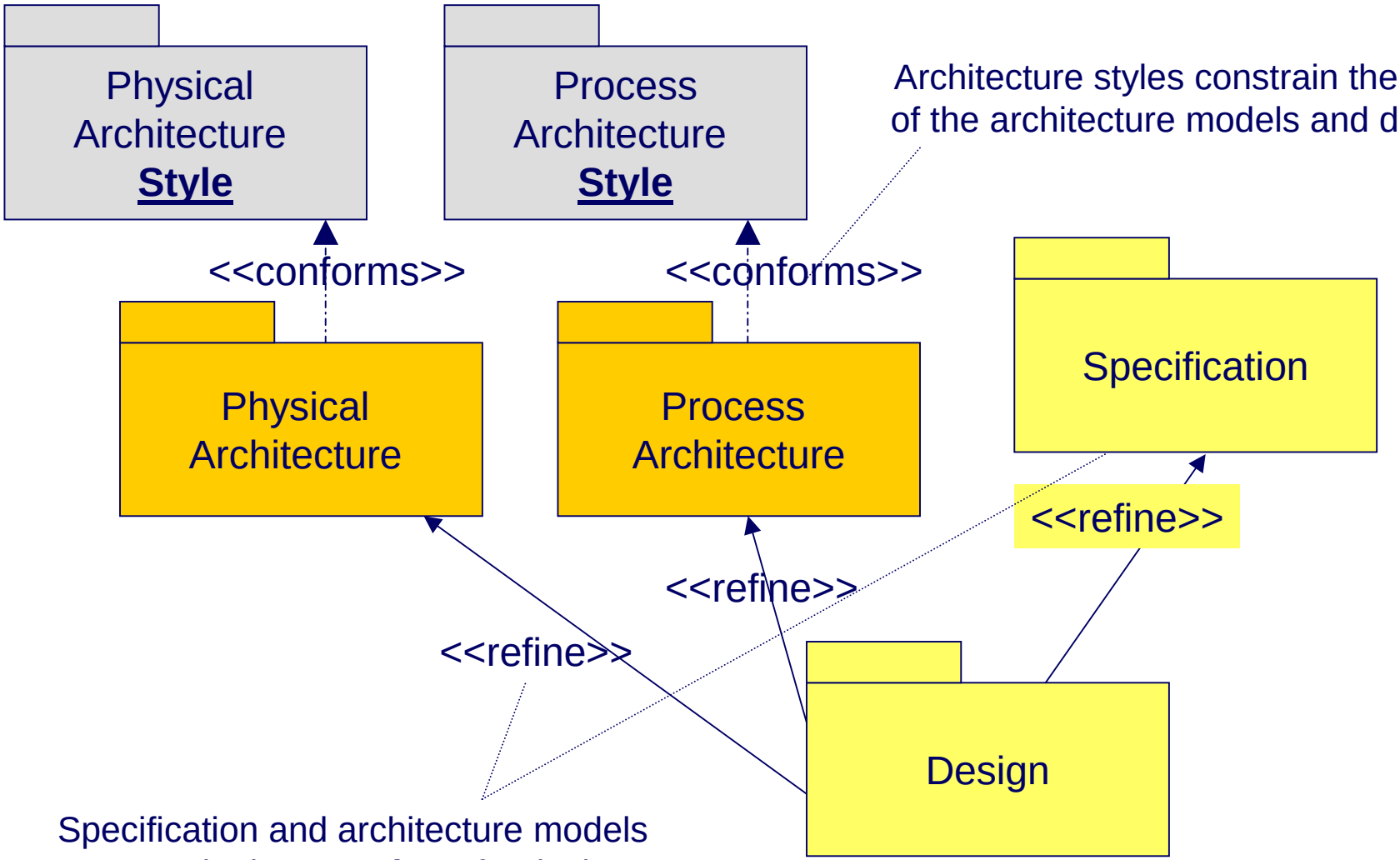
  - ✓ e.g. style constraint = upper bound on McCabe complexity implies
    architecture = flowgraph

  - ✓ style constraint = Law of Demeter implies
    architecture = callgraph

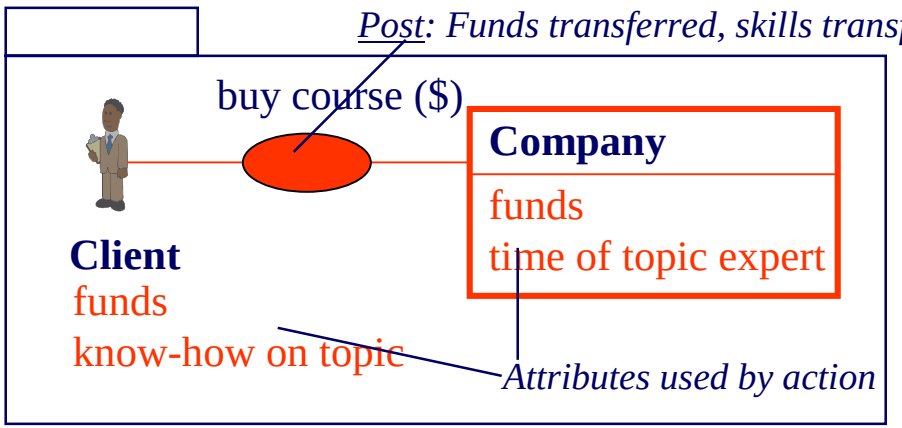
- An architecture **conforms** to a style if it is a valid “instance of” that style

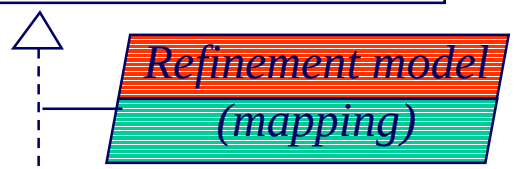
- A style is documented as a collection of (possibly formal) constraints.

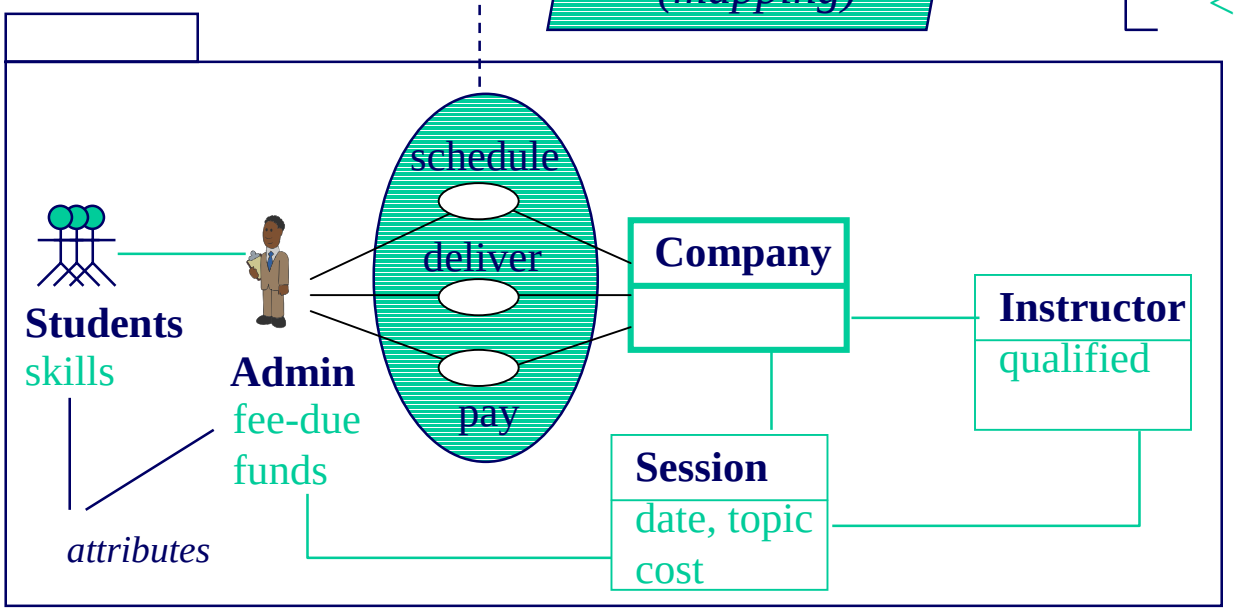
Conform vs. Refine : Form vs. Meaning



Checking refinement vs. conformance

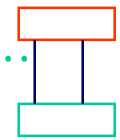


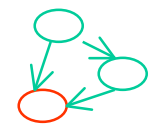




Refinement model, text or diagrams

Time of topic expert =
...instructor, qual, session…
Client know-how =
...students, skills
$ = …session, cost, ...
buy course =
<schedule + deliver + pay>



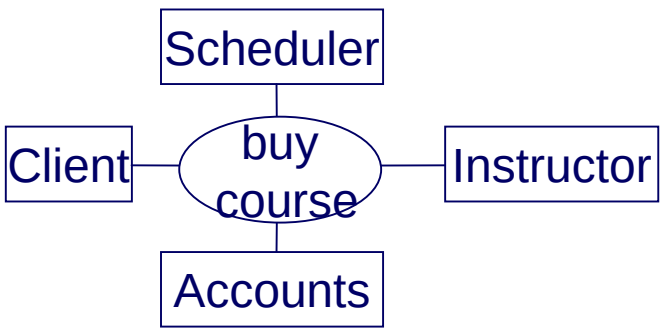


Checking refinement is independent of checking conformance to a "**realization**" style

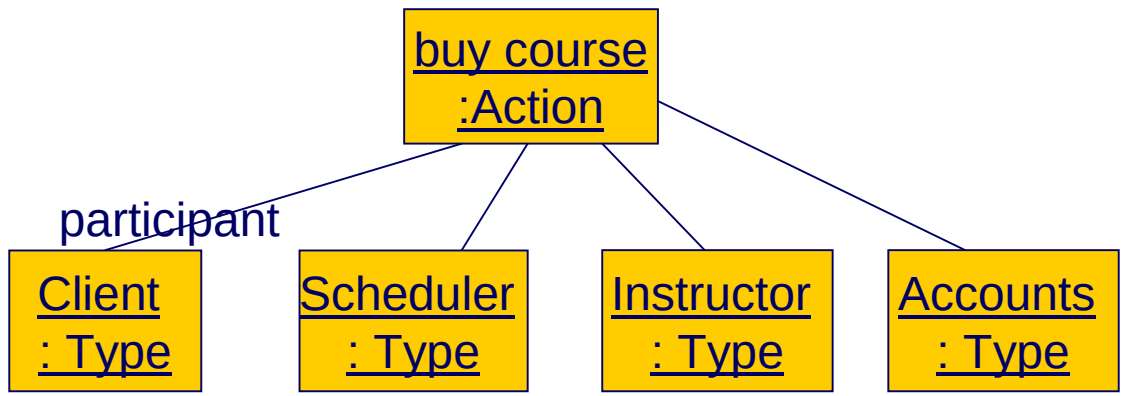
e.g. *"all actions have at most 3 participants and are specified in OCL consistent with the attribute model"*

Conformance Justification

- Validity of design to architecture
  - ✓ design refines architecture
- Validity of architecture to architecture style
  - ✓ check that description (model snapshot) is “instance of” style
  - ✓ much easier than checking refinement (automatable?)



Model



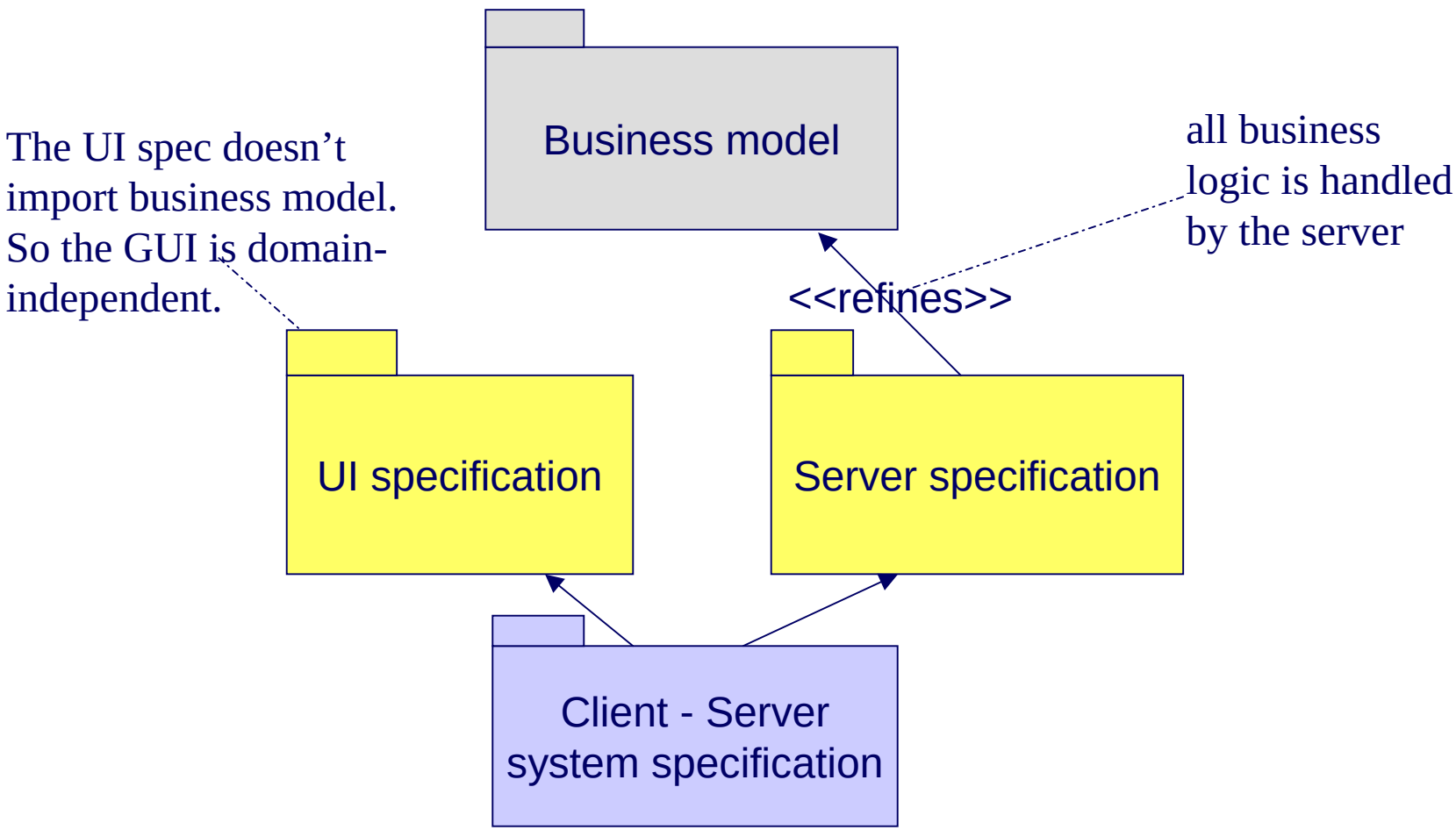
Model as a meta-level snapshot

Constraint: “No action may have more than three participant types”

Examples

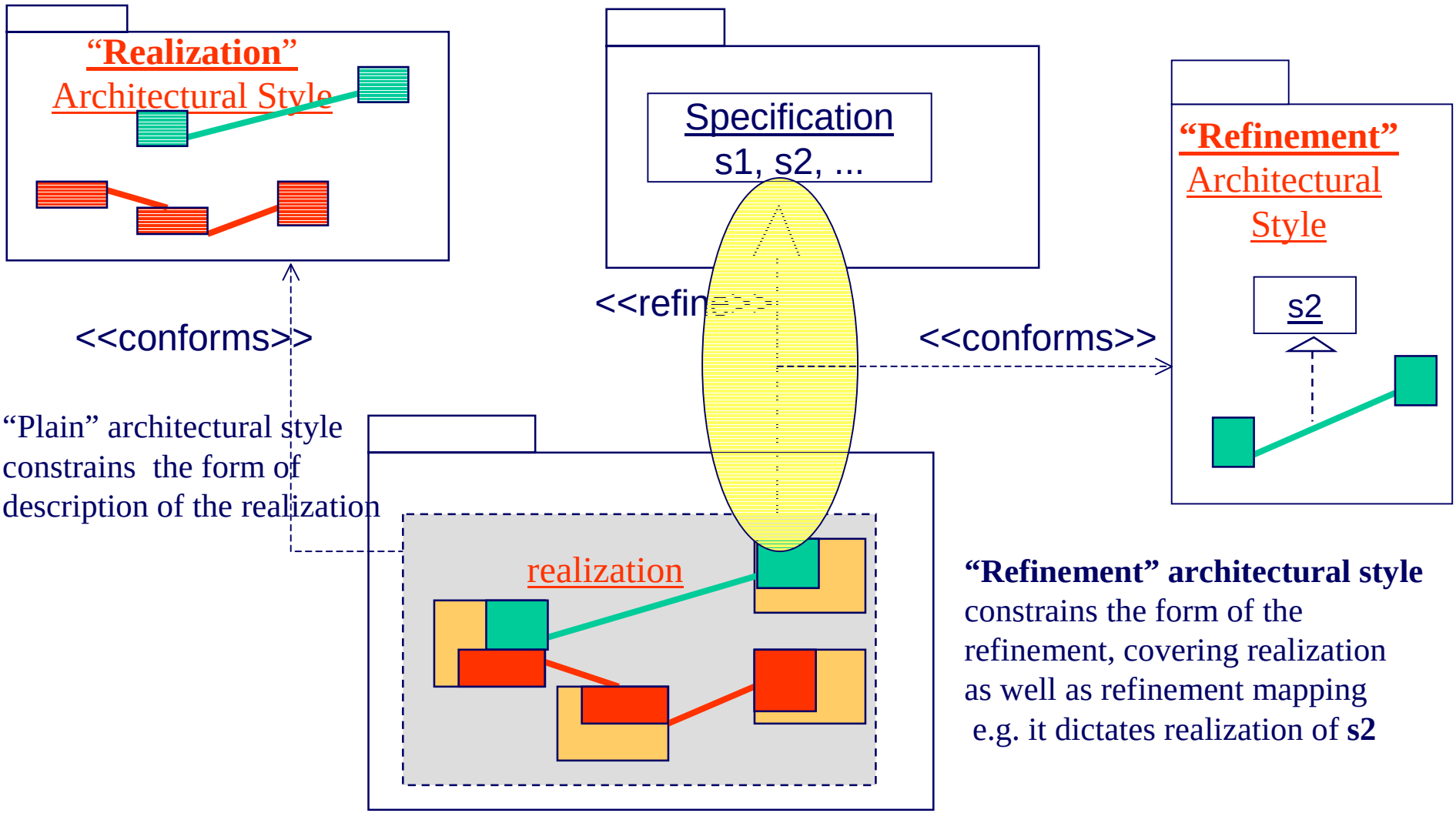
- Coding rules
  - ✓ realization is code
  - ✓ rules are defined in a package that imports the language definition package, can be formalized as OCL “meta” constraints
- Physical architecture
  - ✓ realization is deployed system
  - ✓ rules are constraints on nodes and their properties
- GOF design patterns and UI architecture can both be formalized using frameworks
- Java Beans style (ports and connectors) is an example of a component architecture style and is again defined using frameworks

Tiered Architectures



- 3-tier, 4-tier are similar

“Refinement” Architectural Style



“Plain” architectural style constrains the form of description of the realization

“Refinement” architectural style constrains the form of the refinement, covering realization as well as refinement mapping e.g. it dictates realization of **s2**

Refinement Vs. Realization Styles

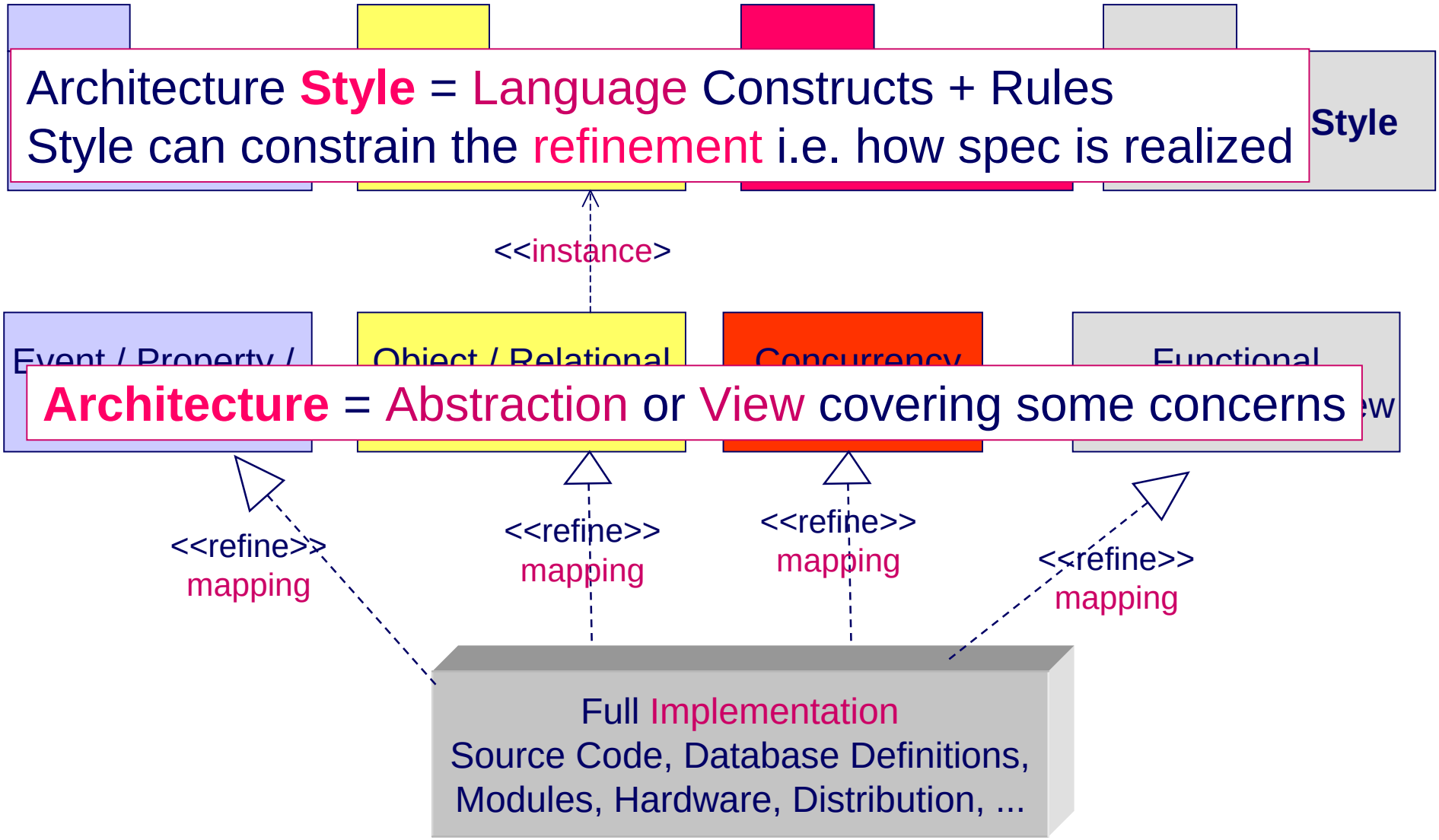
New

- Refinement Architectural Style:
  - ✓ "Anyplace 2 attributes have to be in sync, use…"
    - ✓ Style 1: "… 2 copies with update protocol"
    - ✓ Style 2: "… 1 copy in shared memory"
    - ✓ Style 3: "… 1 copy and query for the other"

- Realization Architectural Style
  - ✓ "2 copies with update" is a pre-defined construct

- Realization style is just a special case of refinement style

Section Summary - Architecture in Catalysis



Outline

- Precise component specifications in UML
  - ✓ Interfaces
- Refinement
  - ✓ Component implementation refines spec
- **What is Software Architecture?**
  - ✓ Common definitions and examples
  - ✓ Catalysis definition of architecture
  - ✓ **Specifying architectural styles using OCL**
  - ✓ Generating architectural styles using Frameworks
  - ✓ Specifying architectural styles with Stereotypes
- Specifying component architectural styles
  - ✓ Components, ports, connectors and assemblies
  - ✓ Static assemblies
  - ✓ Dynamic assemblies
- Realizing component architectural styles
  - ✓ The CORBA component model
- Conclusion

Formalization of Examples in OCL

- multiple inheritance is not permitted
  OclType.allInstances->forAll(T | T.supertypes->size<=1)

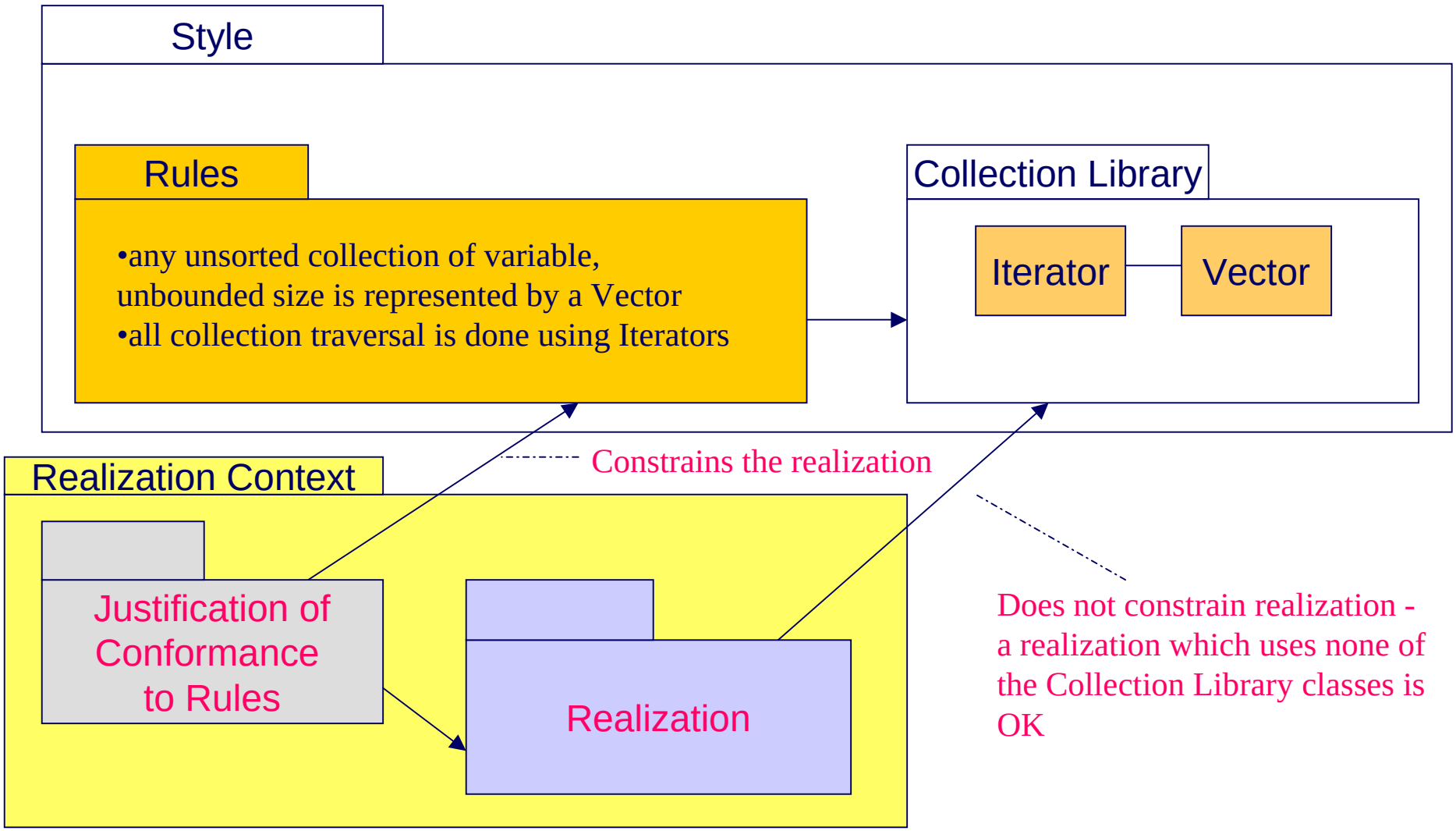
- no type should introduce more than 20 new operations
  OclType::**inv**
  (operations-supertypes.operations)->size<=20

- no operation should have more than 5 parameters
  Operation::**inv** parameters->size<=5

- multiple inheritance OK, unless feature name clashes
  OclType::**inv**
  supertypes.operations = supertypes.operations->asSet

- Law of Demeter

Elements + rules



Outline

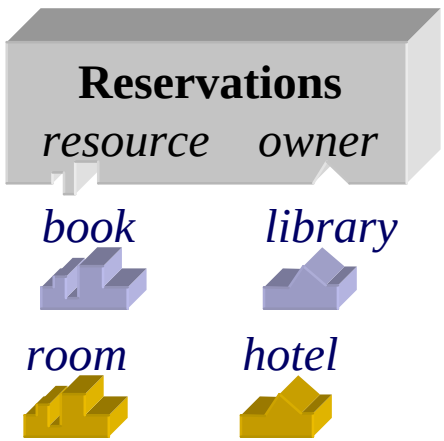
- Precise component specifications in UML
  - ✓ Interfaces
- Refinement
  - ✓ Component implementation refines spec
- **What is Software Architecture?**
  - ✓ Common definitions and examples
  - ✓ Catalysis definition of architecture
  - ✓ Specifying architectural styles using OCL
  - ✓ **Generative architectural styles using Frameworks**
  - ✓ Specifying architectural styles with Stereotypes
- Specifying component architectural styles
  - ✓ Components, ports, connectors and assemblies
  - ✓ Static assemblies
  - ✓ Dynamic assemblies
- Realizing component architectural styles
  - ✓ The CORBA component model
- Conclusion

Frameworks

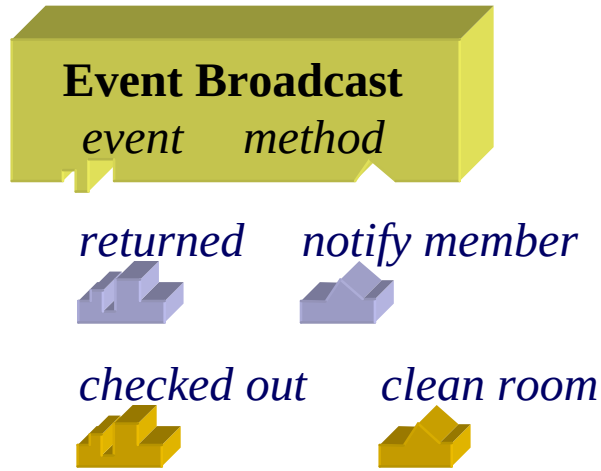
- generic models with placeholders
- generate a realization that conforms to an architectural style
- a collection of frameworks defines an architectural style generatively
- a realization conforms to the architectural style if it can be derived by applying the frameworks in the style to the specification model.
- No need to check if realization conforms to architectural style.

Framework Concept at All Levels

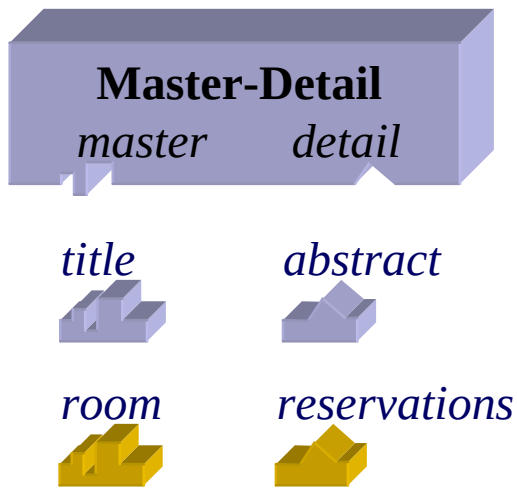
Business Framework



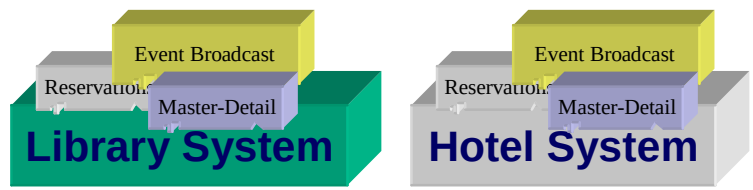
Technical Framework



UI Framework



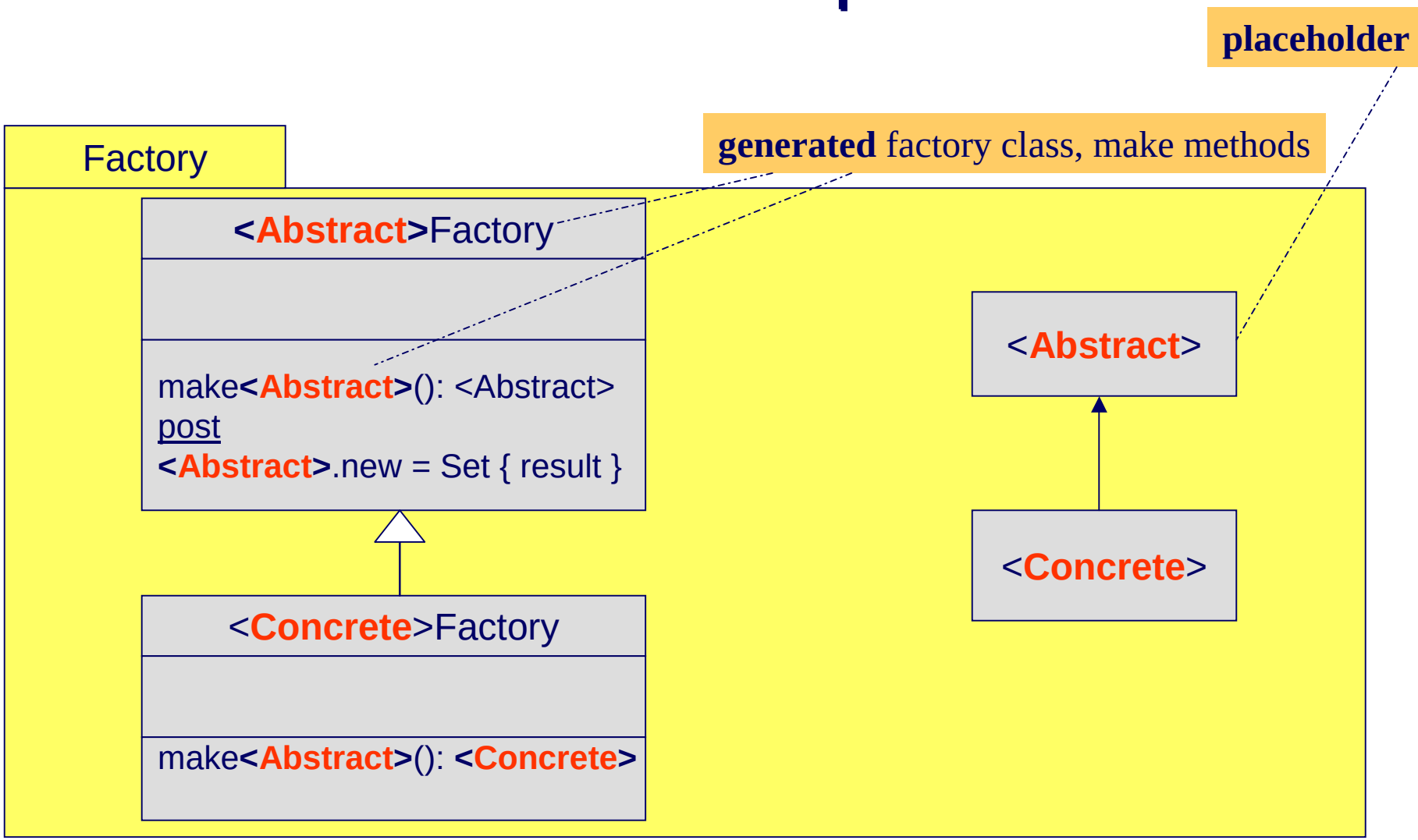
Multiple frameworks used in any app



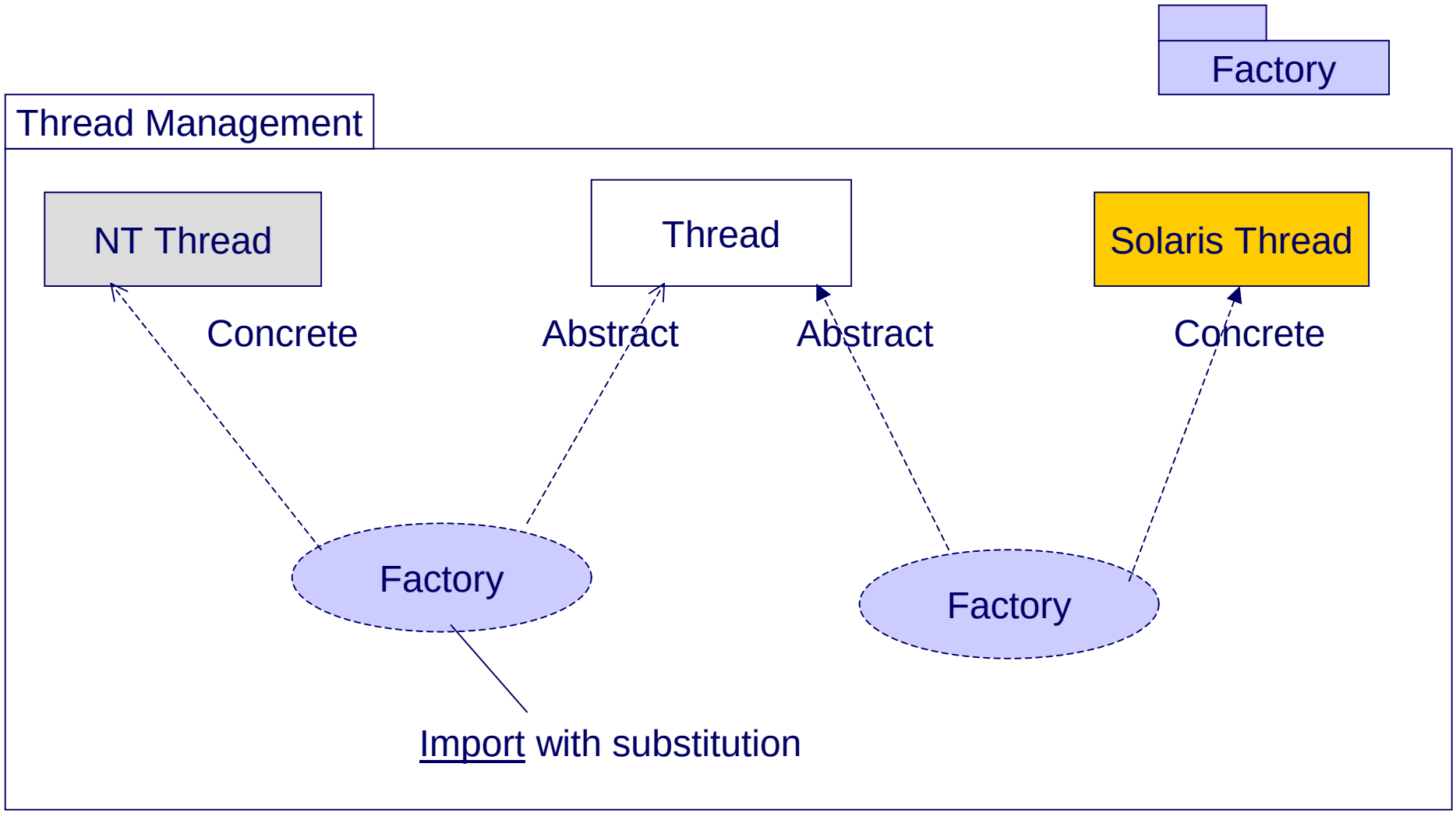
Frameworks Generate Architectures

- E.g. object-relational mapping tells you how to design DB, not just how to check that DB conforms to object model
  - ✓ can also **generate** object model from existing relational design (e.g. legacy DB)
- Multiple inheritance disallowed is **not generative**
  - ✓ but a constructive transformation rule that maps multiple inheritance to delegation, forwarding and implementing multiple interfaces is generative

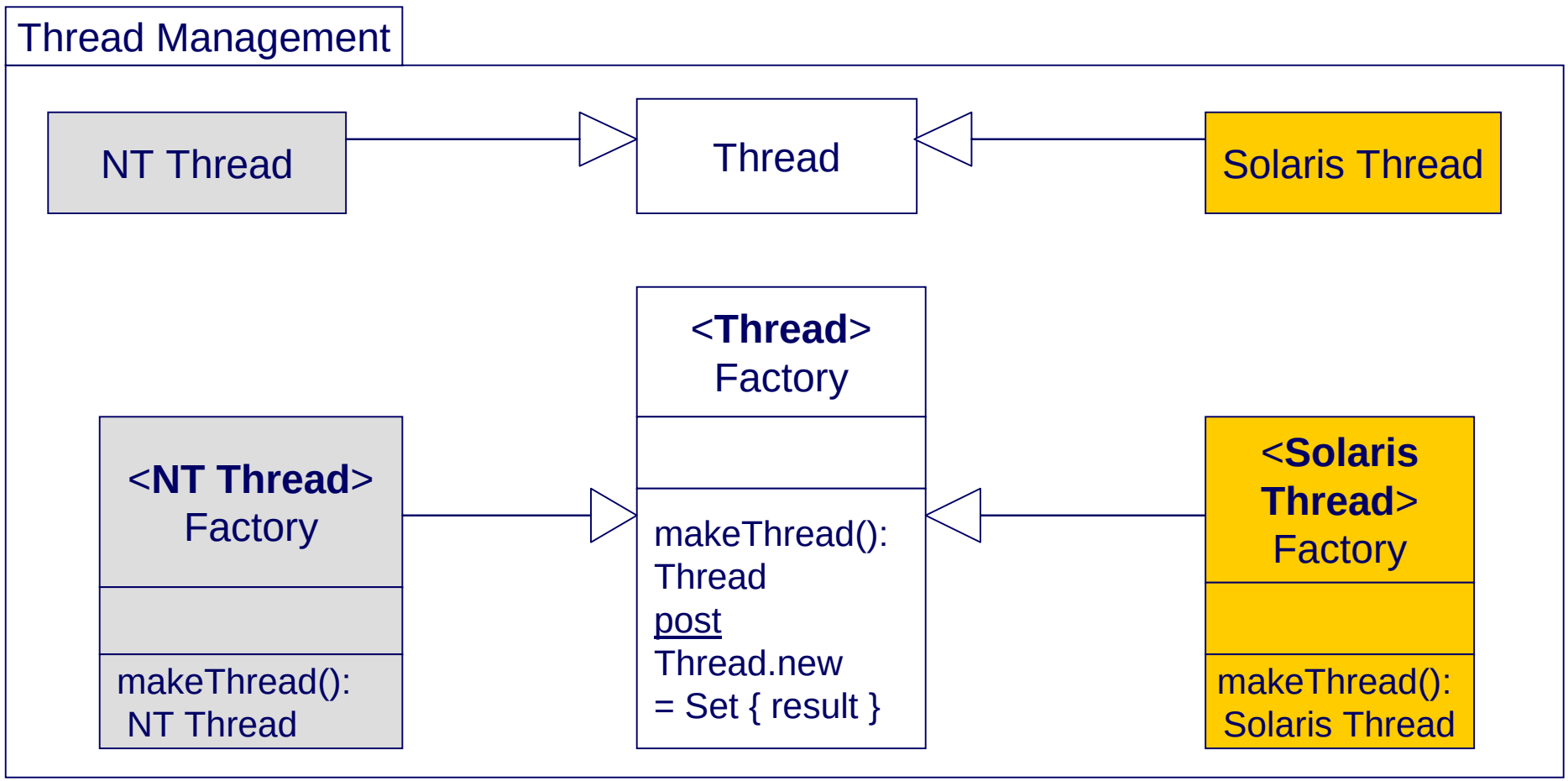
Factory Pattern : Generative Aspects



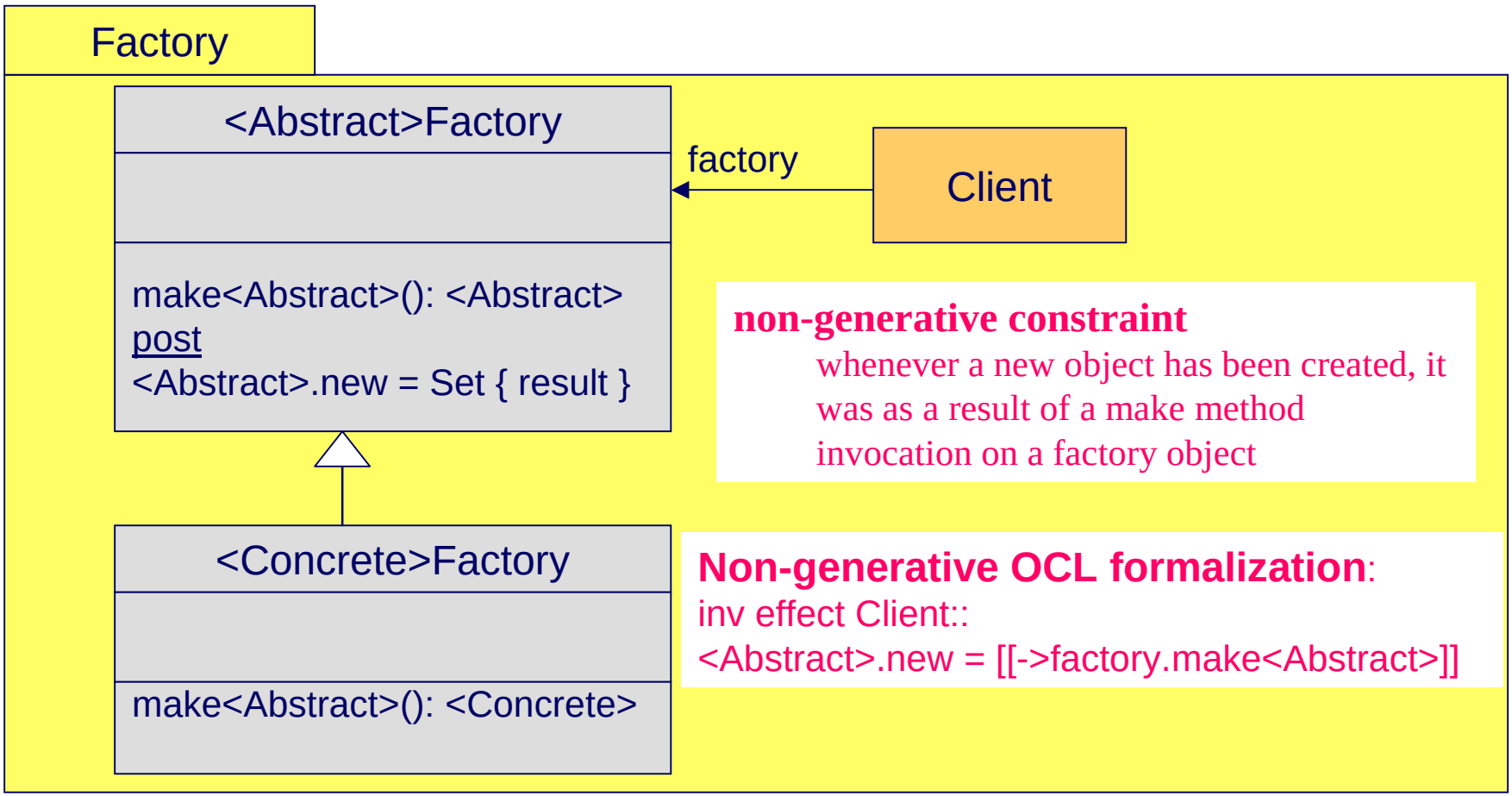
Applying Factory Framework



The Model is Automatically Generated

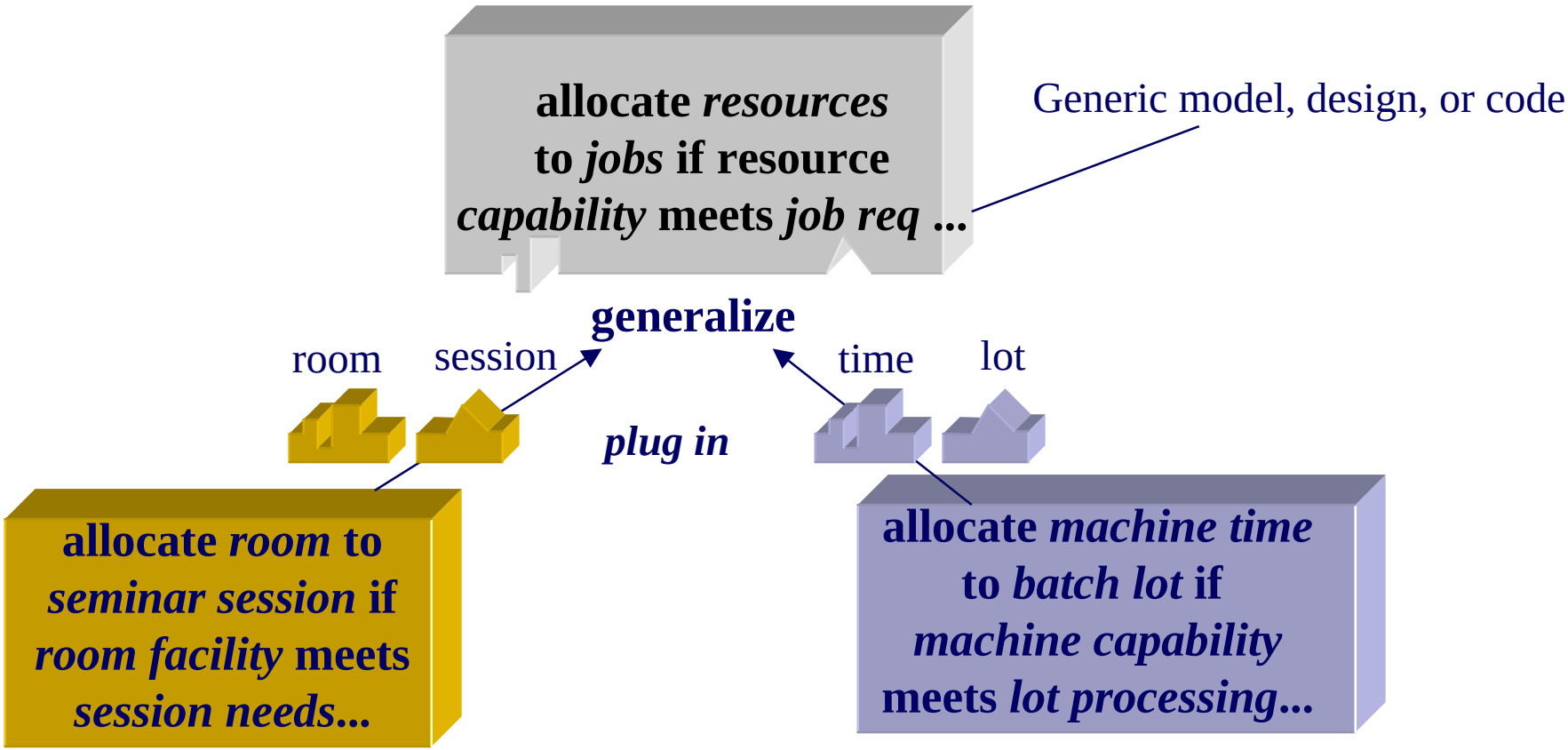


Factory Pattern : Non-Generative Aspects



Constraint can be enforced by translating every **new** <Concrete> into factory.make<Abstract>

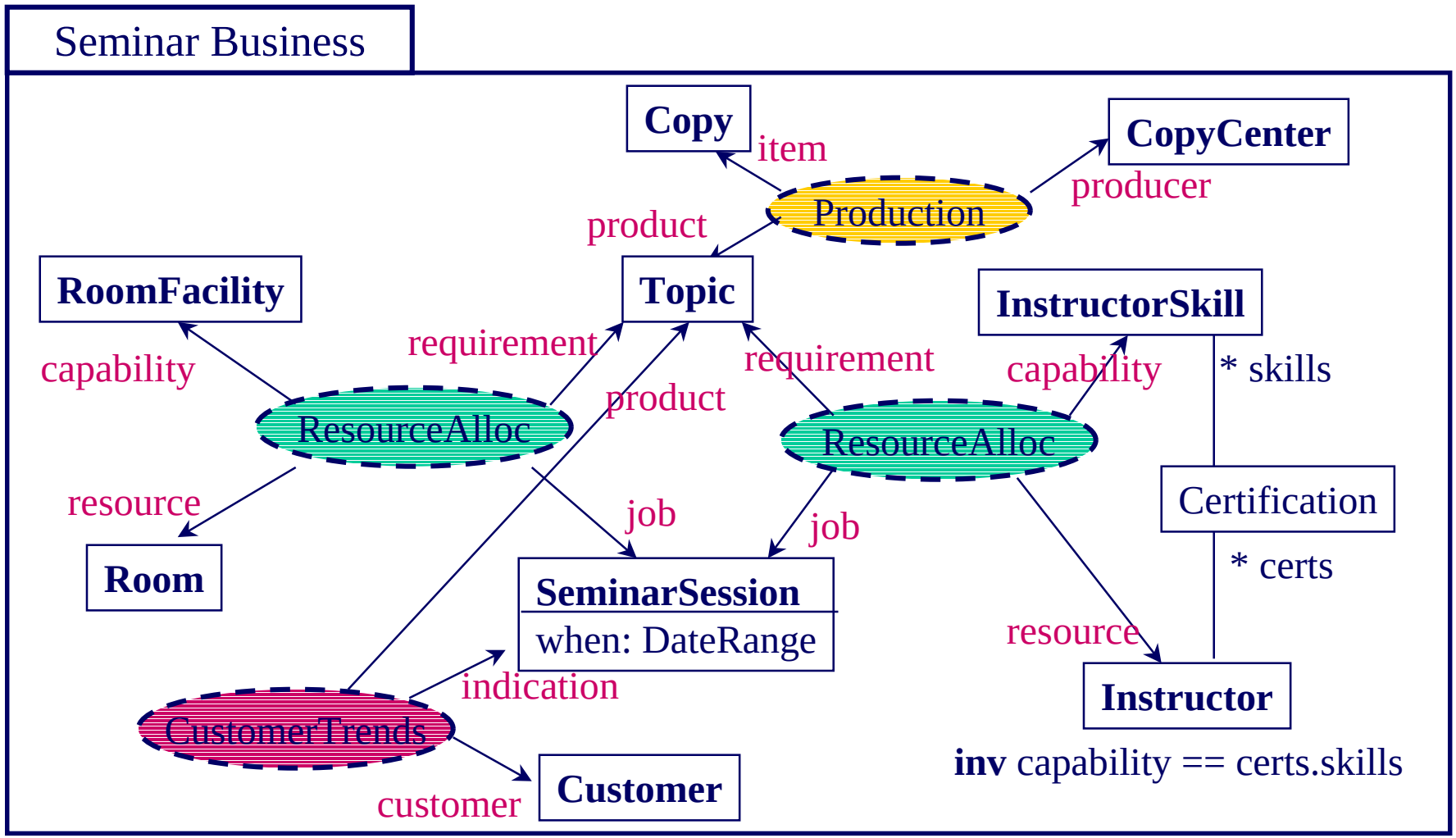
Model Frameworks - *Generic Models*



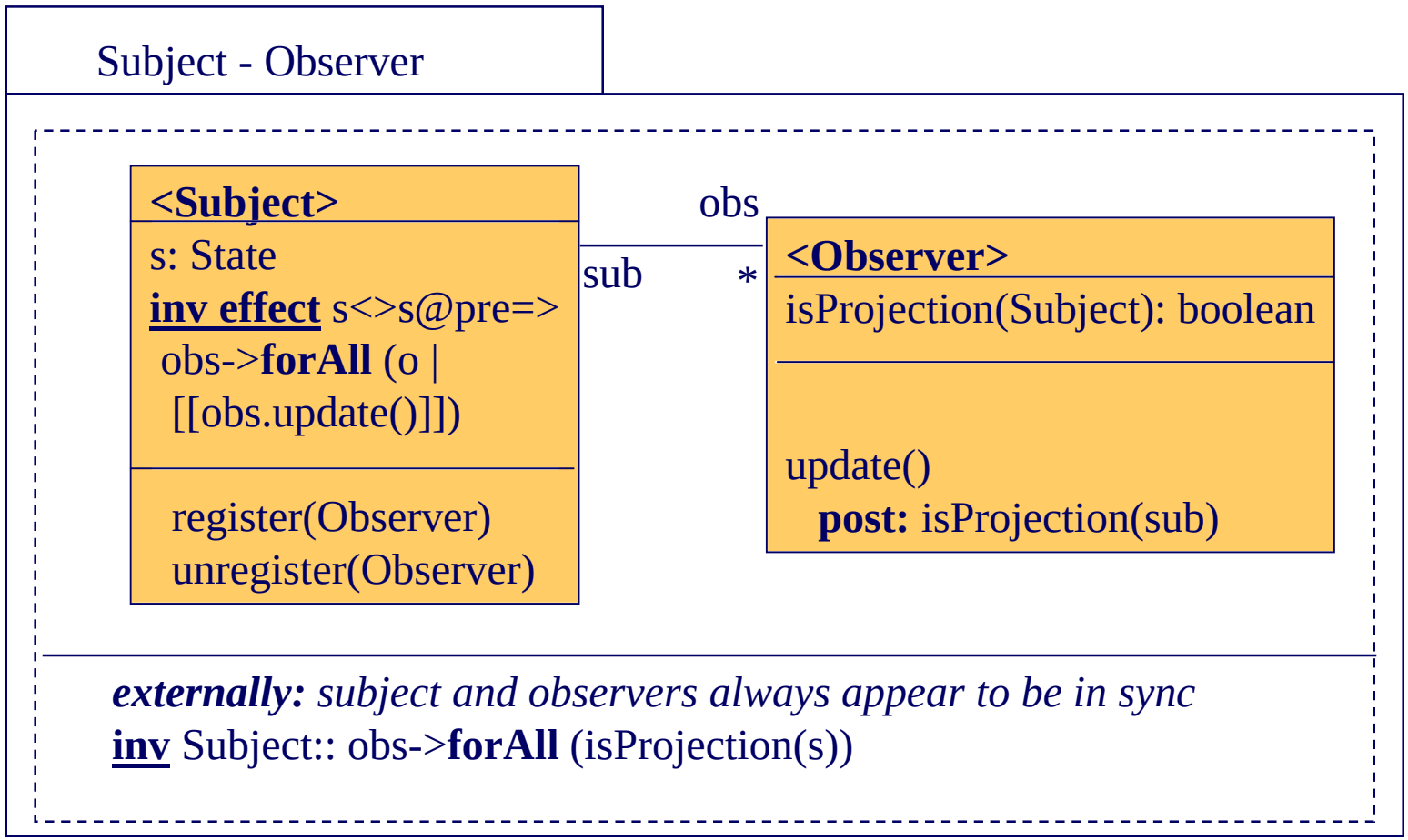
- ✓ A generic model / design / implementation component whose
  - ✓ Defines the broad generic structure and behavior
  - ✓ Provides *plug-points* for adaptation

A Complete Seminar Business Model

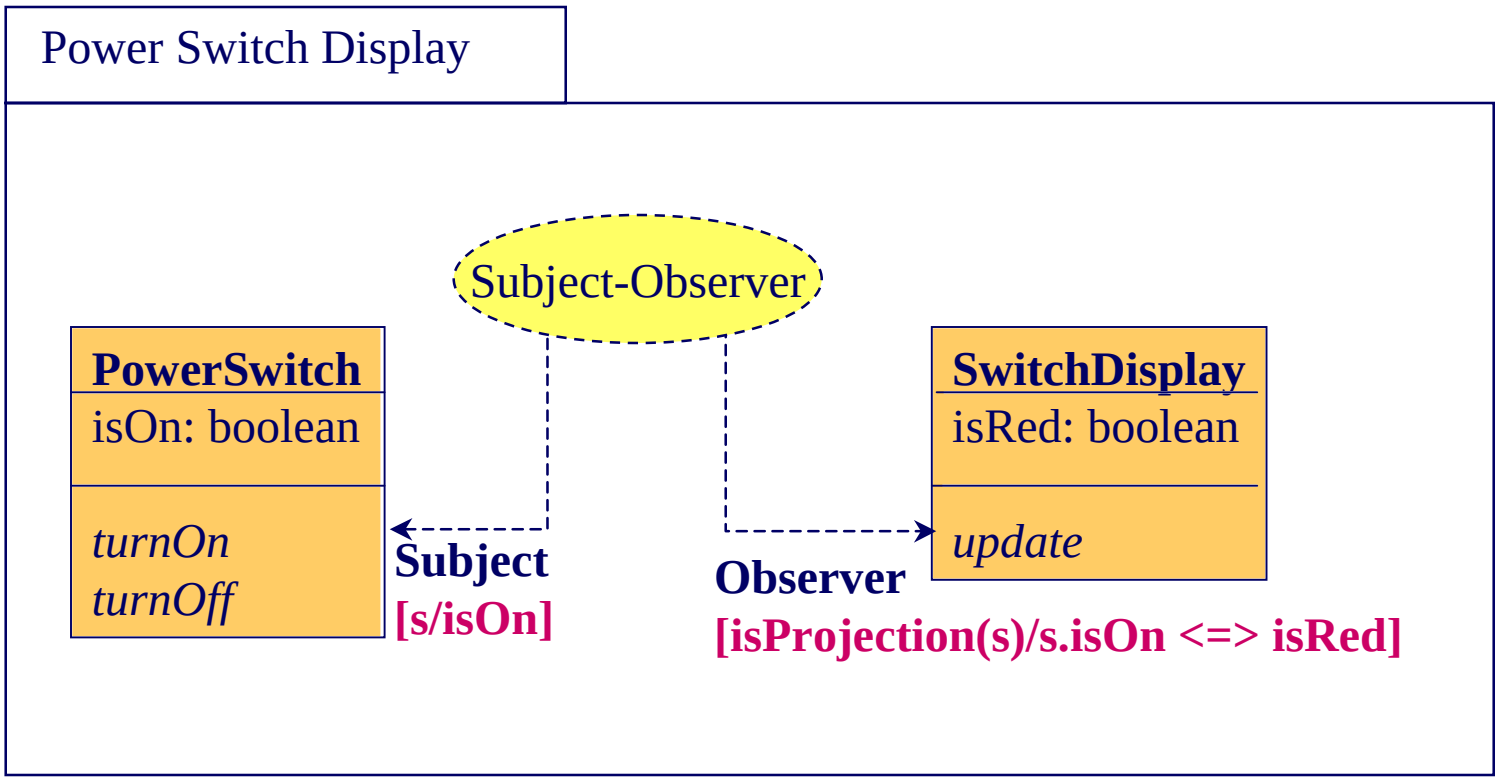
- ✓ Built by specializing three different pre-existing model frameworks



Observer Pattern as a Framework

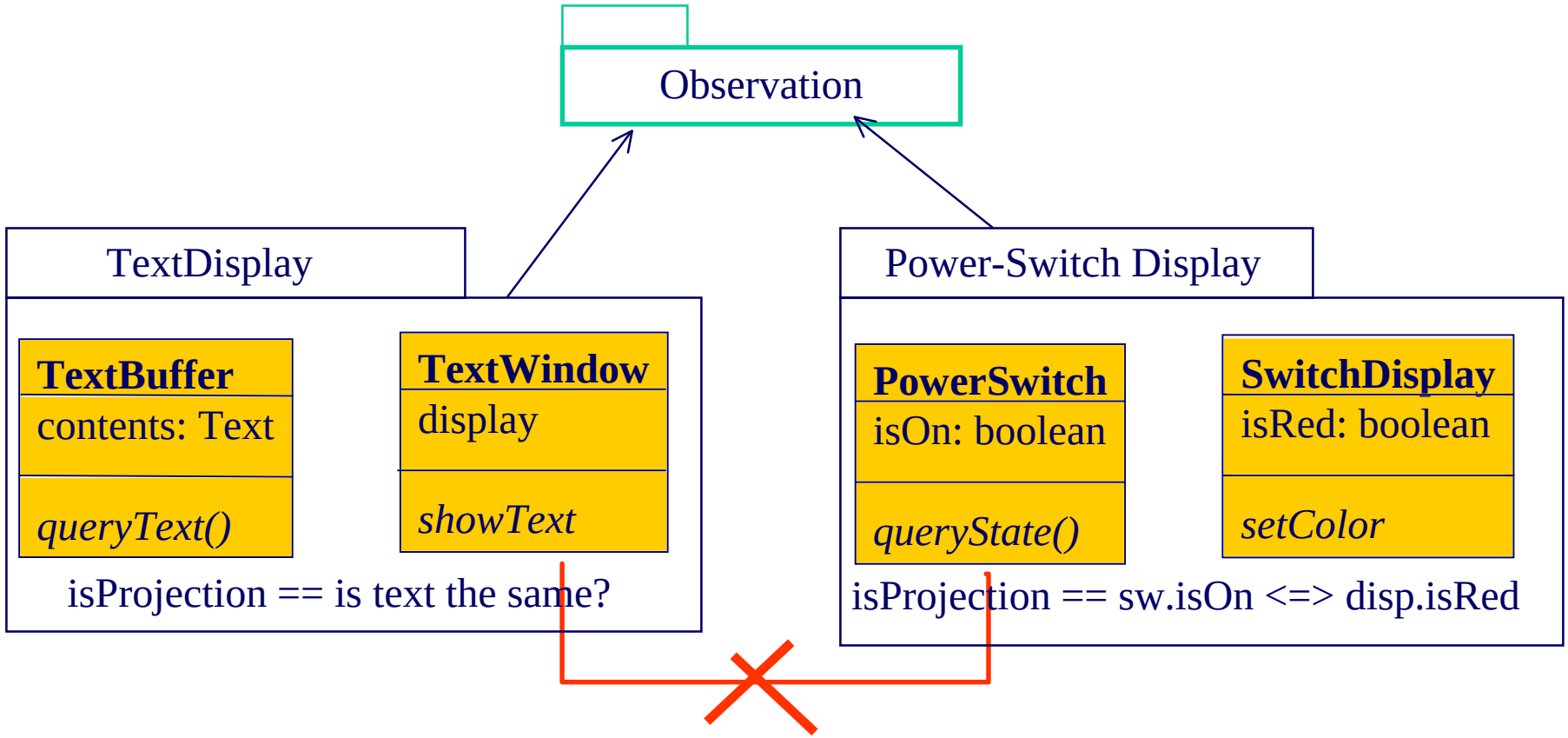


Applying the Observer Pattern



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

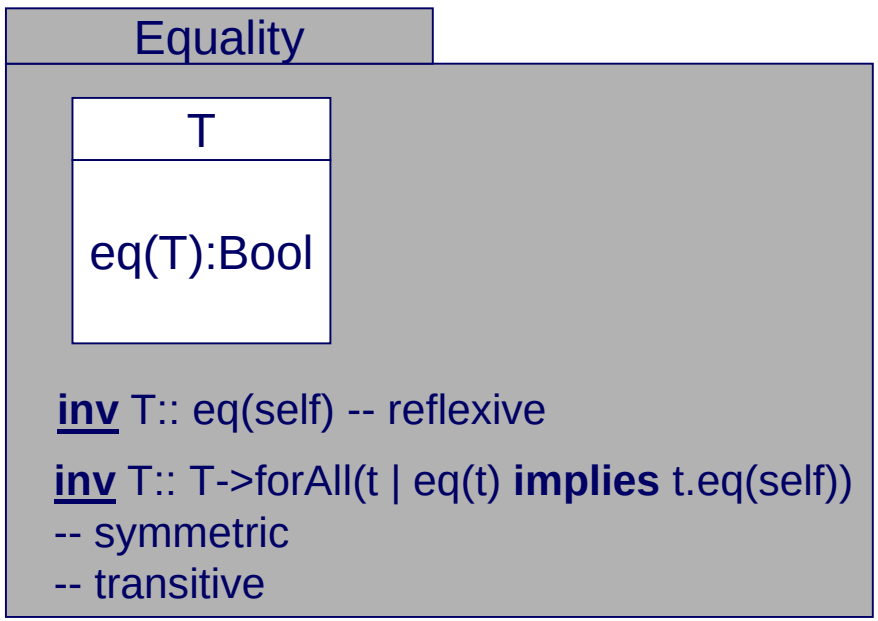
“Specializing” Subject-Observer

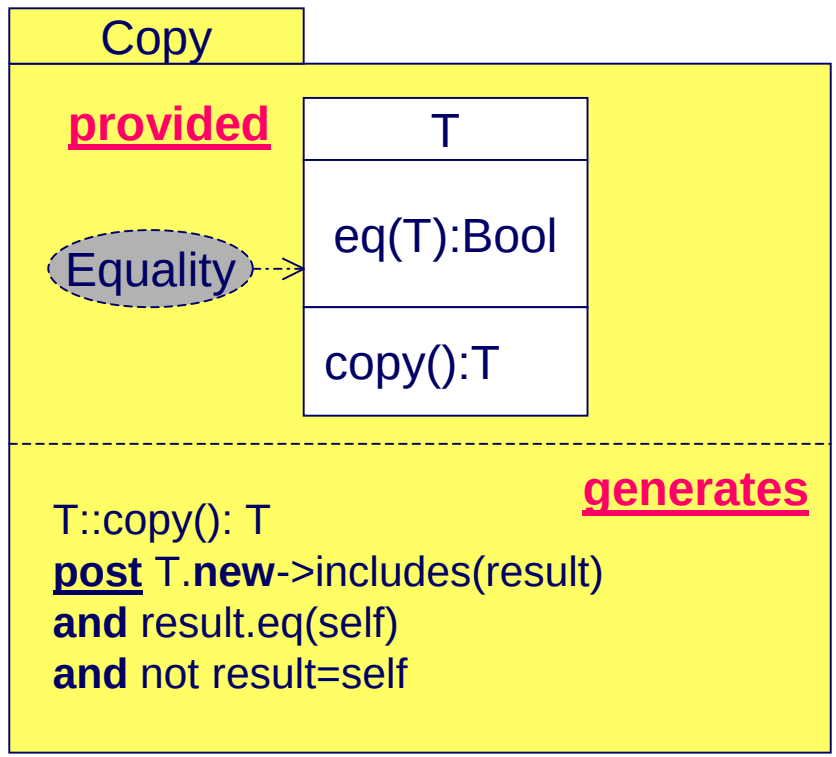


- Do not confuse this with subtype or subclass
  - ✓ The entire family of related types (playing roles) is specialized

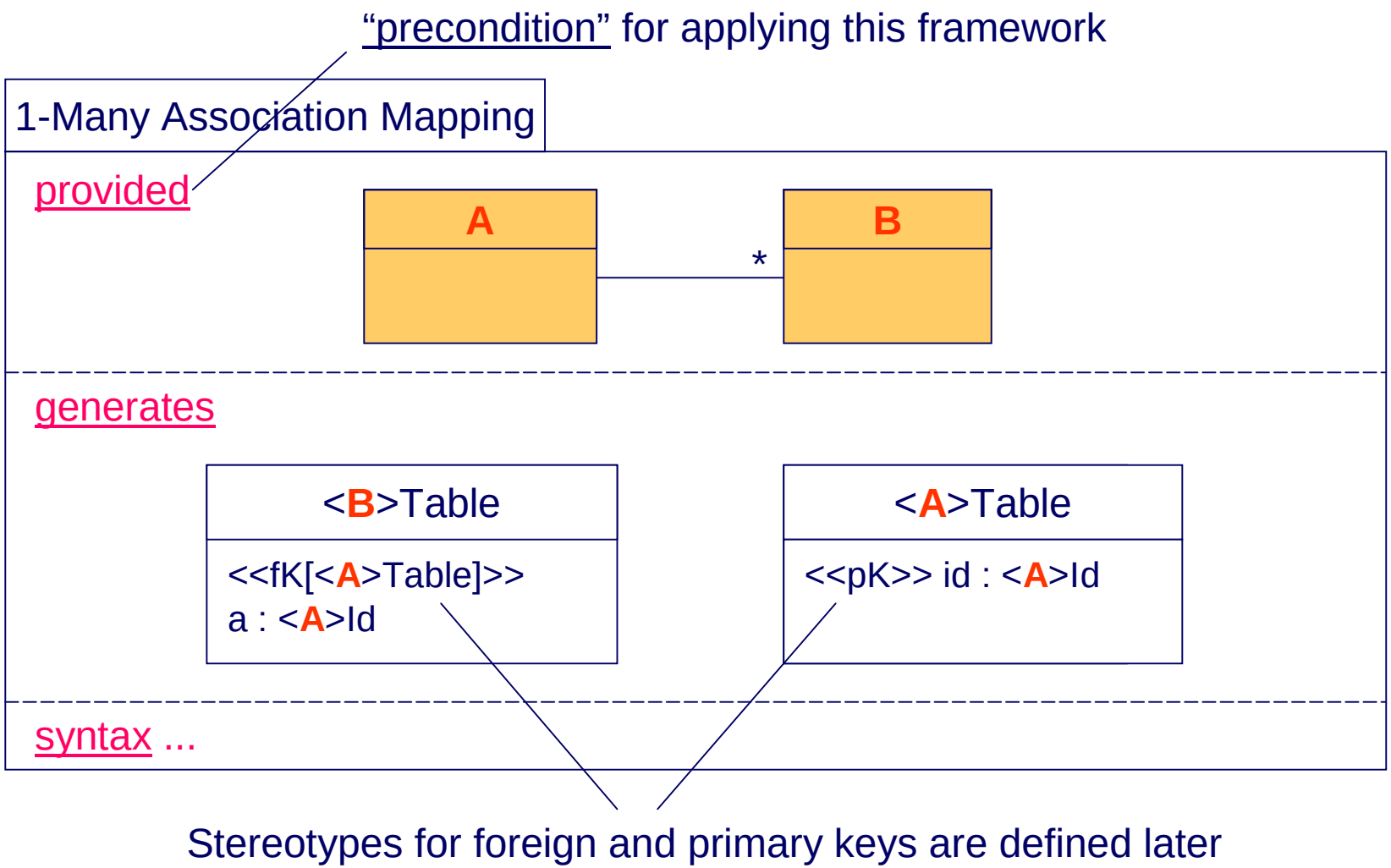
Provisos

- constraints on the applicability of a framework
- defines constraints on valid substitutions

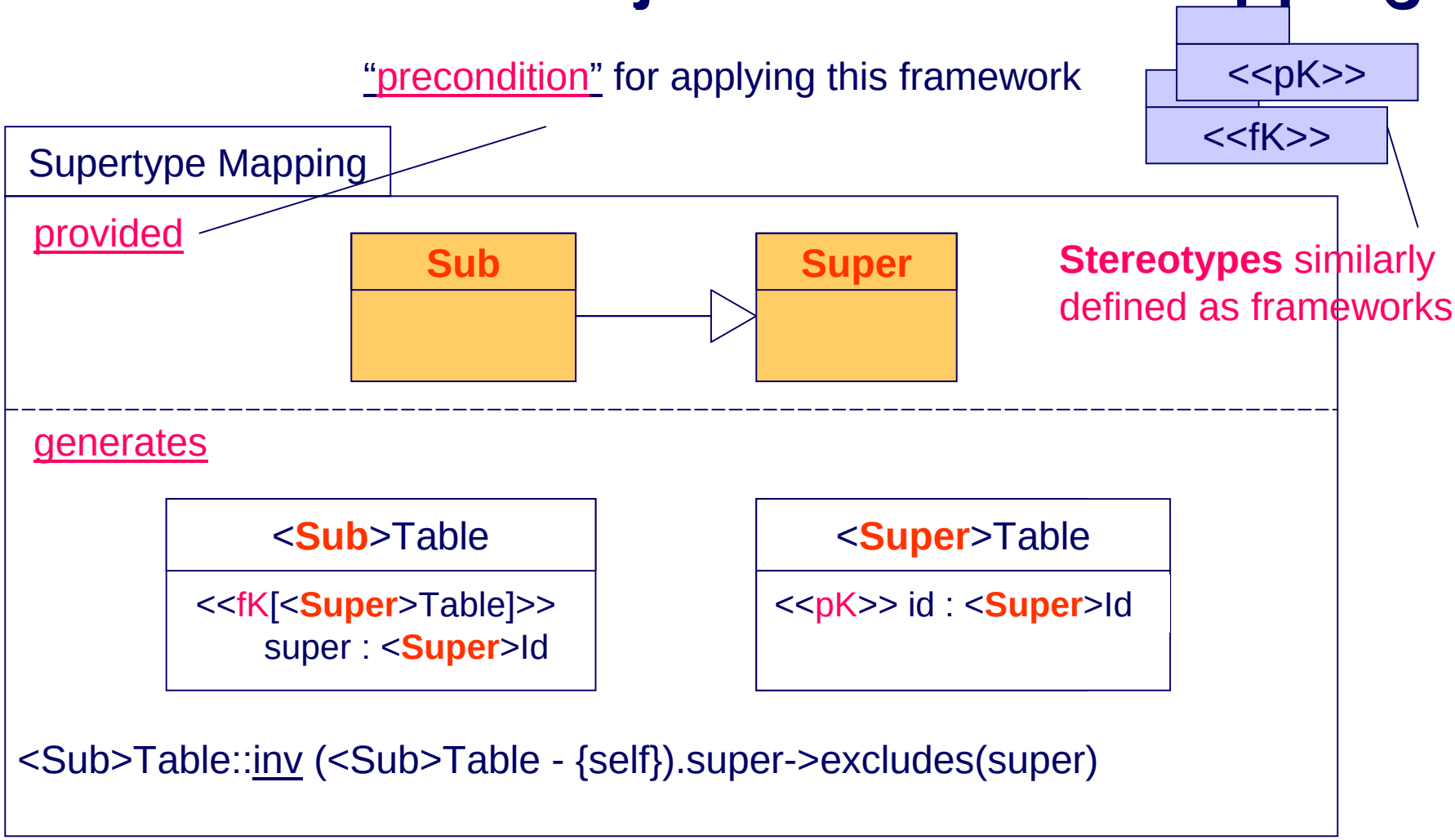




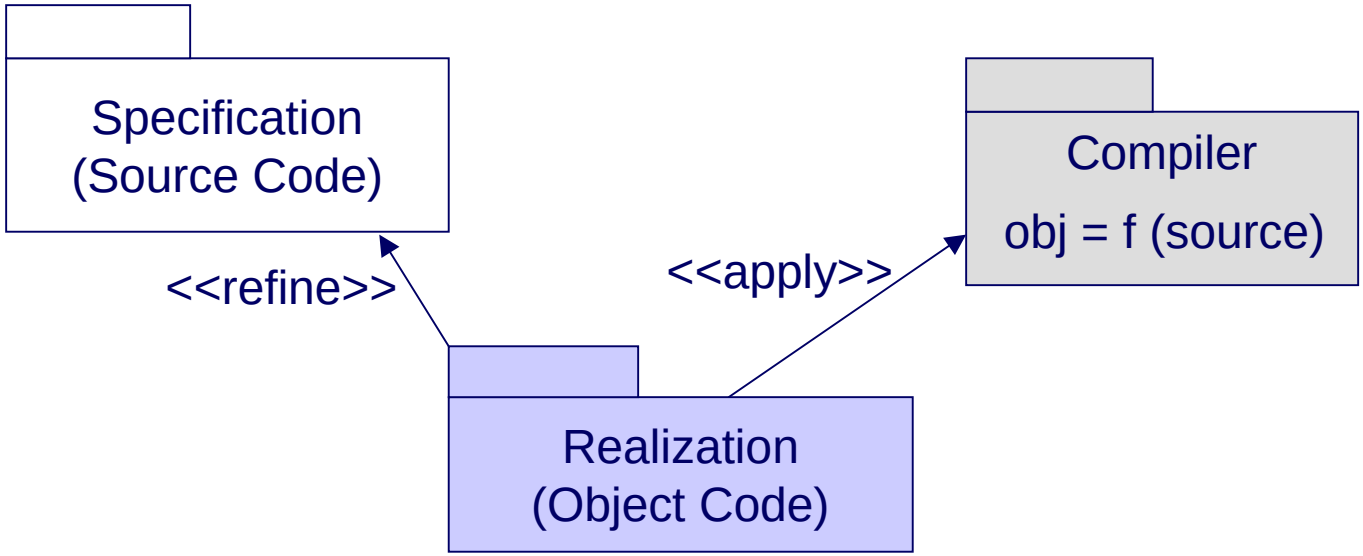
Frameworks for object-relational mapping



Frameworks for object-relational mapping



“Compilers”



- a compiler is the most restrictive form of architecture that is useful
  - ✓ the compiler generates a unique realization from the specification
- a compiler is a mapping from specifications to realizations
  - ✓ a compiler is correct if generated realization is always a refinement

Limitations of generativity - OCL at meta-level

- Not all architectural styles can be defined generatively with frameworks
  - ✓ inheritance hierarchies should be no more than five levels deep
- Language translation (e.g. compilation)
  - ✓ cannot be conveniently defined using framework applications
- In most cases, can only generate part of a realization. Part of it must be handcrafted and then we must check that no constraints are violated.

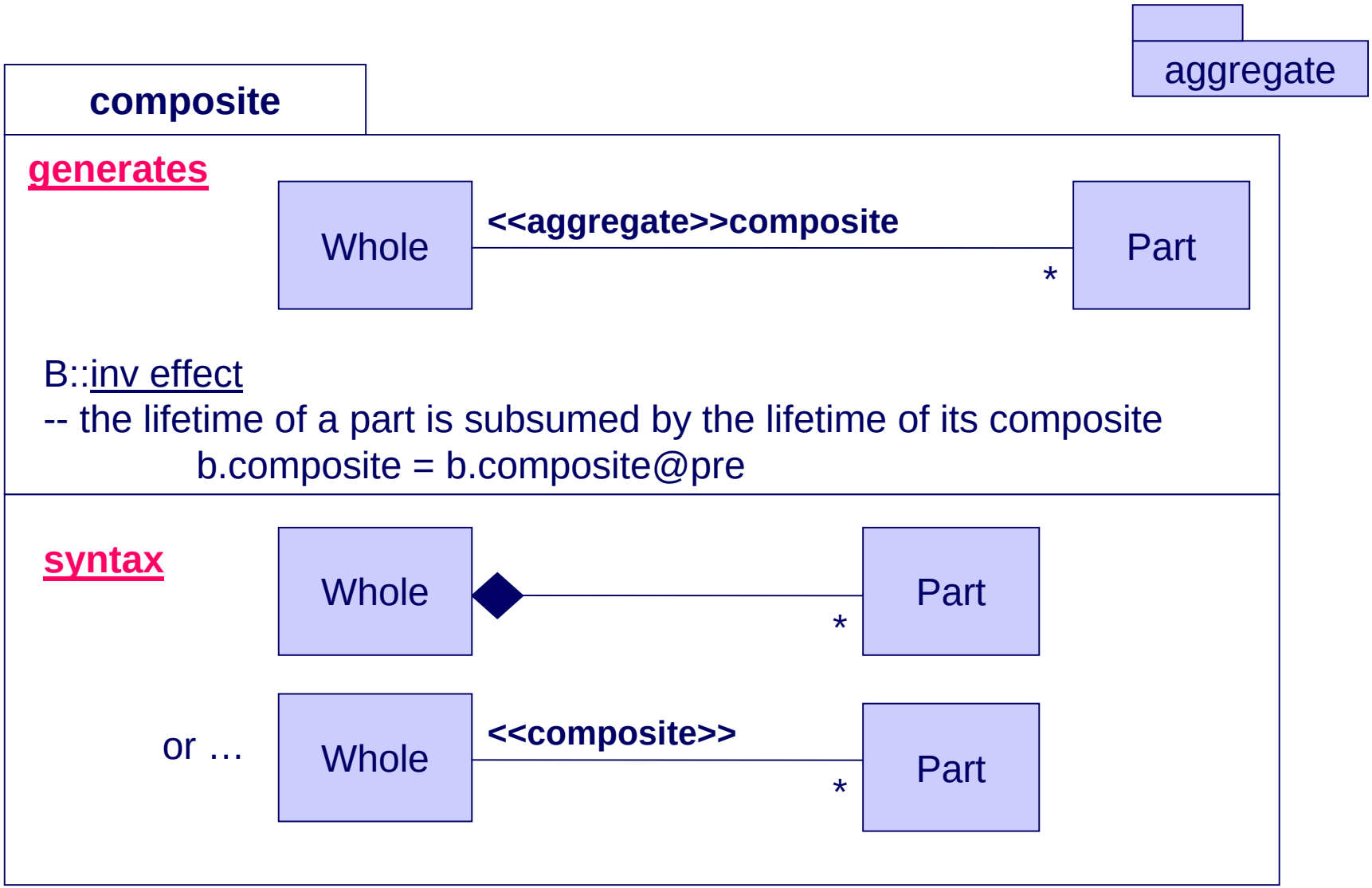
Outline

- Precise component specifications in UML
  - ✓ Interfaces
- Refinement
  - ✓ Component implementation refines spec
- **What is Software Architecture?**
  - ✓ Common definitions and examples
  - ✓ Catalysis definition of architecture
  - ✓ Specifying architectural styles using OCL
  - ✓ Generating architectural styles using Frameworks
  - ✓ **Specifying architectural styles with Stereotypes**
- Specifying component architectural styles
  - ✓ Components, ports, connectors and assemblies
  - ✓ Static assemblies
  - ✓ Dynamic assemblies
- Realizing component architectural styles
  - ✓ The CORBA component model
- Conclusion

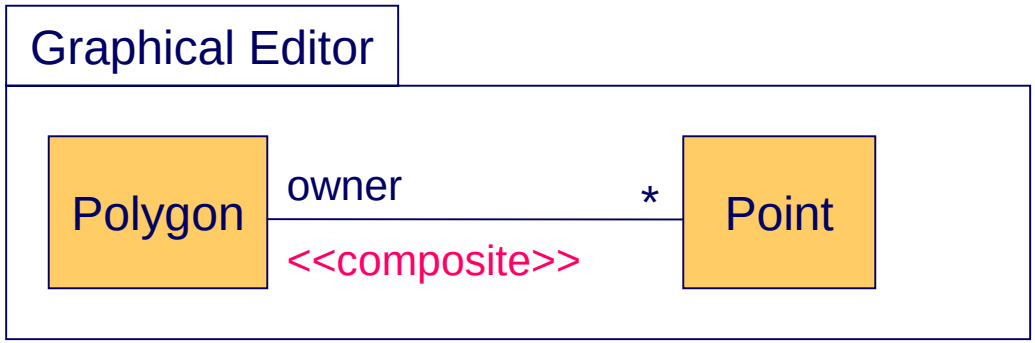
Stereotypes in Catalysis

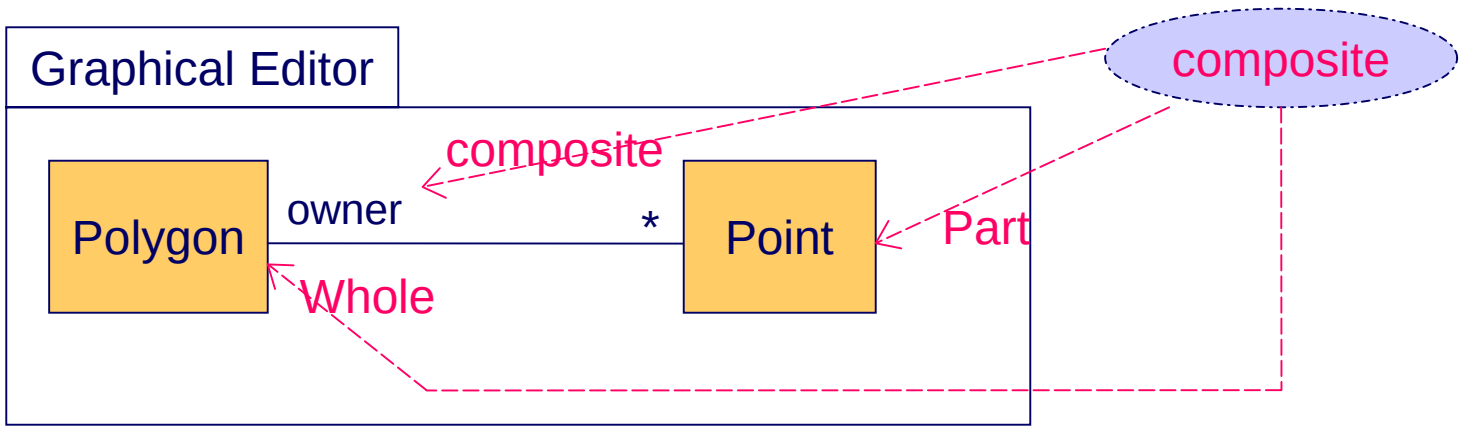
- UML language extension mechanism
- syntactic sugar for frameworks
- stereotype vs. framework
  - ✓ technical difference
    - ✓ stereotype focussed on a single model element - substitutions of other elements inferred implicitly from context
    - ✓ framework explicitly substitutes all parameters
  - ✓ conceptual distinction
    - ✓ stereotypes encourage meta-model extension
    - ✓ frameworks define a generic library in an existing language
- e.g. defining Java Beans language
  - ✓ new elements - beans, events, properties etc.

A Stereotype Definition - UML Composition



A Stereotype Use = Framework Application



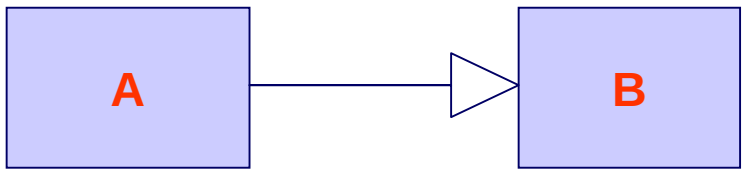


Relational Stereotypes

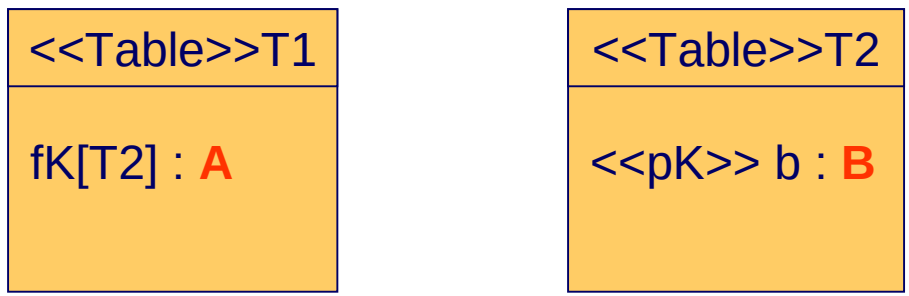
<<Table>>, <<primaryKey>>, …..

fK[Table2]

provided



generates



T1::inv T2.allInstances.b->includes(fK[T2])

Section Summary - Generative Architecture

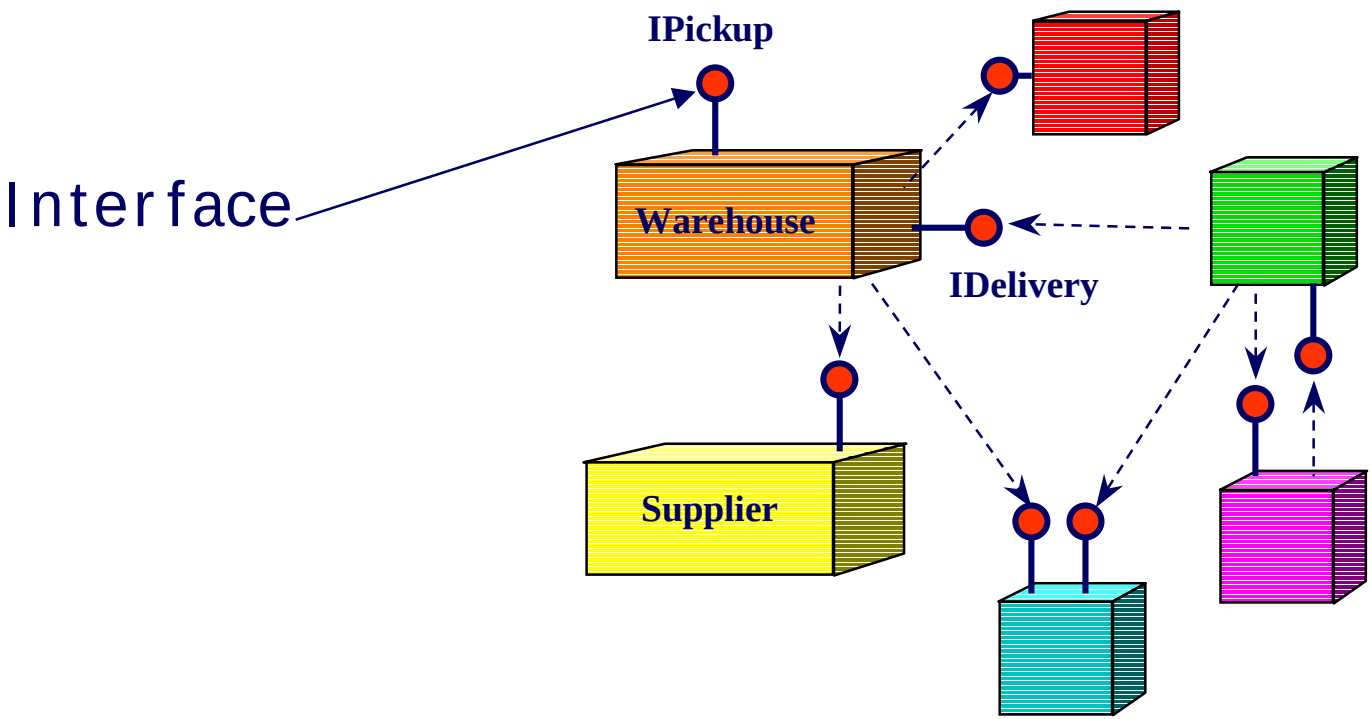
- Architectural style defines language and rules for valid realizations of some specification
  - ✓ Style = Set of <spec, realization, refinement>
- Style either defined as constraint or “generative”
- Generative style = construct + its realization pattern
- Frameworks capture any model pattern
  - ✓ Framework is a package
  - ✓ Pattern application is import + substitute

Outline

- Precise component specifications in UML
  - ✓ Interfaces
- Refinement
  - ✓ Component implementation refines spec
- What is Software Architecture?
  - ✓ Common definitions and examples
  - ✓ Catalysis definition of architecture
  - ✓ Specifying architectural styles using OCL
  - ✓ Generating architectural styles using Frameworks
  - ✓ Specifying architectural styles with Stereotypes
- **Specifying component architectural styles**
  - ✓ **Components, ports, connectors and assemblies**
  - ✓ Static assemblies
  - ✓ Dynamic assemblies
- Realizing component architectural styles
  - ✓ The CORBA component model
- Conclusion

Components in UML

Components interact via clearly specified interfaces

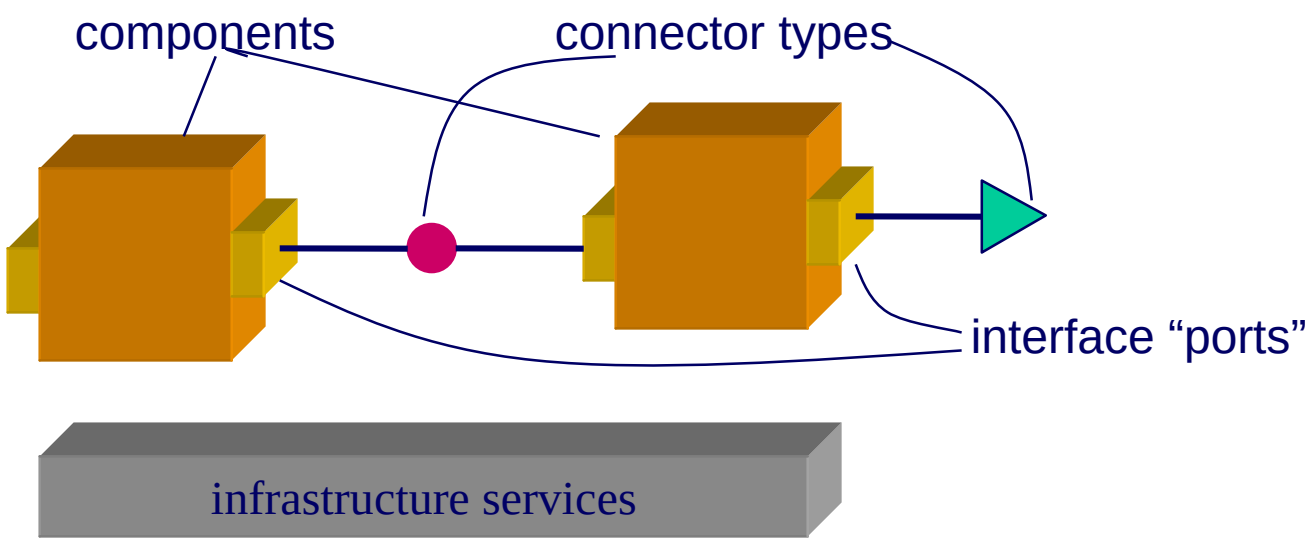


Focus on interfaces

- precise external behavior (provided + required)
- all implementation aspects completely hidden

Component Terminology

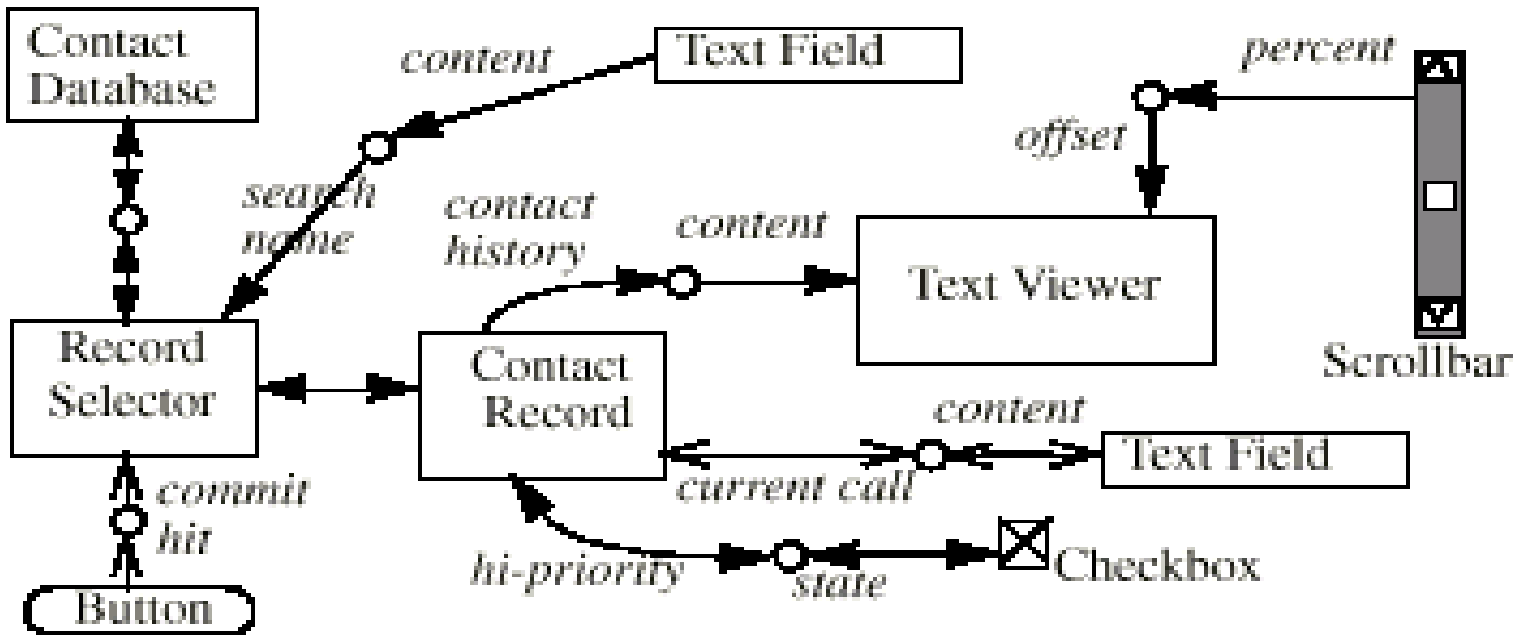
- ✓ *Component:* Software chunk that can be “plugged” together with others
- ✓ *Connector:* A coupling between *ports* of two *components*
- ✓ *Port*: The “plugs” and “sockets” of an individual component
- ✓ *Component Architecture Style:* (a) Standard port “*connector*” types and rules
- ✓ *Component Infrastructure:* (b) Standard services for components and connectors
- ✓ *Component Kit:* Components designed to work together with common architecture



Outline

- Precise component specifications in UML
  - ✓ Interfaces
- Refinement
  - ✓ Component implementation refines spec
- What is Software Architecture?
  - ✓ Common definitions and examples
  - ✓ Catalysis definition of architecture
  - ✓ Specifying architectural styles using OCL
  - ✓ Generating architectural styles using Frameworks
  - ✓ Specifying architectural styles with Stereotypes
- **Specifying component architectural styles**
  - ✓ Components, ports, connectors and assemblies
  - ✓ **Static assemblies**
  - ✓ Dynamic assemblies
- Realizing component architectural styles
  - ✓ The CORBA component model
- Conclusion

Component assembly - specification



inv checkbox.state = contactRecord.hi-priority
inv textField.content = contactRecord.currentCall
inv effect textField.content changed **implies** recordSelector.searchName = textField.content
inv effect button hit **implies** recordSelector committed

Can check that overall behavior is correct uncluttered by connector implementation details

Component assembly - code generation

Port code:

get/set methods for properties

e.g. `CheckBox { getState(): boolean, setState(boolean), . . . }`

registration and de-registration methods for events and output properties

e.g. `Button { registerHitHandler(HitHandler), . . . . }`

code to check for event conditions and generate events

code to check for and signal property changes

Connector code:

adapters to enable events to trigger method invocations

e.g.
```
RecordSelectorHitAdapter implements HitHandler
        { handleHit() { recordSelector.commit(); } }
```

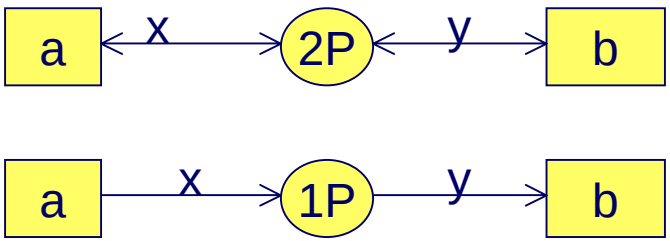
Configuration code:

code to create component and adapter instances and register them appropriately

e.g.
```
b = new Button(); rs = new RecordSelector();
rsha = new RecordSelectorHitAdapter(rs);
b.registerHitHandler(rsha)
```

Connector specifications

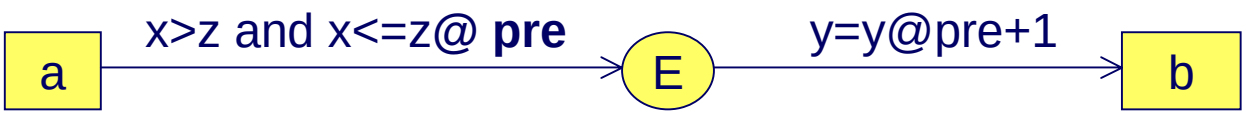
Property connectors: 2-way and 1-way



inv a.x = b.y

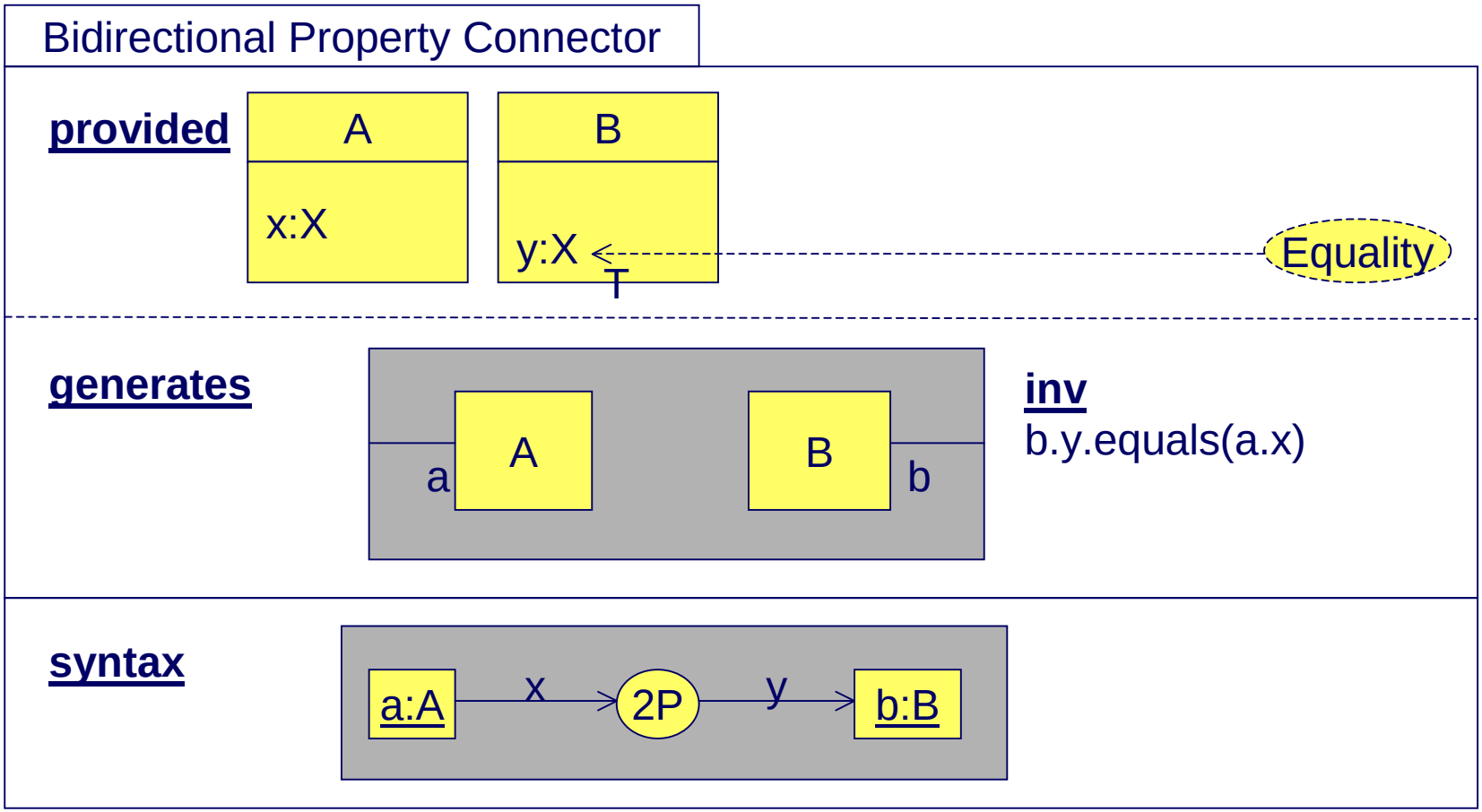
inv effect a.x <>a.x@**pre**
implies b.y.equals(a.x)

Event connectors:



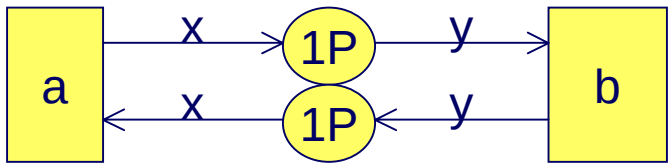
inv effect a.x>a.z and a.x<=a.z@**pre implies** b.y=b.y@**pre**+1

Snapshot connector frameworks



Connector refinements

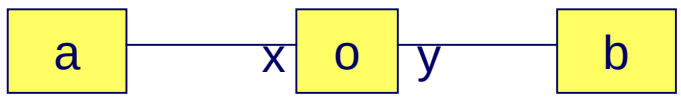
The 2P connector can be realized as 2 1P connectors:



since
inv effect a.x <>a.x@**pre implies** b.y.equals(a.x)
and
inv effect b.y <>b.y@**pre implies** a.x.equals(b.y)

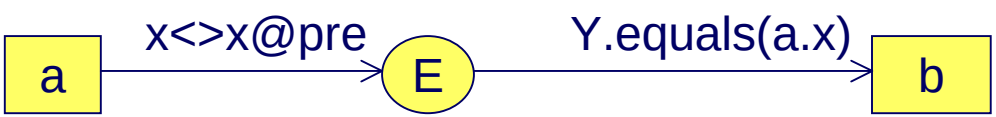
implies **inv** a.x.equals(b.y)

Many alternative realizations e.g. sharing



since a.x=b.y
implies a.x.equals(b.y)

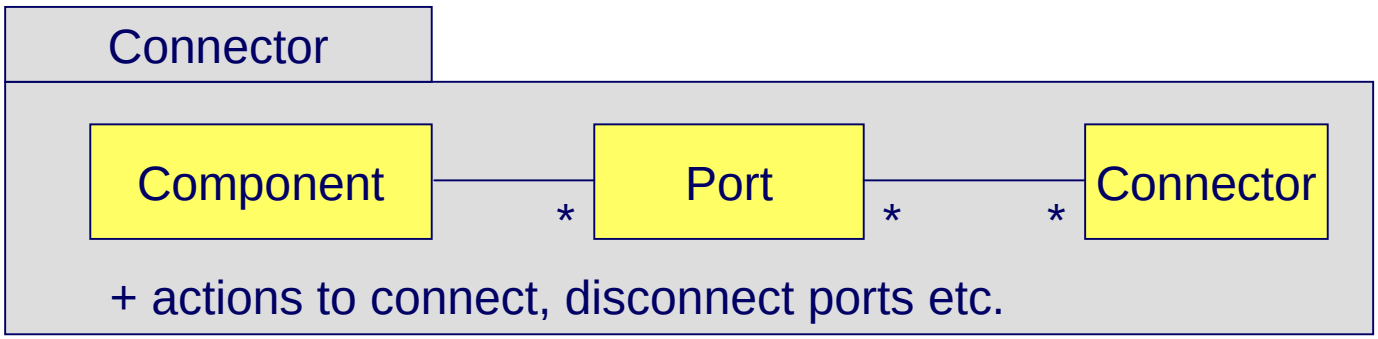
The 1P connector can be realized as an E connector:

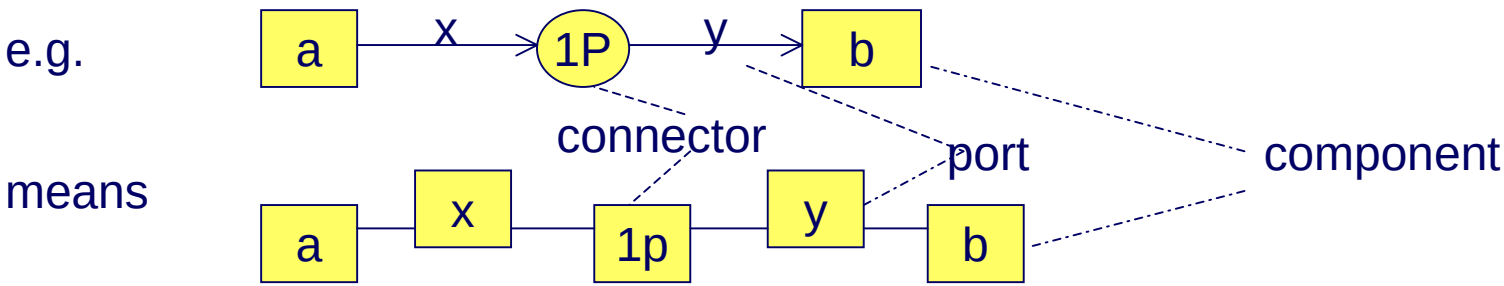


Outline

- Precise component specifications in UML
  - ✓ Interfaces
- Refinement
  - ✓ Component implementation refines spec
- What is Software Architecture?
  - ✓ Common definitions and examples
  - ✓ Catalysis definition of architecture
  - ✓ Specifying architectural styles using OCL
  - ✓ Generating architectural styles using Frameworks
  - ✓ Specifying architectural styles with Stereotypes
- **Specifying component architectural styles**
  - ✓ Components, ports, connectors and assemblies
  - ✓ Static assemblies
  - ✓ **Dynamic assemblies**
- Realizing component architectural styles
  - ✓ The CORBA component model
- Conclusion

Objectifying Ports and Connectors





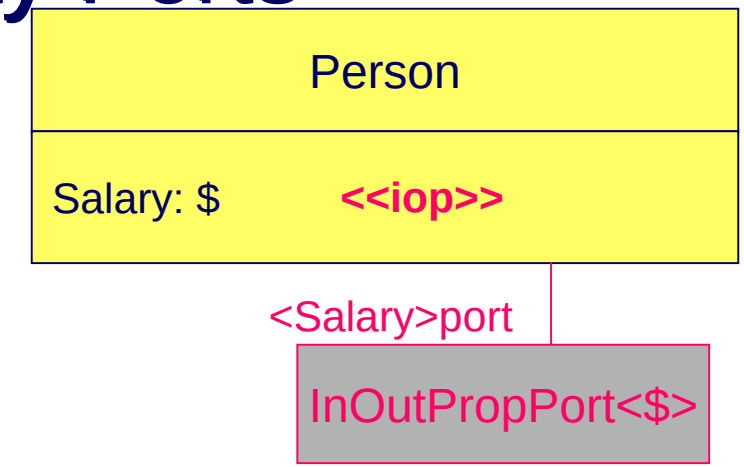
A directional connector is a specialization of a connector which has two ports, and distinguishes them as source and sink.

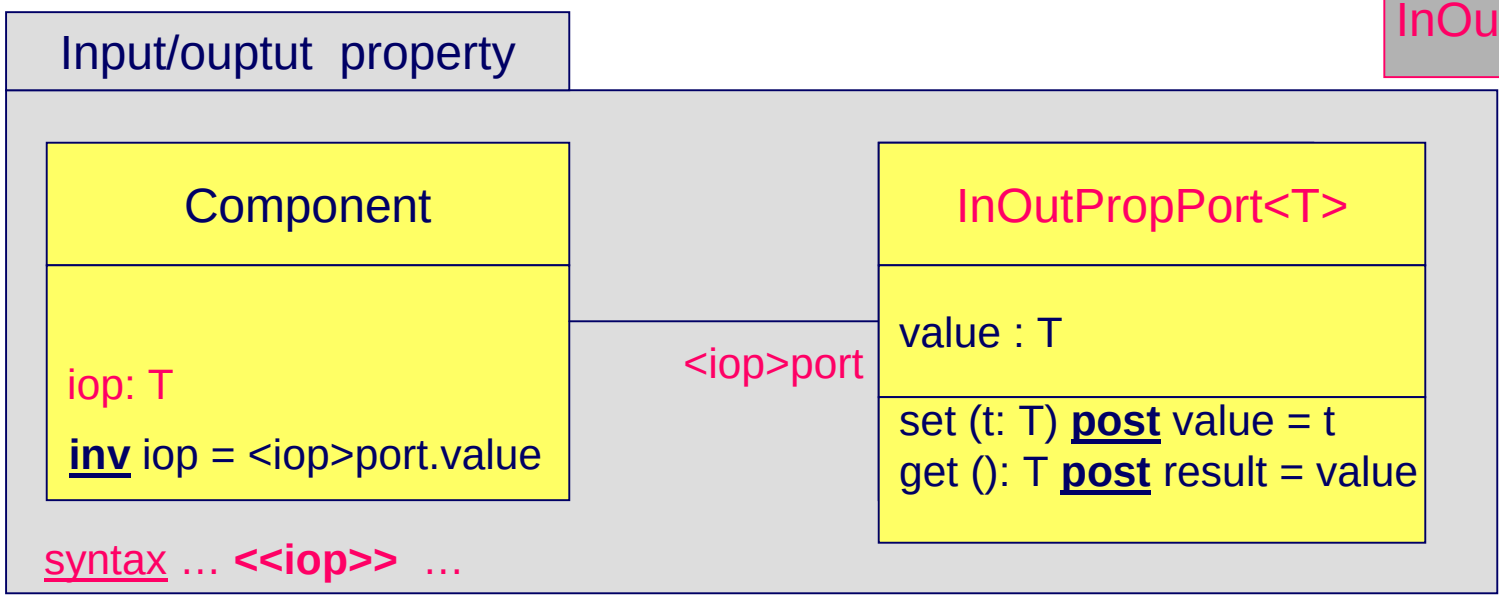
Need to objectify ports and connectors to specify **dynamically evolving** assemblies - e.g. registration and de-registration may be triggered by events or property changes

Attribute/Property Ports

3 basic varieties:
- input/output port has get and set
- input port has set only, output port has get only

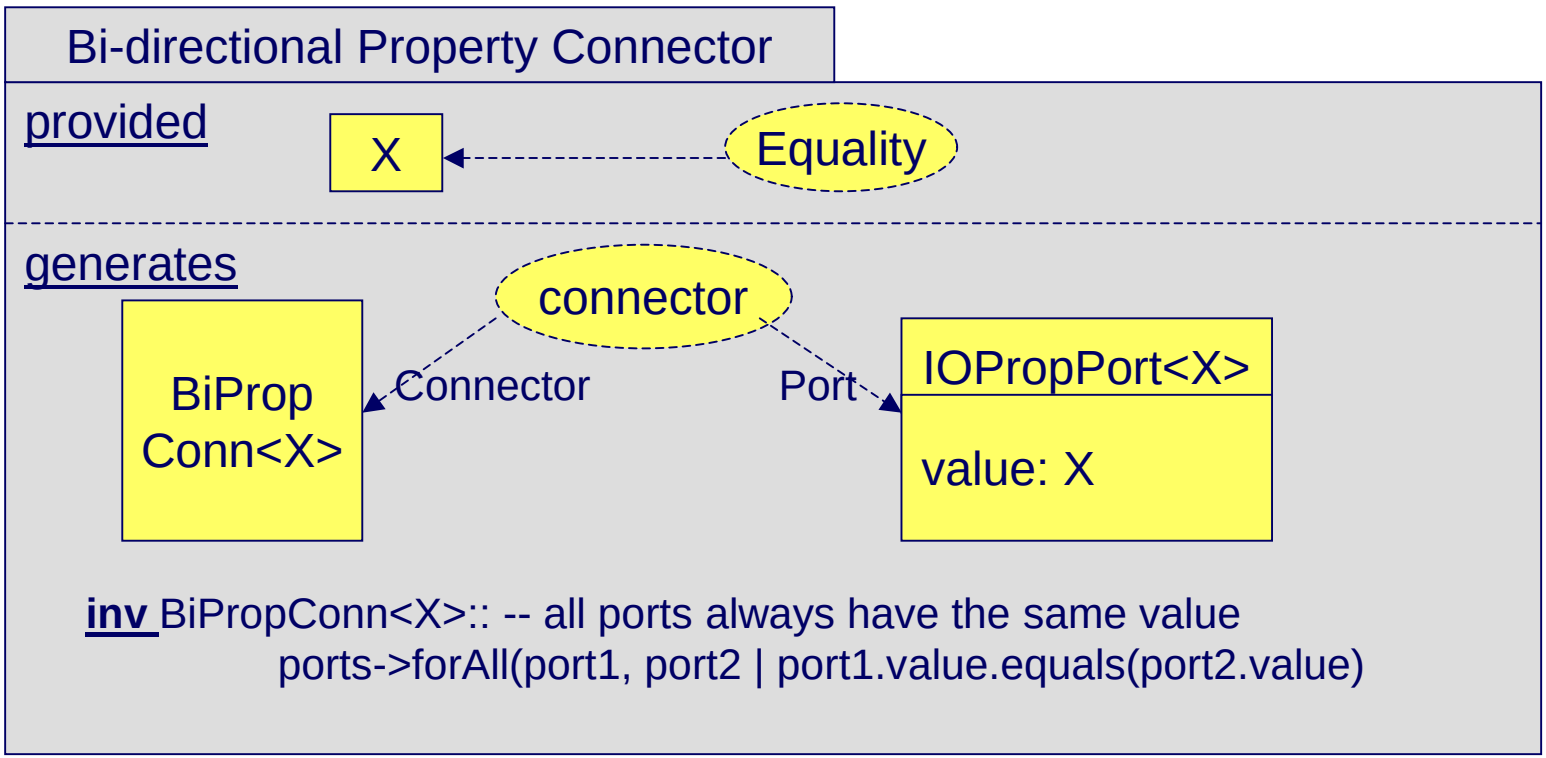
Ports are defined and used as stereotypes e.g.





Other possibilities e.g. constrained input port has set with a precondition

Attribute/Property Connectors

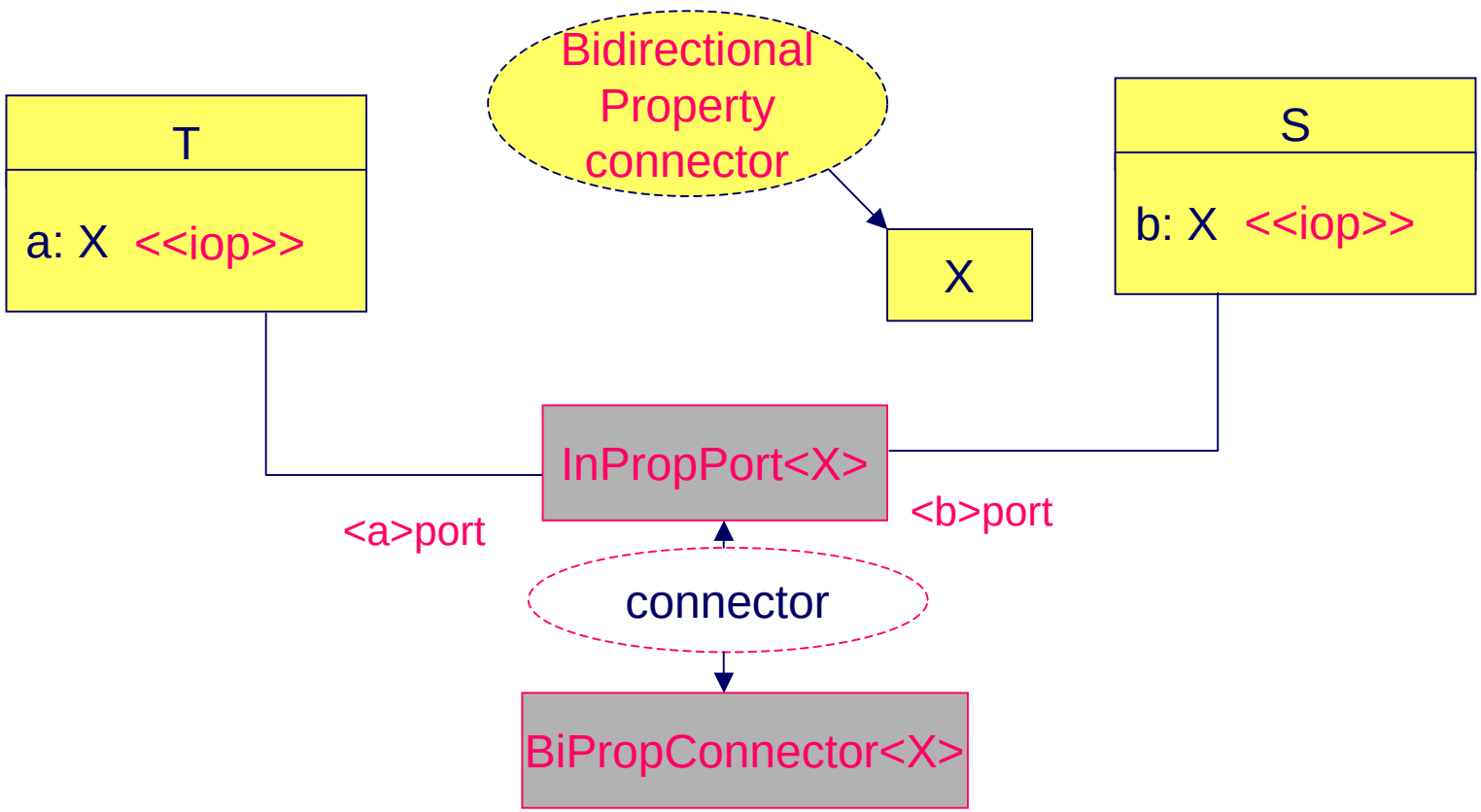


A simple bi-directional property connector always keeps a collection of input/output property ports in synch:

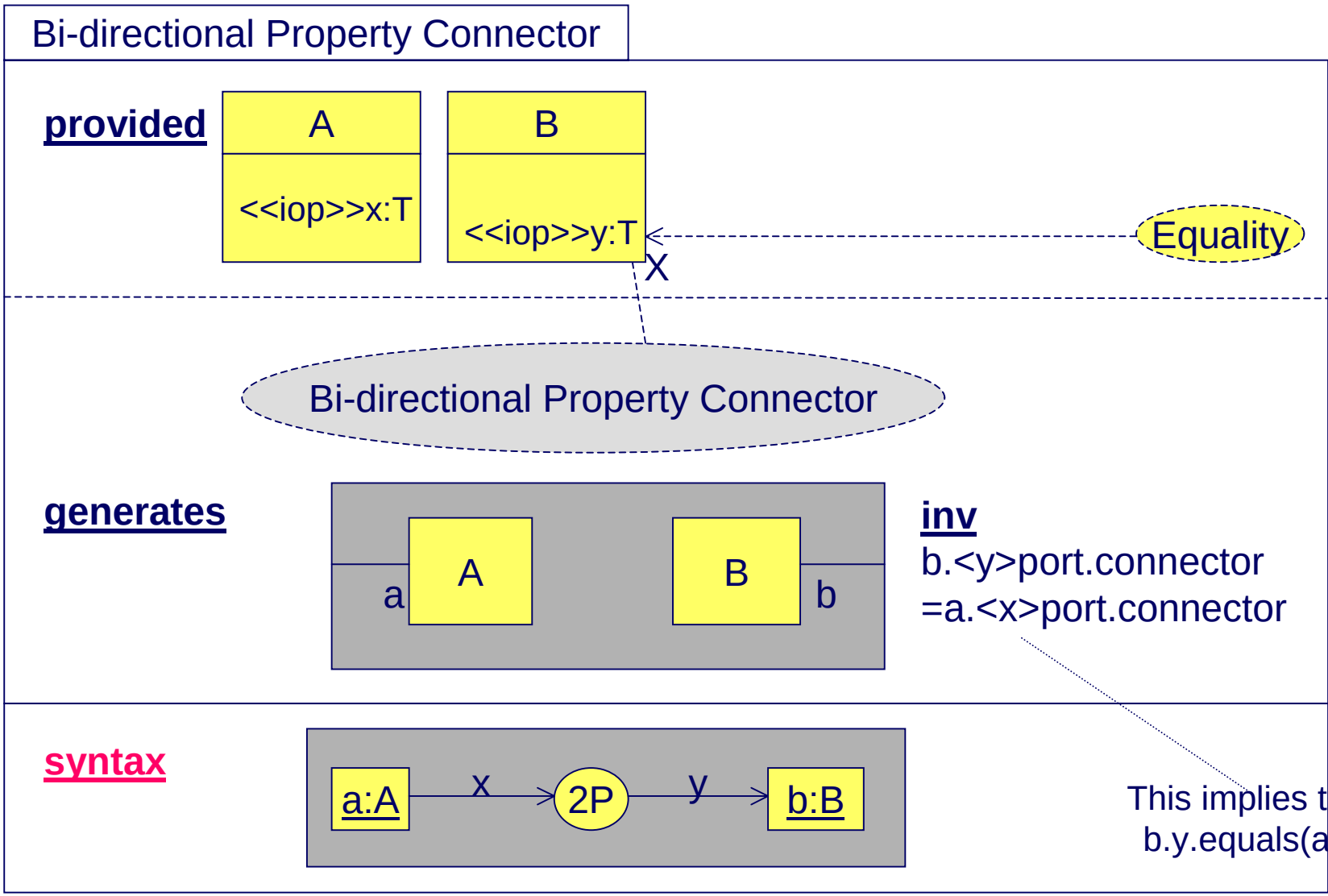
- when any property changes, all the connected properties change to stay in synch

Applying Port / Connector at Type Level

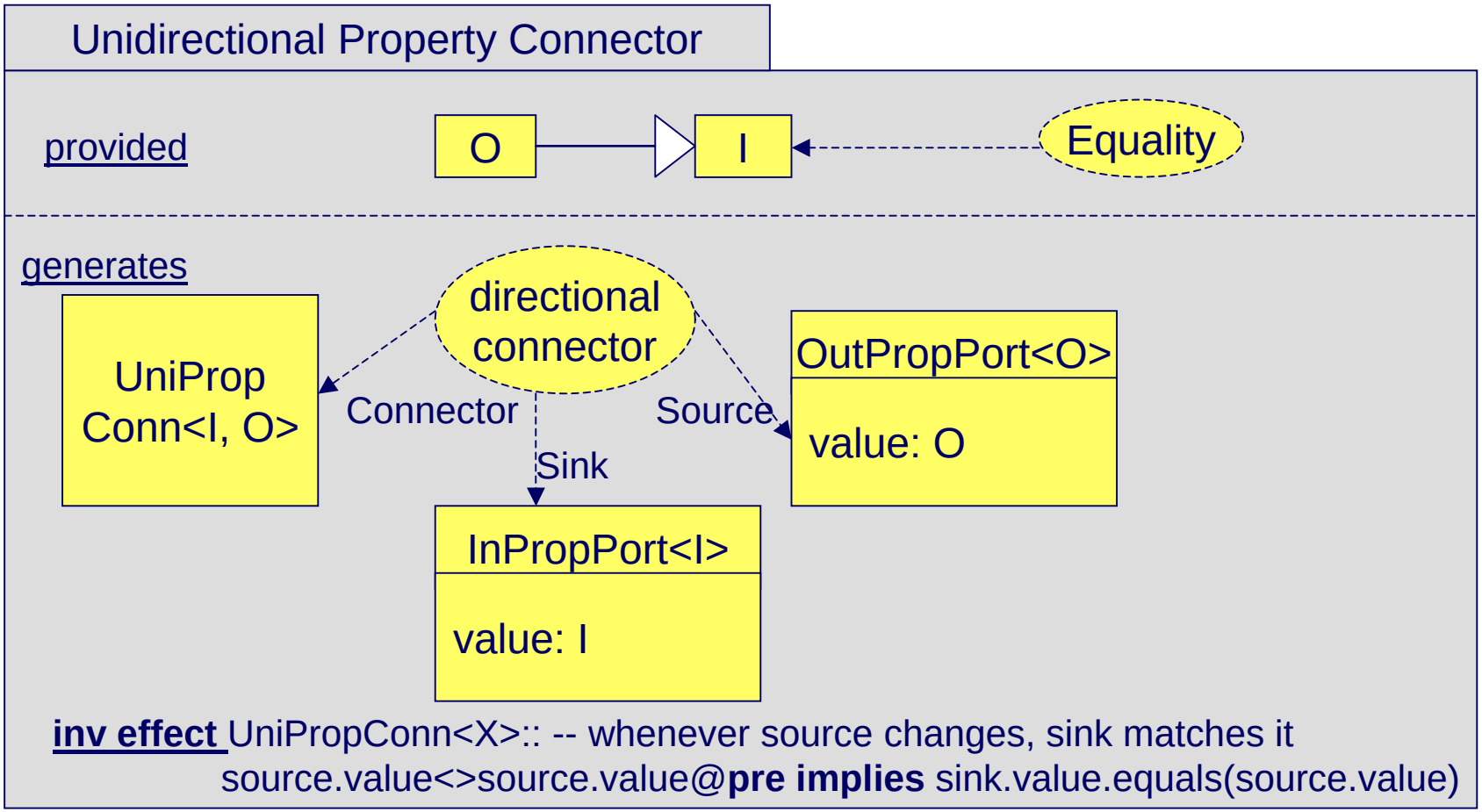
New



Snapshot connector frameworks

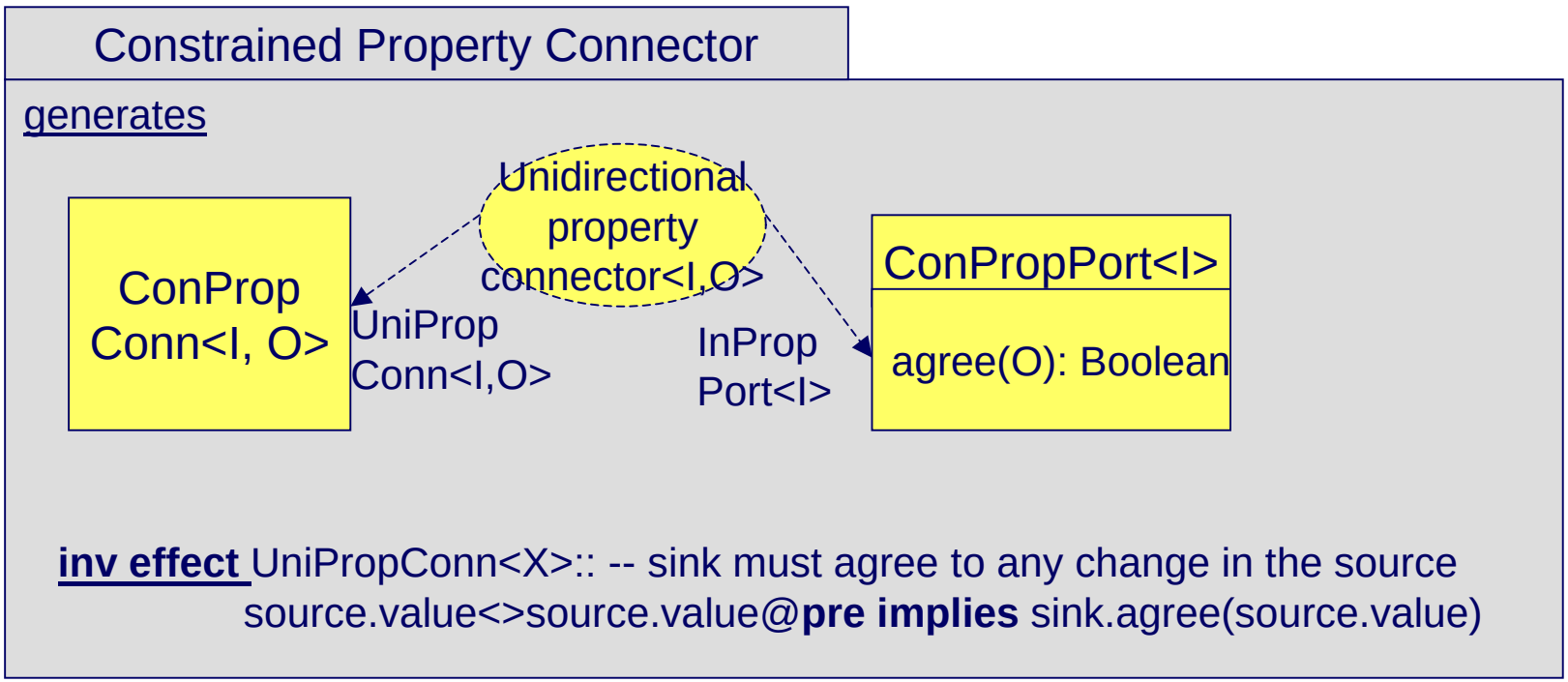


Attribute/Property Connectors



The unidirectional connector guarantees that properties are kept in synch in one direction only.
If the sink value can change independently, it may, at times, be out of synch with the source value.

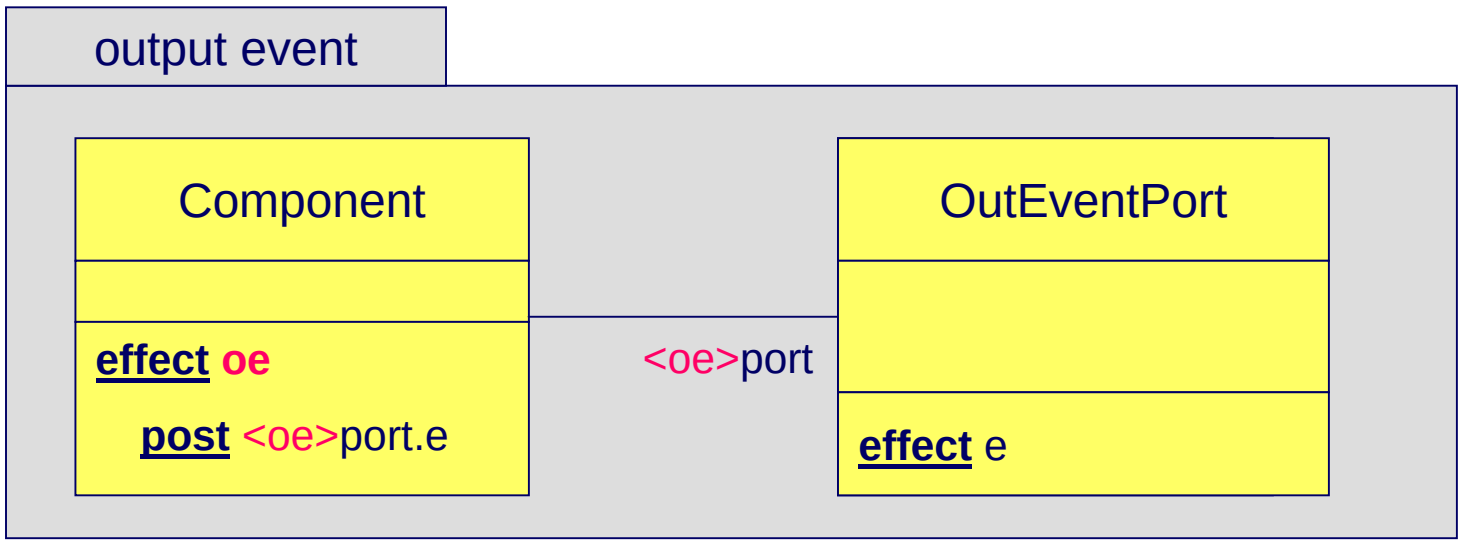
Constrained Property Connector



The constrained property connector is a specialization of the unidirectional property connector, in which the sink can veto a change to the source value.

Event Ports

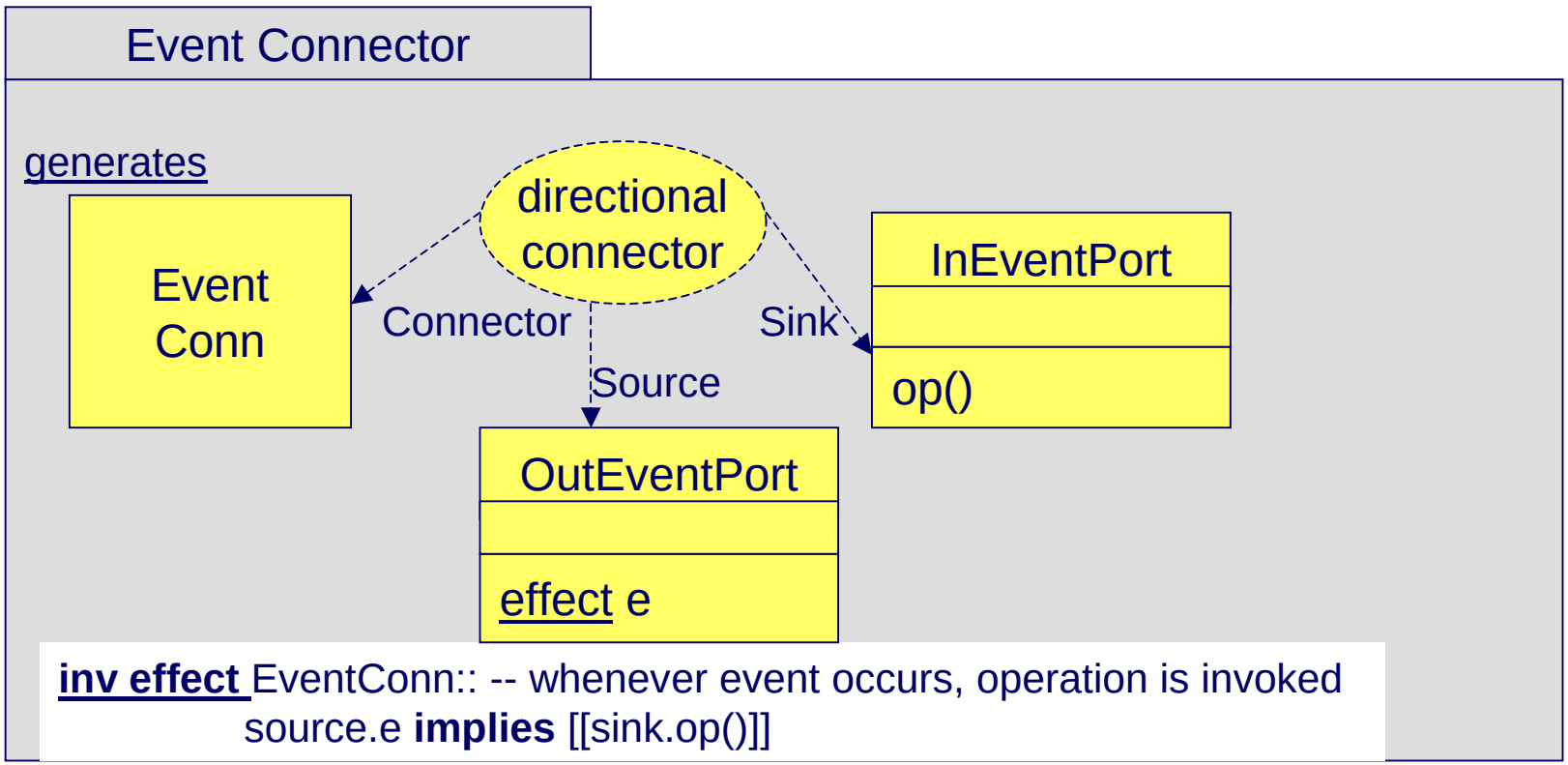
2 varieties - output event ports that generate events, input event ports that handle them



An effect is a named state change.
Events are always generated as a result of state changes.
An **input event port** is an **adapter** for an **operation** on the component.

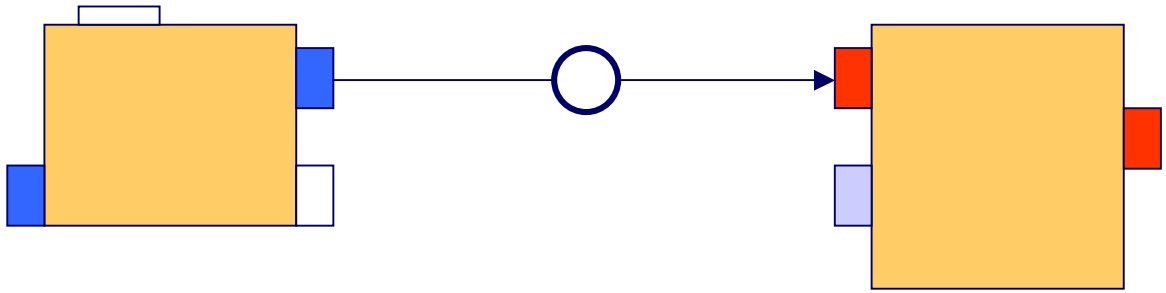
An output event port **realization** will include **registration/de-registration** operations.

Event connector



Generalizations are possible for **parameterized** events and operations.
The event connector is then parameterized by the number and type of parameters.

Section Summary - Component Architecture



- Connectors couple Ports (connection points) of Components
  - ✓ Connector abstracts interaction protocol and intermediaries
  - ✓ Port abstracts internal structure as connection point
  - ✓ Architecture style defines set of port / connector types
- Ports and connectors provide a thinking / design tool
  - ✓ Implementation is considerably more complex
- Dynamic assembly requires objectified port / connector
  - ✓ Alternately, some form of reflective access to components
- Frameworks provide succinct application of all the above

Outline

- Precise component specifications in UML
  - ✓ Interfaces
- Refinement
  - ✓ Component implementation refines spec
- What is Software Architecture?
  - ✓ Common definitions and examples
  - ✓ Catalysis definition of architecture
  - ✓ Specifying architectural styles using OCL
  - ✓ Generating architectural styles using Frameworks
  - ✓ Specifying architectural styles with Stereotypes
- **Specifying** component architectural styles
  - ✓ Components, ports, connectors and assemblies
  - ✓ Static assemblies
  - ✓ Dynamic assemblies
- **Realizing component architectural styles**
  - ✓ **The CORBA component model**
- Conclusion

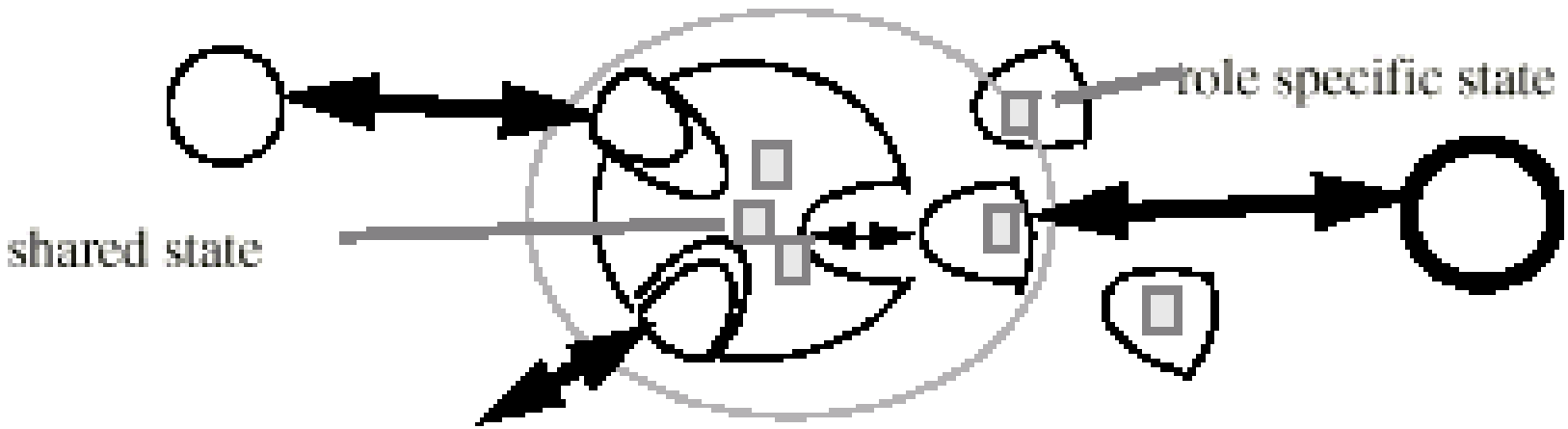
CORBA component model

- Components
- Ports
  - ✓ events sources and sinks - uses cookies instead of object-ids
  - ✓ attributes (properties)
  - ✓ facets - provided interfaces
  - ✓ receptacles - required interfaces
- Connectors
  - ✓ event emission and consumption (unicast) - static assembly
  - ✓ event publication and subscription (multicast) - dynamic assembly
  - ✓ facet and receptacle connection
  - ✓ varieties of attribute connection
- Homes - component managers
- Services - transactions, persistence, security etc.

Facet/Receptacle Ports and Connector

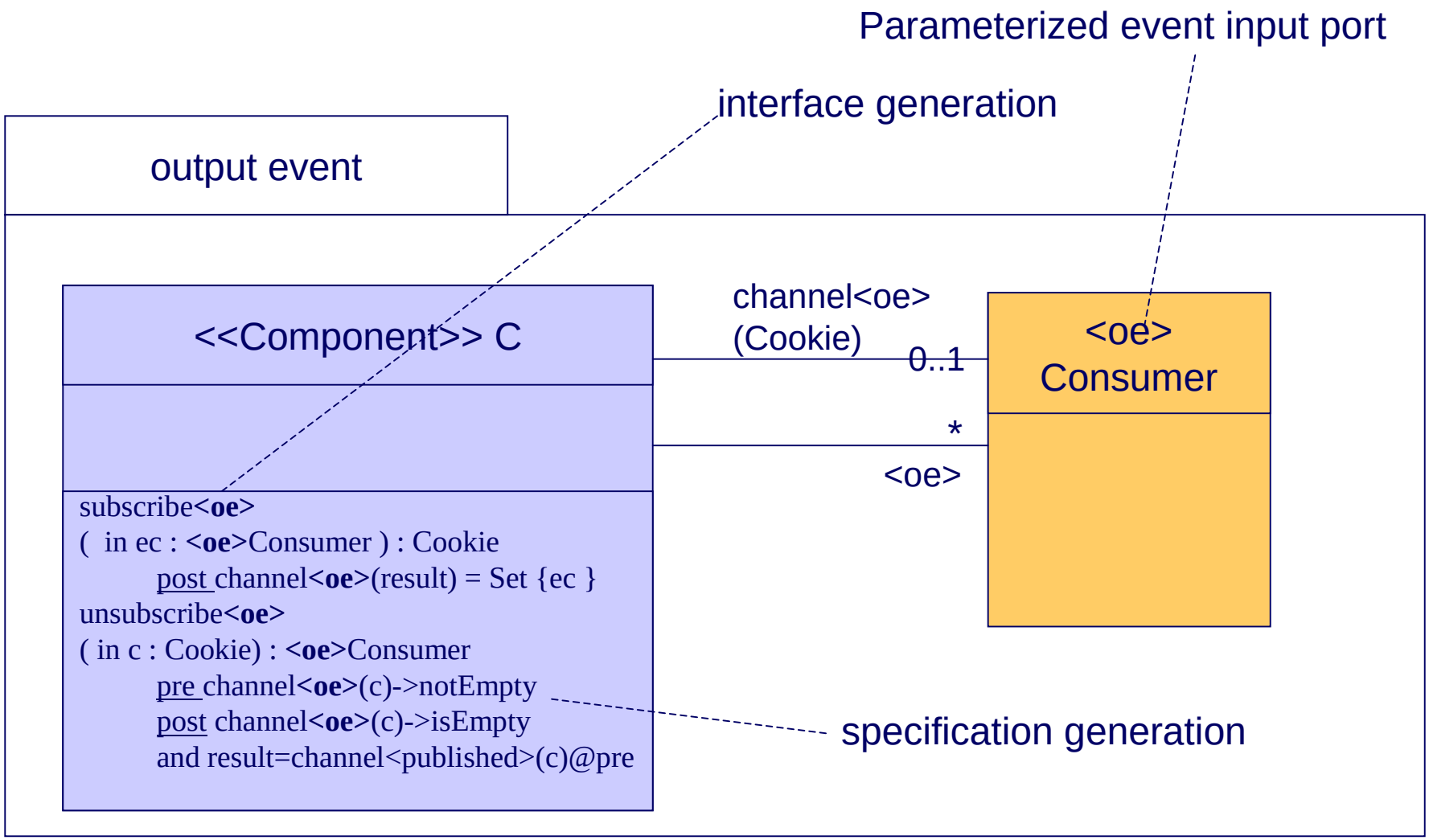
inv a.r = b.f

Facets as role objects:

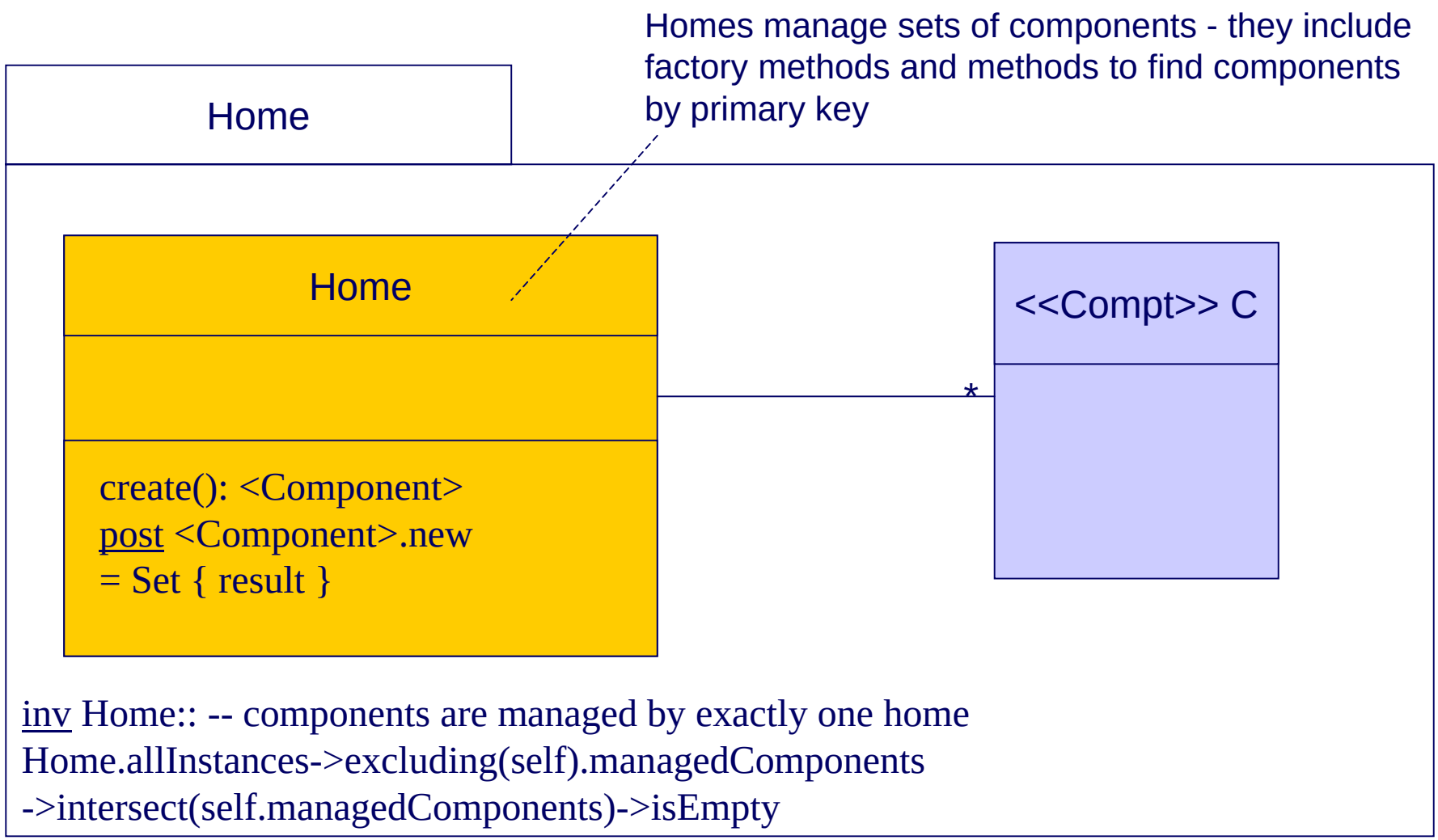


Each role (facet) is an observer of the shared state

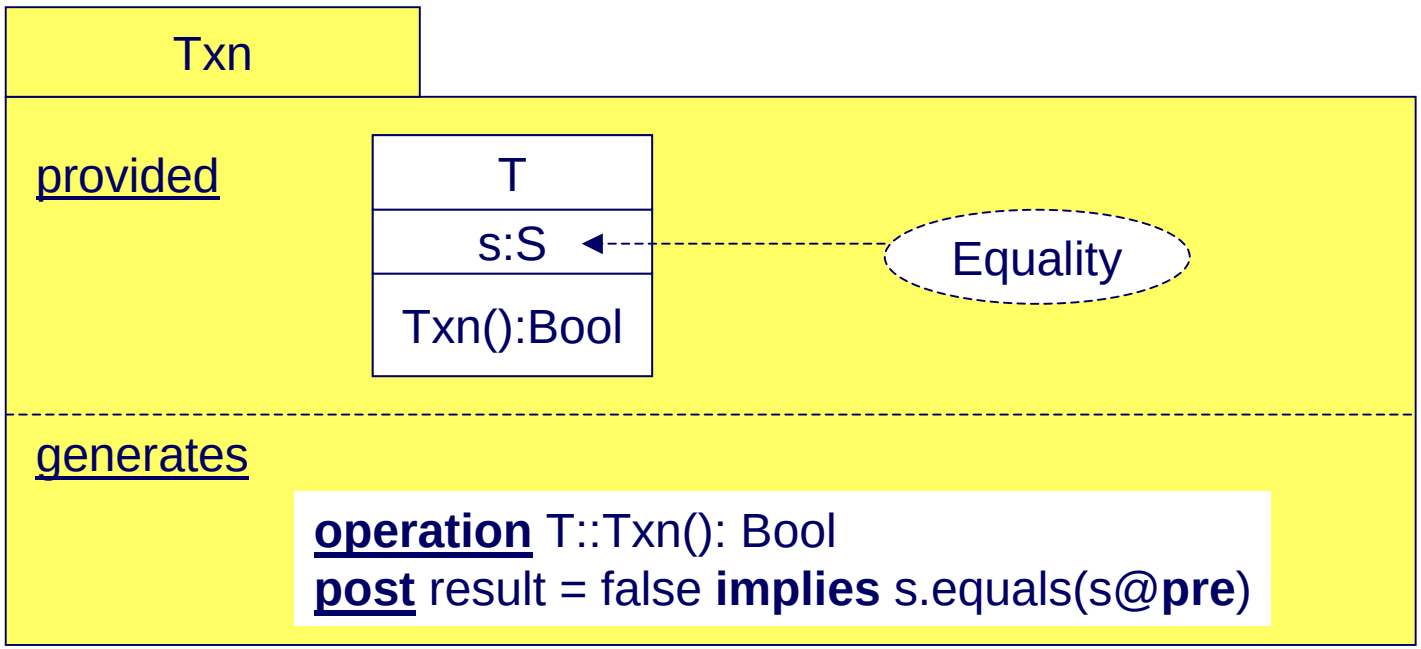
Events CCM realization



CCM Homes

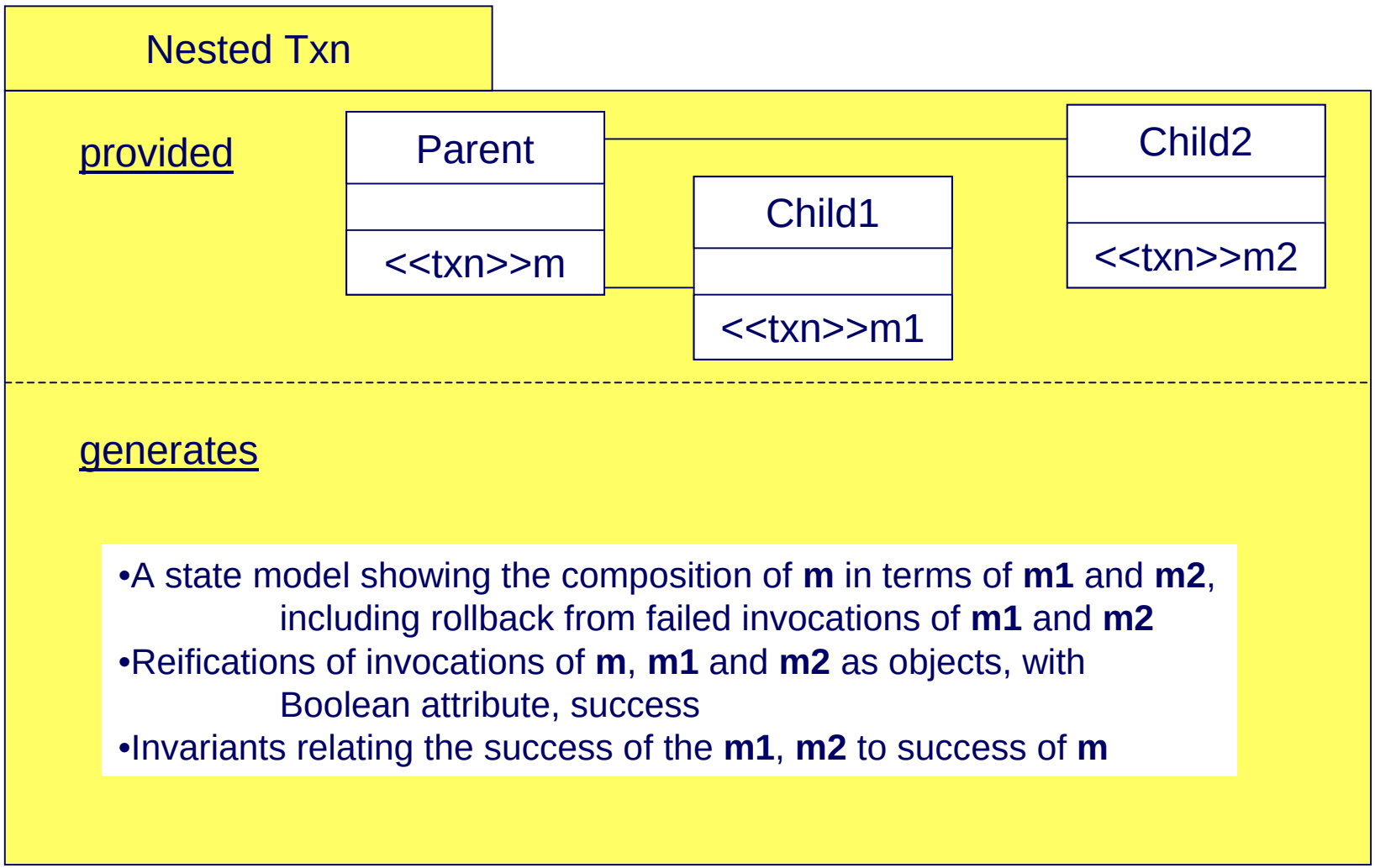

Homes manage sets of components - they include factory methods and methods to find components by primary key
Home
Home
create(): <Component>
post <Component>.new
= Set { result }
<<Compt>> C
*
inv Home:: -- components are managed by exactly one home
Home.allInstances->excluding(self).managedComponents
->intersect(self.managedComponents)->isEmpty

Transactions and Txn Policies

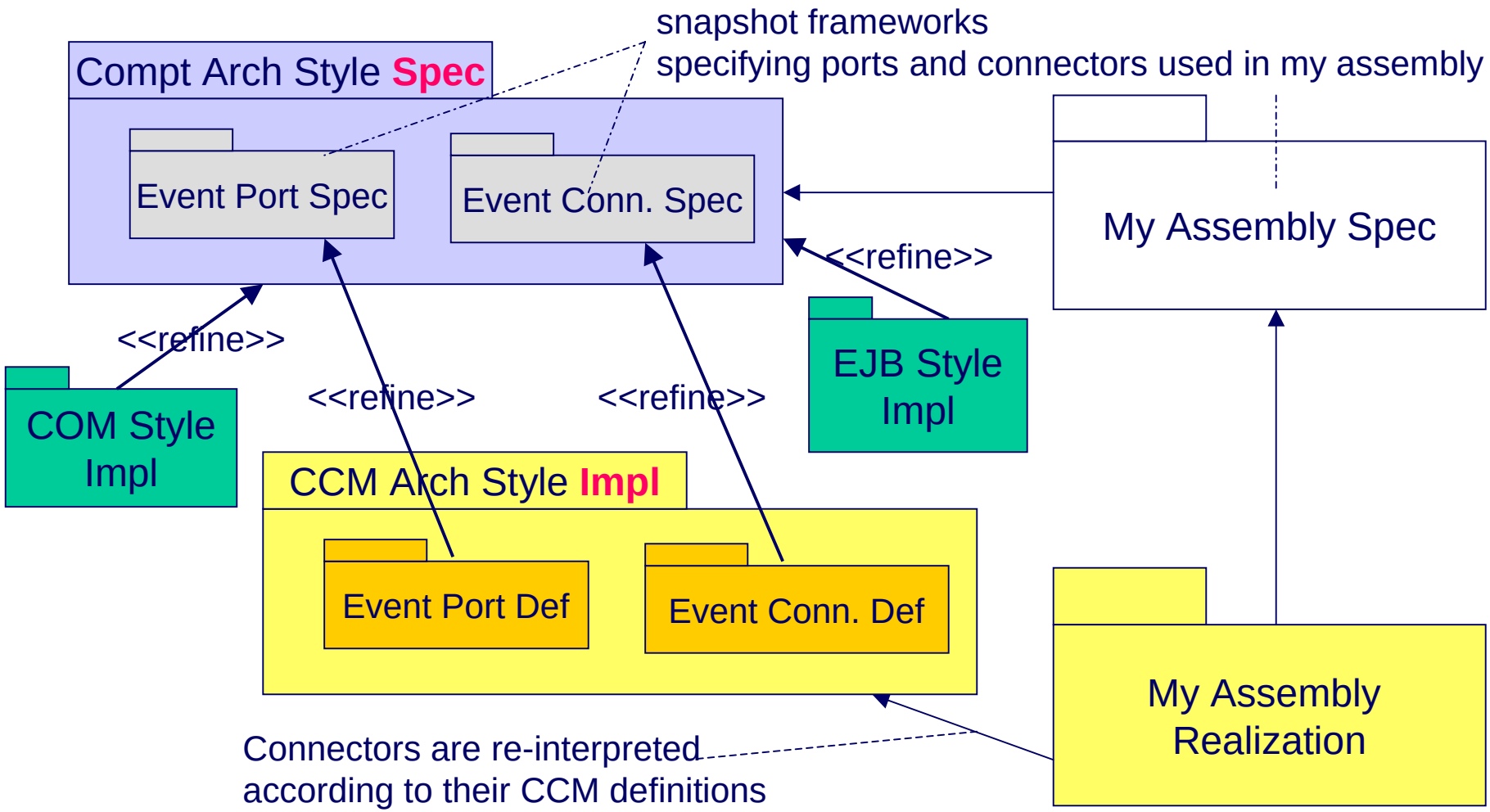


Policies: transaction required, transaction supported etc.

Nested Transactions



Styles Vs. Style Implementations



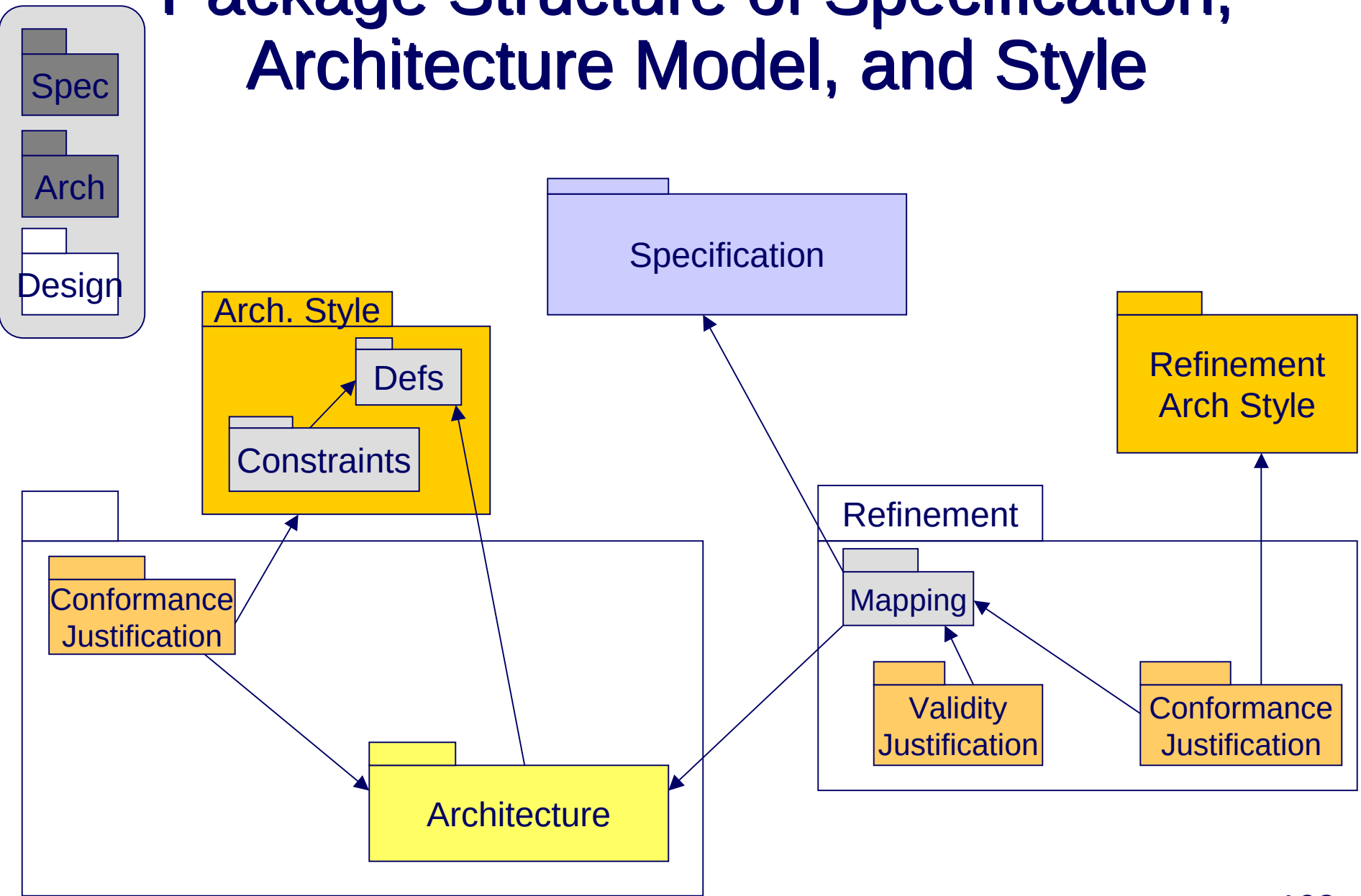
Section Summary - Style Realization

- An architectural style can separately define:
  - ✓ Design constructs and their **specification**
  - ✓ Corresponding **implementations** of those constructs
- Development uses those style constructs to specify
  - ✓ And separately uses their implementations

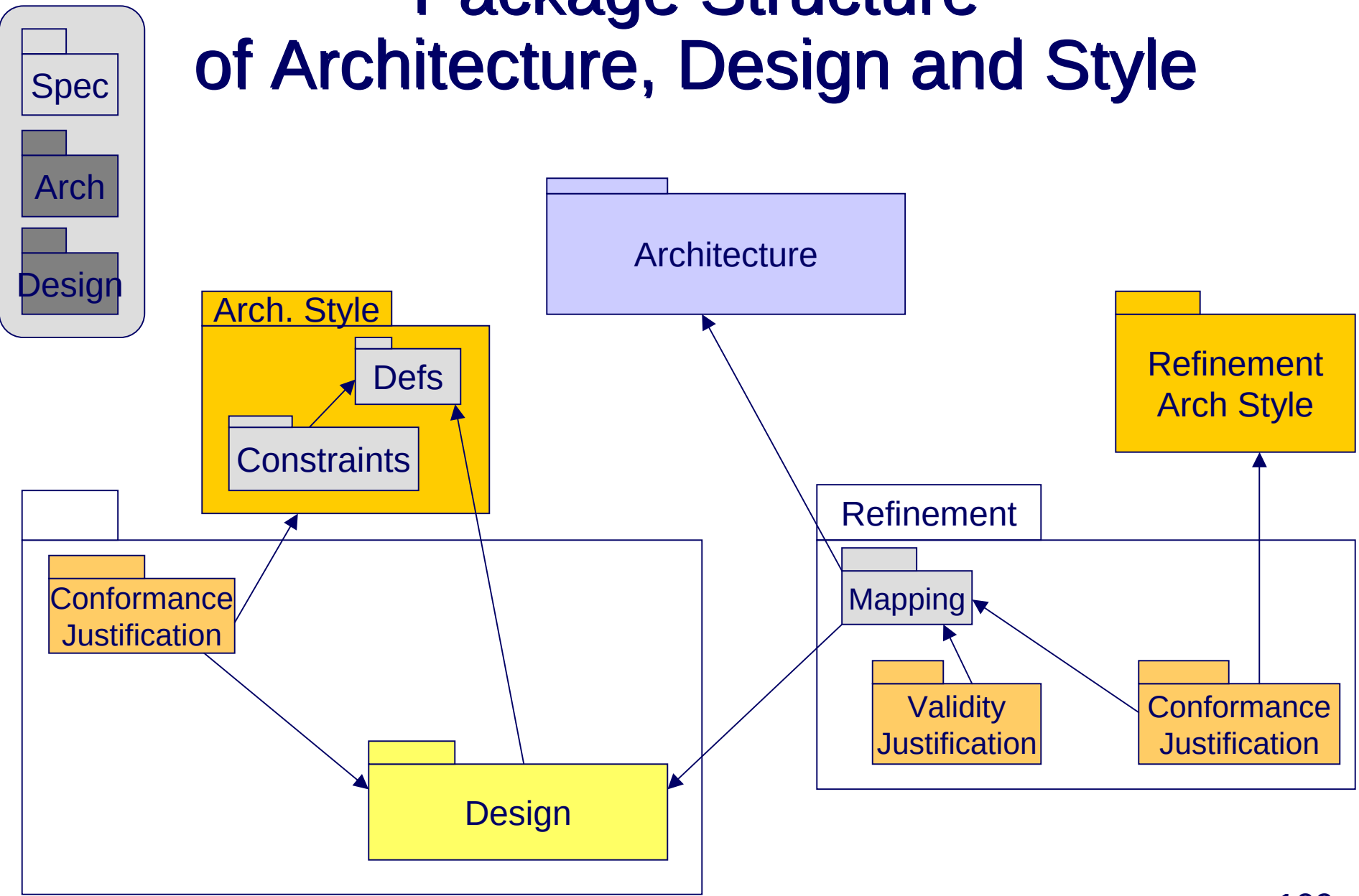
Outline

- Precise component specifications in UML
  - ✓ Interfaces
- Refinement
  - ✓ Component implementation refines spec
- What is Software Architecture?
  - ✓ Common definitions and examples
  - ✓ Catalysis definition of architecture
  - ✓ Specifying architectural styles using OCL
  - ✓ Generating architectural styles using Frameworks
  - ✓ Specifying architectural styles with Stereotypes
- Specifying component architectural styles
  - ✓ Components, ports, connectors and assemblies
  - ✓ Static assemblies
  - ✓ Dynamic assemblies
- Realizing component architectural styles
  - ✓ The CORBA component model
- **Conclusion**

Package Structure of Specification, Architecture Model, and Style



Package Structure of Architecture, Design and Style



Conclusion

- Form vs. Meaning
  - ✓ A **specification** constrains the meaning of its realization
  - ✓ An **architectural style** constrains the form (description) of its realization
    - ✓ An architectural style may be defined relative to a specification
    - ✓ Styles + style implementations together define a refinement architectural style.
  - ✓ A **refinement architectural style** constrains the form of the refinement mapping between a realization and a specification
    - ✓ A **compiler** is a special case of a refinement architectural style that generates a unique conforming realization

Conclusion

- Architectural styles can be documented as:
  - ✓ A set of constraints, possibly formalized using OCL
  - ✓ A collection of model elements with a set of constraints on their use
  - ✓ A collection of stereotype definitions, together with a constraint such as all model elements must be a stereotype defined in the style
- A component architecture can be defined as a collection of frameworks

References

- M. Shaw and D. Garlan, “Software Architecture - Perspectives on an Emerging Discipline”, Prentice Hall, 1996.
- UML 1.3 specifications (http://www.omg.org)
- D. D’Souza and A. Wills ‘Objects, Components and Frameworks with UML: the Catalysis approach’, Addison Wesley 1998.
- Catalysis overviews and discussions at http://www.catalysis.org
- L. Bass et al, “Software Architecture in Practice”, Addison-Wesley 1998