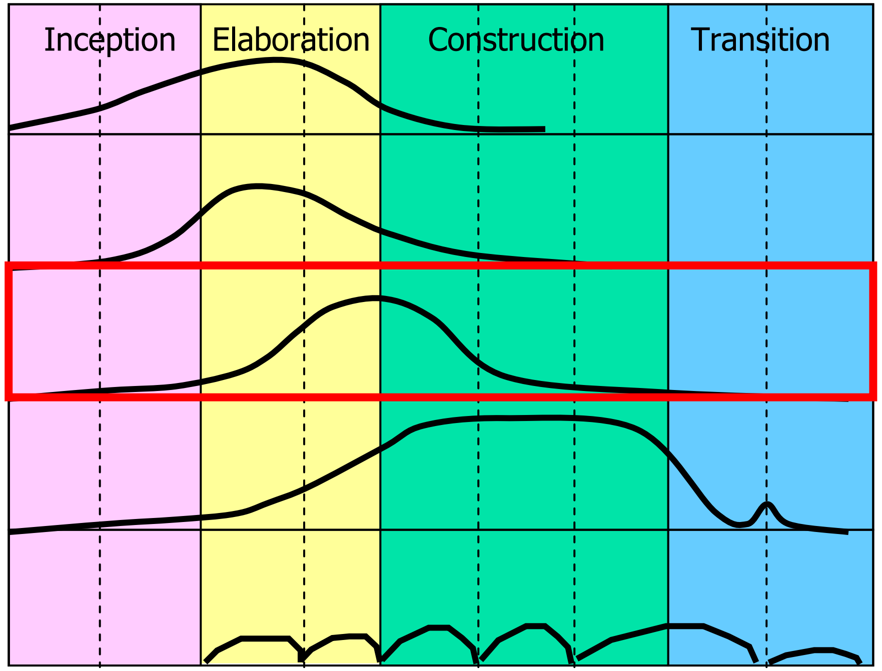
Design - introduction

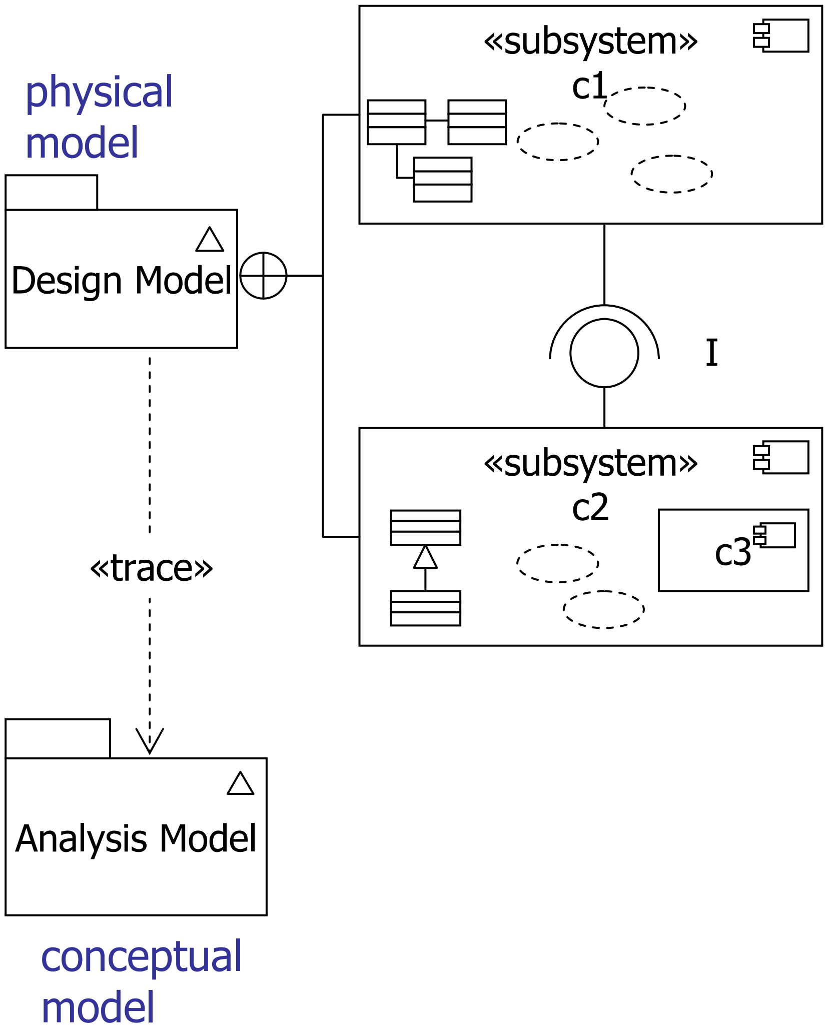
Design - purpose

- Decide how the system's functions are to be implemented
- Decide on strategic design issues such as persistence, distribution etc.
- Create policies to deal with tactical design issues



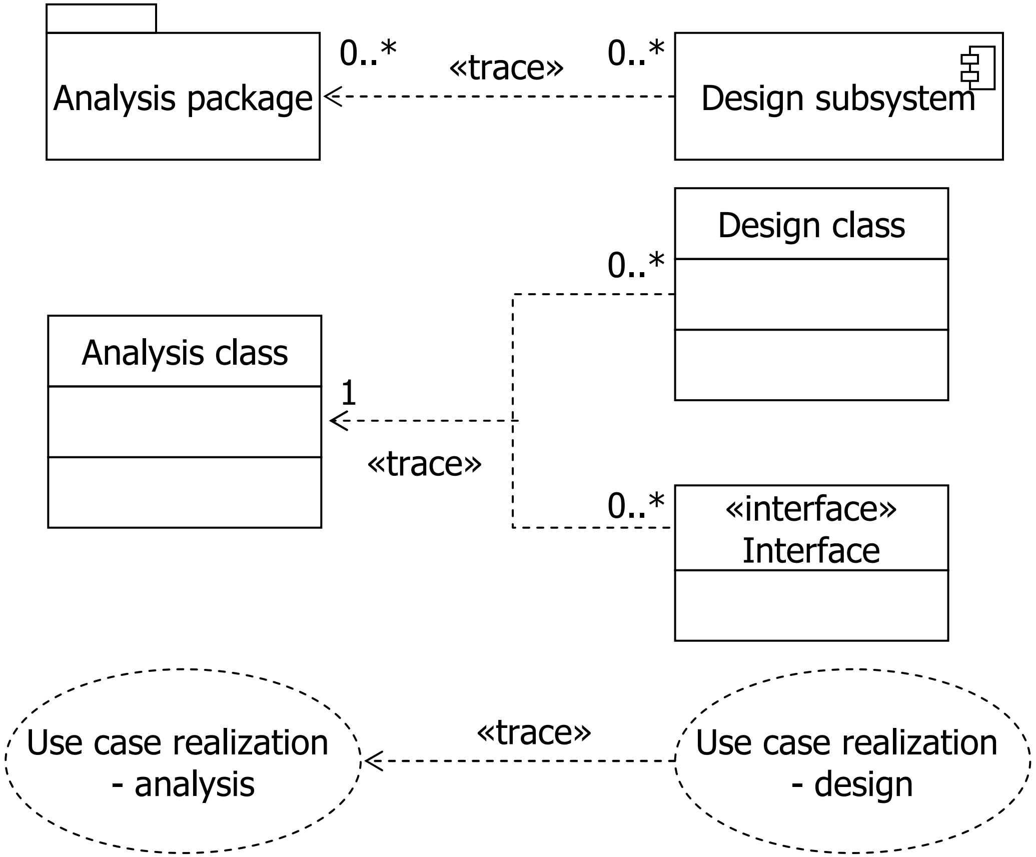
Design artifacts - metamodel

- Subsystems are components that contain UML elements
- We create the design model from the analysis model by adding implementation details
- There is a historical «trace» relationship between the two models



Artifact trace relationships

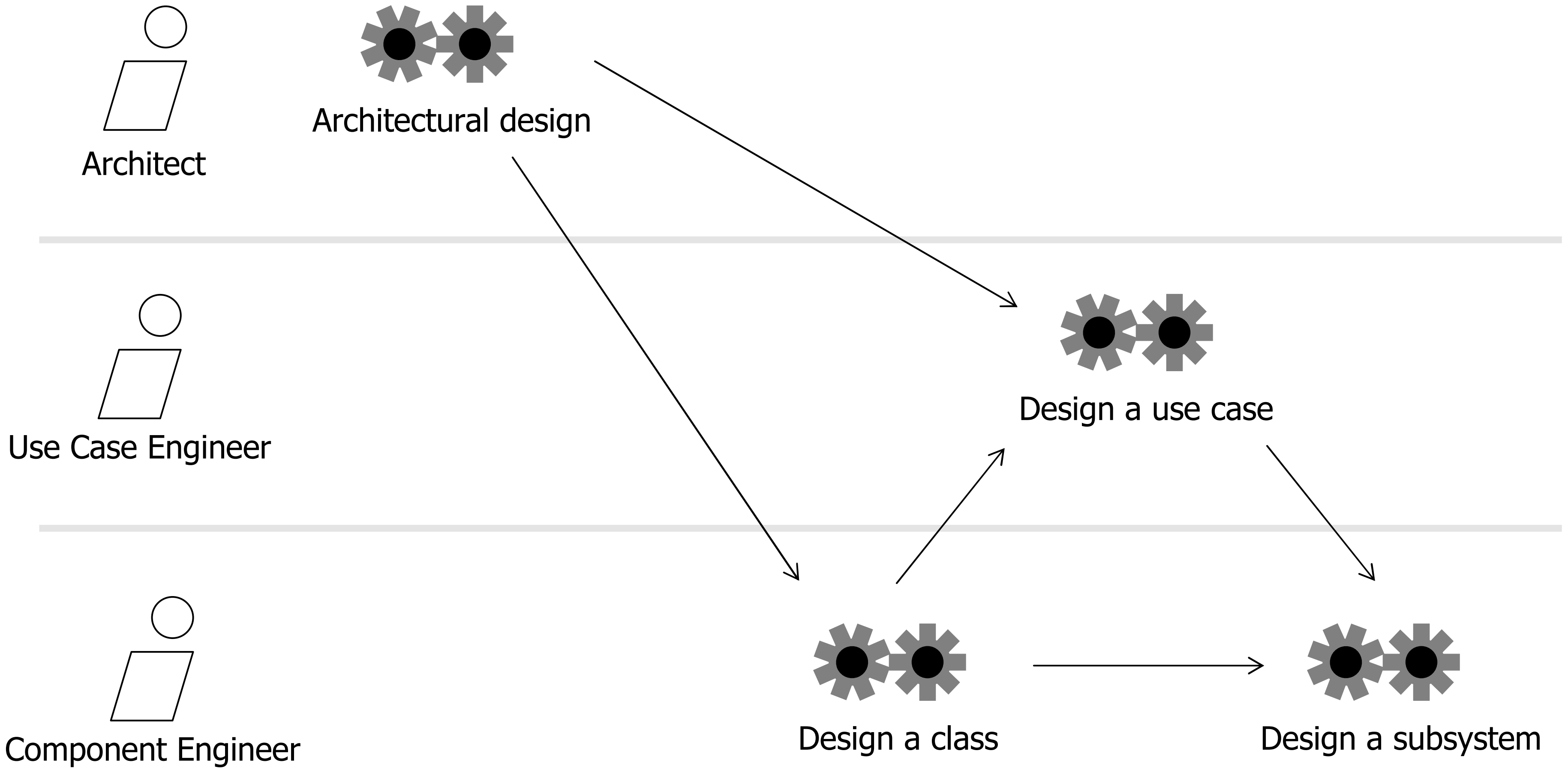
- Design model
  - Design subsystem
  - Design class
  - Interface
  - Use case realization – design
- Deployment model



Should you maintain 2 models?

- A design model may contain 10 to 100 times as many classes as the analysis model
  - The analysis model helps us to see the big picture without getting lost in implementation details
- We need to maintain 2 models if:
  - It is a big system (>200 design classes)
  - It has a long expected lifespan
  - It is a strategic system
  - We are outsourcing construction of the system
- We can make do with only a design model if:
  - It is a small system
  - It has a short lifespan
  - It is not a strategic system

Workflow - Design



Summary

- Design is the primary focus in the last part of the elaboration phase and the first half of the construction phase
- Purpose – to decide *how* the system's functions are to be implemented
- artifacts:
  - Design classes
  - Interfaces
  - Design subsystems
  - Use case realizations – design
  - Deployment model

Design - classes

What are design classes?

- Design classes are classes whose specifications have been completed to such a degree that they can be implemented
  - Specifies an actual piece of code
- Design classes arise from analysis classes:
  - Remember - analysis classes arise from a consideration of the problem domain *only*
  - A refinement of analysis classes to include implementation details
  - One analysis class may become many design classes
  - All attributes are completely specified including type, visibility and default values
  - Analysis operations become fully specified operations (methods) with a return type and parameter list
- Design classes arise from the solution domain
  - Utility classes – String, Date, Time etc.
  - Middleware classes – database access, comms etc.
  - GUI classes – Applet, Button etc.

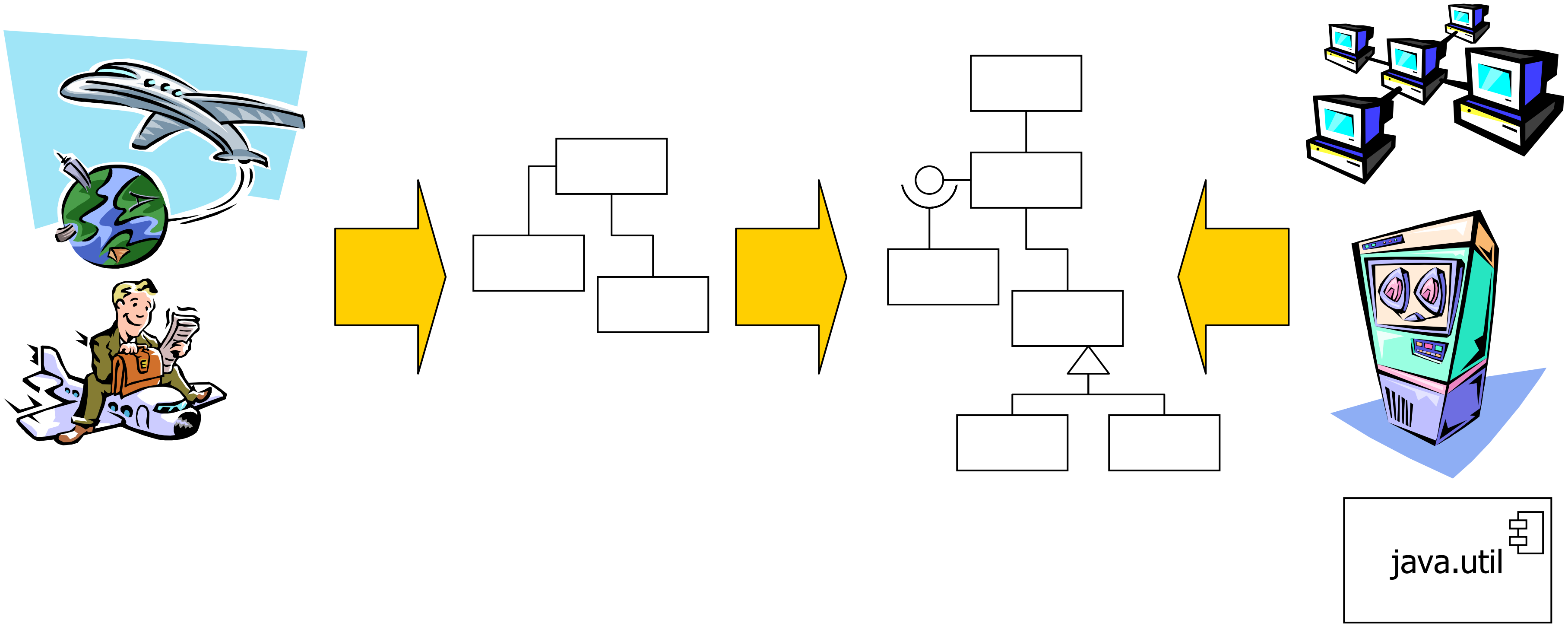
Sources of design classes

Problem domain

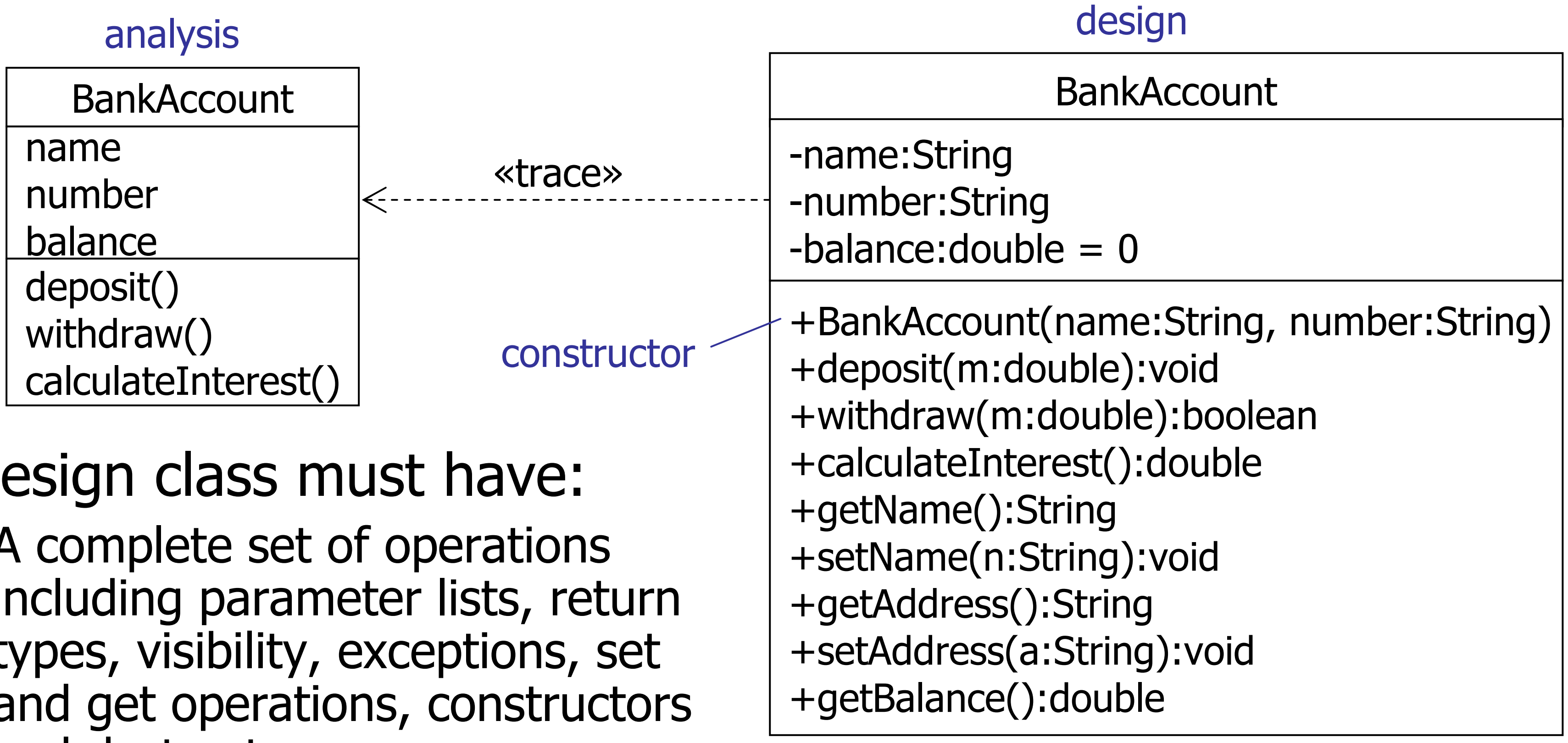
Analysis classes

Design classes

Solution domain



Anatomy of a design class



analysis

BankAccount
name
number
balance
deposit()
withdraw()
calculateInterest()

«trace»

design

BankAccount
-name:String
-number:String
-balance:double = 0
+BankAccount(name:String, number:String)
+deposit(m:double):void
+withdraw(m:double):boolean
+calculateInterest():double
+getName():String
+setName(n:String):void
+getAddress():String
+setAddress(a:String):void
+getBalance():double

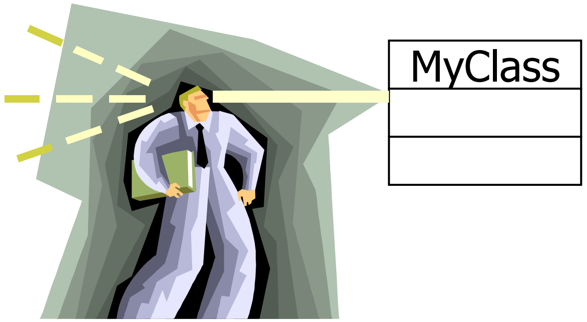
constructor

- A design class must have:
  - A complete set of operations including parameter lists, return types, visibility, exceptions, set and get operations, constructors and destructors
  - A complete set of attributes including types and default values

Well-formed design classes

- Design classes must have the following characteristics to be "well-formed":
  - Complete and sufficient
  - Primitive
  - High cohesion
  - Low coupling

How do the users of your classes see them?
Always look at *your* classes from *their* point of view!



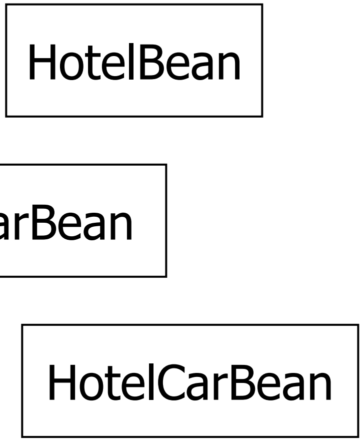
Completeness, sufficiency and primitiveness

- Completeness:
  - Users of the class will make assumptions from the class name about the set of operations that it should make available
  - For example, a BankAccount class that provides a withdraw() operation will be expected to also provide a deposit() operation!
- Sufficiency:
  - A class should never surprise a user – it should contain exactly the expected set of features, no more and no less
- Primitiveness:
  - Operations should be designed to offer a single primitive, atomic service
  - A class should never offer multiple ways of doing the same thing:
    - This is confusing to users of the class, leads to maintenance burdens and can create consistency problems
  - For example, a BankAccount class has a primitive operation to make a single deposit. It should *not* have an operation that makes two or more deposits as we can achieve the same effect by repeated application of the primitive operation

The public members of a class define a "contract" between the class its clients

High cohesion, low coupling

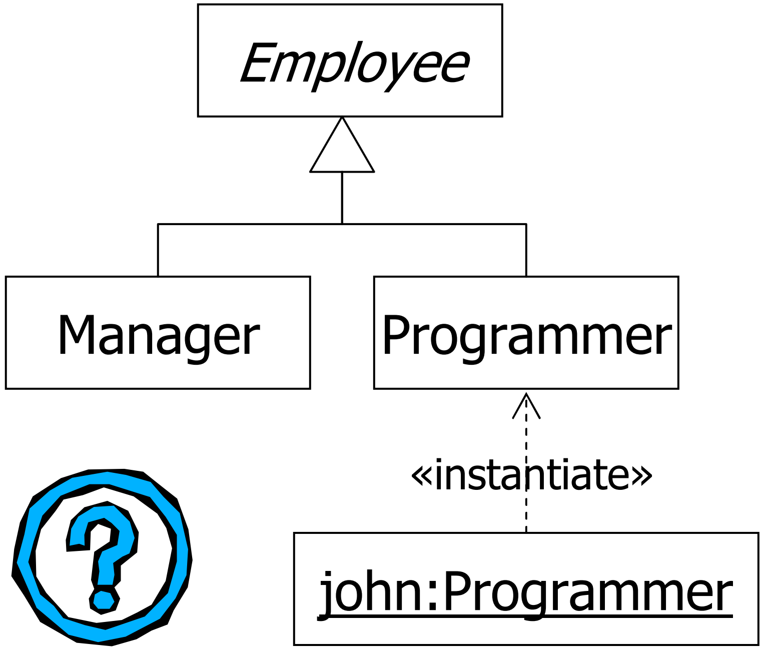
- High cohesion:
  - Each class should have a set of operations that support the intent of the class, no more and no less
  - Each class should model a single abstract concept
  - If a class needs to have many responsibilities, then some of these should be implemented by "helper" classes. The class then delegates to its helpers
- Low coupling:
  - A particular class should be associated with just enough other classes to allow it to realise its responsibilities
  - Only associate classes if there is a true semantic link between them
  - Never form an association just to reuse a fragment of code in another class!
  - Use aggregation rather than inheritance (next slide)



this example comes from a real system! What's wrong with it?

Aggregation vs. inheritance

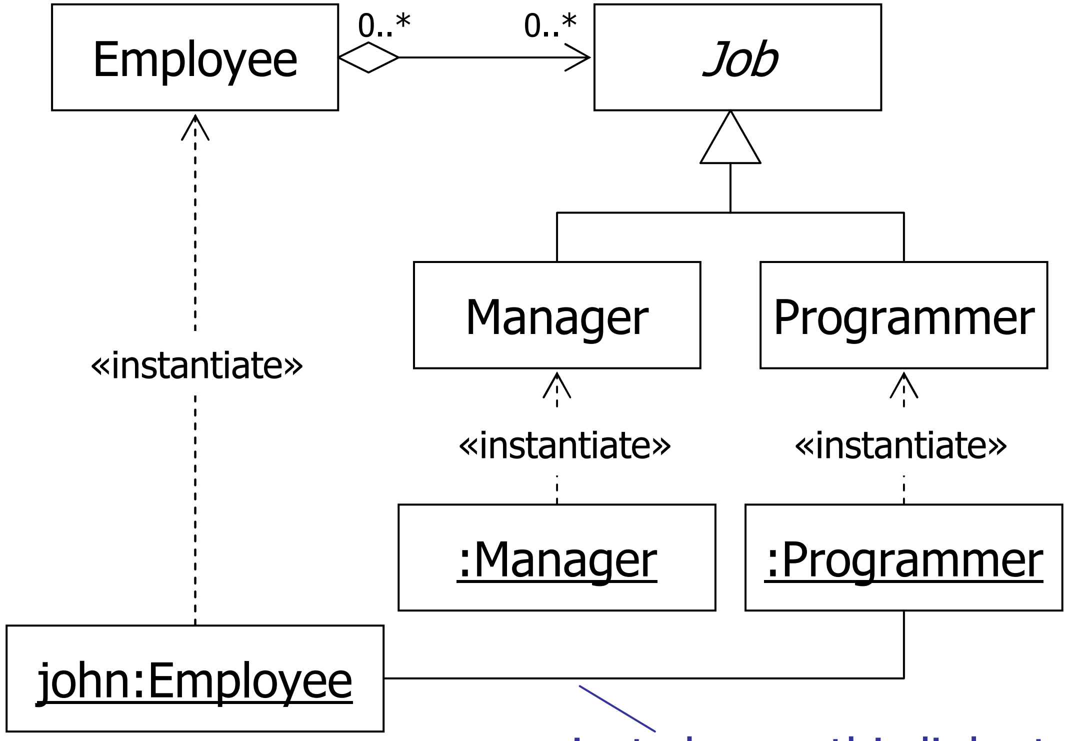
- Inheritance gives you fixed relationships between classes and objects
- You *can't* change the class of an object at runtime
- There is a fundamental semantic error here. Is an Employee *just* their job or does an Employee *have* a job?



1. How can we promote john?
2. Can john have more than one job?

A better solution…

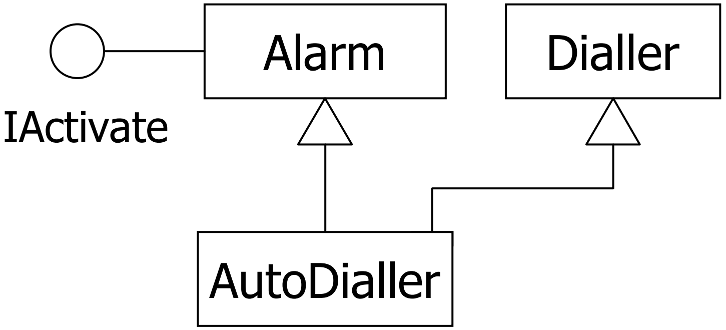
- Using aggregation we get the correct semantics:
  - An *Employee has* a Job
- With this more flexible model, Employees can have more than one Job



just change this link at runtime to promote john!

Multiple inheritance

- Sometimes a class may have more than one superclass
- The "is kind of" and substitutability principles must apply for *all* of the classifications
- Multiple inheritance is sometimes the most elegant way of modelling something. However:
  - Not all languages support it (e.g. Java)
  - It can always be replaced by single inheritance and delegation



in this example the AutoDialler sounds an alarm and rings the police when triggered - it is logically both a *kind of* Alarm *and* a *kind of* Dialler

Inheritance vs. interface realization

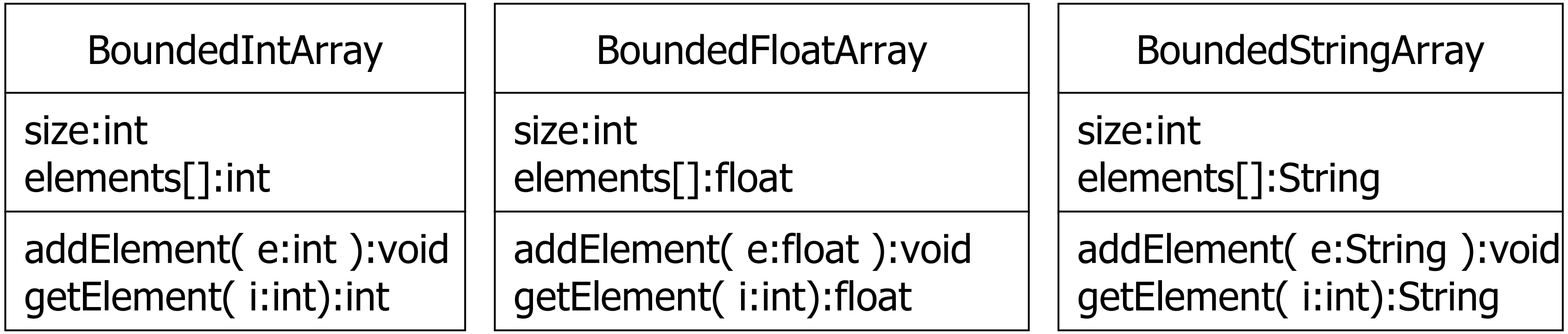
- With inheritance we get two things:
  - Interface – the public operations of the base classes
  - Implementation – the attributes, relationships, protected and private operations of the base classes
- With interface realization we get exactly one thing:
  - An interface – a set of public operations, attributes and relationships that have no implementation

Use inheritance when we want to *inherit implementation*.
Use interface realization when we want to *define a contract*.

Templates

- Up to now, we have had to specify the types of all attributes, method returns and parameters. However, this can be a barrier to reuse
- Consider:

spot the difference!



BoundedIntArray

size:int
elements[]:int

addElement(e:int):void
getElement(i:int):int

BoundedFloatArray

size:int
elements[]:float

addElement(e:float):void
getElement(i:int):float

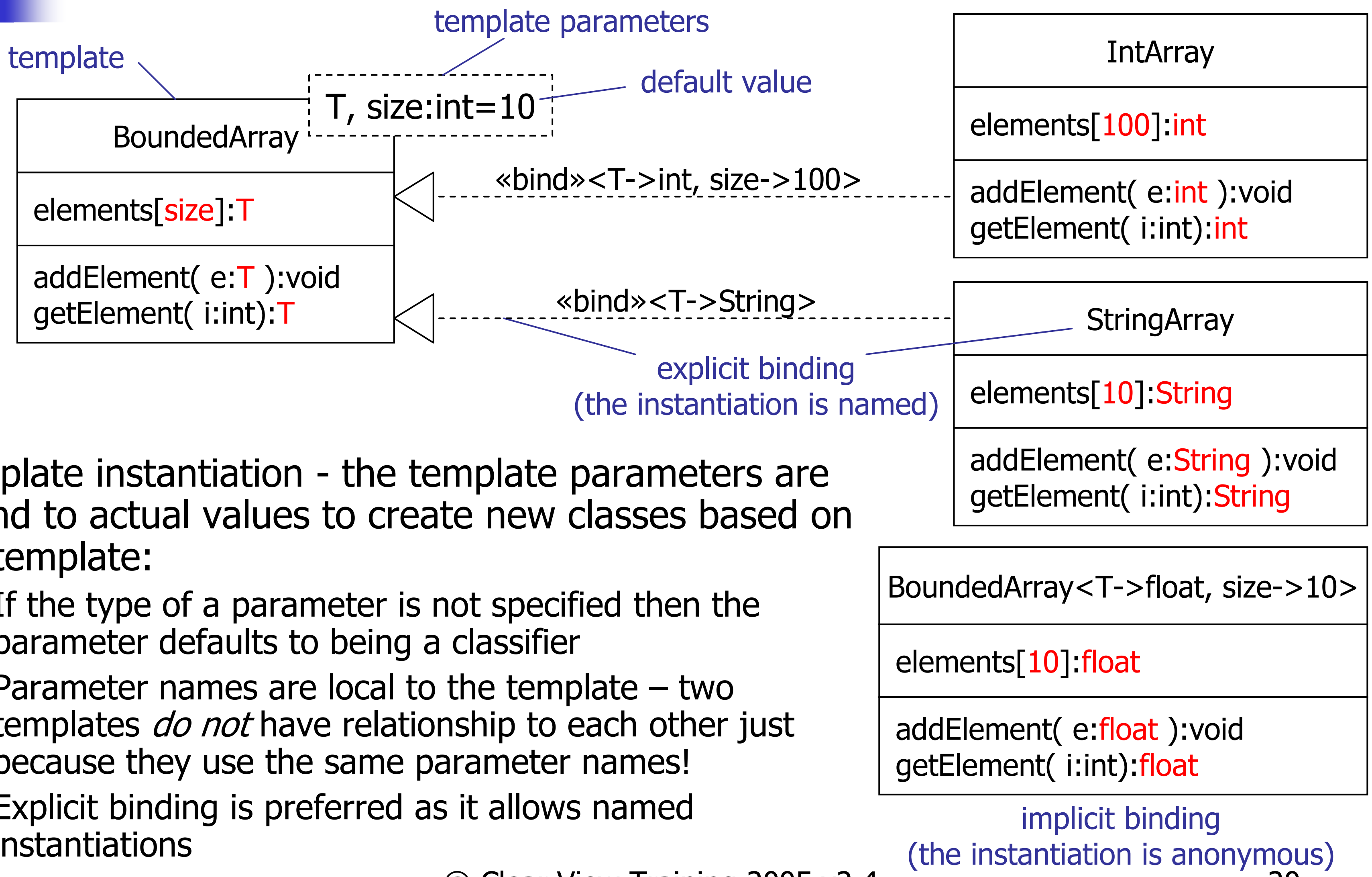
BoundedStringArray

size:int
elements[]:String

addElement(e:String):void
getElement(i:int):String

etc.

Template syntax



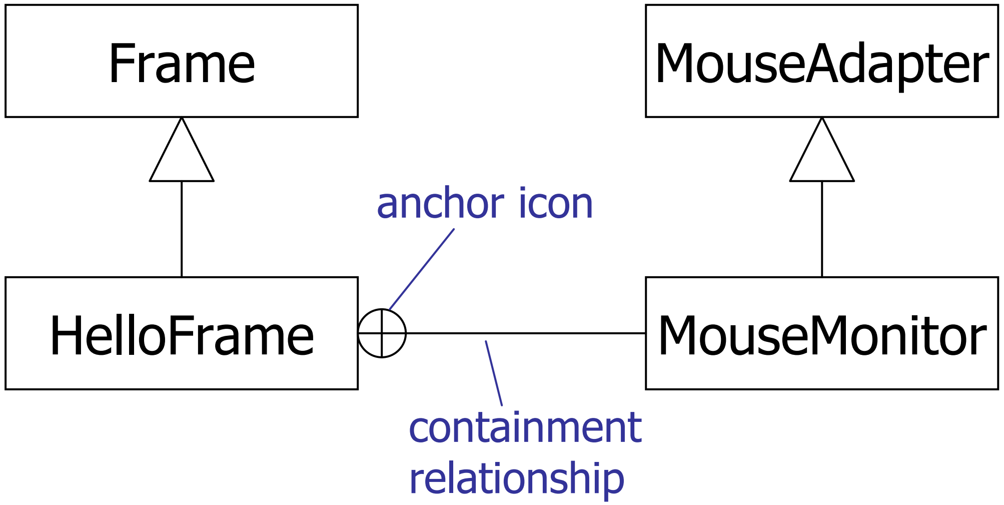
- Template instantiation - the template parameters are bound to actual values to create new classes based on the template:
  - If the type of a parameter is not specified then the parameter defaults to being a classifier
  - Parameter names are local to the template – two templates *do not* have relationship to each other just because they use the same parameter names!
  - Explicit binding is preferred as it allows named instantiations

Templates & multiple inheritance

- Templates and multiple inheritance should only be used in design models where those features are available in the target language:

language	templates	multiple inheritance
C#	Yes	No
Java	Yes	No
C++	Yes	Yes
Smalltalk	No	No
Visual Basic	No	No
Python	No	Yes

Nested classes



- A nested class is a class defined inside another class
  - It is encapsulated inside the namespace of its containing class
  - Nested classes tend to be design artifacts
- Nested classes are only accessible by:
  - their containing class
  - objects of that their containing class

Summary

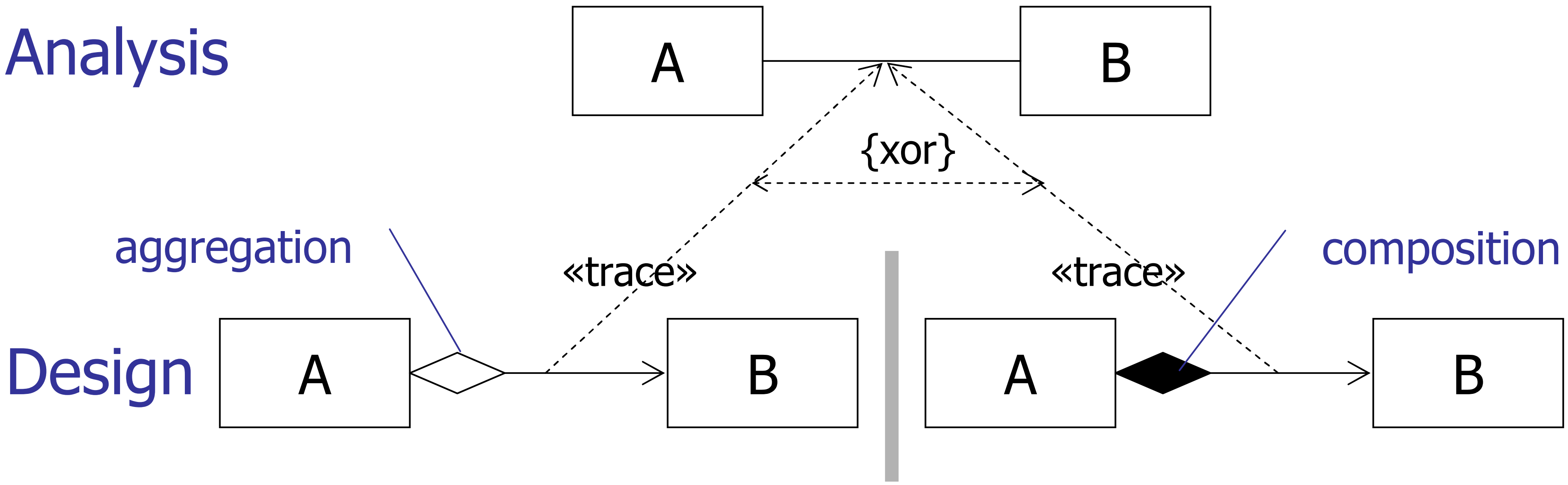
- Design classes come from:
  - A refinement of analysis classes (i.e. the business domain)
  - From the solution domain
- Design classes must be well-formed:
  - Complete and sufficient
  - Primitive operations
  - High cohesion
  - Low coupling
- Don’t overuse inheritance
  - Use inheritance for "is kind of"
  - Use aggregation for "is role played by"
  - Multiple inheritance should be used sparingly (mixins)
  - Use interfaces rather than inheritance to define contracts
- Use templates and nested classes only where the target language supports them

Design - refining analysis relationships

Design relationships

- Refining analysis associations to design associations involves several procedures:
  - refining associations to aggregation or composition relationships where appropriate
  - implementing one-to-many associations
  - implementing many-to-one associations
  - implementing many-to-many associations
  - implementing bidirectional associations
  - implementing association classes
- All design associations must have:
  - navigability
  - multiplicity on both ends

Aggregation and composition

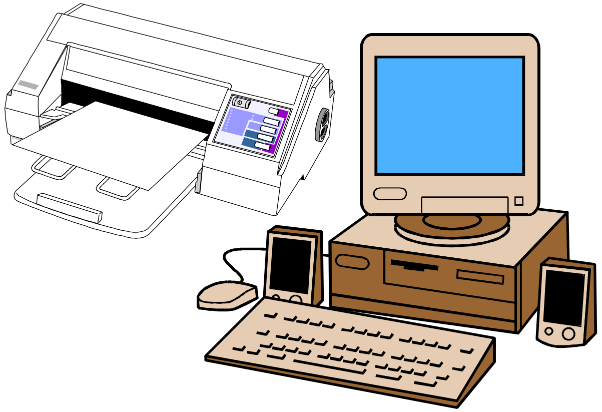


- In analysis, we often use unrefined associations. In design, these can become aggregation or composition relationships
- We must also add navigability, multiplicity and role names

Aggregation and composition

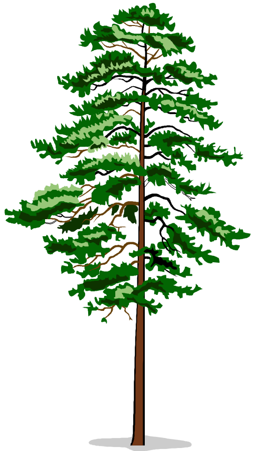
UML defines two types of association:

Aggregation



Some objects are weakly related like a computer and its peripherals

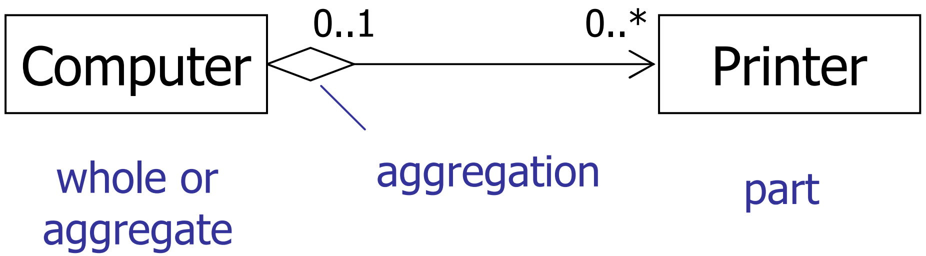
Composition



Some objects are strongly related like a tree and its leaves

Aggregation semantics

aggregation is a *whole–part* relationship



A Computer may be attached to 0 or more Printers

At any one point in time a Printer is connected to 0 or 1 Computer

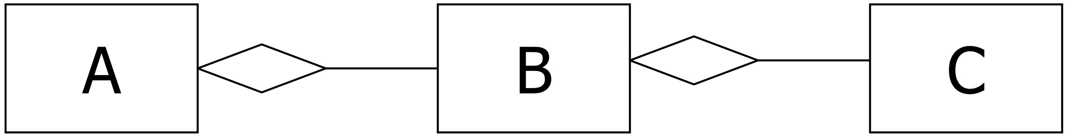
Over time, many Computers may use a given Printer

The Printer exists even if there are no Computers

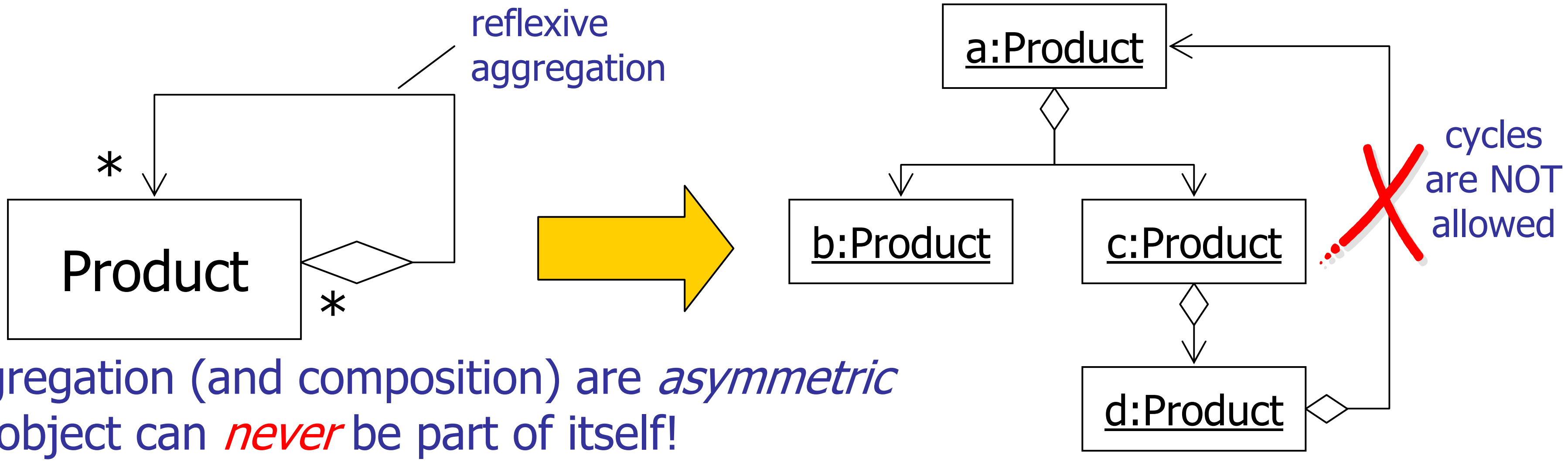
The Printer is independent of the Computer

- The aggregate can sometimes exist independently of the parts, sometimes not
- The parts can exist independently of the aggregate
- The aggregate is in some way incomplete if some of the parts are missing
- It is possible to have shared ownership of the parts by several aggregates

Transitive and asymmetric

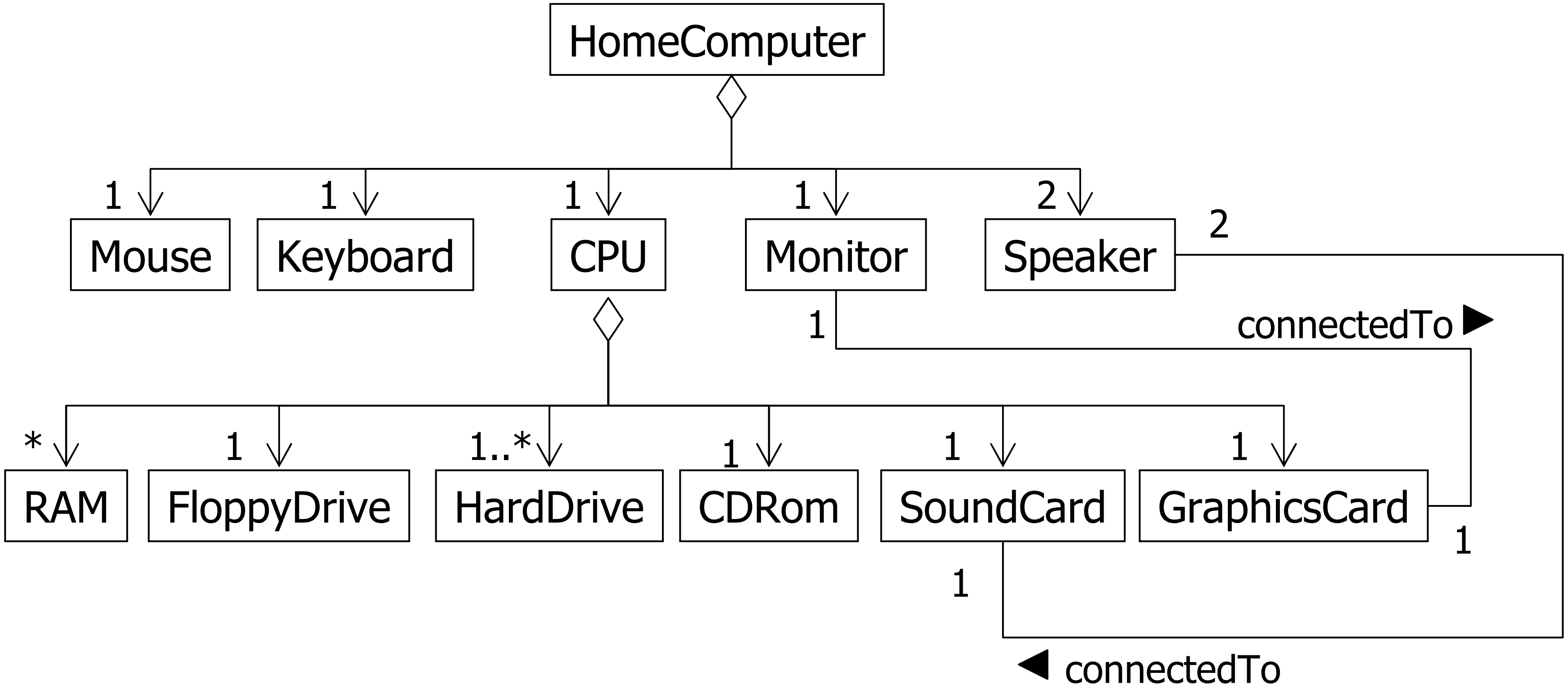


Aggregation (and composition) are *transitive*
If C is a part of B and B is a part of A, then C is a part of A



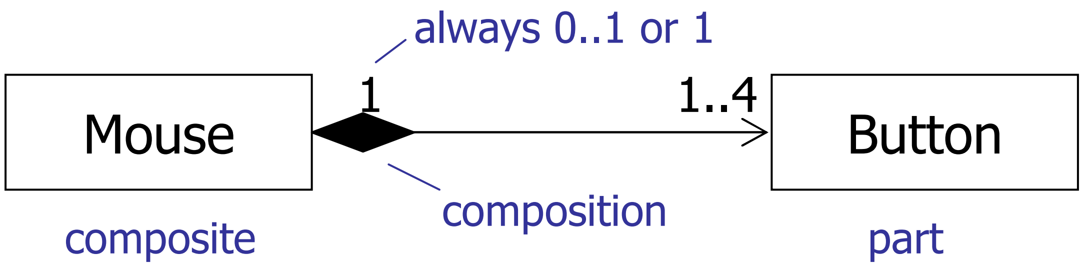
Aggregation (and composition) are *asymmetric*
An object can *never* be part of itself!

Aggregation hierarchy



Composition semantics

composition is a strong form of aggregation



The buttons have no independent existence. If we destroy the mouse, we destroy the buttons. They are an integral part of the mouse

Each button can belong to exactly 1 mouse

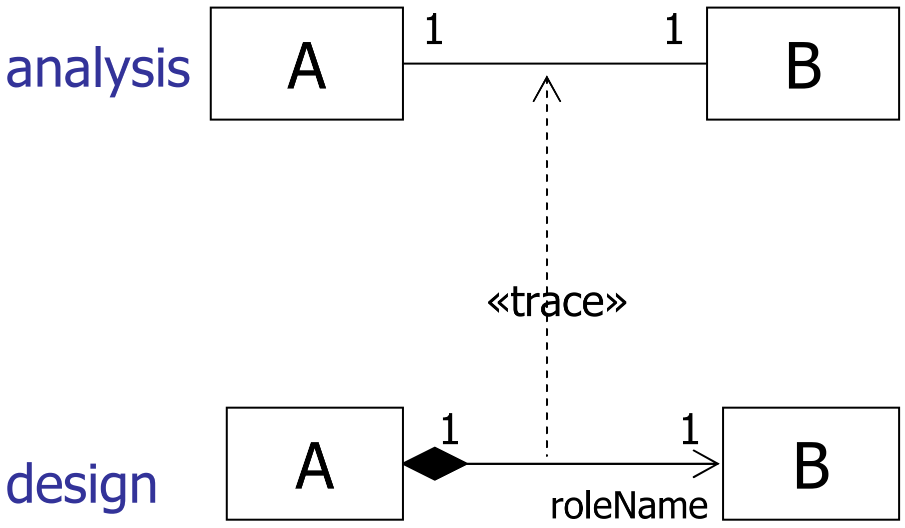
- The parts belong to exactly 0 or 1 whole at a time
- The composite has sole responsibility for the disposition of all its parts. This means responsibility for their creation and destruction
- The composite may also release parts provided responsibility for them is assumed by another object
- If the composite is destroyed, it must either destroy all its parts, OR give responsibility for them over to some other object
- Composition is transitive and asymmetric

Composition and attributes

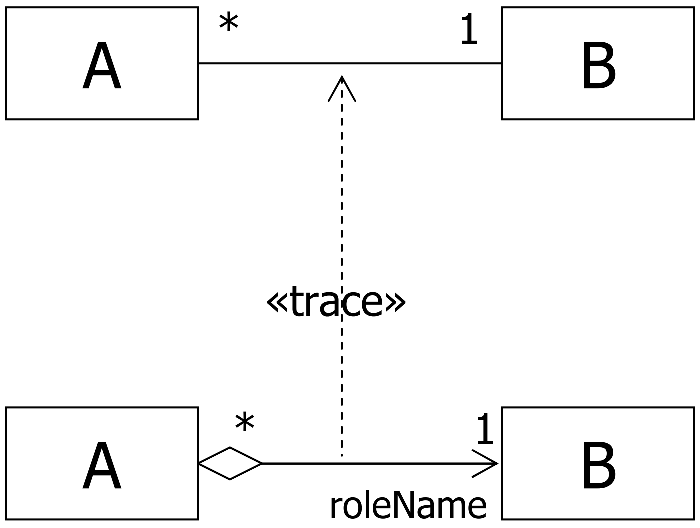
- Attributes are in effect composition relationships between a class and the classes of its attributes
- Attributes should be reserved for primitive data types (int, String, Date etc.) and **not** references to other classes

1 to 1 and many to 1 associations

1 to 1



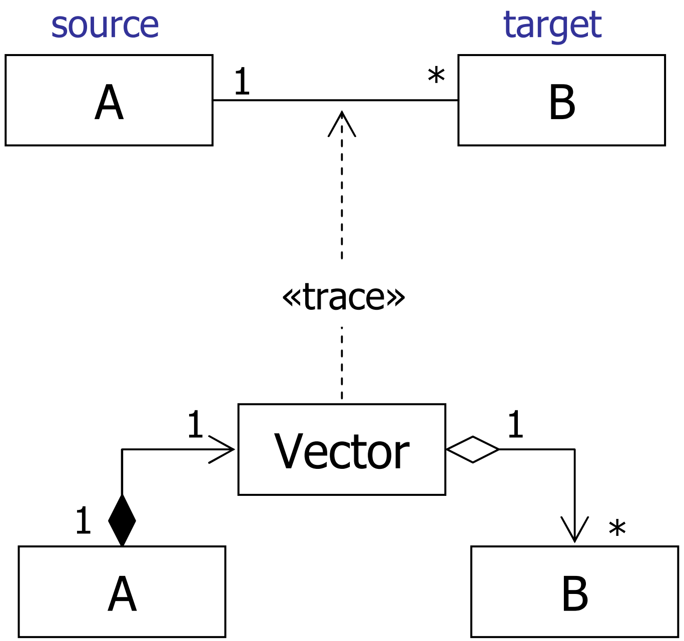
many to 1



- One-to-one associations in analysis *usually* imply single ownership and *usually* refine to compositions
- Many-to-one relationships in analysis imply shared ownership and are refined to aggregations

1 to many associations

- To refine 1-to-many associations we introduce a *collection class*
- Collection classes instances store a collection of object references to objects of the target class
- A collection class always has methods for:
  - Adding an object to the collection
  - Removing an object from the collection
  - Retrieving a reference to an object in the collection
  - Traversing the collection
- Collection classes are typically supplied in libraries that come as part of the implementation language
- In Java we find collection classes in the java.util library

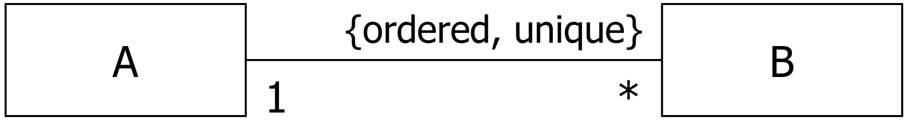


Collection semantics

- You can specify collection semantics by using association end properties:

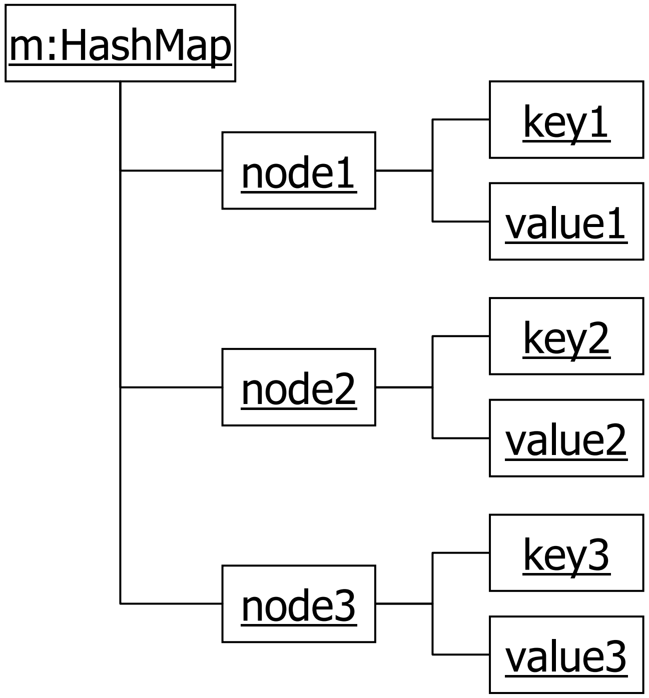
property	semantics
{ordered}	Elements in the collection are maintained in a strict order
{unordered}	There is no ordering of the elements in the collection
{unique}	Elements in the collection are all unique an object appears in the collection once
{nonunique}	Duplicate elements are allowed in the collection

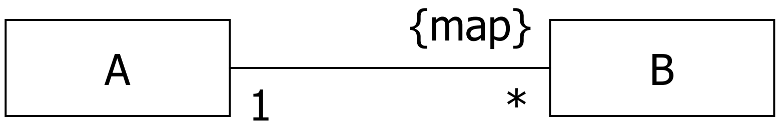
property pair	OCL collection
{unordered, nonunique}	Bag
{unordered, unique}	Set (default)
{ordered, unique}	OrderedSet
{ordered, nonunique}	Sequence



The Map

- Maps (also known as dictionaries) have no equivalent in OCL
- Maps usually work by maintaining a set of nodes
- Each node points to two objects – the "key" and the "value"
- Maps are optimised to find a value given a specific key
- They are a bit like a database table with only two columns, one of which is the primary key
- They are incredibly useful for storing any objects that must be accessed quickly using a key, for example customer details or products

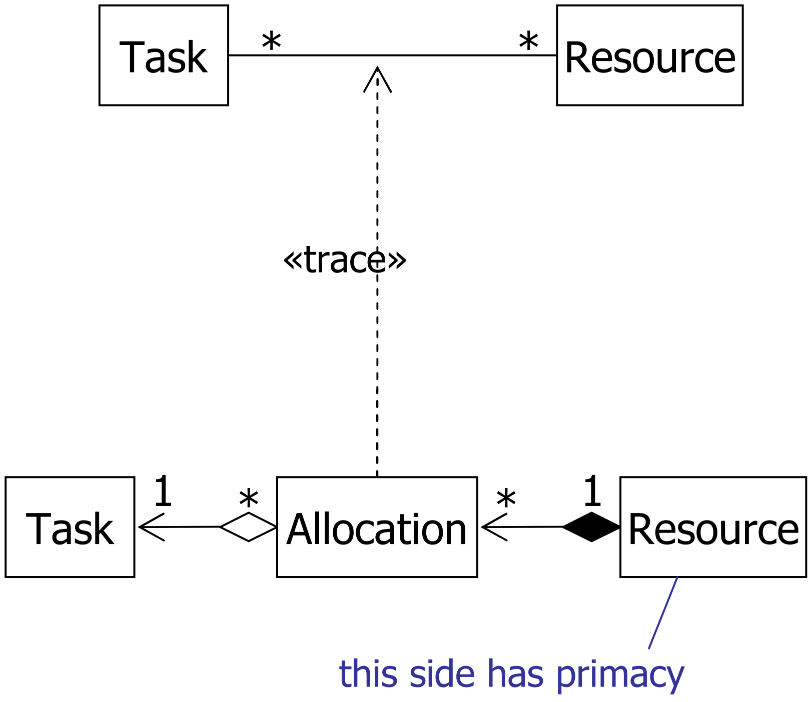




you can indicate the type of collection using a constraint

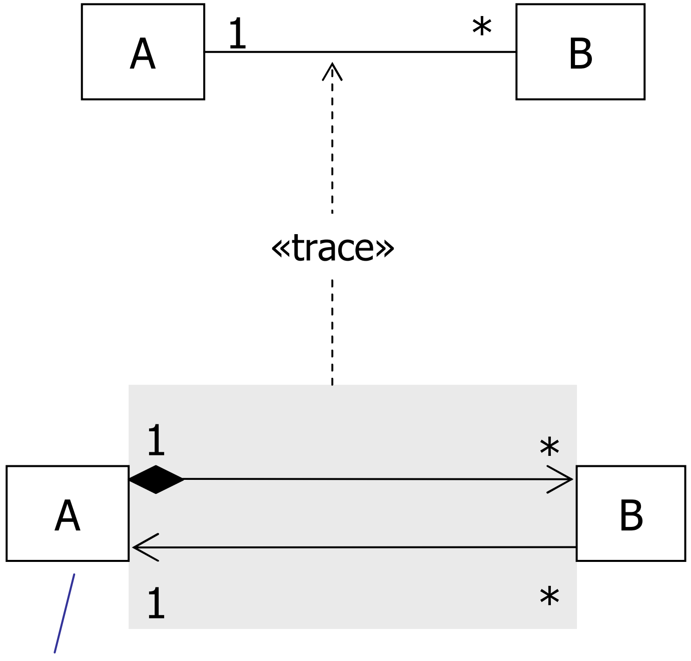
Many to many associations

- There is no commonly used OO language that directly supports many-to-many associations
- We must reify such associations into design classes
- Again, we must decide which side of the association should have primacy and use composition, aggregation and navigability accordingly



Bi-directional associations

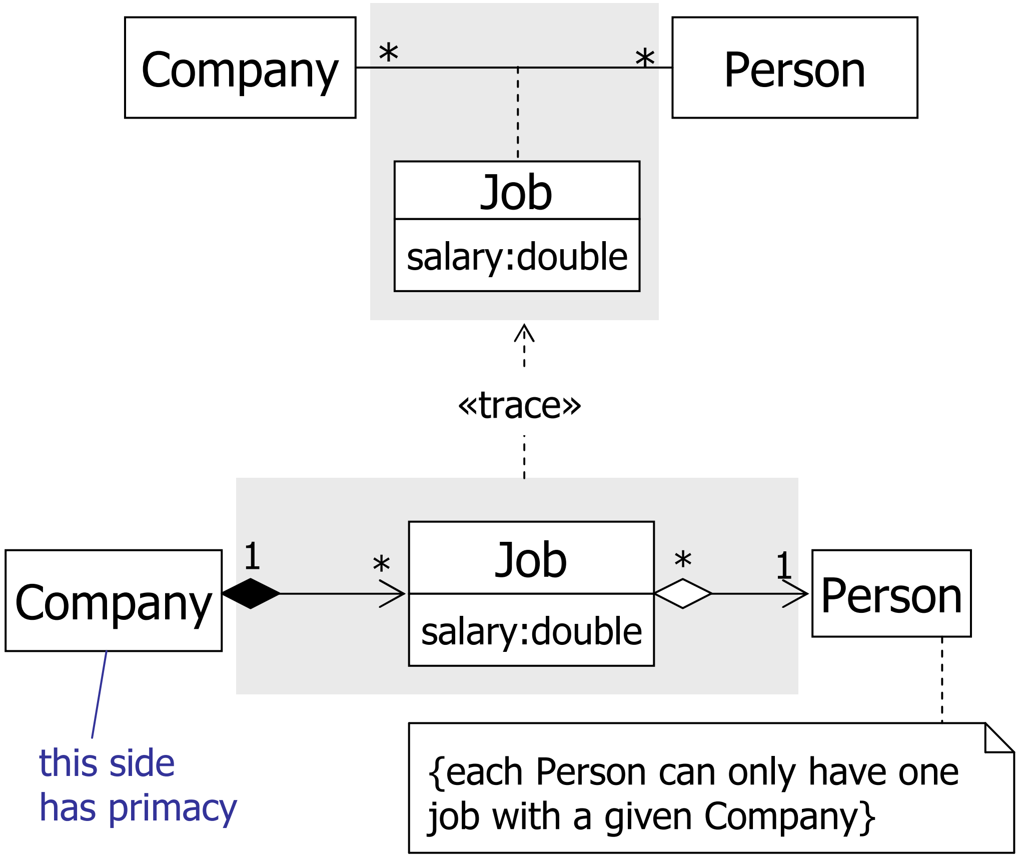
- There is no commonly used OO language that directly supports bi-directional associations
- We must resolve each bi-directional associations into two unidirectional associations
- Again, we must decide which side of the association should have primacy and use composition, aggregation and navigability accordingly



this side has primacy

Association classes

- There is no commonly used OO language that directly supports association classes
- Refine all association classes into a design class
- Decide which side of the association has primacy and use composition, aggregation and navigability accordingly



Summary

- In this section we have seen how we take the incompletely specified associations in an analysis model and refine them to:
  - Aggregation
    - Whole-part relationship
    - Parts are independent of the whole
    - Parts may be shared between wholes
    - The whole is incomplete in some way without the parts
  - Composition
    - A strong form of aggregation
    - Parts are entirely dependent on the whole
    - Parts may not be shared
    - The whole is incomplete without the parts
- One-to-many, many-to-many, bi-directional associations and association classes are refined in design

Design - interfaces and components

What is an interface?

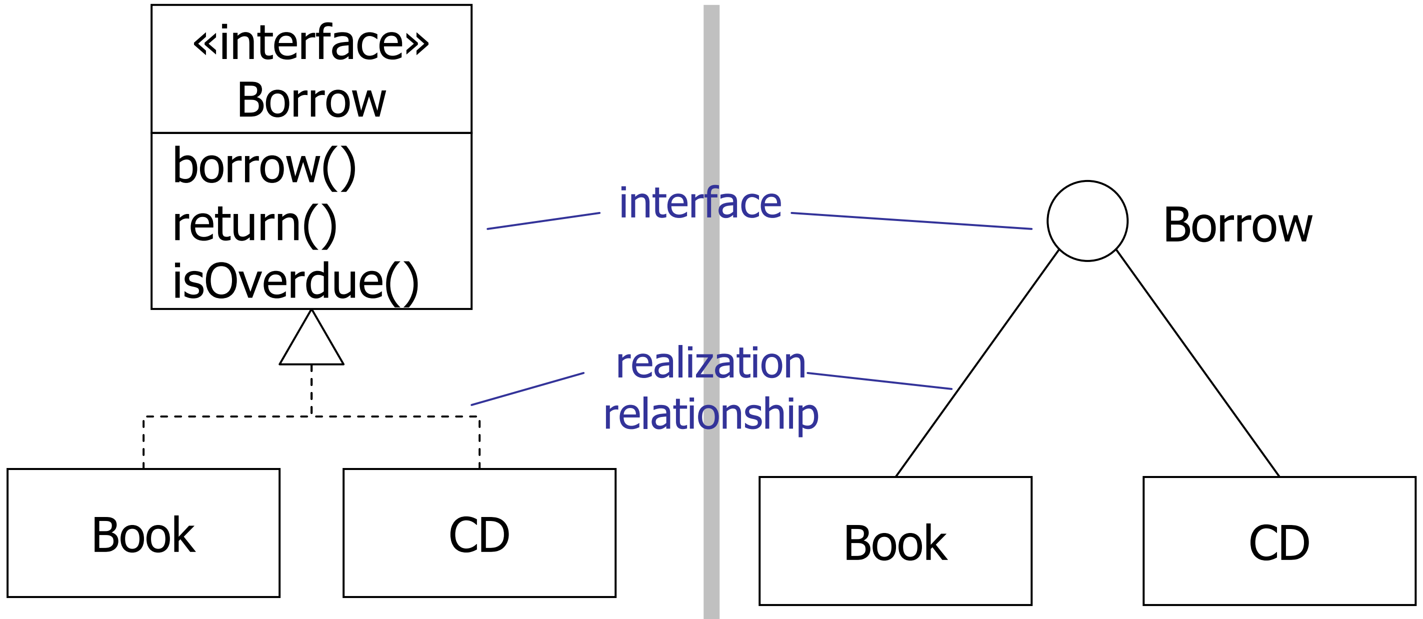
design by contract

- An interface specifies a named set of public features
- It separates the specification of functionality from its implementation
- An interface defines a contract that all realizing classifiers *must* conform to:

Interface specifies	Realizing classifier
operation	Must have an operation with the same signature and semantics
attribute	Must have public operations to set and get the value of the attribute. The realizing classifier is not required to actually have the attribute specified by the interface, but it must behave as though it has
association	Must have an association to the target classifier. If an interface specifies an association to another interface, then the implementing classifiers of these interfaces must have an association between them
constraint	Must support the constraint
stereotype	Has the stereotype
tagged value	Has the tagged value
protocol	Realizes the protocol

Provided interface syntax

- A provided interface indicates that a classifier implements the services defined in an interface

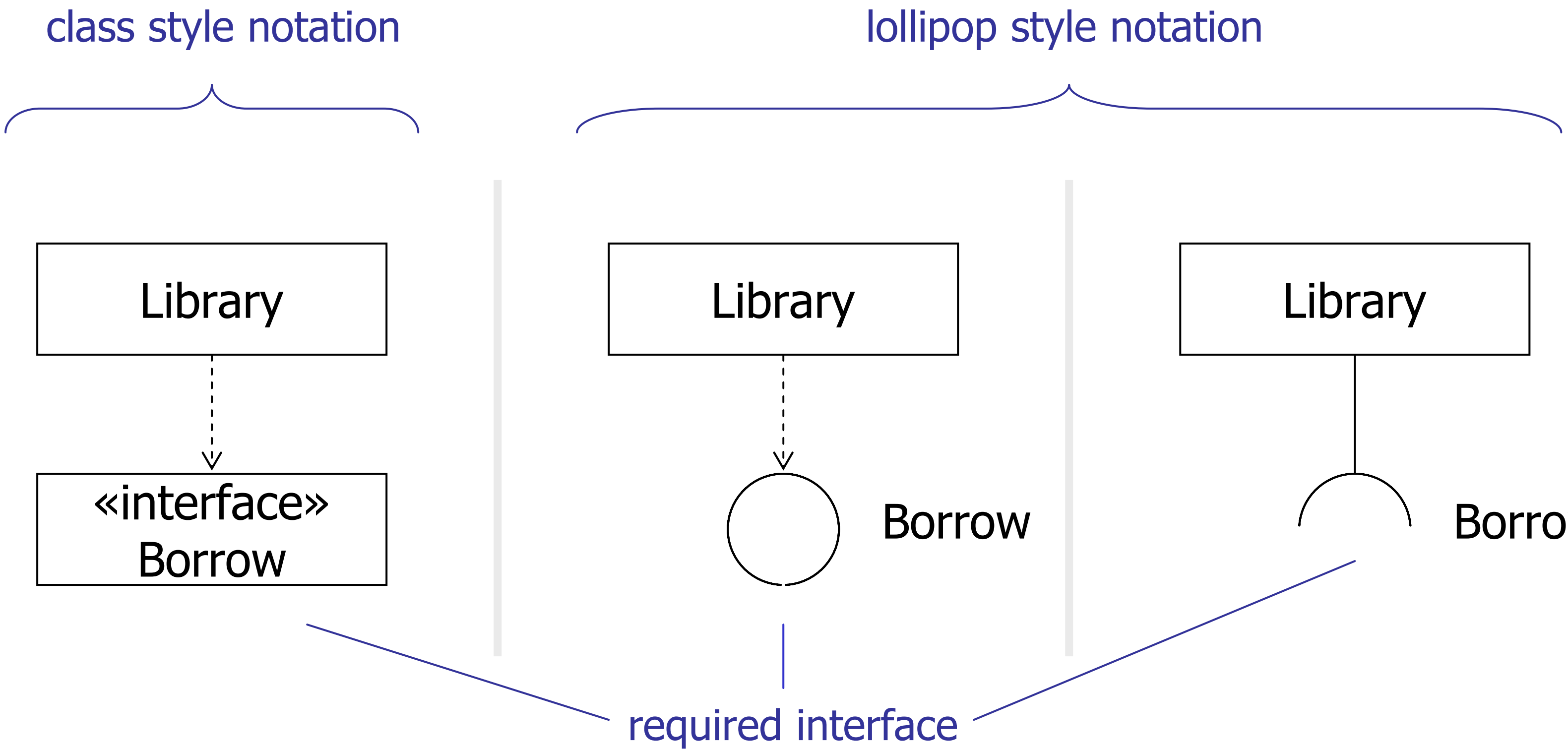


"Class" style notation

"Lollipop" style notation
(note: you can't show the interface operations or attributes with this shorthand style of notation)

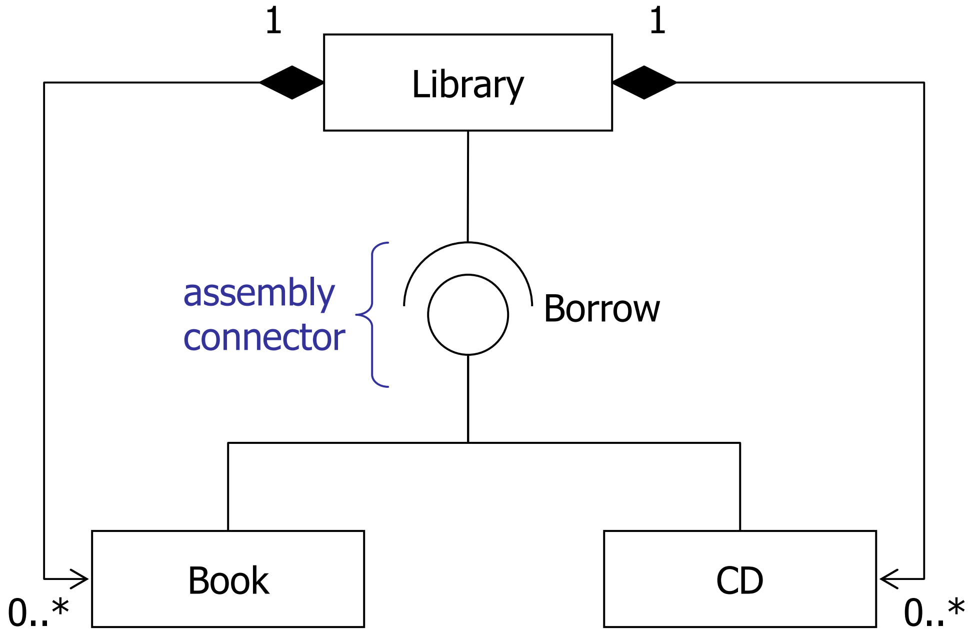
Required interface syntax

- A required interface indicates that a classifier uses the services defined by the interface



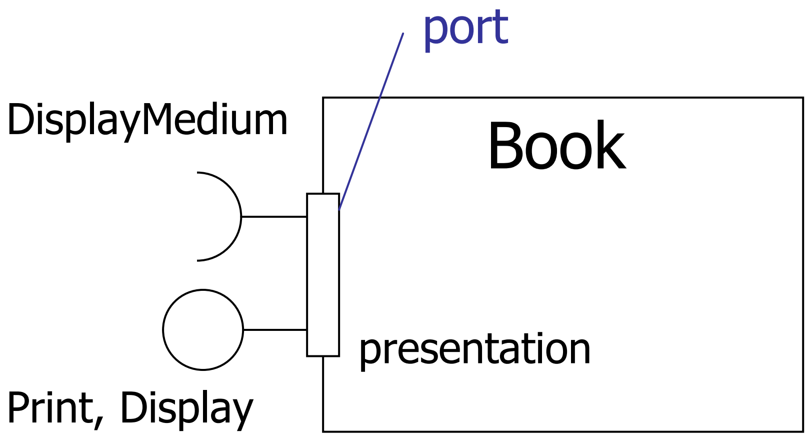
Assembly connectors

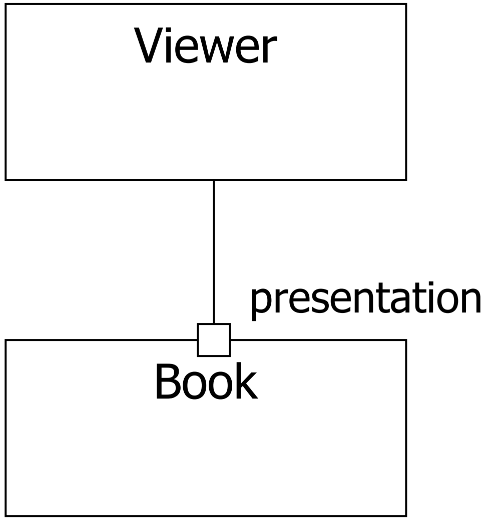
- You can connect provided and required interfaces using an assembly connector



Ports: organizing interfaces

- A port specifies an interaction point between a classifier and its environment
- A port is typed by its provided and required interfaces:
  - It is a semantically cohesive set of provided and required interfaces
  - It may have a name
- If a port has a single required interface, this defines the type of the port
  - You can name the port portName:RequiredInterfaceName





Interfaces and CBD

- Interfaces are the key to component based development (CBD)
- This is constructing software from replaceable, plug-in parts:
  - Plug – the provided interface
  - Socket – the required interface
- Consider:
  - Electrical outlets
  - Computer ports – USB, serial, parallel
- Interfaces define a contract so classifiers that realise the interface agree to abide by the contract and can be used interchangeably

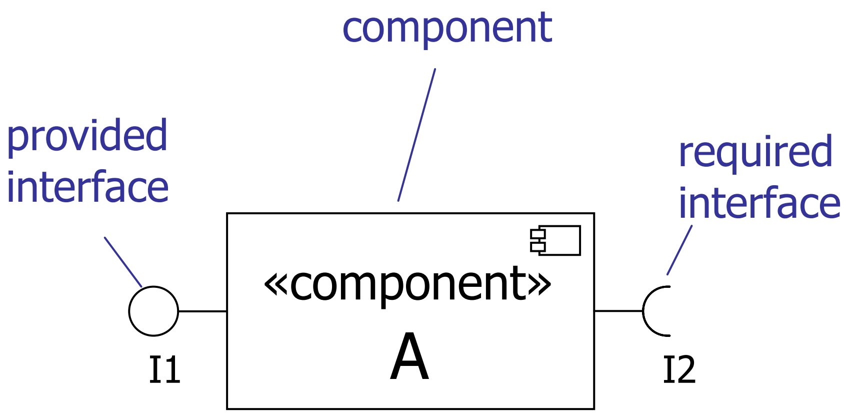
What is a component?

- The UML 2.0 specification states that, "A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment"
  - A black-box whose external behaviour is completely defined by its provided and required interfaces
  - May be substituted for by other components provided they all support the same protocol
- Components can be:
  - Physical - can be directly instantiated at run-time e.g. an Enterprise JavaBean (EJB)
  - Logical - a purely logical construct e.g. a subsystem
    - only instantiated indirectly by virtue of its parts being instantiated

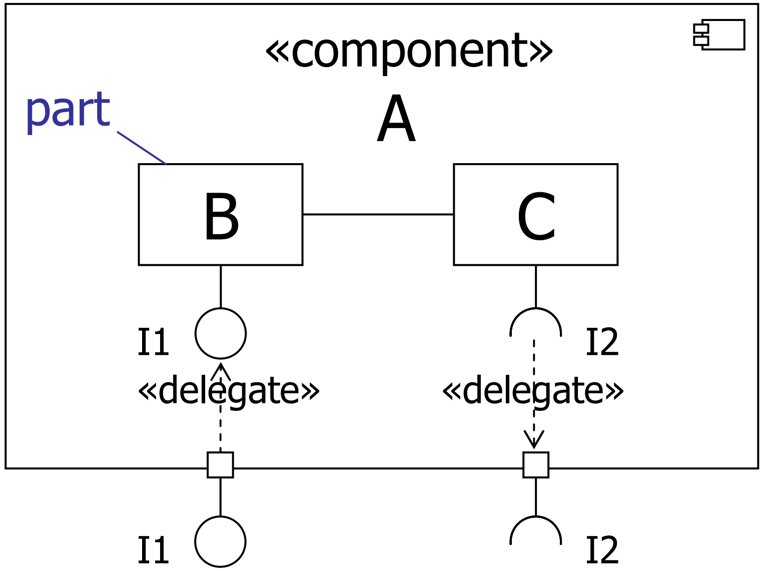
Component syntax

- Components may have provided and required interfaces, ports, internal structure
  - Provided and required interfaces usually delegate to internal parts
  - You can show the parts nested inside the component icon or externally, connected to it by dependency relationships

black box notation



white box notation

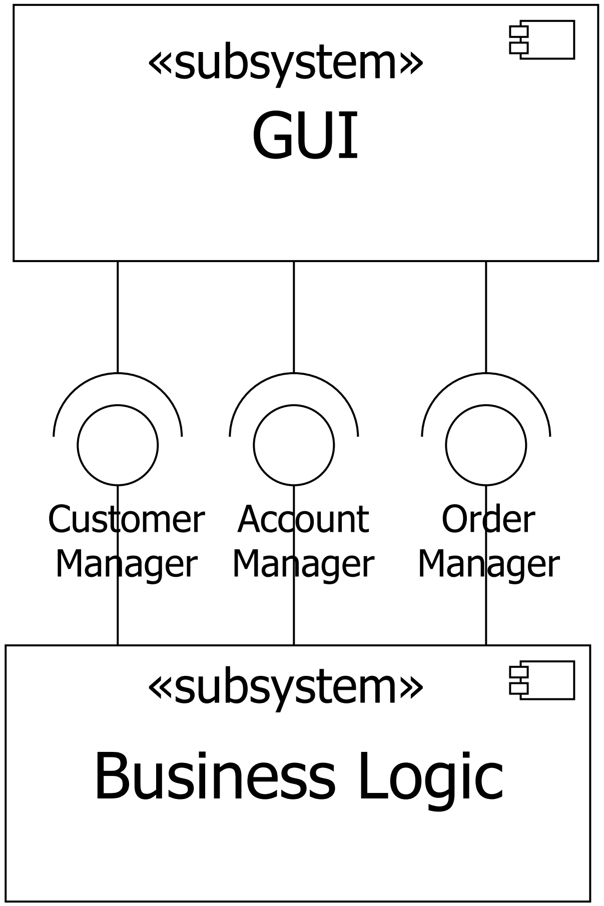


Standard component stereotypes

Stereotype	Semantics
«buildComponent»	A component that defines a set of things for organizational or system level development purposes.
«entity»	A persistent information component representing a business concept.
«implementation»	A component definition that is not intended to have a specification itself. Rather, it is an implementation for a separate «specification» to which it has a dependency.
«specification»	A classifier that specifies a domain of objects without defining the physical implementation of those objects. For example, a Component stereotyped by «specification» only has provided and required interfaces - no realizing classifiers.
«process»	A transaction based component.
«service»	A stateless, functional component (computes a value).
«subsystem»	A unit of hierarchical decomposition for large systems.

Subsystems

- A subsystem is a component that acts as a unit of decomposition for a larger system
- It is a logical construct used to decompose a larger system into manageable chunks
- Subsystems *can't* be instantiated at run-time, but their contents can
- Interfaces connect subsystems together to create a system architecture



Finding interfaces and ports

- Challenge each association:
  - Does the association have to be to another class, or can it be to an interface?
- Challenge each message send:
  - Does the message send have to be to another class, or can it be to an interface?
- Look for repeating groups of operations
- Look for groups of operations that might be useful elsewhere
- Look for possibilities for future expansion
- Look for cohesive sets of provided and required interfaces and organize these into named ports
- Look at the dependencies between subsystems - mediate these by an assembly connector where possible

Designing with interfaces

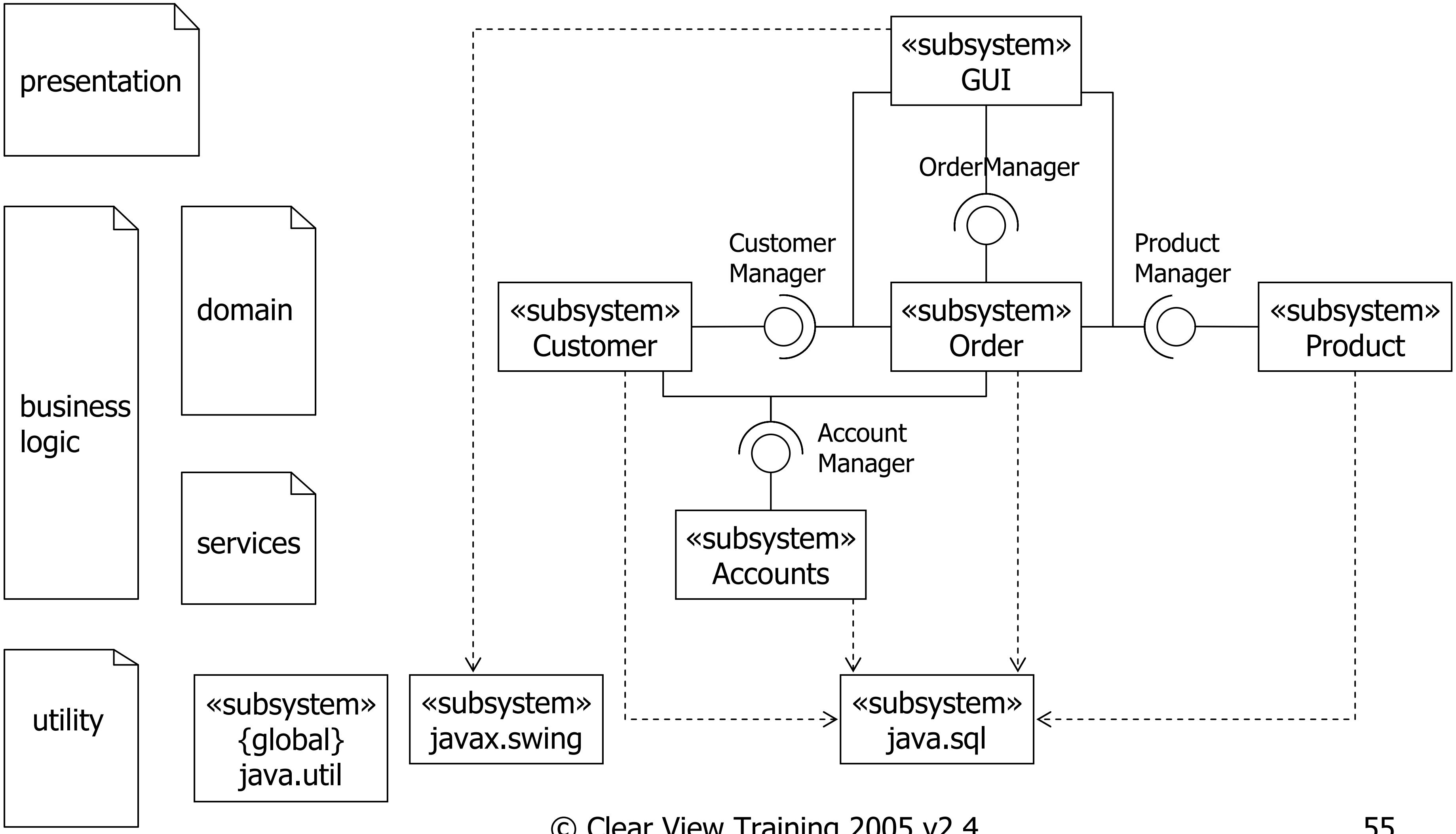
- Design interfaces based on common sets of operations
- Design interfaces based on common roles
  - These roles may be between two classes or even within one class which interacts with itself
  - These roles may also be between two subsystems
- Design interfaces for new plug-in features
- Design interfaces for plug-in algorithms
- The Façade Pattern - use interfaces can be used to create "seams" in a system:
  - Identify cohesive parts of the system
  - Package these into a «subsystem»
  - Define an interface to that subsystem
- Interfaces allow information hiding and separation of concerns

Physical architecture

- Subsystems and interfaces comprise the physical architecture of our model
- We must now organise this collection of interfaces and subsystems to create a coherent architectural picture:
- We can apply the "layering" architectural pattern
  - Subsystems are arranged into layers
  - Each layer contains design subsystems which are semantically cohesive e.g. Presentation layer, Business logic layer, Utility layer
  - Dependencies between layers are very carefully managed
  - Dependencies go one way
  - Dependencies are mediated by interfaces

zühlke

Example layered architecture



Using interfaces

- Advantages:
  - When we design with classes, we are designing to specific implementations
  - When we design with interfaces, we are instead designing to contracts which may be realised by many different implementations (classes)
  - Designing to contracts frees our model from implementation dependencies and thereby increases its flexibility and extensibility
- Disadvantages:
  - Interfaces can add flexibility to systems BUT flexibility may lead to complexity
  - Too many interfaces can make a system too flexible!
  - Too many interfaces can make a system hard to understand

Keep it simple!

Summary

- Interfaces specify a named set of public features:
  - They define a contract that classes and subsystems may realise
  - Programming to interfaces rather than to classes reduces dependencies between the classes and subsystems in our model
  - Programming to interfaces increases flexibility and extensibility
- Design subsystems and interfaces allow us to:
  - Componentize our system
  - Define an architecture

Design - use case realization

Use case realization - design

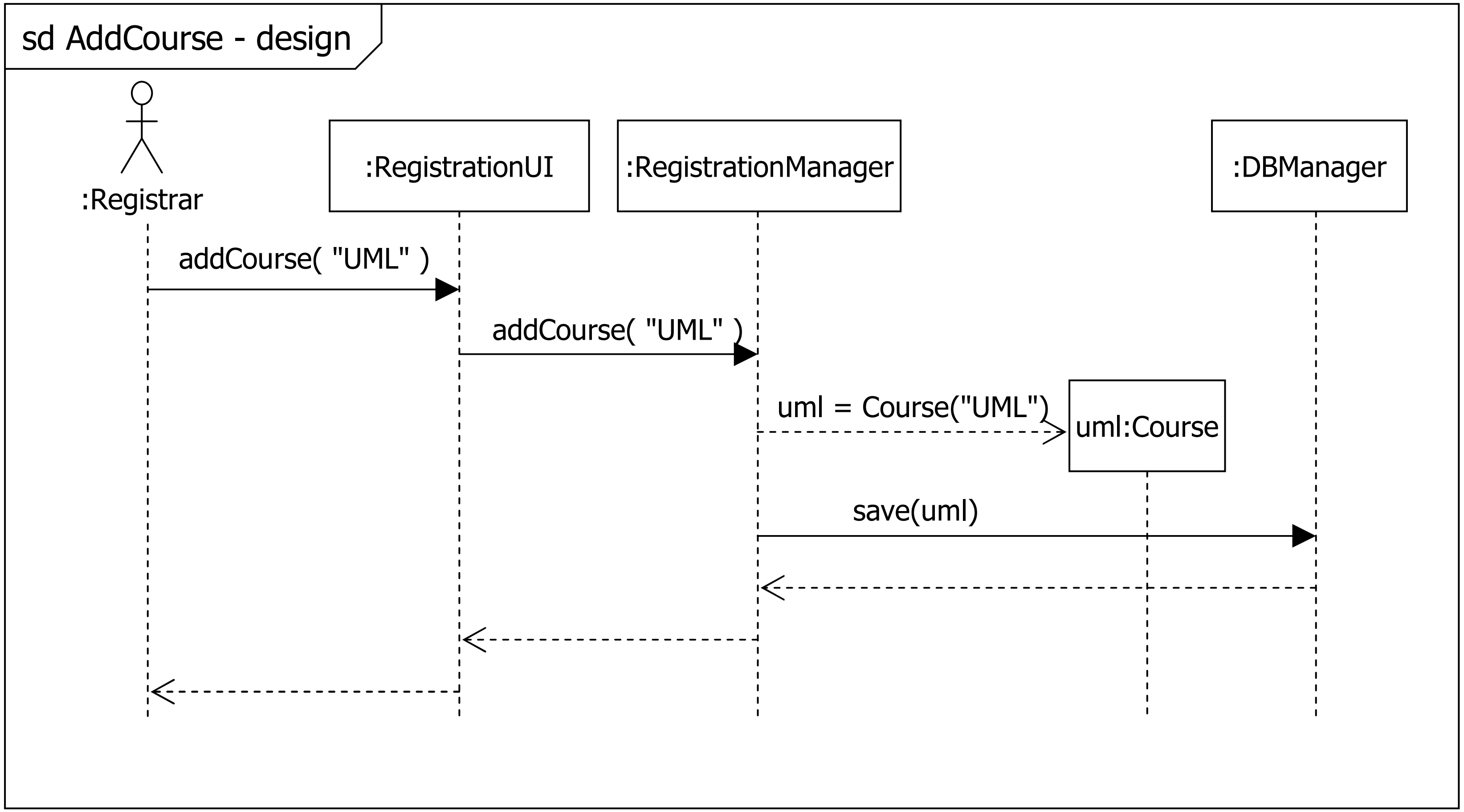
- A collaboration of Design objects and classes that realise a use case
- A Design use case realization contains
  - Design object interaction diagrams
  - Links to class diagrams containing the participating Design classes
  - An explanatory text (flow)
- There is a trace between an Analysis use case realization and a Design use case realization
- The Design use case realization specifies implementation decisions and implements the non-functional requirements

same as in Analysis, but now including implementation details

Interaction diagrams in design

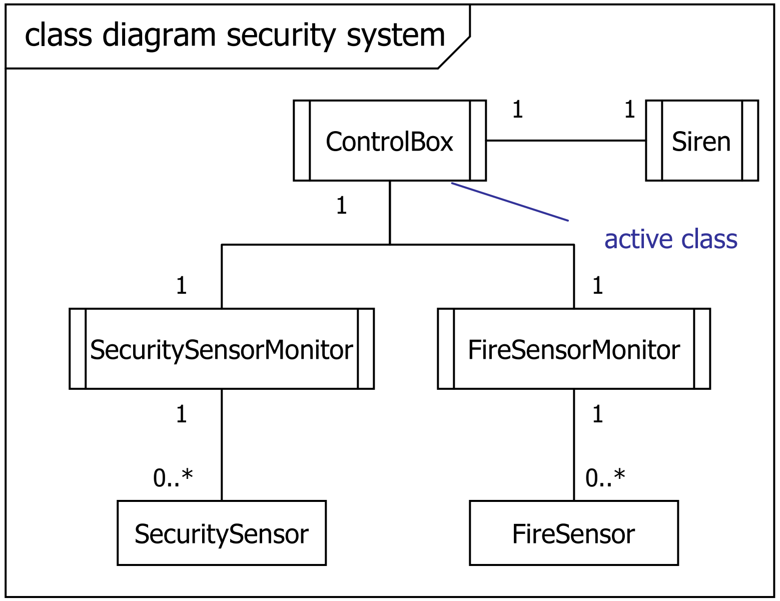
- We only produce a design interaction diagram where it adds value to the project:
  - A refinement of the analysis interaction diagrams to illustrate design issues
  - New diagrams to illustrate technical issues
  - New diagrams to illustrate central mechanisms
- In design:
  - Sequence diagrams are used more than communication diagrams
  - Timing diagrams may be used to capture timing constraints

Sequence diagrams in design

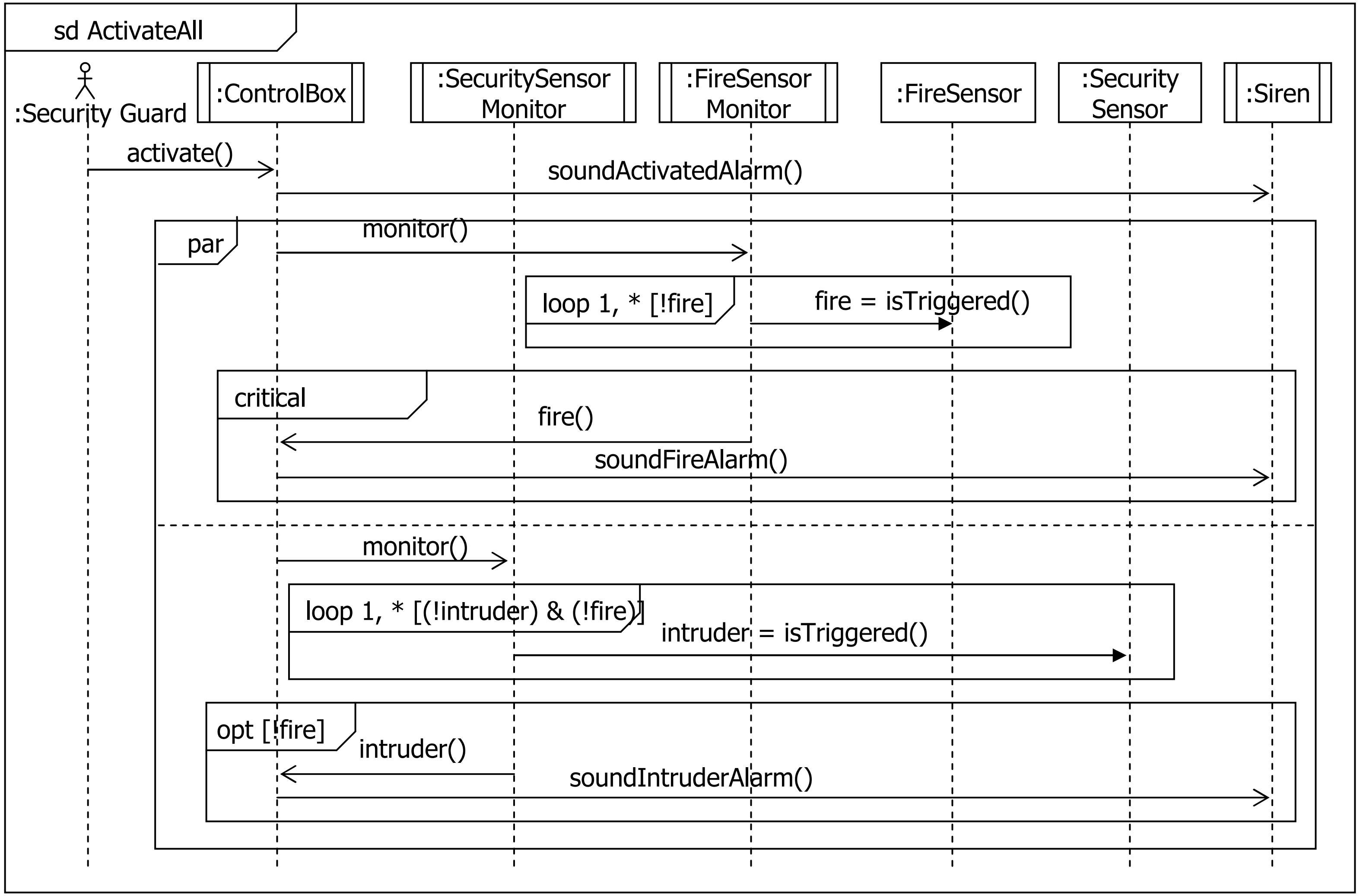


Concurrency – active classes

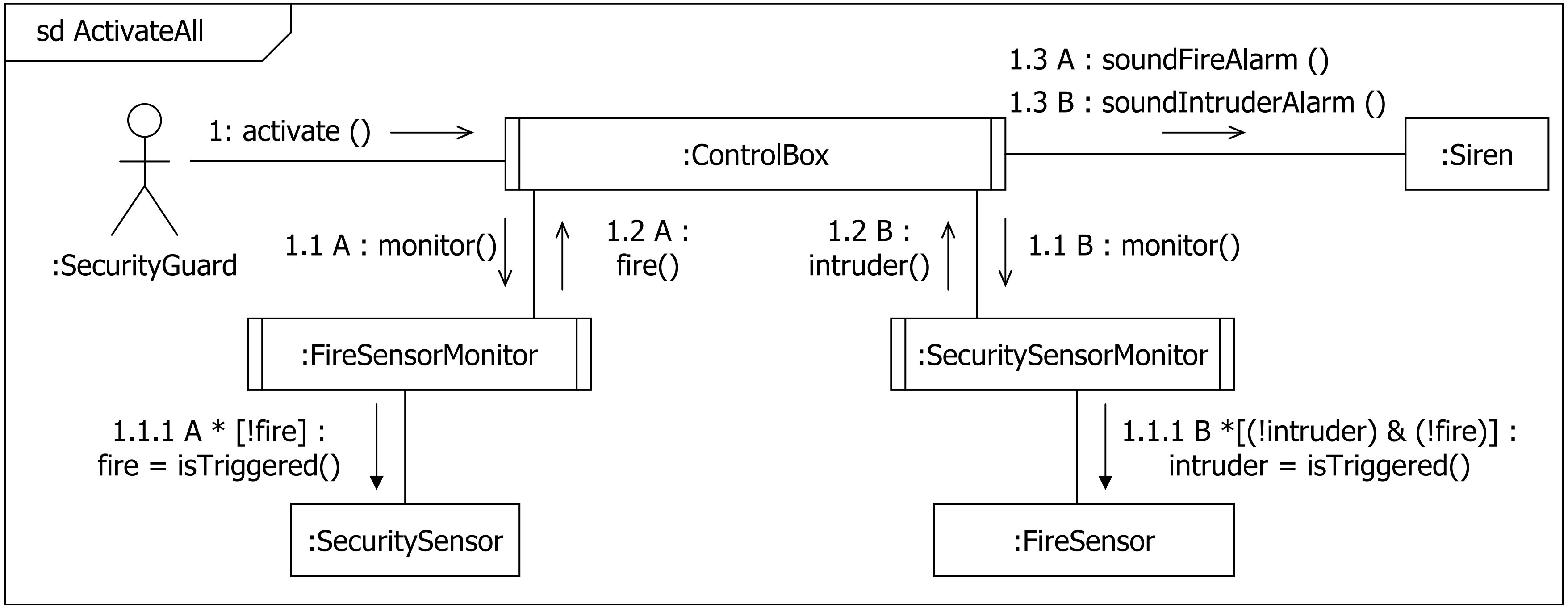
- Active classes are classes whose instances are active objects
  - Active objects have concurrent threads of control
- You can show concurrency on sequence diagrams by giving each thread of execution a name and appending this name to the messages (see next slide)



Concurrency with par



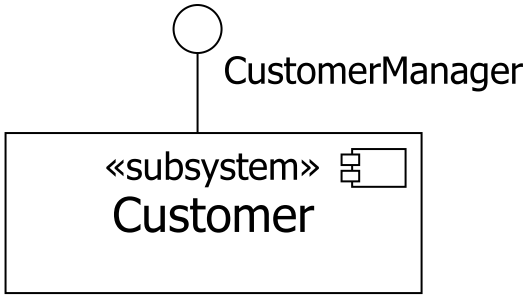
Concurrency – active objects

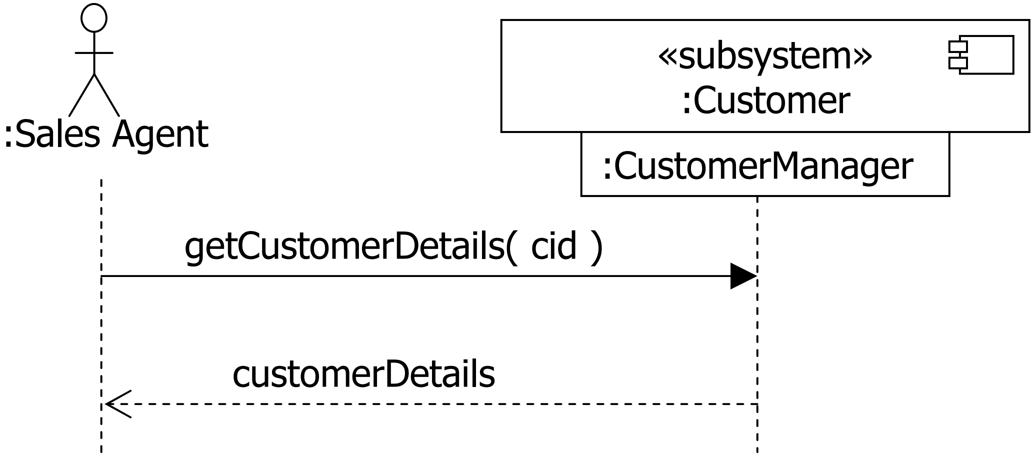


- Each separate thread of execution is given its own name
  - Messages labelled A execute concurrently to messages labelled B
  - e.g. 1.1 A executes concurrently to 1.1 B

Subsystem interactions

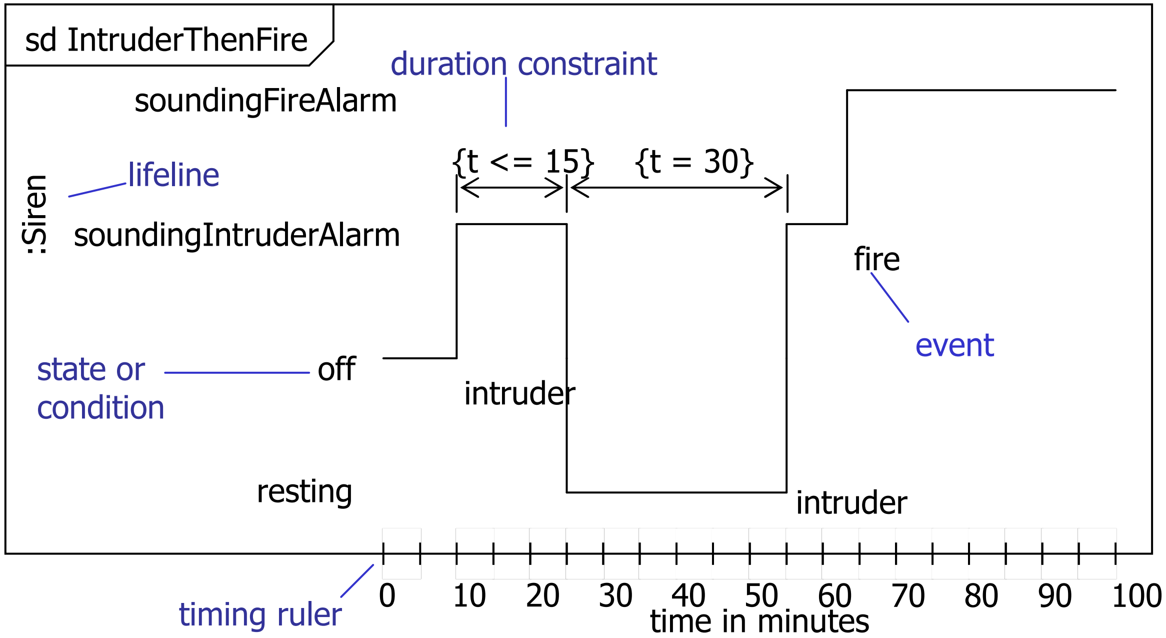
- Sometimes it’s useful to model a use case realization as a high-level interaction between subsystems rather than between classes and interfaces
  - Model the interactions of classes within each subsystem in separate interaction diagrams
- You can show interactions with subsystems on sequence diagrams
  - You can show messages going to parts of the subsystem

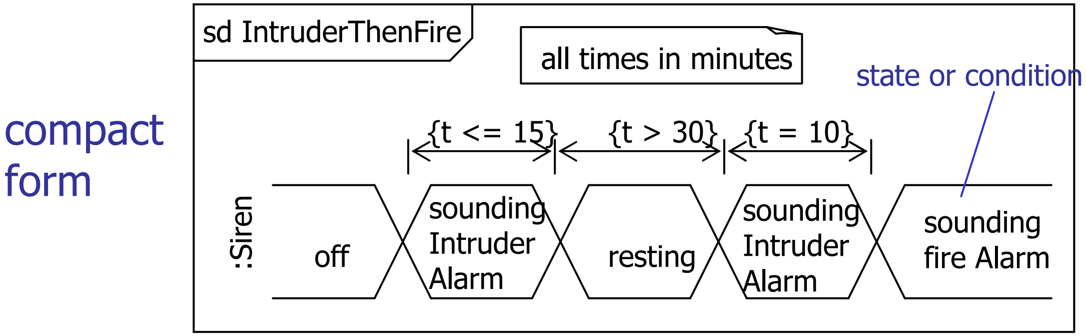




Timing diagrams

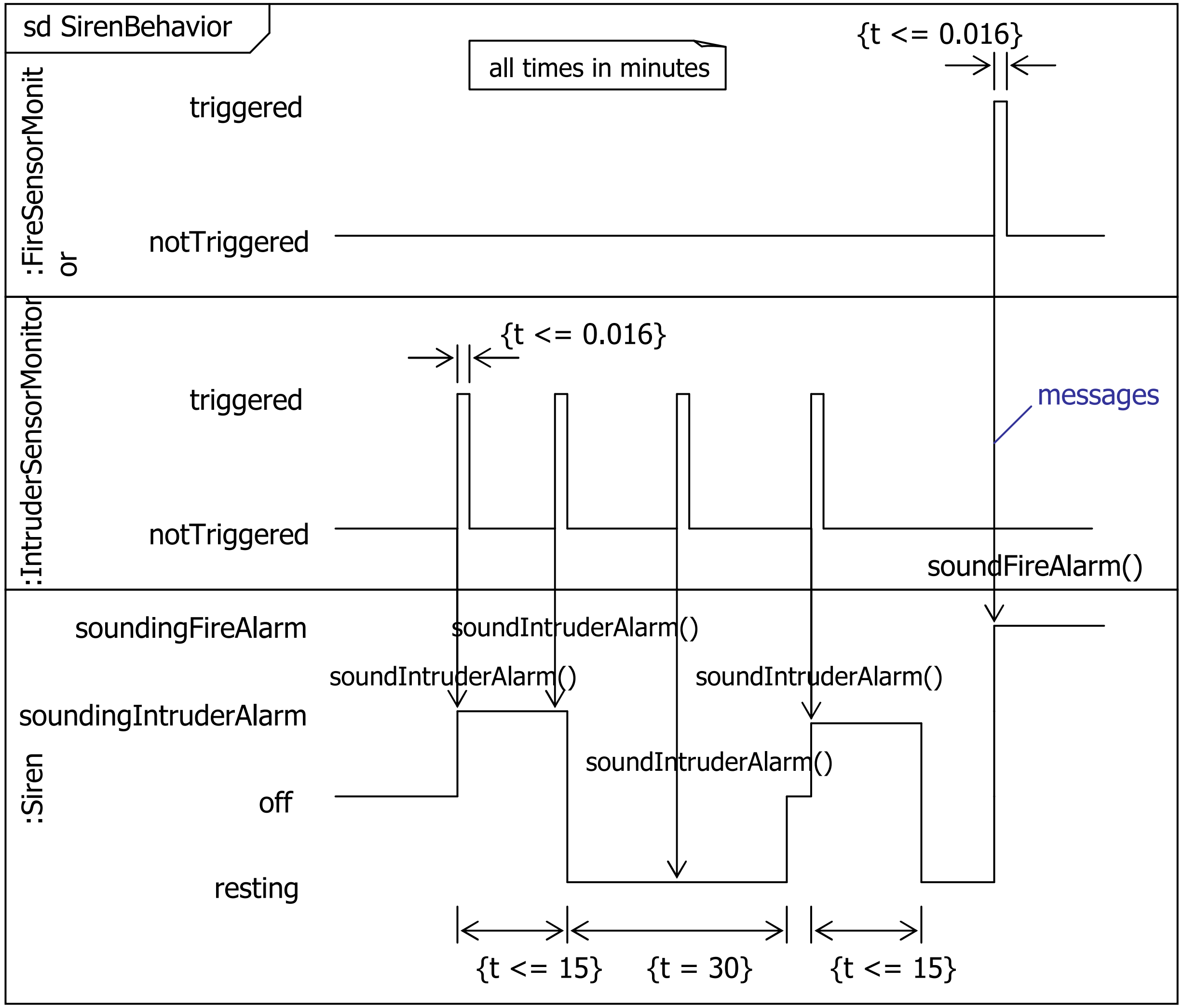
- Emphasize the real-time aspects of an interaction
- Used to model timing constraints
- Lifelines, their states or conditions are drawn vertically, time horizontally
- It's important to state the time units you use in the timing diagram





Messages on timing diagrams

- You can show messages between lifelines on timing diagrams
- Each lifeline has its own partition



Example: use case realization - design

20.8

Summary

- We have looked at:
  - Design sequence diagrams
  - Subsystem interactions
  - Timing diagrams

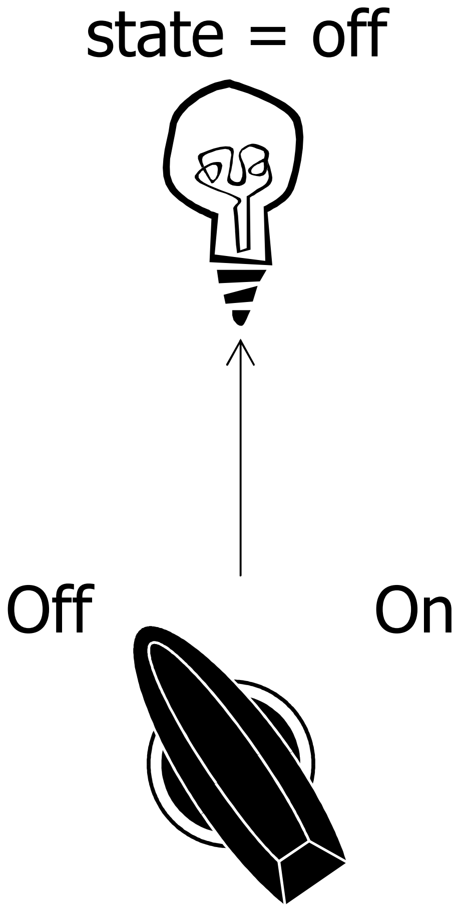
Design - state machines

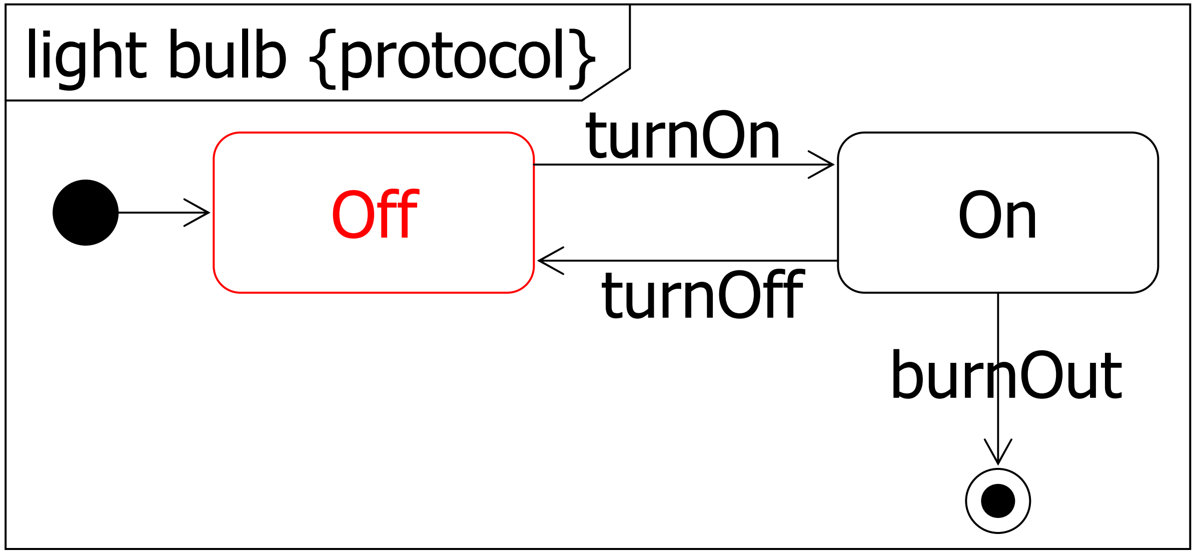
State machines

- Some model elements such as classes, use cases and subsystems, can have interesting dynamic behavior - state machines can be used to model this behaviour
- Every state machine exists in the context of a particular model element that:
  - Responds to events dispatched from outside of the element
  - Has a clear life history modelled as a progression of *states, transitions* and *events.* We'll see what these mean in a minute!
  - Its current behaviour depends on its past
- A state machine diagram always contains exactly one state machine for one model element
- There are two types of state machines (see next slide):
  - *Behavioural* state machines - define the behavior of a model element e.g. the behavior of class instances
  - *Protocol* state machines - Model the protocol of a classifier
    - The conditions under which operations of the classifier can be called
    - The ordering and results of operation calls
    - Can model the protocol of classifiers that have no behavior (e.g. interfaces and ports)

zühlke

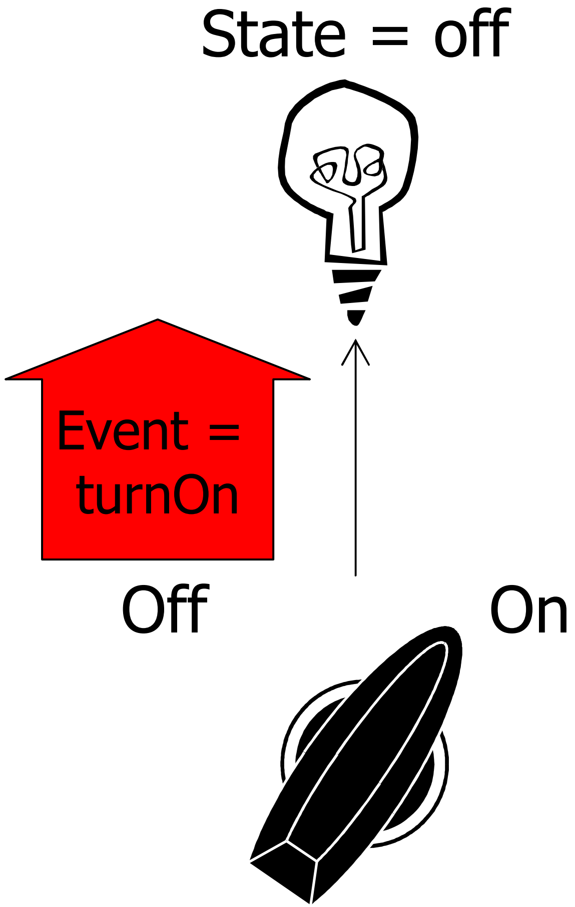
State machine diagrams

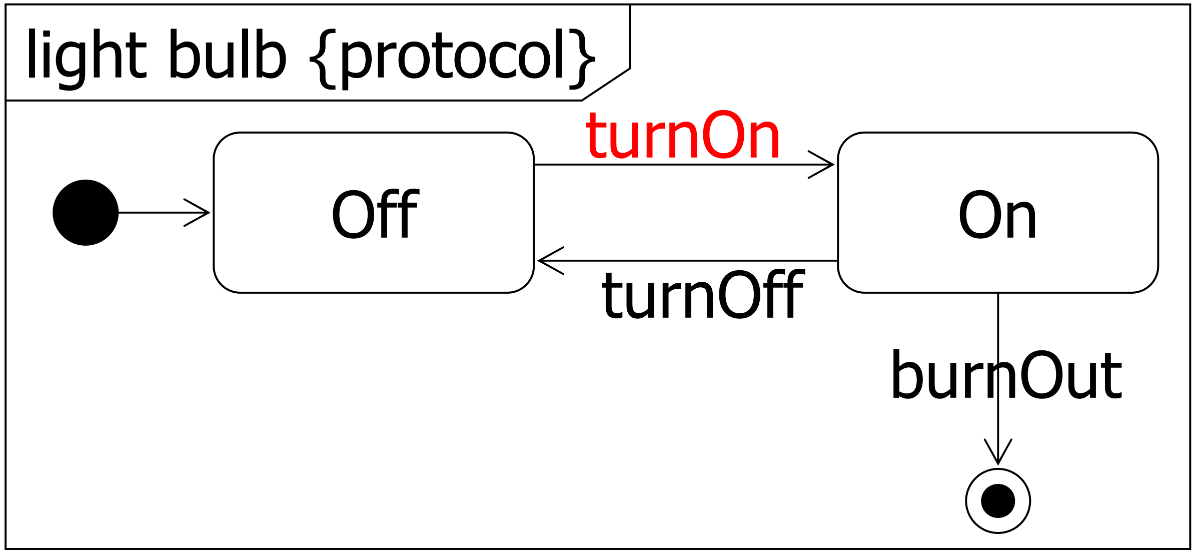




- We begin with the light bulb in the state off

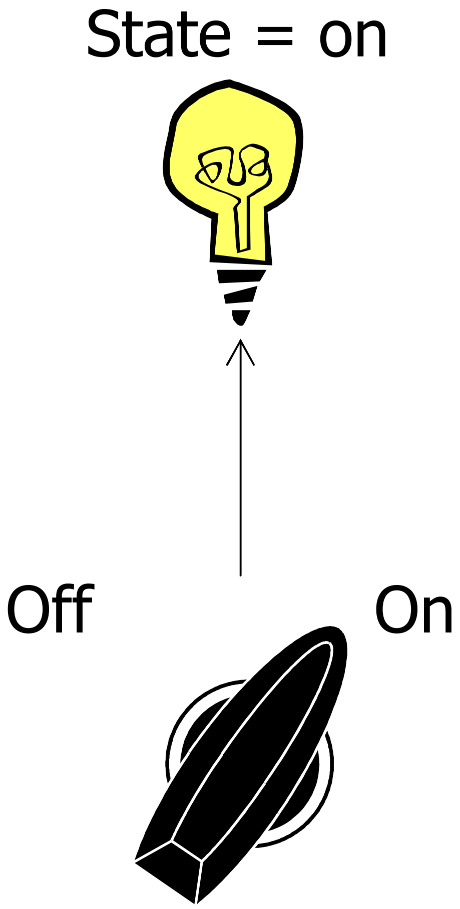
Light bulb turnOn

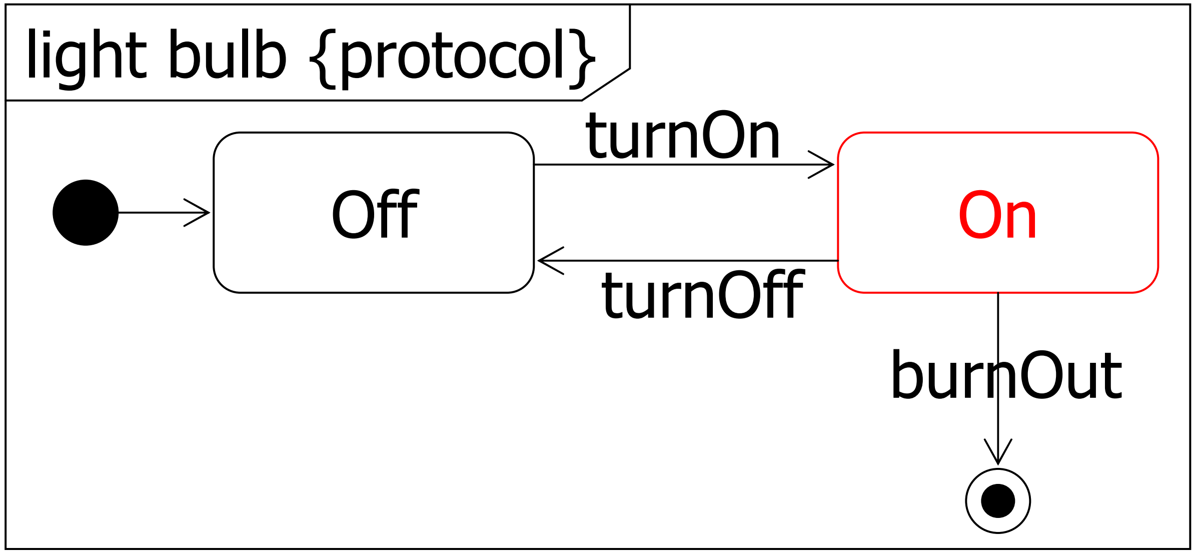




- We throw the switch to On and the event turnOn is sent to the lightbulb

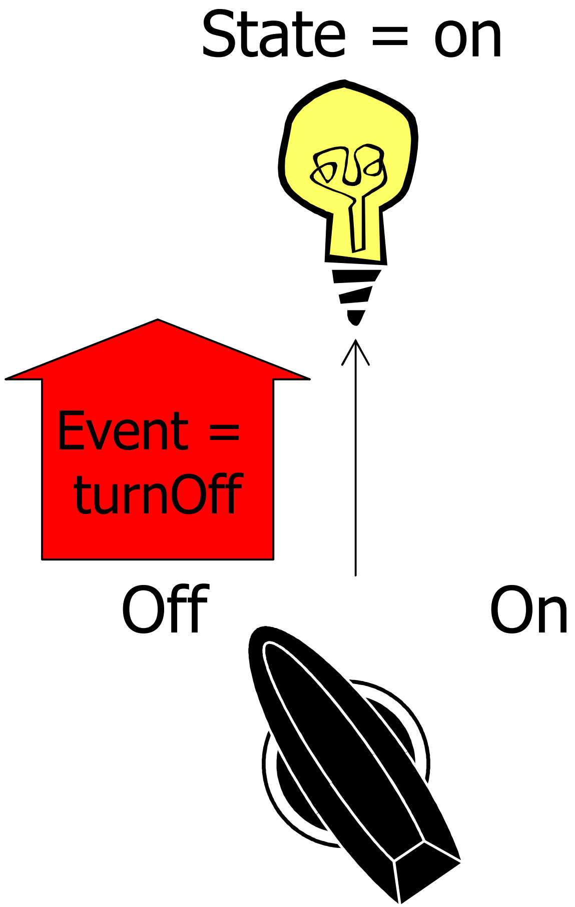
Light bulb On

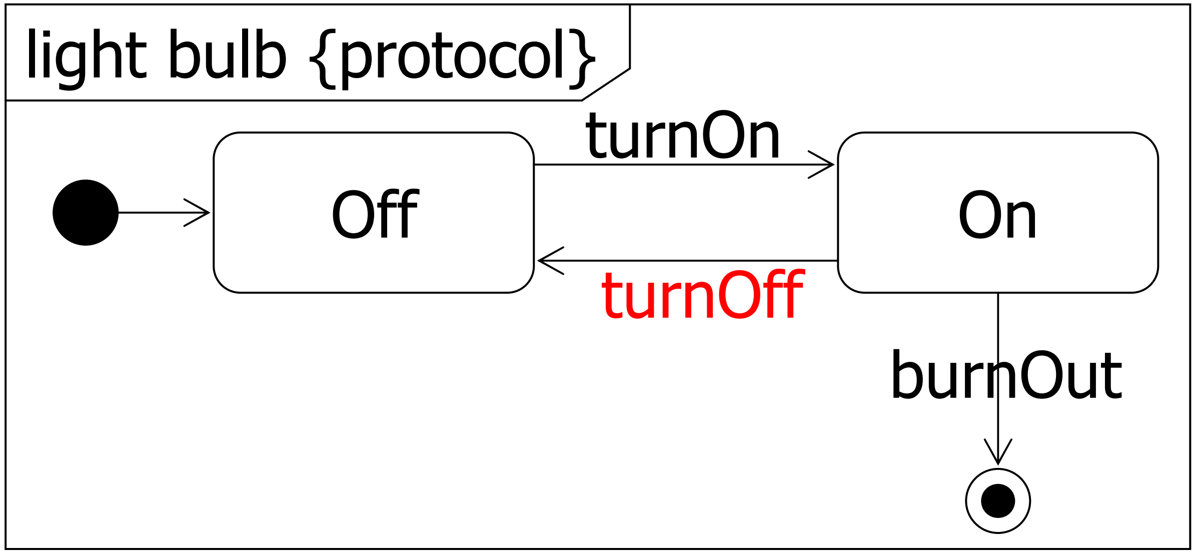




- The light bulb turns on

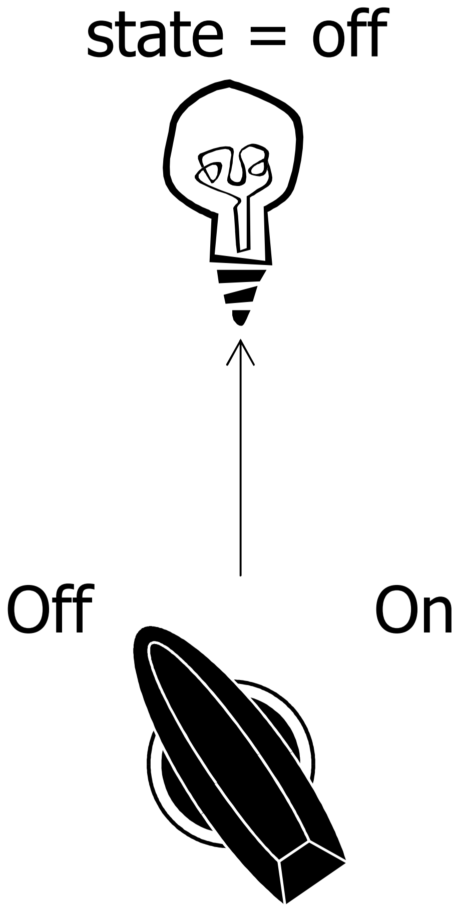
Light bulb turnOff

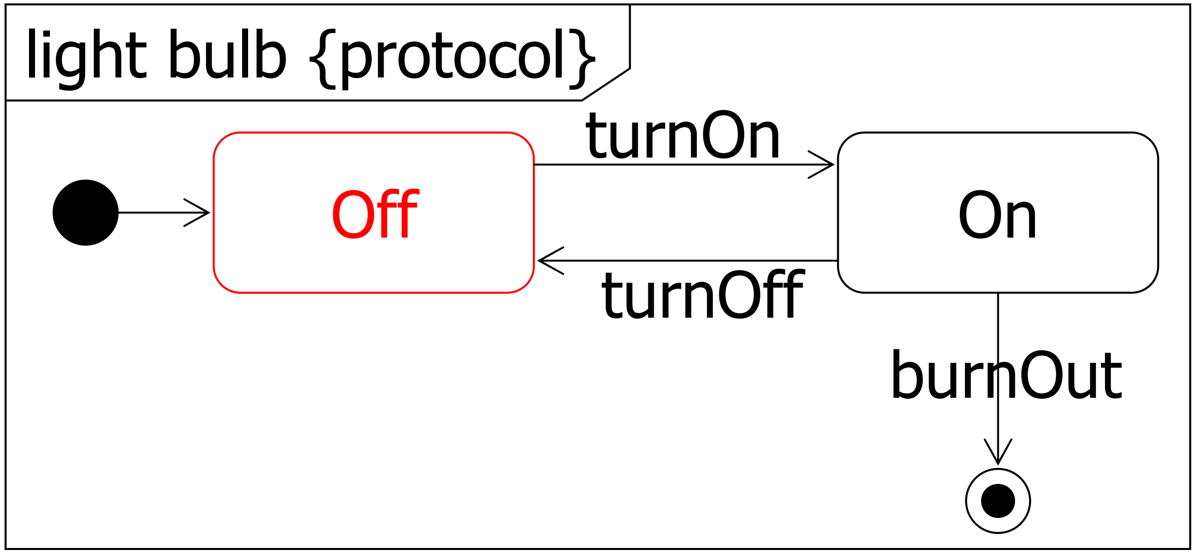




- We turn the switch to Off. The event turnOff is sent to the light bulb

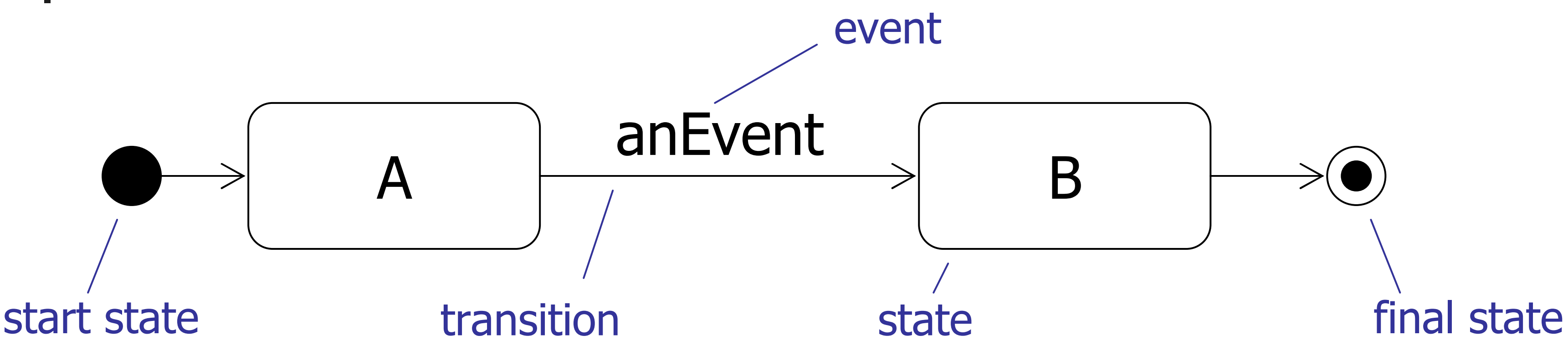
Light bulb Off





- The light bulb turns off

Basic state machine syntax

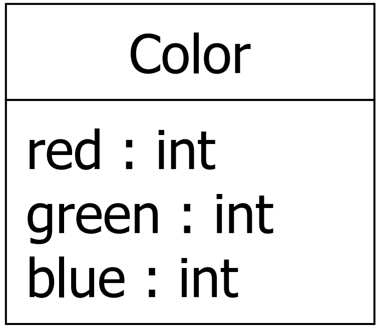


- Every state machine should have a start state which indicates the first state of the sequence
- Unless the states cycle endlessly, state machines should have a final state which terminates the sequence of transitions
- We'll look at each element of the state machine in detail in the next few slides!

States

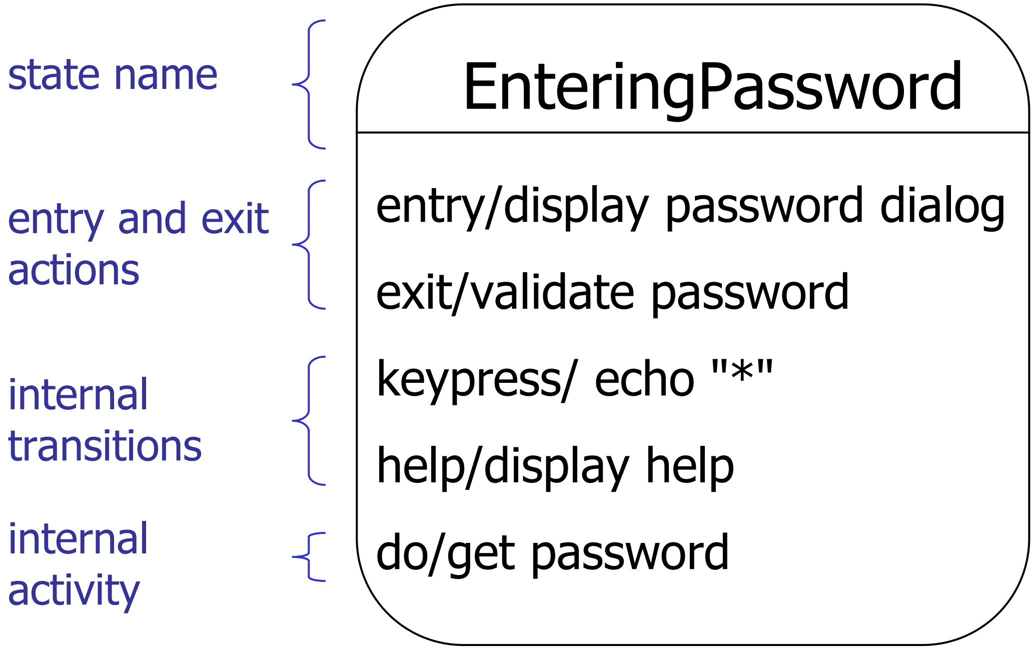
- "A condition or situation during the life of an object during which it satisfies some condition, performs some activity or waits for some event"
- The state of an object at any point in time is determined by:
  - The values of its attributes
  - The relationships it has to other objects
  - The activities it is performing

How many states?



State syntax

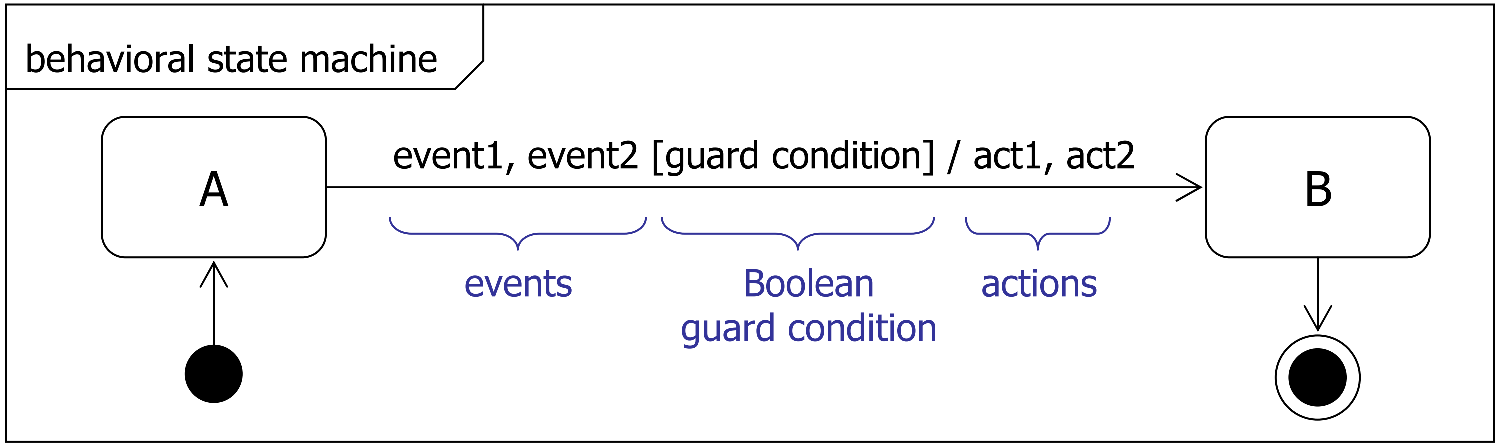
- Actions are *instantaneous* and *uninterruptible*
    - Entry actions occur immediately on entry to the state
    - Exit actions occur immediately on leaving the state
- Internal transitions occur *within* the state. They do *not* transition to a new state
- Activities take a finite amount of time and are interruptible



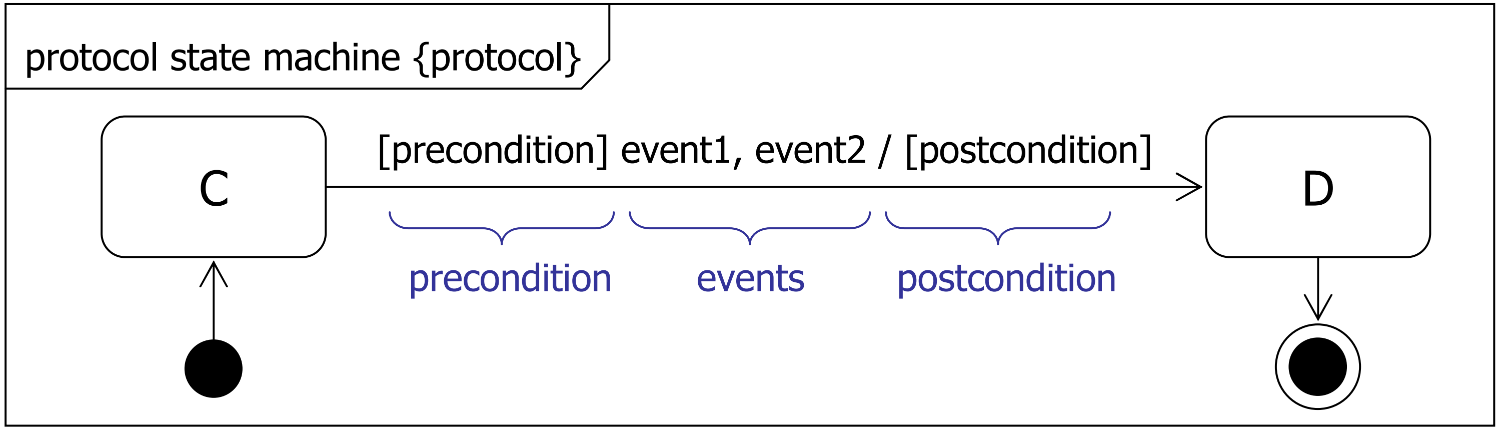
Action syntax: eventTrigger / action
Activity syntax: do / activity

Transitions

protocol state machine

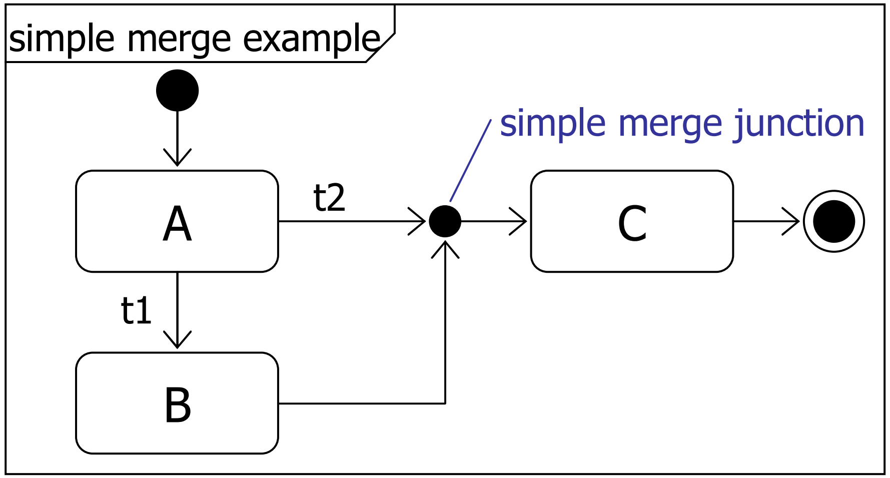


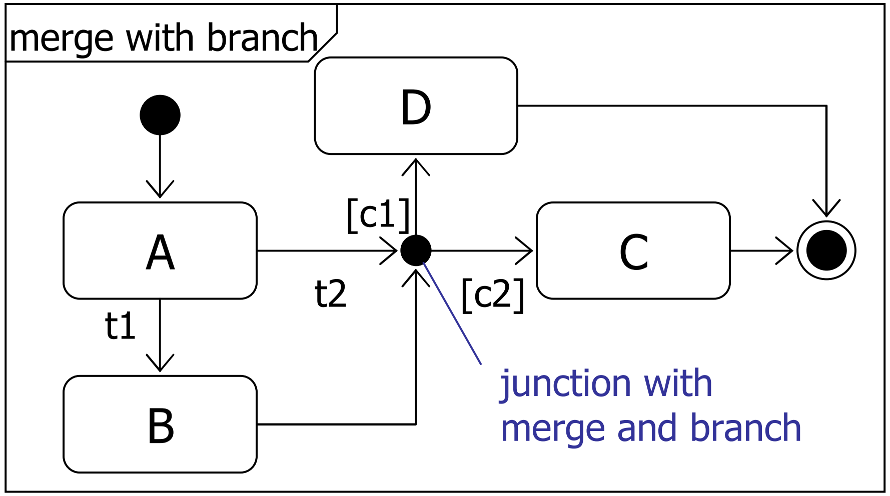
behavioral state machine


protocol state machine {protocol}
C
[precondition] event1, event2 / [postcondition]
D
precondition
events
postcondition

Connecting - the junction pseudo state

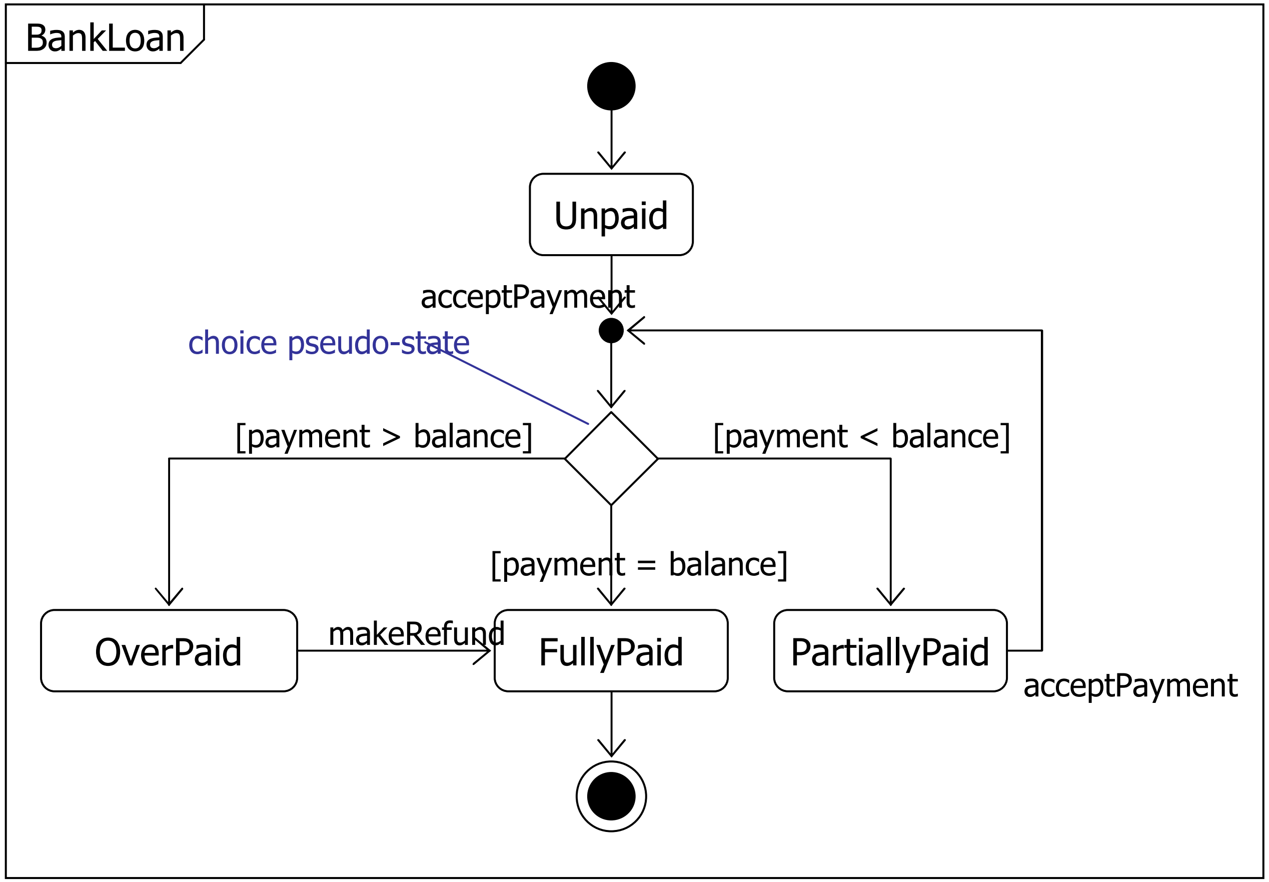
- The junction pseudo state can:
  - connect transitions together (merge)
  - branch transitions
- Each outgoing transition must have a mutually exclusive guard condition





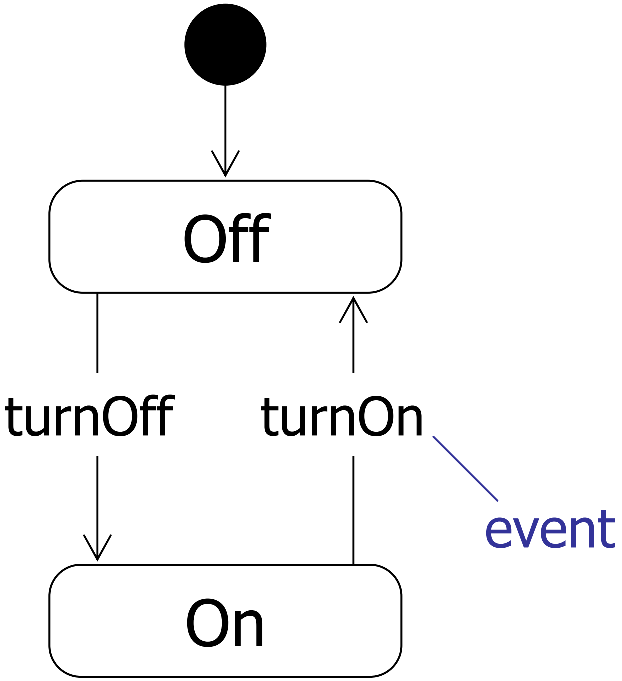
Branching – the choice pseudo state

- The choice pseudo state directs its single incoming transition to one of its outgoing transitions
- Each outgoing transition must have a mutually exclusive guard condition



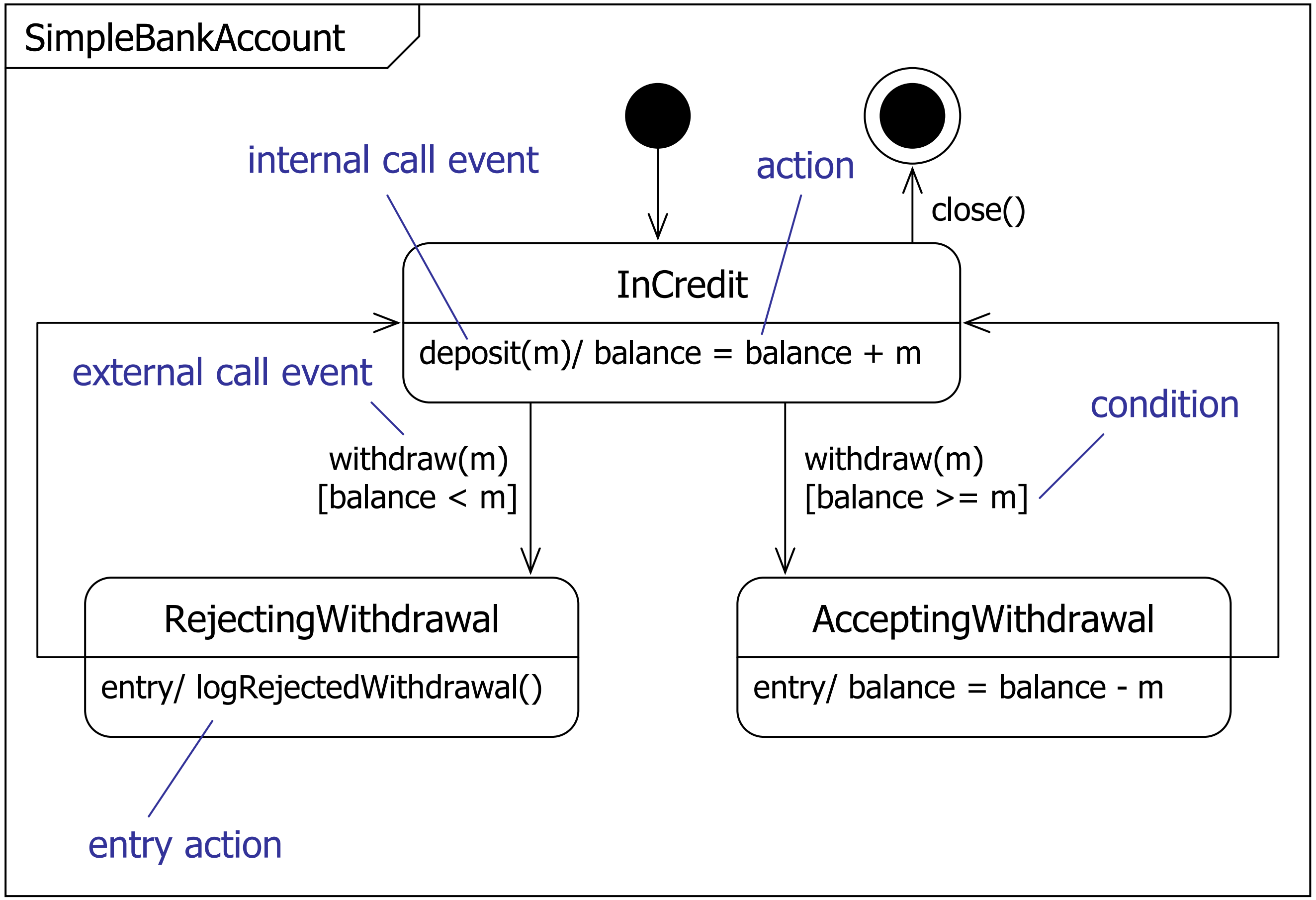
Events

- "The specification of a noteworthy occurrence that has location in time and space"
- Events trigger transitions in state machines
- Events can be shown externally, on transitions, or internally within states (internal transitions)
- There are four types of event:
  - Call event
  - Signal event
  - Change event
  - Time event



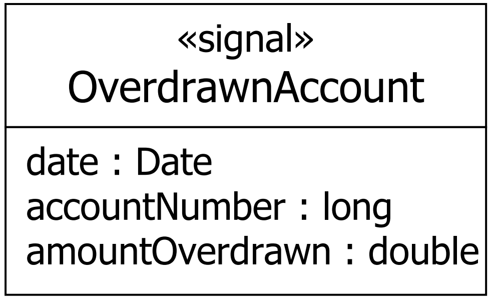
Call event

- A call for an operation executon
- The event should have the same signature as an operation of the context class
- A sequence of actions may be specified for a call event - they may use attributes and operations of the context class
- The return value must match the return type of the operation



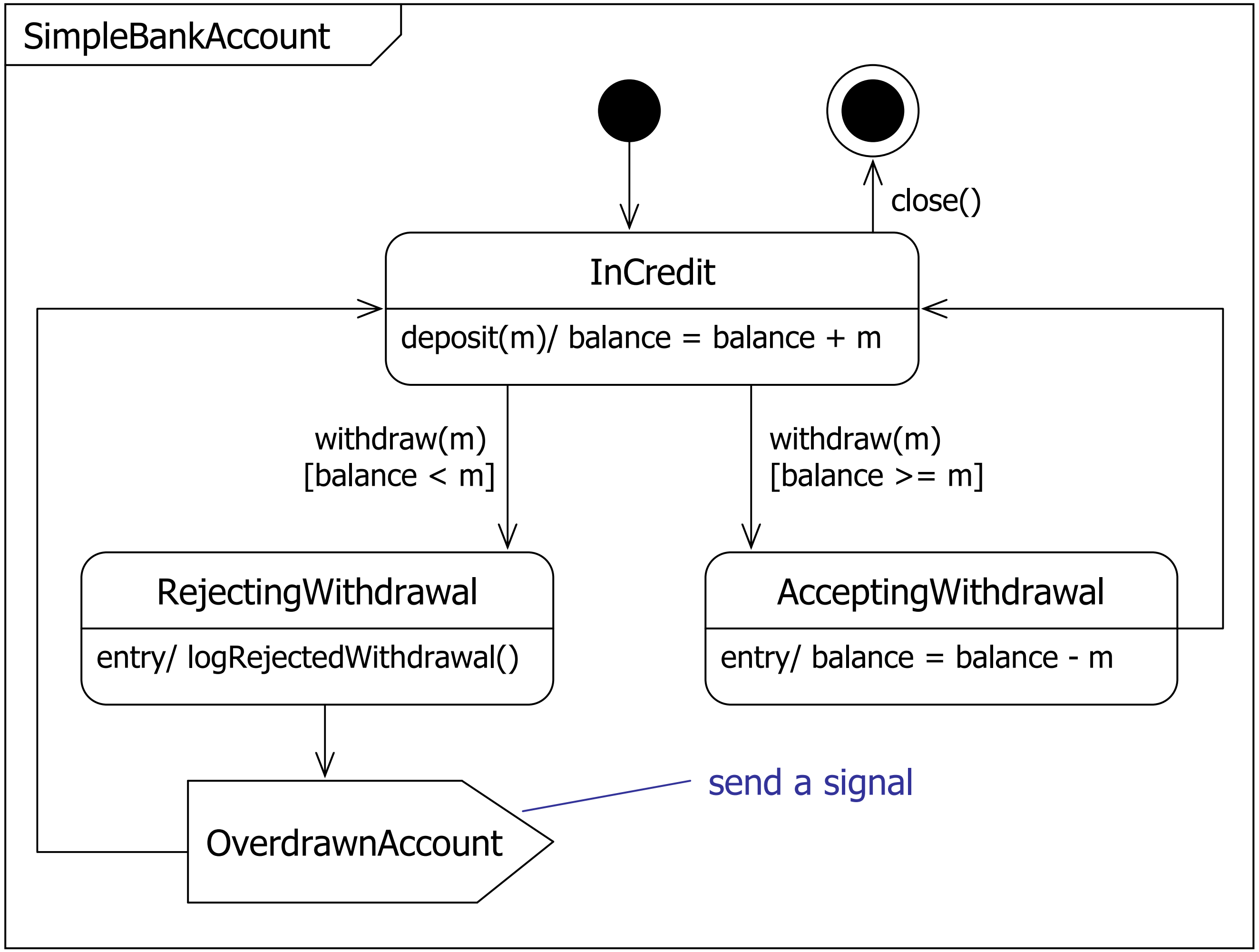
Signal events

- A signal is a package of information that is sent asynchronously between objects
  - the attributes carry the information
  - no operations



«signal»
OverdrawnAccount

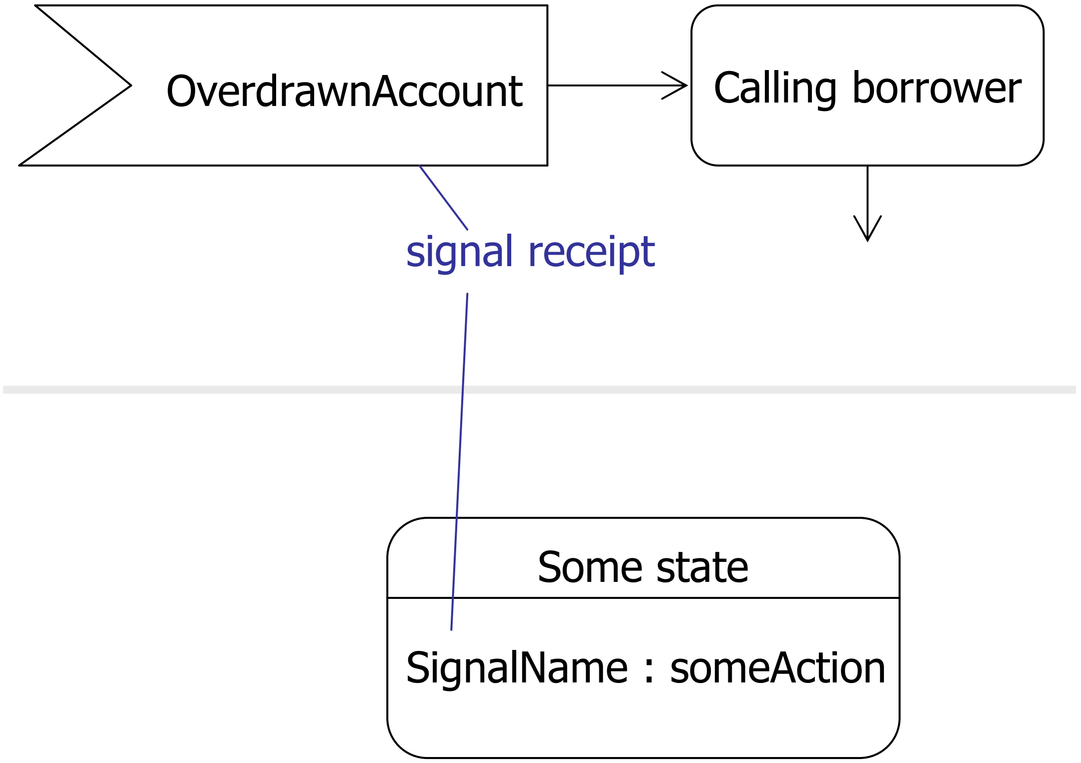
date : Date
accountNumber : long
amountOverdrawn : double



SimpleBankAccount
close()
InCredit
deposit(m)/ balance = balance + m
withdraw(m) [balance < m]
withdraw(m) [balance >= m]
RejectingWithdrawal
entry/ logRejectedWithdrawal()
AcceptingWithdrawal
entry/ balance = balance - m
OverdrawnAccount
send a signal

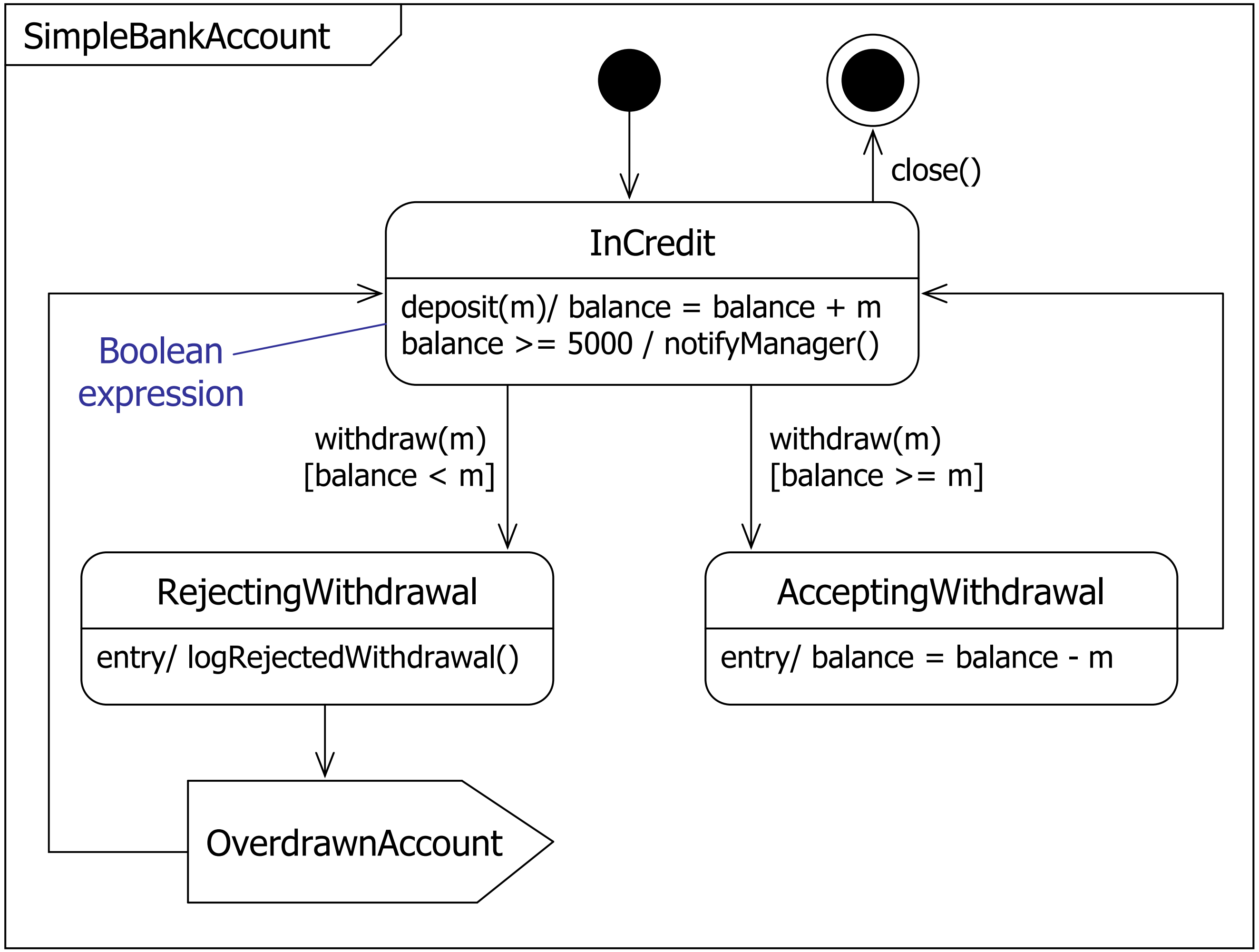
Receiving a signal

- You may show a signal receipt on a transition using a concave pentagon or as an internal transition state using standard notation



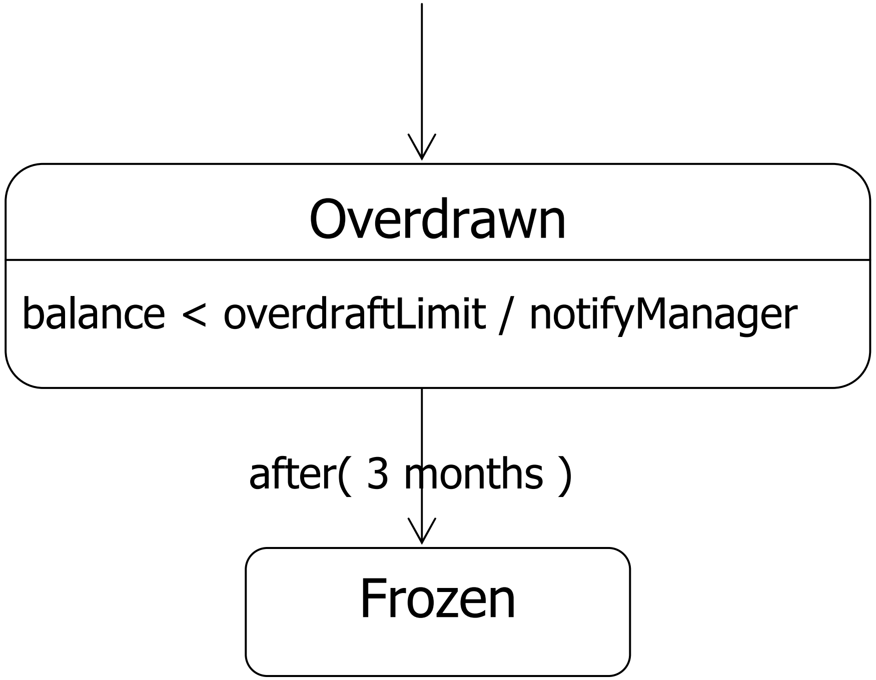
Change events

- The action is performed when the Boolean expression transitions from false to true
  - The event is *edge triggered* on a false to true transition
  - The values in the Boolean expression must be constants, globals or attributes of the context class
- A change event implies continually testing the condition whilst in the state


SimpleBankAccount
close()
InCredit
deposit(m)/ balance = balance + m
balance >= 5000 / notifyManager()
Boolean expression
withdraw(m) [balance < m]
withdraw(m) [balance >= m]
RejectingWithdrawal
entry/ logRejectedWithdrawal()
AcceptingWithdrawal
entry/ balance = balance - m
OverdrawnAccount

Time events

- Time events occur when a time expression becomes true
- There are two keywords, after and when
- Elapsed time:
  - after(3 months)
- Absolute time:
  - when(date =20/3/2000)



Context: CreditAccount class

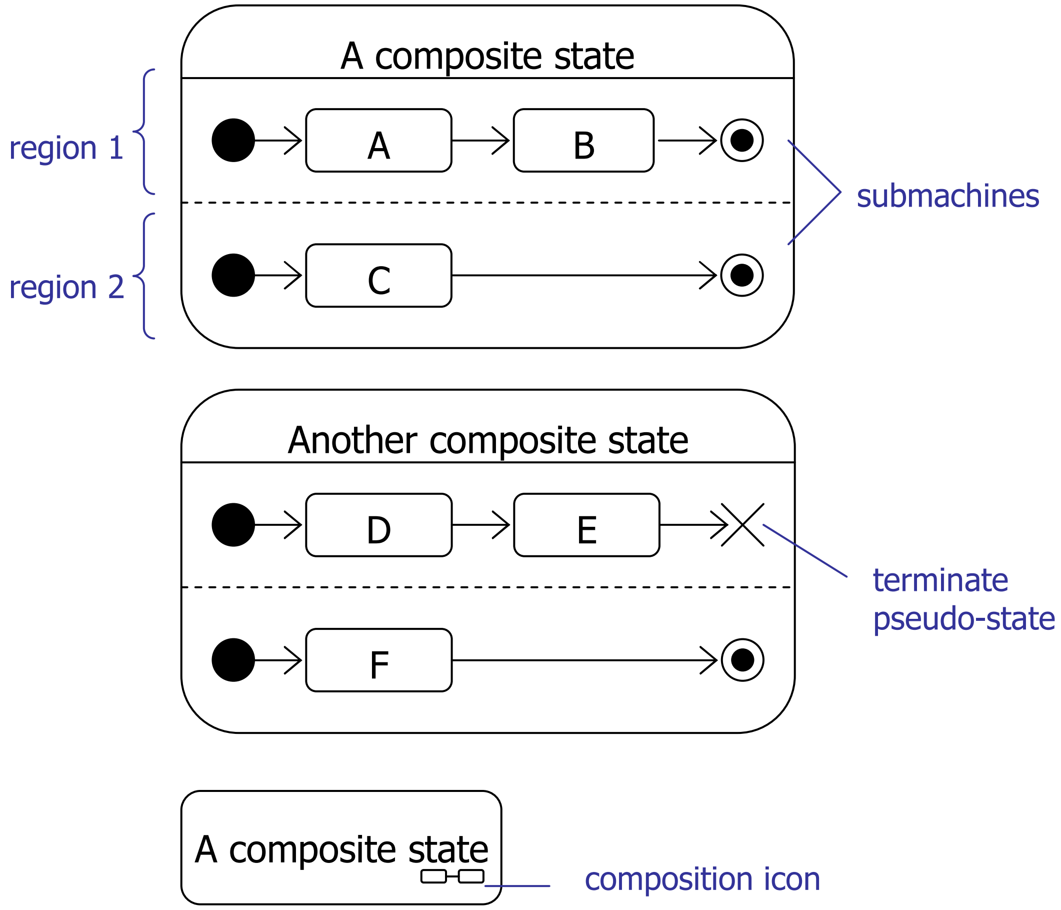
Summary

- We have looked at:
  - Behavioral state machines
  - Protocol state machines
  - States
    - Actions
      - Exit and entry actions
    - Activities
  - Transitions
    - Guard conditions
    - Actions
  - Events
    - Call, signal, change and time

Design - advanced state machines

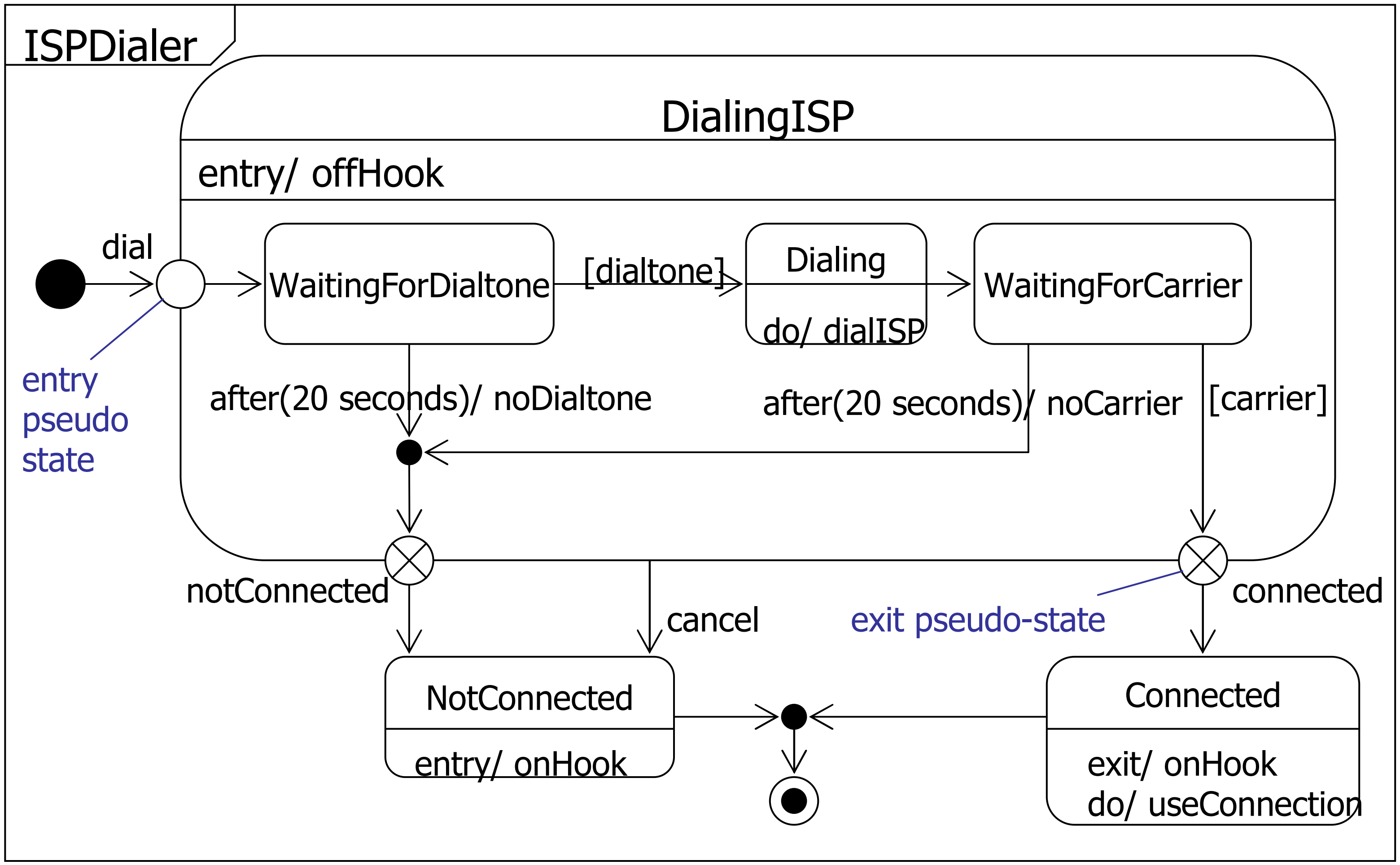
Composite states

- Have one or more regions that each contain a nested submachine
  - Simple composite state
    - exactly one region
  - Orthogonal composite state
    - two or more regions
- The final state terminates its enclosing region – all other regions continue to execute
- The terminate pseudo-state terminates the whole state machine
- Use the composition icon when the submachines are hidden



Simple composite states

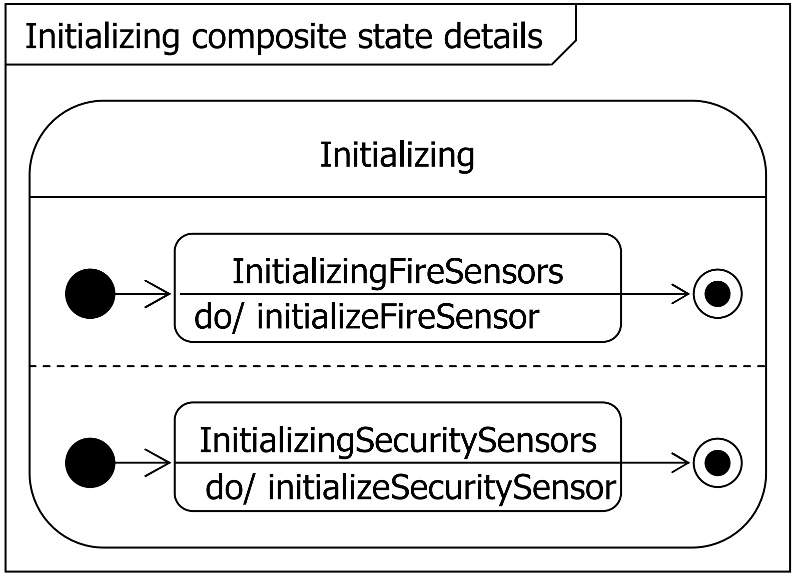
- Contains a single region



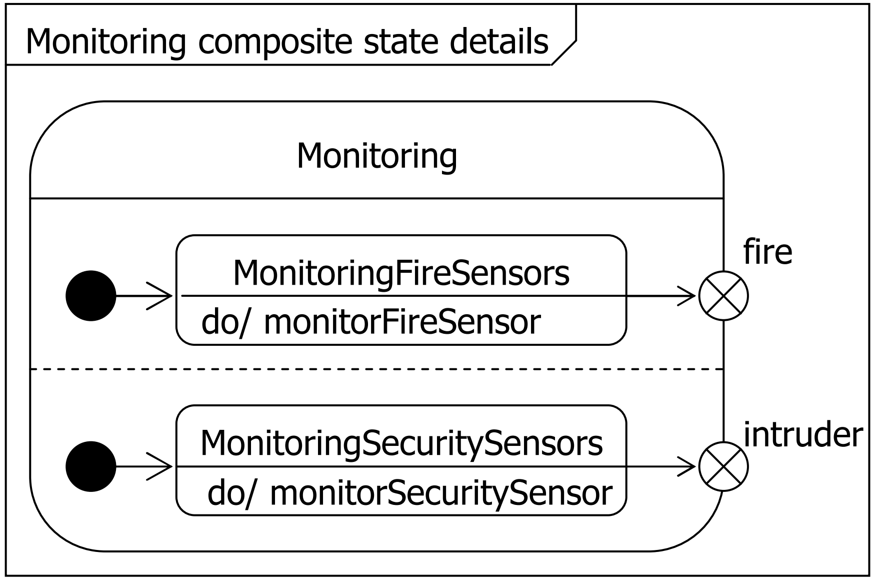
Orthogonal composite states

- Has two or more regions
- When we enter the superstate, both submachines start executing concurrently - this is an implicit fork

Synchronized exit - exit the superstate when *both* regions have terminated

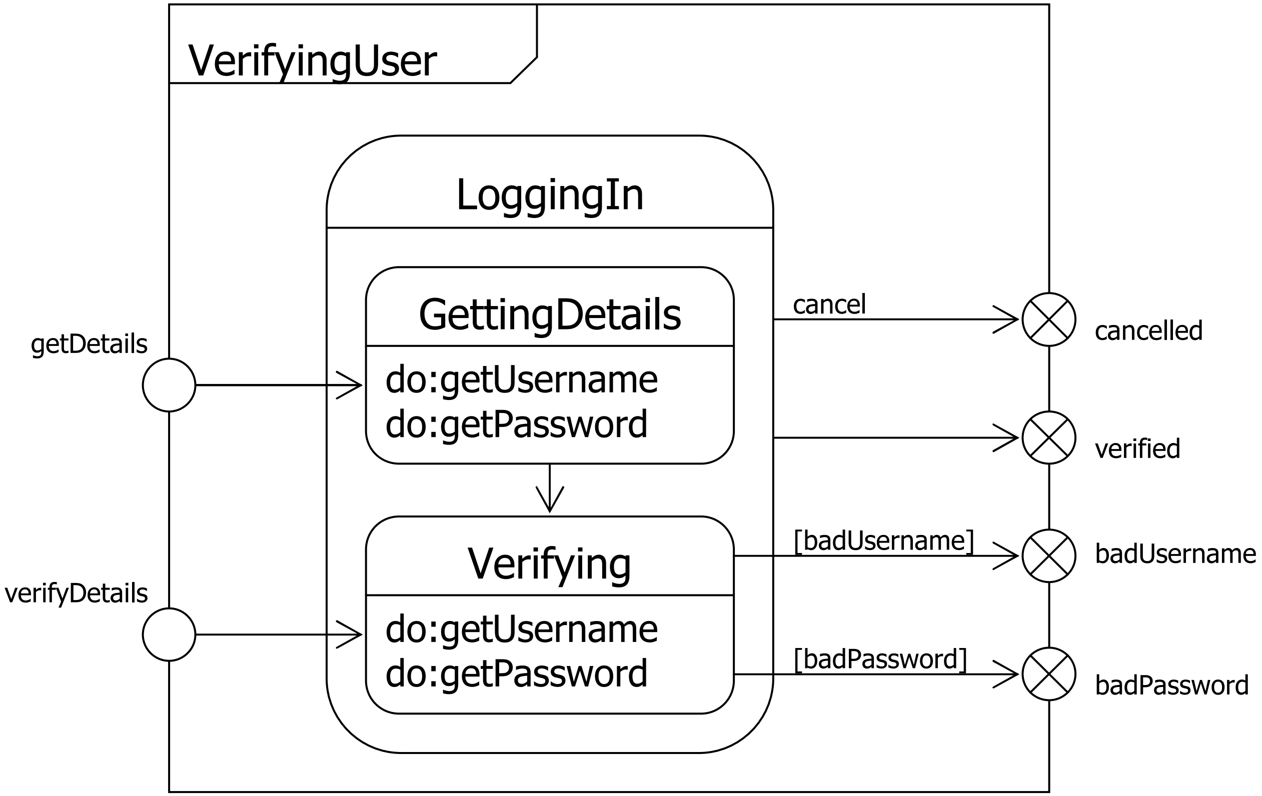

Initializing composite state details
Initializing
InitializingFireSensors
do/ initializeFireSensor
InitializingSecuritySensors
do/ initializeSecuritySensor

Unsynchronized exit - exit the superstate when *either* region terminates. The other region continues


Monitoring composite state details
Monitoring
MonitoringFireSensors
do/ monitorFireSensor
fire
MonitoringSecuritySensors
do/ monitorSecuritySensor
intruder

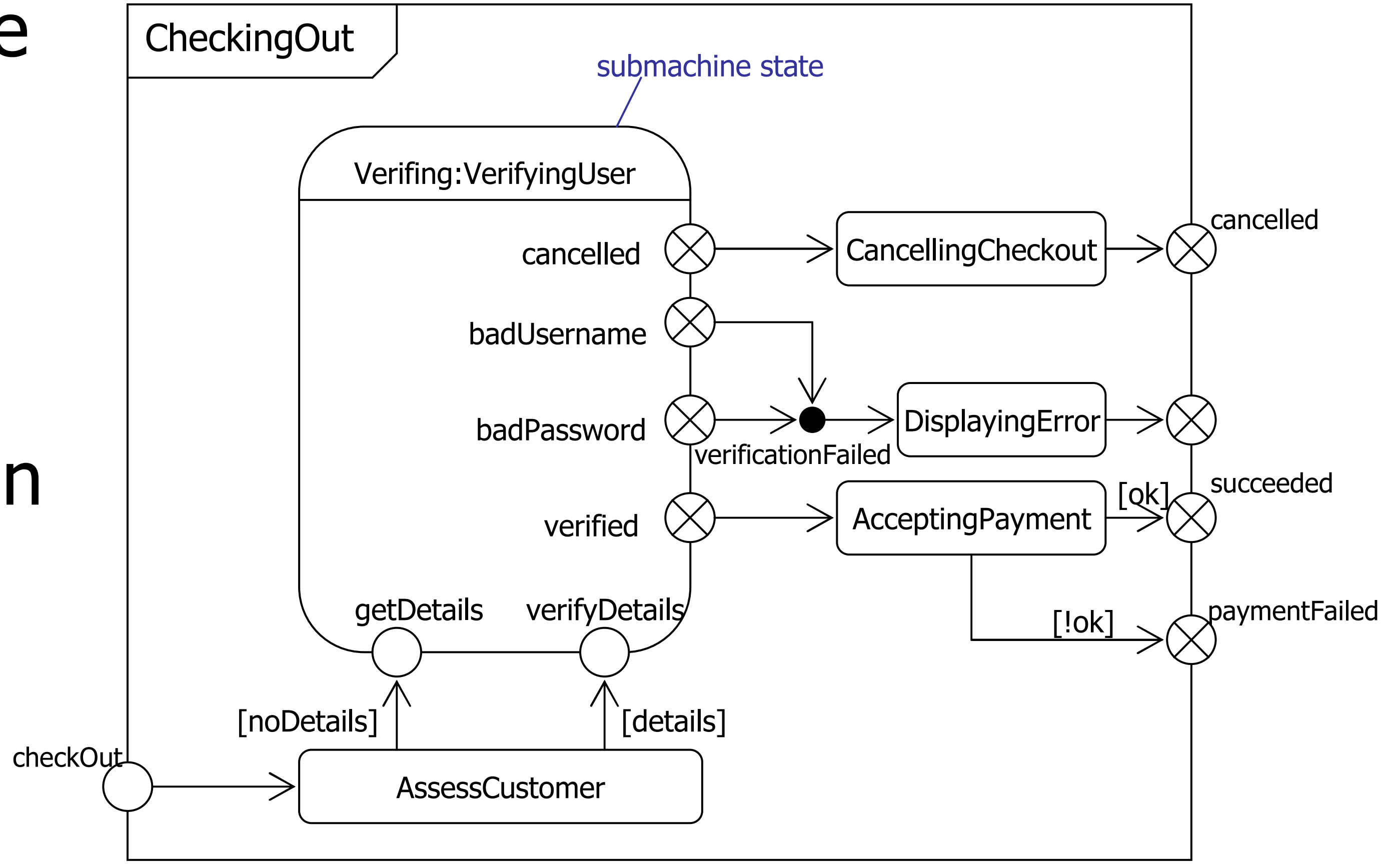
Submachine states

- If we want to refer to this state machine in other state machines, without cluttering the diagrams, then we must use a *submachine state*
- Submachine states reference another state machine
- Submachine states are semantically equivalent to composite states



Submachine state syntax

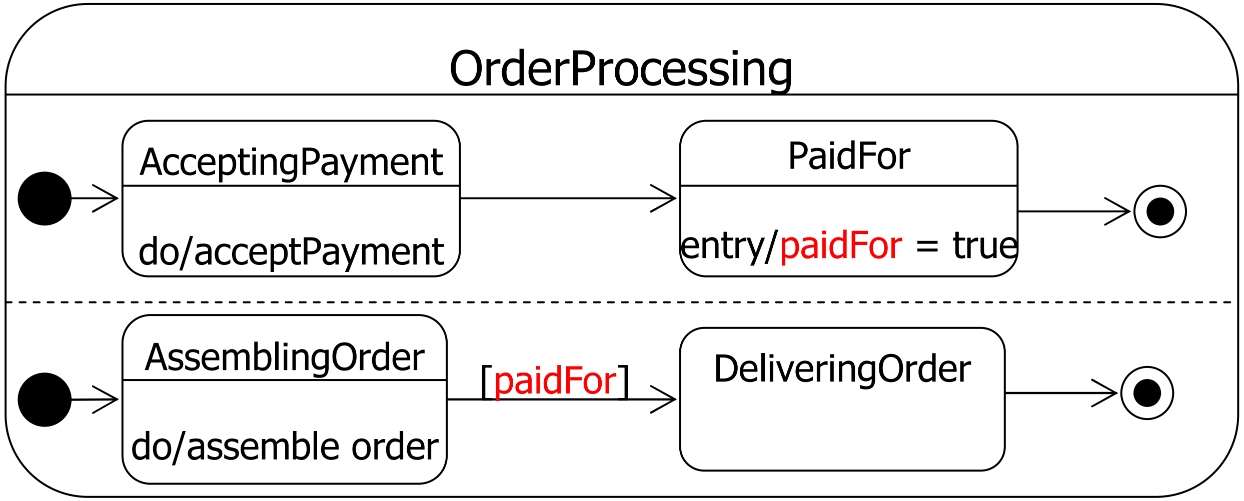
- A submachine state is equivalent to including a copy of the submachine in place of the submachine state



Submachine communication

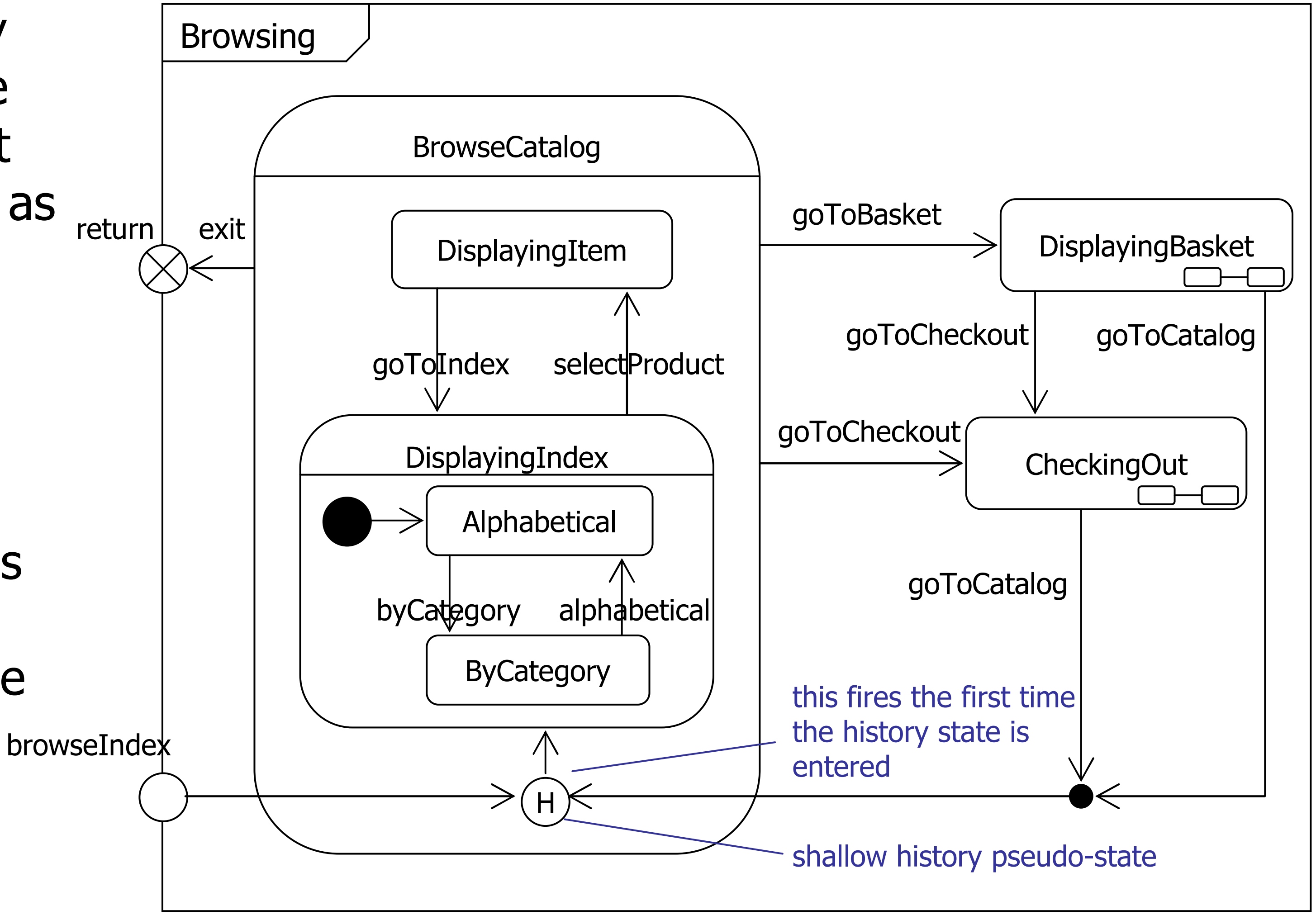
- We often need two submachines to communicate
- Synchronous communication can be achieved by a join
- Asynchronous communication is achieved by one submachine setting a flag for another one to process in its own time.
  - Use attributes of the context object as flags

Submachine communication using the attribute PaidFor as a flag: The upper submachine sets the flag and the lower submachine uses it in a guard condition



Shallow history

- Shallow history remembers the last substate at the same level as the shallow history pseudo state
- Next time the super state is entered there is an automatic transition to the remembered substate

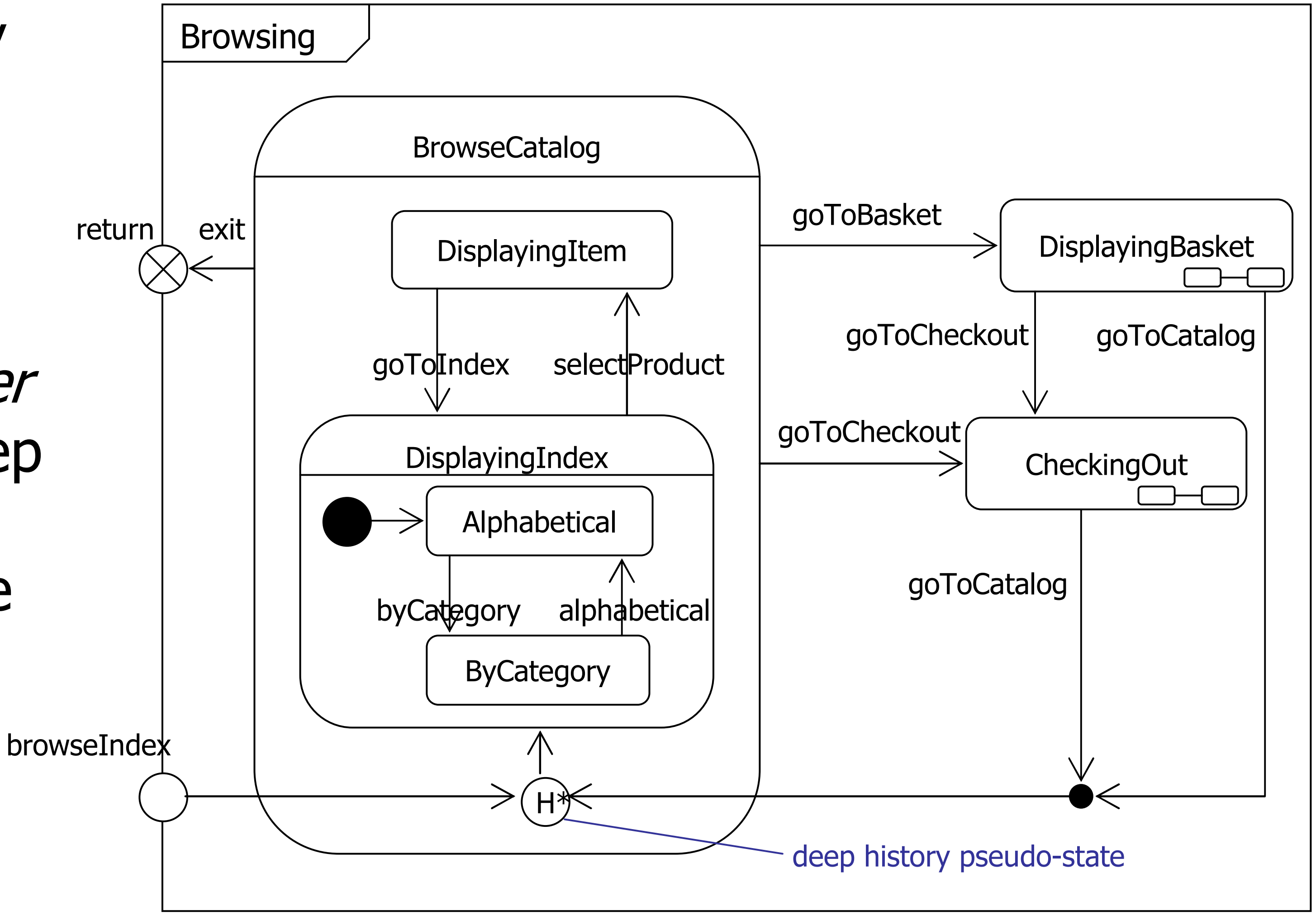


this fires the first time the history state is entered

shallow history pseudo-state

Deep history

- Deep history remembers the last substate at the same level *or lower* than the deep history pseudo state



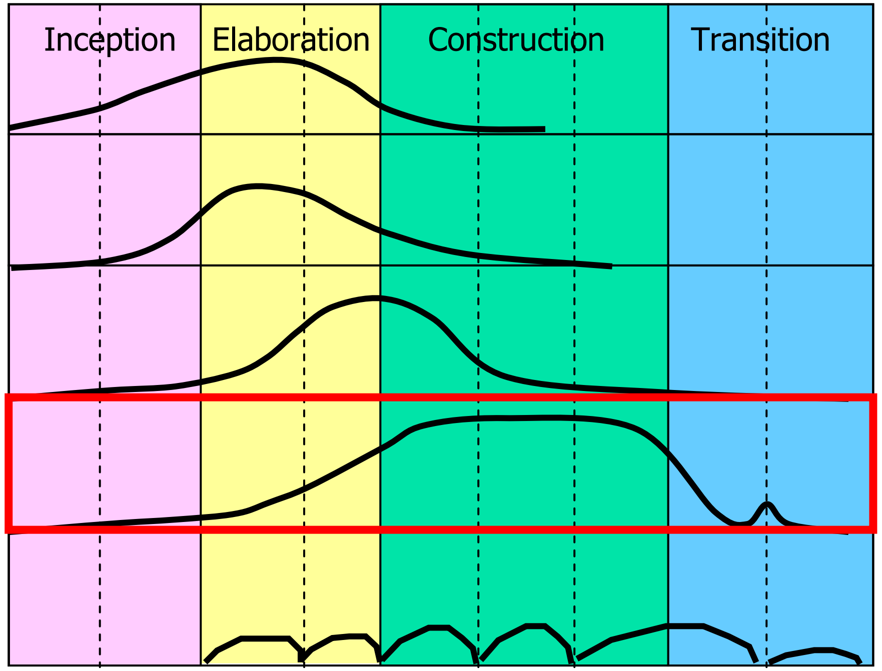
Summary

- We have explored advanced aspects of state machines including:
  - Simple composite states
  - Orthogonal composite states
  - Submachine communication
    - Attribute values
  - Submachine states
  - Shallow history
  - Deep history

Implementation - introduction

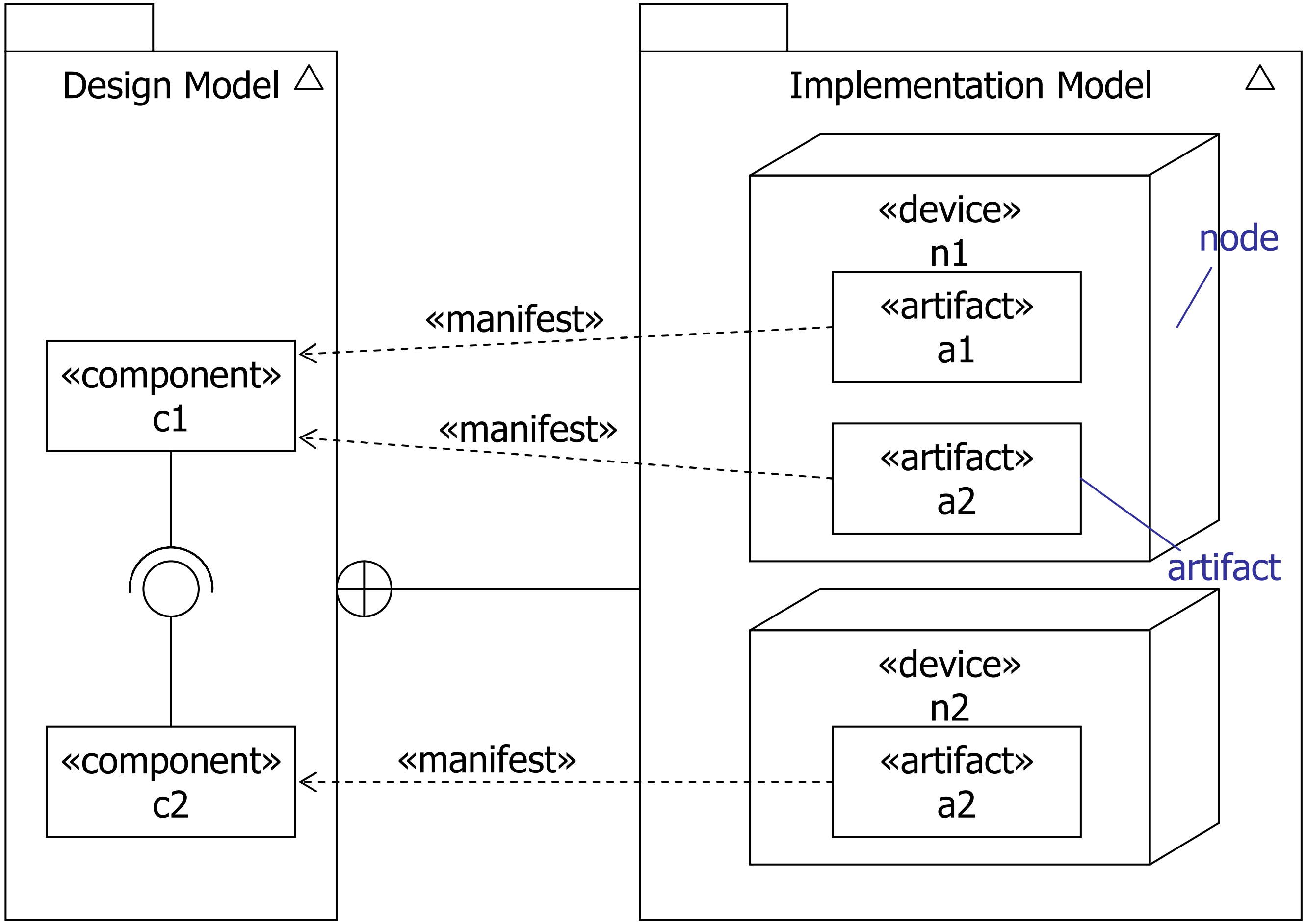
Implementation - purpose

- To implement the design classes and components
  - To create an implementation model
- To convert the Design Model into an executable program

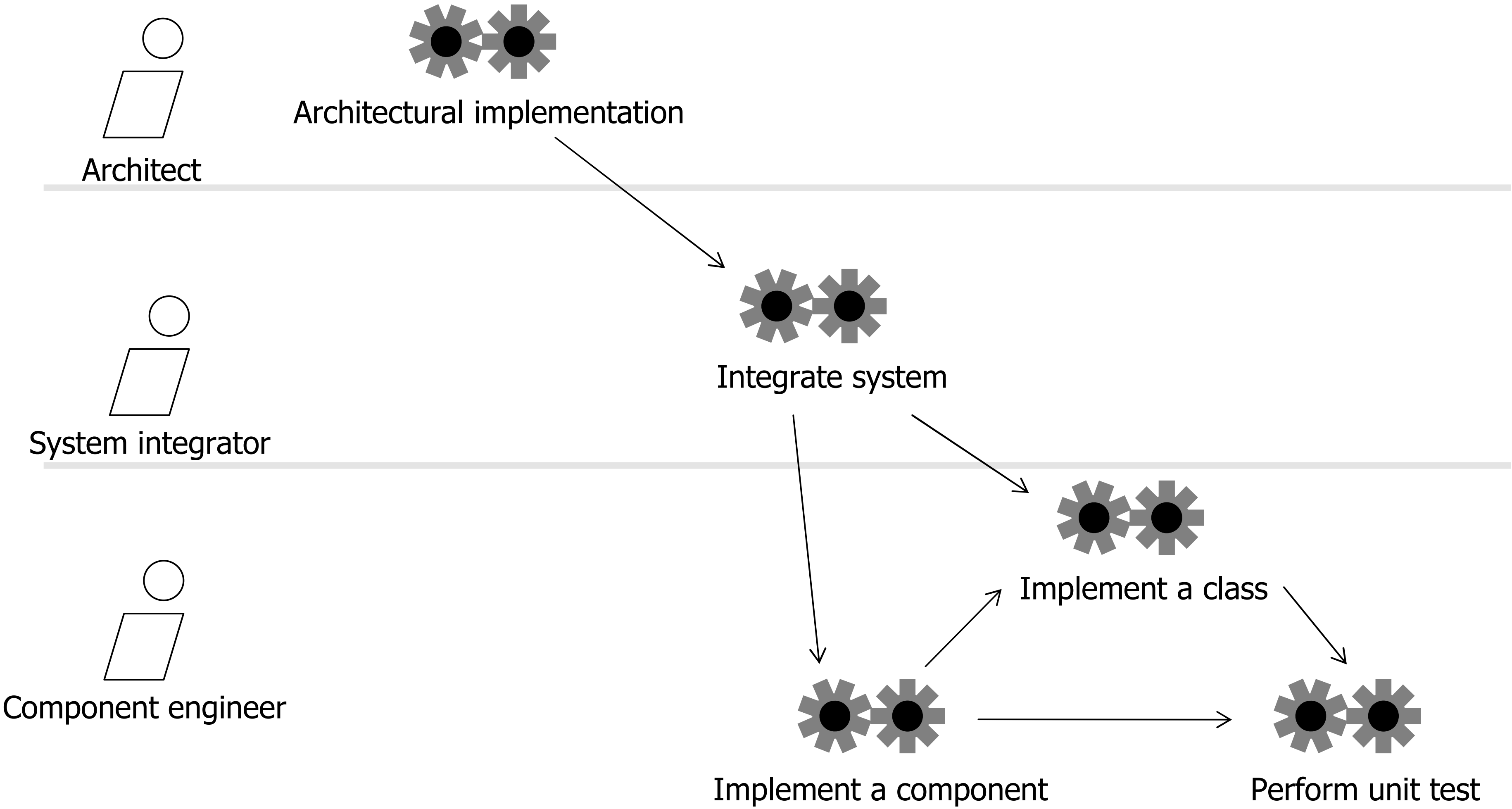


Implementation artifacts - metamodel

- The implementation model is part of the design model. It comprises:
  - Component diagrams showing components and the artifacts that realize them
  - Deployment diagrams showing artifacts deployed on nodes
- Components are manifest by artifacts
- Artifacts are deployed on nodes



Implementation workflow detail



Summary

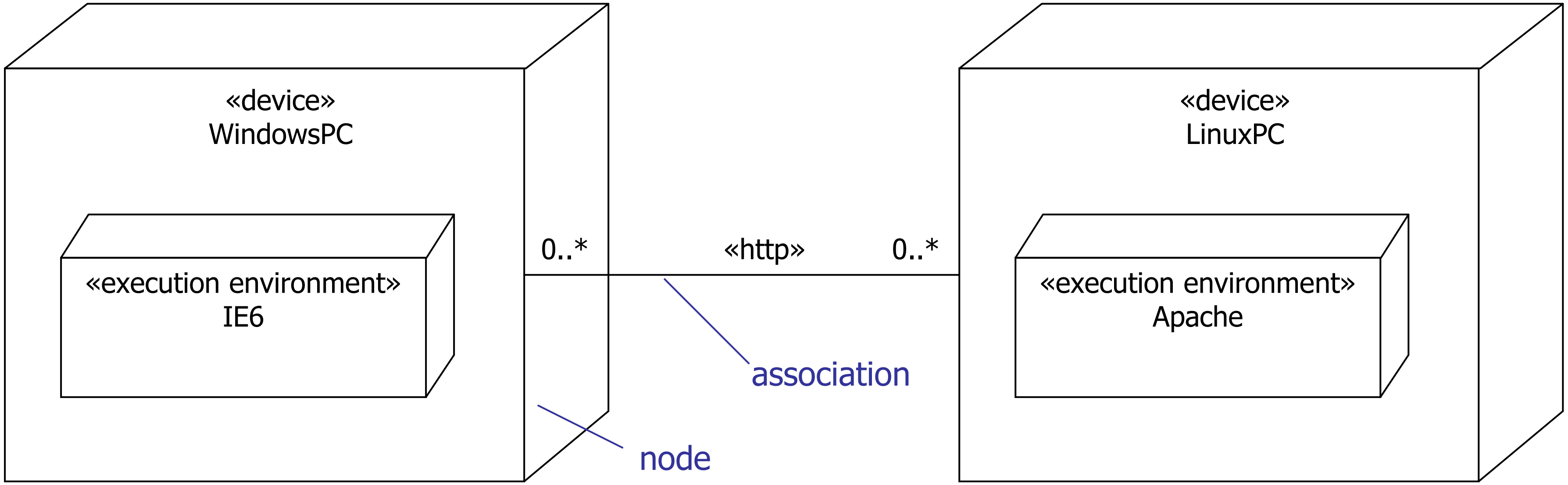
- Implementation begins in the last part of the elaboration phase and is the primary focus throughout later stages of the construction phase
- Purpose – to create an executable system
- artifacts:
    - component diagrams
        - components and artifacts
    - deployment diagrams
        - nodes and artifacts

Implementation - deployment

Deployment model

- The deployment model is an object model that describes how functionality is distributed across physical nodes
    - It models the mapping between the software architecture and the physical system architecture
- It models the system's physical architecture as artifacts deployed on nodes
    - Each node is a type of computational resource
    - Nodes have relationships that represent methods of communication between them e.g. http, iiop, netbios
    - Artifacts represent physical software e.g. a JAR file or .exe file
- Design - we may create a first-cut deployment diagram:
    - Focus on the big picture - nodes or node instances and their connections
    - Leave detailed artifact deployment to the implementation workflow
- Implementation - finish the deployment diagram:
    - Focus on artifact deployment on nodes

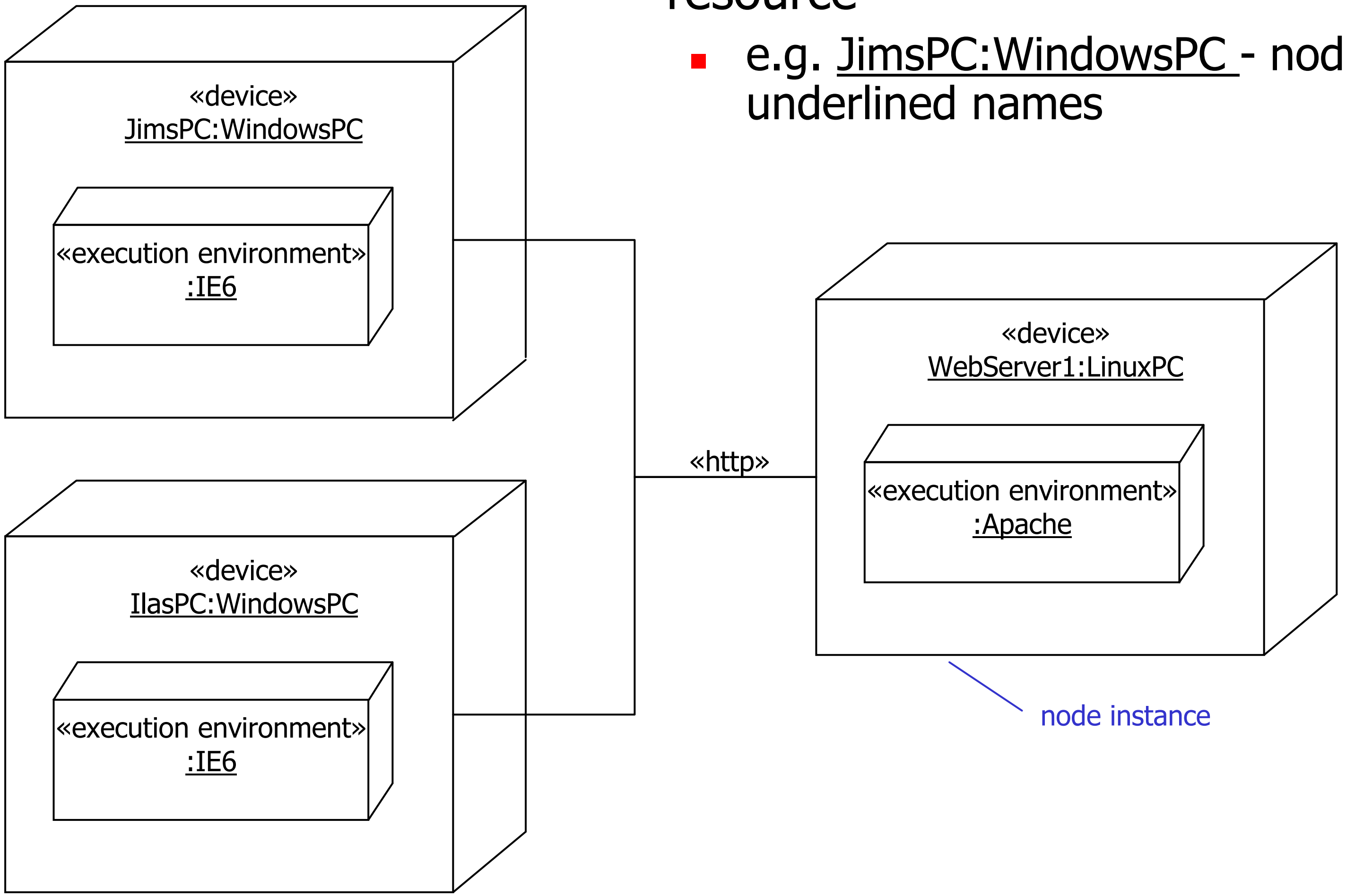
Nodes – descriptor form



- A node represents a type of computational resource
  - e.g. a WindowsPC
- Standard stereotypes are «device» and «execution environment»

Nodes – instance form

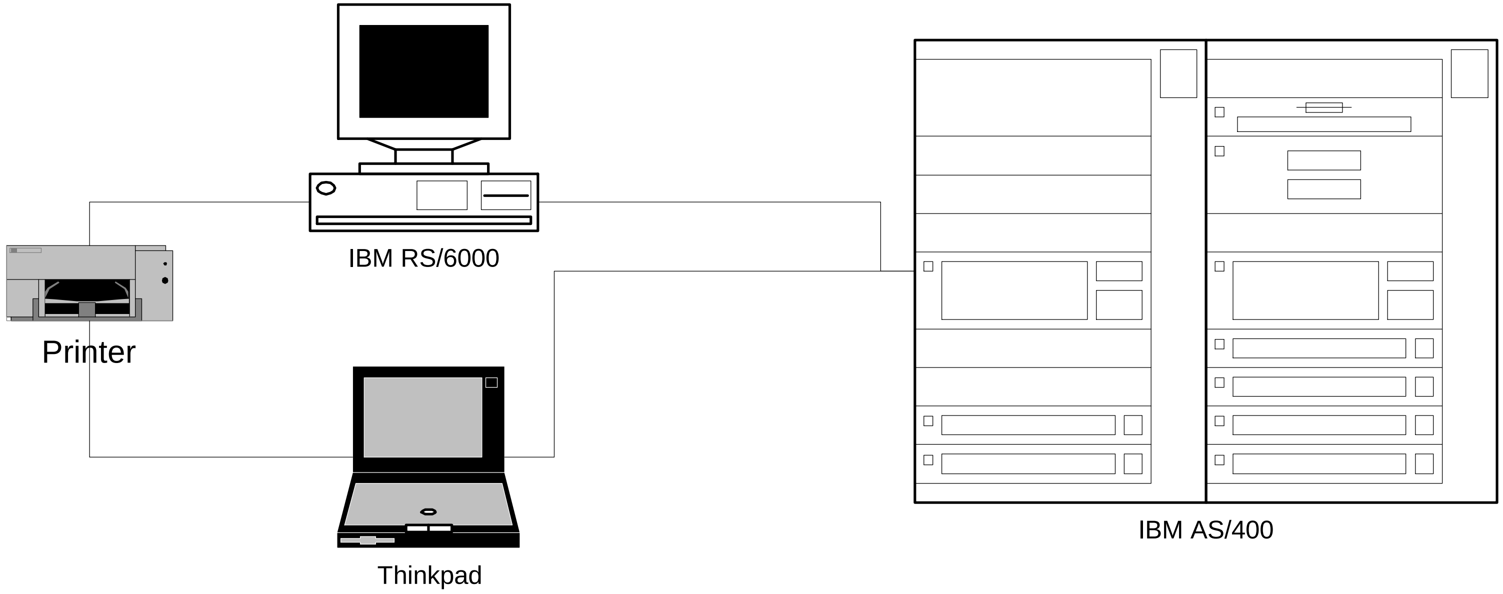
- A node instance represents an actual physical resource
  - e.g. JimsPC:WindowsPC - node instances have underlined names



node instance

zühlke

Stereotyping nodes



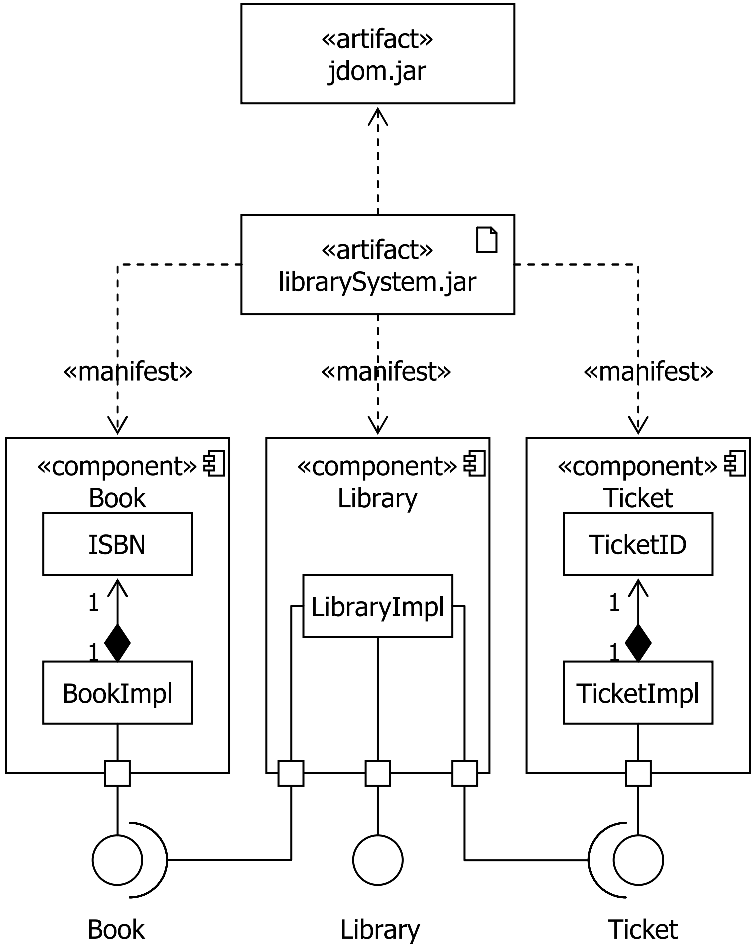
- It’s very useful to use lots of stereotyping on the deployment diagram to make it as clear and readable as possible

Artifacts

- An artifact represents a type of concrete, real-world thing such as a file
  - Can be deployed on nodes
- Artifact instances represent particular *copies* of artifacts
  - Can be deployed on node instances
- An artifact can manifest one or more components
  - The artifact is the represents the thing that is the physical manifestation of the component (e.g. a JAR file)

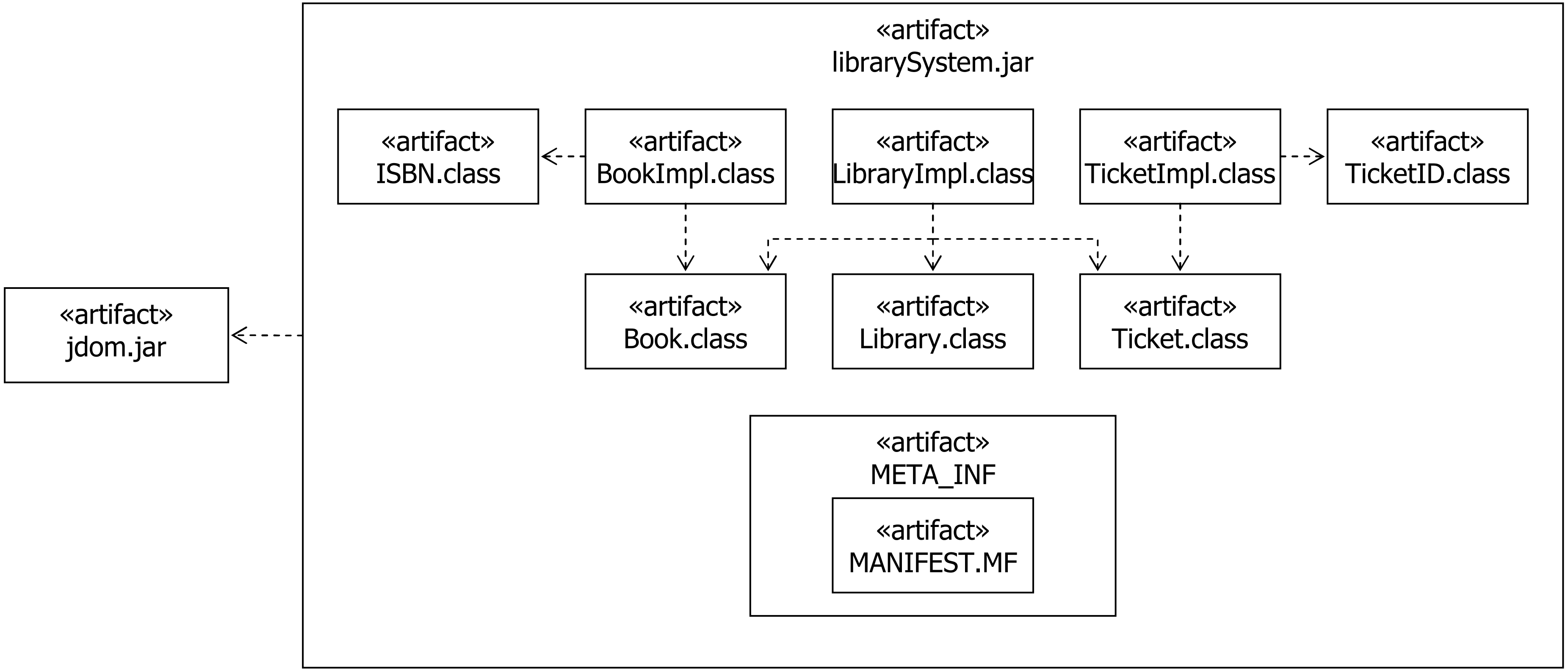
Artifacts and components

- Artifacts provide the physical manifestation for one or more components
- Artifacts may have the artifact icon in their upper right hand corner
- Artifacts can contain other artifacts
- Artifacts can depend on other artifacts



Artifact relationships

- An artifact may depend on other artifacts when a component in the client artifact depends on a component in the supplier artifact in some way



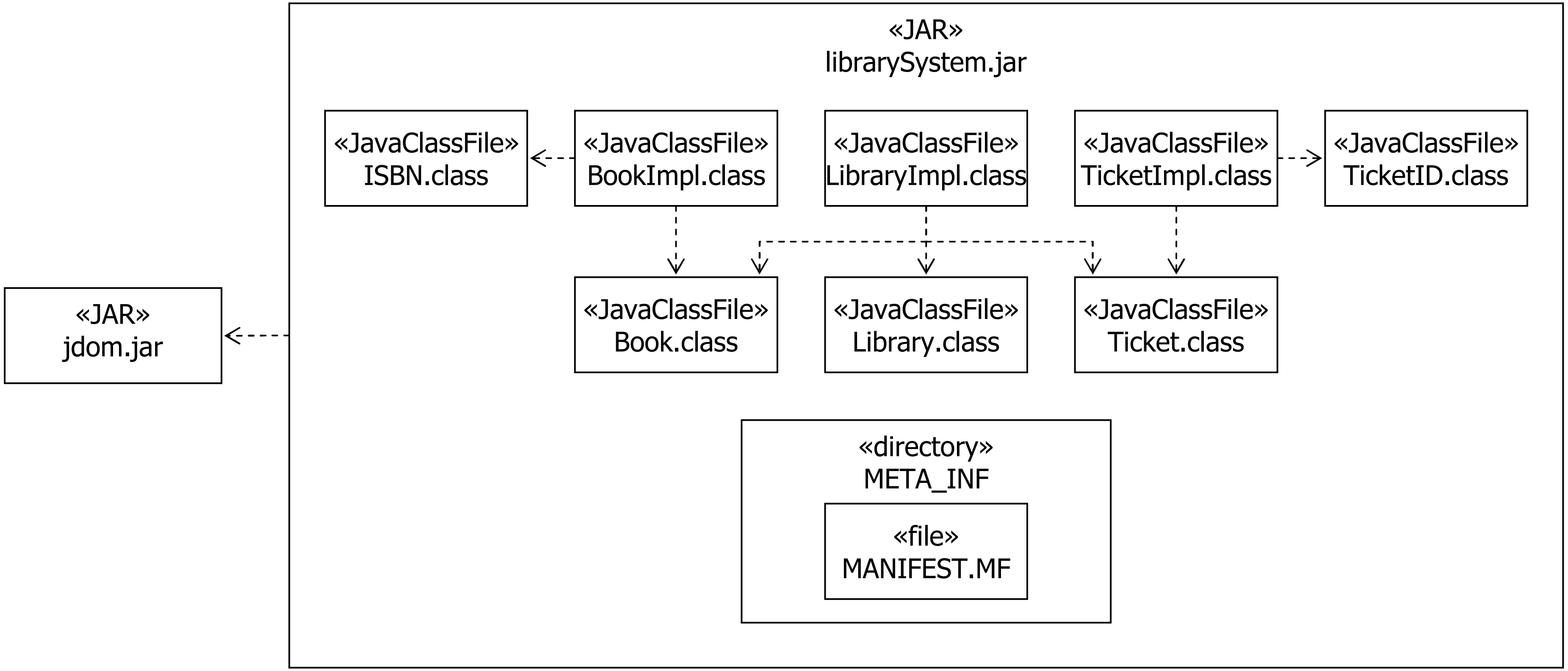
Artifact standard stereotypes

- UML 2 provides a small number of standard stereotypes for artifacts

artifact stereotype	semantics
«file»	A physical file
«deployment spec»	A specification of deployment details (e.g. web.xml in J2EE)
«document»	A generic file that holds some information
«executable»	An executable program file
«library»	A static or dynamic library such as a dynamic link library (DLL) or Java Archive (JAR) file
«script»	A script that can be executed by an interpreter
«source»	A source file that can be compiled into an executable file

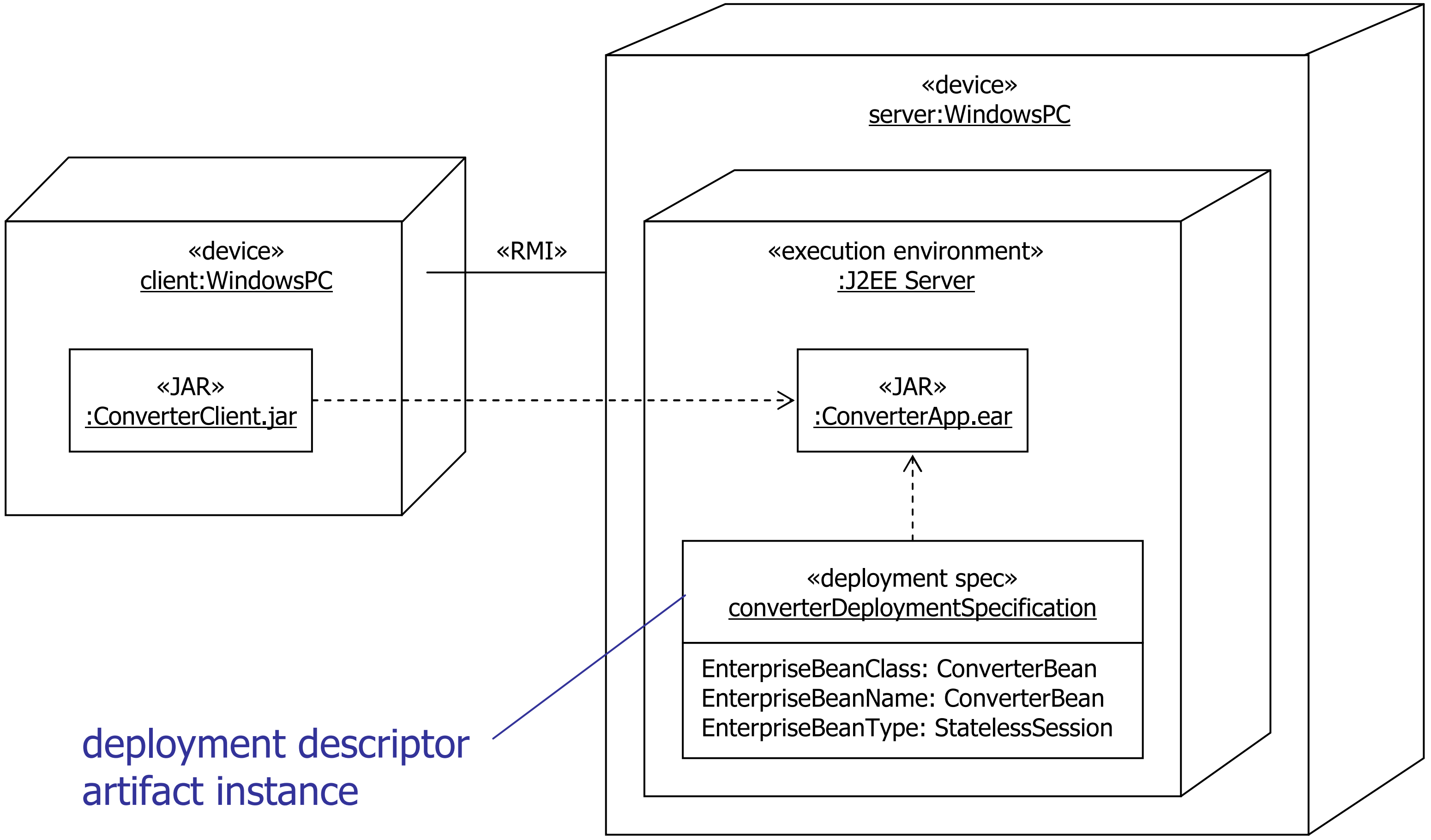
Stereotyping artifacts

- Applying a UML profile can clarify component diagrams
  - e.g. applying the example Java profile from the UML 2 specification…



Deployment

- Artifacts are deployed on nodes, artifact instances are deployed on node instances



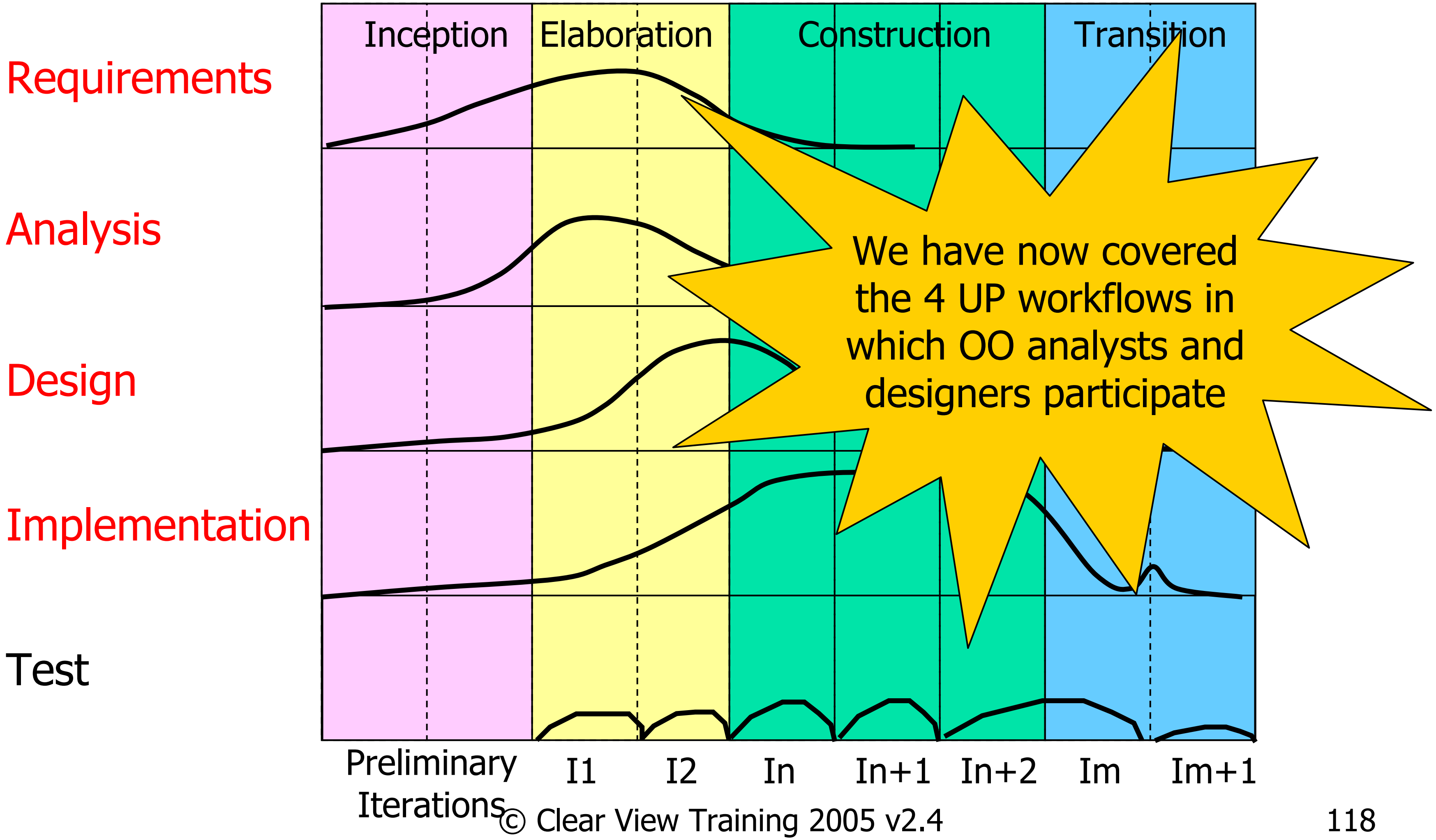
deployment descriptor artifact instance

Summary

- The descriptor form deployment diagram
  - Allows you to show how functionality represented by artefacts is distributed across nodes
  - Nodes represent types of physical hardware or execution environments
- The instance form deployment diagram
  - Allows you to show how functionality represented by artefact instances is distributed across node instances
  - Node instances represent actual physical hardware or execution environments

Course summary

UP phases and workflows



We have now covered the 4 UP workflows in which OO analysts and designers participate

Next steps…

- There is a lot of useful information at www.clearviewtraining.com:
  - UML resources for our books:
    - "UML 2 and the Unified Process"
    - "Enterprise Patterns and MDA"
  - Advanced UML modelling techniques
    - Literate modeling
    - Archetype patterns
  - SUMR - open source use case modeling tools
  - Speak directly to the course author Dr. Jim Arlow:
    - Jim.Arlow@clearviewtraining.com
- Further training, mentoring and consultancy in all aspects of object technology and project management is available from:
  - Zuhlke Engineering Limited (UK), Zühlke Engineering AG (Switzerland) and Zühlke Engineering GmbH (Germany) - www.zuhlke.com

Finally…

- We hope you enjoyed this course and that we'll see you again soon!
- We'd find it really useful if you'd fill in your course evaluation forms before leaving.