OO Analysis and Design with UML 2 and UP

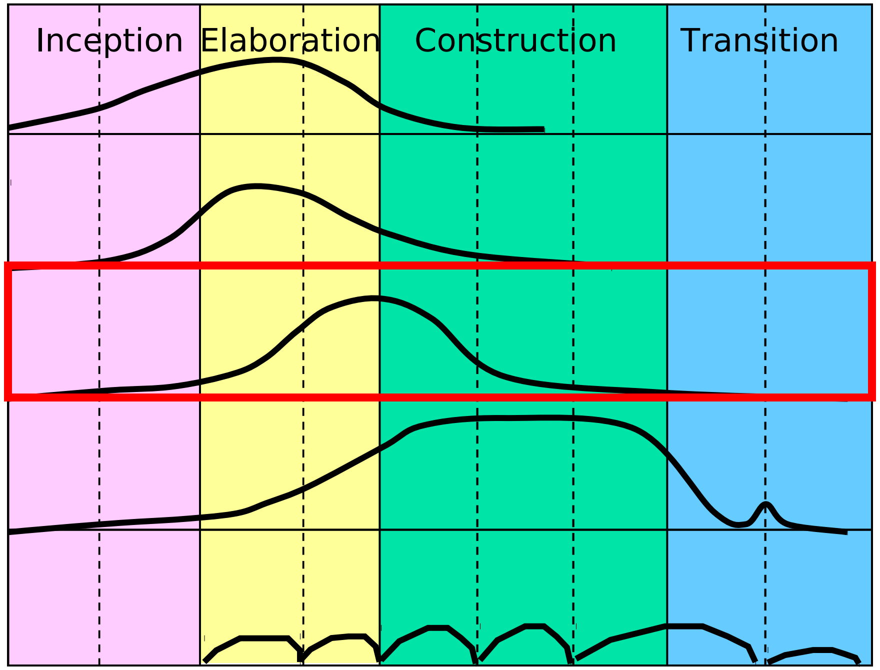
Dr. Jim Arlow,

Zuhlke Engineering Limited

Design - introduction

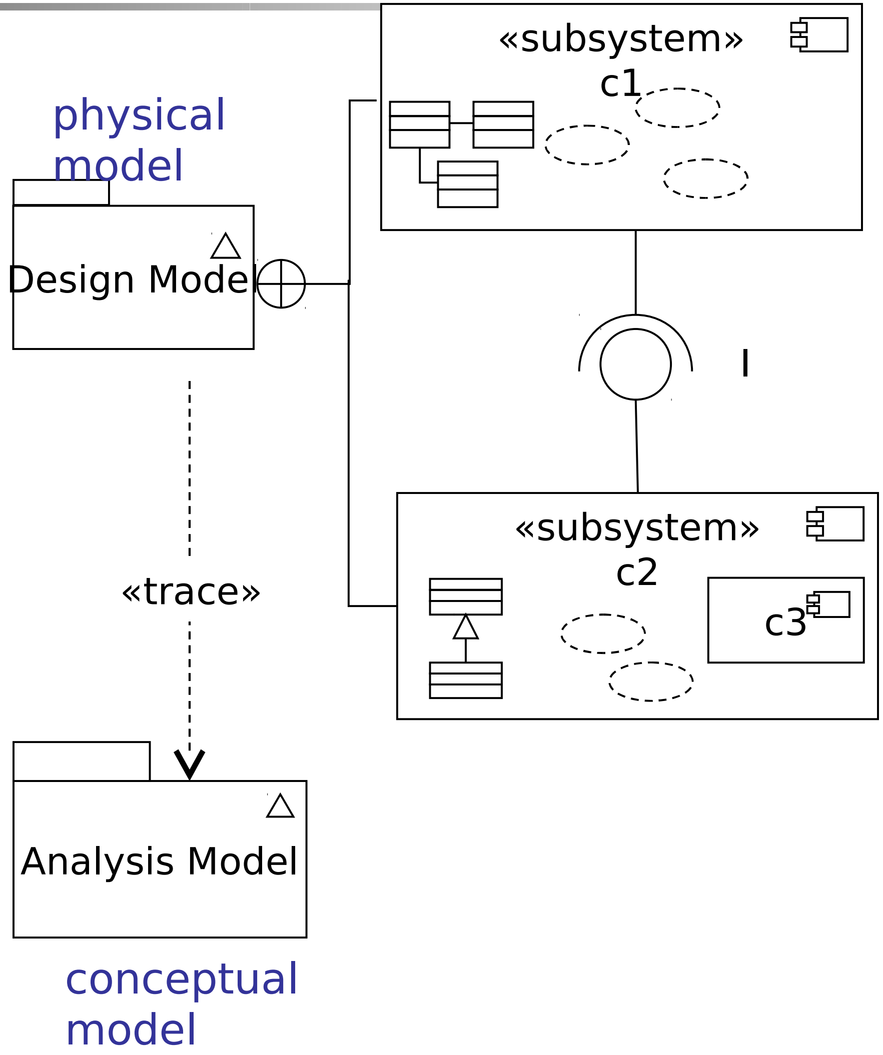
Design - purpose

- 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



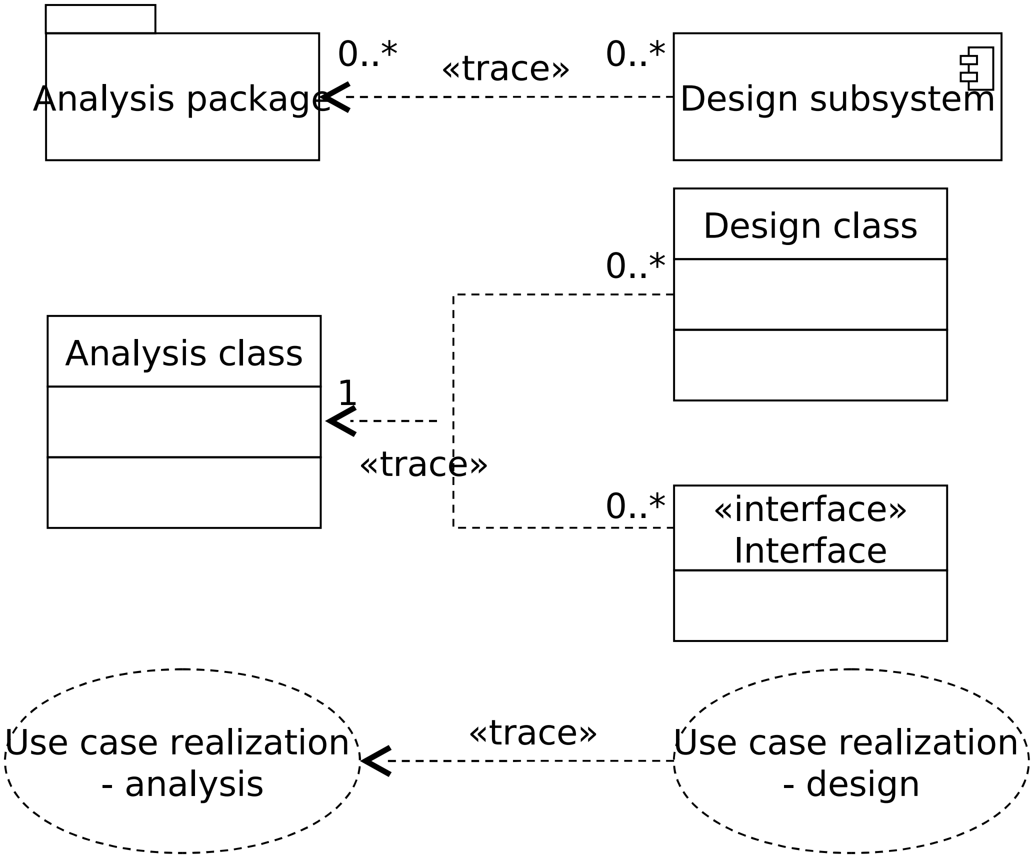
Design artifacts - metamodel

- 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

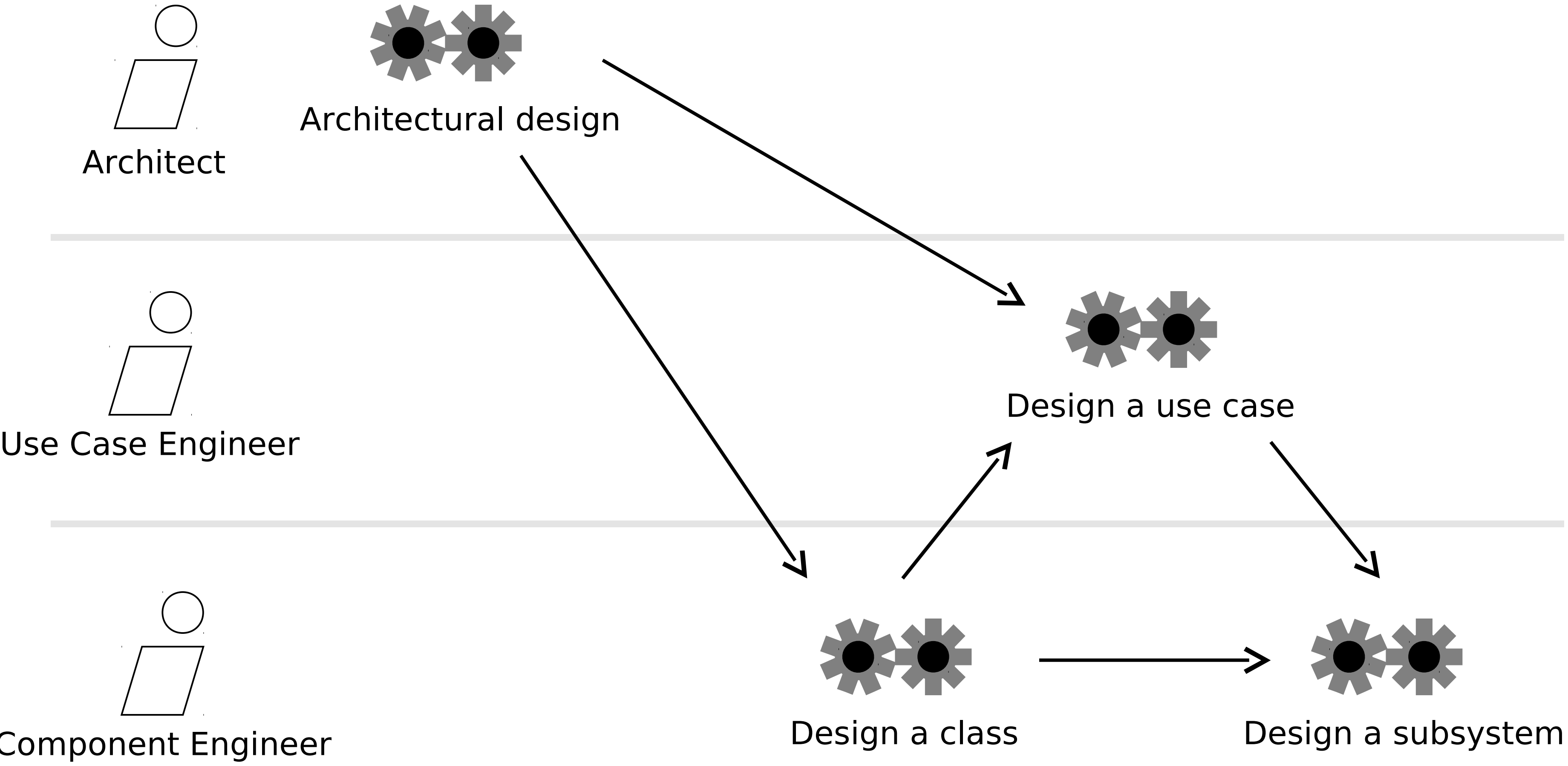


16.3.2

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



16.6

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.

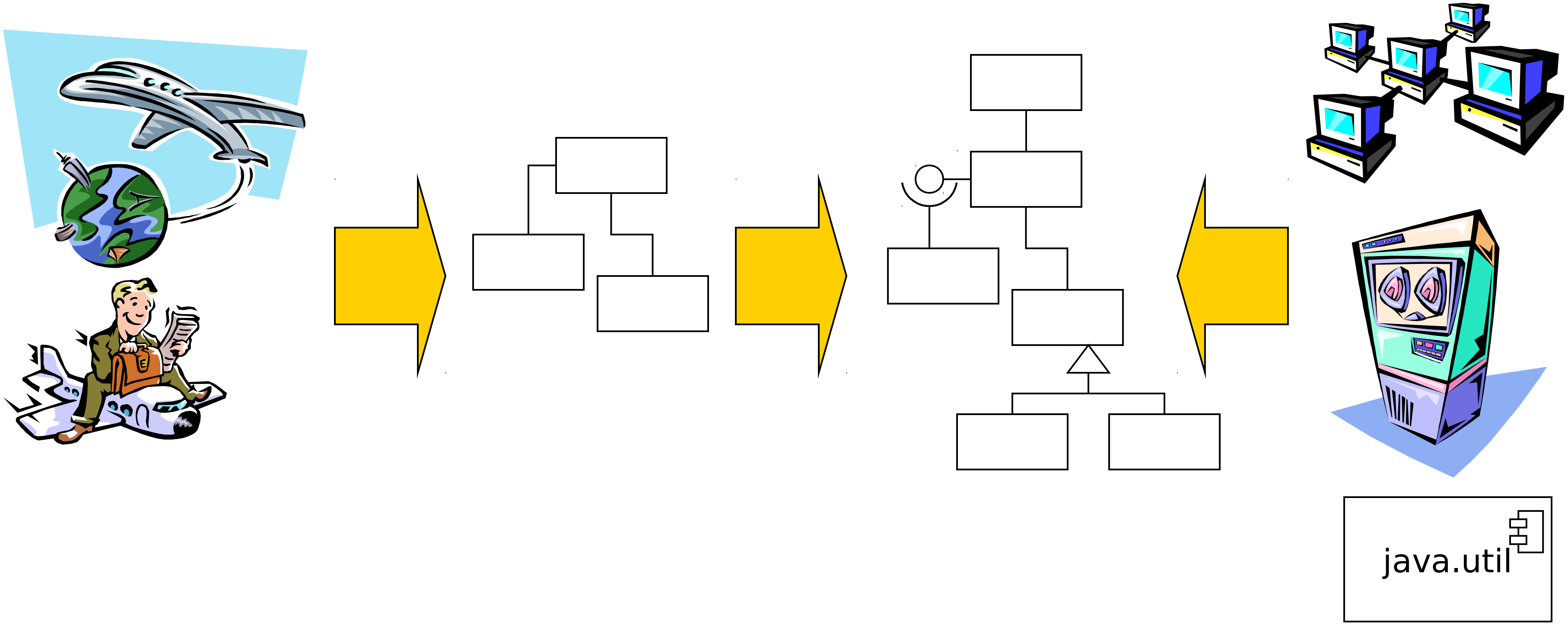
Sources of design classes

Problem domain

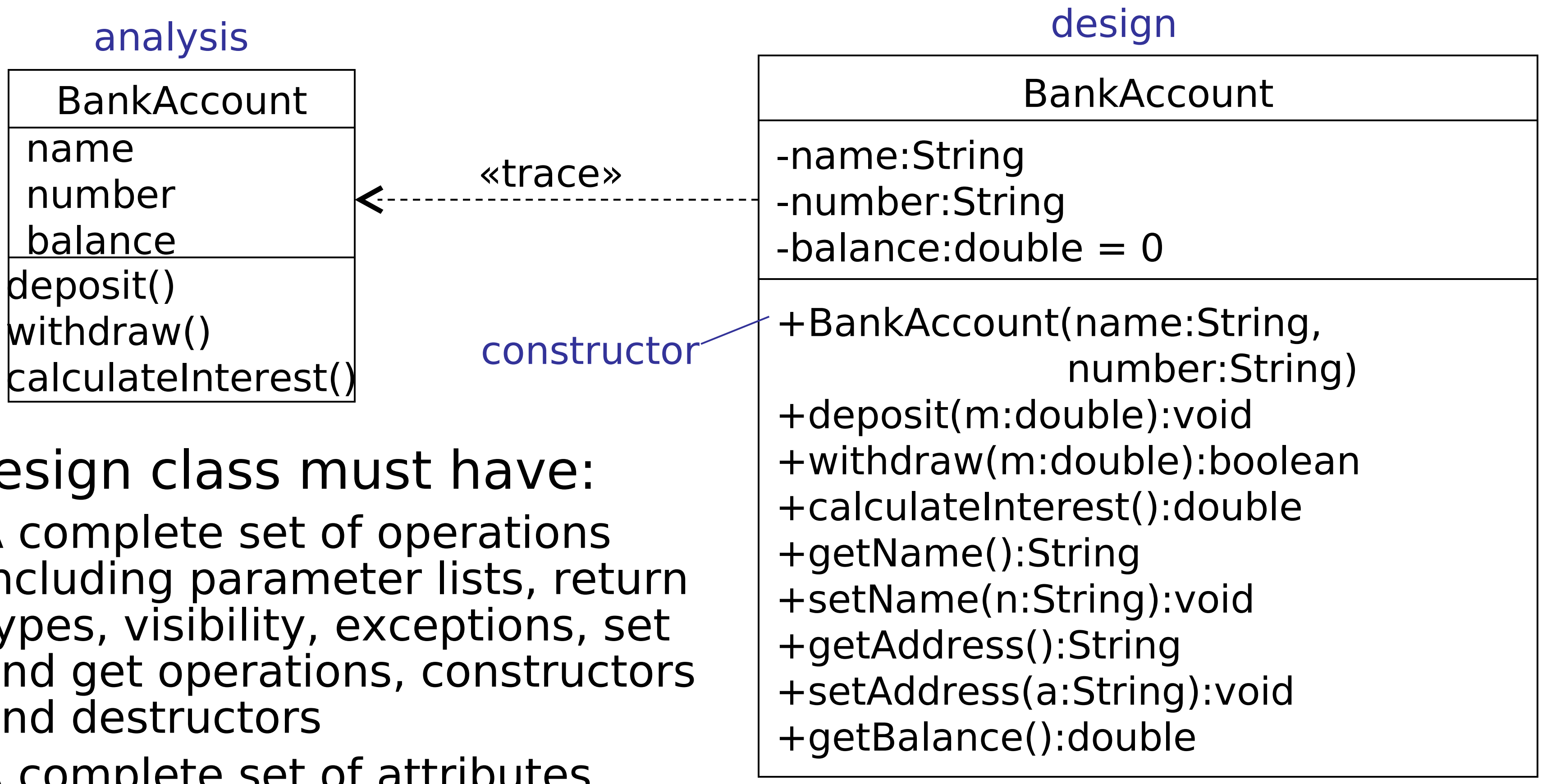
Analysis classes

Design classes

Solution domain



Anatomy of a design class


analysis

design

BankAccount

name
number
balance

deposit()
withdraw()
calculateInterest()

«trace»

constructor

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

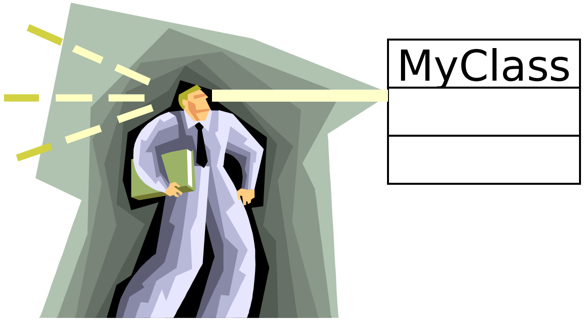
- A design class must have:
  - A complete set of operations including parameter lists, return types, visibility, exceptions, set and get operations, constructors and destructors
  - A complete set of attributes including types and default values

17.5

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

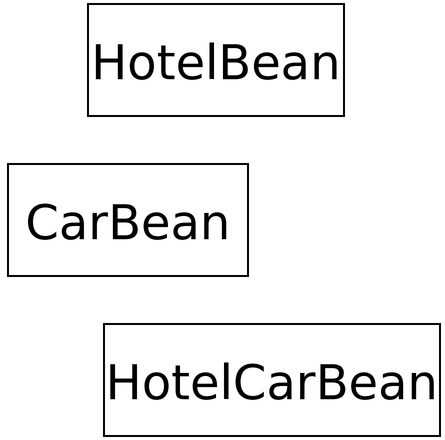
- Completeness:
  - Users of the class will make assumptions from the class name about the set of operations that it should make available
  - For example, a BankAccount class that provides a withdraw() operation will be expected to also provide a deposit() operation!
- Sufficiency:
  - A class should 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

17.5.3 17.5.4

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)

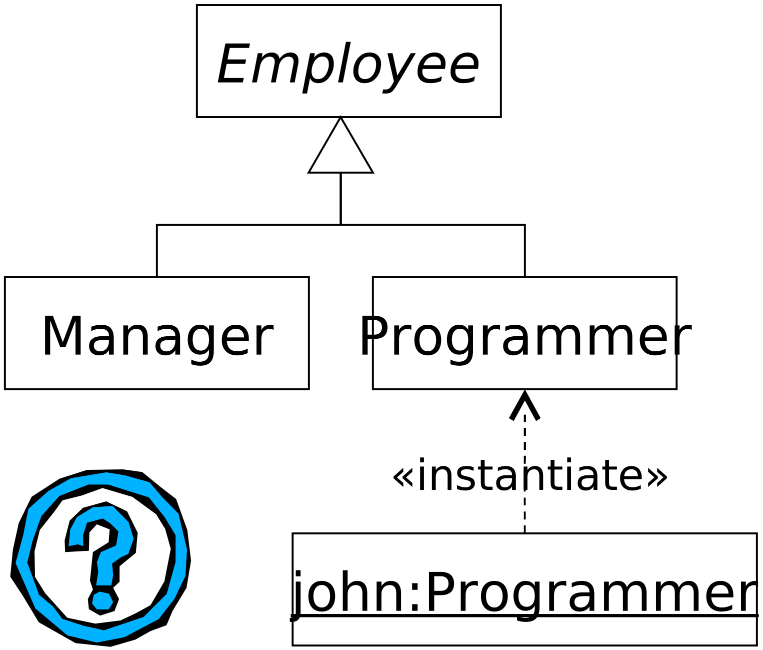


this example comes from a real system!
What’s wrong with it?

17.6

Aggregation vs. inheritance

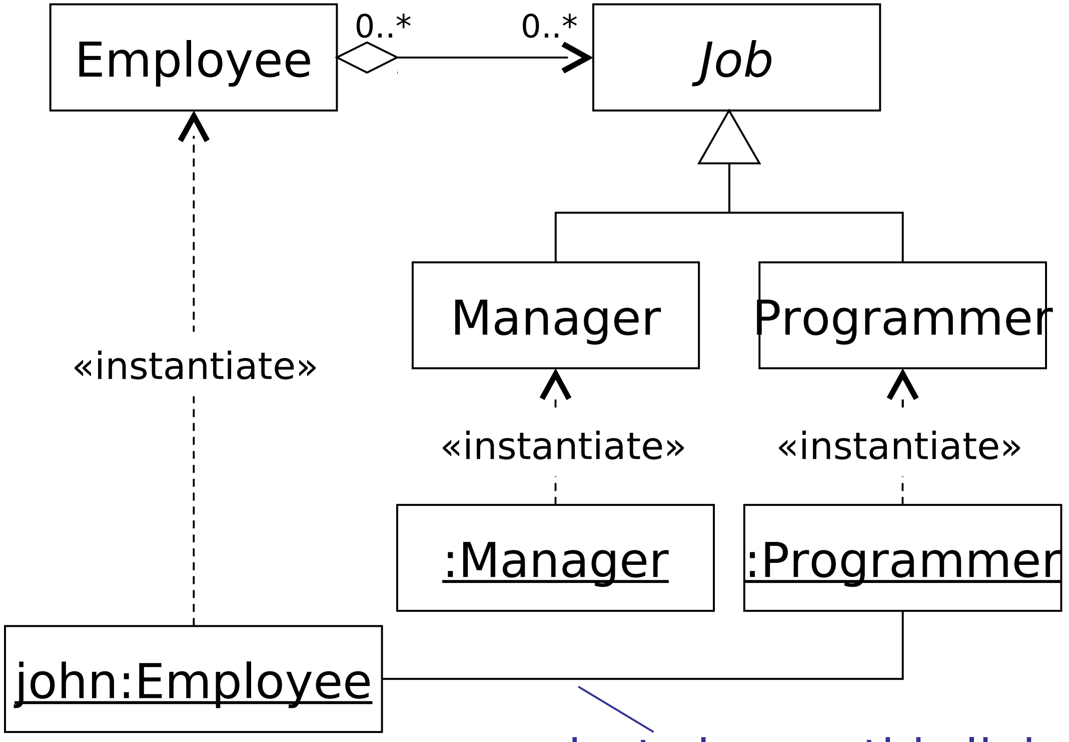
- Inheritance gives you fixed relationships between classes and objects
- You *can't* change the class of an object at runtime
- There is a fundamental semantic error here. Is an Employee *just* their job or does an Employee *have* a job?



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

A better solution…

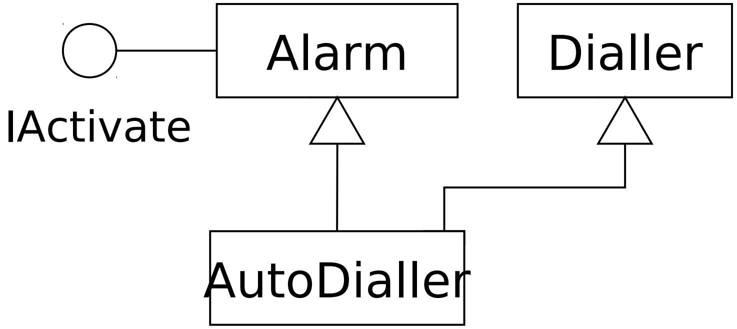
- Using aggregation we get the correct semantics:
  - An *Employee has* a Job
- With this more flexible model, Employees can have more than one Job



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

17.6.3

Inheritance vs. interface realization

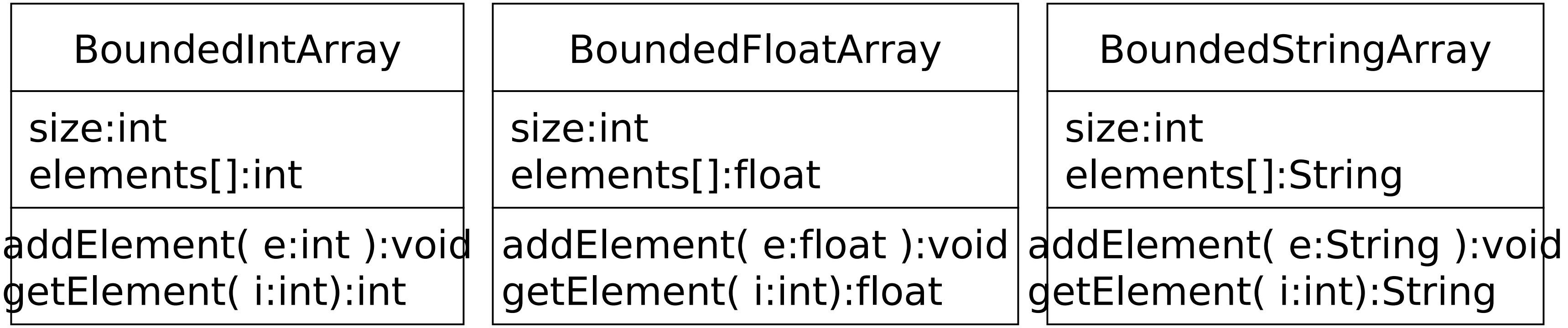
- With inheritance we get two things:
  - Interface – the public operations of the base classes
  - Implementation – the attributes, relationships, protected and private operations of the base classes
- With interface realization we get exactly one thing:
  - An interface – a set of public operations, 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

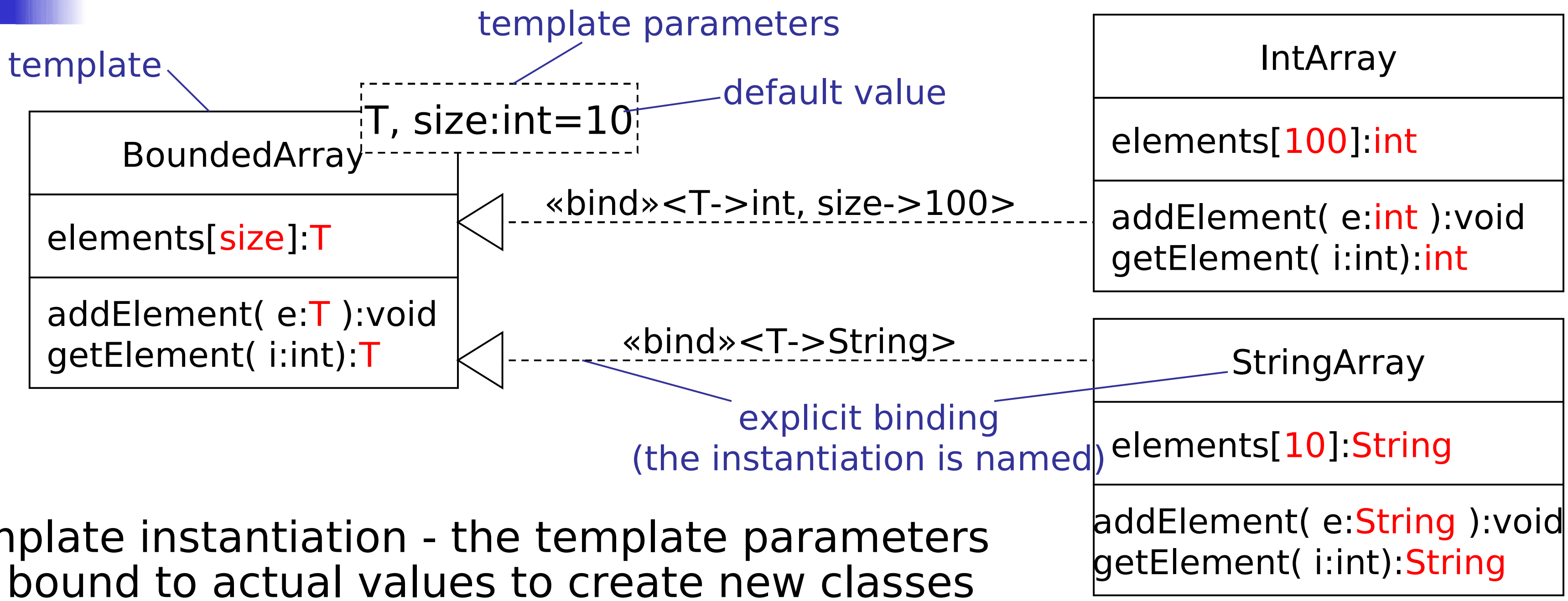
BoundedStringArray

size:int
elements[]:String

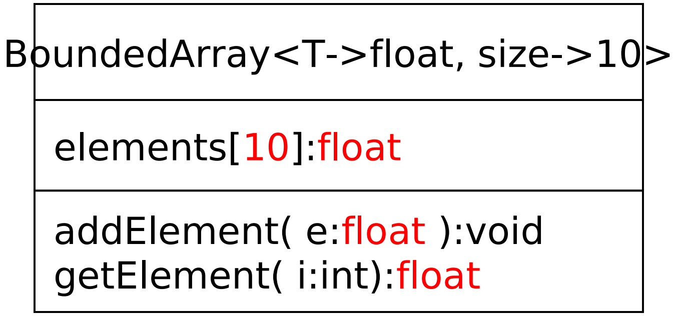
addElement(e:String):void
getElement(i:int):String

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



implicit binding
(the instantiation is anonymous)

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

Summary

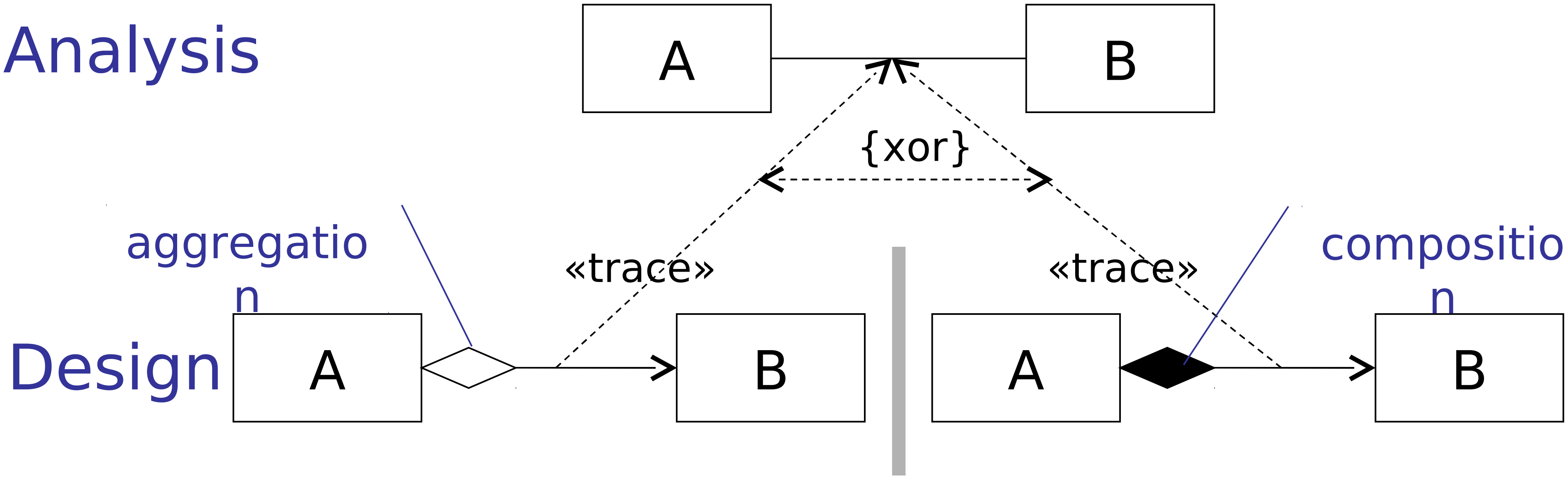
- Design classes come from:
  - A refinement of analysis classes (i.e. the business domain)
  - From the solution domain
- Design classes must be well-formed:
  - Complete and sufficient
  - Primitive operations
  - High cohesion
  - Low coupling
- Don’t overuse inheritance
  - Use inheritance for "is 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

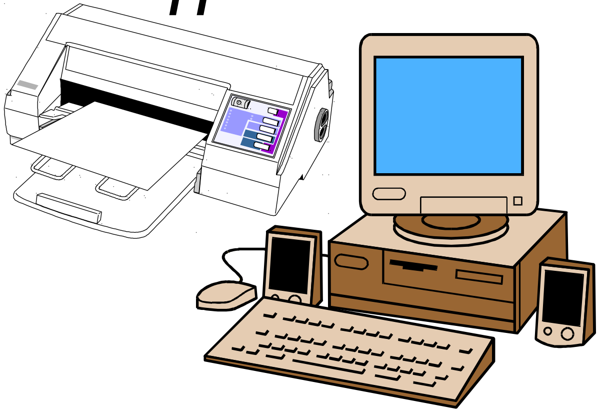


- In analysis, we often use unrefined associations. In design, these can become aggregation or composition relationships
- We must also add navigability, multiplicity and role names

Aggregation and composition

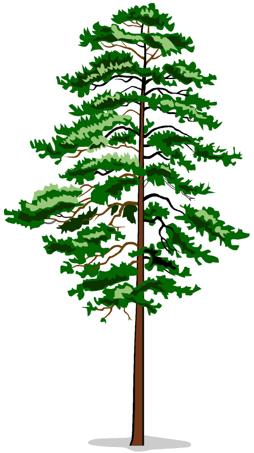
UML defines two types of association:

Aggregation



Some objects are weakly
related like a computer
and

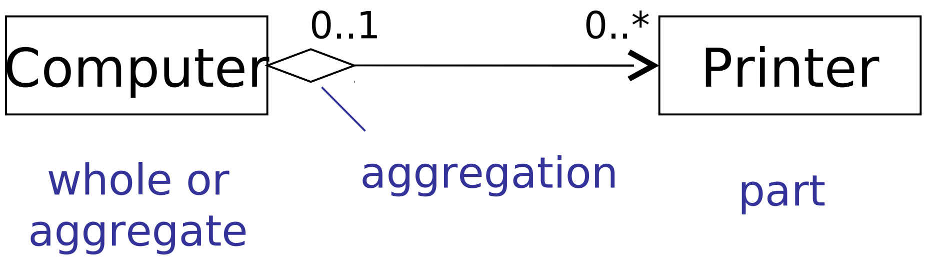
Composition



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

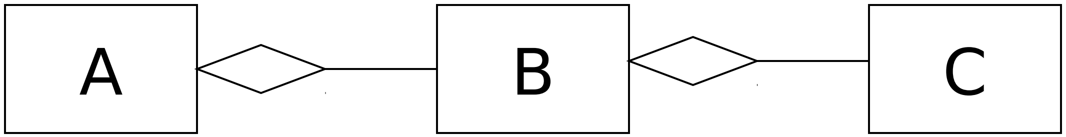
Over time, many Computers may use a given Printer

The Printer exists even if there are no Computers

The Printer is independent of the Computer

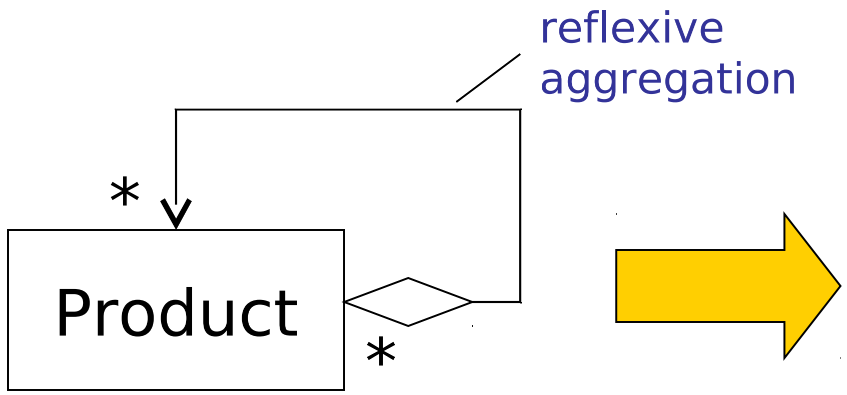
- The aggregate can sometimes exist independently of the parts, sometimes not
- The parts can exist independently of the aggregate
- The aggregate is in some way incomplete if some of the parts are missing
- It is possible to have shared ownership of the parts by several aggregates

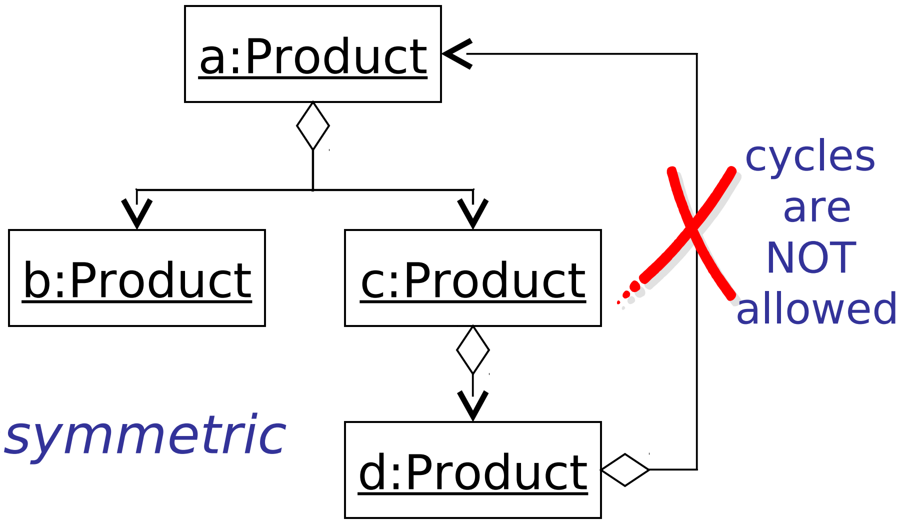
Transitive and asymmetric



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

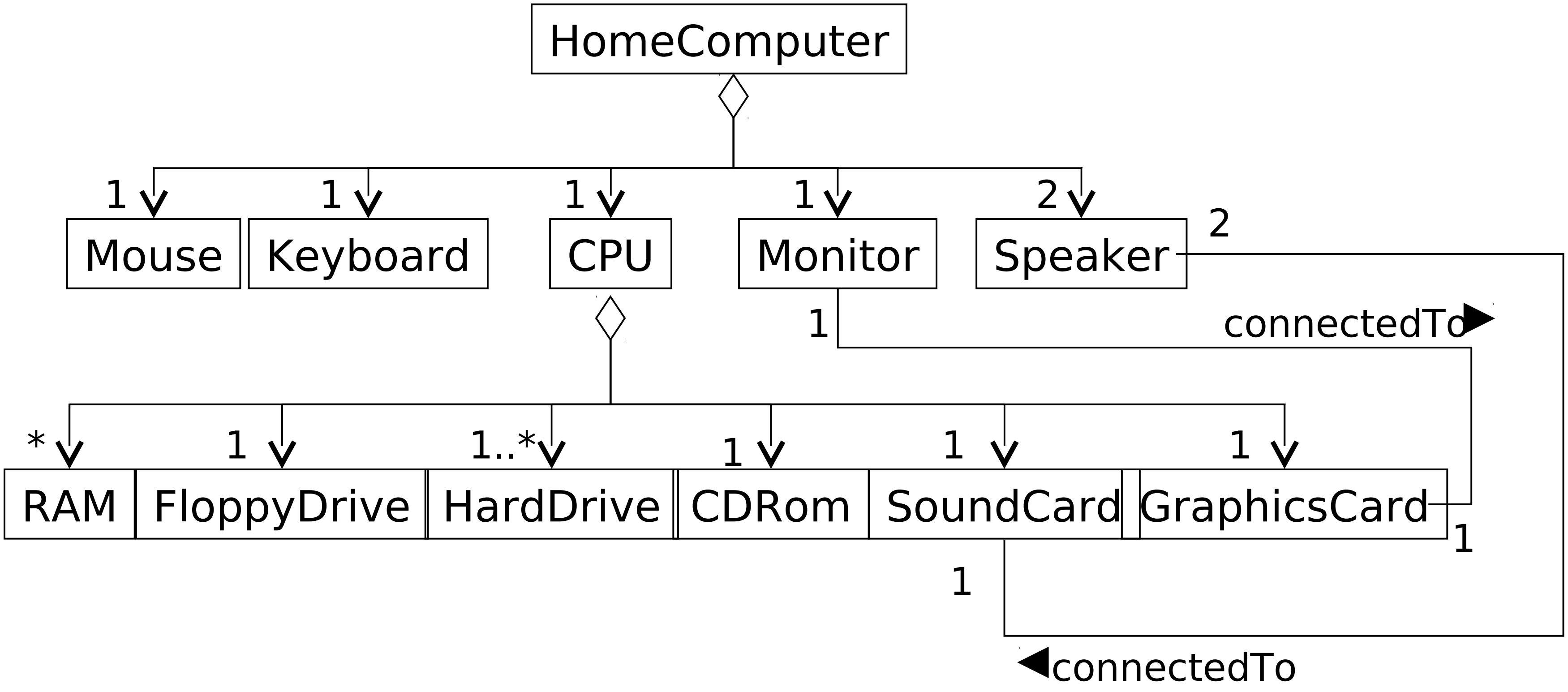




Aggregation (and composition) are *asymmetric*

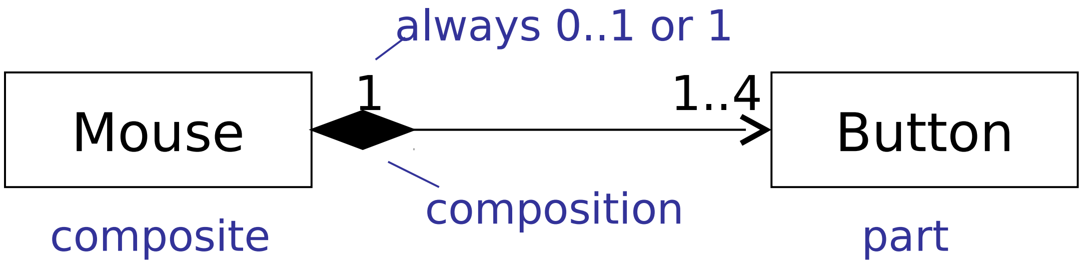
An object can *never* be part of itself!

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

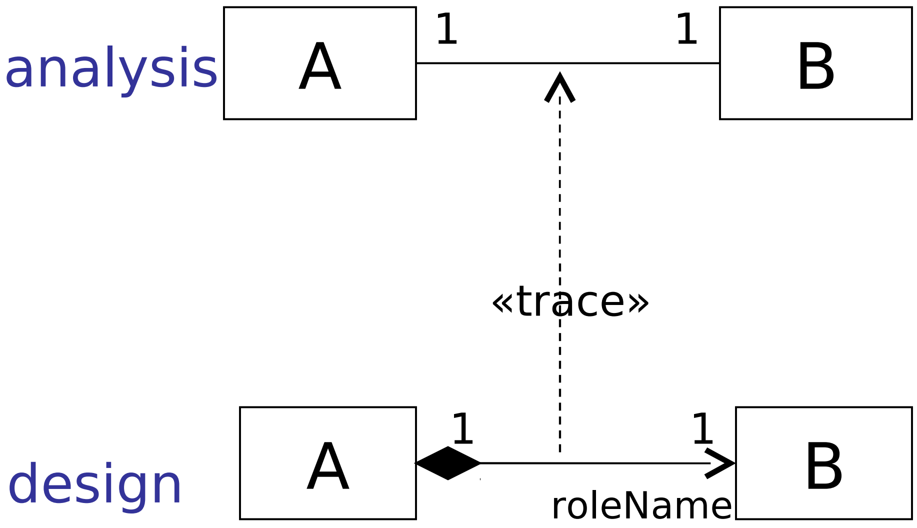
18.5.1

Composition and attributes

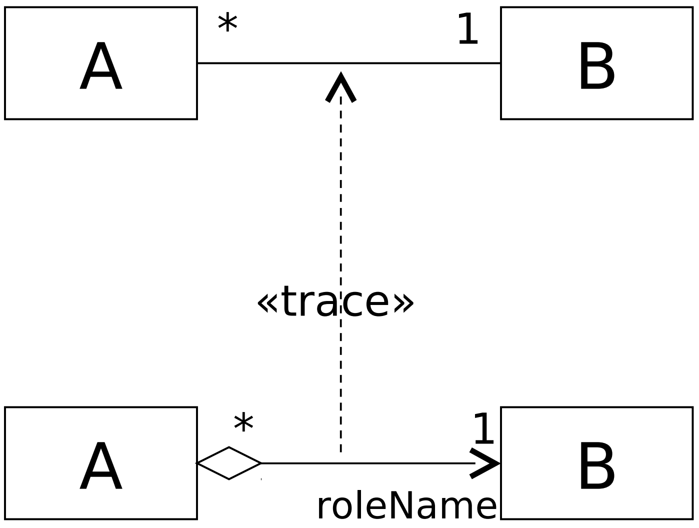
- Attributes are in effect composition relationships between a class and the classes of its attributes
- Attributes should be reserved for primitive data types (int, String, Date etc.) and **not** references to other classes

1 to 1 and many to 1 associations

1 to 1



many to 1

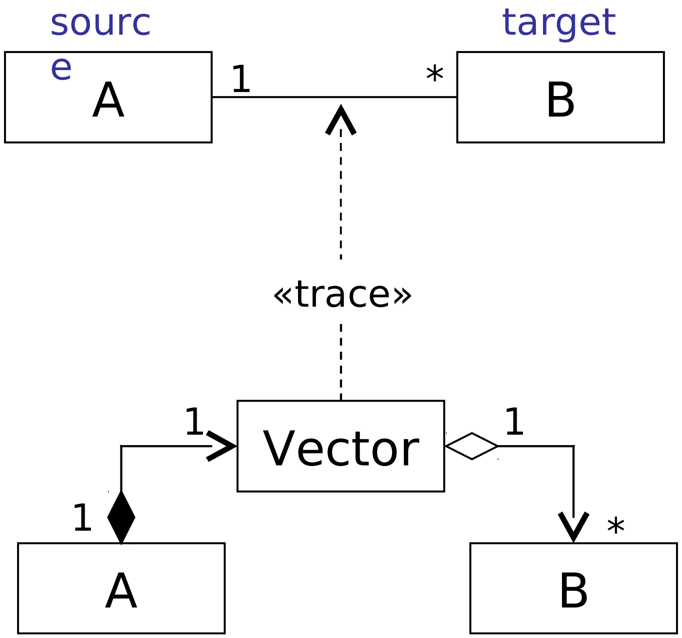


- One-to-one associations in analysis *usually* imply single ownership and *usually* refine to compositions

- 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

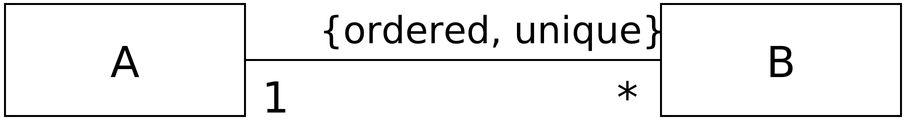


Collection semantics

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

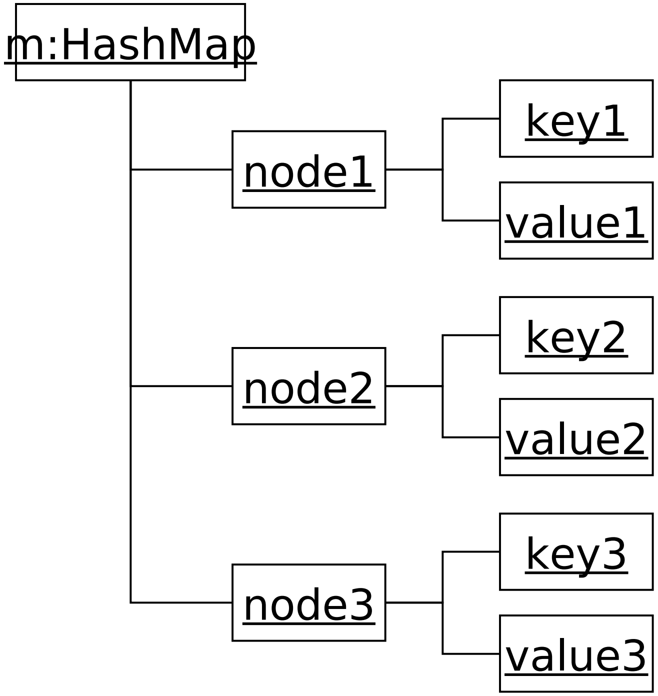
property	semantics
{ordered}	Elements in the collection are maintained in a strict order
{unordered}	There is no ordering of the elements in the collection
{unique}	Elements in the collection are all unique an object appears in the collection once
{nonunique}	Duplicate elements are allowed in the collection

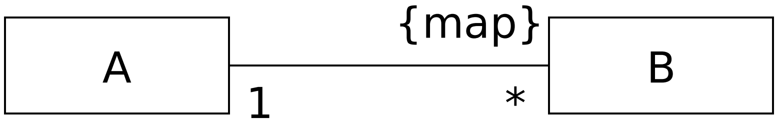
property pair	OCL collection
{unordered, nonunique}	Bag
{unordered, unique}	Set (default)
{ordered, unique}	OrderedSet
{ordered, nonunique}	Sequence



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

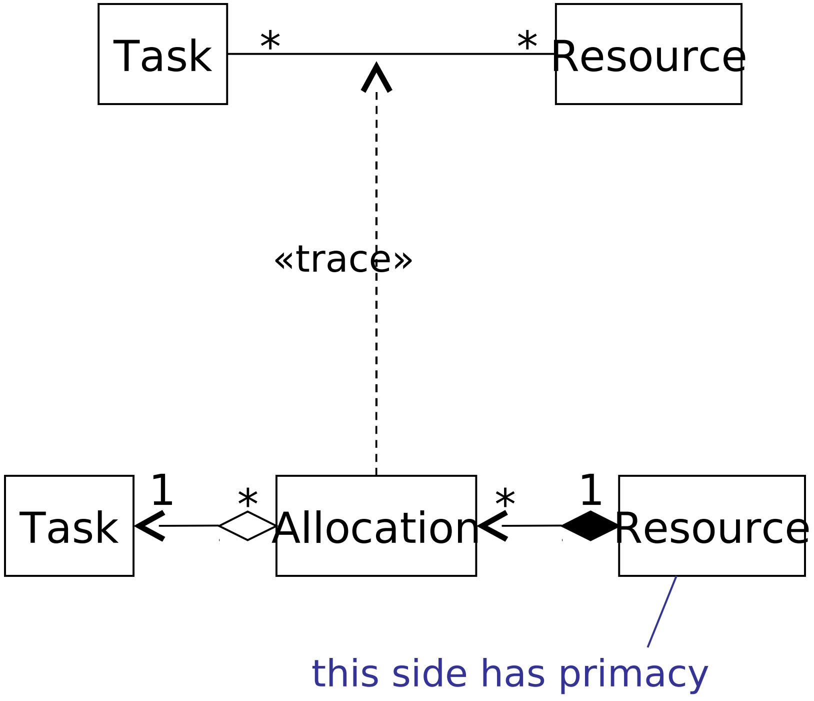




you can indicate the type of collection using a constraint

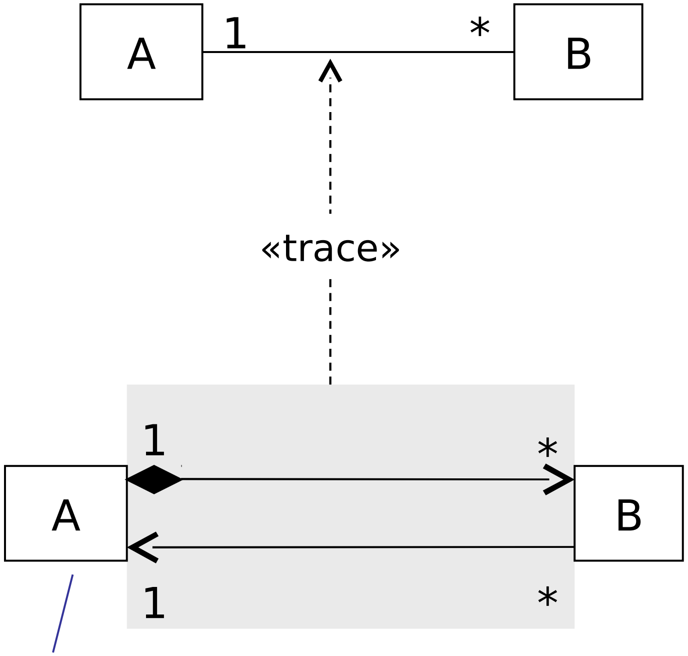
Many to many associations

- 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

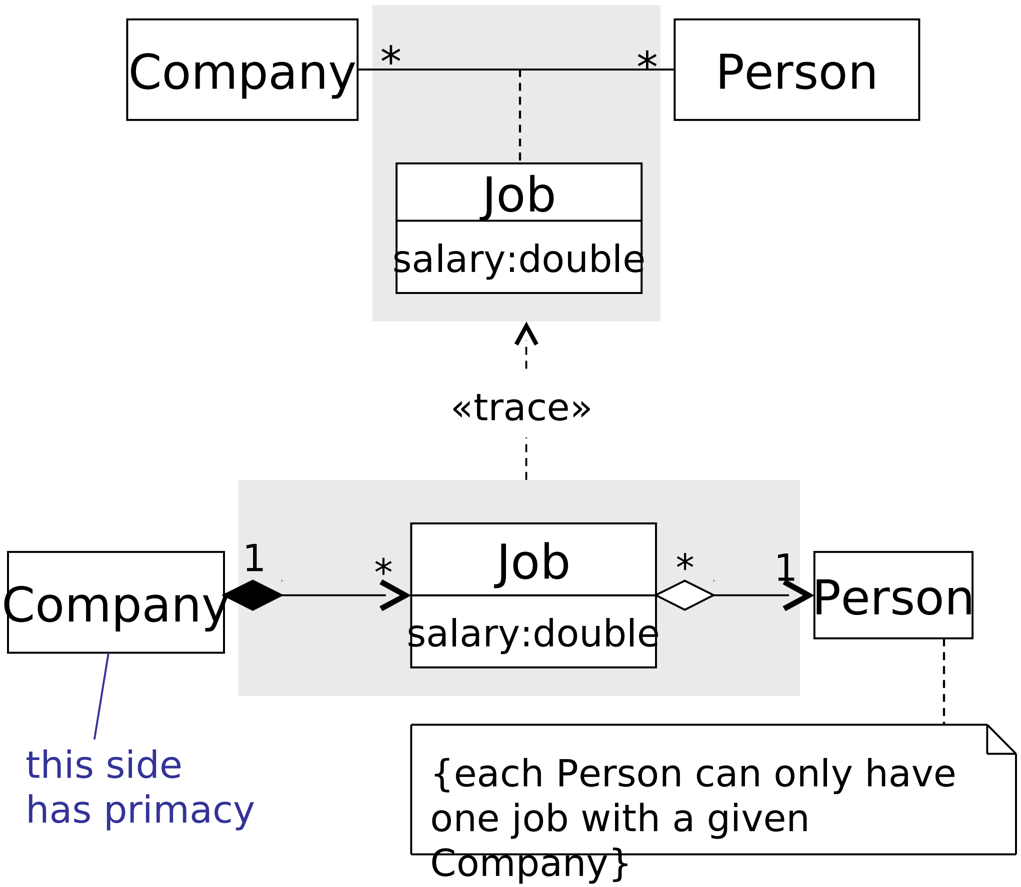
- There is no commonly used OO language that directly supports bi-directional associations
- We must resolve each bi-directional associations into two unidirectional associations
- Again, we must decide which side of the association should have primacy and use composition, aggregation and navigability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