Lecture 3
Object-Oriented Design (Part 2)

Dr Peter T. Popov
Centre for Software Reliability

11th October 2018

Recap from Lecture 2

- Last week we started with software design
  - We looked at the goals of high/low level design
  - We looked at Design classes
  - We introduced the concept of interfaces
- Today we will continue with the ***low level*** software design:
  - Will look at specifications of design classes
    - We will also demonstrate the use of activity diagram to capture the logic of class methods
  - Will look at how class relationships (associations) are refined in design
- Will look at how sequence diagrams in design are refined and will build fragments of software designs:
  - How interfaces get used in sequence diagrams
  - Some advanced techniques such as modelling concurrency and mutex (mutual exclusion)

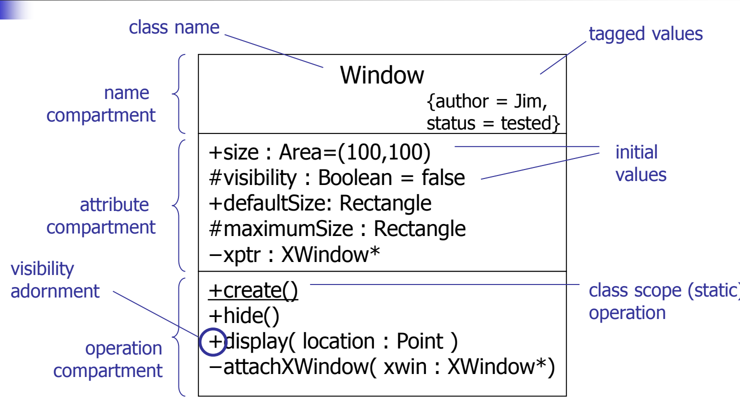
2

UML class notation

3

7.5

UML class notation



- Classes are named in UpperCamelCase
- Use descriptive names that are nouns or noun phrases
- Avoid abbreviations!

 4

7.5.2

Attribute compartment

visibility name : type multiplicity = initialValue

mandatory

- Everything is optional except name
- initialValue is the value the attribute gets when objects of the class are instantiated
- Attributes are named in lowerCamelCase
  - Use descriptive names that are nouns or noun phrases
  - Avoid abbreviations
- Attributes may be prefixed with a stereotype and postfixed with a list of tagged values

7.5.2.1

Visibility

Symbol	Name	Semantics
+	public	Any element that can access the class can access any of its features with public visibility
–	private	Only operations within the class can access features with private visibility
#	protected	Only operations within the class, or within children of the class, can access features with protected visibility
~	package	Any element that is in the same package as the class, or in a nested sub-package, can access any of its features with package visibility

PersonDetails

–name : String [2..*]
–address : String [3]
–emailAddress : String [0..1]

- You may ignore visibility in analysis
- In design, attributes usually have private visibility (encapsulation)

7.5.2.3

Multiplicity

- Multiplicity allows you to model collections of things
  - [0..1] means that the attribute may have the value null

PersonDetails
−name : String [2..*] −address : String [3] −emailAddress : String [0..1]

name is composed of 2 or more Strings
address is composed of 3 Strings
emailAddress is composed of 1 String or null

multiplicity expression

© Clear View Training 2010 v2.6 7

7.5.3

Operation compartment

operation signature

visibility name(direction parameterName: parameterType = default, …) : returnType

parameter list

there may be a comma delimited list of return types - r1, r2,… rn

- Operations are named lowerCamelCase
  - Special symbols and abbreviations are avoided
  - Operation names are usually a verb or verb phrase
- Operations may have ***more than one*** returnType
  - They can return multiple objects (see next slide)
- Operations may be prefixed with a stereotype and postfixed with a list of tagged values

© Clear View Training 2010 v2.6 8

7.5.3.1

Parameter direction

use in detailed design only!

parameter direction	semantics
in	the parameter is an input to the operation. It is not changed by the operation. This is the default
out	the parameter serves as a repository for output from the operation
inout	the parameter is an input to the operation and it may be changed by the operation
return	the parameter is one of the return values of the operation. An alternative way of specifying return values

example of multiple return values:

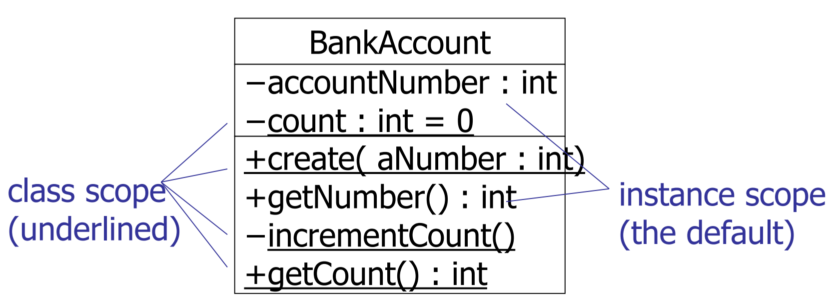
```
maxMin( in a: int, in b:int, return maxValue:int return minValue:int )
…
max, min = maxMin( 5, 10 )
```

 9

7.6

Scope

- There are two kinds of scope for attributes and operations:



  - Class scope in Java and C++ is implemented using the keyword ***static*** for both attributes and operations/methods.
  - Operations with class scope can only operate on attributes with class scope.

 10

7.6.1

Instance scope vs. class scope

	instance scope	class scope
attributes	By default, attributes have instance scope	Attributes may be defined as class scope
attributes	Every object of the class gets its own copy of the instance scope attributes	Every object of the class shares the same, single copy of the class scope attributes
attributes	Each object may therefore have different instance scope attribute values	Each object will therefore have the same class scope attribute values
operations	By default, operations have instance scope	Operations may be defined as class scope
operations	Every invocation of an instance scope operation applies to a specific instance of the class	Invocation of a class scope operation does not apply to any specific instance of the class – instead, you can think of class scope operations as applying to the class itself
operations	You can't invoke an instance scope operation unless you have an instance of the class available. You can't use an instance scope operation of a class to create objects of that class, as you could never create the first object	You can invoke a class scope operation even if there is no instance of the class available – this is ideal for object creation operations

scope determines access

 11

7.7

Object construction

- How do we create instances of classes?
- Each class defines one or more class scope operations which are ***constructors***. These operations create new instances of the class

BankAccount
+create(aNumber : int)

generic constructor name

BankAccount
+BankAccount(aNumber : int)

Java/C++ standard

 12

7.7.1

ClubMember class example

- Each ClubMember object has its own copy of the attribute membershipNumber
- The numberOfMembers attribute exists only once and is shared by all instances of the ClubMember class
- Suppose that in the create operation we increment numberOfMembers:
  - What is the value of count when we have created 3 account objects?

ClubMember
−membershipNumber : String −memberName : String −<u>numberOfMembers : int = 0</u>
<u>+create(number : String, name : String)</u> +getMembershipNumber() : String +getMemberName() : String <u>−incrementNumberOfMembers()</u> <u>+decrementNumberOfMembers()</u> <u>+getNumberOfMembers() : int</u>

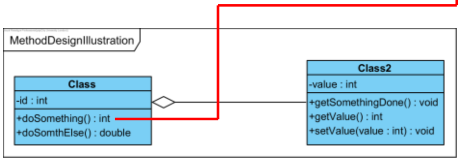
Modelling Class methods

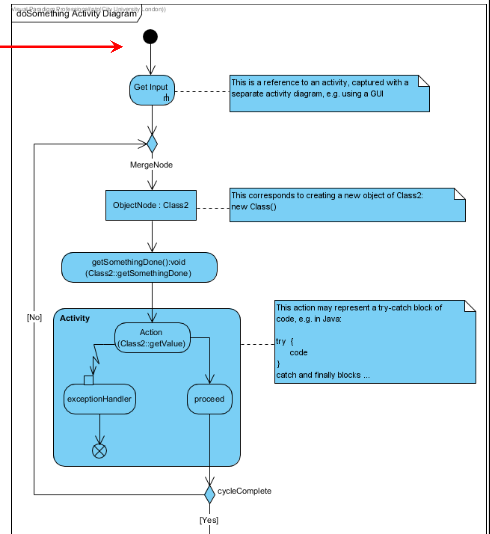
- The logic of the class methods can be captured using activity diagrams.
- Activity diagrams provide a rich set of features:
  - Capture a flow of actions of arbitrary complexity including loops, if-else constructs, etc.
  - Object instantiation/destruction can be captured, too, using the "object flow"
  - Exceptions and exception handlers can be defined as necessary.
- The decision, which methods are worth modelling with an activity diagram should be pragmatic
  - The activity diagram should be only used if it adds value.
  - Activity diagram should only be used if the designer wishes to communicate a specific logic to the programmers.

Activity diagram to model a method

- A fragment of a class diagram is shown.
- The method doSomething() of Class is modelled with an Activity diagram
  - This diagram must be "linked" to the method it models: done as a "sub-diagram" of the method.

ModellingClassMethodWithActivityDiagram.vpp provides further details.





15

Class refinement with Visual Paradigm

- Visual Paradigm allows us to maintain in the same project different models:
  - Analysis model – dealing with problem domain
  - Design model – providing detailed design specifications to be used in implementation
  - Implementation model, e.g. dealing with sources code
- These models define different *namespaces*
  - Copying diagrams from Analysis to Design model is OK
    - Modifications made to diagrams (e.g. class diagrams) in Design model *do not affect* the diagrams in the Analysis model.

In summary: When we move from Analysis to Design we must define a Design model and thus keep both Analysis and Design in the same .vpp project.

16

Design - refining analysis relationships

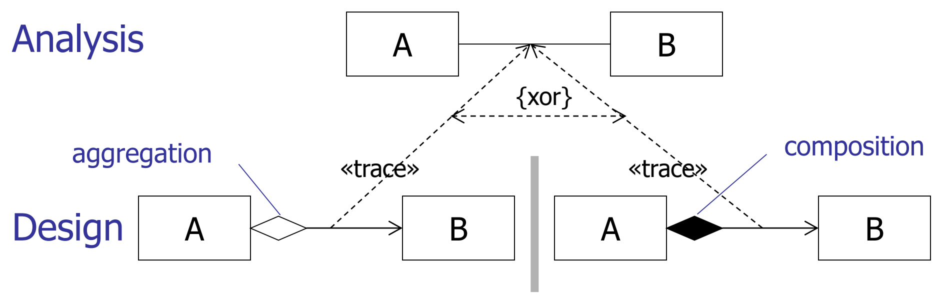
18.2

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

18.3

Aggregation and composition



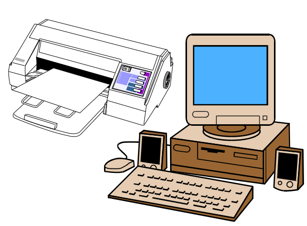
- In analysis, we often use unrefined associations. In design, these ***can*** become aggregation or composition relationships
- We must also add navigability, multiplicity and role names

18.3

Aggregation and composition

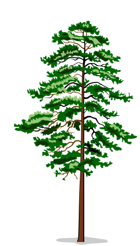
UML defines two types of association:

Aggregation



Some objects are weakly related like a computer and its peripherals

Composition

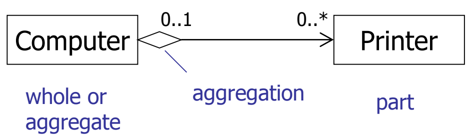


Some objects are strongly related like a tree and its leaves

18.4

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

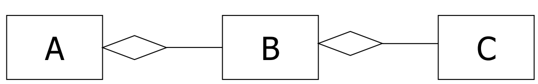
The Printer exists even if there are no Computers

The Printer is independent of the Computer

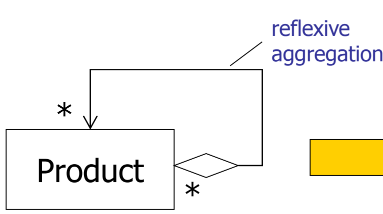
- The aggregate can sometimes exist independently of the parts, sometimes not
- The parts ***can*** exist independently of the aggregate
- The aggregate is in some way incomplete if some of the parts are missing
- It is possible to have ***shared ownership*** of the parts by several aggregates

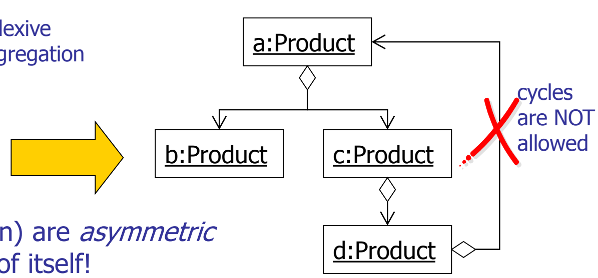
18.4

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

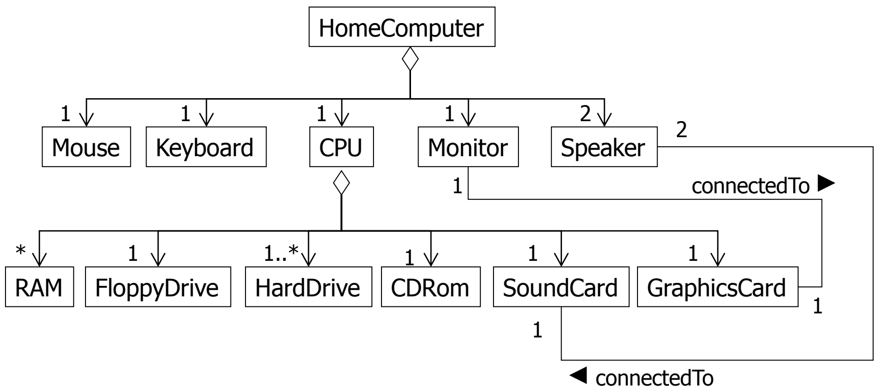




Aggregation (and composition) are *asymmetric*
An object can *never* be part of itself!

18.4

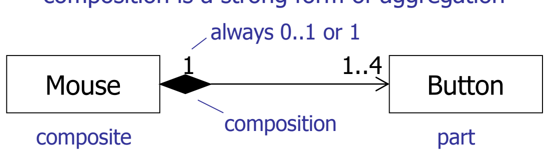
Aggregation hierarchy



18.5

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

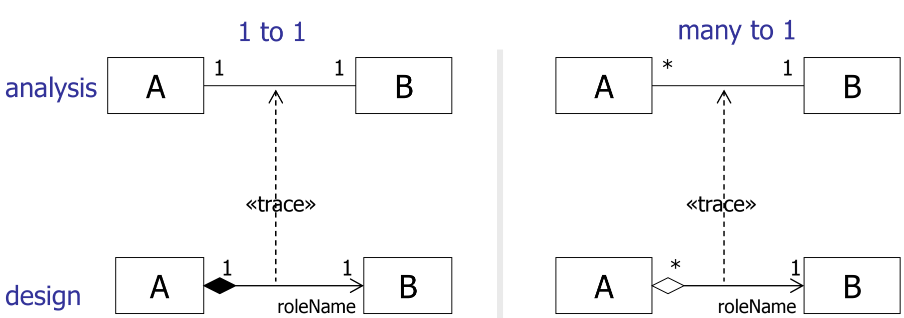
18.5.1

Composition and attributes

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

18.7 18.8

1 to 1 and many to 1 associations

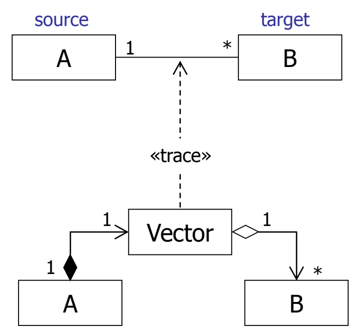


- One-to-one associations in analysis *usually* imply single ownership and *usually* refine to compositions
- Many-to-one relationships in analysis imply shared ownership and are refined to aggregations

18.9

1 to many associations

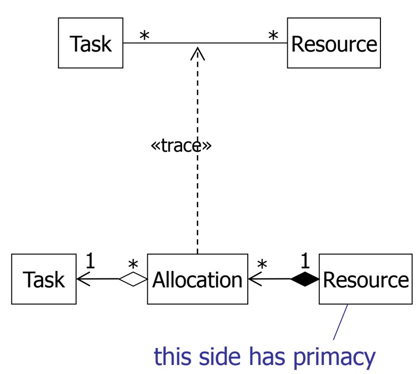
- To refine 1-to-many associations we introduce a *collection class*
- Collection classes instances store a collection of object references to objects of the target class
- A collection class always has methods for:
  - Adding an object to the collection
  - Removing an object from the collection
  - 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



18.11.1

Many to many associations

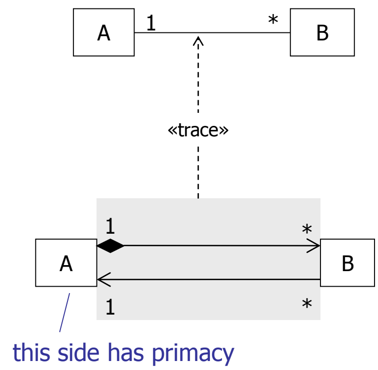
- There is no commonly used OO language that directly supports many-to-many associations
- We must ***reify*** such associations into design classes
- Again, we must decide which side of the association should have ***primacy*** and use composition, aggregation and navigability accordingly



18.11.2

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

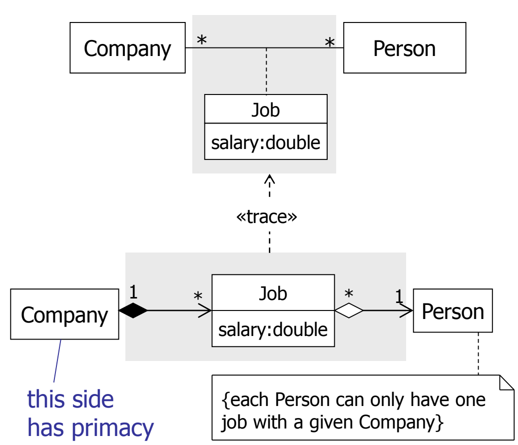


© Clear View Training 2010 v2.6 29

18.11.3

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



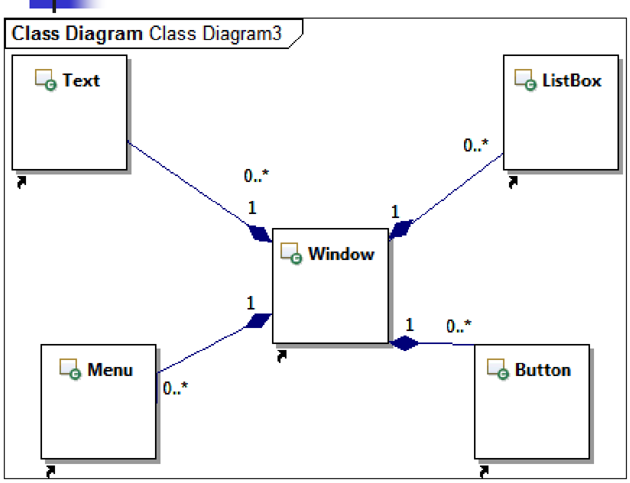
© Clear View Training 2010 v2.6 30

Classes from *other* domains

- Graphic User Interface (GUI)
  - Rarely useful to start with a UML model. IDE Tools provide extensive support to build quickly elaborate GUI without any coding (code is generated by tools).
- Database domain
- Communication domain
- Working with 3rd party (off-the-shelf) software
- Design patterns (make designs efficient and "elegant")
  - Will cover this topic in a separate lecture

31

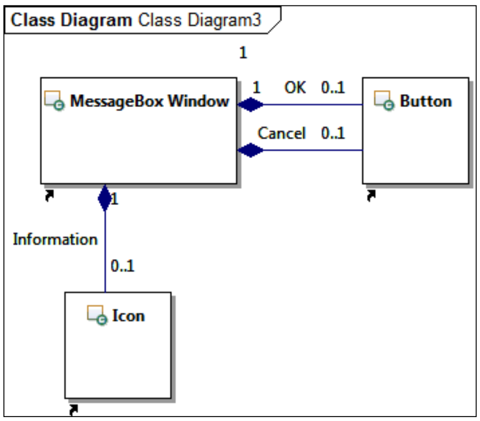
Generic GUI Model

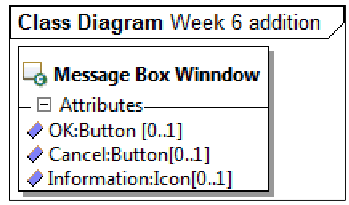


This is a typical model of GUI: the main window (a form) will consist of many visual controls such as Text boxes, List boxes, Buttons, Menus and possibly other visual controls.
This is a general model, but can be made more specific, e.g. by specifying for each Window the *exact* number of buttons, the exact number of boxes, etc.

32

Model of a specific GUI form



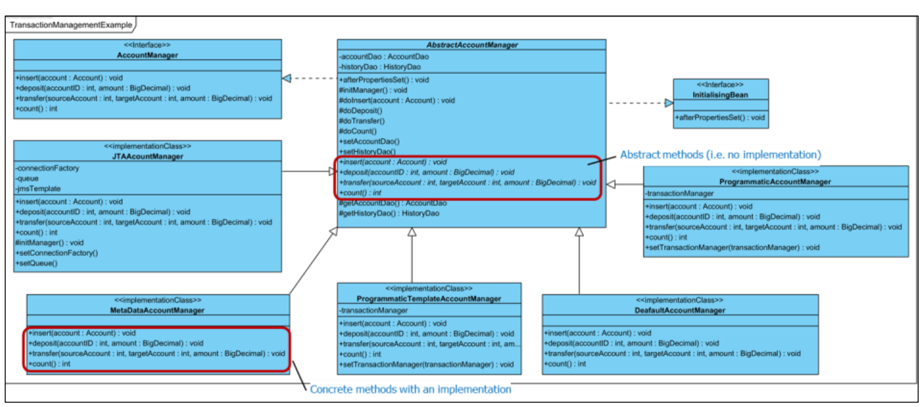


The two fragments are semantically identical: model a MessageBox

33

Relationships between classes and interfaces

- A class may have a "realisation" relationship with an interface, when the class provides an implementation of the methods defined in the interface
  - The same class may implement multiple interfaces
  - Multiple classes may provide an implementation of the same interface. These classes can be used interchangeably to provide implementation of the methods defined in the particular interface.



34

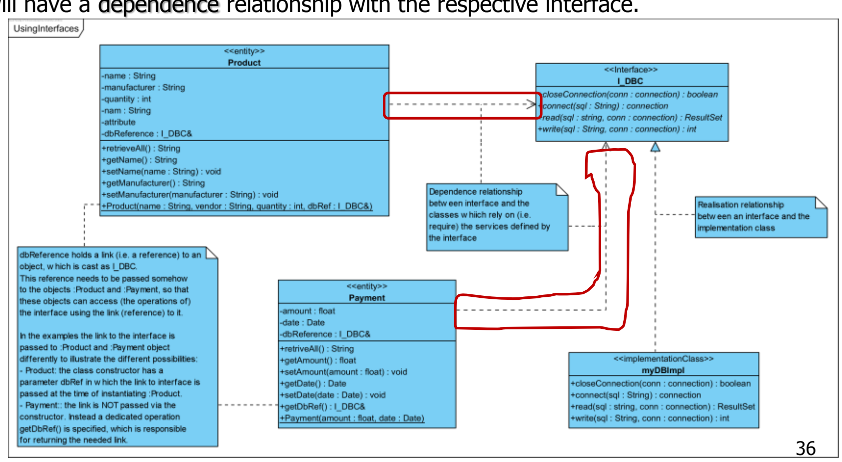
Associations between classes and interfaces

- A class cannot have a normal association with an interface as it cannot be instantiated
  - associations are meant to represent in class diagrams *links between the objects*
    - You may come across (online, in books, event UML tools), in which interfaces are associated with classes (including aggregation/composition). This is simply wrong! The UML standard is very clear about this aspect.
  - An interface cannot be instantiated. It is just a set of abstract methods!
- A class, which *uses* an interface (i.e. invokes the methods defined in an interface) will have a **dependence** relationship with the respective interface.

35

Associations between classes and interfaces

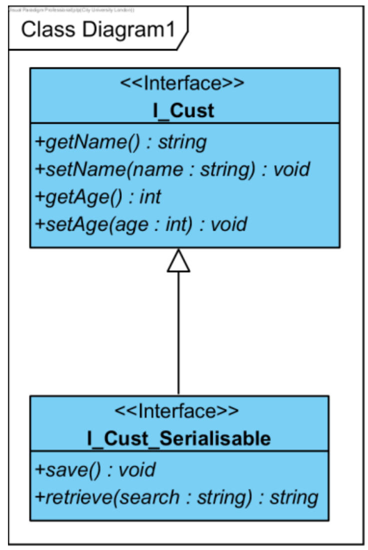
- A class cannot have a normal association with an interface as it cannot be instantiated.
- A class, which *uses* an interface (i.e. invokes the methods defined in an interface) will have a **dependence** relationship with the respective interface.



36

Associations between interfaces

- Inheritances may have *inheritance* relationship.
- the notation used to model inheritance between interfaces is the same as for classes.
- The semantic of the relationship is the same as for classes – substitutability principle applies.



37

18.13

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
- Interfaces may have relationships with classes (realisation and dependence) and with other interfaces (inheritance)

Association refinement are covered in Chapter 18. The examples, however, are not to be found in the book. Check out the notes on Moodle that I released with them.

 38

Design - use case realisation

20.3

Use case realisation - design

- A collaboration of Design objects and classes that realise a use case
- A Design use case realisation contains
  - Design object interaction diagrams
  - Links to class diagrams containing the participating Design classes
  - Interfaces can also be used
  - An explanatory text (flow)

same as in Analysis, but now including ***implementation details***

- There is a trace between an Analysis use case realisation and a Design use case realisation
- The Design use case realisation specifies implementation decisions and implements the non-functional requirements

20.4

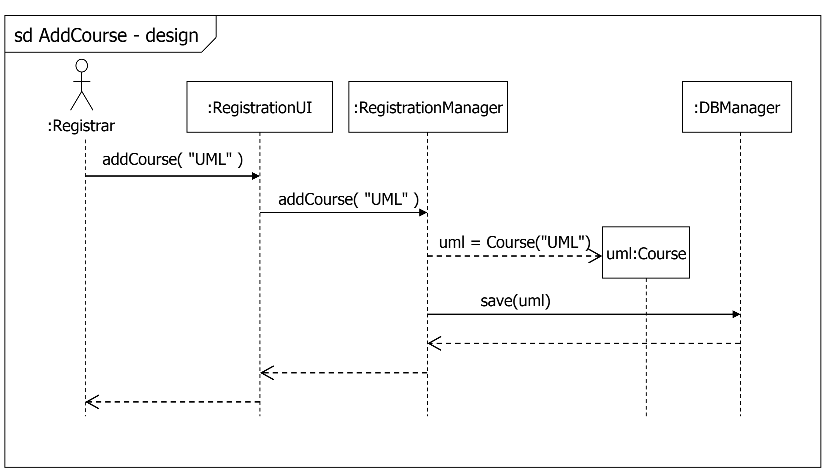
Interaction diagrams in design

- 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***
    - E.g. one may illustrate various *concurrency control* related issues such as mutex between different threads that share a resource
- In design:
  - Sequence diagrams are used more than communication diagrams
  - Timing diagrams may be used to capture timing constraints, e.g. essential for real-time systems.

 41

20.4

Sequence diagrams in design



 42

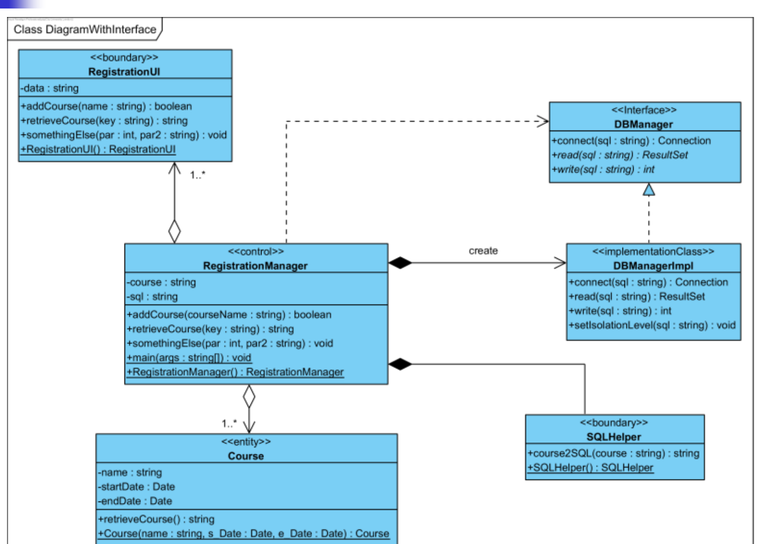
A sequence diagram using an interface

- Let us elaborate on this diagram
  - First provide a sketch of a class diagram in which an interface defines a contract for DB connectivity
  - The sequence diagram is then built to use the interface, which defines DB connectivity.

Further details are provided in VisualParadigm project: DesignSequenceDiagramWithInterfaces.vpp

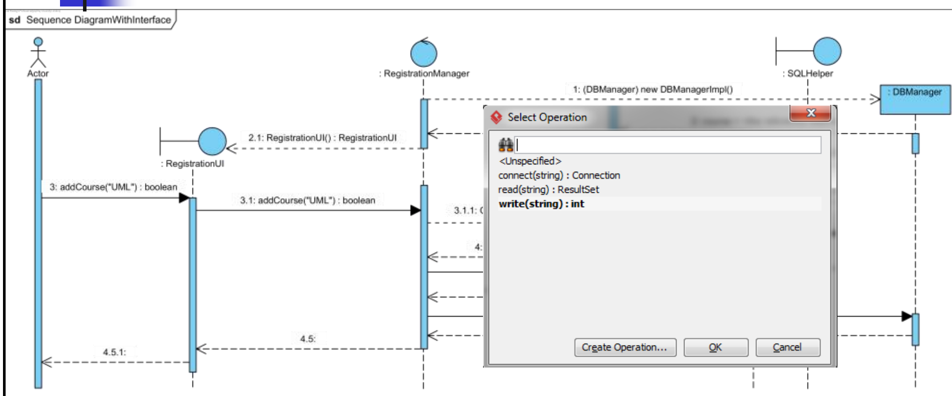
43

The class diagram



44

The sequence diagram



45

Concurrency modelling

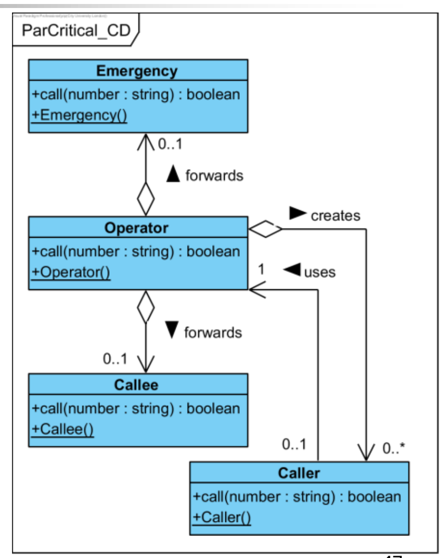
- Concurrency can be modelled with different diagrams
  - Sequence diagram (will be illustrated now)
    - modelled with **par** combined fragment
    - Mutex (mutual exclusion) modelled with **critical** region combined fragment
  - Activity diagram
    - check the UML standard for examples
  - State-machine diagrams
    - will be covered in the next lecture

46

Example: Model of Concurrency in SD

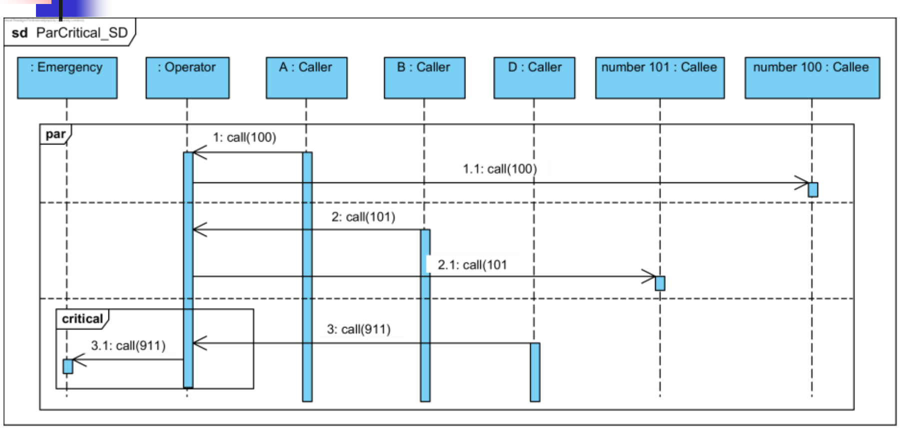
- Consider the following small fragment of a class diagram. It models concurrent phone calls made by a number of Callers to different Callees via an Operator.
  - Each Callee (instance) has a unique number
  - All calls are made asynchronously and truly concurrently (i.e. no need for any synchronisation whatsoever among them)
    - Here we assume that if two different Callers call the same Callee, the Callee can connect to all simultaneously.
  - Calls made to emergency number 911, however, must be "serialised", i.e. at most one call at a time can be connected.

© OMG, UML v2.5.1, p. 629



47

Example: Model of Concurrency in SD (2)



- **par** combined fragment signifies that all calls start in parallel. We illustrate with 3 calls, but in practice can have a much larger number of Callers messaging the :Operator.
- Multiple invocations of :Operator lifeline by Callers (A, B and C) is allowed. The Operator class must be designed accordingly, so that it can handle overlap calls (e.g. the class Operator may start a separate thread to establish a connection with each of the Callers)
- Caller D in the example is calling the emergency number. The :Operator must serialise the messages to the :Emergency lifeline, which is modelled with a **critical** combined fragment.

48

"Dangers" with concurrency

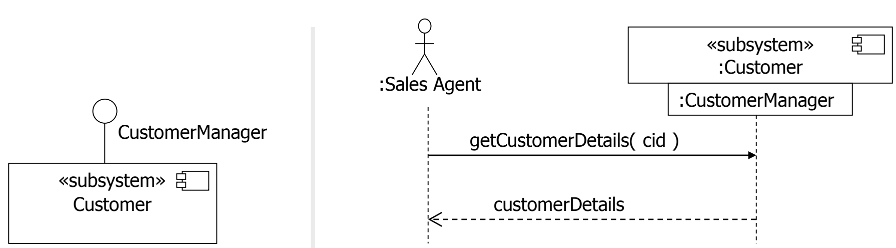
- Implementation of concurrency controlling mechanisms is program language specific.
  - For Java this is achieved via:
    - Threats (or implementation of Runnable interface)
    - "Synchronised" methods (or synchronised blocks).
  - You may read more on the topic at:
    - https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html
- Often concurrency may have "unexpected" consequences
  - With experience we learn to expect more
- Typical problems with concurrency
  - Performance may not improve (or event deteriorate)
  - "Liveliness" may not be guaranteed
    - Fairness of access to shared resource may need to be enforced.

49

20.6

Subsystem interactions

- Sometimes it's useful to model a use case realisation 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



 50

Timing diagrams

- They are necessary when modelling real time systems
  - May be necessary when there are explicit performance related non-functional requirements
  - Jim's book offers a minimalistic coverage of the topic (section 20.7)

51

Summary

- We have looked briefly at
  - Sequence diagrams is design
    - Adding details to the messages exchanged
    - We illustrated also some advanced concepts such as modelling concurrency and mutual exclusion, common in concurrent and distributed programming
  - Subsystem interactions
- Most of the material is covered in Chapter 20 of Arlow's book.
  - Section 20.8 of Jim Arlow's provides an example of a complete use case realisation at the design stage.
  - The VP examples (using interfaces and concurrency), however, are not included in the book.

52