Lecture 2
Analysis Recap and Packages
System Design - Introduction

Dr Peter T. Popov
Centre for Software Reliability

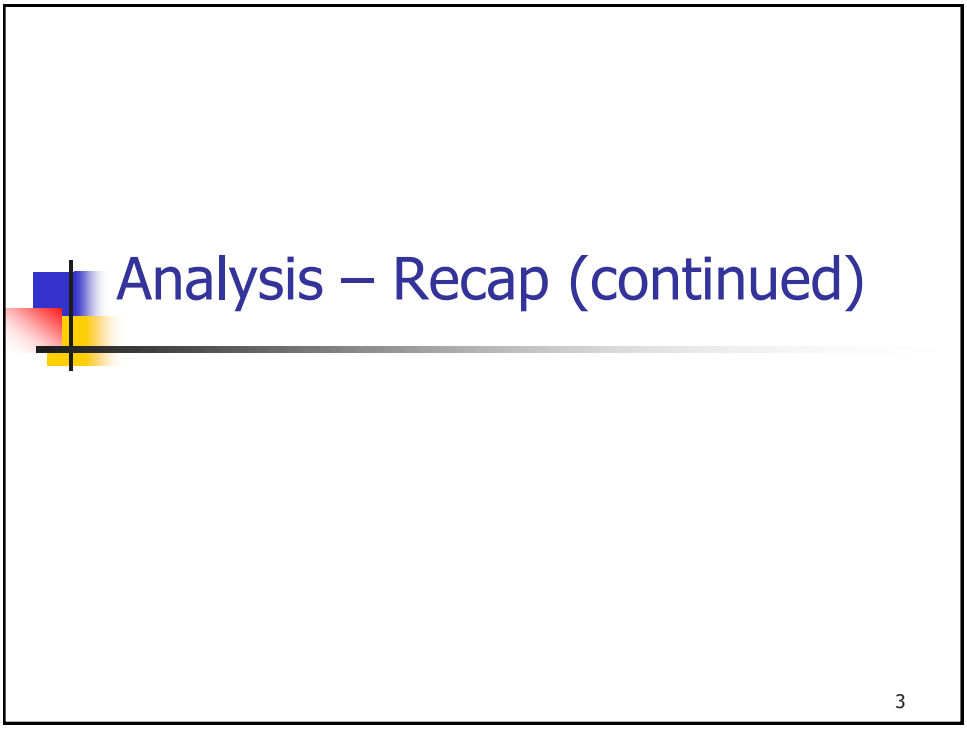
4th October 2018

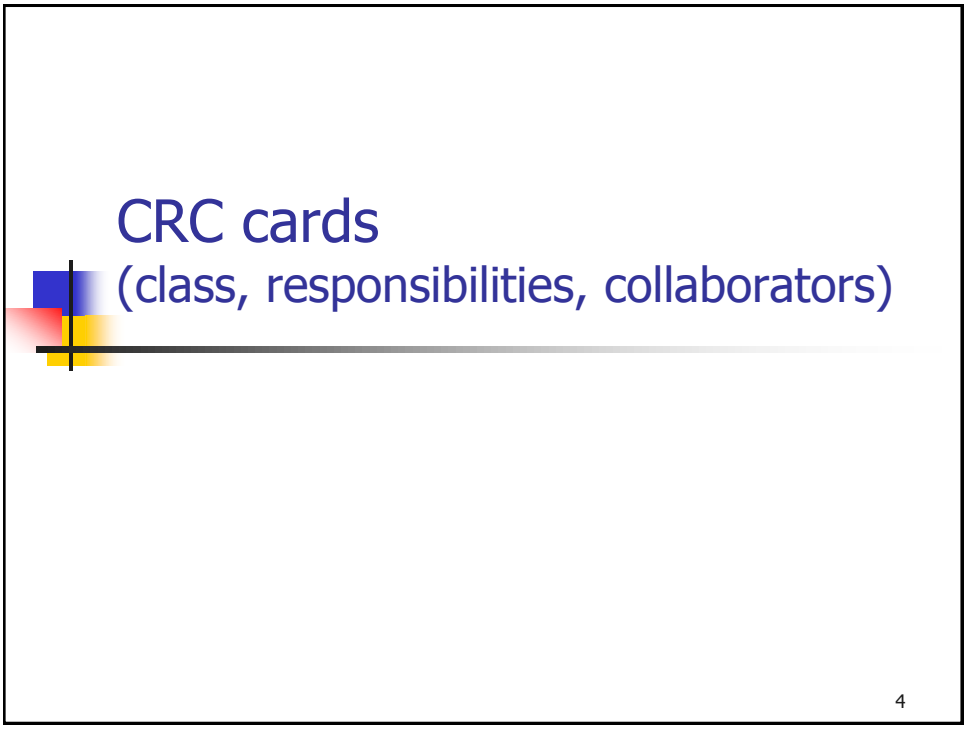
9.2

Plan for today

- Complete with OO analysis
  - CRC cards
  - Introduce UML packages
- Make a start with OO design

2





An extended CRC Card

Front:

Class Name: Patient	ID: 3	Type: Concrete, Domain
Description: An Individual that needs to receive or has received medical attention		Associated Use Cases: 2

Responsibilities	Collaborators
Make appointment	Appointment
Calculate last visit	
Change status	
Provide medical history	Medical history

Source: Alan Dennis, Barbara Haley Wixom, David Tegarden, "System Analysis and Design with UML: Object Oriented Approach", John Wiley & Sons, 2010, 581 p.

5

Back of extended CRC Card

Attributes:

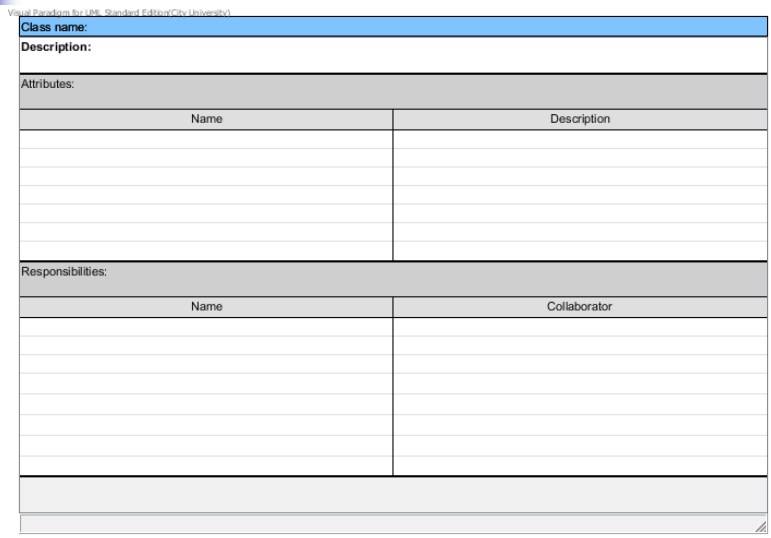
- Amount (double)
- Insurance carrier (text)

Relationships:

- Generalization (a-kind-of): Person
- Aggregation (has-parts): Medical History
- Other Associations: Appointment

6

CRC Cards in Visual Paradigm

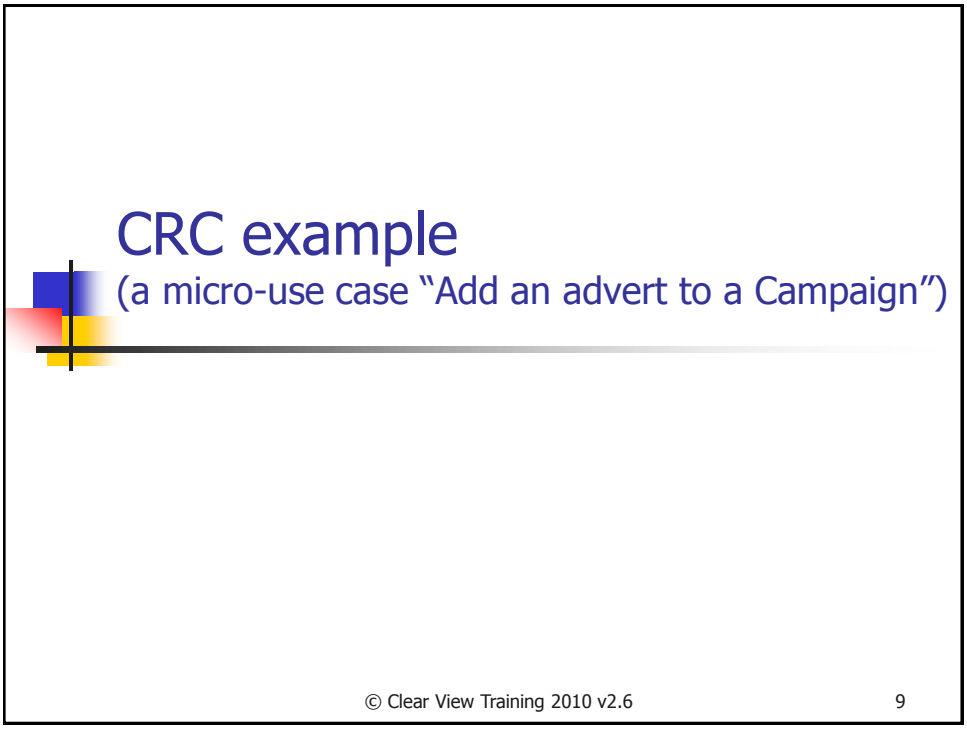


7

CRC cards and Class diagrams consistency

- Every CRC card should be associated with a ***class*** on the class diagram
- Responsibilities of a class are:
  - "knowing". These are captured by the attributes of a class;
    - Class attributes with a type that is another class ***imply a relationship*** (typically a composition) between classes.
  - "doing". These are captured by the operations of a class.
    - Access to attributes (ideally) must be indirect, i.e. via the class operations (encapsulation).
- Collaborators of a class are other classes
  - The class "delegates" some of its responsibilities to the collaborators.
  - If a class A has a collaborator B then there is an ***association*** between class A and class B on the class diagram.

8



11.2

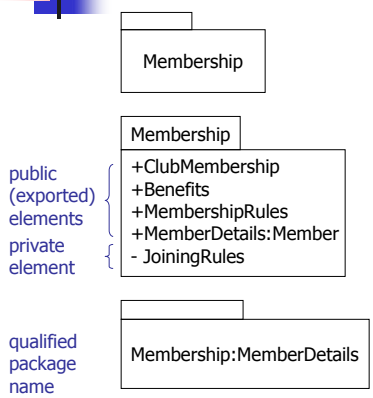
Analysis packages

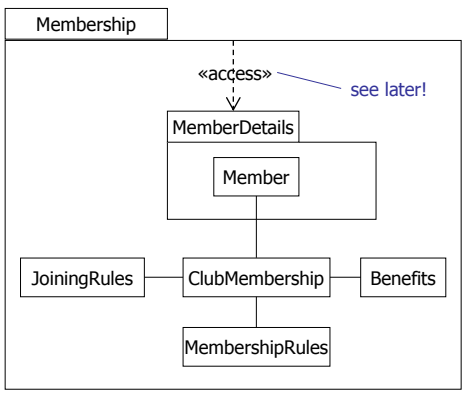
- A package is a *general purpose* mechanism for organising model elements into groups. A package:
  - Groups semantically related elements
  - Defines a "semantic boundary" in the model
  - Provides units for ***parallel working*** and ***configuration management***
  - Each package defines an ***encapsulated namespace***, i.e. all names must be unique within the package
- In UML 2 a package is a purely ***logical grouping*** mechanism
  - Use components for ***physical grouping***
    - We will cover components later in this module
- ***Every*** model element is ***owned by exactly one package***
- Analysis packages contain:
  - Use cases, analysis classes, use case realizations (e.g. sequence diagrams), other analysis packages (i.e. packages can be nested)

11

11.2

Package syntax





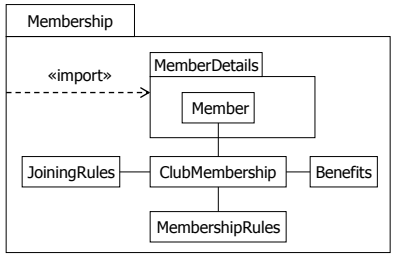
standard UML 2 package stereotypes	
«framework»	A package that contains model elements that specify a reusable architecture
«modelLibrary»	A package that contains elements that are intended to be reused by other packages Analogous to a class library in Java, C# etc.

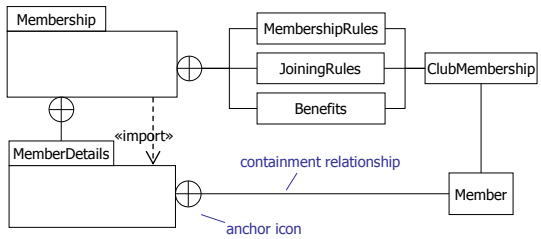
 12

11.4

Nested packages

- If an element is visible within a package then it is visible within all nested packages
  - e.g. Benefits is visible within MemberDetails
- Show containment using nesting or the containment relationship
- Use «access» or «import» to merge the namespace of nested packages with the parent namespace

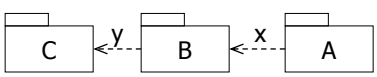




11.5

Package dependencies

dependency	semantics
Supplier «use» Client	An element in the client uses an element in the supplier in some way. The client depends on the supplier. Transitive.
Supplier «import» Client	Public elements of the supplier namespace are added as ***public*** elements to the client namespace. Transitive.
Supplier «access» not transitive Client	Public elements of the supplier namespace are added as ***private*** elements to the client namespace. Not transitive.
Analysis Model «trace» Design Model	«trace» usually represents an historical development of one element into another more refined version. It is an extra-model relationship. Transitive.
Supplier «merge» Client	The client package merges the public contents of its supplier packages. This is a complex relationship only used for metamodeling - you can ignore it.

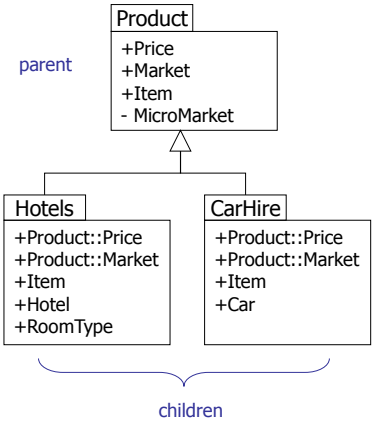


transitivity - if dependencies x and y are transitive, there is an *implicit* dependency between A and C

11.6

Package generalisation

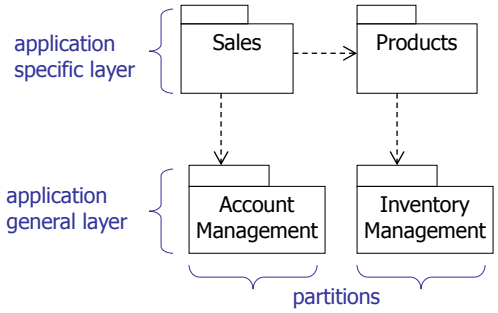
- The more specialised child packages *inherit* the public and protected elements in their parent package
- Child packages may *override* elements in the parent package. Both Hotels and CarHire packages override Product::Item
- Child packages may *add* new elements. Hotels adds Hotel and RoomType, CarHire adds Car



 15

11.7

Architectural analysis



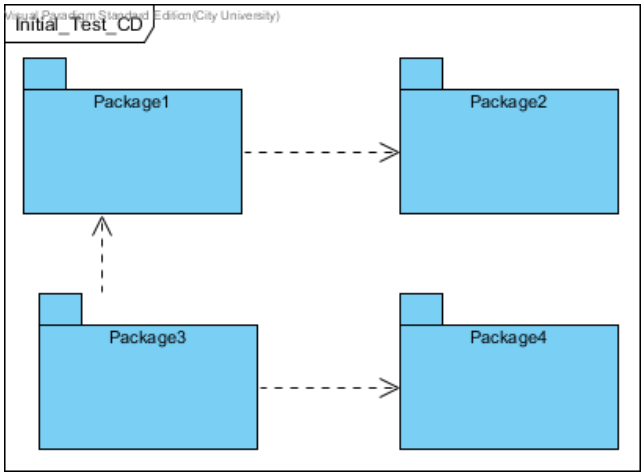
- This involves organising the analysis classes into a set of ***cohesive*** packages
- The architecture should be *layered* and *partitioned* to separate concerns
  - It's useful to layer analysis models into application specific and application general layers
- Coupling between packages should be ***minimised***
  - May need multiple iterations trying different packaging of modelling elements
- Each package should have the minimum number of public or protected elements

 16

Associations between classes in different packages

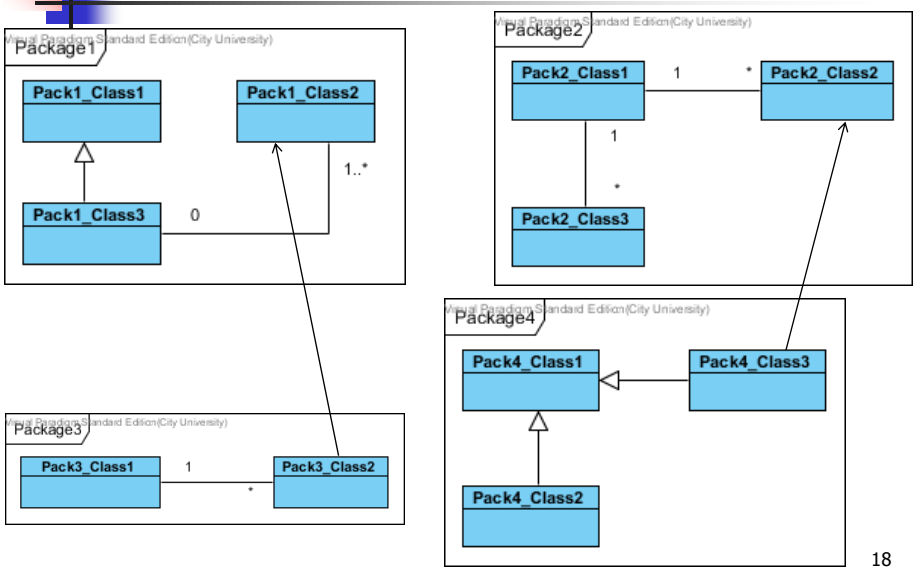
Consider a system of four packages as shown.
The packages contain a number of classes.

Each class is modelled as a separate class diagram, i.e. we can show the associations between the classes in the ***same*** package.
How do we show associations between classes which belong to ***different packages***, e.g. associations between classes in Package 1 and Package 3?



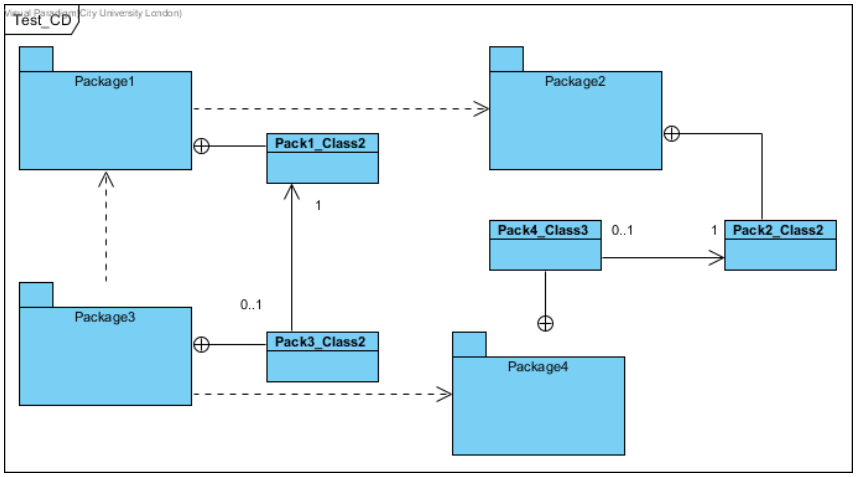
17

Packages internal structure



18

Solution



19

11.7.1

Finding analysis packages

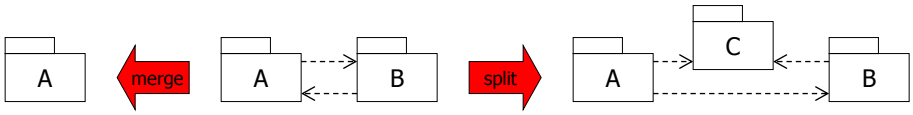
- These are often discovered as the model matures (typically when the number of classes grows)
- We can use the natural groupings in the use case model to help identify analysis packages:
  - One or more use cases that support a particular business process or actor
  - Related use cases (i.e. with <<include>>/<<extend>> or generalisation relationships)
- Analysis classes that realise these use case will often be part of the same analysis package
- Be careful, as it is common for ***use cases*** to *cut across* analysis packages!
  - One class may participate in the realisation of several use cases that are allocated to different packages

 20

11.7.2

Analysis packages: guidelines

- A cohesive group of closely related classes or a class hierarchy and supporting classes
- Minimise dependencies between packages
- Localise business processes in packages where possible
- Minimise nesting of packages
- Don't worry about dependency stereotypes
- Don't worry about package generalisation
- Refine package structure as analysis progresses
- 4 to 10 classes per package
- Avoid cyclic dependencies!



 21

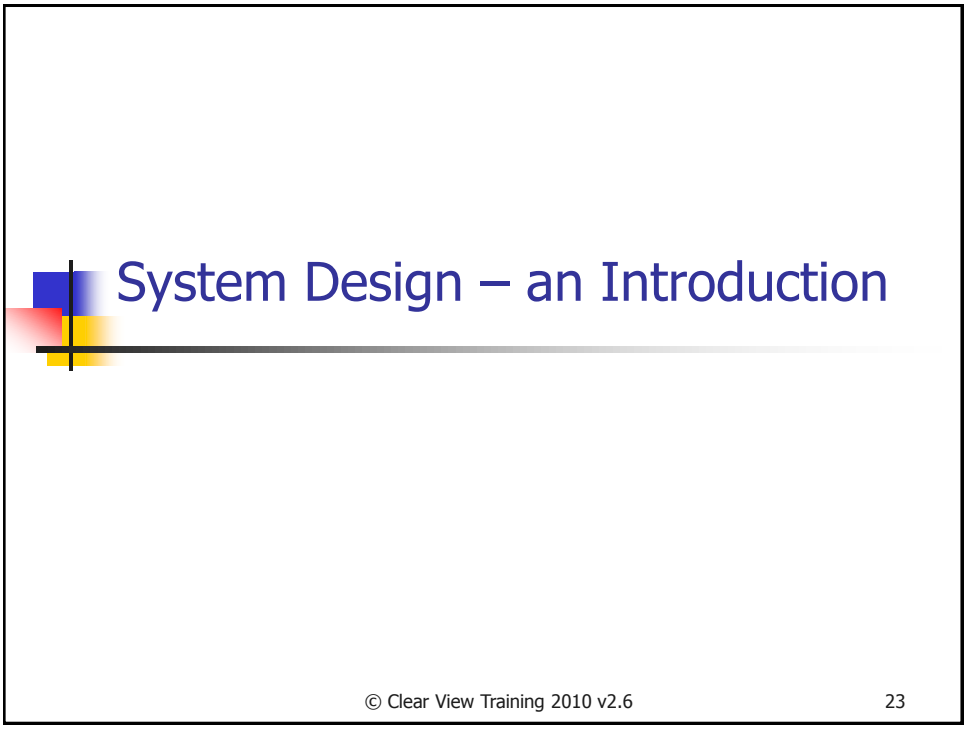
11.8

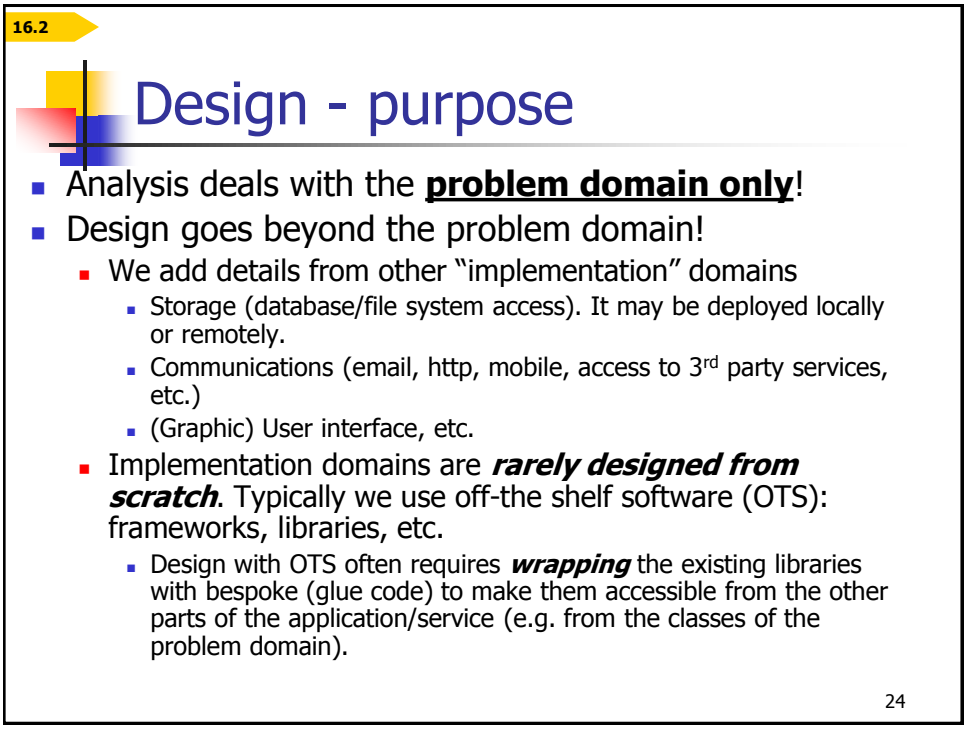
Summary

- Packages are the UML way of grouping modelling elements (i.e. implementing "divide and conquer approach to complexity")
- There are dependency and generalisation relationships between packages
- The package structure of the analysis model defines the logical system architecture

Analysis packages are covered in Chapter 11 of Arlow's book.

 22





16.2

Design - purpose

- Software Design solves two problems:
  - ***High level*** design concentrates on software architecture – s/w components and how they interact.
  - Low level design **specifies** the newly developed classes with ***sufficient details*** so that they can be implemented with an object-oriented programming language (e.g. Java, C++, etc.).
    - Many modern UML tools provide code generation facilities and automate some aspects of software implementation.

The output form low level design are *specification(s)*, not a complete implementation.

25

16.2

High Level design

- Decide on the ***subsystems*** that will form the final product:
  - problem domain subsystem
  - persistence subsystems, i.e. functionality that allows an application/service to access a DB server, a file system, a remote storage (e.g. in the cloud), etc.
  - user interface (e.g. GUI)
  - Communications with users and possibly 3rd party services
- Define the *interfaces* (i.e. API) that each of the subsystems will ***provide*** to the other subsystems and ***will rely upon***
- There is an extensive literature of "software architecture", which is a synonymous to high-level design.
- Software design with off-the-shelf software has been studied very extensively at the turn of the millennium.
  - This is a special form of high level design, when designers are concerned with solving the problem at hand using off-the-shelf software
  - Most of the applications today, even highly innovative ones, use extensively off-the-shelf software (so called "frameworks" for building GUI, for access to databases, the communication stacks) available off-the-shelf.

26

16.2

Low Level (detailed) design

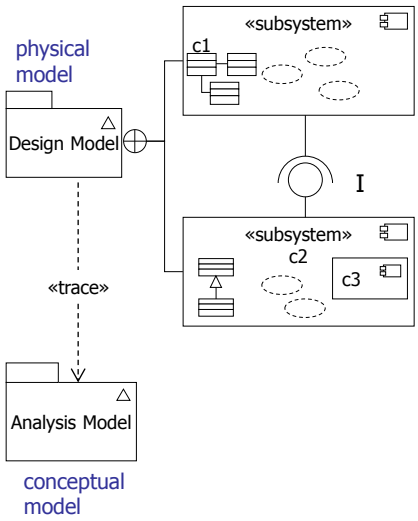
- This part of OO design is focussed on specifying the ***newly developed classes*** with sufficient details, so that they can be implemented using an OO programming language
  - Decide how the system's functions are to be implemented, which typically will require the problem domain classes to be ***refined***
  - Refine the relationships of the analysis model so that they can be implemented using a programming language
    - Some of the concepts from the analysis model (e.g. association classes) may be removed, if they are not supported by the chosen programming language
  - Design patterns can be adopted and detailed for the specific application context
  - Efficiency of computation are addressed in detailed design, too:
    - Synchronous vs. asynchronous (i.e. with callbacks) computations,
    - Use of concurrency (threads, concurrent processes) together with the necessary mechanisms to control access to a shared resource (e.g. locks, mutex, or using non-blocking multicasting protocols), etc.

 27

16.3

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

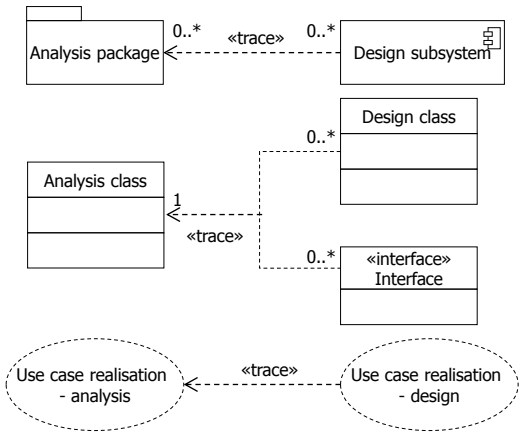


 28

16.3.1

Artifact trace relationships

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



 29

16.3.2

Should we 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 the low level details
- We need to maintain 2 models if:
  - It is a big system (>200 design classes)
  - It has a ***long expected lifespan – this concern is essential!***
  - It is a strategic system which may go through major revisions
  - We are ***outsourcing*** construction of the system – effective communication with 3rd parties requires both analysis and design models.
- 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

 30

16.6

Summary

- Design is a very important part of software construction putting together the problem domain and the various utility domains
- Purpose – to decide *how* the system's functions are to be implemented
- artifacts:
  - Design classes
  - Interfaces
  - Design subsystems
  - Use case realisations – design
  - Deployment

Detailed design

High level design

Introduction to Design is covered in Chapter 16 of Arlow's book.

31

Design - classes

32

17.3

What are design classes?

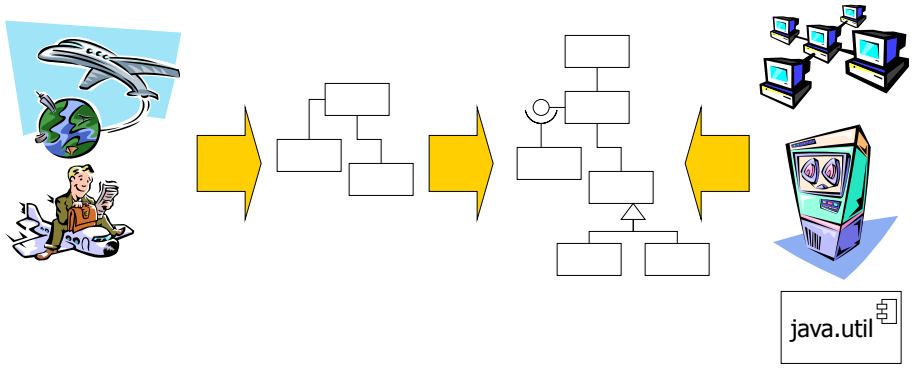
- Design classes are classes whose specifications have been completed to such a degree that they ***can be implemented***
  - They specify an actual piece of code
- Design classes arise from analysis classes:
  - Remember - analysis classes arise from a consideration of the problem domain *only*
  - A refinement of analysis classes to include ***implementation details***
  - One analysis class may become ***many design classes*** (e.g. in design you may add many controllers, a single boundary class may become a fully fledged GUI)
  - 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 also from the ***solution domain***
  - Utility classes – String, Date, Time etc.
  - Middleware classes – database access, communications, etc.
  - GUI classes – Applet, Button, etc. Mobile platforms have their specific GUI frameworks.

33

17.3

Sources of design classes

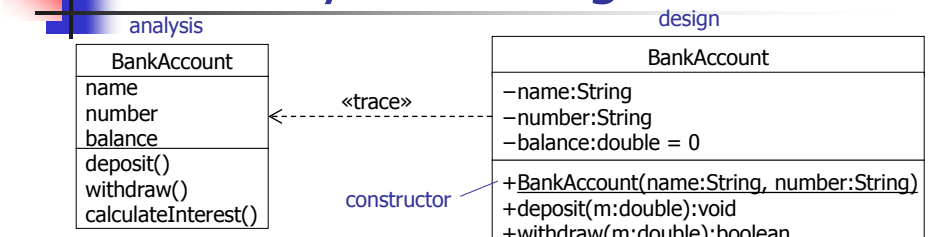
Problem domain | Analysis classes | Design classes | Solution domain



34

17.4

Anatomy of a design class



- 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

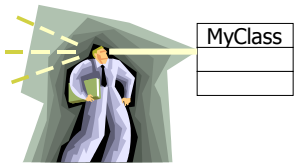
 35

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!



 36

17.5.1 17.5.2

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

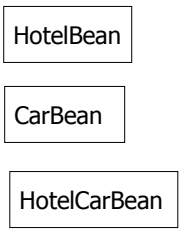
The public members of a class define a "contract" between the class and its clients

 37

17.5.3 17.5.4

High cohesion, low coupling

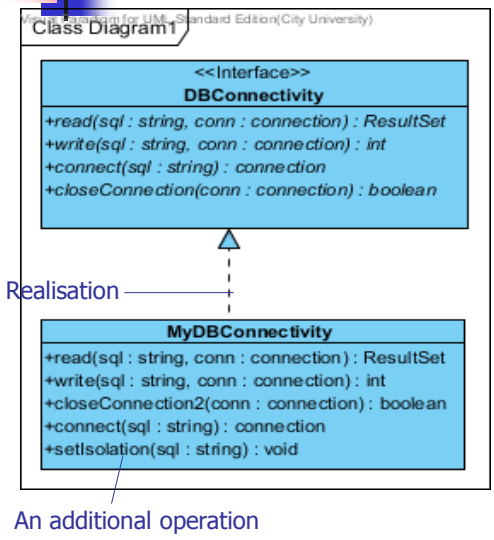
- High cohesion:
  - Each class should have a set of operations that support the ***intent of the class***, no more and no less (remember CRC cards)
  - Each class should model a single abstract concept
  - If a class needs to have ***many responsibilities***, then some of these should be implemented 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 (CRC cards may be useful here)
  - 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



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

 38

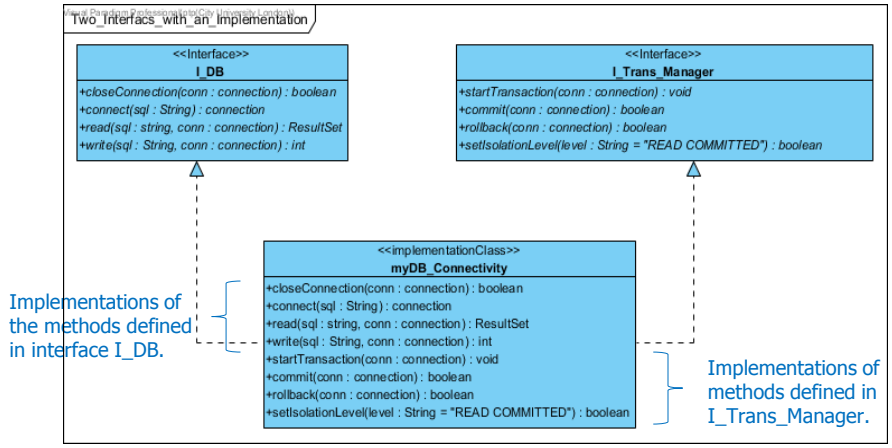
Interfaces in class diagram



Interfaces define a ***contract***, typically by defining a set of abstract (public) operations (i.e. no code).
Designing to interfaces is a ***good practice*** widely used in component based software development (to be discussed in detail in Lecture 5).
A classifier (a class or a component) can provide an implementation (i.e. realisation) of an interface.
Implementation classes must provide implementations (i.e. code) of all operations defined by an interface. It may also provide ***additional operations/methods***.
Designing to interfaces, i.e. not relying on additional operations provided by an implementation class allows for ***easy replacement*** of the implementation classes that realise the same interface.
https://docs.oracle.com/javase/tutorial/java/IandI/usinginterface.html

39

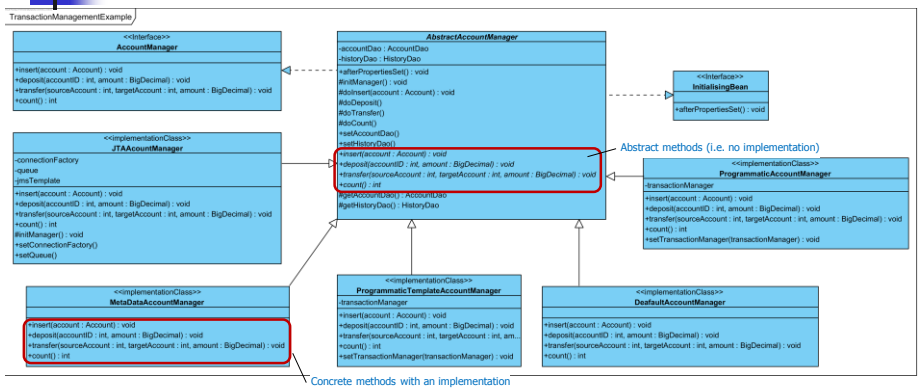
Design with interfaces: example 2



- An implementation class may implement more than 1 interface.

40

Design with interfaces: example 3



- An interface with several implement classes (MetaDataAccountManager, Programmatic TeplateAccountManager, …).
- The same interfaces are used to work with meta-data (i.e. DB schema) and with "normal data" (i.e. tables), but implementation classes for meta and normal data will be different.
- Interestingly, the immediate implementation class is an abstract class (some of its methods are abstract – those shown in italic in the red box of AbstractAccountManager).
  - Each of the concrete implementation classes, which inherit from AbstractAccountManager, must provide an implementation of the abstract methods.

© Rob Harrop, Jan MAchacek, "Pro Spring", Apress, p. 399.

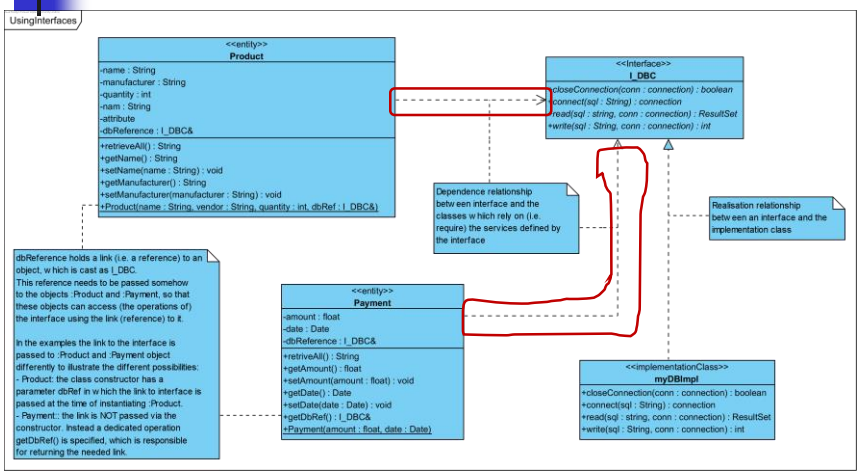
41

Associations between classes and interfaces

- Can a class have an association with an interface?
  - No, associations are meant to represent *links between objects*
  - In other words only if instances of two classes have a link (and therefore can communicate with one another) will an association between the respective classes be *admissible*.
  - Can we instantiate an interface? 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 interface.
  - When an interface method is invoked, the code of the respective realisation class is executed.
  - Replacing the implementation class should (in theory) has little effect on the class invoking the interface methods (the interface defines a contract).
- When at run time a new instance of an implementation class is created, the new reference to the object can be cast as a *reference* to the data type defined interface. Thus the reference to the newly created instance will appears as an instance of the respective interface :
  - i.e. the instance of the implementation class is masqueraded to appear as an instance of the data type defined by the interface.
  - This is very cool!

42

Using interfaces



- Product and Payment are classes, which will ***make use*** of the interface I_DBC, e.g. to store data in a database.
- This usage of the interface is modelled with a dependence relationship between Product/Payment and I_DBC.

43

17.6.3

Inheritance vs. interface realisation

- With inheritance we get two things:
  - Interface – the public operations of the base class
  - Implementation – the attributes, relationships, protected and private operations of the base class
- With ***interface realisation*** we get exactly one thing - a contract defined by the interface:
  - An interface – a set of public operations, attributes and relationships that have no implementation.

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

 44

17.9

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)
    - This is an option only in case the programming language planned for implementation supports multiple inheritance.
    - Some languages, e.g. Java, do not support multiple inheritance between classes.
  - Use interfaces rather than inheritance to define contracts.

Design classes are covered in Chapter 17 of Arlow's book.

45