CITY UNIVERSITY LONDON

IN2013 Lecture 4

Object- Oriented Analysis and Design:

Advanced Class Diagrams

Dr Peter Popov

21st October, 2025

1

Objectives

More techniques for building class models

- CRC cards
  - Discovering classes in the problem domain
  - Discovering relationships between classes
- Class stereotypes (Robustness Analysis)

Dealing with complexity in UML models

- UML Packages

2

Recall: Objects, Classes ...

- An object is a set of data ("attributes") and operations that can work on these data

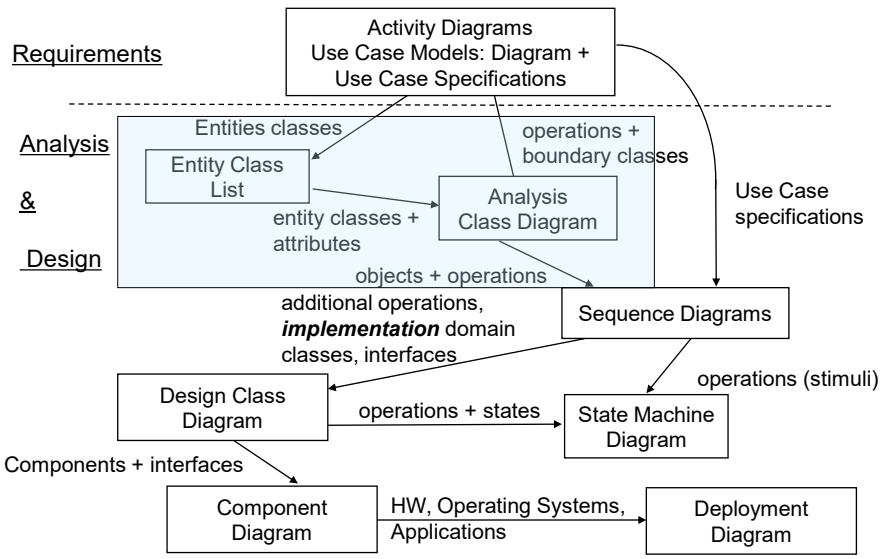
(*information hiding)*

- A class is a classifier, defines a data type from which objects can be instantiated
- All objects instantiated from the same class have the same attributes (types of the pieces information describing their current state), but each one with its own values of the attributes*
- The same operations are available for all objects derived from the same class *
- Objects and classes are to represent entities of interest for developing the system: often real-world entities *but only aspects of interest*

* some operations and attributes have ***class*** scope. This is established in design

3

Building UML Models, Typical Steps



4

Finding Classes for a Specification

- Goal: to list and describe
  - the data items that your system will need to hold about the outside world (the business/activity to be supported)
  - the appropriate ways of manipulating them
    - preliminary stage may be called an *entity class list*
  - Expect to need explanations in words too ("specifications"), not just diagrams!
    - "responsibilities" of classes ("holds all information about ..."), meanings of data
      - even long names for attributes may be ambiguous
    - similarly for operations (what they must do to the data, with constraints and exceptions)

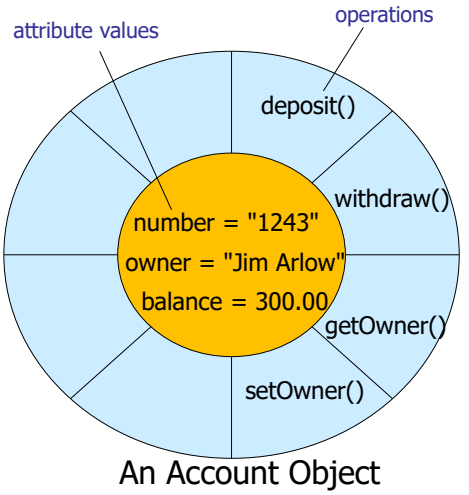
Analysis Class Models: Goals and Warnings

- Function of the model:
  - for the ***developers*** to understand what data and operations they'll need to program for
  - also, for understanding the business problem better
- but note the vital difference between the *real-world* entities and the *objects* that describe them in the computer
  - e.g., for a library automation system, difference between:
    - an object of the "BookDescriptor" class in analysis model (can be recorded as loaned, returned, lost, damaged) vs.
    - a real "book" (may be in a different state from the one recorded, changes are not instantaneous, ...)
- thus, to check that the software system would work *properly* for its business purpose if built as specified (*validation* of the specs), you need to reason about the pertinent parts of the world as well.

Fundamentals of Object Orientation

Objects
- Encapsulation
- Messages

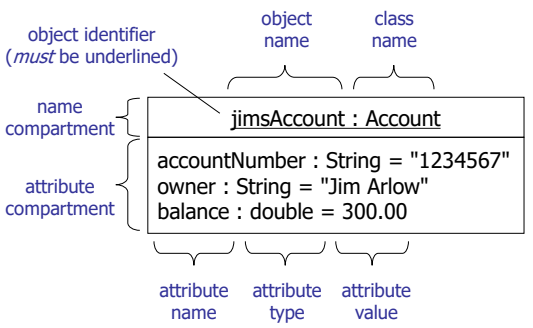
Classes



An Account Object

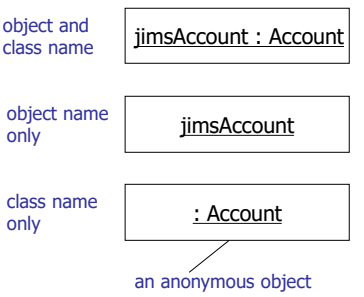
7

UML Object Syntax



variants

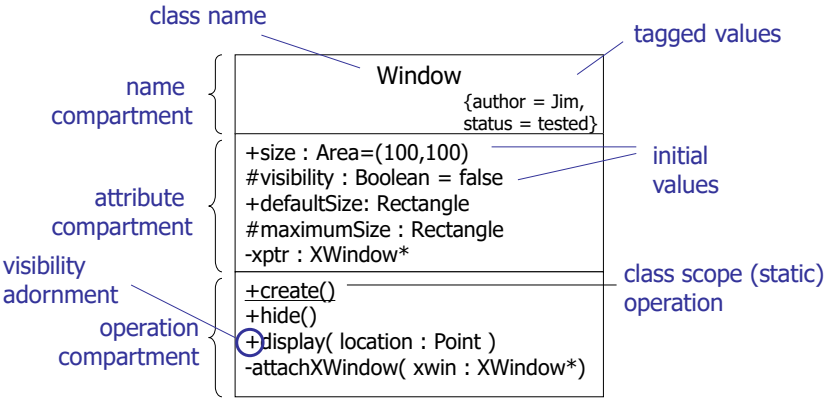
(N.B. we've omitted the attribute compartment)



Why do not we show operations in the object diagram?

8

UML class notation



Classes are named in UpperCamelCase

Problem Domain Analysis

What methods for analysis do we know?

- Noun/verb analysis – leads to "analysis" (i.e. problem domain) classes with:
  - Important attributes
  - Important operations
  - Associations (relationships) between classes

Roadmap

- CRC Cards
- Boundary and Control Classes
- UML Packages
- UML in colour and class archetypes

11

Part 1: CRC Cards
Class – Responsibility - Collaborators

12

Class, Responsibilities and Collaborators (CRC) Cards

The method is applied using CRC ('stick-it' like) cards:

Class Name: BankAccount	
Responsibilities:	Collaborators:
Maintain balance	Bank

things the class does

things the class works with

Each class must have its own CRC card.

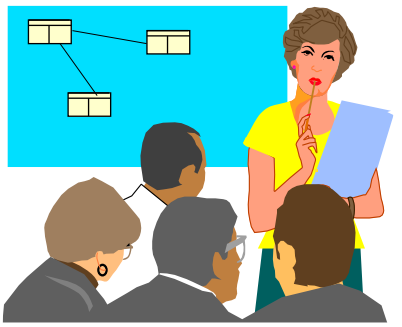
CRC card procedure

Proposed by Ward Cunningham, Kent Beck

Recommended by Extreme Programming (XP) supporters.

Two steps procedure:

- Part 1: Brainstorm
  - *All* ideas are good ideas in CRC analysis
  - Never argue about something – write it down and analyse it later!
  - *Anthropomorphism* (i.e. pretending that the classes have human characteristics)
    - Who or what you are? What do you know? What can you do?
- Part 2: Analyse information - consolidate with noun/verb.



Responsibilities

Responsibilities of a class fall broadly into two categories:

- *Knowing* – what an instance of a class knows
  - The values of own attributes
  - Its relationships (i.e., the classes it is associated with)
- *Doing*
  - An instance can execute its own operations
  - Can request the execution of an operation of another object it knows about

15

Collaborators

Collaborators of a class are ***other classes.***

Instances of a class interact with *instances of the collaborators* (two or more) to achieve a business process described by a use case.

- Useful to think of collaborations in terms of *client-server-contract:*
  - Client is the instance that sends a request (i.e. a message) to an instance of another class (collaborator) for an operation to be executed.
  - Server (supplier) is the object (of the collaborator class) that receives the request from the client.
  - Contract – formalises the interaction between the client and the server (e.g. the Contact between a Patent and a Doctor is to be there for an Appointment).
- Typically, a class "delegates" some of its responsibilities to its collaborators.
  - Class – Collaborator relationship implies that the two classes must have an association in the class diagram (i.e. the instances are linked) so that the delegation of responsibilities can be done.
  - Class – Collaborator relationship is "asymmetric" relationship:
    - Class A being a collaborator of Class B does not imply that Class B is a collaborator of Class A.

16

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.

Back of extended CRC Card

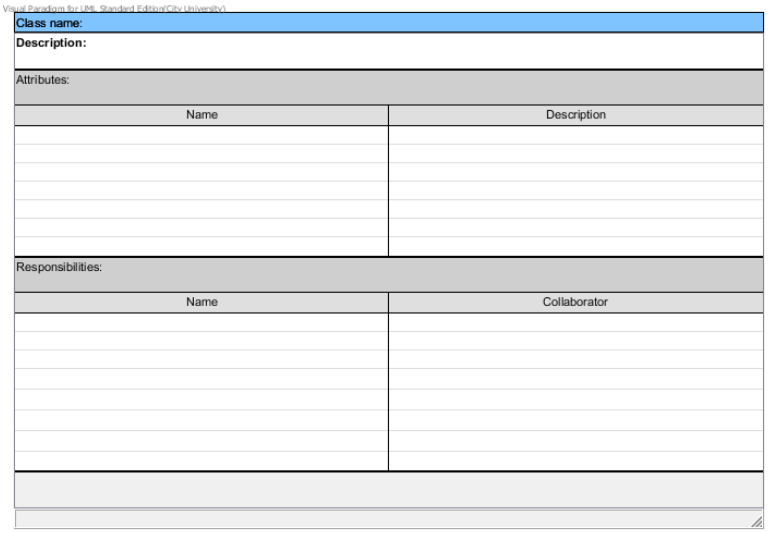
Attributes:

- Amount (double)
- Insurance carrier (text)

Relationships:

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

CRC Cards support in Visual Paradigm



CRC cards: How to use them

Step 1: Create CRC cards for each candidate class
- Nouns from use-case specification (flows) or from short descriptions are candidate classes

Step 2: Examine Common Object Lists.
- Look for: Things, Roles, Relationships

Step 3: Role play the CRC cards
- Team members play the roles of classes and check whether the use cases can be implemented with the current set of CRCs or identify if/how the system can 'break down'.

Step 4: Create the class-diagram
- CRC Responsibilities become either operations (doing responsibilities), or attributes (knowing responsibilities)
- Collaborators become relationships: association, inheritance

Step 5: Review the class diagram
- Missing and unnecessary classes, attributes, operations and relationships identified

Step 6: Incorporate patterns
- Identify useful patterns applicable to the system and modify the CRCs.

Step 7: Review the Model
- Typically done in a review meeting with other team members. Omissions fixed and obsolete classes removed

CRC Cards example

(a micro-use case "Add an advert to a Campaign")

 21

21

Takeaway (Part 1)

Every CRC card should be associated with a single ***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*** 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 its collaborators.
- If a class A has a collaborator B, then there is an ***association*** between class A and class B on the class diagram.
  - Class – Collaborator relationship is ***asymmetric***.

22

Further Reading (Part 1)

Jim Arlow's book "UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design", Edition 2: Chapter 8.

23

Part 2: Robustness Analysis

24

Roadmap

- CRC Cards
- Boundary and Control Classes
- UML Packages
- UML in colour and class archetypes

25

Robustness analysis

Walk through the flow of each use case and identify ***3 kinds of classes (stereotypes)***. These broadly implement the Model-View-Controller pattern, popular in software design:

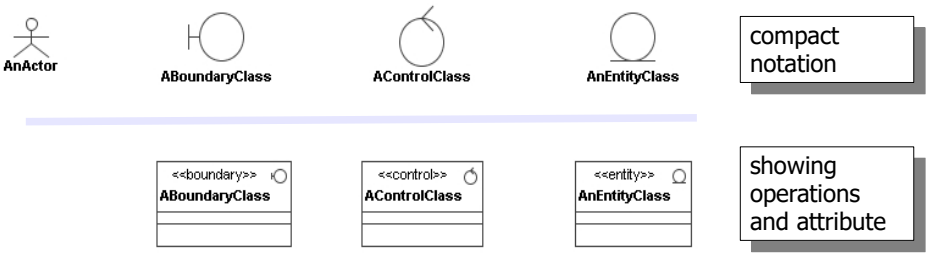
- ***Boundary*** classes – actors use these to communicate with the system (e.g., GUI, but also classes which allow a software system to communicate with other systems, e.g. 3rd party systems).
  - Each actor must have a *dedicated* boundary class which allows the actor to interact with the system.
  - In analysis several actors (roles) may share a boundary class (e.g. all human actors may use the same boundary class).
- ***Entity*** classes – these come from the ***problem domain model*** and often represent persistent data. The classes derived via noun-verb analysis are typically entity classes.
- ***Control*** classes – capture the ***application logic*** defined by use case specifications and glue together the user interface and the entity classes.
  - Typically, in analysis we use a ***small number*** control class, possibly a single control class.

Robustness analysis gives you:

- A first guess at what the right analysis classes *might* be.
- A check that your *use case flow can actually be realized* (via message exchange between objects) of the classes captured in the class diagram.
- Ideas about the user interface.

26

Robustness analysis notation



Stereotypes such as «boundary» extend the UML meta model by introducing new modelling elements based on existing ones

- Each stereotype can have its ***own icon***
- They are one of the UML extensibility mechanisms

Visual Paradigm support both notations.

Robustness analysis rules

- The different types of classes can only be ***associated*** as shown - communication is only allowed between class kinds as shown in the table.

	AnActor	ABoundaryClass	AControlClass	AnEntityClass
AnActor		→		
ABoundaryClass			→	
AControlClass		→	→	→
AnEntityClass			→	→

Takeaway (Part 2): Robustness Analysis in practice

Analyse each use case specification for:

- Nouns
  - If a noun describes something that the system must keep information about, it indicates an ***entity class.***
    - Some nouns may indicate attributes of entity classes.
    - Some nouns may indicate an association class and its attributes.
  - If a noun describes something an actor interacts with, it indicates a ***boundary*** class.
  - Each actor *MUST have a boundary class* to interact with the system.
    - Several actors may share the same boundary class.
- Verbs
  - Describe things the system does – indicate ***control*** classes.
  - May imply relationships (associations) between classes.
  - May indicate operations of a class.

29

Further Reading (Part 2)

Jim Arlow's book "UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design", Edition 2: Chapter 8.

30

Part 3: Analysis Packages

31

Roadmap

- CRC Cards
- Boundary and Control Classes
- UML Packages
- UML in colour and class archetypes

32

Analysis packages

A package is a *general-purpose* mechanism for organising modelling 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 (but name duplications are allowed in different packages, although this is rarely used).

In UML 2 a package is a mechanism of purely ***logical grouping***.

- Use components for ***physical grouping*** (e.g., merging several compiled Java classes in an archive such as a jar file).
  - We will cover components later in this module.

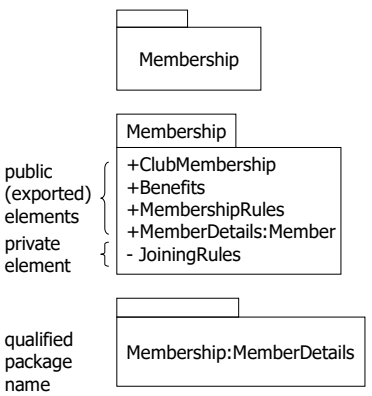
Every model element is ***owned by exactly one package***.

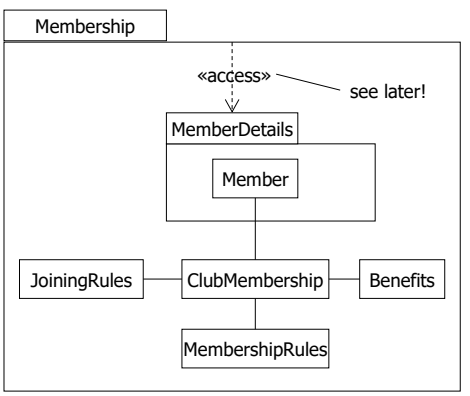
Analysis packages may be used with:

- Use cases, analysis classes, use case realizations (e.g., sequence diagrams).
- Analysis packages may contain other packages, i.e., ***packages can be nested***.

33

Package syntax



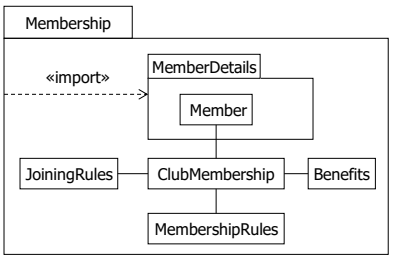


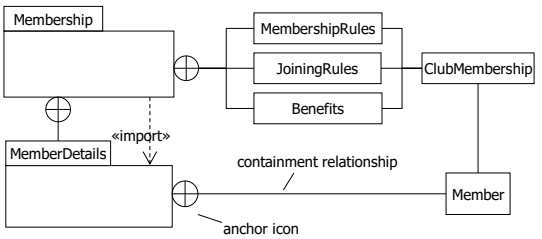
standard UML 2 package stereotypes	
«framework»	A package that contains model elements that specify a reusable architecture
«modelLibrary»	A package that contains elements that are intended to be reused by other packages Analogous to a class library in Java, C# etc.

34

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.



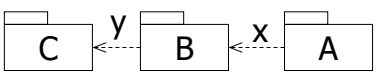


35

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

36

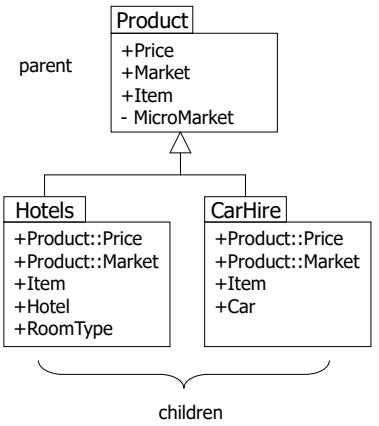
11.6

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

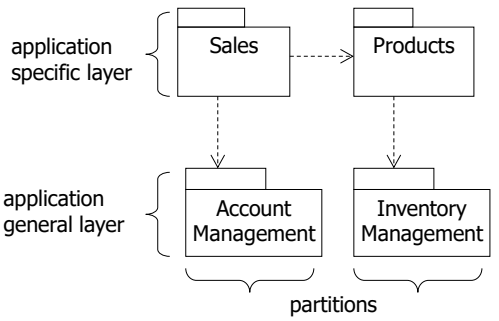
Child packages may ***add*** new elements. Hotels adds Hotel and RoomType, CarHire adds Car



37

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.

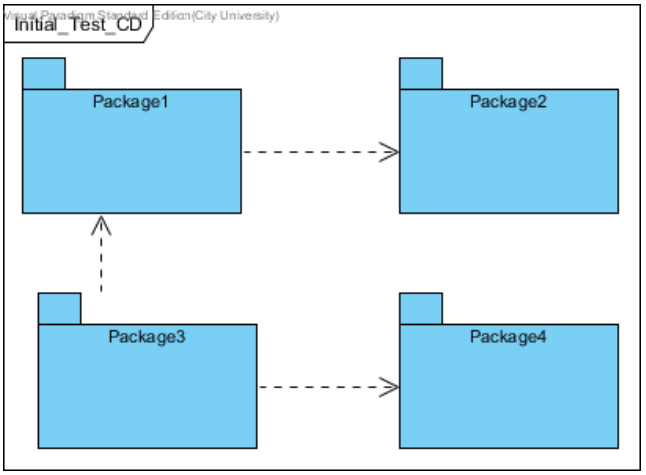
38

Associations between classes in different packages

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

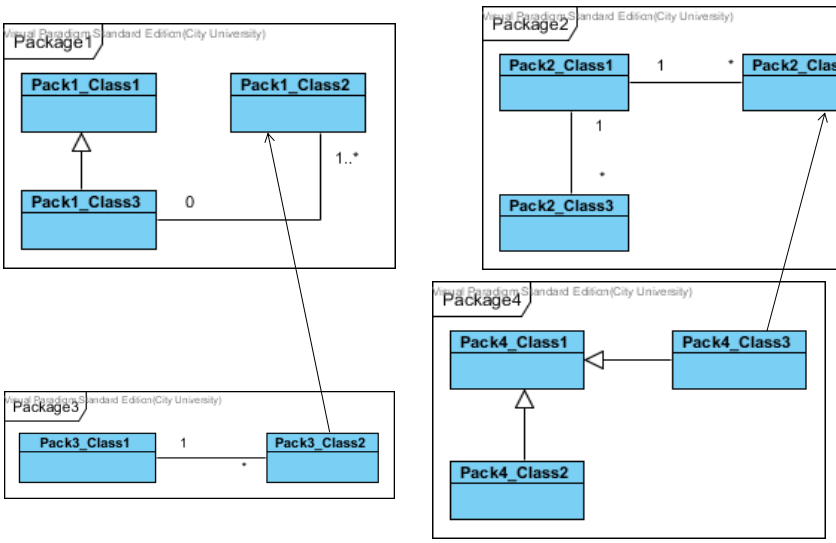
Each package is detailed in a separate class diagram, which shows:
- The classes which belong to the package, and
- the associations between the classes in the ***same*** package (i.e., within the package).

How do we show the associations between classes which belong to ***different packages***, e.g., associations between classes in Package 1 and Package 3?



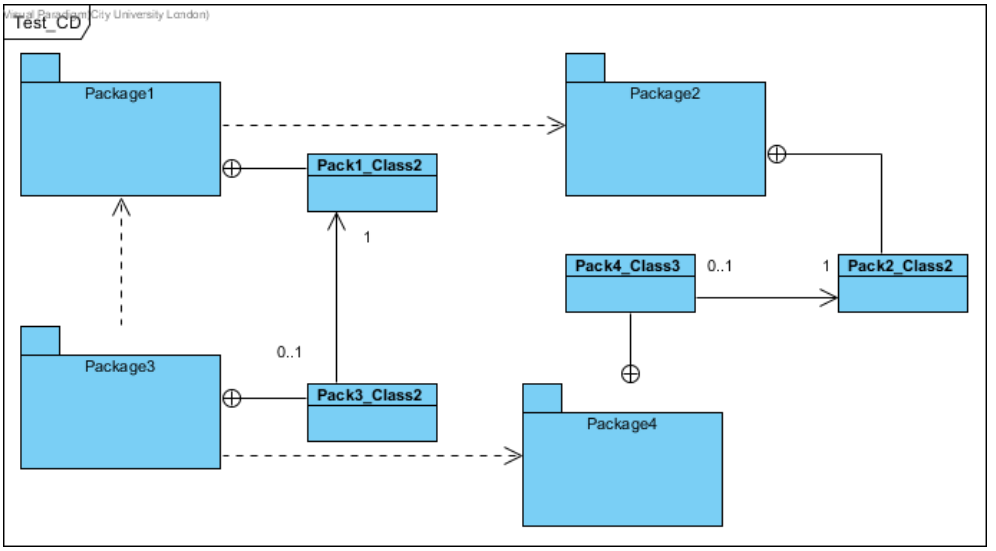
39

Packages internal structure



40

Solution



41

Finding analysis packages

Analysis packages 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 an actor.
- Related use cases (i.e., with <<include>>/<<extend>> or generalisation relationships).

Analysis classes that realise these use cases 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.

Defining a good package structure is a *best effort activity*, i.e., it is difficult to know whether we produce the "optimal allocation to packages" and typically requires several iterations.

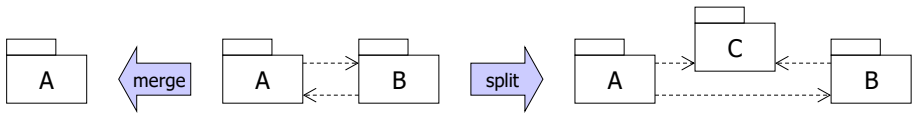
- Trying to minimise the number of cross package association is worth considering, too.

42

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 (whether this is <<use>>, <<access>>, <<import>>, etc.).
- Don't worry about package generalisation.
- Refine package structure as analysis progresses.
- 4 to 10 classes per package.
- Avoid cyclic dependencies! .



Takeaway (Part 3)

- Packages are the UML way of *grouping* modelling elements (i.e. implement "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.

Roadmap

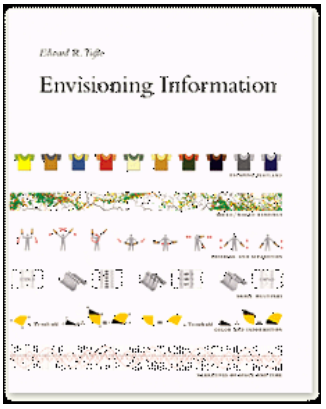
- CRC Cards
- Boundary and Control Classes
- UML Packages
- UML in colour and class archetypes

45

Using colour in models?

Colour is a 'natural' extension of the concept of visual modelling

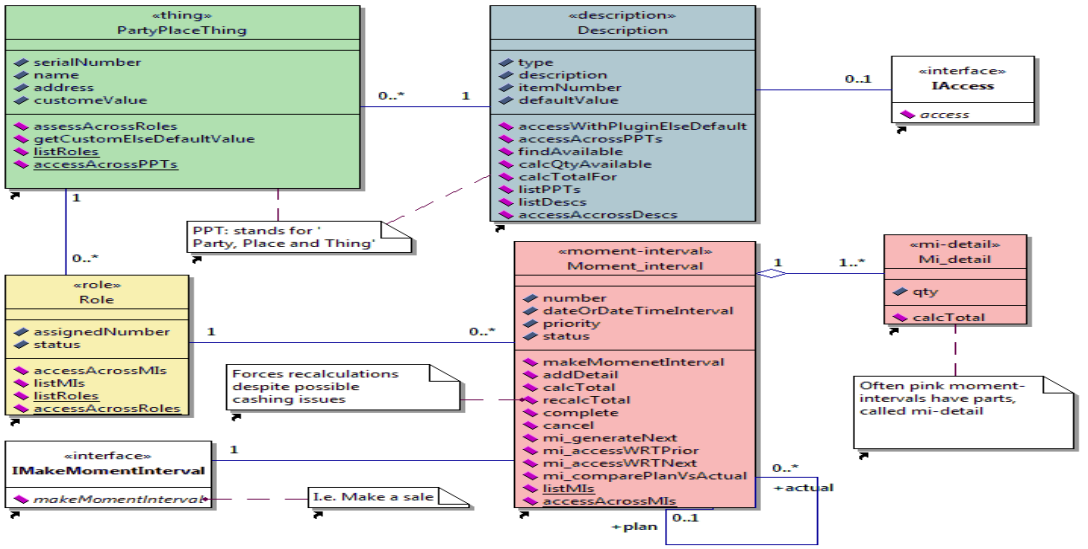
- One can glance over a diagram and realise quickly what is in the diagram from the colour of the modelling elements



- Among the most powerful devices for reducing noise and enriching the content of displays is the technique of layering and separation, ***visually stratifying*** various aspects of the data. [He then describes how to do this: use distinctions and shape, lightness, size, and ***especially colour***]
  - *Edward R. Tufte, Envisioning Information , Cheshire, CT: Graphics Press, 1990*

46

UML 'archetypes' and their plug-ins



47

Why is this useful?

The archetypes and their plug-ins form a 'Domain neutral component', a useful ***reusable component:***

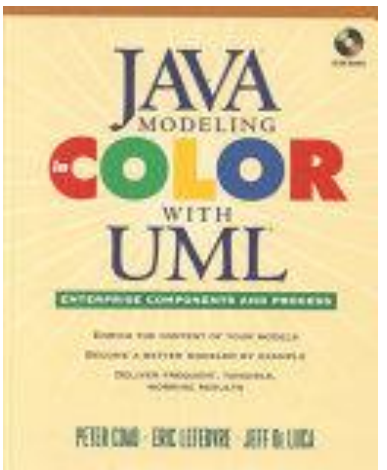
- A model building block.

Multiple domain neutral components can be linked with each other to represent different complex models.

- Peter Coad and co-authors ("UML in Color") claim that they have used the domain neutral component to build ***hundreds different models.***
  - Sales and other business transactions.
  - Production/Manufacturing processes.

Whether you like the colour or not, using archetypes saves a lot of time.

- Archetypes can be thought of as '***analysis patterns***'.
  - CRC cards recommends using patterns at early analysis stages.
- ***Design patterns*** are an essential part of System's Design, which we will look into later in the module.



48

Summary of the Lecture

Problem Domain Analysis is an important step in defining software requirements

A number of methods exist for analysing the problem domain and finding analysis classes, their attributes, operations and relationships.

- Noun/verbs
- CRC cards
- Robustness analysis
- UML Packages
- Archetypes (and patterns) in analysis class diagrams