OO Analysis and Design with UML 2 and UP

Dr. Jim Arlow,
Clear View Training Limited
Introduction
About you…

- Name?
- Company?
- What are you working on?
- Previous experience of OO?
- Previous experience of modelling?
- One thing you hope to gain from this course?
- Any hobbies or interests?
Structure of this course

Introduction
UML Principles
UP
Requirements
Analysis
Design
Implementation
Summary

OO Analysis and Design with UML and UP
Labs
Guiding principles

- This course uses the Unified Software Development Process (UP) to define the activities of OO analysis and design using UML.
- The UP is the industry standard software engineering process for the UML.
Course materials

- For easy reference, all slides in this course are cross referenced to sections in the course book "UML 2 and the Unified Process"
  - There is an example cross reference icon in the top left hand corner of this slide

ISBN: 0321321278
Labs

- This is a practical course, and there is a lot of laboratory work
- Our approach to this work is cooperative rather than competitive
  - Work together
  - Ask each other for help
  - Share ideas and experience
- Don’t get bogged down!
  - If something brings you to a halt for more than 10 minutes, then ask for help
Goals of the course

- To provide a thorough understanding of OO analysis and design with UML
- To follow the process of OO analysis and design from requirements capture through to implementation using the Unified Software Engineering Process as the framework
- To have fun!
Conditions of satisfaction

- You will know you are succeeding when:
  - You can read and understand UML diagrams
  - You can produce UML models in the laboratory work
  - You apply your knowledge effectively back at your workplace

- Questions:
  - You can ask questions at any time!
  - Your participation is always valued
Summary

That’s the end of the introduction so on with the course!
UML principles
What is UML?

- Unified Modelling Language (UML) is a general purpose visual modelling language
  - Can support all existing lifecycles
  - Intended to be supported by CASE tools
- Unifies past modelling techniques and experience
- Incorporates current best practice in software engineering
- UML is *not* a methodology!
  - UML is a visual language
  - UP is a methodology
A major upgrade to UML at the end of 2003:
- Greater consistency
- More precisely defined semantics
- New diagram types
- Backwards compatible
UML future?

- The future of development of UML will be increasingly affected by Model Driven Architecture (MDA)
Why "unified"?

UML is unified across several domains:

- Across historical methods and notations
- Across the development lifecycle
- Across application domains
- Across implementation languages and platforms
- Across development processes
- Across internal concepts
Objects and the UML

- UML models systems as collections of objects that interact to deliver benefit to outside users

- Static structure
  - What kinds of objects are important
  - What are their relationships

- Dynamic behaviour
  - Lifecycles of objects
  - Object interactions to achieve goals
In this section we present an overview of the structure of the UML.

All the modelling elements mentioned here are discussed later, and in much more detail!

- Building blocks
- Common mechanisms
- Architecture
UML building blocks

- Things
  - Modelling elements
- Relationships
  - Tie things together
- Diagrams
  - Views showing interesting collections of things
  - Are views of the model
Things

- Structural things – nouns of a UML model
  - Class, interface, collaboration, use case, active class, component, node

- Behavioural things – verbs of a UML model
  - Interactions, state machine

- Grouping things
  - Package
    - Models, frameworks, subsystems

- Annotational things
  - Notes
  - Tagged values
## Relationships

<table>
<thead>
<tr>
<th>relationship</th>
<th>UML syntax</th>
<th>brief semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>dependency</td>
<td>---</td>
<td>The source element depends on the target element and may be affected by changes to it.</td>
</tr>
<tr>
<td>association</td>
<td>_______</td>
<td>The description of a set of links between objects.</td>
</tr>
<tr>
<td>aggregation</td>
<td>◆</td>
<td>The target element is a part of the source element.</td>
</tr>
<tr>
<td>composition</td>
<td>←</td>
<td>A strong (more constrained) form of aggregation.</td>
</tr>
<tr>
<td>containment</td>
<td>⊆</td>
<td>The source element contains the target element.</td>
</tr>
<tr>
<td>generalization</td>
<td>→</td>
<td>The source element is a specialization of the more general target element and may be substituted for it.</td>
</tr>
<tr>
<td>realization</td>
<td>---</td>
<td>The source element guarantees to carry out the contract specified by the target element</td>
</tr>
</tbody>
</table>
UML has 13 types of diagram

- Structure diagrams model the structure of the system (the static model)
- Behavior diagrams model the dynamic behavior of the system (the dynamic model)
- Each type of diagram gives a different type of view of the model
The heading specifies the kind of diagram, its name and any information (parameters) needed by elements in the diagram.

- The frame may be implied by a diagram area in the UML tool.
UML common mechanisms

- UML has four common mechanisms that apply consistently throughout the language:
  - Specifications
  - Adornments
  - Common divisions
  - Extensibility mechanisms
Behind every UML modelling element is a *specification* which provides a textual statement of the syntax and semantics of that element.

These specifications form the *semantic backplane* of the model.
Adornments

- Every UML modelling element starts with a basic symbol to which can be added a number of *adornments* specific to that symbol.

- We only show adornments to *increase the clarity* of the diagram or to highlight a specific feature of the model.

### Window

- **author = Jim, status = tested**
- `+size : Area=(100,100)`
- `#visibility : Boolean = false`
- `+defaultSize: Rectangle`
- `#maximumSize : Rectangle`
- `-xptr : XWindow`
- `+create()`
- `+hide()`
- `+display( location : Point )`
- `-attachXWindow( xwin : XWindow*)`
Common divisions

- Classifier and instance
  - A classifier is an abstraction, an instance is a concrete manifestation of that abstraction
  - The most common form is class/object e.g. a classifier might be a BankAccount class, and an instance might be an object representing my bank account
  - Generally instances have the same notation as classes, but the instance name is underlined

- Interface and implementation
  - An interface declares a contract and an implementation represents a concrete realization of that contract
Extensibility mechanisms

- Stereotypes
  - A stereotype allows us to define a new UML modelling element based on an existing one
  - We define the semantics of the stereotype ourselves
  - Stereotypes add new elements to the UML metamodel
  - Written as «stereotypeName»

- Constraints
  - Extends the semantics of an element by allowing us to add new rules about the element
  - Written as { some constraint }

- Tagged values
  - Allows us to add new, ad-hoc information to an element’s specification
  - Written as { tag1 = value1, tag2 = value2 … }
Stereotype syntax options

- A stereotype introduces a new modelling element and so we must always define semantics for our stereotypes.
- Each model element can have many stereotypes.

- Stereotype name in guillemets
  - stereotype name in guillemets: "Ticket"
  - icon: preferred

- Stereotype icon
  - icon: preferred
  - stereotype icon: Ticket

- Stereotype name and icon
  - stereotype name and icon: "Ticket"
  - icon: preferred

- Stereotyped relationship
  - stereotype relationship: «control» JobManager «call» Scheduler
UML profiles

- A profile customizes UML for a specific purposes

- A UML profile is a collection of stereotypes, tagged values and constraints
  - The tagged values and constraints are associated with stereotypes

- Stereotypes extend one of the UML meta-model elements (e.g. Class, Association)
  - Any element that gets the stereotype also gets the associated tagged values and constraints
Architecture

"The organisational structure of a software system"

- UML specification & IEEE Std. 610.12-1990
- RUP has a 4+1 view of architecture

Summary

The UML is composed of building blocks:
- Things
- Relationships
- Diagrams

The UML has four common mechanisms:
- Specifications
- Adornments
- Common divisions
- Extensibility mechanisms

The UML is based on a 4+1 view of system architecture
Introduction to the Unified Process
The Unified Process (UP)

- The Unified Software Development Process is an industry standard software engineering process
  - It is commonly referred to as the "Unified Process" or UP
  - It is the generic process for the UML
  - It is free - described in "The Unified Software Development Process", ISBN:0201571692"

- UP is:
  - Use case (requirements) driven
  - Risk driven
  - Architecture centric
  - Iterative and incremental

- UP is a generic software engineering process. It has to be customised (instantiated) for your project
  - In house standards, document templates, tools, databases, lifecycle modifications, ...

- Rational Unified Process (RUP) is an instantiation of UP
  - RUP is a product marketed and owned by Rational Corporation
  - RUP also has to be instantiated for your project
UP history

- 1967: Jacobson working at Ericsson
- 1976: Jacobson establishes Objectory AB
- 1987: Rational acquires Objectory AB
- 1995: UML becomes an industry standard
- 1997: Rational Unified Process (RUP)
- 1998: Rational Unified Process
- 1999: 2001: Ongoing RUP development
- 2004:
Iterations

- Iterations are the key to the UP
- Each iteration is like a mini-project including:
  - Planning
  - Analysis and design
  - Integration and test
  - An internal or external release
- We arrive at a final product release through a sequence of iterations
- Iterations can overlap - this allows parallel development and flexible working in large teams
  - Requires careful planning
- Iterations are organised into phases

get early and continuous feedback
Iteration workflows

- Each iteration may contain all of the core workflows but with a different emphasis depending on where the iteration is in the lifecycle.

UP specifies 5 core workflows:

- Requirements
- Analysis
- Design
- Implementation
- Test

Other workflows:

- Planning
- Project specific...
- Assessment

An iteration
Baselines and increments

- Each iteration generates a baseline
- A baseline is a set of reviewed and approved artefacts that:
  - Provide an agreed basis for further review and development
  - Can be changed only through formal procedures such as configuration and change management
- An *increment* is the difference between the baseline generated by one iteration and the baseline generated by the next iteration
  - This is why the UP is called “iterative and incremental”
Each phase can include several iterations
- The exact number of iterations per phase depends on the size of the project!
  e.g. one iteration per phase for small projects
- Each phase concludes with a major milestone
Phases and Workflows

- This figure is the key to understanding UP!

- For each phase we will consider:
  - The focus in terms of the core workflows
  - The goal for the phase
  - The milestone at the end of the phase
### Inception

#### Focus

<table>
<thead>
<tr>
<th>Focus</th>
<th>Requirements - establish business case and scope. Capture core requirements</th>
<th>Analysis - establish feasibility</th>
<th>Design - design proof of concept or technical prototypes</th>
<th>Implementation - build proof of concept or technical prototype</th>
<th>Test - not generally applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception</td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
</tr>
<tr>
<td>Elaboration</td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
</tr>
<tr>
<td>Construction</td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
</tr>
<tr>
<td>Transition</td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
</tr>
</tbody>
</table>

#### Goals

- Establish feasibility of the project - create proof of concept/technical prototypes
- Create a business case
- Scope the system - capture key requirements
- Identify critical risks

---

© Clear View Training 2010 v2.6
Inception - milestone

- Life Cycle Objectives - conditions of satisfaction:
  - System scope has been defined
  - Key requirements for the system have been captured. These have been defined and agreed with the stakeholders
  - An architectural vision exists. This is just a sketch at this stage
  - A Risk Assessment
  - A Business Case
  - Project feasibility is confirmed
  - The stakeholders agree on the objectives of the project
Elaboration

<table>
<thead>
<tr>
<th>Focus</th>
<th>Requirements - refine system scope and requirements</th>
<th>Analysis - establish what to build</th>
<th>Design - create a stable architectural baseline</th>
<th>Implementation - build the architectural baseline</th>
<th>Test - test the architectural baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goals</td>
<td>Create an executable architectural baseline</td>
<td>Refine Risk Assessment and define quality attributes (defect rates etc.)</td>
<td>Capture use cases to 80% of the functional requirements</td>
<td>Create a plan with sufficient detail for the construction phase</td>
<td>Formulate a bid which includes resources, time, equipment, staff, cost</td>
</tr>
</tbody>
</table>

Inception | Elaboration | Construction | Transition
Elaboration - milestone

- Lifecycle Architecture - conditions of satisfaction:
  - A resilient, robust executable architectural baseline has been created
  - The Risk Assessment has been updated
  - A project plan has been created to enable a realistic bid to be formulated
  - The business case has been verified against the plan
  - The stakeholders agree to continue
# Construction

## Focus

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Inception</th>
<th>Elaboration</th>
<th>Construction</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>- uncover any requirements that had been missed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Inception</th>
<th>Elaboration</th>
<th>Construction</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>- finish the analysis model</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Design</th>
<th>Inception</th>
<th>Elaboration</th>
<th>Construction</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>- finish the design model</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Inception</th>
<th>Elaboration</th>
<th>Construction</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>- build the Initial Operational Capability</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test</th>
<th>Inception</th>
<th>Elaboration</th>
<th>Construction</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>- test the Initial Operational Capability</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Goals

- Complete use case identification, description and realization
- Finish analysis, design, implementation and test
- Maintain the integrity of the system architecture
- Revise the Risk Assessment
Construction - milestone

- Initial Operational Capability - conditions of satisfaction:
  - The product is ready for beta testing in the user environment
## Transition

<table>
<thead>
<tr>
<th>Focus</th>
<th>Requirements - not applicable</th>
<th>Analysis - not applicable</th>
<th>Design - modify the design if problems emerge in beta testing</th>
<th>Implementation - tailor the software for the user site. Fix bugs uncovered in beta testing</th>
<th>Test - perform beta testing and acceptance testing at the user site</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goals</strong></td>
<td>Correct defects</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Prepare the user site for the new software and tailor the software to operate at the user site</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Modify software if unforeseen problems arise</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Create user manuals and other documentation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provide customer consultancy</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Conduct post project review</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Transition – milestone

Product Release - conditions of satisfaction:
- Beta testing, acceptance testing and defect repair are finished
- The product is released into the user community
Summary

- UP is a risk and use case driven, architecture centric, iterative and incremental software development process.

- UP has four phases:
  - Inception
  - Elaboration
  - Construction
  - Transition

- Each iteration has five core workflows:
  - Requirements
  - Analysis
  - Design
  - Implementation
  - Test
Requirements - introduction
Requirements - purpose

- The purpose of the requirements workflow is to create a high-level specification of what should be implemented.

- We interview the stakeholders to find out what they need the system to do for them – their requirements.
Requirements - workflow

1. Find actors and use cases
2. Prioritise use cases
3. Detail a use case
4. Prototype user interface
5. Structure the use case model
In order to adopt a rigorous approach to requirements we need to extend the basic UP workflow with functional and non-functional requirements elicitation and requirements traceability.
The importance of requirements

Incomplete requirements are the primary reason that projects fail!

The Standish Group, "The CHAOS Report (1994)"
What are requirements?

- Requirements - “A specification of what should be implemented”:
  - What behaviour the system should offer
  - A specific property of the system
  - A constraint on the system

- In UP we create a Software Requirements Specification (SRS)
  - The beginning of the OO software construction process it is a statement of the system requirements for all stakeholders
  - Organises related requirements into sections

- The SRS consists of:
  - Requirements model comprising functional and non-functional requirements
  - Use case model comprising actors and use cases
Writing requirements

<id> The <system> **shall** <function>

unique identifier  name of system  keyword  function to be performed

- There is no UML standard way of writing requirements!
  - We recommend the uniform sentence structure above

- Functional Requirements - what the system should do
  - "The ATM system **shall** provide a facility for authenticating the identity of a system user"

- Non-functional Requirements - a constraint on how the functional requirements are implemented
  - "The ATM system **shall** authenticate a customer in four seconds or less"
The map is not the territory

- Everyone filters information to create their own particular model of the world. Noam Chomsky described this as three processes:
  - Deletion – information is filtered out
  - Distortion – information is modified by the related mechanisms of creation and hallucination
  - Generalisation – the creation of rules, beliefs and principles about truth and falsehood

- These filters shape natural language and so we may need to work to recover filtered information
Summary

- We have seen how to capture:
  - Functional requirements
  - Non-functional requirements
- We have had a brief overview of the three filters which people use to construct their model of the world
Requirements –
use case modelling
Use case modelling

Use case modelling is a form of requirements engineering.

Use case modelling proceeds as follows:

- Find the system boundary
- Find actors
- Find use cases
  - Use case specification
  - Scenarios

It lets us identify the system boundary, who or what uses the system, and what functions the system should offer.
Find actors and use cases

- Business model
  - or domain model
- System analyst
- Use case model
  - outlined
- Requirements model
- Find actors and use cases
- Feature list
- Project glossary
Before we can build anything, we need to know:

- Where the boundary of the system lies
- Who or what uses the system
- What functions the system should offer to its users

We create a Use Case model containing:

- Subject – the edge of the system
  - also known as the system boundary
- Actors – who or what uses the system
- Use Cases – things actors do with the system
- Relationships - between actors and use cases
What are actors?

- An actor is anything that interacts directly with the system
  - Actors identify who or what uses the system and so indicate where the system boundary lies
- Actors are external to the system
- An Actor specifies a role that some external entity adopts when interacting with the system

[Diagram: Stick figure labeled 'Customer']
Identifying Actors

When identifying actors ask:

- Who or what uses the system?
- What roles do they play in the interaction?
- Who installs the system?
- Who starts and shuts down the system?
- Who maintains the system?
- What other systems use this system?
- Who gets and provides information to the system?
- Does anything happen at a fixed time?
**What are use cases?**

- A use case is something an actor needs the system to do. It is a “case of use” of the system by a specific actor.

- Use cases are *always* started by an actor:
  - The *primary actor* triggers the use case.
  - Zero or more *secondary actors* interact with the use case in some way.

- Use cases are *always* written from the point of view of the actors.

---

PlaceOrder

GetStatusOnOrder
Identifying use cases

- Start with the list of actors that interact with the system.
- When identifying use cases ask:
  - What functions will a specific actor want from the system?
  - Does the system store and retrieve information? If so, which actors trigger this behaviour?
  - What happens when the system changes state (e.g. system start and stop)? Are any actors notified?
  - Are there any external events that affect the system? What notifies the system about those events?
  - Does the system interact with any external system?
  - Does the system generate any reports?
The use case diagram

Mail Order System use case diagram

Mail Order System

- PlaceOrder
- CancelOrder
- CheckOrderStatus
- SendCatalogue
- ShipProduct

Communication relationship

Actor: Customer

Subject name: Mail Order System

System boundary

Actors:
- Customer
- ShippingCompany
- Dispatcher
In any business domain there is always a certain amount of jargon. It’s important to capture the language of the domain in a project glossary.

The aim of the glossary is to define key terms and to resolve synonyms and homonyms.

You are building a vocabulary that you can use to discuss the system with the stakeholders.
Detail a use case

- Use case model [outlined]
- Supplementary requirements
- Glossary

Use case specifier

Detail a use case

Use case [detailed]

© Clear View Training 2010 v2.6
Use case specification

<table>
<thead>
<tr>
<th>Use case name</th>
<th>Use case: PaySalesTax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID: 1</td>
<td></td>
</tr>
<tr>
<td>Brief description:</td>
<td></td>
</tr>
<tr>
<td>Pay Sales Tax to the Tax Authority at the end of the business quarter.</td>
<td></td>
</tr>
<tr>
<td>Primary actors:</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td></td>
</tr>
<tr>
<td>Secondary actors:</td>
<td></td>
</tr>
<tr>
<td>TaxAuthority</td>
<td></td>
</tr>
<tr>
<td>Preconditions:</td>
<td></td>
</tr>
<tr>
<td>1. It is the end of the business quarter.</td>
<td></td>
</tr>
<tr>
<td>Main flow:</td>
<td></td>
</tr>
<tr>
<td>1. The use case starts when it is the end of the business quarter.</td>
<td></td>
</tr>
<tr>
<td>2. The system determines the amount of Sales Tax owed to the Tax Authority.</td>
<td></td>
</tr>
<tr>
<td>3. The system sends an electronic payment to the Tax Authority.</td>
<td></td>
</tr>
<tr>
<td>Postconditions:</td>
<td></td>
</tr>
<tr>
<td>1. The Tax Authority receives the correct amount of Sales Tax.</td>
<td></td>
</tr>
<tr>
<td>Alternative flows:</td>
<td></td>
</tr>
<tr>
<td>None.</td>
<td></td>
</tr>
</tbody>
</table>
Naming use cases

- Use cases describe something that happens
- They are named using verbs or verb phrases
- Naming standard 1: use cases are named using UpperCamelCase e.g. PaySalesTax

1 UML 2 does not specify any naming standards. All naming standards are our own, based on industry best practice.
Pre and postconditions

- Preconditions and postconditions are *constraints*
- Preconditions constrain the state of the system *before* the use case can start
- Postconditions constrain the state of the system *after* the use case has executed
- If there are no preconditions or postconditions write "None" under the heading

Use case: PlaceOrder

**Preconditions:**
1. A valid user has logged on to the system

**Postconditions:**
1. The order has been marked confirmed and is saved by the system
Main flow

The flow of events lists the steps in a use case
- It always begins by an actor doing something
  - A good way to start a flow of events is:
    1) The use case starts when an <actor> <function>
- The flow of events should be a sequence of short steps that are:
  - Declarative
  - Numbered,
  - Time ordered
- The main flow is always the happy day or perfect world scenario
  - Everything goes as expected and desired, and there are no errors, deviations, interrupts, or branches
  - Alternatives can be shown by branching or by listing under Alternative flows (see later)
**Branching within a flow: If**

- Use the keyword **if** to indicate alternatives within the flow of events
  - There must be a Boolean expression immediately after **if**
- Use indentation and numbering to indicate the conditional part of the flow
- Use **else** to indicate what happens if the condition is false (see next slide)

---

**Use case: ManageBasket**

<table>
<thead>
<tr>
<th>ID: 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Brief description:</strong> The Customer changes the quantity of an item in the basket.</td>
</tr>
<tr>
<td><strong>Primary actors:</strong> Customer</td>
</tr>
<tr>
<td><strong>Secondary actors:</strong> None.</td>
</tr>
<tr>
<td><strong>Preconditions:</strong> 1. The shopping basket contents are visible.</td>
</tr>
<tr>
<td><strong>Main flow:</strong> 1. The use case starts when the Customer selects an item in the basket. 2. <strong>If</strong> the Customer selects &quot;delete item&quot; 2.1 The system removes the item from the basket. 3. <strong>If</strong> the Customer types in a new quantity 3.1 The system updates the quantity of the item in the basket.</td>
</tr>
<tr>
<td><strong>Postconditions:</strong> None.</td>
</tr>
<tr>
<td><strong>Alternative flows:</strong> None.</td>
</tr>
</tbody>
</table>
Repetition within a flow: For

- We can use the keyword `For` to indicate the start of a repetition within the flow of events.
- The iteration expression immediately after the `For` statement indicates the number of repetitions of the indented text beneath the `For` statement.

<table>
<thead>
<tr>
<th>Use case: FindProduct</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ID:</strong> 3</td>
</tr>
<tr>
<td><strong>Brief description:</strong> The system finds some products based on Customer search criteria and displays them to the Customer.</td>
</tr>
<tr>
<td><strong>Actors:</strong> Customer</td>
</tr>
<tr>
<td><strong>Preconditions:</strong> None.</td>
</tr>
<tr>
<td><strong>Main flow:</strong></td>
</tr>
<tr>
<td>1. The use case starts when the Customer selects &quot;find product&quot;.</td>
</tr>
<tr>
<td>2. The system asks the Customer for search criteria.</td>
</tr>
<tr>
<td>3. The Customer enters the requested criteria.</td>
</tr>
<tr>
<td>4. The system searches for products that match the Customer's criteria.</td>
</tr>
<tr>
<td>5. For each product found</td>
</tr>
<tr>
<td>5.1. The system displays a thumbnail sketch of the product.</td>
</tr>
<tr>
<td>5.2. The system displays a summary of the product details.</td>
</tr>
<tr>
<td>5.3. The system displays the product price.</td>
</tr>
<tr>
<td><strong>Postconditions:</strong> None.</td>
</tr>
<tr>
<td><strong>Alternative flows:</strong> NoProductsFound</td>
</tr>
</tbody>
</table>
Repetition within a flow: While

We can use the keyword **while** to indicate that something repeats while some Boolean condition is true.

<table>
<thead>
<tr>
<th>Use case: ShowCompanyDetails</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID: 4</td>
</tr>
<tr>
<td>Brief description:</td>
</tr>
<tr>
<td>The system displays the company details to the Customer.</td>
</tr>
<tr>
<td>Primary actors:</td>
</tr>
<tr>
<td>Customer</td>
</tr>
<tr>
<td>Secondary actors:</td>
</tr>
<tr>
<td>None</td>
</tr>
<tr>
<td>Preconditions:</td>
</tr>
<tr>
<td>None.</td>
</tr>
<tr>
<td>Main flow:</td>
</tr>
<tr>
<td>1. The use case starts when the Customer selects &quot;show company details&quot;.</td>
</tr>
<tr>
<td>2. The system displays a web page showing the company details.</td>
</tr>
<tr>
<td>3. <strong>While</strong> the Customer is browsing the company details</td>
</tr>
<tr>
<td>4. The system searches for products that match the Customer's criteria.</td>
</tr>
<tr>
<td>4.1. The system plays some background music.</td>
</tr>
<tr>
<td>4.2. The system displays special offers in a banner ad.</td>
</tr>
<tr>
<td>Postconditions:</td>
</tr>
<tr>
<td>1. The system has displayed the company details.</td>
</tr>
<tr>
<td>2. The system has played some background music.</td>
</tr>
<tr>
<td>3. The systems has displayed special offers.</td>
</tr>
<tr>
<td>Alternative flows:</td>
</tr>
<tr>
<td>None.</td>
</tr>
</tbody>
</table>
Branching: Alternative flows

- We may specify one or more alternative flows through the flow of events:
  - Alternative flows capture errors, branches, and interrupts
  - Alternative flows *never* return to the main flow
- Potentially very many alternative flows! You need to manage this:
  - Pick the most important alternative flows and document those.
  - If there are groups of similar alternative flows - document one member of the group as an exemplar and (if necessary) add notes to this explaining how the others differ from it.

Only document enough alternative flows to clarify the requirements!
Referencing alternative flows

- List the names of the alternative flows at the end of the use case
- Find alternative flows by examining each step in the main flow and looking for:
  - Alternatives
  - Exceptions
  - Interrupts

<table>
<thead>
<tr>
<th>Use case: CreateNewCustomerAccount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID: 5</td>
</tr>
<tr>
<td>Brief description:</td>
</tr>
<tr>
<td>The system creates a new account for the Customer.</td>
</tr>
<tr>
<td>Primary actors:</td>
</tr>
<tr>
<td>Customer</td>
</tr>
<tr>
<td>Secondary actors:</td>
</tr>
<tr>
<td>None.</td>
</tr>
<tr>
<td>Preconditions:</td>
</tr>
<tr>
<td>None.</td>
</tr>
<tr>
<td>Main flow:</td>
</tr>
<tr>
<td>1. The use case begins when the Customer selects &quot;create new customer account&quot;.</td>
</tr>
<tr>
<td>2. While the Customer details are invalid</td>
</tr>
<tr>
<td>2.1. The system asks the Customer to enter his or her details comprising email address, password and password again for confirmation.</td>
</tr>
<tr>
<td>2.2. The system validates the Customer details.</td>
</tr>
<tr>
<td>3. The system creates a new account for the Customer.</td>
</tr>
<tr>
<td>Postconditions:</td>
</tr>
<tr>
<td>1. A new account has been created for the Customer.</td>
</tr>
<tr>
<td>Alternative flows:</td>
</tr>
<tr>
<td>InvalidEmailAddress</td>
</tr>
<tr>
<td>InvalidPassword</td>
</tr>
<tr>
<td>Cancel</td>
</tr>
</tbody>
</table>
An alternative flow example

- The alternative flow may be triggered *instead* of the main flow - started by an actor
- The alternative flow may be triggered *after a particular step* in the main flow - *after*
- The alternative flow may be triggered *at any time* during the main flow - *at any time*

### Alternative flow: CreateNewCustomerAccount:InvalidEmailAddress

<table>
<thead>
<tr>
<th>ID: 5.1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Brief description:</strong> The system informs the Customer that they have entered an invalid email address.</td>
</tr>
<tr>
<td><strong>Primary actors:</strong> Customer</td>
</tr>
<tr>
<td><strong>Secondary actors:</strong> None.</td>
</tr>
<tr>
<td><strong>Preconditions:</strong> 1. The Customer has entered an invalid email address</td>
</tr>
<tr>
<td><strong>Alternative flow:</strong> 1. The alternative flow begins after step 2.2. of the main flow. 2. The system informs the Customer that he or she entered an invalid email address.</td>
</tr>
<tr>
<td><strong>Postconditions:</strong> None.</td>
</tr>
</tbody>
</table>

Notice how we name and number alternative flows. Always indicate how the alternative flow begins. In this case it starts after step 2.2 in the main flow.
Requirements tracing

- Given that we can capture functional requirements in a requirements model and in a use case model we need some way of relating the two.
- There is a many-to-many relationship between requirements and use cases:
  - One use case covers many individual functional requirements.
  - One functional requirement may be realised by many use cases.
- Hopefully we have CASE support for requirements tracing:
  - With UML tagged values, we can assign numbered requirements to use cases.
  - We can capture use case names in our Requirements Database.
- If there is no CASE support, we can create a Requirements Traceability matrix.

Table: Requirements Traceability Matrix

<table>
<thead>
<tr>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
</tr>
<tr>
<td>U1</td>
</tr>
<tr>
<td>U2</td>
</tr>
<tr>
<td>U3</td>
</tr>
<tr>
<td>U4</td>
</tr>
</tbody>
</table>
When to use use case analysis

- Use cases describe system behaviour from the point of view of one or more actors. They are the *best* choice when:
  - The system is dominated by functional requirements
  - The system has many types of user to which it delivers different functionality
  - The system has many interfaces

- Use cases are designed to capture *functional* requirements. They are a *poor* choice when:
  - The system is dominated by non-functional requirements
  - The system has few users
  - The system has few interfaces
Summary

- We have seen how to capture functional requirements with use cases.
- We have looked at:
  - Use cases
  - Actors
  - Branching with if
  - Repetition with for and while
  - Alternative flows
  - Requirements tracing
Requirements – advanced use case modelling
More relationships...

We have studied basic use case analysis, but there are relationships that we have still to explore:

- Actor generalisation
- Use case generalisation
- «include» – between use cases
- «extend» – between use cases
Actor generalization - example

- The Customer and the Sales Agent actors are very similar.
- They both interact with List products, Order products, Accept payment.
- Additionally, the Sales Agent interacts with Calculate commission.
- Our diagram is a mess - can we simplify it?
Actor generalisation

- If two actors communicate with the same set of use cases in the same way, then we can express this as a generalisation to another (possibly abstract) actor.
- The descendent actors inherit the roles and relationships to use cases held by the ancestor actor.
- We can substitute a descendent actor anywhere the ancestor actor is expected. This is the *substitutability principle*.

Use actor generalization when it simplifies the model.
Use case generalisation

- The ancestor use case must be a more general case of one or more descendant use cases.
- Child use cases are more specific forms of their parent.
- They can inherit, add and override features of their parent.

<table>
<thead>
<tr>
<th>Use case generalization semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case element</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>Relationship</td>
</tr>
<tr>
<td>Extension point</td>
</tr>
<tr>
<td>Precondition</td>
</tr>
<tr>
<td>Postcondition</td>
</tr>
<tr>
<td>Step in main flow</td>
</tr>
<tr>
<td>Alternative flow</td>
</tr>
</tbody>
</table>

Sales system

- **FindProduct**
  - **FindBook**
  - **FindCD**

Customer
The base use case executes until the point of inclusion:

```java
include(InclusionUseCase)
```

- Control passes to the inclusion use case which executes.
- When the inclusion use case is finished, control passes back to the base use case which finishes execution.

**Note:**

- Base use cases are *not complete* without the included use cases.
- Inclusion use cases may be complete use cases, or they may just specify a fragment of behaviour for inclusion elsewhere.

When use cases share common behaviour we can factor this out into a separate inclusion use case and «include» it in base use cases.
«include» example

### Use case: ChangeEmployeeDetails

<table>
<thead>
<tr>
<th>ID: 1</th>
</tr>
</thead>
</table>

Brief description: The Manager changes the employee details.

Primary actors: Manager

Secondary actors: None

Preconditions:
1. The Manager is logged on to the system.

Main flow:
1. include(FindEmployeeDetails).
2. The system displays the employee details.
3. The Manager changes the employee details.

Postconditions:
1. The employee details have been changed.

Alternative flows: None.

### Use case: FindEmployeeDetails

<table>
<thead>
<tr>
<th>ID: 4</th>
</tr>
</thead>
</table>

Brief description: The Manager finds the employee details.

Primary actors: Manager

Secondary actors: None

Preconditions:
1. The Manager is logged on to the system.

Main flow:
1. The Manager enters the employee's ID.
2. The system finds the employee details.

Postconditions:
1. The system has found the employee details.

Alternative flows: None.
«extend»

- «extend» is a way of adding new behaviour into the base use case by inserting behaviour from one or more extension use cases
  - The base use case specifies one or more extension points in its flow of events
- The extension use case may contain several insertion segments
- The «extend» relationship may specify which of the base use case extension points it is extending

The extension use case inserts behaviour into the base use case. The base use case provides extension points, but does not know about the extensions.
There is an extension point **overdueBook** just before step 4 of the flow of events.

Extension points are *not* numbered, as they are *not* part of the flow.
Extension use cases have one or more *insertion segments* which are behaviour fragments that will be inserted at the specified extension points in the base use case.
Multiple insertion points

If more than one extension point is specified in the «extend» relationship then the extension use case must have the same number of insertion segments.

ReturnBook

extension points

overdueBook

payFine

«extend»

(overdueBook, payFine)

IssueFine

Extension Use case: IssueFine

ID: 10

Brief description:
Segment 1: The Librarian records and prints out a fine.
Segment 2: The Librarian accepts payment for a fine.

Primary actors:
Librarian

Secondary actors:
None.

Segment 1 preconditions:
1. The returned book is overdue.

Segment 1 flow:
1. The Librarian enters details of the fine into the system.
2. The system prints out the fine.

Segment 1 postconditions:
1. The fine has been recorded in the system.
2. The system has printed out the fine.

Segment 2 preconditions:
1. A fine is due from the borrower.

Segment 2 flow:
1. The Librarian accepts payment for the fine from the borrower.
2. The Librarian enters the paid fine in the system.
3. The system prints out a receipt for the paid fine.

Segment 2 postconditions:
1. The fine is recorded as paid.
2. The system has printed a receipt for the fine.
Conditional extensions

- We can specify conditions on «extend» relationships
  - Conditions are Boolean expressions
  - The insertion is made if and only if the condition evaluates to true
We have learned about techniques for advanced use case modelling:

- Actor generalisation
- Use case generalisation
- «include»
- «extend»

Use advanced features with discretion only where they simplify the model!
Analysis - introduction
Analysis - purpose

- Produce an Analysis Model of the system’s desired behaviour:
  - This model should be a statement of what the system does not how it does it
  - We can think of the analysis model as a “first-cut” or “high level” design model
  - It is in the language of the business

- In the Analysis Model we identify:
  - Analysis classes
  - Use-case realizations

![Inception Elaboration Construction Transition Diagram]
Packages contain UML modelling elements and diagrams (we only show the elements here).

Each element or diagram is owned by exactly one package.
Workflow - Analysis

Analysis guidelines:
- 50 to 100 classes in the analysis model of a moderately complex system
- Only include classes which are part of the vocabulary of the problem domain
- Don’t worry about classes which define how something is implemented – we will address these in Design
- Focus on classes and associations
- Don’t worry about class inheritance too much
- Keep it simple!!!

Architectural analysis

Analyze a use case

Component Engineer

Use Case Engineer

Architect

Analyze a class

Analyze a package

© Clear View Training 2010 v2.6
Analysis - objects and classes
What are objects?

- Objects consist of data and function packaged together in a reusable unit. Objects *encapsulate* data.
- Every object is an instance of some *class* which defines the common set of *features* (attributes and operations) shared by all of its instances. Objects have:
  - Attribute values – the data part
  - Operations – the behaviour part
- All objects have:
  - *Identity*: Each object has its own unique identity and can be accessed by a unique handle
  - *State*: This is the actual data values stored in an object at any point in time
  - *Behaviour*: The set of operations that an object can perform
Encapsulation

- Data is hidden inside the object. The only way to access the data is via one of the operations.
- This is *encapsulation* or *data hiding* and it is a very powerful idea. It leads to more robust software and reusable code.
In OO systems, objects send messages to each other over links.

These messages cause an object to invoke an operation.

- the Bank object sends the message "withdraw 150.00" to an Account object.

- the Account object responds by invoking its withdraw operation. This operation decrements the account balance by 150.00.
**UML Object Syntax**

- All objects of a particular class have the same set of operations. They are not shown on the object diagram, they are shown on the class diagram (see later).
- Attribute types are often omitted to simplify the diagram.
- Naming:
  - object and attribute names in lowerCamelCase
  - class names in UpperCamelCase

### Examples

<table>
<thead>
<tr>
<th>Object Identifier</th>
<th>Object Name</th>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>jimsAccount</code></td>
<td>: Account</td>
<td></td>
</tr>
</tbody>
</table>

- `accountNumber : String = "1234567"
- `owner : String = "Jim Arlow"
- `balance : double = 300.00

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

- An anonymous object (must be underlined)

© Clear View Training 2010 v2.6
What are classes?

- Every object is an instance of one class - the class describes the "type" of the object
- Classes allow us to model sets of objects that have the same set of features - a class acts as a template for objects:
  - The class determines the structure (set of features) of all objects of that class
  - All objects of a class must have the same set of operations, must have the same attributes, but may have different attribute values
- Classification is one of the most important ways we have of organising our view of the world
- Think of classes as being like:
  - Rubber stamps
  - Cookie cutters
Exercise - how many classes?

7.4
Classes and objects

- Objects are instances of classes
- Instantiation is the creation of new instances of model elements
- Most classes provide special operations called *constructors* to create instances of that class. These operations have class-scope i.e. they belong to the class itself rather than to objects of the class
- We will see instantiation used with other modelling elements later on
- Classes are named in UpperCamelCase
- Use descriptive names that are nouns or noun phrases
- Avoid abbreviations!
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
Visibility

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>public</td>
<td>Any element that can access the class can access any of its features with public visibility</td>
</tr>
<tr>
<td>-</td>
<td>private</td>
<td>Only operations within the class can access features with private visibility</td>
</tr>
<tr>
<td>#</td>
<td>protected</td>
<td>Only operations within the class, or within children of the class, can access features with protected visibility</td>
</tr>
<tr>
<td>~</td>
<td>package</td>
<td>Any element that is in the same package as the class, or in a nested subpackage, can access any of its features with package visibility</td>
</tr>
</tbody>
</table>

PersonDetails

- name : String [2..*]
- address : String [3]
- emailAddress : String [0..1]
## Multiplicity

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

<table>
<thead>
<tr>
<th>PersonDetails</th>
</tr>
</thead>
<tbody>
<tr>
<td>- name : String [2..*]</td>
</tr>
<tr>
<td>- address : String [3]</td>
</tr>
<tr>
<td>- emailAddress : String [0..1]</td>
</tr>
</tbody>
</table>

- Name is composed of 2 or more Strings.
- Address is composed of 3 Strings.
- EmailAddress is composed of 1 String or null.

Multiplicty expression

© Clear View Training 2010 v2.6
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
Parameter direction

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

example of multiple return values:

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

There are two kinds of scope for attributes and operations:

<table>
<thead>
<tr>
<th>BankAccount</th>
</tr>
</thead>
<tbody>
<tr>
<td>-accountNumber : int</td>
</tr>
<tr>
<td>-count : int = 0</td>
</tr>
<tr>
<td>+create( aNumber : int)</td>
</tr>
<tr>
<td>+getNumber() : int</td>
</tr>
<tr>
<td>-incrementCount()</td>
</tr>
<tr>
<td>+getCount() : int</td>
</tr>
</tbody>
</table>

class scope (underlined)

instance scope (the default)
### Instance scope vs. class scope

<table>
<thead>
<tr>
<th>instance scope</th>
<th>class scope</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>attributes</strong></td>
<td></td>
</tr>
<tr>
<td>By default, attributes have instance scope</td>
<td>Attributes may be defined as class scope</td>
</tr>
<tr>
<td>Every object of the class gets its <em>own copy</em> of the instance scope attributes</td>
<td>Every object of the class <em>shares the same, single</em> copy of the class scope attributes</td>
</tr>
<tr>
<td>Each object may therefore have <em>different</em> instance scope attribute values</td>
<td>Each object will therefore have the <em>same</em> class scope attribute values</td>
</tr>
<tr>
<td><strong>operations</strong></td>
<td></td>
</tr>
<tr>
<td>By default, operations have instance scope</td>
<td>Operations may be defined as class scope</td>
</tr>
<tr>
<td>Every invocation of an instance scope operation applies to a specific instance of the class</td>
<td>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</td>
</tr>
<tr>
<td>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</td>
<td>You can invoke a class scope operation even if there is no instance of the class available - this is ideal for object creation operations</td>
</tr>
</tbody>
</table>

**Scope determines access**
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.

<table>
<thead>
<tr>
<th>BankAccount</th>
<th>BankAccount</th>
</tr>
</thead>
<tbody>
<tr>
<td>+create(aNumber : int)</td>
<td>+BankAccount(aNumber : int)</td>
</tr>
</tbody>
</table>

generic constructor name | Java/C++ standard
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?

<table>
<thead>
<tr>
<th>ClubMember</th>
</tr>
</thead>
<tbody>
<tr>
<td>- membershipNumber : String</td>
</tr>
<tr>
<td>- memberName : String</td>
</tr>
<tr>
<td>- numberOfMembers : int = 0</td>
</tr>
<tr>
<td>+ create( number : String, name : String )</td>
</tr>
<tr>
<td>+ getMembershipNumber() : String</td>
</tr>
<tr>
<td>+ getMemberName() : String</td>
</tr>
<tr>
<td>- incrementNumberOfMembers()</td>
</tr>
<tr>
<td>+ decrementNumberOfMembers()</td>
</tr>
<tr>
<td>+ getNumberOfMembers() : int</td>
</tr>
</tbody>
</table>
Summary

- We have looked at objects and classes and examined the relationship between them.
- We have explored the UML syntax for modelling classes including:
  - Attributes
  - Operations
- We have seen that scope controls access:
  - Attributes and operations are normally instance scope
  - We can use class scope operations for constructor and destructors
  - Class scope attributes are shared by all objects of the class and are useful as counters
Analysis - finding analysis classes
8.2

Analyse a use case

- Business model [or domain model]
- Requirements model
- Use case model
- Architecture description
- Use case engineer
- Analysis class
- Use case realization

© Clear View Training 2010 v2.6
What are Analysis classes?

- Analysis classes represent a crisp abstraction in the problem domain
  - They may ultimately be refined into one or more design classes
- All classes in the Analysis model should be Analysis classes
- Analysis classes have:
  - A very “high level” set of attributes. They *indicate* the attributes that the design classes *might* have.
  - Operations that specify at a high level the key services that the class must offer. In Design, they will become actual, implementable, operations.
- Analysis classes must map onto real-world business concepts

<table>
<thead>
<tr>
<th>BankAccount</th>
</tr>
</thead>
<tbody>
<tr>
<td>class name</td>
</tr>
<tr>
<td>attributes</td>
</tr>
<tr>
<td>deposit()</td>
</tr>
<tr>
<td>withdraw()</td>
</tr>
<tr>
<td>balance : double</td>
</tr>
<tr>
<td>calculateInterest()</td>
</tr>
</tbody>
</table>

We always specify attribute types if we know what they are!
What makes a good analysis class?

- Its name reflects its intent
- It is a *crisp abstraction* that models one specific element of the problem domain
  - It maps onto a clearly identifiable feature of the problem domain
- It has *high cohesion*
  - Cohesion is the degree to which a class models a single abstraction
  - Cohesion is the degree to which the *responsibilities* of the class are semantically related
- It has *low coupling*
  - Coupling is the degree to which one class depends on others
- Rules of thumb:
  - 3 to 5 responsibilities per class
  - Each class collaborates with others
  - Beware many very small classes
  - Beware few but very large classes
  - Beware of “functoids”
  - Beware of “omnipotent” classes
  - Avoid deep inheritance trees

A *responsibility* is a contract or obligation of a class - it resolves into operations and attributes
Finding classes

- Perform noun/verb analysis on documents:
  - Nouns are candidate classes
  - Verbs are candidate responsibilities

- Perform CRC card analysis
  - A brainstorming technique using sticky notes
  - Useful for brainstorming, Joint Application Development (JAD) and Rapid Application development (RAD)

- With both techniques, beware of spurious classes:
  - Look for *synonyms* - different words that mean the same
  - Look for *homonyms* - the same word meaning different things

- Look for "hidden" classes!
  - Classes that don't appear as nouns or as cards
Noun/verb analysis procedure

- Collect all of the relevant documentation
  - Requirements document
  - Use cases
  - Project Glossary
  - Anything else!

- Make a list of nouns and noun phrases
  - These are candidate classes or attributes

- Make a list of verbs and verb phrases
  - These are candidate responsibilities

- Tentatively assign attributes and responsibilities to classes
CRC card procedure

- Class Name: BankAccount

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintain balance</td>
<td>Bank</td>
</tr>
</tbody>
</table>

- Class, Responsibilities and Collaborators
- Separate information collection from information analysis
  - Part 1: Brainstorm
    - All ideas are good ideas in CRC analysis
    - Never argue about something - write it down and analyse it later!
  - Part 2: Analyse information - consolidate with noun/verb
Other sources of classes

- Physical objects
- Paperwork, forms etc.
  - Be careful with this one – if the existing business process is very poor, then the paperwork that supports it might be irrelevant
- Known interfaces to the outside world
- Conceptual entities that form a cohesive abstraction e.g. LoyaltyProgramme
Summary

- We’ve looked at what constitutes a well-formed analysis class
- We have looked at two analysis techniques for finding analysis classes:
  - Noun verb analysis of use cases, requirements, glossary and other relevant documentation
  - CRC analysis
Analysis - relationships
What is a relationship?

- A *relationship* is a connection between modelling elements.
- In this section we’ll look at:
  - *Links* between objects
  - *Associations* between classes
    - aggregation
    - composition
    - association classes
What is a link?

- Links are connections between objects
  - Think of a link as a telephone line connecting you and a friend. You can send messages back and forth using this link

- Links are the way that objects communicate
  - Objects send messages to each other via links
  - Messages invoke operations

- OO programming languages implement links as object references or pointers. These are unique handles that refer to specific objects
  - When an object has a reference to another object, we say that there is a *link* between the objects
Object diagrams

- Paths in UML diagrams (lines to you and me!) can be drawn as orthogonal, oblique or curved lines.

- We can combine paths into a tree if each path has the same properties.
What is an association?

- Associations are relationships between classes.
- Associations between classes indicate that there are links between objects of those classes.
- A link is an instantiation of an association just as an object is an instantiation of a class.
An association can have role names or an association name. It’s bad style to have both!

The black triangle indicates the direction in which the association name is read:

“A Company employs many Persons”
Multiplicity

- Multiplicity is a constraint that specifies the number of objects that can participate in a relationship at *any point in time*.

- If multiplicity is not explicitly stated in the model then it is undecided - *there is no default multiplicity*.

<table>
<thead>
<tr>
<th>multiplicity syntax: minimum..maximum</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1</td>
<td>zero or 1</td>
</tr>
<tr>
<td>1</td>
<td>exactly 1</td>
</tr>
<tr>
<td>0..*</td>
<td>zero or more</td>
</tr>
<tr>
<td>*</td>
<td>zero or more</td>
</tr>
<tr>
<td>1..*</td>
<td>1 or more</td>
</tr>
<tr>
<td>1..6</td>
<td>1 to 6</td>
</tr>
</tbody>
</table>
**Multiplicity exercise**

- **How many**
  - Employees can a Company have?
  - Employers can a Person have?
  - Owners can a BankAccount have?
  - Operators can a BankAccount have?
  - BankAccounts can a Person have?
  - BankAccounts can a Person operate?
Exercise

- Model a computer file system. Here are the minimal facts you need:
  - The basic unit of storage is the file
  - Files live in directories
  - Directories can contain other directories
- Use your own knowledge of a specific file system (e.g. Windows 95 or UNIX) to build a model

Hint: a class can have an association to itself!
Reflexive associations

Directed graph showing the relationship between directories and files.
Hierarchies and networks

<table>
<thead>
<tr>
<th>hierarchy</th>
<th>network</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0..*</td>
<td>0..*</td>
</tr>
<tr>
<td>0..1</td>
<td></td>
</tr>
<tr>
<td>a1:A</td>
<td>b1:B</td>
</tr>
<tr>
<td>b1:A</td>
<td>c1:B</td>
</tr>
<tr>
<td>c1:A</td>
<td>d1:B</td>
</tr>
<tr>
<td>d1:A</td>
<td>e1:B</td>
</tr>
<tr>
<td>e1:A</td>
<td>f1:B</td>
</tr>
<tr>
<td>f1:A</td>
<td>g1:B</td>
</tr>
<tr>
<td>g1:A</td>
<td></td>
</tr>
</tbody>
</table>

In a hierarchy, each object has zero or one object directly above it.

In an association network, each object has zero or many objects directly above it.
Navigability indicates that it is possible to traverse from an object of the *source* class to objects of the *target* class.

- Objects of the source class may reference objects of the target class using the role name.

Even if there is *no* navigability it might still be possible to traverse the relationship via some indirect means. However, the computational cost of the traversal might be very high.

An Order object stores a list of Products.

**Navigable**

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

A Product object does not store a list of Orders.

**Not navigable**

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

A to B is navigable, B to A is not navigable.

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

A to B is navigable, B to A is undefined.

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

A to B is undefined, B to A is undefined.
Navigability - standard practice

Strict UML 2 navigability can clutter diagrams so the UML standard suggests three possible modeling idioms:

1. Show navigability explicitly on diagrams with crosses and arrows
2. Omit all navigability from diagrams
3. Omit crosses from diagrams
   - bi-directional associations have no arrows
   - unidirectional associations have a single arrow
   - you can't show associations that are not navigable in either direction (not useful anyway!)

A to B is navigable
B to A is not navigable

A to B is navigable
B to A is navigable
Associations and attributes

If a navigable relationship has a role name, it is as though the source class has a pseudo-attribute whose attribute name is the role name and whose attribute type is the target class.

Objects of the source class can refer to objects of the target class using this pseudo-attribute.

Use associations when:
- The target class is an important part of the model.
- The target class is a class that you have designed yourself and which must be shown on the model.

Use attributes when:
- The target class is not an important part of the model, e.g., a primitive type such as number, string, etc.
- The target class is just an implementation detail such as a bought-in component or a library component, e.g., Java.util.Vector (from the Java standard libraries).
Association classes

Each Person object can work for many Company objects. Each Company object can employ many Person objects. When a Person object is employed by a Company object, the Person has a salary.

But where do we record the Person’s salary?

- Not on the Person class - there is a different salary for each employment
- Not on the Company class - different Person objects have different salaries
- The salary is a property of the employment relationship itself
  - every time a Person object is employed by a Company object, there is a salary
We model the association itself as an association class. One instance of this class exists for each link between a Person object and a Company object.

- Instances of the association class are links that have attributes and operations.
- Can only use association classes when there is one unique link between two specific objects. This is because the identity of links is determined exclusively by the identities of the objects on the ends of the link.

We can place the salary and any other attributes or operations which are really features of the association into this class.
Using association classes

If we use an association class, then a particular Person can have only one Job with a particular Company.

If, however, a particular Person can have multiple jobs with the same Company, then we must use a reified association.
Qualified associations

- Qualified associations reduce an n to many association to an n to 1 association by specifying a unique object (or group of objects) from the set.
- They are useful to show how we can look up or navigate to specific objects.
- Qualifiers usually refer to an attribute on the target class.

The combination (Club, memberId) specifies a unique target object.

In the diagram:
- Club has a 1:1 relationship with Member.
- Member has a one-to-many relationship with Club, with a qualifier member1d.
- The combination (Club, memberId) specifies a unique target object.
Summary

In this section we have looked at:

- Links – relationships between objects
- Associations – relationships between classes
  - role names
  - multiplicity
  - navigability
  - association classes
  - qualified associations
Analysis - dependencies
What is a dependency?

- A dependency is a relationship between two elements where a change to one element (the supplier) may affect or supply information needed by the other element (the client). In other words, the client depends in some way on the supplier.
  - Dependency is really a catch-all that is used to model several different types of relationship. We’ve already seen one type of dependency, the «instantiate» relationship.

- Three types of dependency:
  - Usage - the client uses some of the services made available by the supplier to implement its own behavior - this is the most commonly used type of dependency.
  - Abstraction - a shift in the level of abstraction. The supplier is more abstract than the client.
  - Permission - the supplier grants some sort of permission for the client to access its contents – this is a way for the supplier to control and limit access to its contents.
Usage dependencies

- «use» - the client makes use of the supplier to implement its behaviour
- «call» - the client operation invokes the supplier operation
- «parameter» - the supplier is a parameter of the client operation
- «send» - the client (an operation) sends the supplier (a signal) to some unspecified target
- «instantiate» - the client is an instance of the supplier
A «use» dependency is generated between class A and B when:

1) An operation of class A needs a parameter of class B
2) An operation of class A returns a value of class B
3) An operation of class A uses an object of class B somewhere in its implementation

A :: doSomething()
{
    B myB = new B();
    ...
}

foo( b : B )
bar() : B
doSomething()
Abstraction dependencies

- «trace» - the client and the supplier represent the same concept but at different points in development.

- «substitute» - the client may be substituted for the supplier at runtime. The client and supplier must realize a common contract. Use in environments that _don’t_ support specialization/generalization.

- «refine» - the client represents a fuller specification of the supplier.

- «derive» - the client may be derived from the supplier. The client is logically redundant, but may appear for implementation reasons.
This example shows three possible ways to express a «derive» dependency.
Permission dependencies

- **«access»**
  - The public contents of the supplier package are added as private elements to the namespace of the client package

- **«import»**
  - The public contents of the supplier package are added as public elements to the namespace of the client package

- **«permit»**
  - The client element has access to the supplier element despite the declared visibility of the supplier
Summary

- Dependency
  - The weakest type of association
  - A catch-all

- There are three types of dependency:
  - Usage
  - Abstraction
  - Permission
Analysis –
inheritance and polymorphism
Generalisation

- A relationship between a more general element and a more specific element
- The more specific element is entirely consistent with the more general element but contains more information
- An instance of the more specific element may be used where an instance of the more general element is expected

Substitutability Principle
Example: class generalisation

- More general element
- More specific elements
- "is kind of"

A generalisation hierarchy
Class inheritance

- Subclasses inherit all features of their superclasses:
  - attributes
  - operations
  - relationships
  - stereotypes, tags, constraints

- Subclasses can add new features

- Subclasses can override superclass operations

- We can use a subclass instance anywhere a superclass instance is expected

**Substitutability Principle**

<table>
<thead>
<tr>
<th>Shape</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>origin</td>
<td>Point = (0,0)</td>
</tr>
<tr>
<td>width</td>
<td>int {&gt;0}</td>
</tr>
<tr>
<td>height</td>
<td>int {&gt;0}</td>
</tr>
<tr>
<td>draw</td>
<td>g : Graphics</td>
</tr>
<tr>
<td>getArea</td>
<td>() : int</td>
</tr>
<tr>
<td>getBoundingArea</td>
<td>() : int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Square</th>
<th>Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>{width = height}</td>
<td>radius : int = width/2</td>
</tr>
</tbody>
</table>

But what’s wrong with these subclasses
Subclasses often need to *override* superclass behaviour.

To override a superclass operation, a subclass must provide an operation with the same signature:

- The operation signature is the operation name, return type and types of all the parameters.
- The names of the parameters don’t count as part of the signature.
Abstract operations & classes

We can't provide an implementation for `Shape :: draw( g : Graphics )` or for `Shape :: getArea() : int` because we don't know how to draw or calculate the area for a "shape"!

Operations that lack an implementation are abstract operations

A class with any abstract operations can't be instantiated and is therefore an abstract class
Exercise

what’s wrong with this model?

Vehicle

JaguarXJS

Truck
Polymorphism

- Polymorphism = "many forms"
  - A polymorphic operation has many implementations
  - Square and Circle provide implementations for the polymorphic operations \texttt{Shape::draw()} and \texttt{Shape::getArea()}

- All concrete subclasses of Shape \textit{must} provide concrete \texttt{draw()} and \texttt{getArea()} operations because they are abstract in the superclass
  - For \texttt{draw()} and \texttt{getArea()} we can treat all subclasses of Shape in a similar way - we have defined a contract for Shape subclasses
What happens?

- Each class of object has its own implementation of the draw() operation.
- On receipt of the draw() message, each object invokes the draw() operation specified by its class.
- We can say that each object "decides" how to interpret the draw() message based on its class.

Diagram:

- :Canvas
  - 1.draw() -> s1:Circle
  - 2.draw() -> s2:Square
  - 3.draw() -> s3:Circle
  - 4.draw() -> s4:Circle
We have overridden the deposit() operation even though it is *not* abstract. This is perfectly legal, and quite common, although it is generally considered to be bad style and should be avoided if possible.
Summary

- **Subclasses:**
  - inherit *all* features from their parents including constraints and relationships
  - may add *new* features, constraints and relationships
  - may *override* superclass operations
- A class that can’t be instantiated is an abstract class
Analysis - packages
Analysis packages

- A package is a *general purpose* mechanism for organising model elements into groups
  - Group semantically related elements
  - Define a “semantic boundary” in the model
  - Provide 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
- *Every* model element is owned by exactly one package
  - A hierarchy rooted in a top level package that can be stereotyped «topLevel»
- Analysis packages contain:
  - Use cases, analysis classes, use case realizations, analysis packages
Package syntax

A package that contains model elements that specify a reusable architecture.

A package that contains elements that are intended to be reused by other packages. Analogous to a class library in Java, C# etc.

standard UML 2 package stereotypes

<table>
<thead>
<tr>
<th>stereotype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>«framework»</td>
<td>A package that contains model elements that specify a reusable architecture</td>
</tr>
<tr>
<td>«modelLibrary»</td>
<td>A package that contains elements that are intended to be reused by other packages. Analogous to a class library in Java, C# etc.</td>
</tr>
</tbody>
</table>
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
# Package dependencies

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

**Transitivity** - if dependencies x and y are transitive, there is an *implicit* dependency between A and C.
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`.
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.
- Each package should have the minimum number of public or protected elements.

Diagram:

- **Application Specific Layer**:
  - Sales
  - Products

- **Application General Layer**:
  - Account Management
  - Inventory Management

Partitions:
Finding analysis packages

- These are often discovered as the model matures
- 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
- Analysis classes that realise these groupings 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 realise several use cases that are allocated to different packages
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!

© Clear View Training 2010 v2.6
Summary

- Packages are the UML way of grouping modeling elements
- There are dependency and generalisation relationships between packages
- The package structure of the analysis model defines the logical system architecture
Analysis - use case realization
Analyse a use case

- Business model [or domain model]
- Requirements model
- Use case model
- Architecture description

→ Use case engineer
→ Analyse a use case
→ Analysis class
→ Use case realization
What are use case realizations?

- Each use case has exactly one **use case realization**
  - parts of the model that show how analysis classes collaborate together to realise the behaviour specified by the use case
  - they model *how* the use case is realised by the analysis classes we have identified

- They are rarely modelled explicitly
  - they form an implicit part of the backplane of the model
  - they can be drawn as a stereotyped collaboration
UC realization - elements

Use case realizations consist of the following elements:

- Analysis class diagrams
  - These show relationships between the analysis classes that interact to realise the UC

- Interaction diagrams
  - These show collaborations between specific objects that realise the UC. They are “snapshots” of the running system

- Special requirements
  - UC realization may well uncover new requirements specific to the use case. These must be captured

- Use case refinement
  - We may discover new information during realization that means that we have to update the original UC
Interactions

- Interactions are units of behavior of a context classifier.

- In use case realization, the context classifier is a use case.
  - The interaction shows how the behavior specified by the use case is realized by instances of classifiers.

- Interaction diagrams capture an interaction as:
  - Lifelines – participants in the interaction
  - Messages – communications between lifelines
A lifeline represents a single participant in an interaction
- Shows how a classifier instance may participate in the interaction

Lifelines have:
- name - the name used to refer to the lifeline in the interaction
- selector - a boolean condition that selects a specific instance
- type - the classifier that the lifeline represents an instance of

They *must* be uniquely identifiable within an interaction by name, type or both

The lifeline has the same icon as the classifier that it represents
- The lifeline `jimsAccount` represents an instance of the `Account` class
- The selector `[ id = "1234" ]` selects a specific `Account` instance with the id "1234"
## Messages

A message represents a communication between two lifelines

<table>
<thead>
<tr>
<th>sender</th>
<th>receiver/target</th>
<th>type of message</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>synchronous message</td>
<td>calling an operation synchronously, the sender waits for the receiver to complete</td>
</tr>
<tr>
<td></td>
<td></td>
<td>asynchronous send</td>
<td>calling an operation asynchronously, sending a signal, the sender <em>does not</em> wait for the receiver to complete</td>
</tr>
<tr>
<td></td>
<td></td>
<td>message return</td>
<td>returning from a synchronous operation call, the receiver returns focus of control to the sender</td>
</tr>
<tr>
<td></td>
<td></td>
<td>creation</td>
<td>the sender creates the target</td>
</tr>
<tr>
<td></td>
<td></td>
<td>destruction</td>
<td>the sender destroys the receiver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>found message</td>
<td>the message is sent from outside the scope of the interaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lost message</td>
<td>the message fails to reach its destination</td>
</tr>
</tbody>
</table>
Interaction diagrams

- **Sequence diagrams**
  - Emphasize time-ordered sequence of message sends
  - Show interactions arranged in a time sequence
  - Are the richest and most expressive interaction diagram
  - Do not show object relationships explicitly - these can be inferred from message sends

- **Communication diagrams**
  - Emphasize the structural relationships between lifelines
  - Use communication diagrams to make object relationships explicit

- **Interaction overview diagrams**
  - Show how complex behavior is realized by a set of simpler interactions

- **Timing diagrams**
  - Emphasize the real-time aspects of an interaction
Sequence diagram syntax

- All interaction diagrams may be prefixed `sd` to indicate their type
  - You can generally infer diagram types from diagram syntax
- Activations indicate when a lifeline has focus of control - they are often omitted from sequence diagrams

The Registrar selects "add course".

The system creates the new Course.

object creation message

object is created at this point

notes can form a "script" describing the flow
Deletion and self-delegation

- Self delegation is when a lifeline sends a message to itself
  - Generates a nested activation
- Object deletion is shown by terminating the lifeline's tail at the point of deletion by a large X
State invariants and constraints

sd ProcessAnOrder

:Customer

raiseOrder()

A

acceptPayment()

{B - A <= 28 days}

B

:OrderManager

acceptPayment()

state invariant

unpaid

B

acceptPayment()

paid

deliver()

:DeliveryManager

delivered

deliver()
Sequence diagrams may be divided into areas called combined fragments. Combined fragments have one or more operands. Operators determine how the operands are executed. Guard conditions determine whether operands execute. Execution occurs if the guard condition evaluates to true.

- A single condition may apply to all operands OR
- Each operand may be protected by its own condition
Common operators

<table>
<thead>
<tr>
<th>operator</th>
<th>long name</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>opt</strong></td>
<td>Option</td>
<td>There is a single operand that executes if the condition is true (like if … then)</td>
</tr>
<tr>
<td><strong>alt</strong></td>
<td>Alternatives</td>
<td>The operand whose condition is true is executed. The keyword else may be used in place of a Boolean expression (like select… case)</td>
</tr>
</tbody>
</table>
| **loop** | Loop      | This has a special syntax: 
loop min, max [condition] 
Iterate min times and then up to max times while condition is true |
| **break**| Break     | The combined fragment is executed rather than the rest of the enclosing interaction |
| **ref**  | Reference | The combined fragment refers to another interaction |

ref has a single operand that is a reference to another interaction.
This is an interaction use.
The rest of the operators

These operators are less common

<table>
<thead>
<tr>
<th>operator</th>
<th>long name</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>par</td>
<td>parallel</td>
<td>Both operands execute in parallel</td>
</tr>
<tr>
<td>seq</td>
<td>weak sequencing</td>
<td>The operands execute in parallel subject to the constraint that event occurrences on the same lifeline from different operands must happen in the same sequence as the operands</td>
</tr>
<tr>
<td>strict</td>
<td>strict sequencing</td>
<td>The operands execute in strict sequence</td>
</tr>
<tr>
<td>neg</td>
<td>negative</td>
<td>The combined fragment represents interactions that are invalid</td>
</tr>
<tr>
<td>critical</td>
<td>critical region</td>
<td>The interaction must execute atomically without interruption</td>
</tr>
<tr>
<td>ignore</td>
<td>ignore</td>
<td>Specifies that some messages are intentionally ignored in the interaction</td>
</tr>
<tr>
<td>consider</td>
<td>consider</td>
<td>Lists the messages that are considered in the interaction (all others are ignored)</td>
</tr>
<tr>
<td>assert</td>
<td>assertion</td>
<td>The operands of the combined fragments are the only valid continuations of the interaction</td>
</tr>
</tbody>
</table>
branching with opt and alt

- **opt semantics:**
  - single operand that executes if the condition is true

- **alt semantics:**
  - two or more operands each protected by its own condition
  - an operand executes if its condition is true
  - use `else` to indicate the operand that executes if *none* of the conditions are true

---

sd example of opt and alt

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>opt [condition]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>do this if condition is true</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>[condition1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt</td>
<td></td>
</tr>
<tr>
<td>do this if condition1 is true</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>[condition2]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[else]</td>
</tr>
<tr>
<td>do this if neither condition is true</td>
<td></td>
</tr>
</tbody>
</table>
```
Iteration with loop and break

- **Loop semantics:**
  - Loop min times, then loop (max - min) times while condition is true

- **Loop syntax:**
  - A loop without min, max or condition is an infinite loop
  - If only min is specified then max = min
  - Condition can be
    - Boolean expression
    - Plain text expression *provided* it is clear!

- Break specifies what happens when the loop is broken out of:
  - The break fragment executes
  - The rest of the loop after the break does *not* execute

- The break fragment is *outside* the loop and so should overlap it as shown
## loop idioms

<table>
<thead>
<tr>
<th>type of loop</th>
<th>semantics</th>
<th>loop expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>infinite loop</td>
<td>keep looping forever</td>
<td>loop *</td>
</tr>
<tr>
<td>for i = 1 to n</td>
<td>repeat ( n ) times</td>
<td>loop n</td>
</tr>
<tr>
<td>{body}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>while( booleanExpression )</td>
<td>repeat while booleanExpression is true</td>
<td>loop [ booleanExpression ]</td>
</tr>
<tr>
<td>{body}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>repeat</td>
<td>execute once then repeat while booleanExpression is true</td>
<td>loop 1, * [booleanExpression]</td>
</tr>
<tr>
<td>{body}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>while( booleanExpression )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>forEach object in collection</td>
<td>Execute the loop once for each object in a collection</td>
<td>loop [for each object in collection]</td>
</tr>
<tr>
<td>{body}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>forEach object in ObjectType</td>
<td>Execute the loop once for each object of a particular type</td>
<td>loop [for each object in :ObjectType]</td>
</tr>
</tbody>
</table>
Communication diagrams emphasize the structural aspects of an interaction - how lifelines connect together

- Compared to sequence diagrams they are semantically weak
- Object diagrams are a special case of communication diagrams
Iteration

- Iteration is shown by using the iteration specifier (*), and an optional iteration clause
  - There is no prescribed UML syntax for iteration clauses
  - Use code or pseudo code
- To show that messages are sent in parallel use the parallel iteration specifier, *///
Branching is modelled by prefixing the sequence number with a *guard condition*. There is no prescribed UML syntax for guard conditions! In the example above, we use the variable `found`. This is true if both the student and the course are found, otherwise it is false.
Summary

- In this section we have looked at use case realization using interaction diagrams.

- There are four types of interaction diagram:
  - Sequence diagrams – emphasize time-ordered sequence of message sends
  - Communication diagrams – emphasize the structural relationships between lifelines
  - Interaction overview diagrams – show how complex behavior is realized by a set of simpler interactions
  - Timing diagrams – emphasize the real-time aspects of an interaction

- We have looked at sequence diagrams and communication diagrams in this section - we will look at the other types of diagram later.
Analysis - advanced use case realization
Interaction occurrences

- An interaction use is inserted into the including interaction
  - All lifelines in the interaction use must also be in the including interaction
  - Be very aware of where the interaction use leaves the focus of control!
- Draw the interaction use across the lifelines it uses

Sequence of messages in I2:
- n1
- n2
- m1 from I1
- n3
Parameters

- Interactions may be parameterized
  - This allows specific values to be supplied to the interaction in each of its occurrences
  - Specify parameters using operation syntax
  - Values for the parameters are supplied in the interaction occurrences

- Interactions may return values
  - You can show a specific return value as a value return e.g.
    \[ A::a = I3( "value" ):"ret" \]
Gates are inputs and outputs of interactions (and combined fragments – see next slide)

- Provide connection points that relate messages inside an occurrence or fragment to messages outside it

```
Sequence of messages in l2:
n1
n2
m0
m1
n3

from l1
```
Continuations allow an interaction fragment to terminate in such a way that it can be continued by another fragment.
Summary

In this section we have looked at:

- Interaction occurrences
- Parameters
- Gates
- Continuations
Analysis - activity diagrams
What are activity diagrams?

- Activity diagrams are "OO flowcharts"!
- They allow us to model a process as a collection of nodes and edges between those nodes

Use activity diagrams to model the behavior of:
- use cases
- classes
- interfaces
- components
- collaborations
- operations and methods
- business processes
Activities

- Activities are networks of nodes connected by edges.
- There are three categories of node:
  - Action nodes - represent discrete units of work that are atomic within the activity.
  - Control nodes - control the flow through the activity.
  - Object nodes - represent the flow of objects around the activity.
- Edges represent flow through the activity.
- There are two categories of edge:
  - Control flows - represent the flow of control through the activity.
  - Object flows - represent the flow of objects through the activity.
Activity diagram syntax

- Activities are networks of *nodes* connected by *edges*
  - The control flow is a type of edge
- Activities usually start in an *initial node* and terminate in a *final node*
- Activities can have preconditions and postconditions
- When an action node finishes, it emits a token that may traverse an edge to trigger the next action
  - This is sometimes known as a *transition*
- You can break an edge using connectors:

```
  A  |  A
incoming connector  |  outgoing connector
```

**Send letter**

precondition: know topic for letter  
postcondition: letter sent to address

```
Write letter

Address letter

Post letter
```

- **initial node**
- **action node**
- **edge**
- **control flow**
- **final node**
Activity diagram semantics

- **The token game**
  - Token - an object, some data or a focus of control
  - Imagine tokens flowing around the activity diagram

- Tokens traverse from a source node to a target node via an edge
  - The source node, edge and target node may all have constraints controlling the movement of tokens
  - All constraints must be satisfied before the token can make the traversal

- A node executes when:
  - It has tokens on all of its input edges AND these tokens satisfy predefined conditions (see later)

- When a node starts to execute it takes tokens off its input edges
- When a node has finished executing it offers tokens on its output edges
Activity partitions

- Each activity partition represents a high-level grouping of a set of related actions
  - Partitions can be hierarchical
  - Partitions can be vertical, horizontal or both
- Partitions can refer to many different things e.g. business organisations, classes, components and so on
- If partitions can’t be shown clearly using parallel lines, put their name in brackets directly above the name of the activities

![Diagram](chart.png)

- (London::Marketing) Market product
- (p1, p2) SomeAction

nested partitions, multiple partitions
Action nodes

- Action nodes offer a token on all of their output edges when:
  - There is a token simultaneously on each input edge
  - The input tokens satisfy all preconditions specified by the node

- Action nodes:
  - Perform a logical AND on their input edges when they begin to execute
  - Perform an implicit fork on their output edges when they have finished executing
### Types of action node

<table>
<thead>
<tr>
<th>action node syntax</th>
<th>action node semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Close Order" /></td>
<td><strong>Call action</strong> - invokes an activity, a behavior or an operation. The most common type of action node. See next slide for details.</td>
</tr>
<tr>
<td><img src="image" alt="OrderEvent" /></td>
<td><strong>Send signal action</strong> - sends a signal asynchronously. The sender <em>does not</em> wait for confirmation of signal receipt. It may accept input parameters to create the signal</td>
</tr>
<tr>
<td><img src="image" alt="OrderEvent" /></td>
<td><strong>Accept event action</strong> - waits for events detected by its owning object and offers the event on its output edge. Is enabled when it gets a token on its input edge. If there is <em>no</em> input edge it starts when its containing activity starts and is <em>always</em> enabled.</td>
</tr>
<tr>
<td><img src="image" alt="end of month occurred" /></td>
<td><strong>Accept time event action</strong> - waits for a set amount of time. Generates time events according to its time expression.</td>
</tr>
</tbody>
</table>
Call action node syntax

- The most common type of node
- Call action nodes may invoke:
  - an activity
  - a behavior
  - an operation
- They may contain code fragments in a specific programming language
  - The keyword 'self' refers to the context of the activity that owns the action

Example:
```python
if self.balance <= 0:
    self.status = INCREDIT
else:
    self.status = OVERDRAWN
```

The keyword 'self' refers to the context of the activity that owns the action.
**Control nodes**

<table>
<thead>
<tr>
<th>control node syntax</th>
<th>control node semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Initial node" /></td>
<td>Initial node – indicates where the flow starts when an activity is invoked</td>
</tr>
<tr>
<td><img src="image" alt="Activity final node" /></td>
<td>Activity final node – terminates an activity</td>
</tr>
<tr>
<td><img src="image" alt="Flow final node" /></td>
<td>Flow final node – terminates a specific flow within an activity. The other flows are unaffected</td>
</tr>
<tr>
<td><img src="image" alt="Decision node" /></td>
<td>Decision node – guard conditions on the output edges select one of them for traversal. May optionally have inputs defined by a «decisionInput»</td>
</tr>
<tr>
<td><img src="image" alt="Merge node" /></td>
<td>Merge node – selects one of its input edges</td>
</tr>
<tr>
<td><img src="image" alt="Fork node" /></td>
<td>Fork node – splits the flow into multiple concurrent flows</td>
</tr>
<tr>
<td><img src="image" alt="Join node" /></td>
<td>Join node – synchronizes multiple concurrent flows. May optionally have a join specification to modify its semantics</td>
</tr>
</tbody>
</table>

See examples on next two slides.
A decision node is a control node that has one input edge and two or more alternate output edges
- Each edge out of the decision is protected by a guard condition
- Guard conditions must be mutually exclusive
- The edge can be taken if and only if the guard condition evaluates to true
- The keyword else specifies the path that is taken if none of the guard conditions are true

A merge node accepts one of several alternate flows
- It has two or more input edges and exactly one output edge
Fork and join nodes - concurrency

- Forks nodes model concurrent flows of work
  - Tokens on the single input edge are replicated at the multiple output edges

- Join nodes synchronize two or more concurrent flows
  - Joins have two or more incoming edges and exactly one outgoing edge
  - A token is offered on the outgoing edge when there are tokens on all the incoming edges i.e. when the concurrent flows of work have all finished
Object nodes

- Object nodes indicate that instances of a particular classifier may be available
  - If no classifier is specified, then the object node can hold any type of instance
- Multiple tokens can reside in an object node *at the same time*
  - The upper bound defines the maximum number of tokens (infinity is the default)
- Tokens are presented to the single output edge according to an ordering:
  - FIFO – first in, first out (the default)
  - LI FI – last in, first out
  - Modeler defined – a selection criterion is specified for the object node
Object node syntax

- Object nodes have a flexible syntax. You may show:
  - upper bounds
  - ordering
  - sets of objects
  - selection criteria
  - object in state

- **Order**
  - order objects may be available
  - zero to 12 Order objects may be available
  - last Order object in is the first out (FIFO is the default)

- **Set of Order**
  - sets of Order objects may be available

- **Order**
  - Order objects raised in December may be available
  - select Order objects in the open state
Object nodes can provide input and output parameters to activities
- Input parameters have one or more output object flows into the activity
- Output parameters have one or more input object flows out of the activity
- Draw the object node overlapping the activity boundary
Pins are object nodes for inputs to, and outputs from, actions:

- Same syntax as object nodes
- Input pins have exactly one input edge
- Output pins have exactly one output edge
- Exception pins are marked with an equilateral triangle
- Streaming pins are filled in black or marked with \{stream\}

**Diagram:**

- LogOn
  - GetUserName: UserName[valid]
  - GetPassword: Password[valid]
- Authenticate User
- LogOnException
- LogError

**Annotations:**

- exception pin
- pin
- streaming – see notes

© Clear View Training 2010 v2.6
Summary

- We have seen how we can use activity diagrams to model flows of activities using:
  - Activities
    - Connectors
  - Activity partitions
  - Action nodes
    - Call action node
    - Send signal/accept event action node
    - Accept time event action node
  - Control nodes
    - decision and merge
    - fork and join
  - Object nodes
    - input and output parameters
    - pins
Analysis - advanced activity diagrams
Interruptible activity regions may be interrupted when a token traverses an interrupting edge.

- All flows in the region are aborted.
- Interrupting edges *must* cross the region boundary.
Protected nodes have exception handlers:

- When the exception object is raised in the protected node, flow is directed along an interrupting edge to the exception handler body.
Expansion nodes

- Expansion node – an object node that represents a collection of objects flowing into or out of an expansion region
  - Output collections *must* correspond to input collections in collection type and object type!
- The expansion region is executed once per input element according to the keyword:
  - iterative – process sequentially
  - parallel – process in parallel
  - stream – process a stream of input objects

Expansion regions containing a single action - place the expansion node directly on the action.

© Clear View Training 2010 v2.6
Sending signals and accepting events

- Signals represent information passed asynchronously between objects
  - This information is modelled as attributes of a signal
  - A signal is a classifier stereotyped «signal»
- The accept event action asynchronously accepts event triggers which may be signals or other objects

```
CardDetails
  ∆
  «signal» SecurityEvent

  «signal» AuthorizationRequestEvent
    pin : PIN
    cardDetails : CardDetails

  «signal» AuthorizationEvent
    isAuthorized : Boolean
```

```
Validate card
  CardDetails
  →
  Enter PIN
    PIN
    AuthorizationRequestEvent
      CardDetails
      send signal
    Authorization Event
      [isAuthorized]
      [!isAuthorized]
        Authorized
        Not authorized

Authorization
  isAuthorized : Boolean
  accept event
  send
```
Advanced object flow

- **Input effect**
  - Specifies the effect of the action on objects flowing into it

- **Output effect**
  - Specifies the effect of the action on objects flowing out of it

- **«selection»**
  - The flow to selects objects that meet a specific criterion

- **«transformation»**
  - An object is transformed by the object flow

Diagram:

- `sendPayment`: Order [paid]
- `Order` to `Receipt`
- `acceptPayment` to `Order [paid]`
- `Order` to `recordTransaction` to `Transaction` [create]
- `Order` to `sendReminder`
- `Order [!paid]` to `sendReminder`
- `Order`: `Order.date - now > 28 days`

15.8
Multicast and multireceive

- A «multicast» object flow sends an object to multiple receivers
- A «multireceive» object flow receives an object from multiple receivers
Parameter sets provide alternative sets of input pins and output pins to an action
- Only one input set and one output set may be chosen (XOR)

input condition: ( UserName AND Password ) XOR ( UserName AND Passphrase ) XOR ( Card AND PIN )
output: ( User [Authenticated] ) XOR ( User [!Authenticated] )
Central buffer nodes accept multiple upstream object flows
- They hold the objects until downstream nodes are ready for them
Interaction overview diagrams

- Model the high level flow of control between interactions
- Show interactions and interaction occurrences
- Have activity diagram syntax
In this section we have looked at some of the more advanced features of activity diagrams:

- Interruptible activity regions
- Exception handlers
- Expansion nodes
- Advanced object flow
- Multicast and multireceive
- Parameter sets
- Central buffer nodes
- Interaction overview diagrams
Design - introduction
Design - purpose

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

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

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

- **Deployment model**
Should you maintain 2 models?

- A design model may contain 10 to 100 times as many classes as the analysis model
  - The analysis model helps us to see the big picture without getting lost in implementation details

- We need to maintain 2 models if:
  - It is a big system (>200 design classes)
  - It has a long expected lifespan
  - It is a strategic system
  - We are outsourcing construction of the system

- We can make do with only a design model if:
  - It is a small system
  - It has a short lifespan
  - It is not a strategic system
Workflow - Design

Architect

Use Case Engineer

Component Engineer

Architectural design

Design a use case

Design a class

Design a subsystem
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

java.util
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
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 and its clients.
High cohesion, low coupling

- **High cohesion:**
  - Each class should have a set of operations that support the intent of the class, no more and no less
  - Each class should model a single abstract concept
  - If a class needs to have many responsibilities, then some of these should be implemented by “helper” classes. The class then delegates to its helpers

- **Low coupling:**
  - A particular class should be associated with just enough other classes to allow it to realise its responsibilities
  - Only associate classes if there is a true semantic link between them
  - Never form an association just to reuse a fragment of code in another class!
  - Use aggregation rather than inheritance (next slide)

This example comes from a real system! What’s wrong with it?
Aggregation vs. inheritance

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

1. How can we promote John?
2. Can John have more than one job?
A better solution…

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

![Diagram showing the relationship between Employee, Job, Manager, and Programmer roles with a link for promoting John.]

just change this link at runtime to promote John!
Multiple inheritance

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

In this example, the AutoDialler sounds an alarm and rings the police when triggered - it is logically both a kind of Alarm and a kind of Dialler.
Inheritance vs. interface realization

With inheritance we get two things:
- Interface – the public operations of the base classes
- Implementation – the attributes, relationships, protected and private operations of the base classes

With interface realization we get exactly one thing:
- An interface – a set of public operations, attributes and relationships that have no implementation

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

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

Consider:

<table>
<thead>
<tr>
<th>BoundedIntArray</th>
<th>BoundedFloatArray</th>
<th>BoundedStringArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>size:int</td>
<td>size:int</td>
<td>size:int</td>
</tr>
<tr>
<td>elements[]:int</td>
<td>elements[]:float</td>
<td>elements[]:String</td>
</tr>
<tr>
<td>addElement( e:int ):void</td>
<td>addElement( e:float ):void</td>
<td>addElement( e:String ):void</td>
</tr>
<tr>
<td>getElement( i:int):int</td>
<td>getElement( i:int):float</td>
<td>getElement( i:int):String</td>
</tr>
</tbody>
</table>

spot the difference!
Template instantiation - the template parameters are bound to actual values to create new classes based on the template:

- If the type of a parameter is not specified then the parameter defaults to being a classifier.
- Parameter names are local to the template - two templates do not have relationship to each other just because they use the same parameter names!
- Explicit binding is preferred as it allows named instantiations.
Templates & multiple inheritance

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

<table>
<thead>
<tr>
<th>language</th>
<th>templates</th>
<th>multiple inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Java</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>C++</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Visual Basic</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Python</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
A nested class is a class defined inside another class
- It is encapsulated inside the namespace of its containing class
- Nested classes tend to be design artifacts

Nested classes are only accessible by:
- their containing class
- objects of that their containing class
Summary

- Design classes come from:
  - A refinement of analysis classes (i.e. the business domain)
  - From the solution domain

- Design classes must be well-formed:
  - Complete and sufficient
  - Primitive operations
  - High cohesion
  - Low coupling

- Don’t overuse inheritance
  - Use inheritance for "is kind of"
  - Use aggregation for "is role played by"
  - Multiple inheritance should be used sparingly (mixins)
  - Use interfaces rather than inheritance to define contracts

- Use templates and nested classes only where the target language supports them
Design - refining analysis relationships
Design relationships

- Refining analysis associations to design associations involves several procedures:
  - refining associations to aggregation or composition relationships where appropriate
  - implementing one-to-many associations
  - implementing many-to-one associations
  - implementing many-to-many associations
  - implementing bidirectional associations
  - implementing association classes

- All design associations must have:
  - navigability
  - multiplicity on both ends
Aggregation and composition

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

UML defines two types of association:

**Aggregation**

Some objects are weakly related like a computer and its peripherals.

**Composition**

Some objects are strongly related like a tree and its leaves.
Aggregation semantics

aggregation is a *whole-part* relationship

- 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

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
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.

cycles are NOT allowed
Aggregation hierarchy
Composition semantics

- 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.

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.

Mouse composition is a strong form of aggregation.

always 0..1 or 1

composition

part

Mouse

1

Button

1..4

composition

always 0..1 or 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

- 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:

<table>
<thead>
<tr>
<th>property pair</th>
<th>OCL collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>{unordered, nonunique}</td>
<td>Bag</td>
</tr>
<tr>
<td>{unordered, unique}</td>
<td>Set (default)</td>
</tr>
<tr>
<td>{ordered, unique}</td>
<td>OrderedSet</td>
</tr>
<tr>
<td>{ordered, nonunique}</td>
<td>Sequence</td>
</tr>
</tbody>
</table>

- {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
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.

\[
\text{m: HashMap} \\
\text{node1} \rightarrow \text{key1} \rightarrow \text{value1} \\
\text{node2} \rightarrow \text{key2} \rightarrow \text{value2} \\
\text{node3} \rightarrow \text{key3} \rightarrow \text{value3} \\
\]

\[
\text{A} \rightarrow \{\text{map}\} \rightarrow \text{B} \\
\text{1} \rightarrow \ast \\
\text{you can indicate the type of collection using a constraint}
\]
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 association into two unidirectional associations.
- Again, we must decide which side of the association should have primacy and use composition, aggregation and navigability accordingly.
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

```
Company (*)  Job (*) Person
   |      |    |      |
   |      |    |      |
   |      |    |      |
```

- each Person can only have one job with a given Company
Summary

In this section we have seen how we take the incompletely specified associations in an analysis model and refine them to:

- **Aggregation**
  - Whole-part relationship
  - Parts are independent of the whole
  - Parts may be shared between wholes
  - The whole is incomplete in some way without the parts

- **Composition**
  - A strong form of aggregation
  - Parts are entirely dependent on the whole
  - Parts may not be shared
  - The whole is incomplete without the parts

- One-to-many, many-to-many, bi-directional associations and association classes are refined in design
Design - interfaces and components
What is an interface?

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

<table>
<thead>
<tr>
<th>Interface specifies</th>
<th>Realizing classifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>Must have an operation with the same signature and semantics</td>
</tr>
<tr>
<td>attribute</td>
<td>Must have public operations to set and get the value of the attribute. The realizing classifier is not required to actually have the attribute specified by the interface, but it must behave as though it has</td>
</tr>
<tr>
<td>association</td>
<td>Must have an association to the target classifier. If an interface specifies an association to another interface, then the implementing classifiers of these interfaces must have an association between them</td>
</tr>
<tr>
<td>constraint</td>
<td>Must support the constraint</td>
</tr>
<tr>
<td>stereotype</td>
<td>Has the stereotype</td>
</tr>
<tr>
<td>tagged value</td>
<td>Has the tagged value</td>
</tr>
<tr>
<td>protocol</td>
<td>Realizes the protocol</td>
</tr>
</tbody>
</table>
A provided interface indicates that a classifier implements the services defined in an interface.

- **“Class” style notation**
  - `<<interface>>
    - Borrow
    - borrow()
    - return()
    - isOverdue()`

- **“Lollipop” style notation**
  - Borrow
  - relationship
  - interface
  - `Book`, `CD`

(Note: you can’t show the interface operations or attributes with this shorthand style of notation)
Required interface syntax

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

**Class style notation**

- Library
  - "interface"
  - Borrow

**Lollipop style notation**

- Library
  - Borrow

- Library

© Clear View Training 2010 v2.6
Assembly connectors

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

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

![Diagram](image-url)
Interfaces and CBD

- Interfaces are the key to component based development (CBD)
- This is constructing software from replaceable, plug-in parts:
  - Plug – the provided interface
  - Socket – the required interface

- Consider:
  - Electrical outlets
  - Computer ports – USB, serial, parallel

- Interfaces define a contract so classifiers that realise the interface agree to abide by the contract and can be used interchangeably
What is a component?

- The UML 2.0 specification states that, "A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment"
  - A black-box whose external behaviour is completely defined by its provided and required interfaces
  - May be substituted for by other components provided they all support the same protocol

- Components can be:
  - Physical - can be directly instantiated at run-time e.g. an Enterprise JavaBean (EJB)
  - Logical - a purely logical construct e.g. a subsystem
    - only instantiated indirectly by virtue of its parts being instantiated
Component syntax

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

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>I2</td>
</tr>
</tbody>
</table>
```

black box notation

white box notation
# Standard component stereotypes

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

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

- Challenge each association:
  - Does the association have to be to another class, or can it be to an interface?

- Challenge each message send:
  - Does the message send have to be to another class, or can it be to an interface?

- Look for repeating groups of operations
- Look for groups of operations that might be useful elsewhere
- Look for possibilities for future expansion
- Look for cohesive sets of provided and required interfaces and organize these into named ports
- Look at the dependencies between subsystems - mediate these by an assembly connector where possible
Designing with interfaces

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

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

Advantages:
- When we design with classes, we are designing to specific implementations
- When we design with interfaces, we are instead designing to contracts which may be realised by many different implementations (classes)
- Designing to contracts frees our model from implementation dependencies and thereby increases its flexibility and extensibility

Disadvantages:
- Interfaces can add flexibility to systems BUT flexibility may lead to complexity
- Too many interfaces can make a system too flexible!
- Too many interfaces can make a system hard to understand

Keep it simple!
Summary

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

- Design subsystems and interfaces allow us to:
  - Componentize our system
  - Define an architecture
Design - use case realization
Use case realization - design

- A collaboration of Design objects and classes that realise a use case

- A Design use case realization contains
  - Design object interaction diagrams
  - Links to class diagrams containing the participating Design classes
  - An explanatory text (flow)

- There is a trace between an Analysis use case realization and a Design use case realization

- The Design use case realization specifies implementation decisions and implements the non-functional requirements
Interaction diagrams in design

- We only produce a design interaction diagram where it adds value to the project:
  - A refinement of the analysis interaction diagrams to illustrate design issues
  - New diagrams to illustrate technical issues
  - New diagrams to illustrate central mechanisms

- In design:
  - Sequence diagrams are used more than communication diagrams
  - Timing diagrams may be used to capture timing constraints
Sequence diagrams in design

**sd AddCourse - design**

- :Registrar
  - addCourse( "UML" )
- :RegistrationUI
  - addCourse( "UML" )
- :RegistrationManager
  - uml = Course("UML")
  - uml:Course
- :DBManager
  - save(uml)

© Clear View Training 2010 v2.6
Concurrence – active classes

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

activates

security

fire

intruder

siren

control box

security monitor

fire sensor

security sensor

siren

activate()

soundFireAlarm()

par

monitor()

intruder = isTriggered()

fire = isTriggered()

fire()

intruder()

soundIntruderAlarm()

critical

fire()

soundActivatedAlarm()

loop 1, * ![fire]

loop 1, * [!(intruder) & ![fire]]
Concurrency – active objects

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

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

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

![Timing Diagram Example]

296
Messages on timing diagrams

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

Messages on timing diagrams diagram:

- `soundIntruderAlarm()`
- `soundFireAlarm()`

Messages:

- `trig`rgered
- `notTriggered`

Timings:

- `{t <= 0.016}`
- `{t = 30}`
- `{t <= 15}`
- `{t <= 15}`
- `{t <= 15}`

All times in minutes.
Example: use case realization - design

20.8
Summary

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

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

State = off

Event = turnOn

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

State = on

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

- The light bulb turns off

state = off

The light bulb turns off
Every state machine should have an initial state which indicates the first state of the sequence.

Unless the states cycle endlessly, state machines should have a final state which terminates the sequence of transitions.

We’ll look at each element of the state machine in detail in the next few slides!
States

- "A condition or situation during the life of an object during which it satisfies some condition, performs some activity or waits for some event"

- The state of an object at any point in time is determined by:
  - The values of its attributes
  - The relationships it has to other objects
  - The activities it is performing

<table>
<thead>
<tr>
<th>Color</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>red : int</td>
<td>green : int</td>
</tr>
<tr>
<td>blue : int</td>
<td></td>
</tr>
</tbody>
</table>
State syntax

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

**Action syntax:** eventTrigger / action

**Activity syntax:** do / activity

### EnteringPassword

- entry/display password dialog
- exit/validate password
- keypress/ echo "*"
- help/display help
- do/get password
Transitions

behavioral state machine

A

event1, event2 [guard condition] / act1, act2

B

protocol state machine {protocol}

C

[precondition] event1, event2 / [postcondition]

D

precondition events postcondition
Connecting - the junction pseudo state

- The junction pseudo state can:
  - connect transitions together (merge)
  - branch transitions

- Each outgoing transition must have a mutually exclusive guard condition
Branching – the choice pseudo state

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

The diagram illustrates the choice pseudo-state with transitions labeled as follows:
- Unpaid
- Overpaid
- FullyPaid
- PartiallyPaid

Guard conditions are:
- \( [\text{payment} > \text{balance}] \)
- \( [\text{payment} = \text{balance}] \)
- \( [\text{payment} < \text{balance}] \)

Transition actions include:
- acceptPayment
- makeRefund
Events

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

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

```
InCredit

deposit(m) / balance = balance + m

RejectingWithdrawal
entry / logRejectedWithdrawal()

AcceptingWithdrawal
entry / balance = balance - m
```

```
SimpleBankAccount

withdraw(m) [balance < m]
withdraw(m) [balance >= m]
```

© Clear View Training 2010 v2.6
Signal events

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

```
InCredit
deposit(m)/ balance = balance + m
```

```
RejectingWithdrawal
withdraw(m) [balance < m]
entry/ logRejectedWithdrawal()
```

```
AcceptingWithdrawal
withdraw(m) [balance >= m]
entry/ balance = balance - m
```

```
SimpleBankAccount
```

```
OverdrawnAccount
```

```
«signal»
OverdrawnAccount
```

date : Date
accountNumber : long
amountOverdrawn : double
Receiving a signal

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

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

```
SimpleBankAccount

InCredit
---
deposit(m)/ balance = balance + m
balance >= 5000 / notifyManager()

RejectingWithdrawal
---
entry/ logRejectedWithdrawal()

AcceptingWithdrawal
---
entry/ balance = balance - m

withdraw(m) [balance < m]

withdraw(m) [balance >= m]

OverdrawnAccount
```
Time events

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

Context: CreditAccount class

```
Overdrawn
balance < overdraftLimit / notifyManager
```

```
Frozen
after(3 months)
```
Summary

We have looked at:

- Behavioral state machines
- Protocol state machines
- States
  - Actions
    - Exit and entry actions
  - Activities
- Transitions
  - Guard conditions
  - Actions
- Events
  - Call, signal, change and time
Design - advanced state machines
Composite states

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

- Contains a single region
- The nested states inherit the cancel transition from Dialing ISP

```
<table>
<thead>
<tr>
<th>ISPDialer</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry/ offHook</td>
</tr>
<tr>
<td>dial</td>
</tr>
<tr>
<td>WaitingForDialtone [dialtone]</td>
</tr>
<tr>
<td>Dialing do/ dial ISP</td>
</tr>
<tr>
<td>WaitingForCarrier</td>
</tr>
<tr>
<td>notConnected</td>
</tr>
<tr>
<td>entry pseudo state</td>
</tr>
<tr>
<td>after(20 seconds)/ noDialtone</td>
</tr>
<tr>
<td>connected</td>
</tr>
<tr>
<td>exit pseudo-state</td>
</tr>
<tr>
<td>NotConnected entry/ onHook</td>
</tr>
<tr>
<td>cancel</td>
</tr>
<tr>
<td>Connected exit/ onHook do/ useConnection</td>
</tr>
</tbody>
</table>
```
Orthogonal composite states

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

Synchronized exit - exit the superstate when both regions have terminated

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

- If we want to refer to this state machine in other state machines, without cluttering the diagrams, then we must use a **submachine state**
- Submachine states reference another state machine
- Submachine states are semantically equivalent to composite states
A submachine state is equivalent to including a copy of the submachine in place of the submachine state.
Submachine communication

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

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

- Shallow history remembers the last substate at the same level as the shallow history pseudo-state.
- Next time the super state is entered there is an automatic transition to the remembered substate.
Deep history remembers the last substate at the same level or lower than the deep history pseudo state.
Summary

We have explored advanced aspects of state machines including:

- Simple composite states
- Orthogonal composite states
- Submachine communication
  - Attribute values
- Submachine states
- Shallow history
- Deep history
Implementation - introduction
Implementation - purpose

- To implement the design classes and components
  - To create an implementation model
- To convert the Design Model into an executable program
The implementation model is part of the design model. It comprises:

- Component diagrams showing components and the artifacts that realize them
- Deployment diagrams showing artifacts deployed on nodes

Components are manifest by artifacts
Artifacts are deployed on nodes
Implementation workflow detail

Architectural implementation

Implement a class

Implement a component

Integrate system

Perform unit test

Component engineer

System integrator
Summary

- Implementation begins in the last part of the elaboration phase and is the primary focus throughout later stages of the construction phase.
- Purpose – to create an executable system.
- Artifacts:
  - Component diagrams
    - Components and artifacts
  - Deployment diagrams
    - Nodes and artifacts
Implementation - deployment
Deployment model

- The deployment model is an object model that describes how functionality is distributed across physical nodes
  - It models the mapping between the software architecture and the physical system architecture

- It models the system’s physical architecture as artifacts deployed on nodes
  - Each node is a type of computational resource
  - Nodes have relationships that represent methods of communication between them e.g. http, iiop, netbios
  - Artifacts represent physical software e.g. a JAR file or .exe file

Design - we may create a first-cut deployment diagram:
- Focus on the big picture - nodes or node instances and their connections
- Leave detailed artifact deployment to the implementation workflow

Implementation - finish the deployment diagram:
- Focus on artifact deployment on nodes
Nodes – descriptor form

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

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

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

- An artifact represents a type of concrete, real-world thing such as a file
  - Can be deployed on nodes

- Artifact instances represent particular *copies* of artifacts
  - Can be deployed on node instances

- An artifact can manifest one or more components
  - The artifact is the represents the thing that is the physical manifestation of the component (e.g. a JAR file)
Artifacts and components

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

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

UML 2 provides a small number of standard stereotypes for artifacts

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

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

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

■ The descriptor form deployment diagram
  ■ Allows you to show how functionality represented by artefacts is distributed across nodes
  ■ Nodes represent types of physical hardware or execution environments

■ The instance form deployment diagram
  ■ Allows you to show how functionality represented by artefact instances is distributed across node instances
  ■ Node instances represent actual physical hardware or execution environments
Course summary
We have now covered the 4 UP workflows in which OO analysts and designers participate.
Next steps…

- There is a lot of useful information at www.clearviewtraining.com:
  - UML resources for our books:
    - "UML 2 and the Unified Process"
    - "Enterprise Patterns and MDA"
  - Advanced UML modelling techniques
    - Literate modeling
    - Archetype patterns
  - SUMR - open source use case modeling tools
  - Speak directly to the course author Dr. Jim Arlow:
    - Jim.Arlow@clearviewtraining.com

- Further training, mentoring and consultancy in all aspects of object technology and project management is available from:
  - Zühlke Engineering Limited (UK), Zühlke Engineering AG (Switzerland) and Zühlke Engineering GmbH (Germany) - www.zuhlke.com
Finally…

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