Notes for revision

The notes in this document are designed for last minute revision. They cover the main points of the material in summarised form, but for detailed information and examples, you will need to refer back to the relevant chapters.

Chapter 1

Introduction

Frameworks

- In system development a framework or life cycle helps to identify milestones, structure activities and monitor deliverables.
- A framework provides a foundation for development and ensures a certain level of consistency in how the work is carried out, especially with large teams of developers.
- A framework supports project management (planning, monitoring and controlling the development process).
- The stages of the traditional system life cycle are generally considered to be requirements, analysis, design, implementation and installation.
- Traditional life cycle models include the Waterfall, the V-Model, the Spiral, Prototyping, Iterative development and Incremental development.
- The phases in object-oriented development are inception, elaboration, construction and transition.
- Workflows (activities) in object-oriented development include requirements, analysis, design, implementation and testing.
- A workflow may be carried out during any phase of development, and a phase may involve the whole range of workflows.
- In the development of object-oriented systems, the emphasis is on:
  - the central idea of the object
  - a seamless development process
  - using the same models throughout development
  - the importance of iterative development
  - issues relating to quality, such as maintenance, modification and reuse.

Methods

- A development method is much more prescriptive than a life cycle, providing detailed instructions on how to build a system.
- The Rational Unified Process (RUP) is an example of a widely used object-oriented development method.
- The RUP 'Best Practices' are:
  1. Develop software iteratively
  2. Manage requirements
  3. Use component-based architectures
  4. Visually model software
  5. Verify software quality
6. Control changes to software

UML and modelling

- The Unified Modelling Language (UML) is a set of diagrammatic modelling techniques, designed to be used in object-oriented development.
- The main UML models are the use case, the class diagram, interaction (sequence and collaboration) diagrams, the state diagram, the activity diagram, component diagram and deployment diagram.
- The authors of UML view the architecture of a system from five different perspectives, known as the '4 + 1 view'. These are: the use case view, the design view, the process view, the implementation view and the deployment view.
- A model represents some aspect of a problem or the system that is designed to solve it.
- Abstraction is the process of concentrating on certain aspects of the problem and ignoring others.
- Decomposition is the breaking down of a large complex problem into successively smaller parts that can be worked on as independent units.
- Models help to impose structure on information gathered from clients and users of the system. They are a useful basis for discussions between clients and developers.
- A CASE (Computer Aided Software Engineering) tool is any piece of software that helps people to develop systems.
- The UML is supported by a number of CASE tools, such as Rational Rose™ and Together™.
Chapter 2

Requirements for the Wheels case study system

- The principal stages of requirements engineering are elicitation, specification and validation.

Elicitation

- During requirements elicitation information is gathered about the existing system, current problems and requirements for the future.
- The techniques commonly used in requirements elicitation include interviews, questionnaires, study of documents, observation of people at work and scenarios.
- Interviews are useful for eliciting in-depth information on a one to one basis.
- A plan should be drawn up before the interview with details of the time and place, topics to be covered, and any documents that the interviewee should bring along.
- A questionnaire is useful for gathering a small amount of well-defined information from a wide range of people.
- A scenario records a sequence of interactions between a user and the system that are carried out to achieve a specific goal (e.g. to return a bike).
- In general, scenarios should be written for all the situations relating to the scenario goal, even where the goal is not achieved.

Specification

- During requirements specification the information that has been collected is recorded and documented.
- At this early stage in the development process, the problem definition and the problems and requirements list are the main specification techniques.
- The problem definition is a brief summary of what has been uncovered during elicitation. Typically, sections in the problem definition include problems with the current system, objectives and scope of the new system, preliminary ideas, and recommended action.
- In the problems and requirements list, requirements are recorded more formally and in more detail; this includes a unique identifier for each requirement, the source and date of the requirement, a brief description of it, its priority for development, related requirements and documents, and information about any changes that affect it.

Validation

- During requirements validation the recorded requirements are checked to make sure that they are consistent with what the clients and users want and need.
- Interview summaries, cross-referencing information obtained from different sources, and Fagan inspections are all techniques used to validate requirements.
• A Fagan inspection is a structured method of checking any documented output that is produced during the development process.
Chapter 3

Use cases

- Use cases model what the system does (its functionality) from the user's point of view.
- A use case is an end-to-end use of the computer, a complete path through the system.
- The use case model structures the system into the user's view of its main tasks. It includes a use case diagram, use case descriptions, actor descriptions and scenarios.
- The UML symbols used in use case diagrams are:

  A use case: an ellipse labelled with the name of the use case. Each use case name starts with a verb to make the point that use cases represent processes.

  An actor: a stick figure labelled with the name of the actor. Actor names are capitalized so that they are easy to identify as people or things that interact with the system.

  A use case relationship: a line linking an actor to a use case. The shows us which actors are associated with which use cases.

  The boundary: a line drawn round the use cases to separate them from the actors and delineating the area of interest. The boundary is often omitted.

Identifying use cases

- One way of identifying use cases in a system is to establish what each actor uses it to achieve.
- Scenarios are also a useful basis for use case identification, since each use case represents a group of scenarios that all have a common goal. Each scenario in the group describes a different sequence of interactions that achieve, or fail to achieve, the use case goal.
- The set of scenarios written by the developer should cover:
- a typical sequence of events leading to the achievement of the use case goal
- obvious variations on the norm
- sequences of events where the use case goal is not achieved

Use case descriptions

- The use case description is a narrative document that describes, in general terms, the required functionality of the use case.
- High level descriptions need only document the purpose of the use case, the actors involved and give a general overview of what happens.
- Expanded use case descriptions are more detailed and structured than the high level use case descriptions. They should document:
  - what happens to initiate the use case
  - which actors are involved
  - what data has to be input
  - the use case output
  - what stored data is needed by the use case
  - what happens to signal the completion of the use case
  - minor variations in the sequences of events
  - preconditions (can be omitted)

Actors

- Actors represent people or things that interact with the system and receive some benefit from it.
- An actor description briefly describes the actor in terms of role and job title.

Use case relationships

- The use case relationships are:
  - communication (the line linking an actor to a use case)
  - <<include>> (used when there is a chunk of behaviour that is common to more than one use case)
  - <<extend>> (used to represent significant alternative behaviour in a use case)
  - generalisation (used when one relationship inherits the characteristics of another; another way of specifying alternative behaviour)
- <<include>> and <<extend>> relationships are documented in the use case description using the keyword ‘initiate’.

Use case realisation

- The development of a use case from identification to implementation is called use case realisation.
- The group of classes involved in a particular use case is known as a collaboration.
Working with use cases

- The use case model helps developers to identify classes, produce interaction diagrams, manage the project, check requirements and test the completed system.
- An essential use case is free from detailed design or implementation decisions; a real use case does show design and implementation decisions insofar as they affect the user.
- Use cases can be grouped into packages to simplify a diagram.
Chapter 4

Objects and classes: the basic concepts

Functional decomposition

- In functional decomposition the system is constructed based on the main areas of activity.
- In functional decomposition there is a clear separation between data and process, and data items can often be accessed by any part of the program.
- The problems that can arise from functional decomposition include poor modularity, systems that are not robust, and difficulties with testing and maintenance.

Objects

- A software construct, such as an object, that can be used as the basis for development should be:
  - autonomous
  - cohesive
  - easy to understand
  - easy to adapt
  - based on data

It should also allow encapsulation of data and have a well-defined public interface.
- Since it is based on the concept of an object, object-orientation provides a seamless development process, which allows traceability from requirements to code.
- In the UML an object is represented as a rectangle with two sections. The top section is for the name of the object, and the second section is used for the object’s attribute values.

<table>
<thead>
<tr>
<th>aBike :Bike</th>
</tr>
</thead>
<tbody>
<tr>
<td>type = men's</td>
</tr>
<tr>
<td>dailyHireRate = £8</td>
</tr>
<tr>
<td>deposit = £50</td>
</tr>
</tbody>
</table>

- Object names are always underlined and have two parts, either of which can be used on its own.
- There are a number of ways in which we can refer to an object: e.g. 'an object of the Bike class', 'a Bike object', or ':Bike' — all of these names mean the same thing in this context.
- An object is a concept, abstraction, or thing with clear boundaries and meaning in the current application area.
- We use objects both to model the real-world characteristics of the application area and to provide us with a basis for the computer implementation.
An object is always an abstraction because although we want it to represent a real world thing, we are only interested in certain aspects of the real world thing.

An object has three characteristics:
- behaviour (operations)
- state (determined by the values of its attributes and links to other objects)
- identity (it is unique and has a separate existence from every other object).

Data can be hidden inside an object in such a way that it is protected and cannot be directly accessed by other parts of the program.

Every object has a public interface, which is the only thing the rest of the program knows about the object apart from its name.

The public interface consists of operation signatures.

The public interface provides the services it makes available to other objects.

When one object wants to communicate with another, it does this by sending a message.

For a message to work it must correctly address the object by name and the operation it specifies must be part of that object's public interface.

An object can also have private, internal operations that are not part of the services it makes available to other objects.

**Classes**

A class of objects is a group of objects with the same set of attributes, the same relationships and the same behaviour.

A class is an object factory, it can produce hundreds of objects, all with exactly the same structure and behaviour. When we define a class, we define the structure and the public interface for all objects of that class.

In the software, the code that implements the operations is situated in the class.

The process of creating a new object belonging to a class is called instantiation.

The UML symbol for a class is a rectangle divided into three sections. The top section is used for the class name, the middle section for the attributes and the bottom section for the operations.

<table>
<thead>
<tr>
<th>Bike</th>
</tr>
</thead>
<tbody>
<tr>
<td>bike#</td>
</tr>
<tr>
<td>available</td>
</tr>
<tr>
<td>type</td>
</tr>
<tr>
<td>size</td>
</tr>
<tr>
<td>make</td>
</tr>
<tr>
<td>model</td>
</tr>
<tr>
<td>dailyHireRate</td>
</tr>
<tr>
<td>deposit</td>
</tr>
<tr>
<td>getCharges (no.days)</td>
</tr>
<tr>
<td>findBike(bike#)</td>
</tr>
<tr>
<td>registerBike (bikeDetails)</td>
</tr>
<tr>
<td>getBike# ()</td>
</tr>
</tbody>
</table>
Relationships

- There are three types of relationship between classes:
  - association (models a real-life link between classes)
  - aggregation (models a whole-part relationship between classes)
  - inheritance (models an ‘is-a’ relationship between classes).

- Association is shown by a line joining the classes.

```
| Customer | 0..* | hires | 1..* | Bike |
```

- Aggregation is shown by a line joining the classes with a diamond next to the whole class.

```
Car

| Wheel   | 4   |
| Door    | 2,4,5 |
| Engine  | 1   |
```

- Inheritance is shown by a line joining the classes with an open-headed arrow which points from the specialized class to the general class.

```
Picture

| title   |
| price   |
| updatePrice() |

Photograph

| photographer |
| camera      |
| speed       |
| aperture    |
| alterContrast() |

Painting

| artist  |
| type    |
| owner   |
| printProvenance() |
```
In inheritance, when we create a specialized class, it inherits all the attributes, operations and relationships of the parent class.

Association and aggregation have multiplicity, indicated by numbers and asterisks as follows:

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Example</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>an exact number</td>
<td>e.g. exactly one</td>
<td>1 (or may be omitted)</td>
</tr>
<tr>
<td></td>
<td>exactly six</td>
<td>6</td>
</tr>
<tr>
<td>many</td>
<td>e.g. zero or more</td>
<td>0..*</td>
</tr>
<tr>
<td></td>
<td>one or more, lots of</td>
<td>1..*, *</td>
</tr>
<tr>
<td>a specific range</td>
<td>e.g. one to four, zero to six</td>
<td>1..4, 0..6,</td>
</tr>
<tr>
<td>a choice</td>
<td>e.g. two or four or five</td>
<td>2, 4, 5</td>
</tr>
</tbody>
</table>

More terminology

- Classes that are never instantiated (i.e. no objects of the class exist in the system) are known as abstract classes.
- An operation is a process that an object can perform, part of its behaviour. The word operation refers to the interface of the process. The code that implements the process is known as the method.
- In an inheritance hierarchy, it is possible for a feature in a subclass to use the same name as a feature in the superclass, but to redefine and replace it; this is called over-riding.
- An operation that is implemented by a number of different methods is called polymorphic.
- A subclass object can always be substituted for an object of the class above it in the hierarchy, or indeed for an object of any ancestor class. This is known as substitutability.

Object-orientation and reuse

- Object-orientation offers support for reuse of software as follows:
  - Libraries of classes.
  - The inheritance mechanism allows programmers to tailor library classes to meet the requirements of a new system.
  - Well-designed classes, and compositions of classes are cohesive and easy to understand, so programmers searching a library are able to identify software components to meet their needs.
  - Classes, and compositions of classes encapsulate their internal details so that all that a client component needs to know about is the interface. This helps with the problem of using components originally written for different systems in different languages.
Chapter 5

The class diagram

- The class diagram defines both the software architecture, i.e. the overall structure of the system, and the structure of every object in the system.
- It appears through successive iterations at every stage in the development process.

Use case realisation

- Use case realisation is one way of constructing a class diagram. In use case realisation, the developer looks at each use case in turn and decides what classes would be needed to provide the functionality modelled in the use case.

Constructing a class diagram

- Another way of constructing a class diagram is to develop a domain model, i.e. a class diagram that sets out to model all of the classes in the problem domain in one go, not use case by use case.
- Building a class diagram is essentially an iterative process; no-one, no matter how experienced, gets it right first time.
- Stages in building a class diagram:
  - identify the objects and derive classes;
  - identify attributes;
  - identify relationships between the classes;
  - write a data dictionary to support the class diagram;
  - identify class responsibilities using CRC cards;
  - separate responsibilities into operations and attributes;
  - write process specifications to describe the operations.
- Noun analysis is a useful way of identifying objects and classes. The stages in noun analysis are:
  - Find a complete but concise description of the system requirements (e.g. the problem definition).
  - Pick out all of the nouns and noun phrases and underline them. This usually provides a rather long list of possible (or candidate) objects.
  - Reject unsuitable candidates by applying a list of rejection criteria.
- Objects should be rejected as classes if they are: attributes, redundant, too vague, too tied up with physical inputs and outputs, associations, outside the scope of the system, an operation or an event, a representation of the whole system.
- At the analysis stage, objects will be in categories, such as people, organizations, physical things, conceptual things.
- At this stage of development relationships in the class diagram model real-life relationships that we think may be useful.
Data dictionary

- Detailed information about classes and their attributes is recorded in the data dictionary, which acts as a central repository with agreed terms and their meanings expressed in a standardized notation.
- A typical data dictionary notation is as follows:

<table>
<thead>
<tr>
<th>MEANING</th>
<th>SYMBOL</th>
<th>DESCRIPTION</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>consists of</td>
<td>=</td>
<td>introduces the definition of a data item</td>
<td>Customer =</td>
</tr>
<tr>
<td>and</td>
<td>+</td>
<td>joins components of the definition in sequence</td>
<td>Customer = name + address</td>
</tr>
<tr>
<td>one or more</td>
<td>{ }</td>
<td>attribute may be repeated; any restrictions on the number of repetitions are shown by a subscript</td>
<td>Customer = name + address + {phone}₂</td>
</tr>
<tr>
<td>zero or one</td>
<td>(</td>
<td>attribute is optional</td>
<td>Customer = name + address + {phone}₂ + (email)</td>
</tr>
<tr>
<td>alternatives</td>
<td>[ ]</td>
<td>selection is indicated by enclosing the alternative attributes in square brackets [ ]</td>
<td>Name = [initial</td>
</tr>
<tr>
<td>either.. or</td>
<td></td>
<td>alternatives for selection in [ ] are separated by a vertical bar</td>
<td></td>
</tr>
<tr>
<td>specific value</td>
<td>“ ”</td>
<td>indicates specific values</td>
<td>“individual”, “wholesale”</td>
</tr>
<tr>
<td>*…”</td>
<td>comment</td>
<td>comments are enclosed between asterisks</td>
<td>Customer = name + address + {phone}₂ + (email) + [“individual”</td>
</tr>
</tbody>
</table>

- A data dictionary is structured in the same way as a language dictionary, where items in a definition may have their own entry in the dictionary.

Packages

- Packages provide a mechanism for managing our models by grouping modelling elements.
- The complete class diagram can have many levels; where each level except the bottom one (which will have classes only) can consist of just packages or a mixture of packages and classes.
Chapter 6

Identifying functionality: CRC cards and interaction diagrams

Responsibilities and CRC cards

- A class is regarded as having certain responsibilities to provide services to the user of the system or to another class.
- A responsibility identifies something that a class is expected to do; it is an obligation on the class to provide some kind of service.
- The aim of the CRC card technique is to divide the overall functionality of the system into responsibilities which are then allocated to the most appropriate classes.
- CRC cards are usually index cards about 10 cm. x 15cm. in size, and each card represents one class in the system.
- On the front of the card is a high-level description of the class and the back of the card records the class name, responsibilities and collaborations (if any).
- CRC cards are often used in group role-play. Each member takes the role of an object of one of the classes in the system, and the group then enacts the events that take place during a typical scenario. As each responsibility arising from the scenario is identified, it is allocated to the object of the most suitable class.
- Once responsibilities have been identified, they need to be specified in terms of individual operations and attributes, so that each class has the data and operations needed to fulfil its responsibilities in the way that the user requires.

Interaction diagrams

- Objects collaborate to achieve the functionality required of them by sending messages.
- Interaction diagrams model the messaging between a collaboration of objects that will take place in the execution of a specific scenario.
- There are two types of interaction diagrams: sequence and collaboration. Each shows the same information, but with a different emphasis.

Sequence diagrams

- Sequence diagrams show the flow of control between objects required to execute a scenario. A scenario outlines the sequence of steps in one instance of a use case from the user's side of the computer screen, a sequence diagram shows how these steps translate into messaging between objects on the computer's side of the screen.
- In a sequence diagram the dashed vertical line below each object symbol is called the object's lifeline. It represents the object's life for the duration of the scenario we are enacting. A message is represented as a labelled arrow from one object lifeline (the sender) to another (the recipient). The sequence of messages is read from the top of the page to the bottom, i.e. the time ordering of the messages goes from top to bottom.
• Object *activation* is shown by a thin rectangle on the object's lifeline. An object becomes active as soon as it receives a message. This means that the object is computing; processing is taking place in the object as the invoked operation executes. The activation continues until the operation finishes processing when control returns to the object that sent the message.

**Collaboration diagrams**

• Most CASE tools will automatically generate a collaboration diagram from a sequence diagram or vice-versa.
• In a collaboration diagram links between objects are explicitly modelled, which is not the case with the sequence diagram.

**Operation specifications**

• Operation specifications describe in detail what happens in an operation.
• Techniques used to specify operations include everyday English, specification by contract, structured English, decision trees, decision tables.
• Specification by contract covers:
  - the signature of the operation (its name, any arguments, and the type of values it returns)
  - the purpose of the operation
  - what the client object must provide in order to obtain the required service
  - a description of the internal logic of the operation
  - any other operations that are called by this operation
  - any attributes of objects whose values are changed by the operation.
Chapter 7

State diagrams

- A state diagram models the ways in which the objects of a class behave in response to events that affect them.
- The diagram shows all the possible behaviours of objects of a class, and records the ordering of events.
- A state diagram is only needed for classes where the different ways in which the objects of the class respond to events are complex (i.e. dependent on the state that the object is in).

States and events

- The state of an object is the situation it is in while satisfying some condition or waiting for an event.
- An event is something that happens which has significance for the system and affects an object of at least one of its classes.

Notation for state diagrams

- The symbols used in a state diagram are:

  - state
  - start state
  - stop state
  - transition from one state to another
  - self-transition (no change of state)

  `event / [guard] / action`  transition label (each of the three parts is optional)
• A guard is a condition relating to an event.
• An action is something that the system does in response to an event.
• A state diagram can have only one start state (since all objects of a class are in the same state when created) but many stop states (since the way an object ends its life will depend on the specific series of events that it undergoes).
• A superstate may be used to cater for events that can occur at any stage in the life of an object.

Constructing a state diagram

• The basic steps in drawing a state diagram for a class are:
  - identify the events that affect an object of the class;
  - identify the different states that objects of the class can be in, including the start state and (possibly) multiple stop states;
  - check whether any events that are listed separately should be represented as the same event with different conditions (guards);
  - check whether there are any actions that the system must perform in response to an event; these should be represented as actions;
  - begin to construct the diagram from the start state, the event that creates an object of the class, and the state that the object moves into;
  - build up the diagram, working through the events and states on the list and adding them to the diagram;
  - check that all guards and actions have been included on the relevant transition labels;
  - check whether a superstate should be included to cater for events that may occur at any time during the life of an object;
  - check the completed diagram against the information that has been gathered about the behaviour of the class.
Chapter 8

Activity diagrams

- Activity diagrams show the internal flow of control in a process.
- They are process-based, rather than object-oriented.
- Activity diagrams can be drawn in the initial stages of development to help developers and clients to analyse business workflow processes.
- The diagrams can be used to illustrate the steps involved in achieving a use case goal, showing the activities and the order in which they take place.
- Activity diagrams are a useful means of describing how operations work, particularly when these are based on complex algorithms.
- Activity diagrams can be used to represent sequence, selection and iteration; they can also illustrate where different activities can be carried out in parallel.
- As with all diagrams, it is important to make sure that an activity diagram is easy to read.

Notation for activity diagrams

- The symbols used in activity diagrams are as follows:

  - **activity**
  - **start state; only one allowed on each activity diagram or sub-diagram**
  - **stop state; more than one can appear on the same diagram**
  - **transition from one activity to another**
  - **diamond indicates beginning (branch) and end (merge) of behaviour that depends on certain conditions being satisfied**
  - **condition or guard indicates whether a particular transition will be taken. It is essential that every guard evaluates to true or false, and that the guards on the alternative processing routes are mutually exclusive.**
synchronisation bar indicates start (fork) or end (join) of parallel processing

swimlane indicates which agent, person or object is responsible for a set of activities. They are used when we want to identify who, what, or which object in the system carries out a particular activity. Swimlanes are separated from each other by lines and the top of each is labelled with the name of the person, organisation or object responsible for carrying out the set of activities in the swimlane.
Chapter 9

Design

- The classic distinction between analysis and design is that analysis describes WHAT a system must do and design describes HOW to do it; roughly speaking, analysis decisions are implementation-independent, design decisions tend to be implementation-dependent.
- The design is effectively an abstraction of the final code; it will omit much of the detail in the code but, nevertheless, will provide the essence of the structure and interactions of the programs.
- The product of design activities will be a layered diagram of the system architecture, a set of component and deployment diagrams, a database definition, a test plan and screen and report layouts, a set of detailed class diagrams and supporting documentation such as data dictionary and operation specifications and a set of detailed interaction diagrams.
- The architecture of the system specifies the software and hardware components of the system, how they are structured and related.

Different types of classes

- Entity classes are concerned with modelling the system requirements; they model features of the problem domain.
- Boundary classes model the system's interface with its actors, i.e. with the user or with other systems.
- Control classes control the sequencing of events in use case scenarios.

Visibility

- Visibility means the ability to limit the accessibility of certain features of the model or the program.
- In the UML, any attribute or operation can be declared to be public (+), private (-) or protected (#).

Packages and dependencies

- A package is a UML mechanism for grouping modelling elements. A package itself does not represent anything in the system, but is used to group modelling elements that do represent things in the system.
- Packages can be used to group classes, collaborations, sub-systems or a complete view of the system such as a use case model. We can also use packages to nest models, e.g. in the class diagram. We can have high level class diagrams that show only packages and dependencies – these are sometimes referred to as package diagrams, although this is not a UML term.
- Package diagrams allow us also to specify dependencies between packages. A dependency exists between packages if a change in one can affect the other.
- Localising the effect of dependencies is made much easier if, as far as is possible, we can arrange our classes so that dependencies are one way only.
In a client-server relationship, the dependency is one way – the client depends on the server.

**Layered architecture**

- In a layered architecture, packages are arranged into layers so that each layer only uses the services of the layer below it. This minimises dependencies and the effects of dependencies.
- In a simple four layer architecture might the presentation layer contains a package of boundary or interface classes, the application logic layer contains the control classes, the application layer contains the domain classes (entity and related classes) and the storage layer contains the database and related classes. This is a standard sort of arrangement because generally, interface classes depend on control classes which in turn depend on application classes which depend on database classes.

**Component and deployment diagrams**

- A UML component diagram represents the actual physical software components and their dependencies.
- UML deployment diagrams show the physical arrangement of the hardware elements of a computer system.

**The user interface**

- The user interface is one of the most important aspects of a computer system. General guidelines for designing a user interface include:
  - The system should be consistent
  - User tasks that are boring and prone to error should be minimised
  - Screens should be free of clutter, self-sufficient and self-explanatory.
  - The dialogue with the system should be easy to follow and it should be obvious to users how to navigate their way through the various screens.
  - Feedback should be informative, but not intrusive, so the user always knows what is happening, but is not overwhelmed with details of what the system is doing.
  - The language used in instructions and messages to the user should be clear, concise and free of jargon.
  - There must be adequate user support with clear instructions, messages and comprehensive on-line help.

**Dealing with persistent data**

- Persistent data is data that is stored and remains accessible during the whole life of the system.
- A database stores, organizes and maintains all the data required to support the operations of an organization centrally and in such a way that it can be shared by many different programs.
• An object-oriented database provides all the storage facilities and functionality of a traditional database, but is specifically designed to implement the types of complex data structure frequently found in object-oriented systems.
• It seems unlikely at present that object-oriented databases will replace relational databases for systems with reasonably straightforward data storage requirements.
• The most widely used approach to dealing with persistent data in an object-oriented system is to adapt the object-oriented models to fit a relational database, and to incorporate supplementary code, such as JDBC, to act as an interface between the database and the main program.
• A relational database stores data in tables. Each table represents an entity that is important in the system and about which it needs to store information. The table is made up of rows and columns, where each column stores a field, or attribute of the entity, and each row stores a single record, typically the complete set of values for a single instance of the entity.

Implementing a class diagram in a relational database

• When implementing a class diagram in a relational database, the basic rule is that one class maps onto one table.
• One to many associations can be implemented either by creating a table for each of the two linked classes, and a third table to implement the association, or by creating a table for each of the two linked classes and including a foreign key in the table for the many class.
• For many to many associations separate tables are created for each class and for the association.
• There are three possible ways of implementing an inheritance relationship in a relational database:
  - map each of the classes to a separate table, using a shared number or code to link the tables
  - implement tables for the sub-class or classes only
  - combine all the classes in one table.

Design Patterns

• Design patterns are tried and tested solutions to commonly occurring problems.
• Patterns are not hard-and-fast rules, but descriptions of how particular types of problem may be solved.
• A design pattern belongs in one of three categories, creational, structural or behavioural.
• Patterns are documented using standard templates, providing some or all of the following information:
  - Pattern name
  - Brief description of what the pattern does
  - The problem that the pattern addresses
  - Constraints on the use of the pattern
  - The design of the solution in UML class or sequence diagrams
  - The participants in the pattern (the classes or objects involved)
  - An example of where the pattern has been used successfully
- Frequency of use of the pattern
- Sample code and details of implementation
- Rationale for the pattern, why it has developed in this way
Chapter 10

Designing objects and classes

- Detailed design activities include the detailed specifications of classes, their relationships and interactions.

Classes in the design class diagram

- In an analysis class diagram, most of the classes are entity classes, i.e. they model features of the problem domain.
- A design class diagram introduces more classes. Boundary classes handle the system's interface with the user; they are used, for example, to translate the user's menu selection into a message to an object. Control classes handle the sequencing of events in the execution of the use case.

Association and messages in the design class diagram

- In a design diagram, associations between classes specify how to implement navigable paths between objects of those classes.
- How we implement an association depends on the multiplicity of the association and whether communication is one-way or two-way.
- The direction in which messages are sent is called the navigability of the association. One-way associations are referred to as unidirectional.
- If an object needs to send a message, it must know how to get the message to the recipient. This means it must know the recipient object's identifier. It can do this in various ways.
  - Via a direct link as in an association
  - The calling object can be sent the target object's identifier by another object that has a link to the target object.
  - An object identifier can be passed in as a parameter by a constructor to the object it creates.

Attribute and operation signatures in UML

- The UML format of an attribute signature is:  
  visibility name: type-expression = initial-value
- The UML format of an operation signature is:  
  visibility name (parameter list) : return-type

Detailed design activities

- During detailed design, we look more closely at operations or more precisely at the methods that implement the operations. We design the algorithms that will be used.
During design we focus on the code; we use the interaction diagrams to give an abstract view of what is happening when the program is running.

Chapter 11

The code

This chapter discusses the code written to implement a simple version of the Wheels system. The points made relate to the Java programming language, rather than the general topic of object-oriented development.