Murdoch UNIVERSITY

Data Structures and Abstractions

Introduction to the Unit and UML

Lecture 1

Data Structures

- *". . . It would be more logical first to choose a data structure appropriate to the problem, and then to look around for, or construct with a kit of tools provided, a language suitable for manipulating the structure."* – Maurice Wilkes, Computer Science pioneer and inventor.

Murdoch UNIVERSITY

Contact Details

- Shri Rai
- SC 245.1.019
- s.rai@murdoch.edu.au

Before attempting this unit:

- **Prerequisite knowledge and skills**
  - **This is the third Computer Science programming unit and it assumes knowledge and skills from the previous 2 units.**
  - You should have completed a university level C programming unit within the last 12 months, where you would have done C programming every week for a semester or trimester (at least 12 weeks of C programming)
  - You should have completed a university level Java programming unit in the last six months, where you have written classes and object oriented programs every week for a semester or trimester (at least 12 weeks of Java programming)
  - Written a lot of structured programs in C
  - Written a lot of procedures and functions in C
  - Written procedures and functions with value and variable parameters.
  - Written selection constructs (if and switch statements)
  - Written iteration constructs (for, while and do while loops)
  - Have written code that makes use of arrays of int, char, float, boolean, string.
  - Have written code that does File input/output
  - Written recursive routines
  - Designed and implemented classes and methods in Java
  - Have written code with arrays of objects and/or structs.
  - Know how to test and debug your C and Java programs.
  - You have at least 10 hours a week every week for work on this unit.

Murdoch UNIVERSITY

Before attempting this unit … :

- If you find yourself in a situation where you do not meet any prerequisite knowledge or skill listed earlier, you have two options:
  - Withdraw from this unit/module and build up your knowledge and skills before enrolling again, or
  - Spend additional time and effort in the first two weeks of the unit to build up or refresh your skills and knowledge.

Murdoch UNIVERSITY

The Running of this Unit

- Read the Unit guide to see what you need to do to pass. Do not get caught out at the end and it is too late.
- The term lecture or lecture notes would mean these slides.
- This is very much a *hands-on* unit. Knowledge of theory is embedded in the practical work. You may not pass even if your program is working because you did not take into account the theoretical/design considerations.
  - So don’t hack programs to make them work. Design it right and implement the proper design.
- It is a going to be a very practical unit (including exam), where you experience useful design, testing and programming techniques.
  - The exam is worth half the unit and has a lot of C++ programming questions where you will need to design and implement the required data structures and test them.
- Must not sit back and expect everything to fall in your lap and expect to pass. You are expected to do everything yourself.
  - If students who memorised code pass this unit/module, this unit/ will have a very high pass rate.
  - The only way to fail this unit is when insufficient effort is put in by you.
- You will need pencils/pens and paper at *every* class, both to answer questions and to make annotations. Builds up you ability to solve problems.
- **You will actually be helping yourself if you wrote the code by hand on paper and desk checked/tested the code before coming anywhere near a computer.**

A Note on Lectures, Sessions, Weeks and required preparation.

- A semester/trimester consists of teaching weeks and possibly non-teaching weeks depending on country in which the unit is running.
- Each teaching week is a session and this is the approximate time for the coverage of a topic.
- In this unit, you will have to spend 6-8 hours of productive study and work each week on your own. **[1] //a number in [ ] like this means there are very important notes in the notes pane.**
- There is a lab for each session where you will meet teaching staff.
- Each topic has up to 2 hours lab/workshop, and it is to provide you with help or to assess your work.
- Before coming to the lab, you need (must) to prepare and complete work on your own. [2]
- If you come to any lab without first starting working on your own, you are already behind.
- There is a study group called PASS that runs alongside this unit. You should attend the PASS session as it provides you with additional help and practice.
- Cover the topics/labs in sequence. Do not skip ahead as there are no known short cuts.
- If you do not spend the necessary time on your own on this unit/module, you would have to re-do the unit. [3] [4] [5]
- Attention to detail is critical. [7]

Know this

- What is plagiarism?
- What is cheating?
- What is collusion?
- Can I copy code from the web or a book to use in my assignment?
- What are the assessment components?
- What percentage of the unit is the exam?
- What is a Session?
- What is a Week?
- How much time do you have to spend on your own for each topic?
- Is there a study group for you join?

If you are unsure of any of the above, ask staff.

At The End of this Unit…

- You will know:
  - How to do OO design.
    - Understand and use the S.O.L.I.D principles [1]
    - Appreciate the GRASP principles [2]
  - How to do OO testing.
  - How to code OO in C++.
  - How to change your design.
  - How to refactor code.
  - How to choose a suitable data structure.
  - How to choose a suitable algorithm.
  - How to code in a meticulous, hopefully “bullet-proof” manner.
  - Keep to the specification.

At The End of this Unit…

- You will know how to code:
  - A simple class.
  - A composite class.
  - Class that use other classes
  - A container of the same type of object.
  - A container of pointers to similar objects.
  - Temporary data structures used for processing.
  - Two dimensional containers.
  - Graphs.
  - Algorithms that operate on all of the above.

At The End of this Unit…

- But all the things listed earlier can’t happen by just reading about it. You need to practice – write lots of code.

- The most important advice for the practical work and the unit:
  - Start early.

ICT167/104 Revision

- What does Object Oriented mean?
- What is encapsulation?
- What is polymorphism?
- What is an interface?
- What is a method?
- What is a data attribute?
- What is an API?
- What is an IDE?
- What is a GUI?
- What is the relationship between C and Java?
- Please revise concept material from Foundations of Programming and Principles of Computer Science – we assume that you know and build on the material studied before.

Revision

- Two extremely important concepts for this unit and beyond would be **cohesion** and **coupling**. [1]
  - Cohesion
    - High cohesion is good
      - Easier to understand
      - Easier to test
      - Consequently more reliable and more robust
      - Better re-use
      - We want functional cohesion – single responsibility
    - Low cohesion is bad
      - Consider an example from an ICT167/ICT104 exam – a method that takes two numbers as parameters and returns the sum. [2]
      - Any design/code with low cohesion is bad [2]

Revision

- Coupling
  - A coupling exists if a subroutine (or module or class or component) is dependant on another subroutine (or module or class or component)
  - A change in one can cause the problems for the other.
    - "Jelly-effect" [1]
  - Low or loose or weak coupling is good
    - Aim for this. Modules communicate with each other via data through the parameter passing mechanism.
    - Class objects don't depend directly on other class objects but on interfaces or abstractions. Think of the Java view of what graphics is.
  - High or tight or strong coupling is bad
    - Leads to the "Jelly-effect".
    - Change one thing and other things start to break or not work
- Low cohesion and high coupling would result in violation of the Open-Closed Principle (OCP): [2]
  - Open for extension but closed for modification.
  - OCP is one of the principles in SOLID.

Lets Get Started

- The rest of this material spells out some basic information and ideas that will carry through the rest of the semester.
- Video on OO design and Agile approaches gives a feel for some principles. https://www.youtube.com/watch?v=t86v3N4OshQ (**skip** the first **8** minutes) [1]
- Make sure you understand the materials in the slides and ask questions as needed.
  - I *cannot* guess what you will not understand.
  - There is *no such thing* as a stupid question.
  - If *you* do not understand, I can guarantee that *others* will not: be the sensible one and ask!
  - If you are really that shy, please email me. I will put the question and answer in a targeted QandA file without identifying you.

Murdoch UNIVERSITY

Rules and Standards

- Throughout this unit I will be giving you rules plus design, coding and testing standards that you are required to follow.
- Standards and rules:
  – ensure that code is more readable by others;
  – reduce the incidence of errors;
  – encourage good programming practices.
- Failure to follow these rules and standards will result in significant loss of marks.
  - In an employment situation not following the company's design, coding and testing rules will cost the company, and then it will cost you.
  - So, please follow the standards and rules!!
- In industry you are also required to follow rules and standards – so look on this as preparation for the workplace.
- **Attention to detail is very important. [1]**

Murdoch UNIVERSITY

Design and Coding

- It is possible to insist that all classes, interfaces, etc. are designed in advance.
- However this is may *not the best way (in practice) to build software.*
- This type of design comes from engineering and building, where it is *very* difficult to change your mind half way through building something.
- However in programming, changing the structure (refactoring) part of the way through a program is relatively easy.
  - The drawback is that it can encourage sloppiness.
  - The advantage is that if you find a better way of doing something, you can use the better way.

Murdoch UNIVERSITY

Design and Coding [1]

- *Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it. —Alan Perlis*

- *But I also knew, and forgot, Hoare's dictum that premature optimization is the root of all evil in programming. —Donald Knuth, The Errors of TeX [Knuth89]*
  - "Design Style" in the unit reference book: C++ coding standards: 101 rules, guidelines, and best practices by Herb Sutter, Andrei Alexandrescu
  - http://prospero.murdoch.edu.au/record=b2249386~S10

Murdoch UNIVERSITY

Balance

- When you code there are several things that must be considered:
  - Code readability (**correct, simple and clear**, and thus maintainability)
  - Operating speed of the code
  - Memory usage in RAM
  - Memory usage on Disk
  - Speed of writing the code
  - The total number of lines of code
- It is *never* possible to optimise all of these at once.
- In this day and age, the order given above is considered the best order.
- In other words, your code should be readable and maintainable above all else, and writing code in only a few lines is the least important thing.

Balance [1]

- *Stop and think of some random good software engineering technique— any good technique. Whichever one you picked, in one way or another it will be about reducing dependencies.* (coupling) [2]
  - *Inheritance? Make* (client) *code written to use the base class less dependent on the actual derived class.*
  - *Minimize global variables? Reduce long-distance dependencies through widely visible data.*
  - *Abstraction? Eliminate dependencies between code that manipulates concepts and code that implements them.* (independent of implementation – code relies on the public interface by information hiding)
  - *Information hiding? Make client code less dependent on an entity's implementation details. An appropriate concern for dependency management is reflected in avoiding shared state, applying information hiding , and much more.*
- *Our vote for the most valuable Item in this section goes to Item 6:* ***Correctness, simplicity, and clarity come first. That they really, really must.***

Balance

- “Design Style” in the unit reference book: C++ coding standards: 101 rules, guidelines, and best practices by Herb Sutter, Andrei Alexandrescu http://prospero.murdoch.edu.au/record=b2249386~S10 [1]:
  - *Rule 6. Correctness, simplicity, and clarity come first. Summary KISS (Keep It Simple Software): Correct is better than fast. Simple is better than complex. Clear is better than cute. Safe is better than insecure.*
  - *It’s hard to overstate the value of simple designs and clear code. Your code’s maintainer will thank you for making it understandable— and often that will be your future self, trying to remember what you were thinking six months ago. Hence such classic wisdom as:*
  - *Programs must be written for people to read, and only incidentally for machines to execute. —Harold Abelson and Gerald Jay Sussman*
  - *Write programs for people first, computers second. —Steve McConnell The cheapest, fastest and most reliable components of a computer system are those that aren’t there. —Gordon Bell*
  - *Those missing components are also the most accurate (they never make mistakes), the most secure (they can’t be broken into), and the easiest to design, document, test and maintain. The importance of a simple design can’t be overemphasized. —Jon Bentley*

Murdoch UNIVERSITY

Incremental Programming

- So in OO programming you should:
  1. Do some overall design: at the least a UML diagram
  2. Choose a class to code [1]
  3. **Test plan for this class** [2]
  4. Design the class
  5. Code a **little**
  6. **Test what has been coded so far – all of it** [3]
  7. Refactor as necessary
  8. Go back to (4)
  9. Change the documentation if necessary
  10. Go back to (2)
- This is called incremental or iterative programming.

Agile Programming

- The agile programming concept used in industry is based on this simple incremental coding technique.
- This unit becomes **much** easier if you design, code and test incrementally.
- It becomes much *harder* if you cut out the design and test portions, [1]
  - or if you try to do detailed design all in advance.
- The agile methodology may not be suitable for a particular task but that is outside the scope of this unit.

Minimal But Complete

- All routines (and classes) should be coded in a minimal but complete manner. [1]
- This means that there are some methods that *must* always be included in a class (without them the class is not **complete**).
- Then there are methods that are *required* for a specific application, however these should be kept to a minimum and only added in when needed (the **minimal** class concept).
- Every method must be tested, so *never* code a method until you are sure it is needed.

Data vs Databases

- All data *can* be stored in a database.
- However, relational databases:
  - Require an additional language (e.g. SQL);
  - may require higher memory and disk space;
  - do not readily adapt to the OO concept.
- There is a separate unit for the study of databases.
- So we will be using ordinary (text) files within which to store our data on disk. We need to be able to do file I/O at an early stage in the unit.
  - In fact, you should already know how to do file I/O in your earlier units. In this unit, you will learn the C++ approach.
- We will still learn the data structures used in a database.
- And our own data structures to store them within RAM.

C++ vs Java

- The two languages are very similar.
- The main differences relevant to this unit are:
  - In Java the main program is a class, in C++ it is not: instead it is a main program like in C (similar to ict159/ict102)
  - In Java the class interface and class methods (code) go in the same file. In C++ the API (specification) goes in a header file (.h) and the code goes in a separate implementation file (.cpp). The files must have the same prefix by convention. For example circle.h and circle.cpp. Must do this separation otherwise penalties will may be imposed. [1]
  - C++ has pointer and non-pointer data structures available to the programmer. Pointers are necessary for the building of many important data structures as they provide very fine grained control over how memory is used.
    - This ability comes at a cost – it is dangerous. [2]
    - Obscure bugs are hard to track down.
  - Data structures in Java rely entirely on pointers so the distinction is not available to the programmer.
  - Java does its own garbage collection but the C++ programmer is responsible for writing code to release memory back to the OS when working with raw pointers.

Unit Tests

- In OO a unit is a class.
- Unit testing, for our purposes is therefore the name given to the testing of a single class.
- **Obviously a class must be fully tested before it is used in the main application program or by another class. [1]**
- Therefore every class has its own test program (main program)
- This test program is run whenever the class is changed.
- So as well as the header file (.h) (the API) and an implementation file (.cpp) there will be a test file (.cpp) for *every* class.
- For example: circle.h, circle.cpp and circleTest.cpp. [2]
- Unit Test Plans (a description of what needs testing in the unit test program) are an important part of the assessment in this unit.
- The application test is separate from the unit tests as the application test has to test the actual application that is given to the end user.
  - So an application test plan is also needed.

User Interfaces

- From our point of view you can have two possible user interfaces:
  - Console (terminal)
  - Graphical User Interface (GUI)
- The second of these is, of course, the one used within Windows.
- Of course, you might have full (one day) direct brain-computer interfaces but this is not relevant for this unit.
- However GUI coding can be a unit in itself: it is difficult and requires a vast amount of skill to get right, unless you are using visual programming tools. VB, Delphi…
  - GUI programming is outside the scope of this Data Structures unit.
- Behind every GUI is a set of data classes.
- These must be *separate* to the GUI classes, so that they and the GUI can be easily changed. Look up the model-view-controller paradigm.
- The data classes are what this unit concentrates on.
- To avoid getting distracted by GUI programming, all our output will be done within a Console interface.
- However, for ease of programming we will use the Codeblocks and/or the Visual Studio IDE. [1] [2]

Why Object Orientation

- You need to know why we have Object Orientation. This informs the approach we are taking in the unit.
- The required information is found in: Chapter 1 and 2 “The Object Oriented Paradigm" in the library ebook “**Design Patterns Explained**” by Shalloway and Trott. *https://prospero.murdoch.edu.au:443/record=b3142219~S10* [1]
  - Go through the first two chapters carefully. [1]
  - Objects are given responsibility for their own behaviour. So in OO, you work out who (which object) is responsible for what.
  - Objects are instances of a given class, so a class will define the responsibilities of the instances.
  - The class’s public interface will indicate that class's responsibilities.
- Introduction to OO design principles – **SOLID** https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

Murdoch UNIVERSITY

Important Reading

- If any link to the library content is broken, look for the material using My Unit Readings at http://library.murdoch.edu.au/.
  - **If there is any issue with the library links or library content, please contact the library first.**
- Aside from the first video listed in the previous slide, the following reading informs the approach we are taking in the unit. The required information is found in:
  - Chapters 1 and 2 “The Object Oriented Paradigm" in the library ebook “Design Patterns Explained” by Shalloway and Trott. [1] *https://prospero.murdoch.edu.au:443/record=b3142219~S10*
- Revise **Separate Interface and Implementation, Structs vs Classes** in “Absolute C++. Pages 284-293” by Walter Savitch. https://content.talisaspire.com/murdoch/bundles/5873062446 9eef8318b4571 [2]

Videos to Watch

- Start the theory part of this unit using this video of a presentation by Robert Martin on OO and Agile Design [1] https://www.youtube.com/watch?v=t86v3N4OshQ (**skip** the first 8 minutes). Covers a few SOLID principles so you get a feel for what these are and the video is by the author of the principles that become know as SOLID principles.
- Bucky's C++ tutorials: (you will need for the labs if you prefer to lean by first watching videos) [2] https://www.youtube.com/watch?v=tvC1WCdV1XU&list=PLAE85DE8440AA6B83
- Stanford University series of videos starting from Lecture 2 is aimed at 1st years who know little or nothing about C++. Lecture 1 is mostly admin. Some similarity in course content but at a much more basic (1st year) level at Stanford. [3]
  - https://www.youtube.com/watch?v=wmiD5J8Dw9E&list=PLFE6E58F856038C69&index=2

Overview

- **Before starting here, you need to have gone through the ideas and concepts introduced in Lec-01-intro.ppt.**
- We will look at the use of the Unified Modelling Language (UML) for Object Oriented (OO) design.
- We will be referring to "UML Distilled" by Fowler and Scott.
- UML has it's beginning with works by James Rumbaugh, Grady Booch and Ivar Jacobson. Each of them had their own way of "doing things" but started to collaborate. [1]
- If you end up working with UML in industry, this book is a 'must'.
- Fowler also has a useful website: http://martinfowler.com/
- We will be using **StarUML** to draw UML diagrams; it is available in the labs… maybe not all labs but as it is free you can install it yourself.
- If you want to draw them at home it can be downloaded free from http://www.it.murdoch.edu.au/units/ICT209/staruml/ [2]

Murdoch UNIVERSITY

UML for Design

- The most important design tool for OO is the Unified Modelling Language (UML). The Object Management Group oversees development of UML.
- We will be using UML class diagrams extensively.
- You will be using **StarUML** to draw these diagrams. This is not to say that these diagrams can’t be drawn by hand or by some other tool like Microsoft Visio, ArgoUML, BOUML, .. etc. [1]
- UML shows the relationship between classes.
- It can also show the attributes and methods in a class.
- An initial UML diagram (version 01) is bound to be incorrect, so it gets updated as you change the design (versions 02, 03...).
- It forms the main type of class documentation for this unit.

Standards in UML

- UML is supposed to be "unified" and therefore "standard".
  - Ok, but not everyone plays by the rules ...
- Even finding out exactly what the standards are is difficult.
- And the standards have alternatives in them!
- And, of course, then you have to find software that supports the full standard.
- On top of which, the standards rely very heavily on symbols not words, whereas a word can be clearer.
- So we will stick as "close" to the standards as we can, given that we are using StarUML and we want clear, useful diagrams.

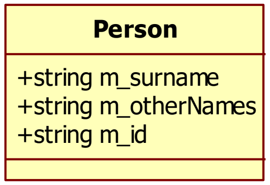
Murdoch UNIVERSITY

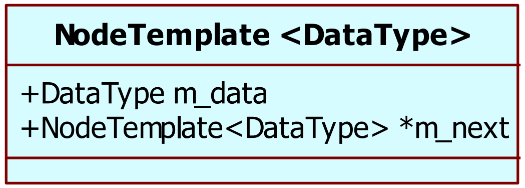
Class Types

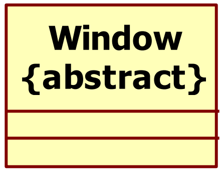
- There are many different types of classes.
- Basic or simple classes are those whose member variables are all of types that are built into the language, for example int, float, bool, string (if available).
- Composite classes are those that have at least one member variable that is of another class type.
- Interface classes are those that have no data members and no implementation. They simply define an interface (set of methods) that other classes must implement.
- Abstract classes are those that provide an interface, but which may also have some data members and some implementation.
- Template classes have data and a complete implementation, however the *type* of the data is not *instantiated* until compile-time.
  - Note that a type that is described using a template and an instantiating type is known as a bound element.

Our UML Standard for Classes

- Simple and composite classes will just be labelled with the class name
- Template classes will be shown with the template parameter in a dashed box.
- Interface and abstract classes (and all other classes with *qualifiers)* will have the class name followed by the qualifying information in {} braces if you are not planning to generate code automatically.





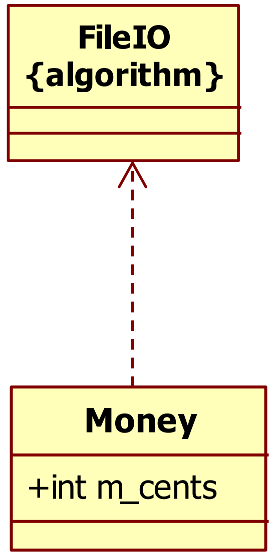


Class Relationships

- There are many different potential relationships (associations) between classes.
- Coupling is introduced when classes are related.
- In this unit we will examine:
  - dependency
  - composition
  - aggregation
  - specialisation
  - realisation

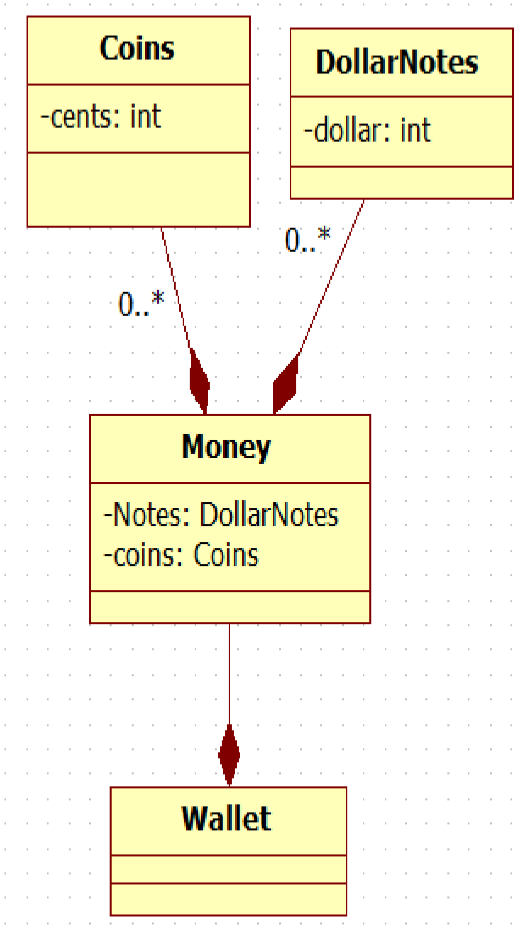
Dependency

- Described as 'needs', 'uses', 'requires'. As Fowler [1] puts it "A dependency exists between two elements if changes to the definition of one element (the supplier) may cause changes to the other (the client)"
- It is used when one class *uses* another within its code.
- For example, writing to a binary file is messy, so quite often one encapsulates the functions for binary file writing, into a class containing only algorithms. (FileIO).
- In UML dependencies are shown with a dotted line and a simple arrow. In this example 'Money' depends on 'FileIO'.
- Note that the FileIO class has the qualifier {algorithm} indicating that it has no member variables, only algorithms.
- You draw this relationship if the other relationships are not appropriate but there is a dependency, via, say, parameter passing, or method call.
- If a class contains another class as a data member, there is a dependency, but you indicate the dependency using an encapsulation (or aggregation) relationship and not this dotted arrow notation.



Composition

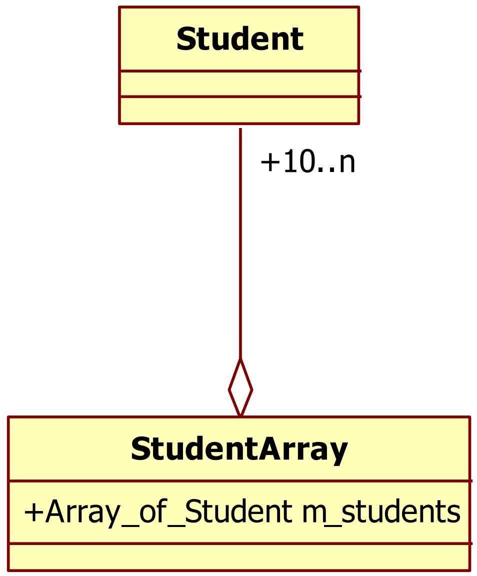
- Conventionally described as ‘has a’.
- One class has a data member that is the type of another class.
- *And* when **the object disappears, all its parts usually disappear too.**
- For example a wallet *has a* set of Money amounts.
- Money consists of Notes and Coins
- This relationship is shown in UML with a solid line and a solid diamond at the container end.
- If the ‘whole’ contains more than one of the same ‘parts’ a *multiplicity* should be included.
- The multiplicity can be a single number (5), a defined range (0..4), or an open-ended range (0..*).
- StarUML allows the connection to be labelled using the attribute name used to represent money inside MoneyArray.
- If you loose your wallet, you loose your money.
- With money, you can give away coins or notes.
- There are other ways to model the relationship.



END

Aggregation

- Described as ‘has a’.
- One class has a data member that is the type of another class.
- However, when the whole disappears, the parts **do not**.
- For example an array of students in a unit. When the array disappears, the students do not.
- This relationship is shown in UML with a solid line and an open diamond at the container end.
- As with composition, there should be a multiplicity if the relationship is not 1 to 1.



END

Specialisation

- When a new class has all the characteristics of another class, but then adds information, it is called a specialisation.
- We *derive* a *child* class from a *parent* class and then add to the child.
- Sometimes the relationship is expressed differently and we say the parent class *generalises* the child class.
- **One class is a specialisation of another class if and only if:**

1. You can relate them in English using 'is a' in a behavioural sense [1]
   For example, 'a car *is a* wheeled vehicle'.

   and

2. The child class uses *every* member variable declared in the parent class.

   and

3. The child class requires *every* method defined in the parent class.

Specialisation

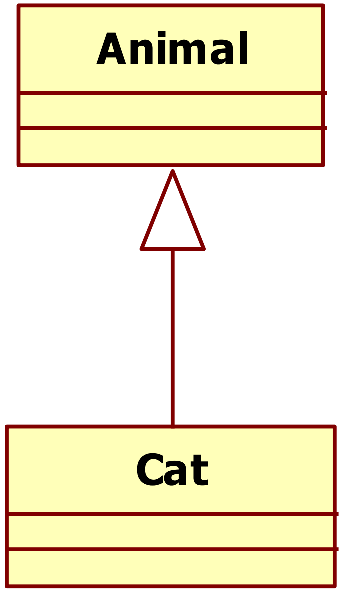
- If you don’t follow all of these rules, your design will violate the **Liskov Substitution Principle (LSP) – the “L” in S.O.L.I.D principles.**
- *“**If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.**”*
- “S is is a subtype of T” means S is a child of T; or T is the parent of S.

- For correct parent-child relationship to occur, a program written in terms of objects of type T (the parent), you can substitute these objects of T (parent) with objects of the subtype S (child) without changing program behaviour.
  – If the behaviour changes, then there is **no** reliable parent child relationship between types T and S. If you have designed an inheritance relationship between Class T and Class S, you have made a mistake.

Murdoch UNIVERSITY

Specialisation cont.

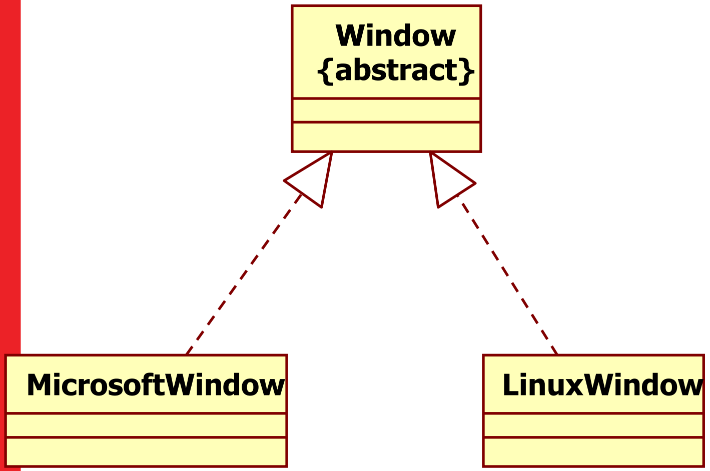
- Examples of correct specialisation:
  - a cat *is an* animal
  - a car *is a* transport vehicle
- Example of **poor** specialisation:
  - a square *is a* rectangle (but it has no width, so it should *not* be derived from rectangle)
- Specialisation (generalisation in StarUML) is shown with a solid line and a hollow triangular arrow.
- In C++ specialisation is achieved by having the child *inherit* from the parent class.
- Arrow head is with the parent.

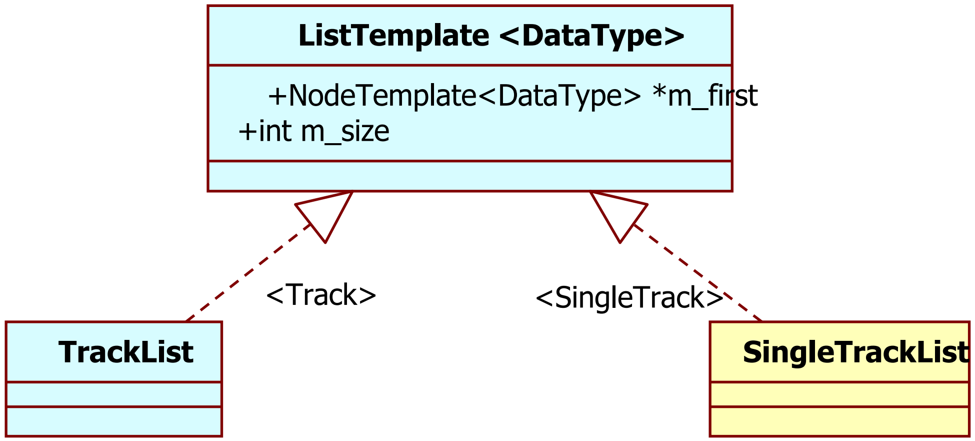


END

Realisation

- Abstract, template and interface classes define all or part of a class.
- None of these types of classes can be used until they are *realised* as a completed class.
- For example an abstract class might be set up called a *Window*, this could then be *realised* as a Microsoft Window or a Linux Window. [1]
- In UML realisation is shown with the same arrowhead as specialisation, but with a ***dotted*** *line*.
- Where it is a realisation of a template, the DataType qualifies the relationship [2]





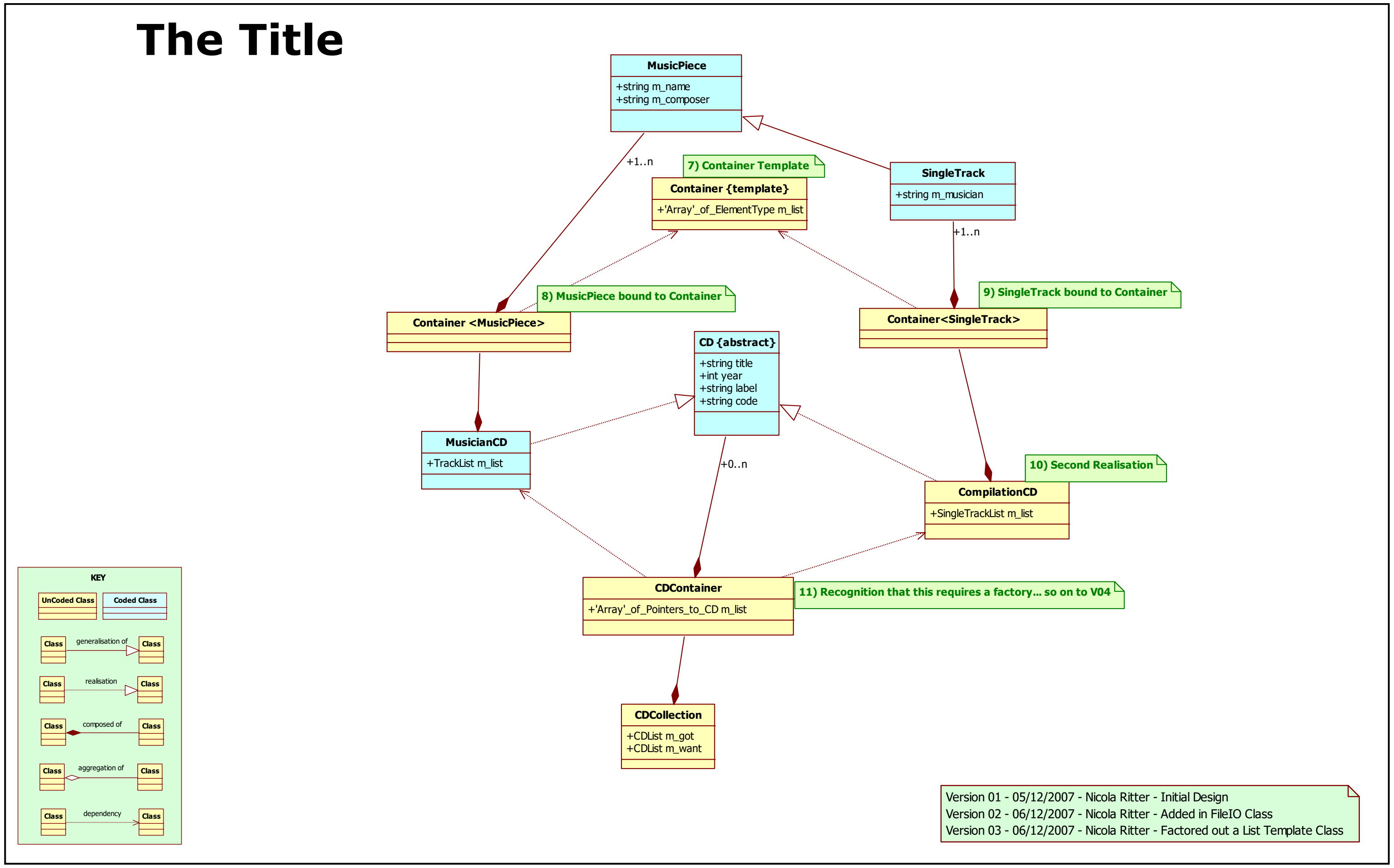
Our Application

- Lecture example (next slide set) shows how to code an application to store data about a CD music collection.
- I have chosen this because it is not too difficult and it is possible to cover almost everything in the unit. But we will be looking at other code too.
- The labs would be using different code and the unit textbook uses other examples, so you get exposure to more than just one or two examples.

Diagram Basics

- All diagrams (whether UML, ERD etc) should have:
  - a title
  - a key
  - version history; for each version
    - the version number
    - the date of the version
    - the author of the version
    - brief description of the changes
- All neatly laid out on the diagram.
- With useful use of *pale* colours as appropriate.

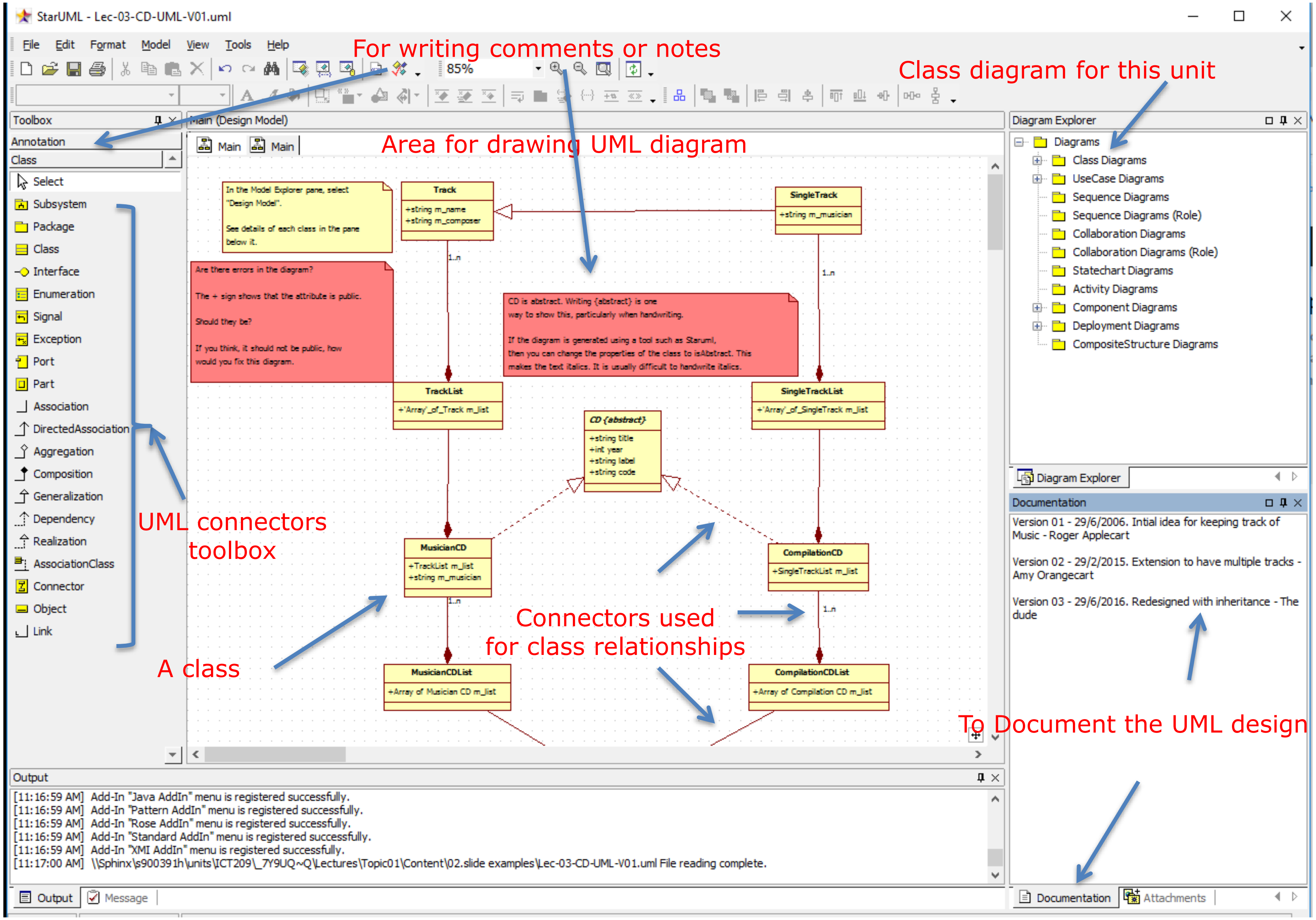
Example Layout



Notes on Diagram Layout

- Try to keep classes in line vertically and horizontally.
- Try to ensure that lines do not cross.
- If the key and history are too large then they may need to go on a separate page, however this risks them getting separated from the main diagram.
- If everyone is using the same software and the software has a key down the side, then parts of the key can be left out, and the history may be able to go elsewhere.
- This, of course, applies in this unit.

Murdoch UNIVERSITY



In the Model Explorer pane, select "Design Model".

See details of each class in the pane below it.

Are there errors in the diagram?

The + sign shows that the attribute is public.

Should they be?

If you think, it should not be public, how would you fix this diagram.

CD is abstract. Writing {abstract} is one way to show this, particularly when handwriting.

If the diagram is generated using a tool such as Staruml, then you can change the properties of the class to isAbstract. This makes the text italics. It is usually difficult to handwrite italics.

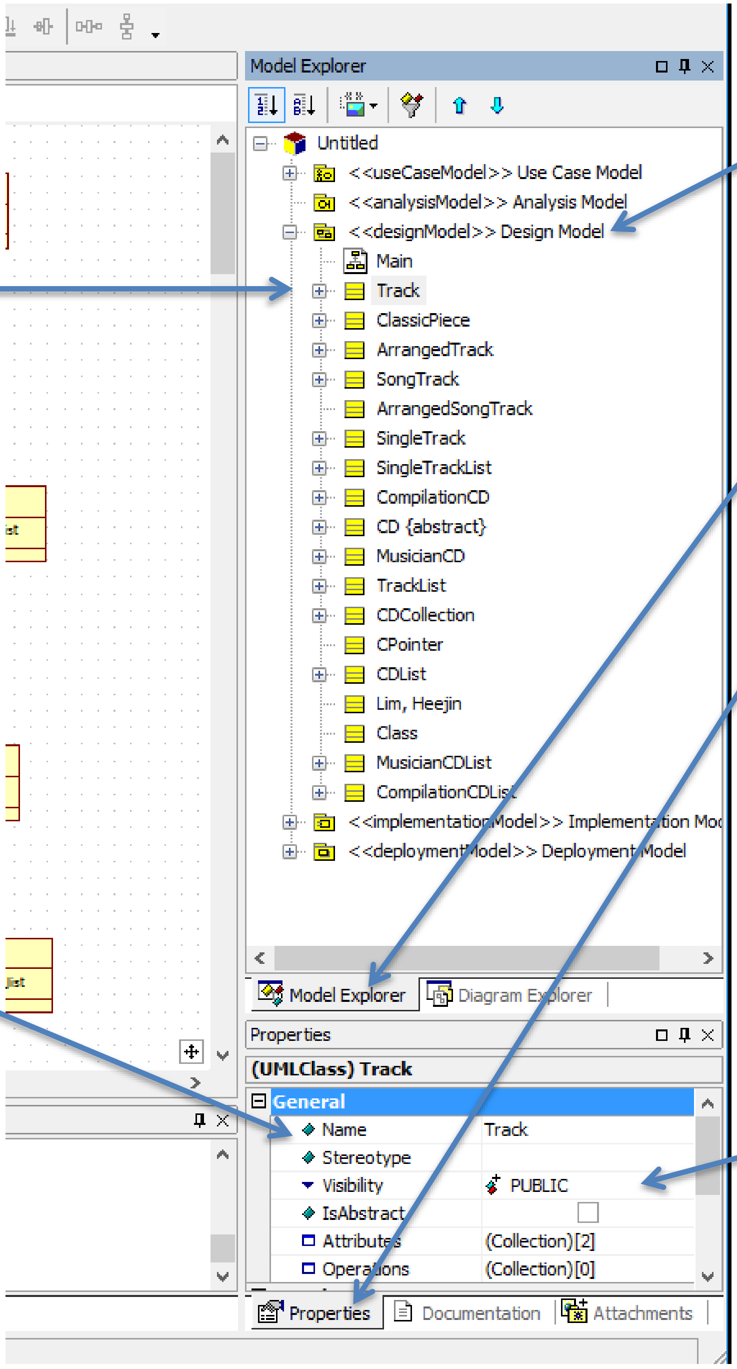
Version 01 - 29/6/2006. Intial idea for keeping track of Music - Roger Applecart

Version 02 - 29/2/2015. Extension to have multiple tracks - Amy Orangecart

Version 03 - 29/6/2016. Redesigned with inheritance - The dude

```
[11:16:59 AM]  Add-In "Java AddIn" menu is registered successfully.
[11:16:59 AM]  Add-In "Pattern AddIn" menu is registered successfully.
[11:16:59 AM]  Add-In "Rose AddIn" menu is registered successfully.
[11:16:59 AM]  Add-In "Standard AddIn" menu is registered successfully.
[11:16:59 AM]  Add-In "XMI AddIn" menu is registered successfully.
[11:17:00 AM]  \\Sphinx\s900391h\units\ICT209\_7Y9UQ~Q\Lectures\Topic01\Content\02.slide examples\Lec-03-CD-UML-V01.uml File reading complete.
```

StarUML interface



Model Explorer

- Untitled
  - <<useCaseModel>> Use Case Model
  - <<analysisModel>> Analysis Model
  - <<designModel>> Design Model
    - Main
    - Track
    - ClassicPiece
    - ArrangedTrack
    - SongTrack
    - ArrangedSongTrack
    - SingleTrack
    - SingleTrackList
    - CompilationCD
    - CD {abstract}
    - MusicianCD
    - TrackList
    - CDCollection
    - CPointer
    - CDList
    - Lim, Heejin
    - Class
    - MusicianCDList
    - CompilationCDList
  - <<implementationModel>> Implementation Mo
  - <<deploymentModel>> Deployment Model

Model Explorer | Diagram Explorer

Properties

(UMLClass) Track

General

Name	Track
Stereotype	
Visibility	PUBLIC
IsAbstract	☐
Attributes	(Collection)[2]
Operations	(Collection)[0]

Properties | Documentation | Attachments

To view/set detailed properties

Selected class

Properties of the selected class

The properties

Readings

- **UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition**
  ebook in library. Lecture notes link keep breaking as the links keep changing.
- **Data Abstraction and Hierarchy by Barbara Liskov.** Original paper describing what became knows as the Liskov Substitution Principle (**LSP**). For an overview, see https://en.wikipedia.org/wiki/Liskov_substitution_principle
- **LSP** is one of the principles of SOLID. This is something that you need to become comfortable with at the end of the unit, and this knowledge is directly relevant to anyone working in a software development role. For an overview see https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)) and the references provided there.

Murdoch UNIVERSITY