Game Design Patterns
01 - Introduction

Administrivia

- Dr Andy Guest
- a.guest@yorksj.ac.uk
- Tel – (1904) (62)6524

Assessment

- Group Assessment or Not?

- Game Demo
- Game Design Report
  - Focusing on design patterns

Module Structure

- Software Engineering Review
- Introduction to Design Patterns
- General Design Patterns
- Gaming Design Patterns
- Unity

01 – Reviewing Software Engineering

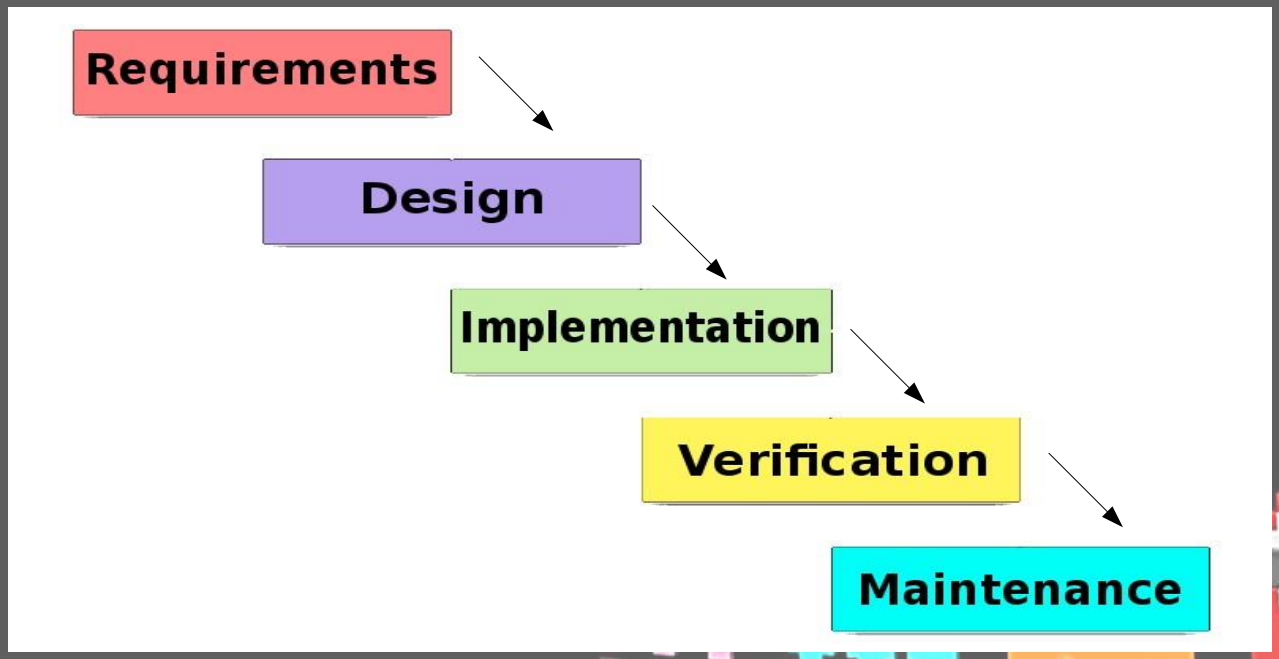
- Tradition Software Engineering
  - The Waterfall Model
- Agile Software Engineering
  - Iterative Approach
- Object Oriented Analysis & Design
- UML

Software Engineering

- According to Wikipedia
  - Software engineering (SE) is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software
- Software gets complex very quickly
- Need a way to control the chaos, especially in team projects

The Waterfall Model

- Structured, layered approach to software development

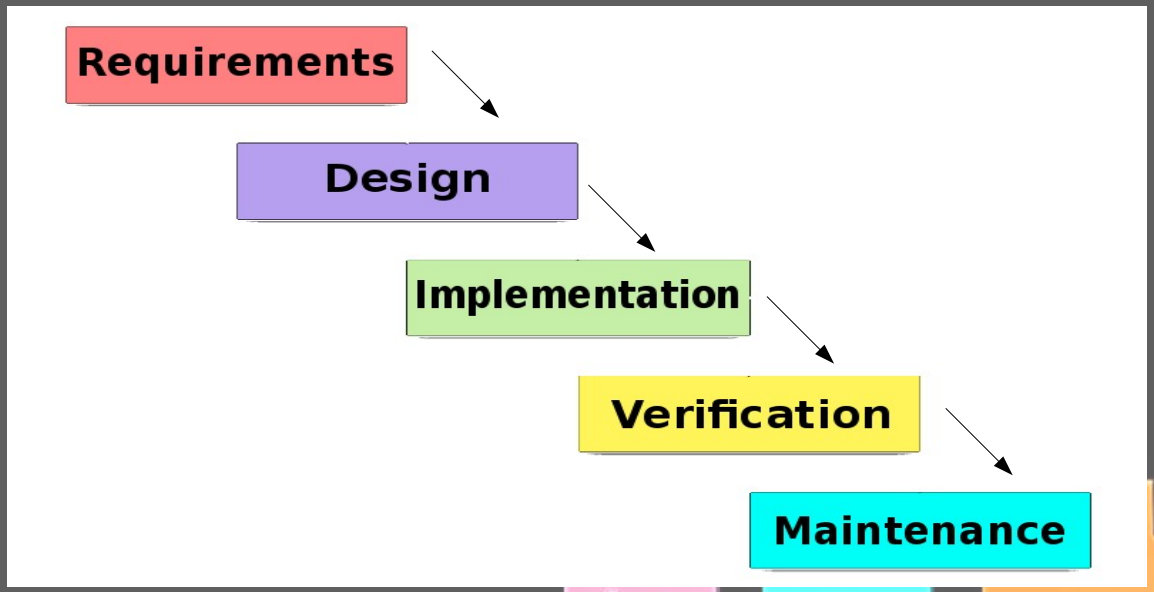


The Waterfall Model

- Leads to “Big Design Up Front” approach
- Plan everything before starting to code
- Design down to detailed pseudo-code
- Coding is just tweaking pseudo-code to code
- Works well if everything can be done like a traditional engineering build
- But this ain't what happens...

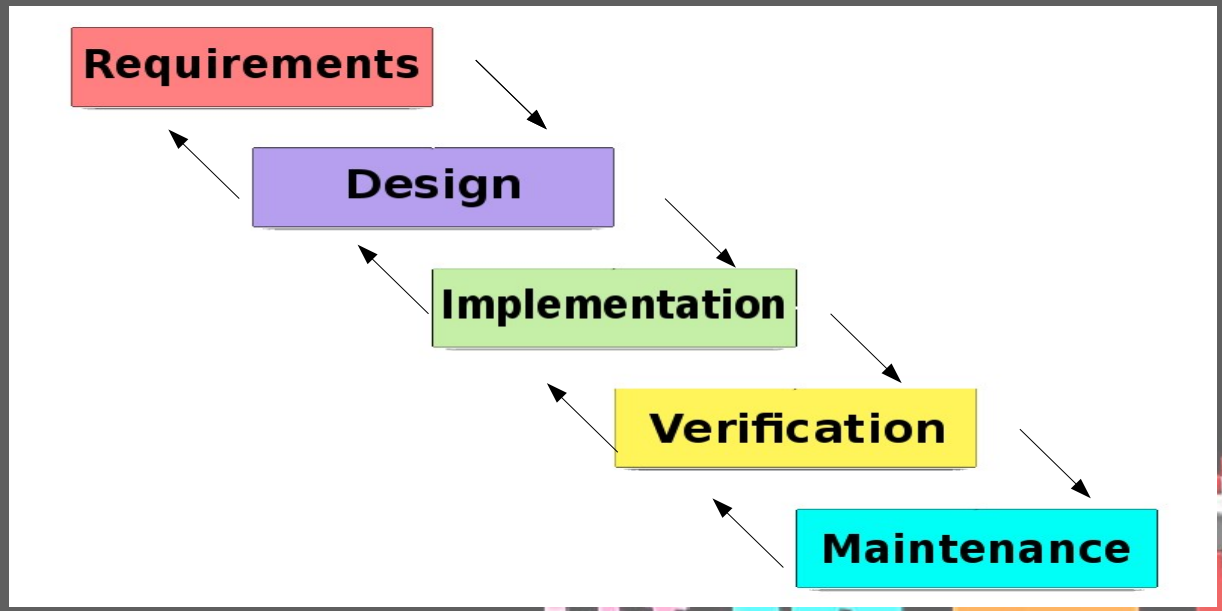
The Waterfall Model

- The theory



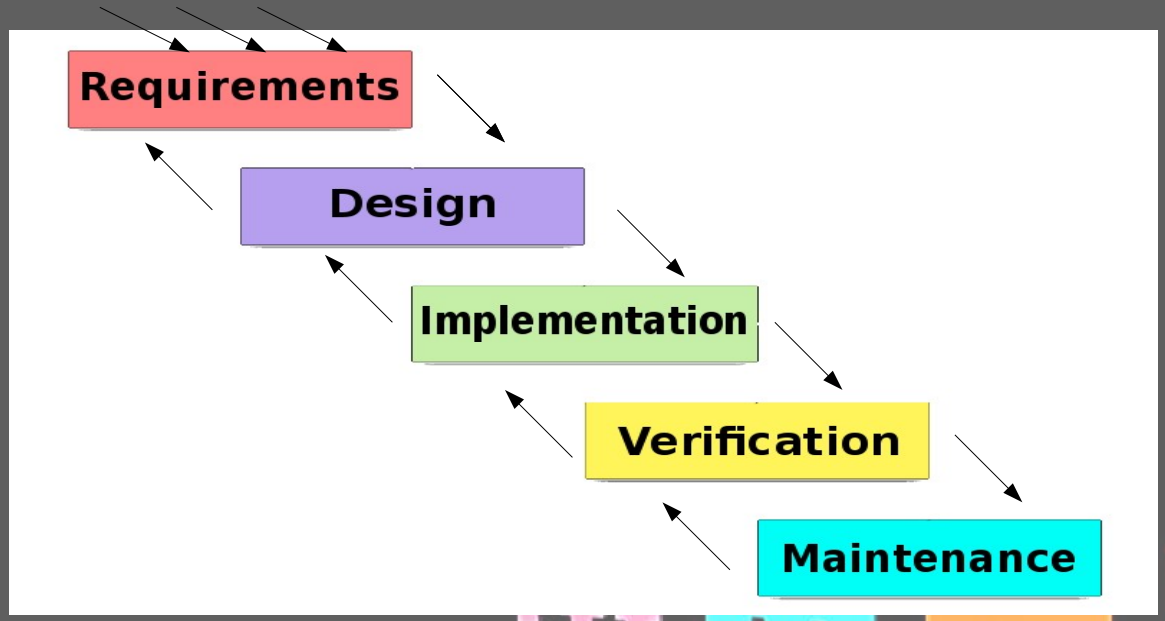
The Waterfall Model

- Problems lead to revisions



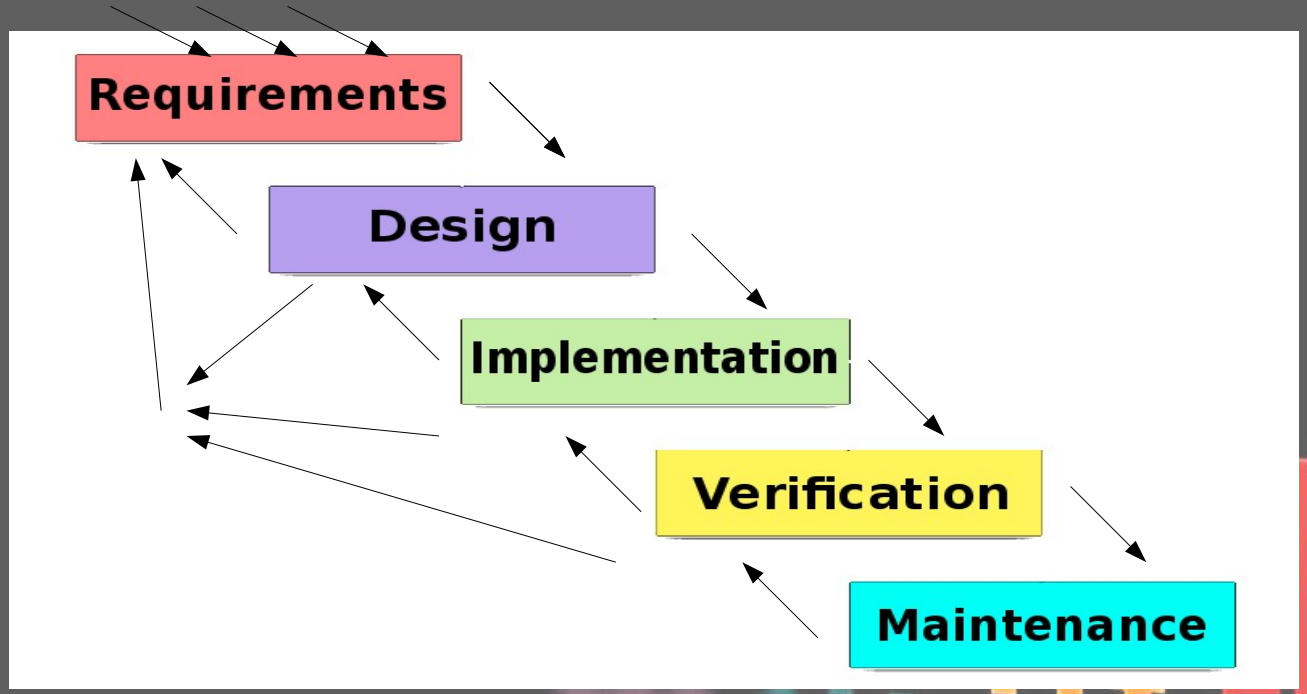
The Waterfall Model

- Customers forgot to mention, or analysts failed to notice



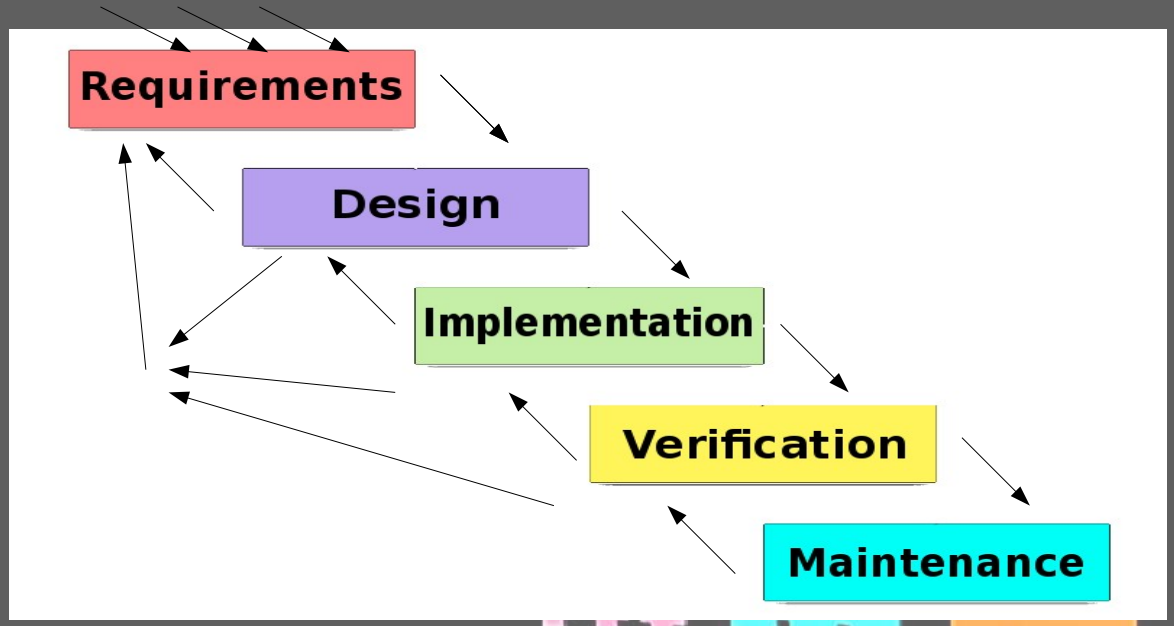
The Waterfall Model

- Serious flaws discovered, faulty premise discovered



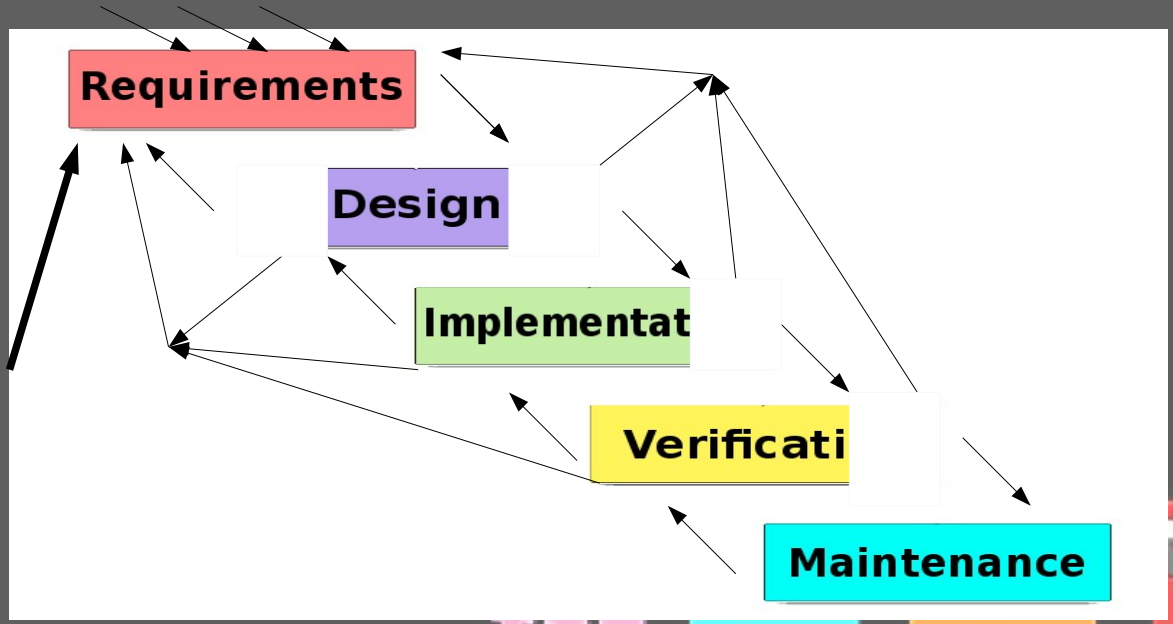
The Waterfall Model

- Alpha/Beta testing makes customers realise they want something else



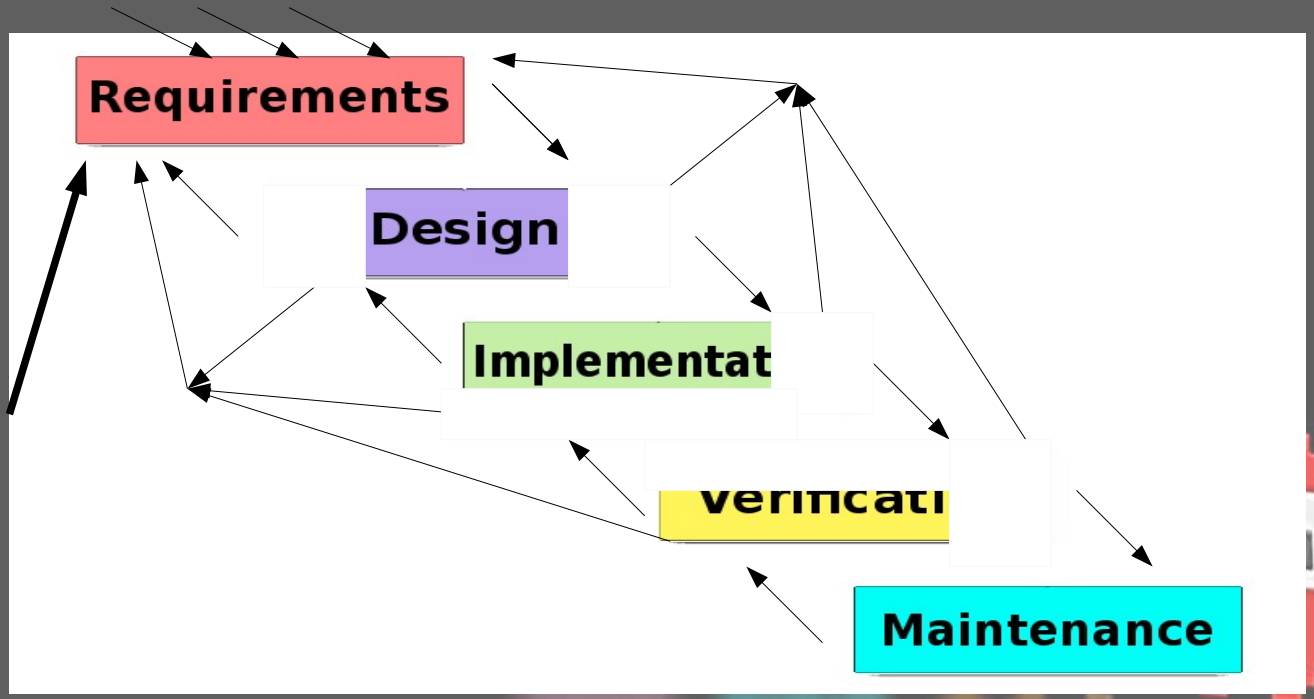
The Waterfall Model

- Time pressures



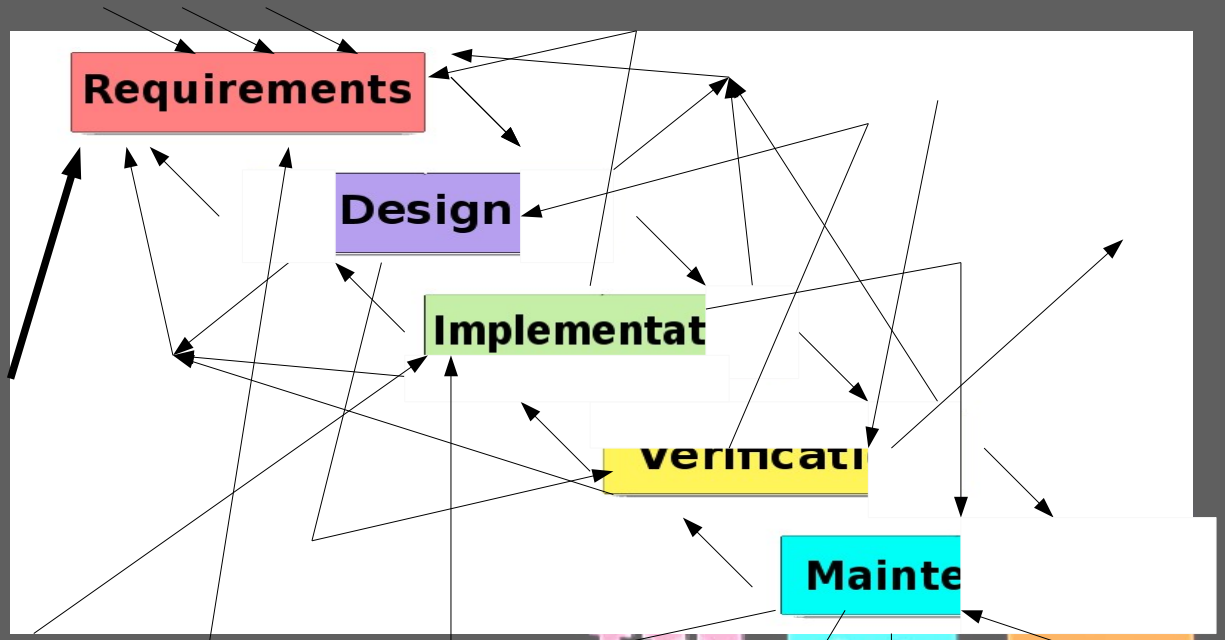
The Waterfall Model

- Delays in one stage cause cuts in later ones



The Spaghetti Model

- Things get a little out of control



Why Use It?

- Coding is the smallest and in many ways the least significant part of the process
- Much more time is spent gathering information, designing and testing than coding
- The waterfall model is simple and effective in other engineering disciplines
- It can work well when everything is clear
- Everyone can understand it

Why Does It Fail?

- There is a lot of paperwork
  - It can't be done and forgotten about it needs to be updated
  - Any change needs to flow through all the documents
- It makes coding the smallest, simplest, least important role
  - We coders don't like that
- It isn't flexible when changes come along

Agile – Modern Software Engineering

- A response to traditional software engineering
- To address the problems and difficulties
- Agile is philosophy, a set of values, rather than a process

Agile Manifesto

- **Individuals and interactions** over process and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

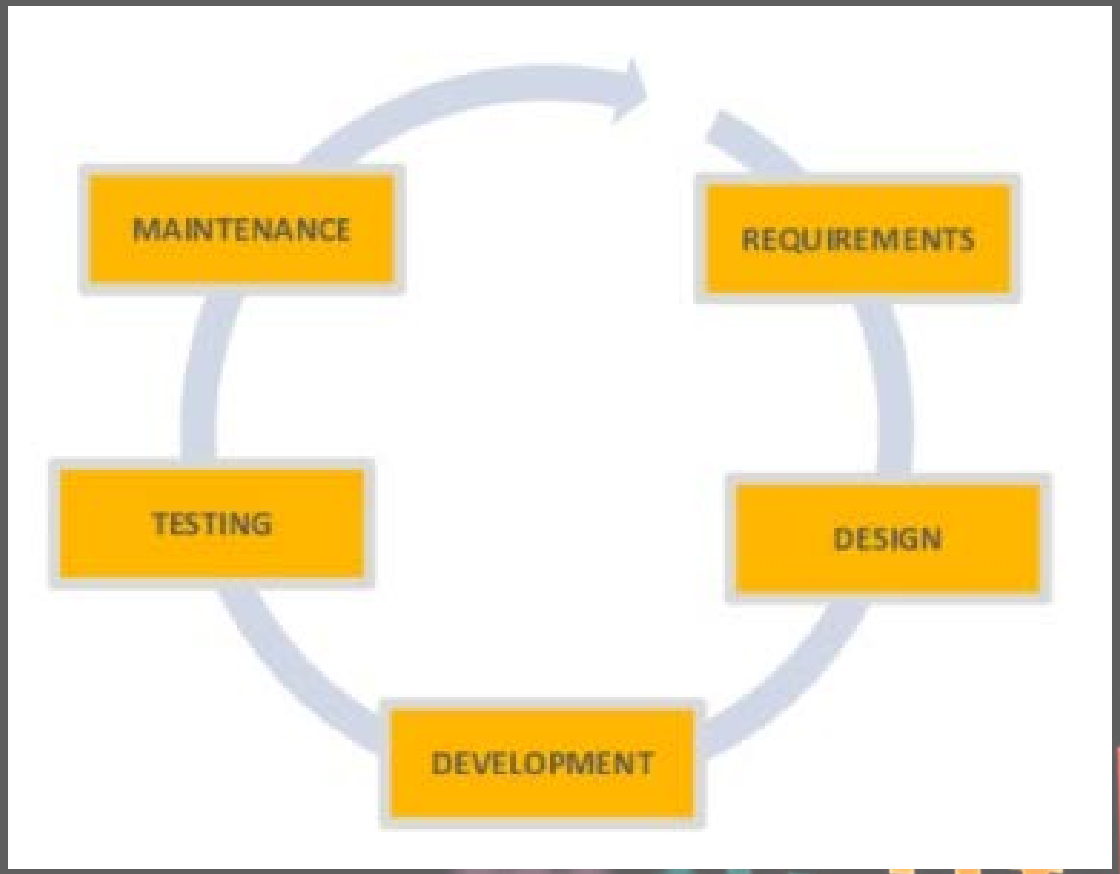
Agile Umbrella

- Scrum, Crystal, Kanban, FDD, XP, DSDM, RUP
- Range in complexity/the number of rules to follow
  - 0 – Do whatever!
  - 3 – Kanban
  - 9 – Scrum
  - 13 – XP
  - 120+ - RUP

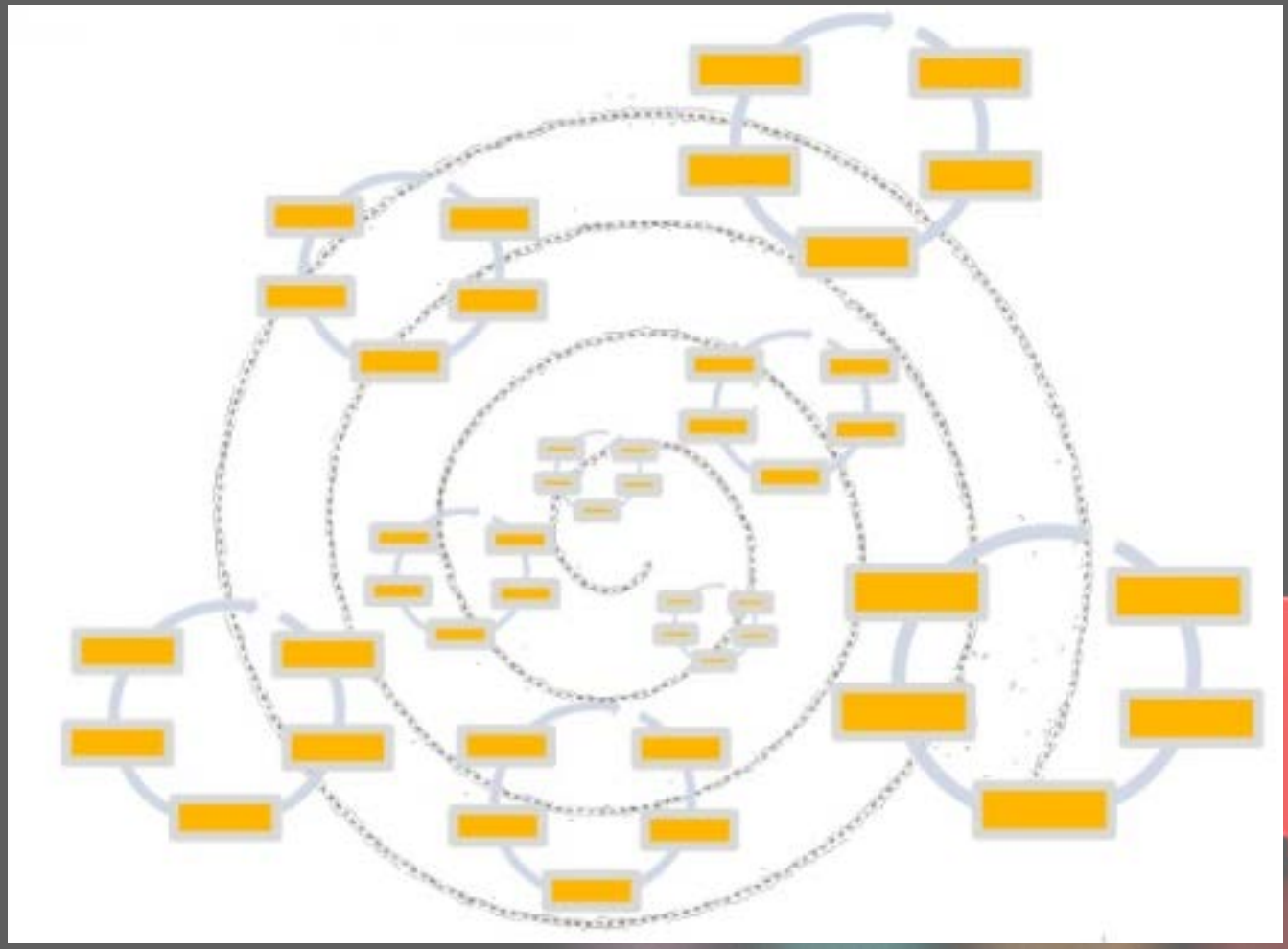
Agile Approach

- Break down code in to small features
- Work on one feature at a time
- Work in short **sprints**
- Work iteratively
- Regularly review feature/work list
- Always have a working build

Iterative Approach



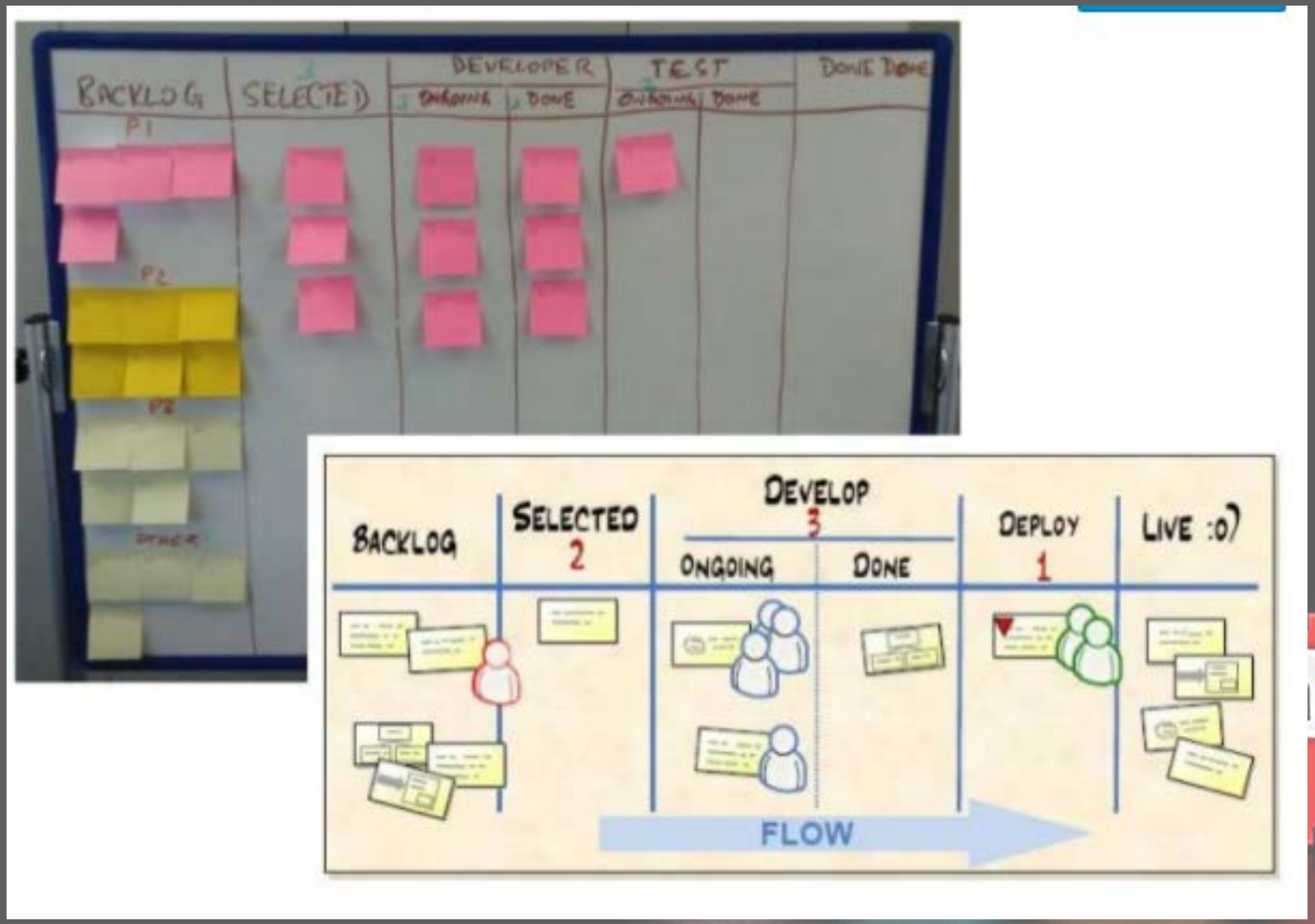
Iterative Approach



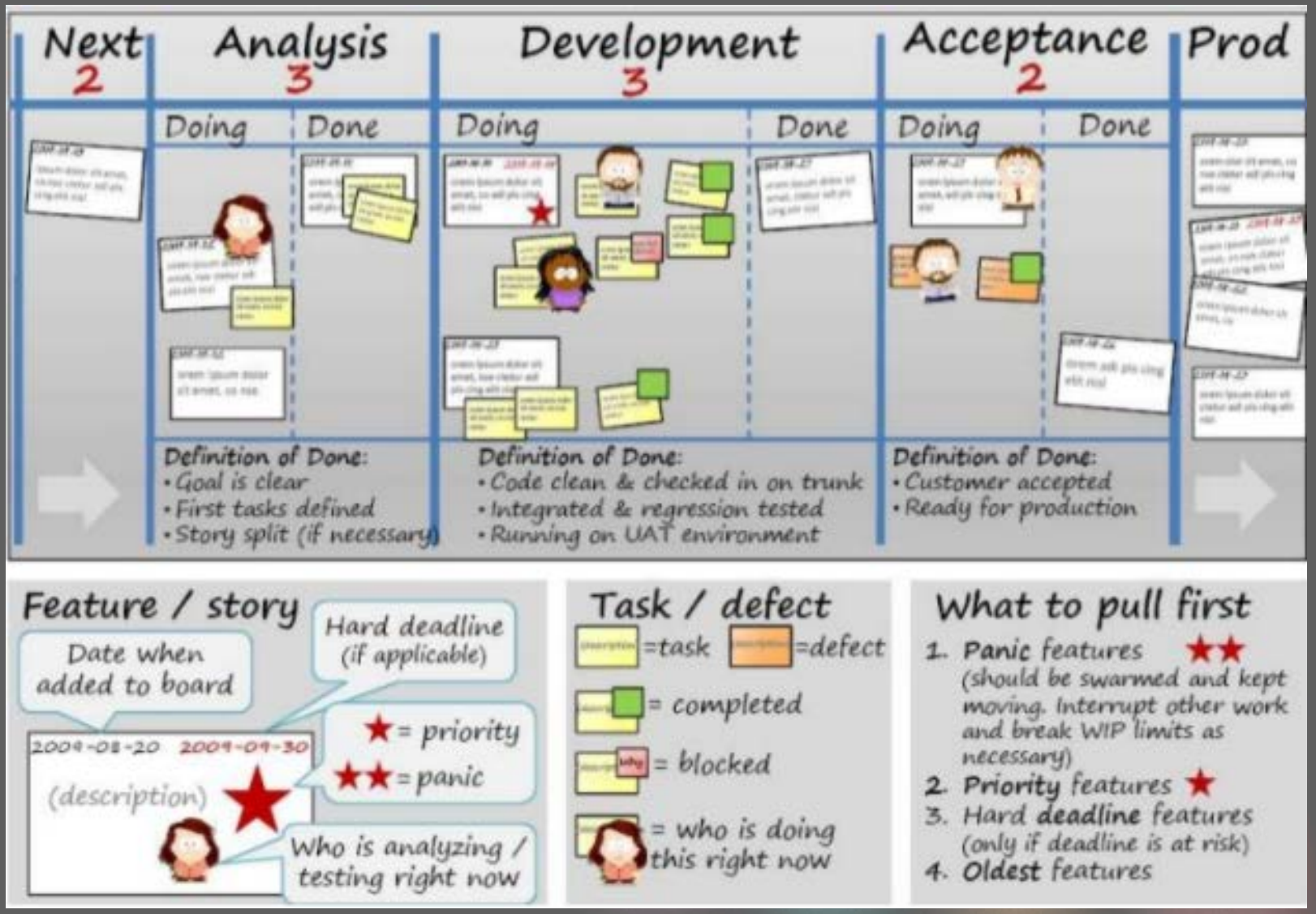
Scrum In A Nutshell

- Instead of a **large group** spending **a long time** building a **big thing**, we have a **small team**, spending a **short time** building a **small thing**, but **integrating regularly** to see the whole.

Kanban Board



Kanban Board



OO Analysis & Design

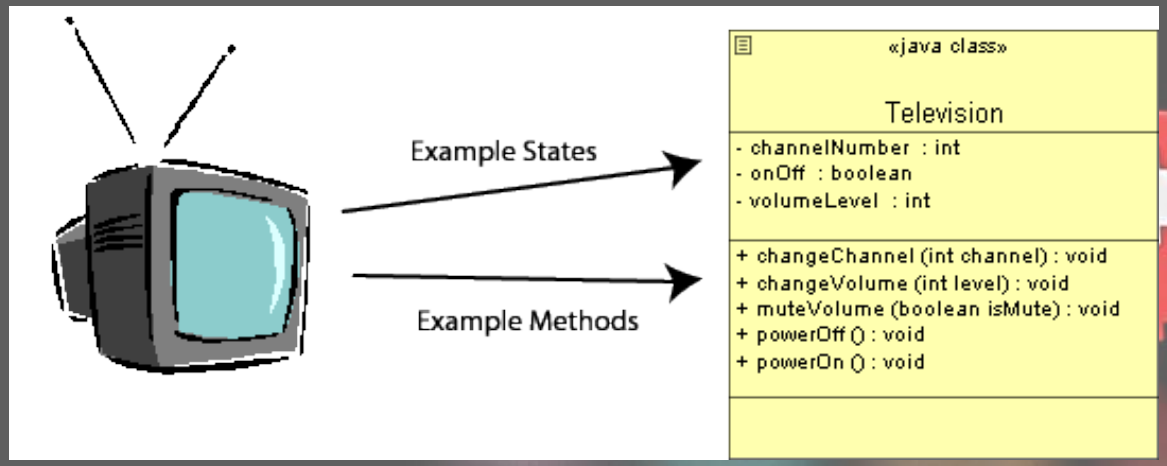
- The object-oriented approach encourages:
  - Modularisation: where the application can be decomposed into modules.
  - Software re-use: where an application can be composed from existing and new modules.
- An object-oriented programming language generally supports five main features:
  - Classes
  - Objects
  - Classification
  - Polymorphism
  - Inheritance

Classes

- A class should
  - Provide a well-defined interface - such as the remote control of the television.
  - Represent a clear concept - such as the concept of a television.
  - Be complete and well-documented - the television should have a plug and should have a manual that documents all features.
  - The code should be robust - it should not crash, like the television.

Classes

- Classes allow us a way to represent complex structures within a programming language. They have two components
  - **States -** (or data) are the values that the object has.
  - **Methods -** (or behaviour) are the ways in which the object can interact with its data, the actions.
- We use UML (right) to illustrate classes



Objects

- An **object** is an instance of a class
  - a *class* is the description of a concept
  - an *object* as the realisation of this description to create an independent distinguishable entity
- Objects can be
  - concrete (a real-world object, a file on a computer)
  - conceptual (such as a database structure)

Object Oriented Analysis

- To write object oriented code we first need to analyse the game description, thinking in terms of
  - Classes
  - Objects
  - Data
  - Methods

Identifying Classes & Objects

- If you haven't been provided with a text description of the game/software then write one
  - It should describe what the player/user does, is trying to do and why
  - You should pay attention to *things* and *actions*
  - It should describe the game/software at a high level allowing someone unfamiliar with it to understand the concept
  - You may want to start at a high level and describe the types of core functionality and then later on repeat the process by focusing on each core function in turn and going in to more detail

Identifying Objects & Classes

- Take your description and
  - Look for *nouns*
    - *Proper noun* – Alice, Ace of Hearts – describes an *object* (incidence of class)
    - *Common noun* – Field officer, playing card, value – describes a *class* (or *attribute*)
  - Look for *verbs*
    - *Doing verb* – creates, submits, shuffles – describes an *operation*
    - *Being verb* – is a kind of, is one of either – describes *inheritance*
    - *Having verb* – has, consists of, includes – describes *aggregation*, *composition*
    - *Modal verb* – must be – describes a *constraint*
  - Look for *adjectives*
    - *adjective* – a *yellow* ball (i.e. colour) – helps identify an *attribute*

Identifying Objects & Classes

- Examine your list of *nouns*
  - Separate them out in to classes, objects and attributes
  - Which attributes are part of which classes?
  - Which objects are instances of which classes?
- Examine your *verbs*
  - Which *doing verbs* attach to which classes ?
  - Which *being verbs* describe a relationship between which classes?
  - Which *having verbs* describe relationships between which classes?
  - Which *modal verbs* describe constraints on classes?
  - Which *modal verbs* describe constraints on objects?
  - Which *modal verbs* describe constraints on attributes?
  - Which *adjectives* describe attributes of which classes?

Identifying Objects & Classes

- Take your description and reading it again looking for
  - **Entity objects** – represent persistent information in the system
    - Account, player, boss monster
    - Look for terms that are domain specific, recurring nouns, real world entities and activities
  - **Boundary objects** – represent interactions between user and system
    - Button, form, display, movement keys
    - Identify controls that initiate a use case, data entry forms and windows, system messages to user
  - **Control objects** – represent the management / control of an activity
    - If a use case is complex and involves many objects, create a control object to manage that case
    - Identify one control object per actor in a use case
    - Life span of control object should last through the use case

UML – Unified Modelling Language

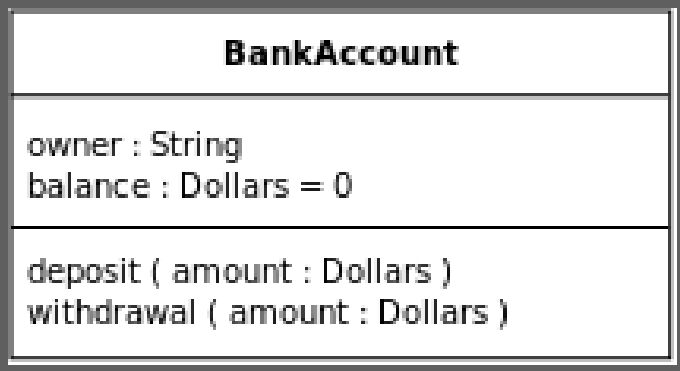
- UML is a general purpose, developmental modelling language intended to provide a standard way to visualise the design of a system
- Consists of many types of diagrams
- Right now we are concerned with class and object diagrams
  - But others may also be useful
    - Use cases describe user interactions with the system / game
    - Sequence diagrams describe the interactions between objects in time
    - State diagrams describe the various states a system (or subsystem) may be in and how changes between states occur
    - Activity diagrams describe workflows step by step (flowcharts)

Class Diagrams

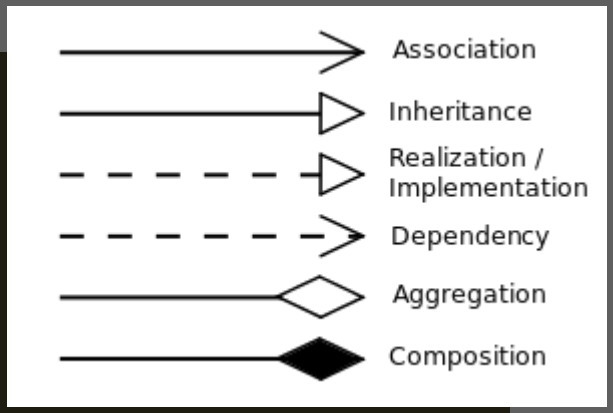
- Class diagrams show the relationships between classes
- These are static
  - They do not show the creation and destruction of objects
  - A class diagram is true for a system no matter where you are in the execution of that system
    - As opposed to an object diagram which is a snap shot of the state of objects at a moment in time and may change constantly
- They show inheritance, composition, aggregation of classes

UML Class Description

- Each class is shown in a standardised form with three sections
  - Name
  - Attributes
    - Data stored as part of the class
  - Operations
    - Class functions
- Attributes and operations may be described as
  - + Public, - Private, # Protected



UML Class Relationships



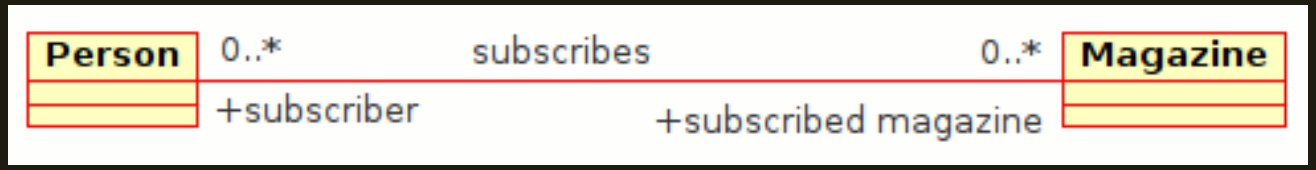
- In class diagrams lines drawn between classes describe
  - the relationship between those classes
- Instance Level Relationships
  - **Association** – shows a connection between the members of two classes
    - **Aggregation**

    implies a relationship where one object can exists independently of the other
    - **Composition**

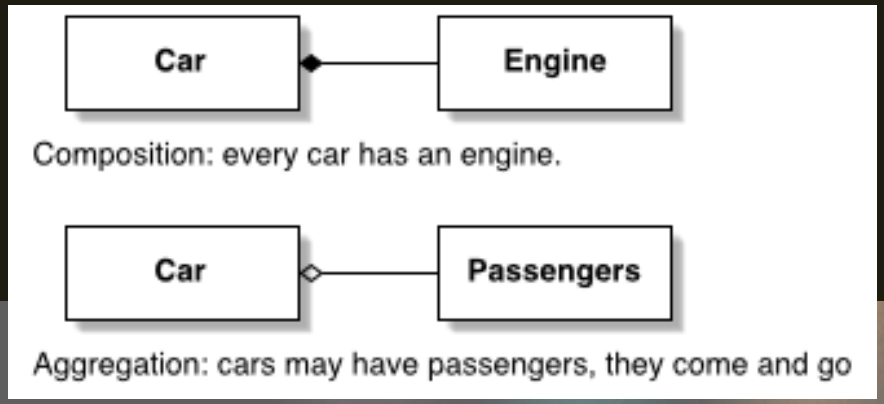
    implies a relationship where one object cannot exist independently of the other
  - **Inheritance** – shows a parent / child relationship
  - **Realisation** – shows the implementation of an interface
  - **Dependency -** a relationship where one class uses another by parameter or return type

UML Associations

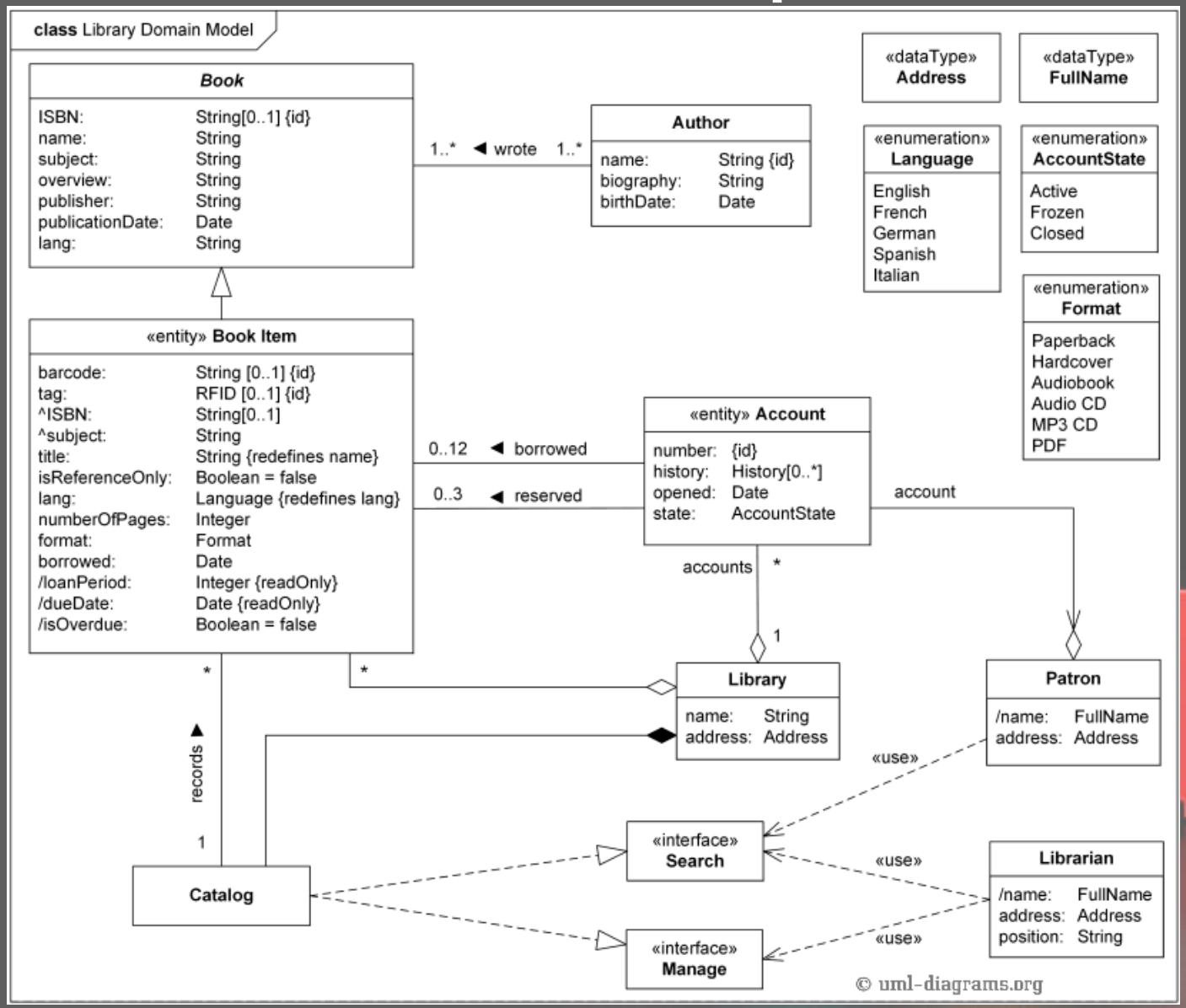
- **Association** is a link between members of two classes.



- **Aggregation** – the two classes can exist independently
- **Composition** – one class cannot exist independently of the other (within the system)



UML Example


class Library Domain Model
Book
ISBN: String[0..1] {id}
name: String
subject: String
overview: String
publisher: String
publicationDate: Date
lang: String
1..* wrote 1..*
Author
name: String {id}
biography: String
birthDate: Date
«dataType» Address
«dataType» FullName
«enumeration» Language
English
French
German
Spanish
Italian
«enumeration» AccountState
Active
Frozen
Closed
«enumeration» Format
Paperback
Hardcover
Audiobook
Audio CD
MP3 CD
PDF
«entity» Book Item
barcode: String [0..1] {id}
tag: RFID [0..1] {id}
^ISBN: String[0..1]
^subject: String
title: String {redefines name}
isReferenceOnly: Boolean = false
lang: Language {redefines lang}
numberOfPages: Integer
format: Format
borrowed: Date
/loanPeriod: Integer {readOnly}
/dueDate: Date {readOnly}
/isOverdue: Boolean = false
0..12 borrowed
0..3 reserved
«entity» Account
number: {id}
history: History[0..*]
opened: Date
state: AccountState
account
accounts
*
1
records
1
Library
name: String
address: Address
Patron
/name: FullName
address: Address
«use»
Catalog
«interface» Search
«interface» Manage
Librarian
/name: FullName
address: Address
position: String
© uml-diagrams.org

Design Patterns

- Design patterns represent the best practices used by experienced object oriented software developers.
- Design patterns are solutions to general problems that software developers faced during software development
- These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

Summary

- Reviewed Tradition Software Engineering
  - The Waterfall Model
- Reviewed Agile Software Engineering
- Reviewed Object Oriented Analysis & Design
- Review UML

Next Week

- Design Patterns Overview
  - Concept
  - Structure
  - Usage
- Design Patterns Examples