COMP 1531
Software Engineering Fundamentals

Week 01

Course Introduction

COMP 1531
Software Engineering Fundamentals

LiC: Aarthi Natarajan
a.natarajan@unsw.edu.au

Web: http://webcms3.cse.unsw.edu.au/COMP1531/

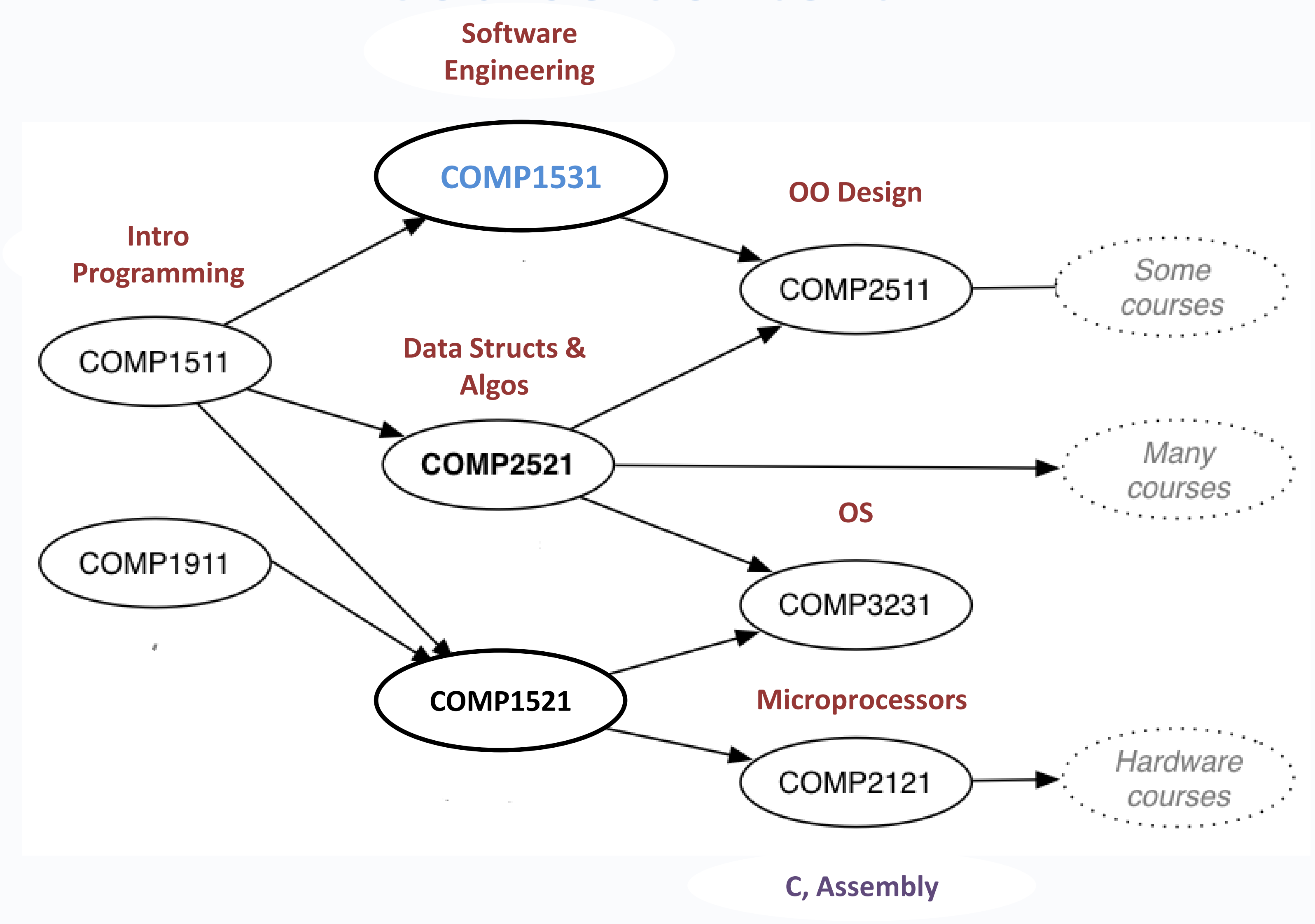
Course Admin:
Ian Park/Rob Everest

About me…

Background:

- 16 years enterprise software development (IBM, Westpac, RTA…)
- 8 years instructor (Oracle Education)
- Email is the best way to contact me
- I believe in interactive lectures, so please ask questions!

Course Context



Pre-requisites

- Completed **COMP 1511**

- Learnt fundamental C programming and are familiar with:
  - variables
  - data types
  - loop structures
  - defining and using functions and returning results

COMP 1531 students

- **COMP 1511**
  - Gets you thinking like a programmer
  - Solving problems by developing programs and expressing your solution in C

- **COMP 1531 Course Goals**
  - gets you thinking like a software engineer
  - teaches you how to deliver value to customers
  - learning building large software systems is more than mere programming

COMP 1531 Major Themes

- **Software Engineering (SE)**
  - Understand the importance of SE
  - Realise SE is **not** Programming
  - Problem solving – apply SE principles to solve a real-world problem
    - Explore the key phases of SE life-cycle
    - Analyse a problem and elicit user requirements
    - Design using sound design principles
    - Effective coding and testing techniques

COMP 1531 Major Themes

❖ Team Collaboration

- Learn how to tackle large projects as a team than individual
- Team communication
- Teamwork essential for future employment
- Gain experience in using collaboration tools (GitHub)

❖ Software Development Life-Cycle

- Understand the software development process
- Understand the impact of choice of software development methodology to a software development project
- Understand and gain practical experience in Agile Software Development practices

COMP 1531 Major Themes

❖ **Software Architecture**

- Learn about software architectural styles
- How to choose a particular style for a given project

❖ **Data Modelling**

- Learn to how build a conceptual domain model based on OO design and ER modelling
- Understand the need for data persistence
- Learn how to map an ER model to a relational model

How to teach software engineering?

❖ Within the constraints of an academic environment, how can we best teach SE?

- Software Engineering is not rocket science, but it is an **art** that requires the use of common sense, discipline and diligence
- No single right way or no single text book to teach software engineering
- My approach: to teach and educate you on what I have learnt from my experience

Credit teaching material

- No text book, comprehensive lecture slides will be provided for COMP 1531 - some content and ideas are drawn from:
  - *Software Engineerings* , by Ivan Marsic, Rutgers, The State University of New Jersey (Available for free download from: http://www.ece.rutgers.edu/~marsic/books/SE/book-SE_marsic.pdf)
  - *Agile Software Development: Principles, Patterns and Practice* , by Robert C Martin, Pearson

- Links to useful tutorials will be uploaded as necessary

How do we obtain our educational objectives?

- **Lectures…** (Recorded through Echo 360)
  - 4 hour lectures
  - Tue: 16:00 – 18:00, Wed: 14:00 – 16:00 (weeks 1-10)
- **Tutorials and Labs…**
  - 3 hours per week (1 hour tut followed by lab)
  - (Some tut-groups will have tut/lab in week 11 due to public holiday in week 9 and week 10)
  - Tutorials re-enforce learning from lectures
  - Labs comprise small design and practical programming exercises, individual or in pairs
- **A Group Project…**
  - Project iteration demos scheduled in some lab sessions

Assessments

- Lab Sessions: 3 hours per week (12%)
  - Lab sessions encompass both design and practical programming exercises
  - To obtain any lab marks for a Week X lab, you must:
    - *upload your completed solution to GitHub*
    - *demonstrate your work to your tutor in the lab class in the week it is due OR in the first hour of the following week (completed solution must be submitted to GitHub by Sunday of Week X, unless otherwise specified). Late demonstrations will not be accepted (e.g., getting lab 2 marked off in lab 4)*
  - **Cannot** obtain marks by emailing solution to tutors

Assessments

- Online Quizzes (3%)
  - Multiple-choice format, similar in style to MCQ in exam
  - Review of content covered in lectures
  - Taken in your own time (via WebCMS3)
  - Due before Sunday 11:59 in weeks 4, 8, 10
- Mid-term exam (10%)
  - Specification will be released in week 2
- Final Exam (50%)
  - 3 hour final exam, more details through the term

Group Project

- Contributes to 25% of the final course mark
- Carried out in teams of 3
- Project specification will be released in Week 05
- Implemented using an *Agile Software Development Model*
  - Working software to be delivered in iterations
  - Project specification **can change** at the end of an iteration (as customer changes mind…), so **design well** !!
  - Marks will be awarded for each iteration demo, which will count towards your overall group project mark
  - Responsibilities to be assigned to each group member during each iteration and all team members **MUST** contribute equally **(Tutors will check GitHub)**
  - Final project demo held in week 10

Course Mark

- Course Work Mark = Lab + Quiz + Group Project (out of 40)
- Mid-Term-Exam-Mark = Mark from mid-term exam(out of 10)
- Final-Exam-Mark = Mark from the 3 hour final exam (out of 50)
- **Exam_OK = (Mid-Term-Exam-Mark + Final-Exam-Mark) >= 30/60**
- **Final Course Mark** (out of 100) = Course Work Mark + Mid-Term-Exam-Mark + Final-Exam-Mark
- Final Grade
  - UF, if !ExamOK (even if final course mark > 50)
  - FL, if Final Course Mark < 50/100
  - PS, if 50/100 ≤ Final Course Mark < 65/100
  - CR, if 65/100 ≤ Final Course Mark < 75/100
  - DN, if 75/100 ≤ Final Course Mark < 85/100
  - HD, if Final Course Mark ≥ 85/100

Supplementary Exam

- Please consult the **UNSW Computing Supplementary Assessment Policy** at https://www.engineering.unsw.edu.au/computer-science-engineering/about-us/organisational-structure/student-services/policies/essential-advice-for-cse-students

- If you are granted a supplementary exam or if you think you are eligible for a supplementary exam, you must make sure that you are available on the scheduled day.

- There will be **only one** supplementary Exam, which is held in the period, scheduled by the exam unit.

System

- Most work done on Linux or Mac
  - Lab work and group project can be done on the CSE machine labs or your own device
  - Technology stack – Python, Flask
  - Must use Python 3.5 onwards (Use virtual environment, more instructions will be provided in the forth-coming lecture and tutorial sessions)
  - Use your own favourite text editor (e.g., vim)
- Collaboration and Versioning Tool - GitHub

Plagiarism

Just don’t do it!

COMP 1531
Software Engineering Fundamentals

Introduction to Software Engineering

Why become a Software Engineer?

Meet software engineers at Google:

- https://www.youtube.com/watch?v=9S_fnC5jPgw

6 reasons to become a software engineer:

https://www.youtube.com/watch?v=lfTwqw_gyKs

- Software is critical to society

  … Software is permeating our society, used to control functions of various machines (e.g, aircrafts, pacemakers)

- Building software is exciting, challenging and fun

  … building software for the next generation that change how billions of people interact, connect and explore

  … learn about team culture, meet new people

- $$$

So far in CSE…

- You have mostly implemented carefully defined specifications
- COMP 1917: Implement a simplified Pokemon
- For this assignment you were given:

… a well-defined specification

… a list of commands for the pokemon

(e.g., a – add a pokemon to the list, > - move to the next pokemon etc)

Requirements

… a header file (typeDefs, function prototypes)

… the abstract data type to use (e.g., list)

Design

… hints about the implementation (e.g., how to traverse the list)

Objective so far…

- Build an implementation that matched the specification in functionality, work complexity (O(N) or O log N)
- And you were given…
- …the specification (what to do or requirements)
- …the design (the Abstract Data Type (ADT) interface)
- …hints about implementation
- …perhaps a pointer to some libraries to use

- Someone had to figure all of the above; that’s software engineering and now it’s your turn

What is software engineering?

“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.” [IEEE]

What is Software Engineering?

- "Software Engineering" is a discipline that enables customers to achieve **business goals** through *developing* software-based systems to solve their **business problems** e.g., develop a course enrolment application or a software to manage inventory

- This discipline places great emphasis on the ***methodology*** or the *method for managing the development process*

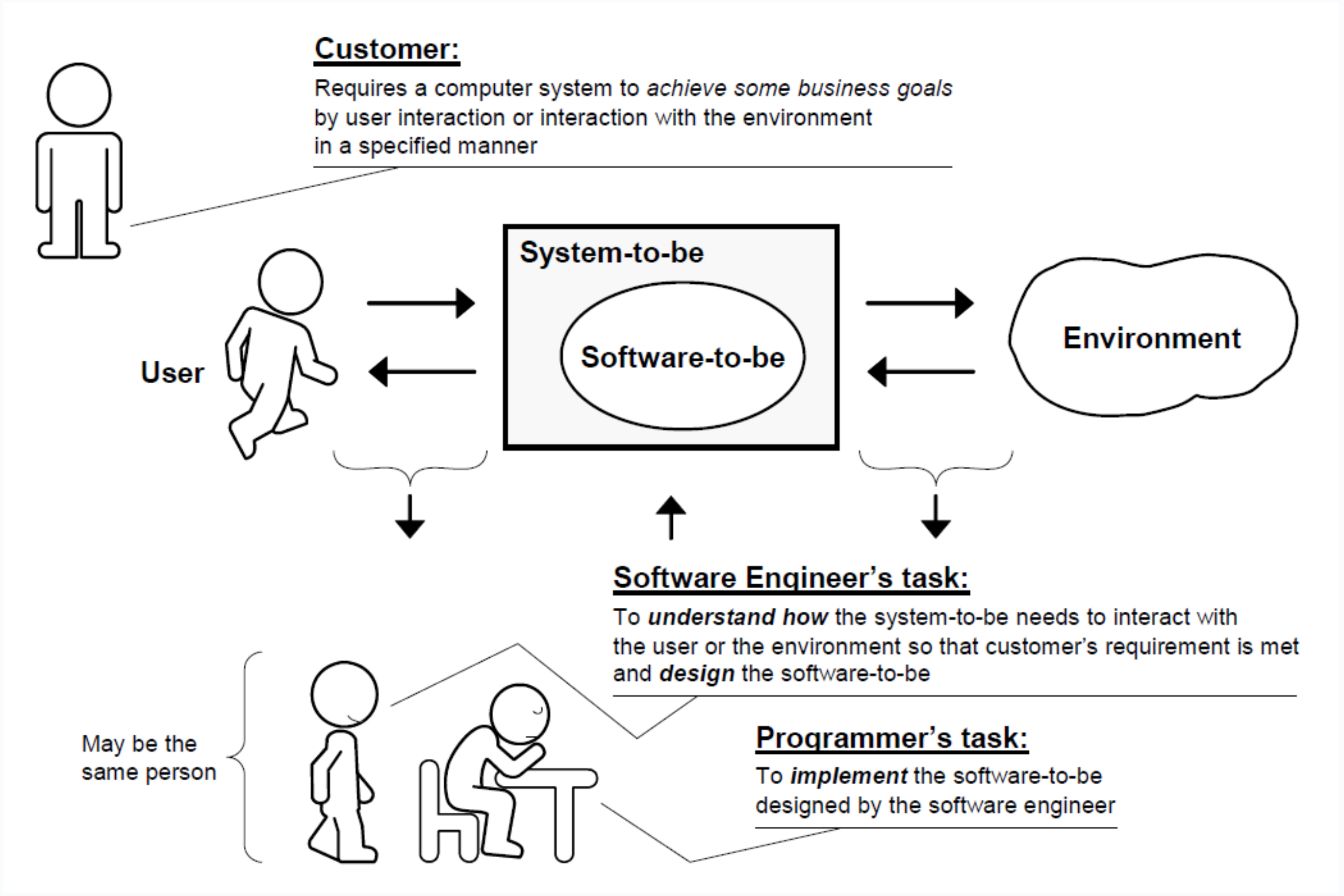
- The methodology is commonly referred to as **Software Development Life-Cycle (SDLC)**

Software Engineering is not Programming

- Software engineering:
  - **Understanding** the business problem (understanding the interaction between the system-to-be, its users and environment)
  - **Creative formulation** of ideas to solve the problem based on this understanding
  - **Designing** the "blueprint" or architecture of the solution
- Programming:
  - **Implementing** the "blueprint" designed by the software engineer

Role of a software engineer

- Software engineer thus acts as a bridge from customer **needs** (problem domain) to programming **implementation** (solution domain)
- This enables the software engineer to design solutions that accurately target the customer's needs, that is, deliver **value** to the customer


Customer:
Requires a computer system to *achieve some business goals* by user interaction or interaction with the environment in a specified manner
System-to-be
Software-to-be
User
Environment
Software Engineer's task:
To ***understand how*** the system-to-be needs to interact with the user or the environment so that customer's requirement is met and ***design*** the software-to-be
May be the same person
Programmer's task:
To ***implement*** the software-to-be designed by the software engineer

Why do we need SE (1)?

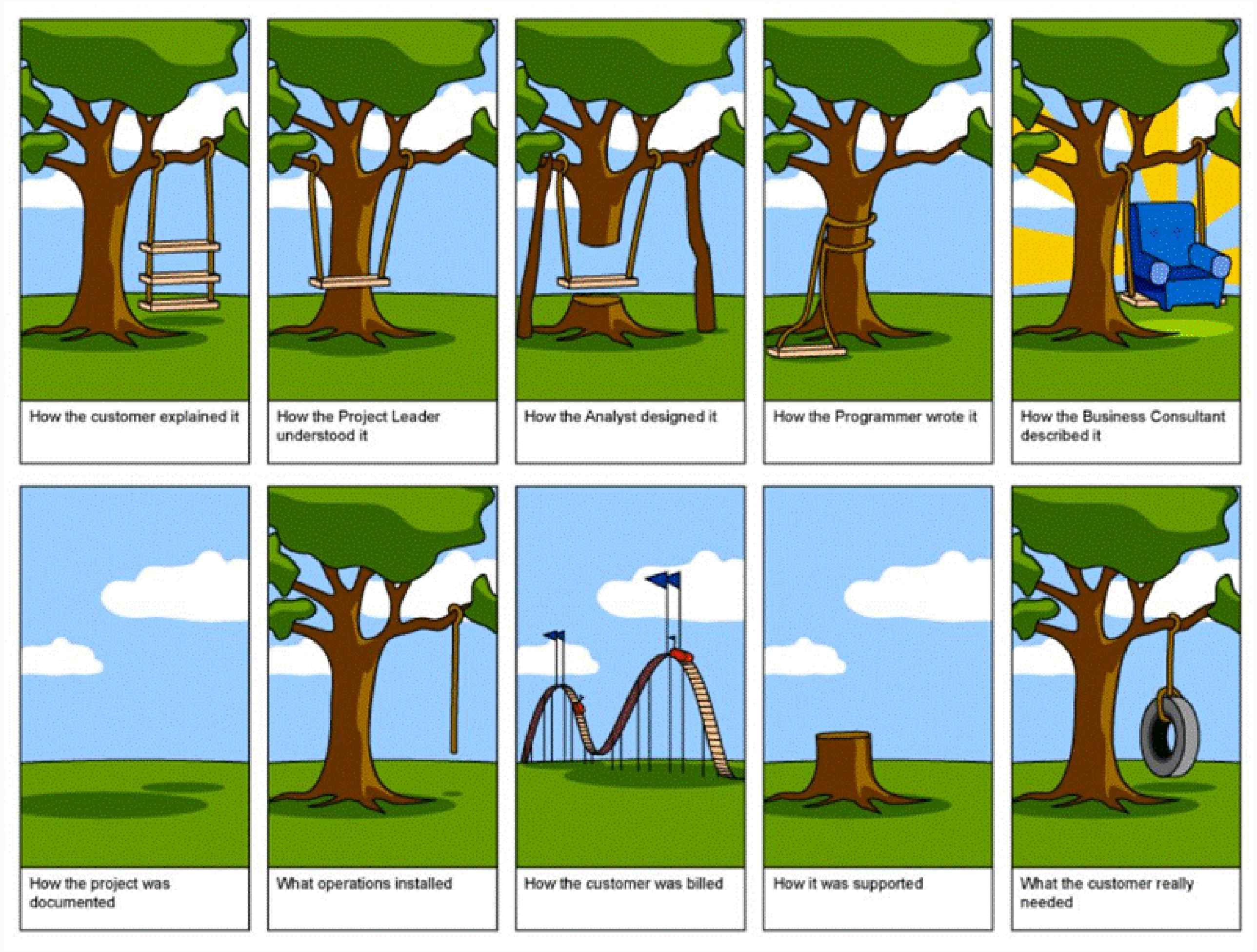
We want to…

- Make sure what we build is what the customer actually wanted
- Deliver the software on time and on budget
- Minimize defects
- Ensure reliability, security, performance, extensibility, usability, maintainability…

To do so, we need a systematic and disciplined approach to software development - and that's what software engineering is about

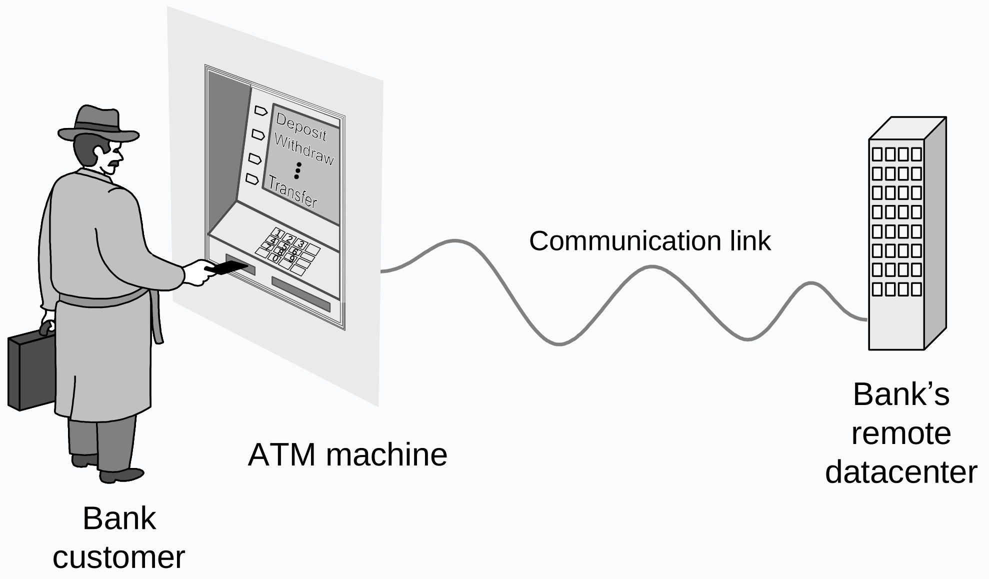
Why do we need SE (2)?

- Software development is a complex process, so building software requires a discipline, to ensure that the product delivered realises customer goals


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

Why do we need SE (3)?

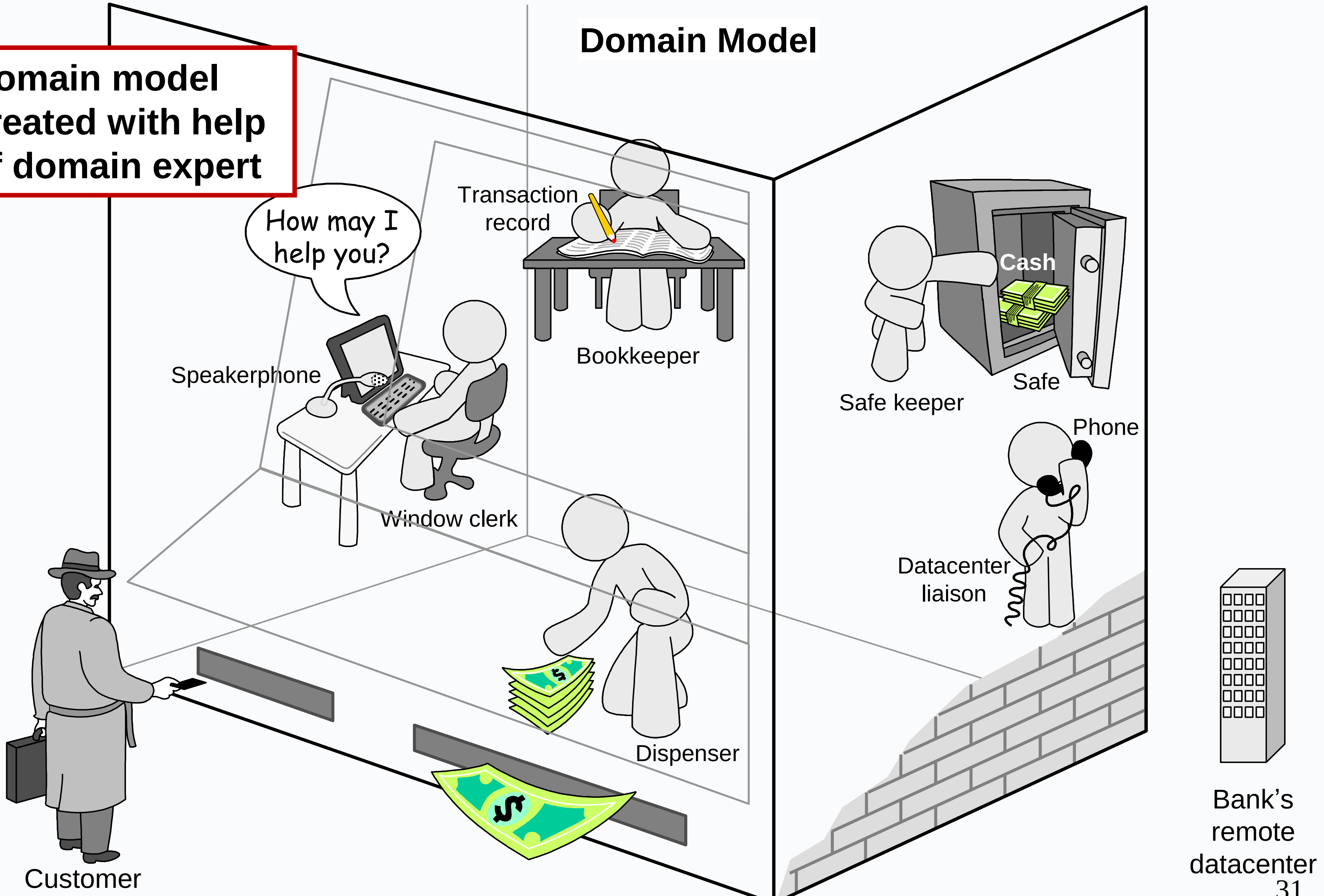
- Software is intangible and software development requires imagination e.g., the ATM Machine



  - Understand both the problem domain and software domain
  - Challenge: find a set of *good abstractions* that is representative of the problem domain and build a conceptual domain model
  - Creative formulation of solutions, design alternatives, trade-offs, difficult to come up with the "correct solution"

How ATM Machine Might Work

Domain model created with help of domain expert



Why do we need SE (4)?

- Software errors, poor design and inadequate testing have led to loss of time, money, human life
  - **Case of the Therac-25:** one of the most well-known killer software bugs in history
  - **Explosion of Ariane-501 in 1996:** blew up 37 seconds after initial launch, cost $7 billion and 10 years of development
  - Or just imagine… if Facebook accidentally leaked out your private information or Amazon started shipping products to wrong customer

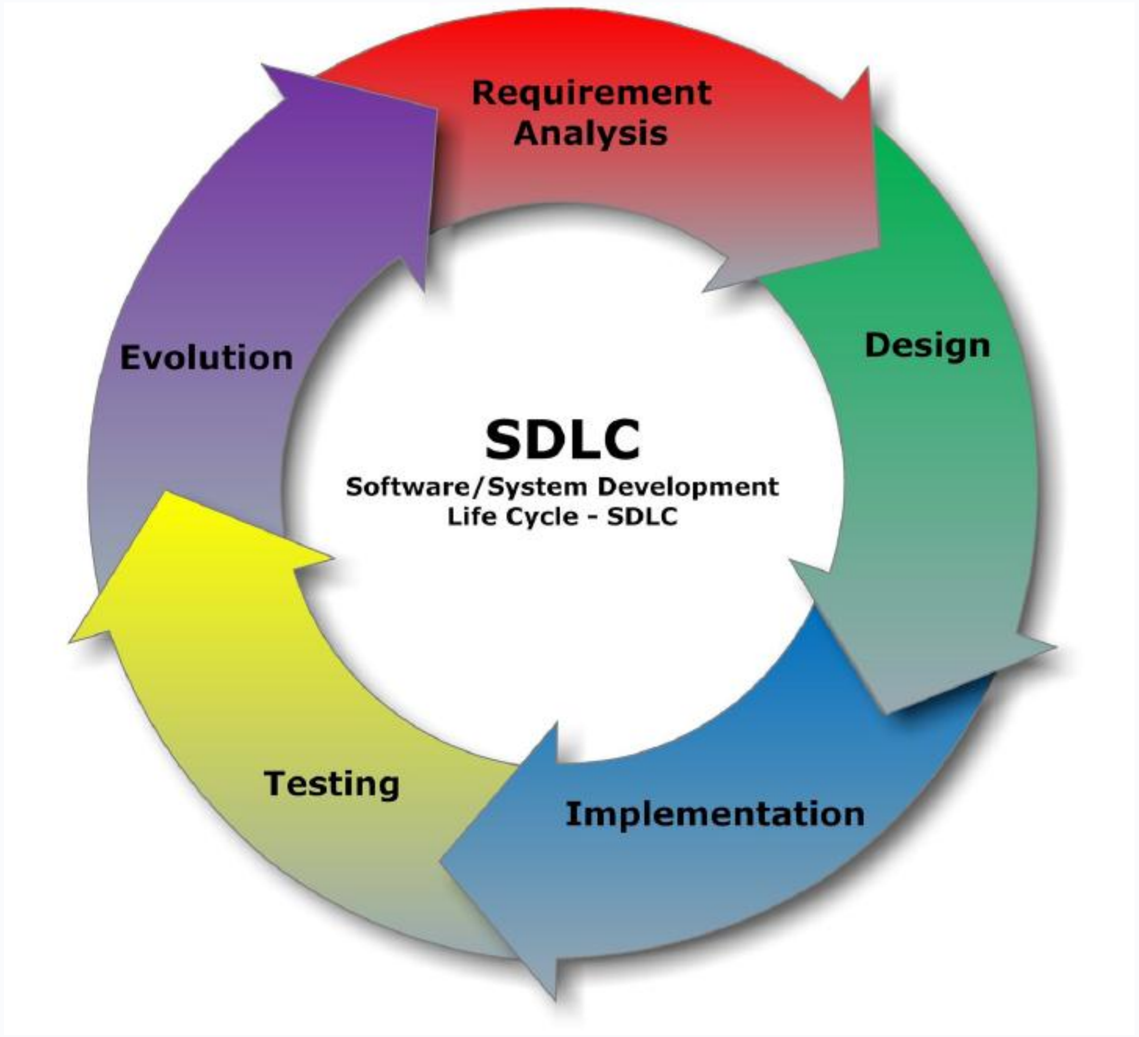
Checkpoint !

Quiz - Software Engineering Basics

Software Development Life-Cycle

- We described software engineering as a complex, organised process with a great emphasis on *methodology*
- This *methodology* is essentially a framework to structure, plan and control the development of the software system and typically consists of the following phases:
  - Analysis and Specification
  - Design
  - Implementation
  - Testing
  - Release & Maintenance
- Each of the above phases can be accompanied by an artifact or deliverable to be achieved at the completion of this phas

Software Development Life-Cycle



SDLC – Requirements Analysis

1. Analysis:

Discover and learn about the **problem domain** and the **"system-to-be"** where software engineers need to:

- Analyse the problem, understand the problem definition – what *features/services* does the system need to provide? ***(behavioural characteristics or external behaviour)***

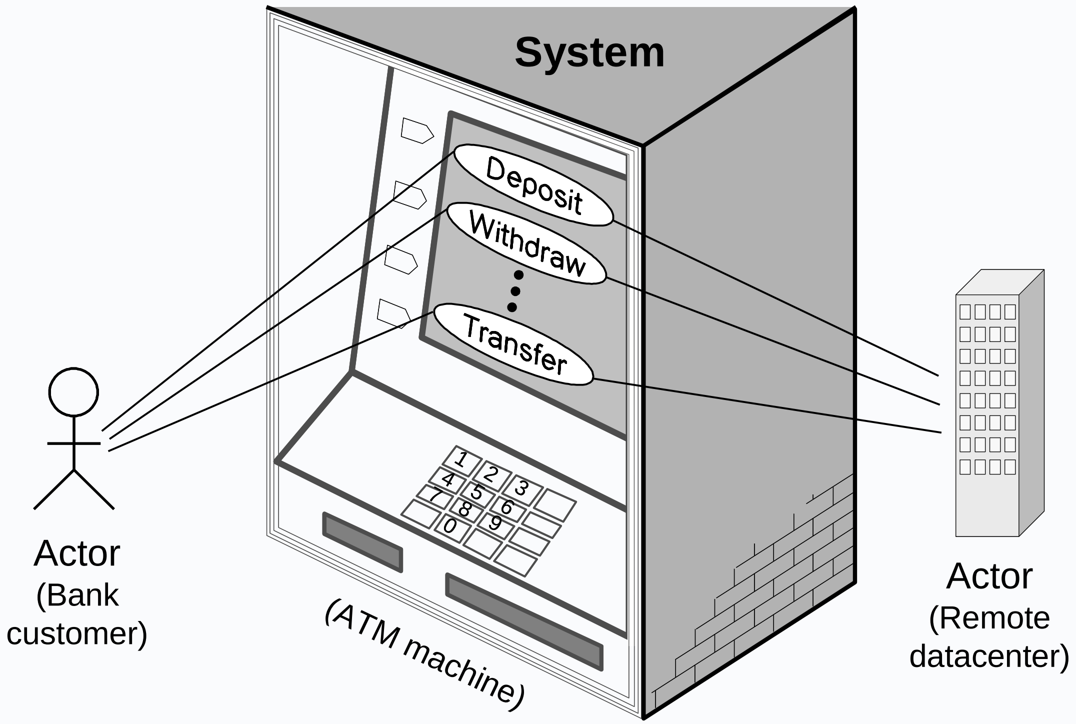
- Determine both functional (inputs and outputs) and non-functional requirements (performance, security, quality, maintainability, extensibility etc.)
- Use-case modelling, user-stories are popular techniques for analysing and documenting the customer requirements

SDLC – Design Phase

2. Design:

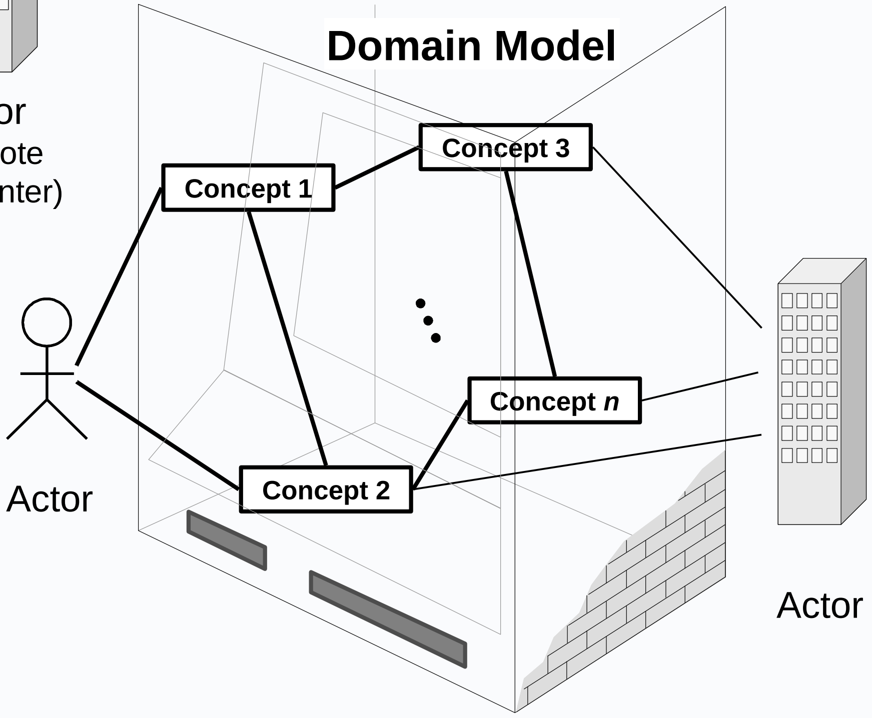
- A problem-solving activity that involves a "creative process of searching how to implement all of the customer's requirements"
- Plan out the system's "internal structure or structural characteristics" that delivers the system's "external behaviour" specified in the previous phase
- Produce software blue-prints (design artifacts e.g., domain model)
- Often the design phase overlaps with previous phase

Requirements Analysis vs Design



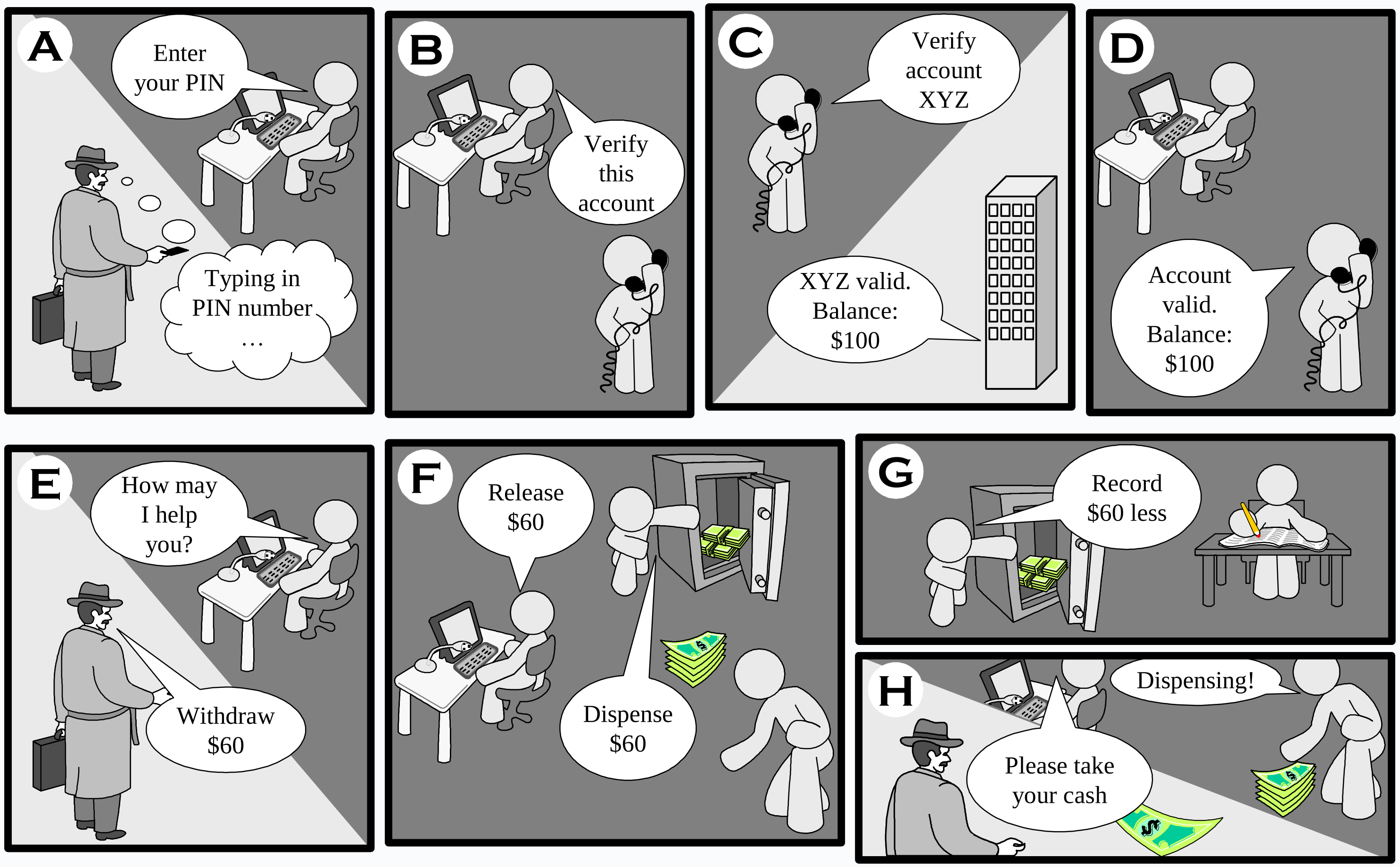
(a)

(b)



SDLC – Design Phase

In generating these “blue-prints”….
Cartoon Strip Writing OR more formal symbols (e.g., UML symbols – use-case diagram, class diagram, component diagram…?



Cartoon Strip: How ATM Machine Works

SDLC Phases

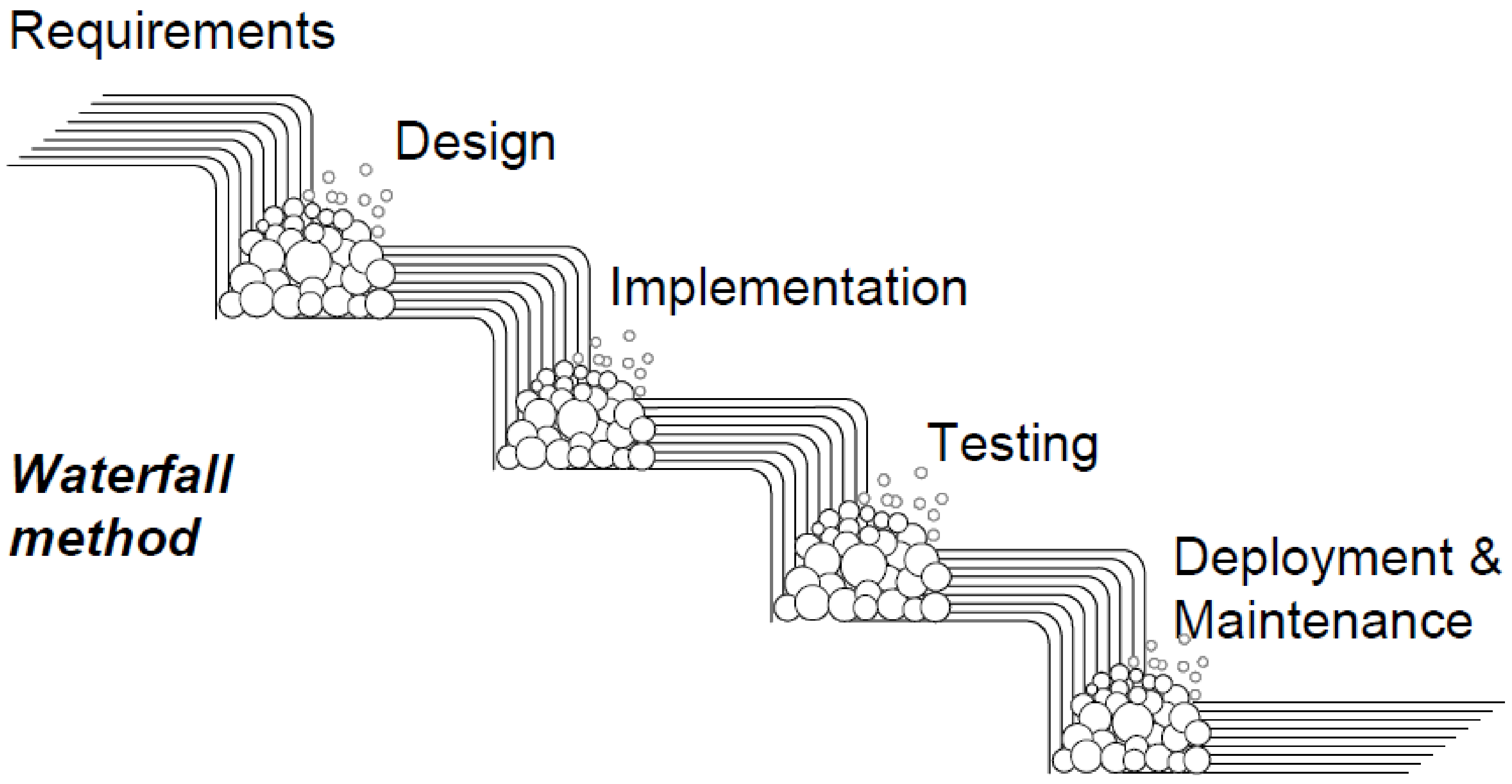
3. Implementation:
   - Encode the design in a programming language to deliver a software product
4. Testing:
   - Verify that the system works correctly and realises the goals
   - Testing process encompasses
     - unit tests (individual components are tested)
     - integration tests (the whole system is testing)
     - user acceptance tests (the system achieves the customer requirements)
5. Release & Maintenance:
   - Deploying the system, fix defects and adding new functionality

Software Development Methodologies

A software development methodology prescribes, how the different phases in SDLC are carried out and can be grouped into two broad categories:

- Waterfall Process:
  - A linear process, where the various SDLC phases are carried out in a sequential manner
- Iterative & Incremental processes
  - which develop increments of functionality and repeat in a feedback loop
  - e.g., Rational Unified Process [Jacobson et al., 1999], Agile methods (e.g., SCRUM, XP)-methods that are more aggressive in terms of short iterations

Waterfall Model (1970's)



Waterfall process for software development.

Waterfall Model (1970's)

- Linear, sequential project life-cycle (*plan-driven development model*) characterized by detailed planning:
  - Problem is identified, documented and designed
  - Implementation tasks identified, scoped and scheduled
  - Development cycle followed by testing cycle
- Each phase must be fully completed, documented and signed off before moving on to the next phase
- Simple to understand and manage due to *project visibility*
- Suitable for risk-free projects with **stable** product statement, clear, well-known requirements with no ambiguities, technical requirements clear and resources ample or mission-critical applications (e.g., NASA)

Waterfall Model Drawbacks

- No working software produced until late into the software life-cycle
- Rigid and not very flexible
  - No support for fine-tuning of requirements through the cycle
  - Good ideas need to identified upfront; a great idea in the release cycle is a threat!
  - All requirements frozen at the end of the requirements phase, once the application is implemented and in the "testing" phase, it is difficult to retract and change something that was not "well-thought out" in the concept phase or design phase
- Heavy documentation (typically model based artifacts, UML)
- Incurs a large management overhead
- Not suitable for projects where requirements are at a moderate risk of changing

Software Development Challenges

- Software is:
  - probably, the most complex artifact
  - intangible and hard to visualise
  - the most flexible artifact – radically modified at any stage of software development when customer changes requirements
- Waterfall model prescribes a sequential process, but this linear order does not **always** produce best results
- Easier to understand a complex problem by implementing and **evaluating pilot solutions**.

Incremental and Iterative Project Life-Cycle

- Incremental and iterative approaches:
  1. Break the big problem down into smaller pieces and prioritize them.
  2. An “iteration” refers to a step in the life-cycle
  3. Each “iteration” results in an “increment” or progress through the overall project
  4. Seek the customer feedback and change course based on improved understanding at the end of each iteration
- An incremental and iterative process
  - seeks to get to a working instance as soon as possible.
  - progressively deepen the understanding or “visualization” of the target product

Incremental and Iterative Models

- Unified Software Development Process (Ivar Jacobson, Grady Booch and James Rumbaugh)
  - an iterative software development process where a system is progressively built and refined through multiple iterations, using feedback and adaptation
  - each iteration will include requirements, analysis, design, and implementation
  - Iterations are **timeboxed**
- Rational Unified Process (A specific adaptation of Unified Process), evolved at IBM
- Agile Methodologies (e.g., XP, SCRUM), that are more aggressive in terms of short iterations

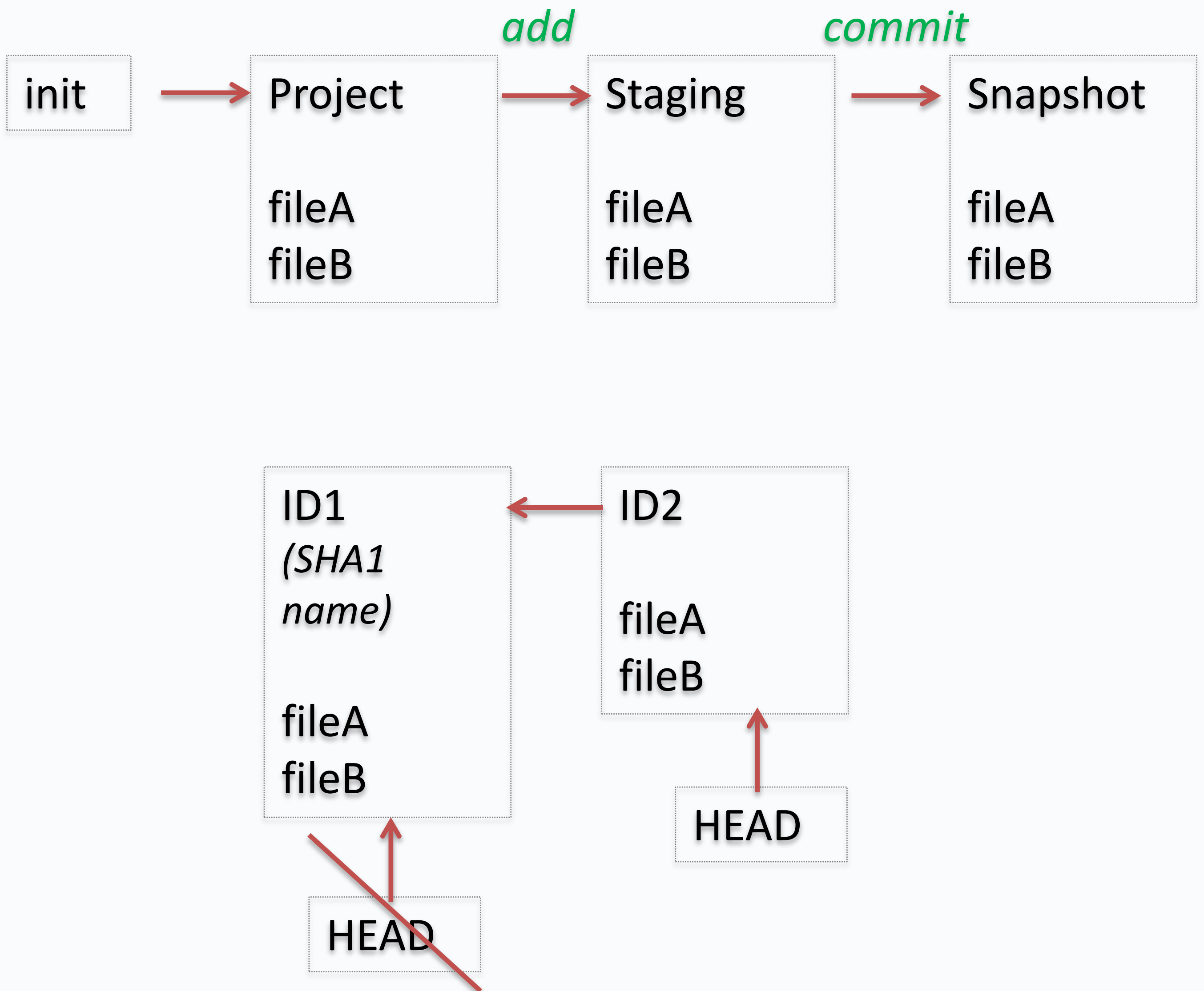
Incremental models explored in detail in week 4

Checkpoint !

Quiz - Software Development Methodologies

Lecture Demo: Git & GitHub

Git Basics – Creating Snapshots



Checkpoint !

Check your Git understanding by watching the series of Git Videos and answering the following questions

1. Quiz - Git Basics
2. Quiz - Branching
3. Quiz - Merging Branches
4. Quiz - Remote Repositories

Tasks to be done prior to your lab in week 1

1. Create a GitHub account using either your UNSW student email address or personal email address.
2. Learn Git
   - Make sure you watch the videos at: https://drive.google.com/drive/folders/1IflPeCjdNpS_eWZnppn_d7wLK83OhUo3
   - Best way to become familiar with Git & Github, is to practice the exercises shown in the video
3. After you have created your GitHub account, please go to this link: https://cgi.cse.unsw.edu.au/~cs1531/github/run.cgi/login and complete the setups outlined in “PreLab” exercise in Lab 01
4. If you encounter any problem while doing this, post it in the course forum (include your GitHub username and user email)

More tasks for week 1 …

- Make yourself familiar with Python

  - Watch the videos at: https://drive.google.com/drive/folders/1uJu0Xp_4APu3WXCFMBw-4yCZ1QlI3F8f

  - Topics covered in week 1:
    - Using python interpreter
    - Defining variables, numbers and strings
    - Control flow (if, for, range, while, break and continue)

- Group Project
  - Start planning your team of **no more than 3**!

Other useful Github and Python Resources

- Python tutorial: https://docs.python.org/3/tutorial/
- Get started with GitHub: https://guides.github.com/activities/hello-world/
- An interactive Git tutorial: https://learngitbranching.js.org/

(Strongly recommended for people who have never used Git before)

Next week…

- Requirements Engineering (User-stories and Use-Cases
- Using UML in Requirements Analysis
- Python (Data-structures – Lists, Tuples, Dictionaries, Defining functions)