Introduction and Process

Lecture 1

Ruzanna Chitchyan, Jon Bird, Pete Bennett
TAs: Alex Elwood, Alex Cockrean, Casper Wang

Purpose of Unit

Previous units aim to improve your knowledge as computer scientists

This unit helps package this knowledge as employability skill-sets

This is achieved by targeting the following industry-relevant issues:

- Transferable skills: Management, Collaboration, Communication
- Platform skills: applying your prior programming language knowledge
- Scalability skills: Development “in the large” at industrial scale

Software Development

But have you heard of ***Software Crisis***

- Software is getting larger and larger
- Software is getting more complex
  - Complex domains (e.g., human behaviours)
  - Systems dependency (e.g., energy and cars)
- Time to market is shorter than ever

And Projects Fail

Ariane 5 Flight 501

Mars Climate Orbiter

Therac-25 Radiotherapy

Why Do Software Projects Fail?

- Bad developers – not the main problem!
- Over budget
  - Poor requirements
  - Over-ambitious requirements
  - Unnecessary requirements
- Contract management
- End-user training
- Operational management

Software Engineering

- Software Engineering
  - Engineering**: cost-effective solutions** to practical problems by applying **scientific knowledge** to building **things** for **people**
    - Cost-effective solutions: process and project management, contracts…
    - Scientific knowledge: modelling, proofs, testing, simulation, patterns...
    - Things=software
    - People: customers and end users

OK, so how do we do *that* ?

Collaboration Tools and Techniques

- UML Designs: Diagramming to reach a consensus on design
- GitHub: Sharing of documents for effective collaboration
- Kanban: Task allocation and progress monitoring
- Test-driven development: Stop other people breaking your code !
- Other techniques are encouraged: Feel free to experiment and explore

Weekly Themes

1. Introduction + Overview
2. Agile Development
3. Requirements Engineering
4. Design
5. Software Quality and Testing
6. [Reading Week]
7. Project Management
8. Human Computer Interaction: Qualitative Analysis
9. Human Computer Interaction: Quantitative Analysis
10. [Easter break: 3 weeks]
11. Advanced Topics in Software Engineering
12. Module Closure

Groupwork

- We can only hope to tackle a real-world scale application...

  ....if we work in teams towards the end goal

- Working together is also a key transferable skill
- In any non-trivial industrial project, you'll be working in a team (maybe summer project!)
- It's not easy – which is why it is essential to practice now!

- Teams of 5 (or 6), assigned randomly

“Tasting Stick” Metaphor

- All of the previously named techniques are large and complex
- We don’t expect you to become experts in everything
- The aim is to give you a “taster” of everything
- So you can at least be able to say that you have tried them
- One way to think of this unit is as a “tasting stick”
- Working in groups, individuals are likely to specialise in some aspects

Software Development Life Cycle Processes:

Which tasks are to be done and in what order?

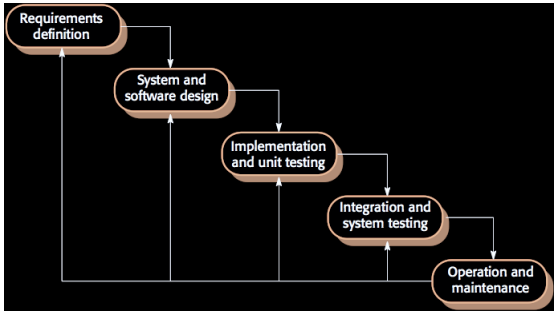
Software Development Tasks

- Requirements Analysis
- Planning
- Design: high level and detailed
- Development
- Testing
- Deployment
- Operation and Maintenance

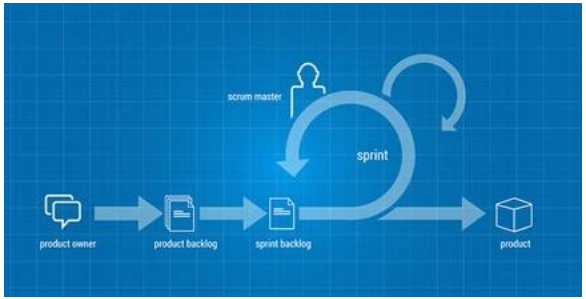
These to-does are combined in various sequences, making up different Software Development Life Cycle Processes

Software Development Life Cycle Examples

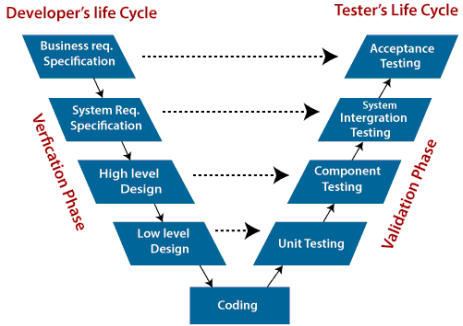
Waterfall



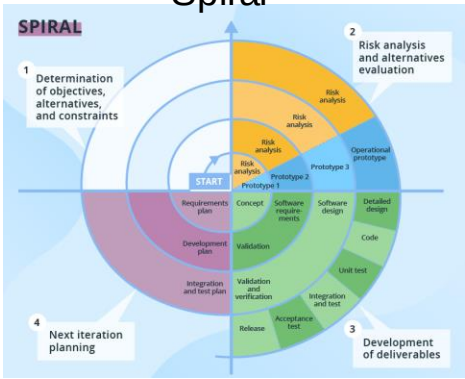
Agile



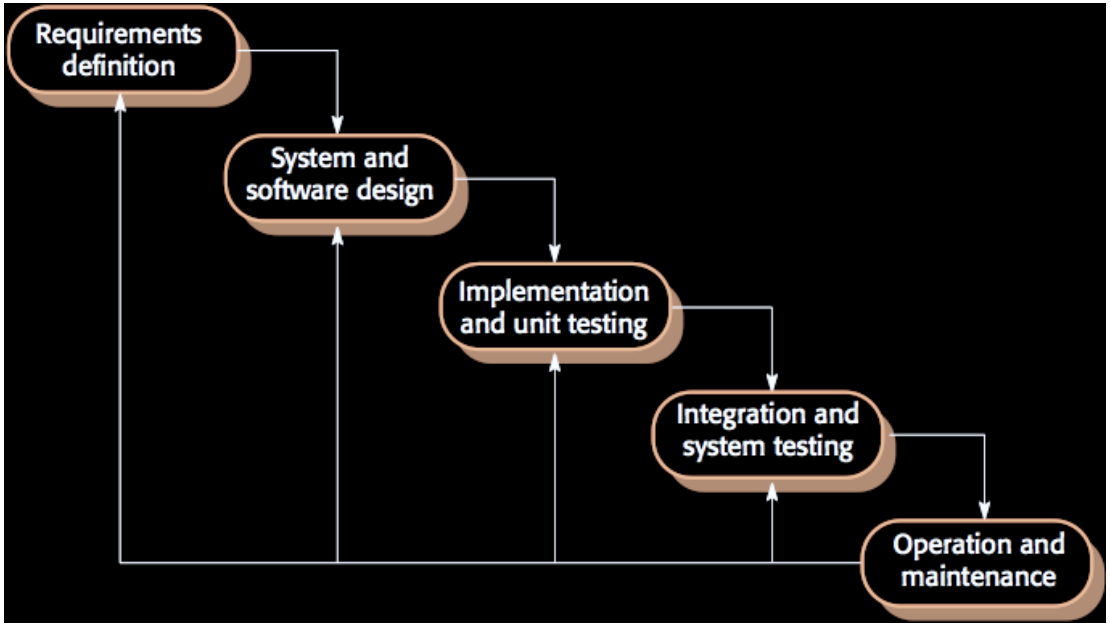
V-Model



Spiral



Waterfall Software Development Life Cycle



Requirements: What to Implement?

- What should the system do ?
- What are the needs of the users ?
- What are the needs of the host organisation ?
- What are interoperability needs of existing systems

We need to think about functional elements: what the system should do?
And also non-functional elements: quality properties of system operation, e.g., security, ease of use, response time etc.

Design: How to Structure Software?

- What objects, databases, servers, services etc. should we create?
- How are these elements structured into a system?

Why to design?

- Helps to decide where to place requirements
- How the parts of the system will be interacting
- Divide up work between team members

Implementation

Not just programming, but also…

- Parallel working and code sharing
- API partitioning and “firewalling”
- Versioning, integration and config management
- Development environments and automated build
- Automated testing
- Docs and training material

Verification and Validation

- Verification: check if the software/service complies with a requirements, constraints, and regulations. ("Are you building it right?")
  - Demonstrates that system meets specifications

- Validation: does this meet the needs of the customers/ stakeholders? ("Are you building the right thing?")
  - Demonstrate that system meets user needs

- Can pass verification but fails validation: if built as per the specifications yet the specifications do not address the user's needs.

- Involves checking, reviewing, evaluating & testing

Waterfall SDLC: Advantages and Disadvantages

- Simple to use and understand
- Every phase has a defined result and process review
- Development stages go one by one
- Perfect for projects where requirements are clear and agreed upon
- Easy to determine the key points in the development cycle
- Easy to classify and prioritize tasks

- The software is ready only after the last stage is over
- High risks and uncertainty
- Misses complexity due to interdependence of decisions
- Not suited for long-term projects where requirements will change
- The progress of the stage is hard to measure while it is still in the development
- Integration is done at the very end, which does not give the option of identifying the problem in advance

Case Study: Insulin Pump Control System

An insulin pump is a medical system that simulates the operation of the pancreas (an internal organ). The software controlling this system is an embedded system, which collects information from a sensor and controls a pump that delivers a controlled dose of insulin to a user.

People who suffer from diabetes use the system. Diabetes is a relatively common condition where the human pancreas is unable to produce sufficient quantities of a hormone called insulin. Insulin metabolises glucose (sugar) in the blood. The conventional treatment of diabetes involves regular injections of genetically engineered insulin. Diabetics measure their blood sugar levels using an external meter and then calculate the dose of insulin that they should inject.

The problem with this treatment is that the level of insulin required does not just depend on the blood glucose level but also on the time of the last insulin injection. This can lead to very low levels of blood glucose (if there is too much insulin) or very high levels of blood sugar (if there is too little insulin). Low blood glucose is, in the short term, a more serious condition as it can result in temporary brain malfunctioning and, ultimately, unconsciousness and death. In the long term, however, continual high levels of blood glucose can lead to eye damage, kidney damage, and heart problems.

Current advances in developing miniaturized sensors have meant that it is now possible to develop automated insulin delivery systems. These systems monitor blood sugar levels and deliver an appropriate dose of insulin when required. Insulin delivery systems like this already exist for the treatment of hospital patients. Many diabetics want to have such systems permanently attached to their bodies.

To Do: Working in Pairs

- Discuss 2 reasons on why Waterfall process could work well for this case study and 2 reasons why it wouldn't work (10 min)

- Class discussion (5 min)

Review

- What is Software Engineering about?
- What is Software Development Life Cycle?
- Can you name any Software Development Life Cycles?
- What are the specific characteristics of Waterfall SDLC?

Intro to Processing + Agile Techniques

Workshop 2

Ruzanna Chitchyan, Jon Bird, Pete Bennett
TAs: Alex Elwood, Alex Cockrean, Casper Wang

Today's Workshop

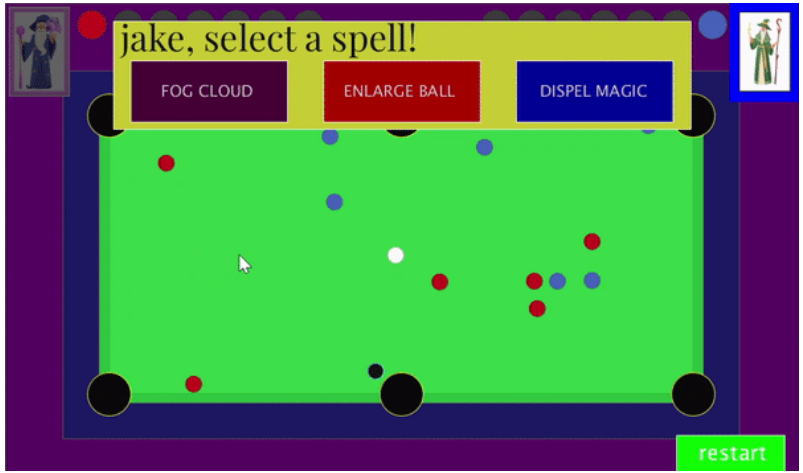
- Project examples from last year (10mins)
- Introduction to Processing (20mins)
- Develop an app (~ 60 mins)
    - Practice Pair Programming
    - Practice Kanban board use
- Try some (very light) evaluation (15mins)

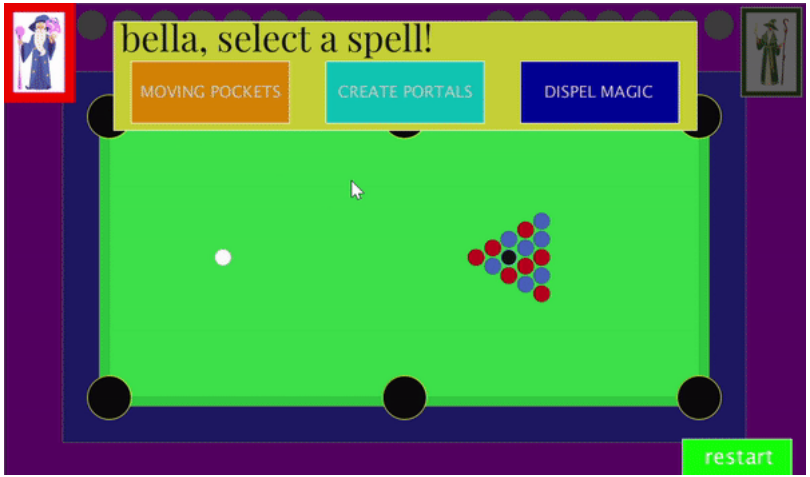
Coursework

- Any questions about the coursework brief?
- Have you made contact with everyone in your team? Let us know now if not!
- Have you made a commit on your repo?
    - Team Photo
    - List of games (inspiration and ideas)

Game Example: Wizard Pool

- "pool with a twist of magic"
- Challenges: Ball collision, spell system, tutorial



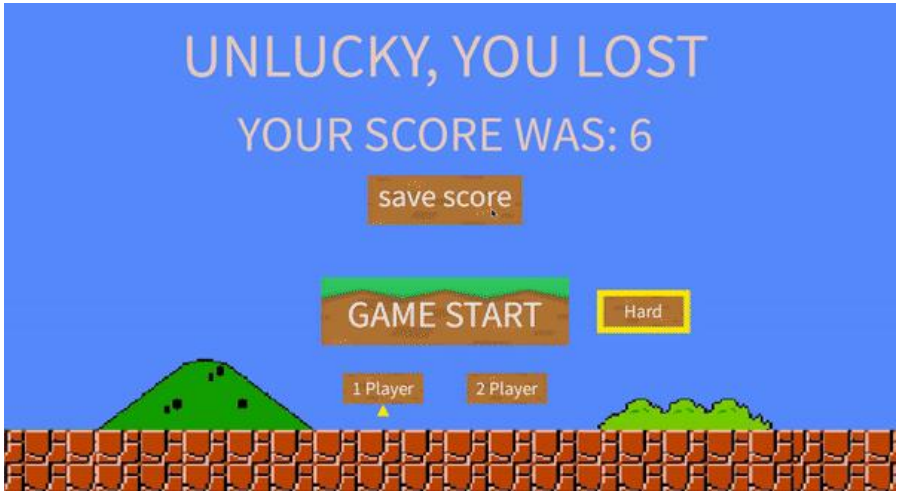


Game Example: JUPITER X Resource War

- Based on the rogue-like game spelunky, but… based in space and has an unkillable ghost enemy [github]
- Challenges: generative map, collision detection, performance optimisation

Game Example : Topsy Turvy

- Classic platformer but… with gravity reversal
- Challenges: physics engine, multi-player mode, high-scores log



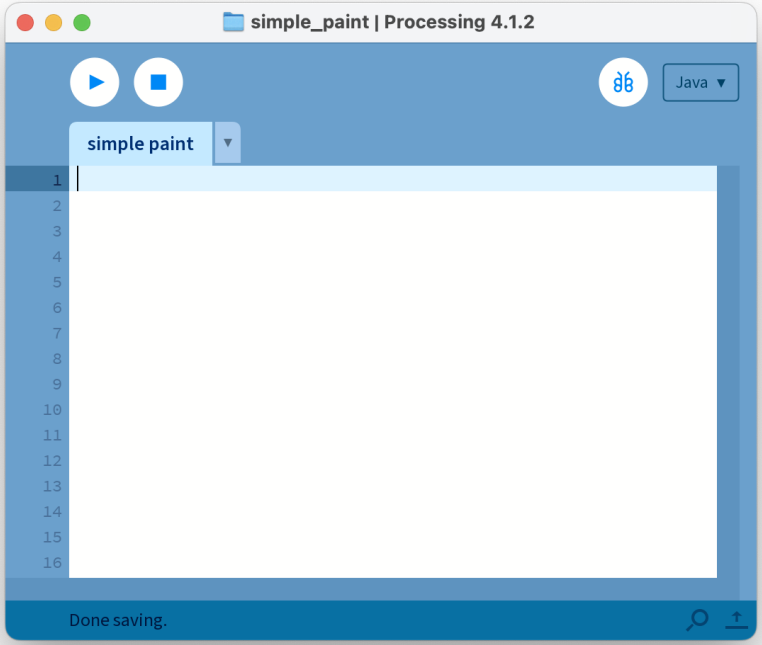
Simple Paint

- Today you are going to develop in pairs a **simple drawing tool** in Processing.
- Inspired by the classic Microsoft Paint, but… this is only a starting point, you will be adding any features you like.
- You will be swapping your paint app with another pair at the end to draw a **portrait**
- *…keep this use in mind before going off and making an abstract generative geometric paint program!*

image

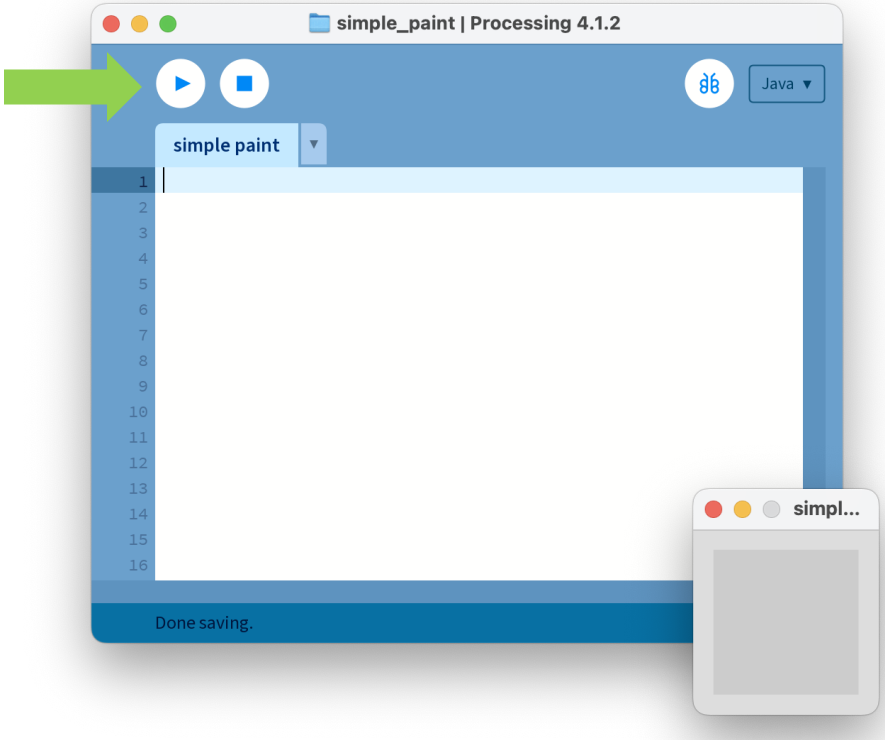
Follow Along Example

- Follow along with this quick demo.
- This will start you all off with a very bare bones Minimum Viable Product (MVP).
- Start by opening up a new Processing ‘sketch’



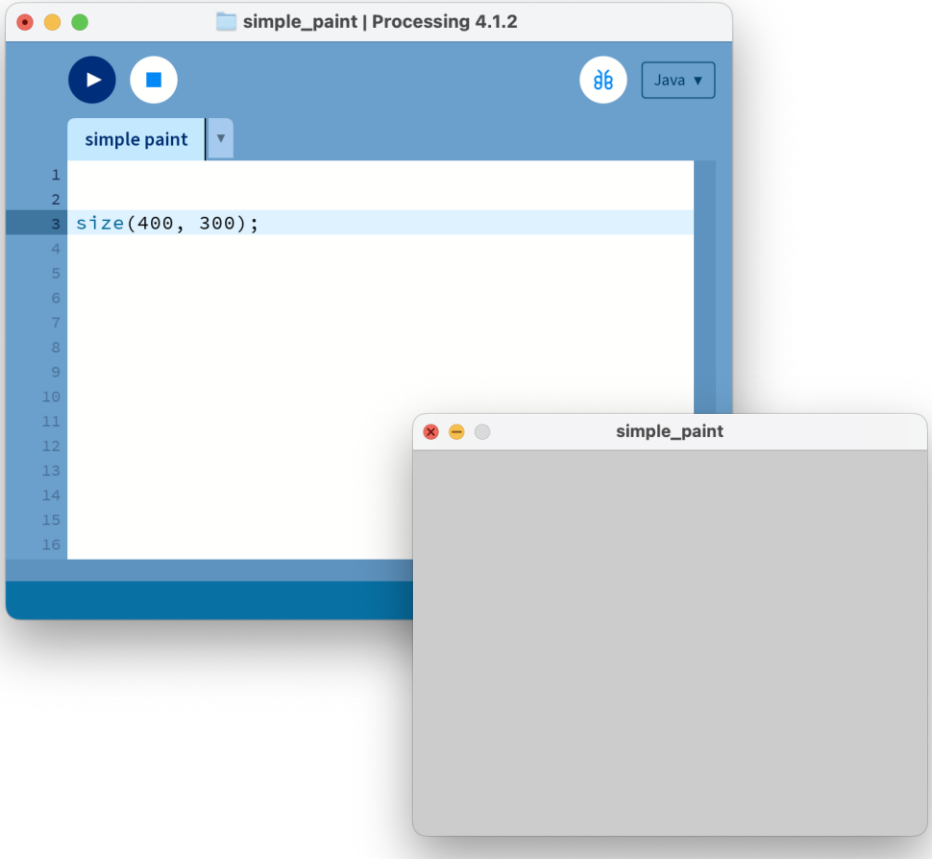
Run + Stop

- Use the start and stop buttons to start and stop your sketch from running.
- If you run and empty sketch, it will still work!



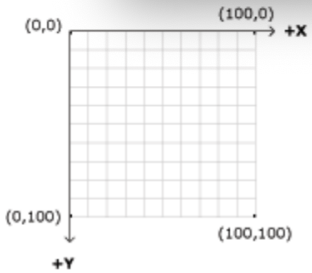
Canvas Size

- First task is to make the canvas a bit larger with the size function:
  - size(*width*, *height*);
  - size(400, 300);
- the size is set in pixels
- Don't forget the semi-colon



Draw a Circle

- Now draw a circle using the ellipse function that takes four arguments:
  - **ellipse**(*x, y, width, height*);
  - **ellipse**(200, 150, 20, 20);
- To draw a rectangle use:
  - **rect**(*x, y, width, height);*
- Coordinate system: **top left is 0, 0**

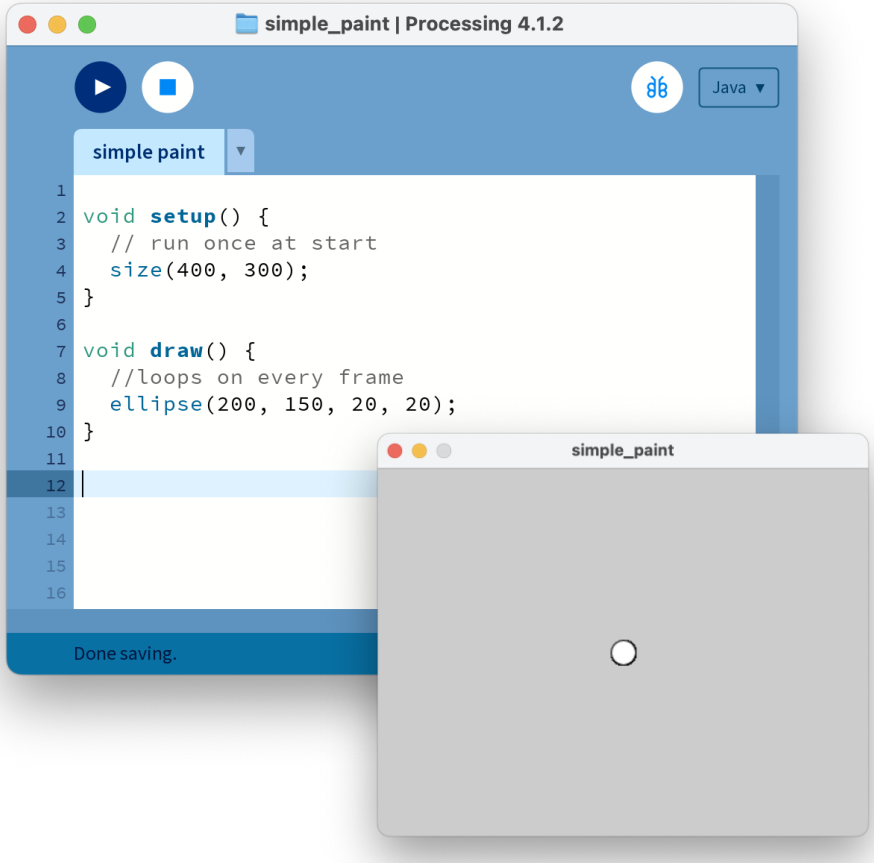


setup & draw

- The last sketch ran through only once then stops. This can work for some non-interactive applications, but we want to be able to animate! So we need to use **setup** and **draw** functions:

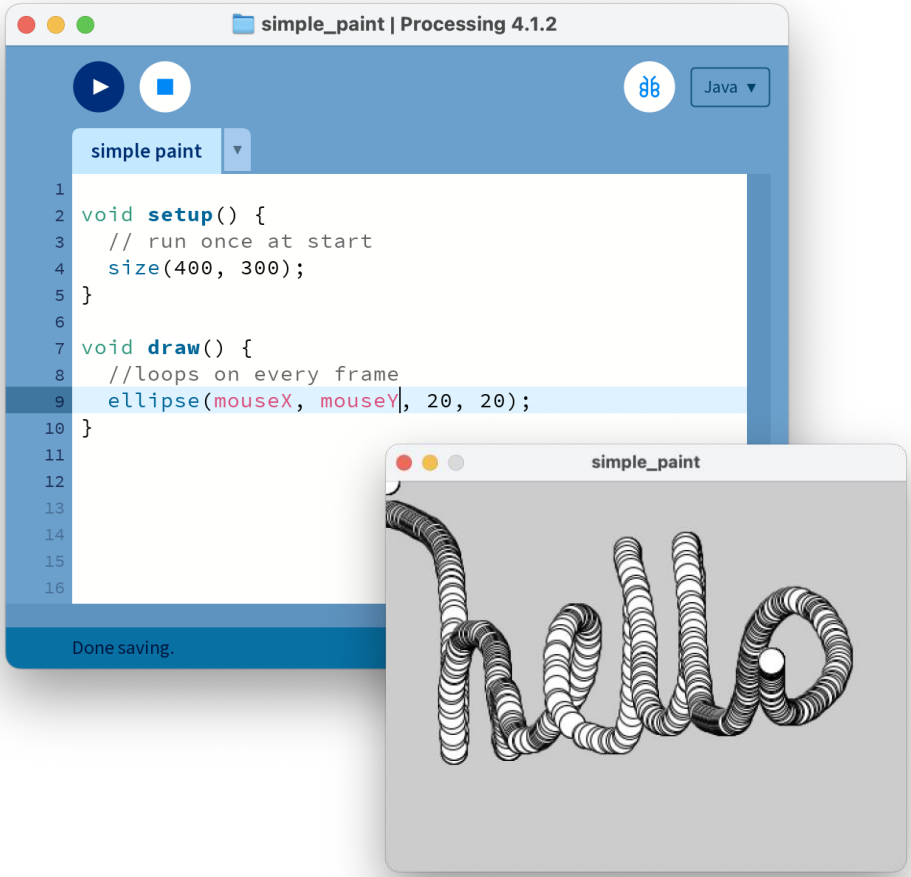
- void **setup**(){
  // runs once at the start
  }

- void **draw**(){
  // runs every frame in a loop
  }



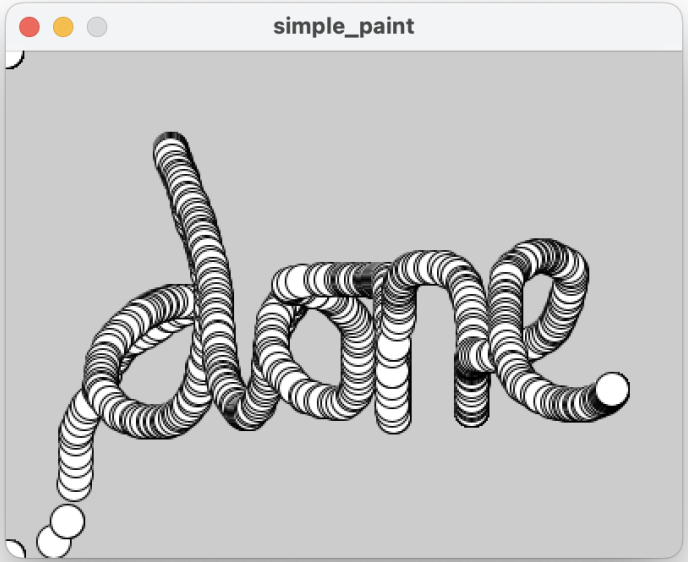
mouseX & mouseY

- Processing has some handy global variables predefined that you can access
  - **mouseX** returns the current mouse x pos
  - **mouseY** returns the current mouse y pos
- The IDE highlights the variable pink to show that it's a predefined variable.
- Other useful ones include **width**, **height** of the sketch window. And previous mouse positions **pmouseX** and **pmouseY**.



MVP achieved!

- You all now have the starting point of a very basic paint program.
    - Try painting the person next to you.
- Your job for the remainder of the workshop is to work in pairs using pair programming and a kanban board to improve on this.
- You will be testing your creation out on another person (with no instructions!) to make a short life drawing portrait at the end of the workshop. But… here are some starting points:



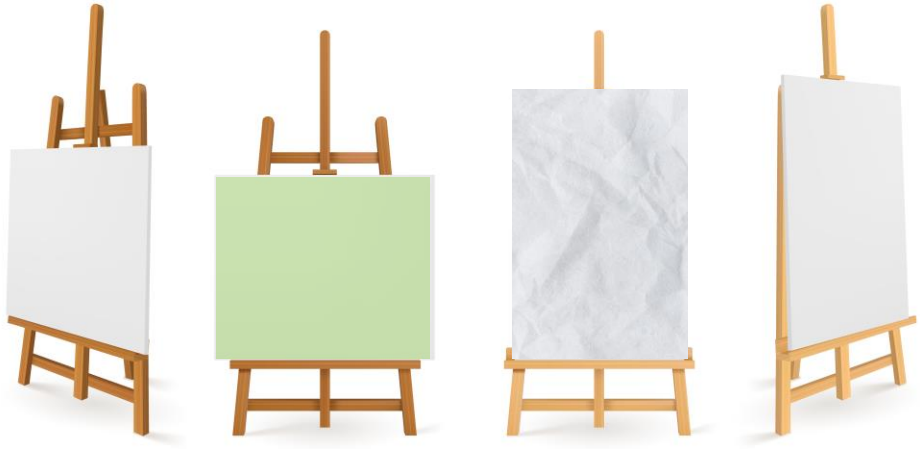
Changing Colour

- Colours values are between 0-255
- **fill**(*red*, *green blue*)
  - Changes the fill colour for the next drawing command.
- **stroke**(*red*, *green*, *blue*)
  - Change the colour for the outline of the next shape

image

Canvas / Background

- Change the background colour of the sketch by calling background(*r,g,b*)
- Try loading an image as a background using image() … note, do this in setup, rather than the draw loop!
- Explore using paper, canvas textures, a sketchbook or use a photograph (say of sky + clouds for a cloud drawing app)



image

Random

- Try randomising some elements with:

  random(*max*)
  random(*min, max*)

- For instance filling a shape with a random colour:

  - **fill**(**random**(255, **random**(255), **random**(255));

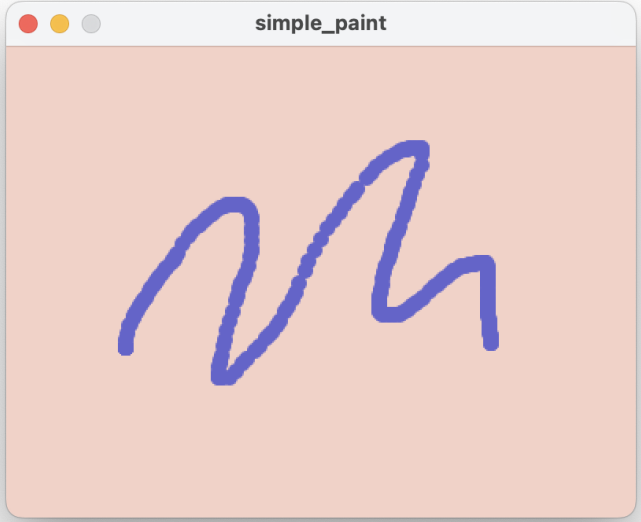
image

simple_paint | Processing 4.1.2

Java

simple paint

```
1
2  void setup() {
3    // run once at start
4    size(400, 300);
5    background(240, 210, 200); // light pink
6  }
7
8  void draw() {
9    //loops on every frame
10   if (mousePressed) {
11     noStroke();
12     fill(100, 100, 200); // light blue
13       ellipse(mouseX, mouseY, 10, 10);
14   }
15 }
16
17
18
```

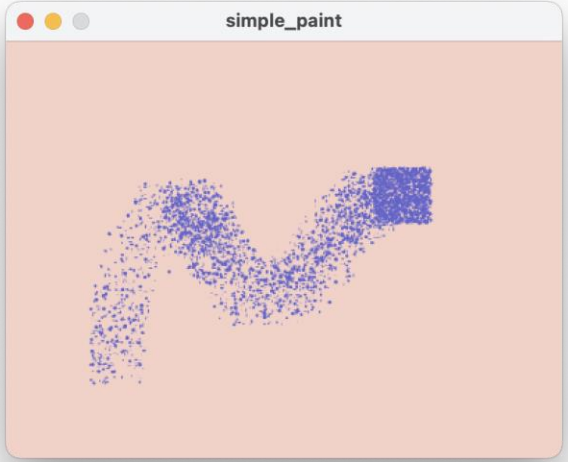


simple_paint | Processing 4.1.2

Java ▾

simple paint

```
1
2  void setup() {
3    // run once at start
4    size(400, 300);
5    background(240, 210, 200); // light pink
6  }
7
8  void draw() {
9    //loops on every frame
10   if (mousePressed) {
11     noStroke();
12     fill(100, 100, 200); // light blue
13     for (int n = 0; n < 10; n++) {
14       ellipse(mouseX+random(-20, 20), mouseY+random(-20, 20), random(3), random(3));
15     }
16   }
17 }
18
19
```



Spraypaint – could be better, perhaps use **cos**() and **sin**() to create a circular area rather than a square?

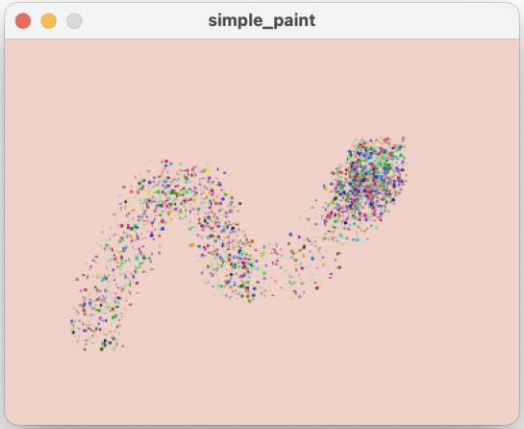
simple_paint | Processing 4.1.2

Java

simple paint

```
1
2  void setup() {
3    // run once at start
4    size(400, 300);
5    background(240, 210, 200); // light pink
6  }
7
8  void draw() {
9    //loops on every frame
10   if (mousePressed) {
11     noStroke();
12     for (int n = 0; n < 10; n++) {
13       fill(random(250), random(250), random(250)); // RANDOM COLOUR DROPS
14       ellipse(mouseX+random(-20, 20), mouseY+random(-20, 20), random(3), random(3));
15     }
16   }
17 }
18
19
```

Done saving.

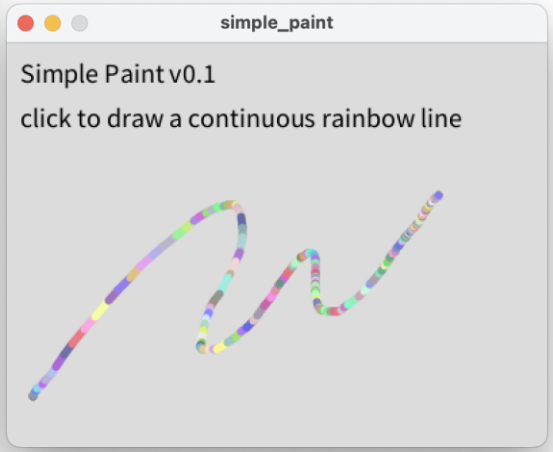


Spraypaint – move the **fill**() command and change the colour for every drop? Try passing two arguments to **random**(*min*, *max*) for a more realistic palette

simple_paint | Processing 4.1.2

simple paint

```
1
2  void setup() {
3    // run once at start
4    size(400, 300);
5    background(220); // light grey (single argument is greyscale)
6    textSize(20);
7    fill(0); // black
8    text("Simple Paint v0.1 \nclick to draw a continuous rainbow line", 10, 30);
9  }
10
11 void draw() {
12   //loops on every frame
13   if (mousePressed) {
14    strokeWeight(6);
15    stroke(random(100, 255), random(100, 255), random(100, 255));
16    line(pmouseX, pmouseY, mouseX, mouseY);
17   }
18 }
19
20
21
```



Documentation + **Commenting** - make sure that a new user can work out how to use your app, and please remember to use comments, either // or /*multiline*/ so that your code is legible

Challenges

These are potential challenges that you could start to fill your Kanban board up with:

- Add a way to change brush colour
- Change the brush size
- Add key commands **keyPressed**()
- Save an image with save("example.jpg")
- Multiple brush types, select with a key
- Use an image() as a brush, rotate randomly to vary the stroke
- Add an eraser. Can this be on right click?
- Add a way to reset/clear the whole canvas
- Auto-scribbler, automatically nudge the pen around the drawing position
- Symmetry – draw a second point on the opposite side of the canvas
- Create smooth lines by drawing between last and current mouse position: line(*p*mouseX, pmouseY, mouseX, mouseY)
- Add opacity to your brush by passing a fourth parameter (0 = transparent, 255 = opaque): fill(red, green, blue, alpha)
- Create lined or grid paper using a for loop.
- Change the **strokeWeight**() of your line based on the speed of mouse movement. Use pmouseX and pmouseY along with mouseX and mouseY to determine the distance moved

- **NOTE**: please add **documentation** – someone's going to be using your system without you being able to explain it. Use **println**() to send text to console or even better **text**() to write to screen.

Going Further

- Many more functions documented in the Processing Reference:
  - https://processing.org/reference
- Look through the reference and see what functions could be interesting to try out
- More info on the Processing environment here:
  - https://processing.org/environment/

`mouseButton`	Shows which mouse button is pressed
`mouseClicked()`	Called once after a mouse button has been pressed and then released
`mouseDragged()`	Called once every time the mouse moves and a mouse button is pressed
`mouseMoved()`	Called every time the mouse moves and a mouse button is not pressed
`mousePressed`	Variable storing if a mouse button is pressed
`mousePressed()`	Called once after every time a mouse button is pressed
`mouseReleased()`	Called every time a mouse button is released
`mouseWheel()`	The code within the `mouseWheel()` event function is run when the mouse wheel is moved
`mouseX`	The system variable that always contains the current horizontal coordinate of the mouse
`mouseY`	The system variable that always contains the current vertical coordinate of the mouse
`pmouseX`	The system variable that always contains the horizontal position of the mouse in the frame previous to the current frame
`pmouseY`	The system variable that always contains the vertical position of the mouse in the frame previous to the current frame

Pair Programming + Kanban

~60 mins

- Partner up. Swap roles regularly.
- Use a Kanban board to plan and track which features you will be working on. Three columns, move features across:
  - *Not Started - > In Progress -> Done*
- Start simple! Keep features small.
- Consider adding a ‘shelved’ or ‘parked’ column to put features in that aren’t working out (given 90min timescale). Don’t get stuck!

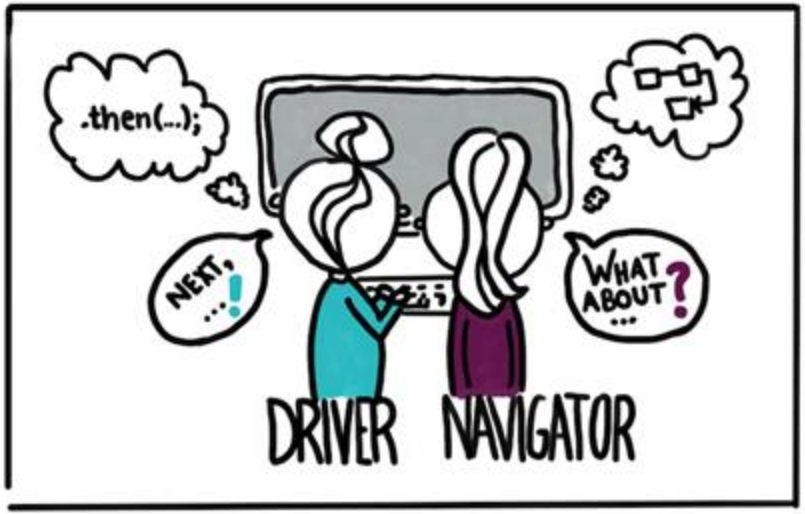
Pair Programming Roles

Driver (Helm)

- The person typing
- Focused on the short-term goal
- Places longer-term goals on backburner
- Talks through what they are doing

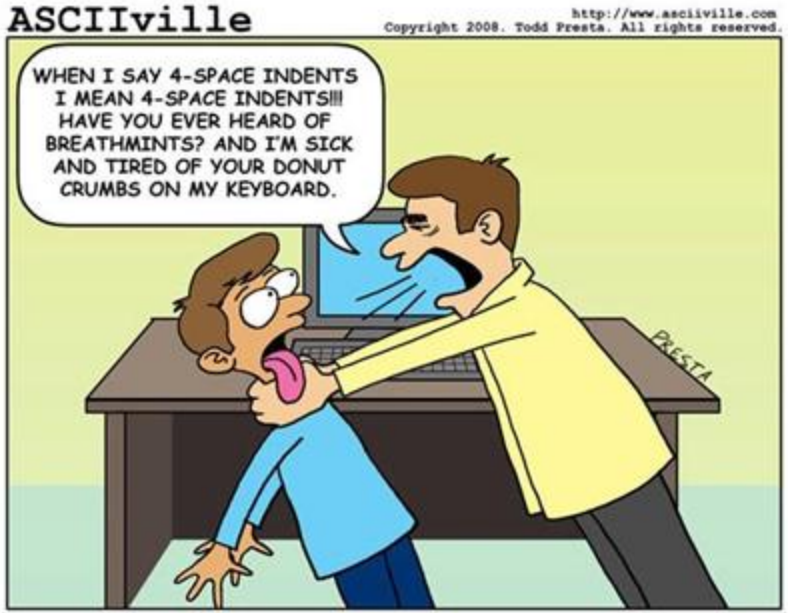
Navigator (Tactician)

- Observes the drivers work
- Reviews code in real-time
- Notes suggestions
- Scans horizon for longer-term issues



Pair Programming Tips

- **Distractions**. Don't check your phone/mail. Stay focused. Build in more individual time if you need it.
- **Micro-management.** Stick to higher level comments, and avoid saying things such as "now type..."
- **Impatience.** Don't jump right in when the driver makes a typo. They may have seen it and just haven't gone back to correct. Avoid breaking their flow.
- **Keyboard hogging.** Make sure to stick to a rotation schedule and avoid sticking to one role.



The dark side of pair programming.

User Testing

15 mins

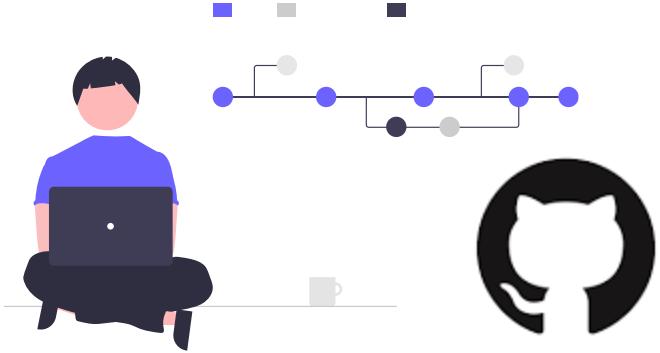
- Find another pair and swap over.
- **Without any instruction**, use their paint app to draw a portrait of your pair programming partner (take turns)
- Show the creators of the app how you used it, and your artistic results!
- What worked? What didn't? Did you enjoy it? What would you improve? What was similar with your own? How was the resulting portrait?

image

Week	Date	**Workshop** Monday 09:00-11:00 *MVB 2.11 PC*	**Lecture** Monday 12:00-13:00 *QUEENS BUILDING, 1.40 PUGSLEY*	**Groupwork**
1	22/01/24	Teams, Project Brief [slides] [project brief] [github intro slides]	Introduction and Process [slides] [materials]	Research games, create list on team repo. Install Processing
2	29/01/24	Intro to Processing [slides]	Agile Software Development [slides]	Decide on two game ideas
3	05/02/24	Paper Prototyping, Agile Techniques, Ideas Clinic [slides]	Requirements Engineering [slides] [materials]	Collect requirements. Decide on final idea
4	12/02/24	Requirements [slides]	Object Orientated Design [slides] [materials]	Add requirements section to report
5	19/02/24	Classes Activity, Mock test, Game jam, Summer project prep [slides]	Software Quality and Testing [slides] [materials]	Develop a working prototype over reading week!
6	26/02/24	GAMES JAM	READING WEEK	
7	04/03/24	IN CLASS TEST (assessing lectures 1-4); Testing [slides]	Project Management [slides] [materials]	Define team roles, Estimate sprint effort with planning poker
8	11/03/24	Planning Poker [slides] [heuristic evaluation sheet]	HCI Evaluation Part One [slides]	Write up evaluations from workshop. Plan qualitative assessment (of your choice). Add two difficulty levels to your game.
9	18/03/24	Think Aloud and Heuristic Evaluation[slides]	HCI Evaluation Part Two [slides]	Add quantitative assessment (of your choice) to report
	25/03/24	SPRINT 1	EASTER week 1	
	01/04/24	SPRINT 2	EASTER week 2	
	08/04/24	SPRINT 3	EASTER week 3	
10	15/04/24	HCI Quantitative Task	Software Engineering Extended - Sustainability [slides] [materials]	Develop Game
11	22/04/24	IN CLASS TEST (assessing lectures 5-9)	Coursework Feedback Discussion of marking scheme [slides]	Finish Report
12	29/04/24	Game Demo Day **Monday 29th April 9am-11am**, MVB 2.11 Demonstrate your game. Markers will have a strict 5min window to assess your game, be ready to allow 1-2mins of gameplay and 2-3mins of answering questions.	Feedback on in-class tests (tbc)	Submit Report + Video to Blackboard **Thursday 2nd May 1pm** Submit entire repo as a single zip file to Blackboard. Make sure link to video is clearly displayed at top of repo. We prefer that you have the video on a streaming service of your choice and not contained within the repo so that we're not having to download video files.

homework / groupwork

- Continue brainstorming game ideas.
- Decide on **TWO IDEAS** to bring along to next weeks workshop.
- **Add these two ideas to your repo**! One paragraph each. Images welcome!



Agile Software Development

Dr Jon Bird
jon.bird@bristol.ac.uk

Thanks to Dr Simon Lock who developed many of these slides for an earlier version of this unit.

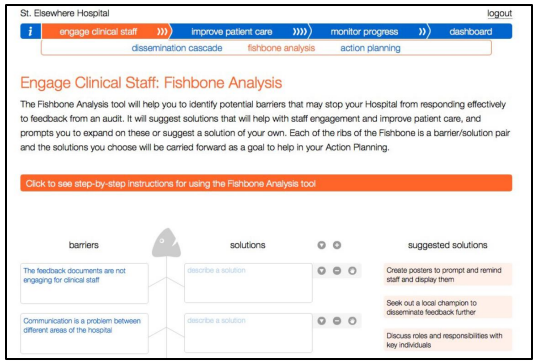
Images are royalty free from www.pexels.com

Today's Lecture

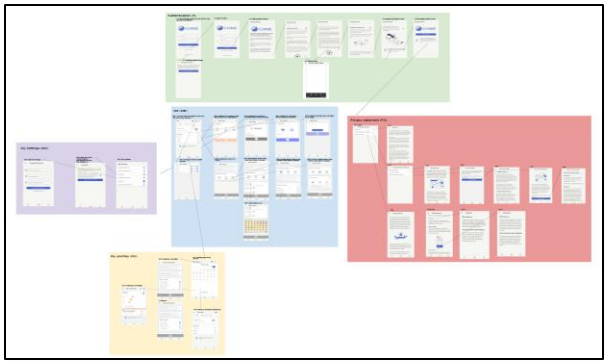
- A digital health project developed using the waterfall approach, which was the focus of last week
- Waterfall versus agile approaches to software development
- Agile software development, including: extreme programming (which includes pair programming); test-driven development; scrum; and Kanban
- A digital health project developed using an agile approach
- Recommended reading

Digital health projects

- My research involves developing digital technologies to address health issues
- What software development process do I employ in my research?



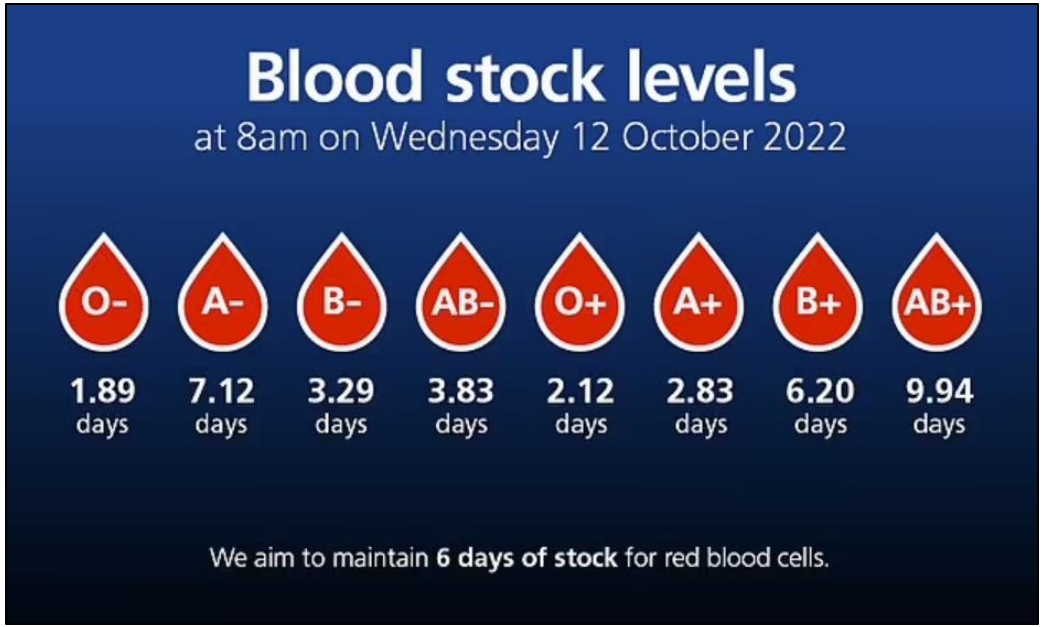
Affinitie



CHIME

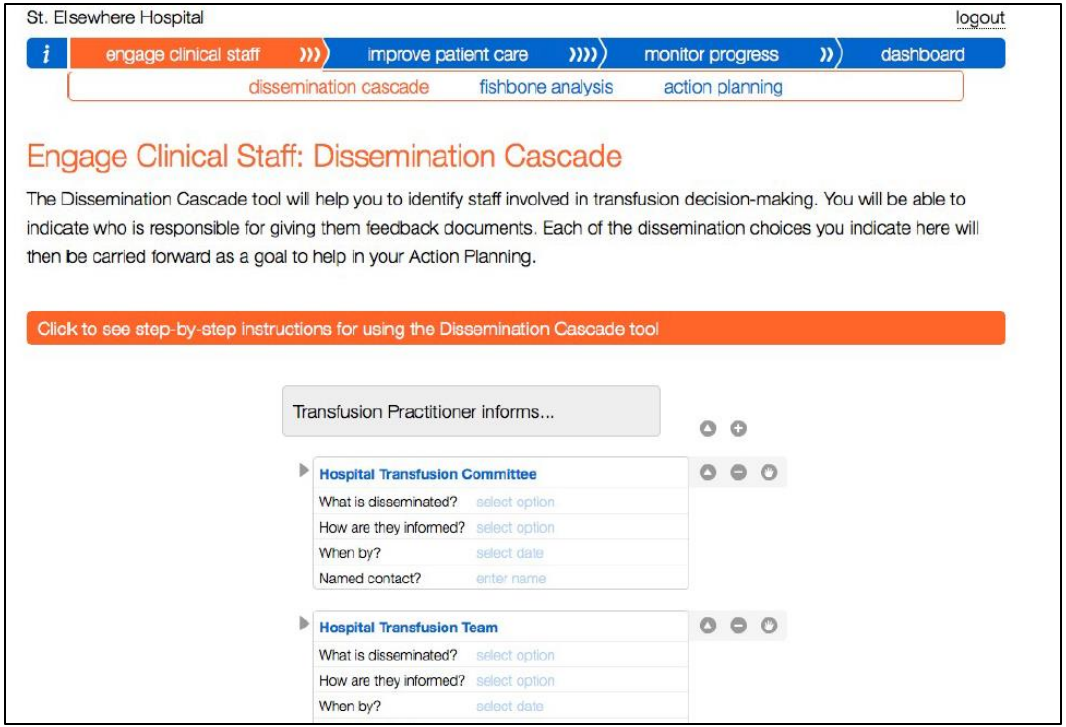
Affinitie - motivation

- There are 3 million blood transfusions annually in the UK
- Around 20% are unnecessary
- Blood components are a scarce resource
- There are health risks associated with blood transfusions
- How can we reduce unnecessary blood transfusions?



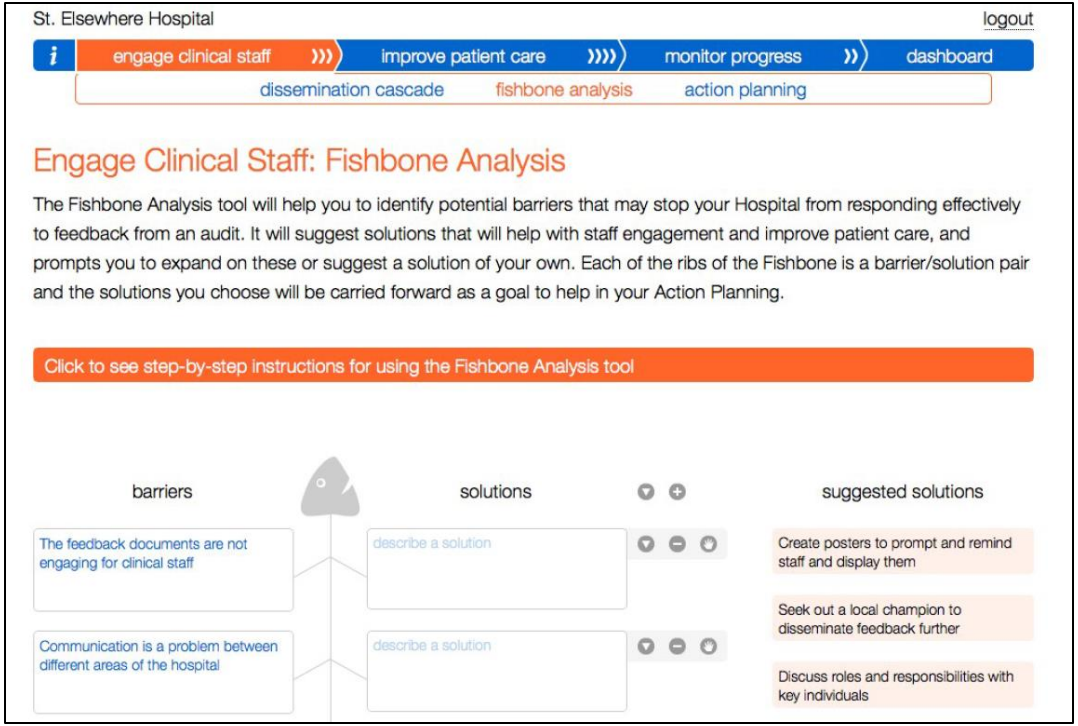
Affinitie – software

- I was asked to join an existing research project to develop a web toolkit for transfusion practitioners
- The research team consisted of health psychologists, statisticians, doctors and NHS Blood and Transplant
- The aim was to help transfusion practitioners disseminate blood transfusion audit results to everyone in the hospital

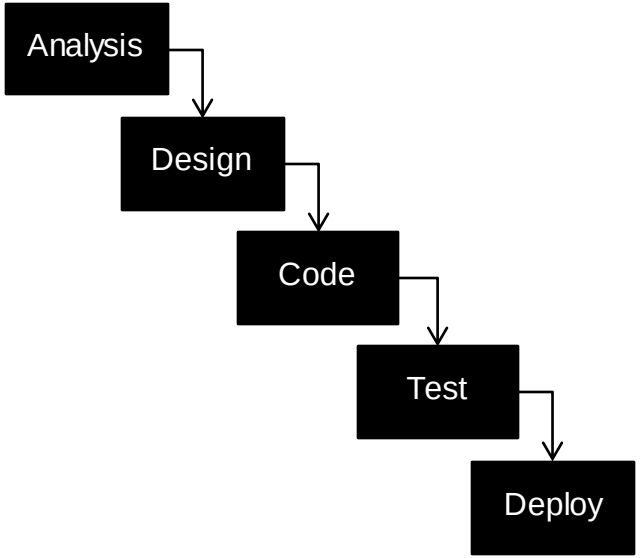


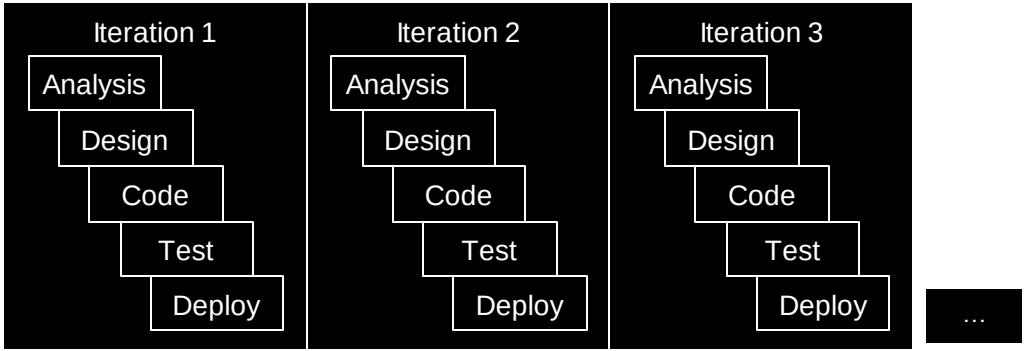
Affinitie – software development

- We were given **a clear set of requirements** by the research team
- They had already developed a set of paper-based tools for transfusion practitioners to help them disseminate blood transfusion audit results
- Software development approach: **waterfall**



Waterfall versus agile life cycles





What is Agile Software Development?

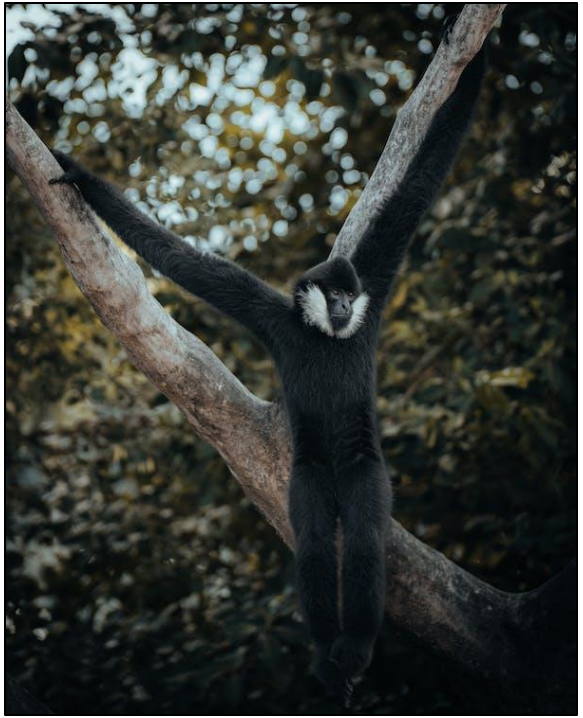
- Agile is a way of thinking about software development
- In winter 2001, 17 software developers met at a ski resort in Utah and drafted a manifesto outlining an alternative way to develop software to the documentation-driven software development processes of the time
- The manifesto is succinct and puts forward four key values for software development

The Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

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

That is, while there is value in the items on the right, we value the items on the left more



How should you read the manifesto?

- The values are purposefully provocative in order to get people to think about software development
- The Agile Manifesto is **not** proposing that we disregard aspects of software development like processes, tools and documentation
- Rather, the Agile Manifesto wants people to think about alternative ways of doing aspects of software development

Agile was created by coders for coders

Coders like
Writing quality code
Ticking things off their to do list
Impressing clients by showing them working software

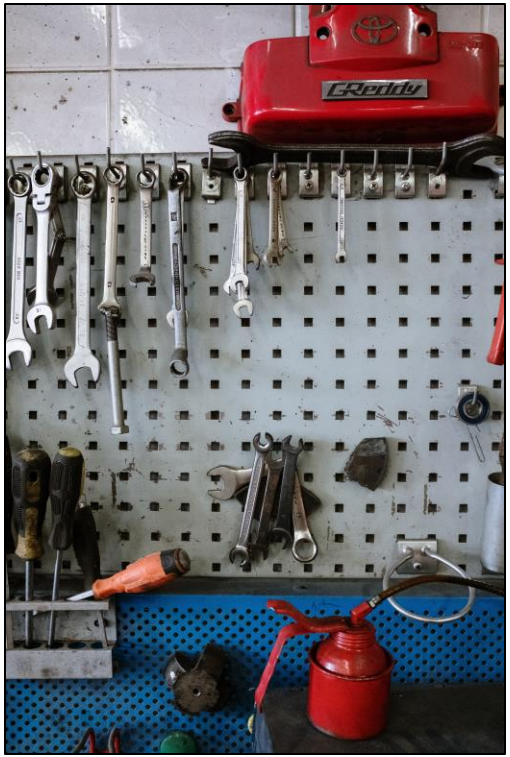
Coders dislike
Writing extensive documentation
Committing to a final design in advance
Being micromanaged
Working to big, immovable deadlines

Twelve agile principles

Satisfy clients' needs	Satisfy coders' needs
The highest priority is satisfying the client (by delivering working software early and continuously)	Work at a steady, sustainable pace (no heroic efforts)
Embrace change (even late in the development cycle)	Rely on self-organising teams
Collaborate every day with the client	Teams reflect regularly on their performance
Use face to face communication	Progress is measured by the amount of working code produced
Deliver working software frequently	Continuous attention to technical excellence
	Minimise the amount of unnecessary work
	Build teams around motivated individuals

Agile Methods

- There are various approaches that adhere to Agile values and principles
- Different companies choose different approaches
- Popular methods include:
    - Extreme Programming (XP) (the two co-creators were signatories of the manifesto)
    - Test-driven development (creator was a signatory)
    - Kanban
    - Scrum (the two co-creators were signatories)
- We'll introduce some key practices from each of these methods that we think you will be useful in this unit and your summer projects
- There are links in the reading to some of these methods



Hands up if any of these apply to you

- Your code structure is complex and “sophisticated”
- You work primarily on your own
- In group projects you are responsible for just your own code
- You write code in your own style
- You work some weekends and so some “all-nighters”
- You code develops in “heroic bursts”

Extreme Programming Ethos

- **Simple design**: use the simplest way to implement features
- **Sustainable pace**: effort is constant and manageable
- **Coding standards**: teams follow an agreed style and format
- **Collective ownership**: everyone owns all the code
- **Whole team approach**: everyone is included in everything

Extreme Programming Practices

- **Pair programming**: two heads are better than one
- **Test driven**: ensure the code runs correctly
- **Small releases**: deliver frequently and get feedback from the client
- **Continuous integration**: ensure the system is operational
- **Refactor**: restructure the system when things get messy

Pair programming in more detail

Code is written by two programmers on one machine:

- The **helm** uses the keyboard and mouse and does the coding
- The **tactician** thinks about implications and potential problems

- Communication is essential for pair programming to work
- Pair programming facilitates project communication

- The pair doesn't "own" that code - anyone can change it
- Pairings can (and should) evolve at any time

- All code is reviewed as it is written
- The **tactician** is ideally positioned to recommend refactoring

The impact of pair programming

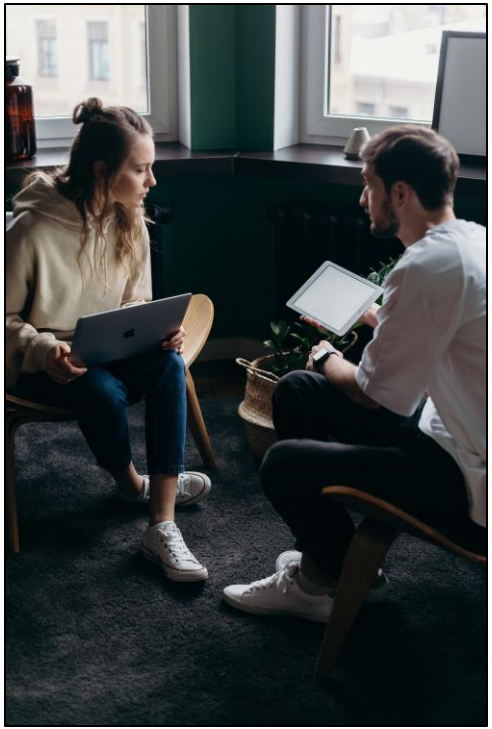
Research studies have assessed the impact of pair programming and identified a number of benefits

Single programmer 77 source lines per month versus pair programming 175 source lines per month

[Jensen, 2005]

15% increase in development-time costs but improves design quality, reduces defects, reduces staffing risk, enhances technical skills, improves team communications and is considered more enjoyable at statistically significant levels.

[Cockburn & Williams, 2000]



Test-driven development in more detail

- Tests are written before any code and they drive all development
- A programmer's job is to write code to pass the tests
- If there's no test for a feature, then it is not implemented
- Tests are the requirements of the system

The benefits of test-driven development

- Code coverage

We can be sure that all code written has at least one test because if there were no test, the code wouldn't exist

- Simplified debugging

If a test fails, then we know it must have been caused by the last change

- System documentation

Tests themselves are one form of documentation because they describe what the code should be doing

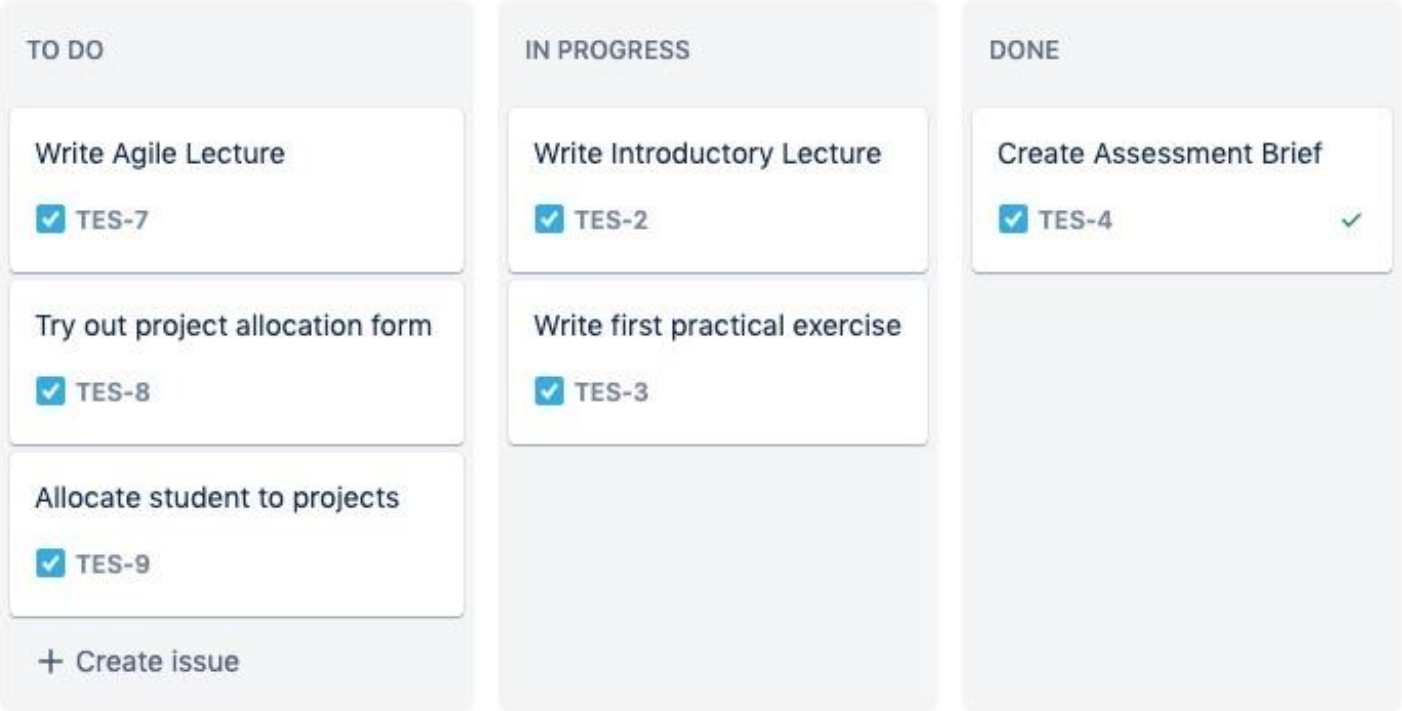
Scrum

- Scrum is a project management approach
- Some key concepts are:
- **The Scrum** – a **stand-up** daily meeting of the entire team
- **Scrum Master** - team Leader
- **Sprint** - a short, rapid development iteration
- **Product Backlog** – To do list of jobs that need doing
- **Product Owner** – the client (or their representative)

Kanban Board

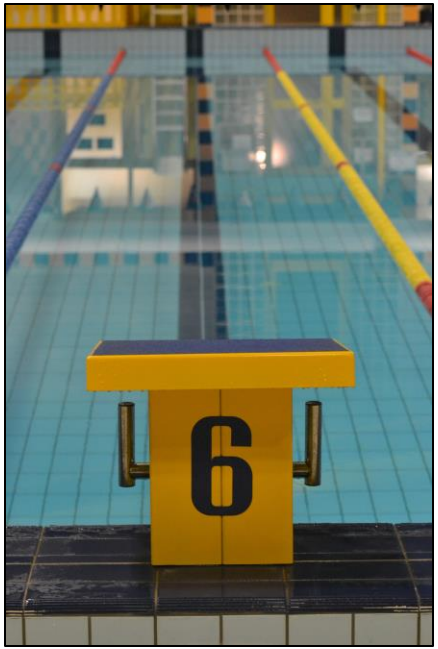
- A concept taken from the Kanban method which was first defined in 2007 but came out of a scheduling system developed by Toyota in the 1950s for just-in-time manufacturing
- In Japanese “kanban” means “visual board” or “sign”
- It’s basically a flexible “to do” list tool
- Issues progress through various states from ”To do” to “Done”
- It was originally implemented as post-it notes on a whiteboard
- Various digital tools now fulfil the same function e.g. Jira

Jira Kanban Board



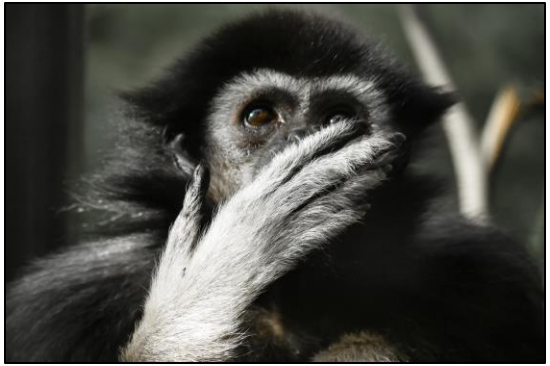
Columns (Swim lanes)

- It is common to use three columns
- But Jira allows you to customize the layout
- For example, you might have columns for:
  - Backlog
  - Being Verified
  - Awaiting integration
- Do what works for your team but make sure you have a “Done” column



Problems with Agile

- Hard to draw up legally binding contracts - a full specification is never written in advance
- Good for green-field development when you have a clean slate and are not constrained by previous work. However, it's not so effective for brownfield development which involves improving and maintaining legacy systems.
- Works well for small co-located teams, but what about large distributed development ?
- Relies on the knowledge of developers in the team but what if they aren't around (holidays, illness, turnover) ?



CHIME - motivation

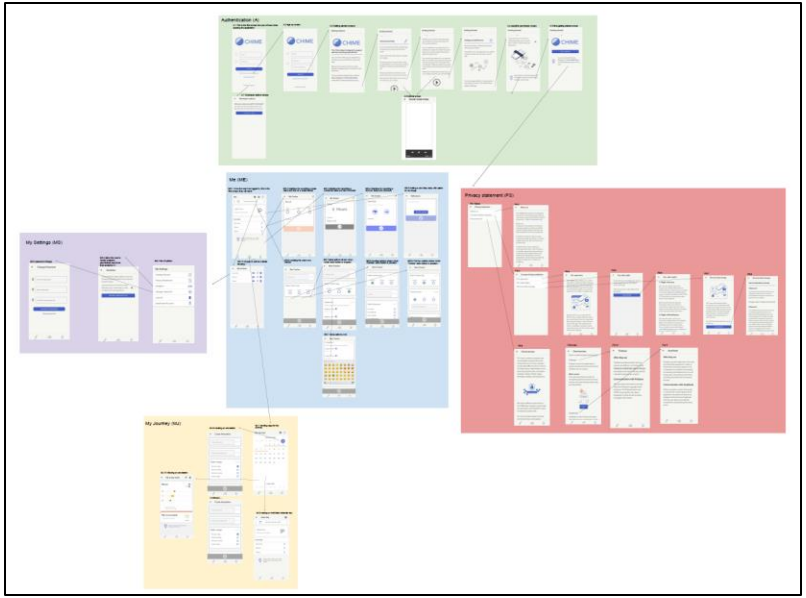
- Tracking health and lifestyle data can help people manage long term conditions
- Sharing these data with healthcare professionals can improve clinical decision making

CHIME - software

- Chime is an app designed for people living with HIV (PLHIV)
- It was developed by researchers at a number of UK universities in collaboration with the Terrence Higgins Trust, the leading HIV charity in the UK

CHIME – software development

- I project manged two programmers who developed the app code at UoB
- We initially designed and built a prototype app based on the requirements identified by other researchers on the project
- The app was evaluated by stakeholders who identified more requirements
- We then carried out a series of four two-week sprints and presented working code at the end of each sprint to other researchers
- The app was then evaluated by PLHIV
- Software development approach: **agile**



Reading

- Two research papers about evaluating pair programming mentioned in the lecture

R. W. Jensen (2005) A Pair Programming Experience, Overload, 13(65)

A. Cockburn & L. Williams (2001) The Costs and benefits of pair programming. Extreme programming examined. G. Succi and M. Marchesi, Addison Wesley: 223-243.

- A good overview of the Agile Manifesto and approach

Overview of the Agile Approach

- The Kanban method

https://kanbanize.com/kanban-resources/getting-started/what-is-kanban

- Unit testing

Unit testing in Java

A level of agility that progammers can aspire to

Paper Prototyping

Workshop 3

Ruzanna Chitchyan, Jon Bird, Pete Bennett
TAs: Alex Elwood, Alex Cockrean, Casper Wang

Today's Workshop

- By now your group should have **2 ideas**
- Paper prototype both game ideas, document with video (60mins)
- User feedback (30mins)
- Update your github (10mins)
- Next week you will have **one idea** which you will start developing

What is Paper Prototyping?

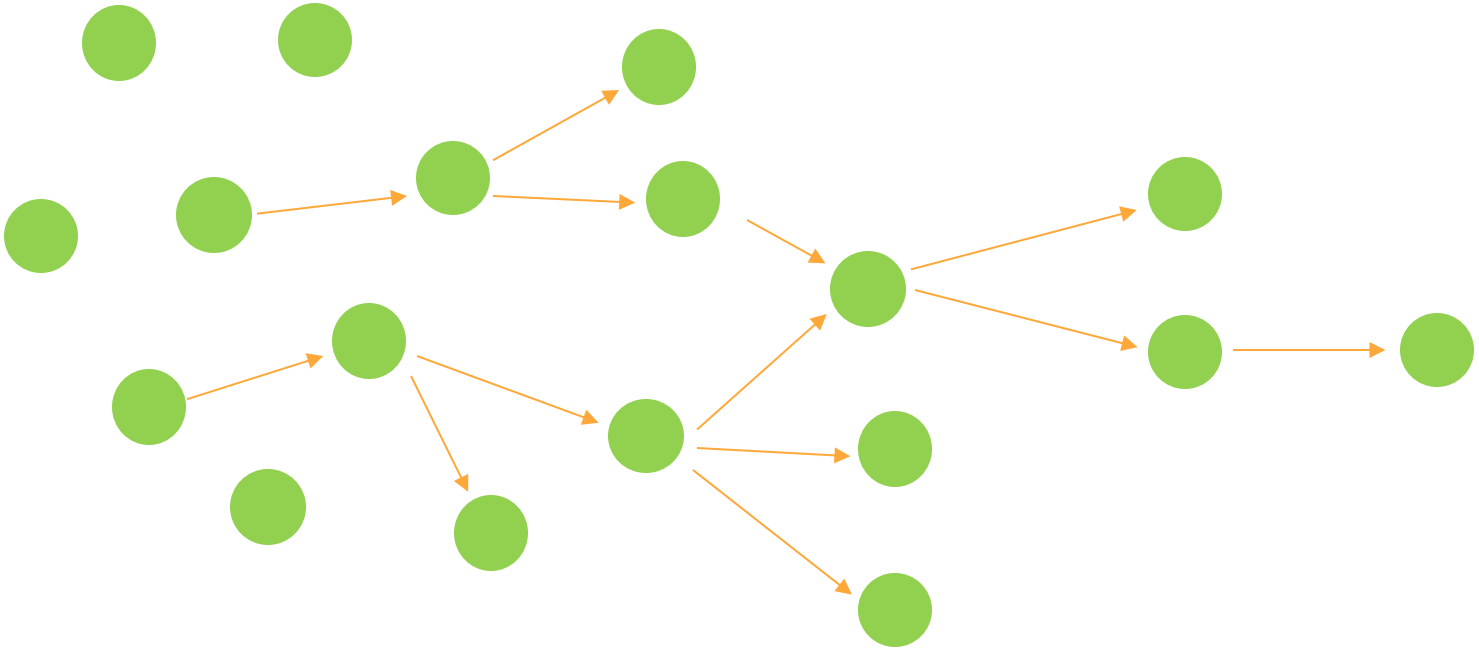
- Paper prototyping is a **user-centred** design method that is widely used in software engineering projects
- Most commonly, it involves the development of user interface mock-ups and drawn sketches which are presented to end users for evaluation
- Can lead to full wireframes and flow diagrams.
- Best done with actual paper – but using a slide deck (PowerPoint prototyping) can also work

What is Paper Prototyping?

- Paper prototypes are often used at the early stages of projects to provide feedback on:
  - Product concept and goals
  - Logic and flow of user journeys
  - The form of the user interface

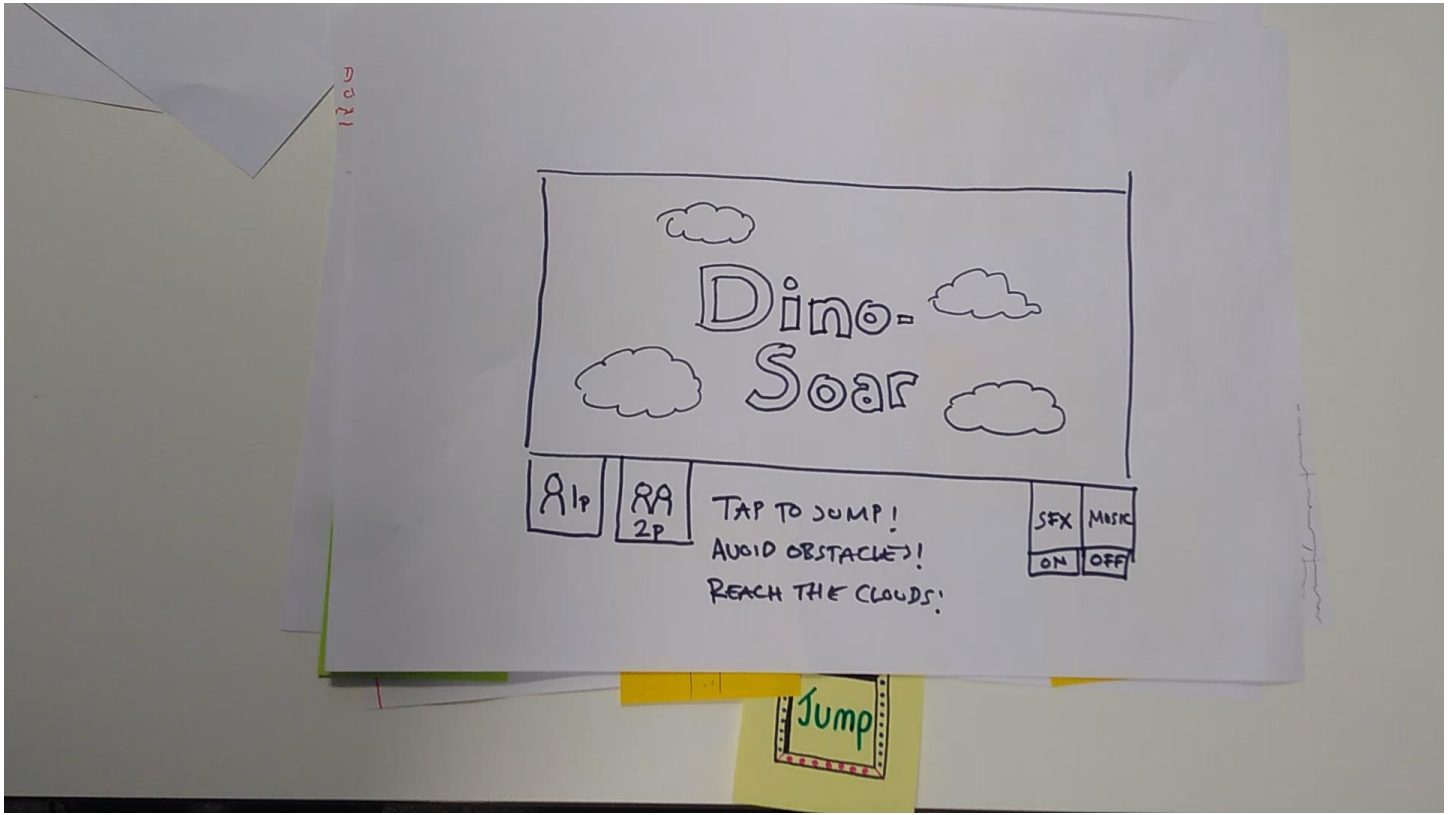
- This is a '**cheap**' process!
  - 1 interactive prototype (10h)
  - = 10 video prototype (1h each)
  - = 100 paper prototypes (6 mins each)
  - = 6000 sketches (6 sec each)

Game idea progress



the last week | today | this week

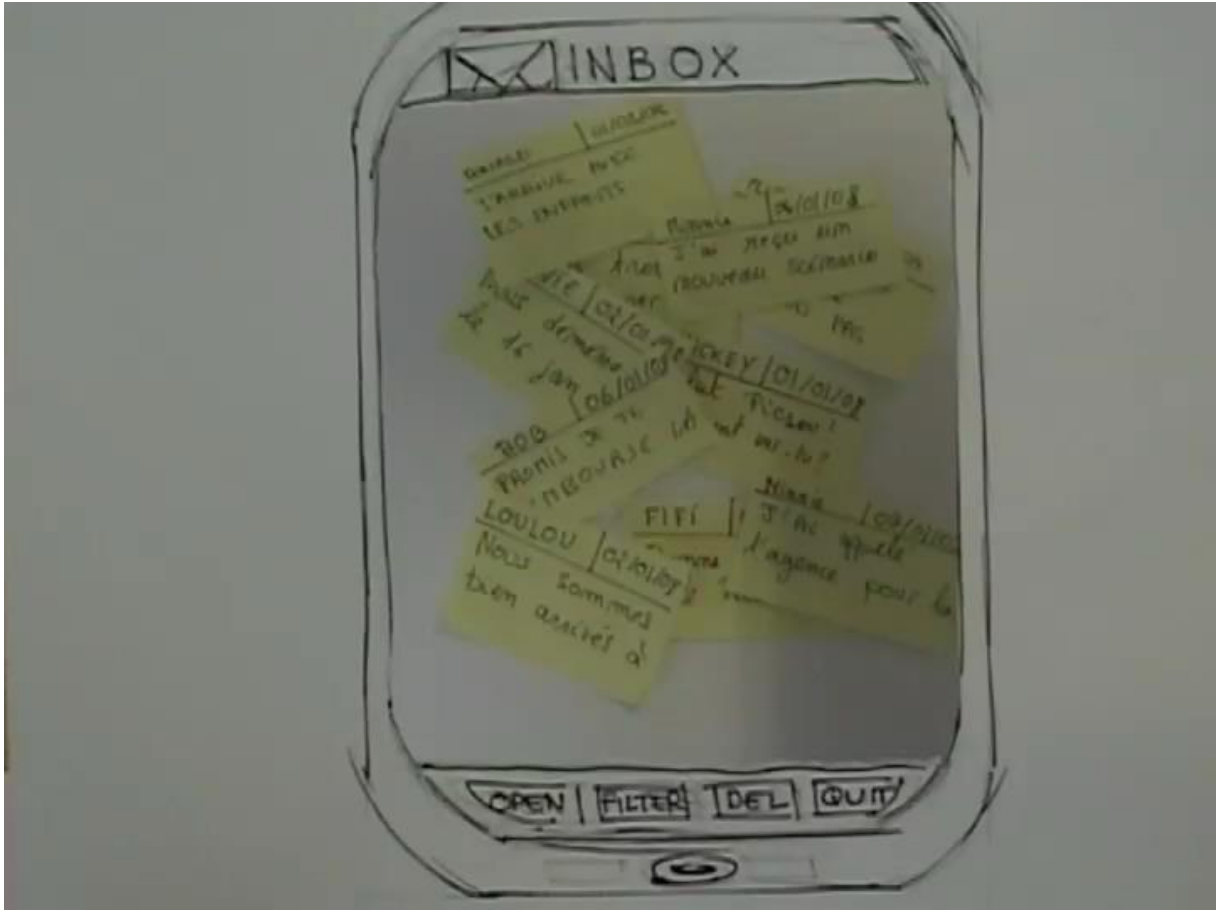
“prototype a one button game”



try **animation** – add **sounds** – plan for some **different outcomes**

use **colours** – use **animation**

plan the **story paths** – use **fill-in text**



use **post-it notes** – can use pause **jump cuts** (don't bother today!)

use **device as frame** – make things **larger** than real life to make it easier!

Why Paper Prototype?

- Quick and cheap design method
- Easy to make instant design change
- *Communication.* Helps the development team and external stakeholders better conceptualise your product and visualise how it will be used
- Helps us to better understand our product and users
- It is a low-investment means of evaluation:
  - Easier for users to give feedback, no fear of causing expensive change
  - Useful for testing multiple ideas or solution variations
- Makes a clearer distinction between conceptual errors and programmatic errors

Paper Prototype your TWO Games

60 mins

Create a ‘working’ version of each game in paper
… the second game may be done in Powerpoint

Animate each game in response to user input

Document each with a video

tips:

- use a (thick) **pen**, not pencil
- choose a **single** task to implement
- don’t forget (manual) **animation**
- use **placeholder text**
- make devices **larger** than life-size
- make sure that it’s **fun**

User feedback

30 mins

Pair up with the team next to you

Demo your two games to the other team and vice versa

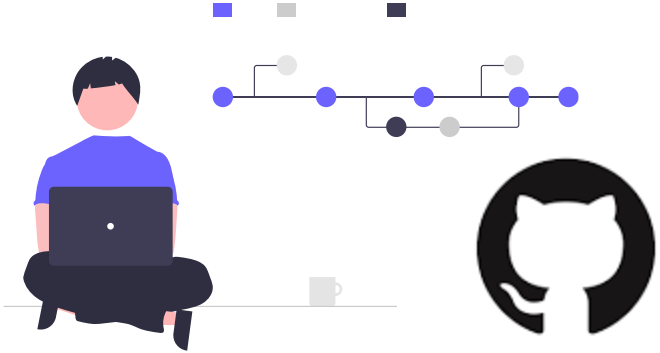
Give Feedback

- Did the game concept make sense?
- Did you understand the controls?
- Was there a good twist?
- Are there obvious challenges to tackle (you need to identify three!)
- Which was you favourite and why?

Week	Date	**Workshop** Monday 09:00-11:00 *MVB 2.11 PC*	**Lecture** Monday 12:00-13:00 *QUEENS BUILDING, 1.40 PUGSLEY*	**Groupwork**
1	22/01/24	Teams, Project Brief [slides] [project brief] [github intro slides]	Introduction and Process [slides] [materials]	Research games, create list on team repo. Install Processing
2	29/01/24	Intro to Processing [slides]	Agile Software Development [slides]	Decide on two game ideas
3	05/02/24	Paper Prototyping, Agile Techniques, Ideas Clinic [slides]	Requirements Engineering [slides] [materials]	Collect requirements. Decide on final idea
4	12/02/24	Requirements [slides]	Object Orientated Design [slides] [materials]	Add requirements section to report
5	19/02/24	Classes Activity, Mock test, Game jam, Summer project prep [slides]	Software Quality and Testing [slides] [materials]	Develop a working prototype over reading week!
6	26/02/24	GAMES JAM	READING WEEK	
7	04/03/24	IN CLASS TEST (assessing lectures 1-4); Testing [slides]	Project Management [slides] [materials]	Define team roles, Estimate sprint effort with planning poker
8	11/03/24	Planning Poker [slides] [heuristic evaluation sheet]	HCI Evaluation Part One [slides]	Write up evaluations from workshop. Plan qualitative assessment (of your choice). Add two difficulty levels to your game.
9	18/03/24	Think Aloud and Heuristic Evaluation[slides]	HCI Evaluation Part Two [slides]	Add quantitative assessment (of your choice) to report
	25/03/24	SPRINT 1	EASTER week 1	
	01/04/24	SPRINT 2	EASTER week 2	
	08/04/24	SPRINT 3	EASTER week 3	
10	15/04/24	HCI Quantitative Task	Software Engineering Extended - Sustainability [slides] [materials]	Develop Game
11	22/04/24	IN CLASS TEST (assessing lectures 5-9)	Coursework Feedback Discussion of marking scheme [slides]	Finish Report
12	29/04/24	Game Demo Day **Monday 29th April 9am-11am**, MVB 2.11 Demonstrate your game. Markers will have a strict 5min window to assess your game, be ready to allow 1-2mins of gameplay and 2-3mins of answering questions.	Feedback on in-class tests (tbc)	Submit Report + Video to Blackboard **Thursday 2nd May 1pm** Submit entire repo as a single zip file to Blackboard. Make sure link to video is clearly displayed at top of repo. We prefer that you have the video on a streaming service of your choice and not contained within the repo so that we're not having to download video files.

homework / groupwork

- Decide on one game idea!
- Update and improve paper prototype as required
- Your Github team page will have:
  - Your names + team photo
  - List of inspiration and initial ideas
  - Your two paper prototype ideas (include videos)
  - Your final idea (one paragraph + paper prototype images/video)



Requirements Engineering

Lecture 3

Ruzanna Chitchyan, Jon Bird, Pete Bennett
TAs: Alex Elwood, Alex Cockrean, Casper Wang

Overview

- What are requirements?
- Stakeholder Identification
- Functional and Non-Functional Requirements
- Describe system behavior and capture it in a model with Use Case Model
  - Use Case Diagram
  - Use Case Specification
- Requirements Quality

Requirements

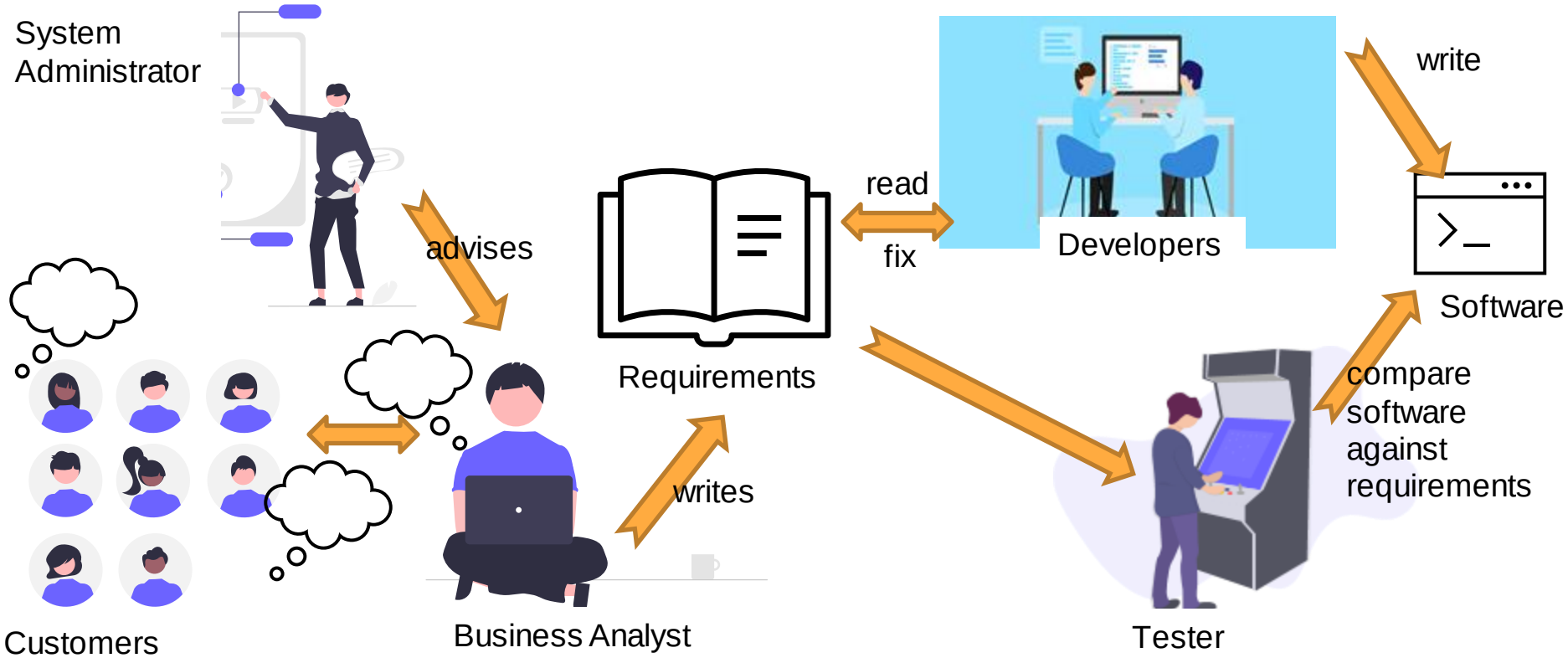
- **System requirements** specify a system, not in terms of system implementation, but in terms of user observation. Requirements record description of the system features and constraints.
  - **Functional requirements** specify user interactions with the system, they say **what** the system is supposed to do:
    - Statements of services the system should provide
    - How the system should react to particular inputs
    - How the system should behave in particular situations
    - May also state what the system should NOT do
  - **Non-functional requirements** specify other system properties, they say **how** the functional requirements are realised:
    - Constraints ON the services or functions offered by system
    - Often apply to whole system, not just individual features

Why do we need Requirements Engineering?

General Problems and Requirements Engineering

- Inconsistent terminology: people express needs in **their own words**
- **Conflicting** needs for the same system
- People frequently **don’t know** what they want (or at least can't explain !)
- Requirements **change** quite frequently
- Relevant people/information may **not be accessible**

Requirements are **communication mechanism**



Requirements are **instructions**

- On your own or in pairs
- Follow some instructions to draw.
- I'll then judge whether you followed the instructions correctly.
- NOT your artistic skill!

Drawing Activity: 5 min.

Requirements are **acceptance criteria**

- To be able to fairly assess whether the team have produced something that matches what you asked for, the thing that you asked for must be:
  - Unambiguous / Precise
  - Complete
  - Understandable / Clear

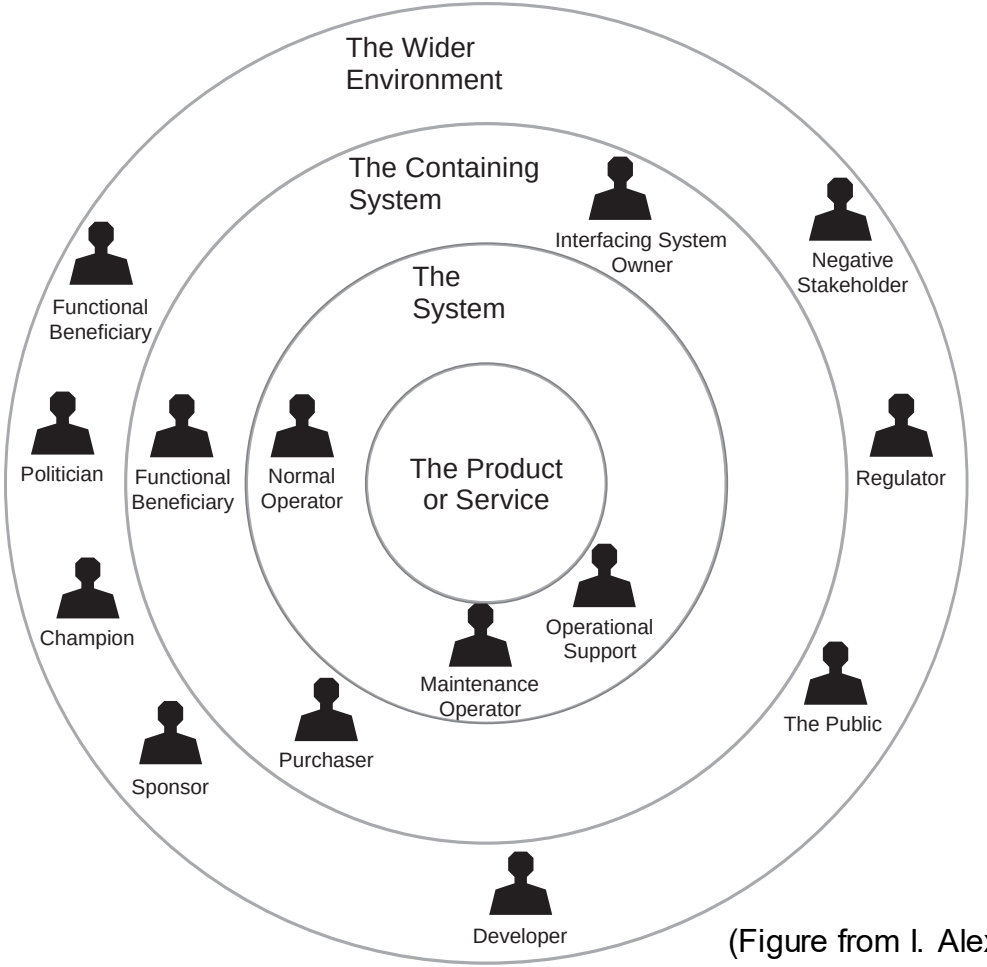
Analysing Requirements

1. Identify stakeholders involved with the system
2. Identify top-level user needs (e.g., as NFRs or “user stories”)
3. Break down stories into individual steps / Refine requirements
4. Specify atomic requirements (e.g., for each step in user stories)

Let’s look at each of these stages in turn…

1. Identify Stakeholders

The Onion model



(Figure from I. Alexander's book)

Stakeholders

Identification

- Clients
- Documentation, e.g., organisation chart
- Templates (e.g., onion model)
- Similar projects
- Analysing the context of the project

Keeping in mind:

- Surrogate stakeholders (e.g., legal, unavailable at present, mass product users)
- Negative stakeholders

2. Identify top level needs/concerns

Identify “User Stories”

A popular way to record user needs…

As a < type of user >, I want to < some goal >
so that < some reason >.

As a student, I want to be able to register for a module, so that I can learn about topics of interest to me.

As a customer, I want to be able to pay for a university course, so that I can attend the lectures to get a degree.

As a customer, I want to have my data kept securely, so that my privacy is protected.

What Is System Behavior?

- System behavior is how a system acts and reacts.
  - It comprises the actions and activities of a system.
- System behavior is captured in use cases.
  - Use cases describe the interactions between the system and (parts of) its environment.

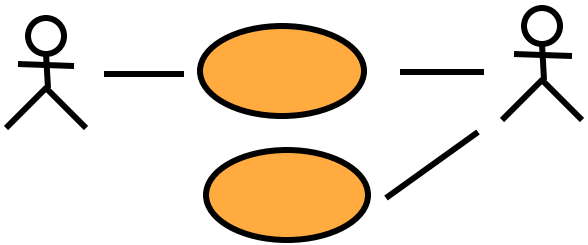
What is a use-case model?

- Describes the functional requirements of a system in terms of use cases
- Links stakeholder needs to software requirements
- Serves as a planning tool
- Consists of **actors** and **use cases**

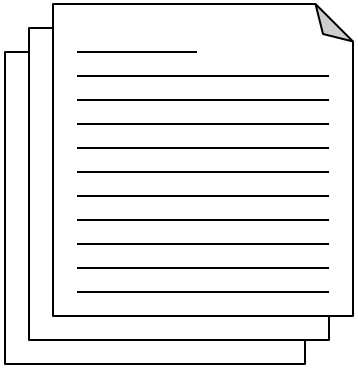
Capture a use-case model

- A use-case model is comprised of:

Use-case diagrams
(visual representation)

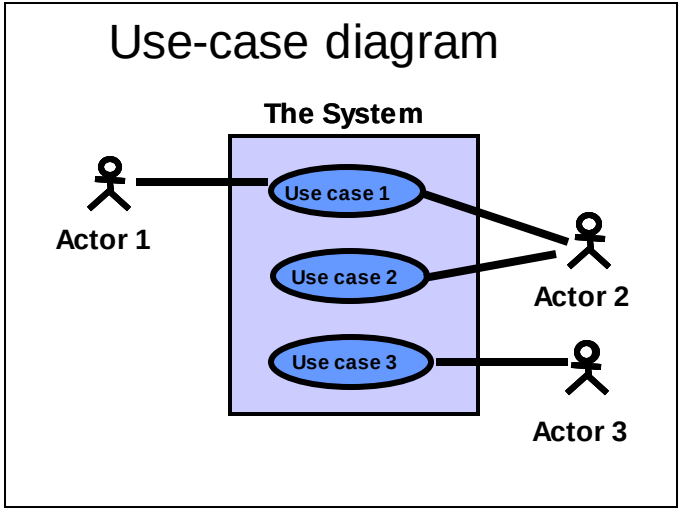


Use-case specifications
(text representation)



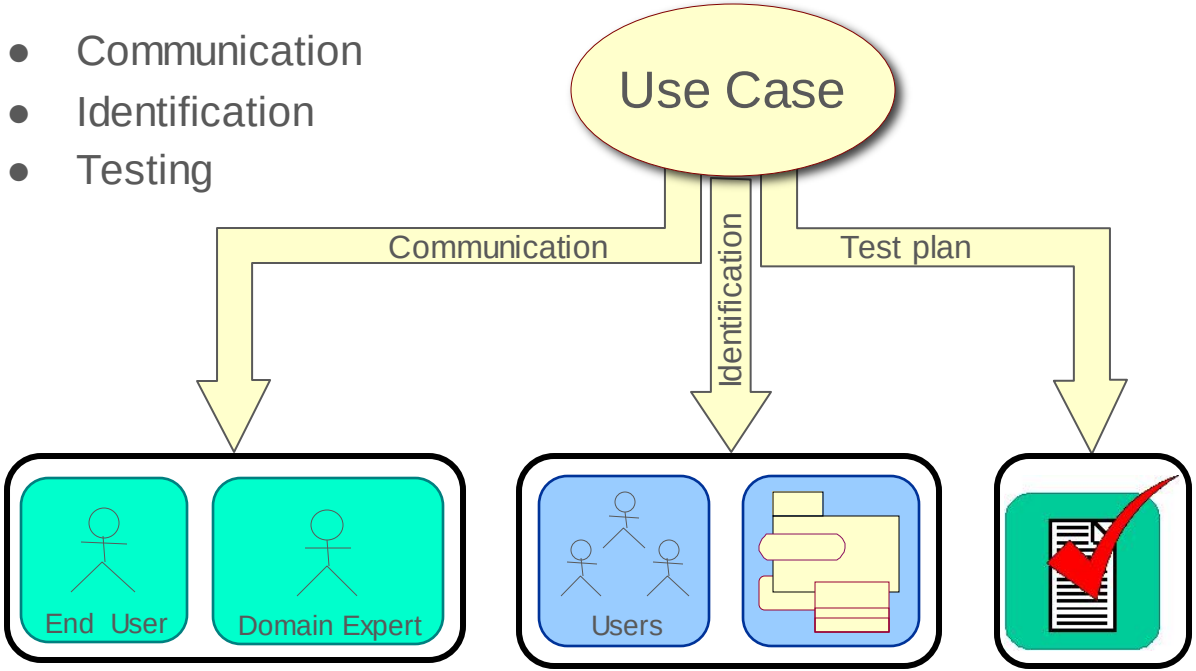
Use-case diagram

- Shows a set of use cases and actors and their relationships
- Defines clear boundaries of a system
- Identifies who or what interacts with the system
- Summarizes the behavior of the system



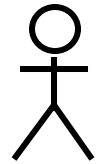
What Are the Benefits of a Use-Case Model?

- Communication
- Identification
- Testing



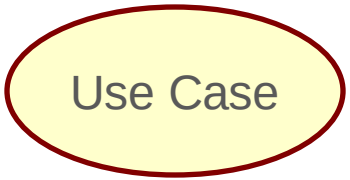
Major Concepts in Use-Case Modeling

- An actor represents anything that interacts with the system.



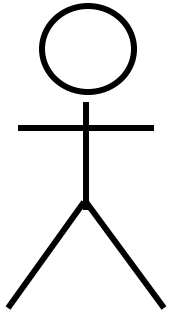
Actor

- A use case describes a sequence of events, performed by the system, that yields an observable result of value to a particular actor.



What Is an Actor?

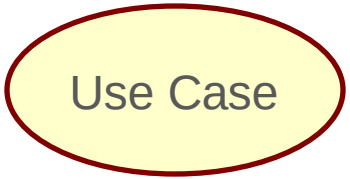
- Actors represent roles a user of the system can play.
- They can represent a human, a machine, or another system.
- They can actively interchange information with the system.
- They can be a giver of information.
- They can be a passive recipient of information.
- Actors are not part of the system.
  - Actors are EXTERNAL.



Actor

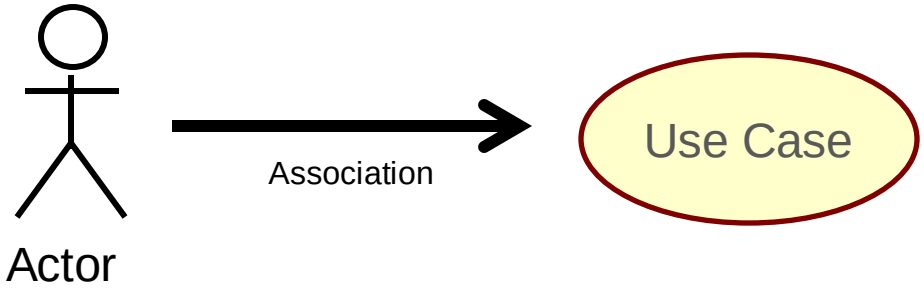
What Is a Use Case?

- Defines a set of use-case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor.
  - A use case models a dialogue between one or more actors and the system
  - A use case describes the actions the system takes to deliver something of value to the actor



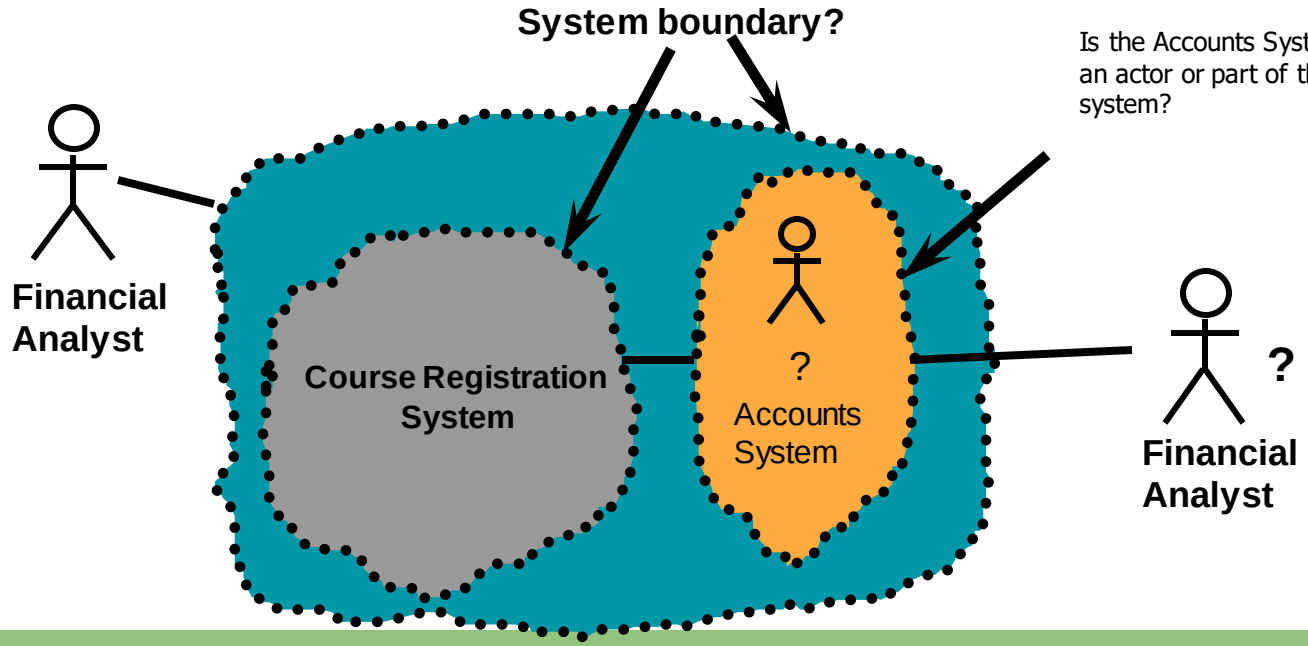
Use Cases and Actors

- A use case models a dialog between actors and the system.
- A use case is initiated by an actor to invoke a certain functionality in the system.



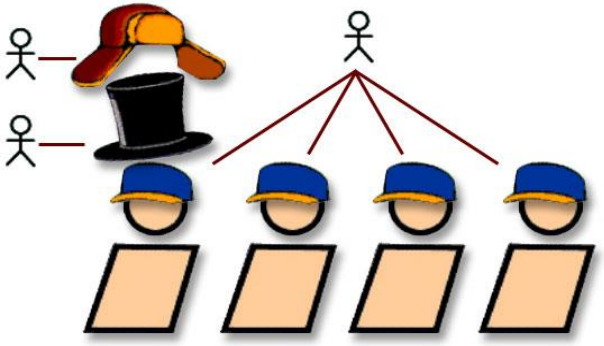
Actors and the system boundary

- Determine what the system boundary is
- Everything beyond the boundary that interacts with the system is an instance of an actor

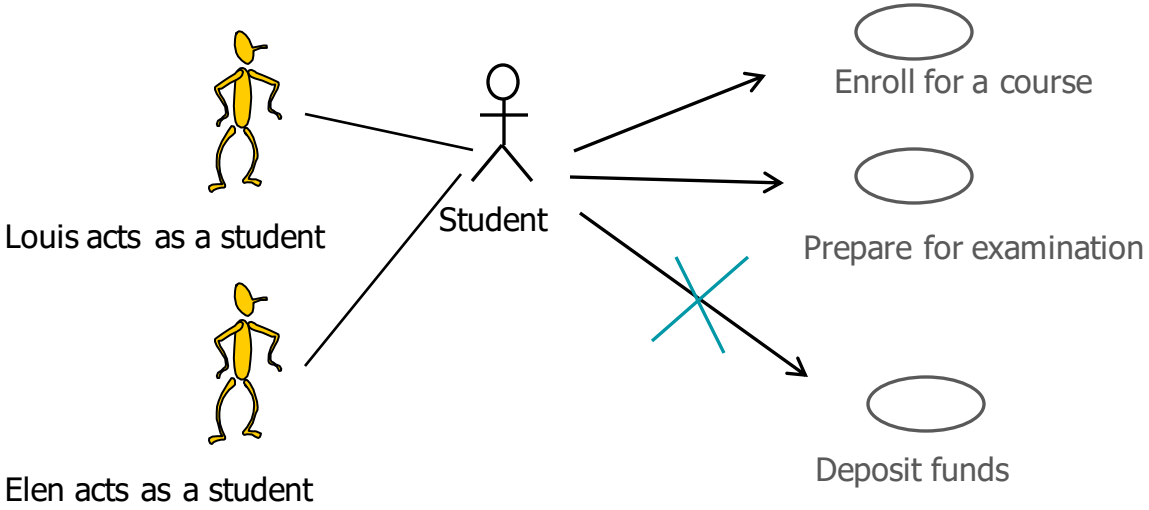


Actors and roles

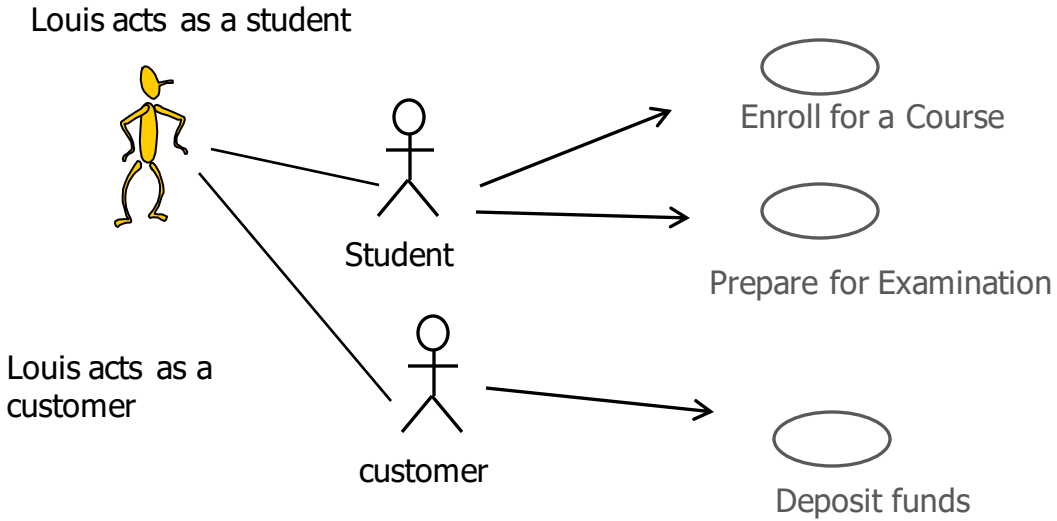
- An actor represents a role that a human, hardware device, or another system can plan in relation to the system.



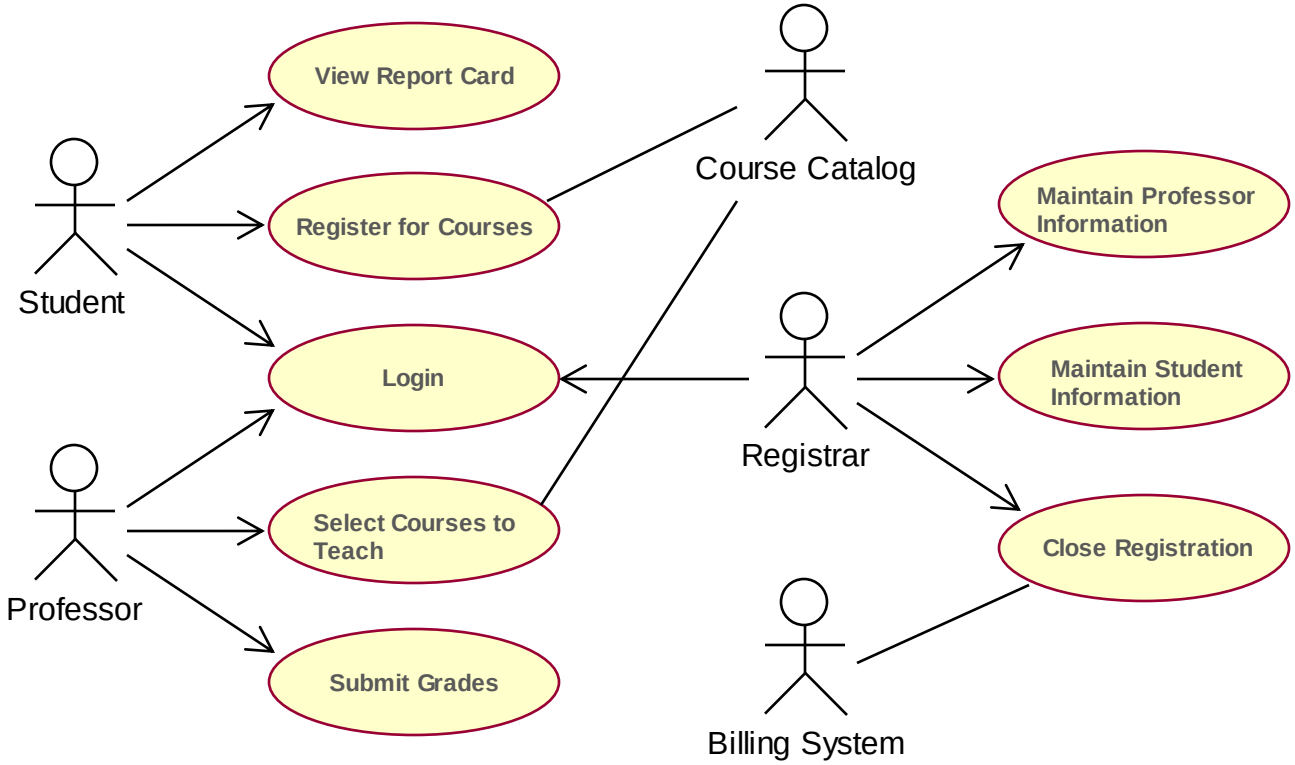
Actors


Louis acts as a student
Elen acts as a student
Student
Enroll for a course
Prepare for examination
Deposit funds

Actors

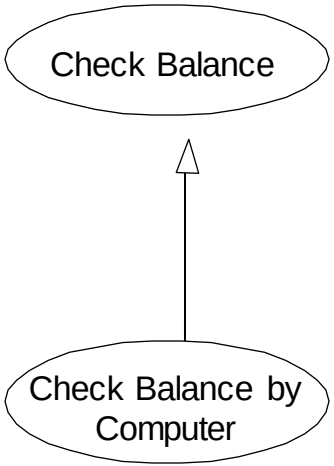

Louis acts as a student
Student
Enroll for a Course
Prepare for Examination
Louis acts as a customer
customer
Deposit funds

How Would You Read This Diagram?

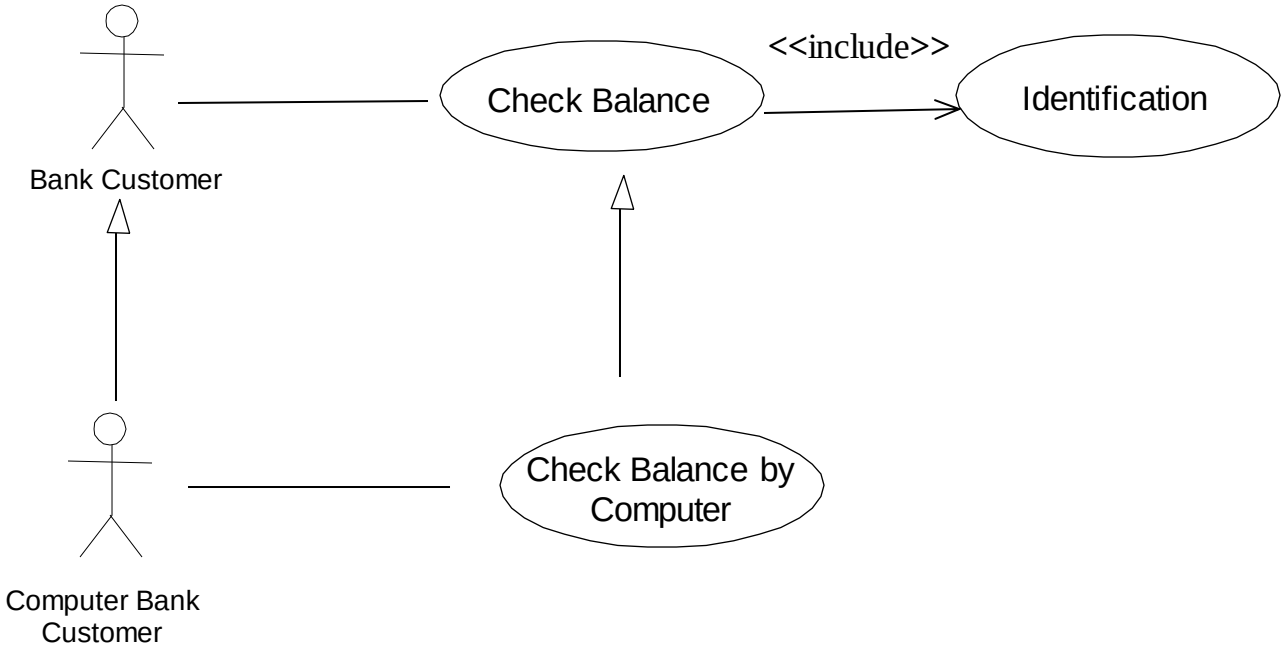


Modelling with Use Cases

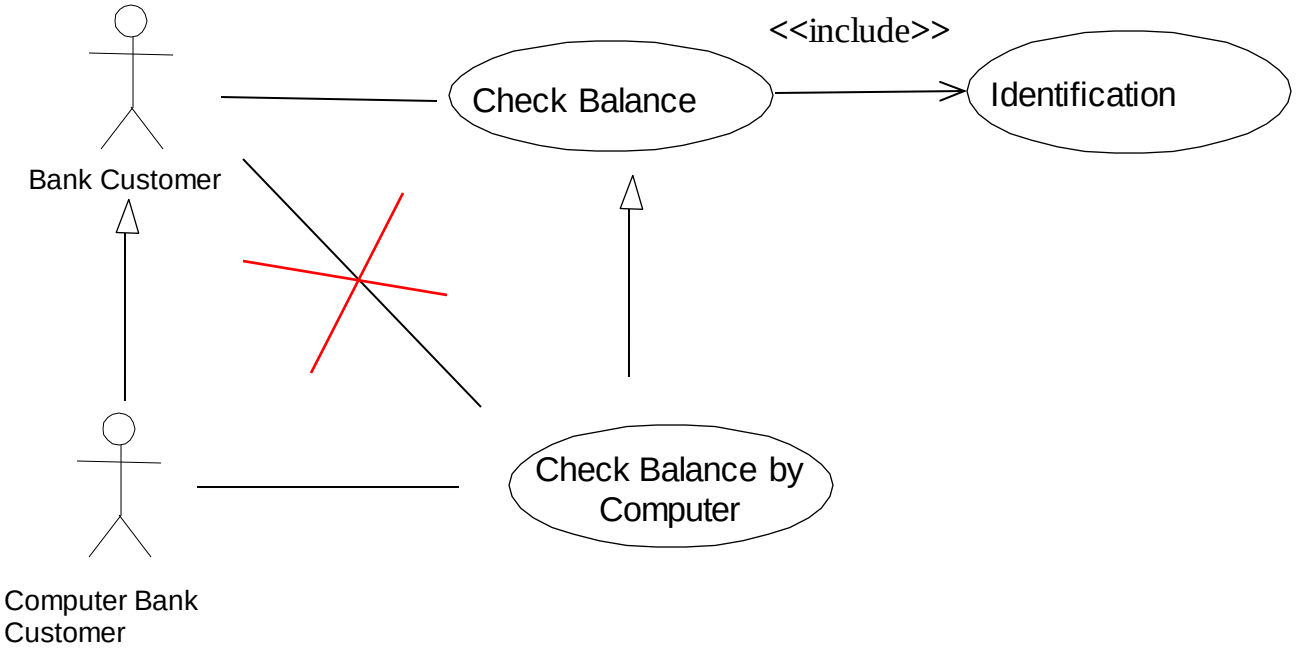
- Generalization between Use Cases means that the child is a more specific than the parent; the child inherits all attributes and associations of the parent, but may add new features



Structuring Use Case Diagrams

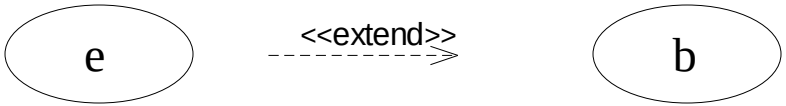


Use Case Diagram: Structuring



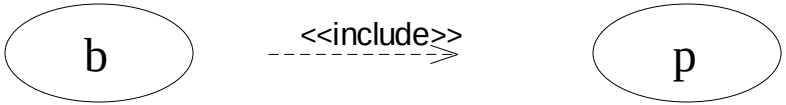
Use Cases

Specifies how the behaviour of the extension use cases `e` can be inserted into the behaviour of the base use case `b`.
`e` is optional.



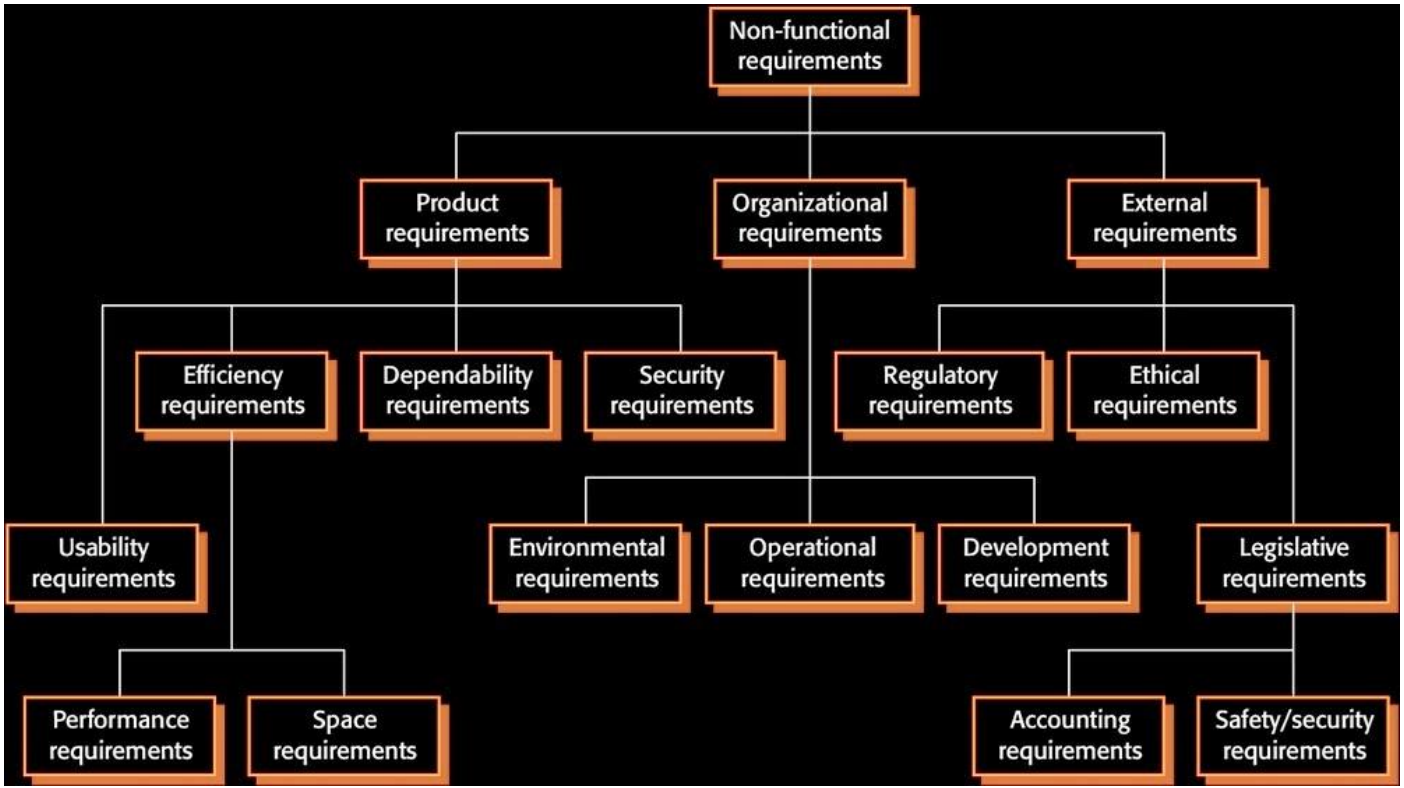
Extend ralationship

Specifies how the behaviour of the included Use Case `p` contributes to the behaviour of the base use case `b`.



Include relationship

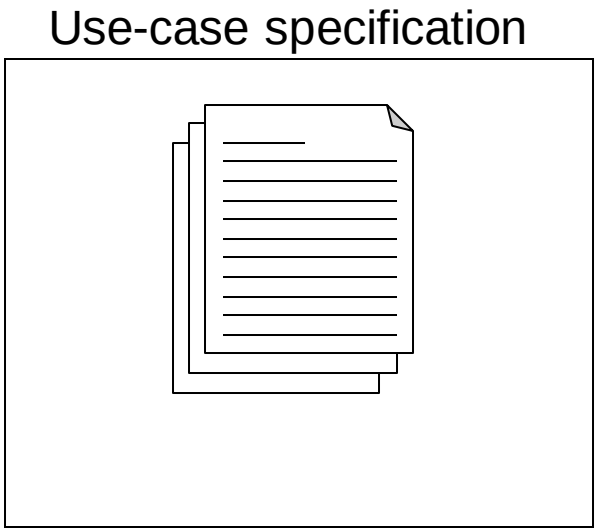
But also: Non-functional Requirements



3. Break down stories into individual steps / Refine requirements

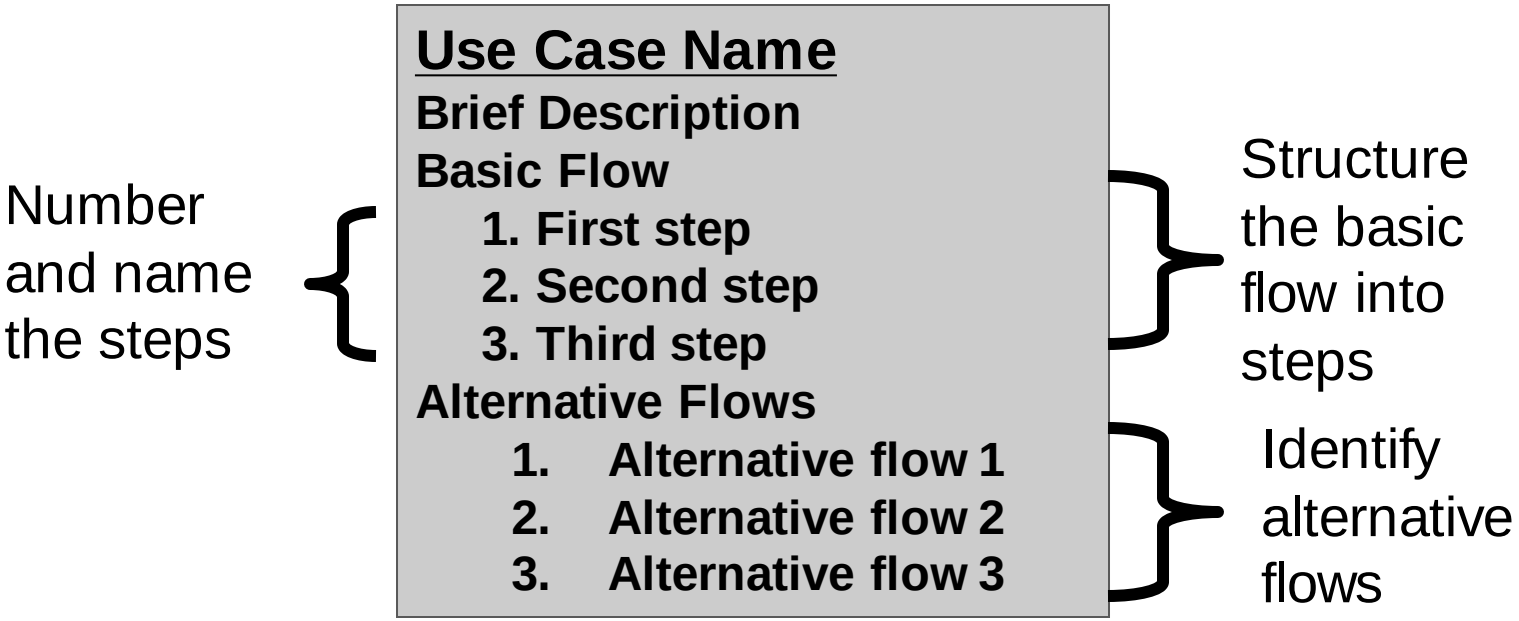
Use-case specification

- A requirements document that contains the text of a use case, including:
  - A description of the flow of events describing the interaction between actors and the system
  - Other information, such as:
    - Preconditions
    - Postconditions
    - Special requirements
    - Key scenarios
    - Subflows



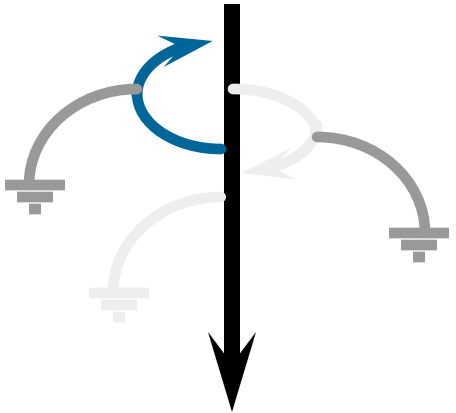
Outline each use case

- An outline captures use case steps in short sentences, organized sequentially



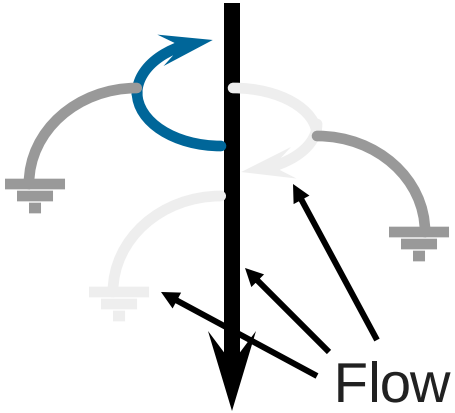
Flows of events (basic and alternative)

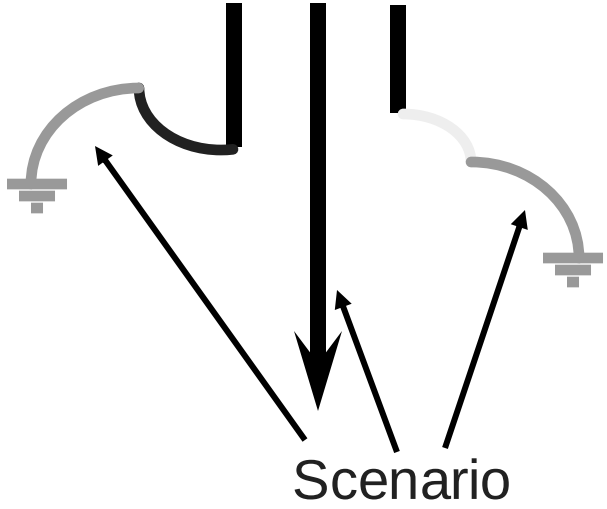
- A flow is a sequence of steps
- One basic flow
  - Successful scenario from start to finish
- Many alternative flows
  - Regular variants
  - Odd cases
  - Exceptional (error) flows



What is a use-case scenario?

- An instance of a use case
- An ordered set of actions from the start of a use case to one of its end points





Note: This diagram illustrates only some of the possible scenarios based on the flows.

Checkpoints for use cases

- ✓ Each use case is independent of the others
- ✓ No use cases have very similar behaviors or flows of events
- ✓ No part of the flow of events has already been modeled as another use case

4. Specify atomic requirements (e.g., for each step in user stories)

- ✓Not detailed in this course, e.g.:
  - ✓Structured language
  - ✓Formal methods

Quality Attributes of Requirements

- Consistency: Are there conflicts between requirements ?
- Completeness: Have all features been included ?
- Comprehendability: Can the requirement be understood ?
- Traceability: Is the origin of requirement clearly recorded ?
- Realism: Can the requirements be implemented given available resources and technology ?
- Verifiability: Can requirements be “ticked off” ?

Verifiability of Requirements

We should ensure that requirements are ***verifiable***
No point specifying something that can't be "ticked off"

For example, the following is an ***unverifiable*** "objective":
The system must be easy to use by waiting staff and should be organised so that user errors are minimised.

In comparison, the following is a ***testable*** requirement:
Waiting staff shall be able to use all system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.

Requirements Elicitation Techniques

- Interviews
- Observations
- Surveys
- Current documentation
- Similar products and solutions
- Co-design
- Prototyping
- …

Review

- What are models for?
- What is system behavior?
- What is an actor?
- A use case?
- What is a role?
- How do we know if our requirements are of good quality?

Requirements

Workshop 4

Ruzanna Chitchyan, Jon Bird, Pete Bennett
TAs: Alex Elwood, Alex Cockrean, Casper Wang

Today's Workshop

- Requirements case study (30mins)
- Discussion (10mins)
- Requirements for your game (60mins)

Running app – brief from SittingPlace Enc. CEO

The SittingPlace office group wants to encourage its employees to be more physically active. In an interview, the company CEO said:

"*The company will provide bonus points to those who use this jogging app, as it will reduce our costs from the employee sickness and stress. The app will allow the employees to jog to work instead of driving, so we will encourage jogging to and from work, if our employees want to.*

We also hear that our employees (as they live at various locations and distances away from the company) are worried that they will not have safe running path to take part in the program or that the distance is too long for them.

So, the app needs to help with this. Where the distance is too long, maybe they can take a ride on bus or tram and then run. That's all fine, as long as they run some distance, we are happy. Obviously the longer they run the healthier it is for

I myself have tried this idea out. I run 3 days a week. So, I have planned several run paths for good days (good with weather and time, I mean) I run the whole distance from home and back. But on wet days my driver picks me up in the mornings and I take bus to the park halfway home and run back only part of the distance. But it works. So, we want to have an app like that."

Requirements Case Study:

Working in your group:

1. Identify the stakeholders for this app (5 min)
2. Write 3 user stories for this app (5 min)
3. Select one User Story and write a Use Case specification (use case steps) for it (10 min)
4. Discuss with another peer group: Why you may wish to use (or not to use) use cases, user stories, or their combinations (15 min).
5. ***Homework**: draw a Use Case diagram for this app*

Note, for models you can use this online tool:

https://www.umletino.com/umlLetino

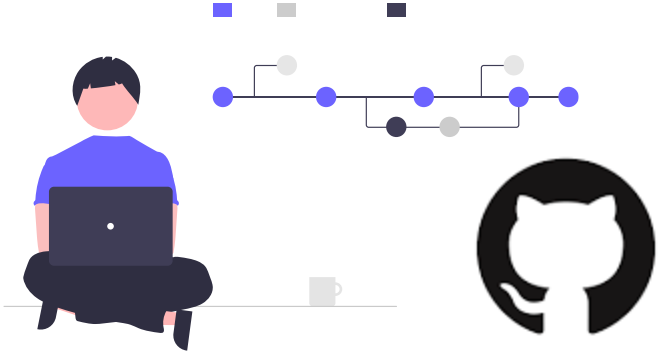
Requirements for your game

Working in your group:

1. Identify the **stakeholders** for your game (5 min)
2. Assign stakeholder roles within the team and write **user stories** for your game from these stakeholders' perspectives (**2 per team member**). Discuss the user stories (15 min)
3. Choose **one** user story and write a **use case specification** for it (10 min)
4. **Discuss with another team** what roles/user stories each team has, what can you learn from each other and what is the utility of user stories/use cases? (15 min)
5. Commit the **stakeholder list** and **Use Case specification** to your **git** repository under a Requirements section (5 min)

homework / groupwork

- Add a requirements section to your Github repo that includes:
  - Stakeholders
  - Add your user stories to the Kanban board
  - The worked through use case specification
  - A Use Case diagram for your game
  - A brief (up to 300 words) **Reflection** on what your team learned on user story/use case use



Summer Project Drop-in

- **Tomorrow!**
  Tuesday 13th February 1pm – 2pm,
  Ivy Gate G01

- Also, next week:
  Tuesday 20th February 1pm – 2pm,
  Ivy Gate G01

- Come along to discuss group formation, individual supervisor choice, find team-mates or if you have general questions.

Object Oriented Design

Lecture 4

Ruzanna Chitchyan, Jon Bird, Pete Bennett
TAs: Alex Elwood, Alex Cockrean, Casper Wang

Overview

- Why would we do OO design?
- Class Diagrams
  - Associations: Composition and Aggregation
  - Generalisation: Inheritance
  - Navigability
- Modelling behaviour
  - Communication diagrams
  - Sequence Diagrams

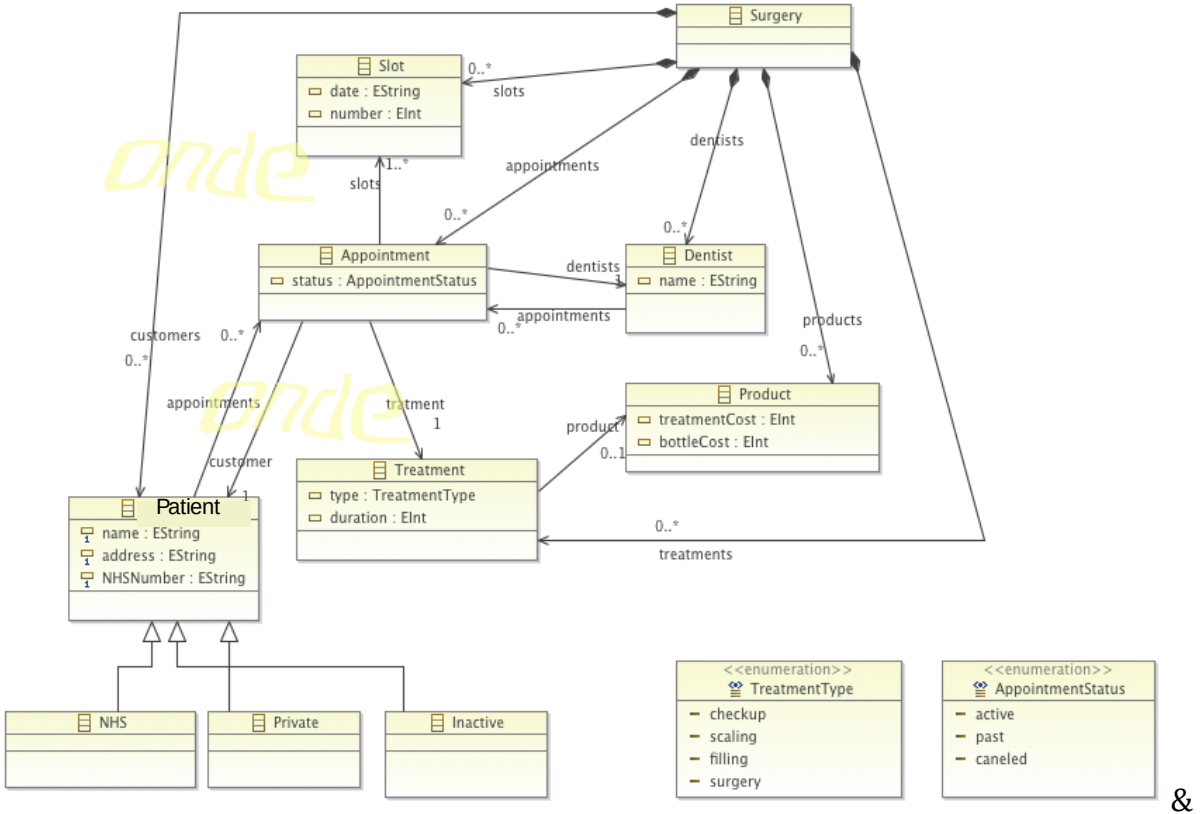
Why OO Design?

- Organise ideas
- Plan work
- Build understanding of the system structure and behavior
- Communicate with development team
- Help (future) maintenance team to understand

Class Diagrams

What Is a Class Diagram?

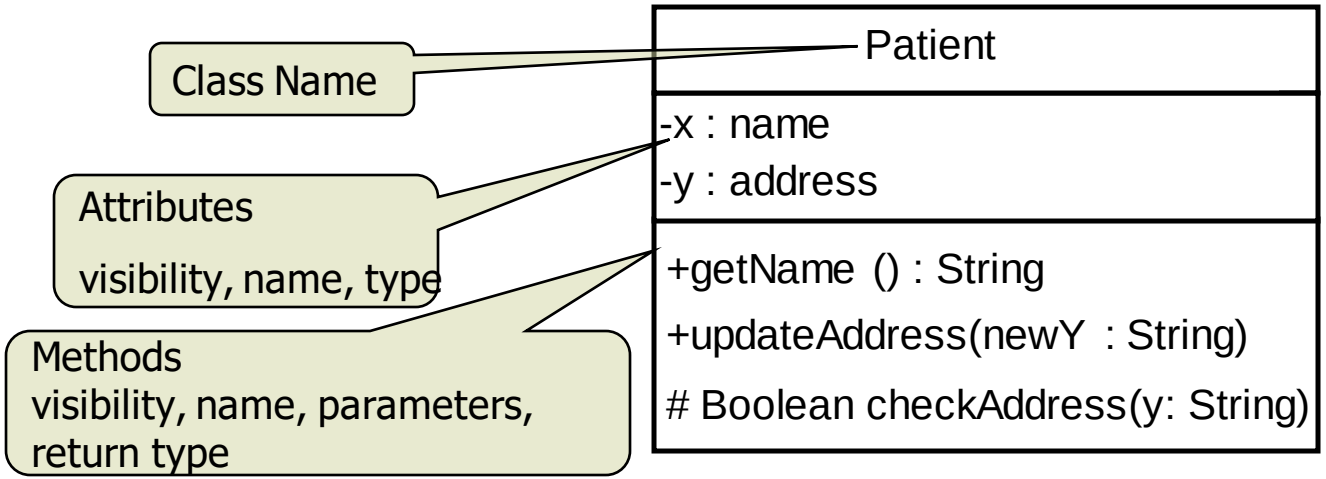
- Static view of a system



&

Class Diagrams

Class can be understood as a template for creating objects with own functionality



Visiblity:

+ public # protected - private

Notation for Attributes

[visibility] name [: type] [multiplicity] [=value] [{property}]

- visibility
  - other package classes
  - - private : available only within the class
  - + public: available for the world
  - # protected: available for subclasses and other package classes
  - ~ package: available only within the package
- [multiplicity], by default 1
- properties: readOnly, union, subsets<property-name>, redefined<property-name>, ordered, bag, seq, composite
- static attributes appear underlined

Notation for Operations

[visibility] name ([parameter-list]) : [{property}]

- visibility
- method name
- formal parameter list, separated by commas:
  - direction name : type [multiplicity] = value [{property}]
  - static operations are underlined
- Examples:
  - display()
  - - hide()
  - + toString() : String
  - createWindow (location: Coordinates, container: Container): Window

How do we find Classes: Grammatical Parse

Classes

- Identify nouns from existing text
- Narrow down to remove
    - Duplicates and variations (e.g., synonyms)
    - Irrelevant
    - Out of scope

Grammatical Parse: Dental Surgery Example

You are responsible for development of a software system for keeping track of the appointments and services of a dental surgery. This business employs several dentists, provides treatments to NHS and non NHS patients, and allows for patients to buy products (e.g., toothbrush, paste, etc.) when they pay for the received services (such as periodontal therapy with plague removal and scaling, and dental surgery).

Grammatical Parse: Dental Surgery Example

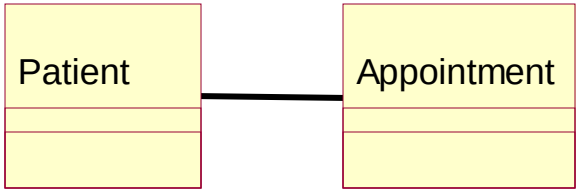
You are responsible for development of a software system for keeping track of the appointments and services of a dental surgery. This business employs several dentists, provides treatments to NHS and non NHS patients, and allows for patients to buy products (e.g., toothbrush, paste, etc.) when they pay for the received services (such as periodontal therapy with plague removal and scaling, and dental surgery).

Structural Relationships in Class Diagrams

What Is an Association?

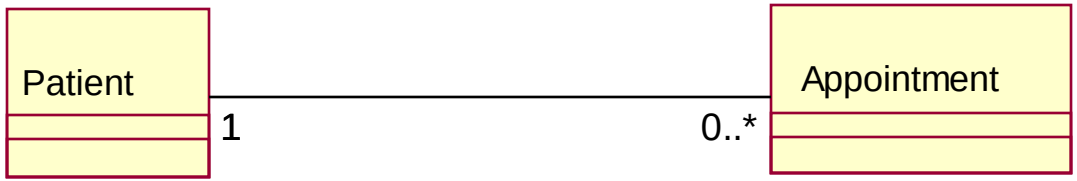
- The semantic relationship between two or more classifiers that specifies connections among their instances.

- A structural relationship specifying that objects of one thing are connected to objects of another thing.



What Is Multiplicity?

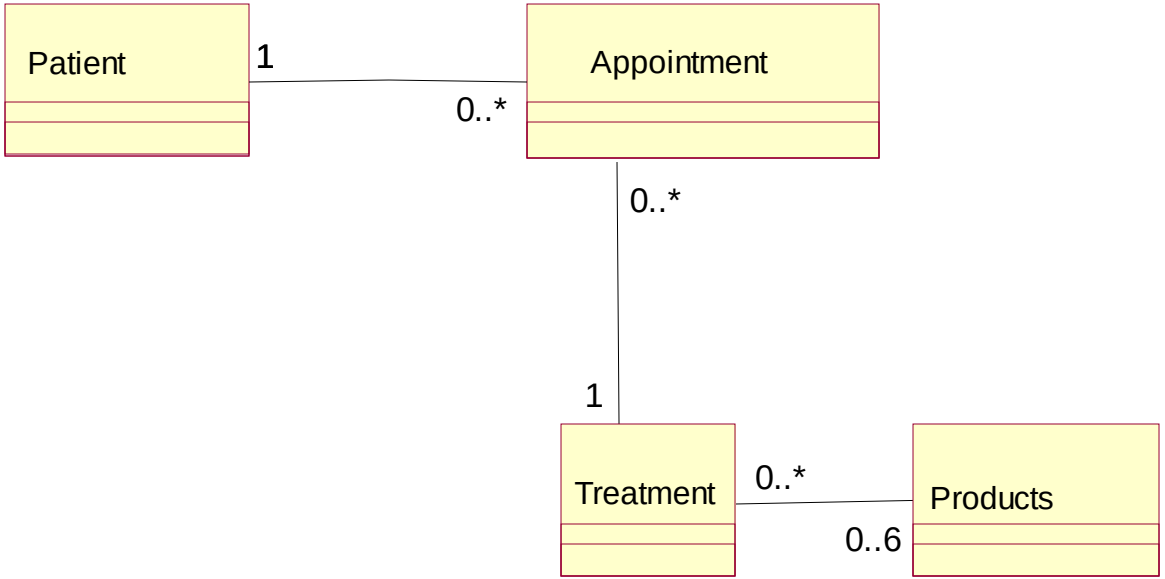
- Multiplicity is the number of instances one class relates to ONE instance of another class.
- For each association, there are two multiplicity decisions to make, one for each end of the association.
  - For each instance of Patient, many or no Appointments can be made.
  - For each instance of Appointment, there will be one Patient to see.



Multiplicity Indicators

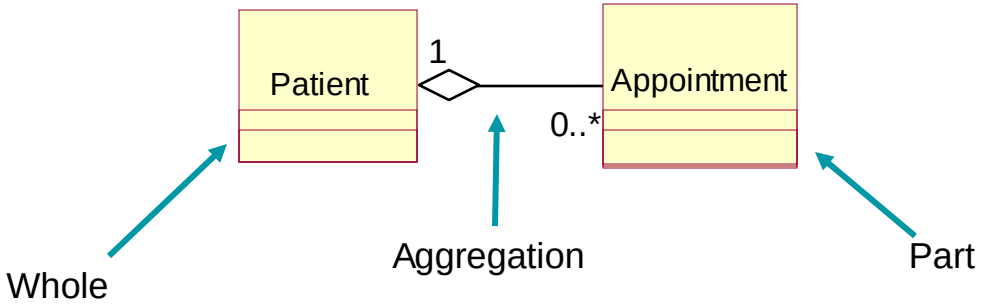
Unspecified	
Exactly One	1
Zero or More	0..*
Zero or More	*
One or More	1..*
Zero or One (optional value)	0..1
Specified Range	2..4
Multiple, Disjoint Ranges	2, 4..6

Example: Multiplicity



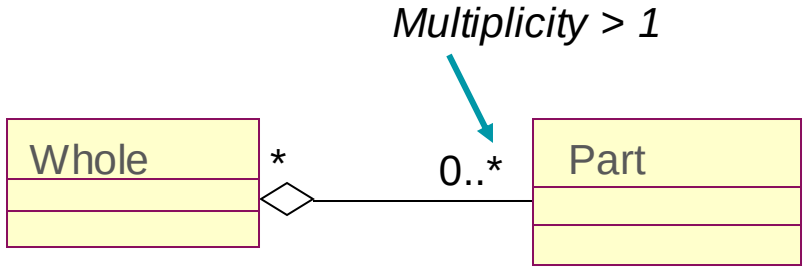
What Is an Aggregation?

- A special form of association that models a whole-part relationship between the aggregate (the whole) and its parts.
  - An aggregation is an “is a part-of” relationship.
- Multiplicity is represented like other associations.

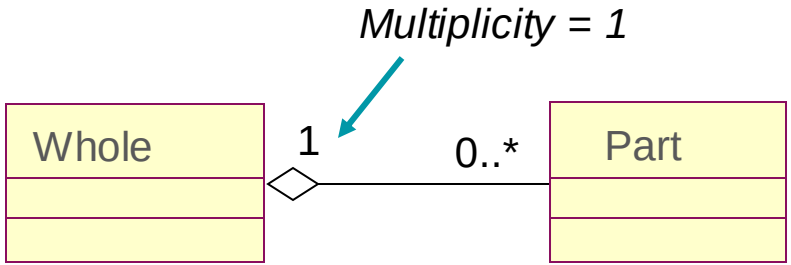


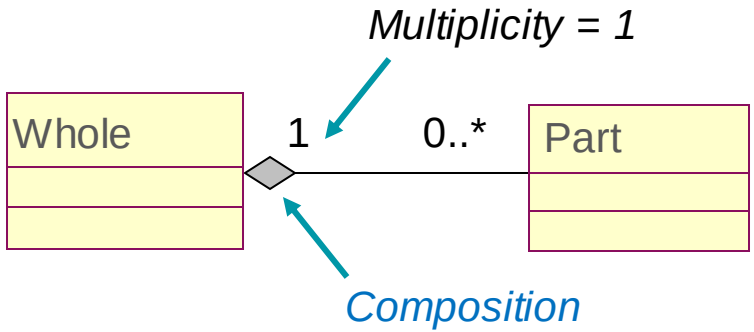
Aggregation: Shared vs. Non-shared

- Shared Aggregation



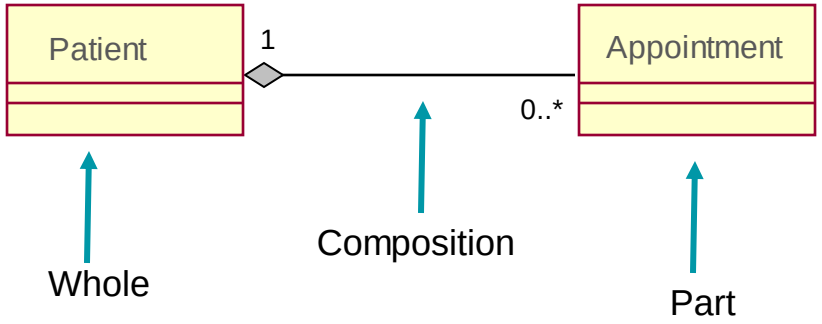
- Non-shared Aggregation





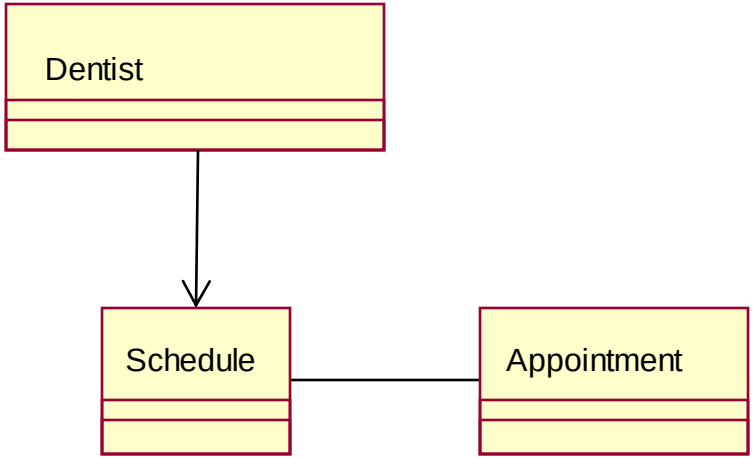
What Is Composition?

- A form of aggregation with strong ownership and coincident lifetimes
  - The parts cannot survive the whole/aggregate



What Is Navigability?

- Indicates that it is possible to navigate from an associating class to the target class using the association



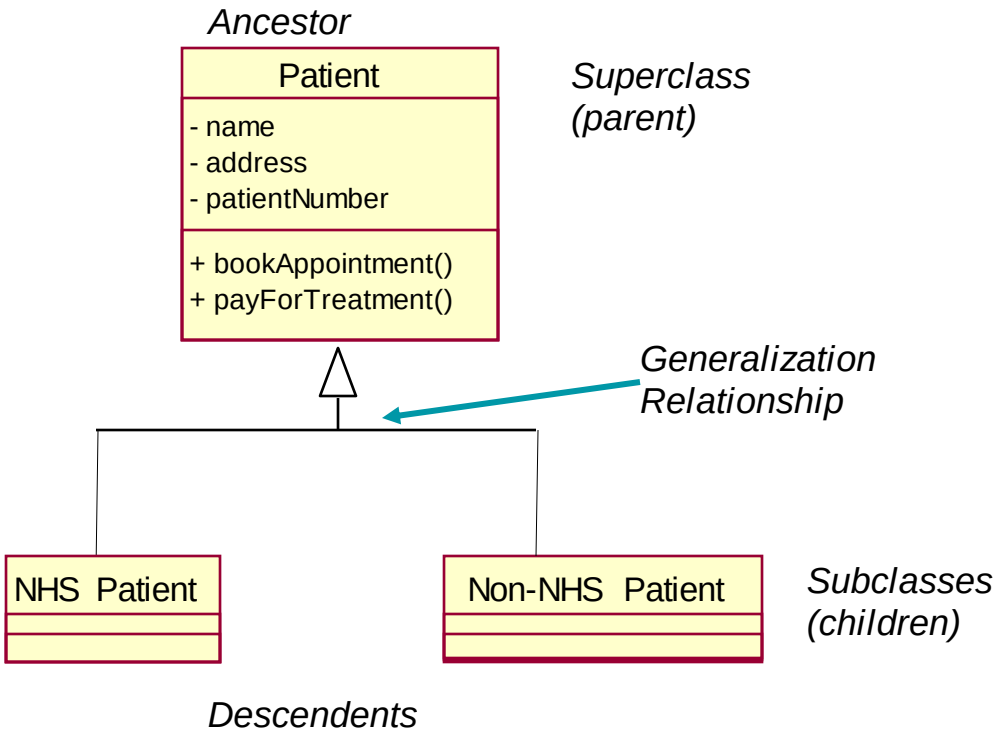
What Is Generalization?

- A relationship among classes where one class shares the *properties and/or behavior* of one or more classes.
- Defines a hierarchy of abstractions where a subclass inherits from one or more superclasses.

- Is an "**is a kind of**" relationship.

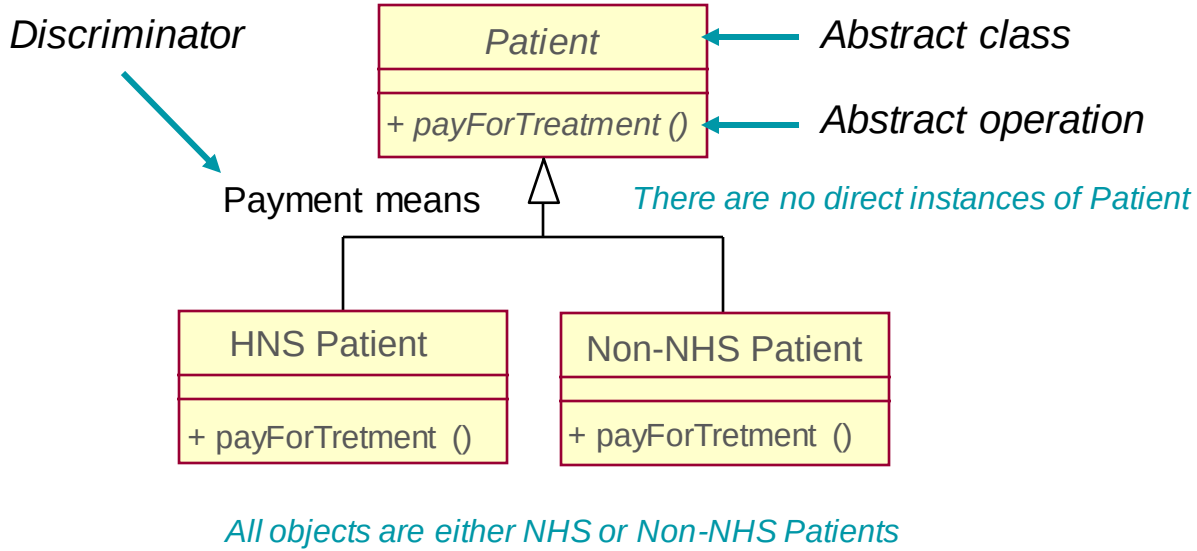
Example: Inheritance

- One class inherits from another
- Follows the “is a” style of programming
- Class substitutability



Abstract and Concrete Classes

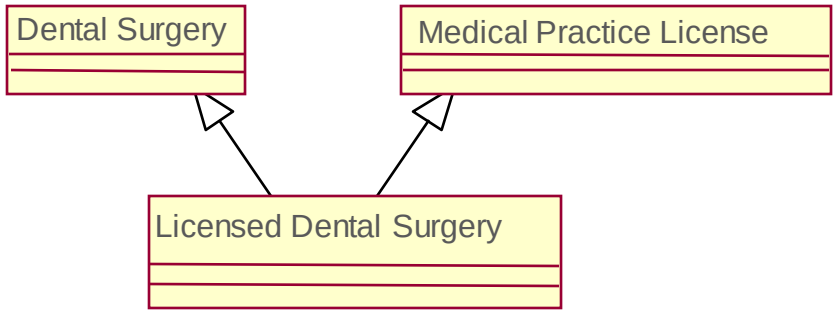
- Abstract classes cannot have any objects
- Concrete classes can have objects



Generalization vs. Aggregation

- Generalization and aggregation are often confused
  - Generalization represents an “is a” or “kind-of” relationship
  - Aggregation represents a “part-of” relationship

Is this correct?



Behaviour Modelling

Objects Need to Collaborate

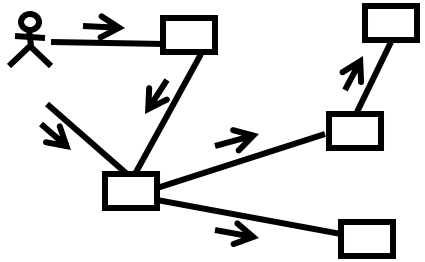
- Objects are useless unless they can collaborate to solve a problem.
  - Each object is responsible for its own behavior and status.
  - No one object can carry out every responsibility on its own.

- How do objects interact with each other?
  - They interact through messages.
  - Message shows how one object asks another object to perform some activity.

Communication Diagrams

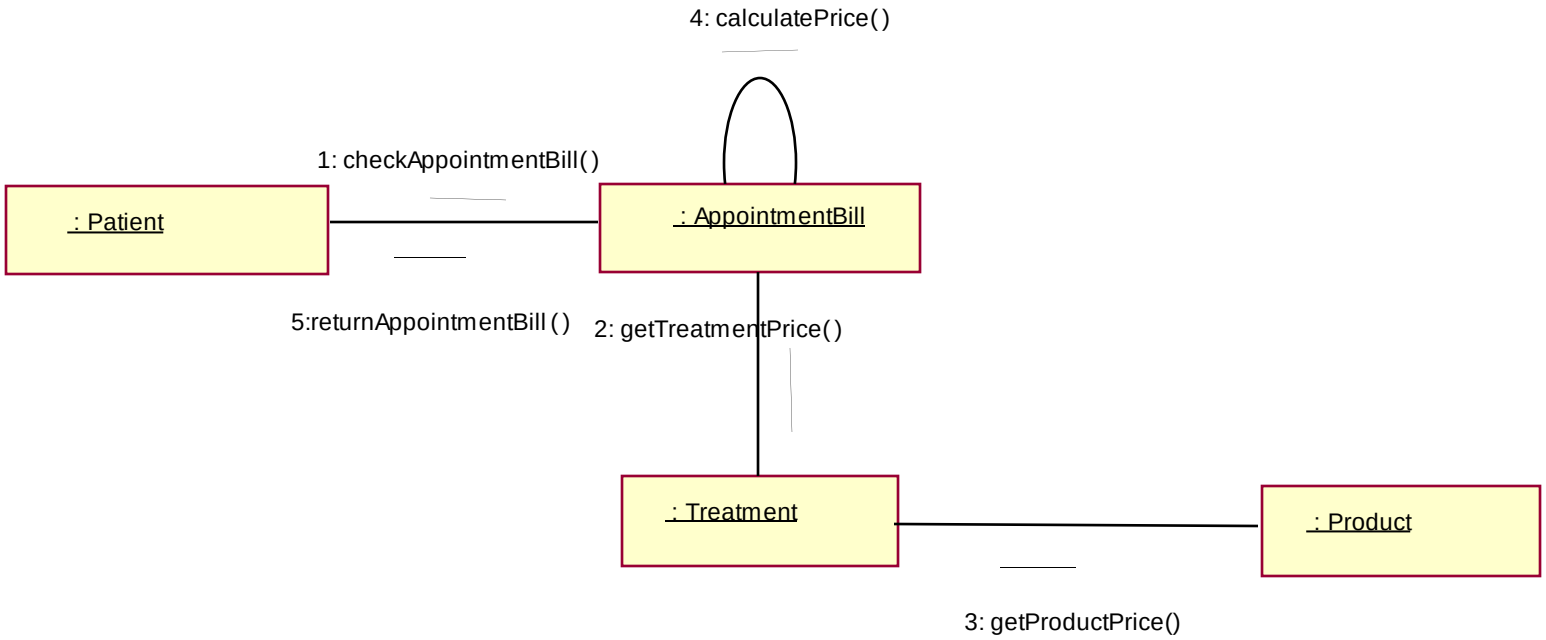
What Is a Communication Diagram?

- A communication diagram emphasizes the organisation of the objects that participate in an interaction.
- The communication diagram shows:
  - The objects participating in the interaction.
  - Links between the objects.
  - Messages passed between the objects.



Communication Diagrams

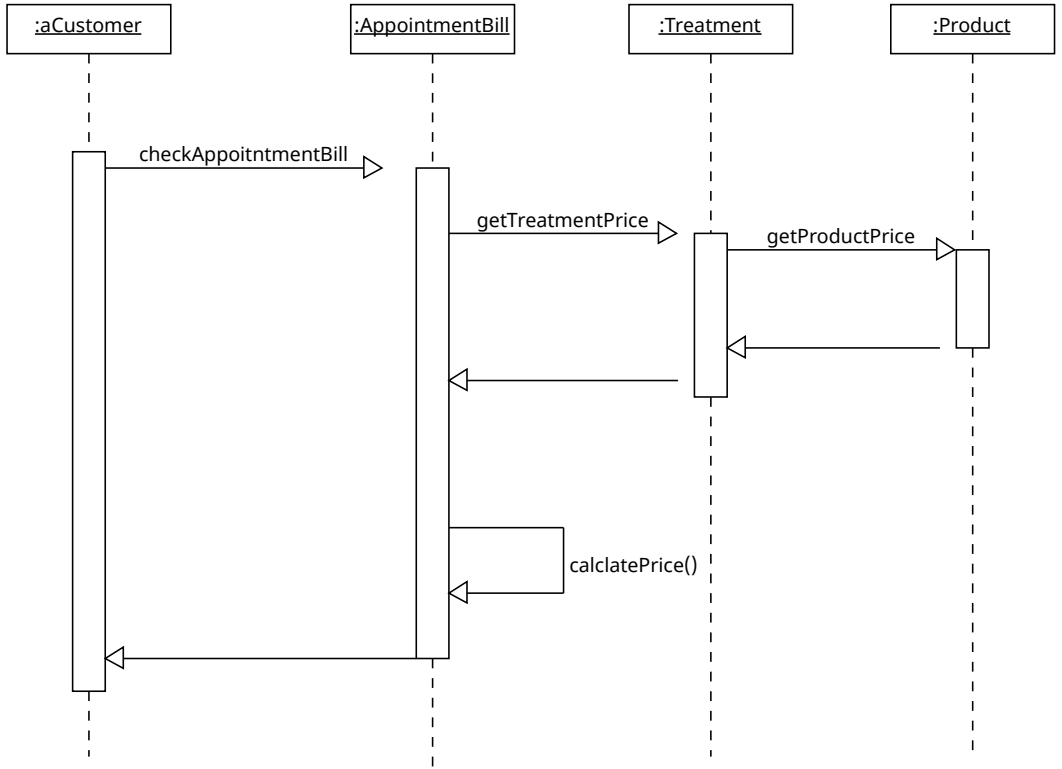
Example: Communication Diagram



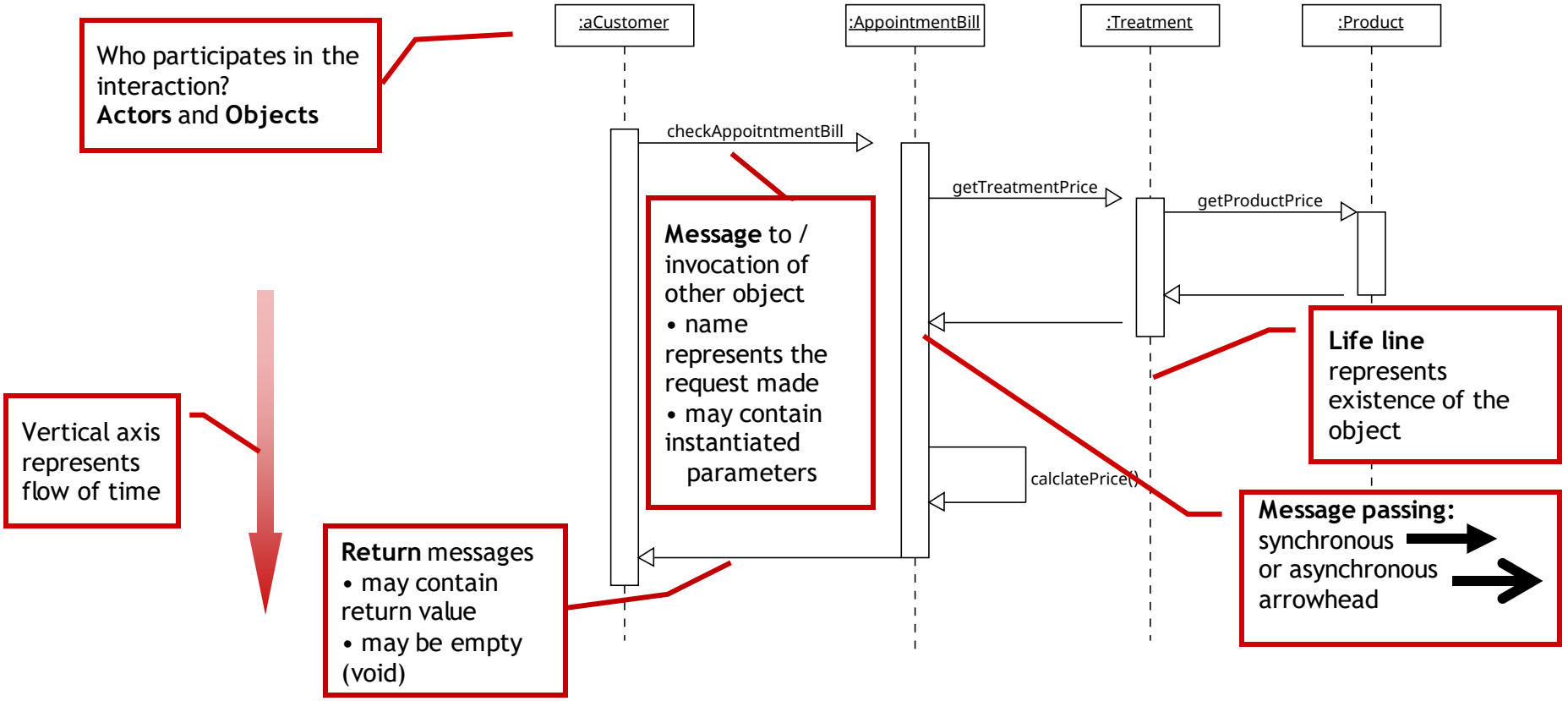
Sequence Diagrams

Sequence Diagrams: Basic Elements

- A set of participants arranged in time sequence
- Good for real-time specifications and complex scenarios



Sequence Diagrams: Basic Elements



Method for Analysis Sequence Diagrams

- for each scenario (high-level sequence diagram)
  - decompose to show what happens to objects inside the system

    ➔ objects and messages
  - Which tasks (operation) does the object perform?

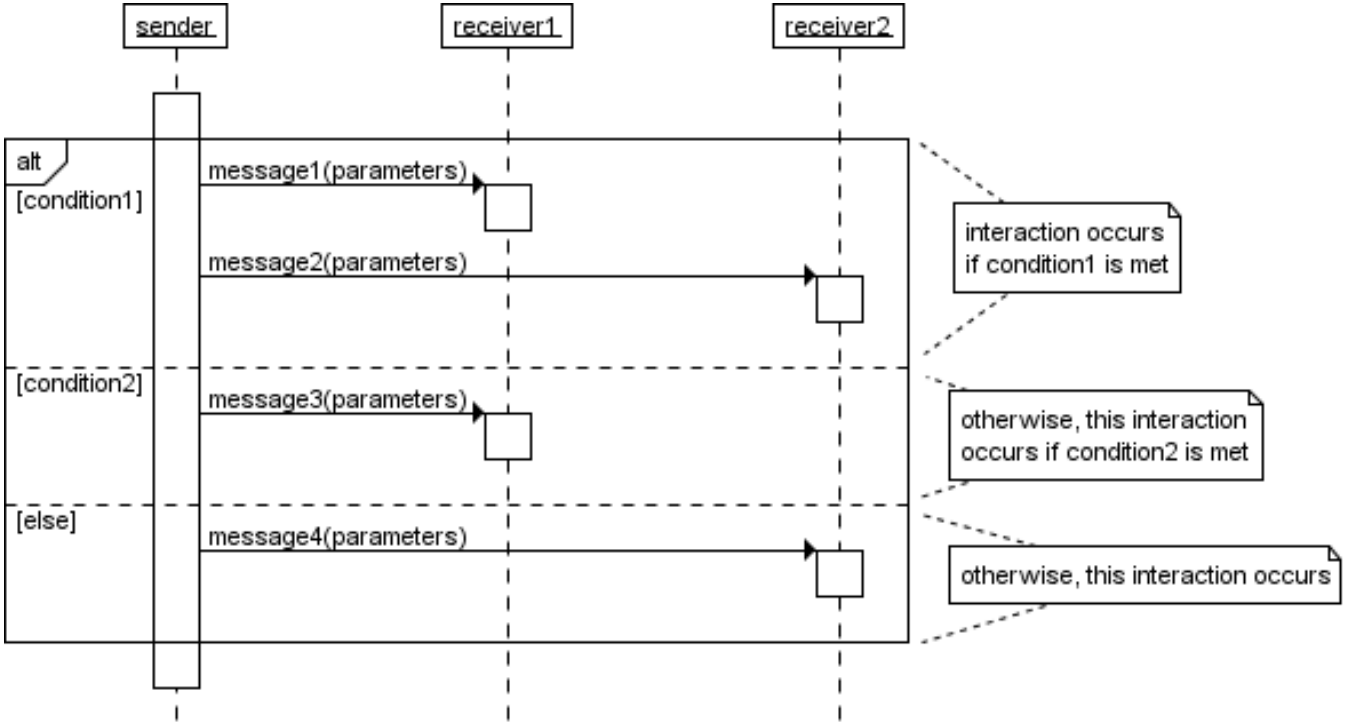
    ➔ label of message arrow
  - Who is to trigger the next step?

    ➔ return message or pass on control flow

Sequence Diagrams

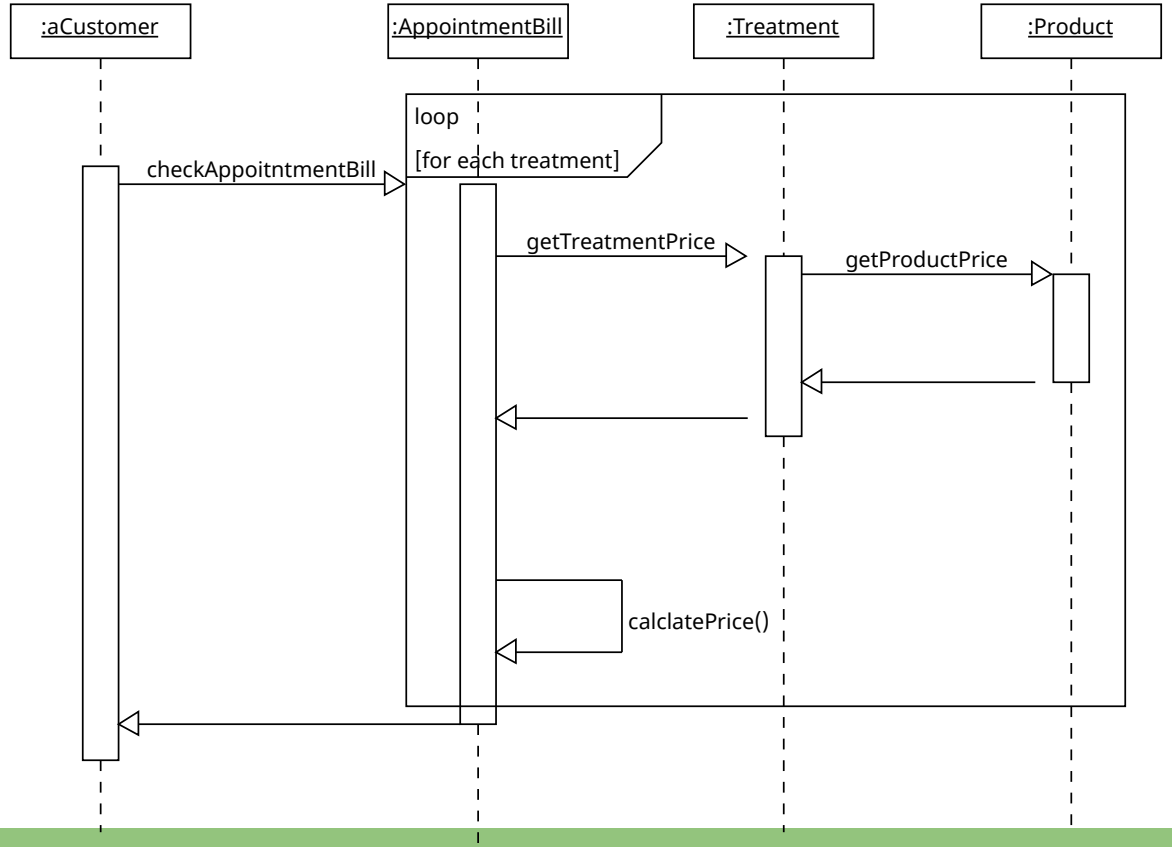
- Sequence Diagrams can model simple sequential flow, branching, iteration, recursion and concurrency
- They may specify different scenarios/runs
    - Primary
    - Variant
    - Exceptions

Interaction frames: alt



https://web.archive.org/web/20231018070441/http://www.tracemodeler.com/articles/a_quick_introduction_to_uml_sequence_diagrams/

Interaction frames: loop



Comparison: Communication and Sequence Diagrams

Sequence and Communication Diagram Similarities

- Semantically equivalent
    - Can convert one diagram to the other without losing any information

- Model the dynamic aspects of a system
- Model a use-case scenario

Sequence and Communication Diagram Differences

Sequence diagrams	Communication diagrams
▪ Show the explicit sequence of messages ▪ Show execution occurrence ▪ Better for visualizing overall flow ▪ Better for real-time specifications and for complex scenarios	▪ Show relationships in addition to interactions ▪ Better for visualizing patterns of communication ▪ Better for visualizing all of the effects on a given object ▪ Easier to use for brainstorming sessions

Review

- What does a class diagram represent?
- Define association, aggregation, and generalization.
- How do you find associations?
- What information does multiplicity provide?

- What is the main purpose of a SD?
- What are the main concepts in a SD?
- What are the communication diagrams?
- What is the difference between SD and communication diagrams?

Object Orientated Design

Workshop 5

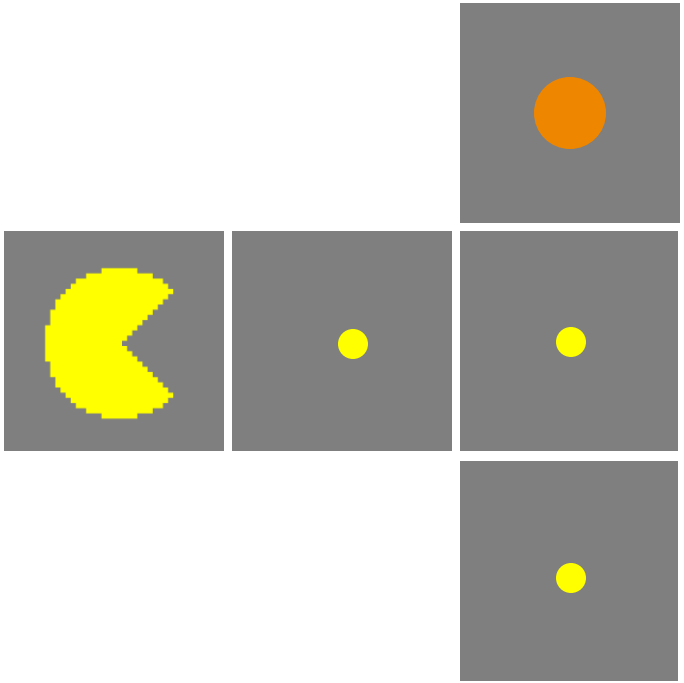
Ruzanna Chitchyan, Jon Bird, Pete Bennett
TAs: Alex Elwood, Alex Cockrean, Casper Wang

Today's Workshop

- Pac-man case study (45mins)
- Then either:
  - Classes challenges in Processing (45mins)
  - Draw class diagram for your game (45mins)
- Homework. Draw up a class diagram for your game, add it to your repo and begin basic implementation.

(Simplified) Pac-man Game

- The game is played on a gridded **board** which consists of **fields**. There are various **figures** that can be placed in fields. These include **pacman** as well as **marbles** and **pills**.
- Marbles and pills are **edible** by pac-man.
- We need to keep a record of eaten figures.
- The game ends when all the marbles have been eaten.



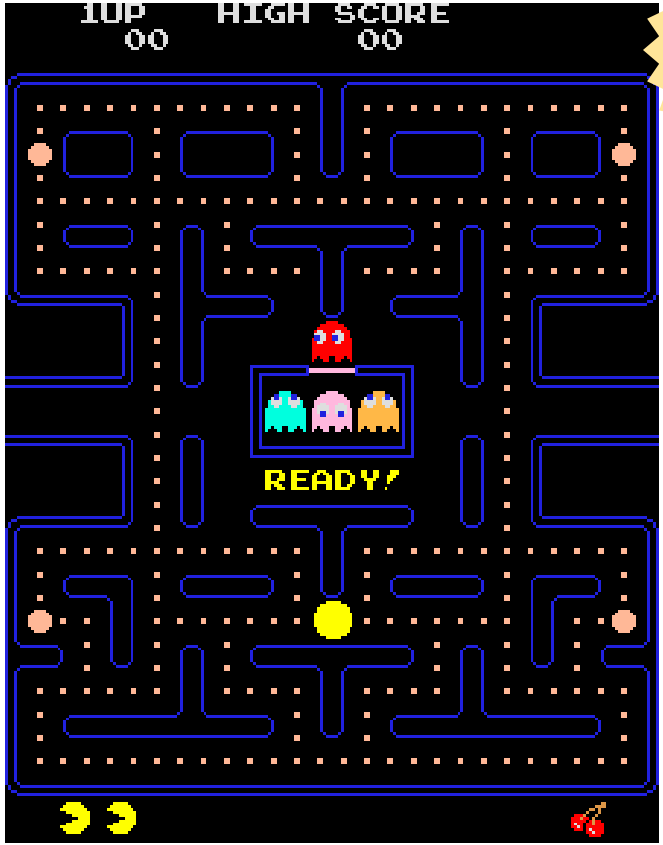
Class Diagram Task

15 mins

Sketch a class diagram of simplified pacman, making sure to note:

- Classes
- Associations
- Inheritance
- Cardinalities

Feel free to note attributes and methods, if you want.

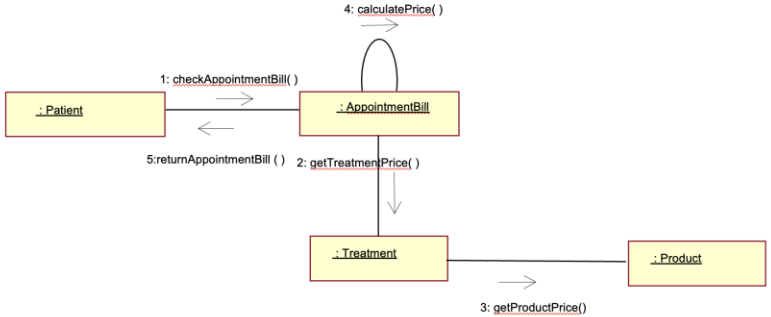


https://en.wikipedia.org/wiki/Pac-Man

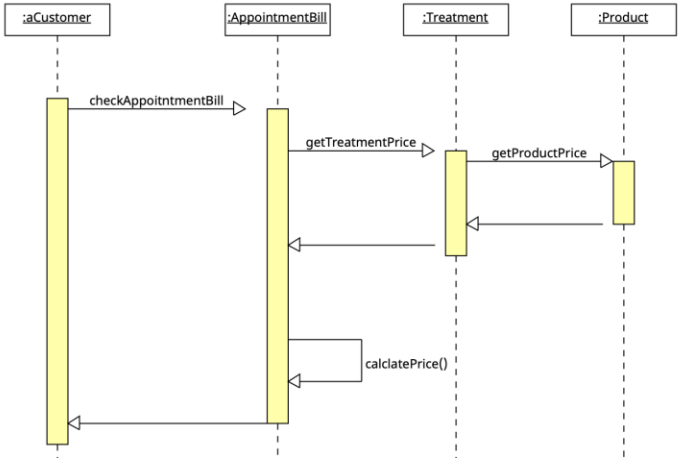
Communication/Sequence Task

15 mins

Sketch a **communication** or a **sequence** diagram for simplified pac-man



communication diagram

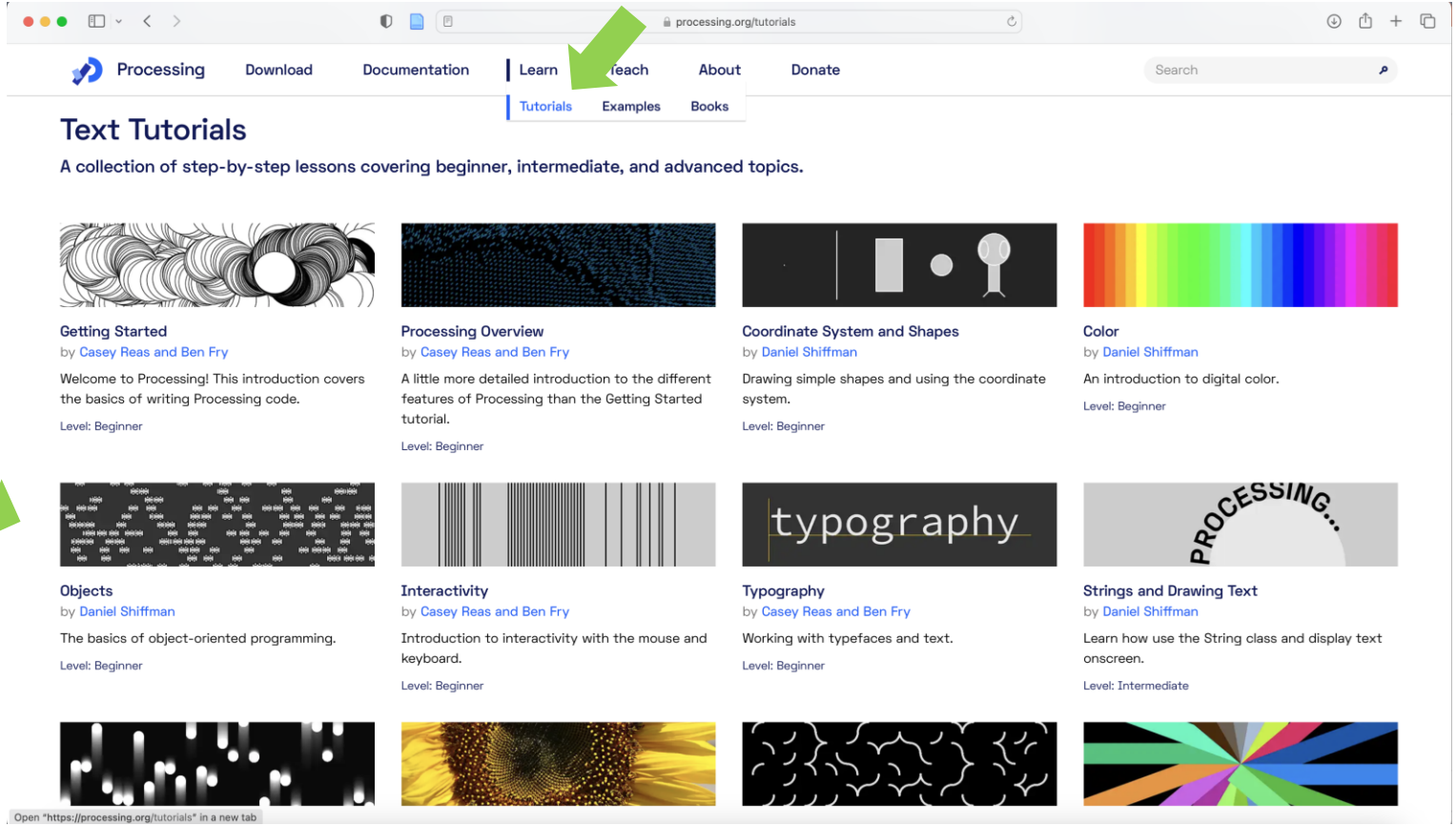


sequence diagram

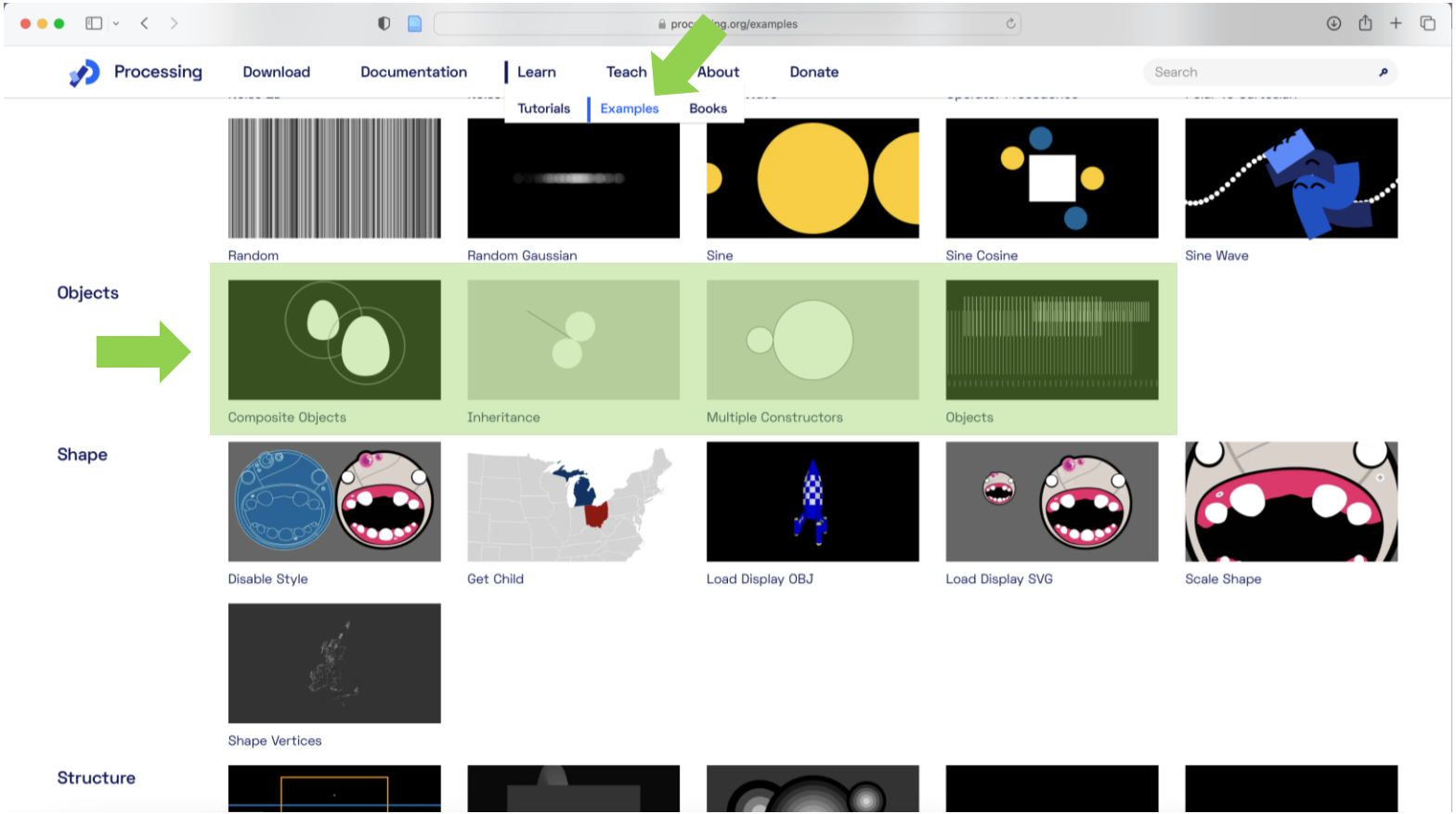
45 mins

Classes in Processing

Optional section if you would like to play around with using classes in Processing with TA support. If you do choose to do this, then please complete the class diagram for your game as homework.



Objects Tutorial - https://processing.org/tutorials/objects



Objects Examples - https://processing.org/examples

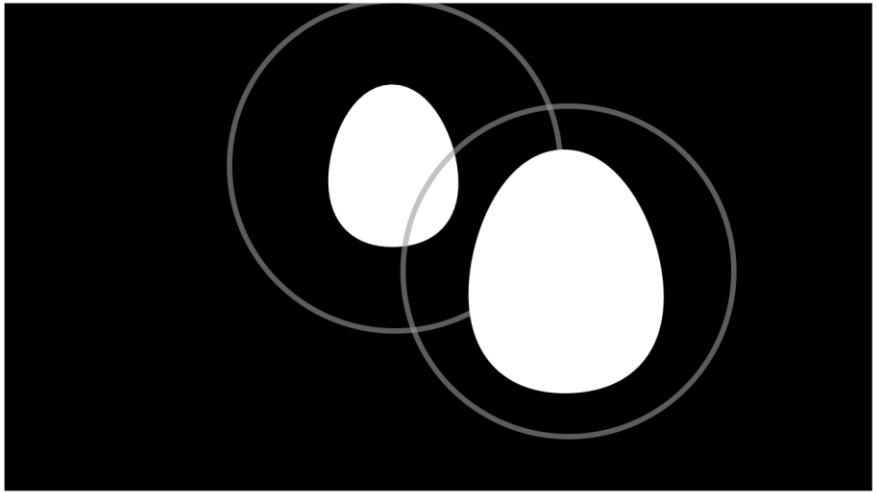
Composite Objects

Warming up:

- Draw a face on the egg using **ellipse**(), **rect**(), **line**(), **arc**() etc. Tip: place these in the Egg Class ‘display’ function, consider where to add them to the function’s sequence of drawing. Don’t forgot colour with **fill**() and **stroke**()

Challenges:

- Create a new class and add it to the composite EggRing class. Consider starting with something simple like placing the egg on a podium (or a hat?).



“An object can include other objects.”

https://processing.org/examples/compositeobjects.html

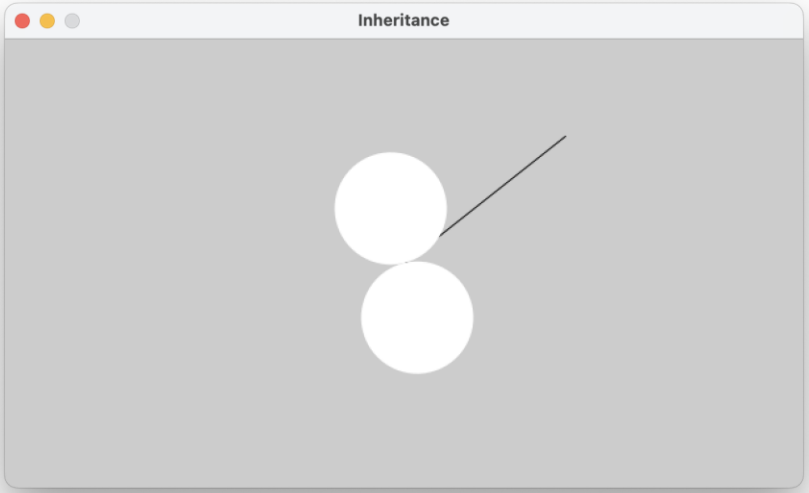
Inheritance

Warming up:

- Increase the speed of both the spin arm and spin spots (from the superclass)
- Update the Spin superclass update method so that the spinning gradually slows down

Challenges:

- Create a new subclass that draws a stationary rectangle. Use the angle variable of the superclass to change the colour or width of the rectangle
- Create a new superclass ‘Bounce’ that enables bouncing rather than spinning



“A class can be defined using another class as a foundation.”

https://processing.org/examples/inheritance.html

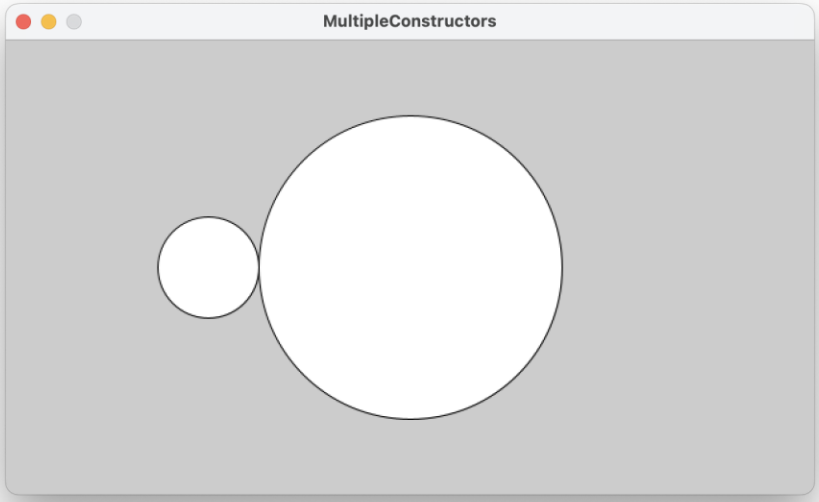
Multiple Constructors

Warming up:

- Comment out noLoop() so that the sketch loops. Add background(200) into draw loop to clear background each frame. Create a mousePressed() function to change the x, y and radius of sp2 with every mouse click (random or mouse pos)

Challenges:

- Add a fourth argument of your choice to the second constructor, perhaps a Boolean determining whether the circle is filled, or an opacity value. Update the code to make use of this fourth argument.



"A class can have multiple constructors that assign the fields in different ways."

https://processing.org/examples/multipleconstructors.html

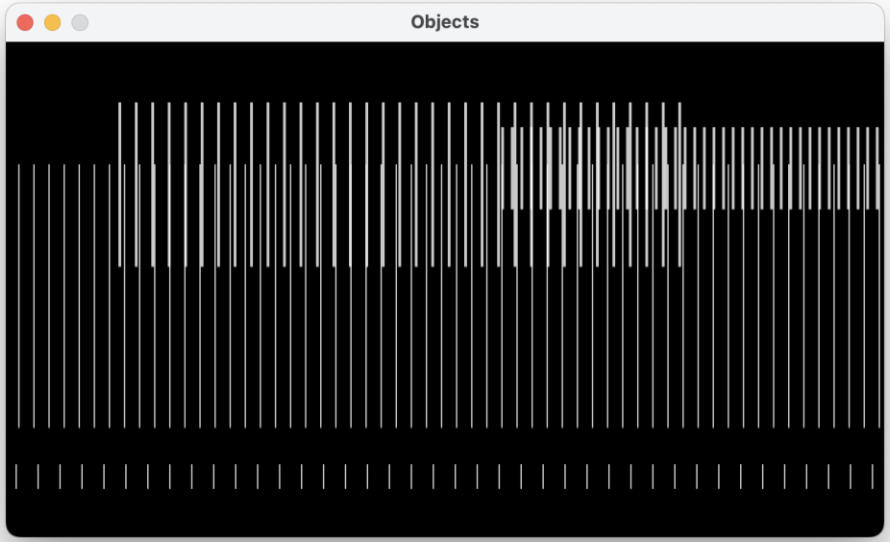
Objects

Warming up:

- Randomise the colour of each set of lines.
- Change the lines to circles.

Challenges:

- Rather than have 4 MRect variables, try creating an array of MRect.
- Create a new ‘update’ function within MRect that reduces the number of bars by one. Try calling this function on every mouse press.



“Move the cursor across the image to change the speed and positions of the geometry.”

https://processing.org/examples/objects.html

Classes for your game

45 mins

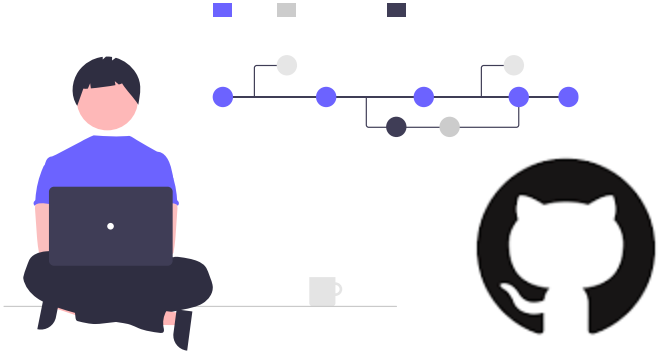
- Create a class diagram for your game making sure to note:
  - Classes
  - Associations
  - Inheritance
  - Cardinalities

Feel free to note attributes and methods, if you want.

- All teams need to do this before next week, please complete as homework and upload to your Github repo.

homework / groupwork

- Finish working through the examples in your team
- Your team should now have one game idea.
- Draw up a class diagram for your game, add it to your repo
- Begin basic implementation of your classes (whilst keeping a Minimum Viable Product in mind)



Summer Project Drop-in

- **Tomorrow!**
  Tuesday 20th February 1pm – 2pm,
  Ivy Gate G01

- Come along to discuss group formation, individual supervisor choice, find team-mates or if you have general questions.

- Please confirm supervisor or submit group choices by **end of reading week**.

Agile Software Development

Dr Jon Bird
jon.bird@bristol.ac.uk

Thanks to Dr Simon Lock who developed many of these slides for an earlier version of this unit.

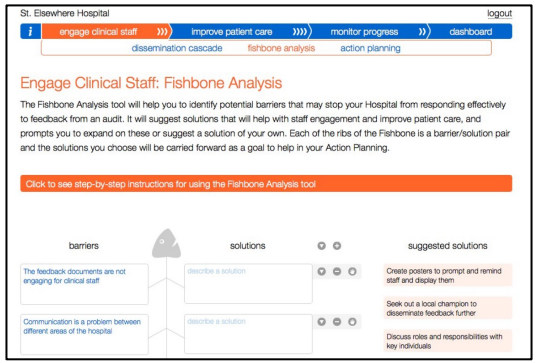
Images are royalty free from www.pexels.com

Today’s Lecture

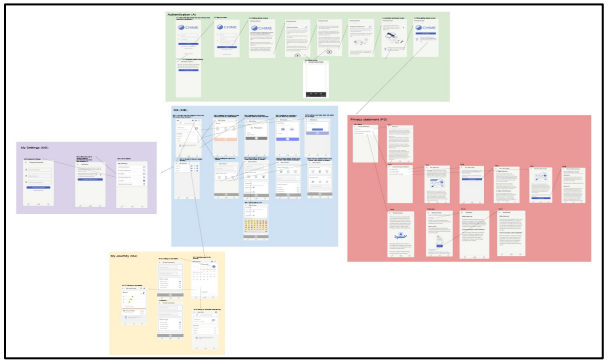
- A digital health project developed using the waterfall approach, which was the focus of last week
- Waterfall versus agile approaches to software development
- Agile software development, including: extreme programming (which includes pair programming); test-driven development; scrum; and Kanban
- A digital health project developed using an agile approach
- Recommended reading

Digital health projects

- My research involves developing digital technologies to address health issues
- What software development process do I employ in my research?



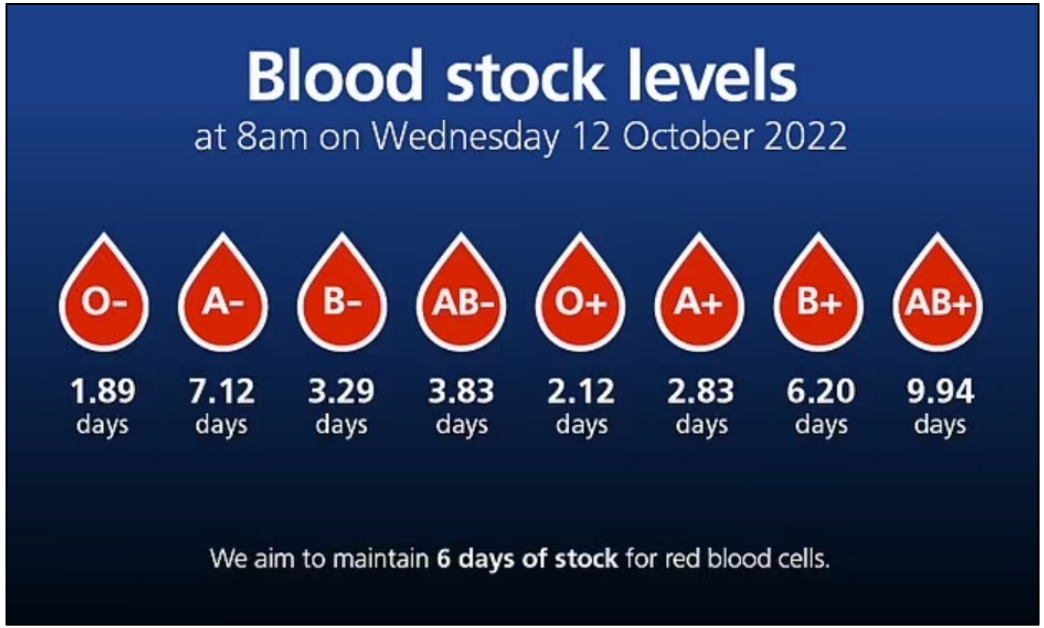
Affinitie



CHIME

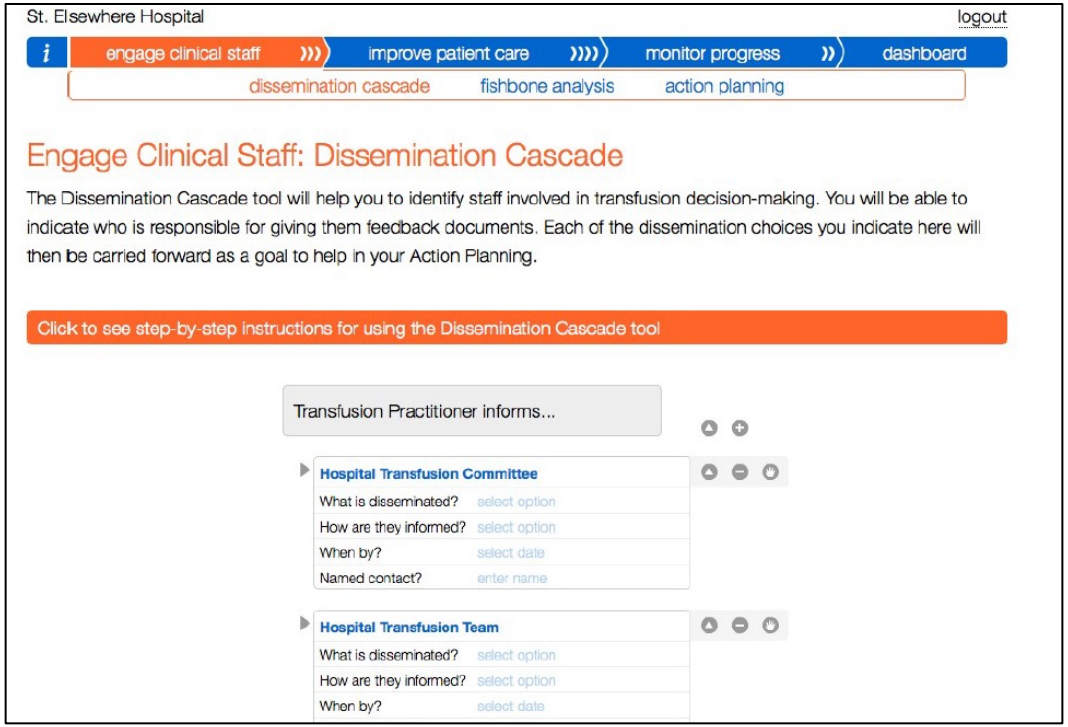
Affinitie - motivation

- There are 3 million blood transfusions annually in the UK
- Around 20% are unnecessary
- Blood components are a scarce resource
- There are health risks associated with blood transfusions
- How can we reduce unnecessary blood transfusions?



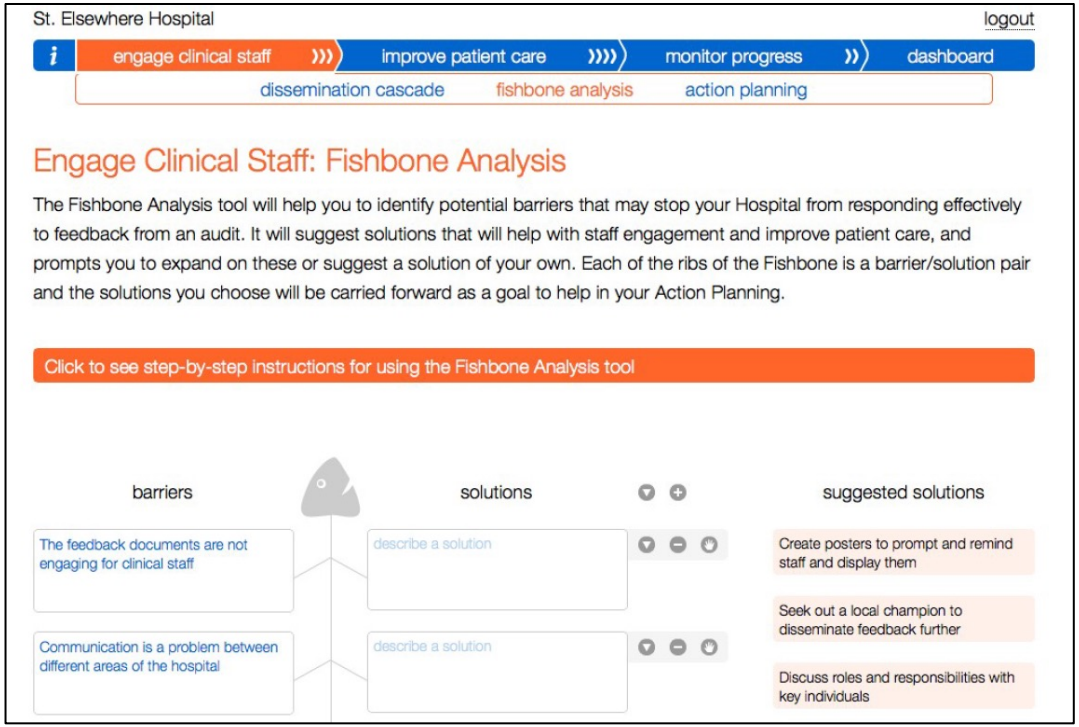
Affinitie – software

- I was asked to join an existing research project to develop a web toolkit for transfusion practitioners
- The research team consisted of health psychologists, statisticians, doctors and NHS Blood and Transplant
- The aim was to help transfusion practitioners disseminate blood transfusion audit results to everyone in the hospital

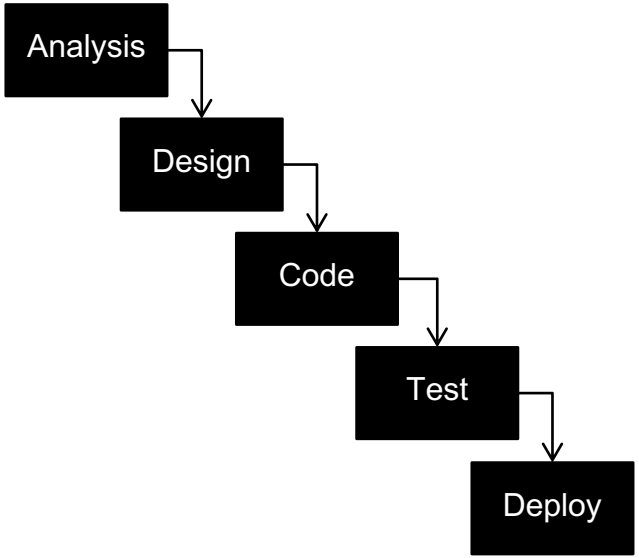


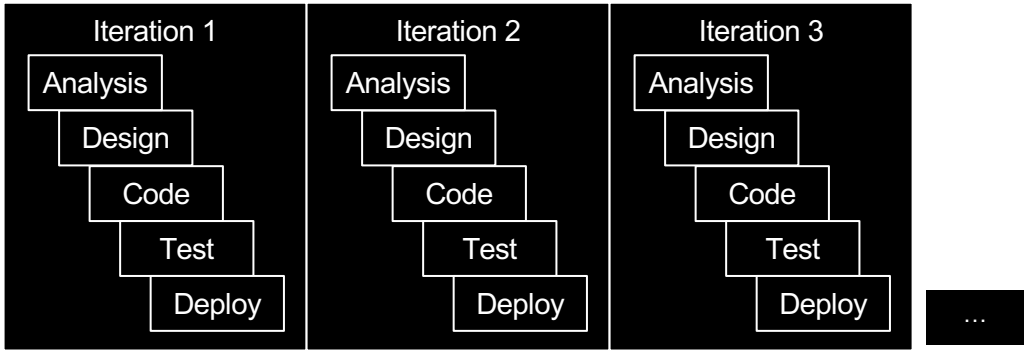
Affinitie – software development

- We were given **a clear set of requirements** by the research team
- They had already developed a set of paper-based tools for transfusion practitioners to help them disseminate blood transfusion audit results
- Software development approach: **waterfall**



Waterfall versus agile life cycles





What is Agile Software Development?

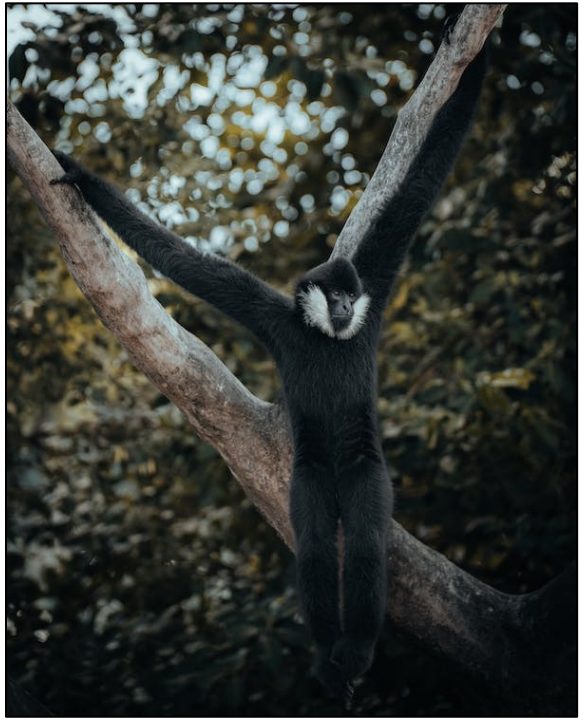
- Agile is a way of thinking about software development
- In winter 2001, 17 software developers met at a ski resort in Utah and drafted a manifesto outlining an alternative way to develop software to the documentation-driven software development processes of the time
- The manifesto is succinct and puts forward four key values for software development

The Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

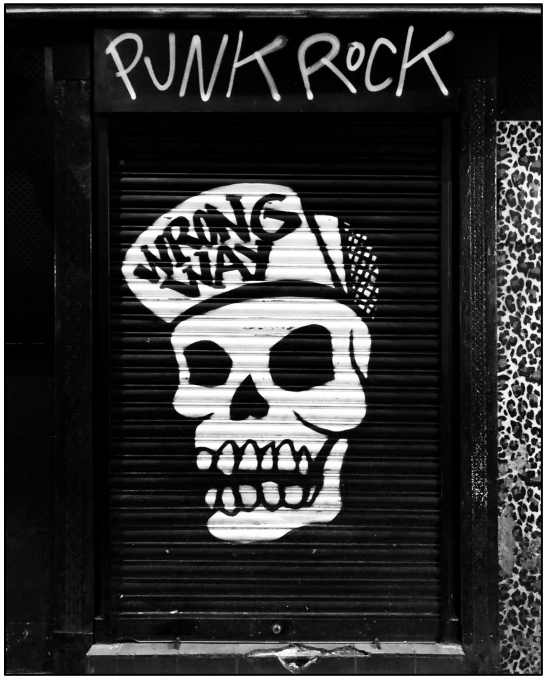
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more



How should you read the manifesto?

- The values are purposefully provocative in order to get people to think about software development
- The Agile Manifesto is **not** proposing that we disregard aspects of software development like processes, tools and documentation
- Rather, the Agile Manifesto wants people to think about alternative ways of doing aspects of software development



Agile was created by coders for coders

Coders like
Writing quality code
Ticking things off their to do list
Impressing clients by showing them working software

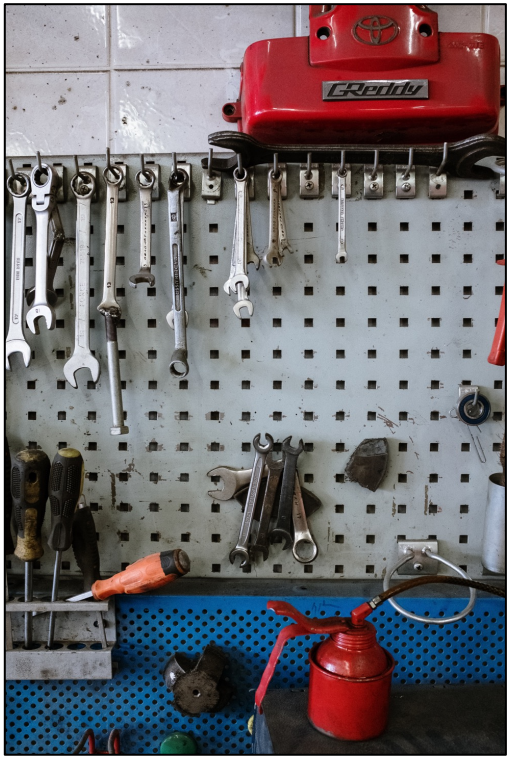
Coders dislike
Writing extensive documentation
Committing to a final design in advance
Being micromanaged
Working to big, immovable deadlines

Twelve agile principles

Satisfy clients' needs	Satisfy coders' needs
The highest priority is satisfying the client (by delivering working software early and continuously)	Work at a steady, sustainable pace (no heroic efforts)
Embrace change (even late in the development cycle)	Rely on self-organising teams
Collaborate every day with the client	Teams reflect regularly on their performance
Use face to face communication	Progress is measured by the amount of working code produced
Deliver working software frequently	Continuous attention to technical excellence
	Minimise the amount of unnecessary work
	Build teams around motivated individuals

Agile Methods

- There are various approaches that adhere to Agile values and principles
- Different companies choose different approaches
- Popular methods include:
    - Extreme Programming (XP) (the two co-creators were signatories of the manifesto)
    - Test-driven development (creator was a signatory)
    - Kanban
    - Scrum (the two co-creators were signatories)
- We'll introduce some key practices from each of these methods that we think you will be useful in this unit and your summer projects
- There are links in the reading to some of these methods



Hands up if any of these apply to you

- Your code structure is complex and “sophisticated”
- You work primarily on your own
- In group projects you are responsible for just your own code
- You write code in your own style
- You work some weekends and so some “all-nighters”
- You code develops in “heroic bursts”

Extreme Programming Ethos

- **Simple design**: use the simplest way to implement features
- **Sustainable pace**: effort is constant and manageable
- **Coding standards**: teams follow an agreed style and format
- **Collective ownership**: everyone owns all the code
- **Whole team approach**: everyone is included in everything

Extreme Programming Practices

- **Pair programming**: two heads are better than one
- **Test driven**: ensure the code runs correctly
- **Small releases**: deliver frequently and get feedback from the client
- **Continuous integration**: ensure the system is operational
- **Refactor**: restructure the system when things get messy

Pair programming in more detail

Code is written by two programmers on one machine:

- The **helm** uses the keyboard and mouse and does the coding
- The **tactician** thinks about implications and potential problems

- Communication is essential for pair programming to work
- Pair programming facilitates project communication

- The pair doesn’t "own" that code - anyone can change it
- Pairings can (and should) evolve at any time

- All code is reviewed as it is written
- The **tactician** is ideally positioned to recommend refactoring

The impact of pair programming

Research studies have assessed the impact of pair programming and identified a number of benefits

Single programmer 77 source lines per month versus pair programming 175 source lines per month

[Jensen, 2005]

15% increase in development-time costs but improves design quality, reduces defects, reduces staffing risk, enhances technical skills, improves team communications and is considered more enjoyable at statistically significant levels.

[Cockburn & Williams, 2000]

Test-driven development in more detail

- Tests are written before any code and they drive all development
- A programmer's job is to write code to pass the tests
- If there's no test for a feature, then it is not implemented
- Tests are the requirements of the system

The benefits of test-driven development

- Code coverage

We can be sure that all code written has at least one test because if there were no test, the code wouldn't exist

- Simplified debugging

If a test fails, then we know it must have been caused by the last change

- System documentation

Tests themselves are one form of documentation because they describe what the code should be doing

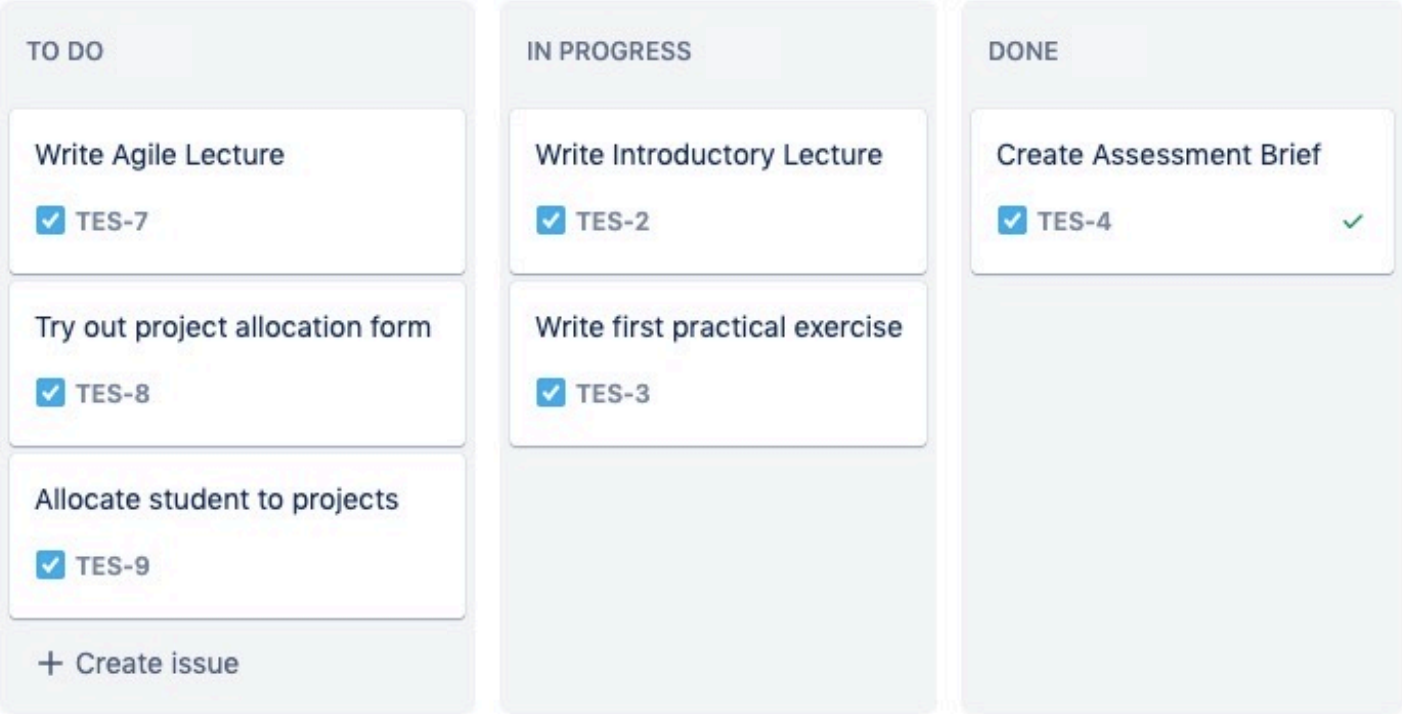
Scrum

- Scrum is a project management approach
- Some key concepts are:
- **The Scrum** – a **stand-up** daily meeting of the entire team
- **Scrum Master** - team Leader
- **Sprint** - a short, rapid development iteration
- **Product Backlog** – To do list of jobs that need doing
- **Product Owner** – the client (or their representative)

Kanban Board

- A concept taken from the Kanban method which was first defined in 2007 but came out of a scheduling system developed by Toyota in the 1950s for just-in-time manufacturing
- In Japanese “kanban” means “visual board” or “sign”
- It’s basically a flexible “to do” list tool
- Issues progress through various states from ”To do” to “Done”
- It was originally implemented as post-it notes on a whiteboard
- Various digital tools now fulfil the same function e.g. Jira

Jira Kanban Board

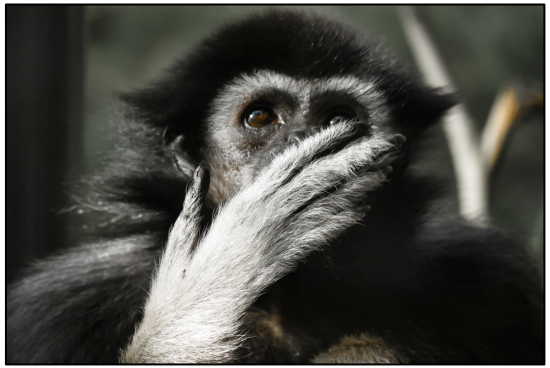


Columns (Swim lanes)

- It is common to use three columns
- But Jira allows you to customize the layout
- For example, you might have columns for:
  - Backlog
  - Being Verified
  - Awaiting integration
- Do what works for your team but make sure you have a “Done” column

Problems with Agile

- Hard to draw up legally binding contracts - a full specification is never written in advance
- Good for green-field development when you have a clean slate and are not constrained by previous work. However, it's not so effective for brownfield development which involves improving and maintaining legacy systems.
- Works well for small co-located teams, but what about large distributed development ?
- Relies on the knowledge of developers in the team but what if they aren't around (holidays, illness, turnover) ?



CHIME - motivation

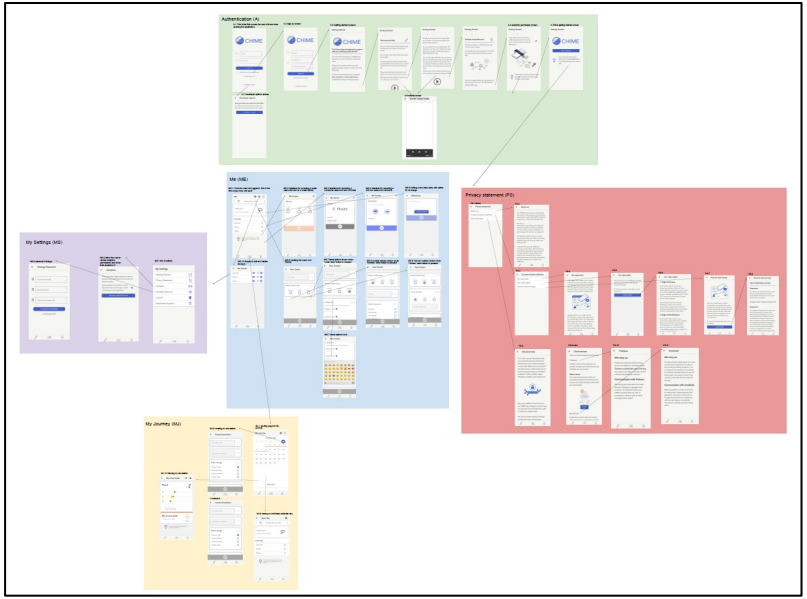
- Tracking health and lifestyle data can help people manage long term conditions
- Sharing these data with healthcare professionals can improve clinical decision making

CHIME - software

- Chime is an app designed for people living with HIV (PLHIV)
- It was developed by researchers at a number of UK universities in collaboration with the Terrence Higgins Trust, the leading HIV charity in the UK

CHIME – software development

- I project manged two programmers who developed the app code at UoB
- We initially designed and built a prototype app based on the requirements identified by other researchers on the project
- The app was evaluated by stakeholders who identified more requirements
- We then carried out a series of four two-week sprints and presented working code at the end of each sprint to other researchers
- The app was then evaluated by PLHIV
- Software development approach: **agile**



Reading

- Two research papers about evaluating pair programming mentioned in the lecture

R. W. Jensen (2005) A Pair Programming Experience, Overload, 13(65)

A. Cockburn & L. Williams (2001) The Costs and benefits of pair programming. Extreme programming examined. G. Succi and M. Marchesi, Addison Wesley: 223-243.

- A good overview of the Agile Manifesto and approach

Overview of the Agile Approach

- The Kanban method

https://kanbanize.com/kanban-resources/getting-started/what-is-kanban

- Unit testing

Unit testing in Java

A level of agility that progammers can aspire to

Agile Software Development

Tags

> **Topic Covered today:**

- Agile Software Development
  - Pair Programming
  - Test Driven Development
- Challanges

Agile Software Development :

1. **Concept:**

- The product owner will determine the scope to the project. This will involve discussing the key requirements with a client and prepare documentation to outline them, including what features will be supported and the proposed end results.
- It is advisable to keep the requirements to a minimum as they can be added on to later stages.
- The time and cost estimation is also done at this stage.

2. **Inception**

- Once the inception stage involves further input from stakeholders to fully flesh out the requirements on a diagram and determine the product functionality.

3. **Iteration**

- The iterative process is the process of breaking down large tasks into small manageable ones these are called iterations or sprints.
- In each sprint you are aiming for a potentially shippable product.
- Each sprint will involve:
  - Meetings / Standups
  - Testing
  - Review and Retrospective : At the end of the sprint the team reviews and presents it to the stakeholders. Which leads onto the next iteration

💡 How does Agile working method suit coders working style?

▼ Ans:

- It does not involve a lot of micromanaged tasks
- Working at a steady pace
- minimise the amount of unnecessary work

💡 Describe the iteration process of the Agile software development

▼ Ans:

- Every iteration aims to provide a shippable project.
- There are clear goals and retrospectives at the end of each sprint process
- Testing code through out on smaller chunks rather than a big code being tested at the end

Benefits of agile iterative development:

- Flexibility for making changes. The methodology allows for modification throughout the development process as each sprint aims to provide a final product
- Customer involvement. Customer / stakeholders feedback is taken more regularly
- Rapid delivery: The iterative approach requires less time on documenting
- Testing during each iteration is easier than testing a large developed final project

Agile Methods:

- There are various approaches that adhere to the agile values and principles :
  - Extreme Programming
  - Test Driven Development
  - Kanban
  - Scrum

Extreme Programming:

Pair Programming :

- Code is written by two programmers on one machine
- The **Helm** (the one writing the code) writes the code where as the **the tactician** thinks about potential implications and problems that could occur
- Communication is a key aspect of pair programming
- No one coder ‘owns’ the code
- Frequent switching of roles is encouraged

> 💡 What are the core concept of Pair Programming

▼ Ans:

Benefits:

- improvement in design and code quality
- Knowledge sharing and enhancement in collaborative efforts
- Focus on testing as you go along

Test Driven Development

- Test Driven Development emphasises writing tests for codes before writing the code itself
- The tests are based on the requirements of each sprint decided at the beginning of the development process
- There are three stages of TDD:
  - Red: Write a test for a functionality you want to add, this test should fail because the functionality doesn't exist yet
  - Green: Write the minimal code for the functionality to pass the test
  - Refactor: Clean up the code and make it more efficient if necessary

💡 What are the main concepts behind TDD

▼ Ans:

Benefits:

- Improved code quality
- Better design and better maintained code
- Documentation
- Confidence in change

Challenges:

- Because the requirements and specifications are not all written in advance drawing up legal contracts and exact milestones before hand can be difficult
- It is good for green field development, when a new system development process is occurring, not for brown field development where your are maintaining legacy systems or previous work.
- Not suited for large multi distributed teams across the country world.

💡 What are the key problems associated with the Agile method

UMLs: Sequence Diagrams

Tags

- Similiar to Class Digrams Sequence diagrams show how classes within a system or objects within code interact with each other

- In particular sequence diagrams show the order of the events

Example: ATM

- A ATM transaction involves the folloiwng steps:

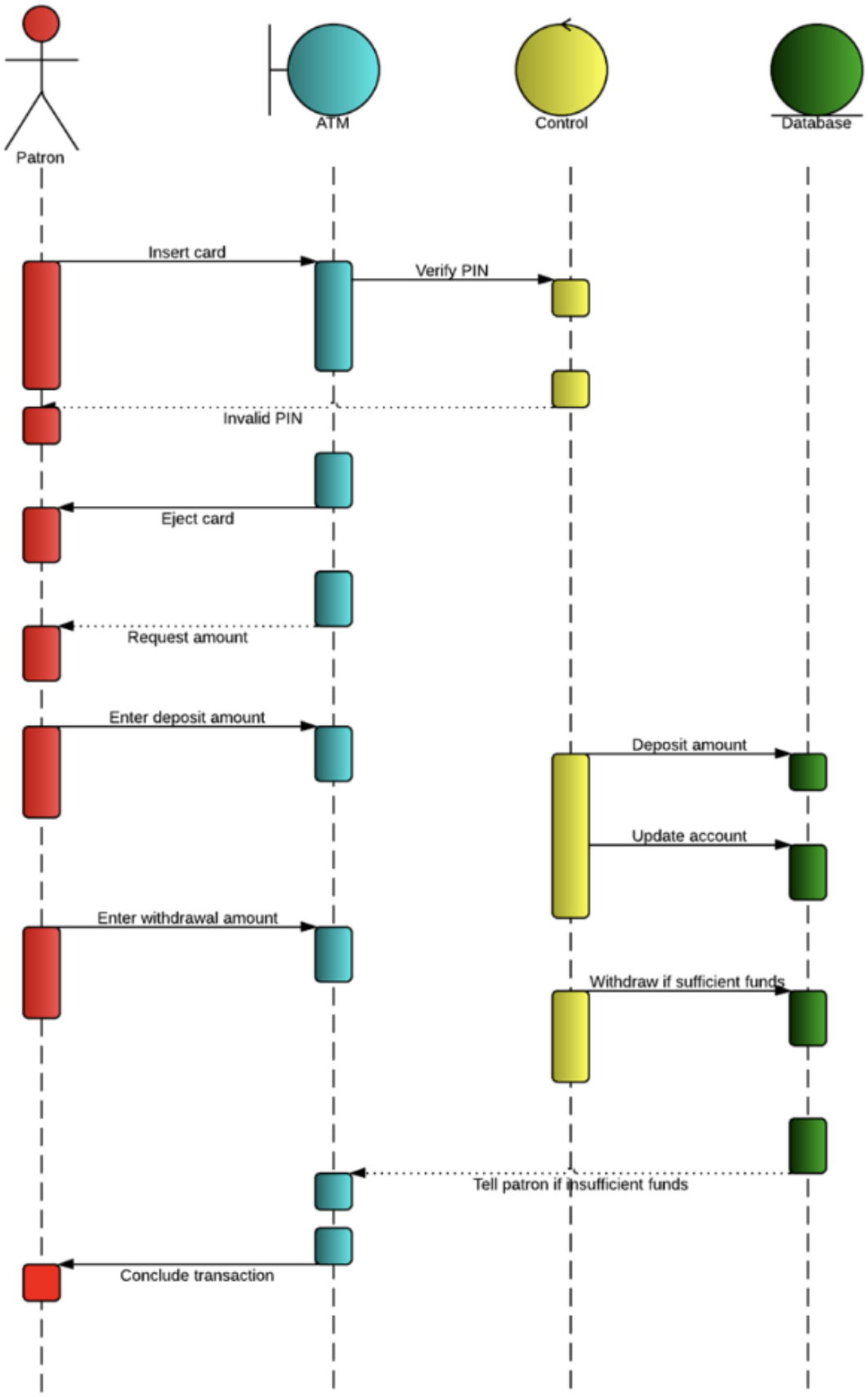
Person → ATM → Bank Server → Bank Account

- A person will use the ATM to access the Banks Server, which will then allow the person access to their own bank account
- In this example the person is called a **actor**
- The ATM / Bank Server / Bank Account are the **objects**

- The the sequence of events go from left to right.
- The dotted lines are called the **lifelines** these show the existance of an object or an actor over time, i.e. moving down the lifeline shows more time is passing

Sequence of events for an ATM transaction as represented by the Sequence Diagram above:

- The first **message** is the person interacting with the ATM by inserting thier card
- The **second message** is the verification of the bank card which happens between the ATM and the bank server
- The **retrun message** is shown by the dotted arrows here the return message shows that the bank card was verfieid ok
- However this is an **alternate case**, where if the card is valid or not, shown by the condition statemetns
- This then triggers the ATM to ask for a pin. But notice this is not a dotted line, because this is not a reply message directly realted to a another request.
- Then the ATM asks the bank server to verify pin
- An alternate case is again created
- If the pin and the card are valid the user is asked to enter a amount to withdraw,. again creating an alternate case, to see if the funds are sifficiant
- The boxes around the dotted lines are the **Activation boxes** these show the activation period of the objects


Patron
ATM
Control
Database
Insert card
Verify PIN
Invalid PIN
Eject card
Request amount
Enter deposit amount
Deposit amount
Update account
Enter withdrawal amount
Withdraw if sufficient funds
Tell patron if insufficient funds
Conclude transaction

Common Message Symbols:

Synchronous message

- Represented by a solid line with a solid arrowhead. This symbol is used when the sender must await for a response before continuing

Asynchronous message

- These messages do not require a response before continuing

Asynchronous retrun message

- Represented by a dashed line, these messages are response or return messages to the synchronous messages.

Use cases for sequence diagrams:

- Usage scenario with a clear timeline of events
- Method logic: Explore the logic of a system

Unified Modeling Language: Class Diagrams

Tags

> **Topics covered today:**

- Class Diagrams
- Sequence Diagrams
- Communication Diagrams

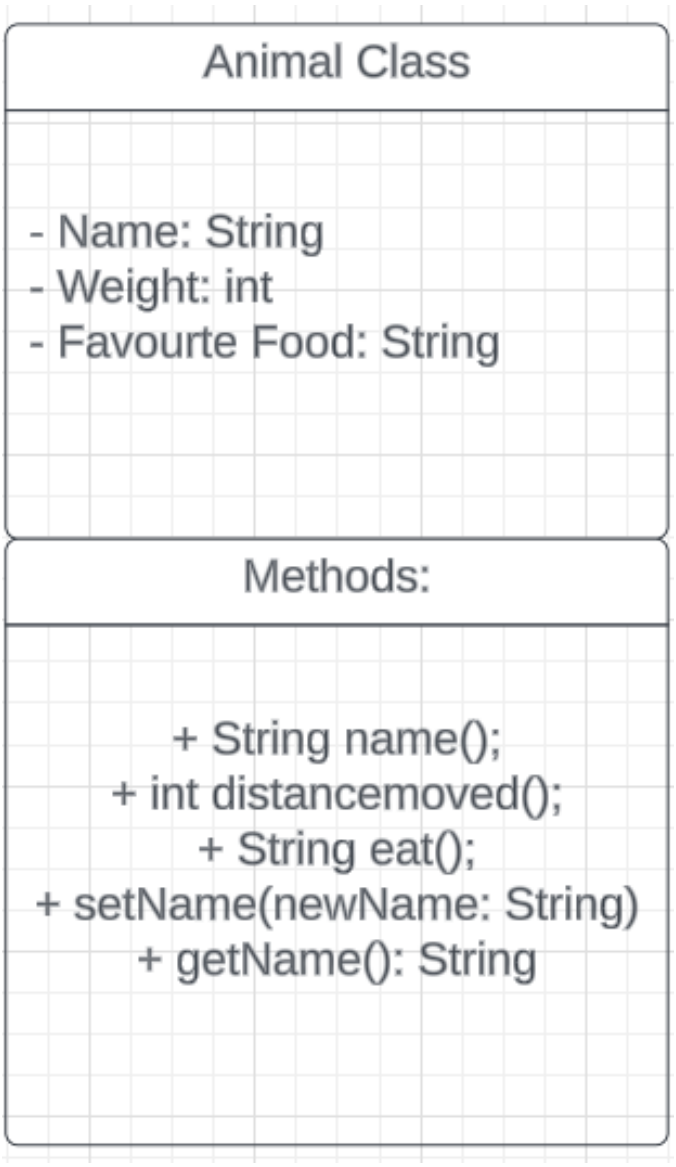
A unified Modeling Language was created to implement in visual representations a complex architectural design of softwares.

- The Agile and Waterfall methods in their graphical form are examples of UMLs
- The Agile development process is a adaptive planned process
- A UML is a graphical way of describing software systems

- a UML sketch is an overall look at the system you are trying to create
- Whereas a UML Blueprint is very detailed

Class Diagrams:

- Classes describe the type of objects that the system is going to use
- A class is made up of states / attributes on the top and Behaviours / Methods
- e.g. The animal class diagram below:



- The '-' and '+' represent visibility i.e. is the field private / protected or public.
- A protected visibility is specified by a #
- A ~ next to a attribute or method specifies package which means as long as the method is in the same package the class can be used by any other class
- Notice how the attributes are all private i.e. only visible to that class becuase class attributes should be private so that they cannot be altered by other fields. A generic abstract class should have private fields.

- The methods / behaviours should be protected i.e. only classes that inherit or extend the class can access the methods of the class.
- The nouns in the task description can be classes. But nouns can also be properties in the class:
-

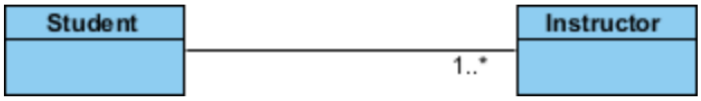
Multiplicity:

- Multiplicity is the number of instances one class relates to ONE instance of another class
- For example a animal class might relate once to a dog class, for each dog class you have identified an animal
- for Favourite food, in the animal class we could have it as a list of favourite foods this is declared using Food: [1 .... 10]
- If the number is unknown then this is represented by a '*' so you can have 0....*
- **Static Attributes** are those attributes that can be shared between all of the class objects that are created from that class

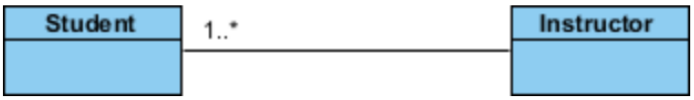
Association

- If two classes in a model need to communicate with each other, there must be a link between them, this can be represented by a line between them.
- The multiplicity is shown on the arrow

A single student can associate with multiple teachers:



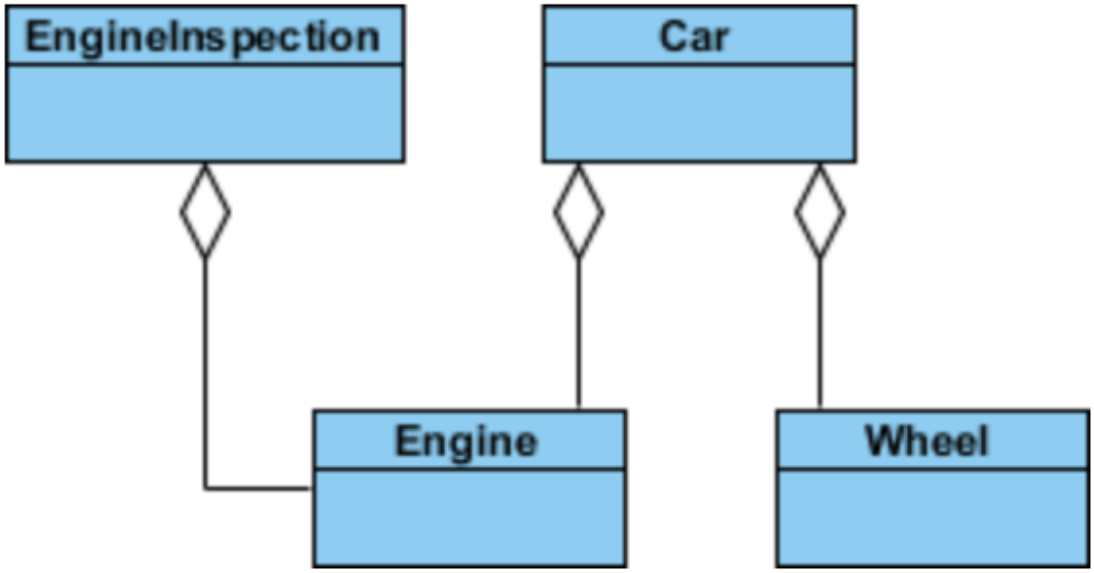
The example indicates that every Instructor has one or more Students:



Aggregation and Composition

- Aggregation and compositions are subsets of association, they each describe a parent child relationship, i.e. one class directly owns the attributes the other class is using. Another term you might see is the parents class being refered to as a “Whole” and the children class being refered to as “parts”
- In aggregation however, the child class can exist independent of the parents class. For example, A classroom (the parent class) can have attributes that describe students, it is the parents class to the student class. However, if you

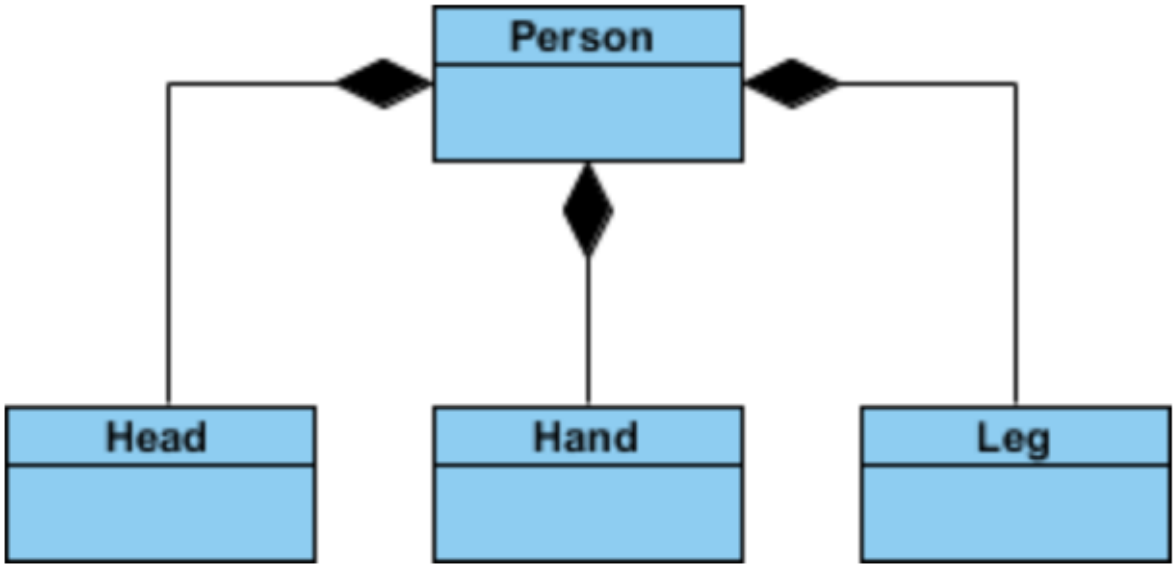
delete the classroom class the students will still exist, it has other attributes for example, height, student ID etc, these attributes are independent from the classroom class.



- Another example you can look at is a zoo, imagine you have a class that represents a group or crows called a “Murder” a single crow class can exist as part of the Murder class but if you deleted the murder class i.e. if you the zoo did no longer have a group of crows, each individual crow would still exist as its own entity.
- A aggregate relationships such as theese are represented by a open diamond arrow

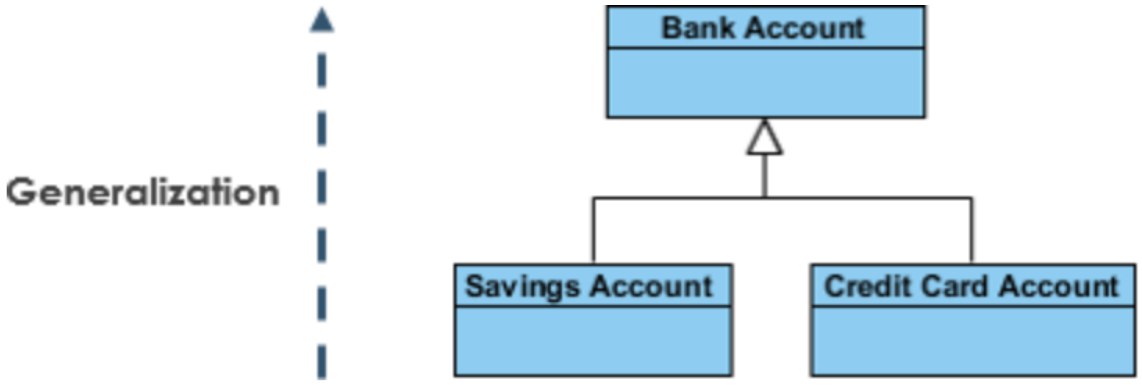
Composition

- A composition is where a child class cannot exist without a parents class e.g. House (parent class) has a child called Room (child class), a room cannot exist without a house



Generalisation / Inheritance

- Generalisation is the mechanism of combining two or more classes that share attributes / methods into a single entity
- Generalisation is represented by a open arrow as shown below:
- For example a savings account and credit car accounts have lots of similar information, such as a persons name, their account info and so on. So these can be generalised to a single class called bank account nt
- You are still keeping the savings account and credit card account as spereat classes however, because they also contain information that is independent form each other like interest will be on the credit card account but not the savings account



- Another example can be: Squares, Rectangle and Circle all belong to parent class called shapes.
- The Common attributes that can describe them is e.g. number of sides
- WIth Inheritance another key word is also useful: **Abstraction**
    - Abstraction is the parent class such as the shape class that all shapes belong to, it is an abstract class becuase the class itself is never instantiated.
    - You can represent an abstract class by either puttong the class name in italics or in “<<>>” e.g.: <<Shapes>>