CS261
Software Engineering

Revision Lecture

James Archbold
James.Archbold@warwick.ac.uk

In association with Deutsche Bank

Introduction

- Welcome to the CS261 Revision Power Hour
- So much power, it actually lasts two hours.
- We’re going to follow a fairly clear format here:
  - Using the 2019 paper as a basis, we’ll be answer exam questions and then covering any extra material around that section.
- Doing this, we’ll go through the majority of the module content.
- After all four sections, there’ll be some quick hints and tips.
- We had a couple questions so far - I’ll cover those at the start of next session.

The Exam

- The exam is 2 hours.
- It consists of 4 questions - and you are expected to answer them all.
- Often in black ink, on white paper, unless you have mitigating circumstances / special conditions.

The Exam

- The four questions cover the four major sections of content we went through in the lectures.
- Past papers are all fairly representative of what you might expect.
- There are few questions where you just regurgitate information from the slides

Hints & Tips

- Read the question - particularly look at what it is asking for.
    - In the past people have answered the wrong question, or not provided enough information.
- Often, the key is to justify your choices or to contextualise your answer properly
- The questions will give you all the information you need.
- The questions also follow the software development life cycle, so it can be helpful to approach them in order.

Topic 1: Software Methodologies

- Question 1 in the CS261 paper typically covers the first few weeks of material.
- The focus is on requirements analysis and software methodologies
- Things to read up on:
  - Requirements engineering - including the different stages
  - Different types of requirements - function vs non-functional, C vs D
  - Writing effective requirements
  - Software methodologies - plan vs agile
  - Pros and cons for methodologies

Question 1A:

(a) Contrast plan-driven development and agile development in terms of:

i. Accommodating requirement changes. [2]

ii. Perceived difficulty to implement. [2]

i. Accommodating requirement changes:

Plan-driven development is meant to be strictly planned before any development starts, and so accommodating changes becomes very difficult.

Agile development believes in minimal planning, where features are incrementally planned and then immediately implemented, and so including a change does not disrupt development.

Question 1A:

(a) Contrast plan-driven development and agile development in terms of:

i. Accommodating requirement changes. [2]

ii. Perceived difficulty to implement. [2]

ii. Perceived difficulty to implement:

Plan-driven development is seen as easier to implement, as with proper planning, implementing should be a simple conversion process from design to code.

Agile development is seen as having a higher level of difficulty to implement. The design is less rigorous, and code must be easier to maintain and modify.

Question 1A:

(a) Contrast plan-driven development and agile development in terms of:

i. Accommodating requirement changes. [2]

ii. Perceived difficulty to implement. [2]

- Things to keep in mind:
  - In this question you are comparing two different methodologies
  - It is not enough just to say ‘this is easier’ or ‘this is harder’
  - You need to provide a reason that demonstrates you understand the methodologies being compared.

Question 1B:

Describe the major processes involved in Extreme Programming, including potential drawbacks. [7]

Extreme programming focuses on building several versions a day, with prototypes delivered to the customer regularly.

It focuses on test driven development, with automated tests running to verify every build. These tests must run quickly to maintain the pace of development.

Code is constantly refactored to maintain simplicity.

Strong customer involvement is essential.

To reduce islands of knowledge and promote collective ownership, pair programming is used.

There is a desire to maintain a sustainable pace to avoid overtime.

In terms of drawbacks, it requires heavy customer involvement otherwise development cannot proceed.

It also requires teams to be small and experienced, due to the difficulty in implementing this process.

Question 1B:

Describe the major processes involved in Extreme Programming, including potential drawbacks. [7]

- Things to keep in mind:
  - You need to know the major steps of software development processes
  - You also need to be able to discuss their positives and negatives
  - This type of question can be easily refocused around any number of the software design processes.

Question 1C:

(c) For each of the following requirements, identify if they are functional or non-functional and if they are C-facing or D-facing requirements. You must justify your reasoning.

i. User details should be stored on a NoSQL database system, that is encrypted using the SHA-512 cryptographic hash functions. [2]

ii. The system should be usable after 15 minutes of training. [2]

i. Developer facing - we see detailed, technical language.
Functional - describes something the system will do / a function that it will perform.

ii. Customer facing - uses non-technical, simple language.
Non-functional - describes a quality of the system.

Question 1C:

(c) For each of the following requirements, identify if they are functional or non-functional and if they are C-facing or D-facing requirements. You must justify your reasoning.

i. User details should be stored on a NoSQL database system, that is encrypted using the SHA-512 cryptographic hash functions. [2]

ii. The system should be usable after 15 minutes of training. [2]

- Keep in mind:
  - How we define the different types of requirements.
  - What makes something D-facing? C-facing?
  - Justify your answers! Demonstrate you understand.

Question 1D:

Detail the four main steps of requirements engineering, and their expected outcomes. You should include a diagram of how these aspects relate to each other. [10]

1. Feasibility Study - This determines if the project is cost effective and will produce a feasibility report.

2. Requirements elicitation and analysis - Derive system requirements via documentation, customer interviews and feature discussion. Produces a system models.

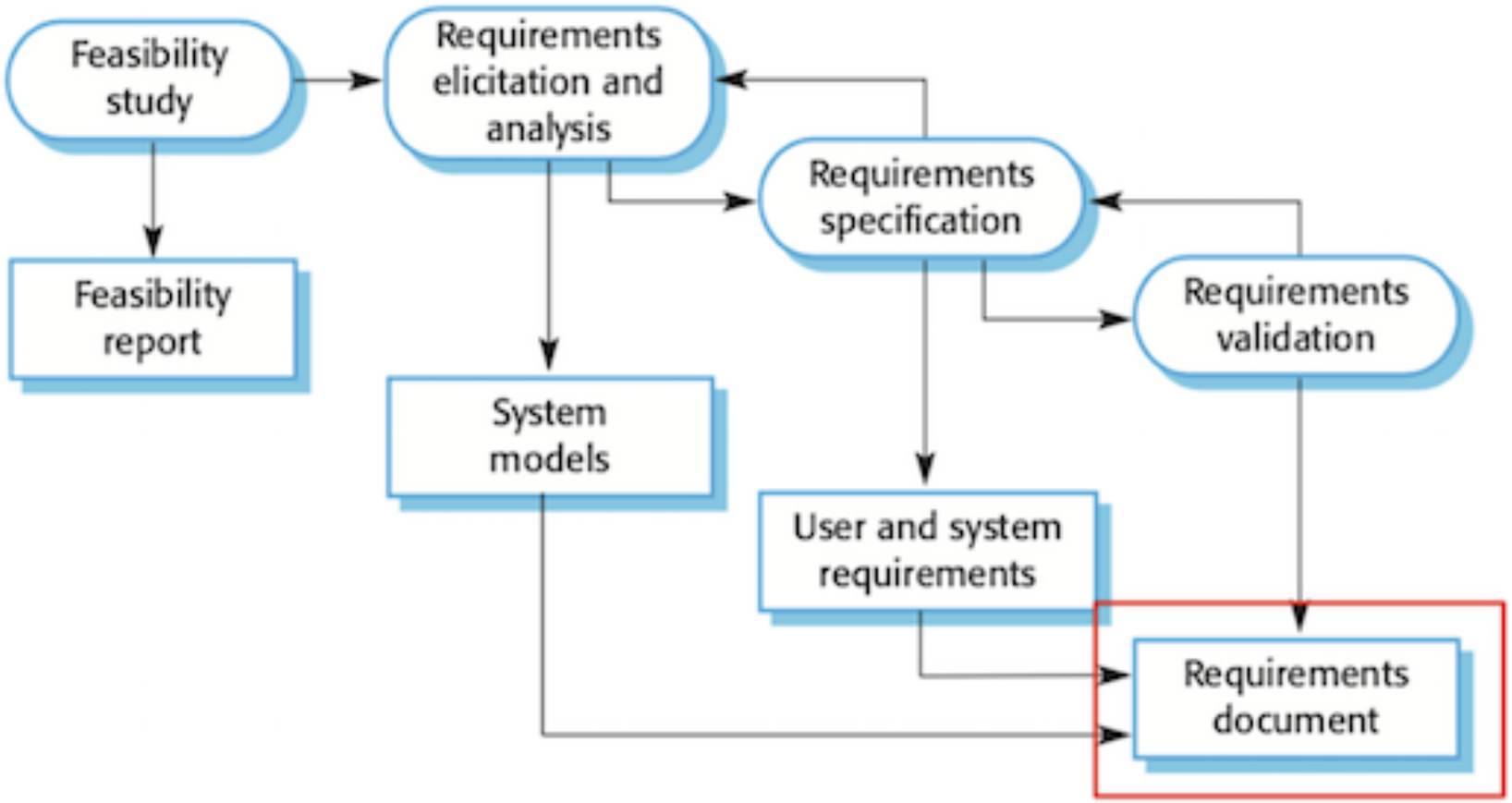
3. Requirements specification - Translates information gathered in elicitation phase into documents that define the set of requirements - Produces user and system requirements.

4. Requirements validation - Ensures requirements are achievable and valid and produces the final requirements documentation.

You should also mention how these steps are not strictly sequential and often, the specification or validation can require an adjustment in a previous step.

Question 1D:

Detail the four main steps of requirements engineering, and their expected outcomes. You should include a diagram of how these aspects relate to each other. [10]



Question 1D:

Detail the four main steps of requirements engineering, and their expected outcomes. You should include a diagram of how these aspects relate to each other. [10]

- Keep in mind:
  - Engineering is different from analysis - engineering is the entire overall process
  - Many questions may ask you to fill in diagrams or identify diagrams
  - The documentation produced is important!

Overview

- Hopefully, by going through this question, you understand the type of knowledge you need for this module's assessment.
- In the next slides, we'll just quickly go through some topics
- This detail will be brief - and you should go over your notes of these topics

Software Process Model

- A sequence of activities that leads to the production of a software product
- There are many different processes, but all involve:
  - *Software specification* - what the software should do
  - *Software design and implementation* - how it should be organised and implemented
  - *Software validation* - checking it does what the customer asked
  - *Software evolution* - changing the software over time

The Waterfall Model

- The Waterfall Model works if the requirements are understood and will not change during development
- Few team constraints (size, location) as development can be distributed in discrete chunks
- Each component can be independently tested against its own specification before integration (outsourcing)
- Easy to add members to the team because the whole system is well documented (churn)

- Customer can wait a long time before they see results
- Using this model, it is very difficult to accommodate change once the process is underway
- Difficult to respond to changing customer requirements
- Could be a major problem if the project is long running
  - Business priorities change
  - Customer expectations change
  - Underlying technologies change

Incremental Development

- ✓ Cost of accommodating changing customer requirements is reduced
- ✓ Software available to user quicker, and therefore feedback can be easily solicited
- ✓ Greater perceived value for money - customers can see development progress
- ✓ Include the user in acceptance testing at each phase.

- ✗ Difficult to estimate the cost of development
- ✗ Difficult to maintain consistency with new features being added - poor design choices at the beginning may hinder later development
- ✗ As progress continues, it becomes harder to include new features or make changes to fundamental components
- ✗ Not cost-effective to produce documentation for every version of the software
- ✗ Increased cost of repeated deployment

Reuse-oriented Software Engineering

- **Component Analysis**
  - Searching for available components to meet the specification.
- **Requirements modification**
  - Analyse the found components to determine if they meet the requirements, modifying requirements if acceptable to do so.
- **System design with reuse**
  - Design the system using the components.
- **Development and integration**
  - Integrate the selected components into a complete system.

Motivating Agile Methods

- Software engineering has changed over the last 20 years
  - Software systems are **common to all businesses**…
  - …and these businesses expect to be able to **evolve rapidly**
- Process-driven approaches become too cumbersome
  - Need to **focus on the code** as opposed to the design
  - Aim is to deliver **working code quickly**, to be able to meet **changing requirements**

Requirements Specification - User Requirements

Functional Requirements

- Describe what the system should do
- Statements of services the system should provide
- Detail how the system should read in certain scenarios

Non-Functional Requirements

- Often termed ‘qualities’ - availability, performance, deployment
- Constraints on the services or functions offered by the system
- Legislative constraints on the system (any applicable laws)
- Often apply to the system as a whole rather than individual components

Requirements Validation

Once requirements have been documented, they must be approved and shown to be acceptable

<u>Consists of checks on the requirements:</u>

- *Validity* - will the system support the customer's needs?
- *Consistency* - are there any conflicts?
- *Realism* - can the system be produced with the available budget and technologies?
- *Verifiability* - once complete, can the system be shown to satisfy the requirements?

CS261
Software Engineering

Revision Lecture
Topic 2: System Modelling

James Archbold
James.Archbold@warwick.ac.uk

In association with Deutsche Bank

Topic 2: System Design

- Question 2 is always focused on the design stage of development.
- Most notably, this requires UML to design systems.
- Read the question carefully, and make sure you provide everything it asks for!

Question 2A:

(a) Discuss how system design is supported by system modelling, including the different perspectives that must be considered. [5]

System modelling helps to clarify functionality of the final system. It informs design approaches and component level decisions. It also provides a basis for development.

Four perspectives to consider:

External view - We model the context of the system.

Interaction view - Model interactions between the system and its environment, or between system components.

Structural view - Model the organisation of the system or the structure of the data being processed.

Behavioural view - Model dynamic behaviour of the system.

Question 2A:

(a) Discuss how system design is supported by system modelling, including the different perspectives that must be considered. [5]

Things to keep in mind:

- All of that information is not necessarily needed.

- You cannot just name the perspectives, but what that perspective offers.

- Again, information needs an explanation and contextualisation.

Question 2B:

(b) You have been asked to model a new software system for sharing and creating short video clips on mobile, less than ten seconds long.

i. Design class UML diagrams for the *VideoClip* object and the *ShareService* itself. Your design must be *complete*. If you include attributes such as creators, genres, date, subscriber and playlist, these must also be included in your design. Where possible, you should also indicate the relationship between designed classes, using appropriate UML notation. [8]

ii. Using appropriate behavioural diagrams, model the process of a user creating and sharing a video with other users. You should include elements such as setting a video to public/private, notifying subscribers and allowing for both internal video creation and just uploading a previously created video. Please include a narrative to explain your design choices. [8]

Question 2B:

- These types of questions can be slightly open-ended - which is why they often ask for explanation along with the diagrams.
- These questions focus on good UML use and sensible software design.
- Let’s start with class diagrams.

VideoClip

+ Genre: String
+ Title: String
+ Length: Long
# DateCreated: Date
- Creator: User

+ shareVideo (targets : List<User>):

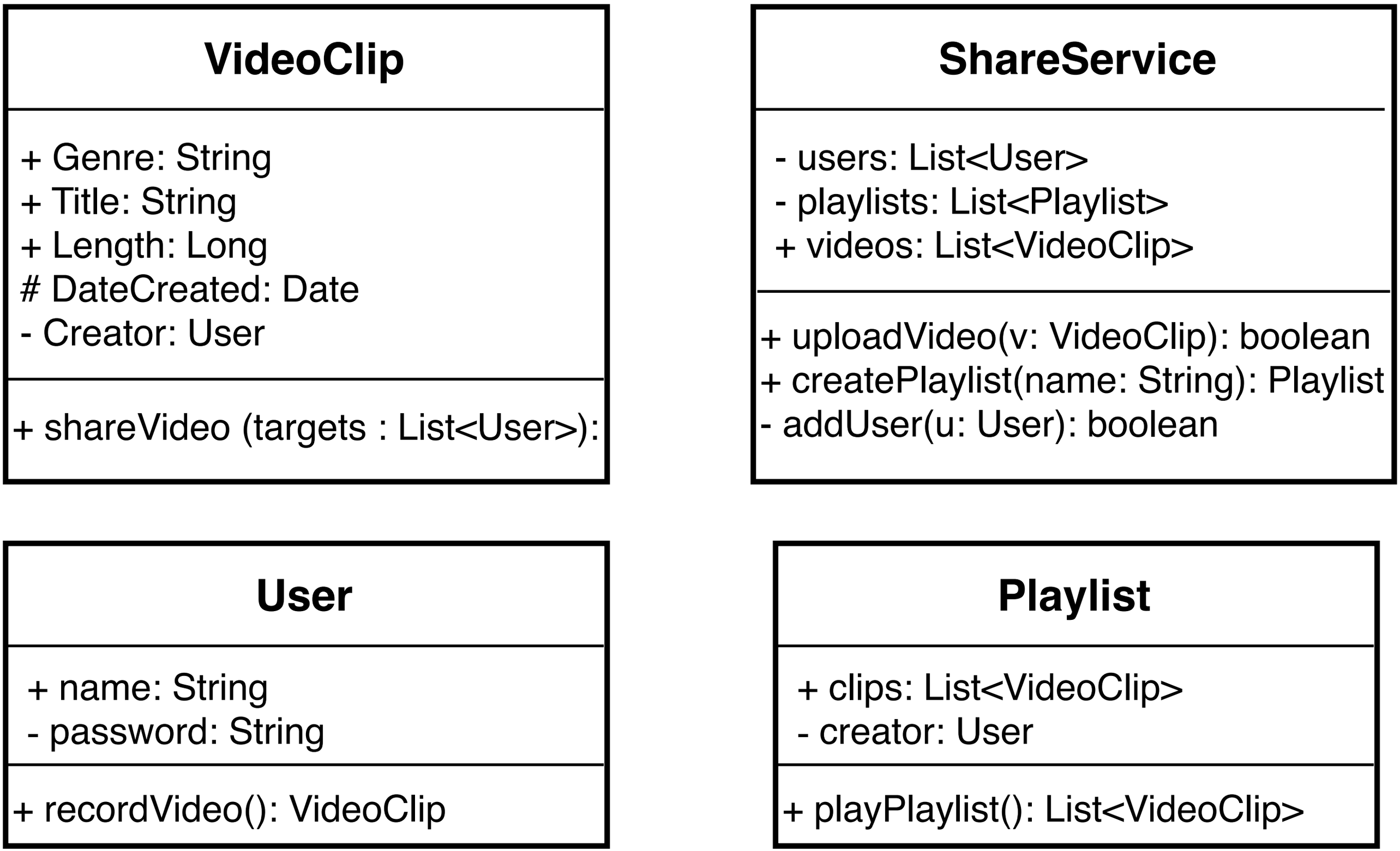
ShareService

- users: List<User>
- playlists: List<Playlist>
+ videos: List<VideoClip>

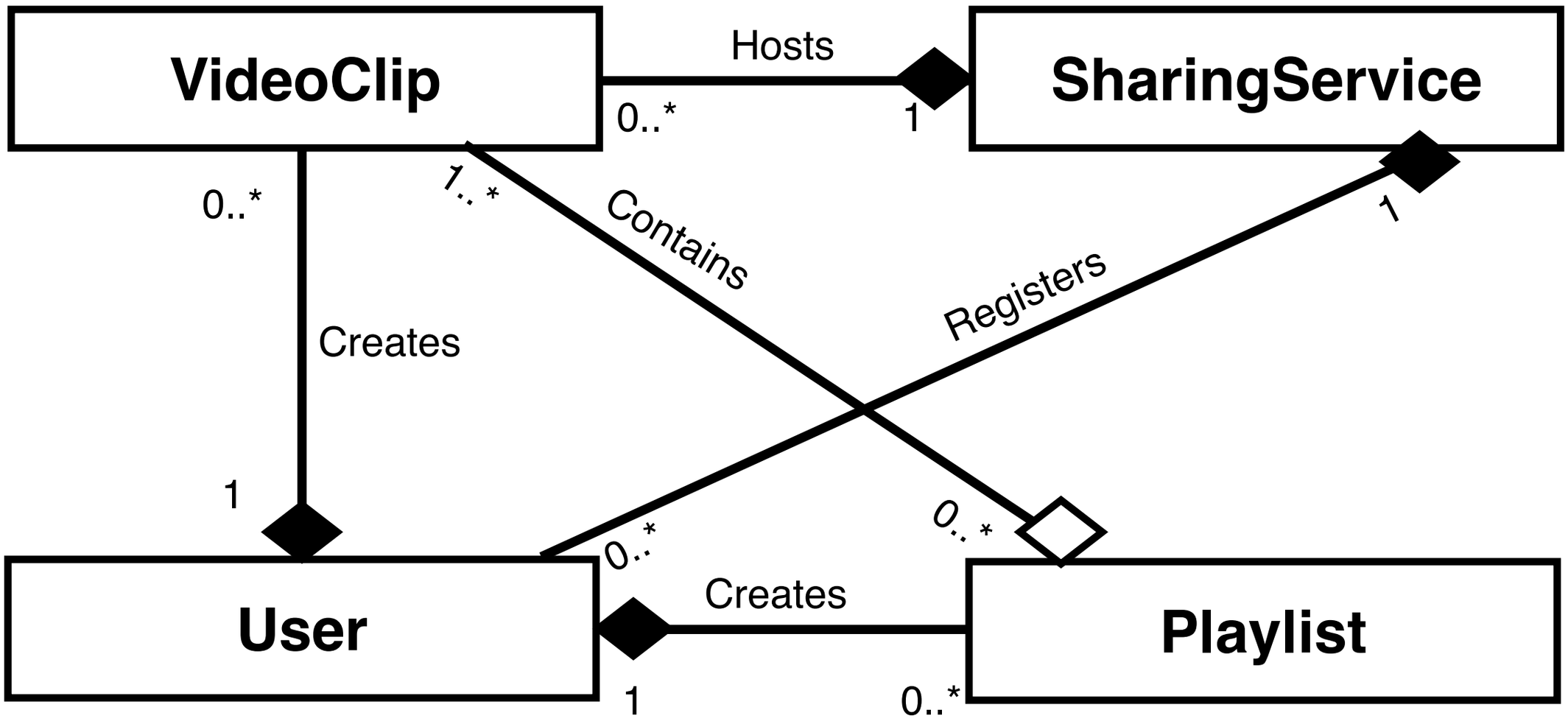
+ uploadVideo(v: VideoClip): boolean
+ createPlaylist(name: String): Playlist
- addUser(u: User): boolean

- Remember to correctly label the access type: public, private, protected etc.
- Your methods need parameters and return type.
- You don’t need these exact methods and attributes - but you should justify your decisions! Don’t be afraid to annotate.
- If you read the question - we’re not done. Our design needs to be *complete.*
- This means defining the extra classes (playlist and user) AND providing correct connections between the classes

Question 2B


VideoClip
+ Genre: String
+ Title: String
+ Length: Long
# DateCreated: Date
- Creator: User
+ shareVideo (targets : List<User>):
ShareService
- users: List<User>
- playlists: List<Playlist>
+ videos: List<VideoClip>
+ uploadVideo(v: VideoClip): boolean
+ createPlaylist(name: String): Playlist
- addUser(u: User): boolean
User
+ name: String
- password: String
+ recordVideo(): VideoClip
Playlist
+ clips: List<VideoClip>
- creator: User
+ playPlaylist(): List<VideoClip>

Question 2B


VideoClip
Hosts
SharingService
0..*
1
0..*
1..*
Contains
1
Registers
Creates
1
0..*
0..*
User
Creates
Playlist
1
0..*

Question 2B

(b) You have been asked to model a new software system for sharing and creating short video clips on mobile, less than ten seconds long.

i. Design class UML diagrams for the *VideoClip* object and the *ShareService* itself. Your design must be *complete*. If you include attributes such as creators, genres, date, subscriber and playlist, these must also be included in your design. Where possible, you should also indicate the relationship between designed classes, using appropriate UML notation. [8]

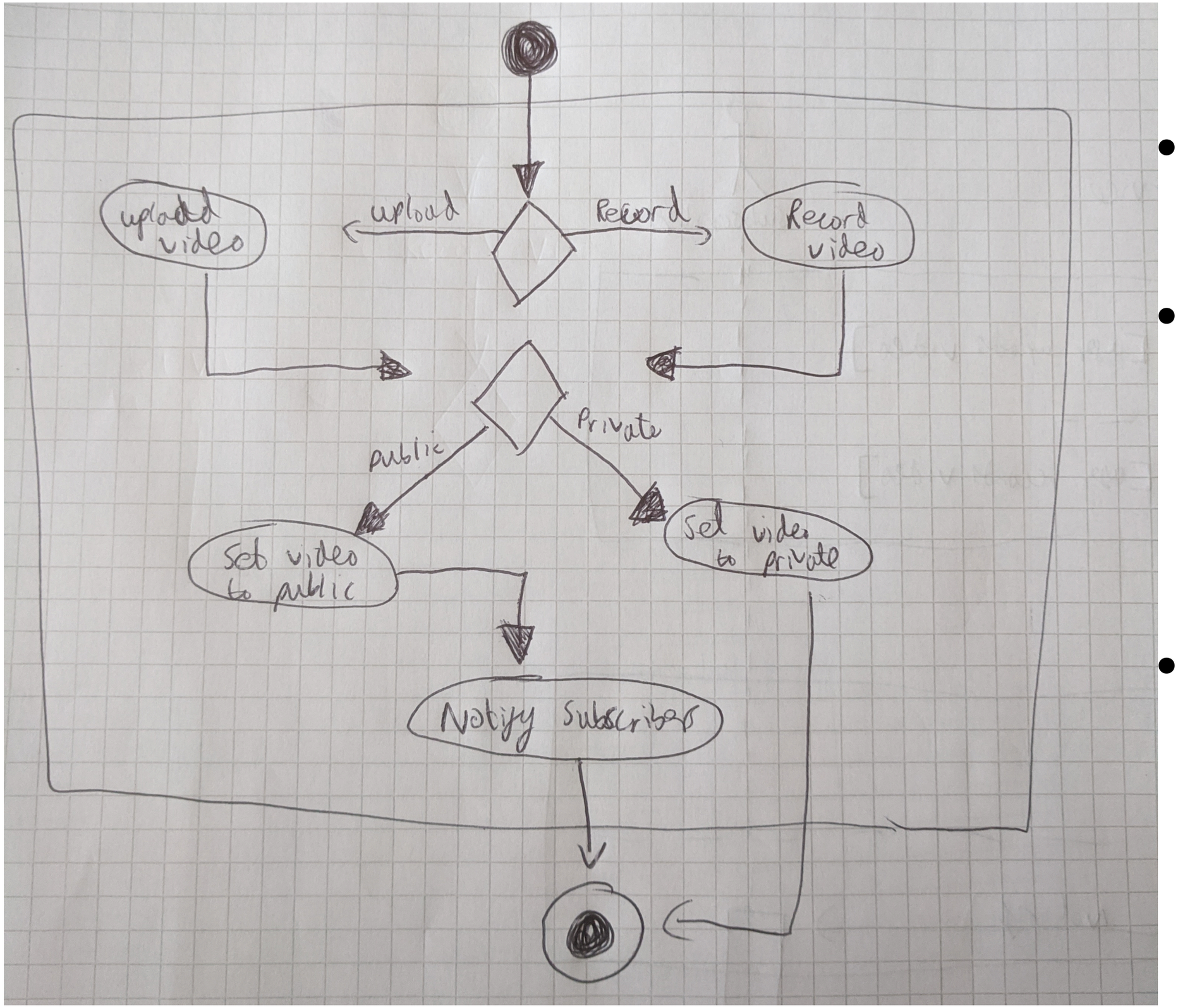
- The relationships and extra classes do not need to be exactly as described here - but your choices need to be explained.
- Make sure you are correctly labelling the relationships and using the proper modifiers!

Question 2B

ii. Using appropriate behavioural diagrams, model the process of a user creating and sharing a video with other users. You should include elements such as setting a video to public/private, notifying subscribers and allowing for both internal video creation and just uploading a previously created video. Please include a narrative to explain your design choices. [8]

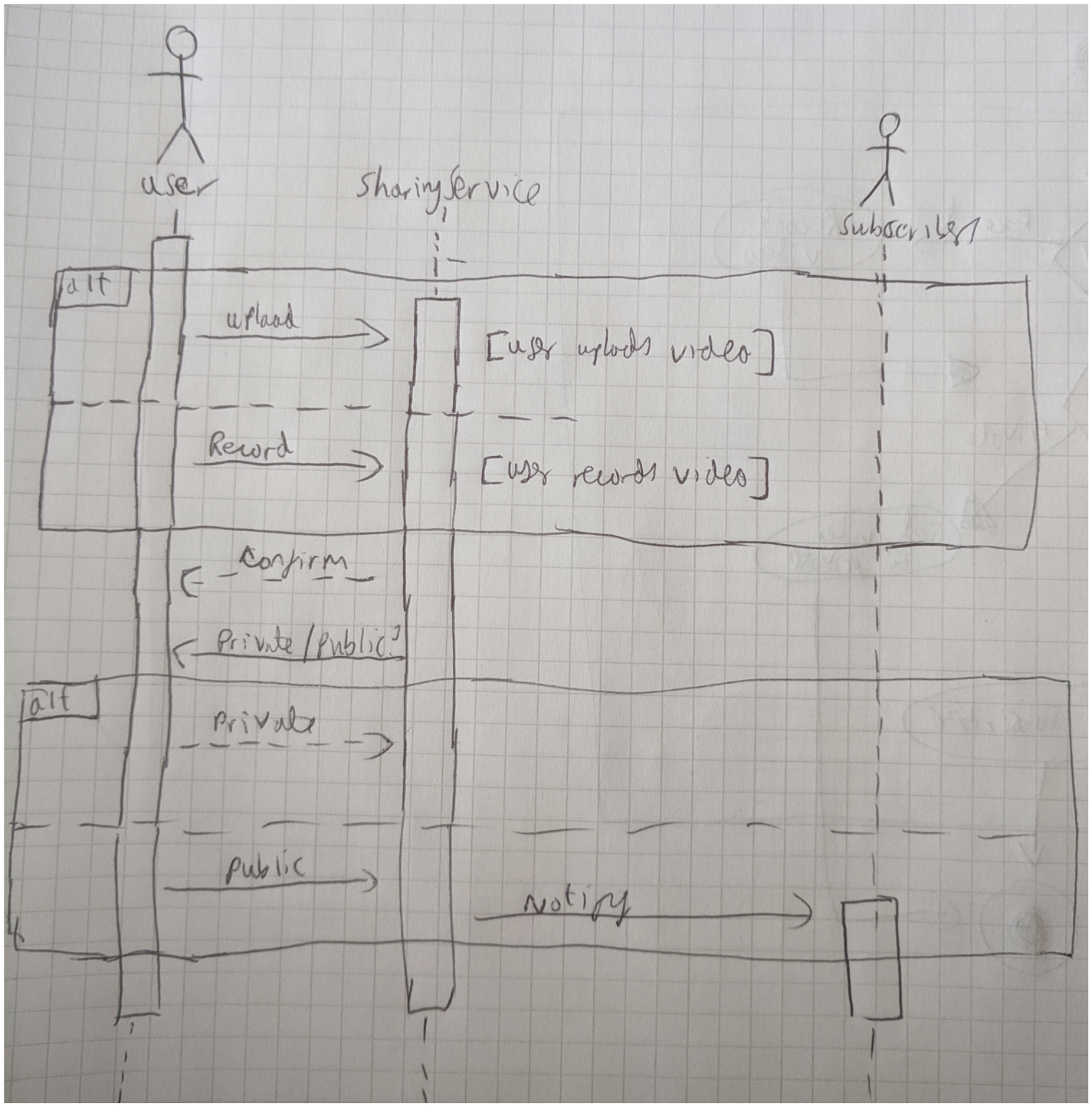
- For this question we probably want to use an activity diagram or a sequence diagram.
- You also probably want to include a use case diagram.
- Note that the question asks for diagrams - plural.

Question 2B



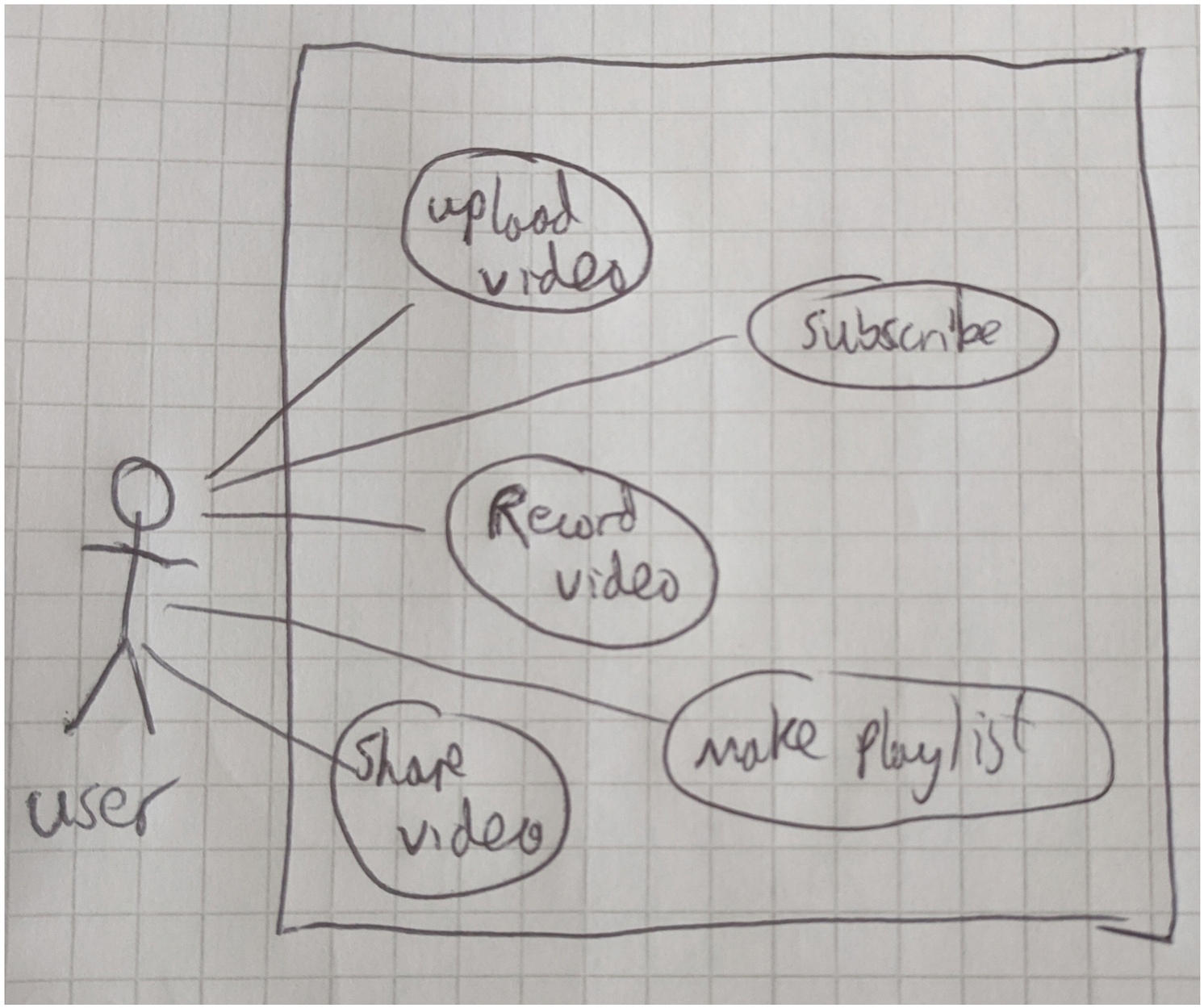
- Enjoy my handwriting.
- Remember, for activity diagrams to use the correct symbols for each action.
- Don’t forget to include a narrative to justify choices!

Question 2B



- Remember to label each system.
- Don’t forget how to do alts and loops
- Remember that we need to show when a system is active - this diagram is actually slightly wrong at the end

Question 2B



- This is a nice easy use case diagram
- Remember, the box is the system boundary!
- The sharing service should not be outside the box.

Question 2B

- It is important to remember the correct way to construct these different diagrams - and the type of thing they are used to represent.
- Diagrams need to be clear and correctly labelled.
- Remember that these diagrams may need some extra discussion and narrative to explain choices.

Question 2C

(c) Discuss how a Finite State Diagram differs from an Activity Diagram, and their relationship with each other. [4]

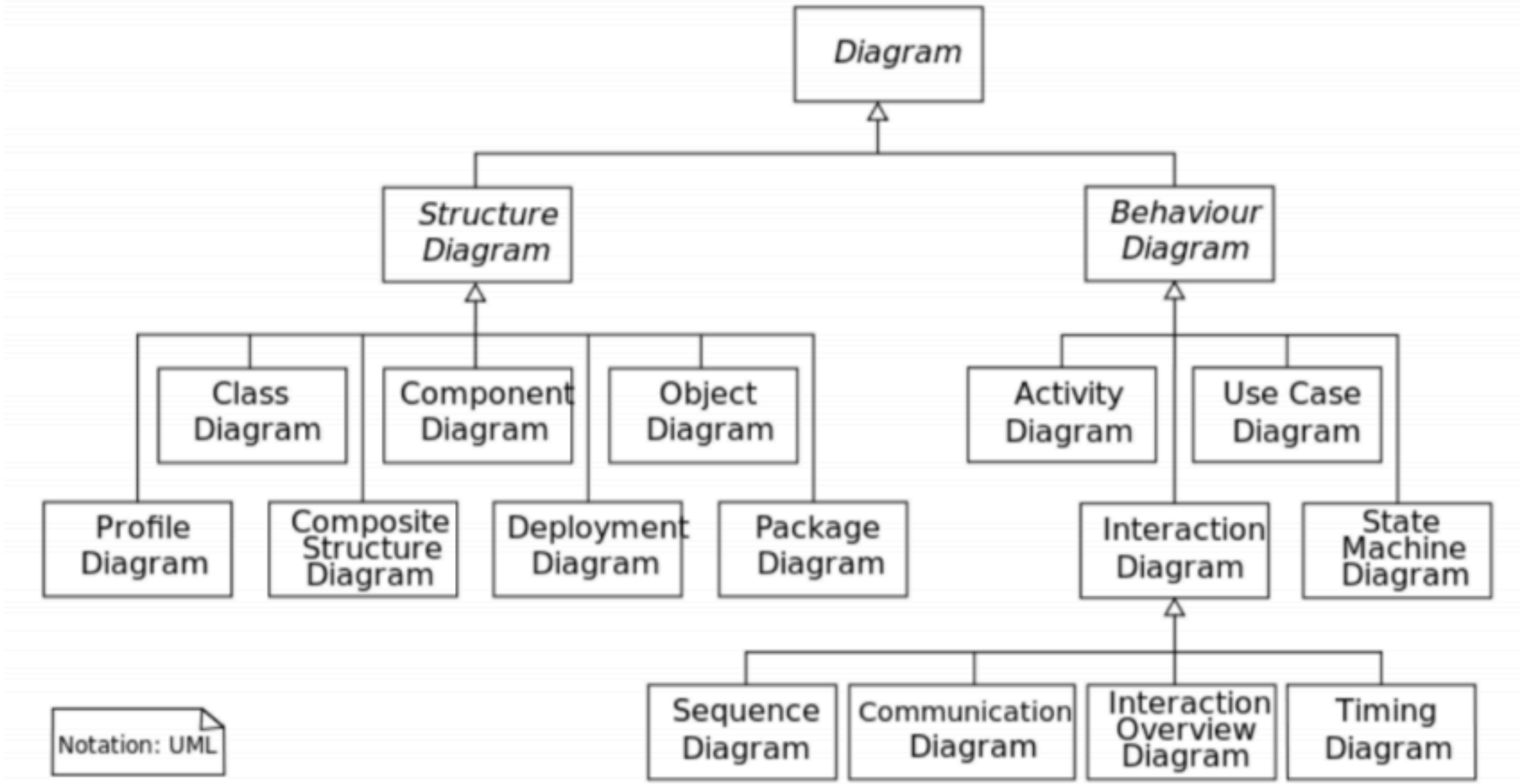
- A State Diagram models how the state of the system changes in response to particular actions.
- An activity diagram models the stepwise actions needed to achieve a given task.
- The actions modelled in an activity diagram are the same actions needed in the state diagram.
- Each action results in a state change, which must be modelled in the state diagram, so the activity diagram tells us the actions our state diagram must include.

Overview

- Clearly, knowing not only how to model a system, but *why* is important.
- This is a very practical section, and so the questions can be quite varied.
- Make sure you're confident in the various diagrams, and their role in system design.

UML Diagram Overview

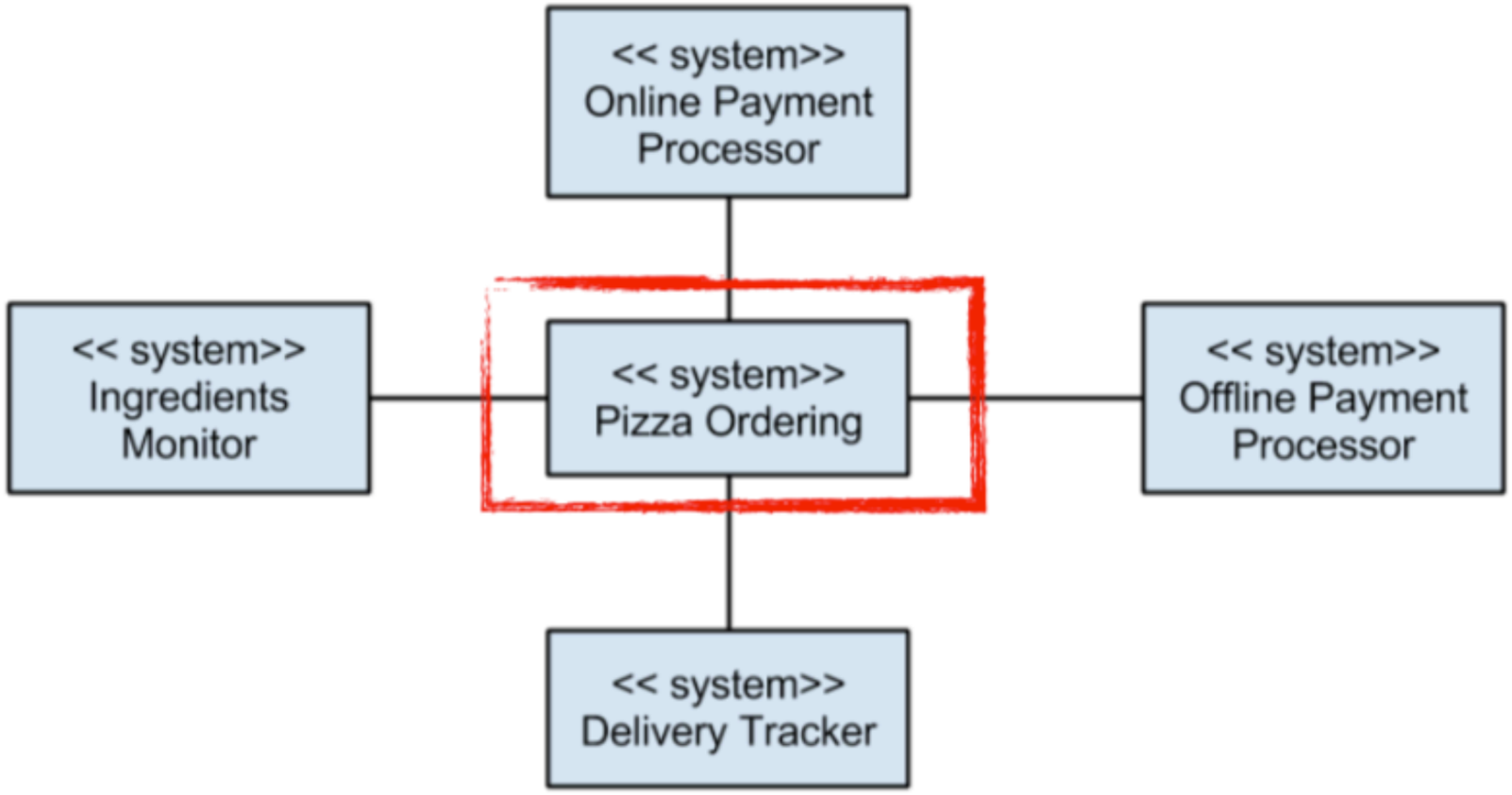
Taxonomy of diagrams (model types) that UML provides



Context Models

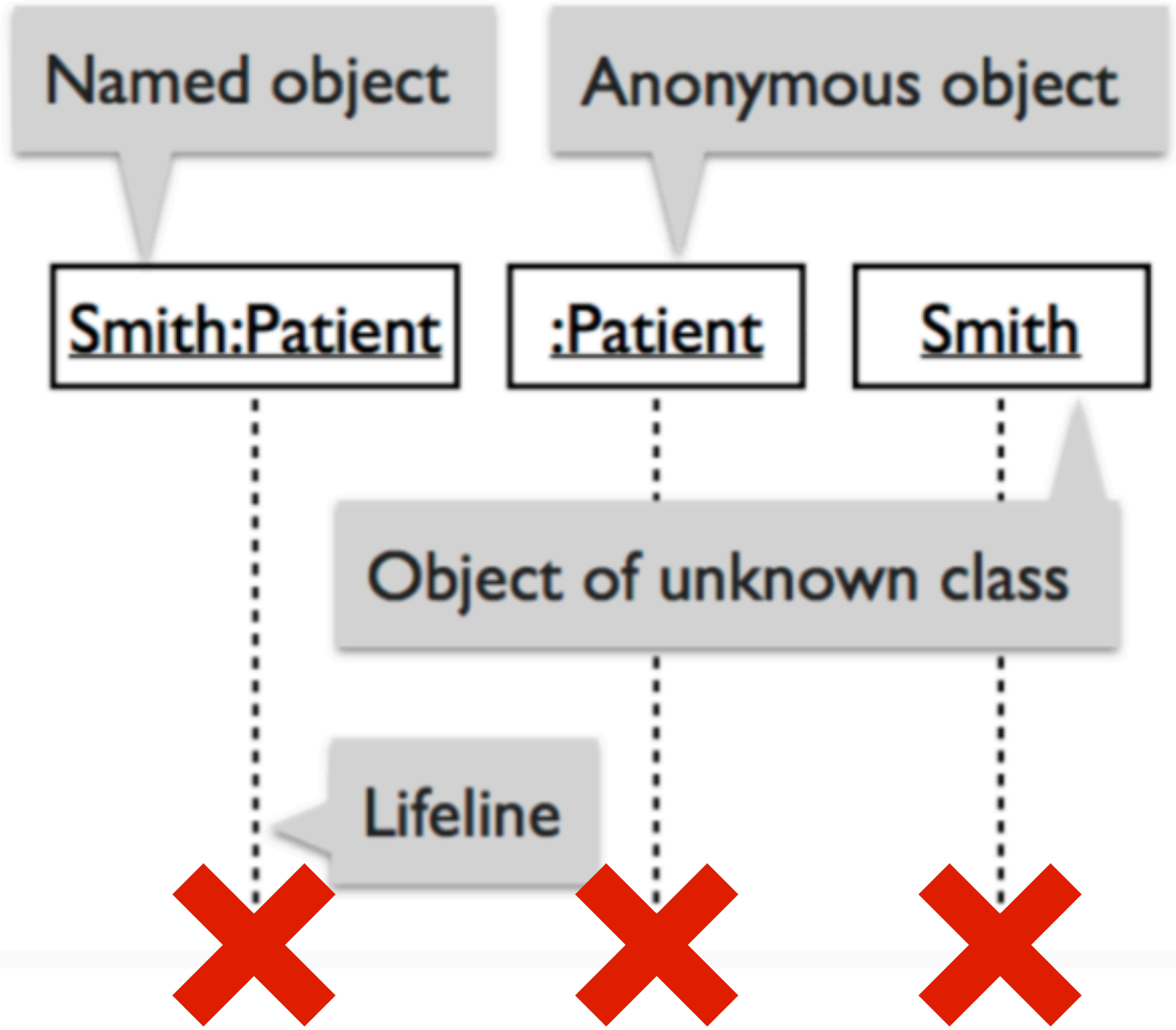
- Context models illustrate the operational context of the system and other systems

- Show what is ***inside*** and ***outside*** of the boundary of the system - Pizza Ordering system being developed shown relative to other systems



Writing correct Sequence Diagrams

- There are various ways of representing an object (depending on what is known)
- The dashed vertical line was the lifetime of the object
- You can terminate the lifetime with a cross
- This means the object is no longer used in the system, not that the patient died.



CS261
Software Engineering

Revision Lecture
Topic 3: Design Patterns

James Archbold
James.Archbold@warwick.ac.uk

In association with Deutsche Bank

Topic 3: Software Design

- Question 3 focuses on software design
- This includes different system patterns and understand how to use them
- That includes the various advantages and disadvantages
- You may have to draw diagrams, or provide a contextual use for a particular pattern.

Question 3A:

(a) What are the advantages to architectural design patterns? [3]

- Increase ease of re-use
- Formalise a good design practice
- Tried and tested approaches
- Better conceptualises solution for customer
- Supports design decisions
- Helps to manage risk
- Allows better costing.

Note what the question asks for - architectural, not software.

Pay attention!

Question 3B

(b) Identify, and justify, an appropriate architectural design pattern for the video streaming service discussed in Question 2. You should include a diagram of your chosen pattern. [5]

This is another ‘open’ question. There are a couple of acceptable answers. First, let’s discuss which answers are *not* appropriate:

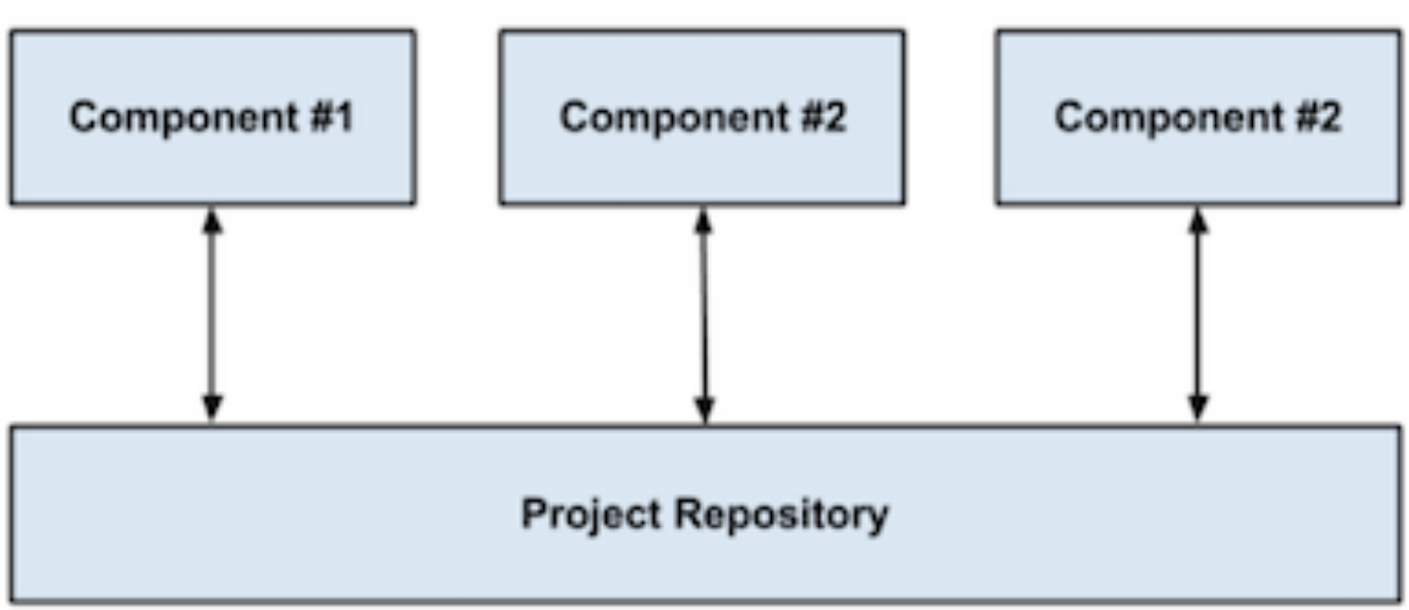
There is a fairly flat architecture - every user has the same functionality and so there is little need for lots of checkpoints. Thus, the layered architecture is not appropriate.

Pipe and filter could be appropriate, but we have to assume every user would upload the same type of video. With many different mobile devices using the app, this is a bold assumption. There are better patterns for this application.

Question 3B

(b) Identify, and justify, an appropriate architectural design pattern for the video streaming service discussed in Question 2. You should include a diagram of your chosen pattern. [5]

The repository pattern is appropriate, due to the app revolving around a large amount of videos that every other aspect interacts with. Outside of the videoclips, other components can generally be developed independently, and it's important to prioritise data management, due to the core aspects of the application focusing on the videos.

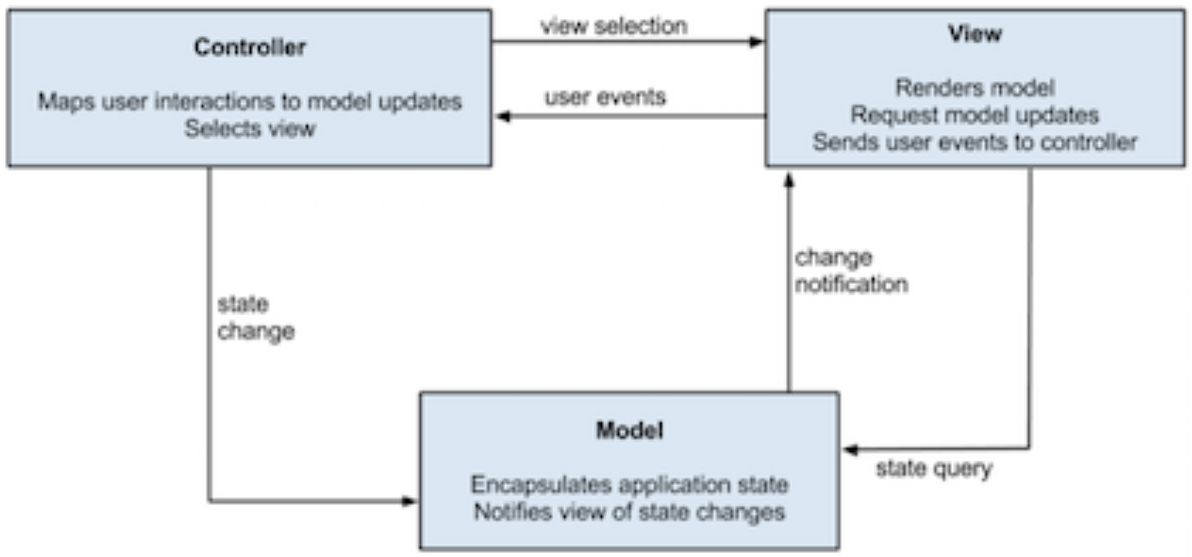


Question 3B

(b) Identify, and justify, an appropriate architectural design pattern for the video streaming service discussed in Question 2. You should include a diagram of your chosen pattern. [5]

The Model View Controller is appropriate, as it focuses on user facing operations, and this application is heavily user oriented. User's query the model about videos, and the controller maps their interactions to model updates, including subscribers and playlists.

The view would be a user's page, where they could upload/share/view videos
The model is the server that keeps track of video information and shows users when others have shared with them.
The controller handles user's uploading videos, editing videos and wishing to share videos, updating the information stored in the model.



Question 3B

(b) Identify, and justify, an appropriate architectural design pattern for the video streaming service discussed in Question 2. You should include a diagram of your chosen pattern. [5]

Your diagram should be labelled in reference to the system in question 2. These questions require thinking about common issues in software, such as compatibility and usability.

Think carefully about the system and issues involved in it.

Question 3C

(c) Contrast your chosen design pattern in Part B with another design pattern of your choice. Again, you should include a diagram and discuss the advantages and disadvantages of both when compared to each other. [5]

This question depends on your answer to B and the second pattern you pick.

We'll use this as an opportunity to go over the advantages and disadvantages of each pattern.

Question 3C

Layered Architecture: Structures system into layers that provide services to layers above it

Layers can be assigned to different development team members

Layers can be easily swapped out

Separation of functionality is hard and requires trust in developers

Can impact performance in the final code

Repository Architecture: Centralised repository stores all data in the system and handles all interactions.

Components can be independent

All data can be managed consistently as it is stored in the same place

Single point of failure

Inefficient as all requests go through repository

Scaling up is difficult

Model-View-Controller Architecture:
Model: manages the system data and associated operations
View: defines how the data is presented to the user
Controller: manages user interaction and passes these interactions to the view and model

Separates components of implementation, in particular user-facing components

Introduces a lot of complexity

Pipe and Filter Architecture: Discrete processing components filter data as it flowed down a linear pathway

Easy to understand

Matches the structure of many applications

Standardised data format required

Each component must stick to a rigorous standard, making modification difficult

Question 3D

(d) Software design patterns are used during the actual implementation of a system. Identify the key properties of a design pattern, and how they are demonstrated by the Factory design pattern. [12]

The key properties of a design pattern are as follows:

1. A standard solution to a common programming problem
2. A technique for making code more flexible by making it meet certain criteria
3. A design or implementation structure that achieves a particular purpose
4. A high-level programming idiom
5. Shorthand for describing certain aspects of program organisation
6. Connections among program components
7. That shape of an object diagram or object model

This is only half the question - we now have to link the Factory design pattern to each of these key properties.

Question 3D

The key properties of a design pattern are as follows:

1. A standard solution to a common programming problem
   - *In many situations, it is necessary to create many different objects of the same type, and the factory pattern offers a way to automate this*
2. A technique for making code more flexible by making it meet certain criteria
   - *Using factories cuts down on hard coded object creation.*
3. A design or implementation structure that achieves a particular purpose
   - *Factories generate default objects through the use of a method, which can be called in the code when a object is needed, as opposed to repeating object creation code.*
4. A high-level programming idiom is the idiom.
5. Shorthand for describing certain aspects of program organisation
   - *The term factory abstracts how this pattern creates many similar objects very easily.*
6. Connections among program components
7. That shape of an object diagram or object model
   - *Factory methods can be extracted into objects, and many involve many other classes*

Overview

- We've discussed architectural design patterns in-depth thanks to the questions.
- We need to make sure we cover the software design patterns as well.
- Make sure you're familiar with their purpose, and their advantages and disadvantages.
- Don't forget patterns are of different types - creational, structural, behavioural.

What do they need?

Design patterns have four essential elements:

A name that is a meaningful reference to the pattern

A description of the problem area that explains when the pattern may be applied

A solution description of the parts of the design solution, their relationships and their responsibilities

A statement of the consequences - results and trade off - of applying this pattern

Architectural Design

Architectural Design is concerned with understanding how a system should be organised

Conceptual Integrity
Conceptual vision, separated from implementation, and hard to change

Quality Driven
Embraces quality attributes, e.g. is fault tolerance important? Should it be easily maintainable? Is backwards compatibility important?

Recurring Styles
Does it adopt architectures seen before?

Separation of concerns
Can we reduce the complexity?

Supports design decisions, helps manage risk, allows better costing, conceptualises solution for customer

Architectural Design

- Often represented using box and line diagrams
  - Used as a basis for discussion about the system design and a way of documenting a proposed architecture
- Two main uses:
  - A way of facilitating discussion about the system design - High-level view is useful for stakeholder communication. The key components are identified so roles can be assigned for development.
  - A way of documenting an architecture that has been designed - Produce a complete system model that shows the different components in the system and their relationship.

Prototypes

- The prototype pattern provides another way of constructing objects of arbitrary types
- Rather than passing a BicycleFactory object, a Bicycle object is passed in
- Then its *clone* method is used to make copies of the given object

```
class Bicycle {

  Object clone() { ... }
}

class Frame {
  Object clone() { ... }
}

class Wheel {
  Object clone() { ... }
}
```

```
class Race {

  Bicycle bproto;

  // constructor
  Race(Bicycle bproto) {
    this.bproto = bproto;
  }

  Race createRace() {
    Bicycle bike1 = (Bicycle) bproto.clone();
    Bicycle bike2 = (Bicycle) bproto.clone();
    ...
```

Essentially, each object is a factory, making objects just like itself

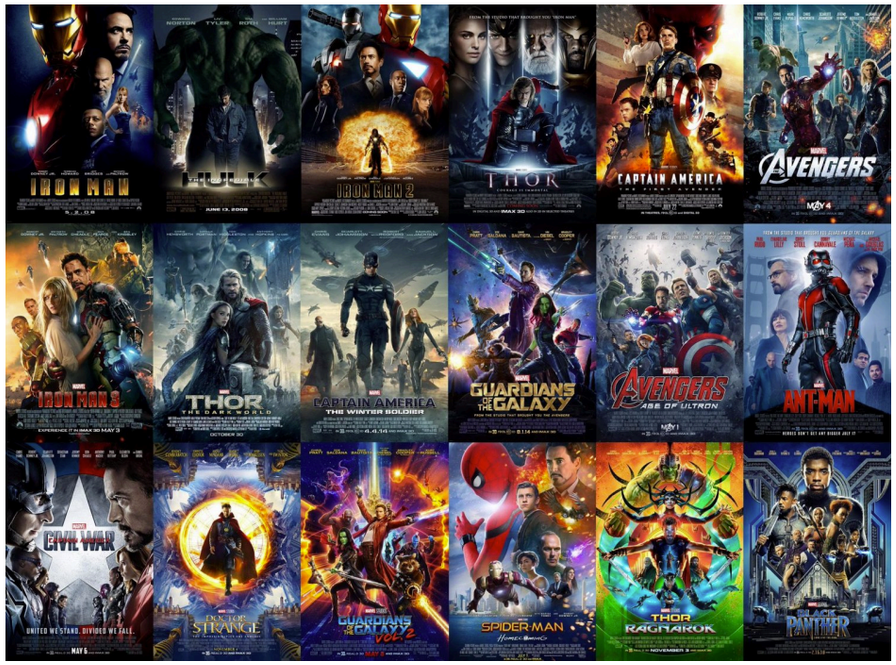
Builders

- The Builder pattern is a slightly different approach to creation, that helps with this specific problem.
- We extract the construction of an object into a set of methods, called builders.
- Object creation is executed in a series of steps - we call only the builders that we need.
- If we want to make a castle - we call our builder that makes the walls out of stone, adds a moat and installs a throne
  - Each of these sub-steps is a different method that could be called by any builder
  - We might have these steps in an abstract class Builder, then have the concrete classes for each type of object we might like to make

Proxy Pattern

Allow us to create placeholders for other objects - often by adding another level of indirection

We may wish to reference an entity (film, image, large database, etc.) without instantiating it
Embedded web-content, cost of streaming, image rights, copyright jurisdiction



In some cases, we might want to ‘load on demand’; the implementation solution is to use a **proxy**

Proxies have wider use:

1. *Remote Proxy* - encoding and decoding requests to a remote object
2. *Virtual Proxy* - similar to 1, but using caching
3. *Protection Proxy* - ensuring access rights before request is enacted

Decorator Pattern

Allow you to add new behaviour to objects at runtime

E.g. visually decorating a graphics component based on the device type
Adding stylised borders, scrolling, zooming etc.

Could add functionality with subclasses (inheritance), this can end up with very large class hierarchies

Decorators allow us to do this more transparently.

Also called wrappers, as they keep the original object and wrap it with additional functionality.

Adaptor Pattern

Allows output from one object to be used by another

We may wish to change the data type we're operating in, in order to properly use another framework

If we update the base class, we might need to make sweeping changes across the entire project.

This could lead to errors and possibly break many other functions.

Adaptors let us convert the data format we're currently operating in to allow us to use other services.

The Flyweight Pattern

Using the flyweight pattern allows us to get more objects in our memory.

Often, we may have many objects that share similar properties.

Some of these properties may be fairly large in size, and if we have a lot of these objects in our memory, we're taxing our resources.

Instead, why don't we just hold one copy of these properties, that all objects can reference?

Iterator Patterns

Traverse a container in order to access the containers elements

Useful as elements are accessed without exposing the container's data structure

New traversal operations can be implemented without changing the interface

This is so ubiquitous in Java that the Iterator interface is defined as part of the java.utils class

```
List users = new LinkedList();
for (Iterator it = users.iterator(); it.hasNext();) {
    User u = (User) it.next();
    ...
}
```

Java data structures such as Array.Map and Set already have iterator patterns implemented

You can of course define your own structures (e.g C.V.s) and define your own iterators over that

Observer Patterns

Allows an object's dependents to be notified automatically if state changes are made

If there is a timetable change, for example, then all students can be notified of this

Note that this can work in a **push model** (timetable sends notifications) or a **pull model** (students periodically ask for updates)

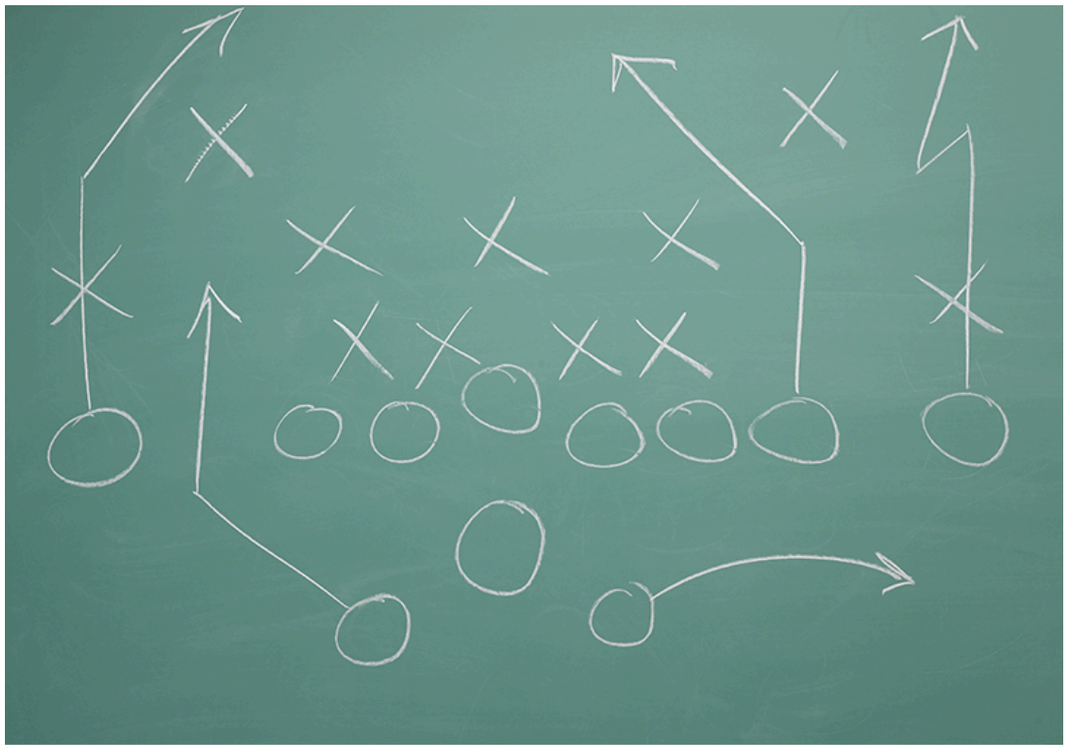
Often referred to as Producer / Consumer or Publish / Subscribe

Memento Pattern

- These issues all revolve around encapsulation
- We're trying to have one object invade the private space of another
- In the memento pattern we make an object responsible for saving its own internal state
- Essentially our originator object has some method that makes a snapshot of itself and stores it an object called a *memento*
- This snapshot can implement a limited interface so that the snapshots can be stored externally (usually in something called a **caretaker**)
- Then, when we want to, we access the mementos from the caretaker and pick which one to restore.

Strategy Pattern

- As usual, we can approach the problem a little differently
- We want a new object to be responsible for choosing our approach to a particular problem
- So, we will have a number of classes, each called a **strategy**, that we can select between
- Our original class now becomes a **context** object and it doesn't even know the details of each strategy
- The client wants the context to solve a problem, and then provides the strategy to use



CS261
Software Engineering

Revision Lecture
Topic 4: Testing and HCI

James Archbold
James.Archbold@warwick.ac.uk

In association with Deutsche Bank

Topic 4: Dependability, UI and Testing

- This focuses on the end of the development cycle
- Often, at this stage, we're reacting to user feedback or solving issues that have become apparent
- This means we need to evaluate and review a particular series of issues, providing the correct context given the application.

Question 4A

In Question 2, you were asked to model a new software system for sharing and creating short video clips on mobile, less than ten seconds long. The service has now been developed, and is called Quickshot.

(a) Using Quickshot as a basis for examples, describe common methods of system failure. [3]

- Making fake app names is where most of my effort goes when writing these exams.
- Note the question asks for examples contextualised through the app.

Question 4A

In Question 2, you were asked to model a new software system for sharing and creating short video clips on mobile, less than ten seconds long. The service has now been developed, and is called Quickshot.

(a) Using Quickshot as a basis for examples, describe common methods of system failure. [3]

- Hardware failure is a result of design or manufacturing errors or components reaching the end of their natural life. For example, the server we store our videos on may breakdown.
- Software failure is a result of errors in specification, design or implementation. For example too many users causing an overflow of userIDs, or videos being limited by file size, meaning lower quality videos can be longer than the specified length.
- Operation failure - An error in the usage of the system. For example, a user deletes all of their videos.

Question 4B

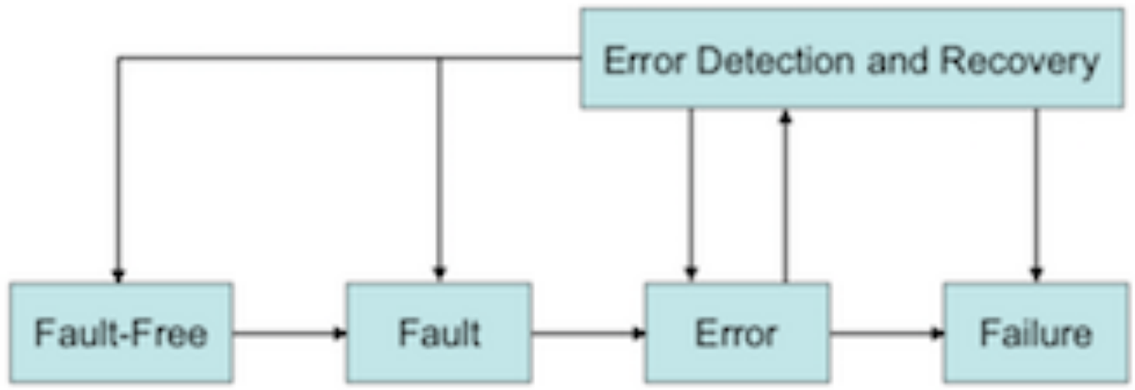
(b) Discuss methods of how we might mitigate errors in software and recover. You should include diagrams where useful and use examples based on Quickshot. [8]

- Again, we are asked to provide examples.
- Examples can be very open ended - but must be reasonable and related to some implementation of the discussed application.

Question 4B

(b) Discuss methods of how we might mitigate errors in software and recover. You should include diagrams where useful and use examples based on Quickshot. [8]

- There are three main methods of error mitigation:
  - Graceful Degradation - enable system to continue functioning with reduced functionality if a component fails. For example, we may prevent users uploading pre-made videos if an exploit has been found using video encoding.
  - Redundancy - Spare capacity that can come online if a part fails. For example, we may have older, slower servers that can be used if needed.
  - Diversity - Redundant components are different types, to prevent similar failure. For example, servers use different encryption protocols so data can still be safe if one is broken.
- What about the diagram? Well, we want to demonstrate how mitigating errors and recovery fit into the ‘fault-error-failure’ model.



Question 4C

(c) Discuss the differences between static testing and dynamic testing, including structural and functional testing. You should use examples in your explanation, based on Quickshot. [7]

- Examples again!
- Hopefully you know the different types of testing, and what they aim to actually test.

Question 4C

(c) Discuss the differences between static testing and dynamic testing, including structural and functional testing. You should use examples in your explanation, based on Quickshot. [7]

- Static testing is testing without execution, and aims to verify the code and ensures it meets the specification. It does not require the code base to be complete and be achieved through code review, inspections or event informally through pair programming.
    - For example, we may inspect the code of the User class to ensure data hiding is properly implemented.
- Dynamic testing involves executing the code with given test cases. Overall, it aims to validate the product and ensure it meets the needs of the customer. There are two types of dynamic testing, structural and functional testing.
    - Structural / White-box - Uses test cases derived from the control/data flow of the system. It aims to ensure the code is structured correctly and that code can be reached and executes as expected.
        - For example we may test branch adequacy by uploading a number of different file formats.
    - Functional / Black-box - Data is input into the system, and data is output and that outcome is compared to the expected outcome. It requires no view of the code.
        - For example we may have a set of unit tests that ensure a user creates a unique username and a sufficiently complex password.

Question 4D

(d) What is User Acceptance Testing? Discuss its uses and benefits, and provide two 'user journeys' that may be used to test Quickshot. [7]

- A lot of people got this wrong last year.
- Remember the steps needed for a user journey!
- You need a **process** and an **expected outcome**
- Lots of people last year just listed the steps. That does not give us anything to evaluate!

Question 4D

(d) What is User Acceptance Testing? Discuss its uses and benefits, and provide two 'user journeys' that may be used to test Quickshot. [7]

- UAT is beta testing, and ensures the user accepts the solution and that the solution meets business practices.
- It ensures the software is ready for service, avoids expensive failures, helps ensure repeat business, and helps to build confidence for the developer and client.
- Requires access to a subject-matter expert, ideally the client, and is executed against scenarios.

Question 4D

(d) What is User Acceptance Testing? Discuss its uses and benefits, and provide two 'user journeys' that may be used to test Quickshot. [7]

- User Journey 1:
  - Expected outcome: User 1 now subscribed to user 2.
  - Process:
    - Log in to system as user 1
    - Search for user 2 in search bar
    - Click 'subscribe'
- User Journey 2:
  - Expected outcome: Video added to user's library
  - Process:
    - Click 'upload video' button
    - Browse phone gallery for video
    - Select video
    - Press 'upload' button

Overview

- We’ve essentially covered the entire testing section
- However, we should also expect questions about HCI, dependability and possibly release management.
- So, let’s quickly review these three sections

Attributes of Dependability

Availability - The likeliness that a service provided by a computer system is ready for use when invoked

Reliability - A measure of how likely the system is to provide its designated service for a specified period of time

Safety - The extent to which a system can operate without damaging or endangering its environment

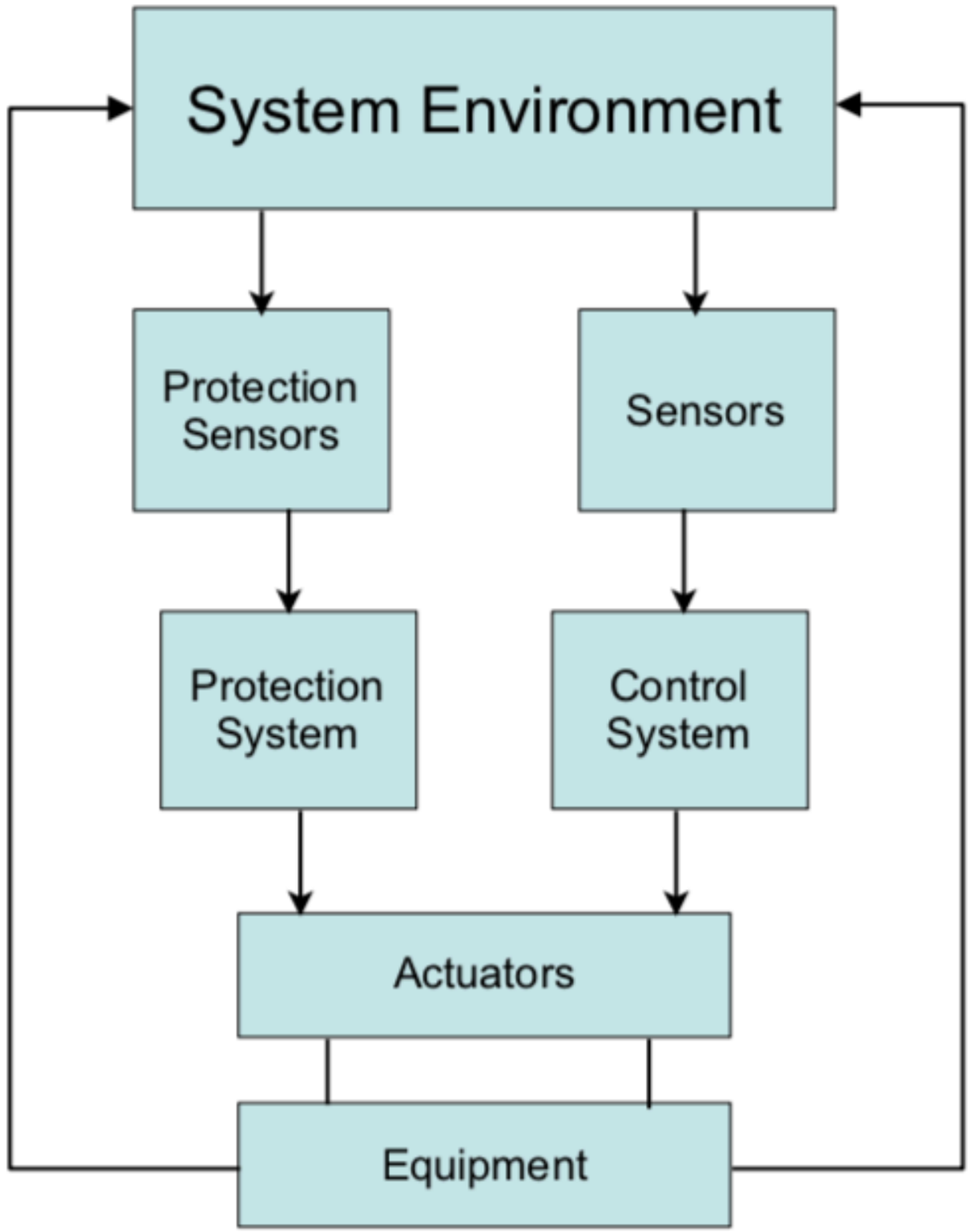
Confidentiality - Concerned with the nondisclosure of undue information to unauthorised entities

Integrity - The capacity of a computer system to ensure the absence of improper alterations with regard to the withholding modification or deletion of information

Maintainability - Defined as a function of time representing the probability that a failed computer system will be repaired in *t* time or less

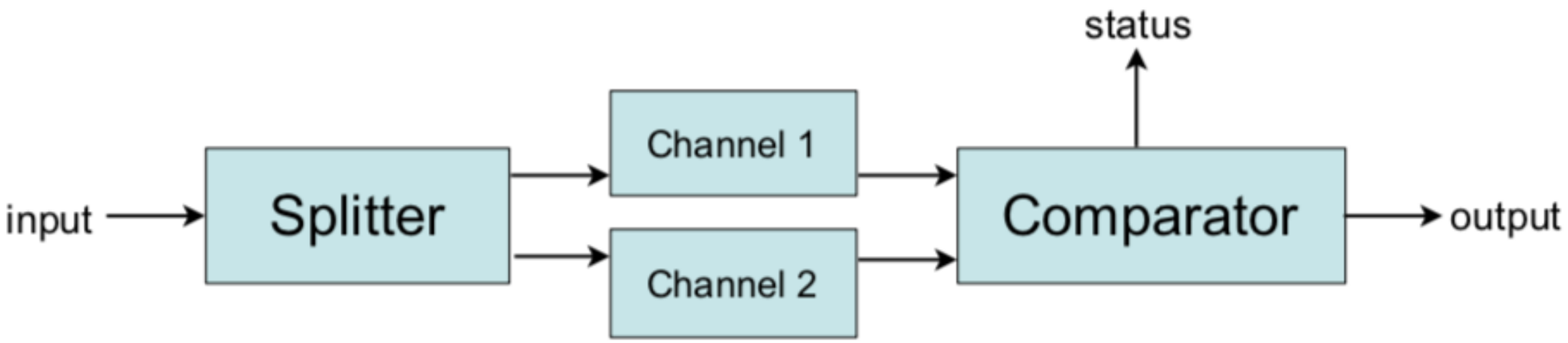
Dependable System Architectures

- Protection Systems
  - A specialised system associated with some other system (usually a control system)
  - Monitors the control system, equipment and the environment
  - Performs some action should a fault be detected
  - Moves the system to a safe state once a problem is detected (shutdown)



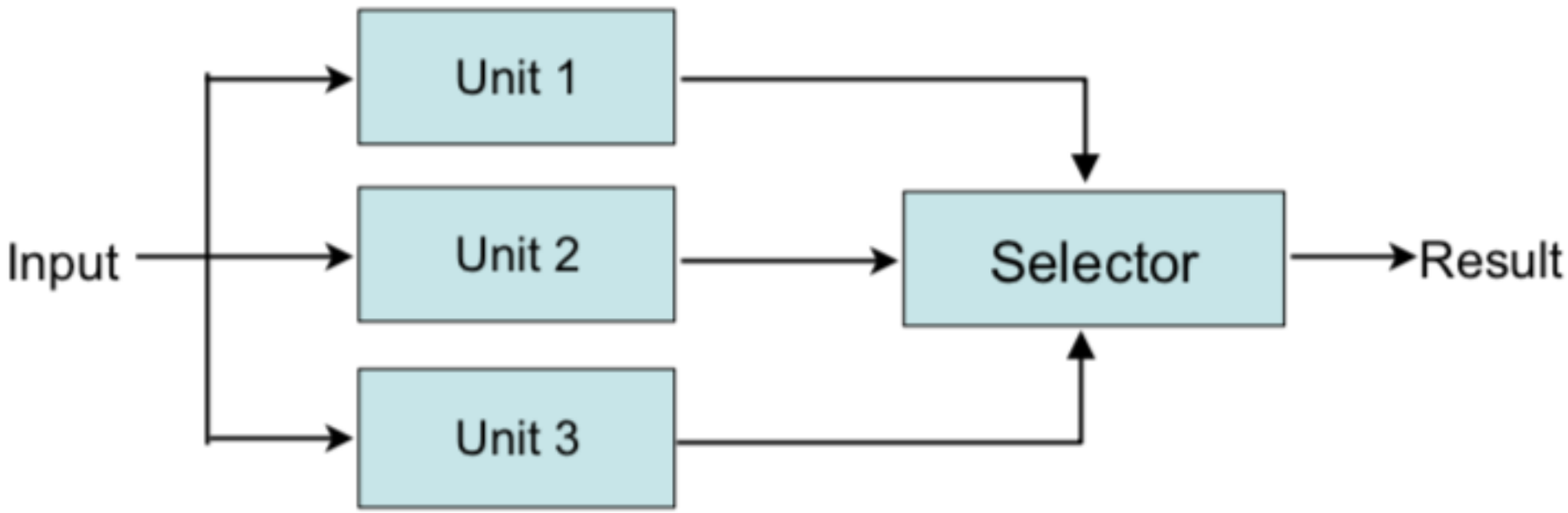
Dependable System Architectures

- Self-monitoring Architectures
  - An architecture designed to monitor its own operation and take action if a problem is detected
  - Computations carried out on separate channels; outputs then compared
  - If the outputs are identical and produced at the same time, the system is operating correctly, if different then a failure is judged to have occurred
  - Hardware and software in each channel should be diverse



Dependable System Architectures

- N-version Programming
  - Multiple replicated hardware units
  - Output from all units is passed to a selector which chooses based on the majority decision
  - When a failure occurs, the units output is ignored until it is fixed
  - Does not cater for same design flaw affecting the output of all units
  - E.g. Triple Module Redundancy (TMR)



Traits for HCI

- Human traits that underpin HCI:
  - Attention - cognitive process of **selectively concentrating** on aspect of the environment **while ignoring others**
  - Memory - episodic vs semantic memory
  - Cognition - Norman's human action cycle, Gestalt Laws of Perceptual Organisation
  - Affordances - what we see as the behaviour of a system or object - what it provides to us

Neilsons Usability Principles

1. Visibility of system status: Keep users informed of status by providing appropriate feedback

2. Match between system and real-world: Use the user's language rather than system terms

3. User control and freedom: Provide escape routes for users ('undo' buttons)

4. Consistency and standards: Avoid making users wonder if different words mean the same thing

5. Help users recognise and recover from errors: Use natural language to describe the error and provide a potential solution

Neilsons Usability Principles

6. Error prevention: Prevent users from making mistakes wherever possible

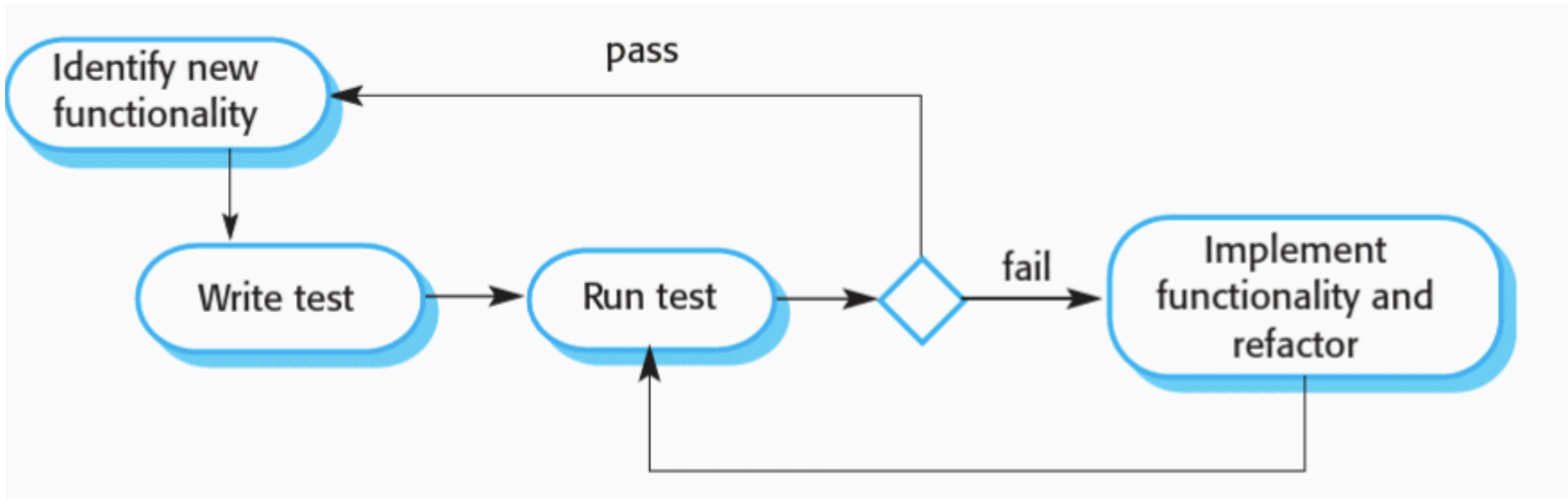
7. Recognition rather than recall: Make objects, actions and options visible rather than force the user to remember how to find them.

8. Flexibility and efficiency of use: Provide accelerators that allow experts to do things faster

9. Aesthetic and minimalist design: Avoid using information that is not needed, keep it simple

10. Help and documentation: Provide searchable information with concrete solutions, write as if for a 13 year old

TDD Process



1. Identify a small, implementable, functional increment.
2. Write a test that is ideally automated.
3. Run this test, along with all other tests, it will fail.
    3.1. It is important to run the test before implementing the function to see the test fail, this shows it is adding a new test to the system.
4. Implement your functionality and re-run the test.
    4.1. This may include refactoring old code.
5. Once all tests pass, you can move onto the next increment.

Conclusion

- The 2019 paper was quite bookwork based - this will not be the case for your paper
- However, many of the tips and discussion points here are still important
- In particular - you will often need to provide examples or properly contextualise your answers
- Don’t forget you must answer all the questions
- Read the questions carefully!
- And of course… good luck!