Lecture 1 - Use Cases

Use Cases: Text stories of some actor using a system to meet goals, aims to emphasize the user's goals and perspective. Should cover both success and failure scenarios.

System under Discussion (SuD): The system that we are attempting to model or explain

Actor: The person interacting with the system, involves behaviour and can be a person, a computer system or an organisation.

Scenario: Also known as a case instance, explains a specific sequence of actions and interactions between actors and the SuD.

Use Case: A collection of related success and failure scenarios that describe an actor using the SuD to support a goal.

> Levels of Detail:
> Brief: Paragraph summary of the use case. Includes success scenario only.
> Casual: Paragraph summary of the use case, but includes both success and alternate scenarios.
> Fully Dressed: Step by step rundown of the use case. Can also include alternate scenarios using 1a

Primary Actor: Has user goals fulfilled through using the SuD. User goals drives the use case.

Supporting Actor: Provides a service or extra information to the SuD to clarify external interfaces and protcols, for example an automated payment authorization service. Generally would be a computer system, but can also be an organization or a person as well.

Offstage Actor: has an interest in the behaviour of the use case, but is not primary or supporting (eg. a government tax agency). Ensures that all interests are identified or satisfied.

Use Case Model: A mode of the system's functionality and enviroment. Primaru includes the set of all written use cases, and optionally can include a UML use case diagram.

UML Use Case Diagram: Gives a context diagram of a system and its environment, displays the name of use cases, actors and their relationships.

Boss Test: Determines if a use case is considered *useful* or not. If you boss asks 'what have you been doing all day?' and you respond with your use case, will your boss be happy? More generally, would you consider this use case to be useful?

Elementary Business Process (EBP) Test: A task performed by one person in one place at one time, in response to a business event and should add measurable business value and leaves the data in a consistent state.

Size Test: A task should seldom be a single step, and should not be several pages long or too complicated. Ideally should be a few steps to test a case.

Lecture 2 - Domain Models and System Sequence Diagrams

Object-oriented Analysis: A process of creating an description of the domain from the perspective of objects.

Analysis: Investigation of the problem and requirements.

Domain Models: Identifies the concepts, attributes and associations that are considered notheworthy in the problem domain. More formally, it is a representation of real-situation conceptual classes.

- It is different to a software object (may not be tangible)
- Only outlines the *noteworthy* concepts or objects.
- It is one of OO artifacts (something that we create)
- It is also visually represented using UML

Unified Process Domain Model: Also the same as the UP Business Object Model, it focuses on explaining things and objects important to a business domain.

Domain Class Diagram: Provides a conceptual perspective to show conceptual classes, their attributes as well as their associations or relationships with other classes

Conceptual Class: Can be consider in terms of its symbol, intension and extension. (Can be a thing idea or an object).

Symbol: Words or images representing a conceptual class.
Intension: Definition of a conceptual class (eg. description of the class/object)
Extension: The set of examples to which the conceptual class applies.

Identifying Conceptual Classes:

Use Existing Models: Can use existing models for common domain problems. Eg. finance
Use Category Lists: Provides examples of catergories, found in references and textbooks.
Identify noun phrases: Identify the nouns in a passage, although it is not as commonly used as the other two options since language can be ambiguous.

Association: A relationship between classes that indicate a meaningful or interesting connection. We tend to add multiplicity and an association name to a relationship.

Attribute: A logical data value of an object. Should be included into a conceptual class when the requirements imply a need to remember a piece of information. Eg. 'Enters item number' or 'Presents item price'

Domain Model: The domain model ends up being a *visual dictionary* of the noteworthy abstractions, domain vocabulary and information content of the domain.

System Sequence Diagrams (SSD): A visualisation of the system events that external actors generate, models the order of the events and possible inter-system events.

Actor: Person engaging with the system
System: Presented as a black box, we can send and recieve events without acknowledging how the system would implement said event.
Event: A brief message outlining what is happening. Can contain parameters if necessary. Should be worded generally rather than mentioning specific implementations. Eg. 'Tap card' and 'Enter card', here 'Enter' would be a better event as it is more general.
Response: A return value associated with the previous message. From the system.
Loop: Can repeat events, represented by a box over relevant events.

Dynamic Context: Captured using SSD. Shows how an actor and system can interact with each pther, as opposed to a Use Case Diagram, which limits the interaction to user-goal level cases.

Inter-system Events: SSD can also illustrate interactions between systems, for example a POS and an external credit card verifier.

Summary: Both Domain models and Sequence System Diagrams are expressed at the abstract level of intention (no need for code yet). Domain models provide a static context of the system, becoming similar to a visual dictionary. Sequence System Diagrams provides a dynamic context of a system, by outlining *noteworthy* interactions between an actor and a system.

Lecture 3 - Design Class Diagrams & Design Sequence Diagrams

Static Design Model: A representation of software objects which define class names, attributes and method signatures (but not method bodies). Attributes also have a type (int, string etc).

· Uses UML notation (like domain models)

· Domain Models focuses on the conceptual perspective of the problem domain (eg. high level ideas and noteworthy concepts)

· Design Models focuses on an implementation perspective of software (eg. how software classes interact and their main role/purpose)

· Domain models inspire the design of software objects, and similar notation is used to reduce the representational gap between the domain as a concept and its representation in software

Responsiblity-Driven Design (RDD): Focuses on assigning responsibility to software objects, i.e what each object should achieve.

Responsiblity: The obligations or behaviours of an object in terms of its role.

Knowing: An object contains information about private, encapsulated data, or knows about related objects, or things it can derive/calculate.
Doing: An object has a method that 'does' something, like creating an object, doing a calculation, controlling other objects or initiating actions in other objects

Dynamic Design Models: A dynamic design model is a representation of how software objects interact via messages. Also illustrates a sequence based of time ordering of messages sent between software objects.

· System Sequence Diagram treats the system as a black box, and focuses on the interaction between actors and the system.

· Design Sequence Diagram illustrates the behaviours within the system, focusing on the interaction between software objects. One entry in the SSD could equate to multiple in the DSD depending on the objects required.

· Design Sequence Diagrams focuse more in realizing the responsiblities of software objects, and again has similar notation to SSDs.

Lecture 4 - GRASP Principles I

Responsibility Driven Design (RDD): Focuses on assigning responsibility to software objects.

- Knowing Responsibilities:
  - knowing about private encapsulated data
  - knowing about related objects
  - knowing about things it can derive or calculate
- Doing Responsibilities:
  - doing something itself, like creating an object or doing a calculation
  - initiating action in other objects
  - controlling or coordinating activities in other objects

GRASP: A set of patterns or principles of assigning responsibilities into an objects. Given a specific category of a problem, we can use GRASP to guide the assignment of responsibilities to objects.

Pattern: A *named* and *well-known* problem/solution pair that can be applied in new contexts. Alternatively, could be a recurring successful application of expertise in a particular domain

Creator: Deals with object creation.

Problem: Who should be responsible for creating a new instance of class A?

Solution: Assign class B responsibility to create instances of class A if one of these is true.

- If more than one of these classes are applicable, then the 'contains' takes precedence. Otherwise the class that has the most would be assigned.
  - B 'contains' or compositely aggregates A
  - B records A
  - B closely uses A
  - B has the initializing data for A

Contradictions: If creation of objects has significant complexity, then it would be advisable to delegate creation to a factory class.

Expert: Also known as Information Expert. Useful for understandibility, maintainability, and extendibility.

Problem:What is a general principle of assigning responsibilities to objects?

Solution: Assign the responsibility to object X if X has the information to fulfill that responsibility.

Contradictions: In some situations, the solution suggested by Expert is undesirable, usually because of problems in high dependencies.

Low Coupling: An *evaluative principle* that should be kept in mind during all design decisions. Useful to maintain a system and keep it efficient and reusable.

Problem: How to support low dependency, low change impact and increased reuse?

- Coupling: how strongly one element is connected to (has knowledge of, or relies on) other elements.
- Low Coupling: An element is not dependent on too many other elements.
- High Coupling: Depends on too many other elements, hard to understand in isolation, hard to reuse, one change impacts many classes (high impact change).

Solution: Assign the responsibility to object X that is the least coupled.

Contradictions: Being highly coupled with stable elements (eg. Standard Java Libraries) should not be a problem, since they are unlikely to be changed often.

High Cohesion: Also an evaluative principle that should be kept in mind during all design decisions. Used to keep a class focused, easing the comprehension of a design, simplify maintainability and enhancement.

Problem: How to keep objects focused, understandable, and manageable, and as a side effect, support low coupling?

- Functional Cohesion: How strongly related and focused the responsibilities of an element are
- Low Cohesion: A class has many unrelated things or does too much work, resulting in code being hard to comprehend, maintain and reuse.

Solution: Assign the responsibility to object X, so that cohesion remains high.

Contradictions: Low cohesion may be preferable in cases where we want the design to meet non-functional requirements. Eg. Calculations in one class can make operations faster.

Controller:

Problem: What first object beyond the UI layer receives and coordinates a system operation?

Solution: Assign the responsibility to a class representing one of:

- Facade Controller:The overall system, a root object, a device the software is running within, or a major subsystem.
- Use Case or Session Controller: A use case scenario that deals with the event.

Contradictions: If the controller is too bloated, low cohesion.
We would have to add in more controllers, or delegate information so that the controller only focuses on user input.

Lecture 5 - GRASP Principles II

Class Hierarchy: Also known as Generalization - Specialization, or Inheritance. Used when concepts are similar, and it is valuable to organize them.

Generalization: The activity of identifying commonality among concepts:

· Define the superclass (general concept)

· Define relationships with subclasses (specialized concepts)

· Conceptual subclasses and superclasses are related in terms of set membership. Generally subclasses are a subset of superclasses.

Creating subclasses:

1. Subclass has additional attributes of interest
2. Subclass has additional associations of interest
3. Subclass is operated on, reacted to, or manipulated differently to the other classes in noteworthy ways

Modelling Guidelines:

1. Declare superclasses abstract
2. Append the superclass name to the subclass

Polymorphism: Goes with generalization, assigning responsibility to subclasses for implementing different behaviours.

Problem: How to handle alternative behaviour based on types?
Problem: How to create pulggable software components?

Solution: When related alternatives or behaviours vary by type (class), assign responsibility for the behaviour - using polymorphic operations - to the types for which the behaviour varies.

Contradictions: If the system is unlikely to have a variation in the future, then using polymorphism would be exerting extra effort.

Pure Fabriaction: Adding a extra class to improve cohesion that doesn't actually exist.

Problem: What object should have responsibility when you do not want to violate high cohesion and low coupling, but solutions offered by other patterns are no appropriate.

Solution: Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept.

Contradictions: Pure fabrication can end up being overused, having too many objects where the objects has one responsibility is not a good a design.
Can also lead to high dependencies if too seperated.

Indirection: Goes with Pure Fabrication.

Problem: Where to assign a responsibility, to avoid direct coupling between two or more things?

Solution: Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled. This creates an indirection.
Eg. A $\rightarrow$ B, if something changes in class A, you have to change class B as well.
A $\rightarrow$ C $\rightarrow$ B, if something changes in class A, then you would just change class C, and leave B alone. This allows us to reuse class A and B.

Protected Variations:

Problem: How to design objects. subsystems, and systems so that the variations or the instability in these elements do not have an undiesirable impact on other elements?

· Points of change:

· Variation point: variations in existing system or requirements

· Evolution point: speculative variations that may arise in the future

Solution: Identify points of known or predicted variation/instability. Assign responsiblities to create a stable interface (point of access) around them.
Eg. a getTax() method may change depending on the system. As long as the adapter has a getTax() method, it can be reused.

Contradiction: Similar to polymorphism, may not be required depending on the scope.

Lecture 6 - State Machine Diagrams

State Machine: A behaviour model that captures the dynamic behaviour of an object in terms of states, events and state transitions.

- State: the condition of an object at a moment in time (circle or box)
- Event: a significant or noteworthy occurence that affects the object to change a state (label on arrow)
- Transition: a directed relationship between two states such that an event can cause the object to change from a prior state to the subsequent state (arrow)

State-dependent object: Reacts differently to events depending on the object's current state.

State-independent object: Reacts to events (of interest) the same way, regardless of the object's state.

State-independent wrt an event: Reacts to a specific event the same way.

Application of State Machine Diagrams:

- Guideline 1: Consider state machines for state-dependent objects with complex behaviour.
  I.e if an object changes according to a state, and if it's complex enough to be modelled.
- Guideline 2: Complex state-dependent classes are less common for business information systems, and more common in communcations and control applications.
  E.g. Better for UI, or systems with actions, rather than storing information.

Transiton Action: An actions that happens when a transition happens, but before a state change. Eg. if a button is clicked, then the action that could happen after is a loading screen, before it arrives at the next page.

Trigger: The event/action that happens in order to change states. Eg. Clicking a button.

Guard: A pre-condition to a transition, if the condition is not true when the trigger happens, then there will be no transition or change of state.

Nested state: A state allows nesting to contain substates. A substate inherits transitions of its superstate. At any point of a substate, it can call any transition of the superstate. Eg. If a superstate has action hang up, then the substate talking, dialing etc. can all hang up.

Choice Pseudostate: Evaluates the different paths of a guard. For example if a guard is true, moves to state A, otherwise if it is false it can move to state B.

Lecture 7 - GoF Part 1

Adapter:

- Problem: We have a system with incompatible interfaces, or want to provide a stable interface to similar components with different interfaces.
- Solution: Convert the interface of a component into another interface, through an adapter object. i.e for every interface, create an adapter interface that is compatiable with the class we want to use it with. Rather than changing the main class, we can change each adapter if the original classes change.
- Issues: Assigning creation of classes, which can be solved using Pure Fabrication.

Factory:

- Problem: Who should be responsible for object creation, when there are special considerations or complex creation logic, and we want to seperate the responsibilities to improve cohesion?
- Solution: Use Pure Fabrication to create a Factory object that handles creation.
- Advantages: Hides complicated logic, allows memory management, and improves cohesion.
- Issues: Who creates the factory, and how do we access it from multiple places? We can use the singleton pattern if global visibility is required.

Singleton:

- Problem: How to create an object with only one instance, and a global, single point of access.
- Solution: Define a static method of the class that returns the singleton. (getInstance)
- Considerations: Not all methods should be static, eg. if we want subclasses, of it the class we're using may not support the method.

Lecture 8 - GoF Part 2

Strategy:

- Problem: How to design for varying, but related algorithms and policies? How to design for the ability to change these algorithms or policies?
- Solution: Define each algorithm/policy/strategy in a seperate class with a common interface. (Same method name but different implementations)

Composite:

- Problem: How to treate a group or composition structure of objects the same way (polymorphically) as a non-composite (atomic) object? (ie. a tree and a leaf should both die, a tree consists of multiple leaves)
- Solution: Define classes for composite and atomic objects so they implement the same interface. Eg. Both the atomic and composite class inherit from the interface, but composite class also uses the interface (extra association).
- Issues: Combining components may result in having multiple methods but we need to specify the method we want to use.

Lecture 9 - GoF Part 3

Facade:

- Problem: Require a common interface to a disparate set of implementations within a subsystem. Eg. having slightly different implementation of the same subsytem.
- Solution: Define a single point of contact to the subsystem - a facade object that wraps the subsystem. The facade object presents a single, unified interface and is responsible for collaborating with the subsystem components. (Rather than attaching different classes to different classes within a subsystem - many to many, we can instead have a facade - many to one to many - which reduces the overall coupling. Changing a class within a subsystem also requires the class to be changed once in the facade as opposed to many times in the different classes)
- Considerations:
  A facade is different to a controller, as it provides a simpler interface of a complex subsystem for a client class. A controller handles user input.
  A facade is different to an adapter, as it wraps an entire system to a single object
  (subsystem $\rightarrow$ facade $\rightarrow$ clients).
  An adaptor wraps each API with a varying interface to produce a single interface
  (specific class $\rightarrow$ specific adapter class $\rightarrow$ generalised adapter class).

Decorator:

- Problem: How to dynamically add behaviour or state to individual objects at run-time without changing the interface presented to the client? (Optional embellishment of an existing class, not replace it)
- Solution: Encapsulating the original concrete object inside an abstract wrapper interface. Then let the decorators that contain the dynamic behaviours also inherit from this abstract interface. The decorators that contain dynamic behaviours also inherits from this abstract interface. The interface then uses recursive composition to allow an unlimited number of decorator 'layers' to be added to each core objects.
- Considerations: The decorator can only decorate one object, and it must exist to be able to be decorated. (Can't make a triangle red if no triangles exist). Also different to Strategy - decorator changes a skin of an object, strategy changes the implementation (guts)

Observer:

- Problem: How to have different 'subscriber' objects react to the changes or events of a 'publisher' object with low coupling?
- Solution: Define a 'subscriber' or 'listener' interface. Subscribers implement this interface and the publisher can dynamically register subscribers (add them to a list) and notify them when an event occurs (go through list and call a generic 'action' method that has the same name but different implementation across subscribers)
- Considerations: An interfaces allows different types of listeners to subscribe.

Lecture 10 - Architectural Analysis

Software Architecture: The large-scale organization of elements in a software system.

Architectureal Analysis: An activity to identify factors that will influence the architecture, understand their variability and priority and resolve them. Identify and resolve the system's non-functional requirements in the context of its functional requirements.

Architecturally signficant requirement: The requirement that can have a large impact on the design (especially when not considered at the beginning). Can also consider variation points and potential evolution points.

Architectural Analysis Steps:

1. Identify/analyse architectural factors: requirements with impact on the architecture (especially non-functional requirements)
2. For the architectureal factors, analyse alternatives and create solutions: architectural decisions (eg. remove requirement, customised solutions, hire an expect etc.)

Priorities:

1. Inflexible constraints
   eg. safety, security, legal compliance
2. Business goals
   eg. demo for clients, deadlines
3. Other goals
   eg. being extendible, like having a new release every 6 months

Architectural Factor Table: A documentation that records the influence of the factors, their priorities and their variability (immediate need for flexibility and future evolution)

1. Factor: What should be considered
2. Measures and quality scenarios: Ideal solutions to a situation
3. Variability (current flexibility and future evolution): How the system will be used in the future
4. Impact of factor(and its variability) on stakeholders, architecture and other factors: How important is this?
5. Priority for success
6. Difficulty or risk

Technical Memo: A documentation that records alternative solutions, decisions and influential factors, and motivations for the noteworthy issues and decisions

1. Issue
2. Solution summary
3. Factors
4. Solution
5. Motivation
6. Unresolved Issues
7. Alternatives considered

Architecural Analysis Summary:

- Concerns are related to non-functional requirements with awareness of business context, but also addresses functional requirements and their vairability
- Concerns involve system-level, large-scale, broad problems, with resolution involving large-scale, or functional design decisions
- Must address interdependencies and trade-offs (eg. security vs performance, or anything vs cost)
- Involves the generation and evaluation of alternatives

Logical Architecture: The large scale organisation of the software classes into packages, subsystems and layers

Layered Architecture: Coarse-grained grouping of classes, packages or subsystems that has a cohesive responsibility for a major aspect of the system. Eg. User interface or domain objects (sale class). UI may consist of many subsystems/logic classes however the main goal is to act as an access point for a user.

Strict layered Architecture: A layer that only calls upon the services of the layer directly below it (eg. a network protocol stack), which is uncommon in information systems

Relaxed layered Architecture: A higher layer can call upon several lower layers. Common in information systems.

Guidelines:

- Organise large-scale logical structure of system into distinct cohesive layers from high application specifics to low general services
  Maintain a seperation of concerns, eg. don't put application logic in the same layer as UI objects, each layer should have a cohesive purpose
- Collaboration and coupling should be from higher layer to lower layers. Avoid coupling from lower to higher layers.
- Don't use external resources as a bottom layer (can be ignored)

Implementation View Architecture: Utilises components and focuses on how to access each component and their relationships.

Component: A modular part of a system that encapsulates its contents is replaceable within its environment.

Component Diagram: Is concerned with how to implement the software system in a high level, defines behaviour in terms of provided and required interfaces, provides the initial architecture landscape for a system. A component can be a class, or external resource (eg. database)

Lecture 11 - Case Study and Software Process

Lessons:

- Client's perspective: Critical, but also need analysis to get the full picture
- Understanding of business requirements is also critical, missing elements can make the system unusable. At minimum should fufill the main business requirements
- Knowing the final prodict is the easy part, planning and executing an acceptable path is the real challenge. Designing helps to provide options.

Unified Process: An iterative and incremental software development process

Schedule Oriented Terms in Unified Processes: The development cycle consists of four main blocks: Inception, Elaboration/Iteration, Construction, Transition

Inception: Inception is the initial short step to establish a common vision and basic scope for the project.
Outcomes:

- Common vision and basic scope for the project
- Creation of a business case (addressing costs)
- Analysis of around 10% of use cases
- Analysis of critical non-functional requirements
- Preparation of the development environment
- Prototypes(maybe): to clarify requirements or technology questions
- Go or no go decision

Elaboration: The initial series of iterations for building the core architecture, resolving high-risk elements, defining more requirements and estimating the overall schedule and resources.

- Produce an executable architecture
- The majority of requirements are discovered and stabilized
- The major risks are mitigated or retired
- May create a domain model, design model, software architecture document, data model, use-case story board or UI prototypes
- Around 80% of the requirements should be reliably defined

Depending on the use case, it may require several *iterations* for the features to be planned. We can split this up into mini blocks in the elaboration process. Eg. Block 1, start with 20% of the use case, then Block 2 50% etc.

Iteration:
Requirements: Do the right thing (validation)
Design: Do the thing right (verification)

Designing Objects:

1. **Code:** From mental model to code. Designing while coding with an IDE which supports refactoring and other high-level operations.
2. **Draw then Code:** Drawing UML on a whiteboard or a UML CASE tool then switching to 1 (Coding).
3. **Only Draw:** Use a tool that generates code from diagrams.

Agile Modelling and Lightweight UML Drawing:

- Modelling with other developers
- Static and dynamic models (Class and interaction diagrams, can create several models in parallel)
- Hand draw or use whiteboards
- UML tools, use IDE integrations

Object Design Skills: The design of an object us more important than just knowing UML Notation. Eg. Principles of responsibility and design patterns

Summary: Inception establishes a common vision and basic scope for the project. Elaboration is where software modelling and design is performed. Translation requirements to design can be done iteratively.