Week 1 Unit Systems ...

Lab 1 Systems thinking

Objectives

- [ ] Modelling and Systems
- [ ] Development Process
- [ ] Requirements Engineering
- [ ] Business Analysis
- [ ] OO Systems Analysis
- [ ] Testing
- [ ] Software Design
- [ ] Architecture
- [ ] OO Detailed Design
- [ ] Design Patterns
- [ ] Implementation
- [ ] Refactoring

Notes

Why Software Engineering

What is Software Engineering?

Software engineering is an art and science. It's a collection of habits and attitudes.

It is the systematic procedures used for the analysis, design, implementation, testing and maintenance of software.

The term became commonly known in 1968 ass the title of a NATO Conference to discuss software issues, guidelines and best practices.

SE aims to solve the software crisis (failures due to increasing demands & low expectations).

it's important to have a systematic way for software development

Importance of software Engineering

Software Engineering sets apart hobbyist programmers from professional software developers.

It is the standardised structure and thorough approach to writing code.

Software Engineering treats the whole development process from start to finish in a formalised manner.

It allows us to deliver programs that meet a certain level of rigour (efficient , secure, viable).

Not everyone knows how to create valuable code that can be maintained and can adhere to a set of standards. Such as documentation.

Software engineering starts from the beginning from hitting a problem to deploying the software and maintaining it.

Software needs to be developed in time and budget and so we need to adhere to software development standards.

Systems and Modelling

Systems Thinking

Systems thinking is the Holistic approach to solving problems that focuses on how a system's components interrelate and change over time, within the context of the overall missions the system is designed to accomplish.

In a rubix cube the individual components don't achieve much on their own, but by putting them together and connecting them we develop a working System.

A system is an entity that is made up of interacting parts, together these parts and their connections make a whole with a purpose and produce a result.

A bicycle is a system because there are multiple components working together to achieve a purpose, but a pile of papers, despite being made up of components isn't. This is because they don't interact with each other.

System Characteristics

Systems have the following characteristics:

▸ Central objective

All components work together to achieve common goal.

▸ Integration

Components are put together to complete a system.

The wholeness of a system , how well do the components work with each other.

▸ Interaction

The system works as a result of components interacting with each other.

We need to describe and document these interactions

▸ Interdependence

A change to one component will affect other components.

High Cohesions - all systems need to be in a working state

Low Cohesion - We don't care about the state of an interracting object as much as its input.

▸ Organisation (hierarchical)

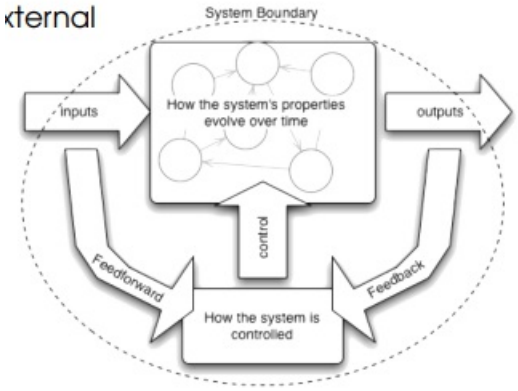
The right components in the right place, in the right time.

Expected Components in a system

We expect systems to have the following components

- Input , process and output
- Environment: internal and External
  - The environment is what describes the factors and circumstances under which the system works
  - The internal enverionment is the one which the system has control over
  - The external is the one it doesnt have control over
- Boundary
  - Separates the internal and external environment

    - The dotted line shown in the image below
- interface
    - How the user or system will interract with our system
- Feedback and Control
    - typically to regulate and maintain the system's behavior



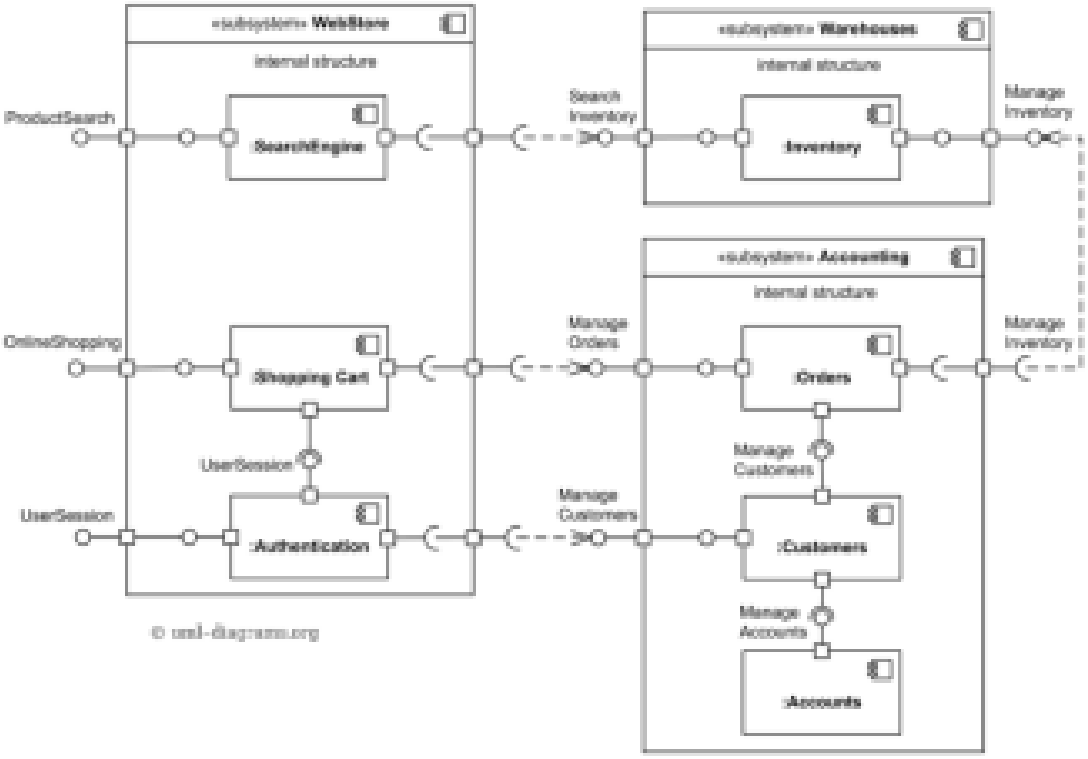
Hierarchy in Systems

Systems are expected to follow a hierarchy.

Complex systems are comprised of various subsystems

Subsystems are arranged in hierarchies and are integrated to achieve the common goal

Subsystems have their own boundaries, thus interfaces and environment



Online shopping UML component diagram example with three related subsystems - WebStore, Warehouse, and Accounting.

You can have a boundary that crosses around , the system can have one and each component can have one too meaning components can have their own internal and external environments.

Systems and External Environments

We expect systems to interact with users and remote other systems, via interface components.

The external (operational) environment affects the functioning of a system.

A system's function may be to change it's environment.

This can be physical or operational. A robot using cameras can analyze it's environment to pick a piece of paper for example.

External Environment can be both physical and organisational.

Emergent Properties / Behaviours

The complex relationships between the components in a system mean that a system is more than simply the sum of its parts.

Properties emerge once the system components are integrated.

Properties emerge as a result of interractions between components

There are 2 types of Emergent Properties:

1. Functional emergent properties are those that describe what the system is trying to achieve or what it is trying to do. For example, if we take the components of a bike and put them on the floor, we can't see the bike's functionality.
2. Non-Functional Properties relate to the behavior of a system, specifically how it performs rather than what it does. In the example of a bicycle, we can consider the safety of riding the bike as a non-functional property. They refer to what the system IS. it is secure , it is fast , it is heavy etc.

UML Component Diagrams

System Modelling

The world is full of systems and sets, with the majority of people only seeing the latter.

What is required to propose or improve and existing system?

- Use our component based vision to create a system's model out of a system
- Model it, no matter how crude the overall insight might be.

You cannot provide a solution to a problem if you don't understand the problem.

A systems model must have the right level of detail to define:

- The components that constitute the system.
- The types of relations between those components.
- How the components interrelate into the whole system to perform some function.
- How the whole system interacts with it's environment.

Ports are used to represent points of interaction between a component and the outside world. They group interfaces together.

Why are models useful

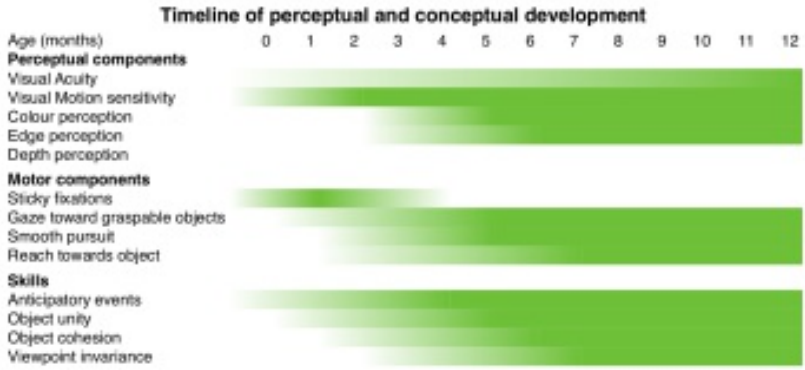
- Abstraction
    - Allows the modeller to focus on the most important features of a real-world situation.

The common mistake when modelling is by starting to build the solution and modelling simultaneously. We start by designing an Abstract model based on assumptions and we gather information as we model.

- Representation
    - Makes one's ideas into a more concrete artefact.
- Communication
    - Help sharing ideas and thoughts that are difficult to put into words
- Early Evaluation

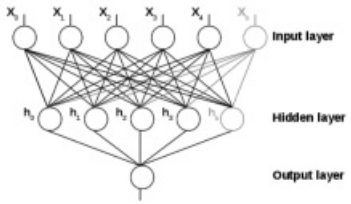
- No runtime failures, check if requirements are met.

Examples of Models

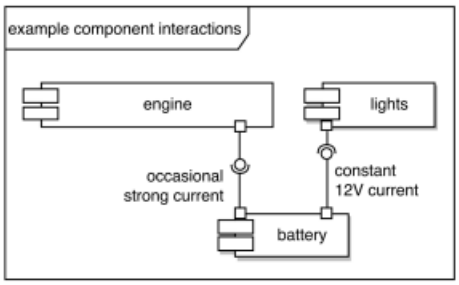


The model below is the OSI model which describes how network layers interract with each other

Neurolink Model



Circuit Model



Models over Modelled Systems

Models are better than model systems , why is it better to start drawing things at an abstract level than putting components together?

- A model is quicker and cheaper to build
- One can change and choose viewpoints and details to include or omit making it easier to analyze
- Some models can be used in simulated environments
- Models evolve, they don't need to be perfect and sometimes don't even make sense

Models in software Engineering

Software is an abstract entity , it's models are abstract too

Software is inheritably abstract and thus its models are too.

is program code a more detailed model?

models of software focus on certain aspects:

- Code organisation
- Runtime Entities and their responsibilities
- Overall data flow
- Messages among entities.

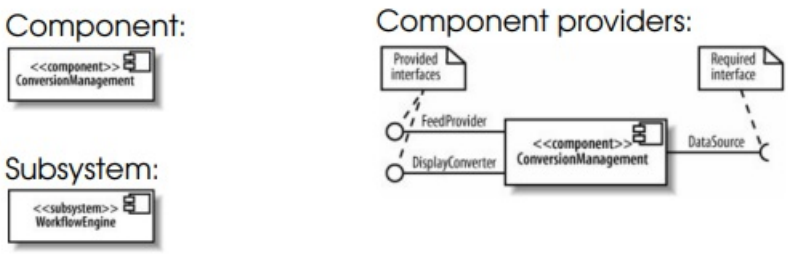
Unified Modelling Language (UML) provides all the tools necessary to produce models

UML Component diagrams Notation

Component diagrams help us plan out the high level pieces of a system to establish the architecture and manage complexity and dependencies amongst components.

Component diagram:

Components are made up of a name and stereotype (the type of that component).



Component diagrams help us plan out the high-level pieces of a system to establish the architecture and manage complexity and dependencies amongst components.

How components work together:

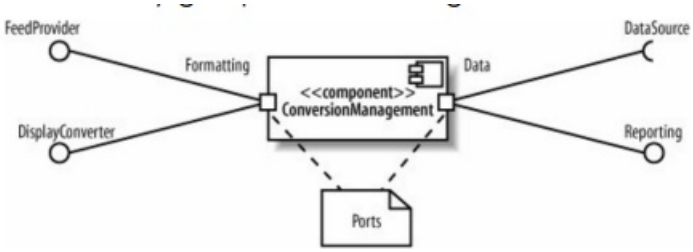
Two ocmmon ways t odepict how components work together:

Arrow dependecy:

ball Socket Connection:

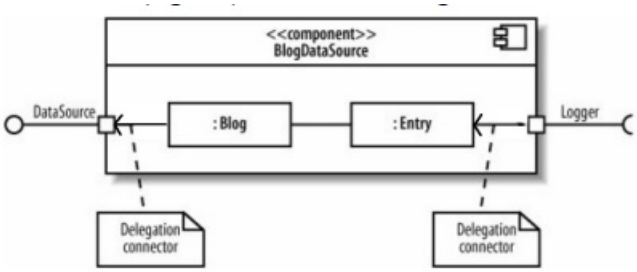
Ports:

Ports are used to depict points of interaction between a component and the outside world.



Ports usually group interfaces together.

One can depict the internal structure of a component and use delegation connectors to show the direction of traffic:



Circuit Example

The battery is dependant on the engine , the engine provides the service , as shown by the ball joint notation , the engine provides an " occasional strong current" and the battery receives the current , likewise as shown by the joint the battery then provides a service to the lights in the form of current.

Summary

Why Software Engineering

- **Definition**: Software engineering is a systematic approach to the analysis, design, implementation, testing, and maintenance of software.
- **Historical Context**: The term "Software Engineering" became well-known in 1968 during a NATO Conference aimed at addressing software-related issues, guidelines, and best practices.
- **Importance**: Software engineering distinguishes professional developers from hobbyist programmers. It emphasizes structured code, thoroughness, and adherence to standards, including documentation. It spans the entire development process, from problem-solving to deployment.

Systems and Modeling

- **Systems Thinking**: Systems thinking involves a holistic approach to problem-solving, focusing on how a system's components interact and change over time to achieve specific objectives.
- **System Characteristics**: Systems have central objectives, integration of components, interaction between components, interdependence of components, and organizational hierarchies.
- **Expected Components in a System**: A system typically consists of input, processing, and output components, internal and external environments, a boundary separating internal and external environments, interfaces, and feedback/control mechanisms.
- **Hierarchy in Systems**: Complex systems often comprise various subsystems organized in hierarchies, each with its own boundaries and environments.
- **Systems and External Environments**: Systems interact with users and other systems via interface components. The external environment can be physical or organizational and may influence the system's functioning.
- **Emergent Properties/Behaviors**: Systems exhibit emergent properties, which are characteristics that arise from interactions between components. These can be functional or non-functional properties.

UML Component Diagrams

- **System Modeling**: Creating a system model is essential to understand and propose solutions to problems. A model should define components, their relationships, interactions, and how the system interacts with its environment.
- **Benefits of Models**: Models provide abstraction, representation, facilitate communication, and enable early evaluation.
- **Examples of Models**: Examples include the OSI model for networking, the Neuralink model, and circuit models.
- **Models vs. Modelled Systems**: Models are often preferable to model systems because they are quicker and cheaper to

build, adaptable, and can be used in simulated environments.

- **Models in Software Engineering**: Software is inherently abstract, so models of software focus on aspects like code organization, runtime entities, data flow, and message passing. The Unified Modeling Language (UML) provides tools for software modeling.
- **UML Component Diagrams Notation**: Component diagrams are used to plan the high-level structure of a system and manage complexity and dependencies among components. They include components with names and stereotypes, arrows and ball-and-socket connections to depict how components work together, and ports to represent points of interaction.
- **Circuit Example**: The lecture notes provide an example of a circuit with components, dependencies, and the use of ports.

Summary

- **Emergent vs. Non-Emergent Properties**: Emergent properties are characteristics that arise when a system is running due to interactions between components. Non-emergent properties are observations that do not require the system to be running.
- Functional vs non-functional:
    - Functional emergent properties are characteristics that describe what a system is trying to achieve or do when operating. (main purpose or functionality).
    - Non-Functional emergent properties relate to the behaviour of a system, specifically how well it performs certain aspects rather than what it does.

Practice After

Certainly! Let's test your knowledge with some questions based on the lecture notes:

1. **What is the main objective of software engineering, and why is it important?**
2. **Define "emergent properties" in the context of systems. Provide an example of both functional and non-functional emergent properties.**
3. **What are the key components expected in a system, and what is the purpose of each component?**
4. **Explain the concept of "systems thinking" and why it is important in problem-solving.**
5. **In UML component diagrams, what is the purpose of ports, and how are they used to represent interactions between components?**
6. **Give an example of a real-world system that exhibits emergent properties. Describe one functional and one non-functional emergent property for this system.**
7. **Why is it better to start modeling a system at an abstract level before adding components?**
8. **What are the benefits of creating models in software engineering, and how do models differ from modelled systems?**
9. **In the context of systems and modeling, why is it important to consider the internal and external environment of a system?**
10. **Briefly explain the significance of the Unified Modeling Language (UML) in software engineering.**

Feel free to answer these questions, and I'll provide feedback and explanations as needed.

Week 2 Software Lifecycle

Lab 2

Objectives

- [ ] this lol

Notes

Software Development Lifecycle

What is the Software Development Lifecycle

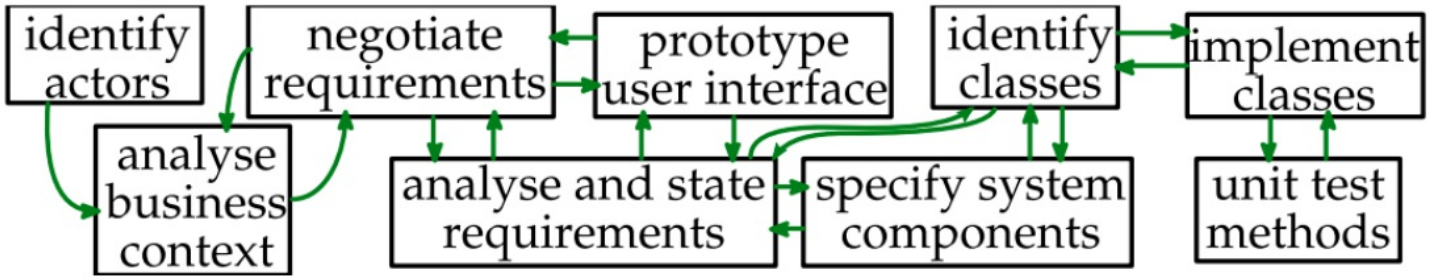
SDLC - Software Development process / Methodology is a sequence of project phases we can follow in order to create a software application.

*SDLC = activities required to develop software **AND** an approach to linking them together in a workflow.*

in the 1960's we had the computational power to apply solutions to complex problems and expertise , but we lacked structure to tackle these problems and apply solutions. SD allowed us to structure our work to develop correct software (meets stakeholder requirements) in a timely way and within budget.

Motivation

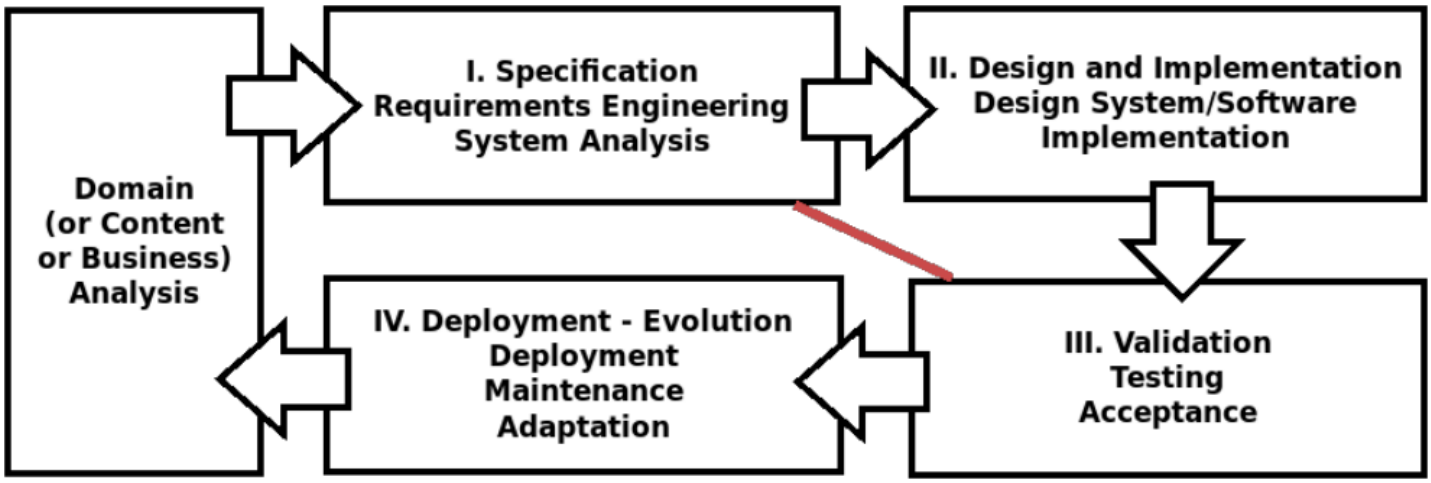
The need for a logical progression of development activities.



1. The actors are anyone who will use the software
    a. Stakeholders are anyone who will benefit from the software
2. Analysing business context is what the business domain is doing , discussing with experts,
3. Negotiating requirements helps us analyse business context in further detail
    a. This stage can repeat in iterations and cycle
4. When we have enough requirement understanding we can begin writing them down and working on a prototype (Designing the solutions and modelling the problem).
    a. Including the stakeholders here can help us get feedback to better tailor the software
    b. We start with abstract modelling and through iteration we develop the details
5. After fleshing out our model we can begin developing the required components and classes.
6. Implement classes
7. Unit tests
8. Specify System components
9. Analyse and state requirements

Logical Progression of Activities

Logical Progression is not necessarily time progression.



We start from the business or domain analysis and collects as much info as possible to begin modelling the problem.

We then move to the specification where we start listing the requirements and thinking of what the solution should look like.

This is also where we can start thinking of how the requirements will be tested.

Design and implementation is where we begin coding

Validation testing and acceptance

Deployment is where we have some software ready to show stakeholders.

Exercise - Example Activities

What do we need to do to develop an "intelligent cookbook app"? Assign activities to their category.

Business context	DB schemas
Specification	observe chefs working
Design+Implementation	update user stories
Validation	set up servers
Deployment+Evolution	regression tests

Answers:

- Business Context - Observe Chefs working
- Specification - update user stories
- Design + Implementation - DB Schemas
- Validation - Regression Tests

- Deployment + Evolution - Set up servers

Part of the very first step is understamd how the experts work , business context.

Updating user stories is working with requirements, writing them down, how will users and stakeholders interract with the system and what do they expect.

Details of Lifecycle phases

The details

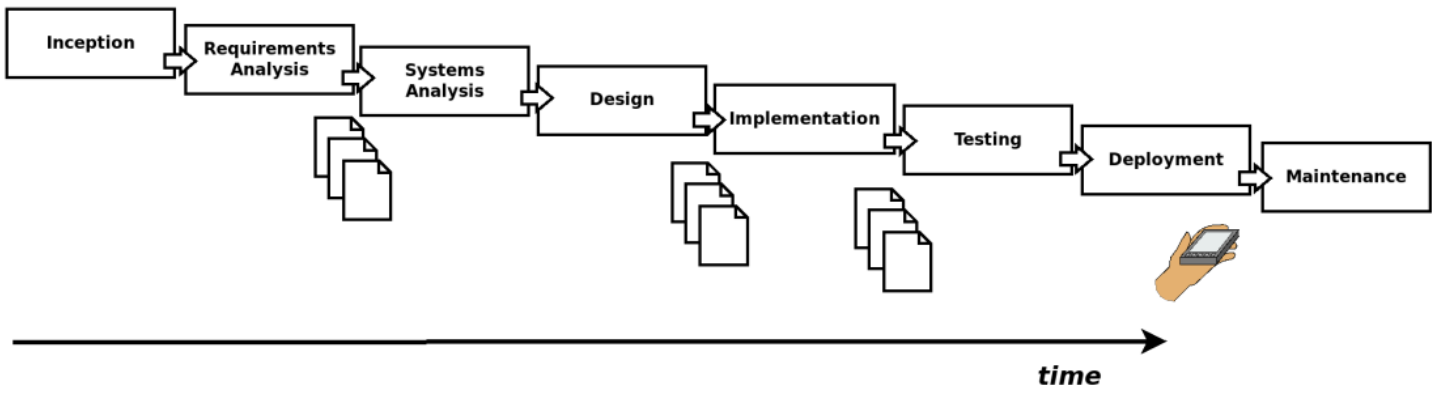
- Domain Analysis - Study what is already there, literature review, conversations with experts, what is there and what is not , study what is already there , literature review. Understand what the problem is.
- Specification (Requirements Engineering) - feasibility study, user requirements elicitation and analysis (input from stakeholders), systems analysis, etc. Overlaps greatly with domain analysis, how do we elicit the requirements? The feasibility study,
- Design - model software structures, software prototypes, interface specifications, component models, etc. Finalising models that describe the solution
- Implementation - source code, database, interfaces, data models, deployment plan, etc.
- Validation - component testing (unit testing, module testing), integration testing (sub-system and system testing), user testing (acceptance testing), etc. How will we test each component, each piece of code, does every component behave correctly when integrated.
- Deployment deployed system, user manuals, training documentation, training staff, fault report, etc. After fixing bugs we can deploy the software. User manuals, training staff, producing documentation

Waterfall Process

Simplest way of organising the development - rarely used nowadays

Large documents and collections of models created at most stages

In the classic model there is no turning back



Each stage needs to be fully complete before moving onto the next.

This methodology used in the 1960's lacked flexibility, and didn't allow for turning back as the previous stages had to be scrapped.

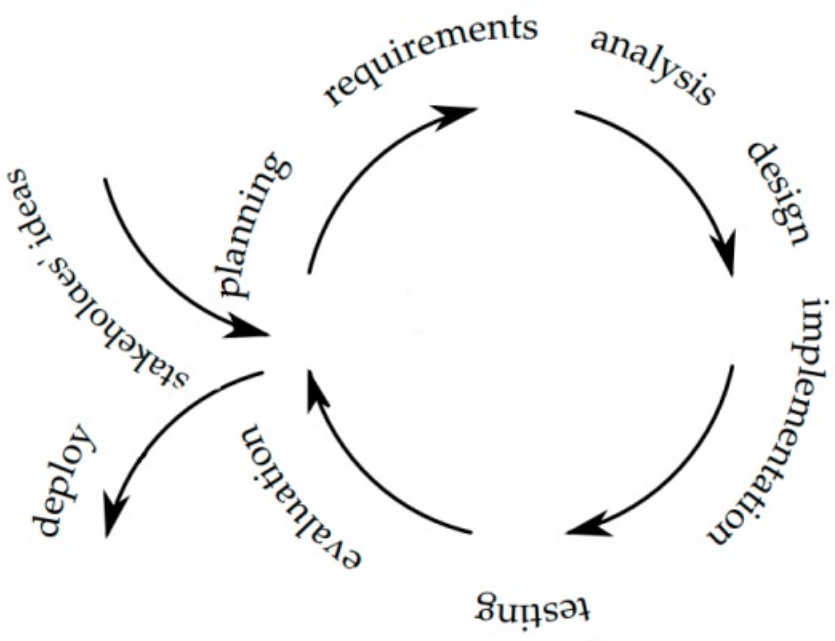
Cyclic

At every cycle of this process we will do a bit of every phase

cycles are between 1 and 4 weeks ,

(cyclically repeating (a subset of) waterfall activities.

when the cycle is complete we invite the stakeholders to give us some input.



Example : The first week might be planning and requirements analysis and the scond might be analysis and design.

When the iteration is over you're expected to produce some sort of output.

Iterative

Making improvements with each cycle

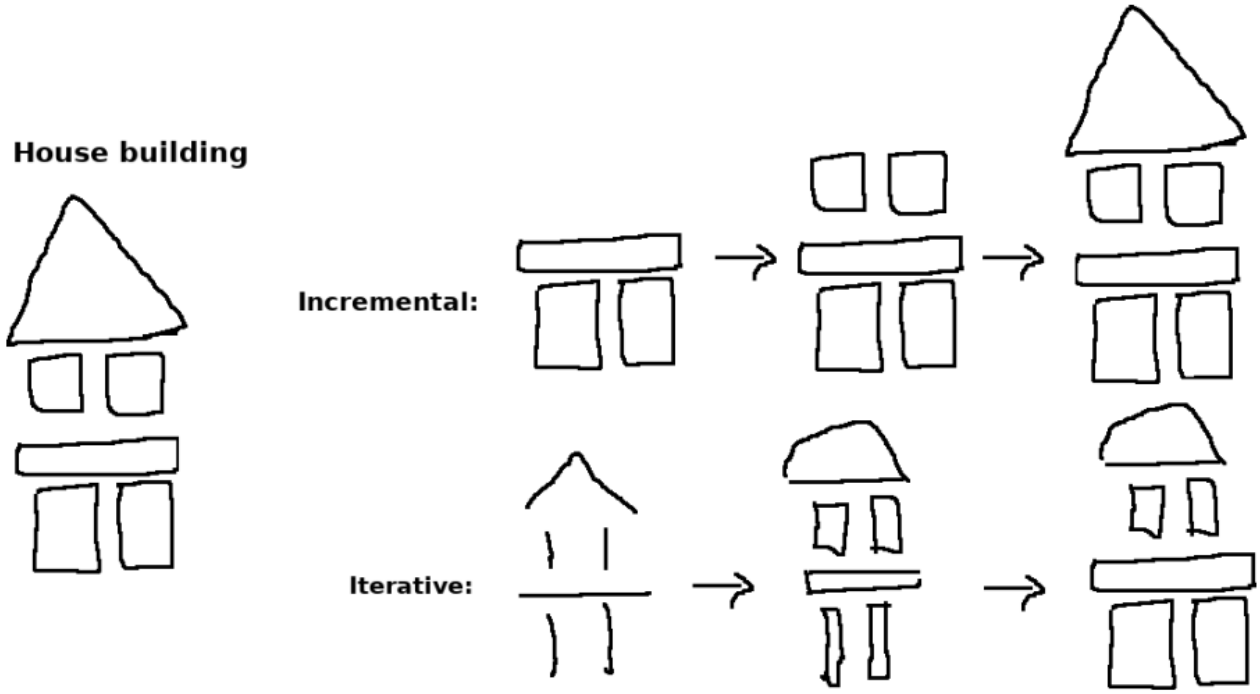
Incremental

Adding new features in each version

Incremental VS Iterative

top text

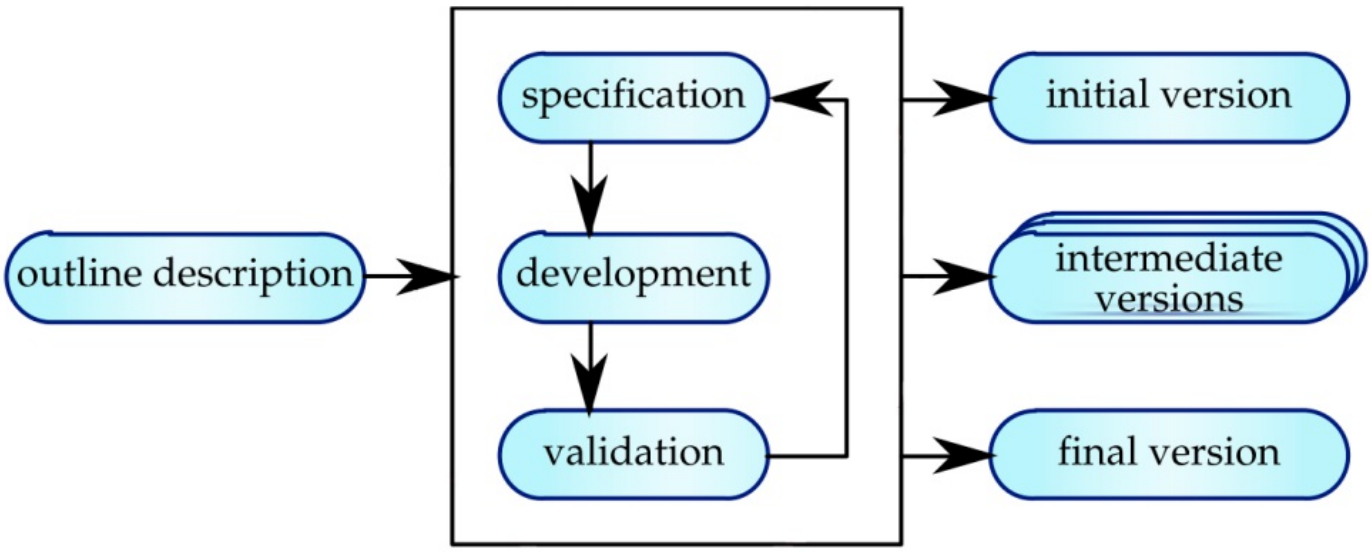
- list
- list



bottom text

Products of Development Cycles

Time is flowing down in the diagram below , we start from producing an abstract concept into a fleshed out version



StakeHolders know what they are paying for and we get feedback on how to improve.

Unified Process

A lifecycle method that allows us to work ourswlves from one activity to the next.

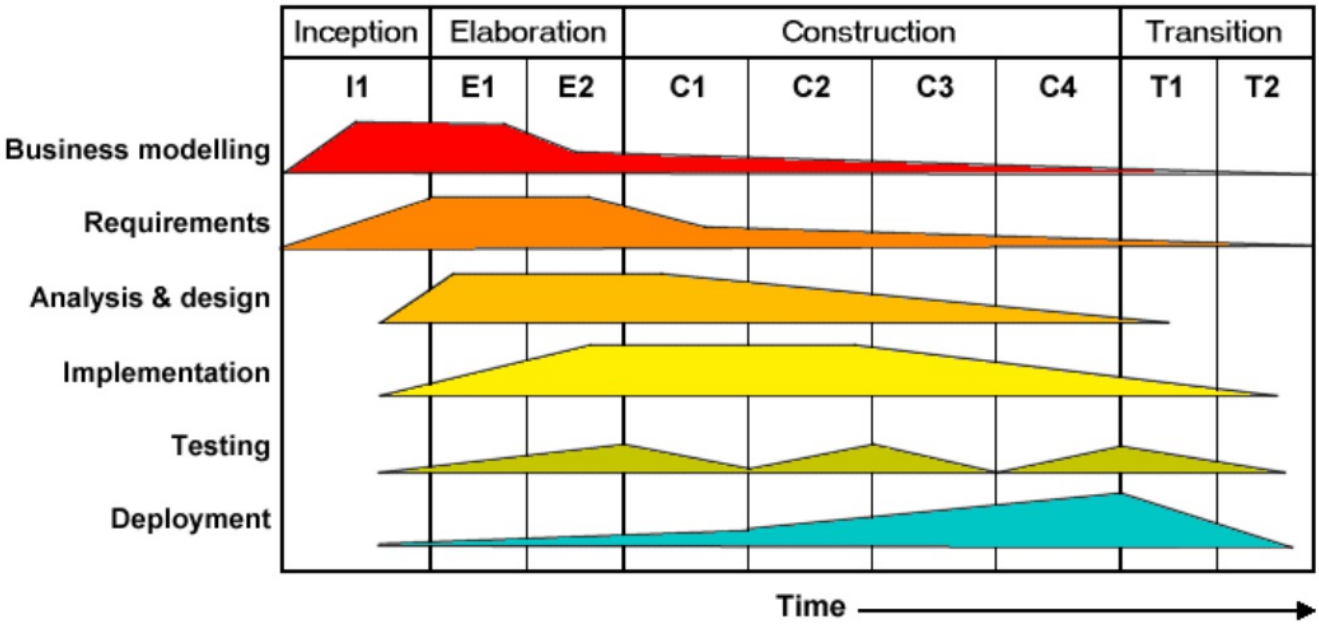
It's like waterfall where it's plan driven, but it also implements iterative features.

- Inception Phase - project scope, communication with stakeholders, goals, planning, costs → vision document, use case models, etc.
- Elaboration phase - analysing problem domain, eliminating major risks, refining plan → actors identified, clarifying requirements, etc.

- Construction phase - delivering the "beta" version, meeting all stakeholders' needs, testing → a working product ready to be deployed, test results, documentation, etc.
- Transition phase - deployment and maintenance of the system, bug reports, training users → project completed and in use, debriefing.

Unified Process Ilustrated

During the inception time we will spend most of our time Business modelling



Inception Phase

This is the initial phase of the project where the feasibility and scope are assessed.

Key activities during this phase are identifying stakeholders and creating a preliminary plan.

By the end of this stage you should have a clear understanding of what the project aims to achieve and whether it's worth pursuing.

Elaboration Ph

In this phase the requirements are refined and the architecture is developed.

Key activities include identifying use cases and creating detailed architectures / models.

At the end of this phase you should have a solid foundation on which to work.

Construction

This is where most of the coding and development takes place.

Key activities include implementing the system, testing and addressing defects.

Incremental development and testing are emphasized to ensure the system evolves steadily.

Transition

This is the final phase of the project that focuses on *transitioning* the system into production.

Activities include user training, system development and final testing.

The goal is to ensure a smooth *transition* to the operational environment.

Throughout the Unified Process, iterations are used to refine and improve the system. After each cycle (typically after the Construction phase), there is an assessment of the project's progress and a decision point to either continue with the next iteration or make adjustments to the project plan.

Agile

Agile is a software development approach as well as a movement and mindset.

> *"We do not act rightly because we have virtue or excellence, but we rather have those because we have acted rightly. We are what we repeatedly do. Excellence, then, is not an act but a habit."*
> *— Aristotle, Nicomachean Ethics*

Agile was the answer to the software crisis in the 1970's

Agile Manifesto

> "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 terms on the right, we value the items on the left more."

Agile process and principles

Text

- Face-to-face communication among team and customers
- Customer representative always available
- Software developed in quick iterations and tested continuously
- Good Quality, well documented code
- No long-term planning, but settling long term goals and visions
- Simple design to meet personal needs, refactorable due to changes

The Scrum

A framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value.

Scrum is used (80%) in the industry, it's not designed for CS or SD , but it has significant applications in agile developments

Scrum adds:

- Communication via several types of regular meetings
- stakeholders have to trust developers
- prioritisation of features via backlogs (todo lists)

eXtreme Programming adds:

- Pair Programming
- Sustainable pace, no overtime
- Shared System metaphors.
- Shared code ownership - anyone can edit

Scrum and Values

Pillars

- Transparency
- Inspection
- Adapting

Values:

- Pillars come to life and build trust for everyone only when
    - commitment
    - courage
    - focus
    - openness
    - respect
- are embodies and lived by the team.

Scrum Roles and Events

Text

Roles:

The product owner is one single person who is expected to represent the stakeholder(s) and is responsible for maximising the value of the product and work of the Development Team.

The Development team is equipped with all necessary skills to successfully deliver the product. Typically equipped with a mix of skills. Can be part of the dev team.

The Scrum Master can be part of the development team, they're responsible for providing support to the dev team and product owner. For example they organise meetings and ensure everyone has access to the same artefact. They're not a manager or boss, they're a support figure.

Events:

Sprint: time-box to produce a "done" product increment. A well defined time-box known as an increment.

Sprint Planning, Daily Scrum, Sprint Review, Sprint Retrospective.

Artifacts

In the context of Scrum, an "artifact" refers to a tangible or visible object that is used to facilitate and document the work of the Scrum team throughout the software development process. These artifacts help provide transparency, communication, and structure to the Scrum framework. Scrum defines three primary artifacts:

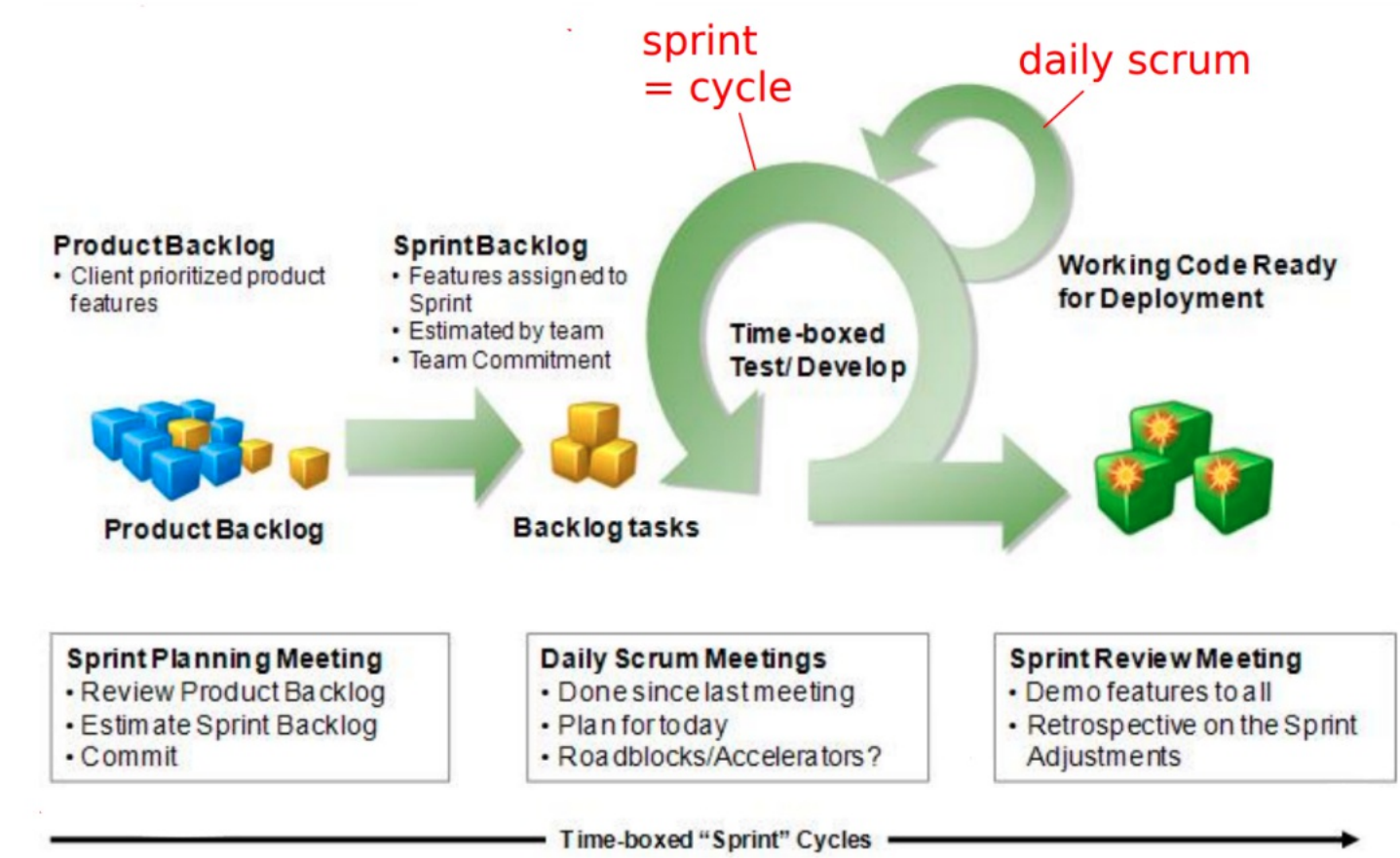
1. **Product Backlog:**
    - The Product Backlog is a prioritized list of all the work that needs to be done on the project. It serves as the single source of requirements for the Scrum team. The Product Owner is responsible for creating and maintaining the Product Backlog.

- Items in the Product Backlog can include user stories, features, bug fixes, technical tasks, and any other work related to the product. These items are often expressed in a way that describes the desired functionality or outcome from the user's perspective.

2. **Sprint Backlog:**
   - The Sprint Backlog is a subset of the items from the Product Backlog that the Development Team commits to completing during a specific sprint. It's essentially the work plan for the sprint.
   - The Sprint Backlog is created during the Sprint Planning meeting and is owned by the Development Team. It helps the team focus on delivering a potentially shippable product increment by the end of the sprint.
3. **Increment:**
   - An Increment is the result of the work completed during a sprint. It is a potentially shippable, working product that includes all the features, enhancements, and fixes that the team has completed during that sprint.
   - The goal of each sprint is to produce a potentially releasable Increment, meaning it's in a state where it could be released to end-users if the Product Owner decides to do so.

Scrum Overview Illustrated

top text



bottom text

UML Activity diagrams

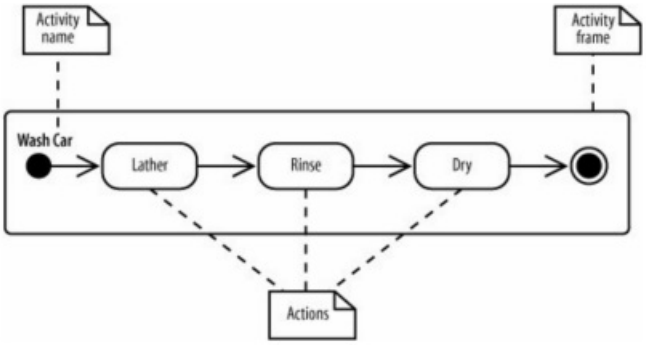
Activity Diagrams show high level actions chained together to represent a business process occurring in our system.

They allow us to understand different conditions and constraints as well as what functionalities are offered by components

Diagram #1

This box represents an activity. This activity is composed of a chain of actions. The black node names wash car is called a toekn

node and follows the direction of the arrows.



The token will move from one action (e.g. lather) to another (e.g. rinse) until it reaches the end node.

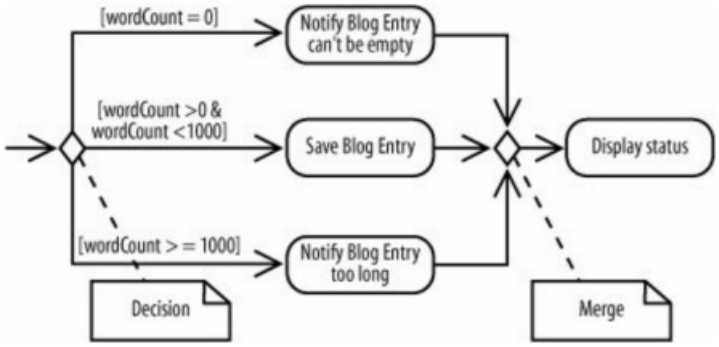
Each of the rounded boxes below is an action.

the arrows between them are called control arrows.

Diagram #2

An action starts when it receives a control token

Diamonds are used to depict a decision or merge nodes.



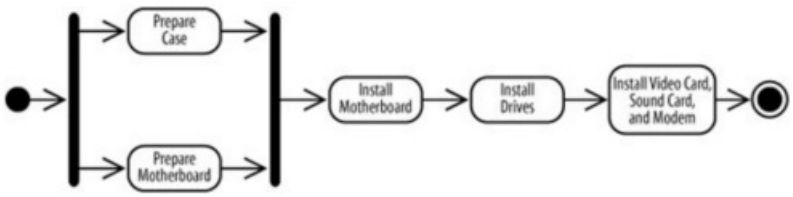
Diamonds can be used to represent a decision.

The arrows from them are called decision edges.

Diagram #3

Multiple actions that run in parallel are depicted using forks and joins.

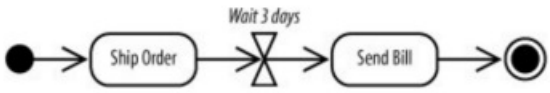
A flow is broken up into two or more simultaneous flows. All incoming actions must finish before the flow must proceed past the join



The first black vertical line is a fork, where the token splits and works on both actions simultaneously, it cannot proceed until they're both complete. It then reaches the 2nd vertical line where it "joins" back together and resumes flow.

Diagram #4

When time is a factor, time events are used to model a waiting period.

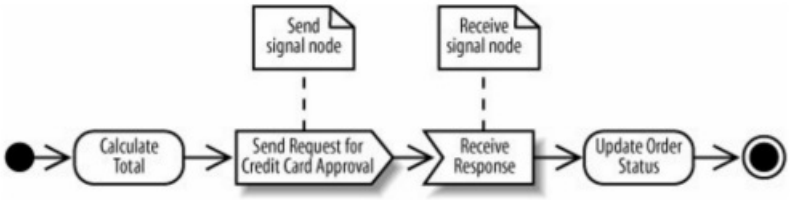


We can use the hourglass to depict time

Diagram #5

When an activity interacts with an external actor or process, messages are sent and received. These are called signals.

A receive signal wake up an action, whereas a send signal is a message to the external participation



The irregular pentagon can represent sending a signal to the external word,

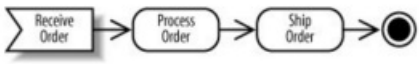
The delta represents receiving a message from the external requirement

Diagram #6

When send and receive signals are combined a synchronus flow is depicted.

If only a send signal is depicted, the flow is asynchronous (does not wait for a response to proceed).

A receive signal can replace a starting node

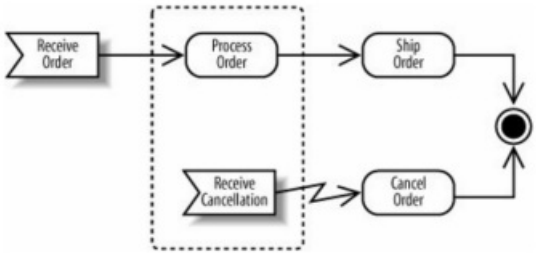


bottom text

Diagram #7

Interruptions are receive signals that cause activities to stop following their “expected” flows. A lightning bolt symbol is used to depict an interruption.

Interruptible regions show the area which an interruptiomn is expected to occus.

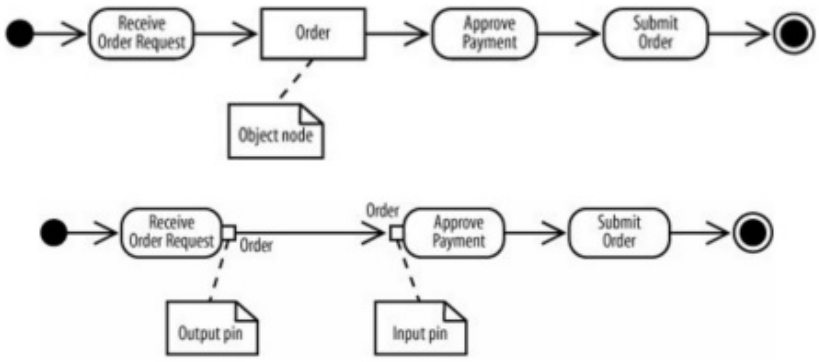


bottom text

Diagram #8

Objects can be depicted either by using object nodes or pins.

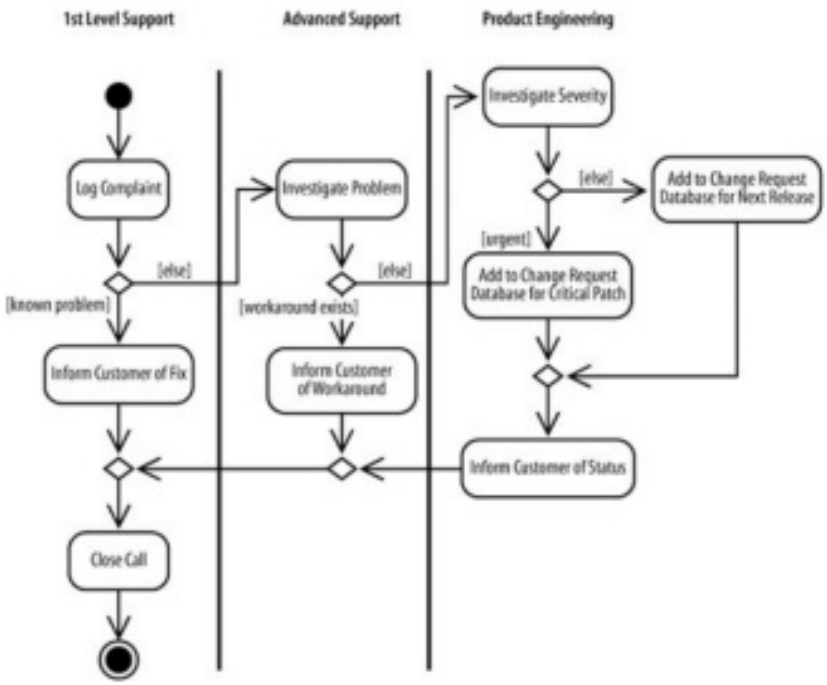
Input pins represent input objects, an action cannot run without an object parameter. Output pins specify that an object is output from an action.



The white boxes represent input and output pins (the node / action is producing or expecting an object as an input or output).

Diagram #9

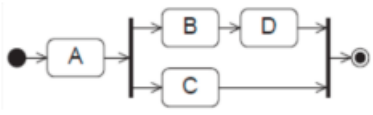
Swimlanes or partitions are used to depict which component or participant is responsible for which actions.



Swimlanes allow us to depict diagrams in which lots of decisions are made.

Exercise - quiz 1

top text



bottom text

Exercise - quiz 2

top text

bottom text

Summary

Revision

Week 3 Requirement Engineering

Objectives

- [ ] this lol

Notes

Requirements engineering

Requirements for a system are a description of:

- what the system should do
- what services should it provide
- what constraints affect its operation

The process of finding said requirements, analysing, documenting and checking these services and constraints is called Requirement Engineering

There are 3 types of Requirements that we will focus on:

- User Requirements - General statements in plain English about services and associated constraints (also diagrams!)
- System Requirements - Detailed descriptions of systems functions, services and operational constraints writting in a more technical language for more technical audiences.
- Domain and Software Requirements - derived from conversations with experts about operations and compliance.

Examples of user and System Requirements

As shown by the highlights in the image below user requirements are more of a generalised statement of what the program should do. This is beacause they're intended to be equally accessible and comprehensive to everyone especially the stakeholders who are the target audience.

▶ User Requirement Definition:

> *1. The LocalCare system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.*

▶ System Requirement Specification:

> ***1.1** On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.*
> ***1.2** The system shall automatically generate the report for printing after 17.30 on the last working day of the month.*
> ***1.3** A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed, and the total cost of the prescribed drugs.*
> ***1.4** If drugs are available in different dose units (e.g., 10 mg, 20 mg) separate reports shall be created for each dose unit.*
> ***1.5** Access to all cost reports shall be restricted to authorized users listed on a management access control list.*

The System requirements are written with more technical language as they're intended for a more technically comprehensive audience.

Types of Requirements

We know the 2 types.

Functional Requirements

▸ Statements of services a system should provide.
▸ How the system should react to particular input.
▸ How the system should behave in particular situations.
▸ In some cases, a functional requirement may explicitly state what the system should NOT do.

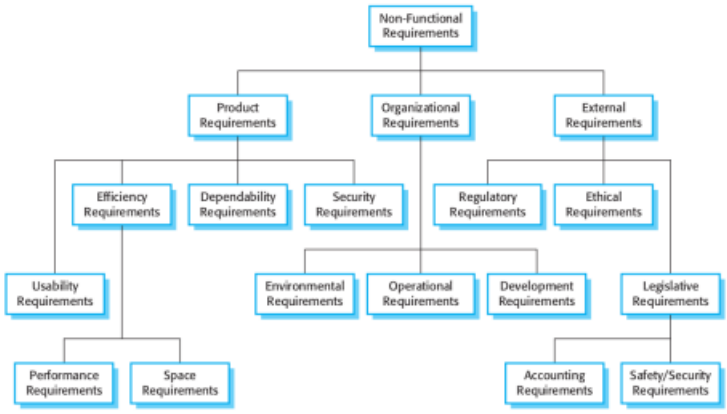
▸ Describe what the system should do.
▸ Depend on the type of software and expected users.
▸ User requirements are described in an abstract way to be understood by system users.
▸ System requirements are more detailed (input/output, exceptions, etc.).
▸ FRs are written in a functional requirements specification document, which needs to be both complete and consistent.

Non-Functional requirements

▸ Constraints on services or functions offered by the system.
▸ Timing constraints, development-related, imposed by standards, etc.
▸ NF requirements often apply to the system as a whole.

▸ Often more critical than functional requirements.

▸ Types of non-functional requirements:



Measuring Non-Functional Requirements

Non functional requirements must be testable / quantifiable / measurable in some way.

This is a non measurable way of writing an *NFR* :

The system should be easy to use by medical staff and should be organised in such a way that user errors are minimised.

This is a Measurable way of writing an *NFR* :

Medical staff shall be able to use all the 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.

Metrics for NFR:

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Software Requirements Document

Once we've collected all of our requirements we can write them down in a *Software Requirements Document*. This document

should be tailored to a wide range of readers from technical to stakeholder.

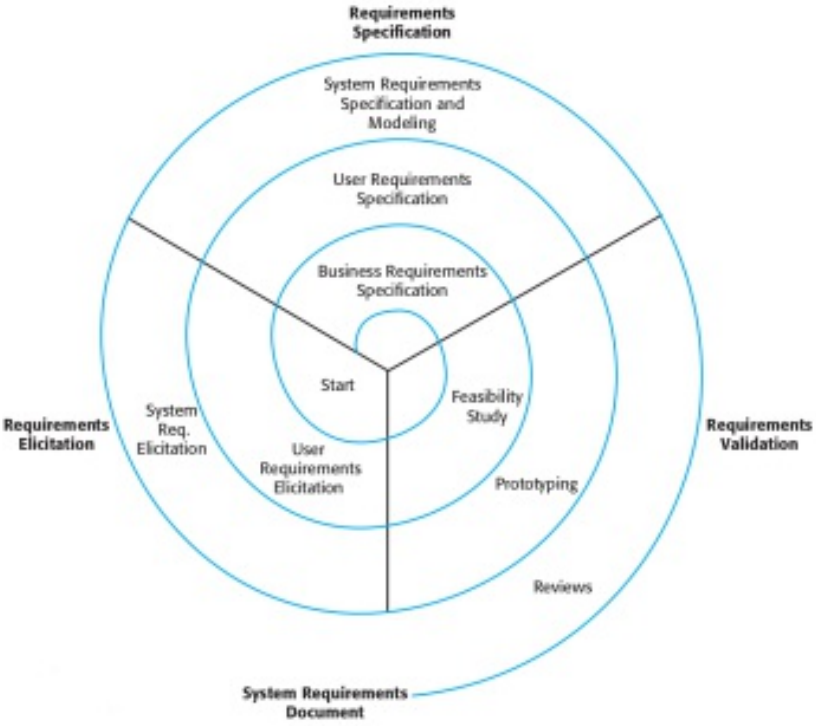
Requirements Specification

Requirement Specification is the process of writing down the requirements in the requirements document, almost always in natural language supplemented by tables & diagrams.

Useful guidelines:

- Invent a standard format.
- Use language consistently to distinguish between mandatory and desirable requirements (e.g., "must/shall", "should", etc.).
- Use text highlights (e.g., bold, italic or colour).
- Avoid jargon, abbreviations and acronyms.
- Explain rationale.

The process of Requirement Specification is also iterative:



Feasibility study

This is a short study focused on the *feasibility* of the project covering:

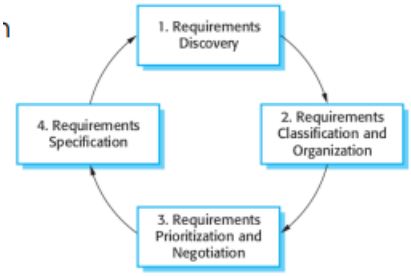
- Is the system useful?
- is it possible?
- what risks are involved?
- Can it be integrated with existing systems?
- Can it be implemented within time and budget constraints?
- Does it contribute to the overall objectives of the organisation?

Exercise

let's go to the gym buddy!

Requirements Elicitation and Analysis

1. Software engineers work closely with stakeholders to *elicit* and *analyse* requirements
2. Discovering user, system and domain requirements by applying several techniques to interact with stakeholders
3. Classifying and organising related requirements by creating clusters of cohesive and similar requirements and associating them with sub-systems of the architecture.
4. Prioritising and Negotiating - ordering requirements and resolving conflicts
5. Specification - validating and formally documenting requirements.
6.



Elicitation Challenges

- Stakeholders often don't know what they want / need (that's why AI won't kill the programming industry).
- Requirements are often expressed in the stakeholders own terms, and with implicit knowledge of their work.
- Different stakeholders have different requirements which may lead to conflict between the requirements.
- Political factors may influence the requirements of the system
    - Managers can demand specifications to increase their own influence in the company
- The importance of each requirement may change with the economic and business environment.
- New Stakeholders will come with new or overlapping sets of requirements.
-

Requirements Discovery

Requirement Discovery can be accomplished via several ways according to the stakeholders:

- Brainstorming - Creating new ideas - requires voting
- Interviews - closed or open , formal or informal.
- Questionnaires - running short, targetted surveys.
- Examination of documents / artefacts - reading current policies / procedures.
- Scenarios - running example interaction sessions.
- Use cases - textual or graphical diagrams musing UML
- Prototyping - best in receiving feedback, can be paper, powerpoint or functional.
- Ethnography - Watching daily activiites and identifying problems.

Scenarios

A scenario is a sequence of events and or actions.

A workflow is a set of scenarios with a common goal usually structured:

- Primary path - msot common scenario
- Alternate path(s) - less common scenario to the goal.
- Exception path(s) - scenario aiming for the goal but missing it

You'll spend 80% of the time dealign with what will happen 20% of the time (alternate paths).

Note the numbering.

Primary path:

1. Fill kettle
2. Switch kettle on
3. Put tea in teapot
4. Pour boiling water
5. Wait for two minutes
6. Pour tea into cup
7. Add milk and sugar

Alternative path:

1.1 Kettle already full

Start with step 2

3.1 Cup instead of teapot

Put tea in cup at 3

Leave out step 6

Exception path:

2.1 Kettle exploding

Kettle explodes after step 2

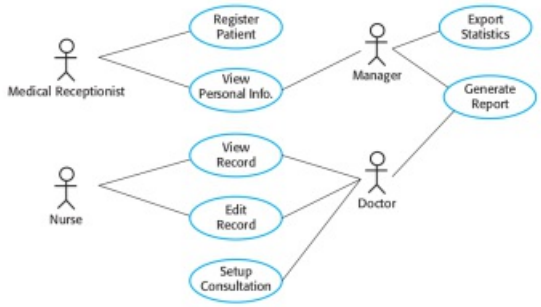
Leave out steps 3-7, stop the fire

Use Cases and use Case Descriptions

A use case is a piece of functionality performed by the system that can be described by a short name.

Each case is associated with its initiators and a use case description.

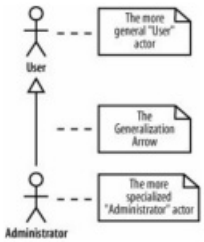
Its name should describe the functionality from the initators POV.



UML Use Case DIagrams

Actors are shown as stickmen or sterotyped boxes for external systems.

Uml

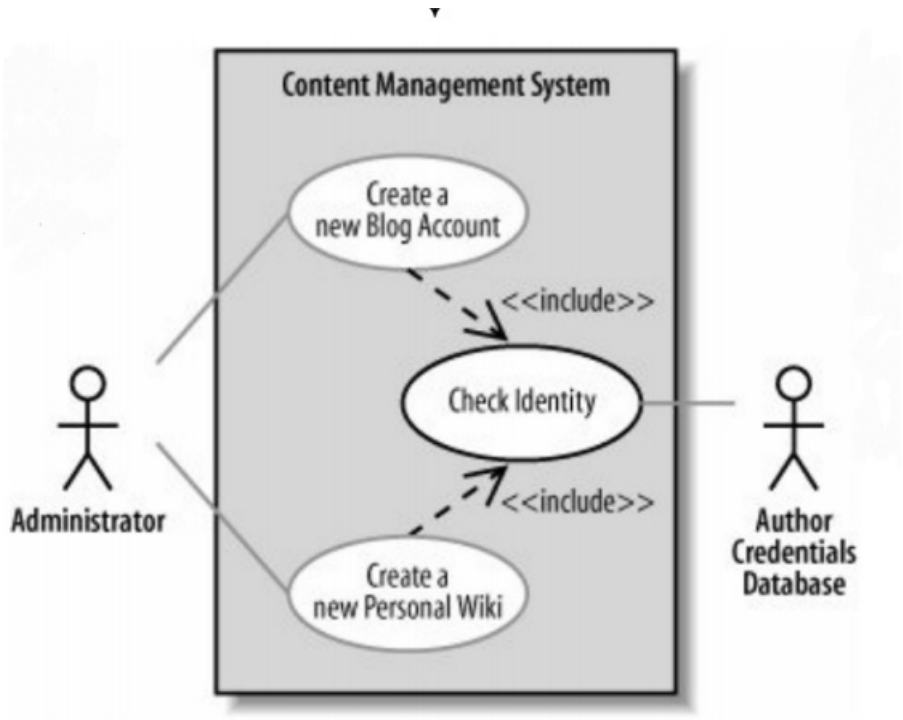


Use case

Relationships in UML Use Case Diagrams

Include

> The content management system shall allow an administrator to create a new personal Wiki, provided the personal details of the applying author are verified using the Author Credentials Database.

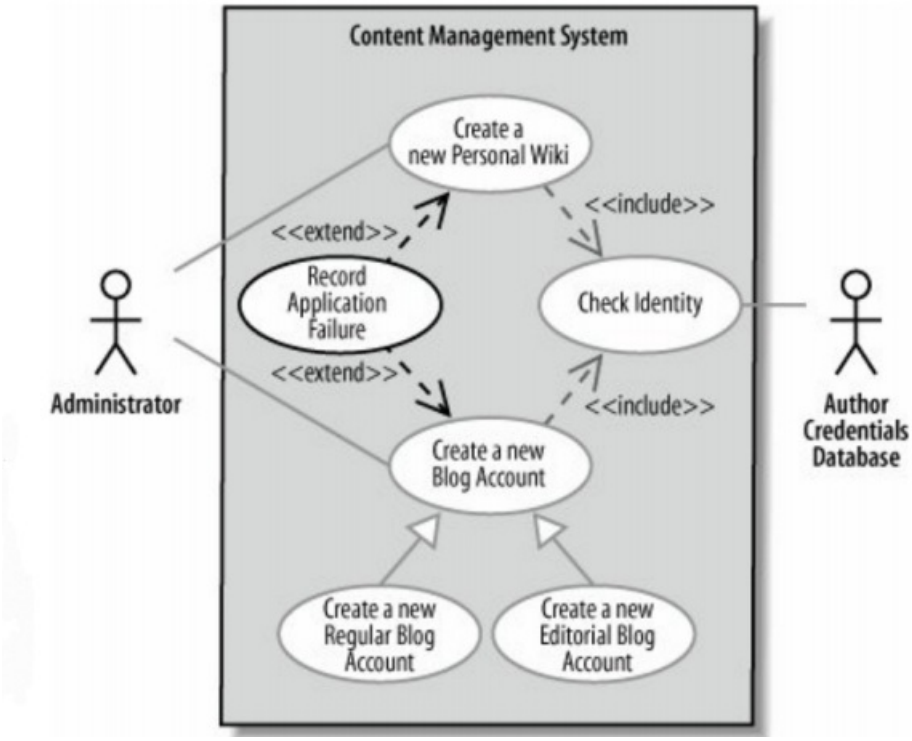


Here, "Create a new Blog Account" uses all the steps from "Check Identity" use case description.

No need to repeat "Check Identity" steps in the use case description of the "Create a new Blog Account" use case.

Extend

<<Extend>> in this context means optionally including, i.e.,
may A may or may not include B.



"Create a new Blog Account" extends "Record Application Failure": the situation may sometimes happen.

Extend has nothing to do with Java's inheritance. For the latter, UML use case diagrams use the generalisation arrow.

Requirements Classification & Organisation

This is abut grouping related requirements and given them numbers to make them traceable (e.g. FR1 , FR2, NFR1, NFR2).

It helps decompose the system into sub-systems and components of related requirements and define relationships between those components to identify which design patterns to use.

- Classifying by audience. Who are requirements written for?
- User requirements: stakeholders
- System requirements: stakeholders who engage in detail.
- Software requirements: mainly developers.

requirements type	client managers	system end users	client engineers	developer managers	system architects	system developers
user reqs	✔	✔	✔	✔	✔	
system reqs		✔?	✔		✔	✔
software specs			✔?		✔	✔

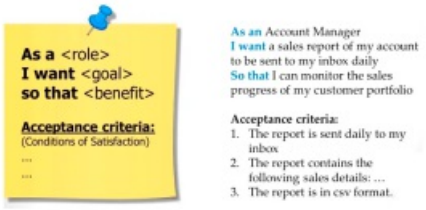
Requirement Prioritisation

- MoSCoW: Must, Should, Could and Would (or Won't, Wish to) have.
- If any of the must requirements is not included, the delivery is considered a failure.
- Should requirements are important but not as time-critical as the must ones.

▸ Requirements related to improving user experience are often classified as could.
▸ Won't requirements are least-critical and can be delivered in the next release.

Requirement Prioritisation with Agile

User stories are used instead of scenarios (use cases).



Cards are created which usually contain a checklist of test to validate the requirements later.

Good User stories:

These use the ***INVEST*** technique:

- Independent - can be understood without having to read some other user story.
- Negotiable - invitation to discuss
- Valuable - Giving useful outcomes to the stakeholders
- Estimable - can be measured
- Small - can fit into an iteration
- Testable - can be declared as completed

Story prioritisation is completed using *maps* :

user story maps

Requirements Validation

The process of checking that requirements actually define the system that the customer really wants.

There are 5 different types of checks :

- Validity Check - Does the system provide the functions which best support the needs of the stakeholders
- Consistency check - are there any conflicts to resolve
- Completeness - are all functions required by the customers included?
- Realism - can the requirements be implemented within the time and budget?
- Verifiability Check - can the requirements be tested ?

Techniques

There are many ways to complete requirement validation:

- Requirements reviews - the requirements are analysed systematically by a team of reviewers who check for errors and inconsistensies.
- Prototyping - an executeable model of the system in question is demonstrated to end users and customers.
- Test-case generations - if a test is difficult or impossible to design, the requirements will be difficult to implement.

Summary

Lecture Notes

Slide 4 - Requirements a short description in plane language

User requirements are the user side of the story what do they expect

System Requirements are what the system needs to do

Domain and Software requirements are the requirements identified from interractions with the experts, related to the organisation, how the organisation works. Software are related to what standards we need to comply with to produce the software

Slide 5 - highlight generate monthly in first quote , 2nd quote highlight last working day etc

We find this information from more interractions but we start with user requirements first. We derive system requirements from a more technical audience.

Slide 6 - Functional what does the system do / provide or how it should behave in a particular situaion or functional environment.

Sometimes they describe what the system should not do

Nonfunctional requirements are not related to the function of the system but can describe the constrains of the system , like following a standard. non-functional are emergent properties. Non-functional describe what the system is. How does the system asdress the constraints

slide 7 - user requirements are described in an abstract way because we expect the stakeholders to understand them, Functional requirements are written ina compleyr document - the document contains all the requirements the user has asked for (desired . neccessary functionality) . the document needs to have consistent language and structurte and that there is no overlap between requirements , there is no requirement that cancels some other requirement out.

User requirements as abstract meaning in a general way to open the discussion to more questions.

System requiremtns are written ina general language but are more specific, they’re intended for a more technical audience

slide 8 - what kind of existing technologies do you need to incorporate in your design ,

slide 9 - the non-functional requirements need to be testable , find a way to quantify performance , understand whether something has passed the test or not

slide 10 - metrics

slide 11 - Assuming we have all the requirements we need to write them down in a software requirmeent document which covers both functional and non-functional requirements.

DOCUMENT - Requirements have a neccessity rating , they have ID’s ,

Slide 12 - process that allows us to generate this document (write it down formally) . We tend to write it in a natural langauge as the document has a broad audience ,

slide 13 - the requirements engineering as a sequence of activities is also iterative ,

slide 14 - feasibility study is an integral part of software engineering , anything that can affect our ability to develop the system. should be short snd focused on if the development will be affected in any way by issues. Do we need to do anythign about them, risk assessment

slide 15 - exercise -

slide 16 - discovery of requirements, once discovered we need to do something clevber / productive with them. Classification how have people grouped requirements together , based on components? sybsystems? hierarchy ?

slide 18 - stakeholders don't ahve a technical background , the business analyst is expected to know the stakeholders alnguage and convey requirements to the devs. Depending on different stakeholders different requirements, there may be some requirement conflicts.

slide 19

slide 20

slide 21

slide 22

slide 23

slide 24

slide 25

slide 26 - a functionality performed by the system , usually described by some name. When written into a document they have a description. this is a document tnhat al;lows us to write scenario types in a formal way, as well as UML to describe a use case diagram as to how the system is used by actors.

slide 27 - stickman - human actor , has actor stereotype , systems can use just a box.

We can use a generalisation arrow to describe a special actor such as an admin being a special user we draw a generalisation arrow to the actor.

When the arrow has a head it emans the actor is initating an activity / interraction

slide 28 - you cannot create an account use case without completing all the steps in check identity use case.

slide 29 - extend - dotted line - optionally included in a use case - can be a part of it, in some cases we may need to use this use case.

36 - User stories are a great way to start , some sya they'll replace use cases but we still need use casie diagrams.

slide 39 -

Revision

quantifiable NFR

Week 4 Business Domain analysis

Objectives

- [ ] this lol

Notes

The Big picture

▸ Unit 1: Systems Thinking (systems, components and properties/behaviours), component diagrams.

Breaking down systems into smaller components we can understand to produce complex solution

▸ Unit 2: SDLC (Waterfall, UP, Agile), activity diagrams.

▸ Unit 3: Requirements Engineering, use case diagrams and descriptions.

▸ Unit 4: Business Analysis, object/class/state machine diagrams.

▸ Unit 5: Systems Analysis, sequence/communication diagrams.

▸ Unit 6: Software Testing.

Analysis vs Synthesis

Before we go into the business Analysts role ,

Analysis

This is an investigation of the component parts of a whole

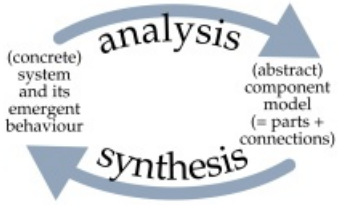
in analysis you're given a problem and you try to break the system int oseveral components which cna be easily modelled to figure out hwo the system will work.

It involves some kind of abstraction, such as creating a component model.

Synthesis:

Creating something real out of abstract models

Trying to combine separate parts in a clever way so when we put them together we end up with a working system



Both of these cases involve a lot of modelling, in software engineering we do both.

Analysis and Software Engineering

In Business Context Domain Analysis

Analyse existing structures and their processes.

In Requirement Analysis

Discover and define desirable external behaviour of a new or modified system

Trying to understand and discover desirable behaviours for the system , user case descriptions etc

In Systems Analysis

Elaborate requirements into a more formal specification of the desirable external behaviours (including users perception of the internal behaviours)

In reality we do more synthesis here than analysis.

Write the requirements formally and fiure out how the users will interract with our system.

How will objects communicate, how does this affect the behaviour of our system etc.

Exercise

Think about creating an activity diagram during a system's design. Would you call it an analysis or synthesis activity?

▸ Analysis.

▸ Synthesis. ←

What is business Analysis

Business Analysis is the set of tasks and techniques we use when working with stakeholders to understand the structure, policies and operations of an organisation.

A set of tasks or techniques, that we apply when we interract with stakeholders to understand what you're looking for. A lot of time is spent doing research as opposed to writing code.

The business analyst will do this, they will interract with the stakehoders and research the company.

We don't expect the BA to be technical, a lot of this implementation will be very abstract.

They will assess and validation a solution to the problem, this too will be very abstract.

We expect them to know the termi ology of the domain but not to provide technical solutions, they help bring stakeholder and devs together.

A business Analyst:

- Identifies problems and opportunities
- Elicits the needs and constraints from stakeholders.
- Analyses the needs and defines requirements
- Assesses and validates the potential and actual solutions.

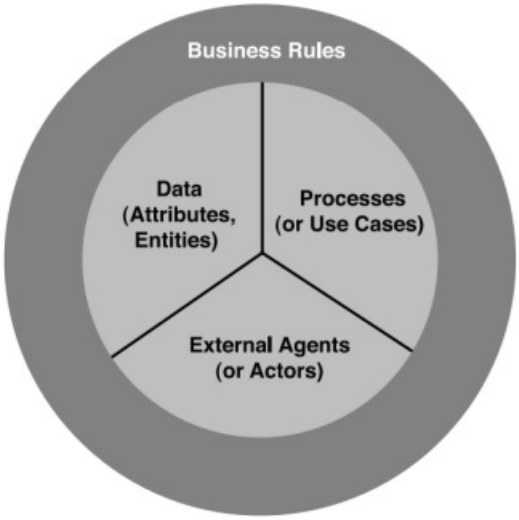
The main focus is on business goals and not technology designs

BA's to Developers ratio 1/6 but now changing towards BA's

Business Analysis towards Requirements

When describing a business, there are four basic requirements components:

1. Information is data
2. Manual or automated activities and procedures the business performs
3. People or systems or departments inside and outside the organisation.
4. Guidelines, cosntraints and policies under which the organisation operates



When we're involved in the dev of a solution for a business, we need to find out what the bsuiness do. All bsuinesses produce data ans that describes internal processes as well, we learn how different departments operate and how clients interface with the bsuiness. This helps us think of hwat objects we will need to design. We can also learn about activities and procedures of the business,

Business Analyst - the "Middle Man":

BA's break requirements down and see the specific parts of the busienss that may need improvement.

More specific questions lead to more detailed requirements

Why document requirements?

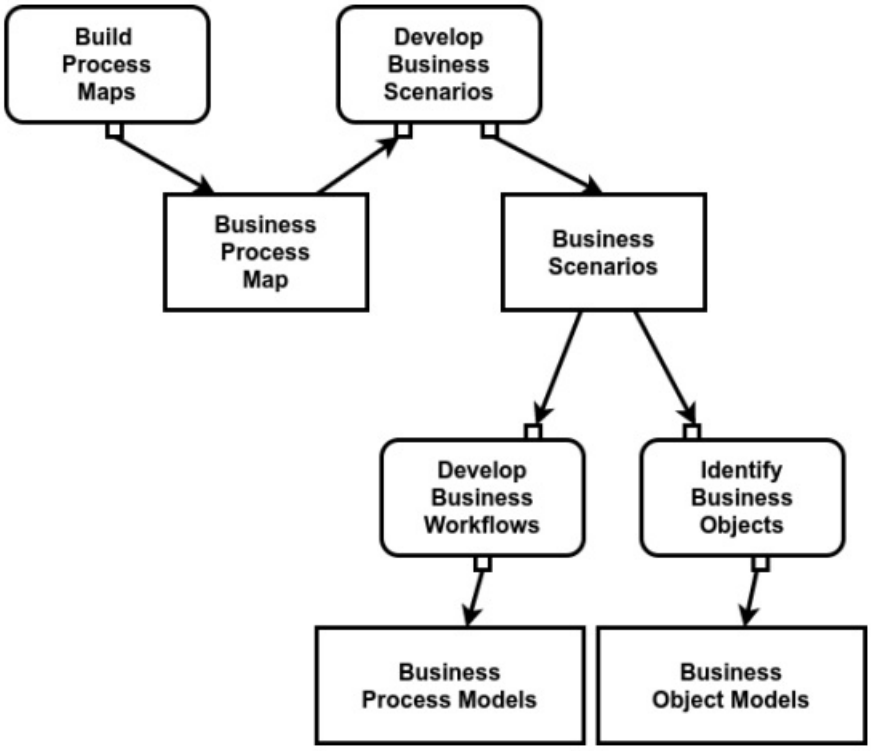
- people forget
- verbal communication is fraught with errros
- People answer the same question differently if asked twice
- Writing down something forces us to think of it carefully
- Stakeholders review what BA's write down
- New people joining a project get up to date sooner.

The business analyst is expected to understand the terminology in the domain and interact with both the developer and stakeholder, they are expected to produce a lot of documentation.

Conducting Business Analysis

This section focuses on Business process Maps and object models

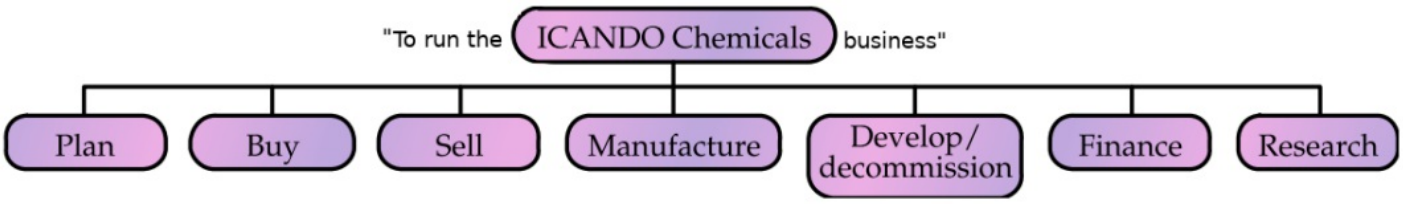
Class & object diagrams and state machines



This is an activity diagram.Each node ahs an output object which is all the business processes.

Business Process Maps

A business process map - a business process is a logicsl grouping of things that are important for business. We can then have a discussion with stakeholders about grouping eleemnts describing what the business does

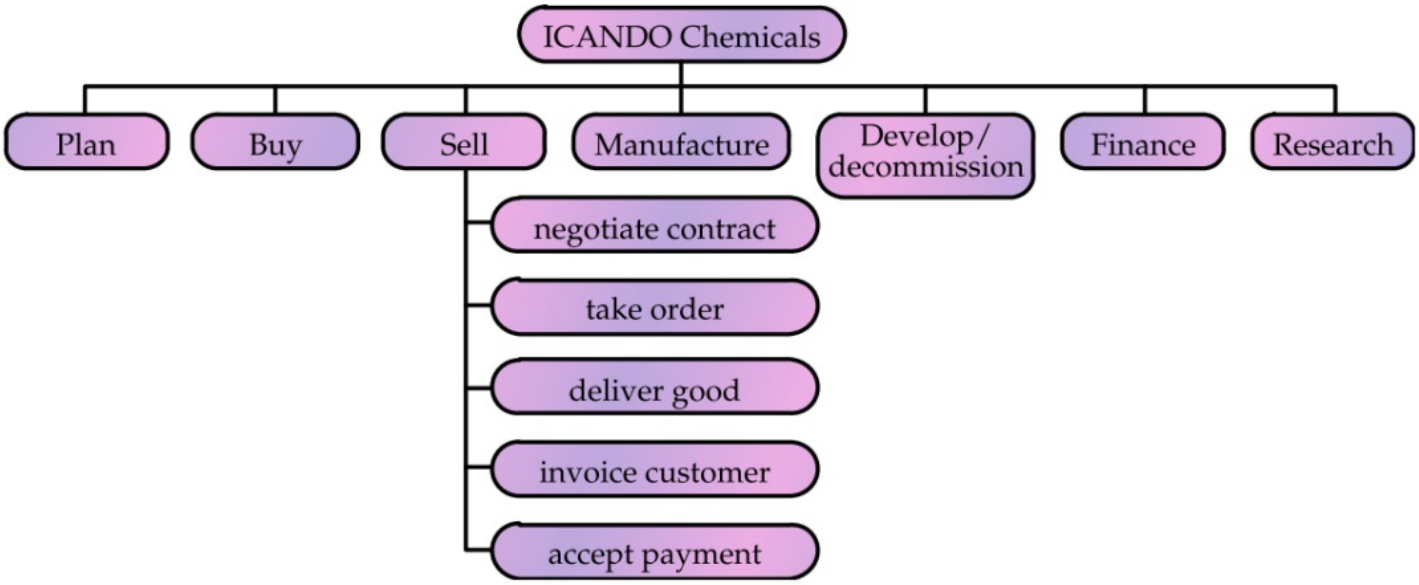


A business process is a logical grouping of events that can be agreed as a fundamental element of the business.

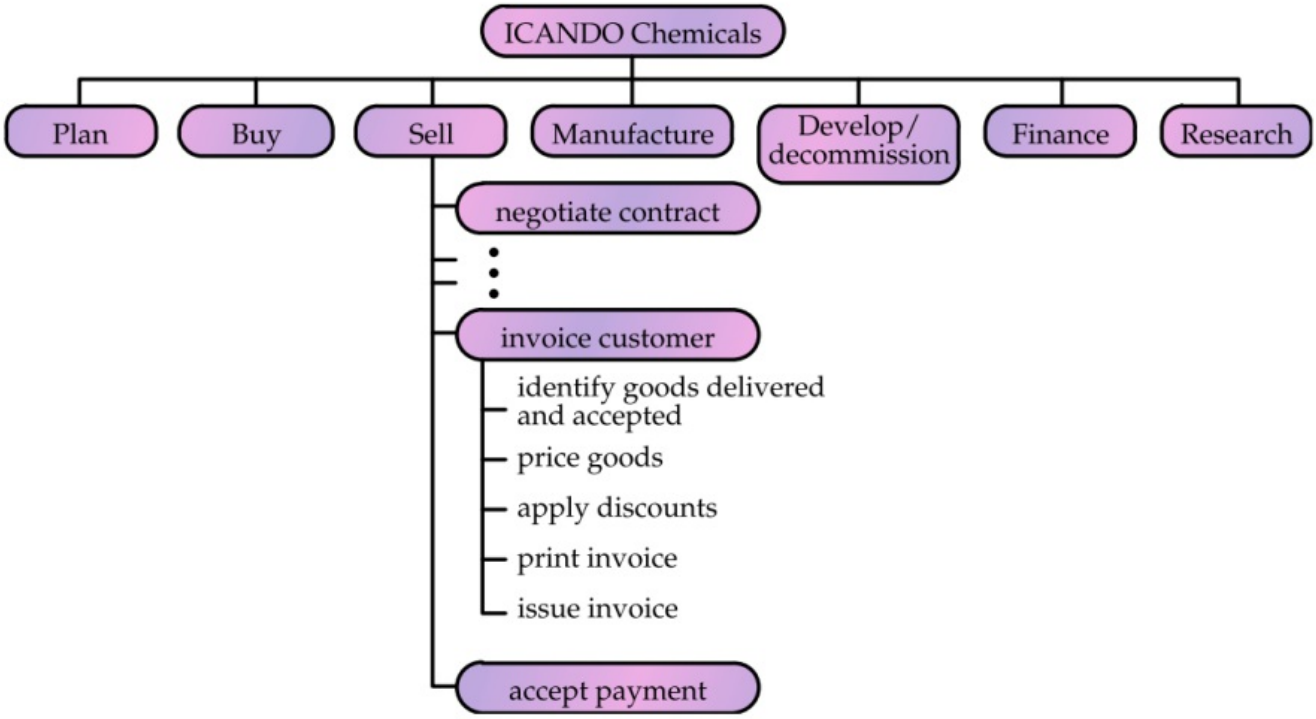
A business process map is a collection of named processes grouped hierarchally on no more than 3 levels.

A few rules:

- The root is the most abstract process
- Level 1 should not have more than 10 high-level processes



A lower-level process is a specialisation of its parent



Braking down to more than 3 levels will render the business process map useless as it has to remain abstract.

Things to avoid in business process Maps

Business process maps help us focus on main functionality

Provide a top-level view and give us scope, clarify terminology, easy to understand for everybody

No need to present a perfect breakdown - Sofwatre dev is an iterative process so the map might not be perfect straight away , but

it has to be good enough to start with.

A business process map does not show:

- The structure of the organisation
- The information flow
- Time considerations
- Interactions among processes

Its about understanding business processes not the interactions

Method to produce a business process Map

Usually half a day workshop attended by business managers and experts

All contribute to knowledge, but need to reach an agreement.

Need for a facilitator:

- Oversees the discussions and forces rules
- Aiming for a high level view and a good but not perfect map (not always achievable).
- Ensures timing (time boxed event).

Conducting business Analysis - Revisited

Identifying Domain Objects'

Noun-verb analysis - the first step to class decomposition

To break a large problem down into a class structure.

Imagine you've done all the initial business analysis steps, like scenarios, use cases and descriptions and you're trying to understand how to derive objects and classes.

Class decomposition - converting things like use cases into classes and objects.

A good method to do this is noun-verb analysis, we try to find the nouns which will represent classes, and the verbs will represent services provided by the class.

Nouns may represent:

- classes
- Specialisations (classes that extend Abstract classes)
- Attributes (a Student has a name).
- Data (Values to attribute variables/containers/etc).

Verbs may represent:

- Services (methods) provided by a class
- Services used by Classes

Exercise

Considering the workflows on the right, identify potential classes using the noun-verb analysis method.

Answer

Note the numbering.

Primary path:

1. Fill kettle
2. Switch kettle on
3. Put tea in teapot
4. Pour boiling water
5. Wait for two minutes
6. Pour tea into cup
7. Add milk and sugar

Alternative path:

1.1 Kettle already full
Start with step 2

3.1 Cup instead of teapot
Put tea in cup at 3
Leave out step 6

Exception path:

3.2 Kettle exploding
Kettle explodes after step 2
Leave out steps 3-7, stop the fire

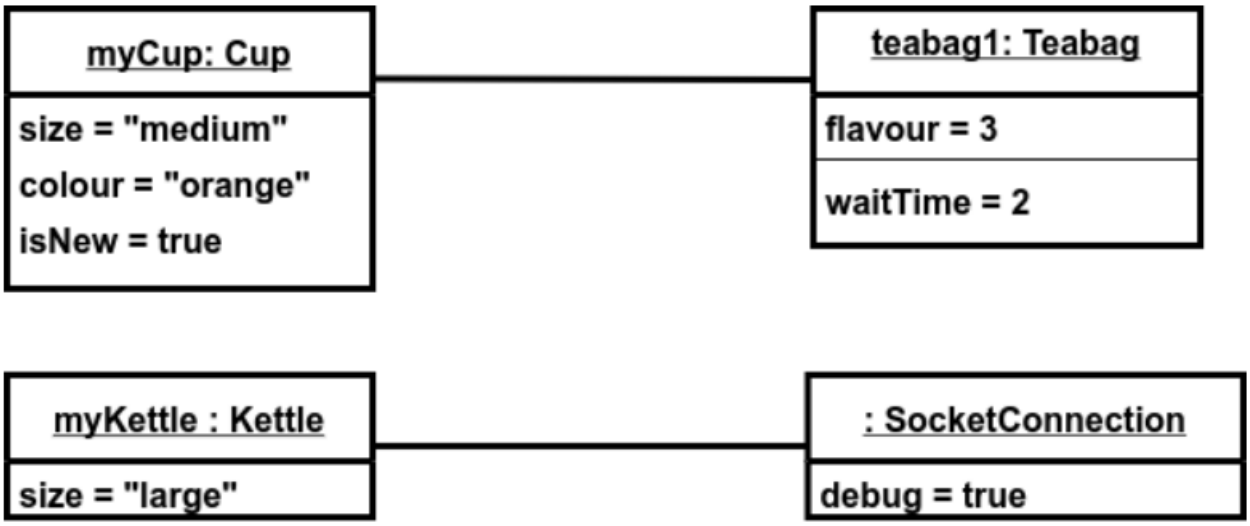
UML Object and class Diagrams

- UML Object Diagrams are used to communicate object information (relationships and static view).
- They represent instances of classes and derive from UML Class Diagrams
- Object diagrams may or may not contain object attribute values.
- Class diagrams are usuua;;y more detailed, include attribute and method information.

Notation: 

myCup is an instance of class Cup.

UML Object diagrams

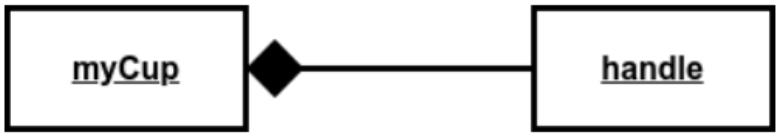


Anonymous objects may exist, but why?

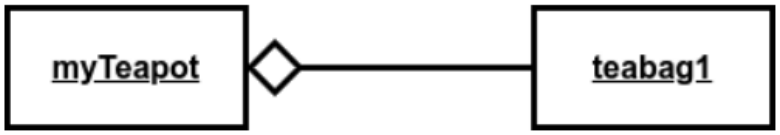
Object Relationships

text

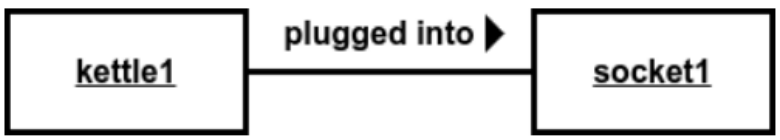
- **Composition** (part does not exist without the whole):



- **Aggregation** (a part of a whole):



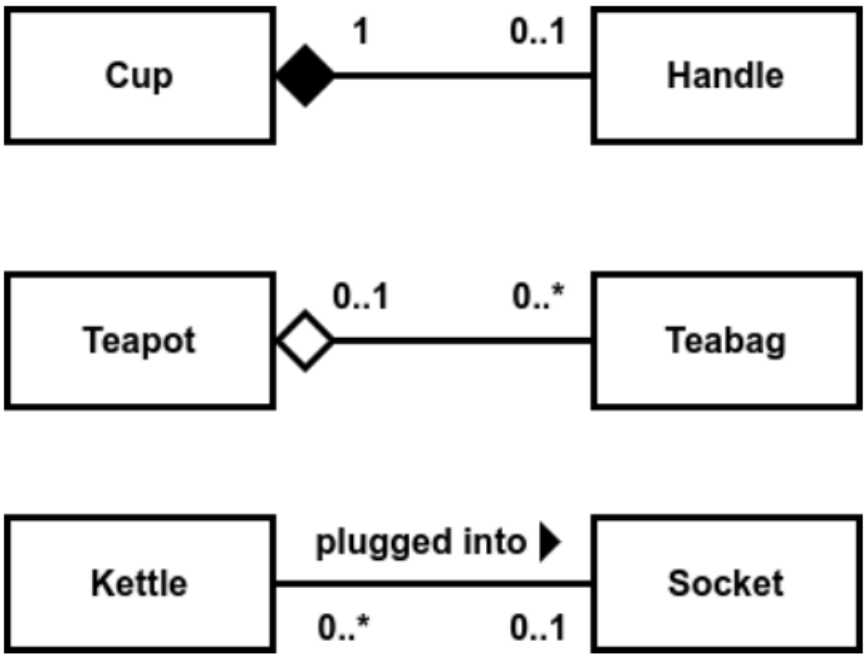
- **Association** (arbitrary relations, can be navigable):



text

Relationships over classes

Multiplicities should be added to provide more information about relationships when showing classes of a system



Practical

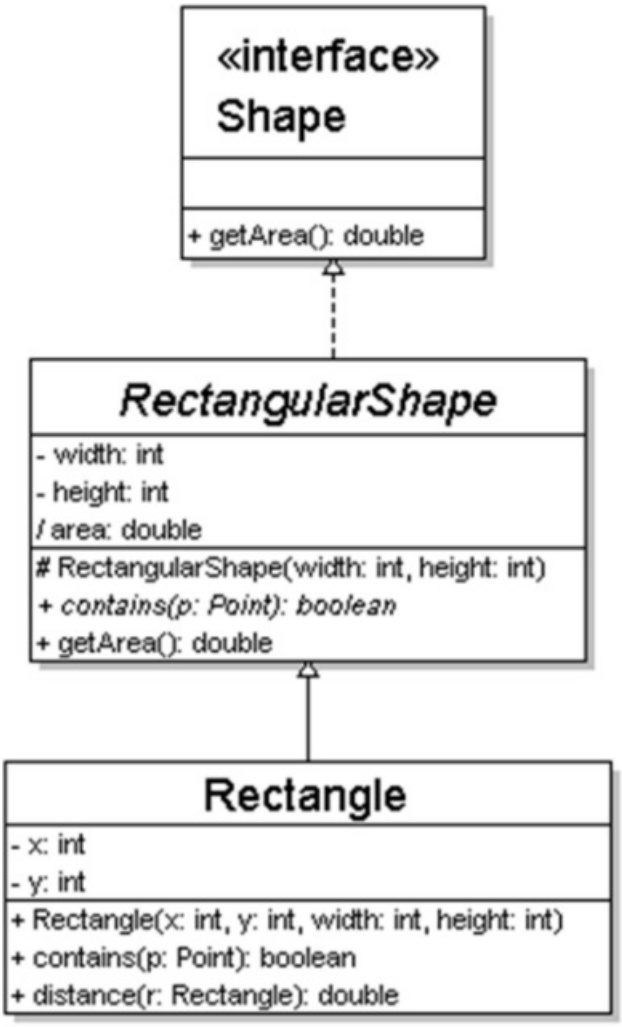
A detailed class Diagram

Class diagrams can be very detailed

- Private -
- Public +
- Protected #
- Derived /

Parameters and return types

Interface and Abstract classes (realisation and specialisation arrows)



Object and Class Diagrams - What to remember

Use a UML Object Diagram to:

- Show a static view of an interaction
- Describe object relationships of a system

Use a UML Class diagram to:

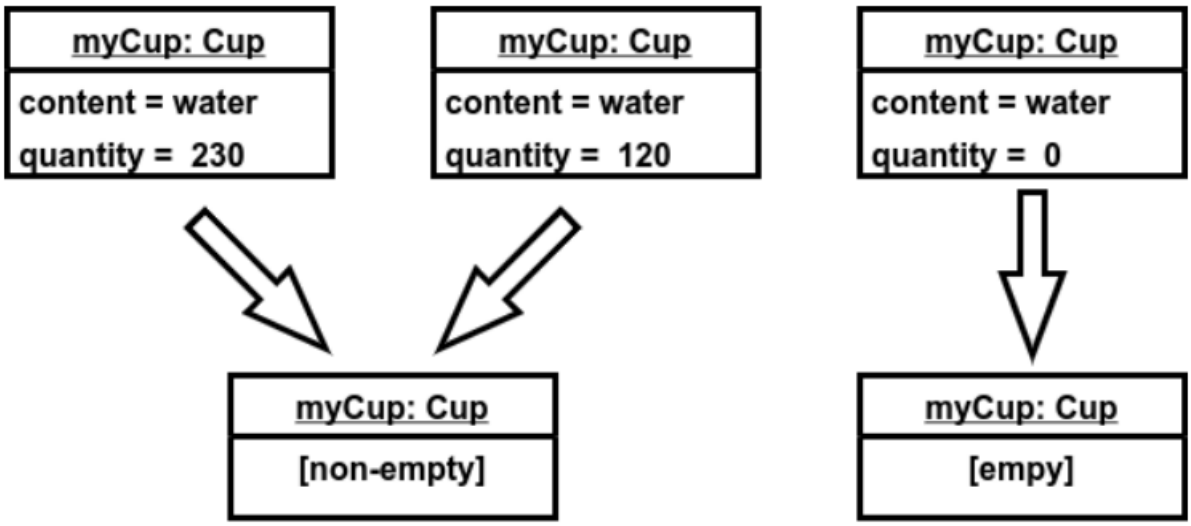
- Better understand the general overview of the schematics of an application
- Provide an implementation-independent description of types used in a system.

Object States and State Machines

State machine diagrams are sued to show how an objects state changes during its lifetime

Object state = attribute values.

Drawing a state machine requires increase of abstraction

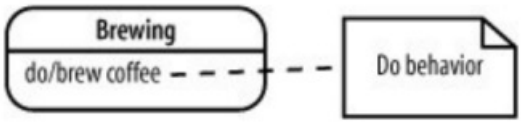


State Machine Diagrams Notation

Object states can be a passive quality or an active quality

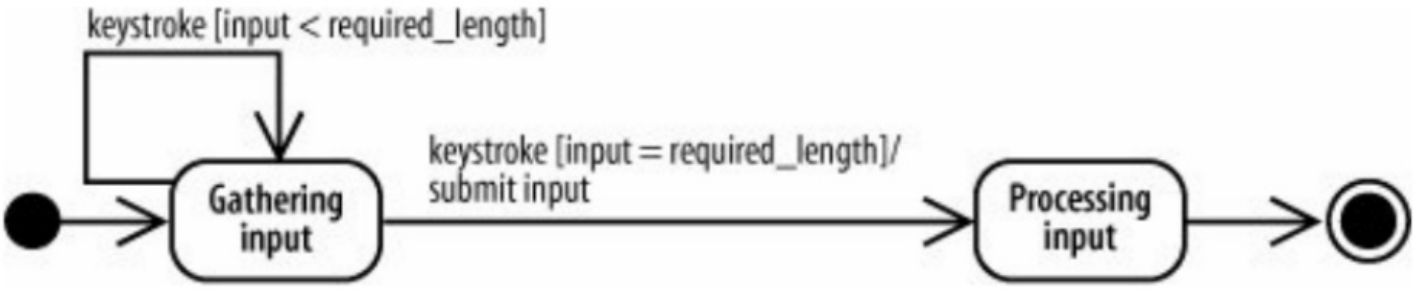
- Object is on or off
- Object is doing something

text



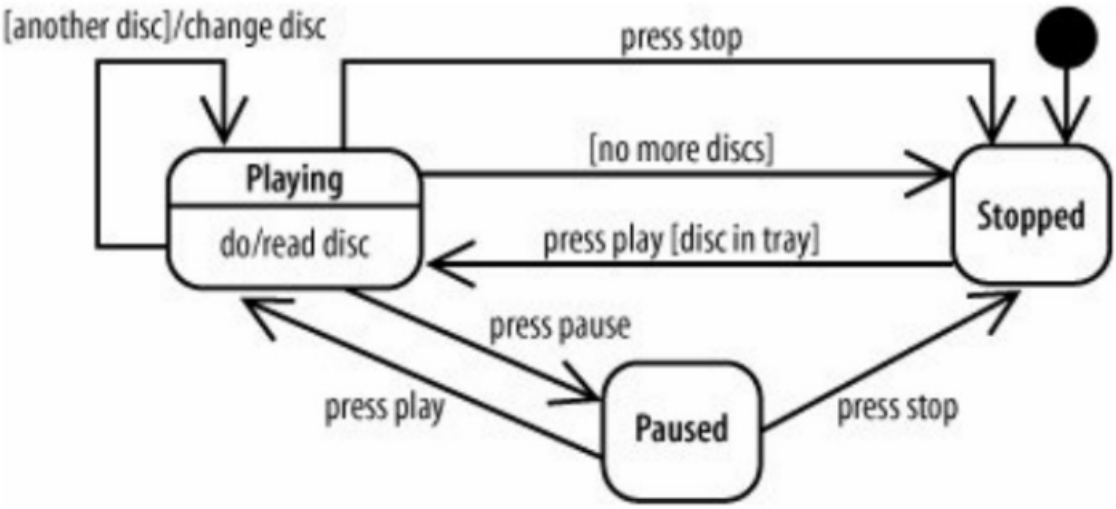
text

Objects transition from source states to target states



A transition syntax: trigger [guard] / behaviour

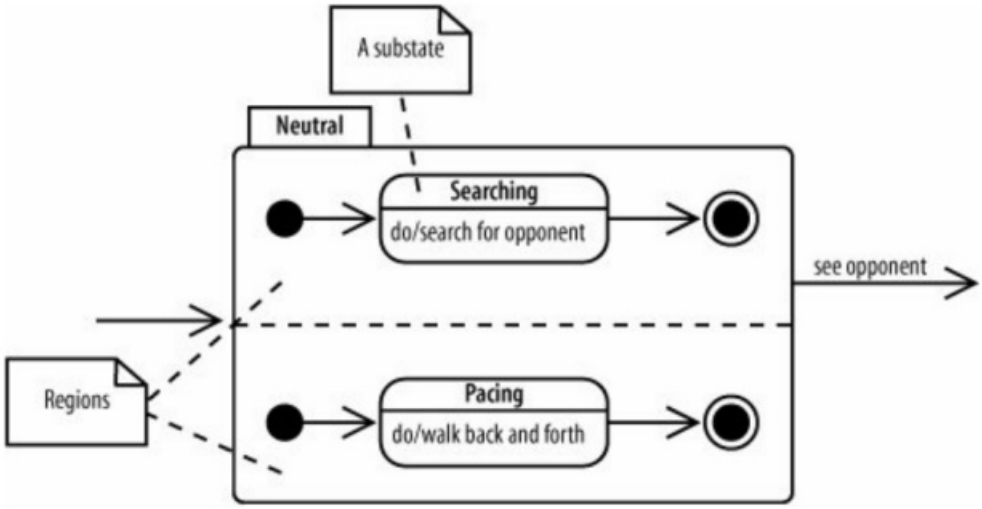
There can also be more than one reasons for a transition



Composite States

UML allows concurrent states to be shown using the composite states notation.

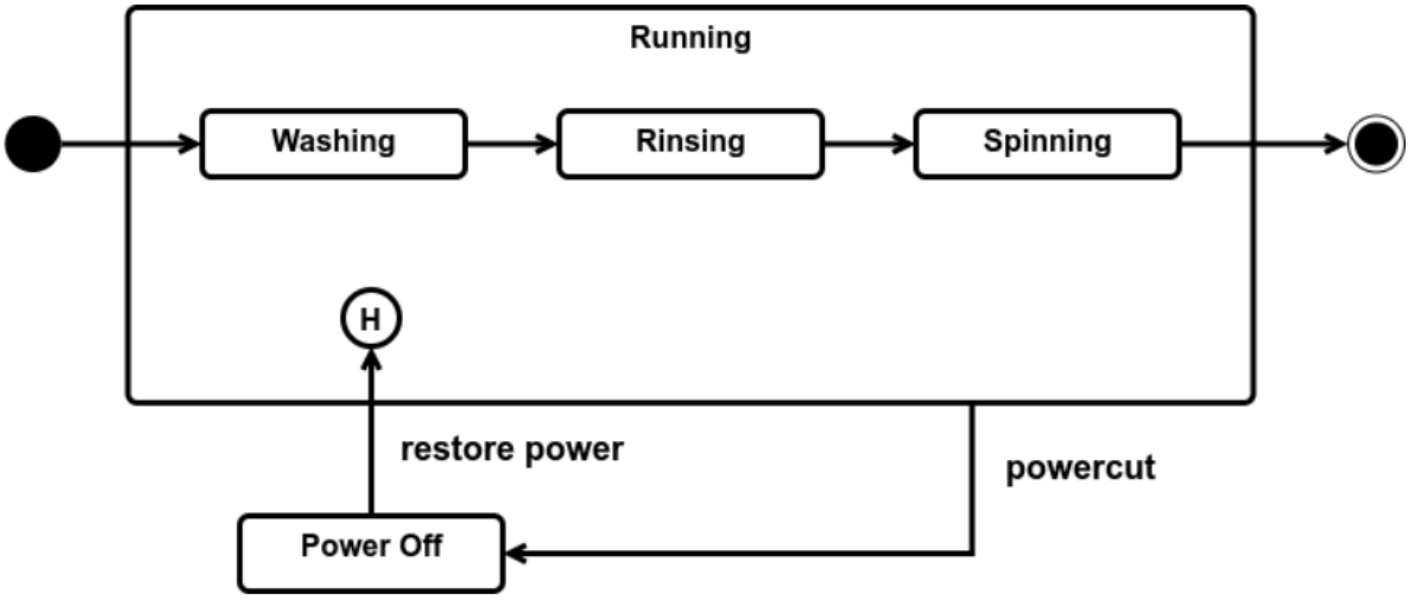
A non-player game character is in the “Neutral” state which consists of two substates “searching” and “pacing”



Even if a substate runs to completion, the composite state is complete when every substate diagram is complete.

History States

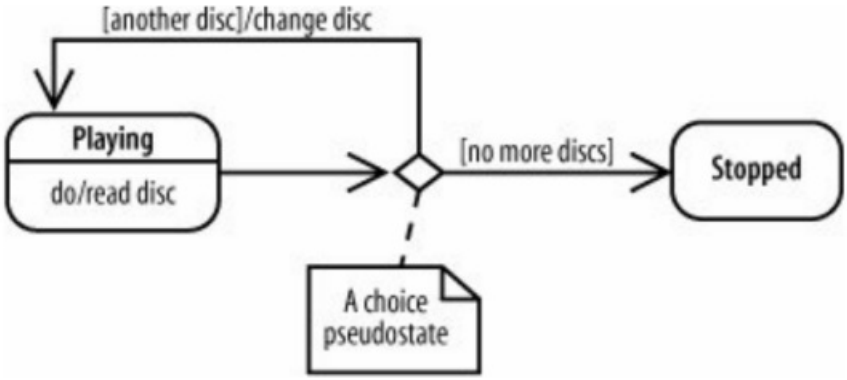
History states are used to remember the state before an interruption has happened



Advanced Pseudostates

UML notation allows chocies, forks and joins to be possbile

text



text

text

text

Summary

Lecture

Revision

Week 5 - Systems Analysis

Objectives

- [ ] this lol

Notes

Systems Analysis

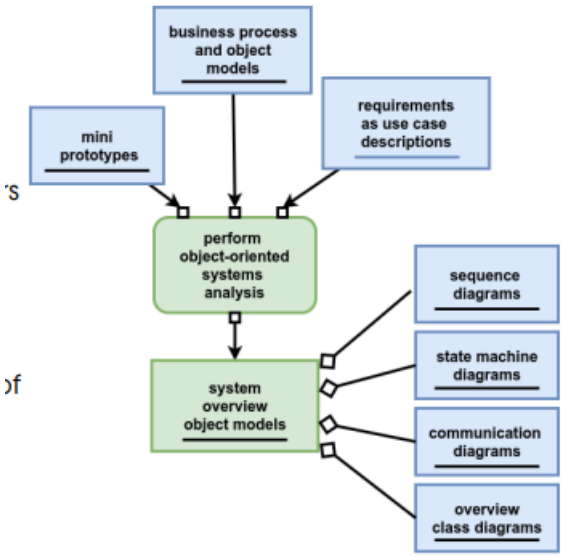
We expect to have collected enough data about user interractions and business domain, to meet the company and user requirements, as well as existing business processes, they will most liekly also have been modelled.

The drafting of technical solutions to our requirements.

The current deliverable is not yet finished but close.

Overlaps with Domain Analysis, but is more technical and definitely a synthesis activity.

We need to start drafting technical solutions to meet the requirements, but the end result isnt expected to be fully implementable



- Domain Analysis feeds Systems Analysis with information
- Detailed models of the system, modelling of internal machanisms.
- Systems Analysis delivers a more technical solution to meet the requirements
- The business analysis would be a node with a square on the bottom , producing an output , feeding into the business process and object models, requirements and use cases as well as prototypes.
- After this we are expected to produce a deliverable,

which will be a set of models applicable to the problem.

Systems Analyst

The OOP SA's role is to:

- Research problems and determine requirements
- Plan technical solutions for an enhanced or new system
    - Derive an outline design from the requirements
        - Business Analyst
    - The design realises all use cases via actor-object and object-object collaboration.
        - how will the objects collaborate, what are the internal mechanisms implemented by them.
    - Objects of the design have a specific responsibility that can be implemented.
- From business processes and their use cases to a comprehensive model of the systems objects, behaviours and interfaces.

OOP Systems Analysis

Software based object orients modelling was first introduced to simulate real-wrold entities (simula in the 60's)

- Real-world entities : Software objects representing the entitites (e.g. employees, students, orders, goods)
- Attributes of entitites: object fields (e.g. employee name).
- Bheaviours:
    - Autonomous behaviours - objects own thread executes objects private methods
    - Object interactions - signals sent between objects

The essence of OO models: objects interact by passing messages to each other.

UML Sequence Diagrams

UML Sequence Diagrams provide the necessary notation to draw objects and sequences of interactions between them.

Provides all annotation to sraw objects and the sequence of messages they send to each other

It is a two-dimensional diagram: objects are arranged along the top of the diagram, and the interactions are listed underneath in time order going down the page.

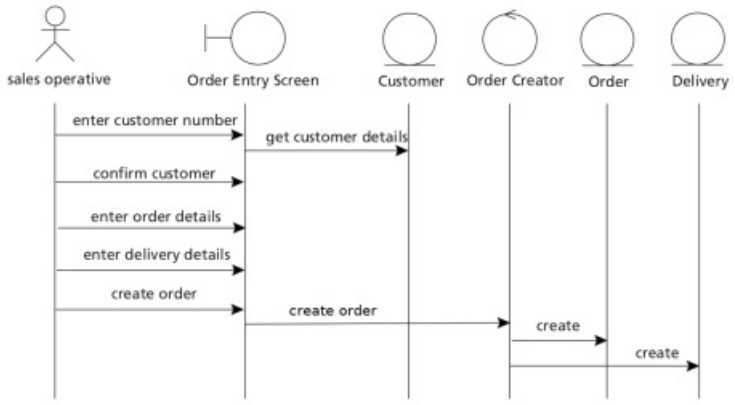
Using analysis, they help us tie together the sequences in use case descriptions.

Notation for three types of objects available.

Example

We have a 2D diagram , one dimension is time and the other is the objects.

The stick figure is an Actor, which *usually* initiates interractions between objects.



The time doesn't have to be strictly measured in a specific unit, it can be any.

The UI is a boundary object.

The order creator is a control object, the order and delivery are entity object , the CO manipulates the entity objects.

Objects in sequence diagrams

Boundary objects:

- Facilitate interactions with the outside world (usually an UI and API of some sort)
- Manage the dialogue between an actor and the system
- Don't store any data, but may wrap it appropriately
- At least one is always present in a sequence diagram
- They may check or fill in incorrect data and send it to other objects.

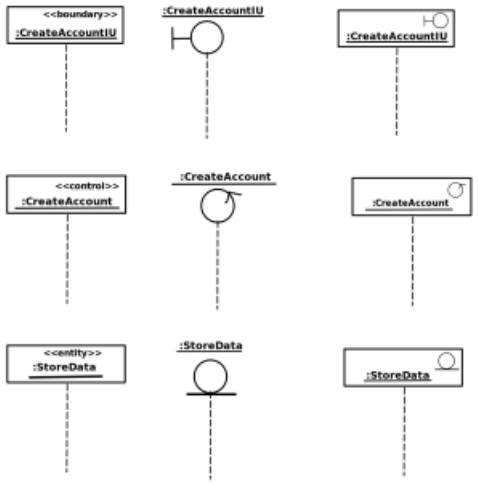
Control Object

- Realise use cases and organise complex behaviours
- interact with boundary objects to send / receive , input / output and interact with actors
- Something that allows us to *realise* behaviours defined in use cases, but they don't reflext come entity in the real world
- We might want to create a list of products or a card on the webpage, we can populate the card with products, the card is a control object and the product is an entity object.
- Used to represent a concept that doesn't exist in the real world.
- We expect them to interact with boundary objects.

Entity Object:

- Model a store or persistence mechanism that casptures data in the system.
- They are the classes that we draw inspiration from to

Notation of Objects in SD



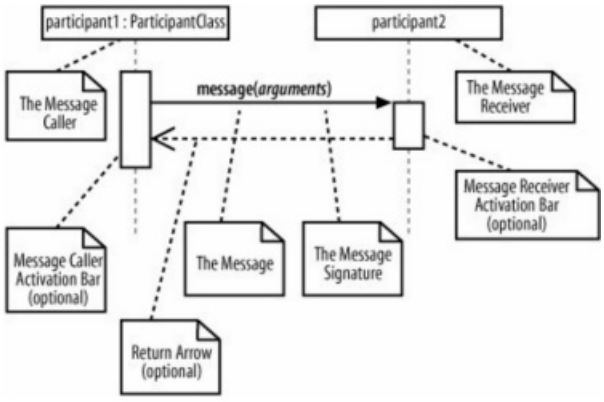
The first row is all about boundary objects

Control objects

Entity Objects

UML Sequence Diagrams Activation Bars

Activation bars indicate that an object is active (doing something) , but can be omitted

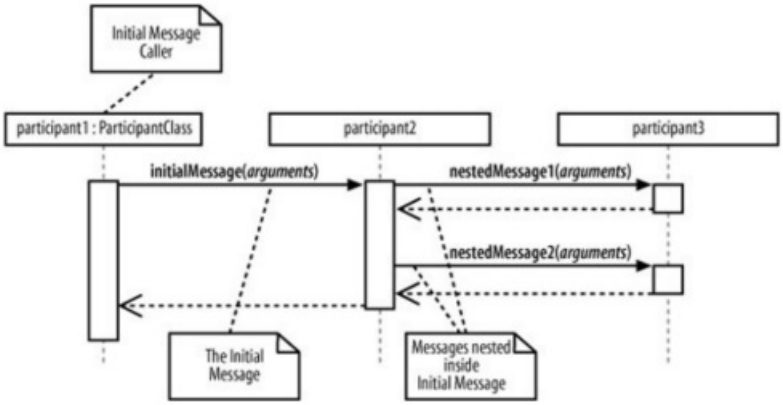


This diagram looks complicated but is just heavily labelled.

The tall gate represents the activation mode, participation in discussion with other objects b during its lifetime, at this point in time

Messages Example

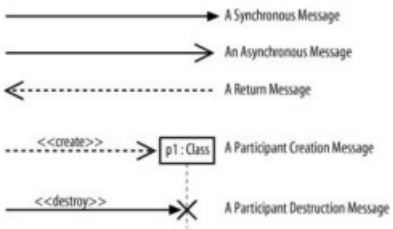
Nested Messages are allowed:



Remember time is flowing down.

There are different types of messages that can be shown:

- Synchronous Messages : caller waits until the receiver finishes and returns
    - They require a return message, the sender has to wait for the return.
- Asynchronous messages : fire and forget
    - If we don't want that behaviour, and want the object to continue operating we send Asynchronous messages.

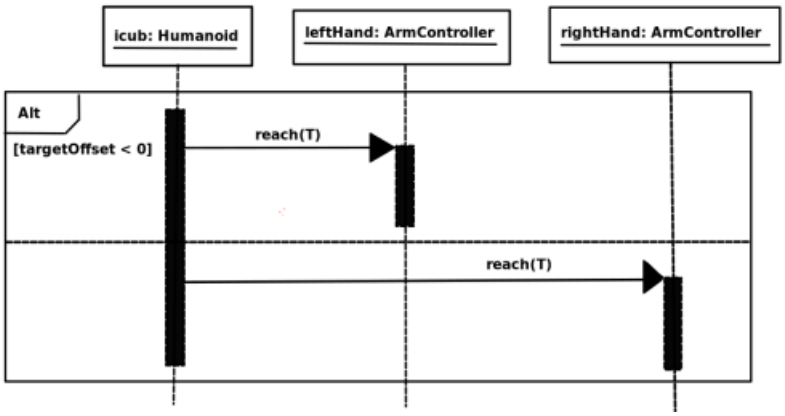


When we see create in this notation we tell a class to create a new object.

UML Sequence Diagrams Alternatives

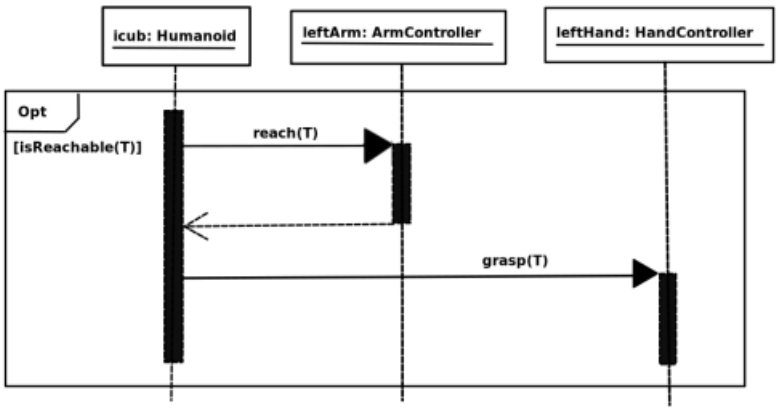
Alternative flows show a choice of behaviour in a workflow. A guard is used to elevate a choice.

Only one of the options is executed:



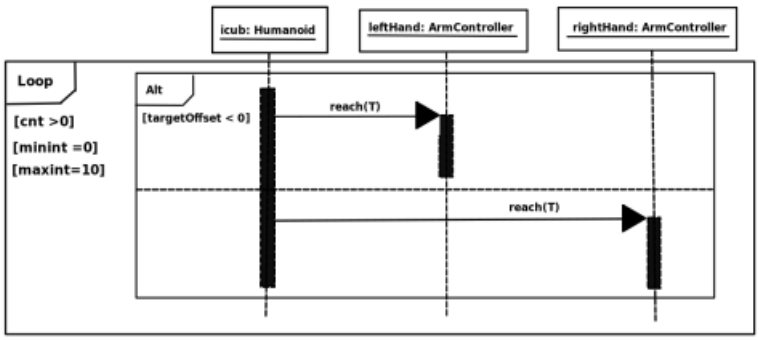
UML Sequence Diagrams Optionals

Optional fragments are only executed when their condition is true:



UML Sequence Diagrams Loops

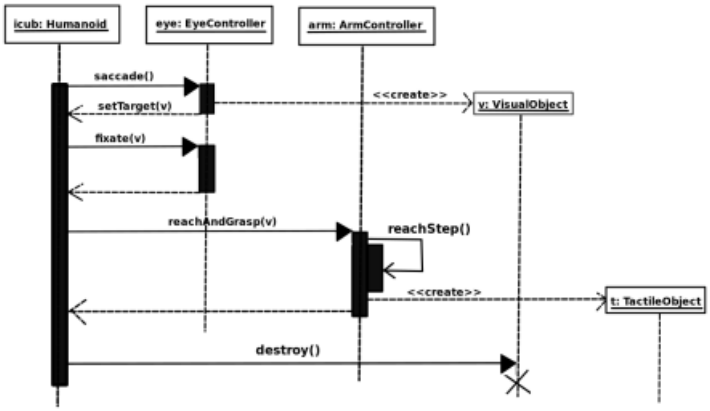
Loop fragments depict repetitive sequences



Fragments ncan be combined to communicate the appropriate models

UML Sequence Diagrams Reflexive messages

Reflexive calls allow an object to send messages to itself



The robot sends the set accade message to the eye controller ,

at some point it sees an interesting object , so it sends a create message to create an entity object representing something in the real world. When you create an object you don't expect a returns arrow, we get a reference to where it is in the memory.

When it "returns"

When we send the reach messages to the arm, we expect it to happen in phases, like extend, expecting several steps. We can use the activation bar to show the activation abr call an internal private method to do the reaching.

After gaining the tactile information , we can destroy the tactile object.

Deriving Objects from use cases

Process:
For each use case:

1. Pick a few representative scenarios, i.e., primary, alternative and exception paths.
2. Express each scenario/path as a sequence diagram, identifying participating domain and system objects and their interactions.
3. Rework the sequence diagrams as communication diagrams.
4. Combine all communication diagrams to derive a outline/overview class diagram.

from a large amount of use case scenarios we pick the ones with high complexity and incorporate other use cases.

We then try to understand what the business analyst has written

Starting from use case description

Primary path:

1. The sales operative takes the customer number and enters it on the screen.
2. The customer details are retrieved and displayed on the screen.
3. The sales operative checks that the customer details match those given by the customer, and ticks a confirm box.
4. The sales operative enters the order details.
5. The sales operative enters the delivery details.
6. The sales operative requests that the order is created.

Determine Actors and Objects

Actors - Sales Operative

Boundary Objects - Screen to display details

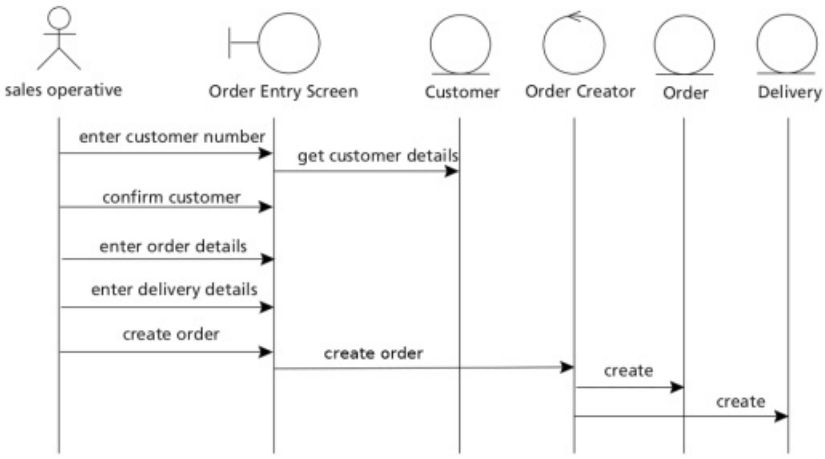
Entity Objects - Customer, Order, Delivery

Deriving a Sequence Diagram

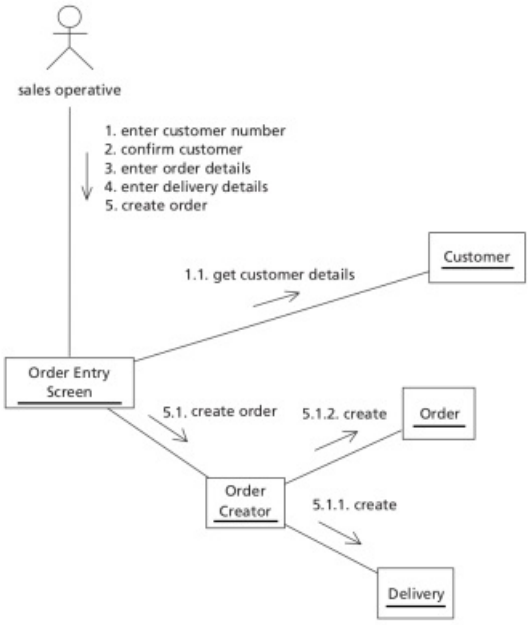
Draw a lifeline on the left of the initiating actor.
Draw next to it lifeline(s) for the actor's boundary object(s).
Draw (generously spaced) interactions between actor and the boundary object(s).
For each boundary object interaction determine whether there is a need to involve another object.

- If the interaction is about a single entity object, create it
- If the system needs to perform an action that involves more than one entity object, consider creating a control object to manage the action.

Resulting Sequence Diagram



Sequence diagrams to communication diagrams



- UML Communication diagrams are another way of interrelating objects in sequence diagrams
- Numbering interactions in sequence diagrams translates them easier to communication diagrams
- Easier to derive class from communication diagrams

Communication Diagrams to Class Diagrams

Knowing the objects, one can derive a UML class diagram and add further details such as attributes, associations and multiplicities.

Note that some of these details may need clarification with the business analyst/stakeholders.

Aggregating all Models

The process of taking your design from use case and use

case descriptions, to sequence diagrams, to communication diagrams and ultimately to class diagrams is iterative.

▸ At the end, we try to aggregate all the generated models into one outline class model of the software system.

▸ Nothing stops you from using other UML diagrams to generate your models. E.g., state machine diagrams are often used in Systems Analysis to model Actor — GUI interactions.

Preparing for Tutorial 5

Preparation:

Week 6 - Software Testing

Objectives

- [ ] This lol

Notes

What is software testing?

The process used to *validate* a software application by:

- Observing
- Recording
- Comparing

The behaviour (output) of the software with the intended behaviour and output.

The main purpose is to find errors.

Testing is all about making sure software meets requoirements, it responds as expected without suprises. Does it work under all possible circumstances? what is and isnt expected.

the main purpose is to ensure there are no “bugs”.

When do we Test?

Testing should be done early and often , whenever a change has been made, ass even a small change on a piece of code may *invalidate* the entire system.

We don’t have to wait until the program is fully implemented.

We start from an incomplete syste, some components are nto finished and soem are missing so we’re expected to emulate some of it.

Testing is most effective when we capture client requirements.

Test Data

Test data should test the software at its limits and test the business rules.

We should include:

- Extreme values
  - E.g. out of bounds
- Borderline Values
  - -1 , 0 , … , 999
- Invalid combinations of values
  - Virgin : yes , kids = 3
- Nonsensical values
  - Negative quantities of orders.
- Heavy loads
  - Are performance requirements met.

We expect tests to be repeatable so we need a lot of data to describe them.

Test data should be there to test the limits of the business rules (describe the requirements set by clients).

Who should perform the testing

Ideally, by specialist test teams who have access to custom software.

The business (analyst) and systems (analyst) should also test the software.

In eXtreme Programming (XP) programmers are expected to write test harnesses for classes before they write the code.

Intended users of the system should test against requirements (user acceptance testing).

Types of testing

Black Box testing

Establishes whether the end product does what it needs to as fast as it should.

Checking FR and is usually done by Systems Analysts.

The idea that each component or system itself , or function , we don't have access to the internal mechanism but we can pass some input to a component and we expect a certain output.

White box testing

seeks to establish whether the end-product delivers a good solution as opposed to just solving a problem.

Checking NFR, usually done by the developers.

Make sure the system / algorithm doesn't have any unnecessary complexity.

We can do this because we have access to the code.

Various Types of testing

Unit Testing

Ensures that individual classes work correctly

Lowest level kind of testing

Interface testing

Validates the functions exposed by modules

How do components communicate with each other.

Integration Testing

Establishes whether all classes in the system work correctly together

After interface testing, we make sure that when joined, components work. we test them simultaneously and make sure there are no errors.

Subsystem testing

Checks if each subsystem works correctly and delivers the required functionality

Make sure that some part of the system works wihtout considering interractions with other parts of the system (focusing on a specif part of the system).

System testing

Checks if the whole system works seamlessly

Testing the WHOLE system

Acceptance testing

Checks if the system works as required by user and according to specification

We don't validate any internal mechanisms, but rather ask end users if the requirements are met and what their opinion is.

Usability testing

Tests if the intended users are satisfied with the software application in addressing its intended purpose.

All about user experience, is the user able to work and be satisfied by the system provided.

Is the user experience met.

Regression testing

Ensures that changes made to the source code has not caused defects in the existing source code.

Umbrella term that describes all the methods and tests woring together to ensure the system is still functional if the system is changed

Installation testing

Checks how well the software application works on the required platform

Robustness testing

Establishes the ability of the software application in handling anomalies (being trustworthy)

Can be things related to the external environment and how it affects the system.

Will the system work equally well under different circumstances

Performance testing

Check if the system is fast enough and it uses acceptable amount of memory (under stress).

Fold injection

Going into the code and manually changing values to cause errors to see hwo they are handled.

Levels of Testing

Level 1

Testing modules (i.e. classes) then program (use classes) then suites (i.e. the application)

Mostly about unit tests. We test units as well as modules.

We start from 0 progress in implementation adn we start building important components and we want to start testing them.

We dont have enough classes yet and the system is not complete.

Level 2

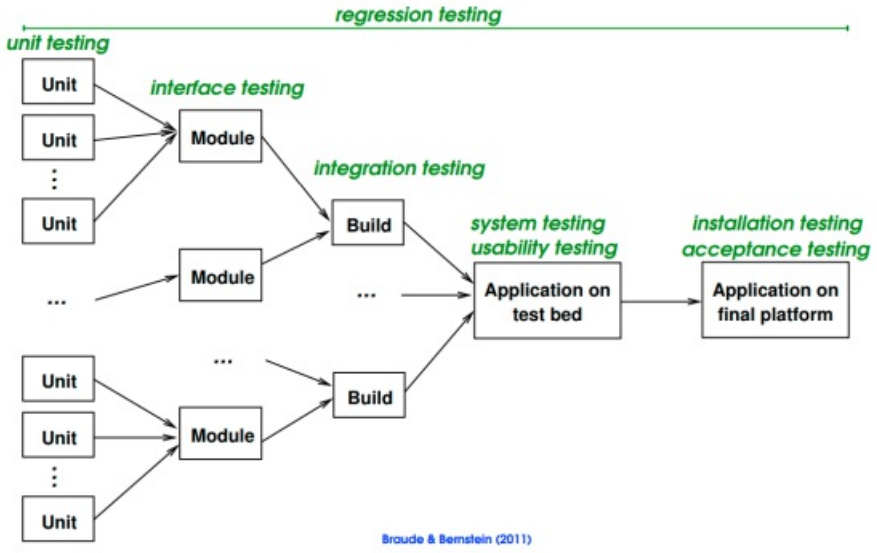
(*Alpha testing or verification)* Execute programs in a simulated environment and test inputs and outputs

We want to see if the programs inputs / outputs work , we run it in a simulated environment, the system is not ready for release yet.

Level 3

(Beta Testing / Validation) test in a live use environment and test for response times, performance under load and recovery from failure etc.

Summary of Testing Activities



Regression testing

A process in which the software is **retested** so as to ensure that its behaviour hasn't been compromised by altering or adding new changes to the code.

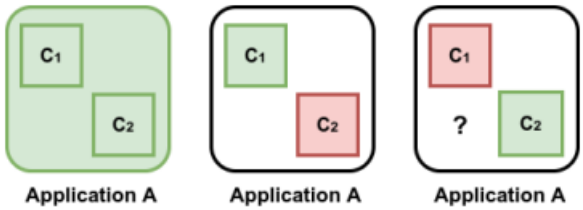
It facilitates faster development:

- It reduces risk of changes.
- Unexpected effects can be found quickly.
- Ability to run thousands of tests in one click.

Regression testing can be very time consuming

(requires capturing data, analysis, reporting etc)

Consider the following:



When should we do regression testing?

(you're retarded haha)

Test Documentation

Contains plans cases and results. When we're given a problem and start doign domain analysis and elicit requireemnts we should start thinking about the plan for the testing

Test Plans

We plan the testing before we start writing the code.

- Written before the tests are carried out.
- Written before the code is written
- Contain test cases

Test Cases

- Test data
- expected outcomes
- description of the test
- test environment and configuration

Test Results

- Used to faciliatate analysis ideally test results should be recorded in a spreadsheet, dataabse or a specialist package like Jira
- Recorded when tests are failed or passed
- Allow reporting percentage passed
- Ideally should be linked to reqirements, showing whether or not each requirement has been met.
- Error results should be recorded in a fault reporting package with enough details for developers to reproduce them with a view to fixing the bugs.

Test Planning steps

1. Define what is meant by a "testing unit"

2. Determine the types of testing to be performed.
3. Determine the extent of testing (i.e., prioritise the tests).
4. Determine how to document the testing procedures.
5. Determine input sources.
6. Decide who will perform which tests.
7. Estimate resources.
8. Identify metrics to be collected.

Determine the Extent of Testing

Software should be thoroughly tested before release, but when do we stop?

(when we're dead lol)

The testing team should establish a set of stopping criteria:

- A tester is unable to find another defect in n minutes of testing
  - (where n may be any +ve integer, e.g., 5, 10, 30, 100).
- When all nominal, boundary and out-of-bounds test data show no defect.
- When a given checklist of test types has been completed.
- When testing runs out of its scheduled time (DANGEROUS).

Unit Testing

Testing applied to parts of an application (classes and their methods) in isolation to the application.

White Box Unit Testing

Test cases cause every line of code (statement) all conditions (branch) and all possible flows (path) to be executed, from entry to exit, to be executed,

Statement coverage - test cases cause every line of code to be executed.

Branch coverage - test cases cause every decision point to be executed

Path coverage - test cases cause every independent code path to be executed.

Black Box Unit Testing

Equivalent Partitioning - divide input values into equivalent groups and checking if the output is as expected.

Boundary value analysis - test at boundary conditions

Path coverage Example:

text

```
1 private static void showResult(int guessed, int toGuess) {
2 if (guessed == toGuess) {
3 System.out.println("Well done! You guessed it!");
4 }
5 else if (guessed > toGuess) {
6 System.out.println("Sorry. Your guess is too high.");
7 }
8 else {
9 System.out.println("Sorry. Your guess is too low.");
10 }
```

```
11 System.out.println("The number I have in mind is %d\n", toGuess);
12 }
```

A test case for each ***distinc path*** is:

`showResult` (8,8): 2 → 3 → 11

`showResult` (9,8): 2 → 5 → 6 → 11

`showResult` (7,8): 2 → 5 → 8 → 9 → 11

Unit testing Checklist

Black box unit testing:

1. Verify operation at normal parameter values.
2. Verify operation at limit parameter values.
3. Verify operation outside parameter values.

White box unit testing:

1. Ensure that all instructions execute.
2. Check all paths, including both sides of all branches.
3. Check the use of all called objects.
4. Verify the handling of all data structures.
5. Verify the handling of all files.
6. Check normal/abnormal termination of all loops.
7. Check normal/abnormal termination of recursions.

Using Assertions

Assertions are statements in Java which ensures the correctness of assumptions.

They allows us to enforce business rules and catch any mistakes

Also allow us to check for parameter and return values

```
1 int age = 16;
2 assert age <= 17 : "Cannot drive";
3 System.out.println("Can drive at " + age);
```

They allow us to check for things that should happen.

Unit testing with JUNIT

JUnit is a framework for unit testing which facilitates the writing of repeatable tests.

To use JUnit 5, typically one imports:

- `org.junit.jupiter.api.Assertions`
- `org.junit.jupiter.api.Test`

Annotates a test method with a @Test

Uses assert- methods, i.e.,
`assertFalse`, `assertTrue`, `assertEquals`, `assertThrows`, etc.

```
1 @Test
2 public void testAddingStaff() {
3 StaffManager sm = new StaffManager();
4 sm.addStaff("Lucy", "Bastin");
5 Assertions.assertFalse(sm.getAllStaff().isEmpty());
6 Assertions.assertEquals(1, sm.getAllStaff().size());
7 }
```

JUNIT Test Licefycle Features

Fixtures allow you to set up a testing environment

(prepare a baseline for your tests).

Usually fixtures have the following lifecycle:

```
1 @BeforeAll
2 @BeforeEach
3 @Test
4 @AfterEach
5 @AfterAll
```

You may want to use `Before` annotations for the preoaration of test data.

You may want to use `After` annotations for housekeeping

You may want to use `BeforeEach` to run a command before each method.

Test-Driven Development

A different approach to writing software.

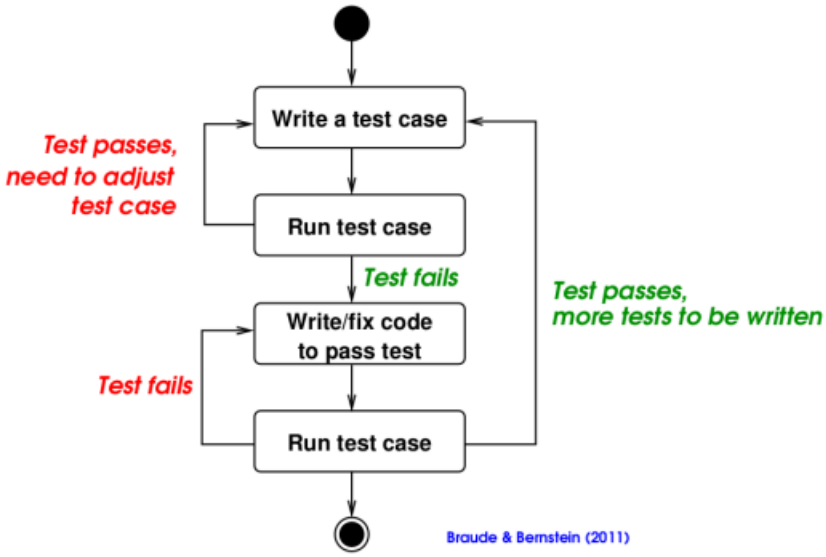
The development of the application is thus driven by the test cases solely.

Writing test cases before writing any source code.

Code that does not address the issues in the test cases will not be written.

A testing framework such as JUnit can be used to facilitate TDD.

Process



Advantages

Test cases ensure a good statement coverage.

Only code required to address the requirements is written; no redundancy.

Rapid feedback on the written code.

Test suite available for further testing purposes, e.g., regression testing.

Week 8 design vs analysis and architecture

Objectives

- [ ] this lol

Notes

Analysis identfies *what* the system must do.

Design specifies the *how*.

in analysis the analyst seeks to understand:

- the organisation
- the requirements
- the objectives

in design, the designer seeks to specify a system that will:

- fit the organisation
- meets its requirements adequately
- assist it to meet its objectives

Consider a computer system which supports day to day work of the advertising company agate:

Analysis identfies that there needs to be an (advertising) campaign class which has a title attribute.

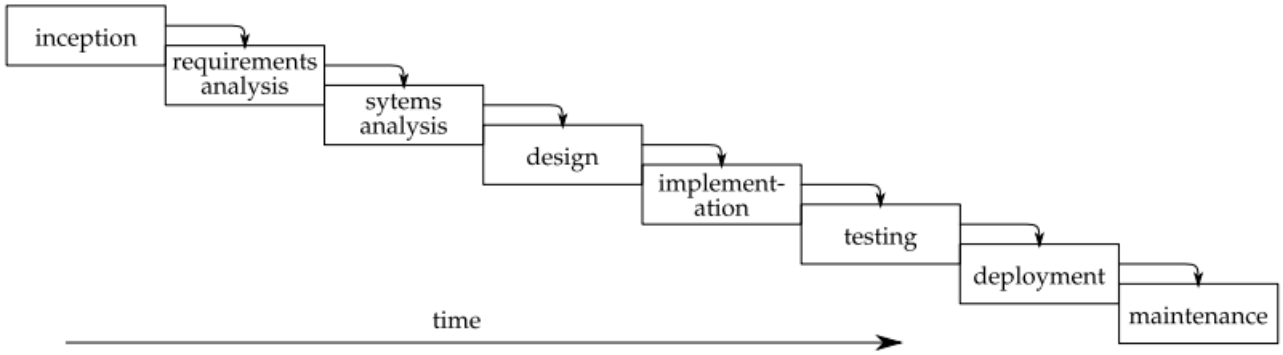
Design determines:

- how the title attribute will be entered into the system,
- how the title attribute will be displayed on screen
- how the title attribute will be stored in a database,
- what other attributes of the class Campaign are
- other classes etc

When does analysis stop and design start?

In a waterfall life cycle, there is a clear transition between analysis and design

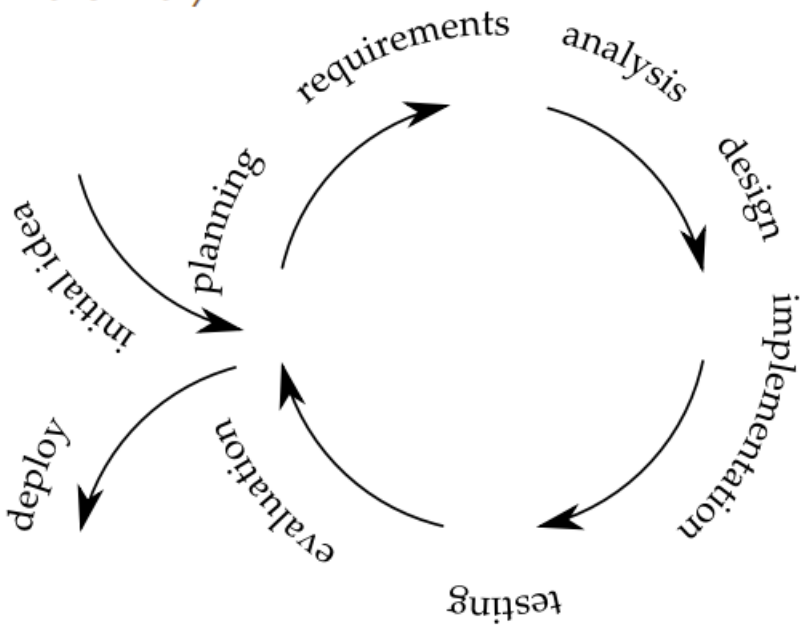
➟ Activities in one stage (analysis, design, construction, etc.) is completed *before* activities in the next stage begins.



When you have the analysis stage complete, you can't go back once you have the design phase, so you start when you're confident nothing will change.

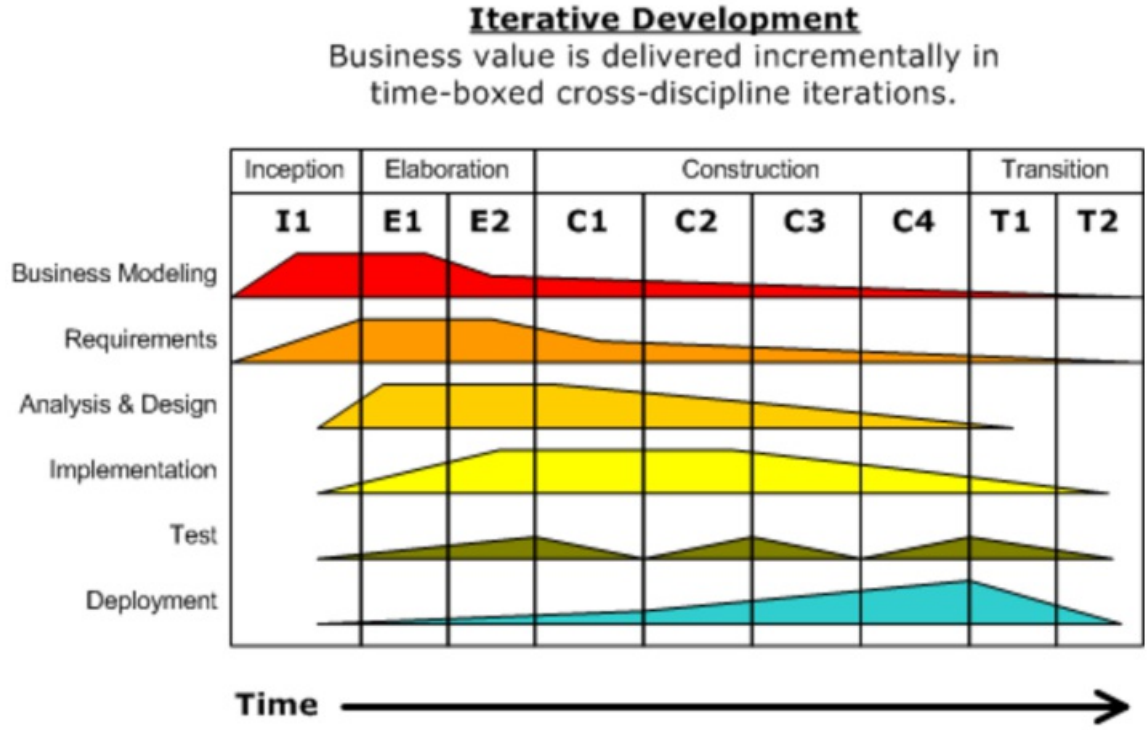
In an iterative lifecycle, the analysis of a particular part of the system will precede its design, but analysis and design may be happening in parallel.

➟ Analysis and design happen throughout the *entire* project development *iteratively*.



In the Unified Process, the amount of analysis and design performed grows from the Inception phase to the Elaboration phase and shrinks in the construction phase.

Each iteration, you're gathering analysis for a system that has changed , hence



Logical and Physical Design

Logicsl Design is independent of the implementation language and platform.

Like a design patternm, its a logical way of doing stuff that can be implmented in any language.

Logical Design of a UI can be done wuthout knowing the implementation langauge

Types of fields, position in windows etc.

Physical Design is based on the actual implementation platform and the language that will be used.

What hardware, software etc

Physical design is specific to the language

Layout managers availible

Separating logical / physical design is useful if the software must be implemented on different platforms.

A platform-independent design that can be tailored as required.

System Design

How granular should it be? how will these subsystems commuinicate and where will we put them , what should be broken down?

Deals with the high level architecture of the system.

Structure of the sub-systems, communication between them, and their distribution processors.

Standards for screen, reports, help.

Detailed Design

Traditional detailed design consists of:

Designing inputs, outputs, processes and file/database structures.

Object-oriented detailed design adds detail to the analysis model:

- types of attributes.
- operation signatures.
- additional classes to handle user interface or data management
- design of reusable components
- assigning classes to packages

Qualities of Design

- Functional - the system will perform its required functions.
- Efficient - the system performs those functions efficiently in terms of time and resources.
- Economical - running costs of system will not be unneccessarily high.
- Reliable - not prone to hardware or software failure, will deliver the functionality when users want it.
- Secure - protected against errors, attacks and loss of valuable data
- Flexible - capable of being adapted to new uses to run in different countries or to be moved to a different platform
- General - general-purpose and portable (mainly applies to utility programs).
- Buildable - the design is not too complex for the developers to be able to implement it.
- Manageable - easy to estimate work involved and to check on progress
- Maintainable - the design makes it possible for the maintenance programmer to understand the designers intention.
- Usable - provides users with a satisfying experience (not a source of dissatisfaction).
- Reusable - elements of the system can be reused in other ... ?

Prioritizing Design Trade-Offs

A designer is often faced with objectives that are mutually incompatible.

Trade-offs have to be applied to resolve these conflicts.

Functionality, reliability and security are likely to conflict with economy

The number of features in the end-product is constrained by the budget availible for the development of the system.

It is helpful if guidelines are prepared for prioritizing design objectives.

If design choice is unclear, users should be consulted

Design objectives may confilc with constraints imposed by requirements.

An example:

- Requirement: The system can be used in different countries by speakers of different languages.
- Entails - Designers will have to agree a list of all prompts, labels and messages and refer to these by some system of naming or numbering
- Trade-Offs - Multilingual support will lead to increases in flexibility but also increase the cost of the design.

Trustworthy Software

Trustworthiness seeks to address the quality and robustness of software, ensuring that software "performs as it should, when it should and how it should".

The trustworthy Software Foundation (TSF) summarises five facets of trustworthiness:

1. Safety - the ability of the software to operate wihout causing harm to anything or anyone.

2. Reliability - The ability of the software to operate correctly
3. Availibility - the ability of the software to operate when required
4. Resilience - the ability of the software to recover from errors quickly and completely
5. Security - The ability of the software to remain protected against hazards posed by malware, hackers or accidental misuse

I Scope for Use (E1)
I Coding Practices (E2)
I Use Tools Effectively (E3)
I Defect Management (E4)
I Artefact Management (E5)
➟ Many of them are well established good software engineering practice.

Design adresses the issue of how to meet the requirements.

Non functional requirements in particular impact on the design.

Measurable objectives often refer to the non-functional requirements of a software development project.

Measurable objectives set clear targets for designers.

Objectives should be formulated ina quantifiable manner so that they can be tested.

Summary and Reading

Notes - Architecture

Architecture defines structure

→ what are the core components, and how do they relate to each other?

Defines behaviour

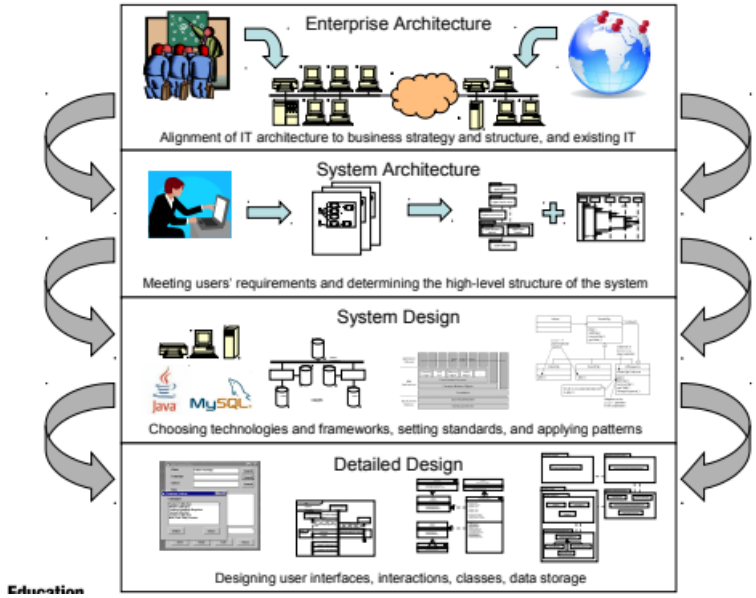
→ How do the core components interact with each other.

Balamces stakeholders needs which may conflict.

Has a particular scope

→ enterprise, system, software, hardware, organisational information

Relationship between Architecture and Design



Enterprise Architecture
Bennett et al. (2010, Section 13.4.2)

- **Enterprise architecture** concerns:
  - the modelling of the *business*,
  - the way the *enterprise* conducts business and
  - how the *information systems* are intended to support the business.
- *Typical questions* to be considered include:
  - How does the system overlap with other systems in the organisation?
  - How will the system need to interface with other systems?
  - Will the system help the organisation to achieve its goals?
  - Is the cost of the system justified?

System Architecture
Key Definitions : Garland & Anthony (2003) and IEEE (2000)

System is a set of co-operating components organised to accomplish a specific function or a set of functions.

→ System architecture describes the hardware and software elements and interactions between them.

System Architect:

Acts on behalf of the client

address the big picture

ensure that the required qualities of the system are accounted for in the design

ensure the required features are provided at the right cost.

What is architecture?

> "A software architecture describes the overall components of an application and how they relate to each other. Its design goals, include sufficiency, understandability, modularity, high cohesion, low coupling, robustness, flexibility, reusability, efficiency, and reliability."
> Braude and Bernstein (2011)

Architecture is not the same as Framework

> "A framework, sometimes called a library, is a collection of software artifacts usable by several different applications. These artifacts are typically implemented as classes, together with the software required to utilize them. A framework is a kind of common denominator for a family of applications. The Java APIs (3D, 2D, Swing, etc.) are frameworks."
> Braude and Bernstein (2011, p.358)

Other examples of frameworks:
- Java: Apache Struts, Spring, JMF, JavaFX...
- PHP: Laravel, Drupal, CakePHP...
- python: Django, Dash, Giotto...
- Javascript, HTML, CSS: JQuery, Bootstrap, React...

Subsystems

A subsystem typically groups together elements of the system that share some common properties.

The subdivision of an information system into subsystems has the following advantages:

- It produces smaller units of development.
- It helps to maximize reuse at the component level.
- It helps the developers to cope with complexity.
- It improves maintainability and portability.

Helps the system more manageable mentally and helps people work on specific parts of the system without affecting the project immediately.

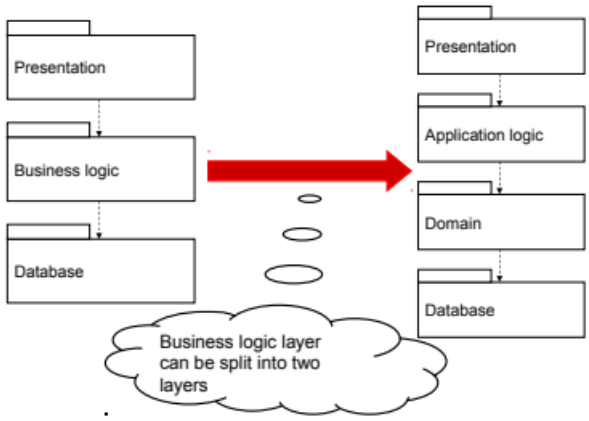
Helps us test specific parts before putting everything together.

Layering and Partitioning:

Two general approaches to divide a software system into subsystems:

- Layering - so called because the different subsystems usually represent different levels of abstraction
  - Abstract functionality
- Partitioning - usually means that each subsystem focuses on a different aspect of the functionality of the system as a whole
  - Actual ?

Both approaches are often used together on one system.



Layered Architecture

- Boundary - classes can be mapped onto the presentation layer (e.g. end user interaction, like forms).
- Control classes can be mapped onto the application logic layer (e.g. I/O, processing inputs).
  - Take actual input
- Entity classes can be mapped onto a domain layer (e.g. data model)
  - Abstract Model of a particular kind of database
- Database layer corresponds to the storage of data within the system (physical storage).

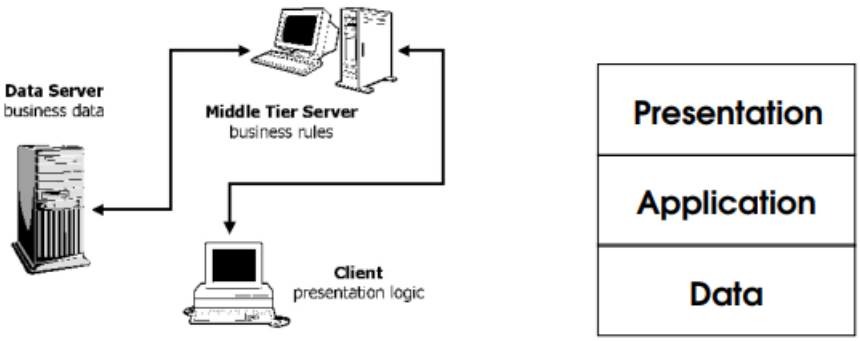
Tiered Architecture

A layered architecutre (logical structuring):

- groups functionality / code based on its level of abstraction.
- All layers may reside on the same computer
- All logical , cna be done on localhost

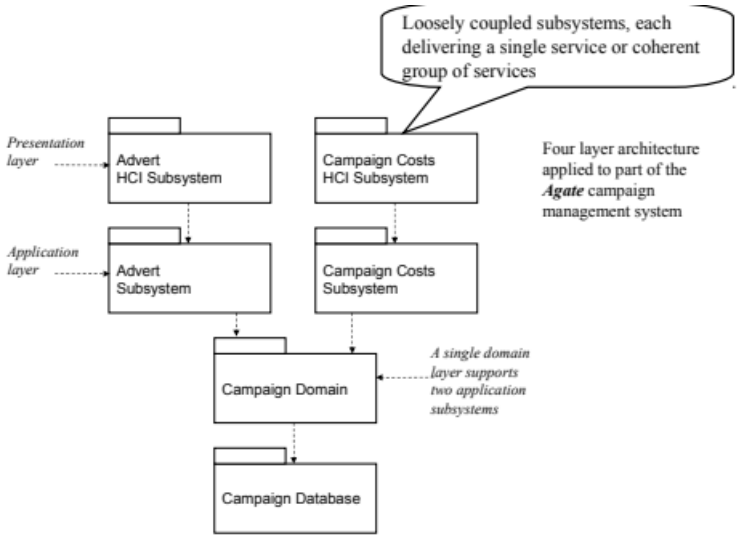
A tiered architecture (physical structuring):

- concerns the logical grouping of functionality / code
- Tiers reside on, and are executed by different computers.
- physical location



Partitioning

The diagram displays a centralised database of advertising campaigns, we then have a sunsystem which will be used by the designer / advertising agents who work with clients in a particular way i.e. multimedia focused.



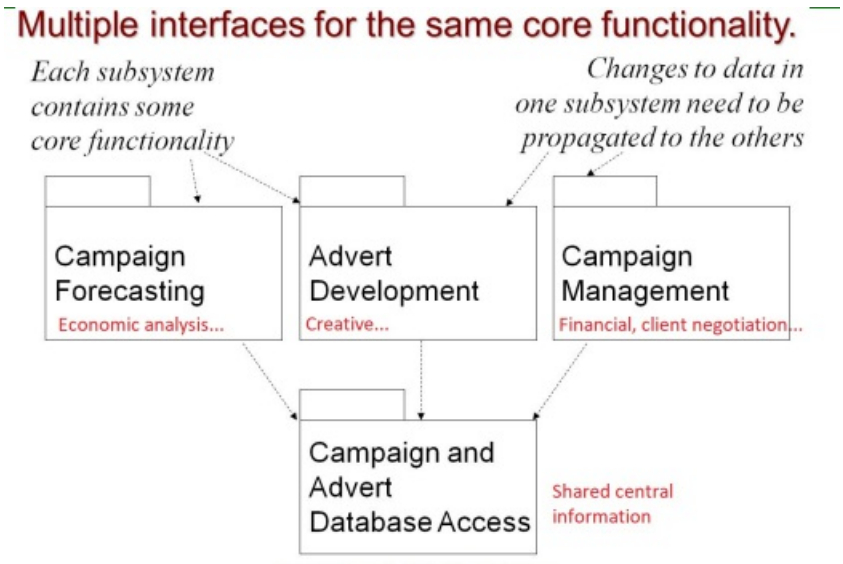
At the same time we have the finance team who needs to look at the cost of these campaigns.

We need these 2 groups to have access to the same resources, we don't want different duplicate versions.

The partitioning happens at the upper level , the database is centralised.

Issues with some architecures

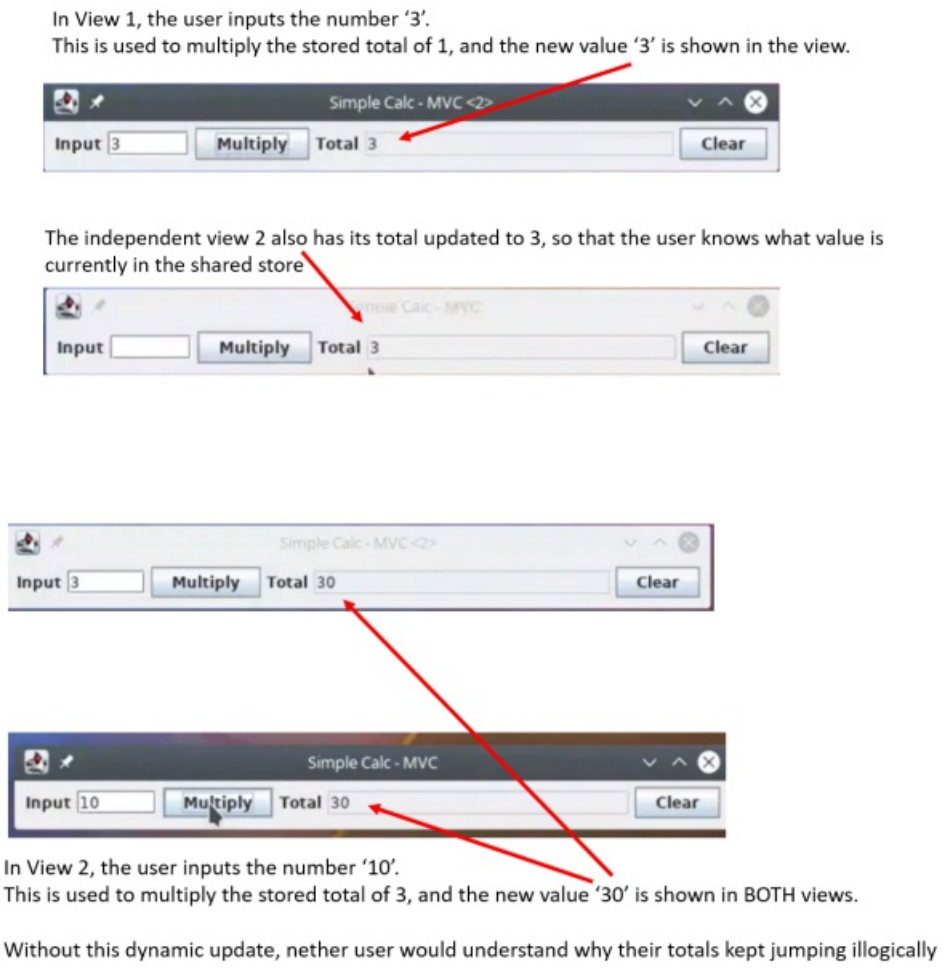
The advert people want a nice clean view without worrying about cost etc , campaign management has different needs and so does forcasting , but it all needs to be stored centrally and up to date for everyone who will have access to this information



The same information should be capable of being presented in different formats in different windows.

Changes made within one view should be reflected immediately in the other views

A simple calculator in MVC

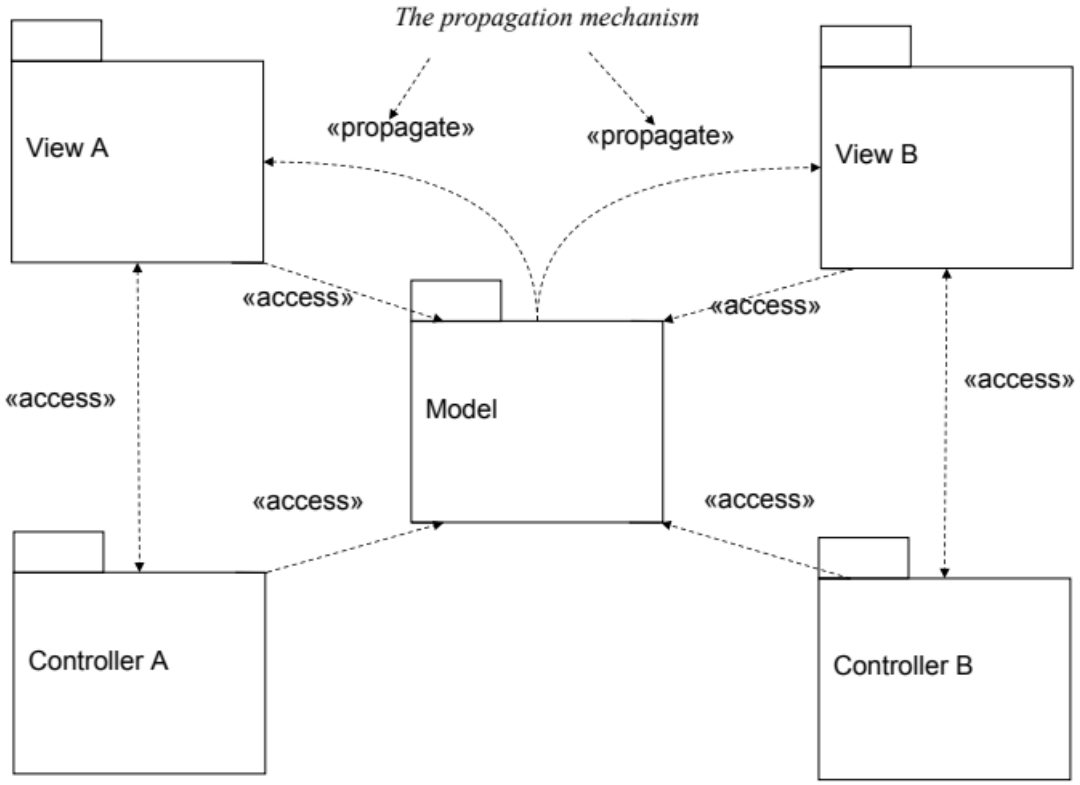


Model View Controler

Bennet et al

propagation Mechanism - enables the model to inform each view that the model data has changed and as a result the view must update itself.

It is also called the dependency mechanism



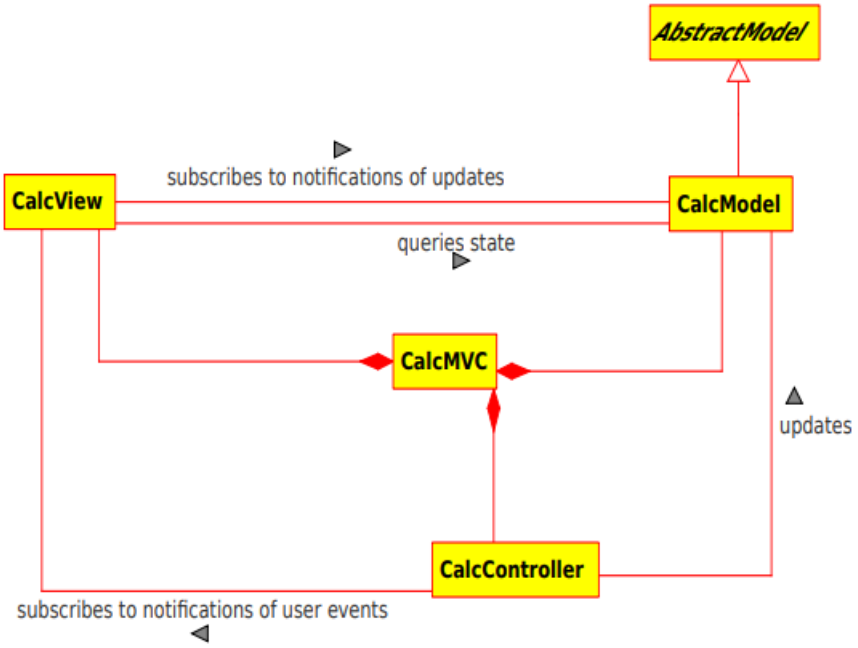
Eckstein

Model - the model represents data and the rules that govern access to and updates this data.

The model is the data , all the code that goes in is allowing access to the data.

View - The view renders the contents of a model and must update its presentation as needed. The view may register itself with the model for change notifications, or may call the model to retrieve the most current data.

Controller - The controller translates the users interactions with the view into actions that the model will perform. In a stand alone GUI client, user interactions could be button clicks or menu selections, whereas in an enterprise web application, they appear as `GET` and `POST` HTTP requests.



The controler needs to know whats happening in the view, so it subscribes changes to that

The view needs to display changes made to the model, so it subscribes to the changes to that

The control adjusts the model according to changes in the view,

Roles of each component

CalcMVC:

- is the top level class
- responsible for creating the *model* , *view* and *controller* objects.

CalcModel

- does not know about the view nor the controller
- Fires property change notifications to its subscribers, in this case the *view*.

I CalcView:

- renders the model using a GUI.
- subscribes to the property change notifications from the model.
- passes user input events to its subscribers.

CalcController:

- subscribes to all user input events.
- defines the event handlers for GUI events.
- calls the model to change its state based on user event.

CalcMVC swartz


```
1 // structure/calc-mvc/CalcMVC.java - Calculator in MVC pattern.
2 // Fred Swartz - December 2004
3
4 import javax.swing.*;
5
6 public class CalcMVC {
7 //... Create model, view, and controller.
8 public static void main(String[] args) {
9
10 CalcModel model = new CalcModel();
11 CalcView view = new CalcView(model);
12 CalcController controller = new CalcController(model, view);
13
14 view.setVisible(true);
15 }
16 }
```

```
1 // structure/calc-mvc/CalcModel.java
2 // Fred Swartz - December 2004
3 // Modified by M Konecny 21-01-2011
4 // Model
5 // This model is completely independent of the user interface.
6 // It could as easily be used by a command line or web interface.
7
8 import java.math.BigInteger;
9
10 public class CalcModel extends AbstractModel {
11 /** name used by property change notification mechanism */
12 public static final String TOTAL_PROPERTY = "Total";
13 private static final String INITIAL_VALUE = "1";
14
15 private BigInteger m_total; // The total current value state.
16
17 /** Constructor */
18 CalcModel() {
19 reset();
20 }
```

2

```
1 /** Reset total to initial value. */
2 public void reset() {
3 BigInteger old_value = m_total;
4 m_total = new BigInteger(INITIAL_VALUE);
5 firePropertyChange(TOTAL_PROPERTY, old_value, m_total);
6 }
7
8 /**
9 * Multiply current total by a number.
10 * @param operand
11 * Number (as string) to multiply total by.
12 */
13 public void multiplyBy(BigInteger operand) {
14 BigInteger old_value = m_total;
15 m_total = m_total.multiply(operand);
16 firePropertyChange(TOTAL_PROPERTY, old_value, m_total);
17 }
```

3

```
1 /**
2 * Set the total value.
3 * @param value
4 * New value that should be used for the calculator total.
5 */
6 public void setValue(BigInteger value) {
7 BigInteger old_value = m_total;
8 m_total = value;
9 firePropertyChange(TOTAL_PROPERTY, old_value, m_total);
10 }
11
12 /** Return current calculator total. */
```

```
13 public BigInteger getValue() {
14 return m_total;
15 }
16 }
```

Abstract Model

1

```
1 import java.beans.PropertyChangeListener;
2 import java.beans.PropertyChangeSupport;
3
4 /**
5 * This class provides base level functionality for all models,
6 * including a support for a property change mechanism (using
7 * the PropertyChangeSupport class), as well as a convenience
8 * method that other objects can use to reset model state.
9 *
10 * @author Robert Eckstein
11 * from: http://www.oracle.com/technetwork/articles/javase/mvc-136693.html
12 */
13 public abstract class AbstractModel {
14
15 /**
16 * Convenience class that allow others to observe changes
17 * to the model properties
18 */
19 protected PropertyChangeSupport propertyChangeSupport;
20
21 /** Default constructor: Instantiates class PropertyChangeSupport. */
22 public AbstractModel() {
23 propertyChangeSupport = new PropertyChangeSupport(this);
24 }
```

2

```
1 /** Adds a property change listener to the observer list. */
2 public void addPropertyChangeListener(PropertyChangeListener l) {
3 propertyChangeSupport.addPropertyChangeListener(l);
4 }
5
6 /** Removes a property change listener from the observer list. */
7 public void removePropertyChangeListener(PropertyChangeListener l) {
8 propertyChangeSupport.removePropertyChangeListener(l);
9 }
10
11 /**
12 * Fires an event to all registered listeners informing them
13 * that a property in this model has changed.
14 */
15 protected void firePropertyChange(String propertyName,
16 Object oldValue, Object newValue) {
17 propertyChangeSupport.firePropertyChange(propertyName,
18 oldValue, newValue);
19 }
20 }
```

CalcView

1

```
1 // structure/calc-mvc/CalcView.java - View component
2 // Presentation only. No user actions.
3 // Fred Swartz - December 2004
4 // Modified by M Konecny 21-01-2011
5
6 import java.awt.event.*;
7 import java.beans.PropertyChangeEvent;
8 import java.beans.PropertyChangeListener;
9 // other import statements omitted
10 public class CalcView extends JFrame {
11
12 private CalcModel m_model;
13 // other field definitions and initialisation omitted.
14
15 /** Constructor */
16 CalcView(CalcModel model) {
17 // ... Set up the logic
18 m_model = model;
19 // ... Initialize components
20 m_totalTf.setText(m_model.getValue().toString());
21 m_totalTf.setEditable(false);
```

2

```
1 // ... Setup automatic updates of the total from model:
2 m_model.addPropertyChangeListener(new PropertyChangeListener() {
3 // will be executed by the model when an event is fired there.
4 public void propertyChange(PropertyChangeEvent evt) {
5 if (evt.getPropertyName().equals(m_model.TOTAL_PROPERTY)){
6 m_totalTf.setText(evt.getNewValue().toString());
7 }
8 }
9 });
10
11 // other GUI code omitted
12 }
13
14 // other GUI methods omitted
15
16 void addMultiplyListener(ActionListener mal) {
17 m_multiplyBtn.addActionListener(mal);
18 }
19
20 void addClearListener(ActionListener cal) {
21 m_clearBtn.addActionListener(cal);
22 }
23 }
```

CalcController

1

```
1 // structure/calc-mvc/CalcController.java - Controller
2 // Handles user interaction with listeners.
3 // Calls View and Model as needed.
4 // Fred Swartz - December 2004, modified by M Konecny 21-01-2011
5
6 import java.awt.event.*;
7 public class CalcController {
8 // The Controller needs to interact with both the Model & View.
9 private CalcModel m_model;
10 private CalcView m_view;
11
12 /** Constructor */
13 CalcController(CalcModel model, CalcView view) {
14 m_model = model;
15 m_view = view;
16 //... Add listeners to the view.
17 view.addMultiplyListener(new MultiplyListener());
18 view.addClearListener(new ClearListener());
19 }
```

2

```
1 ///////////////// inner class MultiplyListener
2 /** When a multiplication is requested.
3 * 1. Get the user input number from the View.
4 * 2. Call the model to multiply by this number.
5 * If there was an error, do nothing.
6 */
7 class MultiplyListener implements ActionListener {
8 public void actionPerformed(ActionEvent event) {
9 try {
10 m_model.multiplyBy(m_view.getUserInput());
11
12 } catch (Exception e) {
13 // ignore the exception, ie ignore the event
14 }
15 }
16 }//end inner class MultiplyListener
```

3

```
1
2 ////////////////////// inner class ClearListener
3 /** Reset model. (View will update automatically)
4 */
5 class ClearListener implements ActionListener {
6 public void actionPerformed(ActionEvent e) {
7 m_model.reset();
8 }
9 }// end inner class ClearListener
10 }// end of class CalcController
```

MVC single model multiple views

The MVC architecture is flexible

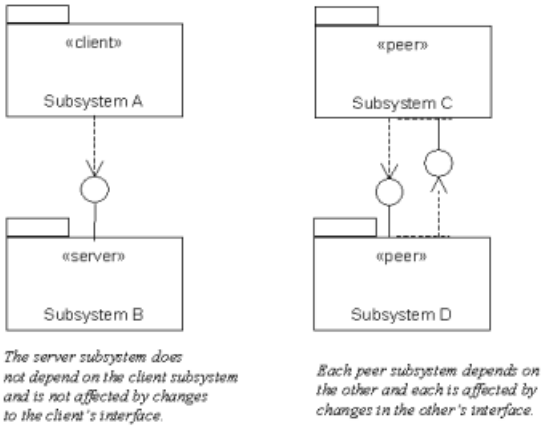
Multiple views can be created in the MVC architecture

Changes made to the model within one view is reflected immediately to all views which use the same model

```
1 import javax.swing.*;
2 public class CalcMVC {
3 public static void main(String[] args) {
4 CalcModel model = new CalcModel();
5 CalcView view1 = new CalcView(model);
6 CalcController controller1 = new CalcController(model, view1);
7 CalcView view2 = new CalcView(model);
8 CalcController controller2 = new CalcController(model, view2);
9
10 view1.setVisible(true);
11 view2.setVisible(true);
12 }
13 }
```

Each subsystem provides services for other subsystems

There are two different styles of communcation that make this possible : Client-server and peer to peer communication



Clinet-server communcation

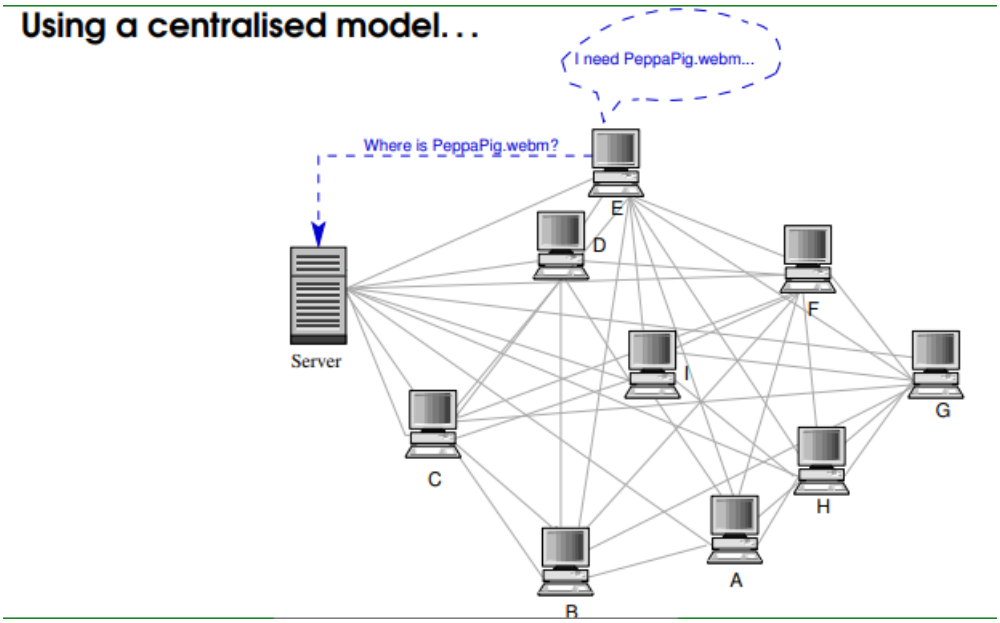
I Client-server communication requires the client to know the interface of the server subsystem.
➟ The communication is only in one direction.
I The client subsystem requests services from the server subsystem and not vice versa.
➟ The client plays the role of a consumer; while the server is considered to be the supplier.
I Examples of client-server communication can be found in most network-based applications.
➟ E.g. email systems and most web applications.

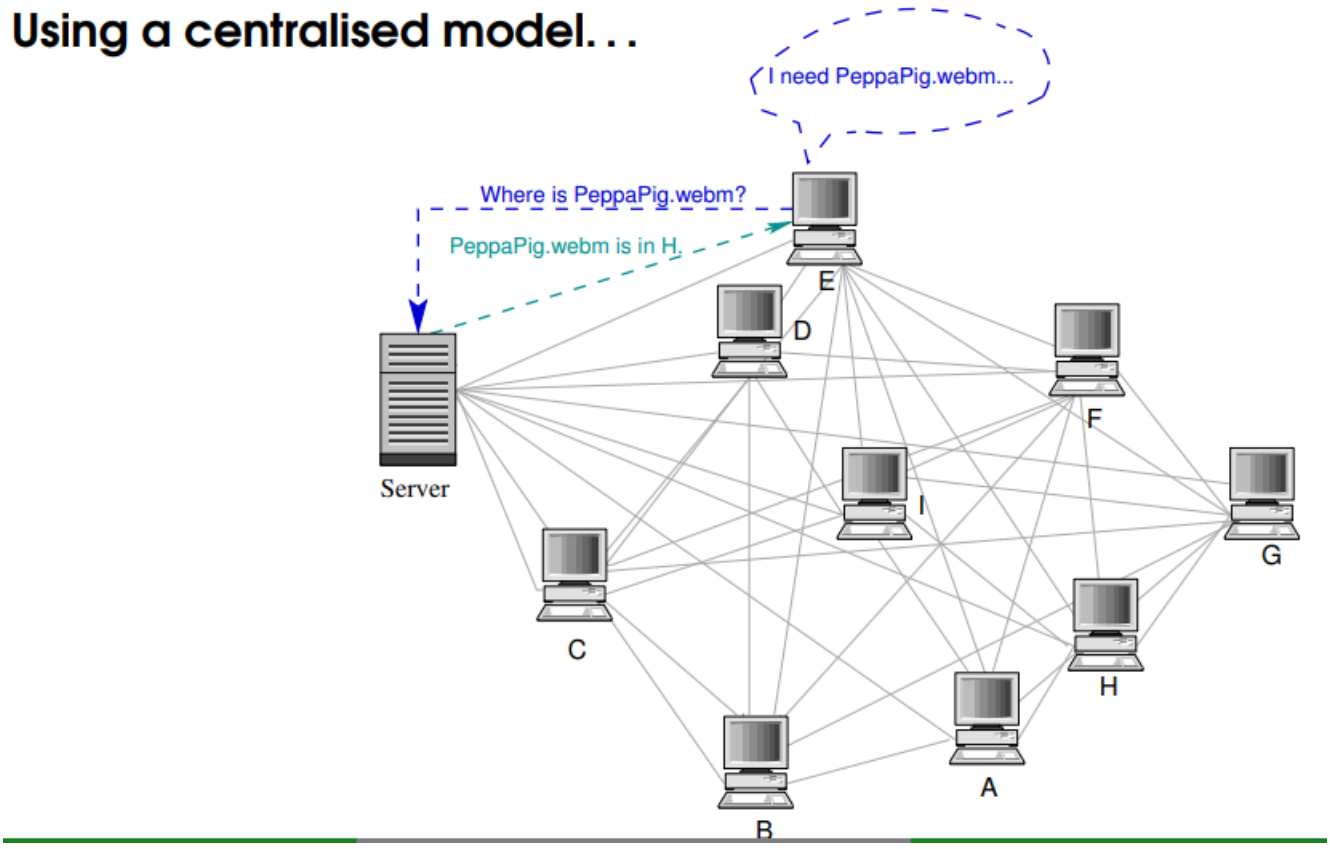
Peer-to-peer communication

I Peer-to-peer (P2P) communication requires each subsystem

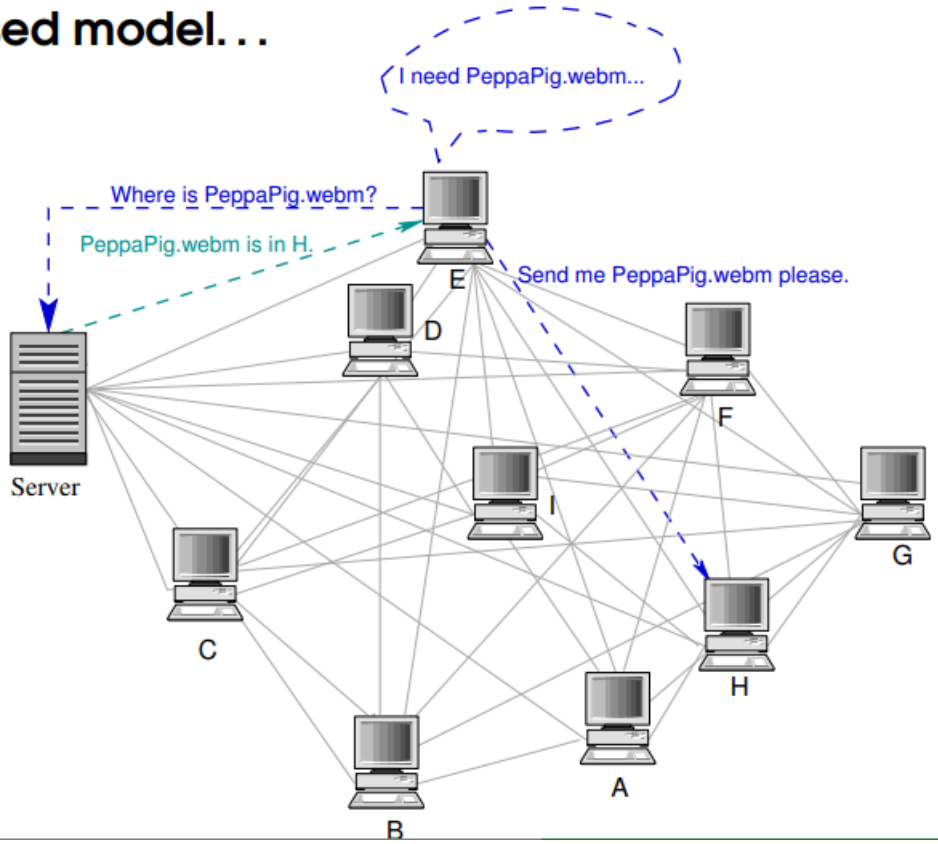
to know the interface of the other, thus coupling them more tightly.
I Each peer has to run exactly the same program and hence all peers provide identical services.
I Peer-to-peer communication is two way since either peer subsystem may request services from the other.

P2P How to find a peer which holds the required information

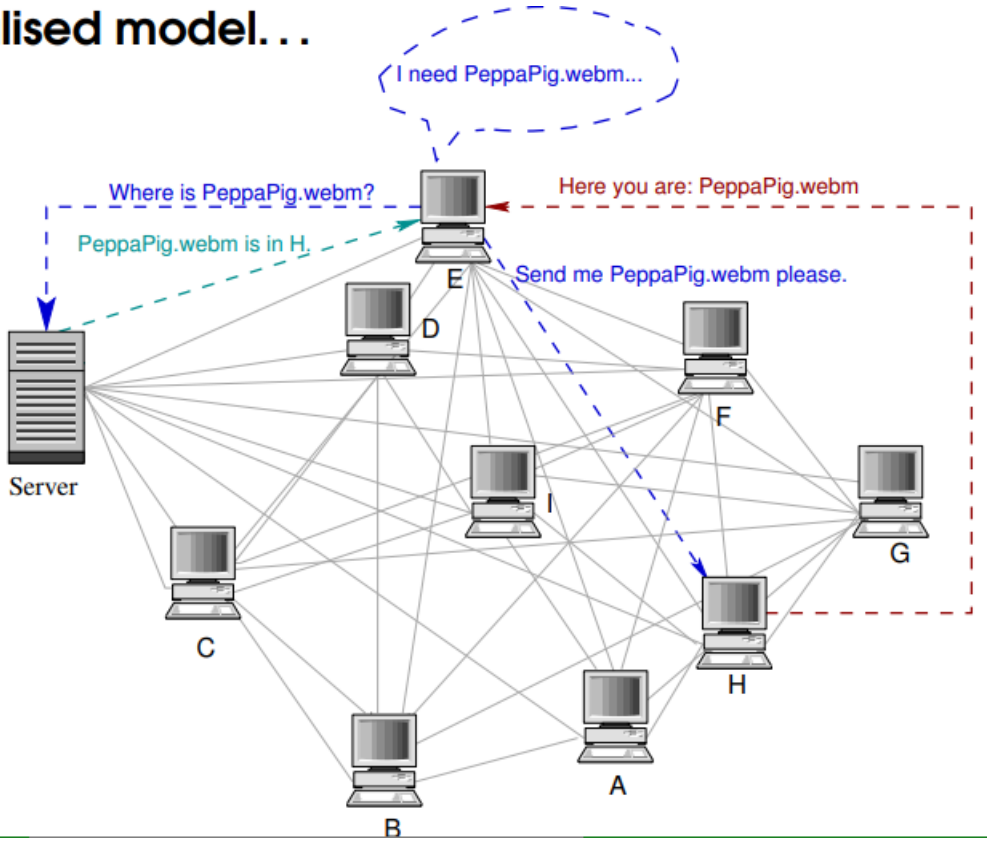




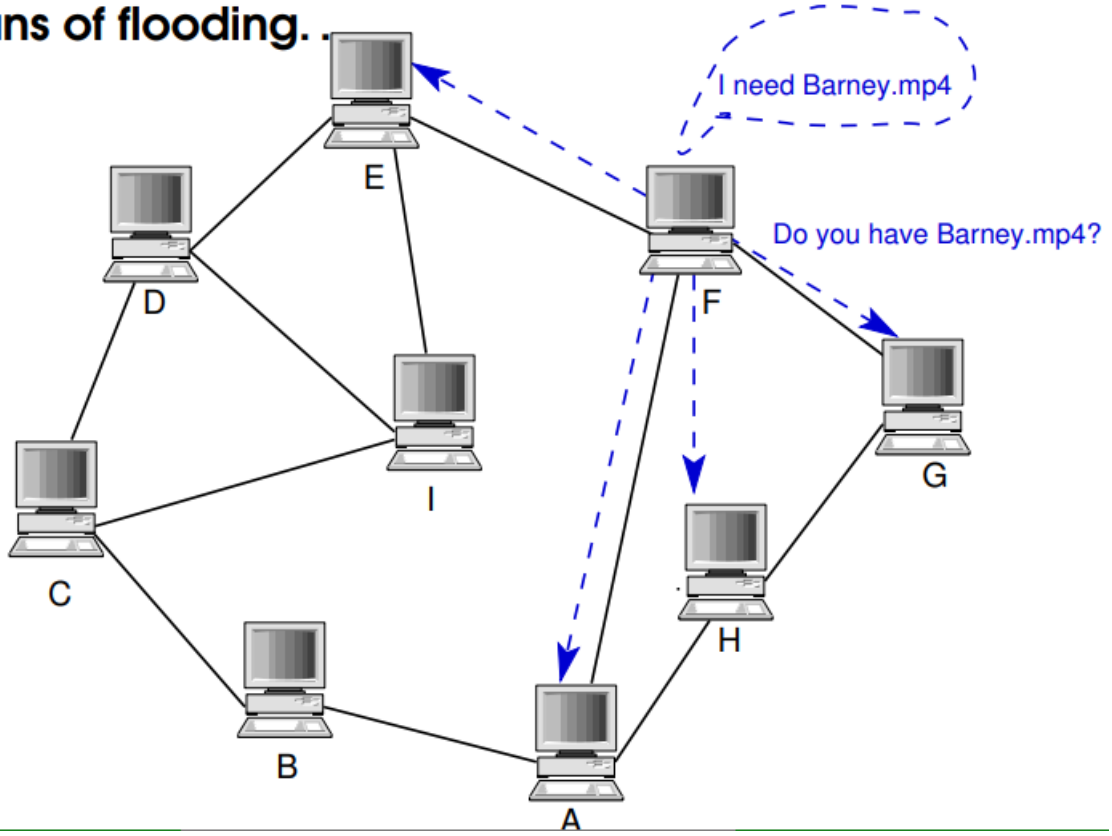
Using a centralised model...



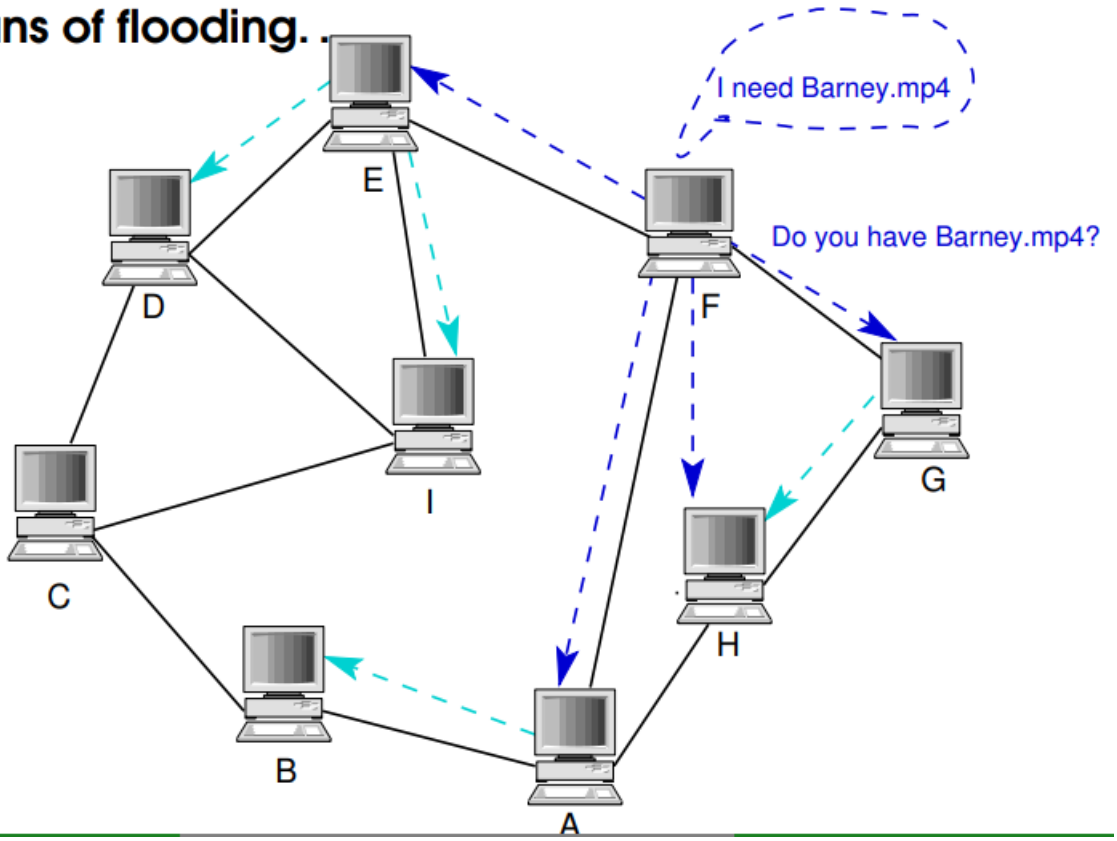
Using a centralised model...



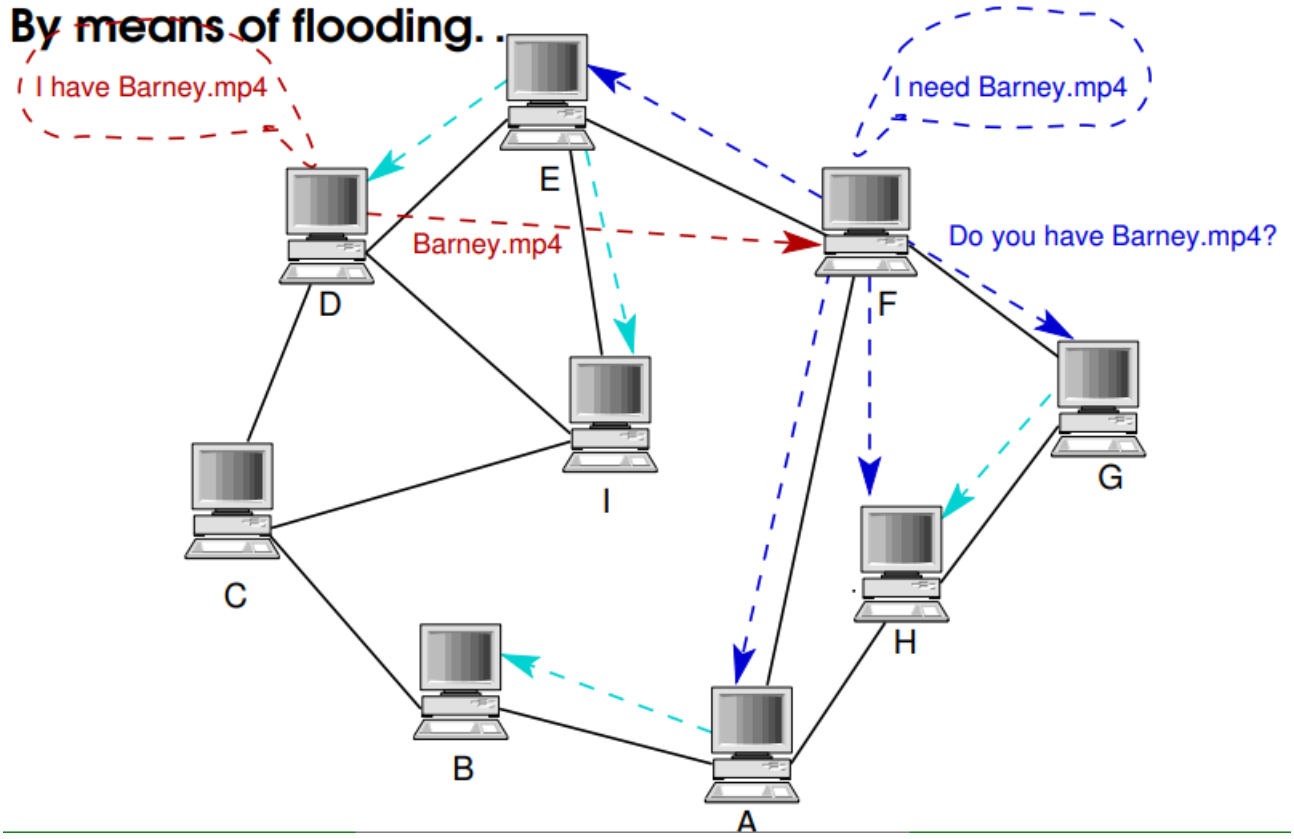
By means of flooding. .



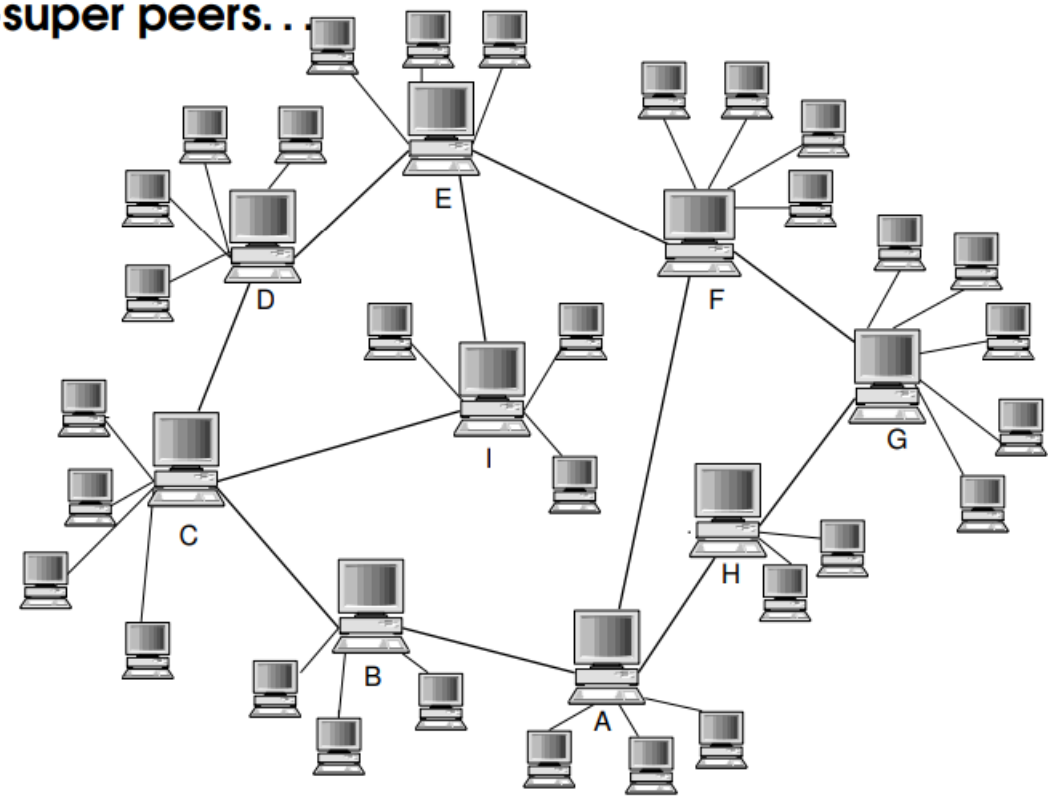
By means of flooding. .

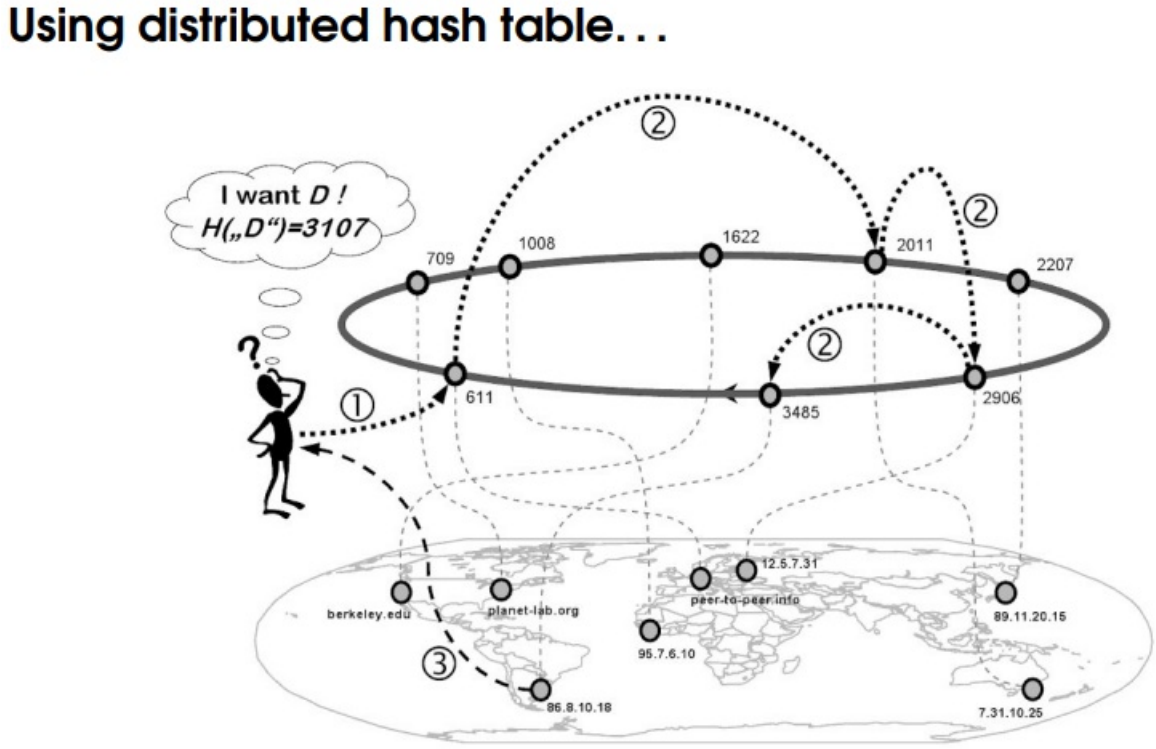


By means of flooding. . .



Using ultra-super peers. . .





An example

I Torrent file sharing systems use a distributed hash table to facilitate storing and retrieving of large files.
I ...a key-value store distributed accross a number of nodes in a network
➟ Each peer knows which portions of files it keeps and also where to look for other chunks from other peers.
I (By contrast, in a blockchain, each node of the network usually stores the full ledger of transactions)
➟ Example of an attempt to combine approaches: https://

ceur-ws.org/Vol-2334/DLTpaper4.pdf

Service Oriented Architecture

Large distributed system may adopt service Oriented Architecutre in their design

The basic unit of communication in SOA is the invocation of remote services

A service typically refers to a Web service:

- communcates via internet protocols (e.g. HTTP)
- Sends and receives structured data e.g. in XML or JSON format

in SOA , services interact via a common communications protocol

→ The resulting system components are loosely coupled with each other.

Holisay Booking Scenario

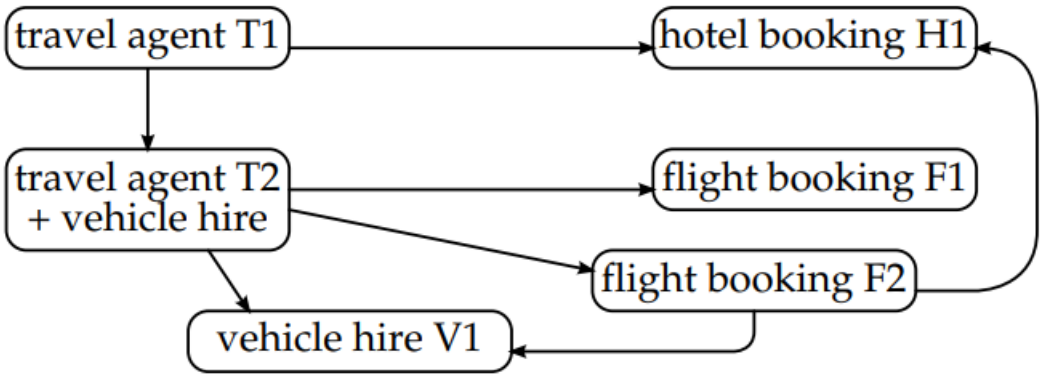
An example of a SOA system is a network of Web servicesbcomprising and supporting travel agents.

This system is composed of services for booking flights, hotels

and car hire.

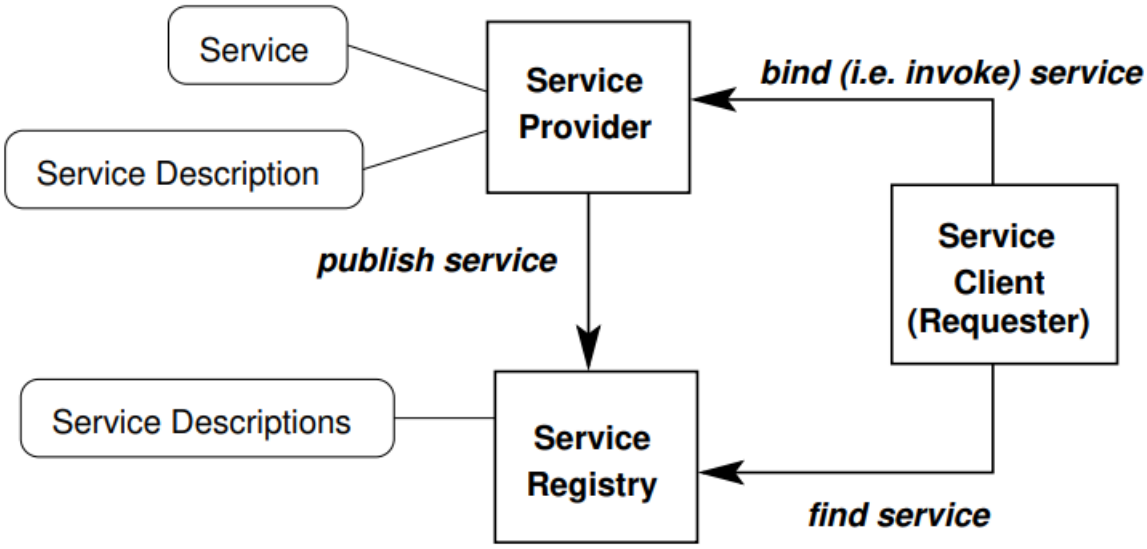
Agent applications use these services to assemble more sophisticated holiday-package services for their clients.

SOA example ^



- Each bubble is a *service* provided by potentially a *different* vendor.
- A *service* tends to use another service in order to complete its task.
- It should be *easy* to switch between the services of the same kind provided by a different vendor.

Operations in SOA



Common principles in SOA

You don’t have to show the operations just what is available, like skyscanner.

Reusable logic is divided into services

Services share a formal contract

→ mainly regarding information exchange

Services are loosely coupled

Services abstract underlying logic

→ The only part of a service that is visible to the outside world is what is exposed via the services description

Services are composable.

Why SOA

Business use of SOA is justified by a promise of :

- Easier reuse of services for multiple purposes
- Better adaptability to changing business environment and availible technologies
- Ability to integrate new and legacy systems
- Ability to cheaply setup e-business links across the world

Week 9 Detailed Design

Objectives

- [ ] Thuis lol

Notes

Detailed Design : Classes

In system analysis , key business concepts are identified and depicted as *classes*.

Classes capture key business concepts and operations

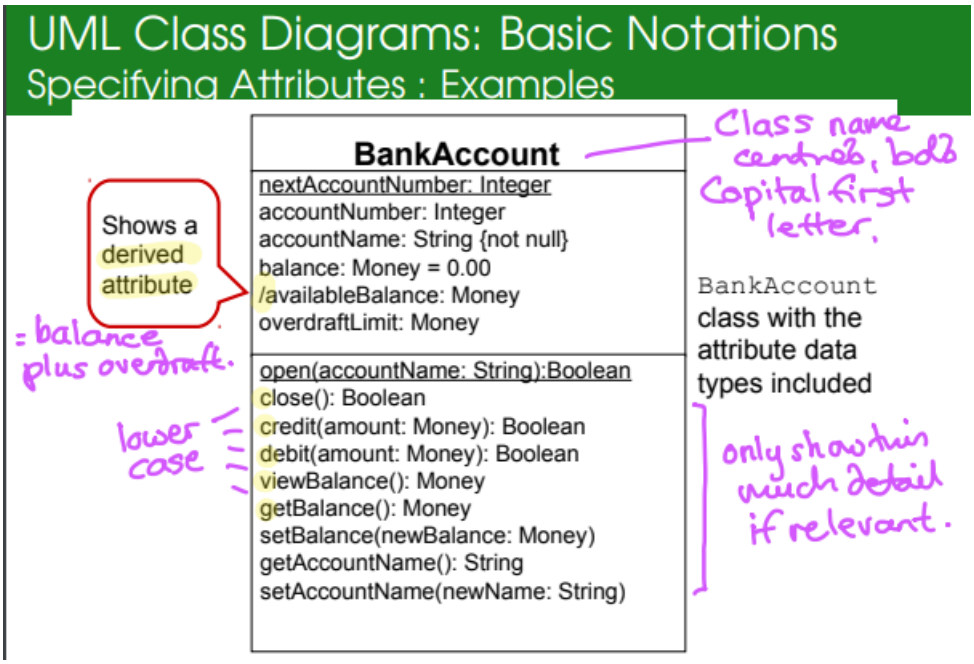
Classes only present a partial view of the required model.

In **Detailed Design** , we add more detail to the class model:

- Deciding the data type of each attribute.
- Deciding how to handle derived attributes.
- Adding primary operations
- Defining the signature of operations including the types of parameters
- Deciding the visibility of attributes and operations.

Style Guidelines for UML Class Diagrams

- Class name should be centered and presented in bold
- The first letter of class names should be *capitalized* .
- Attributes and operations' names should begin with a lowercase letter.
- The name of each abstract class should appear in italics
- Full attributes and operations should be shown only when needed.
    - They should be supressed in other contexts or when merely referring to a class.



The derived attribute is dependant on another , it is indicated by the / symbol.

The derived method helps us protect variables to stop them being amnipulated elsewhere.

Specifying Attributes

The attribute balance in class `BankAccount` might be declared with an *initial value* of zero:

```
balance:Money = 0.00
```

Attributes that may not be null are specified using a property string:

```
accountName:String{not null}
```

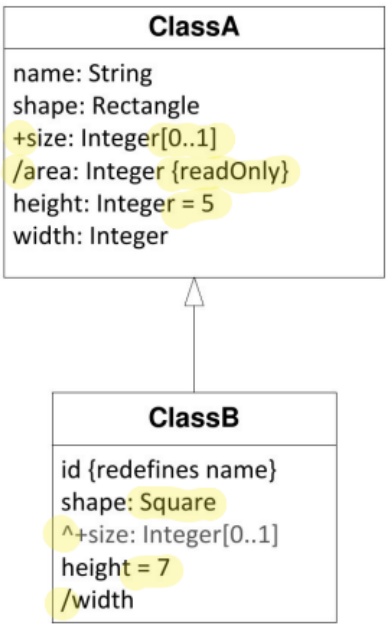
Arrays are specified using multiplicity

```
qualifications:String[0..10]
```

The derived attribute `availibleBalance` whose value is calculated from `balance + overdraftLimit` is marked by:

```
/availibleBalance: Money
```

More Examples:



- ClassA::name is an attribute with type String.
- ClassA::shape is an attribute with type Rectangle.
- ClassA::size is a *public* attribute of type Integer with *multiplicity* 0..1.
- ClassA::area is a *derived attribute* with type Integer and a *read-only* property.
- ClassA::height is an attribute of type Integer with a *default initial value* of 5.
- ClassA::width is an attribute of type Integer.
- ClassB::shape is an attribute that redefines ClassA::shape. It has type Square, a specialization of Rectangle.
- ClassB shows size as an attribute *inherited* from ClassA, as signified by the prepended *caret* symbol.
- ClassB::height is an attribute that redefines ClassA::height. It has a *default* value of 7 for ClassB instances that overrides the ClassA *default* of 5.
- ClassB::width is a *derived attribute* that redefines ClassA::width, which is not derived.

All squares are specific rectangles.

The curly brackets can mean a sidenote or a constraint.

When something is static we underline it and write it in capitals.

the / width is a derived variable meaning we will no longer be able to explicitly set it.

Operation Signatures

An operation's signature is determined by the operation's name, the number and the type of its parameters and the return type (if any).

visibility is the visibility of the operation i.e. one of the following:

- "+" public
- "-" private
- "#" protected
- "~" package

name is the name of the operation

The name of the operation followed by a pair of brackets is the mandatory part of specifying an operation.

The parameter-list is optional

If included, it is a list of parameter names and types separated by commas.

return-type is the type of the return value

If the operation has one defined

`BankAccoutn` might have a `credit()` operation that would be shown in the diagram as:

```
credit(amount: Money): Boolean
```

We're feeding in a money parameter and returning boolean.

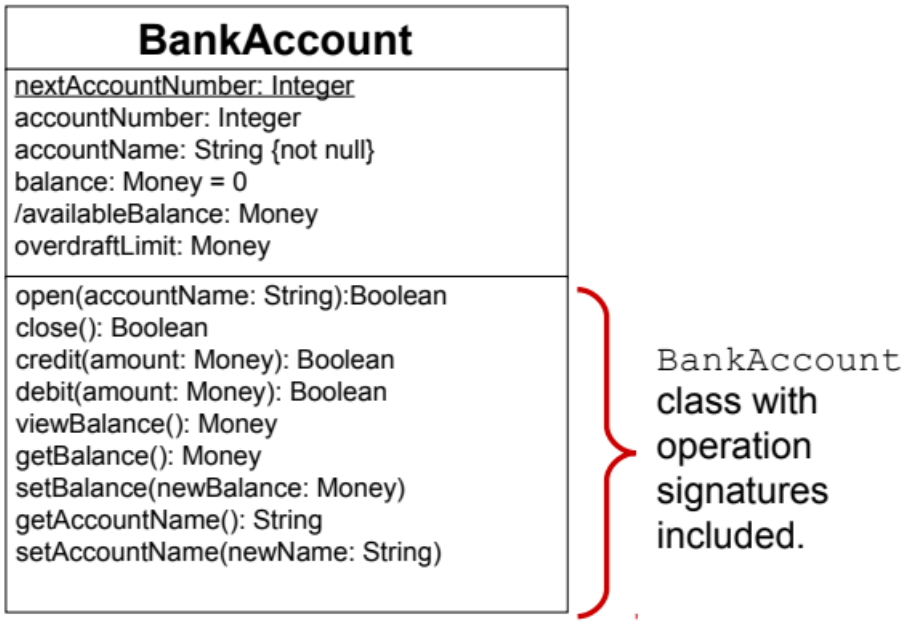
```
- hide()
+ toString(): String
+ insertionSort<T -> Comparable>(data : T[1..*])
```

Basic Notations

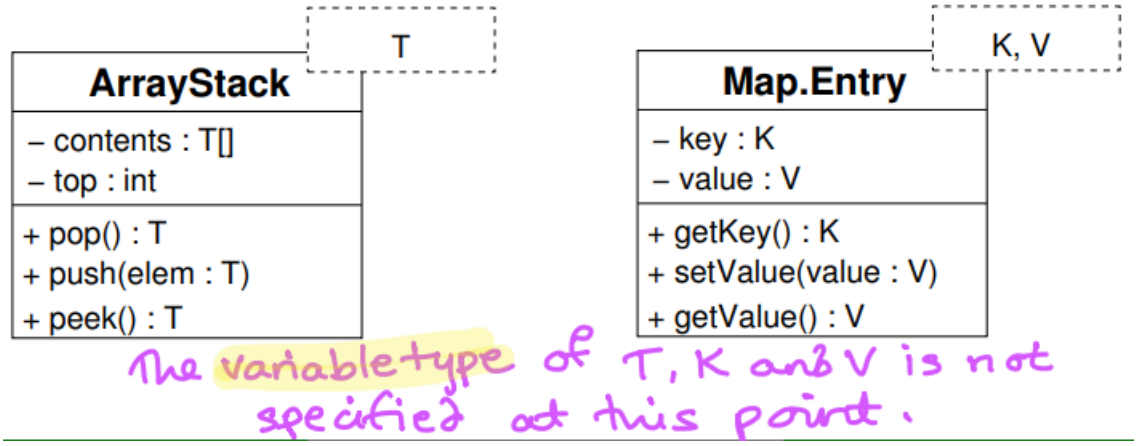
Show primary operations (i.e. constructors, destructors, getters and setters) where it is neccessary

Only show constructors where they have special significance

Varying levels of detail at different stages in the development cycle

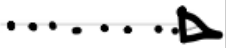
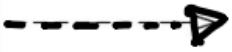


Templates are model Elements that are parameterized by other model Elements



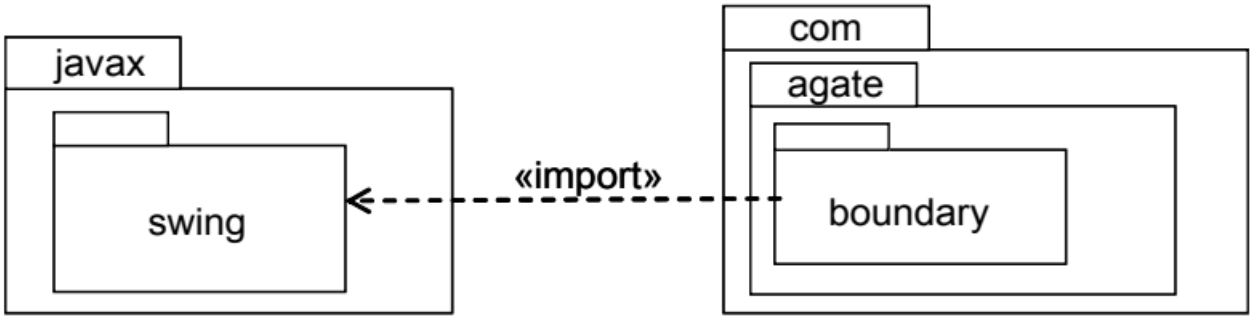
Templates typically appear within the context of specifying a generic type

Relations:

PATH TYPE	NOTATION	REFERENCE
Aggregation		See 7.3.2, 'AggregationKind (from Kernel)'
Association		See 7.3.3, 'Association (from Kernel)'
Composition		See 7.3.2, 'AggregationKind (from Kernel)'
Dependency		See 7.3.12, 'Dependency (from Dependencies)'
Generalization		See 7.3.20, 'Generalization (from Kernel, PowerTypes)'
InterfaceRealization		See 7.3.25, 'InterfaceRealization (from Interfaces)'
Realization		See 7.3.45, 'Realization (from Dependencies)'
Usage		See 7.3.53, 'Usage (from Dependencies)'

e.g
<<import>>
word describes the usage

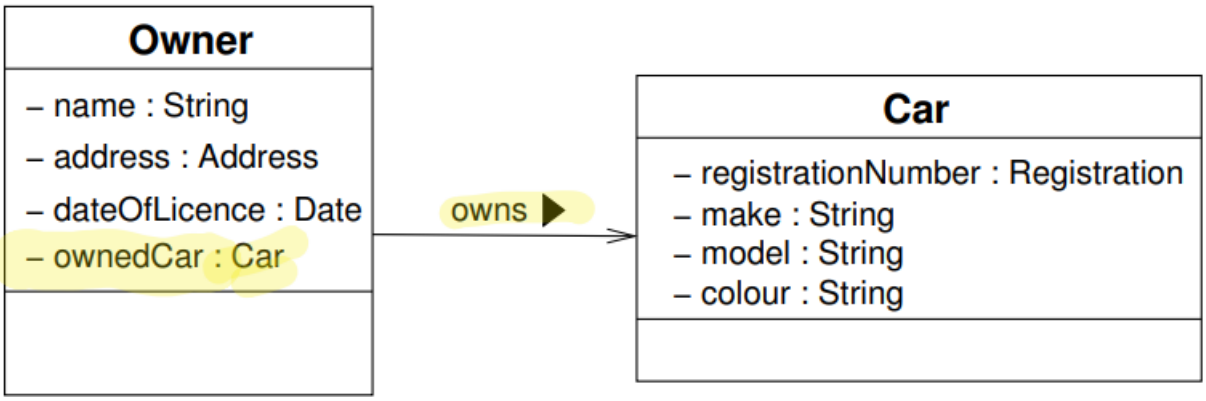
Import:



Designing Associations

Class `Owner` can send messages to class `Car` but not vice versa

Each `Owner` object keeps a reference to the owned `Car` object



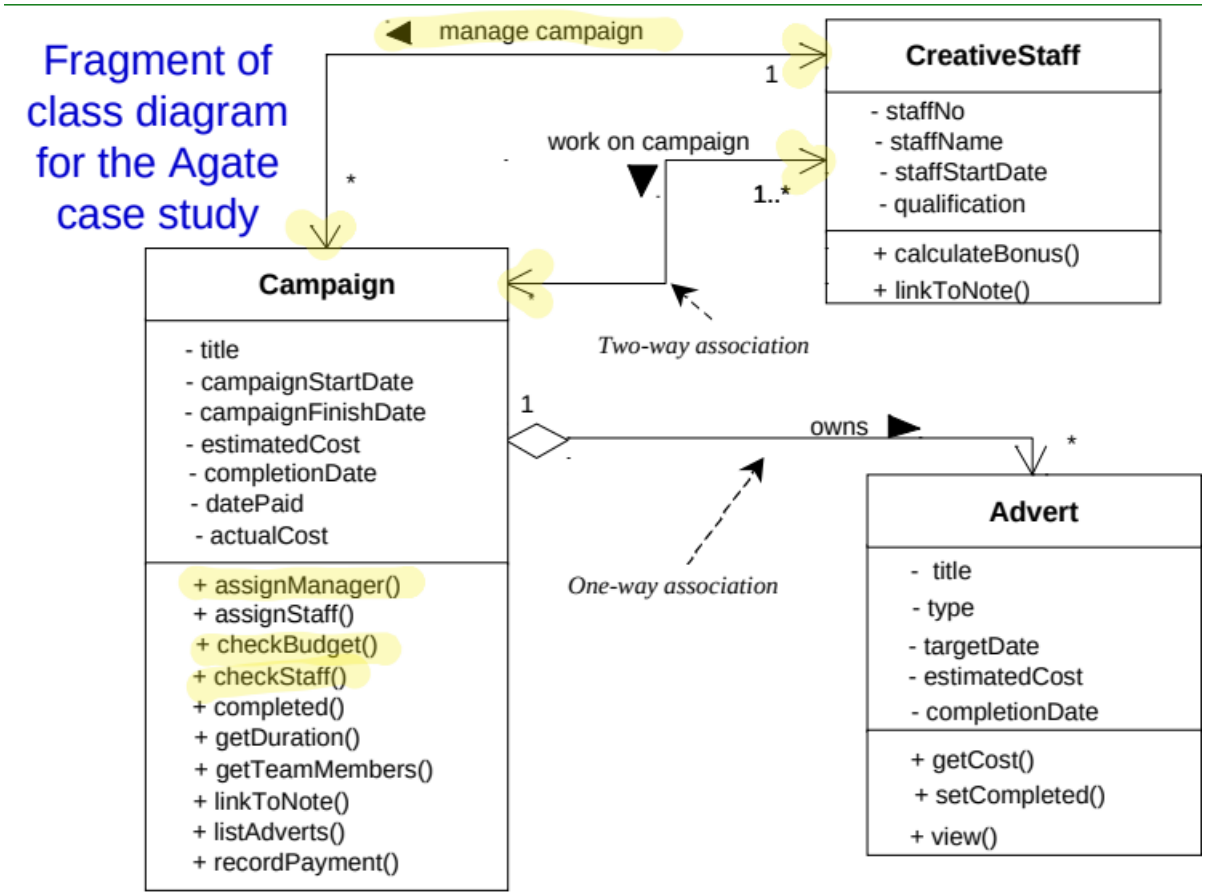
An association that has to support message passing in both directions is a two-way association

A two way association is indicated with arrowheads at both ends, where the arrowheads specify the permissavble direction of navigation

Coupling describes the degree of interconnectedness between classes

A good design seeks to minimise coupling

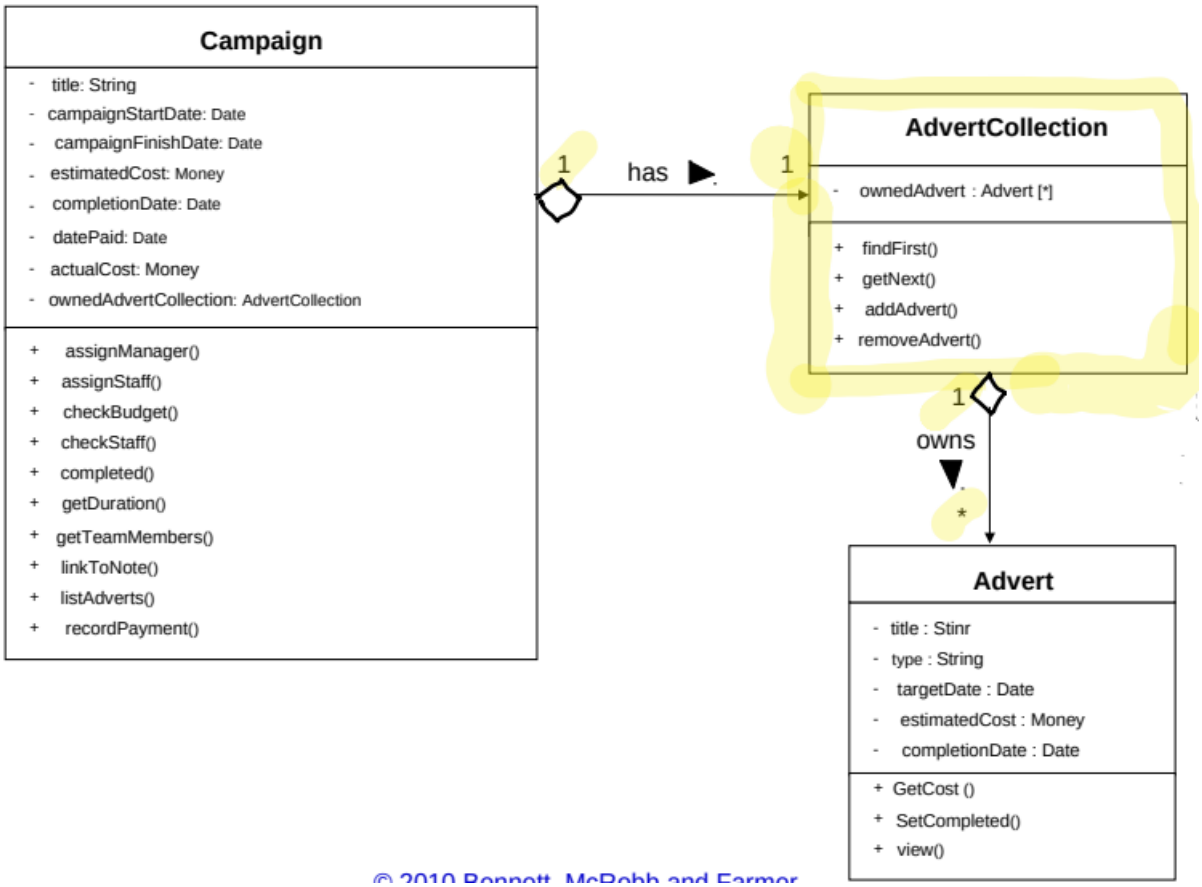
Minimizing the number of two way associations keeps the coupling between objects as low as possible.



Collection classes are used to hold the object identifiers (i.e. object references) when passing is required from one to many along an association

OO languages typically provide support for working with collection classes.

```
private List<Advert> adverts;
```



Integrity Constraints

Referential integrity ensures that an object identifier / reference in an objects is actually referring to an object that exists otherwise it should refer to a null value.

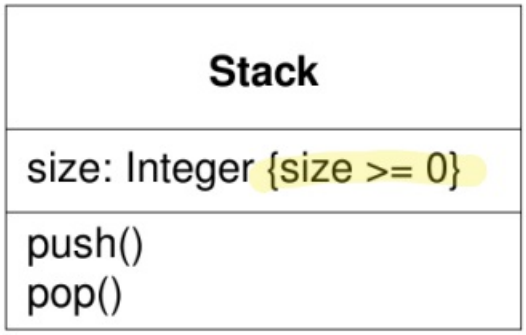
> Campaign object reference(s) which a `CreativeStaff` object keeps must be either null (i.e. the staff is not working on any campaign) or existing Campaign object reference(s).

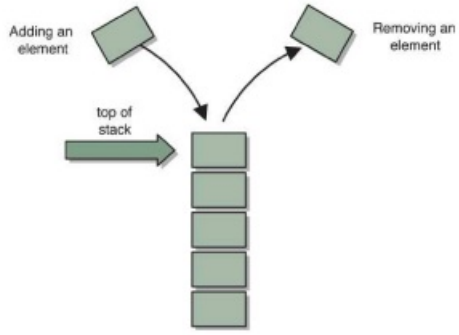
Dependency Constraints ensures that attribute dependencies, where one attribute may be calcualted from other attributes are maintained consistently

> an attribute whose value is calculated from other attributes is a derived attribute marked by a /

Domain Integrity ensures that attributes only hold permissable values

> attributes from the `Cost` domain must be non-negative recorded in 2 decimal places.





Interfaces

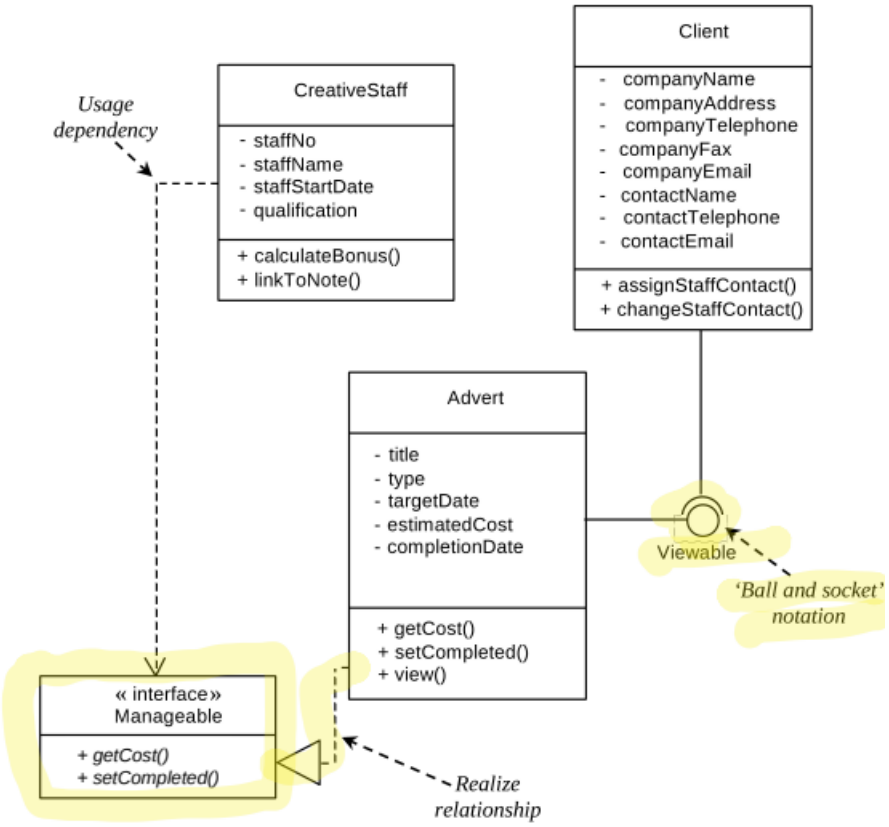
Uml supports two notations to show interfaces

The small circle icon with a list of the operations supported

A stereotyped class icon with a list of the operations supported

Normally only one of these is used on a diagram

The realization relationship represented by the dashed line with a non filled triangular arrowhead indicates that the client class supports at least the operations listed in the interface.



Cohesion and coupling

Bennett, McRobb and Farmer (2010):

> Cohesion is a measure of the degree to which an element contributes to a single purpose.

> Coupling describes the degree of interconnectedness between design components. It is reflected by the number of links an object has and by the degree of interaction the object has with other objects.

Braude and Bernstein (2011):

> Cohesion within a module1
> is the degree to which the
> module's elements belong together. . . it is a measure of how focused a module is.

What are they

Cohesion and coupling aim to support each other:

- Traditional detailed design tried to maximise cohesion elements of module o code so all contribute to the achievement of a single function.
- Traditional detailed design treid to minimise coupling unnecessary linkages between ,odules that made them difficult to maintain or use in isolation from other modules.

Types:

Several ways in which coupling and cohesion can be applied within an object-oriented approach:

- Interaction coupling
- Inheritance coupling
- Operation cohesion
- Class cohesion
- Specialization cohesion
- Temporal cohesion

Coupling

Interaction Coupling is a measure of the number of message types an object sends to other objects and the number of parameters passed with these message types

High interacxtion coupling means objects call each others methods a lot.

> "In general, if a Message Connection involves more than

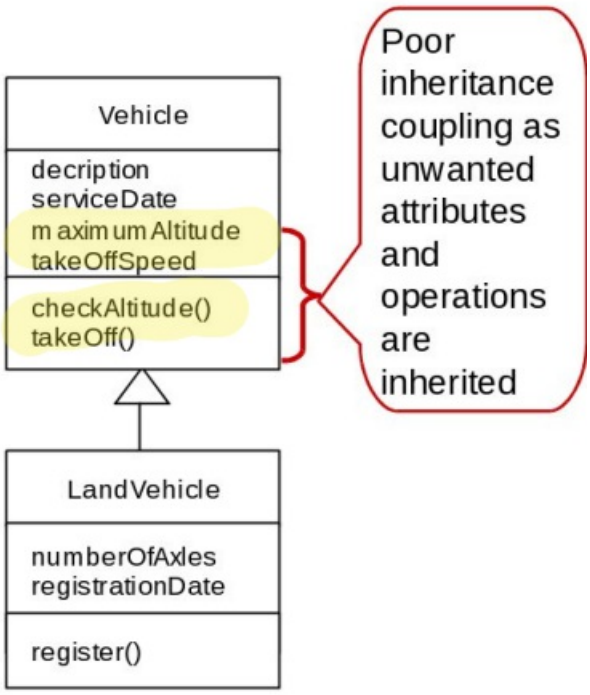
> three parameters, examine it to see if it can be simplified."

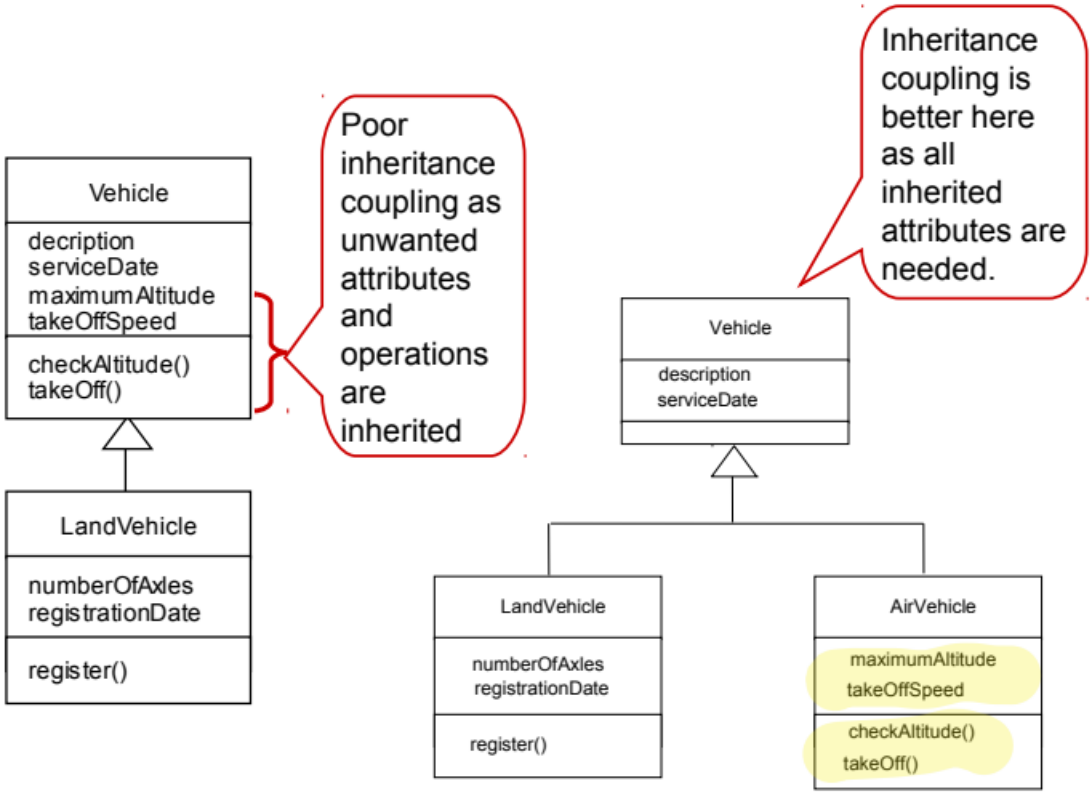
(Coad and Yourdon, 1991, Section 8.2.1)

Interaction coupling in a detailed design should be kept to a minimum to reduce the possibility of changes rippling through the interfaces and to make reuse easier.

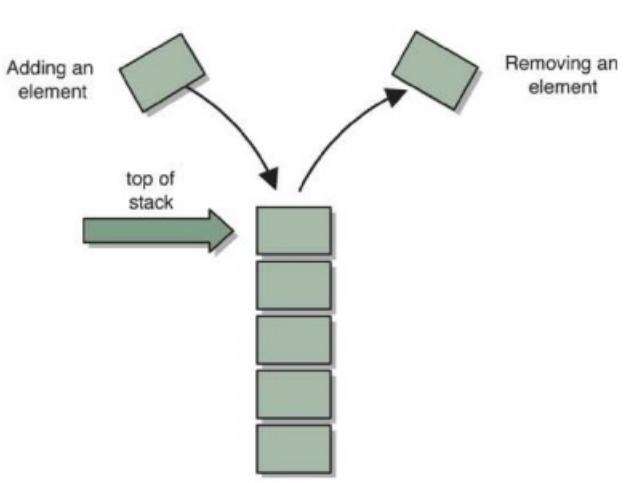
Inheritance Coupling describes the degree to which a subclass needs its inherited features \9 degree of independence).

Inheritance should not be used in abundance or carelessly as it may weaken the degree of information hiding in the classes concerned





Classes `Vector` and `Stack` in package `java.util` is an example of poor inheritance coupling because stack inherits a range of unsuitable methods from `Vector` .

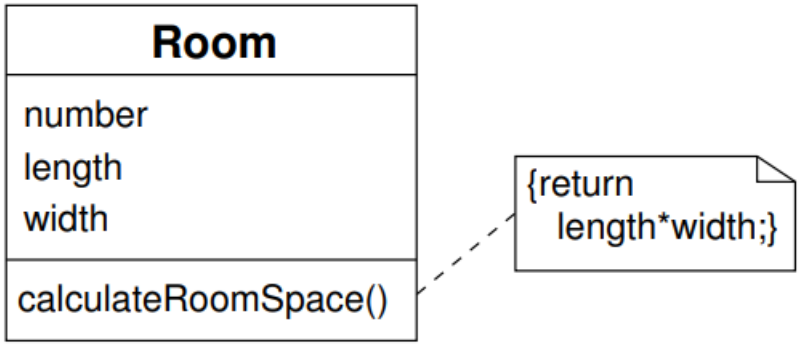


➟ *Stack* is a Last-In-First-Out (LIFO) collection with access to its elements at the **top** of the stack only.

➟ Class `java.util.Stack` inherits inappropriate methods such as insertElementAt, firstElement, setElementAt, removeElementAt and sort from `java.util.Vector`.

Operation Cohesion measures the degree to which an operation focuses on a single functional requirement

Good operation cohesion:



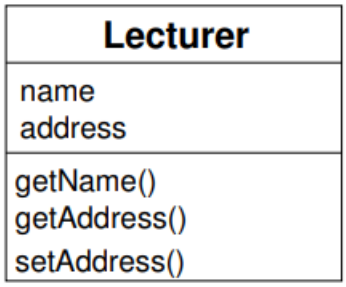
`calculateRoomSpace()` performs an atomic task: compute the total area of the room.

Class Cohesion measures the number of things a class represents

Each class should represent only a single entity

It measures the degree to which a class is focused on a single requirement

(mainitaining aparticualr type of record for example)



➟ Lecturer includes attributes and operations which address the same requirement: *maintain lecturer record*.

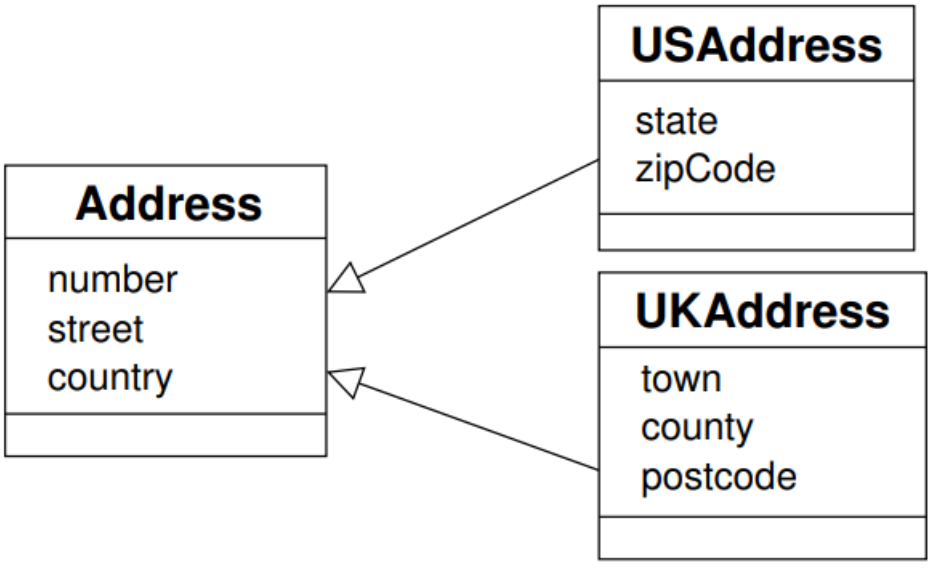
Specialization Cohesion adressses the semantic cohesion of ineritance hierarchies

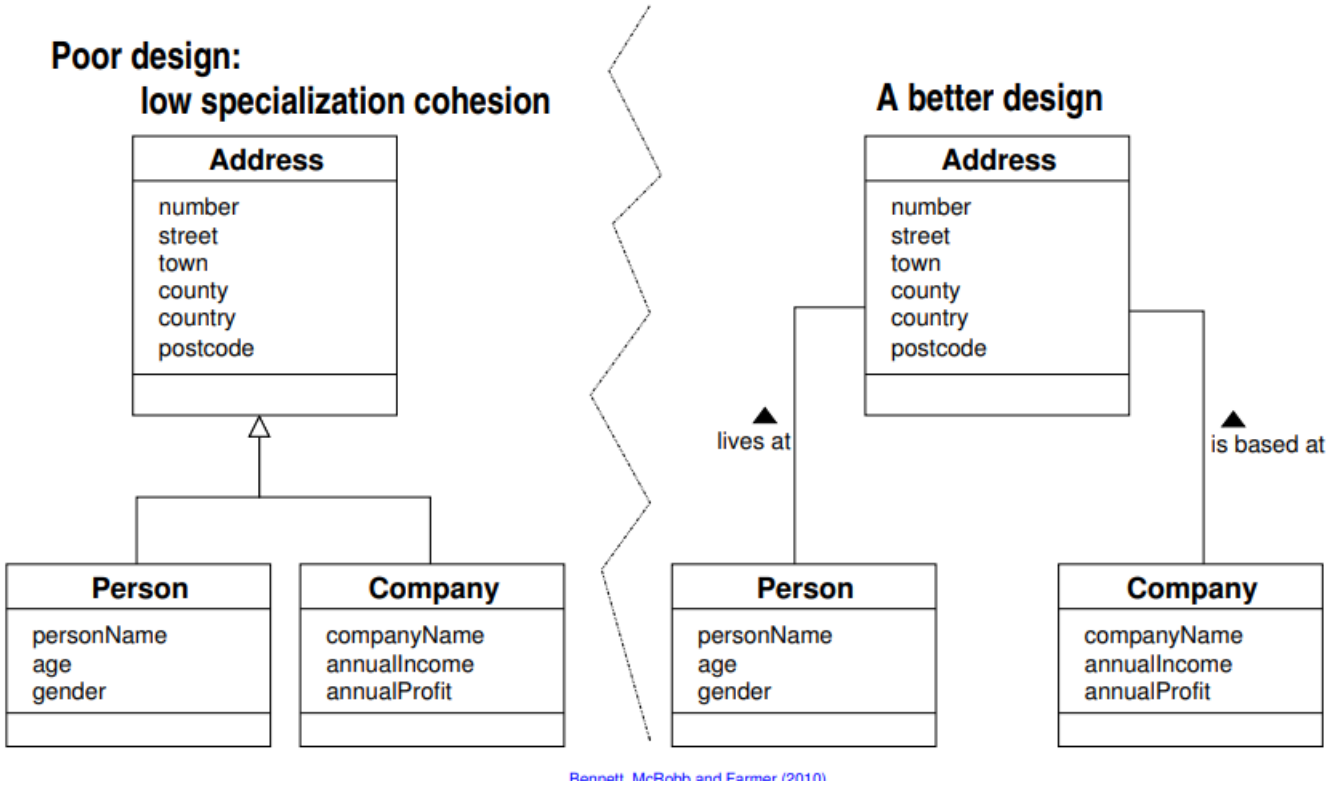
It measures te semantic-relatedness between the inherited attributes and operations to the class itself

are the classes related to each other through a “is_a” relation? Or is the generalisation relation defined out of convenience?

Class A should extend and hence inherit attributes and operations from class B only when A is a aspecialized version of B , not simply because it needs to use the attributes and operations in B.

Example of a good *specialization cohesion*:





Temporal Cohesion module elements linked only by the time at which they need to be done:

an "initialisation" module :

> "Temporal cohesion is present when a subprogram performs a set of functions that are related in time, such as 'initialisation', 'house-keeping', and 'wrap-up'. . . . The only connection between these operations is that they are performed within the same limited time-span."

p.91 Information Systems Engineering: An Introduction, By Arne Soelvberg, David C. Kung

Points to note

> "Low cohesion classes often represent a very 'large grain' of abstraction or have taken on responsibilities that should have been delegated to other objects. . . . As a rule of thumb, a class with high cohesion has a relatively small number of methods, with highly related functionality, and does not do too much work. It collaborates with other objects to share the effort if the task is large."

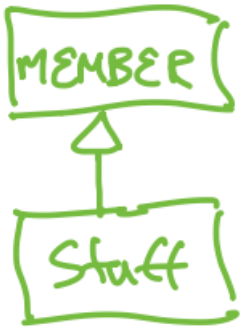
> ". . . no coupling between classes . . . offends against a central

> metaphor of object technology. . . Low coupling taken to excess yields a poor design. . . It is not high coupling per se that is the problem; it is high coupling to elements that are unstable in some dimension, such as their interface, implementation, or mere presence."

Liskov Substitution Principle

States that in object interactions we should not be able to treat a derived object (instance of a subclass) as if it were a base object (instance of a superclass) without integrity problems.

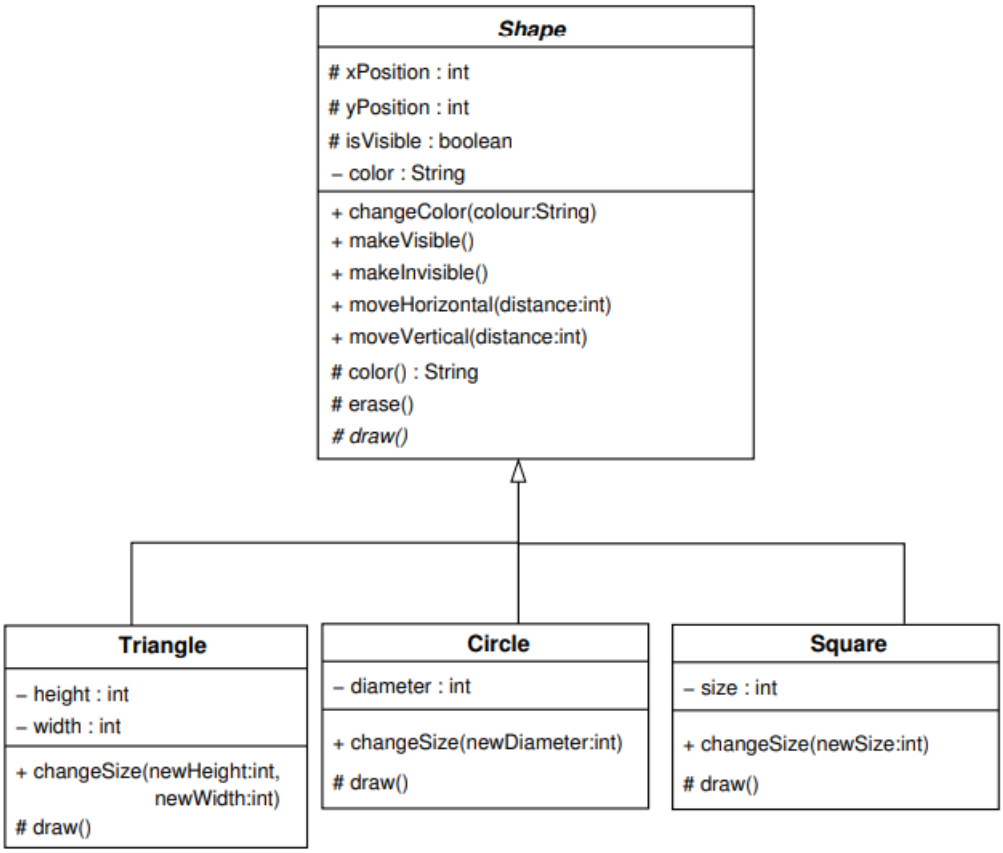
> "What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T ."




```
public void setDriver(Member carUser) {
if (carUser.hasDrivingLicence()) {
driver = carUser;
}
}
```

```
public void setDriver(Staff carUser) {
if (carUser.hasDrivingLicence()) {
driver = carUser;
}
}
```

> "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."

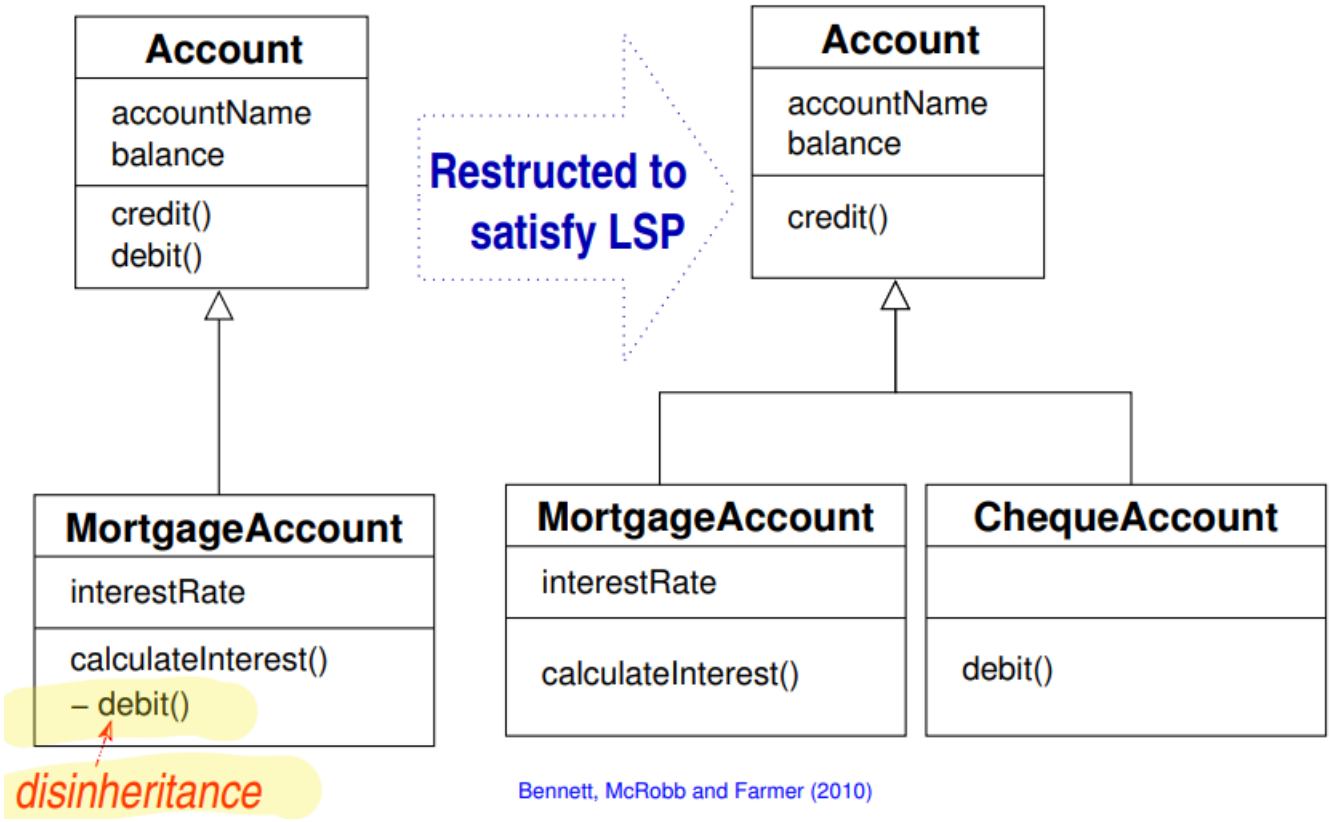


If LSP is not applied, it may be possible to violate the integrity of the derived object, e.g.:

Subclasses typically inherit attributes and operations from their superclass.

If an attribute or operation is unsuitable (not applicable) for the semantic of a subclass, it is possible for the subclass to "disinherit" it by declaring it private.

However, this will violate the Liskov Substitution Principle.



The debit object is now no longer easy to substitute in.

Week 10 Design Patterns

Objectives

- [ ] this lol

Notes

Patterns

What is a design pattern?

A design pattern , as described by Gamma et al. :

> *"description of communicating objects and classes that are customized to solve a general design problem in a particular context"*

Patterns are problem-centred not solution-based

They capture and communicate "best practice" and expertise

Patterns and non-functional Requirements

Patterns are typically applied to address non-functional requirements:

- Chnageability
- Interoperability
- efficiency
- reliability
- testability
- reusability

The pattern Template

According to Bennet et al a pattern should have:

- A *meaningful* name reflecting the knowledge embodied by the pattern
- A description of the problem that the pattern adresses
- Context - the circumstances in which the pattern can be applied.
- Forces:
  - The constraints addressed by the solution
  - Any trade-offs or language specific issues
- Solution:
  - The components involved in the pattern and their relationships
    - May be shown as a UML class diagram or object diagram

GoF Design Patterns

Gang of Four → refers to the book by Gamma & Bennet et al

GoF patterns are classified by 2 criteria:

Purpose: creational , structural , behavioural

Scope: class & object

Creational Patterns

GoF creational design patterns are concerned with the construction of object instances

They separate the operation of an application from how its objects are created

Examples:

Class: Factory

Object: Abstract Factory, Builder, Prototype, Singleton

Structural Patterns

GoF structural patterns are concerned with the wat in which classes and objects are organized.

They offer effective ways of using object oriented constructs like inheritance, aggregation and composition to satisfy particular requirements.

Examples:

Class Adapter

Object: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy

Behavioural Patterns

GoF behavioural patterns address the problems that arise when assigning responsibilities to classes and when designing algorithms.

They suggest particular static relationships between objects and classes and also describe how the objects communicate.

Examples:

Class: Interpreter, Template Method

Object: Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, visitor

Observer

An investment company helps clients find profitable stocks in the market for investment.

- The company keeps an account for each client.
- A client may be investing in multiple stocks.
- The same stock is typically being purchased by multiple clients.
- The price of a stock changes continually.
- The investment company needs to maintain up-to-date stock values for all clients.

How to ensure that when stock value changes, the clients' accounts are updated immediately?

Solution 1

1. Equip class `Stock` with an update method which, when invoked, updates the respective value in all `Account` objects which are referenced by it.

```
1 public class Stock {
```

```
2 private double value;
3 private Set<Account> owners;
4 // other fields, constructor and methods omitted
5
6 private void updateOwners() {
7 for(Account owner : owners) {
8 owners.changeStockValue(this,value);
9 }
10 }
11 }
```

What are the drawbacks of this approach?

idk

Solution 2

The observer pattern...

> "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."

Alternative is to try and lower the coupling by independentaly updating the value for all stock owners. Leave it to the account to change itself.

Solution : Description

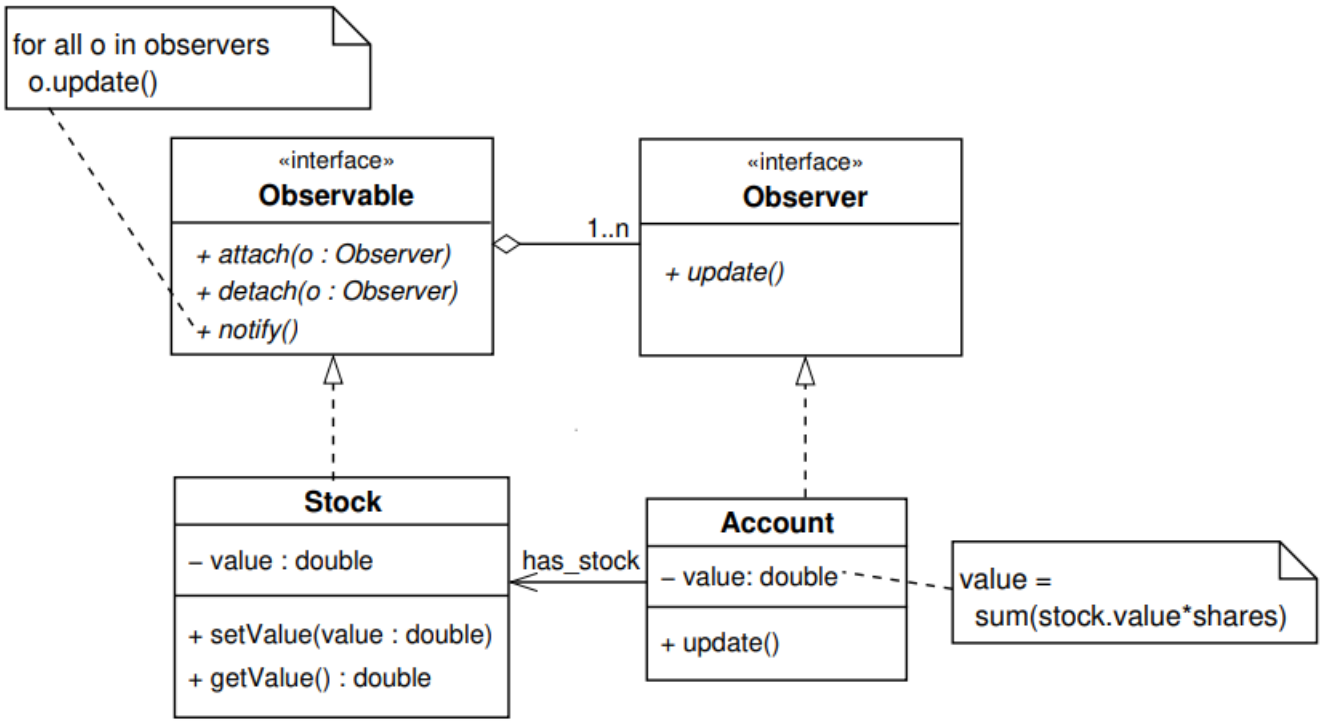
Enable the object acting as the data source to communicate with all objects which use the data (i.e. the Observer objects).

Notify the `Observer` objects of any change in the data source

Upon receiving a change notification, each `Observer` object updates its state.

→ The `observer` pattern promotes good object-oriented design because it enables the observers to keep their own data up to date.

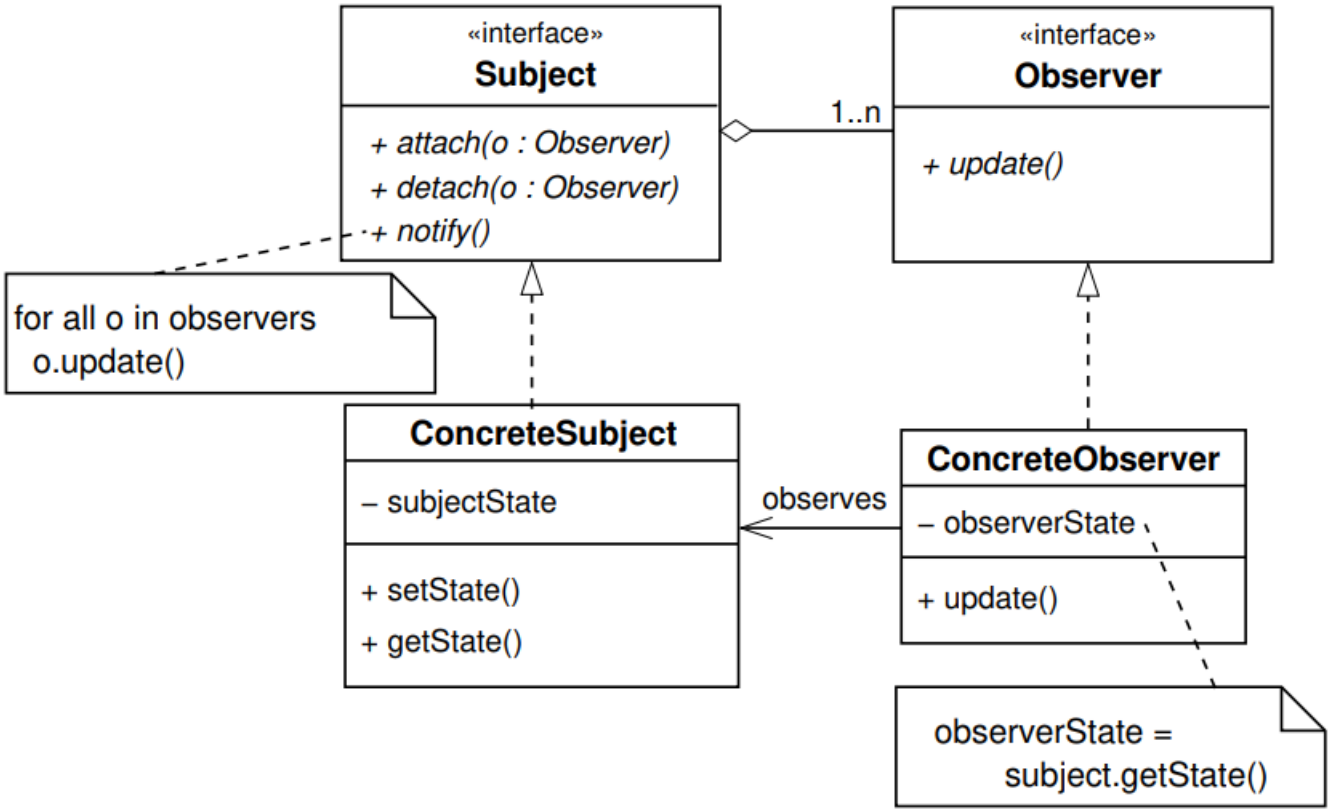
Class Design



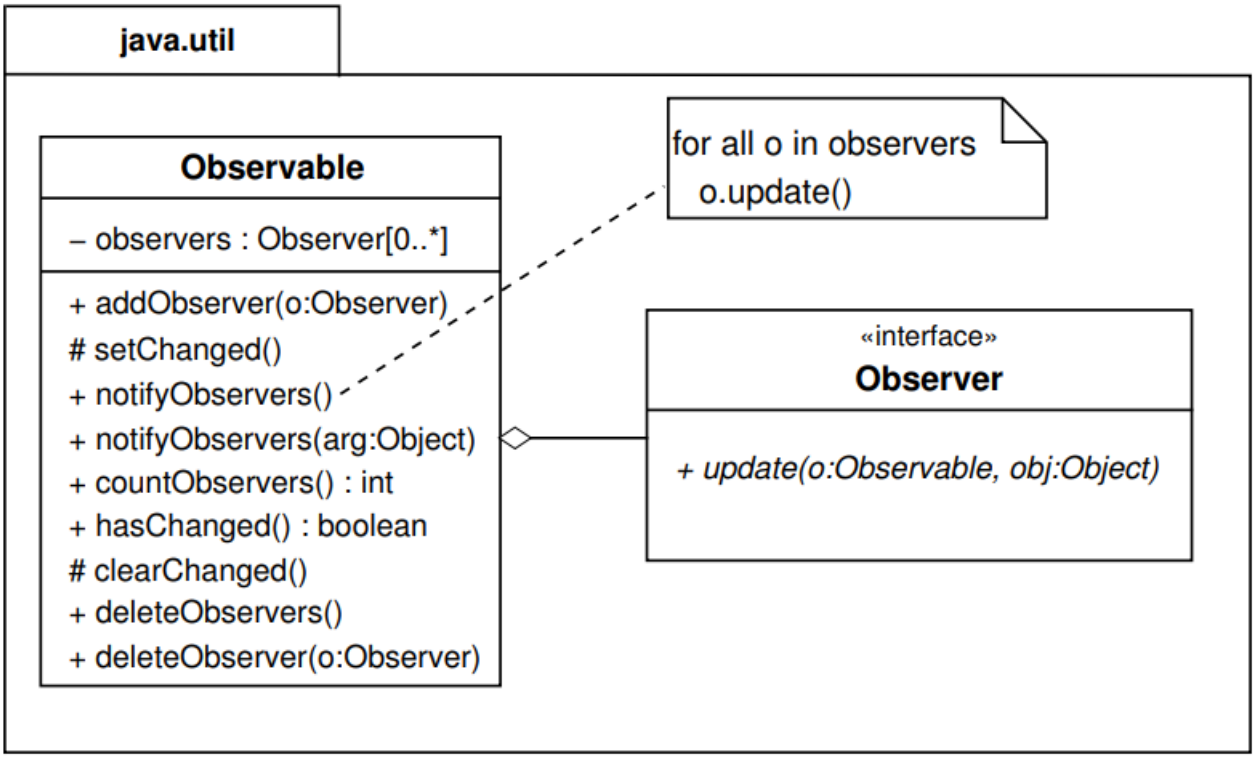
General Form

Subject = observable

We create a relation that allows multip,e observers watching one observable and the notify method would let all watching observes know of any changes.



Observer and Observable in `java.util` :



Interface `java.util.Observer` and class `java.util.Observable` modelled the two key components in the observer pattern.

Classes implementing the interface Observer needed to implement the inherited update method.

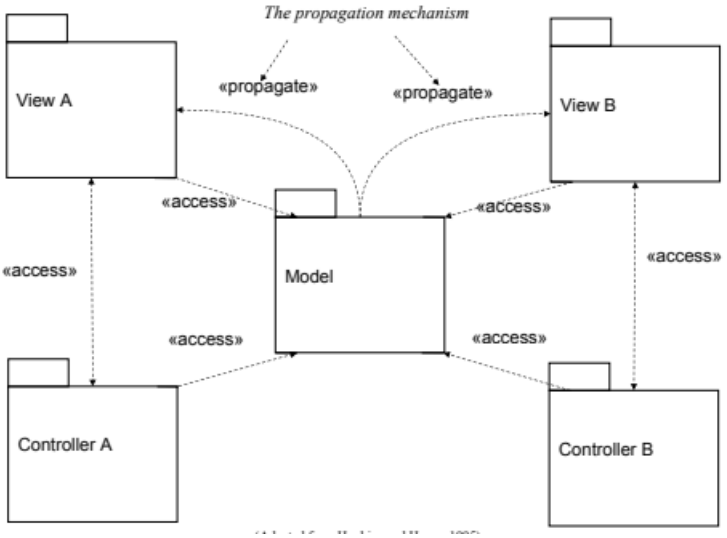
→ Called whenever an object is observed (i.e. Observable) changes its state.

Classes modelling data sources for observation extended class Observable, and included methods such as `addObserver`, `deleteObserver` and `notifyObservers`.

→When an Observable object changed its state, method `notifyObservers` needed to be called in order to notify its Observers to update their states.

Discussion :

The `observer` pattern is used in the `Model-View-Controller` (MVC) architecture in which the view plays the role of the `observer` and the *model* is the `observable` , with the *controllers* acting as clients updating the observables.



Structural Pattern

Composite

Consider the Agate case study: we need to store information about media clips for each advertisement. . .

Each advertisement may be made up of a single or a composite sequence of media clip(s).

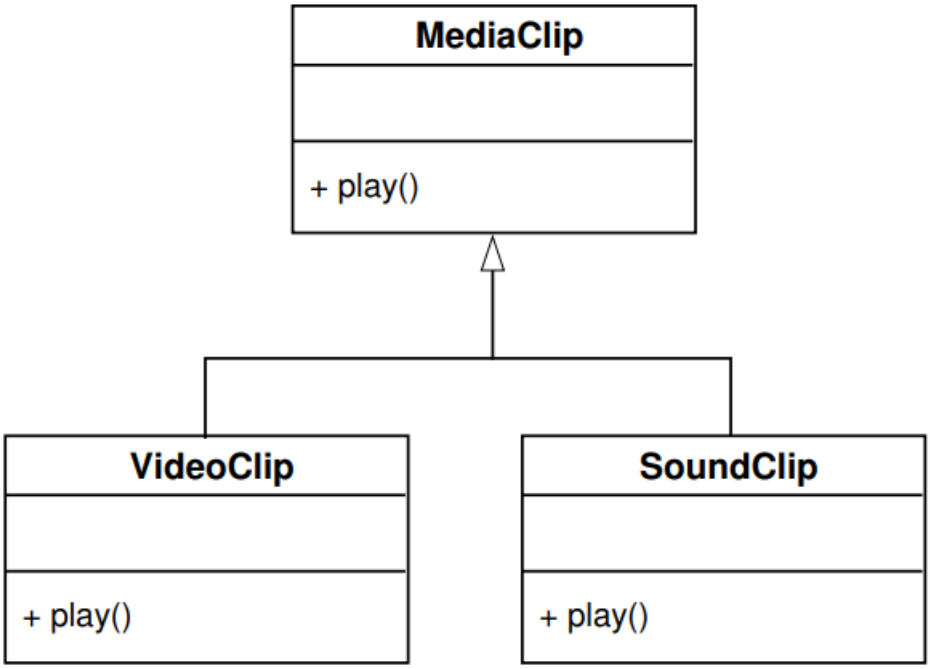
Each media clip in turn may take the form of a video clip or a sound clip, or a combination of several media clips.

How to present the same interface for a media clip whether it is composite or not?

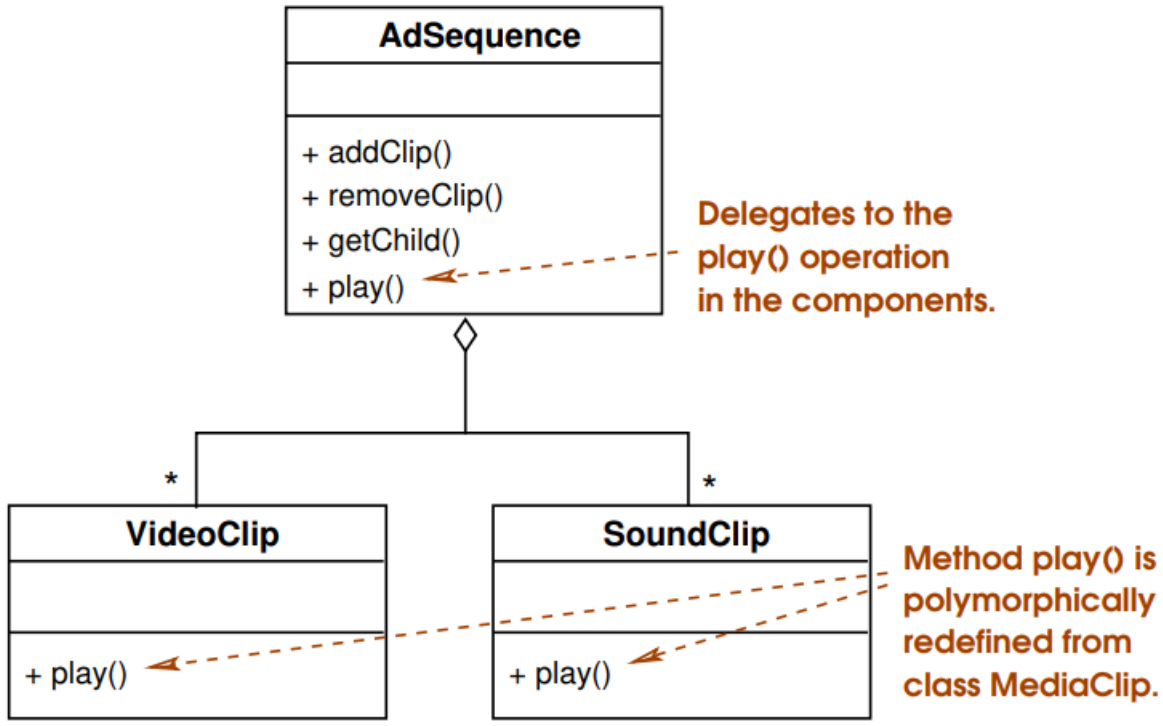
How can we incorporate composite structures?

Context / issue:

How do we present the same interface for a media clip whether it is composite or not?

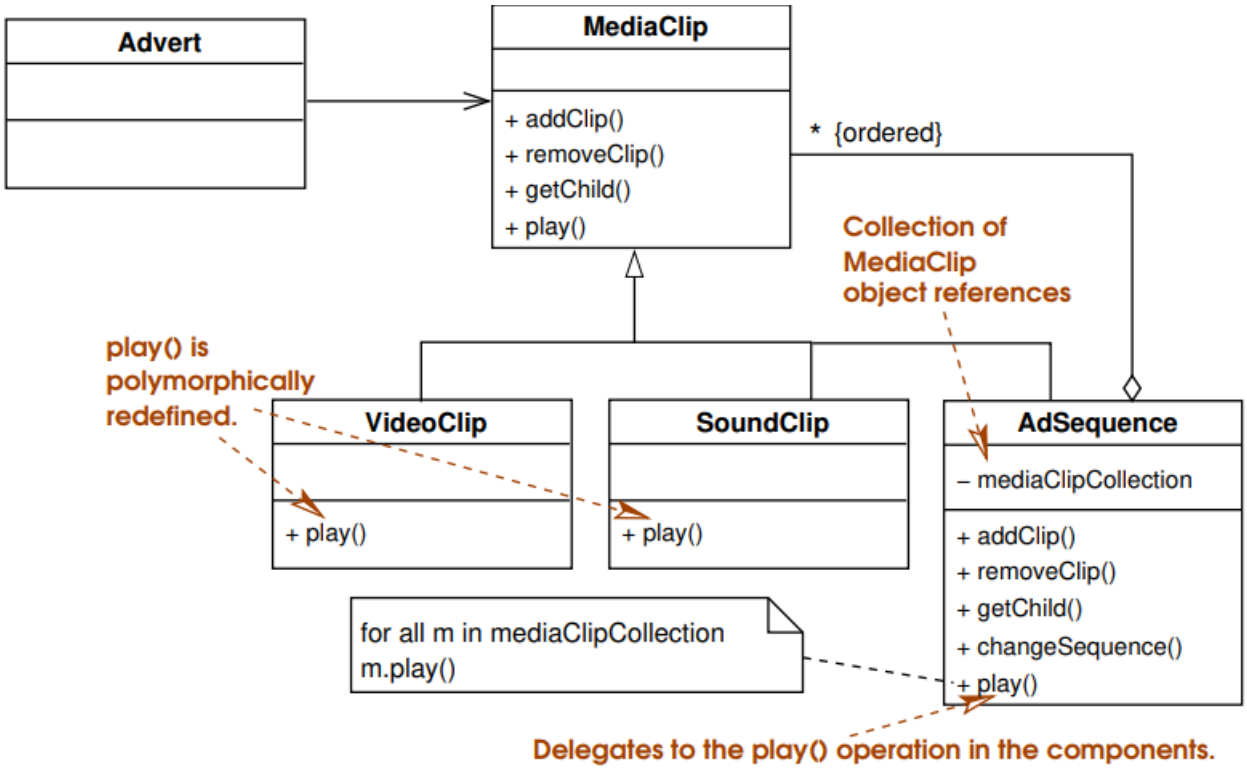


How can we Incorporate composite structures?

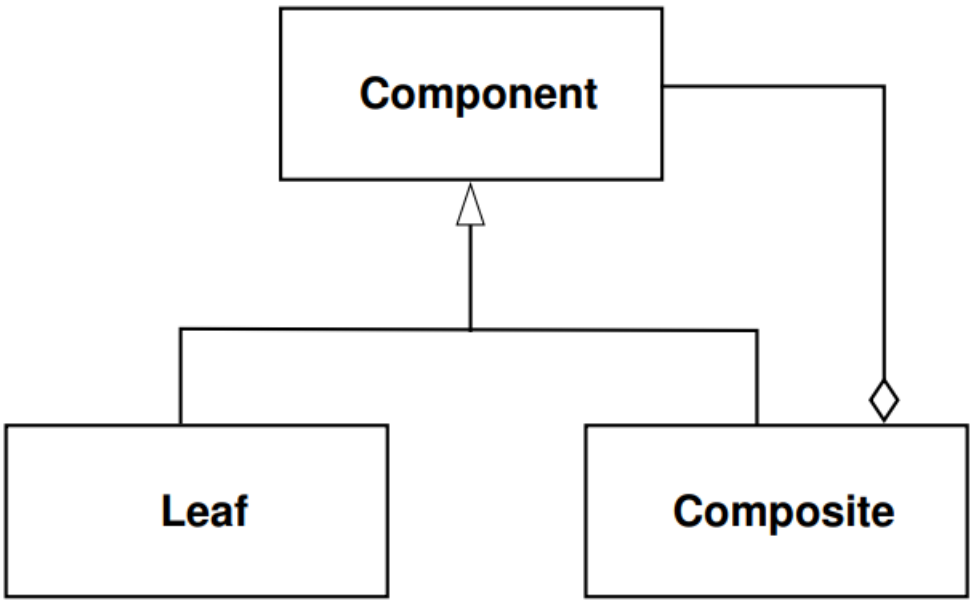


Solution:

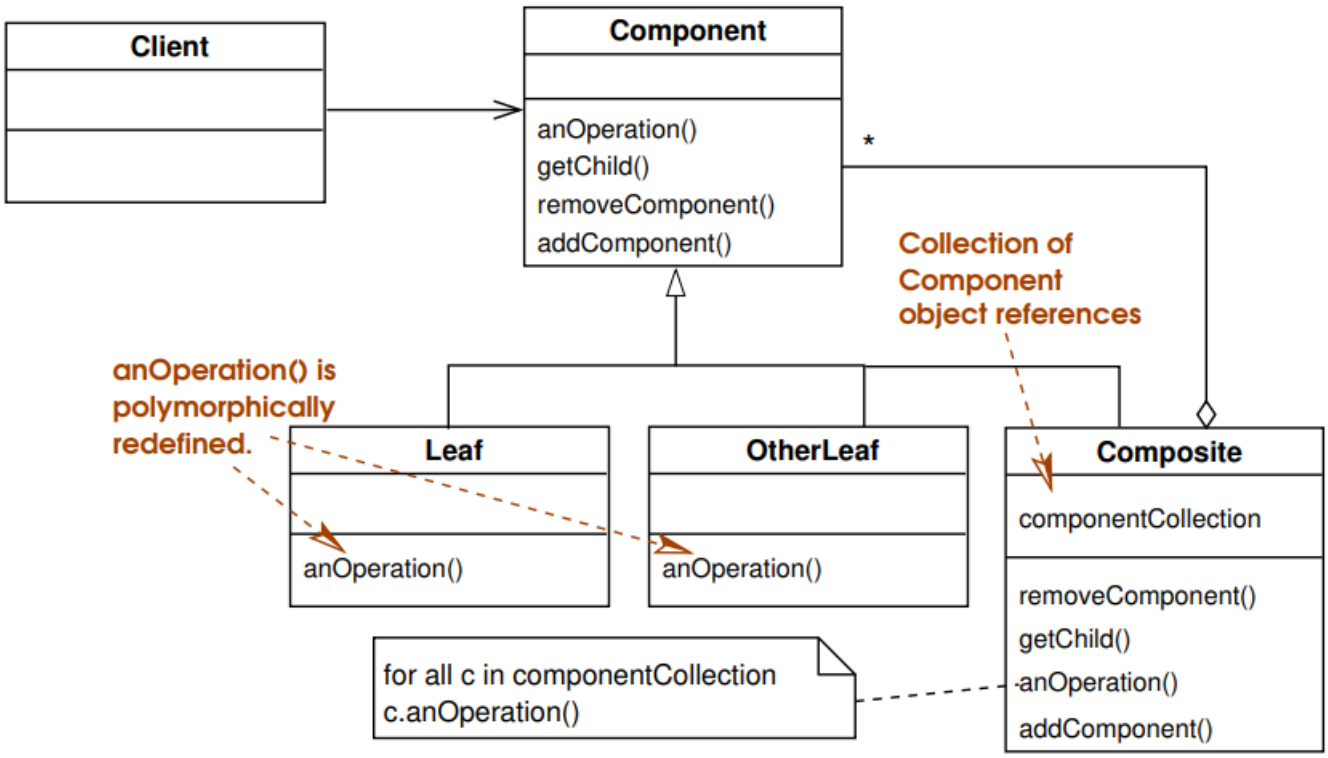
Combine the inheritance and aggregation hierarchies:



Basic Composite Structure:



General Form



Discussion 1:

Consider the *general form* of the Composite pattern:

- The composite pattern is designed to model recursive tree structure to capture a complex *part-whole relation* in a class design.
- The Component may be defined as an interface, an abstract class, or a concrete class, depending whether it has attributes and operations that need to be inherited.

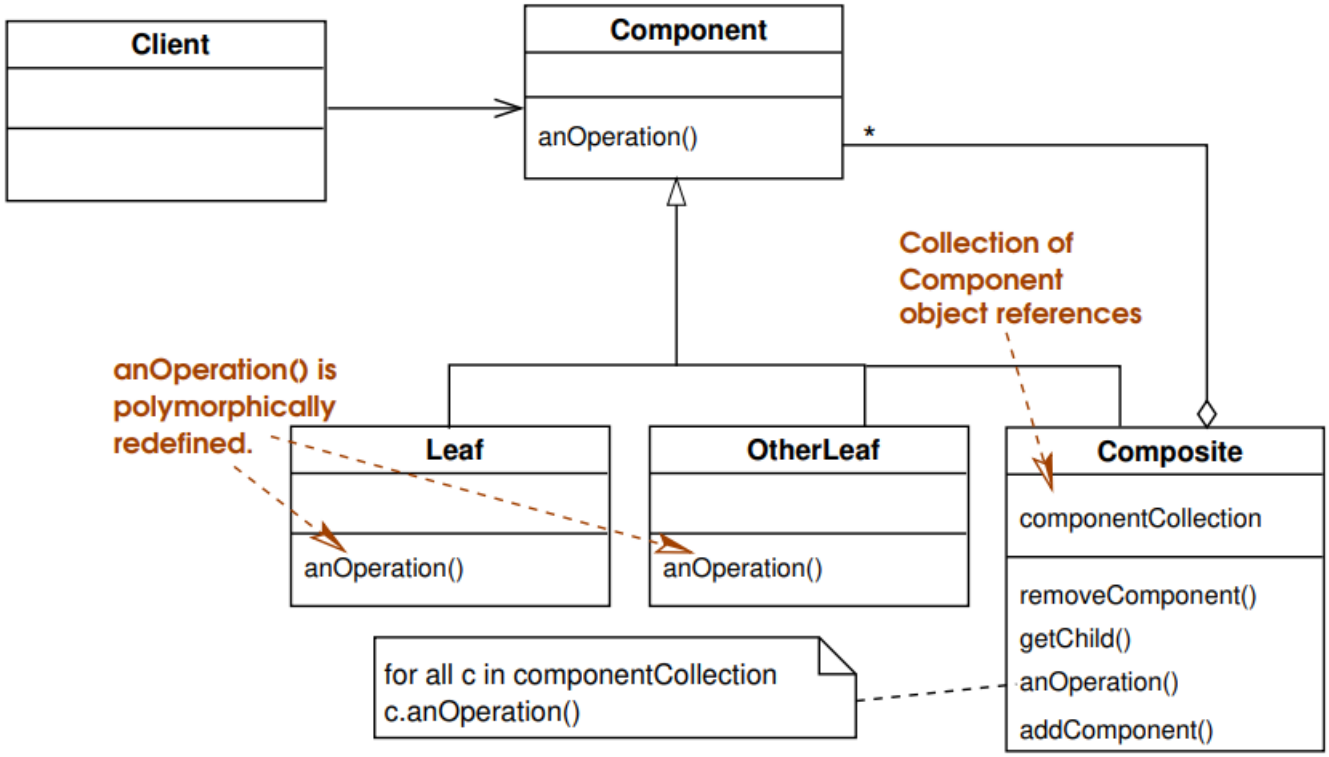
Discussion 2:

When applying the Composite pattern to Java software development, the Composite pattern poses an issue:

- Methods `addComponent` , `removeComponent` , `getChild` are relevant to the Composite, but irrelevant to the leaf classes. Hence it displays a poor level of inheritance coupling and it also violates the Liskov Substitution Principle

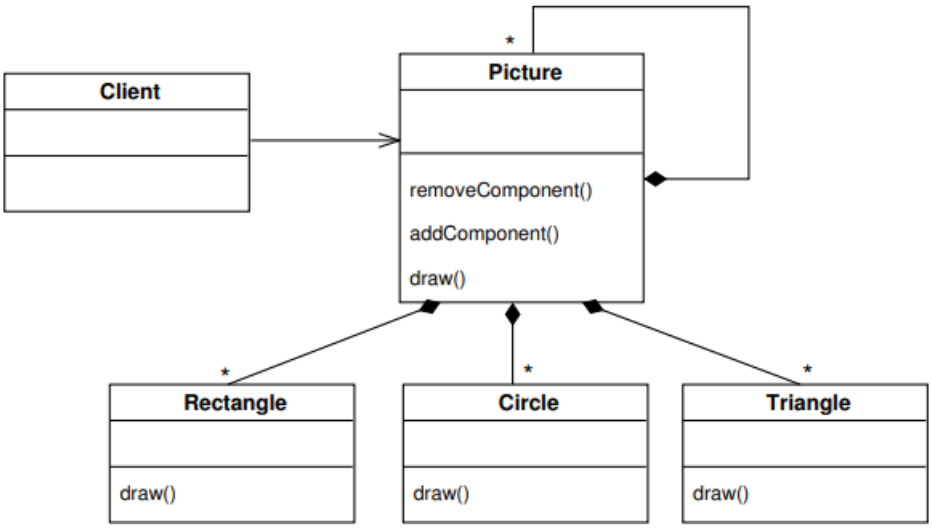
A working solution would be to define the attributes and operations that are common to all the class hierarchy in the Component class and define `addComponent` , `removeComponent` , `getChild` , which are relevant to the Composite only, in the Composite class.

Corrected General Form:



Composite Pattern VS Composite Structure

The part-whole relationship could alternativel be modelled by a simple composition relationship:



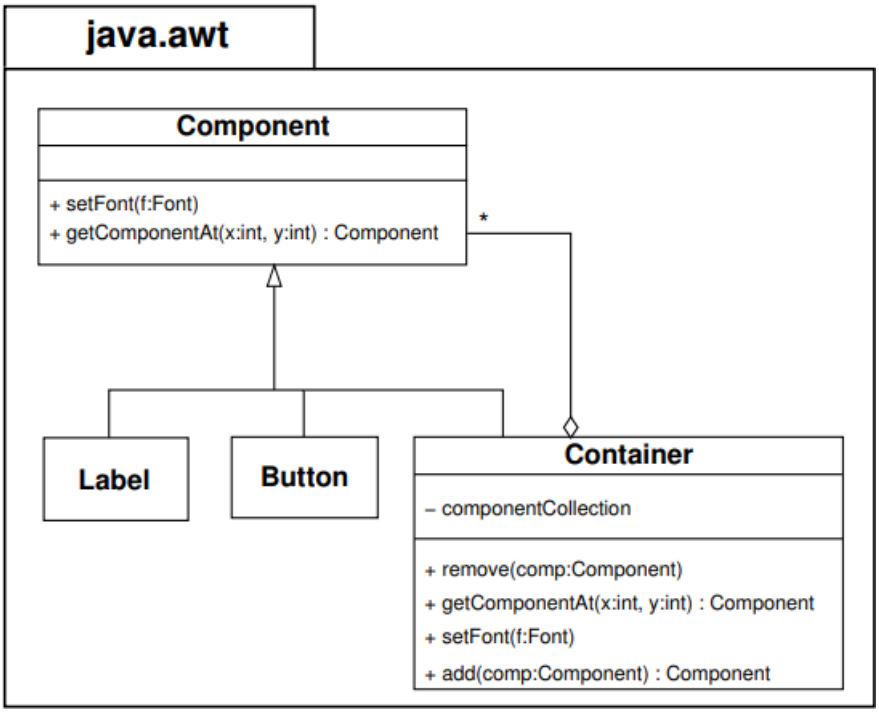
Advantages of using composite pattern:

- Providing more flexibility

    - A client class can collaborate with a part (e.g. Component) or a whole (e.g. Composite) in the same way because they hae the same interface
- Enabling reuse, reducing code duplication
    - Common properties can be defined in the super-class and inherited by both Leaf and composite subclasses
    - E.G. in a graphics application, each graphic component/composite may keep an attribute `isVisible` to indicate if it should be hidden from the view or not.
- Simplify the implementation by avoiding the need of type casting
    - In the composite structure shown earlier, the Composite class keeps a reference to all of its components in a collection
    - However such a collection object will need to be defined of a general type eg Object in java so as to accomodate various types of component objects
    - Type Casting will therefore be unavoidable when operations specific to each type of component need to be carried out.

Example From Java API:

The composite design pattern is used in modelling the part-whole re;ationship between GUI components:




```
1 /** Set the font for a component and all of
2 * its child components to the specified font.
3 * @param component A GUI component
4 * @param font A font to be used within a GUI component
5 */
6 public static void setFont4All(Component component, Font font)
7 {
8 component.setFont(font);
9
10 /* A component may contain other components
11 (e.g. a JPanel object containing JButton objects) */
12 if (component instanceof Container) {
13 /* The given component is a Container object...
14 performs type casting. */
15 Container parent = (Container) component;
16
17 for(Component child : parent.getComponents()) {
18 setFont4All(child, font); // recursion
```

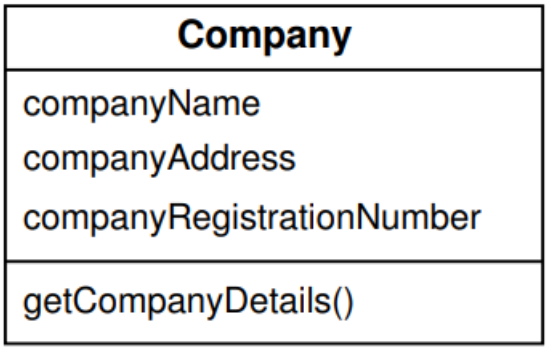
```
19 }
20 }
21 }
```

Creational Pattern : Singleton

(me lol)

Consider Agate case study:

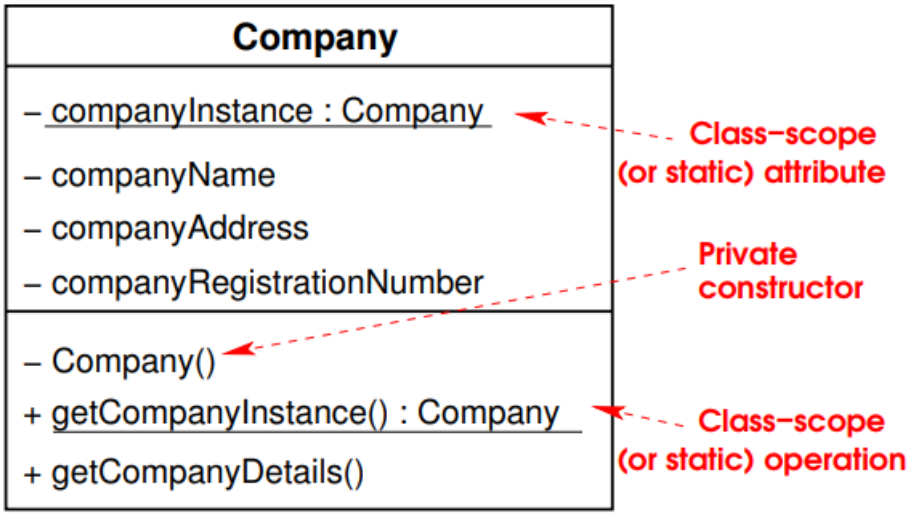
> We need to store information about Agate Company to be accessed by different system objects
>
> How to ensure that onle one instance of the `Company` class is created?



Restrict Access to the constructor!

Use class-scope attribute (i.e. class/static variable) to ensure a single instance

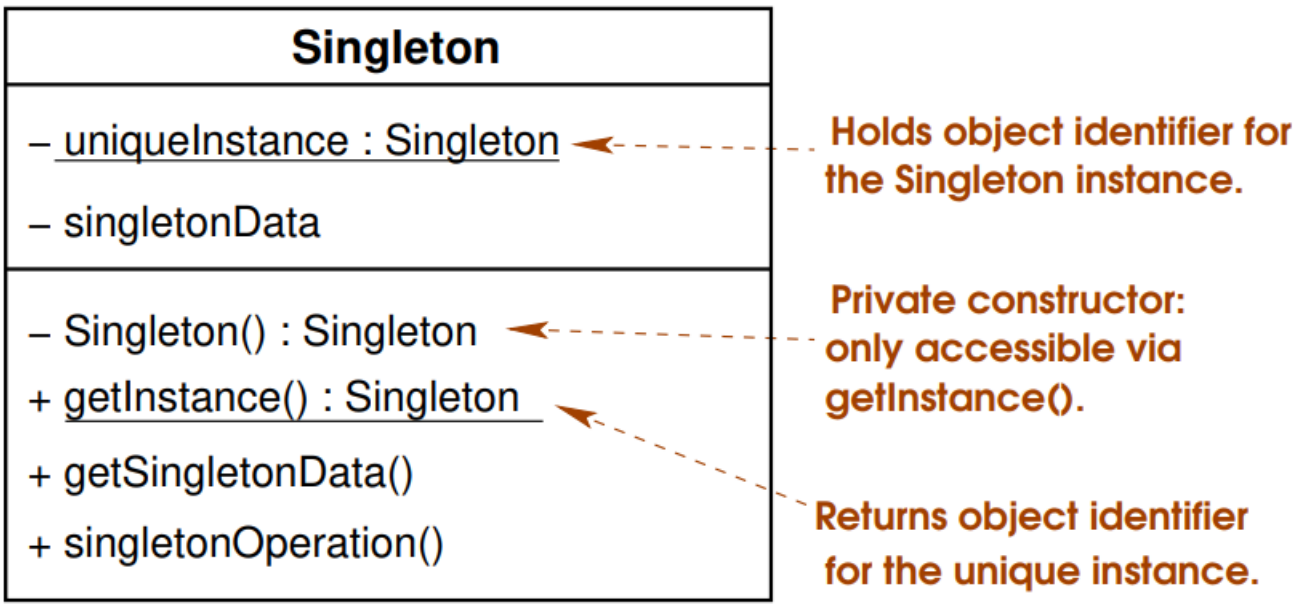
Use class-scope operations (i.e. class/static methods) to allow global access




```
1 public class Company {
2
3 private static Company companyInstance;
4 // other field declaration omitted
5
6 private Company() {
7 // initialise instance variables
8 }
9
```

```
10 public static Company getCompanyInstance() {
11 if(companyInstance == null) {
12 companyInstance = new Company();
13 }
14 return companyInstance;
15 }
16
17 // other class detail omitted
18 }
```

Singleton General Form:



The singleton design pattern is fairly commonly used.

Examples of such in the Java API include:

`java.awt.Desktop.getDesktop`
`java.lang.Runtime.getRuntime`

Context / issue

Consider the class Campaign, which has 4 states:

1. `Commissioned`
2. `Active`
3. `Completed`
4. `Paid`

A `Campaign` object has different behaviour depending on the state it is at.

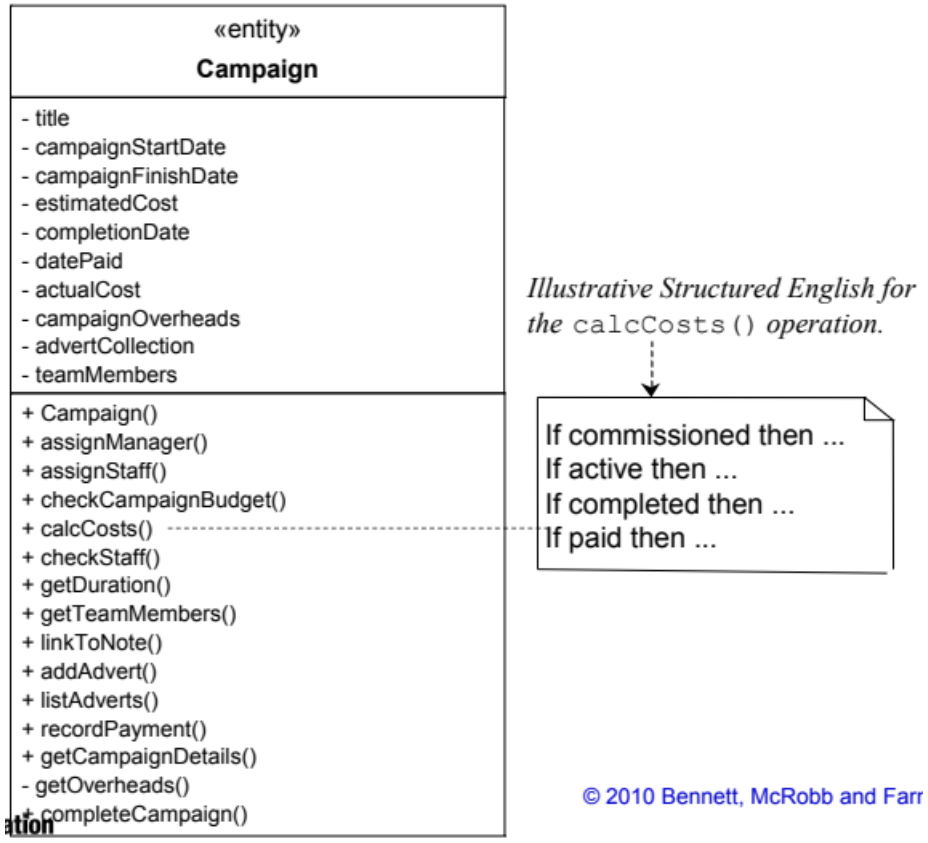
For example, regardless of the state of a Campaign object (i.e. `Comissioned` , `Active` etc) there is the need to calculate costs incurred by this campaign so far. Each state, however, entails a different costing model

How can an object's behaviour change at run-time, based on the state it is at?

Potential Solution:

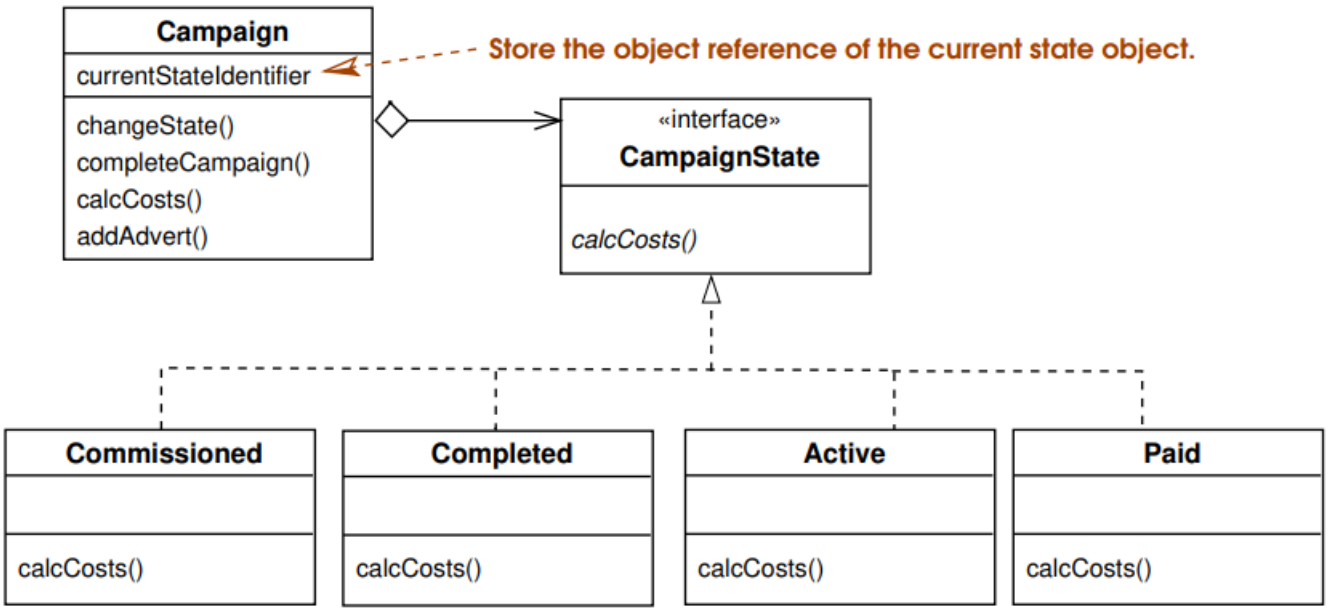
We can use case statements or nested ifs , however this may become very complex , especially if the object has many different states.

Any addition or removal of states will require changes made in the nested if statement.



Potential Solutions: Using separate components

The class may be factored into separate components: one for modelling the required behaviours in each of its states



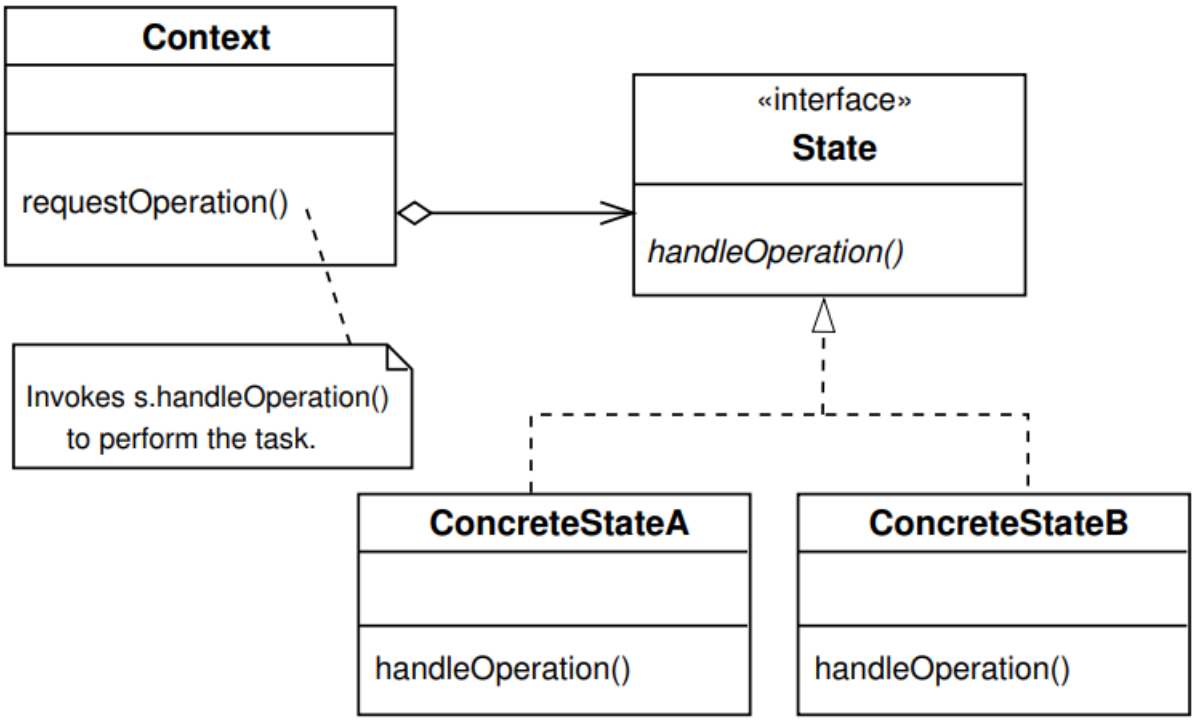
Solution

> 'An object whose behaviour changes depending on the state that it is in delegates its state-dependent operations to other objects which are designed to model state-dependent behaviour'.

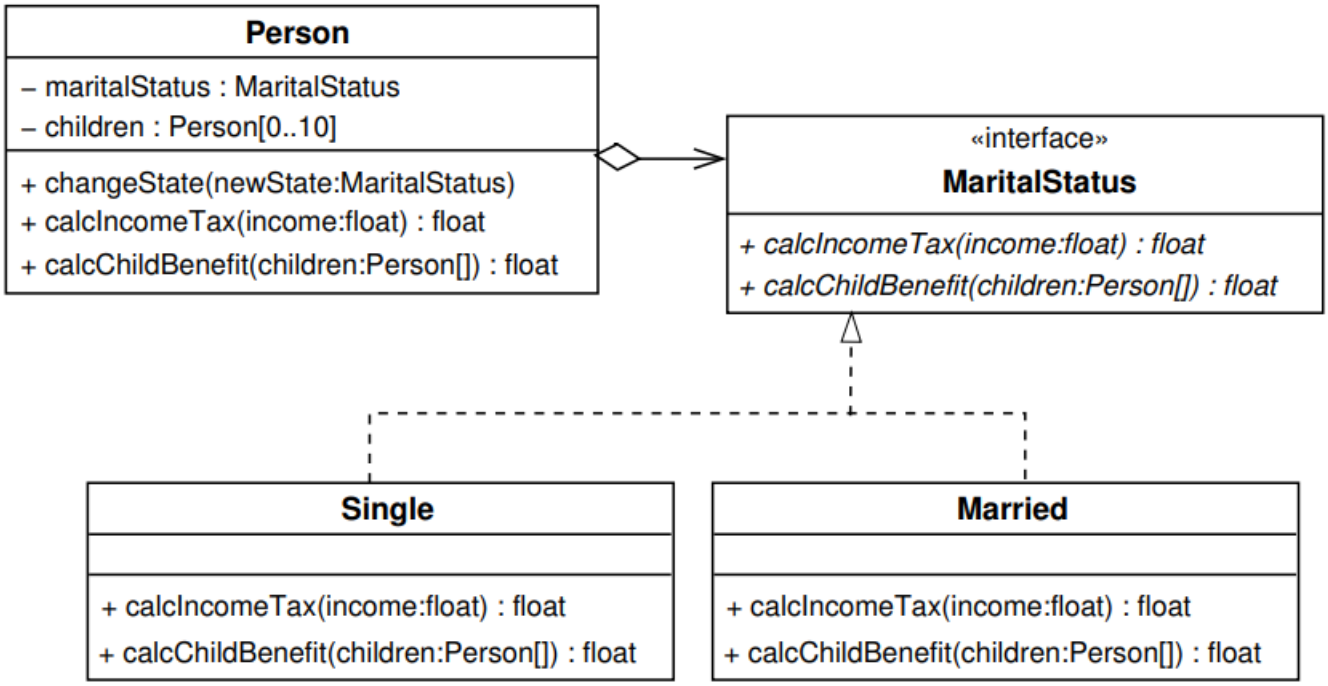
Each state that the object can be in is modelled by a concrete class. These classes implement the same interface which specifies state-dependent operations.

The object keeps a reference to the current state object, which provides polymorphic behaviour.

General Form



Application:



Interface Marital Status:

```
1 public interface MaritalStatus {
2 /** Calculate and returns the amount of income tax to be charged */
3 public abstract float calcIncomeTax(float income);
4 /** Calculate and returns the amount of child benefit entitled. */
5 public abstract float calcChildBenefit(Person[] children);
6
7 }
```

Class Single

```
1 public class Single implements MaritalStatus {
2 public float calcIncomeTax(float income) {
3 // standard income tax charge...
4 // method detail omitted
5 }
6
7 public float calcChildBenefit(Person[] children) {
8 // more child benefit for single parent...
9 // method detail omitted
10 }
11 }
```

Class Person:

```
1 public class Person
2 {
3 private static final int MAX_CHILDREN = 10;
4
5 private MaritalStatus maritalStatus;
6 private Person[] children;
7 private Person spouse;
```

```
8 private float income;
9
10 public Person() {
11 maritalStatus = new Single();
12 children = new Person[MAX_CHILDREN];
13 spouse = null;
14 income = 0.0f;
15 }
16
17 public MaritalStatus getMaritalStatus() {
18 return maritalStatus;
19 }
20 // other fields and methods omitted
21 }
```

To provide state-specific behaviour, the actual calculation of income tax and child benefit entitlement are dony by the state object:

```
public class Person {
private MaritalStatus maritalStatus;
// Other definitions omitted
public float calcIncomeTax(float income) {
return maritalStatus.calcIncomeTax(income);
}
public float calcChildBenefit(Person[] children) {
return maritalStatus.calcChildBenefit(children);
}
}
```

Over their life, a persons marital status in a software system may need to be changed.

Method `registerSpouse(Person)` invokes method `changeState(MaritalStatus)` :

```
1 public void registerSpouse(Person spouse) {
2 this.spouse = spouse;
3 changeState(new Married());
4 }
5
6 public void changeState(MaritalStatus newState) {
7 maritalStatus = newState;
8 }
```

From then on, calculation of income tax and child beenfit entitlement will be done using toperations defined in class Married

Discussion 1

The state pattern enables an object to exhibit different behaviour at run-time in a flexible manner

`state` `objects` which provide different behaviour for the same operation may be created at run-time and attached to the `context` `object` whenever the circumstances of the modelling entity changes

Behavioural Pattern : Strategy

Context / issue

- Consider a word processing application which supports different ways to align the text in each paragraph.

- Depending on the type of document, a different alignment strategy would apply.
- How to avoid hard-coding all the available alignment strategies in the Composition class which is responsible for laying out the text in a document?
- How to ensure that introducing new alignment strategies in future will require minimal changes to existing classes?

Potential Solutions

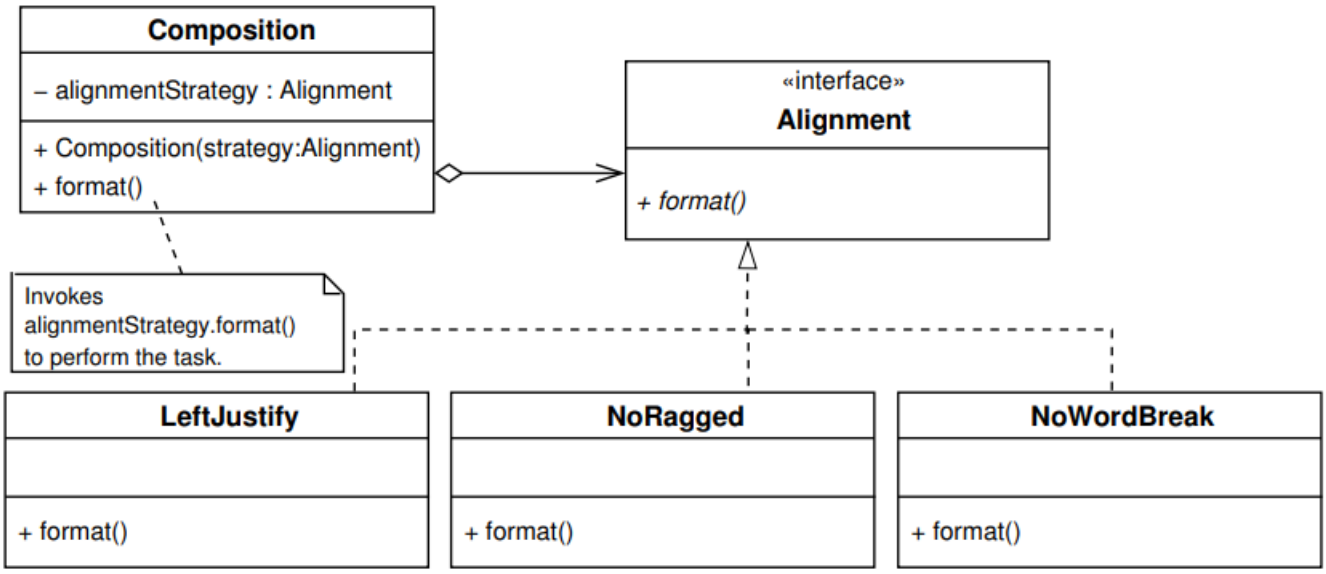
Hard code all alignment strategies in the format method using a set of `if-then-else` statements:

```
1 public void format() {
2 if (isExamPaper(text)) {
3 // allow the text to be ragged along the left margin
4 }
5 else if (isBook(text)) {
6 // disallow ragged along both margins
7 }
8 else {
9 // disallow spreading a word across two lines using hyphens
10 }
11 }
```

Solution 2

Define each alignment strategy in a separate class and make all of them implement the same interface.

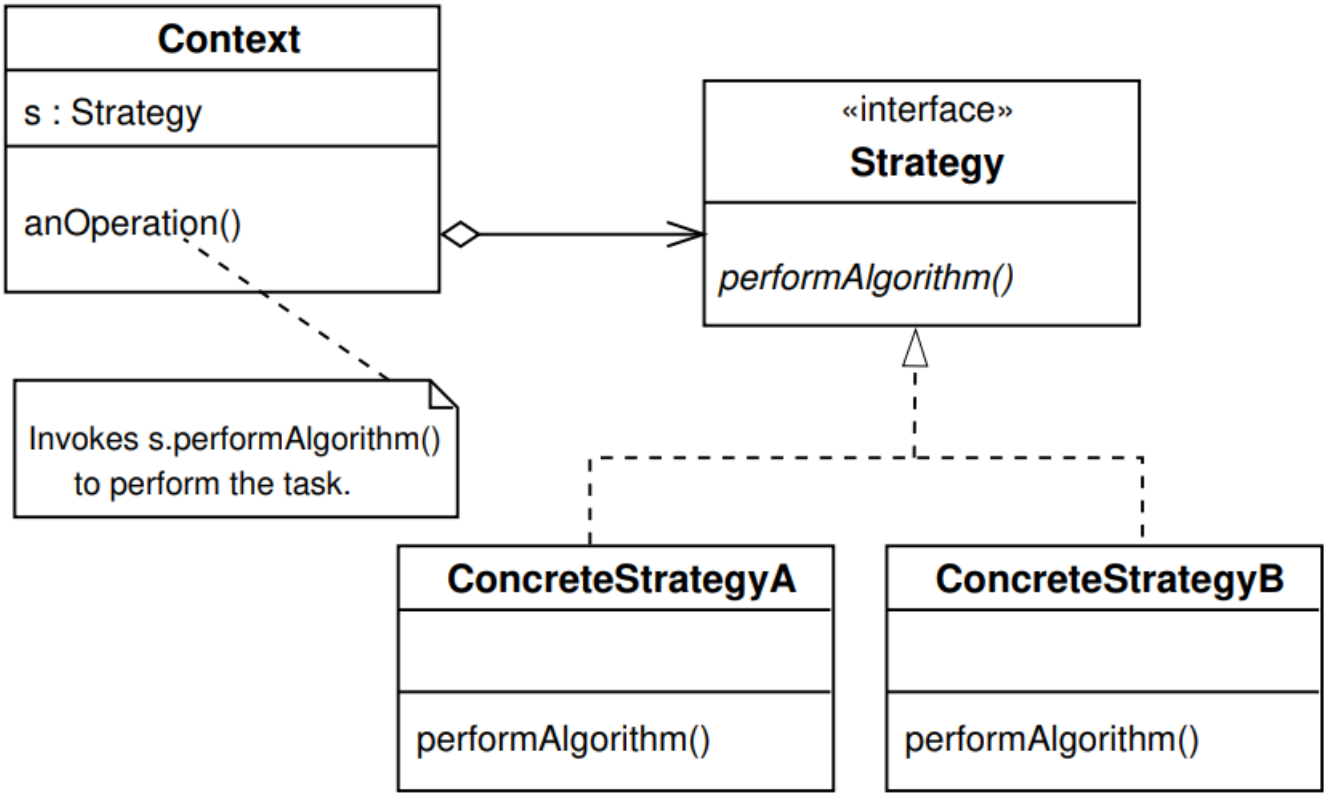
Class Composition will interact with a suitable aligment strategy through the interface



Solution

- Define a family of algorithms using an interface and a set of concrete classes
- the client class may use any one of the algorithms
- Existing algorithms may change without having any impact on the client class

General Form



Discussion

The strategy pattern enables a class to delegate its task to more specialised objects. The context class will not need to keep data required by the family of algorithms. The resulting context class will be less bloated.

In order to perform the required operation, each concrete strategy class needs access to the data in the context class. One way to achieve this is by passing the reference of the context object to the collaborating concrete strategy object.

Code Example

Class `Introductory Offer` :

```
1 public class IntroductoryOffer {
2
3 // constructor and other fields and methods omitted
4
5 public double applyPricingScheme(Campaign c) {
6 // apply 10% discount on calculation of all charges
7 }
8 }
```

Class `Standard` :

```
1 public class Standard {
2
3 // constructor and other fields and methods omitted
4
5 public double applyPricingScheme(Campaign c) {
6 // calculate all charges for this campaign
7 }
8 }
```

Class `Campaign`

```
1 public class Campaign {
2 private Pricing strategy;
3
4 // irrelevant details of the constructor omitted
5 public Campaign(Pricing strategy, ...) {
6 this.strategy = strategy;
7 ...
8 }
9
10 // other fields and methods omitted
11
12 public double calculateCost() {
13 double charge = strategy.applyPricingScheme(this);
14 ...
15 }
16 }
```

Class `salesSubSystem` :

```
1 public class SalesSubsystem {
2
3 // constructor and other fields and methods omitted
4
5 public void addCampaign(...) {
6 ...
7 // The standard pricing scheme applies.
8 Campaign c = new Campaign(new Standard());
9 ...
10 }
11 }
```

Difference between State and Strategy?

- Both use polymorphic classes.
- Both have a context (e.g., editing or browsing).
- Both actually apply different strategies (e.g. use different versions of an algorithm).
- Both are trying to avoid inflexible nested case- or if-statements.
- Both are actually about doing work in alternative ways, and a `ConcreteState` actually encapsulates its own group of strategies for executing functions.
- 'The State-Pattern is a way to vary an object's behavior dynamically, whereas the Strategy-Pattern is a way to vary the implementation'

Structural Pattern Adaptor

Consider a software application (gimp (no the insult we're on about MyDraw here)) for users to draw and arrange graphical elements (e.g. lines, polygons, text etc) into a diagram.

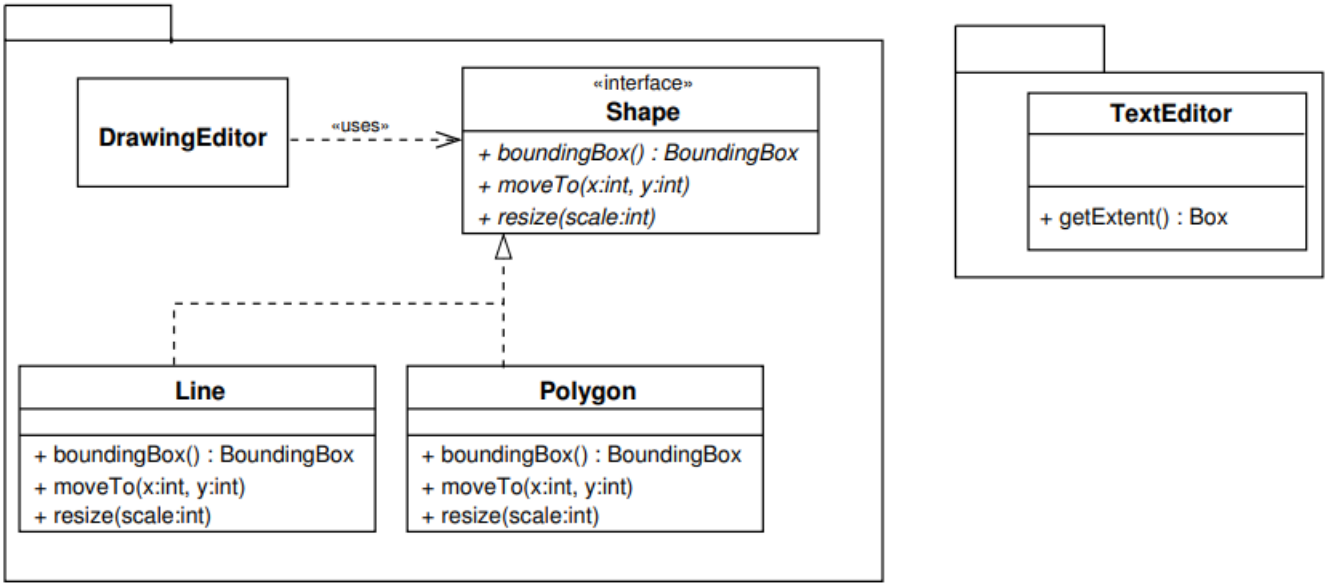
Context / issue

We define an `abstract` `class` (e.g. `Shape`) for modelling a generic graphical element, Subclasses of `SHape` would model `Line` ,

`Polygon` , `Circle` , etc

If there is a `TextEditor` `framework` class which adresses the requirements of our intended `Text` class

However , `TextEditor` does not have the same interface as the `abstract` `class` `Shape` which `MyDraw` expects to work with.



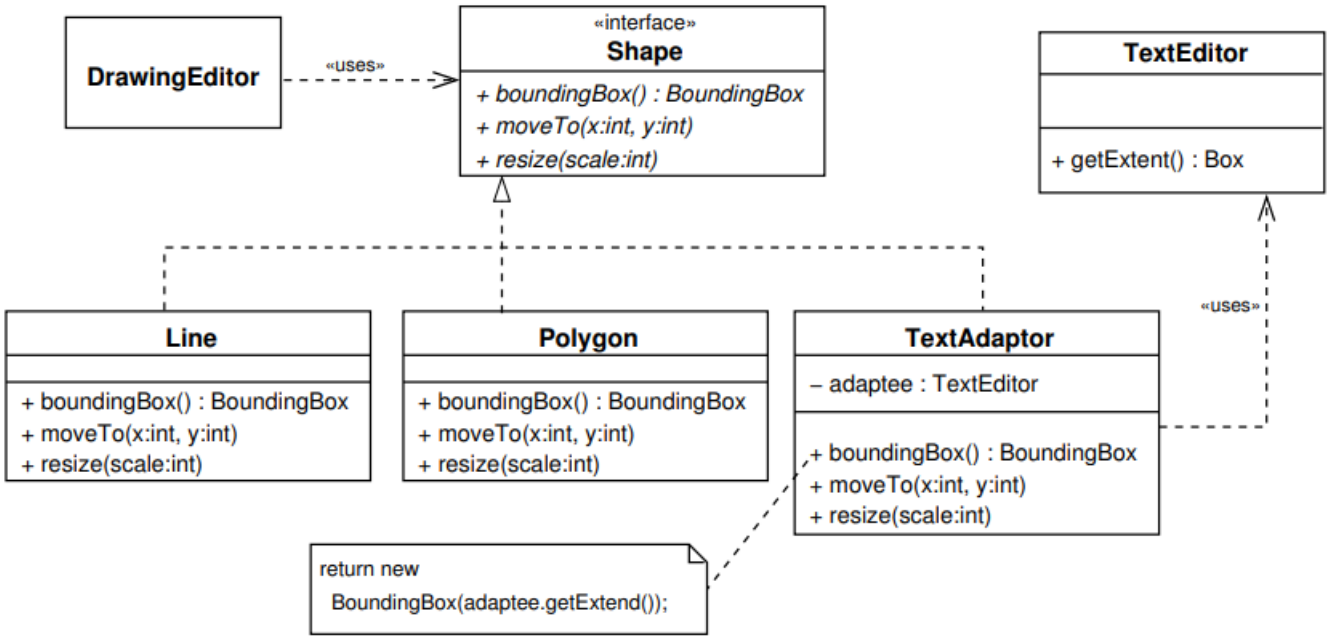
Solution

Using an `Adapter` class , convert the interface of a class into anothe rinterface which clients expect

Client call operations on an `Adaptor` instance

The `adapter` then invokes the respective `Adaptee` operations to complete the task

The Adapter class will also include concrete implementation of other methods which are defined by the implementing interface but not supported by the `Adaptee` class.



General Form

Java API examples

Examples of the adaptor design pattern in the Java API include:

- `java.util.Arrays.asList`
  - Method `asList` adapts an array to the collection type list.
- An adaptor is particularly useful when you want to use an existing class, but its interface does not match the one required by your application.

Applying Patterns

Before using a pattern to solve the problem, cosnider:

- is there a simplet solution?
- Is the context of the pattern consistent with that of the problem
- Are the consequences of using the pattern acceptable
- Are constraints imposed by the software environment that would conflict with the use of the pattern

After selecting a suitable pattern ...

1. Read the pattern to get a complete overview.
2. Study the Structure, Participants and Collaborations of the pattern in detail.
3. Examine Sample Code to see an example of the pattern in use.
4. Choose names for the pattern's participants (i.e. classes) that are meaningful for your application.
5. Define the classes, with operations that perform the responsibilities and collaborations in the pattern.

Week 11 Implementation

Objectives

- [ ] this lol

Notes

Implementation

> "Implementation is concerned with the process of building the new system. This involves writing program code, developing database tables, testing the new system, setting it
> up with data, possibly transferred from an old system, training users and eventually switching over to the new system."

> "consists of programming,
> which is the translation of the software design developed in
> the [design] phase to a programming language. It also involves integration, the assembly of the software parts."

What does Implementation involve

Implementation involves constructing a software system based on a design: this involves writing program code.

To facilitate the coding process we must consider:

- Which programming languages to use
- Which software tools will be used to support the development process
- How to ensure that the code, through written by different programmers, will be easy to read, understand and maintan
- How to effectively manage changes to programs throughout the development process.

Language Choice

The choice of programming languages mainly depends on the nature of the required software system.

- JS, CSS, HTML, for client-side Web applications
- For server side:
  - PHP, JAVA EE, ASP.NET, Ruby, R, Python
- Lots of simultaneous, lightweight communications? maybe Node.js with websockets.
- Where a client prefers to use Microsoft products only the implementation language is likely to be C#.

Language Choice

- If the task requires a substantial amount of processing on tree data structures, a functional programming language such as Lisp, Haskell or possibly Julia is likely to be a good choice.
- If a system is expected to perform a lot of low-level system operations and efficiency is paramount, C may be a preferred language choice.

- If the required software system is expected to run on different platforms a language like Java or Scala would be appropriate.

Programming Conventions

Writing Classes

Class names should be meaningful, e.g. in the Agate example, `CampaignStaff` would be a better class name than Staff.

Avoid using ambiguous or cryptic names such as Record or C1.

The purpose of each class should be clearly and concisely stated in the documentation comment.

> This view is not universally agreed. Some software developers adopt a "comment-free" practice, and opt to achieve clarity by structuring the source code and naming identifiers, methods, variables, etc. in a way that make the code self-explanatory.

Documenting Methods

The purpose of each method should be clearly stated in code comments:

```
/** Tests whether this date is after the specified date. **/
public boolean after(Date when)
// detail omitted.
```

To facilitate maintenance and reuse, it is helpful to specify the following information for methods, especially for the non-trivial ones:

Known issues	intent
Precodnitions	return
post-conditions	exceptions
invariant	

Intent - An informal statement of what the method is intended to do

Return - the balue which the method returns

Known issues - Honest statement of what has to be done, defects that have been repaired

Exceptions - Capture the situation when an abnormality occurred during the execution of the method

> Such situations are often caused by the preconditions not beeing met

Preconditions - what must be true when a method is invoked

Conditions on non-local variables including input parameters, that the method assumes:

```
private int price;
// other detail omitted.
/* Precondition: reduction > 0 */
public void reducePrice(int reduction) {
price -= reduction;
}
```

Verification of these conditions is not guaranteed by the method as it is assumed to be done by the caller.

Post-conditions - what must be true after a method completes successfully.

Specify the effect of the method

Specify the value of non-local variables after execution

Notation x' denotes the value of variable x after the execution:

```
private int price;
// other detail omitted.
/* Post-condition: price' = price - reduction */
public void reducePrice(int reduction) {
price -= reduction;
}
```

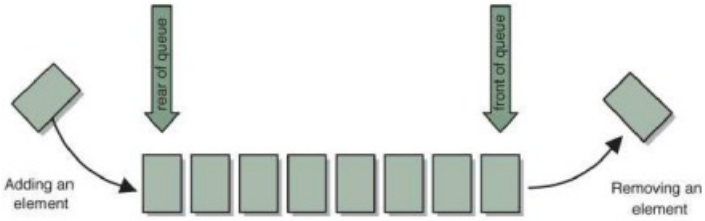
Class Invariants - specify constraints on attributes and their relationships:

```
public class Time {
private int hour; // hour >= 0 && hour < 24
private int minute; // minute >= 0 && minute < 60
private int second; // second >= 0 && second < 60
// other detail omitted.
}
```

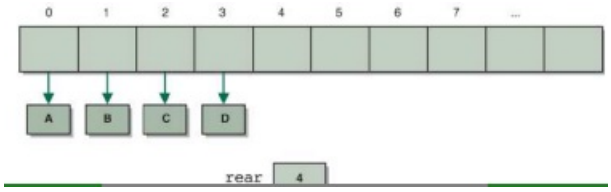
Even though the value of each attribute in an object might change at runtime, its values must conform to the class invariants

Exercise:

consider this queue:



`BoundedArrayQueue` is a queue data structure modelled by an array. The capacity of the queue is fixed.



Invariants for class BoundedArrayQueue:

```
public class BoundedArrayQueue<T> {
/* Class invariant: ... */
private T[] contents;
private int rear; // the index of the next vacant cell
private int size;
public BoundedArrayQueue(int capacity) // detail omitted.
public boolean enqueue(T element) // detail omitted.
public T dequeue() // detail omitted.
```

```
}
```

```
/* Class invariant:
* size >= 0 && size <= contents.length
* rear == size
*/
```

Java Assertions

Preconditions, post-conditions and invariants can be specified in a Java program as assertions.

Assertion in Java provides a convenient tool for run-time checking.

as well as a means for documentation

An assert statement typically:

- performs a check, and
- displays a message when the check fails.
- An assert statement may also:
- throw an exception, and
- include file and line info.

By default, assertions are ignored by the JVM and they are used mainly for debugging.

To switch on assertion checks, a Java program needs to be run with the option -ea (i.e. enable assertions), e.g.:

`java -ea ArrayRemove`

where `ArrayRemove` is a compilation unit that includes a main method.

```
public static String removeElement(String[] array, int pos) {
// check a precondition:
assert 0 <= pos && pos < array.length :
"removeElement: precondition failed";
// remember the original array for post-condition checks:
String[] originalArray = array.clone();
String result = array[pos]; // the result
// shift all elements after pos one position left:
for (int i = pos; i < array.length - 1; i++) {
array[i] = array[i + 1];
}
array[i] = null; // the last element becomes null
// check some post-conditions:
assert array[array.length - 1] == null:
"removeElement: post-condition failed";
assert pos + 1 == array.length
|| array[pos].equals(originalArray[pos + 1]):
"removeElement: post-condition failed";
return result;
}
```

Coding Standards

Naming standards should be agreed early in a project.

A typical object-oriented standard:

Classes with capital letters: Campaign

Attributes and operations with initial lower case letters: `title`, `recordPayment()`

Words are concatenated together with capital letters to show where they are joined, e.g.:

`InternationalCampaign,`
`campaignFinishDate,`
`getNotes()`

Hungarian Notation

The Hungarian notation is typically used in C or C++.
Identifiers are prefixed by an abbreviation to show the type of the variable:

`b` for `boolean`: `bOrderClosed`

`i` for integer: `iOrderLineNumber`

`btn` for button: `btnCloseOrder`

Underscore convention

Another convention commonly used for column names in databases uses underscores to separate parts of a name instead of capital letters, e.g.:

`Order_Closed`
... or, in Postgres...

`order_closed`

This convention makes it easier to replace the underscores with spaces to produce meaningful column headings in reports.

Document Code

The one who wrote a piece of code is unlikely to be the one having to maintain it in future. Hence, it is important that code is readable to facilitate maintenance.

Having good documentation is important to facilitate maintainability and reusability of any written code.

Java has well-defined documentation standards which should be adhered to if coding in Java (Javadoc).

Use XML tags if coding in C#.

Many IDEs include features for generating Javadoc from comments automatically.

Unreadable code

```
// another method to help me
private static void raa (char[] a)
{ final int s = a.length; for (int i = 0;i < s/2;i ++)
{char c = a[i];a[i] = a[s-i-1];
a[s-i-1] = c;
}}
```

Better →

```
/**
```

```
* reverse an array of characters in its own space
* @param array
*/
private static void reverseArray (char[] array)
{
final int size = array.length;
for (int i = 0; i < size/2; i ++) // first half of array
{
// swap i'th and mirrored (size - i - 1)'st element
char c = array[i]; // temporary storage
array[i] = array[size - i - 1];
array[size - i - 1] = c;
}
}
```

improved →

```
private static void reverseArray (char[] array)
{
int index = 0; // start with the first element
int mirrorIndex = array.length - 1; // and the last
while ( index < mirrorIndex )
{
// swap index and mirrorIndex elements
char c = array[index];
array[index] = array[mirrorIndex];
array[mirrorIndex] = c;
// advance counters
index++;
mirrorIndex--;
}
}
```

Naming Conventions

Naming conventions for. . .

- packages: all lower case
- internal use: prefix with name of product, e.g.: cwk1.shapes
- for distributing: prefix with domain name, e.g.:
  - `uk.ac.aston.cs`
- The word in between each `fullstop` corresponds to the name of a folder in the file system.
- reference types (i.e. class/interface names): start with a capital, then mixed case.
- classes: use a noun describing an instance, e.g.: `MyShape`.
- interfaces: either a noun, e.g. `Observer`, or an adjective, e.g. `Cloneable`.

- methods: start with lower case, than mixed case
- Active methods (i.e. `mutator` methods) usually start with verb, e.g. `changeSize()`
- Getter methods (i.e. accessor methods), which instigating no change, are named by return value, e.g. `length()`, or with a '`get`', e.g. `getLength`().
- fields and constants
- Static final fields (i.e. named constants): all capitals, e.g. EQUALS_TOLERANCE, INITIAL_VALUE.
- All other fields use the same capitalisation as method names, e.g. `size`, `isVisible`.

- parameters: same capitalisation as method names, e.g. `newSize` .
- Parameters in public methods appear in the documentation.
- Choose fitting, yet succinct names for parameters.
- Parameters having the same meaning should have the same name, even when they appear in different methods, e.g.:

```
distance(Location anotherLoc)
almostEqual(Location anotherLoc)
```

I local variables: same capitalisation as method names, e.g.: `count`

Always use meaningful names good for achieving readability.

Except for routine cases, e.g. `for(int i=0; ...; i++)`

Layout Conventions

Matching braces must line up vertically or horizontally

every method except main should be preceded by a doc comment.

A method definition 'should not be longer than 30 lines'.

switch statement should not be used, except with enum values.
All keywords and operators should be surrounded by a space:

```
if (x == 0) y = 0; // good
if(x==0) y=0; // bad
```

No "magic numbers" should be used (i.e. direct usage of numbers in the code)

Some exceptions for numbers with generic meanings i.e. (−1, 0, 1, 2)
BUT...

Clearly-named constants or methods are still preferable

e.g. number % 2 === 0

...could be replaced with an `isEven` function

Magic Numbers Ecercise:

```
public static int timeToPension(int ageNow, int currentYear)
{
int statePensionAge = 65; // initial assumption
if (currentYear + 65 - ageNow >= 2024) {
statePensionAge = 66;
}
if (currentYear + 66 - ageNow >= 2034) {
statePensionAge = 67;
}
if (currentYear + 67 - ageNow >= 2044) {
statePensionAge = 68;
}
return statePensionAge - ageNow;
}
```

Improved:

```
final int SPA_66_YEAR = 2024;
final int SPA_67_YEAR = 2034;
final int SPA_68_YEAR = 2044;
public static int timeToPension(int ageNow, int currentYear) {
int statePensionAge = 65; // initial assumption
if (currentYear + 65 - ageNow >= SPA_66_YEAR) {
statePensionAge = 66;
}
if (currentYear + 66 - ageNow >= SPA_67_YEAR) {
statePensionAge = 67;
}
if (currentYear + 67 - ageNow >= SPA_68_YEAR) {
statePensionAge = 68;
}
return statePensionAge - ageNow;
}
```

Comments

start with /** and end with */

are used to automatically generate documentation of types, methods and fields (e.g. in Eclipse, NetBeans, BlueJ)

may contain simple HTML tags (e.g. for bold style, bullet
points)

I may contain a number of special tags, e.g.:
I @author
I @version
I @see hyperlink to documentation of another member, class
I @param every method parameter should have one
I @return describes the return value of a method
I @throws describes potential exception propagation

Defensive Programming:

Defensive Programming is a programming technique which seek to minimise program errors by anticipating potential errors and implementing code to handle them.

Techniques for defensive programming include:

- various error handling techniques
- exception handling
- buffer overflow prevention
- enforcing intentions

Error Handling Techniques

Wait for a legal data value: typically used when getting data from an external source, e.g. user interface

Set a default value: typically used to ensure the continuity in execution of a (critical) process

Use the previous result: typically used when a constant stream of inputs is expected at a regular interval

Log error: store error info for later use or analysis

Throw an exception

Abort: typically used in operations where illegal data may lead to a fatal consequence, hence the system must be aborted and reset.

Buffer Overflow Prevention

Some languages (e.g. C and C++) permit data to be written to memory even the data require more space than what was declared in the code, so long as the required space does not
exceed the space allocated to the program (e.g. through the malloc function).

This may lead to overwriting existing data kept in the memory.

Such overwriting, when exploited, may lead to a security
breach.

Buffer overflow prevention attempts to prevent buffer overflow by checking the size of variables.

Enforce Intention

Use suitable programming constructs to enforce the intended design or use a piece of code.

Use Keywords such as `final` and `abstract` to enforce coding intentions e.g.

```
public final class String {...}
public abstract class AbstractList {...}
```

Make constants, variables and classes as local as possible:

```
for (int i = pos; i < array.length - 1; i++) {...}
```

rather than:

```
int i;
for (i = pos; i < array.length - 1; i++) {...}
```

Enforce Intention

Use the singleton design pattern if only one instance is expected from a class.

Define attributes and operations as private if they are not intended to be accessible by other classes.

If a class is expected to be extended by other classes, make attributes protected and define accessor methods for accessing their values.

If a method is not expected to be used by classes in another package, do not use the public keyword to specify its visibility.

Use enumerate type to enable a variable to be assigned with a predefined set of constants, e.g.:
Rather than:

```
int tShirtSize = 1;
```

we define an enum type in Java to model permissible sizes for a T-shirt:

```
public enum Size {
XS, S, M, L, XL, XXL
}
```

```
Size tShirtSize = XS;
```

Consider introducing new classes to encapsulate legal parameter values so as to prevent bad data:

```
determineRoadTax(String vehicle)
```

we use:

```
determineRoadTax(Vehicle vehicle)
```

Class `Vehicle` ,ight have various factory methods to create the permissible types of vehicles:

```
Vehicle createCar() {...}
Vehicle createLorry() {...}
Vehicle createMotorbike() {...}
```

Software Modelling tools

Modelling tools

Many interactive development environments (IDEs) support UML.

However, the notations used might not always strictly follow the current standard.

Some modelling tools support automatic generation of program code (in Java, C++ and VB) from the models, e.g. Visual Paradigm.

Some also support reverse engineering of code (i.e. generating a design model from existing code).

Some of them map classes to a relational database.

IDE's

Manage files in a project and the dependencies among them.

Link to configuration management tools.

Use compilers to build the project, only recompiling what has changed.

Provide various facilities such as debugging, refactoring, auto-layout, documentation generation, etc.

May include a visual editor for GUI building.

Can be configured to link in third party tools.

Configuration Management Tools

Include elements of version control - to compare source code so as to assist in resolving potential conflicts.

Though configuration management is more than just version control.

Maintain a record of file versions and the changes from one version to the next.

Record all the versions of software and tools that are required to produce a repeatable software build.

Docker provides many of these configuration management functions.

DBMS

A DBMS typically comprises:

- Server system
- Client software (administration interfaces, ODBC and JDBC drivers)
- Tools to manage the database and carry out performance tuning
- Tools to make compiled classes persistent so they can be used with object DBMS
- Large DBMS, such as Oracle, come with many tools, even their own application server

Application Containers

Web containers, e.g. Tomcat (Java), Flask (python)

Run servlets and small-scale applications.

Application servers, e.g. `GlassFish`, `WebSphere`, `nginx`, `Apach Web Server`

Provide a framework in which to run large-scale, enterprise applications.

Installation / Deployment tools

e.g. GitLab CI/CD, AWS CodeDeploy, Jenkins...

Automate extraction of files from an archive and setting up of configuration files and registry entries.

Some maintain information about dependencies on other pieces of software and will install all necessary packages

Uninstall software, removing files, directories and registry entries.

Week 12 Refactoring

Objectives

- [ ] this lol

Notes

Fowler SE tips:

> "Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

> "When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature."

> "Refactoring changes the programs in small steps. If you make a mistake, it is easy to find the bug."

> "Before you start refactoring, check that you have a solid suite of tests. These tests must be self-checking."

What is refactoring?

> "Refactoring is a process of altering source code so as to leave its existing functionality unchanged."

Refactoring refers to the process of restructuring source code in order to impeove its readability, maintainability and extendability without altering any observable behaviour of the software system.

Through a key for refactoring may be to improve the source codes ability to cope with new requirements

- in preparation for system extension, refactoring itself does not, and should not lead to any change in the functional requirements which the source code is to address

This allows the same suite of automated tests to be used

Refactoring : Reanming

Renames an element and if enabled corrects all references to the elements also in other files.

An element may be a:

- method,
- method parameter,
- field,
- local variable,
- type,
- type parameter,
- enum constant,
- compilation unit,
- package

Refactoring: Promoting attribute to class

- Create a new class for modelling an attribute
- Typically, the promoted attribute was defined as a primitive or `String` type previously
- The new class enables further information to be modelled

Before

```
public class Customer {
private String firstName;
private String surname;
private String address;
// other field and method
// definitions omitted
}
```

After

```
public class Customer {
private String firstName;
private String surname;
private Address address;
// other field and method
// definitions omitted
}
```

Why is there need for refactoring?

Refactoring is particularly relevant in Agile development.

In Agile project lifecycle, the typical phases in a waterfall model are performed repeatedly in cycles i.e. requirements, capture, design, implementation, testing, requirements etc

in each cycle, new code is written and added to the existing codebase

At each cycle, te focus is to develop code the addresses the current requirements

At the end of each cycle, the code base needs to be “cleaned up” in order to prevent it from becoming too messy and difficult to maintain.

Refactoring facilitates maintenance and extension of the code base.

Big Refactorings

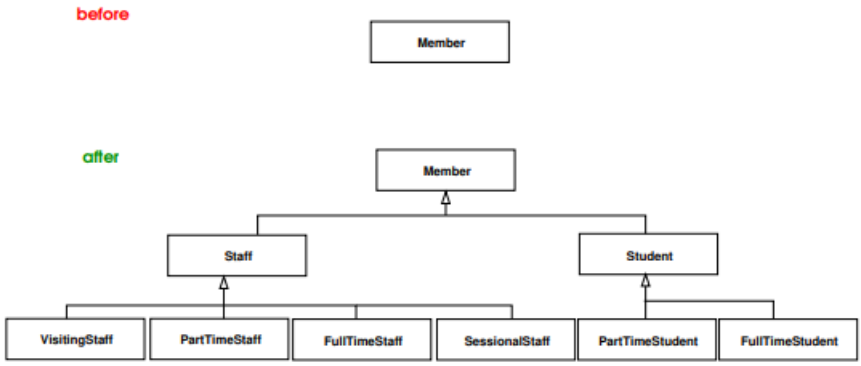
This kind of refactoring changes the overall system design in terms of the number of classes involved

We consider 4 kinds of *big refactorings* :

- Extract Hierarchy
- Tease apart inheritance
- Convert procedural design to objects
- Separate domain from presentation

Extract Hierarchies

Refine the current class design by introducing a new class hierarchy or extending an existing hierarchy.

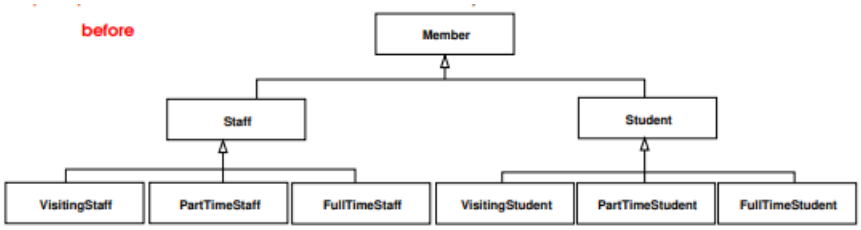


The original member class most likely had code that wasn't used in other sub classes , like students can have scholarships but this doesn't apply to staff.

Classes will inherit what they need but be more specialised than member.

Tease Apart Inheritance

To alleviate the potential for class explosion, Identify common properties and model them as separate class hierarchies:



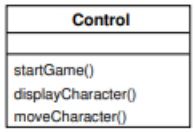
The member class has a status , which allows them to do particular things.

The status object the member owns will have methods that allows them to do a particular thing.

Convert procedural design to objects

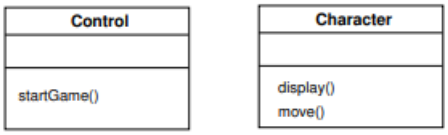
Improve the design by removing procedural components into classes:

before



Taking procedural , long strings of scripts and methods , and thinking about breaking out methods that have entities in common and breaking them out into a different class.

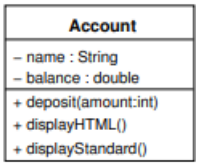
after



Separate domain from presentation

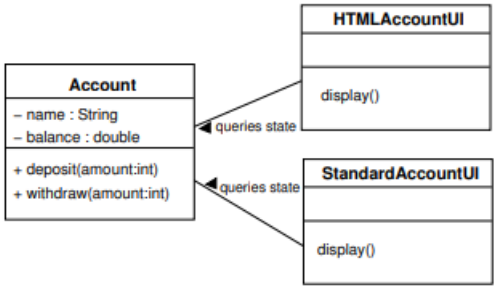
Make the responsibility of each class more focused by separating the presentation operations from the domain classes.

before



It makes sense for an account to have a account holder and name but not display functions , especially since that might be enhanced in the future.

after



We are moving functionalities here and creating new classes, however this method focuses exclusively on extracting the display methods.

Other types of Refactoring

These apply at method level only

- Inline method:
  - Replace a call to a method by the statements of the method
  - *Alt* + *Shift* + *I* in Eclipse
- Why?
  - Where a method body has become condensed down to just one line
  - Inverse of Extract method

- Inline temp
  - Remove a local temporary variable and replace it by the associating expression

```
boolean hasDiscount(Order order) {
double basePrice = order.basePrice();
return basePrice > 1000;
}
```

- Introduce explaining variable
    - use alocal variable to replace a portion of a complex expression

```
function Price(Quantity, ItemPrice: Integer): Extended;
begin
Result := (Quantity * ItemPrice) -
(MaxInt(0, (Quantity - 500)) * ItemPrice * 0.05);
end;
```

```
function Price(Quantity, ItemPrice: Integer): Extended;
var
 Discount: Extended;
begin
 Discount := (MaxInt(0, (Quantity - 500)) * ItemPrice * 0.Result := (Quantity * ItemPrice) - Discount;
end;
```

- Move method
    - Move a method from one class to another
    - Alt + Shift + V in Eclipse
- Move field
    - When a new class is introduced, existing fields may need to be moved to the new class
    - Alt + Shift + V in Eclipse

- Extract class
    - Make a new class from a subset of existing fields and methods
    - Extract Class in Eclipse

- Pull up field
    - When ≥ 2 Subclasses in the same class hierarchy have the same fields, move the common field from the subclasses to the superclass
    - Pull Up in eclipse

- Pull Up Method
- When two methods in different subclasses within the same class hierarchy have essentially the same behaviours, move the method to the superclass.

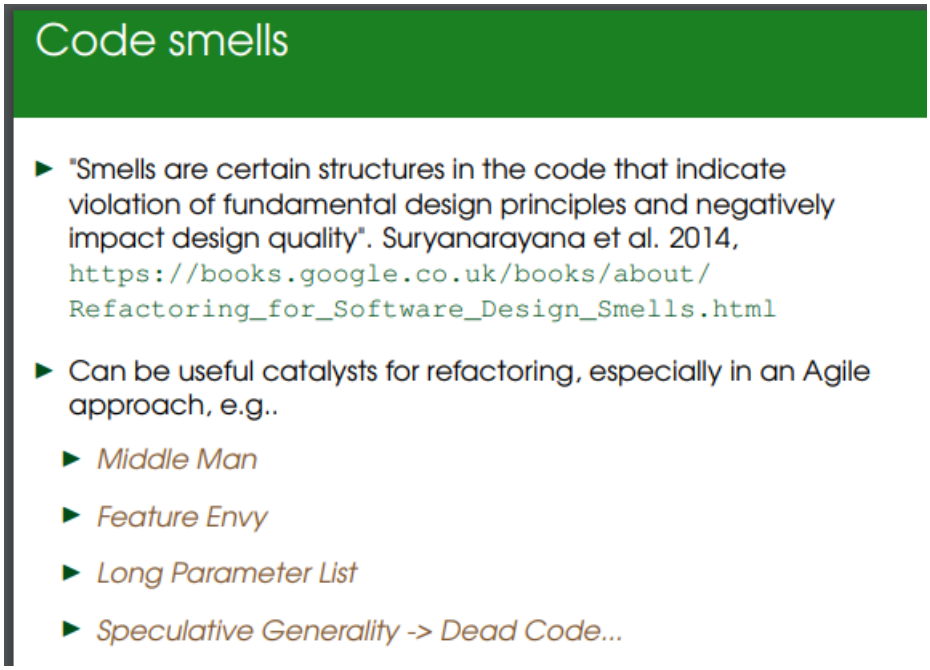
- Push Down Field
    - When a field is used by some subclasses only, but not all subclasses, move the field to those subclasses which use that field.
    - Push Down in eclipse

- Push Down Method
  - When there is a method in a superclass that is relevant to only some its subclasses, move that method to those subclasses.

- Extract Subclasses
  - When there is a feature in a class that is applicable to some instances only, create a subclass for that class and move the feature to it.

- Extract Superclass
  - make a new class from the common field(s) and method(s) in a set of existing classes, with the extracting classes becoming the direct subclasses of the new class
  - Extract Superclass in Eclipse

- Extract Interface
  - make a new interface from an existing class, with the class becoming one which implements the new interface
  - Extract Interface in Eclipse



https://books.google.co.uk/books/about/Refactoring_for_Software_Design_Smells.html