Software development lifecycle

Plan-based methodologies

- **Waterfall**: Each stage is done in order with only one stage being carried out at a time. If you are following the model exactly (which no one does), if you want to backtrack at all you have to restart at the beginning and work your way back down again. Ideally, each stage is only carried out once
  - **Creates lots of documentation**
  - **Does not expect requirements to change** — it is imperative to carry out requirements analysis well
  - **Client does not give feedback during the design and development phases, only requirements analysis, and evaluation**
    - Client doesn't get to feel involved
    - Client can get on with running their business
- **Incremental**: Halfway house between waterfall and agile. Several iterations to produce a series of prototypes, each iteration is a mini waterfall, once all requirements are satisfied that prototype is delivered as the final product
  - The careful analysis of plan-based without the inflexibility to changing requirements of waterfall makes incremental **ideal for risky projects**
    - Risk analysts are highly sought after — adds to the cost of the project
  - Client gives feedback on every prototype
    - Allows requirements to change
- **Reuse-oriented** methodologies add (to the design stage) analysis of which parts of the system can be built around existing components from internal (to the software house) or external (i.e. publicly available) libraries
  - May tweak the requirements (with the sign off of the customer!) to make reuse easier

Agile methodologies

- Agile methodologies allow different stages to be carried out at the same time by different people and have development carried out in short cycles (called sprints) that build on top of each other — sprints are much shorter than iterations
    - At the end of each sprint, the software is given to the user; not temporarily demoed to them as a prototype on a developer's laptop but actually installed on their computers for them to use day-to-day
    - **Allows requirements to evolve**
    - As sprints are so short, developers get feedback very frequently
- "Early, continuous, and frequent delivery of working software"
- Promotes developer wellbeing to try to avoid burnout and thus optimize long term developer productivity
- Concise high quality code
- Commenting is not banned but writing self-documenting code is preferred as out of date documentation lies but code doesn't
    - This can lead to broader laziness in writing documentation
        - Skimping on documentation for the system itself makes training users in the software harder and maintaining the software harder
        - Skimping during requirements analysis can cause legal problems
- **Requires experienced programmers who are capable of making good decisions under time pressure**
- **Requires the client to find the time to provide constant feedback**

Agile methodologies: Extreme Programming (XP)

- A very idealistic form of Agile
- **A user is embedded in the development team** so the developers can get more-or-less real time feedback
    - Having to devote an employe to this can be a major inconvenience for the client
- **Paired programming** is used to ensure extremely high-quality code
    - Significantly adds to the cost of the project for the client
    - One person has the keyboard, the other is checking in real-time everything the person with the keyboard does — switch roles very often and ideally switch pairings regularly
    - Mitigates against the lack of documentation as all developers naturally become familiar with the entire codebase
        - Collective ownership of code
- **Test-driven development**
    - 100% unit test coverage
    - "A bug is a failure of testing not of development"
- "Premature optimization is the root of all evil"
- **Continuous integration: Build often (and so test often)**
- Amount of time developers work is strictly controlled to prevent burnout

When to use which methodology

- **The higher amount of documentation makes waterfall best if we expect changes in developers**
- **If the system is expected to have a short shelf-life, use agile to get it in use ASAP**
- **If the system is expected to have a long lifespan, use waterfall so there is documentation** so the system can be maintained after the original development team have moved on
- **Agile works well for a teams that are small, physically near to each other, communicate easily etc.**
- **Plan-driven works well for teams that are large, already bogged down by bureaucracy (e.g. regulated sectors), spread across significantly different time zones etc.**

Requirements analysis

Steps of requirements engineering

1. **Feasibility study**: Check the project is possible to complete and sensible for you to take on
2. **Requirements elicitation and analysis**: Derive requirements by talking to customers, looking at relevant existing systems and processes, getting customer opinions on mock-ups etc.
3. **Requirements specification**: Turn the natural language requirements into a rigorous specification
4. **Requirements validation**: Check the requirements are achievable
5. Compile all the documentation into a single document and get the customer to sign off on it

Properties of good requirements

- **Prioritised** — MoSCoW (Must (Minimum Usable SubseT), Should, Could, Wish but Won't (bells and whistles that realistically have no chance of being done))
- **Consistent** — requirements don't conflict
- **Modifiable** — requirements can be changed if necessary and a changelog will be maintained if this occurs
- **Traceable** — each requirement is linked to a source (e.g. a particular part of a particular interview with a customer)
- **Correct** — each requirement accurately describes the intention of the source
- **Feasible** — it is possible to implement each requirement within the proposed system and wider deployment environment
- **Necessary** — each requirement should be a feature the customer genuinely wants or is required to interconnect with external systems or is required to conform to standards the customer wishes to or is legally required to adhere to
- **Unambiguous** — there should only be one possible interpretation of each requirement
- **Verifiable** — each requirement should be able to be objectively verified by some measurement

Requirements specification

- Every requirement is written both in layman's terms as a customer-facing (c-facing) requirement and in technical terms (without getting bogged down in implementation details) as a developer-facing (d-facing) requirement
- **Functional requirements describe services that particular components of the system will provide**
  - Results of users' interactions (or lack thereof) with the system
- **Non-functional requirements describe broader qualities of the entire system** e.g. uptime, speed, compliance with standards and legislation
  - Properties of the systems processes
- **To be unambiguous and verifiable, requirements should be very specific**
  - "The method must return a random number between 1 and 100" — restricted to integers? what distribution? is pseudorandomness acceptable?

Requirements validation

- Both the customer and the software house should thoroughly check the requirements before signing off on commencing the project
- **Validity — do the requirements cover all of the customers needs?**
- **Consistency — are the requirements free of conflicts?**
- **Realism — can the requirements be fulfilled with the available resources** (budget, technologies, developers etc.)**?**
- **Verifiability — will it be able to be proved that the system satisfies the requirements?**

Requirements management

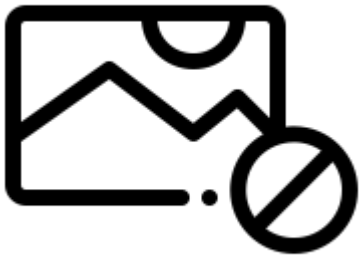
- Requirements management is the process by which new requirements are analysed after validation has been carried out and the project is underway
- **Problem analysis — check the new requirements fully cover the customer's (newly identified) need(s)** — use context to try to cover what they actually need not what they said they needed
    - Check validity and verifiability
- **Change analysis — determine the effect of accepting the new requirements on the rest of the system**
    - Check consistency and realism/feasibility bearing in mind what work has already been committed to by accepting the existing requirements
- **Change implementation** — if everything seems in order, update the requirements document and add implementation of the new requirements to the to do list

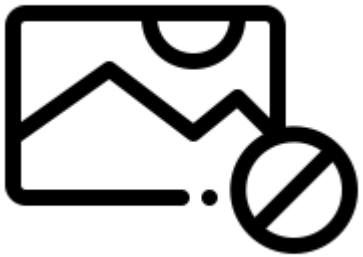
System design (UML)

Class diagrams

- **Class name goes in top sub-box— italicized if an abstract class, preceded by <<interface>> if an interface**
- **Access modifiers: + = public, - = private, # = protected, / = derived, underlined = static**
  - PLus for PubLic, opposite of plus for private, # looks like a gate
- **Attributes go in middle sub-box**
  - **Inherited attributes and methods are omitted** (can be presumed to be implied)
- **Methods go in bottom sub-box**
  - **Getters and setters with the same access as the attribute are implied** so are omitted
- **Inheritance arrows point towards parents**
  - **Solid line and black triangular arrowhead means parent is a concrete class**
  - **Solid line and white triangular arrow head means parent is an abstract class**
  - **Dashed line and white triangular arrowhead means parent is an interface**
- **Arrow from A with solid line and white diamond at B is aggregation** (A is part of B)
  - **Deleting B does not necessarily cascade to A**
- **Arrow from A with solid line and black diamond at B is composition** (A is part of what makes up B)
  - **Deleting B cascades to A**
- **Arrow from A with dotted line and small black triangular arrowhead at B means A depends on B** (A uses B for part of its implementation e.g. B is an instance of Random constructed by A)
- We can add multiplicities to arrows:
  - n (exactly n)
  - a..b (between a and b both inclusive)
  - a.. (a or more)

Class Diagrams



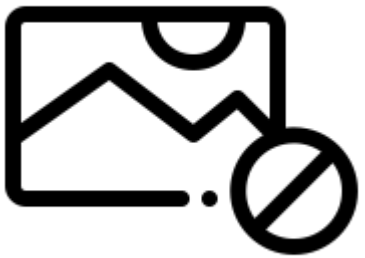


Context diagrams

- A system of ours in the middle inside an extra box representing our system boundary with the external systems it directly uses connected to it passing through the boundary
- Note every system name is prefixed by <<system>>

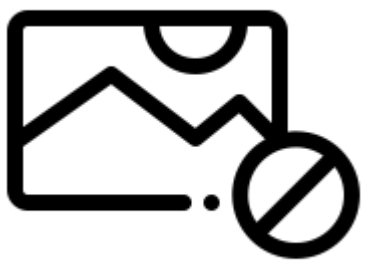
Activity diagrams

- **Flow chart but with the decision at decision diamonds implied by the outcomes**
  - Can take actions in parallel
- **<u>One circle for start, two circles for end</u>**



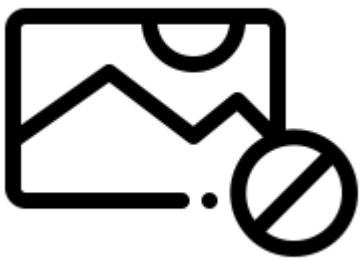
Use case diagrams

- Show how real world actions of users map to events in the system and what abilities different types of user have
- Subclasses must also be connected to their inherited methods (unless overridden in which case they are connected transitively via an extends)



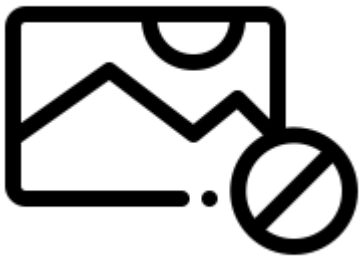
Sequence diagrams

- Show communications between systems on a timeline
    - Somewhat similar to timing diagrams in CS132
- **Horizontal dashed lines show replies to the messages in solid lines**
- **We end the life of an object by ending its dashed line <u>and putting a cross at the bottom</u>**
- **Solid boxes represent time an object is active (doing work or wanting to do work but waiting on some other object)**
- A:B refers to a particular instance of B with name A
- **:B refers to an arbitrary instance of B**
- A refers to an instance with name A of an unknown class



State diagrams

- Finite state machines
- Strongly related to activity diagrams but make the state explicit and the actions implicit whereas activity diagram is vice versa



Software design (idioms)

SOLID principles

- **Single responsibility principle**: Each class should only be responsible for one piece of functionality
  - The class should encapsulate all data and methods needed for that (one) piece of functionality
  - The class should only change if that (one) piece of functionality changes
- **Open/closed principle: Open for extension, closed for modification**
  - **Should be easy to extend but should be extended by subclassing** not changing the class
- **Liskov substitution principle**: Behavioural subtyping
  - **An object that uses the parent class should be able to use the child class instead** in the exact same way
- **Interface segregation principle: Objects should only implement methods when they are going to be useful rather than just because they inherit them** from a high level class whose signature we (poorly) chose to place common (but not necessarily universal) methods in for simplicity
  - **Have small but many interfaces instead of few but large interfaces**
- **Dependency inversion principle: Low level (smaller more specific) classes should use functionality from high level (larger more general) classes not vice versa**
  - There is a temptation to when building complex structures (high level classes) to depend on the details for the building blocks (low level classes) but then we pollute our abstractions with details
  - **When high level classes use functionality from low level classes they should do so via abstracted interfaces** — we can do this because of the Liskov substitution principle

Software patterns

Creational patterns

- **Factory: A factory method creates objects of a particular type** possibly by creating other objects (which may have their own factories in true enterprise code) that are needed to create the object it returns
    - **Factories can act as polymorphic constructors**: Declare factory method in an abstract factory class and implement it calling different constructors in different factory subclasses to return different subtypes of the type in the method signature
- **Builder: Factories are used when we need to create many instances of a (not necessarily complex) class whereas builders are used when we need to create (not necessarily many) instances of a complex (e.g. with the ability to take lots of parameters) class**
    - **Hide the long constructor inside the builder which has a more readable constructor** and hope it is not necessary to look at the builder's implementation very often
    - Could be viewed as a generalisation of factory
- **Prototype: Classes implement a `clone` method which can be used to create many instances of the same class with the same properties**
    - **Can create several different instances with different parameters to use as templates (prototypes) then clone them to create the number of objects you need for each set of initial properties**
    - Having the object be responsible for the production of new instances of itself than an external object (factory/builder) avoids issues with access control of internal state
    - Clone can run into issues with circular references but these should be rare in well designed code

Structural patterns

- **Proxy: A shell client object stands in for a service object**
  - **Virtual proxy: Lazy loading of resource heavy data** (e.g. uses lots of RAM, requires network etc.)
  - **Protection proxy:** Access control, **check authorization of caller** before calling the method in the service
  - **Remote proxy: Provides services actually provided by a different system across the network** and tries to not let the abstraction leak (e.g. cache data and serve from cache iff can't access the service to refresh)
  - **Logging proxy: Keep track of which methods are being called by who**, fulfil them by calling the service
  - **Caching proxy: Maintain a cache (with periodic expiry) and returns from cache if possible**
- **Decorator (wrapper): Add new behaviour to objects without creating a bunch of sub-classes**. **Decorator class is instantiated using the object it is wrapping (composition over inheritance** for the win**)**
  - Inheritance is static (compile-time) but decorators are dynamic (run-time)
  - **Order of wrapping is significant (acts like a stack)**
  - **The decorator has the same interface as what it is wrapping and acts like a proxy** calling the same method in what it is wrapping at some point but also doing its own stuff
- **Adaptor: Converts data to allow an object to be used with a framework it was not originally designed to work with**
  - Create a subclass that makes extensive use of inheritance, overriding the necessary getters leaving everything else alone
- **Flyweight: A single copy of data is shared across objects to free up memory**
  - **Shared data is stored in a (singleton) static data class, other classes store a reference to this shared data instead of a copy of the data itself**
  - Could be viewed as an instance of proxy

Behavioural patterns

- **Iterator: A cursor over an iterable object e.g. `getNext()` and `hasNext()`**
- **Observer: Consumers of a producer object can subscribe to be notified when state changes**
  - **Consumers may register a method with the producer to be called when a change occurs or the producer may provide a method that is polled by consumers**
  - Common in MVC
- **Memento: Objects are saved into a snapshot by themself, snapshots are stored and objects restored from snapshots if necessary by a caretaker**
  - Objects are responsible for saving themselves to avoid access control issues with internal state
- **Strategy: Method is chosen at runtime by choosing which strategy object to call based on the context object**

Architectural patterns

Layered architecture

- Splits the system into layers — **each layer only relies on the layer immediately below and only serves the layer immediately above** — data flow is unidirectional
- **Useful when extending an existing system that you did not design** — only take responsibility for working with whatever output the existing system makes not with its inner workings
- Improves development, debugging, and maintainability through separation of concerns and standardisation
- **Makes it easy to split work across developers as layers are independent** (interfaces should have been agreed in design stage before starting development)
- **Can easily make changes by replacing particular layers**
- **Can easily implement multi-level security** (mandatory access control)
- Can be hard to come up with a suitable separation — any separation is likely to be violated as changes are made over time
- Can decrease performance due to overheads as calls must travel via intermediate layers to reach where they can be acted on
- Each layer must be fairly reliable as if it goes so does everything above it but whats under it will still work

Repository architecture

- **Stores all data in a centralised repository that all other components** (to process the data, give new data etc.) **solely interact with (don't interact with each other directly)** — data flow is bidirectional
- **All other components are completely independent of each other so can be worked on in parallel**
- **Data is consistent as there is a single source of truth responsible for its management and distribution**
- Single point of failure — if the repository breaks, everything breaks
- We may reach a point where the system needs to be scaled to the point of distributing the repository across multiple computers

Pipe and filter architecture

- **Arranges components into a directed acyclic graph of data transformers** to form a data pipeline — data flow is unidirectional
- Easily parallelised code if there are multiple branches
- Easy to evolve
- A very natural shape for lots of tasks
- **Lends itself well to code reuse as many transformers are very common tasks**
- **Only useful when all you are doing is data processing, but that is most backends**

Model-View-Controller (MVC) architecture

- **Model is responsible for data, controller is responsible for interaction, view is responsible for presentation**
- **The view displays the model to the user according to some structure and sends user inputs to the controller**
- **The controller tells the model how the state is changing and tells the view what structure to show**
- **The model manages application state and tells the view what it needs to know about the state to fill in the structure it is showing**
- Data flow is bidirectional
- Very common for web-applications
- Separates the data itself from its presentation — can change how user sees the data (view) without affecting how the data is actually represented internally (model) e.g. **a desktop web page and a mobile app can run off the same model with different views**
- Is the most complex design pattern (that we have seen) — most widely applicable but also the one that removes the least complexity
- Almost every change requires considering every component
    - As every component directly interacts with every other component it can be hard to split work across developers

HCI

Theory of UI design: Nielsen's Usability Principles

- **Visibility of system status — provide feedback to users**
- **Match between system and real-world** e.g. skeuomorphism
- **User control and freedom — provide escape routes** e.g. cancel, undo
- **Consistency and standards**
- **Easy to recognise and recover from errors** — explain errors in plain english and provide guidance on what the user should do about it
- **Prevent errors e.g. confirmation dialogs** for hard to reverse actions
- **Recognition over recall — have everything as easily accessible as possible instead of relying on the user's memory of how to reaccess something**
- **Flexibility and efficiency of use — power-user features (accelerators) that don't get in the way of novice users** e.g. keyboard shortcuts, advanced options tabs
- Aesthetic and minimalist design — **everything should serve a purpose**
- **Help and documentation**

Dependability

Terminology

- **A failure = an instance of the system deviating from its specification**, the result of an error being allowed to propagate
    - **We may be able to handle some errors to prevent them becoming failures** e.g. catching exceptions
- **An error = the manifestation of a fault**
- **A fault = the cause of an error**
- **Reliability = the likelihood that a service works without failure over a period of time** — often quantified as mean time to failure
- **Availability = the likelihood that a service is ready for use when invoked**
- **Confidentiality = the extent to which a system prevents the unauthorised disclosure of information**
- **Integrity = the extent to which a system prevents improper alterations to the information it holds**
- **Safety = the extent to which a system can operating without damaging or endangering its environment** (note this means the wider world, not the technical environment it is running inside)
- **Maintainability = the probability that a system can be repaired from failure in a given period of time**, a probability distribution $P(X \leq t)$ where X is the time taken to repair from failure

Attributes of dependable systems

- **Documented** — create documentation
- **Standardised** — distinct systems should obey a common standard
- **Auditable** — traceability to enable accountability
- **Robust** — system should be able to recover from failures of individual components
- **Diverse**

Creating dependable systems

- Overarching idea: We cannot find and fix all faults but we can try to stop them becoming system-wide failures
- **Graceful degradation enables a system to continue to provide some services (correctly), and crucially to provide no services incorrectly ("fail safely"), if a component fails**
- **Redundancy introduces spare components** (these may be software or hardware) **that can be called upon if a primary component fails**
    - Diversity has the spare components as different as possible to the primaries to try to keep the probabilities of failure independent
- **Protection systems are specialised systems that monitor** (and run separately to) **a control system and its local environment and take action (e.g. safe shutdown) if they detect a fault** — think automated nuclear SCRAM
- **A self-monitoring architecture carries out computations over two channels** with distinct hardware and software **and verifies that the outputs match** declaring a failure otherwise
    - Has fault detection but not fault recovery
- **n-version programming** produces multiple implementations of the same specification by different teams (who can't communicate with each other and ideally will be using different tech stacks) and executes each on a different computer, **output of system is decided by voting by the components** (e.g. majority wins (use odd n to avoid ties))
    - Has fault detection and fault recovery

Testing

Types of test

- **Black box testing (functional testing) = test cases that were created by looking at the system specification** and design documentation in order to create a set of tests that **cover all possible contexts to the code being tested**
  - You do not look at the source code
- **White box testing (structural testing) = test cases that were created by looking at the source code** in order to create a set of tests that **cover every possible path through the code being tested**
- **Unit testing tests a single method**
- **Integration testing tests an entire component**
  - Errors can occur here without occurring in unit tests due to:
    - Interface misuse — API is not as expected
    - Interface misunderstanding — behaviour is not as expected
    - Timing errors
- **System testing tests the whole system**
- **Regression testing verifies no previous bugs have been reintroduced**
- **Alpha testing = select users work with developers during initial development**
- **Beta testing = larger group of users experiment with complete or nearly complete software**
  - Can build up hype around the new software
- **Acceptance testing (custom systems) = customer decides whether system is ready to be deployed**
  - Determines when you get the rest of your money