UNIVERSITY OF CANTERBURY

SENG301

ADVANCED SOFTWARE ENGINEERING STUDY GUIDE

HARRISON COOK
2019

TABLE OF CONTENTS

Part 1: Fabian

1 Software Engineering Methods

1.1 Waterfall

- Sometimes referred to "build it twice" since it can easily lead to misidentification of what the client wants and hence require redesigning the whole product
- Large amount of planning and design – Massive upfront analysis
- Cycle:
    - Requirements Definition
    - System and Software Design
    - Implementation and Unit Testing
    - Integration and System Testing
    - Operation and Maintenance

1.2 Spiral

- No massive upfront analysis
- Cycle:
    - Determine objectives (1)
        - Before each iteration, define the list of objectives
            - i.e. what parts or functions will be developed?
    - Identify and resolve risks (2)
        - Evaluate alternatives and mitigate the risks linked to the selected objectives
    - Development and test (3)
        - Build a solution according to the identified risks
            - i.e. design, write and test code, or, build a prototype
    - Plan next iteration (4)
        - Review and evaluate work done during this iteration release version or adapt requirements plan for next iteration

1.3 Agile

- Fully incremental
- Difference between Agile and Spiral is that there is no formal risk assessment in Agile, if a risk comes up it is addressed
- Cycle:
    - Continuous planning
        - Regular project status updates
        - Monitoring of performance
        - Continuous evaluation of pipeline of tasks
    - Collaborative development and test
        - Team assignments and organisation evolve from task to task
        - System is continuously tested
    - Continuous release
        - Deployment on production environment of new features as they are ready and tested
    - Continuous input and feedback
        - Users are encouraged to provide feedback on system and suggest new features or improvements

1.4 Incremental vs Iterative:

1.4.1 Incremental:

- The requirements are divided into different builds
- Needs a clear and complete definition of the whole system from the start
- Customers can respond to each build (but a build may not represent the whole system)
- Incremental fundamentally means **add onto** and helps you to **improve your process**

1.4.2 Iterative:

- Does not start with a full specification
- Building and improving product step by step
- We can get reliable user feedback
- Good for big projects
- Only major requirements can be defined, details may evolve over time
- Iterative fundamentally means **redo** and helps you to **improve the product**

1.5 Process Activities

1.5.1 Typical activities

- Requirement analysis
- Software design
- Implementation
- Validation and testing
- Software evolution

1.5.2 Requirement analysis

- Elicitation
    - Discussion with stakeholders or users
    - Observation of analogous systems
    - Early prototyping or sketching
- Specification
    - Translate identified system's objectives
    - Formalise into requirements
    - Functional vs non-functional requirements
- Validation
    - Check for realism, consistency and completeness
    - May (will) trigger correct specification
    - May call for further elicitation

1.5.3 Software design

- Architectural design
    - Describe the big picture
    - Follow or adapt architectural style
- Data design and specification
    - Identify domain concepts
    - Specify exchanged data
- Interface design
    - Unambiguous definition of component usage rules
    - Expose implemented features and hide implementation details
- Component design or selection
    - Encapsulate correlated functionalities (or features)
    - Identify off-the-shelf components for reuse

1.5.4 Implementation

- From abstract components to concrete pieces of code
  - Further investigation may be needed before writing code
  - Critical systems may be formally (i.e. mathematically) specified
- Detailed implementation must follow general design
  - Forbid breaches in interfaces' contracts
  - Wrong architectural choices must be raised and documented
- Unit testing is part of the implementation task
  - An erroneous implementation may not be passed to validation phase
  - Any piece of code should be at least specified in terms on in/out/effects

1.5.5 Validation and testing

- Component testing
  - Integrate all parts of a component and test it
  - Verify (or validate) it behaves as expected
- System testing
  - Integrate all components and test the whole thing
  - Check if components interact as expected
  - Early validation of non-functional requirements
- User testing
  - Use meaningful data instead of simulated environment
  - May highlight misunderstandings or omissions
  - Final validation of non-function requirements (mainly ergonomy and performance)

1.5.6 Software evolution

- Do not talk about maintenance as a separate and non-creative activity
- A system is never completed until it dies
  - Despite proper testing, misbehaviours will occur
  - New features will be requested by users or PO
  - Security issues will arise
  - Technological changes will occur
- System changes need
  - A proper understanding of the current system
  - An adequate system transformation plan
  - Clear testing procedure of the whole system

2 AGILE DEVELOPMENT

- Initiated by 17 researchers and IT professionals in 2001

2.1 AGILE CORE VALUES

- Individuals & interactions over processes & tools
- Working software over comprehensive documentation
- Customer Collaboration over contract negotiation
    - Customer is constantly involved
- Responding to change over following a plan

2.2 AGILE PRINCIPLES

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale
- Businesspeople and developers must work together daily throughout the project
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation
- Working software is the primary measure of progress – you must always be delivering something
- Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely
- Continuous attention to technical excellence and good design enhances agility
- Simplicity –the art of maximising the amount of work not done– is essential
- The best architectures, requirements, and designs emerge from self-organising teams
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly

2.3 (MAIN) AGILE VARIANTS

2.3.1 Scrum

- Used when you have enough information to plan around from the beginning
- Team commitment and empowerment
- Objectives in sprint deliveries
- Improved code quality and documentation

2.3.2 Lean

- Also known as optimisation of processes (more so than agile)
- Limit waste and decide as late as possible
    - This means you can gather more information during the process
    - Allows you to make a more informed decision
- No over-production neither over-processing
- Originally based on Toyota's manufacturing in the 1940's and the 7 types of waste (TIMWOOD):
    - Transport – moving the product adds no value
    - Inventory – storing products costs money
    - Motion – unnecessary movements cause stress and waste energy
    - Waiting – by definition is a loss of performance
    - Over processing – putting too much effort than needed
    - Over production – leads to more inventory waste
    - Defects – implies unnecessary maintenance costs

- **Seven godly principles of Lean:**
  - Optimise the whole – avoid thinking on partial optimisations
  - Focus on customer – understand and answer to customer needs
  - Energise workers – unhappy or unrewarded teammates aren't performing
  - Eliminate waste – avoid over-engineering
  - Lean first – welcome changing requirements
  - Deliver fast – time-to-market is critical
  - Keep getting better – focus on people delivering results instead of the results

2.3.3 Kanban

- Lean-oriented (based on Lean) and reuse part of Scrum
- **No** up-front planning (with respect to Scrum)
- Task-based, do a task and then deliver
- Could be used when requirements aren't known (yet) and plans are made as you go
- Continuous changes in one global board
- Aims to reduce waste of any sort
- Uses the seven godly principles from Lean (see above)
- **Kanban's life cycle (focuses on tasks):**
  - Task Planning
  - Development & Test
  - Resolution Release
  - User Feedback
- **Kanban backlog:**
  - To do
  - Review
  - Execute
  - Verify
  - Deploy
  - Done
- **Kanban metrics:**
  - WIP
    - Work in progress does not add value to the customer
    - Limited to avoid too much things in the pipeline
    - Often certain stages of the backlog will have WIP limits (assumedly that is the max number of tasks in progress). To do –> verify
  - Queue – measures efficiency by comparing tasks in WIP and waiting in queues
  - Blocker – external dependency making a task to be blocked in a queue
  - Throughput – average number of units processed per time unit
    - E.g. Tasks per day, story points per week
  - Lead time – total time between when a customer demand arrives and is deployed
  - Cycle time
    - Processing time for a task in (a set of) states, removing queue time
    - Often use variation of Little's law on queue: CycleTime = WIP/Throughput
  - Kanban vs Scrum
  -

	Scrum	Kanban
roles	product owner, scrum master, team	none specified
ceremonies	planning, review, retrospective, daily	daily
delivery	sprint-based	continuous
modification	(almost) none allowed in sprint	any time
productivity	sprint velocity	cycle time
delegation flow	sprint backlog	continuous
work limit	sprint-based	work-in-progress
teams	cross functional	unspecified
environment	relatively stable	evolving priorities

2.4 SCRUM

2.4.1 Scrum values:

- Openness
  - You must be open with your team
- Courage
- Focus
  - You must be focused on your task
  - Results must be functional
- Commitment
- Respect
  - Your client
  - Your team

2.4.2 Scrum roles:

- Product Owner (PO)
  - AKA the customer voice
  - Translates user demands into stories
  - Maintains and prioritises the backlog
  - Defines content of releases and timing
- Scrum Master (SM)
  - AKA process leader and facilitator
  - Acts as a coach – does not command
  - Facilitates communication inside and outside the team
  - Represents management, but protects team members
  - Sometimes who holds this role changes between sprints
- Team – the developers
  - Everyone is a developer (often including the Scrum Master)
  - No hierarchy
  - Self-organising and cross-functional
  - Collective responsibility for achievements

2.4.3 Backlog (aka Agile Board)

- Product Backlog
  - Contains product backlog items (PBIs)
  - Everything that still must be done
  - Prioritised and maintained (by the PO) constantly
  - Some estimations may be missing
- Sprint Backlog
- Items that will be handled this iteration
- Should be fixed for the sprint
- Only high-priority items may modify current sprint backlog
- All estimations must be set
- **Backlog Items**
  - Epic
    - A large piece of work that spans over multiple sprints
    - Abstractly defined high-level requirement
    - May look like a story, but with a fuzzier definition
    - Not really an item by itself
    - Often accidentally created when trying to create a story
    - Needs to be refined into stories

- Story
  - A small and well-defined piece of work
  - Concretely and extensively defined requirement
  - Must be handled in one single sprint
    - The user's point of view regarding a system's functionality
      - One functionality and one goal
      - Written in natural language
      - One set of acceptance criteria
        - These are used to ensure the story is implemented correctly
        - This is test driven development
      - Usually follows the template: As a <role>, I <action> so that <value>
        - E.g. As a user, I want to be able to log in so I can access blah
    - Stories are still negotiable
  - Theme
    - An abstract collection of stories touching the same topic
    - As epics, not a backlog item by itself
  - Feature
    - "System" story
    - Express a needed characteristic of the system
    - Mainly non-functional requirements
    - Not necessarily linked to user stories/demands
  - Improvement
    - (User) feedback on existing product
    - Not a new story or feature by itself
  - Bug/defect
    - Anything that does not work as expected
  - Task
    - Anything concrete and time-boxed that must be done
    - May be split into subtasks
    - May only be assigned to one or one pair of programmers

2.4.4 Story Breakdown

- **Bill Wake's INVEST** – Used to ensure a story is well written
  - Independent – stories should be implementable in any order
    - This is not always the case, and is the only one of these rules that can be broken
  - Negotiable – invitation to a conversion
    - Always in contact with the PO
    - Don't have to interpret anything yourself
    - Can always add more ACs to make things clearer
  - Valuable – every story serves a purpose
    - If it's not something the user wants, don't add it (to the backlog)
    - The must be sure they will want it at some point
  - Estimable – development time
    - If it's too long, it's an epic
  - Small – the holy KISS (Keep It Simple, Stupid)
    - Don't over-engineer, over-develop **(does this mean we should over-develop, or don't over-engineer or over-develop?)**
    - Don't spend time optimising code, if it needs optimisation, we can do it when it comes up later
  - Testable – think early about acceptance criteria
- **Promote stories into epics when there are:**
  - **Vague or undefined terms**
    - E.g. "As a customer, I can add multiple books to my shopping cart which is shown"
      - How many books? Shown where and in what detail

- **Conjunctions or multiple use cases**
  - "I want this **and** this"
  - E.g. "As a customer, I can search for a book and add it to my cart"
    - Problem: multiple use cases (searching and adding)
    - What are the search criteria?
    - How can I search for the book (is there a search bar everywhere?)?
    - Should there be advanced search facilities or filters?
- **Hidden or unexplored business rules**
  - E.g. "As a customer, I can add a book to my cart"
    - What does a customer mean? Any time of customer? Do I have to be logged in?
    - How long is the cart's content retained?
    - What if the book is sold out or otherwise unavailable?
- **Multiple display possibilities**
  - E.g. As a user, I can view the content of my shopping cart"
    - What types of devices are considered for the interface?
    - Do we have to deal with responsiveness?
    - Do we have to build a dedicated mobile app?
- **Possible exception flows**
  - E.g.: "As a user, I can log into my account so that I can view my cart"
    - What details are needed to log in?
    - What happens after multiple failed log in attempts? How many?
    - Is there a 'lost password' procedure?
    - Can users be blocked/banned/deleted by administrators?
- **Unidentified data types**
  - E.g. "As a customer, I can search for books so that I can find the publication I look for"
    - What are the search criteria?
    - How are the result displayed?
    - How are books defined in the system?
- **Unidentified operations (Hidden operations)**
  - E.g. "As a customer, I can review the items in my shopping cart and pay for them"
    - What does 'review' mean? Can I delete items? Update items? Increase quantities?
    - What is the full workflow (steps) between the review and the payment
    - What are the involved third parties for the payment (if any)?
    - Can I cancel the process? When?
    - How can I pay?

2.4.5 Scrum lifecycle

- Focuses on sprints
- Sprint cycle:
  - Sprint retrospective and planning
  - Collaborative development and test
  - Sprint release
  - Sprint review meetings
- **Scrum ceremonies**:
  - Daily scrum meeting (stand up)
  - Sprint planning meeting
  - Sprint review meeting
  - Sprint retrospective meeting
- **Initial start-up:**
  - Before starting:
    - Start from a rough vision of the product
    - Refine its objectives
    - Discuss with the stakeholders (client(s), PO, etc)
    - Create an initial backlog large enough for (at least) one sprint
    - Agree on working mode and standards
    - Agree on coding standards, communication means and tools

- **Fresh restart at every sprint**
    - Every sprint must be planned
    - This requires some cleaning by the PO beforehand
        - Cleaning – going through the backlog looking for things to delete (features no longer wanted/required). Grouping related items into themes. Ensure the priorities of backlog items are still correct. Top items should be (made) ready to work on.
    - Other members may assist cleaning/refining when needed
    - Set up a global goal for the sprint (e.g. "add user profiling features")
    - Priorities must be clearly stated (often with stakeholders)
    - Priorities must be discussed on familiar ground
        - Every task under consideration must be estimated
        - Every implementation task must be sufficiently described
            - i.e. Having wireframes, mock-ups, and updated acceptance criteria, if applicable
    - Sprint planning is no place for requirement engineering

2.4.6 Sprint Planning

- **Parts**
    - 1: What are we going to do?
        - The PO presents the highest priority backlog items
        - First draft based on previous velocity
    - 2: Who's going to do what?
        - Selected backlog items are broken down into tasks by the team
        - Estimations of tasks are set collaboratively (reaching a consensus)
    - 3: How are we going to organise our time as developers
        - Start from higher priority backlog items and related broken-down tasks
        - Iterate until the planning is (almost full)
- **Estimates**
    - Don't fall into the ***student's syndrome***
        - Natural tendencies of under-estimating the job to be done
        - Think about an estimate and double it for the first few sprints
        - Don't forget that standups, reviews, emails, etc also task time
    - Be aware of cascading delays
        - Your delay is passed to any dependent task (blocking)
        - When they occur, re-evaluate the plan
    - Decide between story points (relative) vs hours (absolute)
- **Planning poker**
    - Use a Fibonacci-like progression to reflect progressing uncertainty in estimates
        - Called the rounded Fibonacci
        - Usual values are 1/2, 1, 2, 3, 5, 8, 13, 20, 40, 100, Infinity
    - Occurs after initial backlog or when new backlog items or unexpected impediments arise

2.4.7 Daily stand up

- Short synchronisation point every day
- Used by both Scrum and Kanban
- Must be short and straight to the point
- **Three simple questions**
    - What did you achieve yesterday (or since last stand up)?
    - What will you do today (or until next stand up)?
    - Are you facing any issues or impediments?
- Should last 5 minutes per member at most (this is from notes, in class they've said 15 minutes for whole stand up)
- The daily stand up is the place to identify possible delays and take actions

2.4.8 Review (the task or sprint)

- At the end of the task (if using Kanban) or the sprint (if using Scrum)
- Used to demonstrate the outcomes to the product owner and stakeholders
    - Get and keep track of feedback received from them
- Create a clear flowing scenario by combining the user stories
- Use some realistic test data, don't flood with data
- Ensure you have prepared the necessary logistics
- The whole team is not expected to actively present
- Start by setting the scene, explaining the impediments and how you tackled them
- Possibly include a coming soon... Section
- Stakeholders are often far away from the team, so you have to convince them

2.4.9 Retrospective

- Remember the core Scrum values
- Location is important
    - Large and impersonal; rooms enforce feeling of inquisition
    - Prefer coffee shop, break room, or somewhere casual
- Come prepared
    - Come with you issues and suggestions, even communicating these to your team in advance
- **Discuss issues about**
    - Communication
        - Fix communication problems with the PO, other stakeholders, or within the team
    - Processes
        - Software
        - Platform
        - Standards
        - Versioning
    - Scope
        - Ensure consistent formatting of stories
        - Ensure the scope is accurate
    - Quality
        - Quality is everyone's business (especially code quality)
    - Environment
        - Be sure to maintain a collaborative and pleasant environment where everyone feels comfortable to share their ideas
    - Skills
        - Is addition training or external expertise needed?
- Ask yourself: What are we doing well and what can be improve?
- Maintain a list of issues (action points)
    - You do not need to try solving them all at once
    - The action points must be evaluated at the next retrospective

3 Requirement Engineering/Analysis

3.1 Understand the World You're Entering

- Every business is different
  - They have their own practise
  - They have their own vocabulary
  - They follow their own rules
- Stakeholders (interested parties)
  - Can have different practices
  - Do have divergent objects
  - Have hidden responsibilities and power
- As a requirement engineer, you have to decipher, analyse and identify

3.2 Analyse The People

- Thinking in terms of generic users won't work
  - It is complicated to extract clear requirements
  - It is hard to identify system's added value
  - Very few systems target people with homogeneous characteristics
- Thinking about users help us to design appropriate user interfaces

3.2.1 Getting into the users' mind

- Alan Cooper's Personas
  - User archetypes synthesised from common characteristics
    - Characteristics meaning skills, environment and goals regarding the system
  - Well identified behaviour patterns and goals
  - Fictional characters with fictious details
  - Represents behaviour, there is no real job description associated
  - Possibility to have more than one persona for one job
    - But beware to keep the number of personas as limited as possible
  - Bringing personas to life does not mean role playing
- Alternatively, focus on "classes" of users
  - Like personas, profiles are defined on user interviews, not your imagination =
  - Physical characteristics (i.e. possible disabilities)
  - Cognitive characteristics (i.e. disabilities and motivation)
  - Relevant social, ethnical or religious specificities
  - Educational background and competencies regarding the job
  - Task experience
- Profiling creates abstract users
  - It allows us to highlight the discrepancies between users
  - Helps us in focusing user interfaces

3.3 Analyse The Context

3.3.1 The context does matter

- Each organisation is particular
  - They have their own terminology
  - They manipulate their own concepts
  - They always have dissimilarities in similarities

- From interview and discussion, requirement engineers must
    - Understand the core business concepts
    - Identify the relations between these concepts
    - Specify their concrete semantics
- Contextual information must be validated early with the stakeholders
    - Dictionary of concepts and their attributes
    - Entity-relationship diagrams

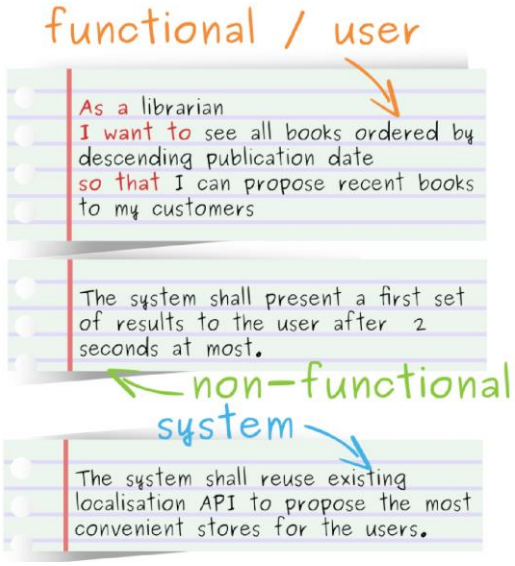
3.3.2 Understand the organisation's style

- Even organisations within an industry
    - Use different business processes
    - Rely on (undocumented) decision workflows
    - Have peculiar habits and customs
    - Have different cultures
- When entering a new organisation, interviews are also meant to
    - Identify the key people
    - Understand the underground power relationships
    - Adjust you to communication and behaviour to fit the organisation's style
- Do not underestimate the critical importance of people

3.4 INTERVIEWS

- Requirement elicitation

- Requirements can be categorised into
  - **Functional**
    - Any intended function offered by the system
  - **Non-functional**
    - Any additional property of the system
- And orthogonally
  - **User**
    - Mainly user stories depicting usages of the system to-be
  - **System**
    - Detailed descriptions of services or technical specifications



3.4.1 The fine art of conducting interviews

- Never start with "tell me what you need"
- Three ways of interviewing stakeholders
  - Conducted – using a predefined set of questions
  - Open – brainstorming-like exploration of their needs
  - Mixed – a bit of both the other approaches
- First interviews are critical
  - Don't get abused by domain jargon
  - Be open-minded
  - Rapidly sketched prototypes are often useful
- Remember that interviewees may not master the day-to-day reality
- **Translate interviews into requirements**
  - Natural language – ordered free-form sentences with no particular characteristics
  - Structured natural language – following a template, like *as a <actor>, I want to <action>*
  - Graphical notations – (annotated) diagrams, like UML Use cases or User Requirement Notation
  - Formal specifications – mathematical descriptions based on, e.g. finite state machines
- **Break down stakeholder's demands**
  - "As a new user, I would like to sign up to XYZ website so that I can start using its awesome services"
  - Attempt 1
    - Task 1 – design function tests
    - Task 2 – generate test data
    - Task 3 – develop data base layer
    - Task 4 – develop business logic layer
    - Task 5 – develop user interface layer
    - Problems with this:
      - Seems like waterfall
      - Even if test-driven designed, tasks are still sequential
      - Are we sure different programmers can work concurrently?
      - Reviews by product owner are quite impossible
    - Good things about this:
      - Logical, straightforward decomposition
      - Systematic and structure decomposition
      - Clear separation of layers
      - Seems like all the needed work is there

- Attempt 2
    - Task 1 – develop username/password functionality (including tests)
    - Task 2 – develop email authentication functionality (including tests)
    - Task 3 – develop landing page functionality (including tests)
    - Benefits of this approach
        - Tasks will not overlap multiple sprints
        - Tasks are testable by themselves
        - Tasks can be verified independently
        - Review by PO is far easier
    - Why not make each task a story? – A story must have value (INVEST) for the PO
    - What about everyone touching all layers at the same time?
        - Stories are only part of the truth
        - Every system needs an architecture (model)
        - Main technological choices must be discussed prior to their use
        - Detailed design is left in the programmer's head

- **Validate requirements**
    - Ensure faithful and accurate translations (on top of INVEST)
        - Valid – does the requirement reflect the user's need?
        - Consistent – are there any conflicting requirements?
        - Complete – requirement definition should be self-contained
        - Realistic – do the current technologies and way of thinking allow such (a) feature(s)
        - Verifiable – aka INVEST's testable
    - Common validation practices
        - Review – sit with the stakeholders and systematically check the requirements
        - Prototype – either build a Proof of Concept or create some sketches
        - Test cases – list sample/fraudulent usages

4 PLANNING ADMIN

4.1 CYCLE:

- Pick a task – Choose an issue to handle
- Execution – Work on the selected issue, refine the story, write code
- Evaluation – Test and integrate or review task outcome
- Release – Integrate code or add document to repository

4.2 BREAKING DOWN TASKS

- INVEST in your stories (INVEST is above)
- **Be SMART with your tasks**
    - Specific – everyone must understand its boundaries
    - Measurable – agree on what 'done' means (Definitions of Done)
    - Achievable – you can achieve it without any help
    - Relevant – tasks do relate to stories, features, improvements or defects
    - Time-boxed – expected estimate, or further break down is needed

4.3 DEFINITIONS OF DONE (AND READY)

- All members must agree on a definition of done and ready
    - Make clear the expected level of quality
    - Minimise arguments between members
- **Different definitions for different elements may exist**
    - Story-level (INVEST)
    - Task-level
    - Code-level (regarding pair programming or code review & unit tests)
    - Documentation-level
    - Release-level
- You need to frequently review these definitions
- Proposed definition of **ready** for **stories**
    - Estimation – story must be estimated
    - Criteria – acceptance criteria must be clearly defined
    - Ordered – story had been ordered into the product backlog
    - Document – additional resources have been attached if necessary (i.e. wireframes)
    - Sprint – all tasks from the story will stay within one sprint
- Proposed definition of **done** for **stories**
    - Acceptance – all tasks have passed the tests
    - Regression – no added feature breaks the current system
    - Deploy – any relevant build and deployment script has been updated
    - Review – the product owner has reviewed and accepted the new functionality
    - Document – end-user documentation has been written or updated
    - Sprint – all stakeholders have reviewed the functionality at a sprint review
- Proposed definition of **done** for **coding tasks**
    - Tests – no code should be marked as done without attached tests (normally unit tests)
    - Review – code should be written in a pair or formally reviewed by a pair
    - Pushed – code should be pushed onto the versioning repository
    - Built – pushed code may not break current build
    - Board – task board should be updated with *RemainingTime* = 0 (i.e. Agilefant time remaining)
    - Document – code must be documented and commented appropriately
- Proposed definition of **done** for **releases**
    - Review – product owner has reviewed the whole content of the release
    - Deployment – all code has been deployed successfully
    - Smoke test – appropriate subset of functionalities have been tested on production data
    - Training – target users (and marketing service) have been trained on new functionalities

5 AGILE TEAM MANAGEMENT

5.1 ACT AS A TEAM

5.1.1 Team's organisation

- Remember the Scrum values
- **Act as a team**
  - Everyone is recognised as an individual
  - But collaboration inside the team is the key
- No hierarchy, everyone is a developer (from technical designers to testers, including GUI designers)
- Get specialised in one subject, but acquire many competencies
  - Everyone has their own preferences in terms of tasks
  - But Scrum relies on continuous and appropriate knowledge transfer

5.1.2 Anyone may take over your job

- **Scrum teams are cross-functional**
  - Developers are encouraged to go outside his/her comfort zone
  - Developers may work on different parts of the system
  - Developers may work on different deliverables (e.g. wireframes, model, code)
  - There is no place for hero culture or knowledge silo in Scrum
- Spread out your knowledge
- Get specialised to develop your set of core technical competencies
- Widen your complementary abilities
  - Easiest way to start is to review someone else's code
- No team is immune to unfortunate events or unexpected member desertions

5.2 DAY-TO-DAY MANAGEMENT

5.2.1 Tools

- **Old school**
  - Large whiteboard, pane or wall
  - Painter's tape for the columns (and rows)
  - Whiteboard marker
  - Sticky notes (number of colours defined by backlog item types)
- **Digital**
  - Suitable and reliable software (Agilefant)
  - Accessible from many devices and from anywhere
  - Technological gap must be considered
- Both have pros and cons, need to find your own way

5.2.2 Setting up the environment

- Make your own working environment
- Make use of the many Software Engineering tools around
  - Git ecosystem (including merge checklists, wiki, issue trackers, etc)
  - Code metrics/quality monitoring (e.g. Sonarlint/Sonarqube)
  - Tools are part of the project standards to decide collaboratively
- Working remotely is possible, but remember about daily stand ups
- **Things to add to your team room**
  - Sprint and retrospection main goals (action points)
  - Agreed definitions of ready and done
  - Stakeholder's quotes and comments about the product (or product to-be)
  - Pin finished tasks on a wall of fame

5.2.3 Setting up your agenda

- **When coming to a sprint planning**
  - Everyone must know their availabilities
  - Don't forget you may have other projects (even in industry)
  - An 8-hour week becomes roughly becomes 6 hours on a project
    - Interruptions occur (emails, phone, issues, teammates, etc)
    - Team communication and synchronisation is time consuming but crucial
  - Don't rely on memory alone
  - Make use of an agenda
    - Put all your deadlines in it as they come
    - Never come to a planning without the agenda
  - Make your non-negotiable(s) clear at the very beginning of the project

5.2.4 Welcome changes

- A sprint plan should be protected and safe
  - It defines the agreed deliverables for the PO, i.e. Product values
  - However, agile requires us to 'welcome changing requirements'
- Stakeholders may look over your shoulder at any time and come with
  - Cosmetic changes – small superficial makeup that can be taken into account
  - Function changes – new stories that should be estimated and planned
- Your job is to know the threshold between a sprint plan and stakeholders making changes and ensure you communicate with stakeholders
  - Decide whether the makeup is worth it during the current sprint or not
  - Decide whether the story can be added into the plan or not (the buffer)

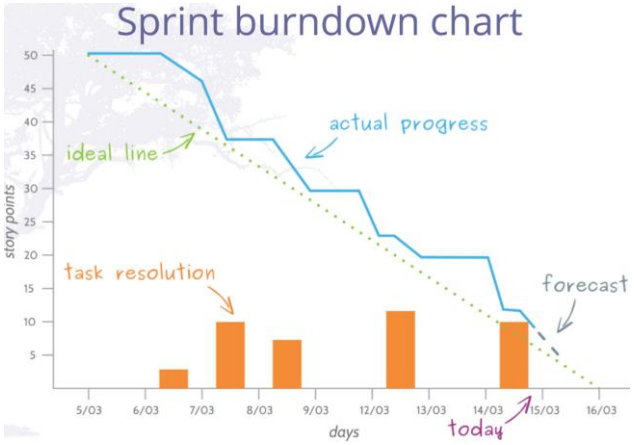
5.2.5 Dealing with the unplanned

- You need to distinguish between development issues and bugs
  - An issue is a problem identified (at the latest) at a review
  - A bug is a problem discovered (at the earliest) during a regression test
    - You can plan for fixing bugs
- **Dealing with impediments (and how do you deal with them?)**
  - Large meetings – stick to the essential stakeholders and keep them time-boxed
  - Illness – unavoidable, but refrain from 'toughing it out'
  - Broken builds – implicit top priority of the whole team to fix
  - Tools – always frustrating, consider alternatives or a work-around
  - Broad backlog – pay special attention to review the stories and broken-down tasks
  - Unreliable PO – unfortunately, you may have no say in this
  - Team problems – Resolve them or seek assistance
  - External incentives – remember Scrum is about the team, so praise the common goal
  - Don't forget to record the impediments and how you resolved (or attempted to resolve) them, to be used in sprint reviews and retrospections

5.3 MONITORING PROGRESSION (BURNDOWN CHARTS)

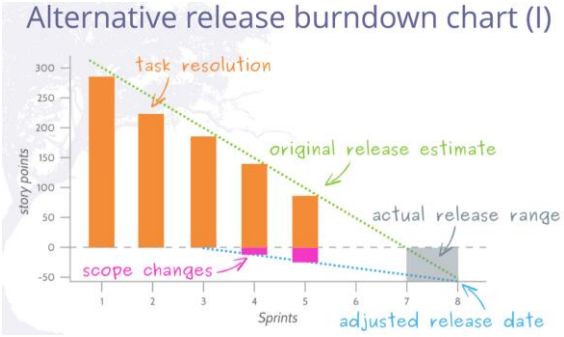
5.3.1 Sprint burndown chart

- Compares actual progress to ideal progress. Used to see if the sprint is on track
- Actual progress line:
  - Straight up – scope change, adding a new story
  - Diagonally up – adding new tasks to stories
  - Straight down – Deferring a story
  - Diagonally down – Normal development
  - Gap at end – Not completing all tasks/stories



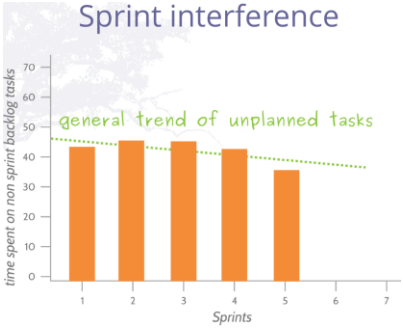
5.3.2 Alternate release burndown chart

- An alternative to the typical burndown chart, this allows you to estimate a release date based on current trends and scope changes
- Scope change – Bar graph below 0 story points, indicates new story or task(s) from the PO
- Task resolution – Bar graph above 0, indicates actual story points after sprint completions
- Allows you to estimate what sprint the product will be completed in

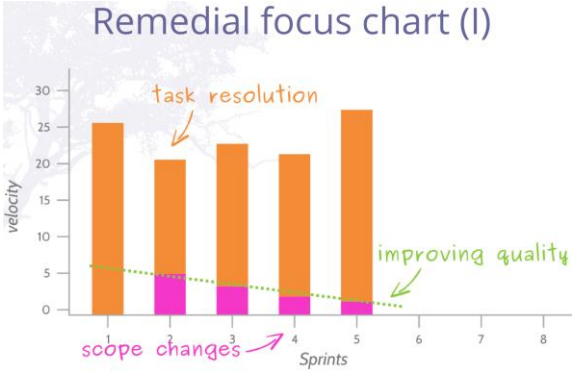


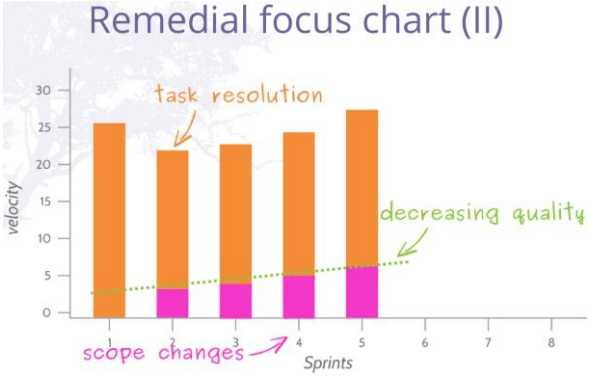
5.3.3 Sprint interference

- This chart assists teams with sprint capacity planning. It indicates the amount of time spent on non-sprint tasks (communication, bug fixing) (unplanned tasks)
- The trend should be decreasing or stabilised, it must not go up



5.3.4 Remedial focus chart





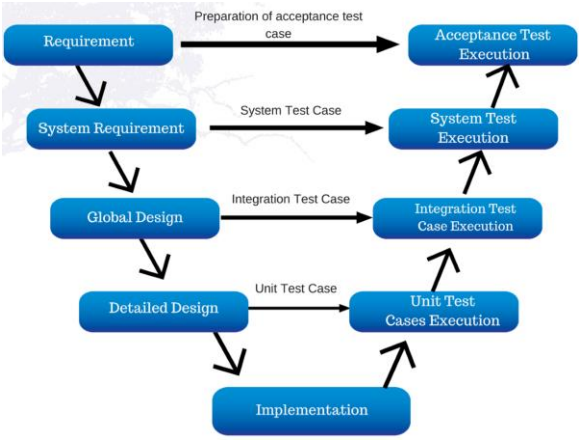
- This chart allows you to monitor fluctuations in product quality
- "Scope changes" – bug fixing, etc. Think this was named wrong by Fabian. More to do with issues found

6 TESTING

6.1 WHITE-BOX VS BLACK-BOX TESTING

6.2 GENERALITIES

- Testing in Agile is similar to the old V method



6.2.1 Objectives of testing

- "Testing can only show the presence of errors, not their absences" (Dijkstra)
- **Two main objectives:**
  - Validation – demonstrate the software fulfils the requirements
  - Verification – Identify erroneous behaviour of the system
- **Determine 'fit for purpose' state**
  - Purpose – safety-critical systems are tested differently (e.g. formally validated)
  - Expectations – For example, users name accept more instability in the product if it is new software
  - Marketing – time-to-market and price also influence testing and validation

6.3 TESTING PHASES

6.3.1 Unit Tests (Whitebox testing of implementation)

- Any piece of code should be tested
- Every feature should be testable and tested
- Avoid human or third-party interactions
- The tests should also prevent missuses
- Identify domains for values that are supposed to have the same effects
- Think about edge cases (values at domains' boundaries)

6.3.2 Component Tests (Blackbox testing of composition)

- Testing of identifiable and isolated parts of the system
  - Implementation is hidden behind interfaces and should be interchangeable
    - So you can change implementation whenever you want
- **Various types of interfaces**
  - Application – behaviour accessible through operations (e.g. Method calls)
  - Memory – shared block of memory between processes
  - Message – passing of data between operations through a communication medium (REST, http, etc)
- On top of being sceptical of input data
  - Test invalid values to make components fail (i.e. check for differing failures)
  - Stress testing
  - If a call order exists for a series of operations, try to call them in any order (out of order)

6.3.3 System Test (Scenario-based user testing)

- Integrate third-party components or systems
- May integrate pieces of code developed by different teams
- May be performed by dedicated testers, and not only by the developers
- **Scenario-based testing**
    - Main usages (full interaction flows) should be modelled
    - Start from (graphical) user interface interactions and go to the data layers
- Trace and record test executions in a tracking sheet
    - Input values, expected output, observed output
    - Metadata like who, when and possibly related bug/issue id

6.4 AGILE TESTING

6.4.1 Cycle

- Commit – Automated unit tests and code review at each commit
- Acceptance tests – Automated story-based tests at each commit
- Manual test – Release candidate testing at each sprint
- Performance – Automated capacity testing at each release

6.4.2 Code Review

- First step to identify problems such as
    - Invalid operation definitions (aka signatures) or usages
    - Missing comments or documentation
    - Missing unit tests
    - Rapid look at the code quality, but stay open-minded to different logic
- Code quality may also be monitored
    - Number of files, classes and lines of code (helps to avoid things like god classes)
    - Code duplications, documentation and comment rates
    - Code smells and technical debt
    - Tools for monitoring code quality, mainly Sonarqube, which can be triggered as part of a git pipeline

6.4.3 Smarter unit testing

- **Java assertion statements**
    - Check incoming parameter values
    - May also ensure invariant properties remain valid (and haven't varied)
    - *Assert*s are useful for regression tests
- Identify common programming mistakes
    - E.g. Database interactions, unmanaged null values...
    - Ensure tests are performed on these particular mistakes
- Thinking about Test-Driven-Development
    - Writing tests before implementation

6.4.4 Integration testing

- Done at each commit, so each commit (to the main branch of the repo)
    - Triggers a full run of all unit tests
    - Prevents system regression
    - Built binaries are passed to subsequent build stages
- Avoid third part components at this stage
- If build time is too long, use subset of tests
    - During off-peak time run a full test suite/build through the Continuous Integration system
- Any successful build becomes a release *candidate* to be further validated (manually)

6.4.5 Automated acceptance testing

- Unit tests are not be all end all, we need automated acceptance tests too
- Run of testing scenarios in a business minded mode
- **User stories are always accompanied by acceptance criteria**
- **What do we need to automate acceptance tests?**
    - Define application interfaces in front of your business logic (i.e. Isolate the UI)
    - Use dependency injection to isolate what you test
    - Create fake, minimal implementations for external dependencies (i.e. Use stubs – method stubs)
        - Method stubs are methods that stand in for another methods, they are methods aren't implemented yet
    - Create fake interactions with the database (i.e. Use mocking – faking the database using other implementation, i.e. using an ArrayList)
    - Split asynchronous scenarios into synchronous pieces
    - Keep the tests simple, think about maintenance
- Using playback tools, like Selenium or Serenity is not always possible
- Testing directly on the GUI may be painful
- If the system was written with testability in mind, UI should be agnostic of any logic
- **Cucumber**
    - Annotate test classes with *given*, *when*, *then* clauses
    - Create a test runner
    - Refer to run reports if there are problems
- Even if playback tools are uses, both types of tests should be written
    - Maintaining playback scripts (i.e. Selenium tests) is more difficult and time consuming
    - With well-designed business logic, the API tends to change less frequently

6.4.6 Capacity testing

- Some non-functional requirements are difficult to test
- Quality attributes are a major part of the system's requirements (e.g. security, performance)
- Maintainability and auditability and not testable properties
- However, capacity-based requirements may be tested (semi-) automatically
    - Response time may be expressed as user stories or required system features
    - Will be prioritised and evaluated through their acceptance criteria
- **Capacity testing environment**
    - It is better to have a clone of the production system, but this is not always feasible
    - You may benchmark and apply scaling factors
    - Canary deployment is also a valid live-testing solution to monitor increases in charge

6.4.7 Sprint review

- Go through your code before a review, if something doesn't work, roll it back
- Last opportunity to ensure everything is okay
    - Do a last-minute-verification, i.e. smoke test
    - Helps to make you confident in the review
- Remember to keep track of testing results appropriately, including for smoke tests

7 Reliability and Resilience Engineering

7.1 Reliability Engineering

- Humans cause faults leading to errors
  - Human error – inputting invalid data into the system or misbehave
  - System fault – characteristic leading to an error
  - System error – visible effects of a misbehaviour of the system
  - System failure – the system is not able to produce an expected at all

7.1.1 Reliability improvement mechanisms

- Fault avoidance – design and development processes, tools and guidelines
- Detection & correction – testing, debugging and validation
- Fault tolerance – system is designed to handle and recover from failures
- A trade-off must be found between correction and consequences

7.1.2 Availability and reliability

- **Availability and reliability are different terms**
  - Availability – probability to successfully access the system at a given time
  - Reliability – probability of failure-free operations
- **Work-as-designed problem**
  - System specifications may be wrong (i.e. Not reflecting the user's truth)
  - System specifications may be erroneous (did someone proof-read them?)
- Reliability may be subjective
  - Concentration of errors in a specific part of the system
  - Systematic longer response time at a specific time
  - May only affect a subset of users
- **Reliability metrics**
  - POFOD – Probability Of Failure On Demand (e.g. Services misbehave on 1/1000 requests)
  - ROCOF – Rate Of Occurrence Of Failures (e.g. Two failures per hour)
  - AVAIL – Availability for servicing (e.g. The magic 99.9% availability)
  - These are common in safety-critical systems but not only in safety-critical systems
  - Reliability metrics are also used to decide if the system is ready for production
- **Reliability requirements**
  - Functional – specifying additional input checks, recovery or redundancy processes
  - Non-functional – using above reliability metrics (or other metrics)
  - These requirements might not necessarily be system-wide
  - Don't need to try reach the 99.9% if it's not critical (this is very costly)
  - Runtime fixes are sometimes economically preferable
- Carefully think about the software architecture, especially for input/output
- Use threads carefully (starvation and deadlocks are bad)
- Write dedicated tests and monitor production system, if possible
- Forget about small efficiencies in your program, roughly 97% of the time, they are not worth the time you spend to solve those efficiencies

7.1.3 Architectural strategies

- Protection systems – system specified in monitoring the execution of another
- Self-monitoring systems – concurrent computation, mismatch detection and transfer of control
- Multi-version programming – mainly for hardware-fault management, e.g. triple-modular-redundancy
  - Have three of the same hardware stack, and the system is good when two give the same output
- These strategies must be put on top of your in-code fault detection

7.1.4 Reliability guidelines for programmers

- Visibility – apply the need-to-know principle
    - Need-to-know principle is effectively the information hiding principle
- Validity – check format and domain of input values explicitly or by using *regression-test-enabled* asserts
- Exceptions – avoid errors to become system failures by capturing them (catching exceptions)
- Erring – avoid error-prone constructs (e.g. Implicit variables) and encapsulate 'nasty' stuff
- Restart – provide recoverable milestones
- Boundaries – take extra care when manipulating indexed structures and make additional checks to avoid security issues or failures
- Constants – express fixed or real-word values with meaningful names, this creates 'self-documented' code

7.2 SECURITY ENGINEERING

7.2.1 The internet

- The emergence on the internet in the 90's created new threats
    - Confidentiality – access to information by unauthorised people and/or programs
    - Integrity – corruption or damage to a system or its data
    - Availability – disallow access to a system or its data
- There are three levels of security to consider
    - Infrastructure – maintain the security of systems and network services
    - Application – security of (a group of) application systems
    - Operational – Security related to the usage of the organisation's system
- All layers of the application stack are threatened, not just the frontend (or whatever)

7.2.2 Security strategies

- Know that attempts to breach a system's security are inevitable and plan for it
    - Avoidance – apply measures to avoid having sensitive data exposed in the first place
    - Detection – monitor possible attacks on a system and take actions against those attacks
    - Recovery – backup, deployment and insurance-related procedures
- A security checklist
    - Permission – ensure file permissions are right
    - Session – terminate user sessions that are inactive for too long
    - Overflow – take extra care to avoid memory overflows
    - Password – enforce sufficient password strength (length, characters, etc)
    - Input – check inputs are correctly formatted and comply with what you expect

7.2.3 Resilience Engineering

- **Resilience Engineering focuses on recovery, not protection**
    - It assumes failures will happen, if your system is even somewhat open to the world, it will be attacked
- In organisations, **security threats may come from**
    - Ignorance – someone doesn't know about a potential risk
    - Design – the system has been developed without (any) security in mind
    - Carelessness – the system has been developed badly by developers or system engineers
    - Trade-offs – not enough emphasis on security to focus on other things
- There are **three levels of protections**
    - Access – provide the right level of authentication mechanism
    - Assets – encrypt sensitive data and separate the database from the software
    - Network – protect and monitor networks and gateway machines

- **The four Rs of a Resilience Engineering plan**
    - Recognition, Resistance, Recovery, Reinstatement
    - **Classification** – hardware, software and human resources, e.g. old hardware is prone to failures or security exploits
    - **Identification** – potential threats to those resources
    - **Recognition** – how an attack may target an identified resource (AKA *asset*), e.g. what are the vulnerabilities
    - **Resistance** – possible strategies to resist each threat
    - **Recovery** – plan data, software and hardware recovery procedures
    - **Reinstatement** – define the process to bring the system back
- Resilience planning is a design activity by itself
    - Identification of resilience requirements (e.g. Availability under attack)
    - Backup and reinstating procedures must be specified on top of the deployment
    - Appropriate classification of critical assets and how to work in degraded mode
    - The above should be tested as well
- **Modelling threats**
    - By creating misuse cases (or stories)
        - Helps to define the tests
        - Depict the misuse together with protection or corrective actions
        - Allows you to create a map of vulnerabilities (classification step above)
        - Allows you to identify critical assets
- **Write dedicated security tests**
    - For your software
    - For brute-type access to data
    - For any interactions with third parties

8 CONTINUOUS INTEGRATION

8.1 THE INTEGRATION PROBLEM

- Process of combining units into a product
- **Even if every part it 100% functional, problems may arise such as**
  - Conflicting dependencies
  - Integration contracts (i.e. APIs) poorly specified
- Diagnosis is harder as more units are combined
- Integration may become a nightmare if everyone builds things and waits until then end before merging
- Focus on structured testing and gradual integrations
- Iterative development also means iterative integration
- Concrete strategy is often hard to decide beforehand
- Testing infrastructure must often be large, even in the early stages
- The more you integrate, the more you need to test (and debug)

8.1.1 Phased integration vs Incremental integration

- **Phased integration**
  - Integrate everything at the same time
  - For example, merging at the end of every sprint
  - There may be multiple variables, interfaces and technologies
  - Makes for a bad time
- **Incremental integration**
  - For example, every time you add something small(ish), try to integrate (merge) it
  - Then re-test the program with the small thing added
  - Still not the way we should do it, see integration strategies

8.2 INTEGRATION STRATEGIES

8.2.1 Top-down integration or bottom-up integration

- **Top-down**
  - Start with interaction points (UI) and go down
  - Requires (many) fake implementations (AKA stubs)
- **Bottom-up integration**
  - Start with business and data layer, and go up
  - Require (many) test executors (AKA drivers)
  - Possible late discovery of entry-point problems

8.2.2 Risk or feature oriented integration

- **Risk-oriented**
  - Start with the riskiest part (usually the 'extreme' ones)
  - Often underestimates the importance of middle layers
- **Feature-oriented**
  - Test and integrate correlated parts (of a feature or some functionality)
  - Feature may cover too many modules

8.3 Fowler's Principles

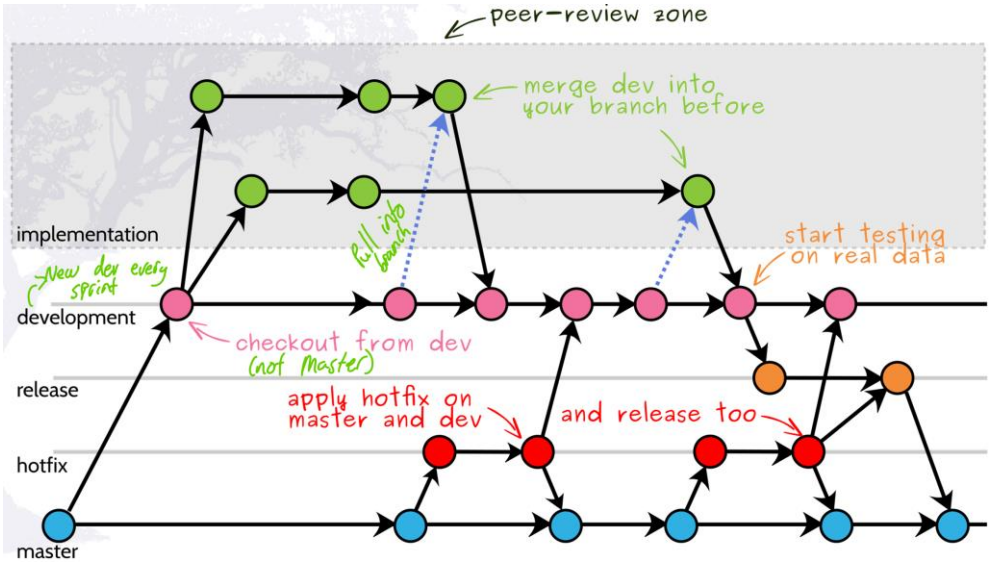
- Martin Fowler defined 11 principles (as part of eXtreme Programming):
  - Maintain a single source repository
  - Automate the build
  - Make your build self-testing
  - Everyone commits to the mainline every day
  - Every commit should build the mainline on an integration machine
  - Fix broken builds immediately
  - Keep the build fast
  - Test in a clone of the production environment
  - Make it easy for any to get the latest executable
  - Everyone can see what's happening
  - Automate the deployment phase

8.4 Managing Sources

8.4.1 Single source repository

- Maintain the code and resources in one single place
  - This means many people can be involved (and not all in the same geographical place)
  - Centralised traceability of changes (able to see who made changes and when)
  - Newcomer may spawn the project and start working (quickly)
- **Tool support for version control**
  - Centralised – one single master repository (e.g. Subversion)
  - Distributed – two levels of commits (e.g. Git)
- Single repository does not mean a single copy, but one mainline
  - Master branch with production code
  - Other branches with development features, plus hotfixes

8.4.2 Branching and merging



8.4.3 Everyday commits

- **Pushing your work every day is meant to**
  - Allow you to identify problems sooner
  - Lower the occurrences of merge conflicts
  - Ease the integration tasks (because there are smaller units to debug)
- Be careful, your code must build at the end of the day
  - Push partial implementations (and don't break the dev line/branch)
  - Ensure tasks are not under-estimated, they need to be completable before the end of the day
- Everyday commits are motivating for developers as you can see your (steady) contributions, and so can others
- Merge requests are another layer of quality assurance
  - Formal peer-reviewing
  - Must not leave merge requests waiting for days

8.5 MANAGING BUILDS

8.5.1 Automated build

- Use an external instance that is accessible to everyone (i.e. An integration server with automated builds)
- Nightly (daily) builds are not enough by themselves, but are still needed
  - This means you are allowed to have two levels of builds so that dev commits are faster
    - A longer build for daily builds
    - A faster build for dev commits
- **Building software requires many tasks**
  - Compiling and linking
  - Loading database schema
  - Running database evolution scripts
  - Moving resources to the right place
- Manual tasks are error-prone and time consuming
  - So automate the build with adequate scripts
  - There are many technologies that do this (maven, gradle, sbt, etc)
- Automated builds allow newcomers to start up in minutes, then don't need to struggle with the technologies and dependencies
- **Self-testing build**
  - Self-testing builds are meant to keep the dev mainline (branch) healthy
    - So do not commit before testing and building (yourself)
  - Setting up a fully working test environment is time-consuming, especially when there is time pressure to deliver
  - Automating the build with upfront tests enables you to
    - Identify bugs as early as possible
    - Provide adequate feedback to developers
    - Lower the risk to commit non-runnable code
  - Tests are then fully apart of commits and builds
    - As the product grows, more tests should be added
    - Unit testing may be accompanied by functional/user tests
    - Non-functional tests are also sometimes added too
      - Non-functional test – testing a non-functional requirement, e.g. testing the speed of a request
  - Automated tests are not the be all end all, but they are valuable health indicators

8.5.2 Fix it immediately

- If the development mainline (branch) may not build any more
  - Fixing this should be the highest priority
- The development mainline must stay as stable as possible
  - This is because developers are checking/pushing/building/testing it many times a day
- **Techniques to avoid blocking the whole team**
  - Minimal local testing and building environments
  - Duplicate mainline that is not updated on failed build

8.5.3 Fast building

- The whole continuous integration process demands time
  - Many commits means many build
  - Long build times mean time lost on development task
- **Subsetting, pipelining and faking**
  - Consider concentrating on a subset of tasks
  - Use fake stubs for external services or resources (e.g. Database)
  - Asynchronously run one or more tests are the CI build
- Fast building can be accomplished through a combination of continuous integration and frequent restrictive builds (having two separate CI build processes, one faster, one slower)

8.5.4 Test for real

- Testing is crucial in continuous integration
  - It allows us to discover problems as they arise
  - And to ensure anyone may get the latest stable build
- The testing environment should mimic the production system, even if it will never be an exact copy
  - Virtualization and snapshot are making this simpler these days, for example
    - Virtual machines (virtual boxes)
    - Other deployable third-party applications (docker, Kubernetes, etc)

8.6 EASE THE SUCCESSION

8.6.1 Access to the latest build

- Everyone should have access to the latest build
- Everyone should have an idea of the running state
  - It's always easier to understand a system when you can see it and interact with it
- As the sources (source code) are accessible, the latest executable is accessible
  - They should be in a dedicated repository everyone knows about
  - This build should be as stable as possible
- This latest build should have passed the pipeline tests

8.6.2 Access to the source (code) and build traces (commit history)

- Traceability is crucial in continuous integration
  - Documentation should be very close to the code
  - Changes are made incrementally (instead of making large commits)
  - Everyone works in a desynchronised manner with individual planning
- Commits must be accompanied by meaningful comments
  - This is so any team member can see what the latest changes are
  - And so, any team member can take over if needed (failed build, for example)
- A summary calendar with major builds/releases is an asset
  - Many source repositories have browsing and visualization facilities for this
  - Add a summary of the build/releases/milestones in your project room

8.6.3 Automated deployment

- Requires many testing environments and instant start-up of the program
  - Requires scripts for building
  - Able to also script the deployment
- Deploying means performing rollbacks as well
  - If an unexpected error occurs
  - Testing on a clone does not provide a 100% guarantee
- Bad deployments are inevitable
  - Real data and users will sometimes trigger unexpected behaviour/bugs/effects
  - We always need a plan B

9 CONTINUOUS DELIVERY

9.1 BACKGROUND AND PRINCIPLES

9.1.1 Deployment time is stressful

- **Release dates are often sources of anxiety**
  - Does everything compile and run on the target environment?
  - Did we forget some dependencies?
  - Have we moved all needed resources?
  - Is the database schema correctly updated?
- **If tasks are executed by hand, many (more) things can go wrong**
  - When new dependencies or new versions of dependencies are needed
  - When the release procedure has been changed
  - When manual fixes are performed
  - When you deploy for the first time on production
- **Reduce the stress by getting used to deployment**
  - By increasing release frequency, you either suffer more or find a way to make it easy
  - This is where continuous integration has virtuous side-effects
    - You have automated builds, with testing
    - You have enforced self-contained sources
    - You have written (partial) deployment scripts
    - You have done it many times for smaller deltas (delta = change(s))
  - Continuous deployment gives the opportunity to test and practice deployment processes

9.1.2 Anti-patterns (ineffective & counterproductive patterns) of deployment

- **Doing everything manually**
  - Extensive and verbose documentation (on deployment, not code)
  - Relying on manual testing during the deployment night or weekend
  - Calling the development team to figure out what is wrong
  - Updating the deployment sheet since the production environment is different
  - Deal with manual 'tweaks'
- **Waiting until the very end to deploy a release**
  - The software is testing on its new platform (deployment platform) for the first time
  - Many organisations are separating developers from IT system engineers
  - The bigger the system is, the more uncertainties exist (i.e. Side effects)

9.1.3 Benefits of automated and continuous delivery

- Empowering teams – test and deploy the build you want
- Reducing errors – avoid unversioned configurations, which are error-prone
- Lowering stress - frequent and smaller changes, accompanied by rollback process
- Deploy where you want – get the version you want where you want when you want
- But decide when you deliver – this is different than continuous deployment wherein the CI triggers the deployment

9.2 DEPLOYMENT STRATEGIES

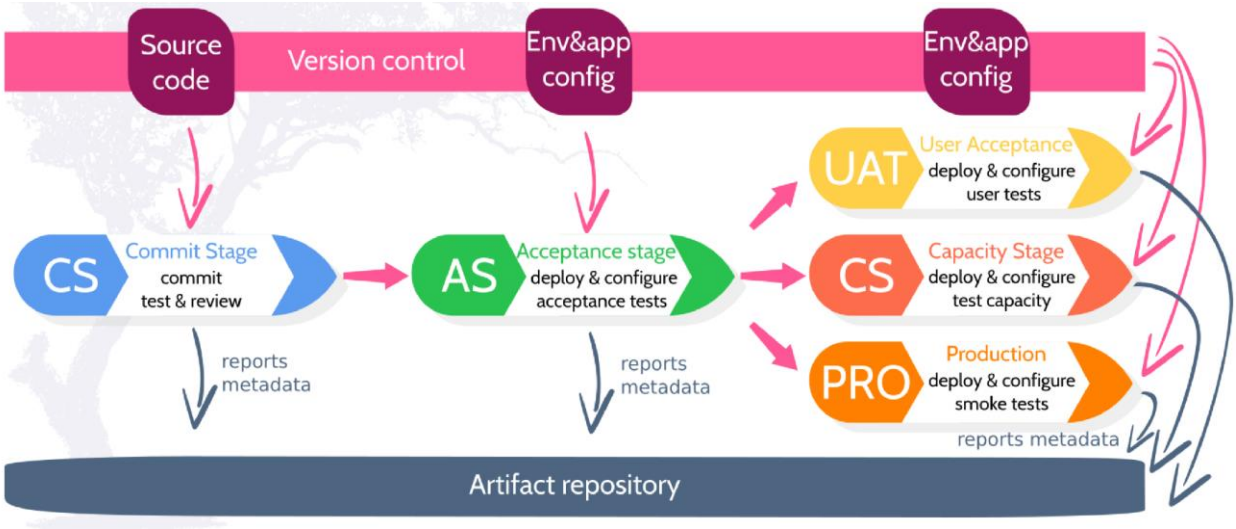
9.2.1 Adequate preparation

- **Upfront work is required**
  - Model the target environment appropriately
  - Prepare the deployment infrastructure carefully
  - Master servers' configuration (avoid obscure GUI-based configs)
  - Define the deployment strategy collaboratively
- **So, automate as much as possible**
  - Many third-party libraries/servers/services allow the use of textual config files
  - Use build chains and dependency management systems (e.g. Maven, ant, sbt)
  - Preparing virtual environments is also a potential solution
- **Don't forget a disaster recovery process in case of a last-minute failure**

9.2.2 Configuration management

- **Keep everything under version control**
  - Your sources (source code), obviously
  - Your configuration files, including DNS zone files or firewall setting
    - But DON'T include passwords or other sensitive data in version control
- **Use your version control smartly**
  - Push regularly to main development branch
  - Use meaningful commit messages
- Think about dependencies
  - Automated retrieval dependencies is handy
    - But avoid having to 'download the internet'

9.2.3 Deployment pipeline



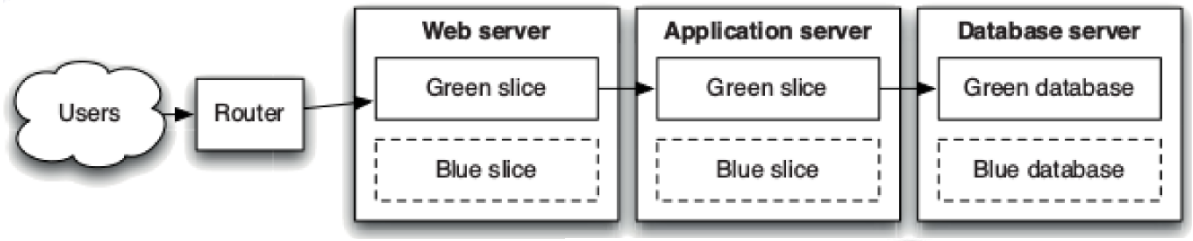
- **Practices**
  - Build only once – even a slight difference in build environments may cause problems
  - Deploy the same way, everywhere – following the same deployment procedure for actual testing on multiple environments
  - Smoke-test your deployments – also script the automated start-up of the system with a few simple requests
    - Smoke-test – basic tests that test the whole system doesn't fail (e.g. turning on car, checking there is not smoke)
  - Deploy into a copy of production – mimic as much as possible (network, firewall(s), OS, application stack)
  - Propagate changes into the whole pipeline as they appear – as changes trigger the pipeline, the latest available version is built and/or tested

9.2.4 Creating deployment strategies

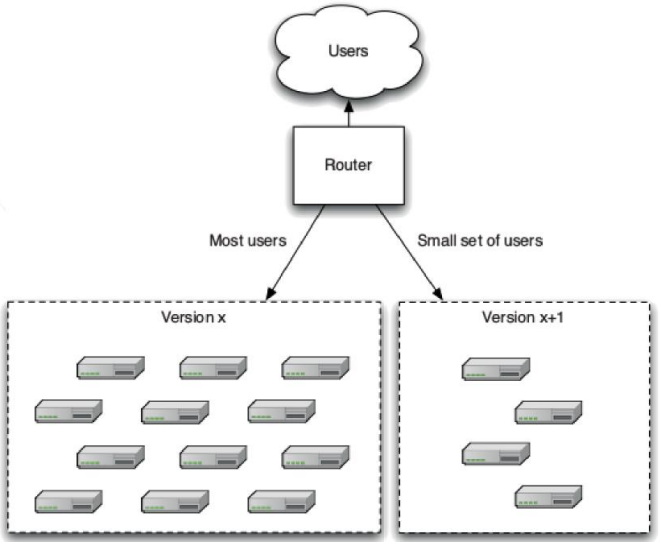
- To prepare your strategy, you need
  - All parties in charge of the various environments
  - A deployment pipeline plan and a configuration management strategy
  - A list of environment variables and process to move from one environment to the other
  - A list of monitoring requirements and solutions
  - To discuss the disaster recovery plan
  - To agree on a Service Level Agreement (SLA) and on support
  - An archiving strategy of outdated data (for auditing)
- The strategy should also encompass data migration, upgrades, patches and bug fixes

9.2.5 Example deployment strategies

- **Zero downtime releases**
  - As much as possible, reduce downtime
    - Downtime is frustrating for users
    - Releases must be minimised or during off-peak time
  - Many *hotplug* facilities exist
    - Decouple the whole system in pieces, replacing pieces one after another
    - Offer roll-back to previous version if necessary
  - Zero-downtime is not always possible
    - Web-based systems may be rerouted, even temporarily
    - But data migrations often need some time
- **Blue-green deployment**
  - In distributed systems (e.g. Web applications)
    - Rerouting is rather easy
    - Possibility to let multiple versions available at the same time
  - Two identical environments
    - Simple rollback of routing rules, if necessary



- **Canary deployment**
  - Real systems are not always easily cloneable
    - Real-world constraints are not fully testable upfront
    - Scalability and response time are crucial features
  - As in blue-green deployments, canary deployment needs a router
    - Deploy a subset of servers with the new version
    - Use a (well defined) subset of users
    - Gradually increase the number of users
  - This allows developers to collect feedback fast



10 AGILE SOFTWARE MODELLING

10.1 AGILE SOFTWARE MODELLING BASICS

10.1.1 Why should we model?

- Do we know about the problem domain?
- Can we see how the whole program should be laid out?
- Is the solution not straightforward?
- Is the transfer of knowledge required between individuals?
- Will the solution need any changes in the future?
- You must understand the business terms
- Concrete solutions aren't always that simple
- You don't work alone
- Software has a beyond its creators
- **Concretely**
    - Handling complexity
        - Many functionalities
        - Many concurrent interactions
        - Many constraints
    - Identifying components
        - Reuse existing facilities
        - Reuse existing resources (external resources, libraries)
    - Externalising aspects

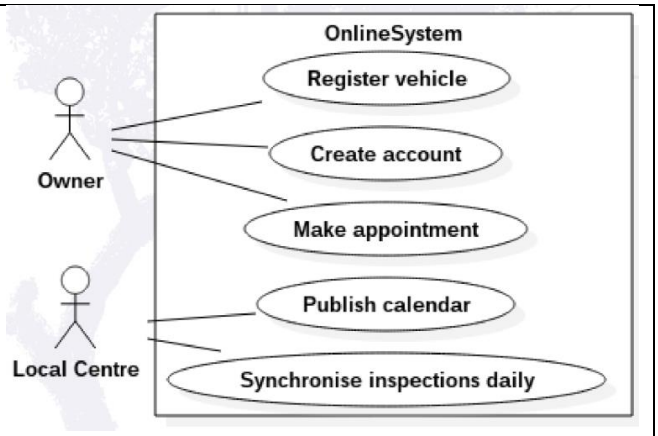
10.1.2 Representations

- Representations (visuals) are good, but you need to know about the
    - Legend
    - Context
    - Syntax
    - Purpose
- To be useful, representations should
    - Be unambiguous
    - State clearly what they represent
- When designing representations, we are modelling the reality

10.2 AGILE AND MODELLING

10.2.1 Use case diagrams to summarise many stories (epics)

- As your backlog grows and stories are added, it may be useful to group stories together into themes and/or epics
- We can do this with use case diagrams
    - Allows us to identify actors, i.e. interested people using the system
    - Gives us a one picture summary of objectives
    - May help to identify missing functionality or features
    - Does create additional documentation workload
    - But offers a higher level of abstraction on epics
    - Allows you to present your system to others in a more concise and simple way



10.2.2 Wireframes

- A rapid and effective way of prototyping and conveying ideas
- Can detail what information will be required for a form
- Allows you to specify domain concepts and their attributes
- Enables you to discuss a general layout without implementing it
- They are a good starting point

10.3 DECISIONS AND DESIGN

10.3.1 Tactics, styles and patterns

- Architecture tactics
    - Relate to one quality attribute (e.g. security – tactic solution is to use auth, etc)
    - Tackle one concern at a time
    - Linked to one (architectural) decision
- Architecture style and design patterns
    - Offer reusable off-the-shelf conceptual solutions
    - Must be taken into account throughout the whole project
    - May encompass multiple tactics
- Both helps shape the design of your system
    - Architecture drift occurs faster than you'd expect
    - Migration may be painful or impractical

10.3.2 Modelling is also about documentation

- README files
    - Allows you to explain the context and objectives of the program
    - Specify the authors, contributors and version of the program
    - Specify the deployment and testing procedure
    - Specify the dependencies
    - Specify copyright permissions and licensing
- Wiki pages
    - Put your external analysis (e.g. wireframes, architecture)
    - Manual tests (with results), DoD/DoR, action points
    - Keep organised with categories and keep it updated

11 User Experience

11.1 Basic principles

11.1.1 User experience is more than usability

- **User experience (UX) encompasses many aspects, such as**
  - Meet the users' requirements (i.e. Usability)
  - No fuss no bother (i.e. Efficiency)
  - Simplicity and elegance (i.e. Satisfactory)
- **UX designers need empathy**
  - Putting themselves/yourselves in your users' shoes
  - Know about their prior knowledge and target knowledge
  - Understand the users' frustrations and desires

11.2 Making a great UX

11.2.1 User experience Honeycomb

- Useful – is your interface answering a need?
- Usable – is it easy to navigate and accomplish tasks?
- Findable – can users find what they are searching for?
- Credible – does it look legit? (does it (not) look like a scam?)
- Accessible – can disabled people access your interface?
- Desirable – does your UI trigger users' emotions?
- Valuable – all the above makes it valuable

11.2.2 Rules of thumb

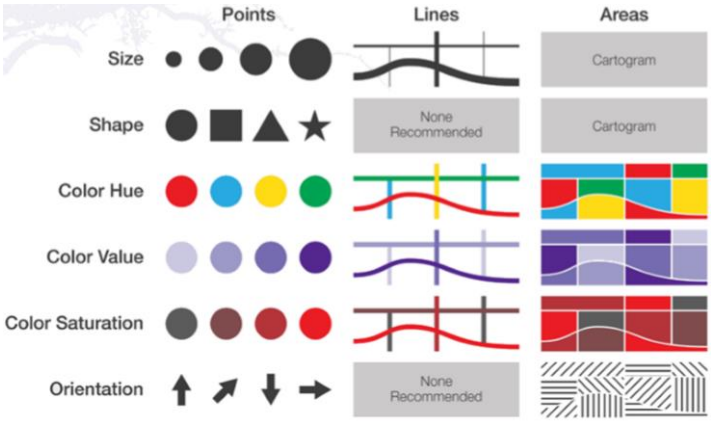
- Simplicity – do not overload the user interface, avoid auto-start
- Consistency – same things in the same place with the same look and feel
- Design makes the difference – two identically functional systems will be distinguished by their UI
- Test, test, test – test a lot
- Keep users informed – give feedback, avoid waiting time, no isolated pages
- Appropriate content – avoid long pages and expired information, use interactive content and infographics

11.2.3 Tools for a UI designers

- Input controls
  - Text/pre-formatted fields
  - Toggle/slider buttons, checkboxes and radio buttons
  - Dropdown and list boxes
- Navigational components
  - Breadcrumbs, search fields and pagination
  - Hyperlinks, tags and icons
- Informational components
  - Tooltips, icons, progress bars
  - Message boxes, notifications, modal windows
- All sorts of containers (e.g. Accordions)

11.2.4 Information graphics (infographics)

- **Infographics design process**
  - Decide what you want to tell? To whom?
  - Recognize the information potential
  - Explore the visualisation possibilities
  - Choose your format and wireframe/prototype your concept
  - Test your concept with users
  - Choose a colour palette
  - Draw shapes and actions
  - Decide on typography
  - Build for print, video or web



12 Developer Experience

12.1 Basic principles

12.1.1 Communication and engagement

- Be open in your communication within and outside the team
- Commit yourself, seek help and give your time

12.1.2 Be consistent in your development

- Stick to code style and existing libraries
- Do not over-engineer

12.1.3 Don't be 'that' guy

- Help the team find the working pace
- Follow the rules, platforms and toolsets your team agreed on

12.2 What is 'building an API'

- Documentation, comments and README
  - Write it as if a junior had to take over your job
- Tests and Minimum Working Example (MWE)
  - An easy way to ensure you deliver the right thing
  - MWE – a collection of source code or other data files which allows a bug/problem to be reproduced
- Release notes and change logs
  - Make it clear what you shipped and how to make it work
- Keep it simple, do not overcomplicate
  - Design usable stuff, you shouldn't need to be a genius to use the API
- Contribute and be part of the community
  - The open source needs you; you could consider contributing to an open source project

Part 2: Moffat

13 COLLECTIONS

13.1 BASICS

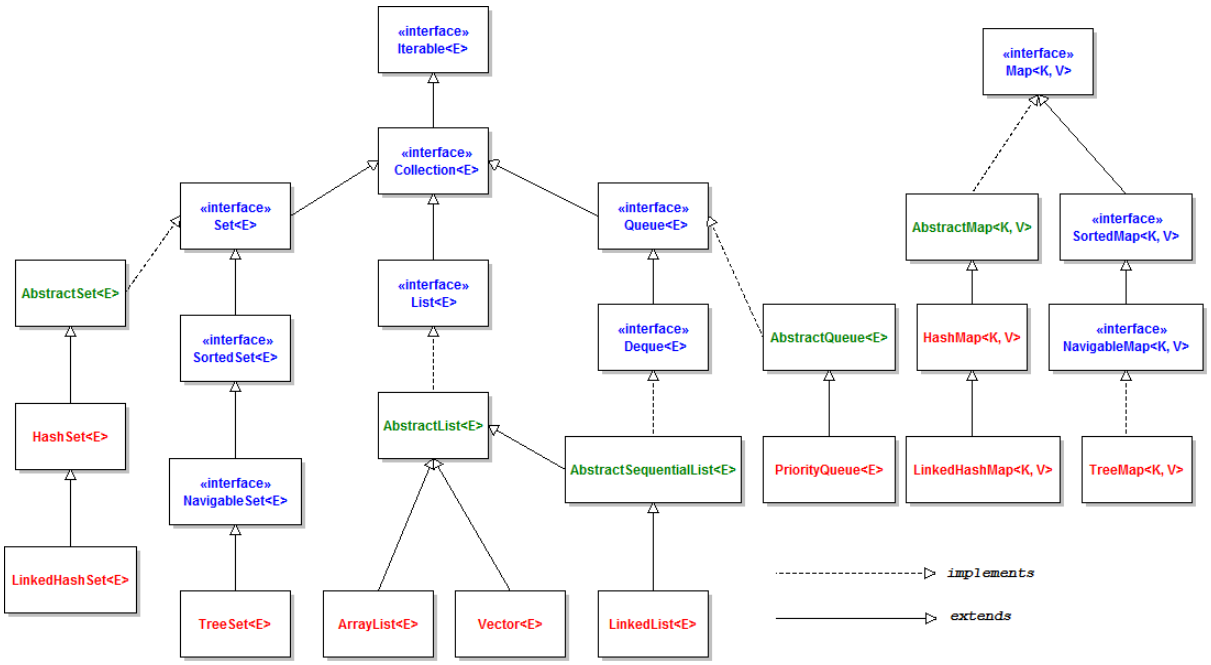
13.1.1 What is a collection?

- **A collection is an object (by itself) that**
  - Contains a bunch of objects
  - Supports iteration
  - Implements a relationship
  - Is implemented by data structures and algorithms
- **Containing a bunch of objects**
  - Without collections
    - Array – We must define functions to access items... etc
  - With collections
    - Set
    - List
    - Queue
    - Map (key & value)
    - Collections already have functions defined (get items, add items, etc)

13.1.2 Why not just DIY?

- I.e. Without collections or arrays, just using references between objects (think Student in Richard Lobb's C programming)
- Because that sort of method is restrictive
  - It is restrictive in how you solve problems; it forces you to think about low-level details
  - This takes your attention away from the design

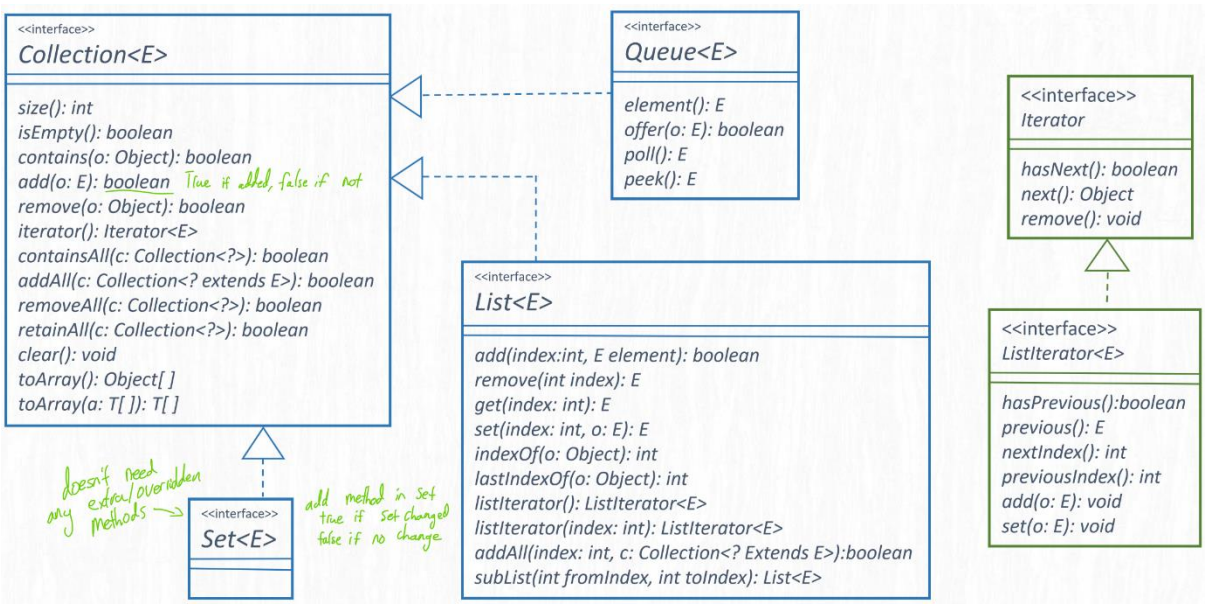
13.2 JAVA COLLECTIONS

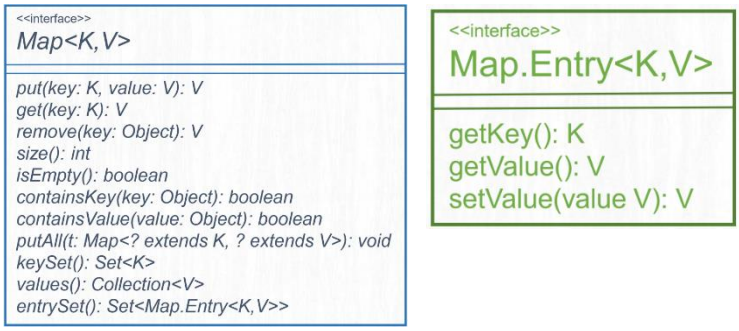


13.2.1 Thread safe collections

- BlockingQueue<E>
  - Parent: Queue<E>
- BlockingDeque<E>
  - Parent: Deque<E>
- ConcurrentMap<K, V>
  - Parent: Map<K, V>
- ConcurrentNavigableMap<K, V>
  - Parents: NavigableMap<E> & ConcurrentMap<K, V>

13.2.2 Collection classes:





13.2.3 Collection interface notes

- **Set**
  - Unique – no duplicates
    - add() method might not actually add, it calls object1.equals(object2)
  - Normal implementations
    - HashSet
    - TreeSet
    - LinkedHashSet
  - Special implementations
    - EnumSet<E extends Enum<E>>
    - ConcurrentSkipListSet
    - CopyOnWriteArraySet

- **List**
  - Ordered – added items maintain order (i.e. The user controls the sequence)
    - add() method appends to the list
  - Normal implementations
    - ArrayList
    - LinkedList
  - Special implementation
    - CopyOnWriteArrayList
  - Legacy implementation
    - Vector
- **Queue**
  - Special ordering
    - First In First Out (FIFO)
    - Last In First Out (LIFO)
    - Priority (queue)
    - And more
  - Normal implementations
    - LinkedList
    - PriorityQueue
- **Map**
  - Not a collection, but plays nicely
  - Normal implementations
    - HashMap
    - TreeMap
    - LinkedHashMap
  - Special implementations
    - EnumMap<E extends Enum<E>>
    - ConcurrentMap
    - IdentityMap
  - Legacy implementations
    - HashTable

14 General Software Design

- Goal: software that is reliable, usable and maintainable
    - Modular design seems to be one of the best available solutions to date
- **Components (units) of a modular system are**
    - Manageably small
    - Physically & logically related
    - Separately maintainable
- **The alternatives to a modular system include**
    - Monolithic systems
        - Components are so highly inter-related that they behave like a single piece
    - Spaghetti code
    - Ravioli code
    - Spaghetti with meatballs code

14.1 Interface vs Implementation

14.1.1 Interface

- Examples of 'interfaces' in real life
    - TradeMe
    - A bookshop
    - A supermarket
- Common theme: Don't know how they sell, but you know what they sell and what you can buy
- Defines and enforces behaviour
- "Services" offered
- Defines a "contract"
- Initial design at the interface level

14.1.2 Implementation

- In reference to the interfaces example, implementation is how they actually run the company
    - Shares
    - Board of directors
    - Bank accounts
    - GST
    - etc

14.2 Modular Design

14.2.1 Cohesion

- The degree of connectedness **within** a module/unit

14.2.2 Coupling

- The degree of interdependence **between** modules/units
- The strength of the relationship between modules/units

14.2.3 Functional Independence

- Software modules/units should have functional independence
- This is measured by **Cohesion**
    - Intra-module
    - How much this module does just one thing
    - Module consists of activities, which belong together
    - Functional strength
- and **Coupling**
    - Inter-module
    - How much this module is connected to another module
    - Modules connected across narrow interfaces & don't refer to each other's internal variables
    - Functional/module independence
- **Related principle: Cohesion: good; coupling: bad**

14.2.4 How to partition large systems into modules

- Determine suitable module contents
- Establish module hierarchy, composition structures
- Inter-module communication (what and how)
- Design QA: good or bad modularisation?
- Independence of scale of problem
- Consistency at all levels: system, module, code
- Repeatability (no 'magic')

14.3 Design Methodologies

- In practice, a hybrid of these approaches may be appropriate

14.3.1 Top-down

- Break up complex problems so it is easier to solve
- **Functional decomposition**
    - What do we decompose with respect to?
        - Time?
        - Data items?
        - Control flow?
        - Resources?
        - Contract?
    - Choice of decomposition criteria can greatly affect cohesion and coupling of the resulting (modular) design
        - No universally right answer exists, it depends on your program
    - Functional decomposition works well with procedural languages and less well with object-oriented languages
- Stepwise refinement (common to many techniques)
- Transaction analysis

14.3.2 Bottom-up

- Combination of lower-level components
- Generally, less favoured than top-down methods
- Not closely related to the structure of the problem
- Leads to proliferation of **potentially** useful functions/methods rather than the most useful ones
- Can be used to hide the low-level details of implementation and be merged with a top-down technique
- Used when we have no idea how to solve the problem

14.3.3 Nucleus-centred

- "Start with the tricky bits"
- Information hiding
- Decide about critical core (algorithms, etc) then build interfaces around it
- Identify design decisions with competing solutions (eg. Choice of sorting algorithm)
- Isolate details; interface should not depend on details
- Motivation for OO (state and access methods encapsulated)

14.3.4 Object Orientated design

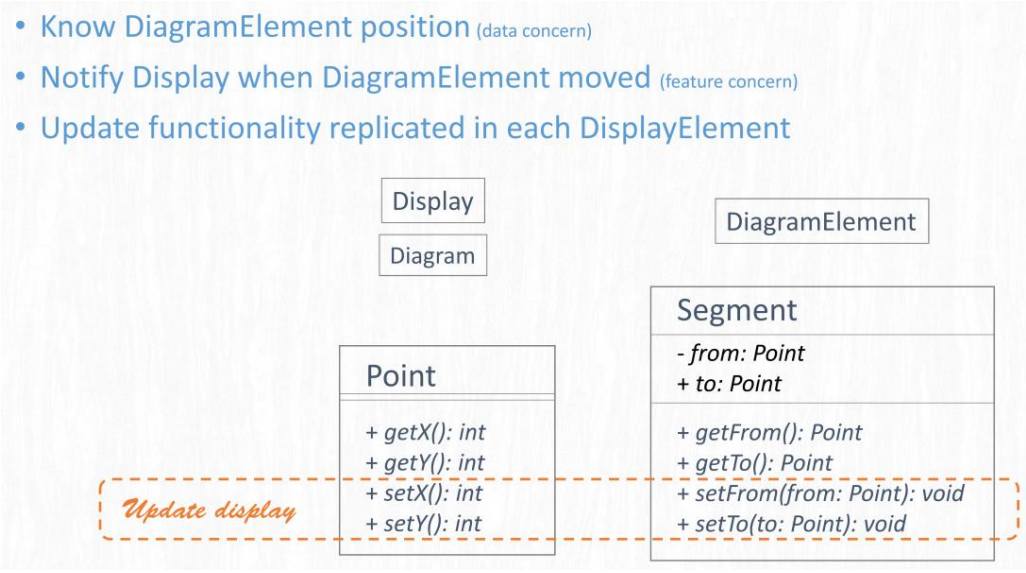
- Encapsulate data and behaviour
- Less shared data
- Low coupling – Ideally only communication is via message passing
    - Assists parallel or distributed implementations
- Easy to change implementation details
- Patterns
- Reuse

14.3.5 Aspect-oriented

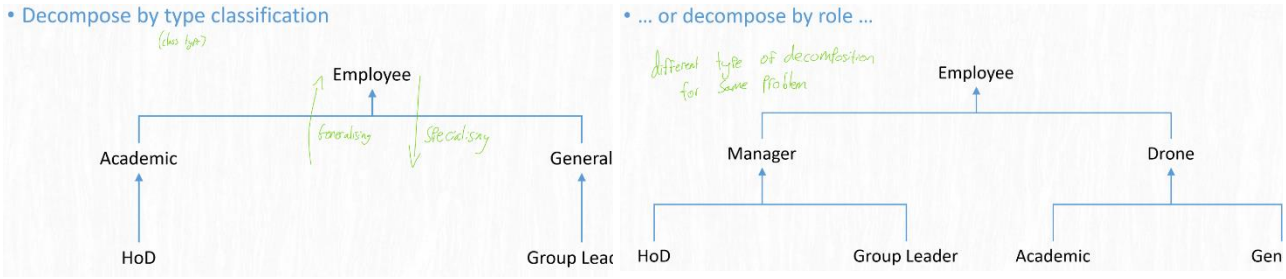
- "Subject-oriented" programming
    - Separation of concerns
    - Data concerns, feature concerns
    - "Crosscutting"
        - Having all your classes do something i.e. I want all my classes to log:

Class 1	Class 2	Class 3	Class 4	Class 5
Logging	Logging	Logging	Logging	Logging
Func1	Func2	Func3	Func4	Func5

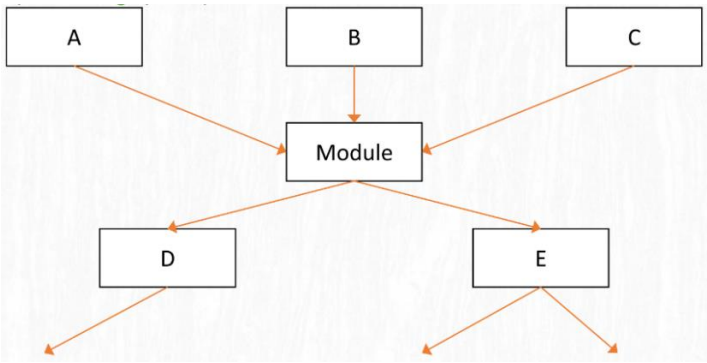
- Can be used for things like
    - Persistence
    - Exception/error handling
    - Security
    - Logging
    - Transactional management
    - HR feature
    - Payroll feature
- Represent system in terms of aspects, join points
- Tools:
    - AspectJ (eclipse)
    - Spring Framework
- Example:



14.3.6 Decomposition example



14.4 STRUCTURAL FAN-IN AND FAN-OUT



14.4.1 Fan-in

- How many things are call me?
- Number of superordinate (calling) modules
- Maximise to reduce code duplication
- If a module breaks, all 'in' (calling) modules also break

14.4.2 Fan-out

- How many things am I calling?
- Number of immediately subordinate (called) modules
- Neither too low (0, 1) nor too high (7 ± 2)

15 Object-oriented (OO) Design

15.1 Fundamentals & principles

- We have only one problem: **complexity**
  - Size
  - Number of parts
  - Connectedness
  - Change
- We have only one solution: **decomposition**
  - Break up
  - Hide parts
  - Decouple
  - Abstract

15.1.1 Encapsulation / Information Hiding

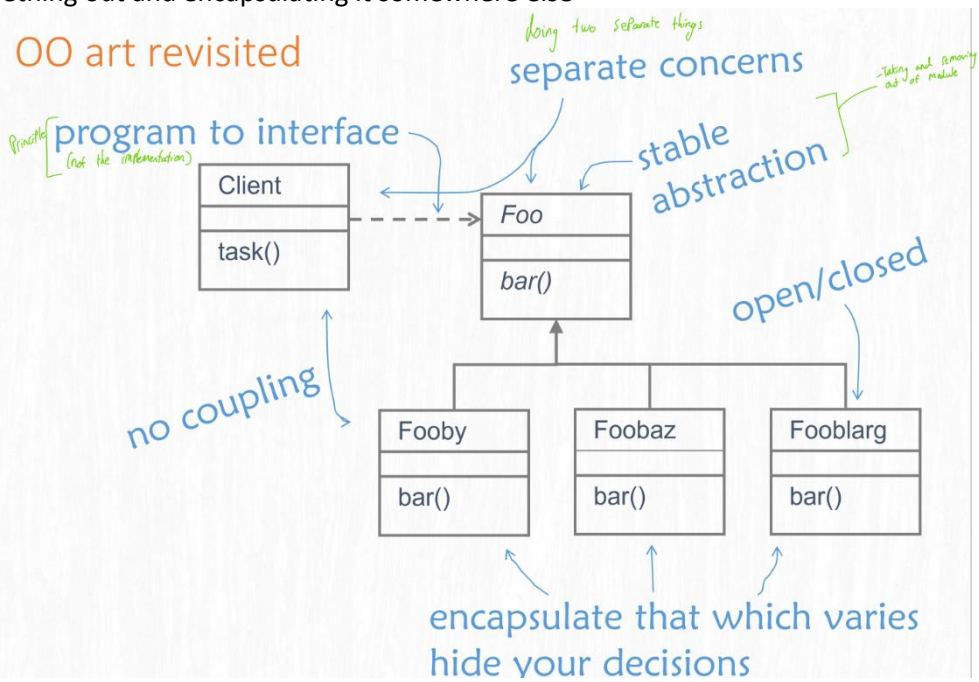
- **Encapsulation**
  - Means drawing a boundary around something
  - Means being able to talk about the inside and the outside of it
  - Is a programming construct that can be made used to hide information
- **Information Hiding**
  - The idea that a design decision should be hidden from the rest of the system to prevent unintended coupling
- Encapsulation is a programming language feature whereas Information hiding is a design principle
- Information hiding should inform the way you encapsulate things, but it doesn't **have** to
- They definitely aren't the same thing

15.1.2 Encapsulation leak

- Happens when you pass a modifiable object through a getter, allowing the client to modify the object when it shouldn't
- Solution: pass back an unmodifiable object instead

15.1.3 Abstraction

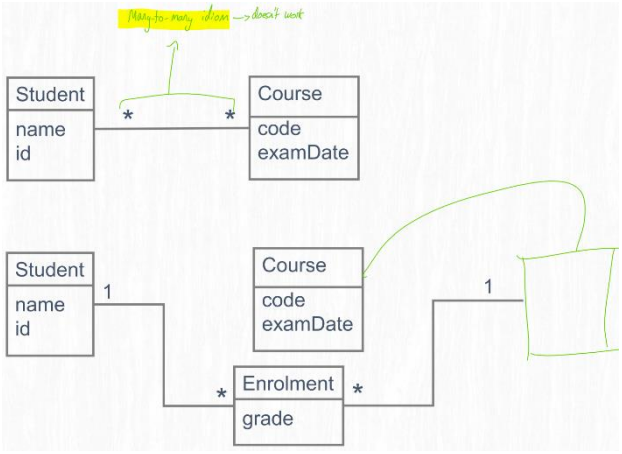
- Taking something out and encapsulating it somewhere else



15.2 DATA & BEHAVIOUR MODELLING

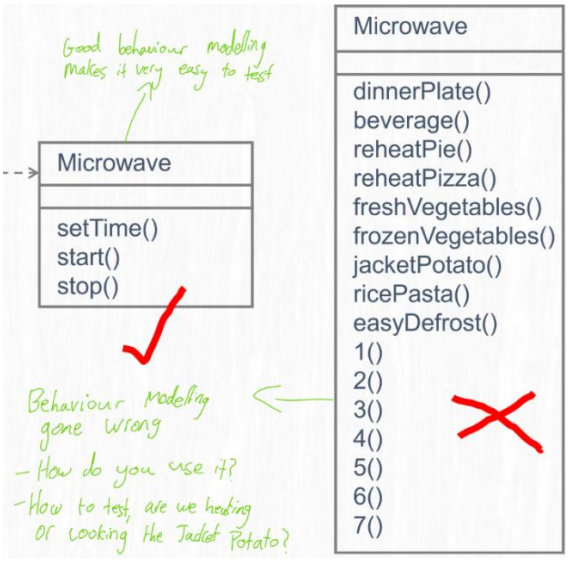
15.2.1 OO is data modelling (i.e. Address of a person/student)

- Focus on data internal to object
- Figure:
  - Top: many-to-many idiom, doesn't work
  - Bottom: solution, bridging class/table



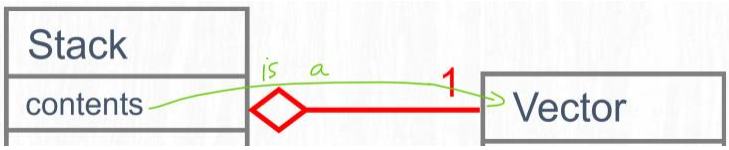
15.2.2 OO is behaviour modelling (i.e. Display() function)

- Focus on services to external world



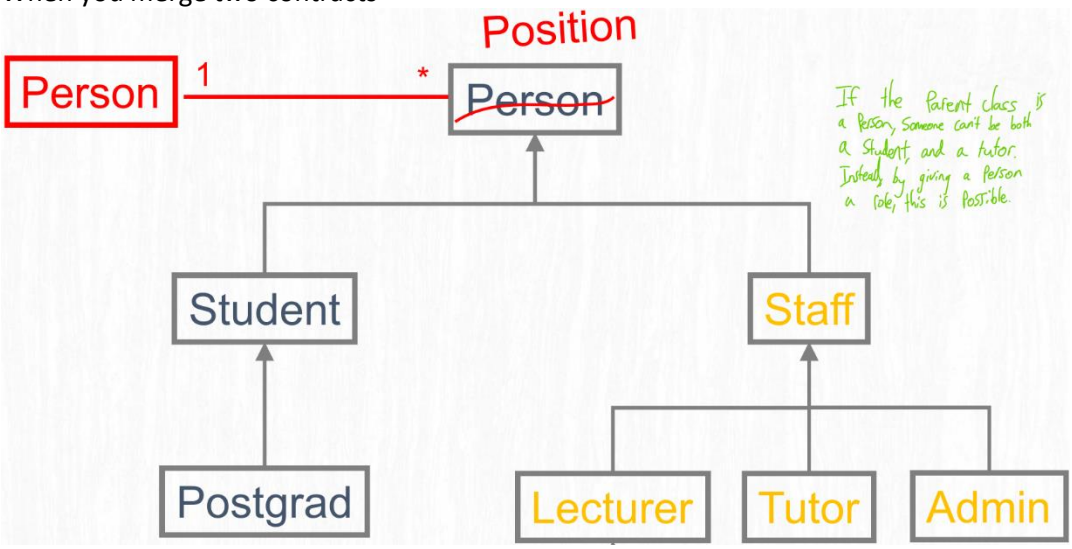
15.3 INHERITANCE

15.3.1 The dark side – inheritance mistakes

- **Inheritance for implementation**
  - When you have no intention of honouring the inherited contract
  - The Vector class implements a growable array of objects
  - Like an array, it contains components that can be accessed using an integer index
  - The size of a Vector can grow or shrink as needed to accommodate, adding and removing items at a given index after creation
  - Stack is a subclass of Vector, but you shouldn't be able to index a stack, or insert an element at a given index
  - Instead Stack could have a (private) variable inside of it say, called "contents", this variable could be a Vector, and that can be the underlying data structure for a stack
  - Then you can control how items are added and removed from the Stack's *contents* (a Vector), and hence the Stack can properly conform to how a stack should add, and doesn't have to break to contract of Vector
  - 

  - Principles:
    - Favour composition over inheritance
    - Hide your decisions

- **"Is a-role-of"**
  - When you merge two contracts



- **The "becomes" problem**
  - When you switch contracts
  - Inheritance isn't dynamic, you cannot change the type of child class on the fly
  - Example – both subclasses of Student: EligibleStudent becomes an IneligibleStudent
    - Solution, instead have variable in Student class like isEligible (boolean)
- **Over-specialisation**
  - When a contract is more specific than necessary
  - Happens when you program to them implementation instead of the interface
    - Example: `ArrayList<Object> list = new ArrayList<>()`
    - Instead of: `List<Object> list = new ArrayList<>()`
  - Don't need to be so specific
  - Related principles:
    - Hide your decisions
    - Program to the interface not the implementation
- **Violating the Liskov Substitution Principle**
  - Breaking the contract
  - If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T.
    - In plainer English: if S is a subtype of T, then objects of type T may be replaced with objects of Type S without (negatively) altering the contract
  - For example: if a square is a subtype of rectangle, then the square breaks the Liskov Substitution Principle since a square's contract is more restrictive than a rectangle's
  - For more details see design principles
- **Changing the super class contract – Principle: If it can change, it ain't inheritance**

15.4 DESIGN BY CONTRACT

15.4.1 Preconditions and postconditions

- **Preconditions**
  - Before calling a service (a method), client checks everything is ready
  - An action must be completed before calling a service to ensure that the service performs as expected
  - Loosening preconditions is okay – you are requiring less from the client (require no more, promise no less)
- **Postconditions**
  - After calling a service, server promises it has done its job
  - The server promises to give a certain set of things to the client
  - Tightening postconditions is okay – you are giving more to the client (require no more, promise no less)
- Invariant
  - Something the service (method from server) promises will always be true
  - You can tighten invariants

15.4.2 Documenting contracts

```
public class Dictionary {
   …

  /** Inserts value into the dictionary at the given key.
   * A later call to lookup(key) will return the value.
   *
   * Preconditions:   The dictionary is not full.
   *                  The key is not null or an empty string.
   *                  (It is OK if the dictionary already has an entry
   *                  at this key; the new value replaces the old one.)
   * Postconditions: The dictionary contains the value indexed by the key.
   */
  public void add(String key, Object value) {
  ….
  }
```

15.4.3 Guidelines

- **No preconditions on queries**
  - It should be safe to ask a question
- **No fine print**
  - Don't require something the client cant determine, i.e. preconditions should be supported by public methods
  - It is OK to have postconditions that the client cannot verify
- **Use real code wherever possible**
  - Better to say "!isEmpty()" than "the stack must not be empty"
- **Can't show all semantics in code (use English)**
  - Example: "pop() returns last pushed object"
- **No hidden clauses**
  - The preconditions are sufficient and complete, and the client doesn't need to do anymore to get the function to correctly work
- **No redundant checking**
  - Don't do defensive programming (checking input variables etc)
  - Don't check that preconditions are satisfied, trust that the contract is being adhered to

15.4.4 Contract inheritance

- **Contracts are inherited (extends another contract)**
  - Preconditions can be loosened (expect less from the client)
  - Postconditions can be tightened (give more to the client)
- **Principal: Require no more, promise no less**
- **Inheritance**
  - Given Bar extends Foo, we previously said that a Bar is a Foo
  - We now say a Bar conforms to the contract of Foo

15.4.5 Formal support for contracts

- **Java interfaces are contracts**
- **Eiffel**
  - Keywords in method declarations, require (precondition), and ensure (postcondition)
- **Support in java**
  - Contracts for java
  - Jcontracter
  - Java Modelling Language (JML)
  - Oval
  - Valid4j

16 CODE AND TEST SMELLS

- Two approaches to code smells
  - **Pragmatic**
    - Have a look at the discovered code smells and decide whether it is actually a problem
    - Or if the code smell is worth the time to remove
  - **Purist**
    - Always fix code smells, no matter what

16.1 TYPES OF CODE SMELL

16.1.1 Long method smell

- Issue: methods are too long
- This is a problem because
  - Method is probably doing too many things
  - Violates the Single Responsibility principle
- Solution(s)
  - Break into multiple methods – **Extract method** (this is important vocabulary)

16.1.2 Large class smell

- Potential issues
  - Class could be overly complex
  - Might be violating the Single Responsibility principle
  - Might be a God class
  - Is this a single point of failure?
- This is a problem because
  - A class with too much code is prime breeding ground for duplicated code, chaos and death
- Solution(s)
  - Break class into smaller cohesive classes – **Extract class**
  - **Extract interface**

16.1.3 Long parameter list

- Issue: method takes too many parameters
  - How complex is the method? Does it need all the parameters?
  - Does the method have a Single Responsibility?
  - Is it a God method?
  - Is coupling too high?
- Solution(s)
  - Introduce parameter object
  - Preserve whole (parameter) object
  - Replace parameter with method call
- You must be careful that you do not introduce more dependencies

16.1.4 Duplicated code

- Code should be written “once and only once” (ideally)
- Duplicated code can look different but is solving the same solution
    - This is called an Oddball solution: a different way of solving the same problem
- Issues
    - How many places do we need to change if we want to make a change?
    - There is not a single place of truth
- Why is this occurring?
    - Due to “copy and paste programming”
    - Due to having multiple programmers
- Solution(s)
    - **Extract method**

16.1.5 Dead, unreachable, deactivated, commented out code

- **Dead code**
    - Source code that might be executed, but the result is never used
    - Unreferenced variables/functions aren’t called are **not** dead code
        - These are automatically removed via compiler/linker
- **Unreachable code**
    - Code you can’t get to
        - Often in a switch statement
    - Could be after a return statement
    - Makes the code harder to read and takes up more memory, might also take up more cache
- **Deactivated code**
    - Executable code that isn’t execute now/anymore (as it is/was part of a different version/release)
- **Commented code**
    - Code that has been commented out
    - This can be misleading
    - It is difficult to read (and maintain)
    - Why was it commented out? Is it needed? Can we delete it?

16.1.6 Switch statement smell

- Issue: Large switch/if statement
    - Increases conditional (and cognitive) complexity
    - Object-oriented programming should only rarely have switch statements (or large if conditional blocks)
- Solution(s)
    - If you are switching on types: replace conditional with polymorphism
    - If you are switching on type code: replace type code with subclasses
        - Type code example: ENUM (studentType = ENROLLED)
        - This is an issue because of the ‘becomes’ problem, one type can become another
    - Are you often checking against null? Introduce a null object
- If the switch performs simple actions, then you can leave it

16.1.7 Comments

- Issues
  - Integrity: are the comments up to date?
  - Do we need the comment? Is the code readable (with/without them)?
- This is a problem because
  - Keeping comments up to date takes up the time and energy of developers
- Solution(s)
  - If an expression is difficult to understand, **extract variable**
  - If code if difficult to understand, **extract method**
  - Possibly, rename method for clarity

16.1.8 Type embedded in name

- Issue: strFirstName, intDateOfBirth, intGetAge()
- This is a problem because
  - What happens if you change the type?
    - We should be hiding our decisions
- Solution(s)
  - Rename method
  - Rename variable

16.1.9 Uncommunicative name

- Issue: a, b, convertFromStrToDblAndCalcGST()
  - Is the name descriptive?
  - Is that name too hard to read?
  - Is the name succinct
- Naming should be consistent (**Inconsistent code smell**)
  - E.g. open/close, read/write
- Solution(s)
  - Rename method
  - Rename variable

16.2 MORE TYPES OF CODE SMELL

16.2.1 Speculative generality

- Issue: making general solutions because you are speculating or anticipating what we might need in the future
- This is a problem because
  - You are **over-engineering** your solution (principle)
  - You should not be speculating about tomorrow's problems
    - This is different to knowing about problems in the future
- We must balance our solution with planning for extensibility (**open/closed** principle)
- Solution(s)
  - Collapse hierarchy
  - Rename method
  - **Inline class** (opposite of **Extract class**)

16.2.2 Inappropriate intimacy

- Issue: One class using the internal fields/methods of another class
- This is a problem because
  - It violates **information Hiding**
  - It has high coupling
  - Classes should know as little as possible about each other (this is effectively reworded coupling)
- Solution(s)
  - Move function/field (to class that uses it, only works if the other class truly doesn't need those parts)
  - Change bi-directional associated to unidirectional
  - Replace superclass with delegate

16.2.3 Indecent exposure

- Issue: a class exposing fields/methods unnecessarily (exposing its internals)
- This is a problem because
  - It violates **information hiding**
- Solution(s)
  - Minimise public exposure (private methods/fields)
  - **Hide your decisions** (principle)
  - **Encapsulate that which varies** (principle)

16.2.4 Feature envy

- Issue: a method making extensive use of another class, i.e. they are envious of another class' methods and wish they had them
- Cohesive elements should probably be in the same module/class
- Solution(s)
  - Move method
  - Extract method

16.2.5 Message chains

- Issue: Long sequences of method calls or temporary variables (a long message stack)
- This is a problem because
  - Increases complexity
  - Changes in any relationship (in the chain) can cause large cascading changes
  - Causes dependency between classes in the chain

16.3 TYPES OF TEST SMELL

16.3.1 Hard to test code

- Issue: code is difficult to test
  - Code might be highly coupled
  - It could be asynchronous code
  - Some types of code are just difficult to (automatically) test, e.g. GUI code
- Solution(s)
  - Need to make code more amenable to testing. This is a big topic in and of itself and is often hard to do

16.3.2 Obscure test

- Issue: the test is difficult to understand (at a glance)
    - Could be caused by too much or too little information in the test
        - Eager test: test too much functionality
        - There could be irrelevant information
        - There could be hard-coded test data
        - There could be indirect tests
- This is a problem because
    - There may be issues understanding what behaviour a test is testing/verifying
- Solution depends on what the issue is

16.3.3 Production bugs

- Issue: we find too many bugs during formal testing (i.e. during pipeline or similar), or in production
- Too many production bugs means that bugs are getting to the customer
    - Tests are not sufficiently finding bugs in code
- This is a problem because
    - It might indicate poor test quality
        - Are the tests good?
        - Do the tests have sufficient coverage?
        - Are the tests actually testing or just checking?
    - Automated tests might not be covering all possibilities
    - Tests might be buggy
- Solution depends on what the issue is

16.3.4 High test maintenance cost

- Issue: tests need to be modified often
- This is a problem because the tests might be
    - Too complex
    - Too obscure
    - Not adhering to **Single Responsibility** (principle)
    - Duplicated
    - Too coupled to SUT (System Under Test)
- Solution depends on what the issue is

16.3.5 Fragile tests

- Issues
    - The interface is too sensitive
        - E.g. recorded tests for GUIs
        - Bad UI testing frameworks
        - The test is sensitive to UI resizing
    - The context of the test is too sensitive
        - The test passing is very sensitive to the data in the system, even simple/small changes can break the test
        - The test is sensitive to the time/date or server state
    - The test fails to compile/run when the system is changed in a way that do not affect the part of the system the test is exercising
- This is a problem because fragile tests increase the cost of test maintenance

16.3.6 Erratic tests

- Issues
    - Sometimes the test fails, sometimes it passes
    - Test failing for no reason (no changes to code/test)
    - Tests fail in one environment but not another (OS, development, production)

17 DESIGN PATTERNS

17.1 WHAT IS A DESIGN PATTERN?

- It is distilled wisdom about a specific problem that occurs frequently in Object-Oriented design
- It is a reusable design micro-architecture
- "A solution to a problem in context"
    - Solution – a canonical design form that someone can apply to resolve these forces
    - Problem – a set of forces (goals and constraints) that occur in this context
    - Context – a recurring set of situations in which the pattern applies
    - What are the forces?
        - **Correctness** – completeness, type safety, fault tolerance, security, transnationality, thread safety, robustness, validity, verification…
        - **Resources** – efficiency, space, "on-demand-ness", fairness, equilibrium, stability…
        - **Structure** – modularity, encapsulation, coupling, independence, extensibility, reusability, context dependence, interoperability…
        - **Construction** – understandability, minimality, simplicity, elegance, error-proneness, co-existence with other software, maintainability, impact on processes, teams, users…
        - **Usage** – ethics, adaptability, human factors, aesthetics, economics…
- The core of a design pattern is a simple class diagram with extensive explanation
    - It documents an elegant, widely accepted way of solving a common OO design problem
- Patterns are discovered, as opposed to written/invented
- Design patterns are
    - Reusable chunks of good design
    - A solution to a common problem
    - Not a perfect solution
    - Essential knowledge for OO developers
- The Gang of Four (GoF)
    - Produced the original design patterns book
    - Documented 23 patterns classified as
        - Creational patterns
        - Structural patterns
        - Behavioural patterns

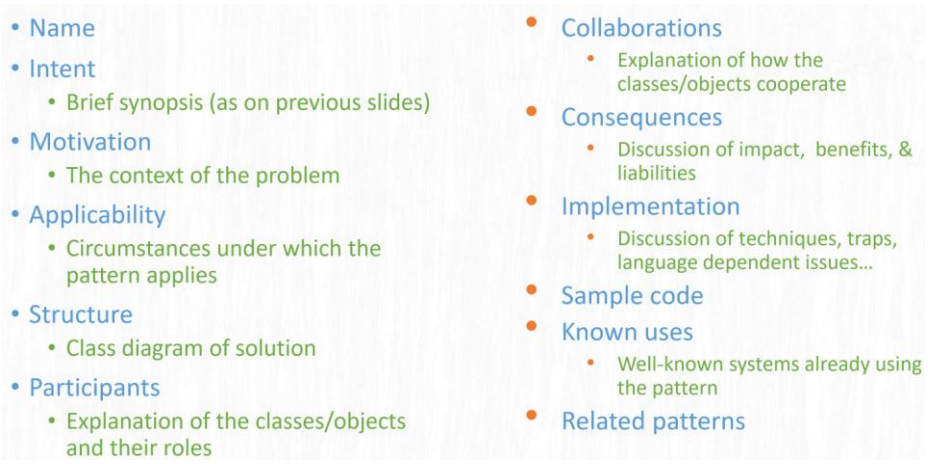
17.1.1 Verification and Validation (V&V)

- Both are used to make sure that a product or service meets the requirements/specifications and fulfils its intended purpose
- **Verification:** have we built the software right
    - The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
    - Unit tests, reviews, walkthrough, metrics analyses
- **Validation:** have we built the right software
    - The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements
    - Internal: acceptance tests, Blackbox, Whitebox
    - External: client tests

17.1.2 Resolution of forces

- "A pattern should represent a kind of equilibrium of forces"
- It is impossible to prove a solution is optimal; arguments must be backed up with
    - **Empirical evidence for goodness** – the rule of 3: don't claim something is a pattern unless you can point to three independent usages
    - Comparisons – with other solutions, (possibly includes failed ones)
    - Independent authorship – not written solely by their inventors
    - Reviews – by independent domain and pattern experts

17.1.3 Documenting Patterns (Gang of Four style)



17.2 Overview of the different design patterns

17.2.1 Creational patterns

- **Abstract Factory** – provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Builder** – separate the construction of a complex object from its representation so that the same construction process can create different representations
- **Factory Method** – define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.
- **Prototype** – specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype.
- **Singleton** – ensure a class only has one instance and provide a global point of access to it

17.2.2 Structural patterns

- **Adapter** – convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
- **Bridge** – decouple an abstraction from its implementation so that the two can vary independently
- **Composite** – compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly
- **Decorator** – attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality
- **Façade** – Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use
- **Flyweight** – Use sharing to support large numbers of fine-grained objects efficiently. Proxy Provide a surrogate or placeholder for another object to control access to it

17.2.3 Behavioural patterns

- **Chain of Responsibility** – avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it
- **Command** – encapsulate the request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations
- **Interpreter** – given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in that language
- **Iterator** – provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- **Mediator** – define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently
- **Memento** – without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later
- **Observer** – defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- **State** – allow an object to alter its behaviour when internal state changes. The object will appear to change its class
- **Strategy** – define a family of algorithms, encapsulate each one, and make them interchangeable, Strategy lets the algorithm vary independently from clients that use it
- **Template Method** – define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure
- **Visitor** – represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates

17.3 ITERATOR

17.3.1 Problem

- Sequentially access the elements of a collection without exposing implementation
- Allow for different types of traversals (e.g. different ordering, filtering)
- Allow multiple traversals at the same time

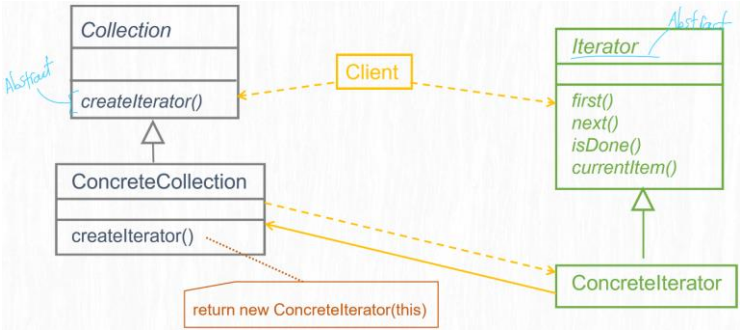
17.3.2 Solution

- Move responsibility for traversal from the collection into an Iterator object. It knows the current position and traversal mechanism
- The collection creates an appropriate iterator

17.3.3 Principles

- **Introduced/reinforced**
  - Encapsulate that which varies

17.3.4 UML Diagram



17.4 Singleton

Ensure a class has only one instance and provide a global point of access to it

17.4.1 Problem

- Some classes should only have one instance
- How can we ensure someone doesn't construct another?
- How should other code find the one instance?

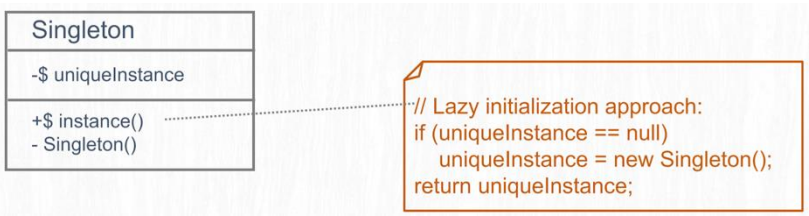
17.4.2 Solution

- Make the constructor private
- Use a static attribute in the class to hold it's one instance (of the class itself)
- Add a static getter to the instance
    - If the class is already created (and hence in the attribute), return it
    - Otherwise, create a new instance and return it

17.4.3 Issues

- **Principle violated**
    - The cohesion good, coupling bad principle as there is tight coupling between objects using a singleton
- Goes against OO design, no object is instantiated
- Can be a god class

GOF: Singleton
Singleton
uniqueInstance
instance()
Singleton()



17.5 Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate

- It is common to have more than one factory method

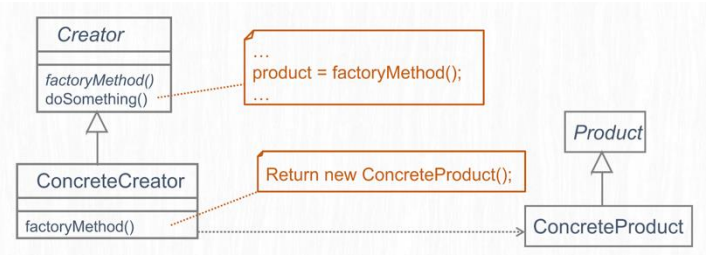
17.5.1 Problem

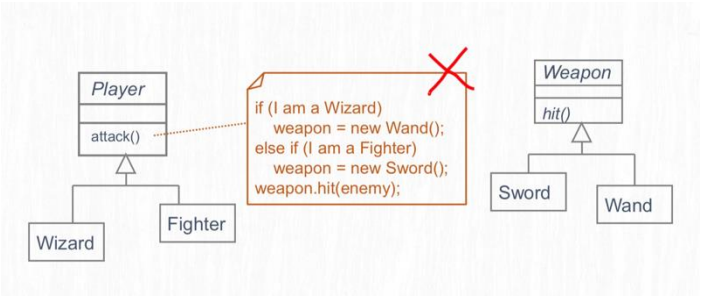
- Normally, code that expects an object of a particular class does not need to know which subclass that object belongs to
    - Except when you create an object, you need to know its class. Hence the "new" operator in Java increases coupling
- Need a way to create the right kind of object, without knowing its exact class

17.5.2 Solution

- Move the "new" operator call into an abstract method (can be parameterised)
- Override that method to create the right subclass object

GOF: Factory method
Creator
factoryMethod()
doSomething()
ConcreteCreator
Product
ConcreteProduct
Creator -- Product Relationship
ConcreteCreator -- ConcreteProduct Relationship





17.6 Template Method

Template Method allows subclasses to redefine certain steps of an algorithm without changing the algorithm's structure

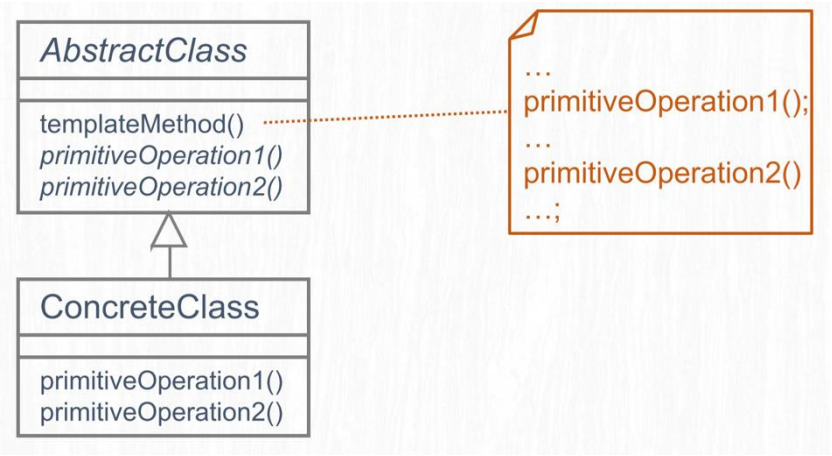
17.6.1 Problem

- Implement the skeleton of an algorithm, but not the details

17.6.2 Solution

- Put the skeleton in an abstract superclass and use subclass operations to provide the details

GOF: Template Method
AbstractClass
templateMethod()
primitiveOperation1()
ConcreteClassA
Client ???
Client -- AbstractClass relationship ???



17.7 Abstract Factory

Provides an interface for creating families of related or dependent object without specifying their concrete classes

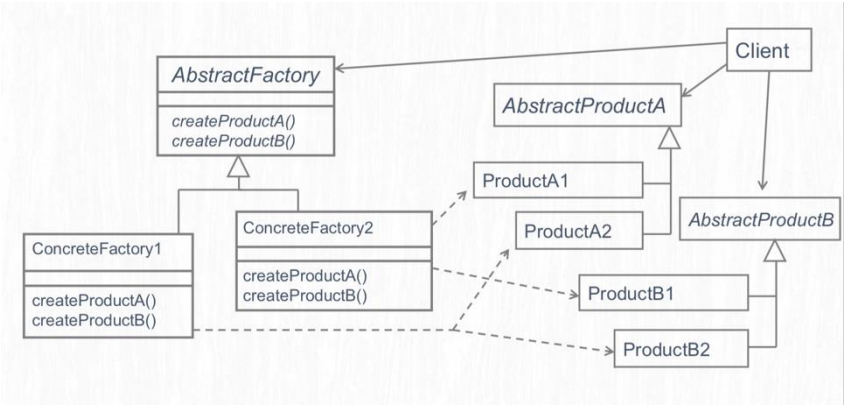
17.7.1 Problem

- Same as factory method, but want to create whole families of related objects

17.7.2 Solution

- Move all the factoryMethods into factory classes

GOF: Abstract Factory
Client
AbstractProductA
ProductA1
ProductA2
AbstractProductB
ProductB1
ProductB2
AbstractFactory
createProductA()
createProductB()
ConcreteFactory1
ConcreteFactory2
Client -- AbstractProduct Relationship
Client -- AbstractFactory Relationship
ConcreteFactory -- Product Relationship



17.8 Strategy

Define a family of algorithms, encapsulate each one and make them interchangeable, Strategy lets the algorithm vary independently from clients that use it

17.8.1 Problem

- Change and object's algorithm dynamically, rather than through inheritance

17.8.2 Solution

- Move the algorithms into their own class hierarchy

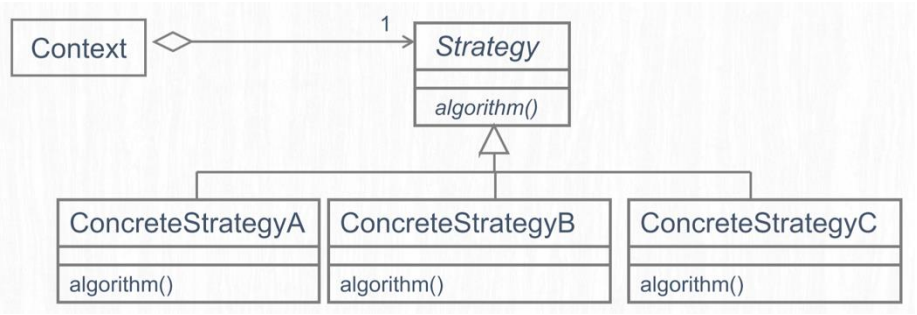
17.8.3 Principles improved/introduced

- Open/closed principle
- Encapsulate that which varies
- Favour composition over inheritance

17.8.4 Notes

- Contexts know different strategies exist (because they must choose one)
- Strategy needs access to relevant context data

Context
Strategy
algorithm()
ConcreteStrategyA
Context -- Strategy Relationship



17.9 OBSERVER

Defines a one-to-many dependency between objects so that when one object changes state, all its dependencies are notified and updated automatically

17.9.1 Problem

- Separate concerns into different classes, but keep them in sync
- Avoid tight coupling

17.9.2 Solution

- Separate into Subject and Observers – can have many observers for one subject
- The Subject knows which objects are observing it, but it doesn't know anything else about them
- When the Subject changes, all Observers are notified

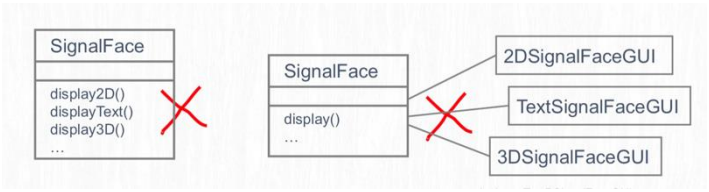
17.9.3 Benefits

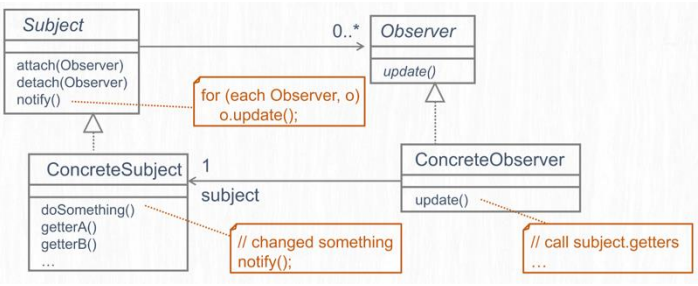
- Reduces coupling as we just broadcast change and whoever is listening can act as they wish
- Often used to decouple the GUI from the backend

17.9.4 Notes

- Changes are broadcast to all observers
    - It is up to each observer to decide if it wants to do something about the change
- Observers don't know about each other
    - They are unaware of the true cost of changes to the Subject
    - Complex dependencies and cycles are possible (and should be avoider)
- Observers aren't told what changed
    - They use getters to find out what changed
    - This can be a lot of work
    - A variant of this pattern allows the update method to contain details of what has changed (more efficient, but tighter coupling)
- The Subject should call notify() only when it is in a consistent state (at the end of a transaction)
    - Beware of subclasses that call base methods and the base class does the notify()

GOF: Observer
Observer
Update()
ConcreteObserverA
Subject
attach(Observer)
detach(Observer)
notify()
ConcreteSubjectA
doSomething()
getterA()
Subject -- Observer relationship
Concrete Subject -- Concrete Observer Relationship

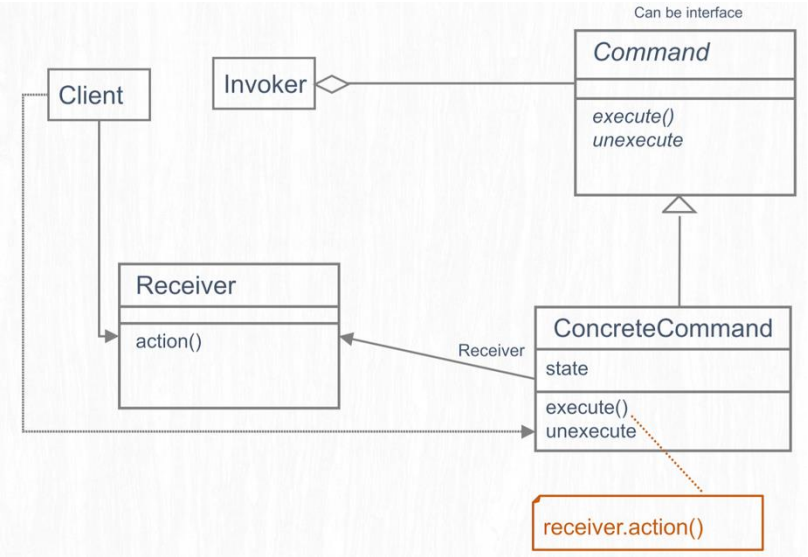




17.10 COMMAND

Encapsulate the request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

Invoker
Command
execute()
unexecute()
ConcreteCommandA
state
Receiver
action()
Client
Invoker -- Command Relationship
Receiver -- ConcreteCommand Relationship



17.11 COMPOSITE

17.11.1 Problem

- When objects contain other objects to form a tree (i.e. a containment hierarchy), how can client code treat the composite objects and the atomic objects uniformly

17.11.2 Solution

- Create an abstract superclass that represents both composite and atomic objects

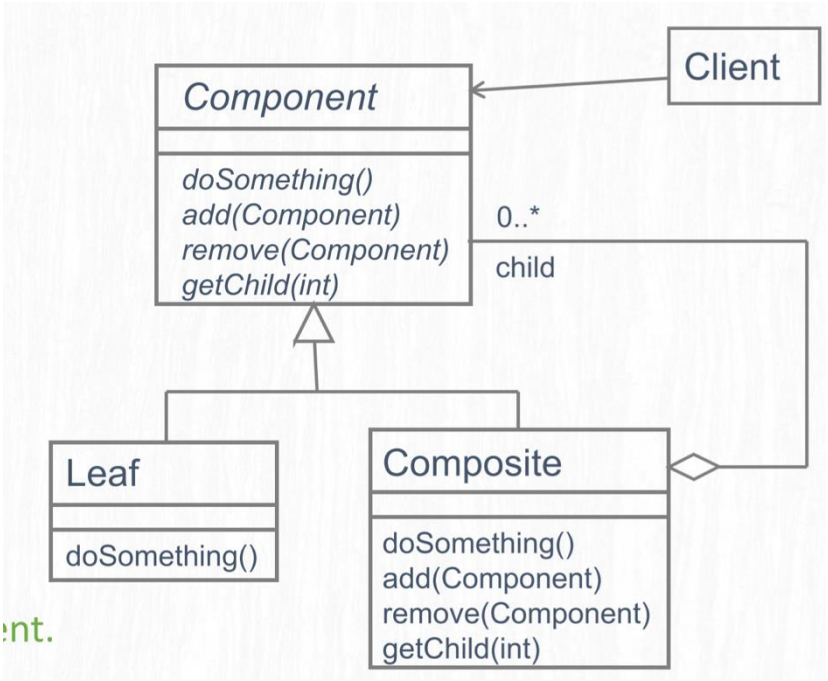
17.11.3 Notes

- Common for child to know parent
- Easy to add new components
- Can make containment too general

17.11.4 Principles improved

- Open/closed principle
- Stable abstraction

GOF: Composite pattern
Client
Component
doSomething()
add(Component)
remove(Component)
getChild() (less common nowadays)
Leaf
Composite
Client -- Component Relationship
Composite -- Component Relationship



17.12 DECORATOR

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality

17.12.1 Problem

- When objects contain other objects to form a tree (i.e. a containment hierarchy), how can client code treat the composite objects and the atomic objects uniformly

17.12.2 Solution

- Use aggregation instead of sub classing

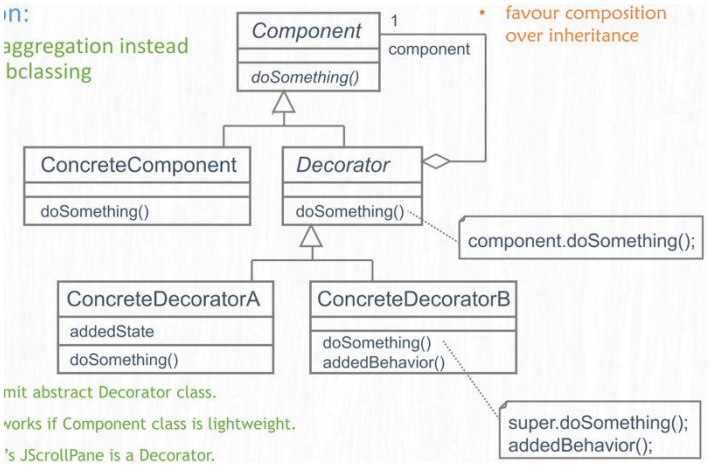
17.12.3 Notes

- Can omit abstract Decorator class
- Only works if Component class is lightweight
- A solution to the “becomes” problem
- Used to add extra methods/state to subclasses

17.12.4 Principles improved

- Open/closed principle
- Encapsulate that which varies
- Favour composition over inheritance

GOF: Decorator pattern
Component
doSomething
ConcreteComponent
Decorator
ConcreteDecoratorA
addedState
addedBehaviour()
Component – Decorator Relationship
ConcreteComponent – Decorator Relationship ???



17.13 The Dark Side of Design Patterns (Anti-patterns)

- An anti-pattern is a pattern that tells you how to go from a problem to a bad solution (it is a poor solution to a problem)
- An anti-pattern looks like a good idea but has some fatal flaw which causes it to be a poor solution
- Examples
    - Cash cow
    - Groupthink
    - Hero culture
    - Over user of pattern
    - Spaghetti code
    - Yet another meeting will solve it

17.13.1 Analysis paralysis

- What: getting stuck in the analysis phase
- Seems like a good idea to get all the requirements to plan before design
- Desire to lower risk
- To fix
    - Be agile
    - Have milestone dates

17.13.2 Copy paste programming

- What: a different context requires a similar solution. Copy/paste it from the original place and make the small changes
- This usually creeps in over the entire development period
- Duplicate code
    - Multiple changes necessary
    - Difficult to maintain and debug
    - Trust decreases
- To fix
    - Extract solution into its own method
    - Use parameters to shape method to the required context

17.13.3 Bike shedding (aka Yak Shaving)

- Parkinson's law of triviality
    - Disproportionate weight/time to trivial issues
- E.g. time/energy spent
    - Discussing low-level implementation details at stand-ups
    - Colour of GUI, button placement
    - Working on chores vs implementation
- Each person feels like they need to cooperate and add value
- The discussion is about a known and familiar context
- To fix
    - Have an agenda
    - Timebox
    - Explicitly specify the object of the timeboxed session
    - Document decisions

17.13.4 God class

- What: one class that has several responsibilities
- Often creeps in over the whole development
- A place to start processes. A central controller
- Multiple responsibilities makes it
    - Harder to test
    - Harder to debug
    - Difficult to maintain
- Look for "manager", "controller", "driver"
- To fix
    - Move responsibilities into their own class – distributed control

17.13.5 Death march

- It is obvious that the commitment will not be met for the sprint/release but the team keeps driving without addressing the issue
- Seems like a good idea because you have "faith" that if you worked harder, you will be able to complete it even though the velocity shows you can't
- Usually management driven
- To fix
    - Do not over commit
    - If you see you won't make the commitment, talk to the PO and remove stories

17.13.6 Singletons are evil

- Tight coupling between objects
- Carries state that lasts as long as the program lasts
- Becomes a god class
- Goes against OO – no object is instantiated

17.13.7 Poltergeist classes (aka Gypsy wagon, Proliferation of classes)

- What: classes that have no responsibility of their own
- Occurs when trying to add modularity to a program
- Life cycle is very brief
- Names usually: _manager, _controller
- E.g. a controller class that calls methods from other classes and nothing else
    - Extra complexity
    - Harder to understand
    - Harder to maintain
    - Redundant navigation paths
    - Stateless change
    - Difficult to test

17.13.8 Magic numbers, strings

- What: numbers/strings that have meaning but meaning is not explicitly states
- Problems
    - Difficult to understand
    - Could have multiple occurrences
    - Harder to change
- To fix
    - Use named constants/Enums/etc
- Code should be self-documenting

18 TERMS

18.1.1 Polymorphism

In terms of object-oriented programming, polymorphism refers to the ability of a programming language to process objects differently dependent on their type or their class. Furthermore, it is the ability to redefine methods for derived (extended) classes.

18.1.2 Eering

Avoiding error prone constructs such as implicit variables, or encapsulating “nasty” stuff

18.1.3 Regression testing

Ensuring that all tests that exist for a system will pass when new features are added. Ensures that no new feature breaks the current system

18.1.4 User testing

Main usages that a user will be performing when using the system are tested, from the UI to the data layer

18.1.5 Smoke testing

A non-exhaustive set of tests that ensure the most important functions of a system. This set of tests are usually used to determine if a system is ready for more exhaustive testing or if it is too unstable

18.1.6 Reflection testing

Reflection testing should be used as a last resort as it requires inspecting private methods and variables, which is indicative of untestable code and should be redesigned.

19 Design Principles

19.1.1 Separate interface from implementation

19.1.2 If you override equals(), you must also override hashCode()

19.1.3 Do not assume, use the documentation

19.1.4 Always think about the client, how will this implementation be used

19.1.5 Cohesion: good. Coupling: bad.

- Cohesion is intra-module, measures how much a particular module does just one thing
- Coupling is inter-module, measures how much a particular module is related to other module(s), some coupling is required

19.1.6 Separation of concerns

- Kind of equal to coupling: bad
- Getters/setters (asking part of "tell, don't ask)

19.1.7 Keep related data and behaviour together

- Separation of concern and this principle are both correct and conflicting, which one you use depends on the context of the situation
- Kind of equal to cohesion: good
- Not using getters/setters (tell, don't ask)

19.1.8 Information hiding

- Information hiding is a design decision
- Encapsulation is a programming construct that can be used to hide information
- The idea that a design decision should be hidden from the rest of the system to prevent unintended coupling

19.1.9 Program to the interface, not the implementation

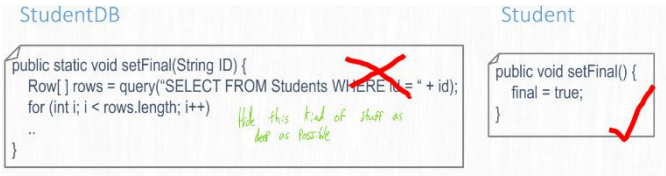
- Program what the client are going to be using, not the implementations
- No encapsulation leaks
- Information hiding

19.1.10 Encapsulate that which varies

- Find the solid bits and make them the framework of your program
- Find the wobbly bits and hide them away

19.1.11 Hide your decisions

- If you choose it, hide it away
  - Data representations
  - Algorithms
  - IO formats
  - Mechanisms (inter-process communication, persistence)
  - Lower-level interfaces
- Find the solid bits and make them the framework of your program
- Find the wobbly bits and hide them away



19.1.12 Make stable abstractions

- Abstracting: taking something out and encapsulating it somewhere else
- Usually increases cohesion (good) and coupling (bad)

19.1.13 Open/closed principle

- Make your system open for extension, but closed for modification
- You should be able to add new functionality without modifying upper levels (which could cause issues with other clients – clients being things that us it, not people)

19.1.14 Do not over-engineer your solution

- Aim for simplicity
- All the other principles are aiming at simplicity
- KISS – Keep It Simple Stupid
- YAGNI – You Ain't Gonna Need It

19.1.15 Overarching principle: Leave the world in a better place than you found it

- Refactor vs re-engineer vs rewrite
- Refactor
    - Doesn't affect functionality
    - A controlled technique for improving the design of an existing codebase by applying a series of small behaviour-preserving transformations
    - By making many small changes, you avoid having a broken system whilst refactoring
- Re-engineer
    - Changing how the program interacts with itself
    - Changes functionality (especially when changing public methods)
    - Changing private methods is somewhere between refactoring and re-engineering, but probably closer to re-engineering
- Rewrite
    - Delete and redo a whole block of code or section of your program
    - Breaking something that is already working
    - Getting rid of work already done

19.1.16 Favour composition over inheritance

- See OO Design → Inheritance → The dark side → Inheritance for implementation (p.48)

19.1.17 Inheritance isn't dynamic || If it can change, it ain't inheritance

- We don't change inheritance on the fly
- Don't change the class of an object on the fly (don't change a superclass to a subclass when something changes, or a subclass to another subclass)
- The "becomes" problem

19.1.18 Liskov substitution principle:

- If we can see a parent, then we should be able to act as though the child adheres to the parent's contract
- i.e. we can use any child in place of the parent without the contract being broken

19.1.19 Single responsibility principle:

- Every module or class should have responsibility over only one part of the functionality
    - A class/module should not be doing multiple things
- Responsibility = reason to change
- It is an issue when one class/module knows (or does) too much
    - The more responsibilities your class has, the more you need to change it
- Typically, violations are found in controllers, main class, initialisers, etc

19.1.20 Interface segregation principle:

- No client should be forced to depend on methods/interfaces it doesn't use
- For example:
    - Having a large parent class that does a lot of different things, forcing a subclass to depend on all of those things regardless of whether it uses them
- To solve:
    - Split interfaces into cohesive, smaller, more specific interfaces
    - This also reduces coupling

19.1.21 Dependency inversion principle:

- Two parts:
    - High-level modules should not depend on low-level modules. Instead, both should depend on abstractions (e.g. interfaces)
    - Abstractions should not depend on details, details (i.e. concrete implementations) should depend on abstractions
- Reworded:
    - High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features
- Solving this gives benefits:
    - Allows for greater scalability and extensibility
    - Reduces coupling

19.1.22 SOLID: The first five object orientated design principles

- **S:** Single responsibility
- **O:** Open/close principle
- **L:** Liskov substitution principle
- **I:** Interface segregation principle
- **D:** Dependency inversion principle

19.1.23 Require no more, promise no less

- Require no more: pre-conditions
- Promise no less: post-conditions