# Week 2: Agile Team Organisation

Why Agile – Empirical process control

Agile Manifesto:

* **Individuals and interactions** over processes and tools
* **Working software** over comprehensive documentation
* **Customer collaboration** over contract negotiation
* **Responding to change** over following a plan

In practice:

* Small teams (3-12)
* Frequent meetings
  + Short, **structured** team meetings and informal communication
  + Small deliveries of features for review
  + **Customer** engagement
* Maintenance of **design and code quality** through continual testing, analysis, review, and refactoring
* Automation wherever possible
* Frequent review and change of **project objectives** and **priorities**
* Frequent measurement and review of **software processes** and adjustment to enhance performance

Methods:

* Xtreme Programming
* **Scrum**
* Lean
* Feature-Driven Development
* Crystal Clear
* Kanban
* …

Practices:

* Test-driven development
* Behaviour-driven development
* Planning poker
* Refactoring
* Pair programming
* Retrospectives
* Continuous integration and deployment

Risks of Agile Development:

* Lack of customer engagement
* Stakeholder conflict
* Complex contractual arrangements
* Loss of organisational memory
* Poor code quality
* Poor team coordination or cohesion

Not agile:

* Doing every single agile practice within a single project regardless of context
* Achieving flexibility through uncontrolled change to objectives or process

Scrum

* Basics
  + Focuses on the team, project and process management aspects of soft dev
  + Can be used as a wrap-around for other agile practices that cover technical concerns
  + Name derived from rugby
* Roles (in Scrum and Beyond)
  + **Scrum master** (coach/mentor)
  + **Product owner** (liaison)
  + Team Manager
  + Quality assurance manager
  + Toolsmith
  + Chief architect
  + UX designer
  + Dev
* Managing work
  + Organised into **Releases**, which are marked by **release planning** meetings
    - Longer projects will comprise multiple **releases**, each of several sprints
    - Identify the high-level set of features/improvements to be delivered in a release
    - Create a roadmap of **milestones** aligned with your customer priorities and the customer days
    - Populate the roadmap with key features from the backlog
    - Assume that these will be changed in the **sprint planning meetings**
  + Some teams begin a first sprint with a **project launch meeting**
    - Determines the major features to be delivered to a customer over a series of sprints
    - Understand customers’ long-term objectives and identify a **minimum viable product**
    - Decide on the goals for within the project course
    - Develop an initial set of **user stories**
    - Refine the user stories into **tasks** and populate a **backlog** of issues on GitLab
    - **Triage** items in the backlog to establish **cost estimates** and **priorities**
  + Release = 2+ sprints
  + Sprint = 1-3 weeks
  + Sprint starts with a **planning meeting** and ends with a **review meeting** and **retrospective**
    - **Sprint planning meeting**
      * Decide on the main goal for the sprint
        + Major new feature set
        + Performance/other enhancements
        + Bugs to be corrected
        + Code to be improved
      * Select tasks from the **backlog** on your issue tracker that match the goal
      * Ensure all selected issues are sufficiently detailed
        + Cost estimates
        + Priorities
        + Assignees
      * Ensure that task cost estimates are within the **project** **velocity**
        + **Diagram

          Description automatically generated**
  + **Stand-ups**/scrums take place throughout the sprint
    - (For TP(H), at least once a week)
    - Facilitated by the **scrum master**
    - Each team member is asked:
      * What did you accomplish last week?
      * What are you working on now?
      * Do you have any blockers?
    - Should not last more than 10 mins
    - Try documenting the stand-up on the wiki at the start
    - Experiment with more frequent stand-ups, especially at start
* Workflow
  + **Diagram

    Description automatically generated**

Managing Delays – Deadlines or Feature-Driven?

* Deadlines
  + Ask for extension
    - Risky because feedback from customer is delayed
* Feature-Driven
  + Reduce the feature set
    - Better because more feedback
* Adding more resources?
  + Ideal project:
    - Shape

      Description automatically generated
  + Factoring in training
    - Shape

      Description automatically generated with low confidence
  + Unpartitionable projects:
    - 
  + Complex
    - Shape

      Description automatically generated with low confidence
  + Conclusion – not always

Reviewing a Sprint

* Review the progress on the **project** in the **Sprint Review Meeting**
  + **Deliver** and **demonstrate** new version of software developed over the course of the sprint
  + Summarise progress compares with planned work for a sprint
    - Additional work completed
    - Missed features
  + Explain reason for deviations from plan
  + Identify new features to be added to the system
* Review the software team’s **process** in the **Retrospective Meeting**

**Summary:**

* **Agile methods manage project risk through frequent reviews and small adjustments to project objectives and process**

# Week 3: Change Management

Change management is fundamental to many other SE practices:

* Compilation
* Dependency management
* Testing
* Code reviews
* Static analysis
* Refactoring
* Deployment and rollback
* Bug reporting and triage
* …

Change Control items

* Any items directly edited by the dev team
* Yes:
  + Source code files
  + Build scripts or config files
  + User documentation
  + Requirements specifications
  + Version control tool configurations
* No:
  + Compiled binaries
  + Third-party libraries
  + Compiled document formats (PDF, PS)
  + Auto-generated source code files
  + Client-side IDE config
  + Log files
* Issues:
  + Bad Reasons for Storing non-Control Items in an SCM:
    - “The source code takes too long to compile, so we need to store the binary”
    - “Compilation can’t be automated; someone has to press a button on the GUI”
    - “Third party libraries aren’t available in a public release repository”
  + Solve through *Release and Dependency Management*
    - Break the code into smaller modules with separate compilation pipelines
    - Agree and configure a release management process for each module using a release repository
    - …

Revision/Version/Source Change/Control Systems/Repositories (Version Control Systems)

* Diagram

  Description automatically generated
* Centralised vs. Distributed
  + Diagram

    Description automatically generatedDiagram

    Description automatically generated
  + Diagram

    Description automatically generated

Commits (Changesets/Revisions)

* \* - package of information describing changes made/a complete snapshot of the project at any one time
* Comprises:
  + Unique ID
  + ID(s) of the parent commit(s)
  + Description of changed change control items
    - Changed
    - Added
    - Removed
  + Metadata:
    - Change author
    - Timestamp
    - Log message
* IDs
  + Diagram

    Description automatically generated
* Log Message (Good Practice)
  + Short, meaningful title
  + Intent of the commit
  + Link to the issue in the project issue tracker/merge request
  + Explanation on how the commit addresses the issue

Branching

* The practice of maintaining multiple development lines based on a common history within a single version control repository
* Useful for:
  + Experimenting with the implementation of new features
  + Undertaking a substantial reorganisation
  + Supporting a specialised variant to an application
  + Creation of a release
* Different branching *practices* can be combined into a branching *strategy* for a team
  + Trunk-Based Development
    - Only one branch (main/master)
    - Benefits:
      * Simpler to work on
      * Clearer visibility of the project
      * Ability to break project into small increments of changes
      * Reduces risk of merge conflicts
    - Drawbacks:
      * Increases risk of broken code
  + Feature Branching
    - Creating a new branch for each new feature of project
  + Staging Branches
    - All branches are first merged into the **staging branch**, where they are tested in the User Acceptance Testing Environment, then merged onto the deployment branch
* Naming Conventions
  + Main ones:
    - master/trunk
    - uat, staging or preprod[uction]
    - deploy[ment] or prod[uction]
  + Hierarchical naming strategy for feature branches, like
    - feature/webapp/splash
    - feature/mobapp/login
* Branching Strategies (Workflows)
  + A collection of (often customised) branching strategies
  + Many standard published ones:
    - Gitflow
    - GitHub Flow
    - GitLab Flow
  + Most teams tailor their strategy to suit their circumstances
* Managing Branch Growth
  + Delete branches after they’re done with
  + Commit squashing
    - Merge all of the commits before merging to main branch

# Week 4: Customer Management and First Customer Day

Objectives for the First Meeting

* Undertake a **release planning meeting** and a **sprint planning meeting**
* Decide on the rules of engagement with customer
  + Communication channels
  + Access to GitLab
* Primary contact (product owner)
* Clarify the overall goals for the project
  + The customer’s ‘big picture’ vision (the **why**)
  + The realistic goal within the scope of the project (the ‘**what**’)
  + The high-level features to be implemented (the **detail**)
  + The **minimum viable product** (the customer’s **priorities**)
* Identify the key stakeholders, including:
  + the person/people in front of you
  + their boss/colleagues who may also represent them
  + users
  + data/API providers
  + Who might you need to talk to in the project?
* Determine appropriate intellectual property/licensing model
* Identify any significant requirements/constraints/risks for the project
  + Special technologies (hardware/software)
  + Visits to customer/user site for requirements gathering
* Decide on the goals for the first sprint
  + What can be achieved by the time you next meet?
  + What demonstrable feature/prototype can you show?

Customer Responsibilities

* Committed to a time budget for the project of 40 hours
* Attending on the specified customer days
* Responsible for providing resources beyond software available free or in the laboratory
  + Additional costs, e.g., for travelling to a customer site off campus
  + The School/University may be able to help with lending hardware/certain software licenses

Your Responsibilities

* Customers come in all shapes, sizes, experiences, and expectations. They may:
  + not know what is realistic
  + not know how to explain what they really want (or you may misunderstand)
  + change their mind (frequently)
  + not be reliable
  + But they are all *real* (Dawson 2000)
* Managing customers’ expectations and own workload
  + Be flexible about requirements changes, but also realistic
  + Be prepared to say something isn’t possible
* You are responsible for ensuring the customer delivers on commitments

Project Scope

* Pick 2: High Quality/Quick/Complete
* Within a fixed time frame are you building a:
  + Proof of concept prototype?
  + Sustainable basis for future development?

Intellectual Property (IP) and Licensing

* A complex issue in collaborative projects – the key is for both parties to be **reasonable**
* Both parties may contribute **background IP**
* University considers students to own all **foreground IP** for coursework, including the project
  + The Uni won’t routinely help protect this
  + Transfer of IP for commercial gain to be compensated
  + Open-source licenses are often the best solution
* Decide whether:
  + the project is likely to have a **direct** commercial value for either party
  + the customer will be in a position to further exploit the outputs after the end of the course
* Any agreement should not prevent coursework being submitted
* If in doubt, ask for help

Purpose of the Review and Planning Meetings

* To review progress since the last meeting
  + Compare what has been done to what was agreed
  + Report any additional work items completed
  + Explains reasons for deviation
* Discuss and elaborate key new requirements
  + New features that emerge from the review
  + Obstacles that have been uncovered
* Agree on a plan for the next iteration
  + Have a proposal ready, but be ready to negotiate
  + New features and other enhancements
  + Significant defects to be fixed
* Have the plan “witnessed” by the academic for the team

Roles in a Customer Meeting

* Meeting chair
  + Manages the agenda
    - Summarise the agenda at the start of the meeting (1 min)
    - The agenda should be specific to the project, now just review/discuss/plan
    - Summarise specific work:
      * done in the last sprint
      * not completed in the last sprint
      * planned for the next sprint
    - Signpost key questions/issues that the customer will be asked to resolve
    - Time Boxing
      * Good practice to estimate the time needed for each agenda item and record this on the agenda
      * Benefits:
        + Helps the team decide how best to use the time in the meeting
        + Helps the chair decide when to pause a discussion of an agenda item
      * Discussion that overrun should be paused
        + Use extra time at the end of the meeting if available
        + Otherwise, make a task to resolve the issue with the customer offline
  + Keeps the meeting to time
  + Constrains discussions
    - A careful balance needs to be maintained between:
      * ensuring useful ideas and suggestions are discussed
      * preventing a discussion from deviating too much from the agenda
    - The chair has authority over who talks. Be prepared to:
      * Interrupt a speaker if you think they’ve dominated too much
      * Ask the customer to elaborate some more if you don’t think a point is clear
      * Ask someone to contribute to stimulate discussion
    - Takes practice
* Product owner (may combine with chair)
* Lead demonstrator
* Note Taker
  + The only job for **at least 1 person** should be to document the minutes for the meeting
  + Consider writing notes directly on the GitLab wiki, rather than handwriting them first
  + Ask the customer if they mind you recording the meeting (most won’t mind)
* Checker

Effective Requirements Gathering

* Use visual aides to focus your customer’s attention during different phases of the meeting
* Use customer meetings to highlight complex decisions to be made and follow up in detail later
* Try experimenting with structures techniques, like a **value proposition canvas**, **user story workshop** or **design studio**
* Use live walkthroughs of new features to illustrate progress

Ask focused questions that:

* elicit concrete tasks to implement
  + Not “Do you like this UI?”, but “What parts of these 3 UI mock-ups do you prefer?”
* help your customer make choices with respect to finite resources
  + Not “Which of these 10 features do you like?”, but “Which 2 features from the list should be implemented first? Which 2 after that?”
* reveal hidden assumptions
  + Not “The system will have a mobile UI, right?”, but “A mobile app will be the only UI, right?”

Setting a Plan for Future Work

* Have a proposal ready for future work
* Incorporates:
  + Work delayed from previous sprints
  + Existing backlog items
  + New features identified during previous sprint
* Anticipate the **priority** of work items for your customer
* Be realistic:
  + Have person-time estimates for each proposed work item
  + Ensure that the total amount of work planned is less than time available
  + Assume you have 1 person-day per week per team member to work on the project

Negotiating the Plan with the Customer

* Customer expectations and priorities *will* at some point vary from your own
* Radical changes in direction should be approached cautiously
  + Don’t say yes immediately if the impact is unclear
  + Take the time to evaluate the impact on the project schedule
* Be honest with customer about the choice to be made
  + Be prepared to negotiate a new schedule
  + Incorporate previous delays into new schedules
* If in doubt, ask for advice from coach/academic

Managing Meeting Outcomes

* Summarise the agreed overall goal and work items to the customer *before* the end of the meeting
* Create backlog items in issue tracker for newly agreed work items
* Create issues for follow up tasks, such as additional meetings/resource requests

Communication between Meetings

* Many customers will want to meey more frequently than once a month
* Customer commits to 40 hours of contact time
* Also useful to establish other communication mechanisms
  + Invite customer to Slack
  + Invite customer to GitLab
* Use additional meetings to conduct more detailed requirements gathering and resolve uncertainties

Summary

* Managing customer and expectations requires preparation and organisation – customers are not being deliberately difficult, they are being human

# Week 5: Requirements Management

Requirements specify the system boundary

Diagram, venn diagram

Description automatically generated

Requirements versus Design

Diagram

Description automatically generated

Managing Requirements

* Maintaining requirements as a project evolves is important
  + Focus on on-going discussion with the customer
  + Drives acceptance testing
* Different requirements specification notations:
  + Formal specifications (e.g., Z)
  + View frames
  + UML use case and activity diagrams
  + Pseudo code
  + Natural language
  + …
* User stories are commonly used in agile projects
  + Actors/Roles/Personas
    - Describe the **categories** of users that will perform actions on the system
      * Avoid actors that are too generic or too specific
    - Includes the actor’s specific **motivation** for interacting with the system
      * Goals
      * Frustrations
    - Actors are essential for justifying the existence of features
    - Can help identify one of the types of system stakeholder to **negotiate** requirements with
    - A system actor does not need to be a human, though
  + Features
    - As an **[actor]**
    - I want to **[action on or by the system]**
    - So that **[rationale]**

Requirements Scope

* In theory, every action on a system should be a user story, but this means:
  + the scope would be unbounded
  + many user stories would be duplicated across multiple systems
  + effort would be wasted on acceptance testing of underlying framework(s)/dependencies
* Instead, focus on the business logic that is particular to the app under dev
* Treat framework features as assumptions about user roles

Non-functional requirements

* Can also be expressed as user stories

Refining User Stories

Diagram

Description automatically generated

* Adding more Detail
* User Story Tasks
  + User stories must be refined into implementation tasks
  + User stories may share tasks
* Story Prioritisation – MOSCOW
  + Must have
  + Should have (convenience)
  + Could have (useful)
  + Would be nice to have (or won’t have this time) (out of scope)
* Estimating Tasks
  + Estimation of software tasks is difficult
    - Some practitioners argue that estimation is useless
    - Sometimes necessary for budgeting and wider planning
    - Helps identify poorly understood tasks and force refinement
  + Various methods proposed:
    - Delphi (e.g., planning poker)
      * Based on developing consensus among experts
    - Market testing (priced to win, bidding)
    - Empirical (COCOMO)
      * Comparing to previous similar projects
    - Algorithmic (e.g., function point analysis)
      * Analyse features
* Scenarios
  + Useful for:
    - Describe workflows
    - Specifying acceptance tests
  + Follows an arrange/act/assert flow, similar to a unit test:
    - Text

      Description automatically generated

Storage of User Stories, Scenarios and Tasks

* Hand-written postcards on wall/whiteboard
  + Easy to reorganise and review as a whole
  + Harder to share across team locations
  + Requires effort to maintain
* User stories in text files on GitLab, Tasks in Issue Tracker

Assessing User Story Quality – INVEST

* Independent
* Negotiable
* Valuable
  + Why actor cares
* Estimable
* Small
* Testable

# Week 6: Software Process Improvement

Process Improvement Frameworks:

* ISO 9001
* Six Sigma
* Capability Maturity Model Integration (CMMI)

Principles (pre-conditions) for agile process improvement

* Measurement is domain-specific
* Improvement is an on-going activity
* Requires whole team participation
* Best effort assumption
  + Working harder won’t improve the process
  + Blaming is pointless and counterproductive
* Root causes, not symptoms

Process:

Diagram

Description automatically generated

Scrum Retrospective

* Timing
  + End of sprint, after review meeting with customer
* Participants
  + Usually whole team (sometimes including product owner, customer)
  + Scrum master as facilitator
* Duration
  + At least 30 mins
* Data Sources
  + Project artifacts
    - Objective evidence
  + Team members
    - Detailed insights
    - Need to be validated

Gathering data from team members:

* General method:
  + Team members populate the **template** board independently using sticky notes
  + Similar items are grouped together
  + Team votes on priority issues for discussion
  + Discussion to reveal the **root cause**
  + Select actions for improvement
* Variety of templates, e.g.:
  + Stop, start, continue
    - What did we do that did not help?
    - What should we begin doing?
    - What did we do well and should keep doing/do more of?
    - Diagram

      Description automatically generated
  + Mad, sad, glad
  + Loved, learned, lacked/longed for
  + Sailboat

Getting to a root cause – 5 ‘why’s:

* Not necessarily 5

Monitoring and measuring

* Create tickets as appropriate for next milestone/release
* Ensure someone is responsible for each action
* Include previous decisions in future retrospectives
* Review the process review process

Continuous process improvement

* Benefits:
  + Problems can be dealt with quickly and effectively
* Drawbacks:
  + May mean team isn’t involved in diagnosis/implementation of the solution
  + May be disruptive to software process

Improving Process Improvement

* Retrospectives can become stale
* Important to:
  + Vary the structure
  + Reflect on the retrospective itself
  + Allow other team members to facilitate

Key Point

* Process improvement is an essential quality assurance process

# Week 7: Code Reviews

What can be reviewed:

* App source code
* Source code docs
* Test code
* Design descriptions
* Requirements specifications

Why undertake code reviews:

* Detect defects in software
* Identify refactoring opportunities for poorly structured code
* Develop a shared understanding of code between developers
* Share good practice between team members
* Gain knowledge about *legacy* systems

When

* Periodically, on a portion of the accepted code base
* On a proposed change to a code base
* Before a proposed change to a legacy system

Do they work:

* Defect Detection Rate:
  + total defects = defects(m) + undetected defects
  + defect detection rate(method) = |defects(m)| / |total defects|
    - Simply: found / all

Conducting – Workflow

Diagram

Description automatically generated

Merge request good practice

* Keep small
  + Single review should be max 300 lines of code
  + Should be possible to complete a review within 30 mins
  + Break larger merge requests in smaller change packages
  + Incorporate code review time into the team’s workload model
* When describing:
  + Give succinct title
  + Explain single purpose. Pick: (Lientz and Swanson, 1980)
    - Corrective
      * Resolve bug reports and refer explicitly to the bug that is being fixed. Request should summarise defective behaviour, and correct behaviour realised by the change
    - Adaptive
      * Responses to changes in a system’s environment.
      * Ex: need to migrate to a new release of a dependency/language implementation
    - Perfective
      * Add new features. Change request should state what the new feature does and refer to the requirement
    - Preventative
      * Enhance maintainability of code
      * Ex: refactoring
  + Report wider impacts of change, e.g., breaking changes
* Automate
  + Block merge requests that don’t conform to standards
    - Maximum size
    - Breaking tests
    - Code style violations
    - Commented out code
    - Spelling mistakes
    - Inadequate tests
  + Static and dynamic analysis can be useful for locating *smells*
    - Source code metrics
      * Lines, ratio between executable code and comments
    - Design metrics
      * Fan-in and fan-out coupling, cyclometric complexity
    - Test suite metrics
      * Effectiveness and efficiency of test suite, like test code coverage and mutation testing
    - Incorrect usage patterns
      * Memory and IO leaks, type cast errors
  + Create a merge request template to ensure consistent

Questions to ask during a review (non-exhaustive)

* Purpose
  + Does this change solve a problem?
  + How would I solve this problem?
  + Has dependant docs (user guides) been updated?
* QA
  + What are the typical/extreme cases for use of this code? Are they covered by test cases?
  + Has the dev thought about corner cases (empty collections, parameter boundaries)?
  + Are the test cases readable and well-organised?
  + How does this code handle exceptional situations (loss of Internet, missing file)?
* Architecture
  + Does the change follow the existing architectural convention for the software?
  + Are there missed opportunities for reuse of existing code?
  + Are there clones (duplicate code) present in the change?
* Code
  + Do identifiers follow project naming conventions? Is the purpose evident?
  + Is the code self-documenting?
  + Are all source comments necessary (explain why, not how)?
* Non-functional considerations
  + Are performance optimisations possible?
  + Are efficiency optimisations possible?
  + Are relevant security patterns followed?

Realtime Code Reviews

* Pair programming
  + One driver, one allocator (instructor)
* Mob programming
  + Team works collectively

# Week 8: Build, Release and Dependency Management

Why use it

* All projects depend on other software
* Compiling dependencies may be slow and not completely automatable
* Enables *caching* of ‘oven-ready’ components

Tooling

* Ex: make/configure (C), Maven (Java), Ant + Ivy, etc

Build Management

* Terminology
  + Build configuration file
    - Contains declarative and imperative statements, describes the targets, mappings, and tasks
  + Targets – abstract goals within a software build management lifecycle
    - e.g., resolving dependencies, compiling code, testing binary
    - Targets may depend on one another
    - Typical:
      * Diagram

        Description automatically generated
  + Mappings – relationship between source and generated artifacts to reach targets
  + Tasks – actions to satisfy the mappings
    - e.g., execute compiler/test suite
* Should be repeatable (idempotent)
  + Reliable
  + Possible to definitively generate a particular release of a software system from its source commit
* Configuration by Convention
  + Many software projects have similar structures
  + Workflow of tasks for managing releases is also similar
  + Follows conventions imposed by the build/release tool
    - (but allows for exceptions)
  + Benefits:
    - Minimises configuration code to be maintained
    - Facilitates project comprehension and readability

Dependency and Release Management

* Types of dependency
  + Environmental (conditions that must exist in surrounding software infrastructure to work)
    - Explicit (specified in dependency strict)
      * e.g., Java SDK version
    - Implicit (assumed to exist, not documented)
  + Application (on other software components within the infrastructure, like Java file/library)
* Types of repository
  + Public (better for distribution to customers)
  + Private/local (better for maintaining pre-public stable versions of software)
* Tasks

1. Establish requirements
2. Obtain list of available releases for each dependency
3. Choose and check feasibility of release version combination
4. Retrieve missing artefacts

* Release/dependency versioning
  + Useful to have a labelling scheme to describe different releases of a component
  + Included in dependency spec
  + Each release label is a reference to a commit in the version control repo
  + May incorporate info about the release
    - Temporal relationship to previous and future releases
    - Contents of the release
    - Purpose of the release
* Specifying dependencies
  + A project almost always has **transitive** dependencies (dep on one might mean another dependency)
  + Dependency graphs may be cyclic
  + Dependency resolution is known to be NP-complete
  + Different strategies employed to find a satisfactory combination of dependencies
    - ‘Nearest’ version wins (e.g., Maven, pip)
    - Allow multiple releases to be used in the same assembly
  + Good practices
    - Dependency resolution optimisations are hard
      * Under constrained specifications may result in incompatible combinations
      * Over constrained specifications limit flexibility, require more effort to maintain and may persist the use of older, buggier releases
    - Migrate to new releases when available
    - Don’t rely on transitive dependencies
* Types of Release
  + Composition
    - Core Executable
    - Tailored Executable
    - Optional extensions
    - Sources
    - Documentation
    - Datasets
    - Complete
  + Schedule/intent
    - Bleeding edge/snapshot (a release for each commit)
    - Incremental
    - Nightly builds
    - Beta test releases (to release to trusted 3rd parties)
    - Release candidates (beta test that could be final)
    - Production releases
* Release branches
  + New branch for each release of software product
  + Diagram, schematic

    Description automatically generated
  + Allows the dev team to prepare a new stable release to a customer
  + Only changes to support bug fixes are permitted in the release branch
    - Bug fixes in the release can be merged into master
  + New features continue to be developed in master
* Changes to different types of APIs
  + Private APIs (minimal docs)
    - Changes should not have any impact on dependencies
  + Public
    - Not explicitly documented, but accessible
  + Published
    - Externally accessible
    - Fairly extensible documentation
* Semantic versioning
  + Format: **major.minor.incremental[-tag]**
    - e.g., 4.0.3-SNAPSHOT
    - Tag – release status/type/other info
    - Patch/incremental = non-breaking **bug fixing** change
    - Minor – non-breaking **feature addition** change
    - Major – **breaking change** to a published API
* Deprecating features
  + Feature that has been left in place for compatibility reasons, but will be removed from a future release of the wider system
  + Document:
    - The scope of the deprecation
    - The release version of the software containing the new mechanism
    - A schedule for when the deprecated feature will be removed
    - An explanation of why the feature has been deprecated
    - A description of how to change dependent code

# Week 9: Continuous Integration

Integration Hell – situation where dev spends more time trying to integrate feature than making that feature

* Exacerbated by:
  + Devs committing simultaneously
  + Committing infrequently/late

Practices (Fowler)

* Change management
  + Maintain a single source repo
  + Everyone commits to the mainline (trunk/master) every day
    - Diagram

      Description automatically generated
    - Diagram

      Description automatically generated
  + Every commit builds the mainline on an integration machine
  + Everyone can see what’s happening
* Quality Assurance
  + Automate the build
  + Make your build self-testing
  + Test in a clone of the production environment
  + Keep the build fast (within 10 mins)
* Deployment
  + Make it easy for anyone to get the latest executable
  + Automate deployment

Feature branching options

* Implement the new feature on the trunk, with frequent commits of small changes
  + Advocated by CI purists
* Implement the new feature in a branch and merge changes from the trunk frequently
* Implement a feature as a prototype branch and then re-implement the feature on the trunk in small changes
* Implement the feature as a permanent branch

Maintaining multiple CI processes

* The mainline trunk development
* Forked developments for specific customers/purposes
* The latest *stable release* that only receives defect fixes
* Significant feature branches

Monitoring and maintaining software quality

* Detecting broken builds
  + The system should be **compiled from source** in a clone of the production environment
  + A suite of **automated regression** tests should be executed on the complete system
  + Appropriate **static analysis** checks (e.g., code style conformity) should be performed and compared to benchmarks
* Preventing broken public builds
  + Diagram

    Description automatically generated
* Preventing broken builds with branches
  + Diagram

    Description automatically generated

Staging Environments

* Limitations of staging environments:
  + The system will be distributed on too many computing nodes to test realistic scenarios
  + The system will be used on too many platform configurations to test them all
  + The system will have too many simultaneous users
  + There are too many diverse user types for all to be tested
  + Only one platform for the software is available and is used for production
  + Software dependencies, such as network end points and data sets, cannot be accessed from outside of the production

CI Environments:

* Ex: Hudson/Jenkins, TeamCity, CruiseControl, BuildBot, Integrity, GitLab Runner, TravisCI
* Workflow:
  + Diagram

    Description automatically generated

Maintaining visibility

* Push notifications (SMS, email)
* Broadcast mechanisms like displaying a status monitor on a large screen display in the dev area

Metrics

* Unsuccessful builds
* Average unsuccessful tests
* Time to build
  + Fast builds are essential
  + The longer the build, the later problems are noticed
  + If too long:
    - Reconfigure process to optimise
    - Split into different purposes

Advanced topics:

* Continuous deployment
  + Automate to deliver to users/customers
  + Benefits:
    - Deployment process is regularly used (better understood)
    - Allows delivering different deployments to different users (like A/B testing)
* Chaos Engineering
  + Defects/failures deliberately introduced to system to see how it copes

Summary

* Continuous integration practices minimise the disruption caused by rapid, concurrent changes to software systems

# Week 10: Static Analysis, Code Readability

Static vs dynamic

* Static applied on program artifacts at rest
* Dynamic applied during execution
  + e.g., time in methods
* Program artifacts may be:
  + Source code
  + Compiled object code

Use cases:

* Developers check changes before submission to review
* CI pipeline can block merge requests that fail static analysis checks
  + Prevents defects entering the mainline
  + Optimises code review process
* Assists reviewers in assessing proposed changes

Spectrum of opinionation for automated code analysis

Chart

Description automatically generated

Self-documenting code

Code Style Standard

* ‘Official’/community language guides, e.g., PEP8 for Python
* Vendor-driven, e.g., many guides for Ruby, Oracle, Eclipse, Atlassian for Java
* *Linting* tools for many languages
  + <X>Lint for most languages
  + Sonarqube
  + CheckStyle, FindBugs (Java)
* Rules cover:
  + Line length (e.g., max 80/120)
  + Identifier naming conventions
    - f or F or f\_hand vs last\_commit\_file\_handle
  + Use of white space (horizontal and vertical)
    - y = x + z \*\* 2 / 3
* Others:
  + Control structure usage
  + Use of parentheses
  + Redundant code
  + Implicit usage patterns
    - type(x) == int vs isinstance(x, int)
  + Some tools also check for spelling

Static Analysis can be integrated with GitLab

Bug detection

* Ex:
  + ‘Property’ methods that can return null
  + Access of null object properties
  + Inconsistent return statements in a method/function
  + Overloaded method names
  + Redefined/reassigned variables/parameters

Vulnerability Analysis

* SQL Injection in Java automatically detected

Detect Poor usage patterns

* Ex: IO and Memory Management

Measuring Designs

* Cyclomatic Complexity
  + complexity = edges (transitions between statements) – nodes (statements) + terminals (exit points) x 2
  + McCabe recommends having it below 10
* Nested Scope Depth
  + Counting, starting from inside biggest scope (it being 0)
* Coupling
  + Raw Measures
    - Method calls and variable accesses
    - Import statements
    - Type usages
    - Inheritance relations
  + Scope
    - Method
    - Class
    - Module
    - Package
    - Application
  + Fan-in complexity (afferent coupling)
    - For precisely identifying classes that change if the subject class changes
  + Fan-out complexity (efferent coupling)
    - Outbound, how dependent on other classes
* Inheritance Depth and Width
  + Tall inheritance hierarchy (a lot of depth) may mean an over-abstraction
    - Deeper => harder to change top-level classes
  + Wider => should introduce more intermediate abstraction

Other topics

* Bad smells (detecting, refactoring)
* Dynamic Analysis

Summary

* Static tools are an effective means of automatically detecting bugs and poor style that can be eliminated prior to code review as part of a CI pipeline

# Week 17: Behaviour-Driven Development

Terms:

* Test case – description of a set of actions to be performed on software and an expected outcome
* Test – the execution of a test case, producing a test outcome
* Testing – the practice of creating, maintaining, executing and evaluating test cases

Reasons for testing

* Detection of defects
* Support the design and implementation of a module
* Prevent the introduction of defects
* Document the behaviour of a system
* Demonstrate that a system meets its specification

Scales of testing

* Unit tests
  + Checks behaviour of a single module/class
  + Should avoid slow activities
* Integration tests
  + Performed on subsets
* Acceptance (system) tests
  + Performed on a fully integrated system
  + Smaller number of unit test cases
  + May be manually implemented
  + Non-functional properties as test cases

Behaviour-Driven Development

* Creation and maintenance of a requirements specification in a structured natural language (Gherkin)
* Generally used for system (acceptance) test cases
* Life-cycle for feature development:
  + Diagram

    Description automatically generated
  + Not actually used consecutively, but concurrently
  + Graph shows hierarchy at least
* Step types
  + Given – **description** how to set up the test case fixture before the test case is executed
  + When – **actions** to be taken during the test case itself
  + Then – **assertions** that are made about the state of the system and/or output at the end of a test case
  + (And) – synonym for the previous step
  + Background – grouping steps that run before first step in a scenario
* Frameworks
  + Diagram

    Description automatically generated
  + Ex:
    - JBehave (Java)
    - Behave (Python)
    - Cucumber (Ruby, Python, Java)
    - Behat (PHP)
    - Specflow
* Step Abstraction
  + “Dirty trick” in JBehave and Behave
* Scenario outlines and example tables
  + When the same sequence of steps can lead to different outcomes depending on the values used
  + Often used in conjunction with parameterised steps
* Where to implement step functions:
  + On the API (code steps are a façade)
  + At the ‘system’ interface, e.g., user interface/REST API
* Good practices
  + Keep scenarios short
  + Keep individual steps short
  + Comply with AAA convention (three steps – given, when, then)
  + Write features in user domain language (avoid technical language)
  + Use frameworks to cut down on redundancy
  + Develop features and implementation gradually
  + Work with the customer to validate features
* Limitations
  + Maintainability
    - Hard, ex:
    - Difficult to make succinct
    - Requires hardcoding requirements
    - Only 1 level of abstraction available
    - Implicitly sequential (even when it doesn’t matter)

Summary

* BDD provides a means, through executable specifications, of ensuring traceability between user-facing requirements and implementations

# Week 18: Evaluating Test Suites

Effectiveness and Efficiency of Tests

* Disproportionate (trade-off)
* Effectiveness – ability to prevent defect being introduced
* Efficiency – cost/resources needed to keep/maintain test suite

Useful metrics

* Post release reported defects (effectiveness)
* Lines of code (LoC) reached during test suite execution (efficiency, effectiveness)
* Test suite execution time (efficiency)
  + Usually, less than 10 mins
* Failed tests per defect (efficiency)
* Test Suite LoC (efficiency)

Test code coverage (dynamic analysis)

* During test suite execution:
  + effectiveness = unique LoC executed / total LoC (in the app)
  + efficiency = total LoC (in the app) / (total LoC + LoC executed count)
* Granularity:
  + LoC
  + Statements
  + Expressions
* If using LoC, can ignore:
  + Declarations (import statements)
  + Parentheses
  + White space
  + Source code docs
  + ‘Uninteresting’ methods, like getters and setters
* Limitations:
  + Doesn’t measure defects

Mutation testing

* Estimate likelihood that a change to system will be detected by 1+ failing test cases
* Defects – combinations of small-scale code mutations
* Mutant – mutation operation
  + ex: replace infix mathematical conditionals (> to <=)
  + ex: replace infix mathematical operators (\* to /)
  + ex: replace incrementor with decrementor (++ to --)
  + Types:
    - Conditional operators with their boundary counterparts
    - Infix mathematical operators with other operations
    - Member field values with default/other values
    - Variable values with default values
    - Constructor calls with NULL assignments
    - Returned values with default values
    - Method call results with default values
* Process:
  + Diagram

    Description automatically generated
  + Classifying mutants after testing:
    - Killed mutants – successfully detected
    - Survivor mutants – successfully pass all tests – undetected
    - Undetermined mutants – programs that do not halt
* Metrics
  + mutant survival rate (effectiveness) = #survivors / all mutants
  + efficiency = killed mutants / failed tests
* Configuration space and optimisation
  + Choice of mutation operations
  + Number and combination of mutations applied to a program
  + Timeout period for completing a test case
  + Scope of test suite for each mutant
* Comparing results across apps
  + 2 **different configs** for different projects will have different effects on the performance of the underlying test suite
  + 2 projects may have very **different characteristics**, which are better tested using very different configs
* Limitations
  + The effectiveness is dependent on careful configuration of the process
  + Mutation operations (or their random combinations) may not be representative of the types of defect that are introduced into a project
  + Mutation testing takes a long time to run because the entire test suite must be executed on each mutant
* Tool ex: PiTest (Java), Mutpy (Python), Stryker (JS)

# Week 19: System Scale Testing

Non-functional requirements – emergent characteristics or properties of the overall system that must be measured to be satisfied rather than observed as a provided feature as for functional requirements

* **Cannot** typically be checked for in **unit tests** or through static analysis. They can only be tested once an implementation of a system has been realised.
* The implementation tested may be the final system, or a prototype.
* Alternatively, it may be possible to build a model from which the characteristics of the final system can be approximated.
* An advantage of testing non-functional requirements is that they may act as an aggregate for more complex functional properties that are too complex to test individually.
* Info needed to test:
  + Property of the system to be measured
  + Metric to be used
  + Operating conditions in which the requirement must be satisfied
  + Threshold the system behaviour must exceed to satisfy the requirement

Safety-critical environments

* Automatic pilots for transportation, such as aeroplanes
* Equipment used in the operation of particle beam radiotherapy equipment
* Space-based technology, e.g., satellites, spacecraft, planetary rovers
* Nuclear power station management systems

Reliability testing

* Extent to which a system performs according to its **specification**
* Metrics:
  + Probability of failure on demand (access)
  + Mean time to failure
  + Down time (how much time can be lost to system outages over a long period of time)

Testing system safety

* Model-driven development
* Cleanroom development (probability of failure on demand)
* HAZOPS (identify flows of info between software components)
  + Used to gain assurance

Safety case – justification that a system meets its system requirements

Systems operating in hostile environments

* Active agents (threats/attackers)
* Ex:
  + Electronic voting systems
  + Friend-or-foe detection systems
  + Digital rights management systems
  + Network security systems
  + Authentication systems
  + Digital cash

Security testing metrics

* Attacker capabilities/knowledge required to penetrate the system
* Time, resources to penetration
* Unpatched vulnerabilities per line of code
* Patches issued per year
* Successful penetrations per year
* Attack surface exposure

Security testing methods

* Vulnerability testing
  + SQL injection testing
  + Port scanning
  + Fuzz testing (supplied with unusual/malformed inputs)
    - Checks validation done by system
    - ex: buffer overflow detection
* Penetration testing

Performance testing

* Properties
  + Throughput – amount of transactions system can process in a fixed amount of time
  + Demand – maximum rate of transactions system can process concurrently
  + Response – average/maximum amount of time to respond to individual transaction request
* Threshold
  + Limit
    - System behaviour within limits
  + Stress
    - System behaviour beyond limits

Scaling effects in system testing

* A picture containing diagram

  Description automatically generated
* Dimensions of scale:
  + Number of lines of code
  + Number of software modules and inter-dependencies
  + Number of software platforms
  + Variation in build and configuration of deployments
  + Duration and variation in inputs
  + Software development team(s) size, locations, experience
  + Number of geographic locations and network connections
  + Number and variation in users

Heterogeneous systems

* May be composed of:
  + Different business units within a single organisation
  + Business units from different subsidiaries within the same umbrella organisation
  + Project teams within different organisations brought together as part of a consortia
  + Many individuals from different organisations
* Ex: ATLAS Project
  + 7 million lines of code
  + Testing:
    - Nightly
    - Full Chain Test

Social-technical systems

* Both computer software, hardware, users, and the surrounding organisational and cultural practices
* Testing difficult:
  + The members of the org may not be available to take parts in tests
  + Obtaining realistic scenarios may not be practical
  + Identifying and covering the real working practices may prove difficult
  + The organisation may evolve independently of the system

Systems of Systems (Coalitions of Systems/Joint Systems)

* Multiple heterogeneous semi-autonomous systems that cooperate to produce emergent effects
* Testing difficult:
  + Each of the member systems will have different cultures and practices
  + The system cannot be isolated/configured for a test
  + One of the member systems may evolve *during* a test
  + The expected outputs for the system are not easy to define

Getting feedback from the users

* Using users as testers
* Trade-off between newer systems having fewer users, but more defects, and vice versa
* Examples:
  + Automated crash reporting
  + Integrated defect reporting
  + Beta testing
    - New features to subset of users
  + A/B testing
    - Evaluate different candidates of design to different users

Summary

* Increasing scale in a software/computer-based system reduces the coverage and effectiveness of testing efforts, even if a proportionate number of resources are applied

# Week 20-21: Software Architecture

Software component

* “Physical manifestation of an object that has a **well-defined interface** and a set of implementations for the interface” (Hopkins 2000)
* “A coherent **package** of software artifacts that can be independently developed and delivered as a unit and that can be composed, unchanged, with other components to build something larger” (D’Souza and Wills 1999)
* “Components extend [object oriented] principles by strengthening the role of the interface and by a separate notion of component specification. Components must conform to a component standard” (Cheesman and Daniels 2001)

Component-oriented system – collection of interacting components that have been combined to realise some wider purpose

Objects vs components:

* A component is a **self-contained**, **self-controlled** and evolving bundle of smaller scale objects
* A component-oriented system can be distinguished by a **middleware framework** to mediate interactions between components
* The middleware can incorporate an **interface definition language** (IDL) that is used to define the published interfaces of each component in the system
* Components are orchestrated by defining an **assembly** either during system initialisation or dynamically at run time
* Components in the same component-oriented system may be implemented in several different programming languages

Types of components:

* General purpose
  + Provide common functionality that is intended for reuse in many different applications
  + Very well documented and stored in a component repository
* Application-specific
  + Implement the problem specific functionality and business logic of the application
  + Orchestrate the behaviour of the general-purpose components to realise application specific needs

Problem: dilemma between simple components with complex interactions vs vice versa:

* Solution – nest simpler components within more complex ones
  + some of the functionality of a component is encapsulated (and replaceable) with other sub-components
  + minimise the management of interactions between components at any given level of decomposition in the system

Component diagram notation

* Diagram

  Description automatically generated
* Box (shape – plug widget stereotype) – component
  + Label (A/B)
* **A provides** an interface I (shape – **facet**) to B
* **B requires** interface I (denoted by connector/receptacle around the facet)

Separation of concerns for component systems

* Components have clearly defined responsibilities for some function
* The component should not be affected by changes to the implementation details of other components
* Design by contract
  + The interface definition for a component constitutes a ‘contract’ between the provider and the supplier of some functionality
  + Contract describing (Meyer (Eiffel dev)):
    - **Benefits** of using the interface that are offered by the providing component
    - **Obligations** imposed on the component that proposes to use the interface
  + Design by contract – process that works with these very precisely documented interfaces to compose software systems
  + Preparing component interface contract:
    - Visible component state (component attribute)
      * realised by the providing component
    - Invariants
      * describing the legal states of the providing component
    - For each method in the interface:
      * Signature
        + comprising a method identifier, argument identifiers and types, method return type and any exceptions that might be raised as a result of improperly invoking the method
      * Pre-conditions
        + required state of the providing component before the method can be invoked and any restrictions on supplied arguments that cannot be expressed in the method signature
      * Post-conditions
        + describe the state of the providing component after the method invocation has been completed and constraints on any values returned to the calling component that cannot be expressed by the IDL’s type system
      * Semantics
        + including, for example, the correct sequence that methods should be invoked in
      * The visibility of each method
    - Constraints: invariants, pre-, post-conditions

Checking component contracts:

* Statically at compile time.
* Using test frameworks, such as JUnit.
* Within the program logic at runtime.
* By the component middleware at runtime

Leaky abstractions

* (Joel Spolsky)
* whenever two component implementations (a provider and a requirer of an interface) are wired together in an assembly, their future evolution is influenced by assumptions about:
  + The way that a providing interface will be utilised
  + The way that the providing interface is realised
* Assumption examples:
  + The ordering of objects received from an iterator over an-unordered set
  + The behaviour and structure of a database system - knowing how queries will be evaluated can have a big impact on performance

Separating concerns in UI design

* Storing user data
  + text, images, video, audio, etc
* Presenting different views of the data
  + tables, graphs, streams, thumbnails, etc
* Changing the data
  + creation, deletion, editing, etc

Architectural patterns

* **Model-view-controller**
  + Diagram

    Description automatically generated
  + Adapting:
    - The retention of some model info in the View
    - The View can be updated as a result of notifications from the model that something has changed or by the View polling the model
    - The View and Control interact directly with the system’s user at the system boundary
* Distributed systems
  + Motivations:
    - Desirable to access services from different locations.
    - Infeasible to store and process data on a single computer.
    - Composed of many different software components from different organisations
  + Patterns:
    - **Client-server**
      * Benefits of centralising info storage and service provision:
        + The management and organising of data can be maintained (and controlled) in a single, consistent structure.
        + The problem of ensuring consistency between multiple copies of the same data item is avoided.
        + The approach provides a single, globally known, point of access to users.
        + The management of changes to service functionality can be undertaken in the server rather than in multiple clients.
      * Diagram

        Description automatically generated
      * Interfaces:
        + Connection
        + The main API
      * Session component
      * Types:
        + **Thin client architecture**

Clients contain very little program logic (all info and most functionality collected in server)

Client only issues instructions

* + - * + **Fat client architecture**

devolving much more responsibility for information management, validation, processing and storage to the clients

Advantages:

reduce the amount of communication required between the client and the server by caching some information in the client

some of the processing is delegated to the clients (reduces computation in server)

Disadvantages:

Risks introducing inconsistencies in stored data

Clients have to be updated more frequently (about software defects)

* + - * Limitations:
        + Communication between servers and clients may suffer from unacceptable **latency**
        + The **scalability** of the architecture is **bounded by** the physical resources (computational power, network bandwidth, memory) available to the **server**.
        + The single point of reference in the architecture (the server) is also a **single point of failure**
    - **Peer-to-peer**
      * moving all services and data in the system into the clients themselves so that every peer is both a client and a server
      * As the demand for resources grows due to an increase in clients, there is an equivalent increase in the resources available for servers, because each peer fulfils both roles
      * useful when a system has responsibility for extensive computational processing, or hosting large amounts of information
      * constrained by the physical capabilities of the hardware
      * Ex: Tor, BitTorrent
      * Peer discovery
        + Using a globally known registry to record the addresses of peers in the system.

if the registry becomes unavailable then none of the peers will be able to discover new peers that might have the resources they need

* + - * + Searches across the peer-based network.

no guarantee of finding the required resources if the search times out before the right peer is reached

* + - * + A hybrid of both patterns

significant additional architectural complexity

* Information processing patterns
  + Disadvantages of synchronous communication mechanisms:
    - Underlying communication channels may be **unreliable** or subject to delays that disrupt the expected flow of computation in a component
    - The client must choose which components will provide information for a computation before processing begins. because only a single request can be made at a time
    - Components are forced to request information from other components in a pre-determined sequence because they need to wait for a response from each component in turn
    - The overall system design **cannot exploit computational parallelism** to improve response time
  + **Message-oriented architecture**
    - Asynchronous
    - all communication occurs as discrete messages that pass through a message bus
    - The bus routes messages to the appropriate client based on a routing policy
    - Types:
      * Message-Driven
        + all internal computation in a component is the result of receiving a message from another component
      * Message Broker
        + a broker is responsible for deciding which component receives which message
    - Interfaces:
      * IBroker
        + send messages to other clients via the bus
      * IQueue
        + read messages from the bus that have been sent specifically to the client by other clients
    - Configuring
      * Relationship between queues and clients
      * Policy applied by the broker to distributing messages
    - Advantages
      * Communications security is centralised by ensuring that all messages pass through secured channels on the message bus
      * The broker can be configured dynamically to route messages according to the state of the application
      * The message bus provides a central location for monitoring inter-component communication
  + Pipe and filter
    - Sequential data processing apps
      * Video and audio media transcoding, where video is decoded, potentially re-formatted (size, colour model etc.) and then encoded
      * Textual analysis, where prose is checked for spelling and grammar errors in one language before being translated into another.
      * Inter-bank payment processing, where very large numbers of small-scale currency transfers must undergo a series of validations before being approved.
      * Processing and synthesising data gathered from scientific instruments.
    - Design Problems (that pipe and filter solves):
      * Each of the steps in the transformation can take a variable amount of processing time for a given data item.
      * Steps may need to replaced or re-ordered as the application is adapted to different needs.
      * It may be desirable to have a convenient means of re-using some functions in combination
    - Each transformation that must be applied to the data is implemented as a single component called a **filter**
    - Each filter provides and implements the same interface, sometimes called the **pipe**
    - The filters are then wired into a single sequential assembly called the **pipeline**, with each successive component obtaining data from a component on the left and passing output to the component on the right
    - Bounding pipeline:
      * **data source** component provides input on the right and requires the pipe interface
      * **data sink** component provides the pipe interface on the right to accept the system’s output on the right
    - Scheduler component is responsible for deciding which of the other components should execute at any one time
      * Interface – Control
    - Solution to problems:
      * Processor time is allocated to the different filters based on load. This ensures an even flow of data through the pipeline.
      * All of the filters provide and require exactly one pipe interface, so can be re-ordered and composed as necessary
    - Variants:
      * push or pull driven data flows
      * sequential or concurrent data processing filters
      * re-orderable or inter-changeable filters
      * branching data filters
* Plugin architecture
  + Motivation
    - It is anticipated that new features may need to be added over time as requirements change.
    - Different users of the application have different requirements.
    - It is anticipated that some users will want to develop their own functionality for the system.
    - The application will run on a variety of different platforms
  + Components:
    - *Registry* that stores the specification of all plugins available to the system
    - *Application* can query the registry for available plugin specifications of a particular type
    - *Loader* component is used to instantiate and configure the component for use by the main application, using the specification supplied by the *Registry*
  + Should be used for extension only
    - If used for whole app, *inner platform effect*
      * Developers reinvent app as a whole new programming language

# Week 22: Software Refactoring

Maintaining source code and system design

* effective use of a version control system
* an automated test suite
* a clear architecture and design
* a simple and consistent coding style, enforced through automation where possible; and
* appropriate use of comments and other documentation

Def

* ``Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.’’

The refactoring process

* Diagram

  Description automatically generated

Bad smell – pattern in the source code indicating poor structure or inappropriate coupling between modules

When to refactor

* implementing new functionality
* correcting a defect
* doing a code review
* when you are trying to understand how a software artifact works
* reduce poor software design (bad smells)

Fowler’s ‘Bad smells’

* cloning
  + **duplicate code**
    - Refactoring: extract method → (pull up method, form template method) or substitute algorithm
* complex structures
  + **long method**
    - The behaviour of long sequences of instructions is harder to understand, because all of the details of the implementation are presented to the reader at once. In general, methods that can’t be read on a few lines in an editor are too long
    - Refactoring: extract method → (introduce parameter object, replace temp with query) or replace method with method object → extract method
  + **large class**
    - Indicates that too many responsibilities have been allocated to a single class. It is also possible that the large class contains duplicate code
    - Refactoring: extract class, extract subclass
* Variables and parameters
  + **Long parameter list**
    - Imposes stamp coupling on the design of the method, because each time the parameter list changes, every call of the method must also be changed. A long parameter list also indicates that a method that is used in different ways, depending on how it is called
    - Refactoring: replace parameter with method, preserve whole object
  + **Feature envy**
    - Indicates that a method is in the wrong class because it obtains most of its data from another class. If either the data or the accessing method change, then so will the other
    - Refactoring: move method, extract method
  + **Data clumps**
    - Occurs when the same, independently sourced, data is used together in different locations in a system. This might be identifiable from duplicate parameter lists for methods, or from duplicate blocks of query method calls. Both of these examples are also indicators of cloning
    - Refactoring: extract class → (introduce parameter object, preserve whole object)
  + **Primitive obsession**
    - Indicated by a preference for using primitive data types for representing more complex values with additional semantics.
    - This is problematic because the semantics and supplementary properties of a data type is not explicit when it is represented as a primitive value, and has to be maintained independently of the data.
    - Ex: A date in the Gregorian calendar could be represented by three integers, for example (one each for day, month and year). However, this means the relationships between these values has to be maintained elsewhere in the system
    - Refactoring: replace data value with object, replace type code with class, extract class, introduce parameter object
  + **Temporary field**
    - These are fields that are used as variables in method bodies but aren’t always needed. Fields should only be used to record information about an object’s state that must persist between method calls
    - Refactoring: extract class, introduce null object
* Making changes
  + **Divergent change**
    - Another indicator that a class has too many responsibilities, identifiable when a single class must be altered in different ways to respond to different changes in the system’s environment. This suggests that the class has two or more different sets of responsibilities
    - Refactoring: extract class
  + **Shotgun surgery**
    - The opposite of divergent change, identifiable when a change in the environment necessitates a number of different changes in several different classes in a system. This makes maintenance more time consuming, and easier to get wrong
    - Refactoring: move method, move field, inline class
  + **Parallel inheritance hierarchies**
    - Identifiable by dependencies between two inheritance hierarchies, such that a change in one hierarchy causes a change in another,
    - increasing coupling and maintenance costs
    - Refactoring: move method, move field
* Control structures and polymorphism
  + **Switch statements**
    - Indicates a reluctance to exploit object oriented polymorphism and increases maintenance costs, because each time a new option is introduced into the system, a flag and additional line must be added to the switch
    - Refactoring: replace type code with subclasses → replace conditional with polymorphism
  + **Refused bequest**
    - Sometimes a sub-class doesn’t need all of the operations provided by a super class. This means that all the methods of the super class are unnecessarily coupled to the sub-class
    - Refactoring: push down method, push down field, replace inheritance with delegation
  + **Alternative classes with different interfaces**
    - benefits of object-oriented polymorphism cannot be exploited. The two or more classes must be manually selected for use in the code
    - Refactoring: rename method, move method, extract superclass
* Design uncertainty
  + **Lazy class**
    - difficult smell to identify, given that many of the smells are about classes that do too much}. However, lazy classes, that have insufficient independent\* responsibilities can be unnecessarily expensive to maintain
    - Refactoring: collapse hierarchy, inline class
  + **Speculative generality**
    - move all design decisions into the configuration for a software system. At its most extreme, this phenomenon is known as the inner platform anti-pattern, because the system is designed as a software platform that must be configured, on top of an existing software platform
    - Refactoring: collapse hierarchy, inline class, remove parameter, rename method
  + **Incomplete library class**
    - when it is discovered that a class from a library doesn’t support all the operations needed by the client. In addition, the library class can’t be altered, because it is used in many other projects
    - Refactoring: introduce foreign method, introduce local extension
* Delegation
  + **Message chains**
    - Occurs when a message one object accesses a data item through a series of intermediary objects. The requesting object is therefore bound to all the other objects in the chain
    - Refactoring: hide delegate, extract method → move method
  + **Middleman**
    - Occurs when one object acts as an information broker to another object but doesn’t actually provide any information itself. There may be good reasons for this, if the broker is a proxy, for example. If the object is just passing on information, however, it may be better for the communication to occur directly
    - Refactoring: remove middleman, inline method
  + **Inappropriate intimacy** (object-orgy anti-pattern)
    - the opposite of accessing information via middlemen or chains. Instead, all objects freely interact with the properties of other objects, causing an `orgy’ of couplings between them
    - Refactoring: move method, move field, extract class, hide delegate, change bidirectional association to unidirectional, replace inheritance with delegation
  + **Data class**
    - Classes should generally have behavioural responsibilities as well as data items. If a class lacks behaviour, this is often because the behaviour has been implemented somewhere else
    - Refactoring: encapsulate field, extract method, move method, hide method
* **Excessive comments**
  + indicative of poor structure that can be improved through the application of refactoring(s). This process helps to create **self-documenting code**. Once the code has been refactored, its purpose and functionality become much more self-evident, reducing the need for supplementary documentation. This excessive documentation can be largely removed when the refactoring process is complete

Types of refactorings:

* Fixing methods
  + Extract method <-> inline method
  + Replace method with method object
* Moving functionality
  + Move method/field
  + Extract class <-> inline class
* Organising data
  + Encapsulate field
  + Replace data value with object
  + Replace magic number with symbolic constant
* Simplifying method calls
  + Parameterise method <-> remove parameter
  + Use parameter object
* Simplifying conditions
  + Decompose conditional
  + Consolidate duplicate conditional fragments
  + Replace conditional with polymorphism
* Reorganising classes
  + Pull up method/field <-> push down method/field
  + Extract superclass/subclass
  + Collapse hierarchy

Automated refactoring done by IDEs

Limits of refactoring:

* Changes can affect:
  + non-functional properties
    - might decrease a system’s overall memory footprint, whilst simultaneously reducing the system’s response time
    - may increase the number of method calls that must be made, increasing execution time for a transaction
      * considerations are particularly important for mobile, embedded or real-time applications in which computing resources may be limited
  + the application programming interface (API) of a system
    - class member moved/renamed
    - visibility of class member changed
    - list of parameters to an operation changed
    - return type of an operation is changed
    - exceptions that may be raised are changed

API

* set of public operations and attributes of one module that can be accessed by other modules in a system at runtime
* In object-oriented systems, every object has an API defined by the public members (operations and attributes) of its class

# Week 23: Software Licensing

Licence – assertion of the permitted uses of a software artifact and the conditions of use

Licence covers:

* Ownership
* End usage rights,
  + such as display and performance
* Distribution rights,
  + such as reproduction (copyright)
* ‘Engineering’ rights,
  + such as disassembly, modification and incorporation in other software
* Warranties and liability

Roles:

* Producer (receives benefit from distribution)
  + Author
  + Owner
  + Distributor
  + Developer
  + Contributor
* Consumer (receives benefit from use)
  + Customer
  + Infrastructure engineer
  + End-User
  + Developer

Licence ecosystem

* Producer can also be consumer of some other product

Cycles in Licensing

Licences can be applied to:

* Source code
* Binaries
* Documentation
* Outputs

Copyright

* form of intellectual property
* legal right to control the reproduction of a creative work for a limited period of time
* typically increased to at least 50 years since the publication or authors death
* Derivation
  + in many jurisdictions, such as the United Kingdom, a copyright holder is also entitled to the copyright on any derivative works
  + modifications to an existing source code base
  + Without specific authorisation in a licence, it is generally not permitted to attempt to make modifications to a copyrighted work
  + original author has the right to decide under what conditions copying/distribution/modification may be done and document them in a licence to consumers
* Contributors
  + Con: may make protection of the copyright harder to defend

Open-source contribution

* Different ways software may be incorporated in other software (derivation):
  + Source code that is incorporated into another project’s code base
    - If an existing work of source code (P) is incorporated into another source code base (C), then this may count as derivation
    - Factors to consider are the extent to which the functionality of C is dependent on the functionality of (P)
      * ex: Single line of code – maybe not derivation
  + A compiled library that is linked to an application at compile or run time
    - may depend on how dependent the linking program is on the library
  + Outputs from a program that contain elements of the producing program
    - ex: compilers or parser generators
    - program outputs generally aren’t considered derived works
      * ex: using IDE
* Copyleft and Incorporation
  + terms of a software licence can also be transitive (if a software artifact is incorporated into another software project, some or all of its licence conditions may also need be imposed on consumers of the incorporating software)
  + “viral licensing”

Warranties and liability

* Warranty asserts the period of time in which the provider is responsible for ensuring the software functions
* Liability clauses assert who is responsible should software cause damages

Enforcement

* Preventing a violation
  + implement technical mechanisms that are able to prevent a licence condition being breached
  + ex:
    - Reproductions conditions might be enforced through the use of digital rights management hardware and software
    - Making programs harder to modify by using just in time decryption and obfuscation techniques
    - Preventing unauthorised usage through licence servers or subscription mechanisms
* Discovering a violation and seeking redress

Reputational issues

* Requiring intrusive access to end user information
  + information is often valuable and part of the business model for the provider, but may cause reputational harm if widely reported
* Only issuing update to licenced users
  + software used to determine whether a licence has been procured or not may be defective, meaning legitimate users are denied the update
  + if the update addresses a security vulnerability it means that not all running instances of the software are patched
* Installing monitoring software to detect violations
  + likely to annoy legitimate users who may have good reason for the functionality
  + may expose their devices to vulnerabilities created by the monitoring software
* Modifying or disabling devices running unlicensed software.
* Taking legal action against licence violators

Considerations when choosing:

* Who are the target end users of your software?
* Do you want your software to be used by other software projects?
* Who are the customers? What are their resources?
* What benefits are you trying to achieve through the distribution of the software?
  + Revenue?
  + Reputation?
* What is the business model for the software?
* Who will own each distributed copy of the software?
* What are the risks of using your software?
* What restrictions do you want to impose on legitimate use?
* What warranties are your required to offer?
* How well equipped are you to continue to develop your software?
* Do you want to encourage others to make contributions to your software?
  + Who will they be? What capability are you looking for?
  + What incentives can you offer?
* What are the dependencies for your software?
  + What are their dependencies?
  + What licences have the distributors applied to them?
  + How rigorous have the immediate distributors been in asking this question?
* What are the threats to your business model?
* What will be the public perception of your licence?
* Are your licence conditions enforceable through legal action?
* How well equipped are you to enforce the conditions of the licence?

Example Licences:

* Creative Commons
  + Attribution
    - a consumer should acknowledge the contribution of the licensed work
  + ShareAlike
    - imposes a condition similar to copyleft
  + NoDerivs
    - prevents the creation of derivative works
  + NonCommercial
    - only permits the licensed work to be used in the creation of non-commercial works
* BSD Licence
* Eclipse Public Licence
* [GNU General Public Licences]
  + goal of the licence is to incentivise sharing and contributions to FOSS projects, by ensuring that the resulting artifacts remain in the public domain, as well as limiting liability and warranty as far as is possible
  + V3 – much longer
  + LGPL has a weaker notion of copyleft enabling consumers to link their code to LGPL artifacts without imposing the condition that the linking software must be distributed under the same conditions
* Microsoft Windows 10 EULA
* mac OS Catalina

Caveats

* Interpretation
  + Derivation?
* Technological evolution
* Jurisdiction

# Week 24: Start-up Growth Engineering 1

Potential for start-ups:

Chart

Description automatically generated with low confidence

* Exponential ex:
  + Twitter
  + Facebook
  + Instagram
  + SkyScanner
  + Slack
  + LinkedIn

Growth Engineering

* **Techniques for systematically introducing a new product idea into a large-scale market and driving it to scale**

Frequent graphs: “jumping the shark” (looks like shark fin) – sudden rise in use/interest, gradual but steady decline

Silicon Valley

* Generational, Systematised Innovation
* Semiconductors, hardware -> Microprocessor, PC, Software -> Internet, Platforms -> Mobile

Properties of Growth Engineering:

* GE creates and exploits **Compounding Growth** effects
  + Generic:
    - Arrow

      Description automatically generated
    - Ex:
      * WhatsApp users invite their friends <-> Some invitees sign up for WA
      * LinkedIn colleagues
    - Currency: **users**
  + Content-Driven
    - Uses **content** as currency
    - ex: LinkedIn
      * Users creating profile, indexed by Google, found on Google, people signing up and creating profiles
* GE models and optimises the entire **user journey**
  + Diagram

    Description automatically generated
    - Acquire – market and get users to sign up
      * Linear Marketing
        + Attract initial and further users
        + Paid, stories, magazines
        + SEO (Search Engine Optimisation), SEM, PR Events
    - Activate – build habits in users to use product (search for Google, buy for Amazon etc)
    - Retain – keep in habit state
    - Monetize – get them to subscribe
    - Refer new users – new source of acquisition

**Growth Model**

1. Linear Marketing
2. Optimise user journey
3. Generate referrals for Compounding Growth

# Week 25: Start-up Growth Engineering II

Retention

* Foundation of Compounding Growth
* Zero Churn
  + No users stop using a product
* 1% Churn
  + Slow, easy to improve product while active users decrease
* 33% Churn
  + Failure
* A trend, not a number
  + Cannot draw conclusions from 1 data point (80% after 4 weeks/50% after 3 months), need to see general trend
  + First part: Activation
    - Initial loss of active users due to them forming habits
    - Make it easier to form habits (UI, fewer obstructions, etc)
  + Then: Ongoing Retention
* Growth Model
  + Chart, bubble chart

    Description automatically generated

Optimising Customer Journey

**?print-pdf&showNotes=separate-page**

**Then CTRL+P for print to PDF**