FIT2101

Software Engineering Process and

Management



Agile and Software Process Models



**Topics Covered**

Features of Agile

What Agile Isn’t

Agile Process Models

Software Process Models

In 2001 leaders of lightweight process models met at a summit with which the result was the Agile Manifesto.

The manifesto is a statement of priorities and principles

**Priorities**

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

**Principles**

1. Early and continuous delivery
2. Welcome changing environments
3. Deliver working software frequently
4. Businessmen and devs work together daily
5. Build projects around motivated individuals
6. Convey information in a team face-to-face
7. Working software is the primary measure of progress
8. Maintain a constant pace indefinitely
9. Continuous attention to technical excellence and good design
10. Simplicity
11. Self-organising teams
12. Regular reflection on becoming more effective

**Features of Agile**

* Risk and risk management are important
* Development is conducted in short iterations
* Continuous customer involvement
* Only short-term plans are detailed
* Testing is integrated into the development
* Requirements, design, and implementation are continuous
* Small, self-organising teams work face-to-face

**What Agile Isn’t**

* Never writing documentation
* Code first, think second
* Ignoring architecture and design
* No measure or management
* Projects always succeed
* Anyone can write software
* Mind changes are cost-free
* Pre-agile knowledge is useless

**Agile Process Models**

* **Lean:** Inspired**​** by Japanese lean manufacturing methods
* **Extreme Programming (XP):​**Emphasised testing and pair programming
* **Test Driven Development (TDD):​**Write unit tests before features
* **Behaviour Driven Development (BDD):** TDD**​** with acceptance tests first
* **Scrum:** Work**​** is broken into sprints

**Software Process Models**

A software process model tells you which activities should happen and when. Systematic methods help to develop software on time, in the budget, and to specifications

**Ad-Hoc**

Ad-Hoc is definitively simple; you’d write code, edit it until requirements were met, and then delivered. As systems became more complicated ad-hoc could not be a viable option; there’s no documentation, testing, and unknown length of development.

**Waterfall Model**

1. **Requirements:** Find**​** out what stakeholders want
2. **Analysis: ​**Find out what needs to be done
3. **Design:** Determine**​** how to build the product
4. **Implementation:** Create**​** a working product
5. **Maintenance: ​**Repair or enhance
6. **Retirement:** Remove**​** from service

This allows development to be broken up, non-overlapping, document producing, and ensure testing and QA. Its document-driven approach makes it expensive to maintain, so waterfall is inappropriate for projects predisposed to change.

If a requirement has changed once maintenance is reached, return to analysis and redo the process.

**Prototyping and spiral**

The idea of prototyping is building a mock of the project to nail down requirements. This is expensive, however, and may require many prototypes, thus spiral was created.

Spiral consists of stages which loop, and become broader, until the development of the product

1. Risk analysis and prototype
2. Build and validate
3. Plan next phase
4. Determine objectives, alternatives and constraints

**Agile**

The most recent of all process models which is unique in its approach to risk; waterfall eliminates change, spiral manages change, but agile embraces change.

Scrum



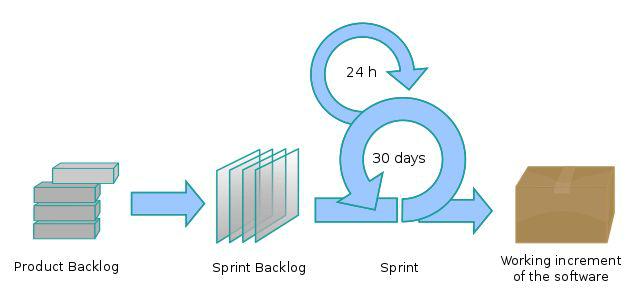
**Topics Covered**

Setting Up Your Project

Product Backlogs

Sprints

Scrum is a way of organising small teams of people to get something done. It is one of the Agile process models.



**Setting Up Your Project**

* **Stakeholder Management:** You**​** need to ensure everyone is aware of all thepeople who have an interest in your project. You need to know who needs to be kept informed about the project.
* **Team Organisation: ​**Who is on the team? What roles will they play? What roleswill they abide by?
* **Make Fundamental Architectural Decisions: ​**Some decision have a biggerimpact than others
* **Decide on a Technology Stack: ​**Decide the collection of software, languages,and tools for the project
* **Establish Who & What the Product is for**
* **Manage Risks**
* **Create Initial Product Backlog**

These decisions are typically made during the inception phase

**Product Backlogs**

A list of Product Backlog Items (PBIs) that need to be done before the end of the project.

It fulfils the same roles as the Software Requirements Specifications, however

* Not limited to customer wants
* Features are less detailed user stories
* Will change over time
* Can exist in any format

A product backlog should be **DEEP**​

* **Detailed Appropriately: ​**Detailed based on priority
* **Estimated: ​**High priority tasks should have an estimated completion time forsprints
* **Emergent:** It**​** changes over time
* **Prioritised: ​**High priority tasks should be done first based on importance,effort and risk

For each product backlog item, it should list

* A description
* An estimate of effort
* A priority

**Sprints**

All Scrum development work is done in sprints. A sprint is a time-based iteration. The features you add need to be small enough to fit many into a sprint, if not, you need to break features into components

The goal of each print is to produce a potentially shippable increment.

During a sprint planning meeting, the team collectively decides how much of the product backlog is implemented.

A daily standup meeting ensures team members know what everyone is doing and who needs help

After each sprint, the team conducts a product review, sprint retrospective (to use if the process can be made more efficient), then backlog refinement.

* **Story Points: ​**A unit of measurement of the progress through a sprint. Theunits with which user stories are measured
* **Velocity: ​**The number of story points usually completed in a sprint
* **Burndown Chart: ​**Plots the number of story points remaining in the backlog

**Sprint Planning**

This is a two-phase process

* **Phase 1:** Product**​** owner goes over high priority items to answer questions andget feedback. The team breaks down high-level stories into small tasks
* **Phase 2:** The**​** team decides which tasks to complete in the sprint

A 15-minute meeting to minimise wasteful incentives. The team answers three questions

1. What did you do yesterday?
2. What are you doing today?
3. What is preventing these things?

**Product Reviews and Retrospectives**

* **Product Review**
  + At the end of each sprint, the team reviews the status of the product.
  + This includes a demo of what it can do and a conversation between the product owner and the team.
* **Retrospectives**
  + The team discusses what went well and what went poorly to refine the team’s process

**Backlog Refinement**

Before the next sprint can begin the product owner needs to read the next few PBIs. The team can create some informal estimates to help the owner understand if a task is difficult.

This is also an opportunity to go over the backlog and check everything is relevant.

**Scrum Stakeholders**

* **The Team**
  + Small, self-managing, and cross-functional
  + Focuses on one task at a time
  + Ideally working in the same room
* **Scrum Master**
  + Helps the team come to a consensus rather than issuing commands
  + Stops interferences in the project
  + Organise and facilitate Scrum activities
* **Product Owner**
  + Decides the contents of the product backlog and their priorities
  + Responsible for maintaining the product backlog
  + An expert on client/customer wants and needs
* **Internal Stakeholders**
  + Other stakeholders within the organisation’s environment

Risk and Risk Management



**Topics Covered**

Risks

Risk Monitoring and Risk Mitigation

Spikes

**Risks**

A risk is a situation involving danger. Risk involves uncertainty if a danger is certain, then you’re dealing with a problem.

**Dealing With Risk**

Mistakes happen all the time, so we need a strategy to deal with risk. Recall we aim to deliver

* A working system
* On time
* For an agreed price

**Effects of Risk**

* Timeliness
* Cost
* Fulfilment of requirements
* Usefulness to customers
* Legal compliance

This is just the surface of what risks can affect.

**Kinds of Risks**

* **General Risks**
  + Some risks can affect any project
    - Loss of team member
    - Change of requirements
    - Natural disaster
* **Specific Risks**
  + Some are project specific
    - Tools used are inadequate
    - Requirements misunderstood
* **Technical and Organisation Risks**
  + Arise from the technical aspects of a project, due to lack of knowledge
* **Organisation Risks**
  + Relate to organisational or process aspects of your project

**Measuring Risks**

* **Impact: ​**How badly the risk will affect a project from extreme or minimal
* **Likelihood: ​**How likely it is for a risk to happen from 0% to 100% chance

**Identifying Risks**

This should be considered in inception and during development. It helps to consider that parts that might cause risks

* Employees
* Customers
* Hardware/Software
* Legal
* Physical

**Documenting Risks**

Risks should be written in a risk management plan

* **Risk Matrix**
  + A table of probability and impact to determine which risks to tackle first

**Risk Monitoring and Risk Mitigation**

For each risk you identify, you should develop a risk monitoring strategy. To take action against a risk you need to know when to do it.

**Risk Mitigation**

We mitigate risks by making it less likely or less severe Approaches fall into

* **Avoidance: ​**Don’t do the risky thing
* **Reduction: ​**Minimise the likelihood or impact of the risk
* **Transfer: ​**Make it someone else's problem
* **Acceptance: ​**Take the costs of the worst case

**Spikes**

A spike is a way to deal with risks in a technical environment. Technical risks are from lack of knowledge, so one way to gain knowledge is to do an experiment. A spike is a small programming experiment that answers a technical question:

* How do I...?
* How long does it take to...?
* Is it easy to...?

**Spikes During Development**

In Agile new technical risks can crop up at any time. You aren’t guaranteed to understand the technical requirements and the environment may change.

This means you can’t assume you’ll do all spiking before development. You need some way to integrate spikes into regular sprints.

If you’re managing functional requirements with user stories, introduce a spike story.

**Documenting Spikes**

Spikes need to be written up. A spike for reducing a technical risk will generate example code, this needs to be documented for others to understand it. You should also document what you’re trying to achieve and whether you achieve it.

Inception



**Topics Covered**

Agile Inception Deck

Project Plan

During inception, there are two goals

* Alignment
* Expectation Setting

The team needs to have the same understanding and expectations of the project before commencing

**Agile Inception Deck**

**Why Are We Here**

To build a great product you need to know why you’re building it. Knowing and understanding the drive of a project is going to give you insight into making trade-offs during development.

**Elevator Pitch**

A good elevator Pitch tells people what your product is, who it’s for and why it’s special in a small amount of time.

**Design a Product Box**

Thinking about your product from the customer’s perspective helps get you in their

head and build the team

Just ask

* Why will people buy the product
* What slogan captures the spirit of the product

**Create a Not List**

Saying what you’re not going to do is powerful. It eliminates waste by letting the team focus on the stuff that it clearly is.

**Meet Your Neighbours**

The project community is always bigger than you think; it’s more than you and your team.

Identifying neighbours will allow you to know who to meet as a courtesy and to prepare for their arrivals when needed.

**Show the Solution**

If you need to use certain tools or architectures you need to let others know. And, you should also let them know if you don’t have all the answers.

**Ask What Keeps Us Up At Night**

You need to identify what risks are worth worrying about and reserving the ones that aren’t

**Size it Up**

You need to figure out an estimation of how long the project is going to take. Some high-level story planning and estimation will need to be done. This is done to determine just how feasible the project is with the resources you have.

**Be Clear on What’s Going to Give**

All projects have parameters of what’s important, such as scope, money, time, or quality.

You need this conversation to decide what can flex or has some wiggle, or what should be static.

You should also decide what’s going to knock it out of the ballpark.

**Show What it’s Going to Take**

You need to answer two questions

* How long will it take
* How much will it cost

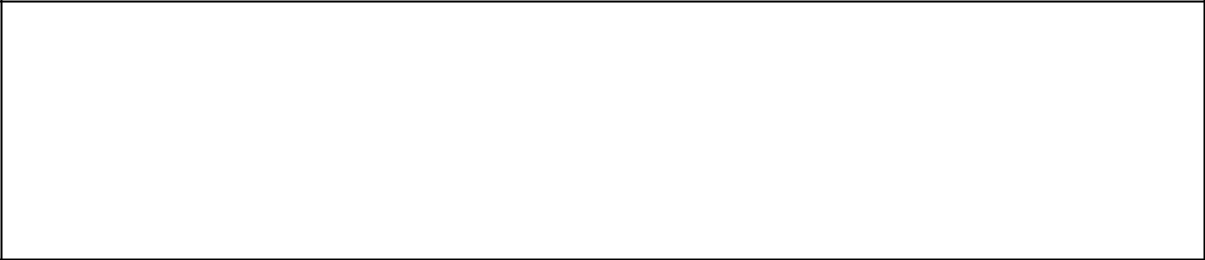
All you need to do is some simple math to see if the project’s feasible.

**Project Plan**

Although Agile is focused on embracing change rather than following a plan, making certain decisions at the start of a project ensures that team members are on the same page. The contents should derive from the following

1. Summary of the project
2. Deliverables and their deadlines
3. Project Context
   1. Process Model
   2. Tools and Techniques
   3. Roles and Responsibilities of the team
   4. Other groups the team may need to work with
   5. Contact Details
4. Vision Statement
5. Time and Task Management
   1. Task Allocation and Time Tracking
6. Architectural Decision

Story Mapping



**Topics Covered**

Story Workshops

Identifying User Stories

Release Planning

Story Mapping

User Stories

**Story Workshops**

Story writing workshops are gatherings of users, developers, project managers, and other stakeholders. The objective is to write as many stories as possible. The primary goal is writing stories, they can be refined and prioritised later.

**Conducting Workshops**

* **Brainstorming**
  + Get the team together with sticky notes and stationary
    - It’s a good idea to write personas on some display so they will be on people’s mind as they work
  + The Scrum Master should facilitate
  + All participants come up with ideas about what the product might do
  + During brainstorming only come up with ideas
    - If people get sidetracked the Scrum Master should step in
    - If people get slack, use personas as inspiration
    - If ideas are unreasonable they will be sorted out with the product owner during maintenance
  + As stories are created they can be sorted into themes or program functions

**After the Workshop**

The team and product owner will need to work together to estimate how large and valuable each story is.

Stories should be negotiable; they can be split up, fixed, defined, or consolidated. It doesn’t need to be right the first time.

* **Now Versus Later**
  + You can’t avoid this work, it’ll eventually be completed during backlog maintenance.
  + It may be a good idea to have a schedule spike to clarify mockup stories.
* It’s usually a bad idea to fully complete or leave everything in the backlog. You should begin each sprint with twice as many stories ready than you need and focus on what’s most important

**Release Planning**

Requirements are flexible in Agile so it’s hard to predict what you’re going to do, and therefore, it’s hard to tell stakeholders time estimated.

You need a bit of planning to get a rough idea of where you’re heading in future iterations.

Release planning is a meeting held after features are identified, prioritised, and estimated to establish a release schedule and determine which features can be released in each release.

**Story Mapping**

Collect related stories into epic. Each part of the epic can be mapped

1. Collect stories in a format that can be rearranged
2. Find space to arrange stories
3. Identify themes along the top row
4. Arrange theme’s components below it in the order of implementation

**Iteration Planning & Release Planning**

Story maps let you plan which functionality will go through each iteration.

It also gives you a place where stories can be reorganised to changing desires of the product owner.

They also allow planning for the functionality of the release and estimating a schedule until delivery of the release.

**Flexibility**

Story maps may be made early on but the idea is it’s easy to move stories around.

**User Stories**

User story requirements involve the INVEST criteria

* **Independent:** Self**​**-contained and can stand alone
* **Negotiable: ​**Easy to change, discard, or replace
* **Valuable:** Describe**​** something the user cares about
* **Estimable: ​**Should be well understood/defined enough that the team canestimate implementation time
* **Small: ​**Able to make confident estimates
* **Testable:** Be**​** able to be sure when the story is implemented

**Structure**

As a <role> I want to <do something> so that <business value>

**Common Problems**

* **Excessive ‘So That’**
  + The team typically expects requirements before the conjunction ‘so that’. You can usually identify this problem when the ‘so that’ section is complex or has multiple parts. Requirements should me moved to before the ‘so that’ section and broken down if needed.
* **Odyssey**
  + A story larger than an epic; so compounded or convoluted that it has no value. Odysseys are usually difficult to estimate and need to be split into many smaller stories.
* **Waterfall**
  + Stories which only consider technical parts of development rather than the product. Usually in the form of a developer story. These should be used as a technical story.
* **Rigidity (Inflexible)**
  + Stories with too much specificity, leaving little room for creativity, better solutions, or dynamic scope. Try to generalise inflexible requirements
* **For Whom? (Non-User)**
  + Stories which have users so specific that it excludes other users which may also find use in the story. Try to encapsulate all people within the rle, but avoid making it too broad and vague.
* **Parakeet Value**
  + Stories which restate the requirements in the ‘so that’ section, making the goal of the story unclear. Try and analyse what value the story will provide and consider that in the ‘so that’ phrase.

Project Inception - Requirements



**Topics Covered**

Requirements in Traditional Process Models

Requirements in Agile Process Models

Definition of Done

User Stories

Requirements are what determine whether your software is complete and correct. Requirements are determined by the product owner, but other stakeholders may have input too since they tell you what behaviours are correct, they’re important to testing.

**Requirements in Traditional Process Models** Traditional models are text-based

* Requirements must be written down
* The written description must contain all the information necessary to begin design and implementation
* Typically completed before design begins

**Requirements in Agile Process Models** Complete requirements aren’t made in advance. The product owner is involved in development.

* Get something up and running as quickly as possible
* Get product owner to pick which features to add next
* Try to add as much value per iteration
* Repeat for each sprint

Collaboration is preferred over exhaustive documentation.

**Definition of Done**

Each project needs a definition of done

* What has to happen before a feature is considered complete and ready to roll out
* Usually includes reviews and tests

Sprint Planning



**Topics Covered**

Sprint Planning Meeting

Conducting Sprint Planning Meeting

Tasks

At the start of every sprint the team needs to decide what it aims to achieve during that sprint, that is, what stories the team will implement. This involves selecting stories from the backlog, breaking them down into tasks, and putting them in the sprint backlog.

This process is print planning.

For sprint planning there must be stories ready to implement, stories shouldn’t be removed from the backlog until done.

**Sprint Planning Meeting**

The whole team needs to plan a sprint

* Product owner to explain the client’s values
* Team members to learn values and explain technical constraints
* Scrum master to facilitate meeting

This meeting should be time-based

**Objectives**

* Create the sprint backlog
* Organise team to plan how to get backlog done

**Sprint Goals**

A short description of the sprint focus

**Conduction Sprint Planning Meeting**

There are two parts

1. Understanding user stories and choose stories
2. Figure out how to organise development to complete stories

**Preparing Stories**

You shouldn’t select stories until

1. INVEST is met
2. The team understands the client’s needs
3. They’re estimated
4. Their priority is known

The team needs to ensure there are enough stories ready by

* Reviewing unreviewed high priority stories
* Running estimation sessions
* Confirming story importance with the product owner

**Selecting User Stories**

Stories should be selected to provide the most velocity, that is, the number of story points in a run

* **Stable & Unstable Velocity**
  + If your velocity is stable you can use last iterations velocity to estimate story points you need
* **Tech Stories**
  + You may need to consider refactoring needs for technical debt

**Detailed Planning**

After deciding stories the team needs to figure out how it will be implemented

* Break down the story into component tasks
* Decides who will perform each task
* Ensure members aren’t over/under loaded during the sprint

**Tasks**

A task is simply a thing that a team member does. It typically forms part of a story and involves one technology.

Estimates



**Topics Covered**

Absolution & Relative Estimation

Difficulty of Estimation

Ideal Time

Story Points

Software estimation is difficult, one good way to know how long a job will take is to look at similar jobs in the past, but we seldom write the same program twice making experience a poorer guide than other professions. Agile estimation is especially difficult

**Absolute & Relative Estimation**

* **Absolute:​**​Awill​take​xdays,​Bwill​take​ydays​
* **Relative:​**​Awill​take longer than​B

Absolute is usually less accurate than relative

**Difficulty of Estimation**

* Don’t have complete control over the project and its environment
* All projects are different and difficult to prepare
* Human cognitive bias
* Dunning-Kruger Effect: You don’t know how much you don’t know
* Hofstadter's Law: It always takes longer than you expect, even when considering Hofstadter’s Law

**Ideal Time**

Estimate your tasks in terms of the ideal time they would take if nothing goes wrong. It’s easy to understand, takes into account cognitive bias, but doesn’t translate directly into calendar time.

**Story Points**

Agile teams estimate the size of user stories with a number of points. They’re a non-time estimation. Small stories are easier to estimate than larger ones, so often points are restricted to numbers that are further apart as they get larger (eg Fibonacci Sequence). If stories are too large they should be broken down.

**Velocity**

A measure of the speed of progress. After some time expected velocity moves from guessing to a steady development

**Sprint Burndown Charts**

The progress of sprints can be made into a graph showing expected progress and actual progress.

Backlogs & Backlog Refinement



**Topics Covered**

Contents of the Backlog

Backlog Refinement

A backlog is a todo list. It contains user stories and their priorities. Backlogs are usually drawn on a whiteboard or spreadsheet. If you need to store more information about a story it’s usually kept elsewhere.

In a typical project, you’ll have at least two backlogs

* The product backlog of stories
* The sprint backlog of stories for the current sprint

**Contents of the Backlog**

User stories are valuable towards users but they don’t see all of the activities done for them

* Spikes
* Maintenance issues
* Refactoring

Usually, these are also stored in the backlog, as you need to schedule them

**Product Backlog Item (PBI)**

A user story is only one type of PBI, there’s no definite format of a PBI, it can be whatever your team decides it to be.

All they need it a priority and the ability to be considered complete

**Backlog Refinement**

An activity designed to make sure the backlog is of high quality.

The team and product owner look through the backlog to verify items are correct and complete.

Going through the backlog helps the team focus on upcoming stories

**Attendants of Backlog Refinement Meeting**

* The product owner must be there to prioritise, update, and correct items
* Scrum master must be there to ensure a smooth meeting
* Team members can be there to gain knowledge and provide technical information

**Considerations of Backlog Refinement Meetings**

* **Value of Stories**
  + The product owner should decide if stories are worth keeping. Sometimes they become redundant or do not reflect needs
* **Adding Stories**
  + The product owner may have new ideas for functionality or there are new opportunities
* **INVEST**
  + Stories should be reviewed to match the INVEST criteria
* **Estimations**
  + Some stories may be estimated if the team wishes
* **Long-Term Planning**
  + The product owner may talk to the team about possible themes for sprints
* **Communication**
  + Backlog refinement should show how the client’s wishes and priorities are changing over time, and what we’re likely to do over the next few months

Reviews and Retrospectives



**Topics Covered**

Sprint Review

Retrospectives

**Sprint Review**

A meeting where the current state of the product is established. It’s done at the end of a sprint.

It’s a good opportunity for client representatives to collaborate with the team.

**Conducting a Sprint**

1. Product owner explains what has and hasn’t been done in a sprint
2. Team talks about how a sprint went
3. Team and product owner demonstrate completed stories and answer other stakeholders questions

**Retrospectives**

A meeting to consider the team's process at the end of a sprint. Think about your practices and policies and whether they’re effective.

**Individual Reflection**

Before the meeting spend a little time noting

* What worked
* What didn’t work
* Suggestions for improvements

**Share Information**

* Go through people’s lists
* Focus on the meeting if actual velocity differs significantly from expected
* Look at proposed improvements from last time
* Brainstorm solutions or consider noise/errors in estimates

Leadership and Accountability



**Topics Covered**

Leadership

Accountability

**Leadership**

There’s a distinction to be made between leadership and management. Management is about the organisation: making sure things are done, how things are done, and policies are followed. It’s their job to ensure the team has their resources. Leadership is about interpersonal skills: motivation, communication and empowerment

**What Leaders Do**

* Don’t do all the work
  + Delegate: hand parts of your responsibilities to others
* Listen
  + Listen to the team and demonstrate that you’ve heard
* Motivate
  + Team members should feel
    - Trusted
    - Respected
    - Supported
    - Safe
* Take Responsibility
  + Take the blame for one’s mistakes

**Accountability**

To be accountable is to take responsibility for your actions. That means credit and blame should go where it’s due.

Good software processes support accountability by knowing

* Who worked on what
* How each team member contributed to the development

**Accountability and Git**

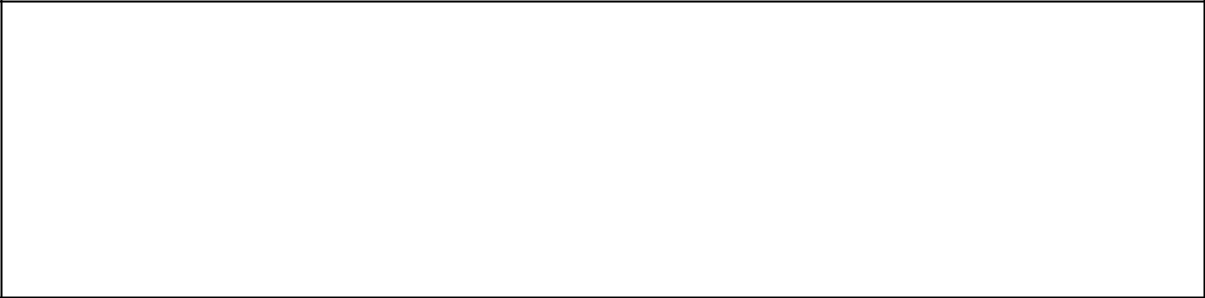
Git records each change that’s made and who made them. With this, you can access the most recent change and older states

* **Explain Changes**
  + With each commit, you should add a comment to describe what you did. This lets your teammates understand your thought process and improve accountability.
  + Write good comments by considering why teammates are looking at your code

**Other Accountability Mechanisms**

There are other tools such as workflow automation software, issue/bug trackers, etc.

Teams and Teamwork



**Topics Covered**

Teams

Conway’s Law

Attributes of a Poor Team

Collaborative Teams

Team Development

Teamicide

**Teams**

A team is a group, a collection of people, that are cohesive and focused on a common purpose or goal.

Software engineering is about delivering systems that are large, complicated, or mission critical. To make sure these systems work you need lots of people working together effectively.

**Conway’s Law**

“Organisations which design systems are constrained to produce designs which are copies of the communication structures of these organisations.”

* To create a software module developers must communicate and collaborate closely
* It’s harder to collaborate outside your team, and easier to collaborate with teammates

**Attributes of a Poor Team**

* **Autocracy**
  + Leaders hand out orders for a team to do
  + Little opportunity for team collaboration
  + Poor morale
  + The leader is a single point of failure
* **Anarchy**
  + No leader
  + Nobody knows what to do
* **Democracy**
  + All decisions are made collaboratively
  + There can be a lot of decisions
  + The delusion of individual responsibility
    - Nobody is at fault for problems
    - Less incentive for good decisions

**Collaborative Teams**

Effective teams tend to have a flat structure, with each member having different responsibilities, members make decisions within their expertise

**Team Development**

Teams go through five stages

1. **Forming**
   * The team has just been put together
     + Team members may not know each other
   * Little progress is likely to be made on the team’s task
   * The team hasn’t learnt how to work with each other yet so the group dynamic doesn’t involve collaboration
2. **Storming**
   * Members may come into conflict
   * Productivity rises slowly as conflicts are resolved
3. **Norming**
   * Most problems are resolved
   * Trust, respect, and identity within the team
   * Standards begin to develop
   * Individuals take on unofficial roles
4. **Performing**
   * Everyone understands the team
   * Trust between teammates
   * Team functions at a high level
5. **Adjourning**
   * The team completes it’s tasks and breaks away

**Teamicide**

Bad practices in making teams

* **Defensive Management:** If**​** management can’t trust a team they lose morale
* **Bureaucracy:** Hard**​** to be enthusiastic about mindless paperwork
* **Physical Separation: ​**Teams work better with interaction
* **Fragmentation of Time:** Hard**​** to concentrate when switching between tasks
* **Quality-Reduced Product: ​**Demotivating to ship a poor product
* **Phony Deadlines:** Lost**​** trust and gain stress when deadlines don’t matter
* **Clique Control: ​**Management has policies which interfere with teams

DevOps



**Topics Covered**

What is DevOps

How DevOps Works

What DevOps Isn’t

Tools and Practices

**What is DevOps**

DevOps is the unification of developers and operations staff. Usually, developers and IT operations staff are in different teams; they have different responsibilities. Dev is responsible for creating new software, Ops is responsible for managing IT infrastructure.

**Conflicting Objectives**

Friction between Dev and Ops is very common. If siloed, they tend to be natural enemies; they each blame each other.

It’s the developers’ responsibility to create new functionality and get it to the end users quickly, but it’s the operators’ responsibility to keep the company’s systems stable.

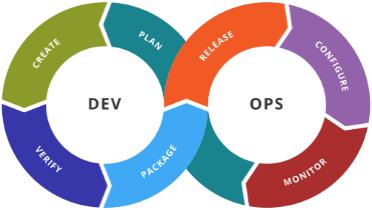
When a new version of software is produced and deployed there’s a risk that it won’t work. So it’s in Ops’ interest to say no to Devs’ request for the deployment, while it’s in Dev’s interest to deploy enough functionality.

DevOps arises from a single insight: *it*​ *is everybody’s responsibility to support the* *business’s goals and activities by enabling change.​*Everyone should share in themanagement of risk; Dev and Ops need to become unified.

**Breaking Down Silos**

DevOps means breaking down silos so that operators, developers, and QA staff all collaborate.

**How DevOps Works**



* Devs and Ops jointly plan the software
* Devs create, verify, and package the software
* Ops manage releases, configuration, and monitoring of the software
* The next planning phase considers the observations from monitoring and experience from deployment and configuration

Dev doesn’t ignore Ops concerns, which makes it more likely for software to be kept running without incidents.

**Infrastructure as Code**

DevOps aims to make it easier to deploy software frequently, requiring deployment and maintenance to be less risky and predictable. One way to do this is to automate as much as possible, referred to as infrastructure as code.

Instead of managing hardware, software, and networks by hand, devs can write programs these tasks. This means writing scripts to control company hardware and software.

Devs can apply their skills to produce and maintain these scripts. This also gives developers a more sophisticated understanding of how software is maintained and the risks in deployment.

**What DevOps Isn’t**

DevOps is not

* A process model (though it fits better with some models than others)
* A toolchain (though it is supported by tools)
* A methodology

DevOps requires a cultural shift in production and deployment the same way moving from a traditional process model to agile would.

DevOps extends the idea of the Agile cross-functional to include more of the organisation in development.

**Tools and Practices**

**Version Control**

Using a mature version control system lets you distinguish between the versions that devs are working on.

What is new to the DevOps approach is to put Ops resources such as scripts and configuration under version control.

**Continuous Integration and Continuous Deployment**

A continuous integration server monitors your version control system. When the code is checked in, the server builds it and runs unit tests. If any of this fails the server notifies the team that there’s a problem.

Under continuous deployment, changes to the code are also deployed. A staging server is set up to be as close as possible to a production environment so that it can go through acceptance testing before being fully deployed.

It’s possible to configure the system to deploy production if all tests pass, meaning new features pushed to master are given to users immediately, though you need to be confident that all your testing is adequate.

**Virtualisation**

* **Virtual Machines**
  + A virtual machine is a piece of software that emulates a computer. Instead of going through the same process each time you roll out equipment, you can install an OS on a VM with your software and distribute the image around.
* **Virtual Servers**
  + Instead of setting up native OSs on your servers, you can copy VM images to servers. You can run your system on equipment redundant to the company to add more capacity if the network is at high capacity.
* **Containers**
  + VMs have a potential problem, every VM has its own copy of the OS, meaning you’re storing many copies of the same files, taking up a lot of space. A containerization system solves this problem. Instead, applications are isolated in containers, which share an operating system, but each has their own copy of the application.

Non-Functional Requirements



**Topics Covered**

Difficulty of Usability

Functional Requirements

Non-Functional Requirements

Requirements Elicitation

**Difficulty of Usability**

* Stakeholders want their software to be usable, but usually, don’t know what it would look like
* Different stakeholders have competing needs
* Hard to put usability in a user story
* How do you acceptance test for usability
* How much are clients willing to pay for it

If your stakeholders can tell you what they need you to build, you can probably build it

* They have to be able to focus where usability problems may lie
* They have to be able to come up with a workaround

**Functional Requirements**

The functional requirements of a software system are the requirements that tell us what the software was supposed to do.

**Non-Functional Requirements**

Usually properties of the system as a whole rather than a single component

**Usability as a Non-Functional Requirement**

Usability is a property that emerges from the system as a whole, however many aspects of usability can be expressed as functional requirements

**Requirements Elicitation**

How do you elicit requirements from people who aren’t consciously aware of them

**Asking Stakeholders**

Construct interview protocols and questionnaires to prompt them to think about the qualities of the software.

**Try and See**

Produce a prototype to get feedback from users, take their feedback into account in the next iteration.

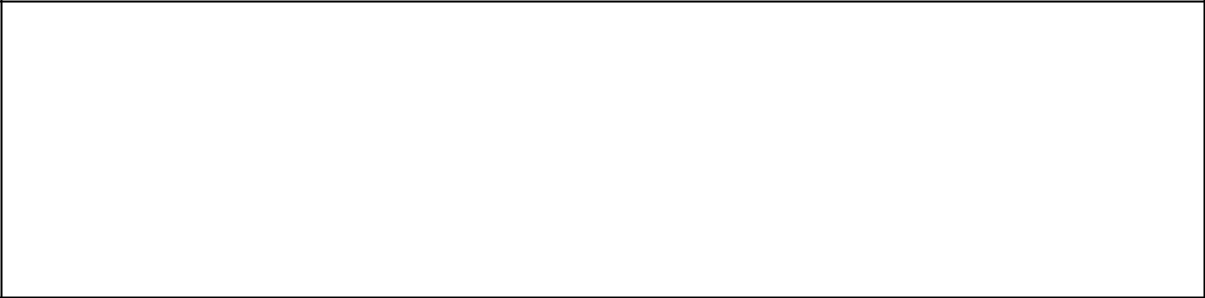
**Domain Knowledge**

A domain is a sphere of knowledge. As you develop experience in a particular domain you’ll get a sense of where the risks and requirements lie

**Hire an Expert**

For some kinds of non-functional requirements, you’ll require help from outside the team

Metrics and Evaluation



**Topics Covered**

Measuring Your Work

Metrics

LOC, SLOC, KLOC

Function Point Analysis

Story Points

Metrics and Targets

**Measuring You Work**

If we want to be able to produce high-quality software efficiently we need to be able to measure both the quality of our work and the efficiency of our process

**Why Measure**

* Deliver high-quality work, on time, and in budget
* We need to know
  + Quality of our software
  + When we’d deliver
  + Cost of software
  + Cost of maintenance
* We measure these to
  + Control them better
  + Predict them better
  + Understand our progress
  + Target resources effectively

**Metrics**

A metric is an outcome of measuring something. It’s the result of applying a well defined, repeatable process. It’s a quantitative measure and gives us information about something we care about.

For a metric to be useful it must

* Be easy to compute
* Be computable before we need it
* Be strongly related to the thing you’re trying to measure

**Choose Metrics Well**

* Some things we care about are hard to measure, sometimes the information isn’t helpful. You should collect information you need, not what’s easy
* We need to collect information about our process, product, and project to interpret them

**LOC, SLOC, KLOC**

* Lines of Code (LOC)
  + SLOC (Source Lines of Code)
  + KLOC (Kilo LOC)
* More than just lines in a file
  + What about comments, blank space, and multi-line statements

**LOC as a Metric**

LOC is easy to compute and automate

* Useful if you’re trying to measure code size
  + But why do you want to know the code size

**Problems With LOC**

Each LOC counts up the same, but not all LOCs are equal. We care more about functionality than code size.

**Function Point Analysis**

**Function Points**

A metric that involves computing an estimate of functionality rather than just code size

Function Point Analysis

1. Identify functions of the system
2. Estimate the complexity of each function
3. Add estimates together
4. Compute a Value Adjustments Factor (VAF) based on the importance to the system

**Advantages**

* FPA is standardised by ISO/IEC
* You don’t need to code
* Independent of programming, based on requirements

**Disadvantages**

* Designed for business information systems, doesn’t adapt well to other domains

**Story Points**

Like FPA but estimated

* Canc compute other metrics like velocity

**Metrics and Targets**

Metrics are useful for finding out about your project, product, and process, but beware of using them as targets or KPIs

* **Key Performance Indicators (KPI):** Measure**​** of how well an individual or teamis doing
* **Goodhart’s Law: ​**When a measure is used as a target it ceases to be a goodmeasure

Ethics in Software Engineering



**Topics Covered**

Ethics in Software Engineering

**Ethics in Software Engineering**

Software engineers shall commit themselves to make the analysis, specification, design, development, testing, and maintenance of software a beneficial and respectful profession.

In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following eight principles

1. **Public: ​**Act consistently with public interest
2. **Client & Employee:** Act**​** in a manner that’s in the best interest of their clientand employer consistent with the public interest
3. **Product: ​**Ensure products and related modifications meet the highestprofessional standards possible
4. **Judgement:** Maintain**​** integrity and independence in their professionaljudgement
5. **Management:** Subscribe**​** to and promote an ethical approach of softwaredevelopment and maintenance
6. **Profession: ​**Advance the integrity and reputation of the profession consistentwith the public interest
7. **Colleagues:** Be**​** fair and supportive of colleagues
8. **Self:** Participate**​** in lifelong learning regarding the practice of their professionand shall promote an ethical approach to the practice of the profession

Limitations of Agile



**Topics Covered**

Causes of Project Failure

When Not to use Agile

Agile Alternatives

**Causes of Project Failure**

**Higher Management Doesn’t Get It**

* It’s not enough to put the team through a Scum training course.
* The rest of the organisation has expectations for reporting, budget, etc.
* Senior management might impose un-Agile conditions
  + Frequently written reporting
  + Frequent personnel changes
  + Insisting on micromanaging the process

**Team Doesn’t Collaborate Well**

Agile teams are self-organising so team members need to communicate amongst themselves. Lone wolf developers will find this challenging and so will developers who are unduly competitive or don’t know their teammates

**Customer Isn’t Interested**

* Agile processes require frequent input from a customer representative
* If a customer doesn’t want to hear from you again, Agile will fail
* If the customer wants a requirements document Agile methods won’t work

**Overplanning**

If you try to plan as religiously as traditional development you’ll lose all the benefits

**Unreasonable Expectations**

* Expecting faster development
* Expecting to cope with no documentation or planning

**Scope Creep**

Agile puts less effort into nailing down the requirements at the start of the project. If the customer’s unreasonable or over-enthusiastic it can lead to scope creep. The team needs to say no to some stories if necessary.

**When Not to use Agile**

* Bad for your organisation
  + Culture is opposed to uncertainty or change
  + Reporting requirements
  + The structure requires a traditional project management
* Bad for your customers
  + Can’t spend time with developers
  + Require a predictable process
* Bad for the product
  + Needs a large team
  + Has stringent quality requirements
  + Requirements are known in advance
* Bad for the team
  + Can’t meet frequently
  + Very large
  + Don’t know each other
  + Lacks maturity to self-manage

**Agile Alternatives**

**Rational Unified Process (RUP)**

A family of iterative heavyweight process models that make extensive use of UML for documentation

* **Phases**
  + **Inception: ​**Define scope, elicit requirements, determine risks
  + **Elaboration: ​**Figure out the architecture
  + **Construction: ​**Develop the software
  + **Transition: ​**Get software ready for release and deploy it
  + RUP iterations take place in these phases
* **Principles**
  + Develop iteratively with risk as the primary iteration driven
  + Manage requirements
  + Employ a component-based architecture
  + Model software visually
  + Continuously verify quality
  + Control changes
  + These are all compatible with Agile

**Model Based Software Engineering (MBSE)**

All approaches to software engineering use models. MBSE is centred around models.

The idea is to generate design from requirements, then automatically generate code from the design.

This works well where the models are a good fit for the problem and where quality requirements are high and need to be proven to be met. Not ideal if you need control over implementation details.