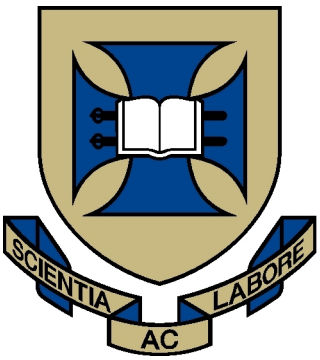
Daniel **Fitz**
43961229



University of Queensland

CSSE3002 – The Software Process

Lecture Notes

Table of Contents

Software Engineering

- Application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.
  - That is, the application of engineering to software.
  - IEEE Standard 610.12-1990
- Concerned with theories, methods and tools that enable professional software development.

“The topic that we call software engineering is both exciting and frustrating. Exciting because it draws on many technical disciplines and provides a harness that binds each discipline to the next. Frustrating, because it demands knowledge in a multitude of topic areas and seems to be infinitely expandable.” - Roger Pressman, 1992

Software Engineering Process

A structured set of activities followed to develop a software system

- Tools
- Methods
- Practices

Well Engineered Software

- Usable
- Dependable
- Maintainable
- Efficient
- How do costs come into this?
  - Trade-offs may be involved
    - appropriate
    - cost-effective

Process Models

- Abstract representation of a process
- Plan Driven
  - Structured / Traditional
- Incremental
- Agile
- Lean
- Formal

Plan Driven Processes

Waterfall

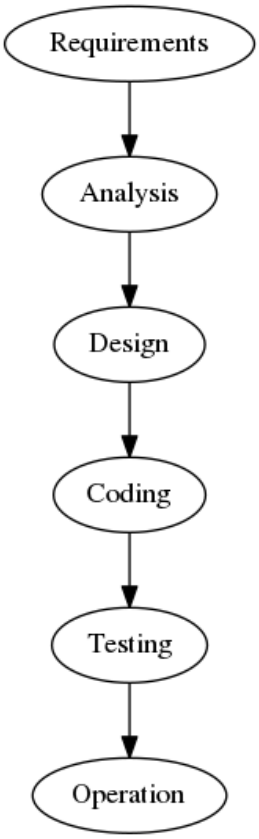


Figure 1: Diagram explaining Waterfall

- Introduced iteration between phases
- Prototyping
  - Requirements
  - Design

VModel

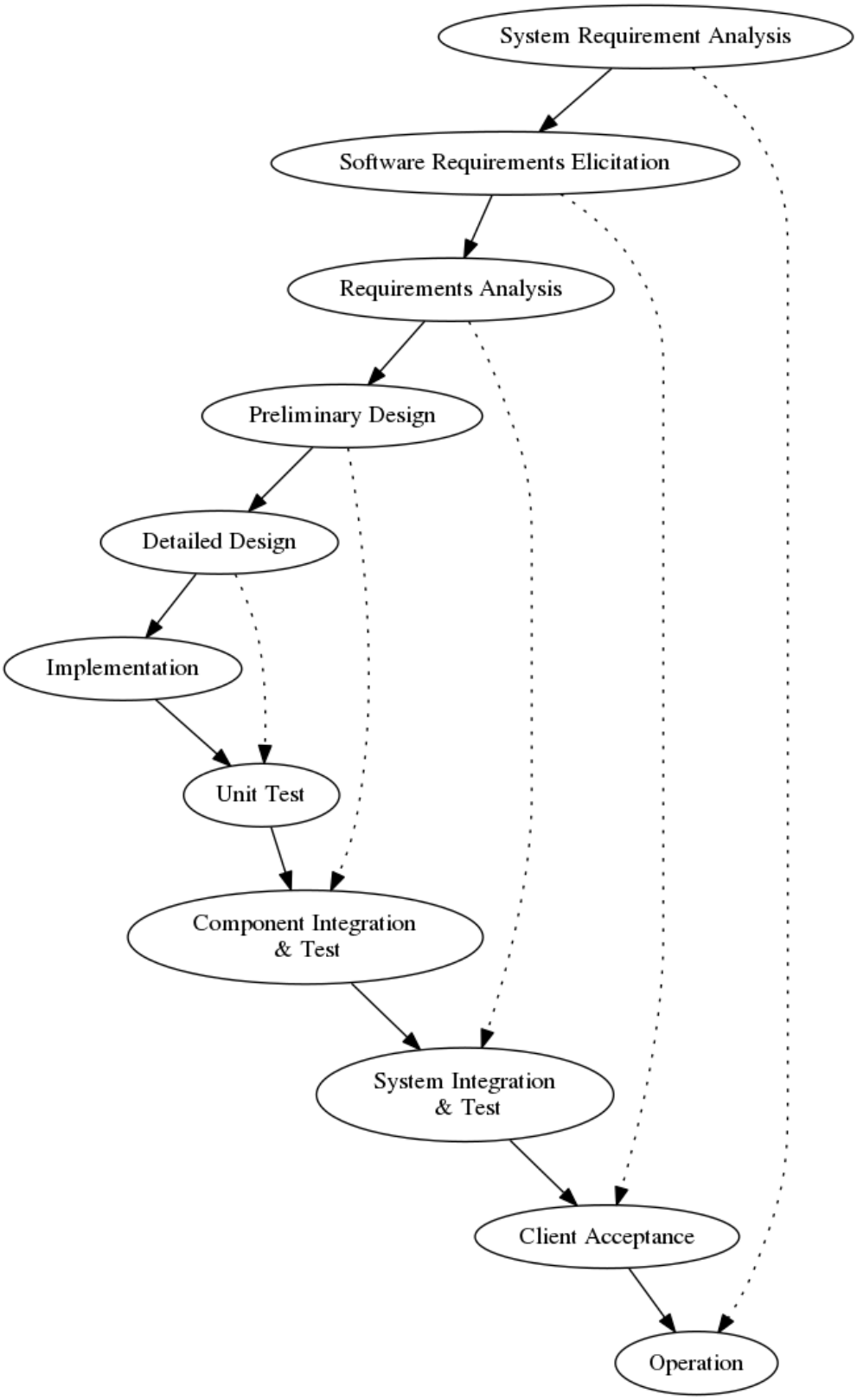


Figure 2: Diagram explaining V Model

Spiral

- Focus on process control
- See

`http://csse.usc.edu/TECHRPTS/1988/usccse88-500/uscsse88-500.pdf`

Incremental Processes

Unified Process

Unified Process is allied closely with UML

- Four distinct phases
  - Inception, Elaboration, Construction and Transition
- Considers activity balance across workflows and phases

OPEN

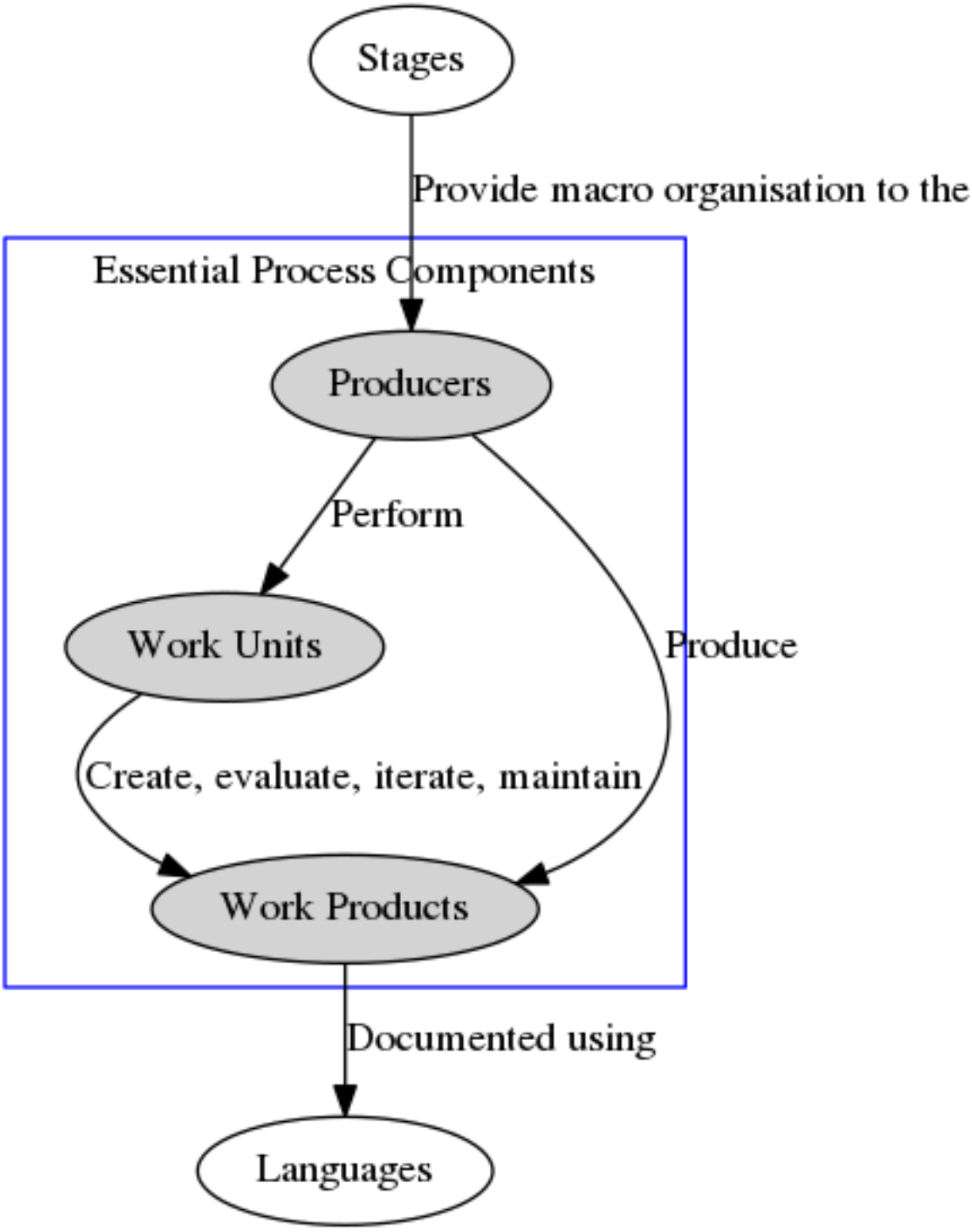


Figure 3: Diagram of OPEN Process

- Process framework
  - process is instantiated from the framework
  - metamodel documents the framework
- Contracts between components
  - process construction

  - scheduling

Agile Processes

- Scrum, XP, FDD, DSDM
- Embrace change
  - Requirements are never fixed
  - Stop pretending and get used to it
- Deliver early and deliver often
  - A working system delivers value
  - A deployed system generates revenue

Lean Development

- More a philosophy than a process
  - Think Big
  - Act Small
  - Fail Fast
- Eliminate Waste
- Amplify Learning
- Decide as Late as Possible
- Deliver as Fast as Possible
- Empower the Team
- Build Integrity In
- See the Whole

Formal Processes

- Application of mathematical formality to software development
  - formal specification
  - transformation of specification to code

Process

- All SE Processes involved phases
  - Requirements
  - Design
  - Development (implementation, coding)
  - Testing (Verification)
  - Delivery and Maintenance
- These are never disjoint, never just sequential
- We iterate between them, and we blur the distinctions because we want to get it right
- Software Engineering cannot work without a defined development process
  - anything else is randomised hacking
- Processes cannot work if they are not usable
  - people don't read telephone books cover to cover
- Good processes should engage the team
  - support technical excellence and innovation
  - embed a culture of trust and responsibility

Standards

- Rules, guidelines and heuristics
- De facto – implicit agreement
  - easily changed

- De jure – formal agreement
  - usually debated and documented

Standard Adoption

- Voluntary
  - achieving good practice
  - safety net
- Required
  - demands of clients
  - certification requirements
  - follow on from other standards
  - process improvement activity

SE Standards

- Normative and informative
- Document centred
- Adaptable

Main SE Standards

- ISO/IEC 12207:2008
  - Systems and software engineering – Software life cycle processes
- ISO/IEC/IEEE 15288:2015
  - Systems and software engineering – System life cycle processes
- ISO/IEC/IEEE 15289:2015
  - System and software engineering – Content of life-cycle information items (documentation)

ISO/IEC 12207

- Framework for lifecycle modelling
- Focus on bespoke software
  - including product and services
- Includes process for defining, controlling and improving software processes
- Last reviewed in 2013

ISO/IEC/IEEE 15288

- Framework for process descriptions
- Focus on system engineering
  - software as a component of system
- Focus on bespoke system development
- Includes process for defining, controlling and improving processes
- Ratified in 2015

12207 vs 15288

- 15288 focuesses on systems
  - hardware, software, people, facilities, material, …
- 12207 focusses on software
  - intended to be used for software component of 15288

ISO/IEC/IEEE 15289

- Standard project documentation
- Focus on purpose and content
  - not necessarily a formal document (e.g. central data repository)
- Ratified in 2015

ISO/IEC 29110

- Software engineering – Lifecycle profiles for Very Small Entities (up to 25 people)
- Subset of 12207 and 15289
- Profiles for different scales of complexity
  - component of a system

  - up to multiple commercial projects
- Ratified in 2016

IEEE Standards

- Terminology
- QA Plans
- Configuration Management
- Requirements Specification
- Unit Testing
- V&V
- Reviews & Audits
- Productivity Metrics
- Quality Metrics
- Project Management Plans
- User Documentation
- Maintenance

Ethics

Code of Ethics

- Agreed standard of behaviour
- Mark of professionalism
- most professional bodies have one
- Enforceable?

Australian Computer Society (ACS)

Primary of Public Interest place interests of public above personal, business or sectional interests
Enhancement of Quality of Life strive to enhance quality of life of those affected
Honesty honest representation of skills, knowledge, services and products
Competence work competently and diligently for stakeholders
Professional Development enhance your own development and your colleagues and staff
Professionalism enhance integrity of the ACS and respect of members for each other

What is Requirements Engineering

- Requirements engineering is a term often used for a systematic approach to acquire, analyse, validate, document and manage requirements
- Typically implemented as a cyclic or iterative process
- Requirements validation may include prototype construction and evaluation
- Applied at both system and software levels, often with interleaved system architecture design

What is a Requirement?

1. A condition or capability needed by a user to solve a problem or achieve an objective
2. A condition by a system or system component to satisfy a contract, standard, specification or other formally imposed document
3. A documented representation of a condition or capability as in 1 or 2

- There is a relationship between the quality / cost / timeliness / etc. of the product and the quality of the process
- Both functional and non-functional requirements are essential for successful software
  - Both must be verified
    - Consequently they must be testable
      - Non-functional Requirements should be measurable

Requirements Engineering Products

- Primary outcome is a requirements specification
  - Essentially a contract between user and developer
  - Basis for all subsequent development and verification processes
- Secondary outcome is usually system and software acceptance test criteria

Why is RE important?

- Most faults observed in a software project are from incorrect, incomplete, or misinterpreted functional specifications or requirements
- Helps earlier detection of mistakes, which are much more costly to correct if discovered later
- Forces clients to articulate and review requirements
- Enhances communications between participants
- Helps record and refine requirements
- All this is about producing good requirements

Advice/Perspective

- … a systematic approach to finding, documenting, organizing, and tracking the changing requirements of a system
- In practice it is impossible to produce a complete and consistent requirements document
- Getting the requirements right is critical for success
- Requirements are rarely right at the start of a large project
  - Expect change
  - Manage it
  - Agile mantra “Embrace Change”

Functional Requirements

- Requirements (or capabilities) for functions (specific behaviour) that must be performed by the system
  - e.g. read a bar code, change a user name
- Primary focus of most requirements activities

Non-Functional Requirements

- Constraints on performance or quality

Product Properties

- Requirements on the behaviour of the product
  - System shall process a minimum of 8 transactions per second
  - User credit card details shall be secured

Process Properties

- Requirements on the practices used to develop / produce the system
  - Control software shall be verified in accordance with IEEE STD 1012-1998

Classification of Non-Functional Requirements

According to the ISO standard

- Safety requirements
- Security requirements
- Interface requirements
- Human engineering requirements
- Qualification requirements
- Operational requirements
- Maintenance requirements
- Design constraints

Sources of Requirements

- Users
  - e.g. customers or end-users – user requirements
- Other Stakeholders
  - e.g. marketing experts, regulators, managers, business owners, developers
- Non-Human Sources

- e.g. other devices or systems in the environment

The Requirements Engineering Process

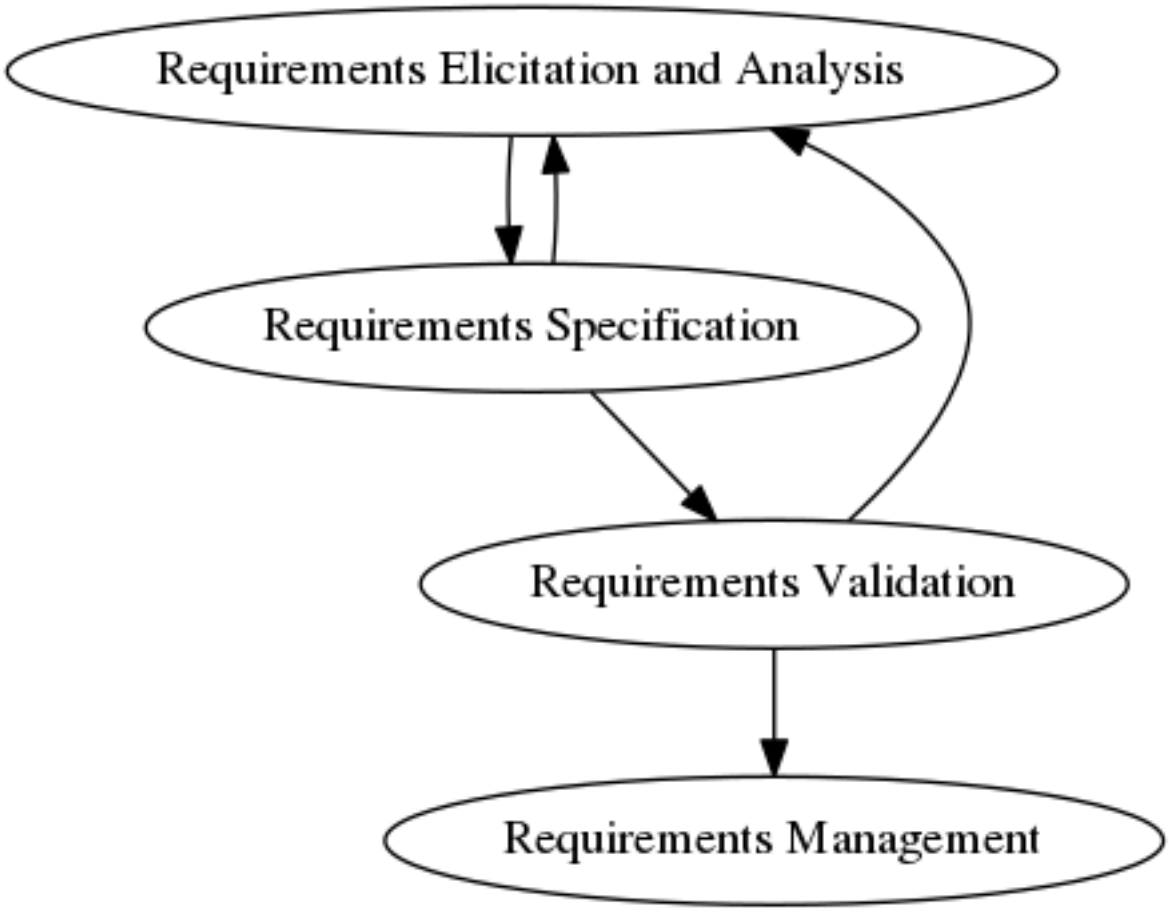


Figure 4: Requirements Engineering Process

Summary

- Requirements engineering is concerned with eliciting, analysing, documenting, validating and managing requirements
- The process, and the specification it produces, is the primary link between the user and the system developer
- The requirements specification is also the basis for all subsequent development activity, and must guide all decisions that determine the resultant product quality
- Requirements engineering is the key to product quality

Project Charter

Vision Statement

- Long term purpose of system
  - a bit idealistic
- For – target audience
- Who – statement of need
- The – product
- Is – category
- That – reason to use
- Unlike – alternative

- Out Product – advantage

Goals

- High-level
- What project will accomplish

Objectives

- Specific
- Supports a goal
  - think “how” it does this
- Describe with an action verb
  - measurable
  - address project end result

SMART

- Specific
  - what is to be accomplished
  - only essential aspects
- Measurable
  - need success/completion criteria
- Agreed-upon
  - common understanding amongst stakeholders
- Realistic
  - achievable with available resources
- Time-based
  - realistic deadline

Business Benefits

- High-level but concrete
  - Increased revenue
  - Reduced costs
  - Improved efficiency
  - Improved customer satisfaction
  - …

Scope

- What is to be delivered
  - by end of project
  - releases determined later
- What is explicitly out of scope
  - does not relate to business benefit

Stakeholders

- Sponsor
- Influencers
- Users
  - key
  - restricted
  - super
- Anti-Users
- Others
  - e.g. infrastructure team

Assumptions

- Expected to occur

Constraints

- Restrictions

- project
- development team

Business Value Not System Requirements

Understanding the Business

- Developers and Stakeholders need a shared understanding of the project's purpose
  - easier when they collaborate continuously
- Focus on value to be delivered
  - not just the requirements
- Enables better decisions, designs and suggestions
  - developers are part of the value chain
    - not just serving it

Business Model

A business model describes the rationale of how an organisation creates, delivers, and captures value

Business Model Canvas

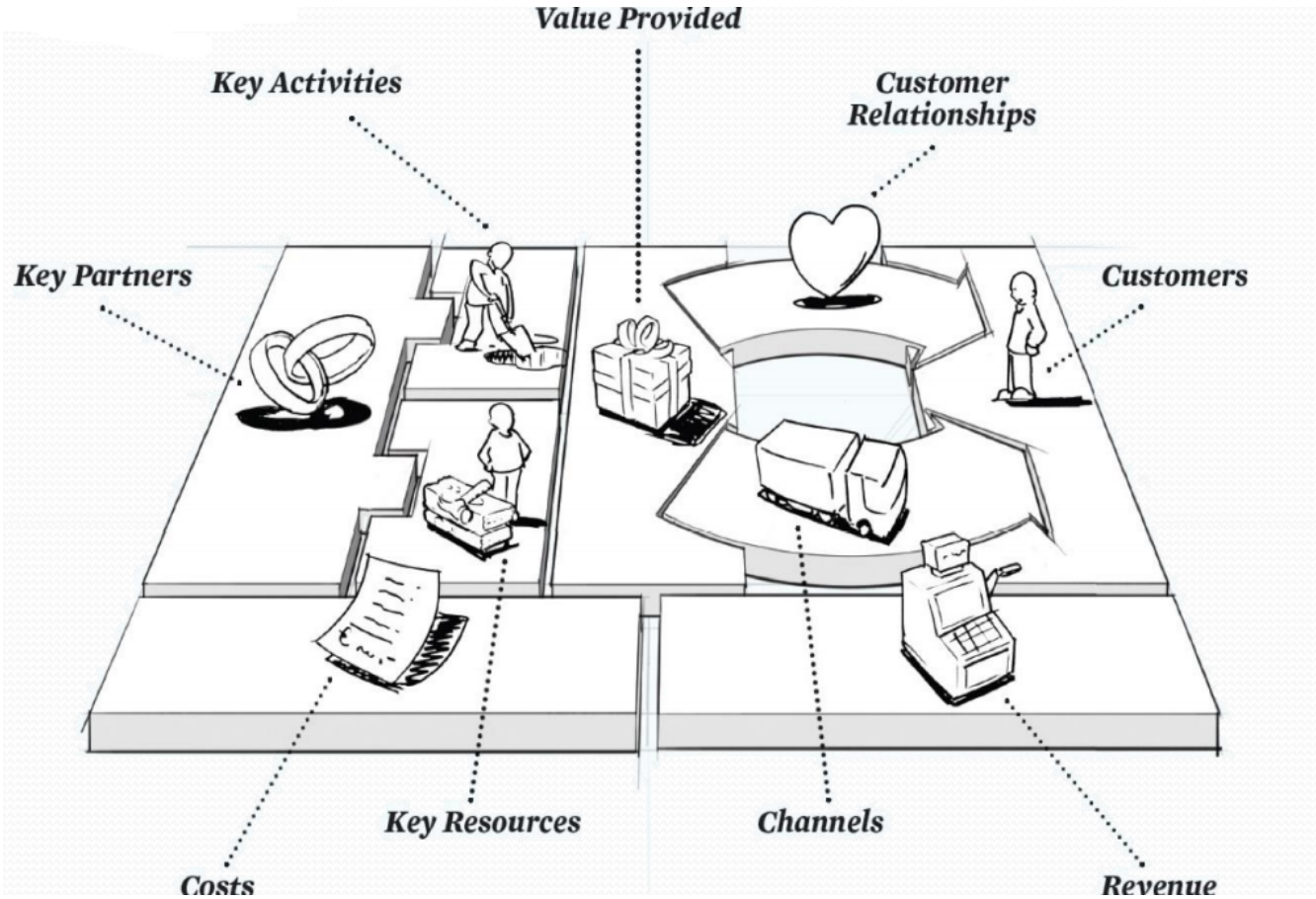


Figure 5: Business Model Canvas

Customers

- Personas
- Who's Impacted

- Stakeholders

Value Proposition
- Use Cases
- Specification by Example
- Customer Savings / Revenue
- Improvements
- Experience Improvements

Channels
- Systems
- Methods
- Related Features

Relationship
- Direct / Indirect
- Human / Automated
- Assisted / Self Service
- Individually / Collaboratively

Revenue Streams
- Opportunity
- Savings
- Profit
- Improvements

Key Resources
- Systems
  - Primary
  - Secondary / Connected
- Team
  - Development
  - Business

Key Activities
- Use Cases
- Who's Activities
- Connected Activities

Key Partnerships
- Development Team
- Business
- Secondary / Connected Teams
- Impacted Teams
- Related Teams

Cost Structure
- Opportunity
- Development Estimates
- Quantity of Customers

Start-Up
- Enthusiastic developers
- No clear business model
  - Or how to monetise success
- Now have focus

- What to out source
- What to develop
- Costs
- Revenue streams

Requirements Elicitation

Elicitation Process

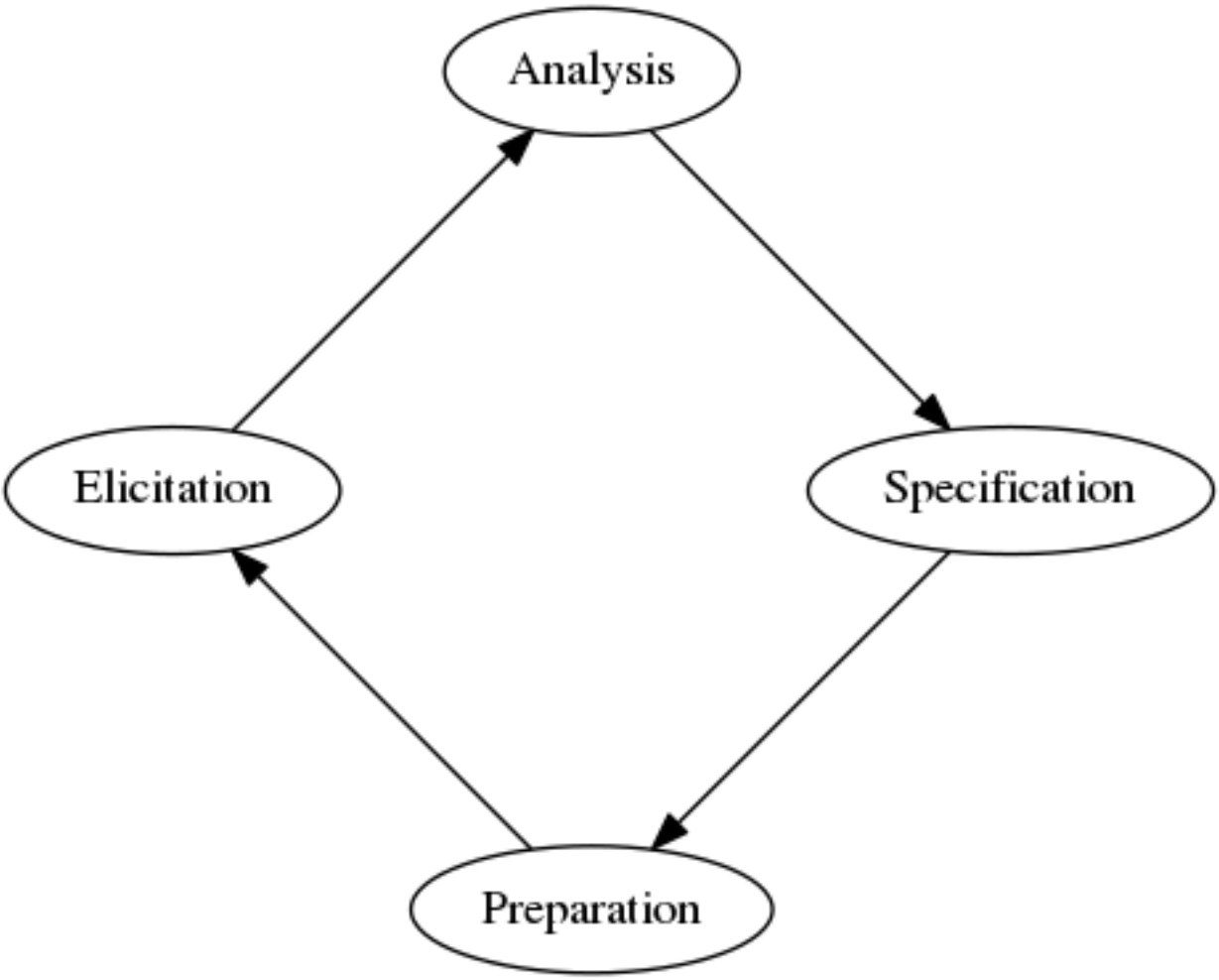


Figure 6: Elicitation Process

Preparation – Sources of Requirements

- Stakeholders
- Users
- Environment
  - application domain
  - organisation
  - operations
    - other system dependencies
      - interface requirements
      - timing constraints

        * execution environment
            - platform
            - reliability and performance
        * criticality
            - mission
            - safety

Know Your Users – User Role Modelling

- What types of people will use the system?
- Don't think of an anonymous user
- Identify different user roles
    - brainstorm initial set
    - group related roles
    - consolidate roles
    - refine roles
- Don't get stuck on organisational roles

Elicitation Challenges

- Stakeholders and users may not be able to describe their tasks well
    - make assumptions and leave things unstated
- No-one knows everything
- Requirements conflict
- Implicit requirements

Elicitation Techniques

Interviews

- Effective for understanding problem and eliciting *general* requirements
- Prepare questions in advance
    - discussion needs a starting point
    - primarily open-ended questions
    - strawman model if you have some data
- Suggest ideas and alternatives
    - users may not realise what is possible
- Active listening
    - paraphrase what you understand
- Clarify what's unclear
- Maintain focus

Workshops

- Structured meeting
    - formal roles
    - clear goals
- Multiple stakeholders
    - resolve conflicting requirements
    - quickly gather broad system usage

Focus Groups

- Less structure
    - still need clear goals
- Exploratory discussion
    - needs
    - preferences
    - expectations
- Broad stakeholder representation
- Gather broad-based ideas

Observations
- Observe how users perform their tasks
- Users often cannot describe everything they do
  - too many fine details or habitual tasks
- Time consuming
  - silent observation
  - interactive

Questionnaires
- Inexpensive and easily administered to remote sites
- Collect data from many users
- May feed into interviews or workshops
- Good questionnaires difficult to write

Independent Elicitation Techniques
- Discover information on your own

System Interface Analysis
- Look at other system's functionality
- Data exchange
  - including formats and validation rules
- Services

User Interface Analysis
- Study existing systems
- What should be replicated and avoided
- Good way to learn existing system and processes

Document Analysis
- Business process descriptions
- Existing system documentation
- Industry standards or legislation
- Gain understanding of domain or system

Requirements Modelling

Product vs User Centred

Product-Centred
- Focus on features to be delivered
  - expect users will use features to complete tasks

User-Centred
- Focus on anticipated usage
  - what do users need to accomplish
- Reveal necessary functionality
- Assists with prioritisation

Use Case Modelling
- Models and documents the functional requirements of a problem domain
- Results in the production of the
  - functional requirements
  - which are the detailed, role based, functional account of the requirements

Why Use Cases?

- Formalises users' expectations of what the system is to do and how the system is to be used
- Easy technique to understand
  - documents actual paths through the system
- User-driven process
  - encourages user involvement
- Basis for scoping and prioritising development work
- Basis for acceptance testing
- Well aligned with Business Process Modelling

What is a Use Case?

- A way to use the system
- Externally required functionality
- What the system does
  - not how it does it
- Specify the behaviour of a use case as a flow of events

Actors, Use Cases and Association

Actors things outside of the system that interact with the system
Use Cases features of the system that an actor uses
Associations indicates a relationship between an actor and a use case

Actors

- Everything that interacts with the system
- Not described in detail (they're outside of system)
- Normally act in several use cases
- Represents a role that a user can play
  - many users are represented by a single actor
  - one user can be different actors at different times

Actors are External to System

- Make sure that actors are people or other systems that would actually use the system
- The system does not use itself

Primary and Secondary Actors

- Primary actors are those actors that the system is designed to serve
  - Main users of the system
- Secondary actors are support roles
  - secondary actors only exist so that primary actors can use the system

<<include>> Relationship

- Factor out common behaviour in use cases
  - sequence of steps appearing in multiple use cases
- Scenario always uses the included steps
  - included steps do not have to be a complete use case

<<extend>> Relationship

- Factors out optional behaviour in use cases
  - used when a scenario produces a different result in certain situations
    - when there are optional or uncommon steps that may occur
- The extended scenario may be completed without including the steps from the extending use case
  - removes the need to have many primary scenarios to capture the optional paths of a use case
- Delivery of extending use cases can occur in later phases of development

Extension Point

- Included in the use case description to indicate the point at which the extending use case begins

  - condition that causes the extension is highlighted
- Scenario for the base use case runs as normal until the extension point
- Under certain conditions the extending use case then begins and runs to completion
- Base use case then resumes

Use Cases Limitations

- Interaction focus
  - usage scenarios
- Not suitable
  - batch processing
  - complex business rules
  - computationally intensive
  - real-time systems
  - embedded systems

Activity Diagram

- Shows the steps involved in performing a task
- Special form of state diagram
- Describes a complex task for objects of a class, operations, or use cases
  - focuses on internal processing
  - use where all the events represent the completion of internally generated actions
  - use state diagrams where asynchronous events occur

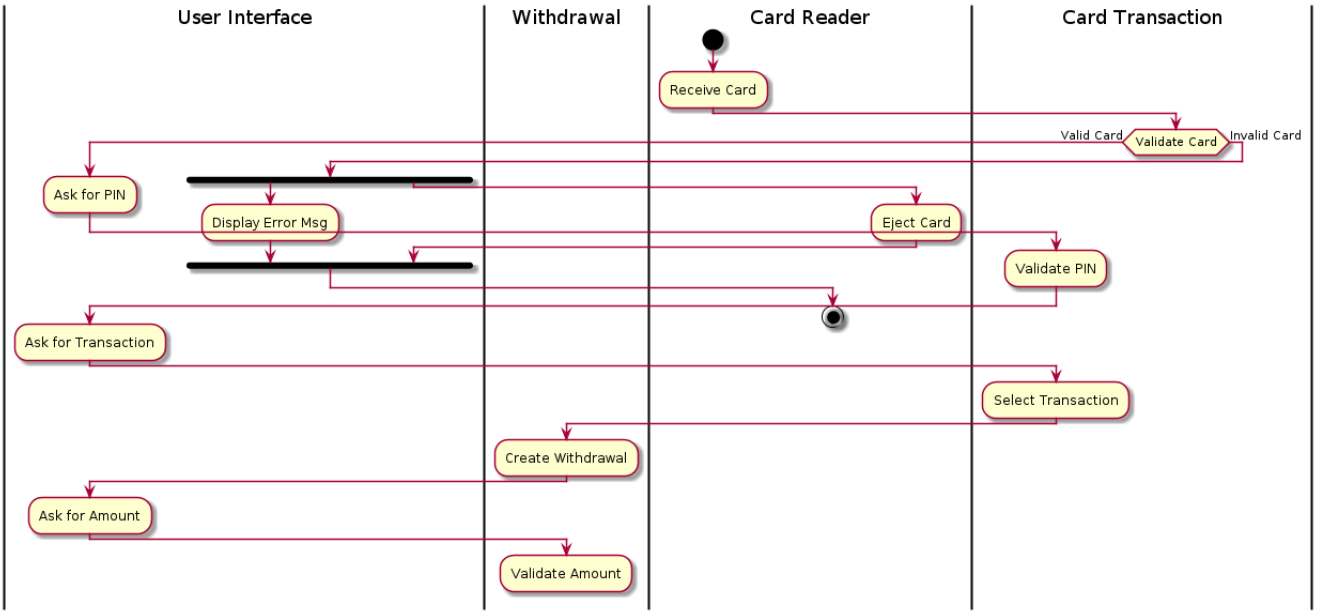


Figure 7: Activity Diagram Example

Activity Diagrams in Use Case Modelling

- Determine triggering event that starts use case flow
- Identify actions and determine control flow
- Add guard conditions and decision points
- Add forking and joining to show parallel activity
- Create invoke activities if complexity requires it

- Group activities into partitions if needed
- Add flows corresponding to alternative scenarios
- Each path should correspond to an individual scenario

User Stories

- Short description of functionality
- From the user's perspective
- Provides value to the user or customer
  - consider both types of clients
- Must be testable
- Provides enough information for developers to make rough estimates

Prioritisation

First Things First

- Rank high value use cases so they are delivered first
- Create shared understanding of how use cases contribute to business objectives
- Leads to estimation and release planning

Prioritisation Activities

- Driven by customer representatives
  - may require a team facilitator
  - may require a few iterations
- Verify results against agreed success criteria

Setting Priorities

- MoSCoW
  - Must have
  - Should have
  - Could have
  - Wont have

Prioritisation Process

- Conducted in a workshop
  - customer representatives
  - developers
- Write use case names on index cards
- Group cards on table / wall
- Review and revise

Prioritisation Factors

- Importance of actor
  - broad base of key users
  - small group of important stakeholders
- Importance of use cases to actor
- Cohesiveness of functionality
  - does use case relate to other higher priority use cases
- Dependencies between use cases
- Risk involved in implementing use case

Prioritisation Strategies

- Performed by the development team
  - customer *decides* on priorities

  - developers provide input
- Deliver important business value early
- Focus on the Must Haves vs the rest
- Split use cases with mixed priorities

Software Review

What is a Software Review?

- Generic term for a variety of techniques for evaluating software development products
  - Collaborative
  - Common aim to detect errors and improve software quality
- Can be applied to any product
  - requirements
  - design
  - code
  - test cases
- Checking for
  - completeness
  - consistency with other project documents
  - correctness
  - feasibility

Review vs Testing

- **Reviews**
  - Concerned with analysis of the static system representation to discover problems
  - May be supplemented by tool-based document and code analysis
- **Testing**
  - Concerned with exercising and observing product behaviour
  - System is executed with test data and its operational behaviour is observed

Benefits of Reviews

- Complexity of software development means major cost benefits from early detection of errors
- Can be applied earlier and to untestable products
  - requirements specifications, design documents, test plans, …
- Incomplete versions of a system can be inspected without additional costs
  - incomplete programs require specialized test harnesses
- During testing, errors can mask (hide) other errors
  - inspection is a static process
- Reduce rework – improves schedule performance
- Author receives timely feedback on defects
- Provides project management status (milestones)
- Can also consider broader quality attributes
  - e.g. compliance with standards, portability, maintainability
- Collaboration in reviews motivates better work
  - distributes expertise and helps team building
- Helps subsequent maintenance
  - documentation exists and is consistent

Limitations of Reviews

- Can check conformance with a specification
  - but not conformance with the customer's real requirements
- Cannot check non-functional characteristics
  - e.g. performance, usability

Error Amplification

- Errors from previous activity
- Some errors are passed through
- Some get amplified
- New errors are generated
- Reviews filter out errors
- Errors passed on to next phase

Types of Review

Technical Review

- review for conformance to standards or achievement of project milestones
- led by team leader
- often management participation

Software (Fagan) Inspection

- peer review with formal process
- led by independent moderator
- systematic data collection
- focus on defect detection and description
- process improvement goal

Structured Walkthrough

- less formal than inspection
- usually led by producer
- no formal data collection
- no participant preparation

Audit

- external review of work product
- independently managed
- usually late in the process

Basic Inspection Principles

- Formal structured process with checklists and defined roles for participants
- Participants prepare in advance for meeting
- Focus is on identifying problems
  - not solving them
- Conducted by technical people for technical people
- Inspection data is recorded to monitor
  - effectiveness of the inspection process
  - relationships with product quality
- Looking for defects in product, not in developers

Inspection Participants

Moderator

- responsible for leading inspection process
- schedules and conducts review
- prepares reports and follows up action items

Recorder

- keeps records of all significant inspection results
- help prepare reports

Producer(s)

- responsible for work under review

Reader

- presents work in lieu of producer in formal inspection

Reviewers
- directly concerned with, and aware of work under review
- required to prepare for review
- should be objective and accountable

Inspection team typically between 3 and 7 people

Inspection Process

Request
- producer requests inspection of document
- moderator is selected

Entry
- moderator checks document is ready

Planning
- moderator plans inspection process

Overview (optional)
- meeting to distribute documents

Preparation
- participants work alone using checklists

Inspection Meeting
- consolidate errors found by individual reviewers
- classify errors by severity for future analysis
- find additional errors via synergy

Rework
- producer resolves issues from the meeting

Follow-up
- moderator checks rework
- schedules additional inspection if needed

Exit
- moderator checks document against exit criteria

Release
- document is released

Issue Classification

Major
- defects that are likely to
  - cause incorrect behaviour, or
  - require external resolution before development can be completed

Minor
- defects that are likely to cause limited or no loss of functionality

Grammatical
- defects that are
  - spelling mistakes
  - grammar defects
  - typographical faults

Questions
- potential defects that the reviewer is unsure about and which s/he wants to discuss at the meeting

Inspection Preparation
- Product must be ready for inspection (entry criteria)
- Inspection team must be selected, briefed and supplied with review and source documents
- Checklists and standards give guidance to reviewers

- Reviewers must prepare individually, recording time and errors identified
- Reviewers may be given special roles
- Reviewers are seeking to find as many defects as possible
- Reviewers should concentrate on major defects
  - actual classification is not critical – re-assessed at meeting
- Queries about the documents should be recorded for meeting
- Experience shows about 75% of all errors are typically located during individual preparation

Reading Techniques

Ad-hoc rely on reviewers' knowledge and experience

Checklist focus on know problem types

Scenario-based assigns specific responsbilities for reviewers – attempts to focus each reviewer on a different class of defects

Perspective-based enhanced version of scenario-based reading – based on viewpoint or needs of stakeholders

Stepwise Abstraction reviewers derive specification from the code and then compare with original specification

Non-Functional Requirements

- System properties and constraints
  - e.g. reliability, response time and storage requirements
- Non-functional requirements may be more critical than functional requirements

NFR Sources

- Product requirements
  - behaviour constraints (execution speed, reliability, etc)
- Process requirements
  - restrictions on the development process (standards to follow, etc)
- External requirements
  - factors external to the system (inter-operability, legislative requirements, etc)

Verifiable

- Imprecise requirements cannot be verified
- NFR should be a measurable statement
  - The system should be easy to use by experienced controllers and should be organised in such a way that user errors are minimised
  - vs
  - Experienced controllers shall be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users shall not exceed two per day

Quality Attributes

- Safety
- Security
- Reliability
- Resilience
- Robustness
- Understandability
- Testability
- Adaptability
- Modularity
- Complexity
- Portability
- Usability
- Reusability
- Efficiency

- Learnability

Software Testing

There is a myth that if we were really good at programming there would be no bugs to catch

- If only we could really concentrate
- If only everyone used agile methods
- If programs were written in Scala
- If we had the right silver bullets

Then there would be no bugs. So goes the myth

Validation and Verification

Validation

- "Are we building the right product?"
- Demonstrate that the software meets its requirements

Verification

- "Are we building the product right?"
- Also called Defect Testing
- To discover faults or defects in the software where its behaviour is incorrect or not in conformance with its specification
- **Successful** tests make the system perform incorrectly
  - expose a defect in the system

Inspections and Testing

- Inspections and testing are complementary and not opposing verification techniques
- Both should be used during the V & V process
- Inspections can check conformance with a specification but not conformance with the customer's real requirements
- Inspections cannot check non-functional characteristics such as performance, usability, etc

Stages of Testing

Development testing

System is tested during development to discover bugs and defects

- All testing activities carried out by the team developing the system
  - Unit testing
    - individual program units (methods or classes) are tested
    - focus on correct functioning of objects or methods
  - Integration / Component testing
    - several units are integrated to create composite components
    - focus on testing component interfaces
  - System testing
    - some or all of the components in a system are integrated and the system is tested as a whole
    - focus on testing component interactions

System Testing

- Involves integrating components to create a version of the system and then testing the integrated system
- Focus is on testing interactions between components
- Checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces
- Tests the emergent behaviour of a system
- Components developed by different teams may be integrated at this stage
- Collective rather than an individual process
- In some companies, system testing may involve a separate testing team with no involvement from designers and programmers

Testing Policies

- Exhaustive system testing is impossible so testing policies which define the required system test coverage may be developed

Release testing

Separate testing team test a complete version of the system before it is released to users

- Testing a particular release of a system that is intended for use outside of the development team
- Primary goal is to convince the supplier of the system that it is good enough for use
  - Show that the system delivers its specified
    - functionality
    - performance
    - dependability
    - does not fail during normal use
- Usually a black-box testing process where tests are derived from the system specification

Release Testing and System Testing

- Release testing is a form of system testing
- Release testing should be conducted by a team that was not involved in the system development
- System testing by the development team should focus on discovering bugs (verification, defect testing)
- Release testing determines if the system meets its requirements and is good enough for external use (validation)

User testing

Users or potential users of a system test the system in their own environment

- User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing
- User testing is essential, even when comprehensive system and release testing have been carried out
  - Influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system
    - cannot be replicated in a testing environment

Types of User Testing

- Alpha Testing
  - Users of the software work with the development team to test the software at the developer's site
- Beta Testing
  - A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers
- Acceptance Testing
  - Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment
    - primarily for custom systems

Requirements Based Testing

- Requirements-based testing involves examining each requirement and developing a test or tests for it
  - functional requirements
  - non-functional requirements

Use Cases and Release Testing

- Use cases describe functional requirements (these need to be validated during release testing)
- Generate test scripts and scenarios from use case descriptions
  - each sequence of events should generate at least one test script (typical and alternatives)
  - pre and post conditions must be tested
- Test scripts need to be traceable to their originating use case sequence of events

Use Case Testing and Sequence Diagrams

- Sequence diagrams are usually associated with use cases

  - describe how the system model provides the functionality
- Can be used to drive integration and system testing
  - provides view of what software components to test
- Provides traceability between requirements, design, code and system tests

Performance Testing

- Part of release testing may involve testing the emergent properties of a system (e.g. performance and reliability)
- Tests should reflect the profile of use of the system
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable
- Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behaviour

Regression Testing

- Check that changes have not 'broken' previously working code
  - applies to both development and release testing
- Manual testing – regression testing is expensive
- Automated testing – it is simple and straightforward
- All tests are rerun every time a change is made to the program
  - must run 'successfully' before the change is committed

Test Automation

- Testing should be automated, as much as possible
- Write test drivers that execute the code being tested and capture the errors
- Rerun tests whenever changes are made to the code

Agile Methods and Acceptance Testing

- Customer writes acceptance criteria for stories
  - ideally for stories going into the next iteration
- Developers turn these into automated test cases during the development iteration
  - working with customer to elaborate details of the criteria
- In theory no separate acceptance testing process
  - reality is the customer will still run through scenarios themselves on the system to do UAT
- Main problem here is whether or not the embedded user is 'typical' and can represent the interests of all system stakeholders

Key Points

- Testing can only show the presence of errors in a program. It cannot demonstrate that there are no remaining faults
- Development testing is the responsibility of the software development team. A separate team should be responsible for testing a system before it is released to customers
- Development testing includes unit testing, in which you test individual object and methods component testing in which you test related groups of objects and system testing, in which you test partial or complete systems
- When testing software, you should try to 'break' the software by using experience and guidelines to choose types of test case that have been effective in discovering defects in other systems
- Wherever possible, you should write automated tests. The tests are embedded in a program that can be run every time a change is made to a system
- Test-first development is an approach to development where tests are written before the code to be tested
- Scenario testing involves inventing a typical usage scenario and using this to derive test cases
- Acceptance testing is a user testing process where the aim is to decide if the software is good enough to be deployed and used in its operational environment

UML Structural Modelling

- Class modelling concepts
- Different approaches to discover classes
- Specifying classes

  - Discovering attributes
  - Discovering operations
  - Discovering associations
  - Discovering generalizations
- UML Class Diagrams

Class Diagram

High Level Class Diagram

- Provides an overview of the system
- During analysis the class diagram shows a conceptual model of the system
  - during system elaboration (as the design becomes more detailed)
    - more detail is added to existing classes
    - new classes are often introduced
  - ideally the implemented system should just be a more detailed version of the high level OOA model
- High level class diagram focueses on the components of a system and their relationships
  - details of the components are fleshed out later

Class

- Description (or template) for a set of objects that share common structure and behaviour in terms of attributes and operations
- Relationship between classes and objects
  - an object is simply an instance of a class
- Objects contains attribute values that conform to the attribute types defined in their class
- Objects can invoke operations defined in their class

Properties of UML Classes

- **Name** – a unique name within its namespace
- Set of attributes and operations
- isActive – active or passive
  - an object of an active class has its own thread of control and runs concurrently with other active objects
  - operations of a passive object are controlled by an active object

Class Modelling

- Capture the system state
  - What a system consists of in terms of classes
- Class modelling activities
  - Identifying classes (or objects)
  - Identifying associations among objects
  - Modelling generalisation relationship
  - Modelling non-trivial behaviour with state machines

Different Types of Classes

- Reusable robust designs do not match the problem domain exactly
- Software systems can be thought of having three dimensions:
  - Behaviour, Presentation, Information
- Classes encompass one or two dimensions

Boundary Class

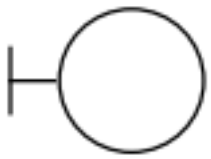


Figure 8: Boundary Class

- separate the interfaces from the rest of the system
- handles communication with the environment
  - users and other systems

Entity Class

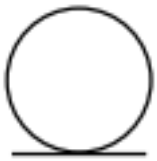


Figure 9: Entity Class

- functionality dealing with the storage and handling of long-lived (potential persistent) information
- often are general to many use cases

Control Class

Figure 10: Control Class

- controls interactions between a group of objects
- functionality specific to one, or a few use cases, and not naturally placed in the other class types

Finding Classes and Use Cases

- Identify participating classes for each use case
  - reuse classes from one use case to another
    - refining existing responsibilities
    - adding more responsibilities if needed
- Deciding on type of classes and what goes in each class can be difficult
  - often the reasoning is based on what are the likely potential changes to the system, and how can the changes be localised

Finding Boundary Classes

- Boundary classes interact with actors outside the system and with all types of classes within the system
- Most boundary classes are relatively easy to identify
  - system interface descriptions
  - actors
    - each concrete actor needs its own interface to the system
    - often an actor will have several boundary classes it deals with, at the implementation level
  - extracted from use case descriptions
    - from the functionality that is interface specific

Finding Entity Classes

- Entity classes often correspond to problem domain entities
- Typically represent information that survives longer than the sequence of events that a use case represents
  - allocate responsibilities that belong to this information to the entity class
    - what operations will need this data

- Parition the data and responsibilities into classes that represent a single abstraction

Finding Control Classes

- Extract remaining functionality from a use case and partition into control classes
  - could start with one control class and break it up as it gets complicated
- These are behaviours that don't belong to the interface, nor do they belong to how the information is handled
- Control classes control (manage) complex interactions between a group of objects in a use case
- Objects of a control class tend to exist only within the sequence of events that make up a use case
- Tie only one actor to a control class
  - system changes often start from actors, if control classes are dependent on only one actor this helps isolate changes in the system
- Some use cases may have no control classes
  - all control flow is modelled between interface and entity classes

Approaches to Identifying Classes

Noun Phrase Approach

- Perform a "grammatical parse" of a system description
- Underline each noun or noun clause and enter it in a table
- Synonyms should be noted
- Assumes that the requirements document is complete and correct
- Tedious

Common Class Patterns Approach

- Derive candidate classes from the generic classification theory of objects
- Example Patterns:
  - *External entities* that produce or consume information to be used by a computer-based system
  - *Things* that are part of the information domain for the problem
  - *Concepts* that are shared and agreed upon by a large community of people
  - *Occurences or events* that occur within the context of system operation
  - *Roles* played by people who interact with the system

CRC Approach

- CRC – Class, Responsibility, Collaboration
- More than a technique for class discovery
- Often used in brainstorming sessions
- Developers fill in CRC cards
- CRCs are discovered from the analysis of use cases
- Suitable for the verification of classes discovered by other methods and to determine class properties

Mixed Approach

- Combination of all previous approaches
- Middle-out approach rather than top-down or bottom-up
- One possible scenario
  - initial classes from general knowledge and experience of analysts
  - common class pattern approach to guide
  - noun phrase approach to identify new classes and to verify discovered classes
  - CRC approach to brainstorm

Selection Characteristics for Classes

1. **Retained information** – the potential objects will be useful during analysis only if information about them must be remembered so that the system can function
2. **Needed Service** – the potential object must have a set of identifiable operations that can change the values of its attributes in some way
3. **Multiple attributes** – an object with a single attribute is probably better represented as an attribute of another object during the analysis activity
4. **Common attributes** – a set of attributes can be defined for the potential object

5. **Common operations** – a set of operations can be defined for the potential object
6. **Essential requirements** – external entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as objects in the requirements model

Specifying Classes

- Once classes are selected, they should be further specified by placing them on a class diagram and by defining their properties
- Class naming guidelines
  - start with a capital letter
  - join multiple words, with each word starting with a capital letter
  - should be a singular noun whenever possible
  - should be meaningful
  - should be drawn from the application domain
  - no longer than 30 letters

Attributes

- Describe a static feature of a class or objects of the class
- Named slots within a class, particularly, the range of values that instances of the class (objects) may hold
- Classes define attribute types and objects contain attribute values

Discovering Attributes

- Perform a "grammatical parse" on a processing narrative for the problem and select things that reasonably "belong" to the object
- Discovering attributes is a side effect of class determination, but it is a demanding task
- In the initial specification models, focus on attributes that are essential to understand the state of objects

Operations

- Describe a behavioural feature of a class
- A service that can be requested from an object to effect behaviour
- Invoked by a message sent to an object of the class
- Signature describes actual parameters
  - including return values
- A procedure that implements the operation is called a method

Discovering Operations

- Perform a "grammatical parse" on a processing narrative for the problem
- From expected object responsibilities including four primitive operations
  - create (an object)
  - read (access to the state of an object)
  - update (modification of the state of an object)
  - delete (an object itself)
- Three types of operations
  - manipulating data in some way (adding, selecting, deleting)
  - performing computation
  - monitoring an object for the occurrence of a controlling event

Relationships between Classes

- Two major relationships between classes

Association

- Defines a semantic relationship between classes
  - more precisely relationships between instances of the classes
  - ability of one instance to send a message to another instance
- An association has
  - a name
  - at least two association ends, and

- each end is attached to one of the classes in the association

Properties of Association Ends

- name: role name
- aggregation: none, aggregate, composite
- changeability: changeable, frozen, addOnly
- ordering: unordered, ordered
- isNavigable: true or false
- multiplicity: 0, 0…1, 1…1, 1..,
- targetScope: instance (default) or class
- visibility: public, protected, private, package

Aggregation and Composition

- Aggregation is a whole-part relationship
  - aggregate is the whole, which has an assembly of part
  - aggregate are the parts that contribute to the whole
- Aggregation is the same as an association
  - except that instances cannot have cyclic aggregation relationships
- Composition is a strong form of aggregation
  - lifetime of the 'part' is controlled by the 'whole'

Property	Aggregation	Composition
Shareable (part can be shared)	Yes	No
Existence Dependent (part can exist without its whole)	Yes	No
Asymmetric	Yes	Yes
Transitive	Yes	Yes

Properties of Aggregation

Generalisation

- A taxonomic (a kind-of) relationship between a more general class (superclass, parent or supertype) and a more specific class (subclass, child or subtype)
  - inheritance relationship
- A subtyping relationship (substitutability)
  - instance of a subclass can always be used where an instance of its superclass is expected

Polymorphism

- A method inherited by a subclass can be overridden (modified) in the subclass to correspond to semantic variations of the subclass
- We say that the operation is polymorphic

Inheritance

- A way to increment specifications by reusing and extending the superclass description
- Subclass inherits all the features of its parent class
  - attributes, operations, methods, constraints and relationships
- Operations may have many implementations (methods) but the same signature
  - name, and the number and types of its parameters
- Polymorphic behaviour relies on inheritance

Multiple Inheritance

- A subclass can inherit from more than one superclass
- If two or more methods of an operation conflict, the model is ill-formed
- If there are conflicts in inherited constraints, the model is also ill-formed

Discovering Associations

- Side-effect of discovering classes
- Some attributes are associations
  - non-primitive data type attributes
- From the processing narrative for the problem
  - has, is part of, manages, reports to, is triggered by, is contained in, talks to, includes, etc
- From use cases or activity diagrams
- Avoid tertiary associations (if possible) to reduce semantic misinterpretations

Discovering Aggregations

- From the processing narrative for the problem
  - having verb (has, is part of, is contained in, includes, etc)
- From analysing whether associations satisfy the four properties of aggregations and compositions
  - transitivity and asymmetric
  - shareability and existence-dependency

Discovering Generalisations

- From the processing narrative for the problem
  - being verb (is a kind of, is one of either)
- From analysing whether the relationships satisfy the properties of inheritance
  - substitutability (subtyping)
  - polymorphism

State Machines

State Machine Modelling

- Two different levels
  - Protocol – legal usage scenarios (e.g. a sub-system interface, …)
  - Behavioural – individual entities (e.g. objects)
- Supports both Moore and Mealy machines
  - Moore – actions associated with states or transitions
  - Mealy – actions may depend on system state as well as trigger
- Objects have qualitative and quantitative state
- Quantitative state is the current values of all of an object's attributes
- Qualitative state indicates current status of an object during its lifespan
  - why we use state machine modelling
- Extended state machines consider how some quantitative state impacts qualitative state
- Model each class that exhibits interesting behaviour

UML Notation for a State

- States may have an entry action, an exit action, and an activity
- Action – takes no execution time
- Activity – takes time and is interruptible

Event-Driven Behaviour

- Event – a type of observable occurrence
- Interactions
  - Call event – synchronous object operation invocation

  - Signal event – asynchronous signal reception
- Occurrence of time instants (time event)
  - Interval expiry: after(t)
  - Calendar/clock time: at(t)
- Change in value of some entity
  - Change event: when(e)

UML Notation for Composite States

- Two or more concurrent sub-states (called regions)
- Or mutually exclusive disjoint sub-states

Measurement and Estimation

- Collection and analysis of data about software projects, processes and products
- Fundamental to any engineering discipline
  - provides objective insight
  - enhances purely subjective evaluation
- Immature with few standardised measures and metrics and inconsistent industrial practice:

Reasons for Measuring

Characterise

- gain an understanding
- establish a baseline for future assessments

Evaluate

- determine status relative to plans
  - assess achievement of goals or changes in technology or processes

Predict

- make plans based on relationships between observable attributes and others

Improve

- identify problems constraining current performance that, when overcome, allow improved performance

What can we measure

Products

- System – size, defect density
- Module – length, percent reused, independent flow paths
- Defect – type, origin, detection, severity, effort to fix

Processes

- Development
  - elapsed time – calendar days, working days
  - effort – staff hours, days, months
  - phase containment – defects found in same phase
- Testing
  - tests passed divided by tests executed
- Maintenance
  - staff hours per request
  - number of pending change requests

Projects

- Resources – staff, equipment, tools
- Progress relative to schedule
- Productivity
- Product characteristics
- Process characteristics

Characteristics of Effective Metrics

- Simple and computable
- Empirically and intuitively persuasive
- Consistent and objective
- Consistent in use of units and dimensions
- Programming language independent – where possible
- An effective mechanism for quality feedback

Goal, Question, Metric (GQM)

- Define the principal goals for your activity
- Construct a comprehensive set of questions to help you know whether you are achieving these goals
- Define and gather the data to answer these questions

GQM Example

- Goal 1: Planning
  - How much does the inspection process cost?
  - How much calendar time do inspections take?
- Goal 2: Managing and Controlling
  - What is the quality of the inspected software?
  - To what degree did staff conform to procedures?
- Goal 3: Improving
  - How effective is the inspection process?
  - What is the productivity of the inspection process?

Measuring Internal Product Attributes

- We are interested in the size of our work products
  - effort generally correlates with size
  - other measures can be normalised relative to size
- Common size measures are **LOC** or **Function Points** (abstract measure of functionality)
- Derived measures include
  - LOC / person-hour
  - Defects / KLOC
  - Cost / KLOC

Structure-Related Metrics

- Problem Complexity
- Algorithm Complexity – time and space
- Structural Complexity – control flow, data flow
- Cognitive Complexity – human understanding

Measuring External Product Attributes

- Quality factors are difficult to measure directly – but indirect measures are possible and useful
- A single number metric for software quality is unrealistic – software quality is multidimensional
- Defect density (defects / KLOC or defects / FP) is an accepted measure – but it can be confounded by the defect finding and reporting process

Software Reliability Metrics

- Reliability is a user-focused metric
- Software reliability research builds on hardware reliability theory although the underlying assumptions are not generally valid
  - software defects are design defects not aging
  - software reliability should improve as defects are fixed (modulo ineffective fixes or new defects)
  - reliability depends on inputs and usage

Resource Management

- Productivity – measure of output produced relative to input required
- In software – productivity equals size of product divided by effort (where effort is typically developer time)

- This measure can be problematic
  - size versus value
  - measure of effort
  - quality of output

Making Process Predictions

- It is often important to be able to estimate cost and effort for a project
- An estimate is a prediction that need not be exact – should be sufficiently accurate to be useful
- An estimate should be seen as an interval rather than just a point
- Estimates can and should be based on real data – the choice of the data is important

Measurement Etiquette

Do

- Use common sense and organisational sensitivity when interpreting metrics data
- Provide regular feedback to the individuals and teams who have collected the data
- Work with engineers and teams to set clear goals and the metrics that will be used to measure them

Don't

- Don't use metrics to appraise individuals
- Never use metrics to threaten teams or individuals
- Don't obsess about a single metric to the exclusion of others
- Don't treat metrics data that identifies a problem as "negative" – the data is merely an indicator for process improvement

Software Estimation

Estimation

- It is usually important to be able to estimate cost and effort for a project
- Predication that need not be exact – should be sufficiently accurate to be useful
- Should be seen as an interval rather than just a point
- Should be based on real data – choice of data is important
- Get initial feel for project cost
- Determine if project is still feasible
- Cull and re-prioritise features based on cost
- Estimates are very coarse-grained
  - this is a first pass
  - we inevitably revise these as more info arrives
- Done as a team

Estimation Guidelines

- Use a structured and define process
- Apply throughout the project
- Collect and use historical data
- Adjust to suit new projects
- Apply statistical analysis where possible
- Look for automation opportunities
- Apply more than one technique and compare the results

Estimation Challenges

- Lack of historical data leads to instinctive estimation
  - flawed project plans
  - unhappy customers and programmers
- Estimates tend to be overly optimistic
- Estimation depends on experience and historical data
- Unique aspects of software projects compound the difficulties

How to Estimate

- There are many methods available but most organisations use very primitive methods
- Classes of techniques
  - analogy
  - expert judgement
  - parametric (algorithmic) models
  - "price to win"

How not to Estimate (van Vliet)

- We were given 12 months to do the job, so it will take 12 months
- We know our competitor put in a bid of $1M so we need to deliver a bid of $0.9M

Estimating by Analogy

- Identify one or more similar projects – use these (or parts of them) to produce an estimate for the new project (size or effort)
- Accuracy is often improved by partitioning a project into parts and estimating each part – errors cancel out so long as estimating is unbiased
- Use a database of projects
  - from your own organisation
  - from multiple organisations (International Software Benchmarking Standards Group (ISBSG))

Wide-band Delphi

- Get multiple experts/stakeholders
- Share project information
- Each participant provides an estimate independently and anonymously

Story Points

- Are not units of time – may be "ideal" days
- Consistency – all 2's require the same amount of effort
- Relativity – a 4 is twice as big as a 2

Software Size Measures

- We need to be able to quantify software size in an objective manner – enables comparison
- Types of size measures
  - syntactic (e.g. Lines of Code (LOC))
  - semantic (e.g. Function Points)

Lines of Code

- Widely used, even though there are obvious limitations
  - need a counting standard
  - language dependent
  - hard to visualize early in a project
  - hard for clients to understand
  - does not account for complexity or environmental factors

Function Points

- Developed by Albrecht (1979) at IBM
  - for data processing domain
- Based on a user view of a system
  - external inputs (e.g. input screen)
  - external outputs (e.g. output screen)
  - external enquiries (e.g. prompt for input)
  - internal logical files (e.g. database table)
  - external interface files (e.g. database table shared between applications)

$$\text{Function Points} = 4 \times EI + 5 \times EO + 4 \times EQ + 10 \times ILF + 7 \times EIF$$

- Value Adjustment Factor (VAF) – determined by 14 general system characteristics

  - e.g. operational ease
  - transaction volume
  - distributed data processing
- VAF ranges from 0.65 to 1.35

Difficulties with Function Points

- Counting function points is subjective – even with standards in place
- Counting can not be automated – even for finished system
- Factors are dated and do not account for newer types of software systems – e.g. real-time, GUI, web
- Many extensions to the original function points that attempt to address new types of system

Simple Estimation Technique

- Each entity class in a model will need
  - 1 user interface, 1 data access, and 1/2 a helper class
  - assumes a 3-tier architecture
- Assumes you know developer productivity
  - typical industry ranges – 2 - 20 classes/month (average 4 - 8)

$$E = 3.5 \times \frac{\text{Model Classes}}{\text{Productivity}}$$

COCOMO 2

- Algorithmic model for cost estimation – developed by Barry Boehm at USC
- Based on data from a large number of projects – many were large defence projects
- 3 sub-models
  - account for different development approaches
  - reflect information known at different stages of development

Target Sectors

- Application Composition
  - assumes component reuse, scripting, or DB programming
  - uses application points for size estimates
- Application Development / System Integration / Infrastructure Development – normal development regime

Application Development, System Integration & Infrastructure Development

- Initial Prototyping – uses application composition model
- Early Design
  - applicable after requirements are understood
  - may use function points translated to LOC
- Post-Architecture
  - applicable after architectural design
  - uses LOC for estimates
  - takes into account component reuse and code generation
  - considers complexity factors

Object / Application Points

- Alternative function-related meaure to FP
  - suited to DB programming or scripting languages (4GLs)
  - much easier to estimate than FP
- Object points are not the same as objects in an OO sense
- Can be estimated early in the development process

Application Points Estimation

- Number of separate screens that are displayed
  - simple screens are 1 object point
  - average screens are 2
  - complex screens are 3

- Number of reports that are produced
  - simple reports are 2
  - average reports are 5
  - complex/difficult reports are 8
- Number of 3GL modules that must be developed to supplement the 4GL code – 10 object points per module (e.g. a class in Java)

Application Composition Model

- Early "order of magnitude" estimate – can be more accurate for 4GL intensive projects
- Based on standard estimates of developer productivity in application points / month
  - considers developer experience and CASE tool use
  - doesn't consider system complexity or developer variability

Estimation Formula

$$PM = \frac{NAP \times \left(1 - \frac{\%reuse}{100}\right)}{PROD}$$

Effect of Complexity Multipliers

- Multipliers reflect complexity of project for this team

Multiplier	Complex Project	Simple Project
RCPX (reliability and complexity)	1.5 (very high)	0.75 (low)
RUSE (reuse)	1.0 (none)	0.9 (good amount)
PDIF (platform difficulty)	1.25 (complex)	1.0 (common)
PREX (personnel experience)	1.25 (v. little)	0.9 (experienced)
PERS (personnel capability)	1.5 (v. inexperienced)	0.75 (v. capable)
SCED (schedule difficulties)	1.25 (tight)	1.0 (none)
FCIL (team support facilities)	1.25 (minimal)	0.9 (good toolset)

Project Duration and Staffing

- Minimal calendar time formula

$$TDEV = 3 \times PM^{0.33+0.2\times(B-1.01)}$$

- PM is the effort computation
- B is the exponent calculated previously
  - B is 1 for the early prototyping model
- Assumes an unlimited number of people are available
  - too few people will increase duration

Staffing Requirements

- Staff numbers can't be computed by dividing development time by the required schedule
- Number of people required depends on activities being performed
- The more people who work on the project, the more total effort is usually required
- Rapid build-up of people often correlates with schedule slippage

Agile and Scheduling

- Assumes a stable team – most non-agile processes assume a variable team (adds "spin up" cost)
- Team velocity determines schedule

Velocity

- Team counts number of story points completed at end of each iteration
- Probably will complete about the same number in the next iteration
  - assuming no changes in the team
  - assuming similar technology

- Use velocity to convert story points to calendar effort
  - requires consistent iteration length
- Requires same team on next project

Release Planning

GOAL: Assemble use cases into logical groups for releases and subsequently decomposition of the first release into iterations

1. High-level Solution Refinement
2. Story Identification
3. Story Prioritisation
4. Story Estimation
5. Release Planning
6. Iteration Planning

- Collect use cases into coherent groups of functionality
- Identify the smallest set of use cases that delivers immediate business value
  - initial release
  - subsequent releases are smallest increment that delivers additional business value
- Deployment overhead will influence the size and frequency of releases