Software Engineering

Agile Software Engineering

Software Engineering lifecycle:

- Software developed from scratch, (modern) business applications developed by extending and modifying existing systems, integrating components
- Identifiable states that lead to a software product: four activities that are fundamental to SE…
    - Specification: functionality and constraints on its operation
    - Design and development/implementation: to meet the specification
    - Validation: software must be validated to ensure that it does what the customer wants
    - Evolution (updates): to meet changing customer needs
- More activities:
    - Sub activities: requirement validation, architectural design, unit testing
    - Supporting process activities: documentation, software configuration management

Software Engineering Methodologies:

- Waterfall (plan-driven process): unidirectional, one step at a time
- Incremental (iterative): one step at a time (incremental functionality), but can loop
- Agile: SCRUM
- Reuse-oriented (component based)

Software Process Category:

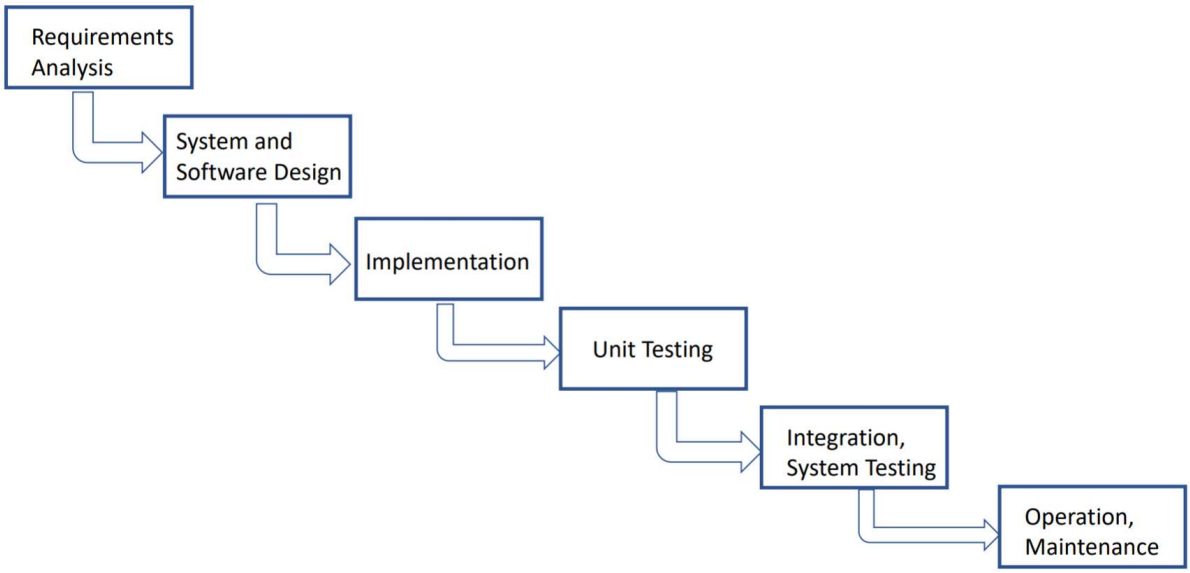
- Plan-driven process:
    - All of the process activities are planned in advance
    - Progress is made and measured again
- Agile Progress:
    - Planning is incremental
    - Easier to change the process

- Reflect changing customer requirements

Which process to use?

- Each process suitable for different types of software
- Find a balance between the two processes

Waterfall:

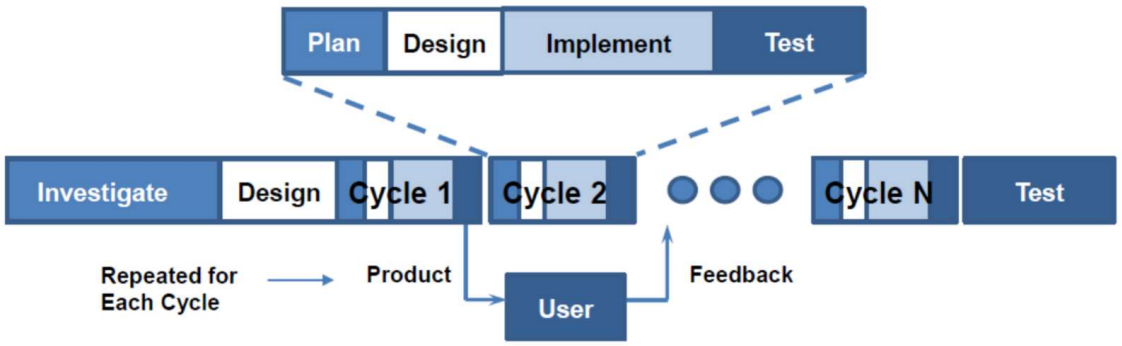


- Used in safety/security-critical systems
- When requirements are very well defined and static, not so common these days
- Small project
- E.g. banking systems – no errors in customer transaction
- Documentation – produced at each phased, manager can monitor progress against plan
- A simple linear model – feedback from one phase to another, documents may be modified to reflect the changes. May involve significant rework
- Advantages:
  - Process visibility
  - Processes are well organised
  - Quality and cost are identifiable
  - Simple and easy to use
- Disadvantages
  - You have to commit early
  - Makes responding to change difficult

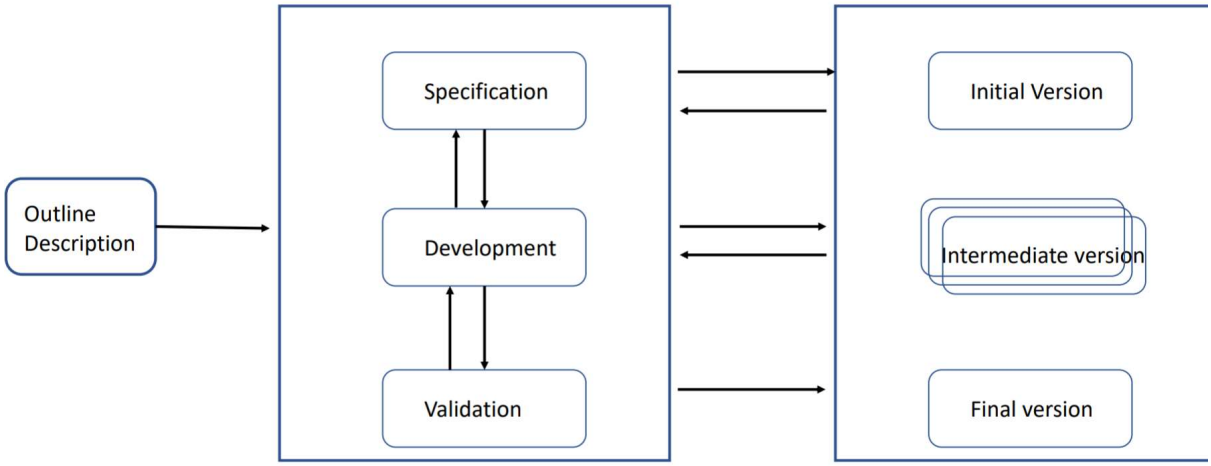
    - Long time to deliver
    - Little/no feedback from customers
- Earliest methodology in software engineering, was the best way to achieve better software needs, careful project planning, rigorous software development process
    - View from developing large, long lived software systems
    - E.g. aerospace, government, banking systems
- Large systems need large teams from different companies
    - Teams geographically dispersed
    - Long periods of development
    - Need significant overhead in planning, designing, and documenting the system

Evolutionary/incremental development

- An example of rapid software development processes: an iterative approach to specification and delivery in order to deliver software quickly.



Evolutionary/incremental Development



Requirements due to change:

- Due to changes in software engineering, it is often impossible to arrive at a stable, consistent set of requirements
- Businesses operate in a global, rapidly changing environment – responds to new opportunities, markets, changing economic conditions. Software is part of almost all business operations
- Impossible to derive a complete set of stable software requirements
- Requirements change or problems recovered, system design and implementation to be reworked and retested
- Some may not change e.g. for a safety-critical control system, complete analysis of system is essential, so waterfall method is the right choice. Takes a long time, slow development.
- However, fast-moving business environments change quickly. Need to focus on rapid software development

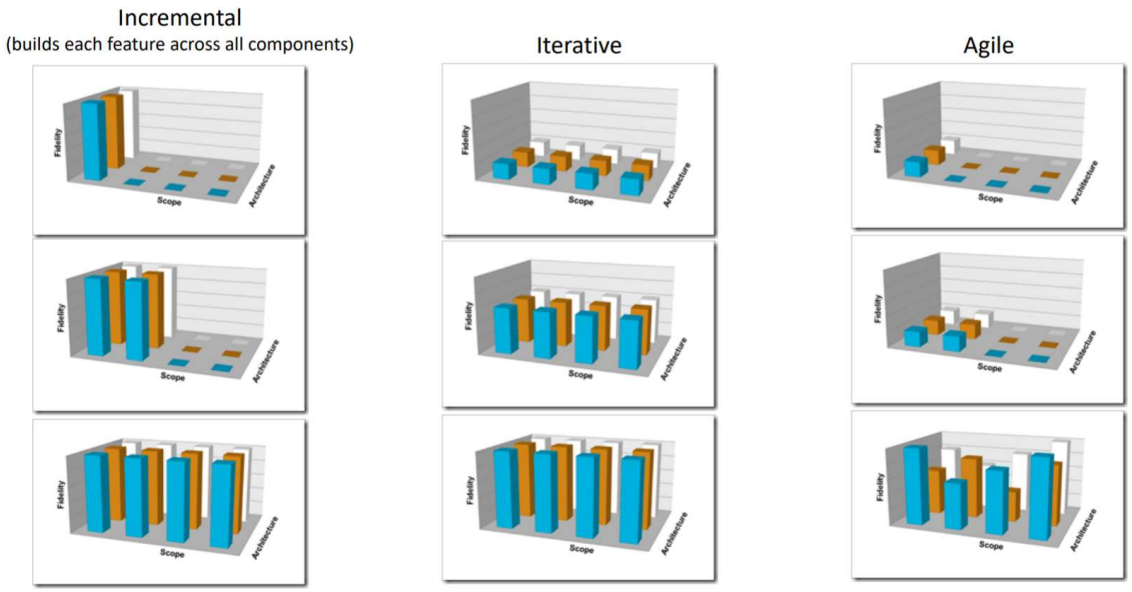
Rapid software development: characteristics:

- The process of specification, design, and implementation are interleaved
    - No detailed system specification
    - Design documentation minimised
    - Requirement documents only define most important characteristics
- System developed in a series of versions
    - End-users and stakeholders involved in specifying and evaluating each version
    - End-users may propose to change requirements in later versions

Agile Software Development:

- An incremental and iterative approach to software specification and development
- Support business where system requirements change rapidly
- A collection of software methodologies and processes: Extreme Programming, SCRUM, adaptive software development, feature-driven development

Incremental, Iterative, and Agile



Agile manifesto:

The four values…

1. Individuals and interactions over processes and tools: valuing people more highly than processes or tools
2. Working software over comprehensive documentation: user stories are sufficient
3. Customer collaboration over contract negotiation: customers involved in the development
4. Responding to change over following a plan: embrace changes, requirement priorities can be shifted, new features can be added

The twelve principles…

1. Customer satisfaction through early and continuous software delivery: customer happier receiving working software at regular intervals
2. Accommodate changing requirements throughout the development process: ability to avoid delays when feature request changes
3. Frequent delivery of working software: SCRUM - team operates in software sprints or iterations, regular delivery
4. Collaboration between the business stakeholders and developers throughout the project: business involved results in better decisions
5. Support, trust, and motivate the people involved: motivated teams work better than unhappy teams

6. Enable face-to face interactions: communication is more successful
7. Working software is the primary measure of progress: delivering functional software is the ultimate factor measuring progress
8. Agile processes to support a consistent development pace: teams establish repeatable and maintainable speed
9. Attention to technical detail and design enhances agility: right skills and good design ensures the team to maintain the pace
10. Simplicity: develop just enough to get the job done for now
11. Self-organizing teams encourage great architectures, requirements, and designs: sharing ideas delivers quality products
12. Regular reflections on how to become more effective: self-improvements, process involvements to help team work more efficiently

Advantages…

- Fast delivery (iterations)
- Allows for lots of client feedback
- Does not try to tackle the whole problem right away

Disadvantages…

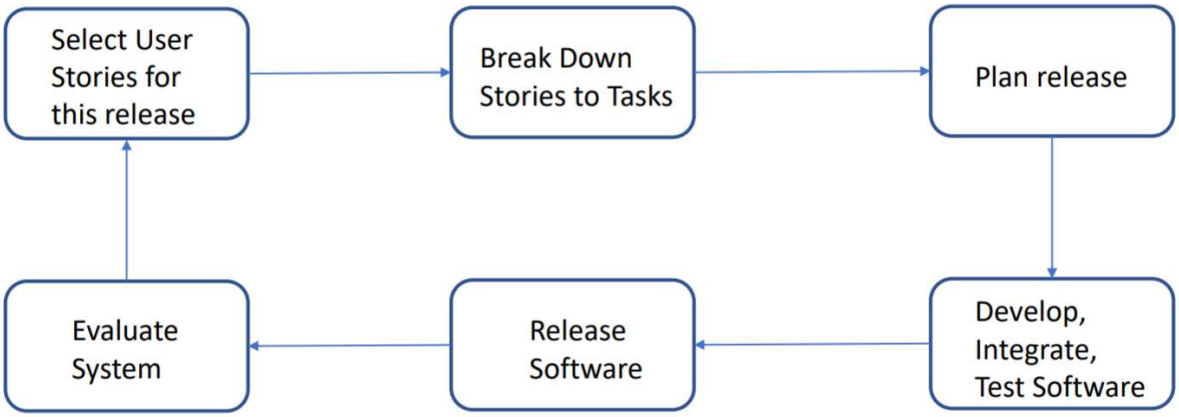
- Lots of time with clients, not developing
- Clients may not be willing and able to spend time with the development team (subject to other pressures)
- No long term planning
- Prioritising change can be difficult: may have stakeholders, each gives different priorities
- Contracts: customers pay for the time required for development, rather than the development of a specific set of requirements

When it should be used…

- Product development: developing a small or medium-sized product for sale
- Custom system development within an organisation:
    - A clear commitment from customer
    - Customer involved in development process
    - Not a lot of external rules and regulations that affect the software

Extreme Programming (XP):

- An agile process: the focus is on code rather than design or documentation
- XP takes an 'extreme' approach to iterative development
    - New versions can be built several times per day
    - Increments are delivered to customers every 2 weeks
    - All tests must be run for every build: build is only accepted if tests run successfully
- In XP, requirements expressed as 'stories'
- These stories are written on cards, the development team breaks into tasks and the tasks are the basis of schedule and costs estimates
- Customer chooses the stories for inclusion in the next release based on priorities and schedule estimates



The XP release cycle

User Stories:

- Convenient format for expressing the desired business values
- Crafted to be understandable to both business people and technical people
- Serve the same purpose as Use Cases but are not the same as they are used to create time estimates for release planning
- Used instead of a large requirement document
- Written by the customer: the things the system needs to do for them and they are written in about 3 sentences
- Should only provide enough detail to make a low risk estimate of how long it will take to implement

- Different form requirement specifications as these need enough detail to implement
- When it is time to implement, developers go back to customer for a detailed description of requirements
- User stories are cards, conversation, confirmation

What exactly are User Stories?

- **Card**
- Stories could be written in the form:

As a <*type of user*>, I want <*some goal*> so that <*some reason*>.

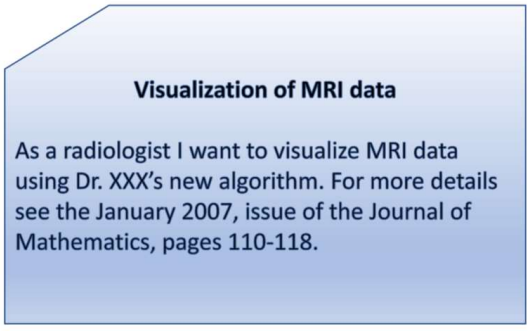
What exactly are User Stories?

- **Conversation**

Details of a requirement are **communicated** among the development team, product owner and stakeholders

- **Conversation and collaboration** ensure that the **correct requirements** are expressed and understood by everyone

Conversation is verbal, but may need supplemented documents



What exactly are User Stories?

- **Confirmation**

Contains confirmation information in the form of conditions of satisfaction

- **Acceptance criteria** that clarify the desired behaviour

Better understand what to build and test.

Upload File

As a wiki user I want to upload a file to the wiki so that I can share it with my colleagues.

Conditions of Satisfaction

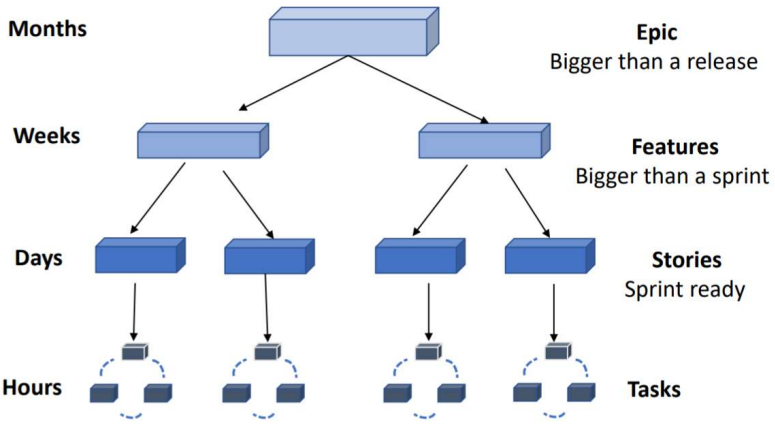
Verify with .txt and .doc files
Verify with .jpg, .gif, and .png files
Verify with .mpg files <= 1 GB
Verify files with no digital rights management restrictions

1. Stakeholders write them: simple and straightforward and the domain expert should write them
2. Use the simplest tool: write them on index cards
3. Indicate the estimated size: estimate the effort or team size that is needed to implement
4. Indicate the priority: what is the priority of the story?
5. Use a unique identifier: allows for traceability

User story abstraction hierarchy

❑User stories are written at various levels of abstraction:

Epic, feature, theme, stories (sprintable), tasks (not stories)



Example

- Epic:
  - As a hotel operator, I want to set the optimal rate for rooms in my hotel
- Room rate is a function of several factors.
- Stories:
  a) As a hotel operator, I want to set the optimal rate for rooms based on previours year pricing
  b) As a hotel operator, I want to set the optimal rate for rooms based on what hotels comparable to mine are charging
- Further split for b):
- Stories:
  1) As a hotel operator, I can define a `Comparable Set' of hotels
  2) As a hotel operator, I can add a hotel to a Comparable Set
  3) As a hotel operator, I can remove a hotel from a Comparable Set
  4) As a hotel operator, I can delete a Comparable Set I no longer wish to use
  5) As a hotel operator, rates charged at hotels in a Comparable Set are used to determine rates at my hotel

XP Customer:

- Part of the team
  - On site, available at all times
  - XP principles: communication and feedback
  - Making sure we build what the customer wants
- Actively involved in all stages:
  - Clarify the requirements
  - Negotiation with the team, what to do next
  - Define acceptance tests
  - Constantly evaluate intermediate versions

Testing in XP:

- Test-driven development
  - Write tests before code – instead of writing some code and then writing the tests for that code, you write the tests before you write the code
  - Tests are automated – you can run the tests as the code is being written and discover problems during development
- Acceptance testing
  - The process where system is tested using customer data to check it meets customer's real needs
  - Written with customer, act as a 'contract', measure of progress
- Unit testing: automate testing of functionality as developers create it

Test-Driven Development:

- Test first: Before we write any code, write a test for that feature
- Automated: tests are run automatically each time a release is built
- Unit tests: automate testing of functionality as developers write it
- Acceptance tests: specified by the customer to test the overall system

Pair Programming:

- Two programmers work side-by side at one computer
- Helps develop common ownership of code – spread knowledge across the team
- Continuously collaborate on same design, algorithm, code, test, etc.
- It serves as an informal review process – each line of code is looked at by more than 1 person
- It encourages refactoring as the whole team can benefit from this
    - Refactoring: a practice of software development that allows you to improve the code without changing or breaking its functionality
    - All developers expected to refactor the code continuously as soon as possible code improvements are found
    - Helps to keep the code simple and maintainable

Reuse-oriented software engineering:

- For a long time, software engineering has been focused on original development
    - We've realised that we only need to invent one wheel
    - We need to adopt a design process that is based on systematic reuse
- Reuse is the ability to use already written code in a program it wasn't written for
- OOP supports this (classes, encapsulation)
- Application system reuse: the whole of an application system may be reused by incorporating it without change (e.g. web application framework – don't change framework code, add concrete classes that inherit operations form abstract classes in the framework)
- Component reuse: components of an application from sub systems to single objects may be reused (e.g. data mining, text-processing system may be used in different applications)

- Function reuse: software components that implement a single well-defined function may be used (e.g. mathematical functions, object classes)

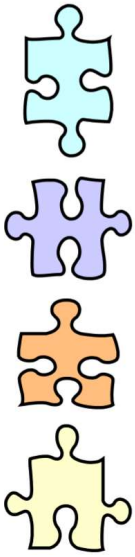
Requirements for design with reuse:

- It must be possible to find appropriate reusable components
    - The re-user of the component must be confident that the components will be reliable and behave as specified
- The components must be documented so that they can be understood, and where appropriate, modified

Benefits of Reuse:

- Increased reliability – components exercised in working systems
- Reduced process risk – less uncertainty in development costs
- Effective use of specialists – reuse components instead of people
- Accelerated development – avoid original development and hence speed-up production

Advantage 1: Software Construction

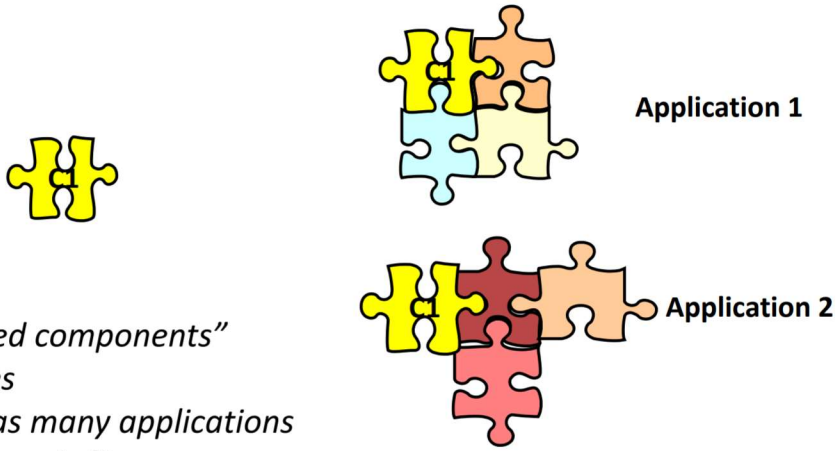


Construction?

Creation?

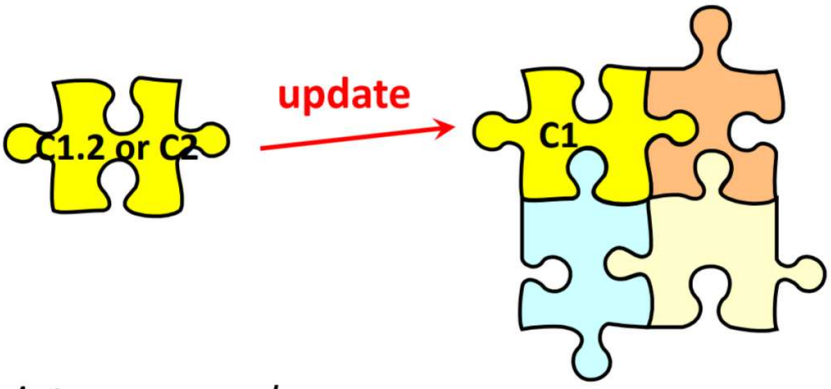
- ***Software construction*** *vs.* ***creation****: application is developed as an assembly of "integrated components"*

Advantage 2: Reuse



Software "integrated components" are reusable entities
It pays off to have as many applications that reuse an entity as is fit

Advantage 3: Maintenance & Evolution



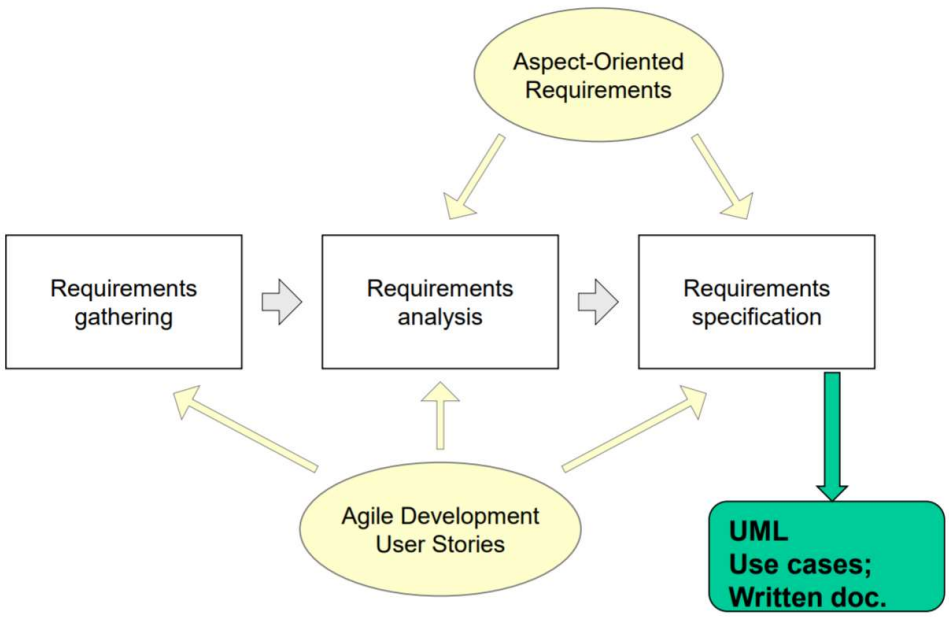
Maintenance and upgrading can be done by replacing parts, maybe even at runtime

Requirements Engineering

Software Requirements:

- A requirement specifies the business functions that the user will be able to perform using the system-to-be in different situations of contexts, and the kind of experience the user will have during this work, and other concerns such as how the system will manage resource (computing, network, etc.) or how the systems will manage and protect the users' data
- User requirements will often be high-level, vague and incomplete, they are more like high-level goals, or business goals, rather than software requirements needed by the developer
- When trying to achieve a give high-level goal, we will need to consider what matters, what are the important parameters, so that we can derive the detailed technical requirements
- Only based on deeper understanding of detailed issues, we can identify important scenarios or situations and identify what parameters should be considered in each situation
- Then using these parameters, we decide what the system should do, or how to respond to this situation (i.e. inputs)

Requirements Process

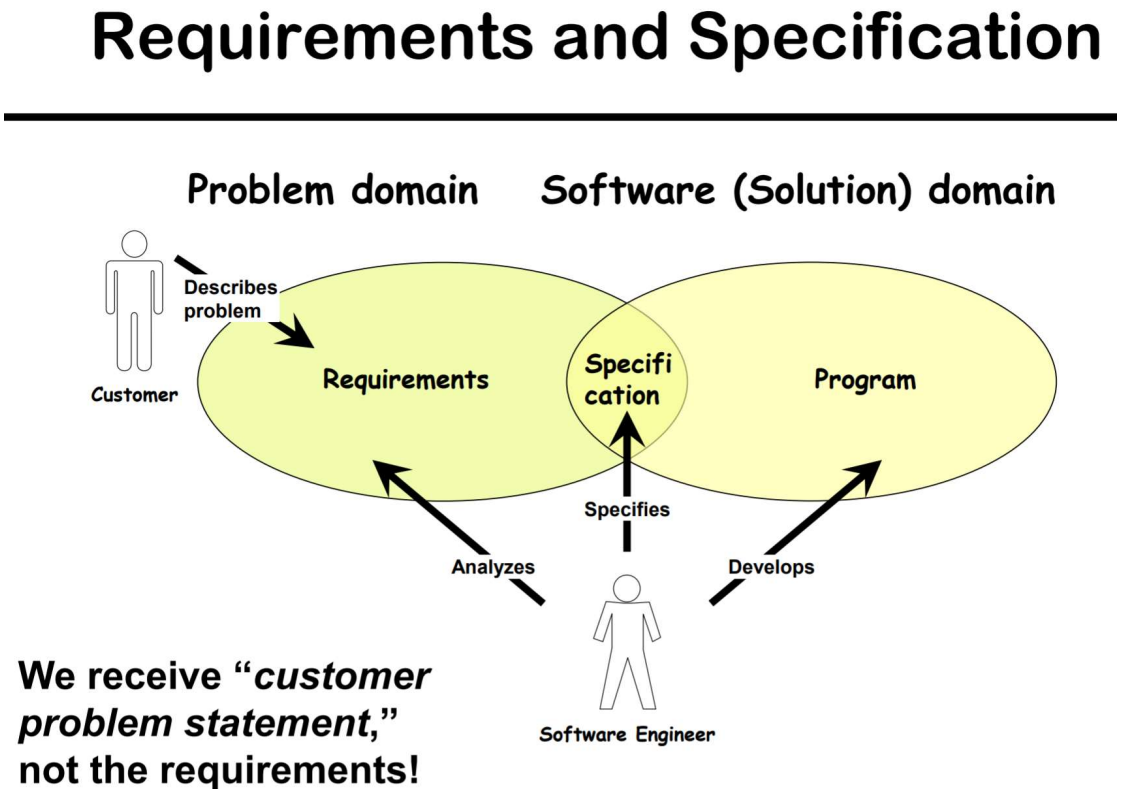


Requirements Engineering Components:

- Requirement gathering – (aka. Requirements Elicitation) helps the customer to define what is required: what is to be accomplished, how the system will fit into the needs of the business, and how the system will be used on a day-to-day basis
- Requirements analysis – refining and modifying the gathered requirements
- Requirements specification – documenting the system requirements in a semiformal or formal manner to ensure clarity, consistency, and completeness

Practical Requirements Engineering:

- Test your idea in practice and use the result in further work, iterating through these creative and evaluative steps until a solution is reached – nobody can know all the constraints of a solution before they go through the solving experience
- Define the criteria for measuring the success – "acceptance tests"
- Avoid random trail-and-error by relying on domain knowledge (from publications or customer expertise)



Requirements Derivation:

Key tasks of Requirements Engineering…
Defining the problem and its causes, and solution constraints (different from detecting that a problem exists)
Depending on the cause, the solution will be different
Requirements are determined by:

- Judgement about customers' business needs and causes preventing their achievement

- Conditions on solutions imposed by real-world constraints
    - Physical
    - Social/cultural
    - Legal
    - Financial
- Threats created by adversaries

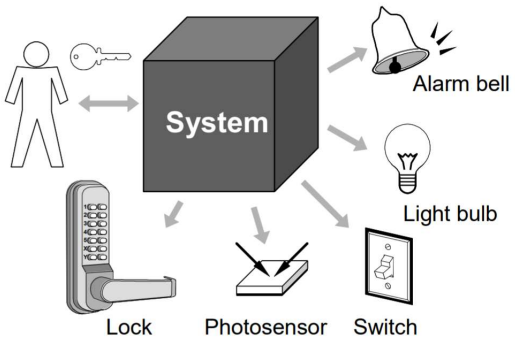
Requirements are not simply desires
Requirements are desires adjusted to real-world constraints and threats

Problem Analysis Examples

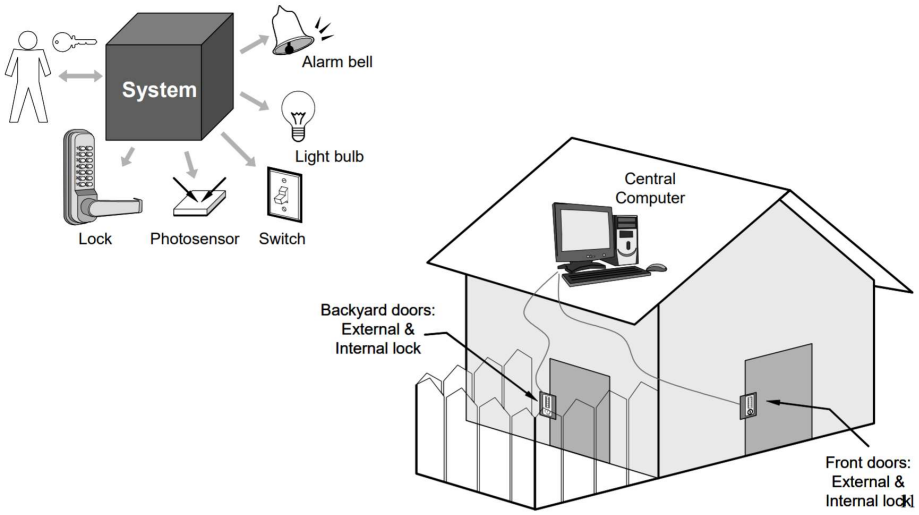
- Traffic Monitoring:
  - Problem: User is delayed to work and needs help
  - Cause A: Traffic is unpredictable (predictably high if repeated delays?!)
  - Cause B: User is unfamiliar with the route
  - Cause C: User has a habit of starting late
- Restaurant Automation:
  - Problem: Restaurant is operating at a loss and customers are unhappy
  - Cause A: Staff is encumbered with poor communication and clerical duties, resulting in inefficiencies
  - Cause B: The menu does not match the likeliest customer's taste
  - Cause C: Staff is misbehaving with customers or mismanaging the food supplies
- Health Monitoring:
  - Problem: User is leading unhealthy lifestyle and experiencing health problems
  - Cause A: User is trying to be active but unsure about sufficient level of activity
  - Cause B: User is trying to be active but too busy with other duties
  - Cause C: User cannot find affordable solutions for active lifestyle
  - Cause D: User has poor sleep
  - Cause E: User is aware of issues but not motivated to act

Case Study: Home Access Control

- Objective: Design an electronic system for:
  - Home access control
    - Locks and lighting operation
  - Intrusion detection and warning



Case Study – More Details



Problem Example: Safe Home Access

- Problem detected:
  difficult access or unwanted intrusion
  (plus: operating household devices and minimizing living expenses)
- Analysis of the Causes:
  - User forgets to lock the door or turn off the devices
  - User loses the physical key
  - Inability to track the history of accesses
  - Inability to remotely operate the lock and devices
  - Intruder gains access
- System Requirements: based on the selected causes

Requirements as User Stories:

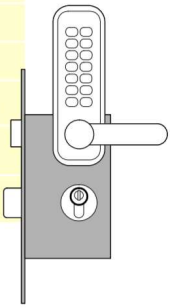
As a tenant, I can unlock the doors to enter my apartment.

user-role (benefactor) | capability | business-value

- Preferred tool in agile methods
- Focus on the user benefits
- User stories are less formal and empathise automated acceptance tests

Example User Story Requirements

Identifier	User Story	Size
REQ-1	As an authorized user, I will be able to keep the doors by default always locked.	4 points
REQ-2	As an authorized user, I will be able to unlock the doors using a valid key.	7 points
REQ-3	An intruder will not be able to unlock the doors by guessing a valid key; the system will block upon a "dictionary attack."	7 points
REQ-4	The door will be automatically locked after being open for a defined period of time.	6 pts
REQ-5	As a user, I will have the door keypad backlit when dark for visibility.	3 pts
REQ-6	Anyone will be able to lock the doors on demand.	2 pts
REQ-7	As a landlord, I will be able at runtime to manage user authorization status.	10 pts
REQ-8	As an authorized user, I will be able to view the history of accesses and investigate "suspicious" accesses.	6 pts
REQ-9	As an authorized user, I will be able to configure the preferences for activation of household devices.	6 pts



- Note no priorities for user stories
- Story priority is given by its order of appearance on the work backlog (described later)
- Estimated size points (last column) will be described later

14

Requirements Analysis

- Requirement REQ-3 states that intruders will not be able to succeed with a "dictionary attack," but many details need to be considered and many parameters determined ("business policies")
  - What distinguishes user's mistakes from "dictionary attacks"
    - The number of allowed failed attempts, relative to a predefined threshold value
      - The threshold shall be small, say three ← business policy!
  - How is the mechanical lock related to the "blocked" state?
    - Can the user use the mechanical key when the system is "blocked"?
- Requirement REQ-5 states that the keypad should be backlit when dark
  - Is it cost-effective to detect darkness vs. keep it always lit?
- Etc.

Requirements analysis should not be exhaustive,
but should neither be avoided.

Requirements Analysis Activities:

- Not only refinement of customer requirements, but also feasibility and how realistic
- Needs to identify the points where business policies need to be applied
- Explicit identification of business policies (BP) is important

Types of Requirements:

- Functional Requirements
- Non-Functional Requirements (or Quality Requirements)
  - FURPS+
  - Functionality (security), Usability, Reliability, Performance, Supportability
- User Interface Requirements

Example: Safe Home Access

User interface requirement:
The UI must be simple for occasional visitors who will not have time to familiarize or to study user's documentation.

Initial design of the door keypad:

Analysis:

- If "Unlock" button is *not* available, then the system could simply take the **first** *N* digits and validate as the key.
- If "Unlock" button *is* available, then the system could take the **last** *N* digits and validate as the key (and ignore any number >*N* of previously entered digits).
- The advantage of the latter approach is that it allows correcting for unintentional mistakes during typing, so the *legitimate* user can have more opportunities to attempt.
- Note that this feature will not help the burglar trying a dictionary attack.

18

Example: Safe Home Access

Redesigned door keypad: includes the "Unlock"&"Lock" buttons

Analysis:

- When a user types in the key and the system reports an invalid key, the user may not know whether the problem is in his fingers or in his memory: "*did I mistype the correct key, or I forgot the key*?"
- To help the user in such a situation, we may propose to include a numerical display that shows the digits as the user types.

19

Example: Safe Home Access

Re-redesigned door keypad: includes the keycode display

Analysis:

- There are several issues to consider about the display feature:
  - How much this feature would increase the overall cost?
  - Would the display make the door device bulky and inconvenient to install?
  - Would it be significantly more sensitive to rain and other elements?
  - Would the displayed information be helpful to an intruder trying to guess a valid key?

20

Acceptance Tests:

- Each requirement describes for a given 'situation' (i.e. system inputs), the output or behaviour the system will produce – the output represents the user's need or business goal
- An acceptance test specifies a set of scenarios for determining whether the system meets the customer requirements
- An acceptance test case specifies, for a given situation or context (defined by current system inputs), the output or behaviour the system will produce in response

Agile Methods for Effort Estimation and Work Organization

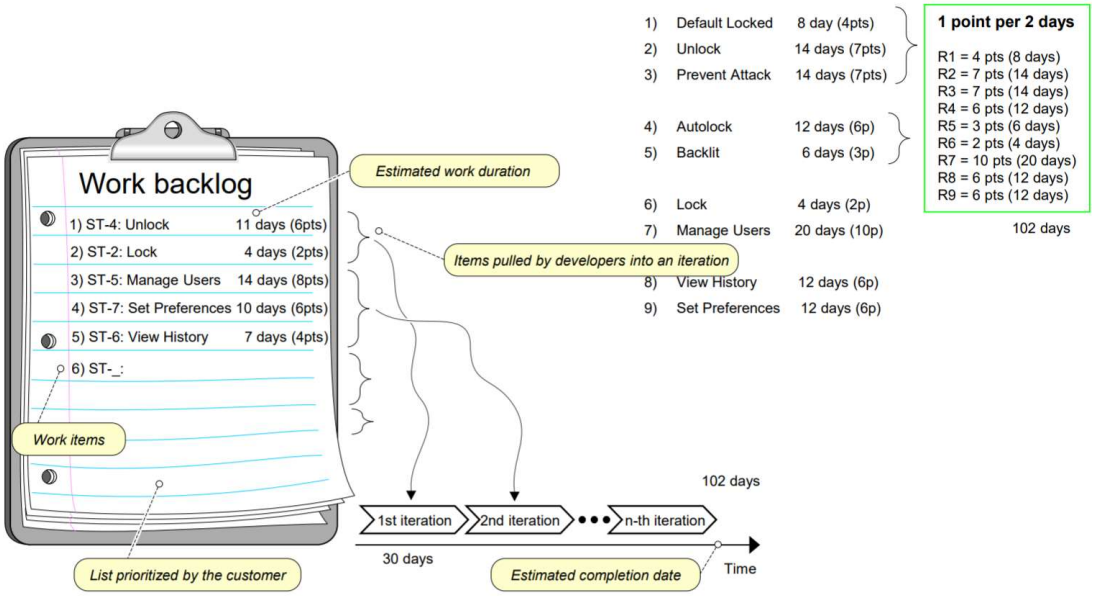
- Size points assigned to each user story
- Total work size estimate:
  - Total size = $\sum$ (points-for-story i), $i = 1..N$
- Velocity (= productivity) estimated from experience
- Estimate the work duration

$$\text{Project duration} = \frac{\text{Path size}}{\text{Travel velocity}}$$

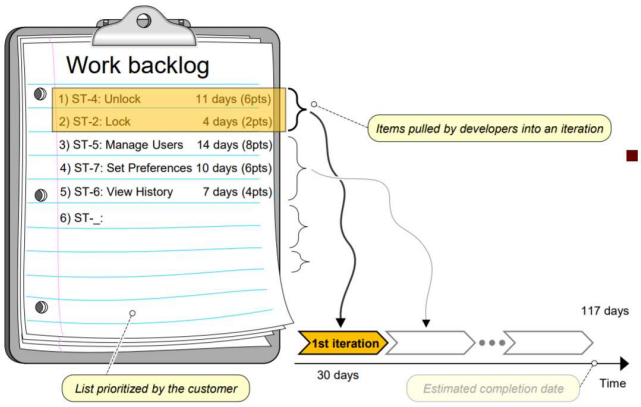
Example User Stories

Identifier	User Story	Size
REQ-1	As an authorized user, I will be able to keep the doors by default always locked.	**4 points**
REQ-2	As an authorized user, I will be able to unlock the doors using a valid key.	**7 points**
REQ-3	An intruder will not be able to unlock the doors by guessing a valid key; the system will block upon a “dictionary attack.”	**7 points**
REQ-4	The door will be automatically locked after being open for a defined period of time.	**6 pts**
REQ-5	As a user, I will have the door keypad backlit when dark for visibility.	**3 pts**
REQ-6	Anyone will be able to lock the doors on demand.	**2 pts**
REQ-7	As a landlord, I will be able at runtime to manage user authorization status.	**10 pts**
REQ-8	As an authorized user, I will be able to view the history of accesses and investigate “suspicious” accesses.	**6 pts**
REQ-9	As an authorized user, I will be able to configure the preferences for activation of household devices.	**6 pts**

Agile Estimation of Project Effort

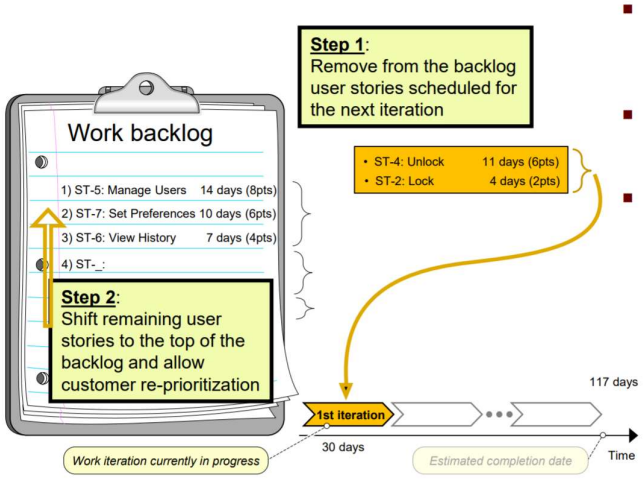


Agile Prioritization of Work



- Instead of assigning priorities, the customer creates an ordered list of user stories
- Developers simply remove the top list items and work on them in the next iteration

Tradeoff between Customer Flexibility and Developer Stability



- Items pulled by developers into an iteration are not subject to further customer prioritization
- Developers have a **steady goal** until the end of the current iteration
- Customer has **flexibility** to change priorities in response to changing market forces

Software Engineering Problems:

- We assume that the computer/software helps the user to achieve a business goal in the problem domain
- Problem domains can be virtual (e.g. text documents, relational databases) or physical world (a device to control, a person to notify and prompt action
- Problem Types:
    - Transforming one virtual object to another (document format conversion)
    - Modifying s virtual object (e.g. document editing)
    - Automatically controlling behaviour of a physical object (e.g. thermostat)
    - Manually controlling behaviour of a physical object (e.g. drone flying)
    - Observing behaviour of a physical object (e.g. traffic monitoring)

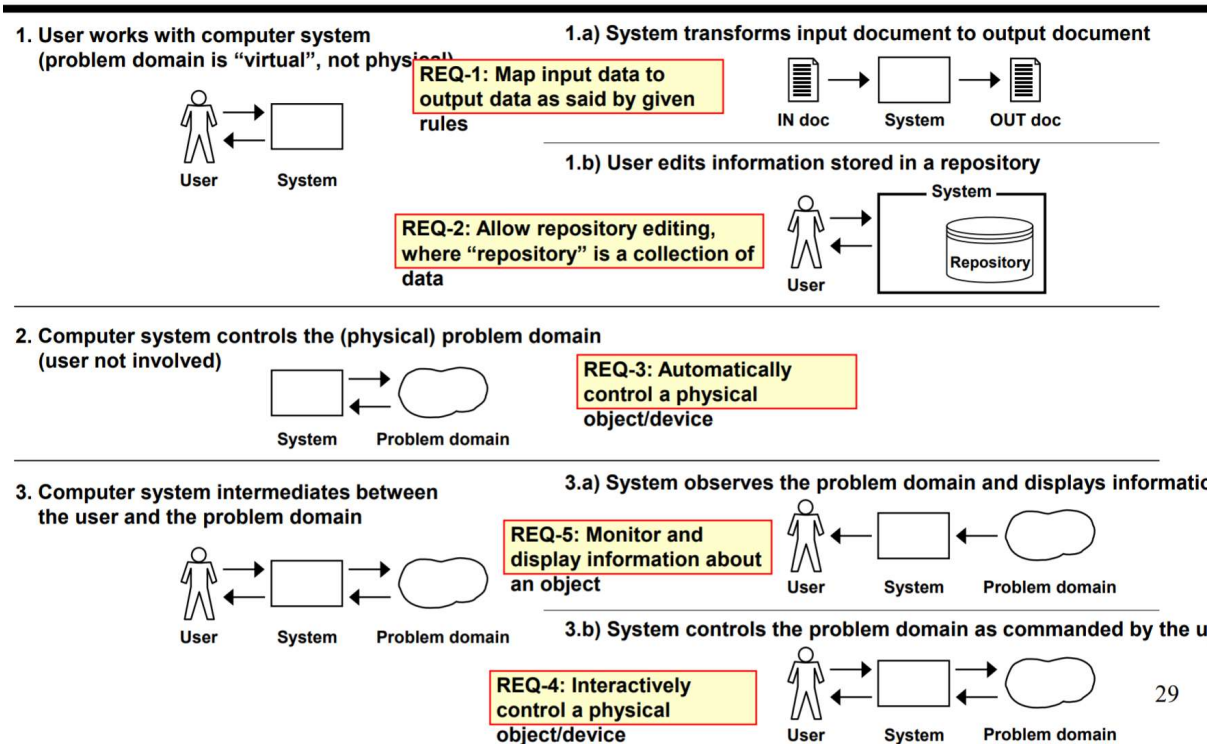
Typical System Requirements

- **REQ-1**: Map/transform input data to output data as said by given rules
- **REQ-2**: Support repository (or *document*) editing, where "repository" is a collection of data items
- **REQ-3**: Automatically control a physical object/device
- **REQ-4**: Interactively control a physical object/device
- **REQ-5**: Monitor and display information about an object

➔ Only a "5-dimensional" problem space!

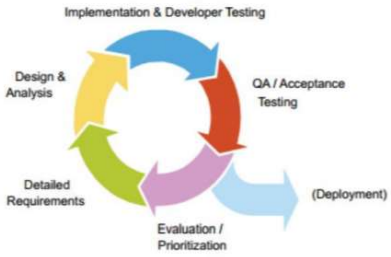
(Problem complexity can vary independently along each dimension)

Typical Software Eng. Problems



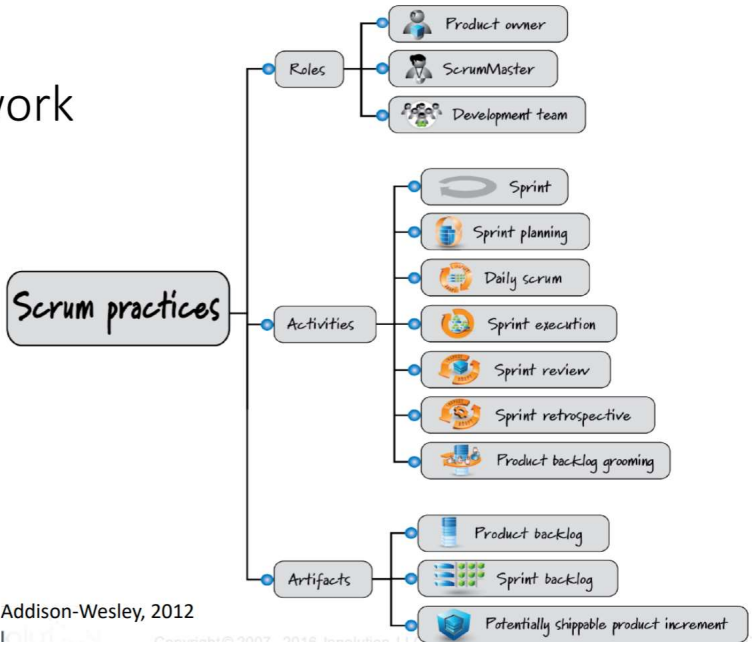
SCRUM

SCRUM is an agile approach for developing products and services



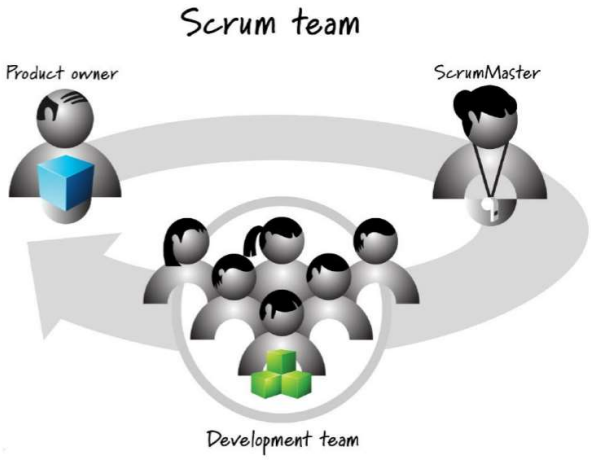
SCRUM framework

- A **framework** for organising and managing work



Source: `Essential Scrum', K.S.Rubin, Addison-Wesley, 2012

SCRUM Team (Roles):



Product Owner:

- Responsible for what will be developed and in what order
- Empowered central point of product leadership
- Deciding which features and functionality to build

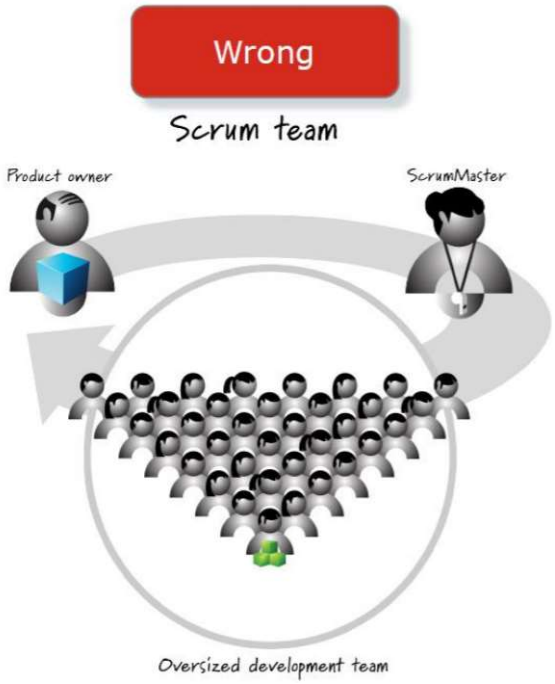
Scrum Master:

- Responsible for guiding the team in creating and following its own process based on the broader framework
- A coach, providing process leadership, helping the team

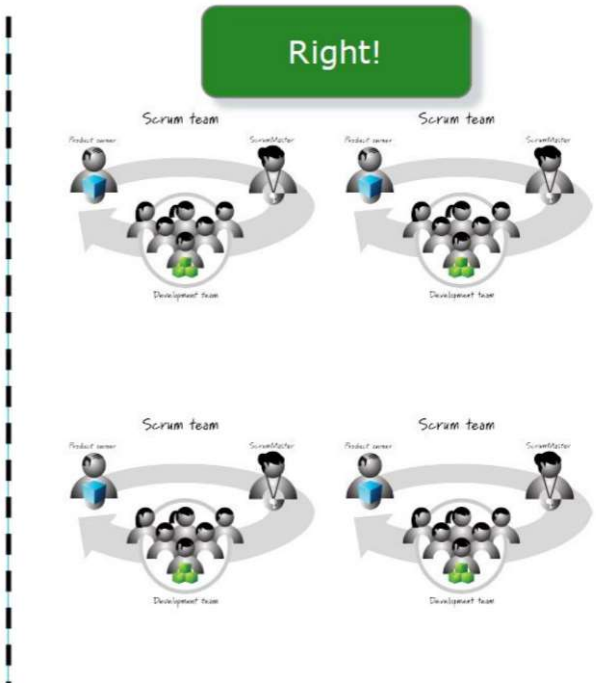
Development Team:

- Responsible for determining how to deliver what the product owner has asked for
- 5-9 people in size
- Must collectively have all of the skills needed to produce good quality, working software – programmers, testers, user experience designers, etc.

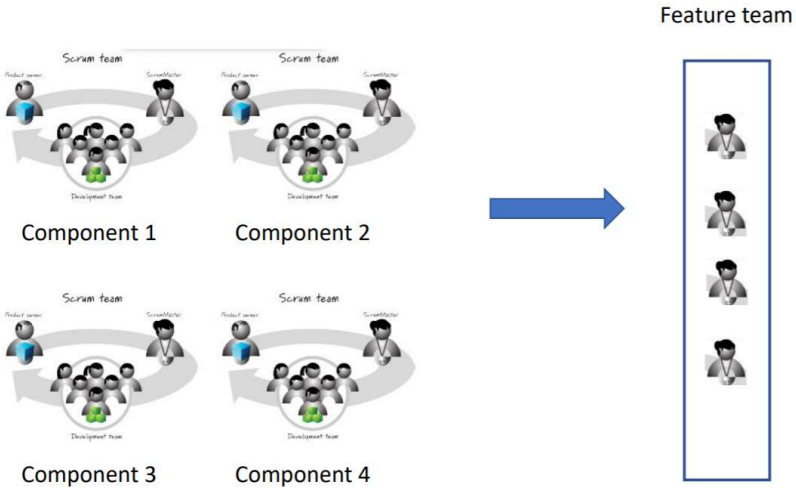
The team: too many



The team: 5-9 people

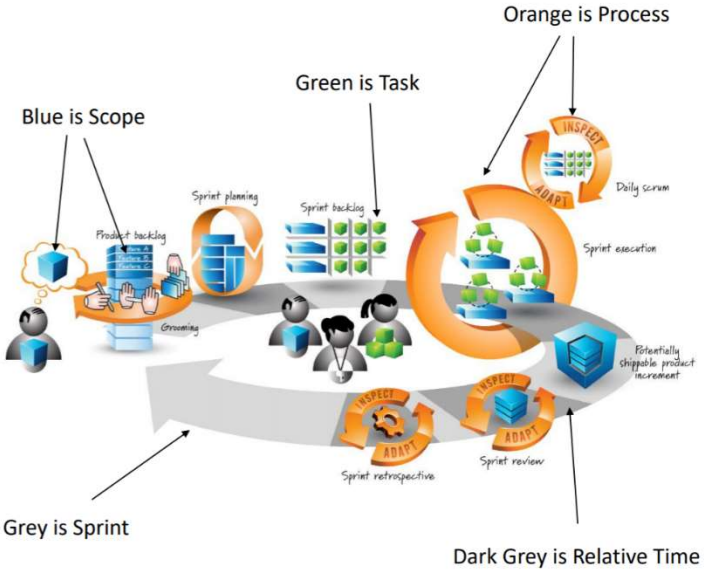


SCRUM team: combined component teams

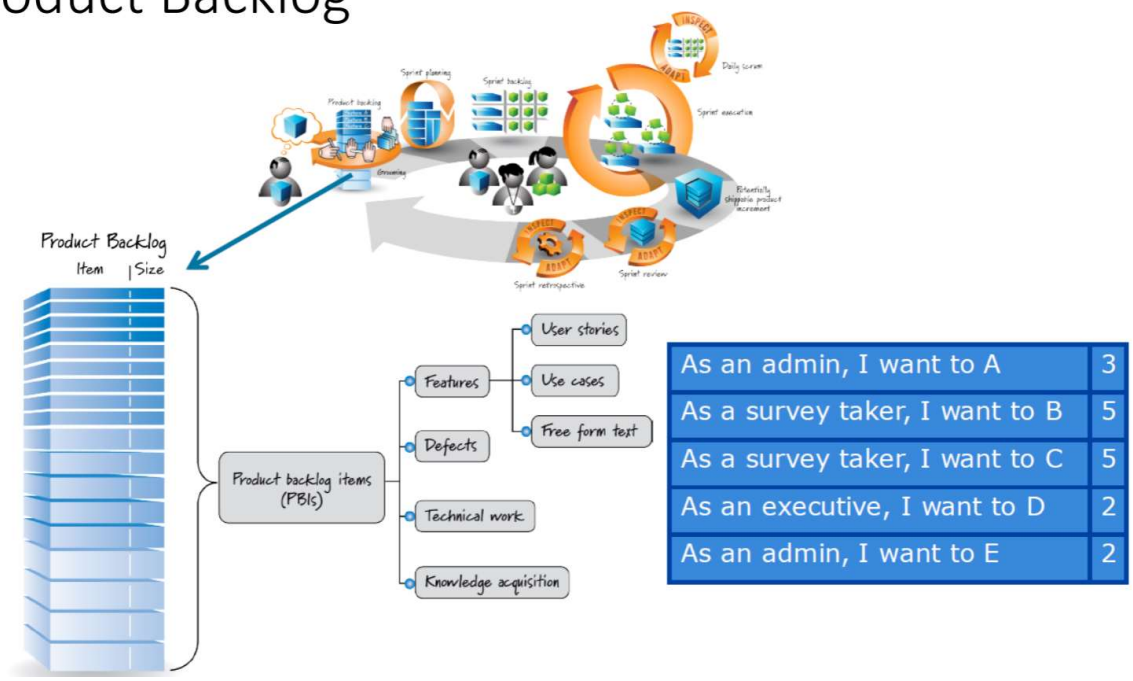


SCRUM Activities:

- Product owner has a vision of what to create – cube is too large
- Grooming to set of features – results in a prioritised list, product backlog
- A sprint…
    - Starts with sprint planning
    - Team determines a subset of product backlog
    - Encompasses the development working during sprint – sprint execution
    - Ends with review and retrospective

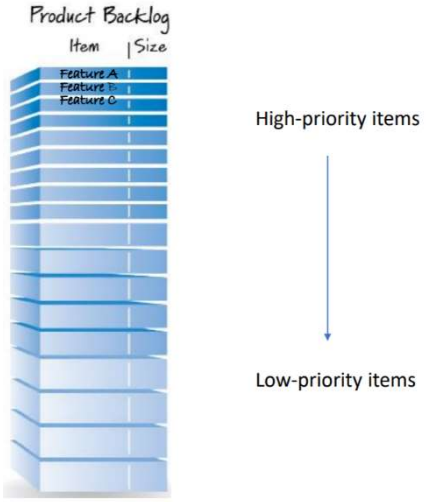


Product Backlog

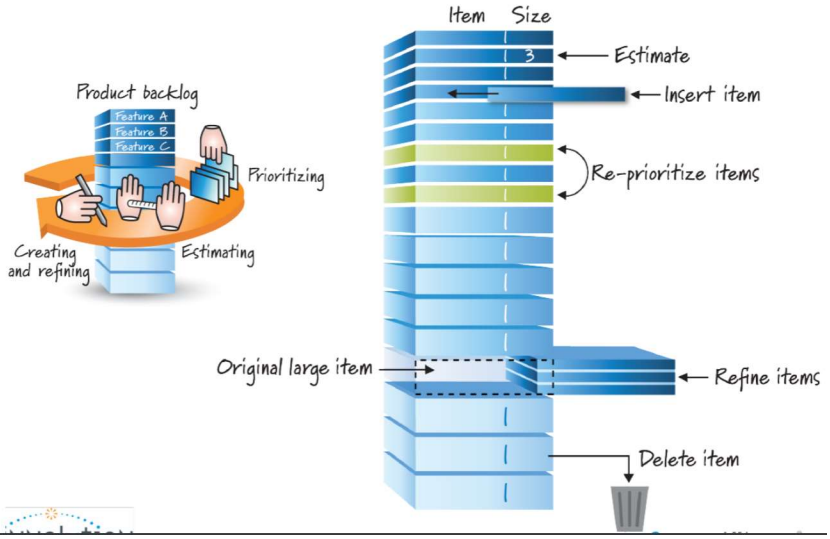


Product Backlog:

- Product owner is responsible for determining the product backlog
- Product backlog are features required to meet the product owner's vision
- Product backlog items are placed in the correct sequence – value, cost, knowledge, and risk

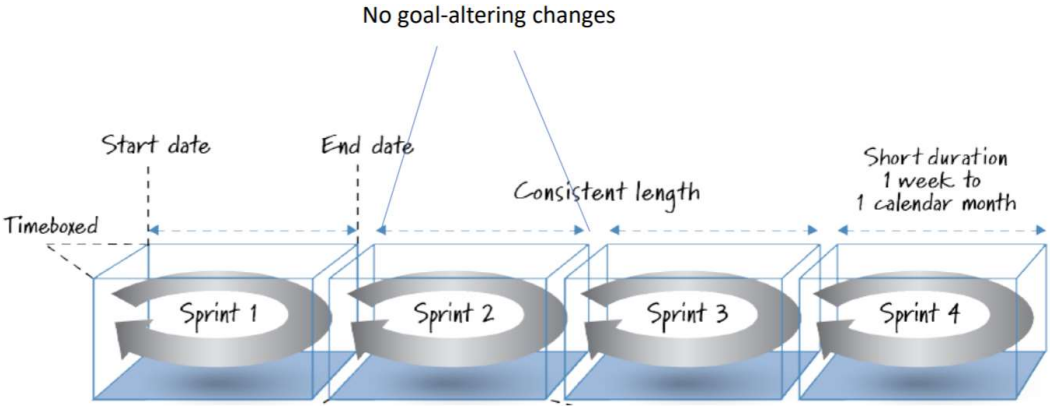


Grooming Activities



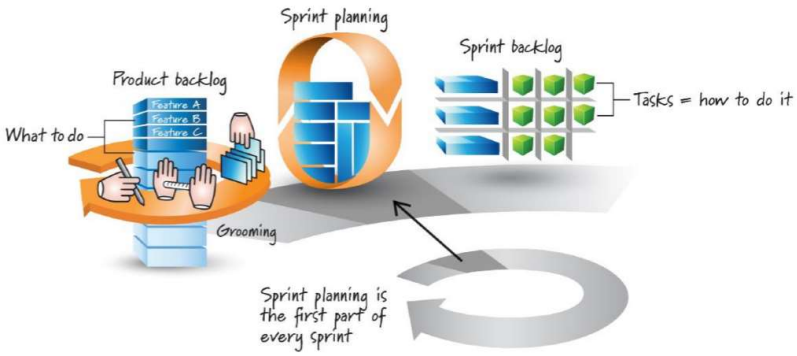
Sprints:

The work completed in each sprint should create something of tangible value to the customer or user

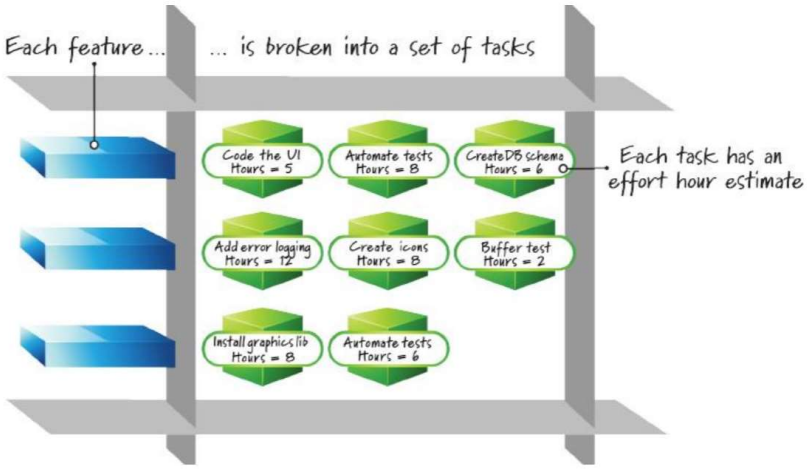


Sprint Planning:

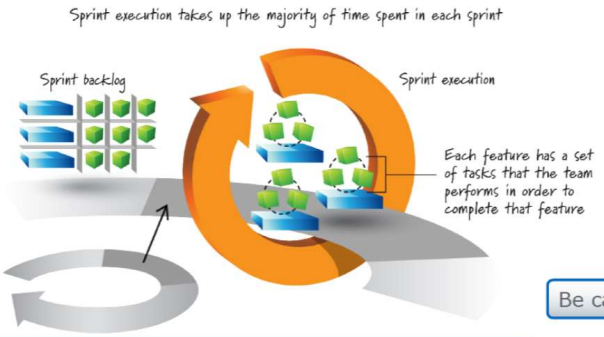
- Determine the most important subset of product backlog items
- During sprint planning, product owner and development team agree on a sprint goal – what the upcoming sprint is supposed to achieve
- In order to acquire confidence in what can get done, the selected features forms a second backlog – sprint backlog



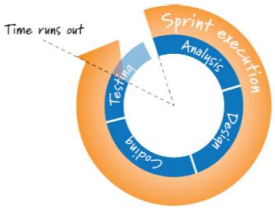
Sprint Backlog



Sprint Execution

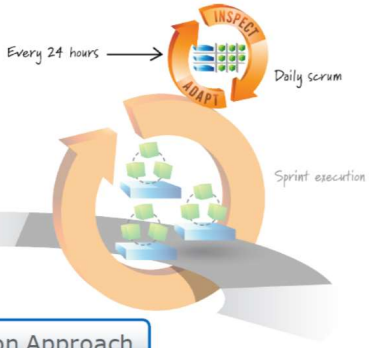


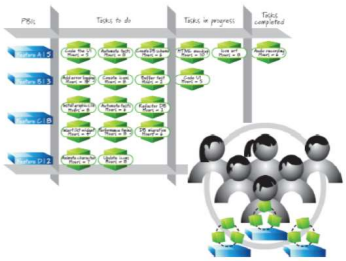
Be careful of mini-waterfall!



- How many PBIs should the team work on in parallel?
- When should work begin on a specific PBI?
- How should task-level work be organized?
- Who should do the work?

Daily Scrum





Common Approach

- What did I accomplish since the last daily scrum?
- What do I plan to work on by the next daily scrum?
- What obstacles are preventing me from making progress?

Every day

15 minutes

August

S	M	T	W	TH	F	S
31	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

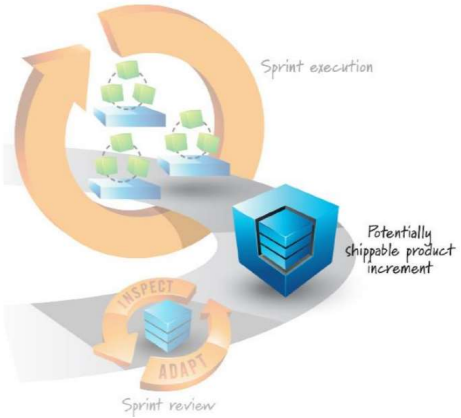
Story	To Do	In Process	To Verify	Done
As a user, I... 8 points	Code the... 9; Test the... 8; Code the... 2; Code the... 8; Test the... 8; Test the... 4	Code the... DC 4; Test the... SC 8	Test the... SC 6	Code the... D; Test the... SC 8; Test the... SC; Test the... SC; Test the... SC 6
As a user, I... 5 points	Code the... 8; Test the... 8; Code the... 4; Code the... 6	Code the... DC 8		Test the... SC; Test the... SC; Test the... SC 6

Daily SCRUM:

- Everyone answers 3 questions…
  - What did I accomplish since the last daily scrum?
  - What do I plan to work on by the next daily scrum?
  - What are the obstacles or impediments that are preventing progress?
- These are not for the Scrum Master to check on people, they are commitments in front of peers

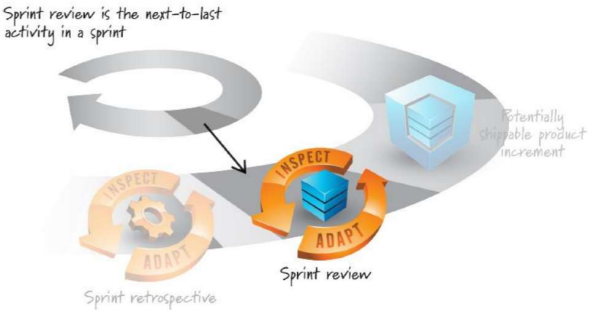
Done: potentially shippable product increment

- Whatever the Scrum team agreed to do is really done according to its agreed-upon definition of done



Sprint Review

❑Give everyone with input into product development an opportunity to inspect and adapt what has been built so far

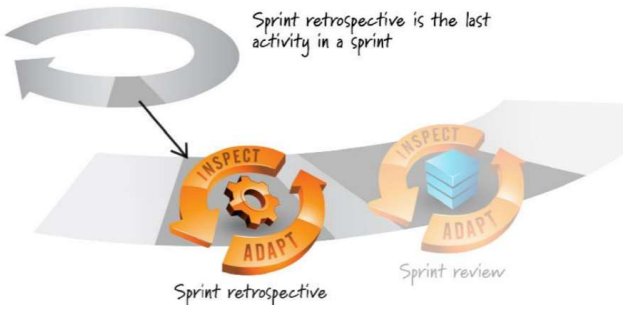


Sprint Retrospective

❑After sprint review and before next sprint planning

❑Check:

- What is working?
- What isn't working?
- What should we change?

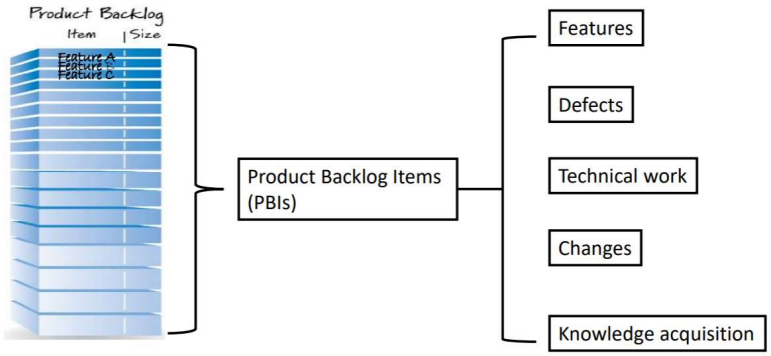


Product Backlog

- ❑A prioritized list of desired product functionality.
- ❑A highly visible artefact at the heart of Scrum framework

Product Backlog Items

- ❑ Product Backlog contains backlog items – **Product Backlog Items** (PBIs)
- ❑ Most PBIs are **features** – items of functionality that will have tangible value to user and customer



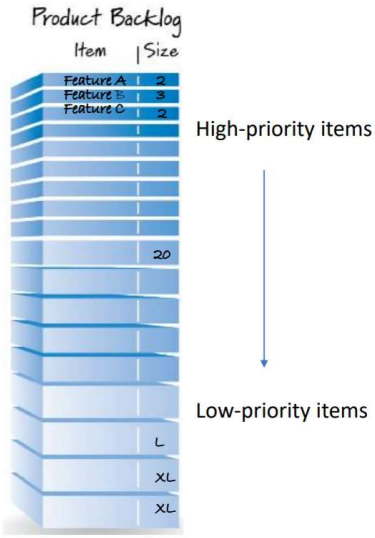
Product Backlog Items

PBI Type	Example
Feature	As a customer service representative I want to create a ticket for a customer support issue so that I can record and manage a customer's request for support.
Change	As a customer service representative I want the default ordering of search results to be by last name instead of ticket number so that it's easier to find a support ticket.
Defect	Fix defect #256 in the defect-tracking system so that special characters in search terms won't make customer searches crash.
Technical improvement	Move to the latest version of the Oracle DBMS.
Knowledge acquisition	Create a prototype or proof of concept of two architectures and run three tests to determine which would be a better approach for our product.

Good Product Backlog Characteristics:

The DEEP Rule…

- Detailed Appropriately
  - Not all items at the same level of detail
  - Near top of backlog: small in size, very detailed – can work on in near-term sprint
  - Larger in size, less derailed at bottom
- Emergent
  - Customer might change their mind
  - Designed to adapt to these occurrences
- Estimated
  - A size estimate of the effort required to develop them
  - Story point or ideal day estimates
- Prioritised
  - Prioritise the near-term items that are designed for the next few sprints



Example

ID	Product Backlog Item	Story Points	Business Value	Priority
1	As a bank customer I want to register to the online banking system so I can see my profile	5	9	9
2	As a registered online banking customer I want to login to online banking system so that I can view my profile	5	9	9
3	As a logged customer I want to make my account transactions	8	9	9
4	As a logged in customer I want to create a new account	8	7	7
5	As a logged in customer I want to change my password	3	4	4

Question

❑What is the difference between the Product Backlog and the Sprint Backlog?

❑**Answer**: The Product Backlog contains everything we might ever work on, while the Sprint Backlog contains just the things we will work on during **one** Sprint.

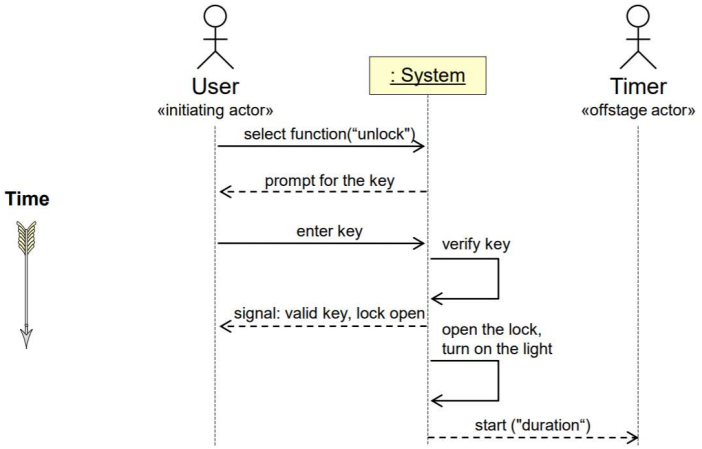
Question

❑Should the team expect to know all the tasks necessary to complete the committed backlog items during the Sprint Planning Meeting?

- **Answer**: No. According to Agile Project Management with Scrum, only 60% of tasks are likely to be identified during the Sprint Planning Meeting.
  - Other tasks will be discovered during Sprint Execution.

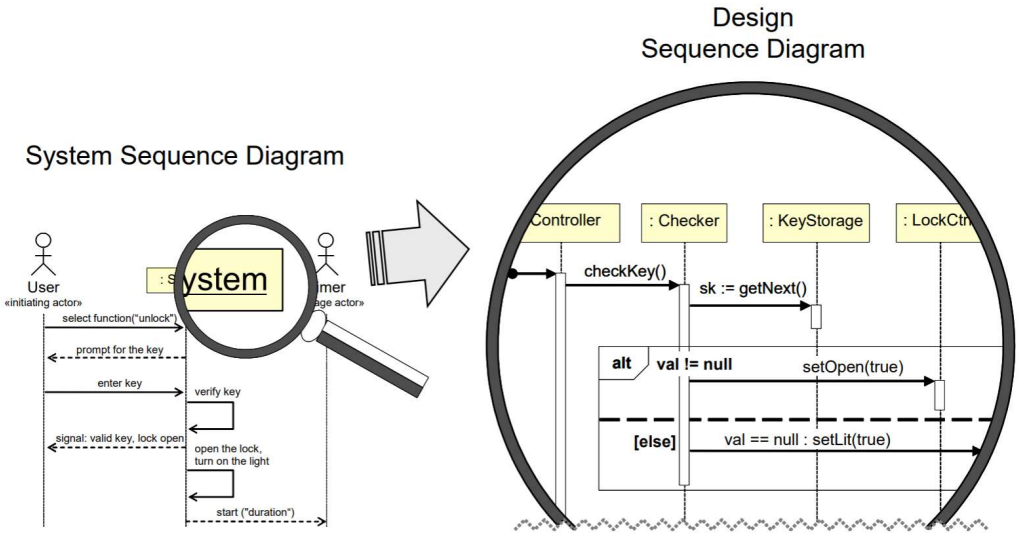
System Sequence Diagrams

We already worked with interaction diagrams: System Sequence Diagrams



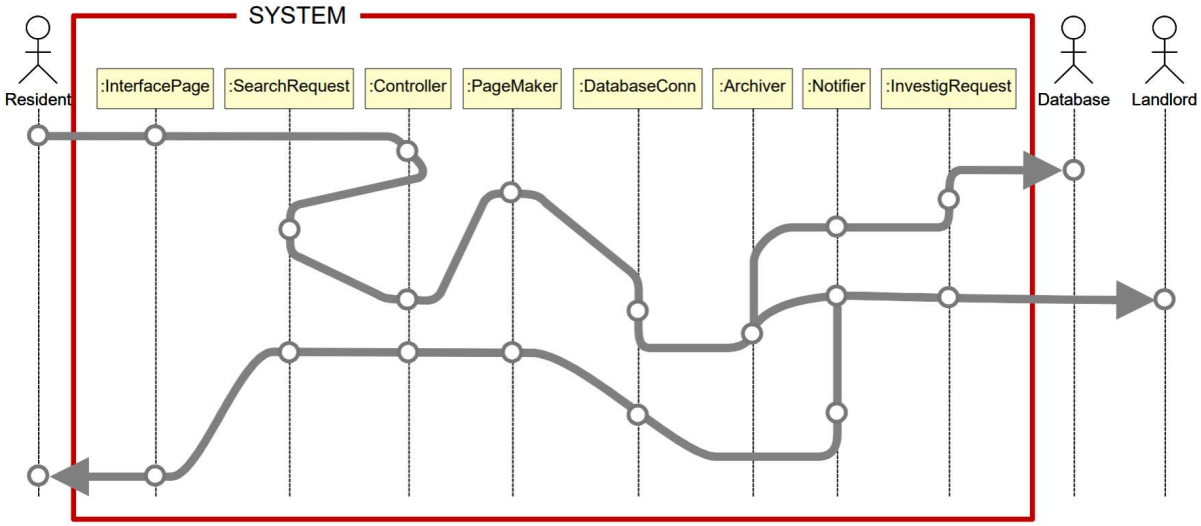
System Sequence Diagrams represent interactions between the **actors**
(system is also an "actor")

Design: Object Interactions



- System Sequence Diagrams represent interactions of **external actors**
- Object Sequence Diagrams represent interactions of **objects inside the system**

Metaphor for Software Design: "Connecting the Dots" within the System Box



We start System Sequence Diagrams (which show only actors and the system as a "black box") to design the internal behavior using conceptual objects from the Domain Model and modify or introduce new objects, as needed to make the system function work.

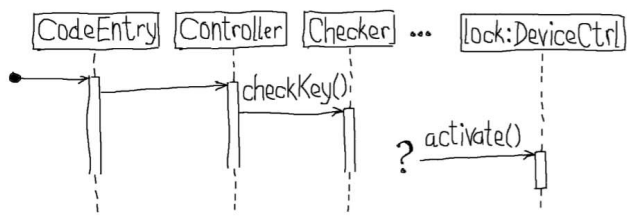
Types of Responsibilities:

- Knowing responsibility: memorizing data or references, such as data values, data collections, or references to other objects, represented as an attribute
- Doing responsibility: performing computations, such as data processing, control of physical devices, etc, represented as a method
- Communicating responsibility: communicating with object's dependencies to delegate work, represented as message sending (method invocation)

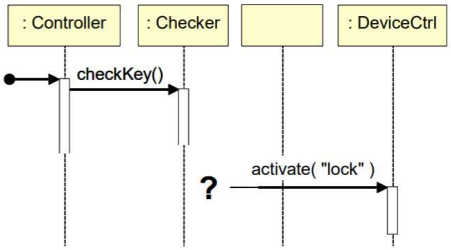
Example

Communicating responsibilities identified for the system function "enter key":

Responsibility Description
Send message to Key Checker to validate the key entered by the user.
Send message to a DeviceCtrl to disarm the lock device.
Send message to a DeviceCtrl to switch the light bulb on.
Send message to PhotoObserver to report whether daylight is sensed.
Send message to a DeviceCtrl to sound the alarm bell.



(a) Hand-drawn during creation



(b) Computer-based UML specification of the design

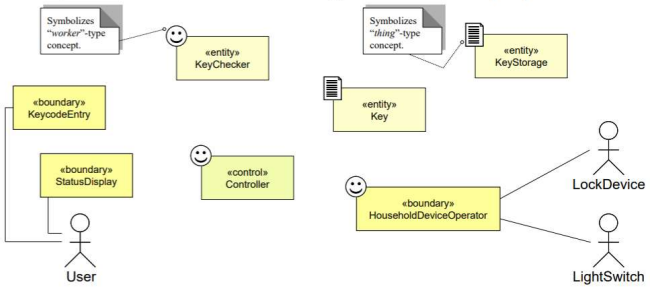
- Two purposes of design diagrams:
  a) Communication tool, during the creative process ➔ use hand drawings and take an image by phone camera (don't waste time on polishing something until you're feel confident that you reached a good design)
  b) Specification tool for documentation ➔ use UML design tools to produce neat and presentable diagrams

How To "Connect the Dots"

Starting Points:

Use Case UC-1: **Unlock** (flow of events):
1. User enters the key code
2. System verifies that the key is valid
3. System signals the key validity
4. System signals:
   (a) to LockDevice to disarm the lock
   (b) to LightSwitch to turn the light on

Domain Model from UC-1 (domain concepts):



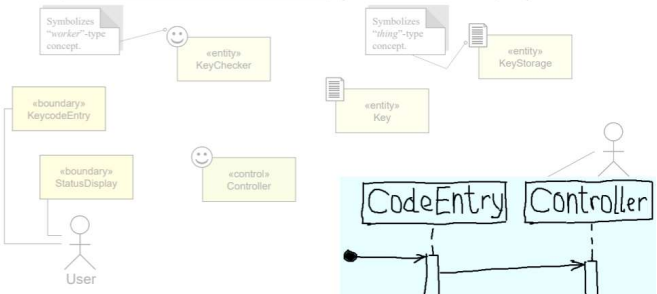
How To “Connect the Dots”

Starting Points:

Use Case UC-1: **Unlock** (flow of events):

1. User enters the key code
2. System verifies that the key is valid
3. System signals the key validity
4. System signals:
   (a) to LockDevice to disarm the lock
   (b) to LightSwitch to turn the light on

Domain Model from UC-1 (domain concepts):



Scenario Walkthrough:

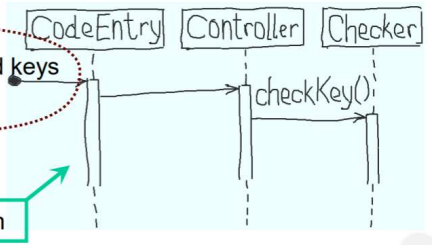
for mapping a Use Case scenario to the Domain Model

Q: who handles this data?

Interface objects and Controller ····· message: checkKey(k)

Q: who performs the verification? Based on what data? ····· return value

Key Checker, based on entered key-code and stored valid keys



Design Sequence Diagram

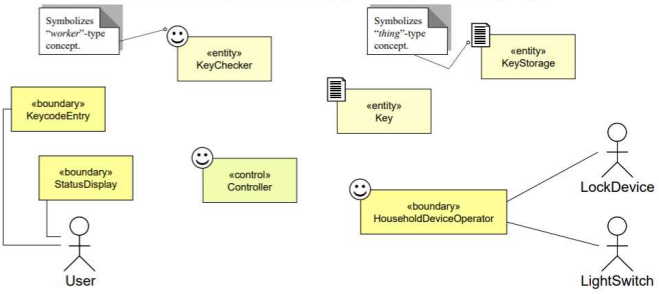
How To “Connect the Dots”

Starting Points:

Use Case UC-1: **Unlock** (flow of events):

1. User enters the key code
2. System verifies that the key is valid
3. System signals the key validity
4. System signals:
   (a) to LockDevice to disarm the lock
   (b) to LightSwitch to turn the light on

Domain Model from UC-1 (domain concepts):



Scenario Walkthrough:

for mapping a Use Case scenario to the Domain Model

Q: who handles this data?

Interface objects and Controller ····· message: checkKey(k)

Q: who performs the verification? Based on what data? ····· return value

Key Checker, based on entered key-code and stored valid keys

message: ???

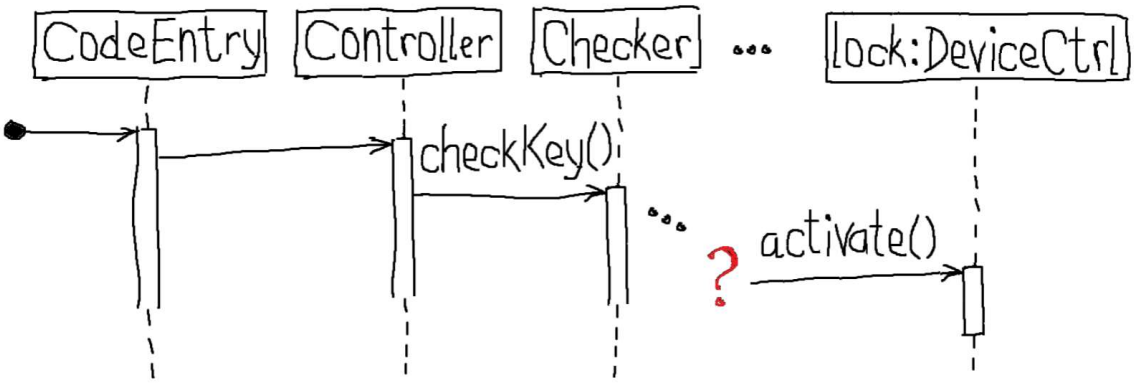
Q: who signals? Based on what data?

Controller and Interface objects, based on key verification; because they are «boundary»

Q: who signals? Based on what data?

Controller or Key checker **???**, based on key verification

Sequence Diagram (in progress)



Q: who performs the verification? Based on what data?

return value

Key Checker, based on entered key-code and stored valid keys

message: ???

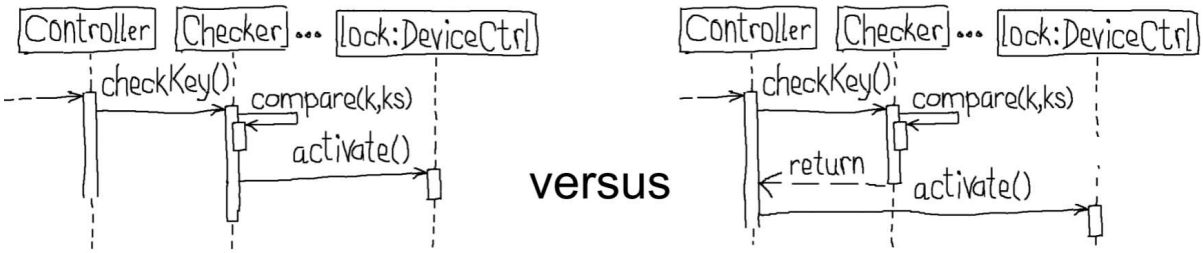
Q: who signals? Based on what data?

Controller and Interface objects, based on key verification; because they are «boundary»

Q: who signals? Based on what data?

Controller or Key checker ???, based on key verification

Alternative Designs:

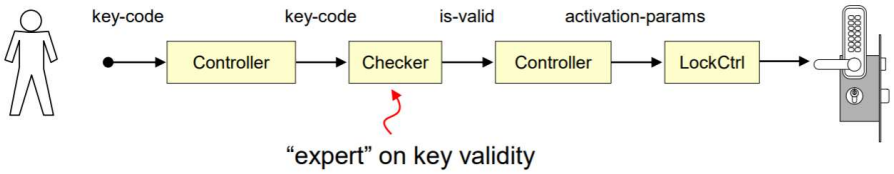


- Which design is better?
- How to evaluate the "goodness" of a design?

How Data Travels

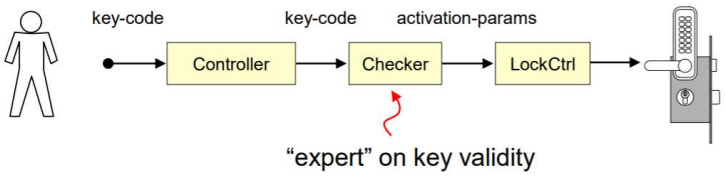
Option A:

"expert" (Key Checker) passes the information (key validity) to another object (Controller) which uses it to perform some work (activate the lock device)



Option B:

"expert" (Key Checker) directly uses the information (key validity) to perform some work (activate the lock device)

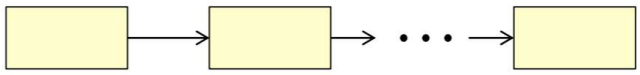


Advantage:
Shorter communication chain

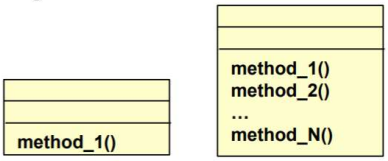
Drawback:
Extra responsibility for Checker

Characteristics of Good Designs

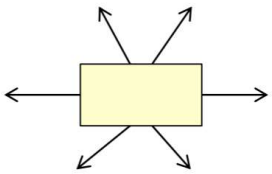
- Short communication chains between the objects



- Balanced workload across the objects



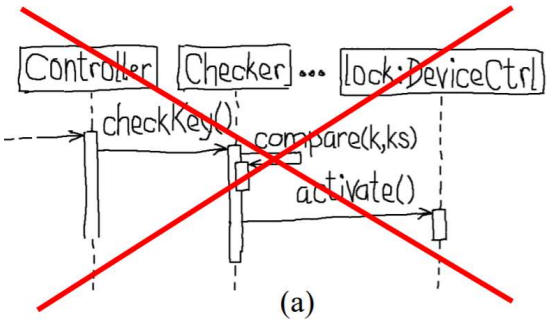
- Low degree of connectivity (associations) among the objects



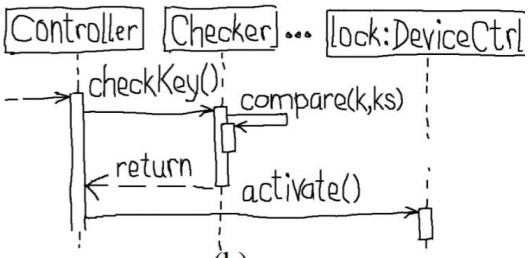
Design Principles:

- Expert doer principle: object that knows should communicate information to all who need it. How to recognise a violation – if a method call passes any parameters
- High cohesion principle: object should not take on too many computation responsibilities. How to recognise a violation – if a class has many loosely or not-related attributes and methods
- Low coupling principle: object should not take on too many communication (delegating) responsibilities. Dependency on other objects (outgoing links) versus object being reused by other objects (incoming links)

Design: Assigning Responsibilities



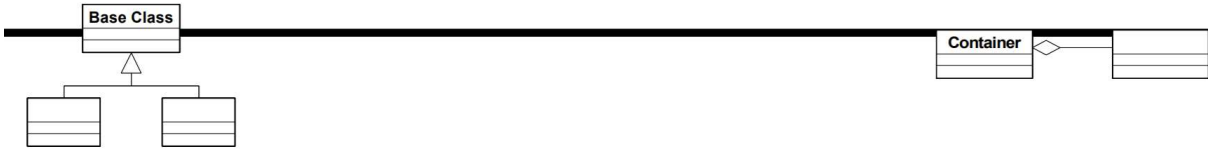
(a)



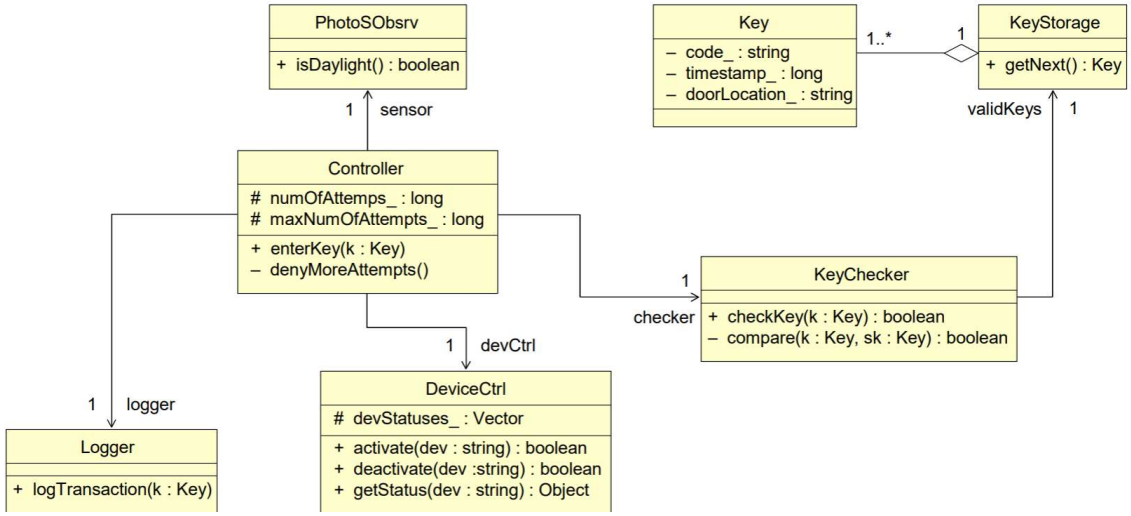
(b)

- Although the Checker is the first to acquire the information about the key validity, we decide to assign the responsibility to activate the DeviceCtrl to the Controller
- This is because the Controller would need to know key-validity information anyway—to inform the user about the outcome of the key validity checking
- In this way, we maintain the Checker focused on its specialty and avoid assigning unrelated responsibilities to it

Class Diagram



Class diagram should be derived by looking-up the sequence diagrams



Traceability Matrix

Mapping: Domain model to Class diagram

Domain Concepts	Controller-SS1	Key	KeyStorage	KeyChecker	DeviceCtrl	PhotoSObsrv	Logger	Controller-SS2	SearchRequest	«html» interfacePage	PageMaker	DatabaseConnection
	Software Classes											
Controller-SS1	X											
StatusDisplay												
Key		X										
KeyStorage			X									
KeyChecker				X								
HouseholdDeviceOperator					X							
IlluminationDetector						X						
Controller-SS2								X				
SearchRequest									X			
InterfacePage										X		
PageMaker											X	
Archiver												
DatabaseConnection												X
Notifier												
InvestigationRequest												

Domain Analysis and Modelling

- Domain analysis and modelling identifies the system elements needed to solve the problem (i.e. meet the requirements)
- The goal of domain modelling is to understand how the system will work
    - Requirements analysis determined how users will interact with the system (external behaviour)
    - Domain modelling determines how elements of the system interact (internal behaviour) to produce the external behaviour
- We do domain modelling based on sources…
    - Knowledge of how the system is supposed to behave (from requirements analysis e.g. use cases)
    - Studying the work domain (or problem domain)
    - Knowledge base of software designs
    - Developers' past experience with software design

Typical Problems with Domain Models:

- Unaware that requirements are not simply a wish list – ignoring real-world constraints and problems
    - Physical I/O devices, networks, etc, are prone to failure
    - Economic, legal, cultural, etc, constraints
- Unaware of dependencies between requirements (or use cases)
- Unaware of incompatible data across concepts/modules
    - Different concepts/modules may receive or output different data formats

Why Domain Modelling:

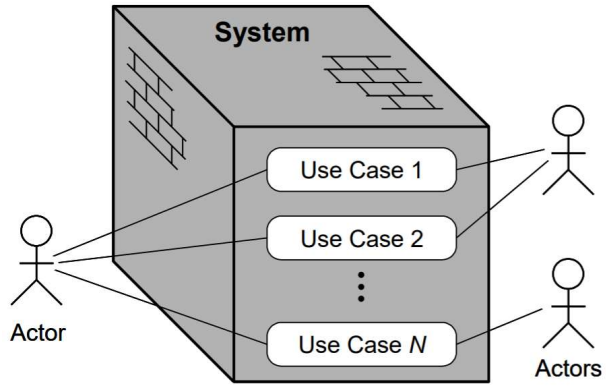
- To achieve N different things, we need N different tasks – like ensuring that every row in the traceability matrix crosses at least one column
- The problem for the beginners is that they don't know what needs to be achieved
- Experienced developers will at least know or be able to guess some things that are common to many problems e.g. generic issues for networks or I/O devices
- The only way to know what is needed is to study the problem domain and get help from domain experts

Use Cases vs. Domain Model

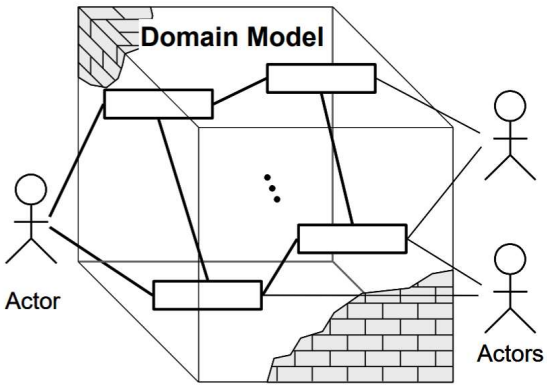
In **use case analysis**, we consider the system as a "**black box**"

In **domain analysis**, we consider the system as a "**transparent box**"

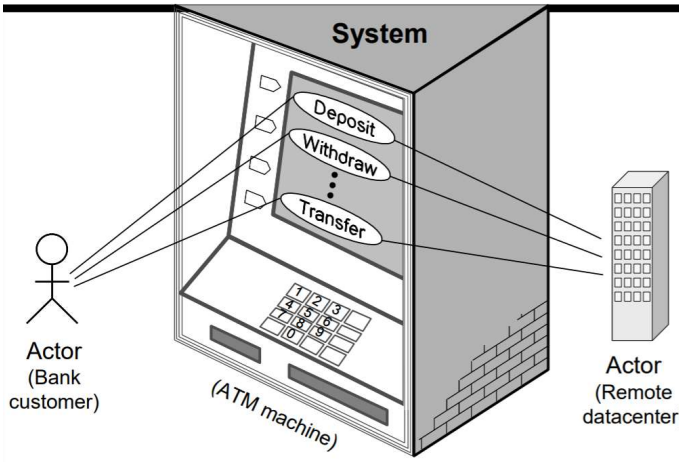
(a)



(b)

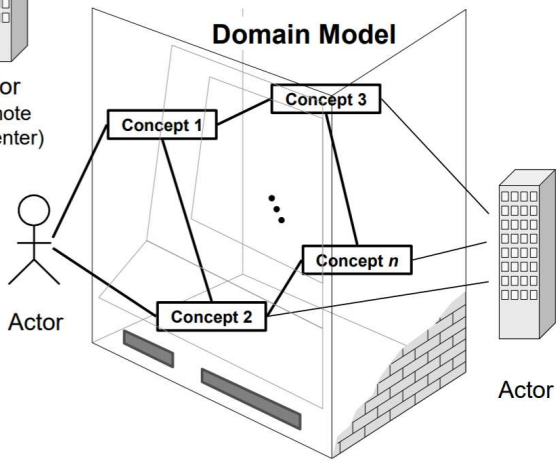


What is Domain Modeling
Example: ATM Machine



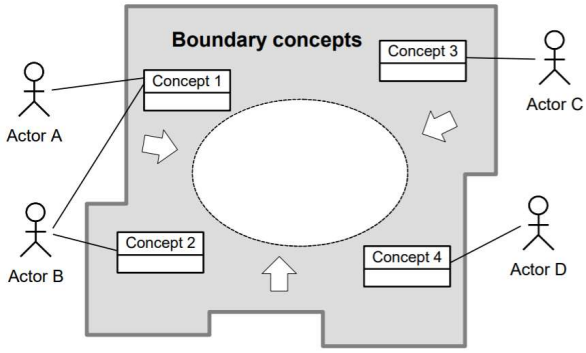
(a) **Use Case Model**

(b) **Domain Modelling**

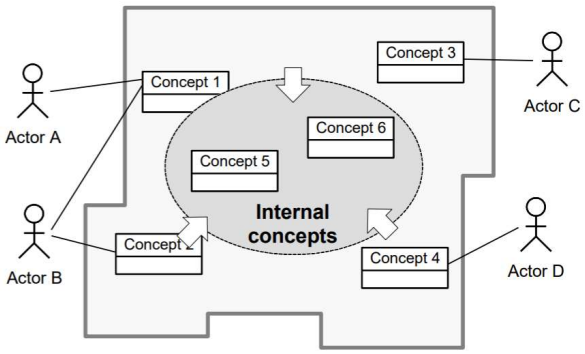


Building Domain Model from Use Cases

Step 1: Identifying the boundary concepts



Step 2: Identifying the internal concepts



Use Case 1: Unlock

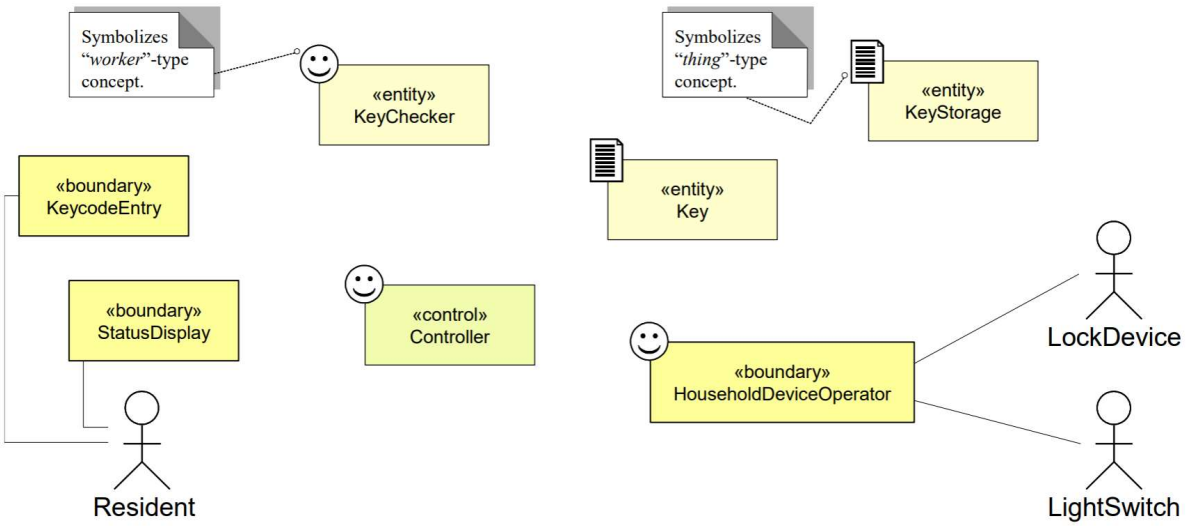
Use Case UC-1:	**Unlock**
Related Requirements:	REQ1, REQ3, REQ4, and REQ5 stated in Table 2-1
Initiating Actor:	Any of: Tenant, Landlord
Actor's Goal:	To disarm the lock and enter, and get space lighted up automatically.
Participating Actors:	LockDevice, LightSwitch, Timer
Preconditions:	• The set of valid keys stored in the system database is non-empty. • The system displays the menu of available functions; at the door keypad the menu choices are "Lock" and "Unlock."
Postconditions:	The auto-lock timer has started countdown from autoLockInterval.
Flow of Events for Main Success Scenario:	
→	1. Tenant/Landlord arrives at the door and selects the menu item "Unlock"
	2. include::*AuthenticateUser* (UC-7)
←	3. System (a) signals to the Tenant/Landlord the lock status, e.g., "disarmed," (b) signals to LockDevice to disarm the lock, and (c) signals to LightSwitch to turn the light on
←	4. System signals to the Timer to start the auto-lock timer countdown
→	5. Tenant/Landlord opens the door, enters the home [and shuts the door and locks]

Extracting the Responsibilities

Responsibility Description	Type	Concept Name
Coordinate actions of all concepts associated with a use case, a logical grouping of use cases, or the entire system and delegate the work to other concepts.	D	Controller
Container for user's authentication data, such as pass-code, timestamp, door identification, etc.	K	Key
Verify whether or not the key-code entered by the user is valid.	D	KeyChecker
Container for the collection of valid keys associated with doors and users.	K	KeyStorage
Operate the lock device to armed/disarmed positions.	D	LockOperator
Operate the light switch to turn the light on/off.	D	LightOperator
Operate the alarm bell to signal possible break-ins.	D	AlarmOperator
Block the input to deny more attempts if too many unsuccessful attempts.	D	Controller
Log all interactions with the system in persistent storage.	D	Logger

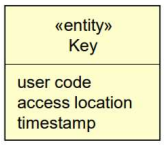
D = Doing; K=Knowing

Domain Model (1)

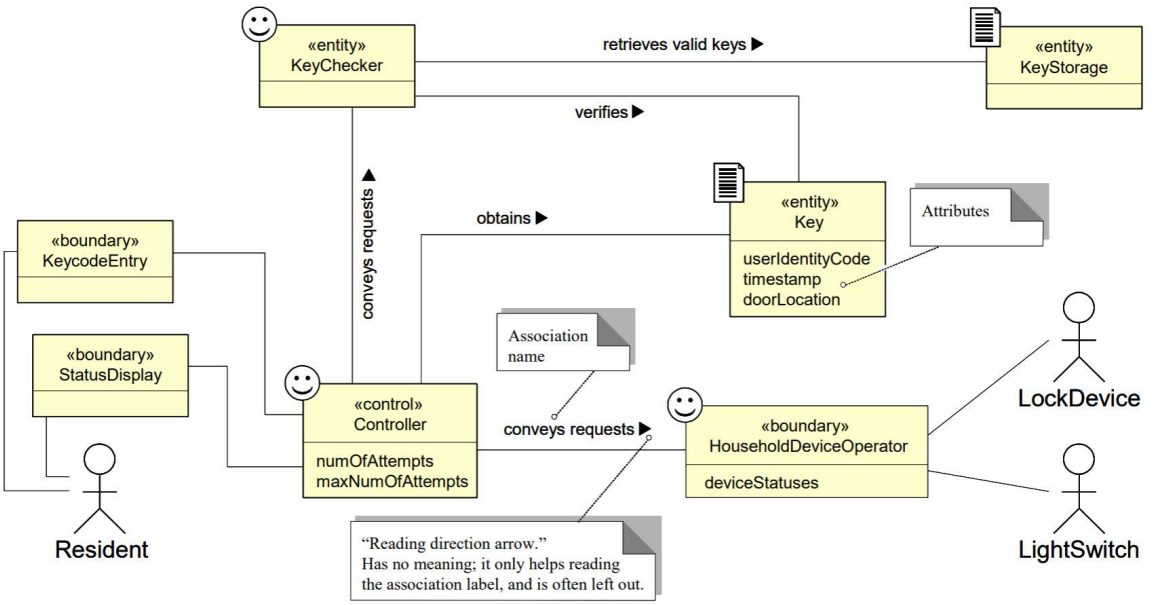


Domain concepts for subsystem #1 of safe home access

Domain Model (2)

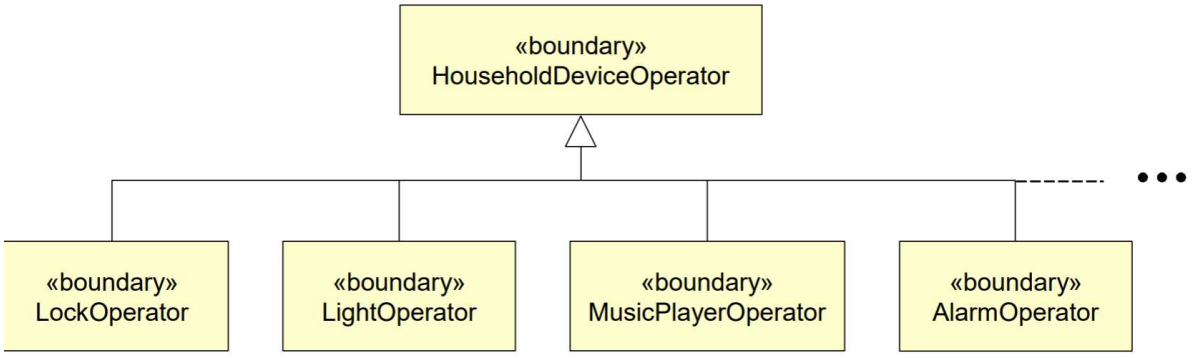


- Domain model for UC-1: Unlock



Degrees of Domain Model Refinement

- Simplest case: all household devices are conceptually the same—just an on/off switch to activate or deactivate
- If each device will provide additional functionality, use different conceptual objects
- The correct approach depends on the requirements



Traceability Matrix (1)

Mapping: System requirements to Use cases

REQ1: Keep door locked and auto-lock
REQ2: Lock when "LOCK" pressed
REQ3: Unlock when valid key provided
REQ4: Allow mistakes but prevent dictionary attacks
REQ5: Maintain a history log
REQ6: Adding/removing users at runtime
REQ7: Configuring the device activation preferences
REQ8: Inspecting the access history
REQ9: Filing inquiries

UC1: Unlock
UC2: Lock
UC3: AddUser
UC4: RemoveUser
UC5: InspectAccessHistory
UC6: SetDevicePrefs
UC7: AuthenticateUser
UC8: Login

Req't	PW	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8
REQ1	5	X	X						
REQ2	2		X						
REQ3	5	X						X	
REQ4	4	X						X	
REQ5	2	X	X						
REQ6	1			X	X				X
REQ7	2						X		X
REQ8	1					X			X
REQ9	1					X			X
Max PW		5	2	2	2	1	5	2	1
Total PW		15	3	2	2	3	9	2	3

Traceability Matrix (2)

Mapping: Use cases to Domain model

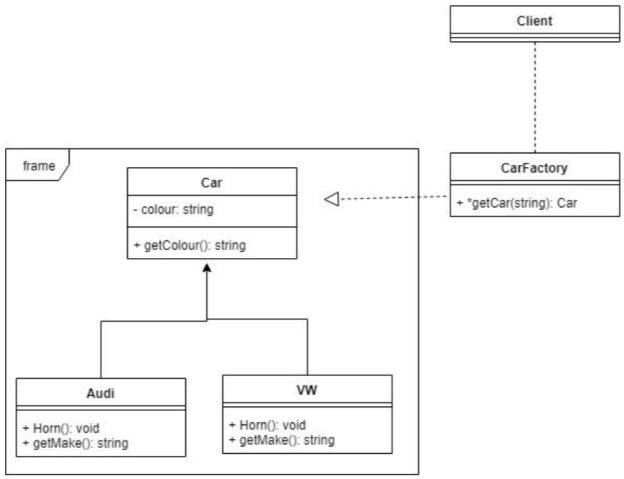
UC1: Unlock
UC2: Lock
UC3: AddUser
UC4: RemoveUser
UC5: InspectAccessHistory
UC6: SetDevicePrefs
UC7: AuthenticateUser
UC8: Login

		Domain Concepts														
Use Case	PW	Controller-SS1	StatusDisplay	KeycodeEntry	Key	KeyStorage	KeyChecker	HouseholdDeviceOperator	Controller-SS2	SearchRequest	InterfacePage	PageMaker	Archiver	DatabaseConnection	Notifier	InvestigationRequest
UC1	15	X	X	X	X			X								
UC2	3	X	X					X								
UC3	2								X		X	X		X		
UC4	2								X		X	X		X		
UC5	3								X	X	X	X	X	X	X	X
UC6	9								X		X	X		X		
UC7	2	X	X		X	X	X									
UC8	3								X		X	X		X		

Design Patterns

Factory Pattern:

- Problem: we have a number of classes to choose from to instantiate, but we don't know which one we need
- The factory pattern shows how we can create objects, without specifying the exact class to create it from
- Creates objects without specifying the exact class of object that will be created
- The factory design pattern defines an interface for creating an object, but lets the classes that implement the interface decide which class to instantiate
- The factory method lets a class defer instantiation to subclasses
- Returns an instance of one of several possible classes…
    - Depending on data we provide
    - Classes returned have a common parent class
    - Each class performs differently



- The programming principle that the factory pattern gives a solution to is polymorphism
- A factory can return an instance of several possible classes that have a common parent class – determined by our need
- The parent class can be an abstracted class, or an interface
- The calling method tells the factory what it wants
- It is the factory that makes the decision on which subclass to return – an instance is created, and returned

Factory Pattern variants:

- Static factory pattern:
    - Creates a method which is responsible for creating objects of a particular type
    - If the method is 'static', then it is a static factory pattern
- Factory method pattern
    - The method creating the objects becomes 'abstract'
    - Causes the class to be abstract and must be used as a base class

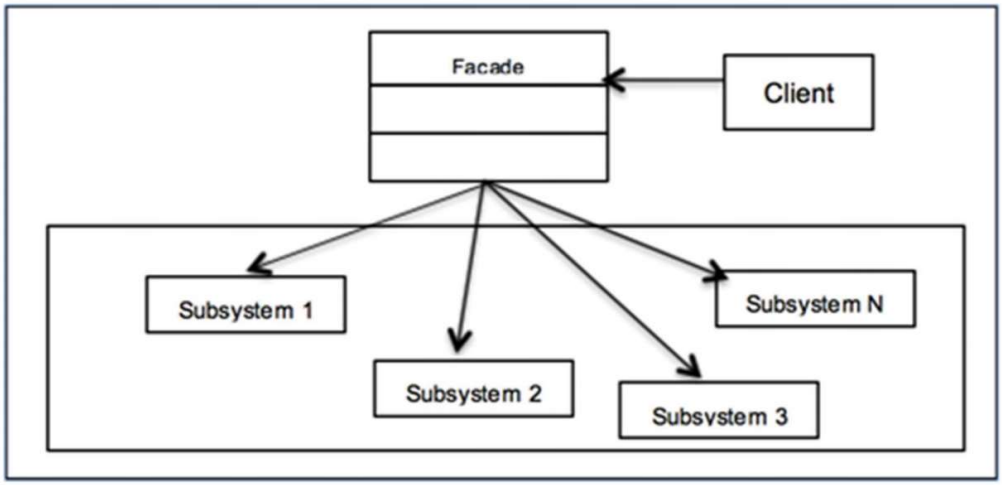
    - The method needs to be implemented in a 'concrete client'
    - E.g. client doesn't know which classes are needed at runtime
- Abstract factory pattern:
    - Group together individual factory methods
    - E.g. client creates a library of products without exposing implementation details

Structural Patterns:

- Structural patterns modify the structure or design of the system
- They are responsible for building simple and efficient class hierarchies and relations between different classes

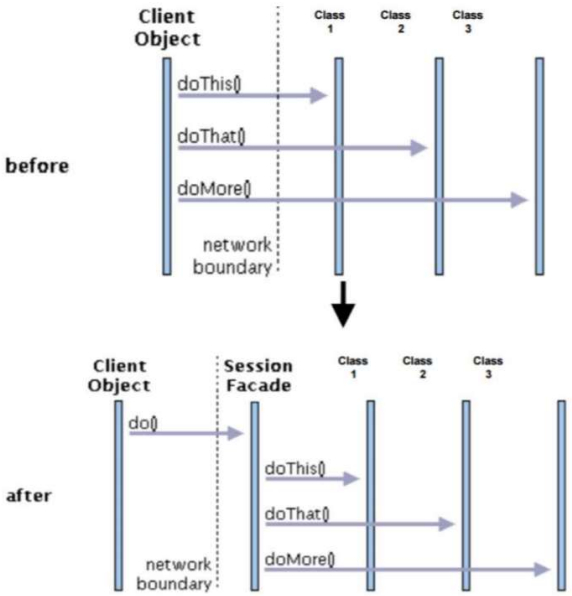
Façade Pattern:

- Façade is a structural design pattern that lets you provide a simplified interface to a complex system of classes, library, or framework
- The problem…
    - Imagine some code that has to work with a large set of objects of some complex library or framework
    - You have to manually initialise all these objects, keep track of the dependencies, correct order, etc.
    - The business logic of your classes becomes tightly coupled to the implementation details of third-party library. Such code is pretty hard to comprehend and maintain
- The solution…
    - The façade is a class that provides a simple interface to a complex subsystem containing dozens of classes
    - The façade may have limited functionality in comparison to working with the subsystem directly
    - However, it includes only those features that clients really care about



- Façade: the main responsibility of a façade is to wrap up a complex group of subsystems so that it can provide a simple look to the outside world

- System: this represents a set of varied subsystems that make the whole system compound and difficult to view or work with
- Client: the client interacts with the façade so that it can easily communicate with the subsystem and get the work completed – it doesn't have to bother about the complex nature of the system
- For example, you (client) contact IT Help Desk (Façade). He/she decides which IT help function (sub-systems) you could be directed to



- Advantages:
    - Simplified interface to a complex system
    - Modify the inner workings of the subsystem without the client knowing – as long as the façade interface to the client remains the same

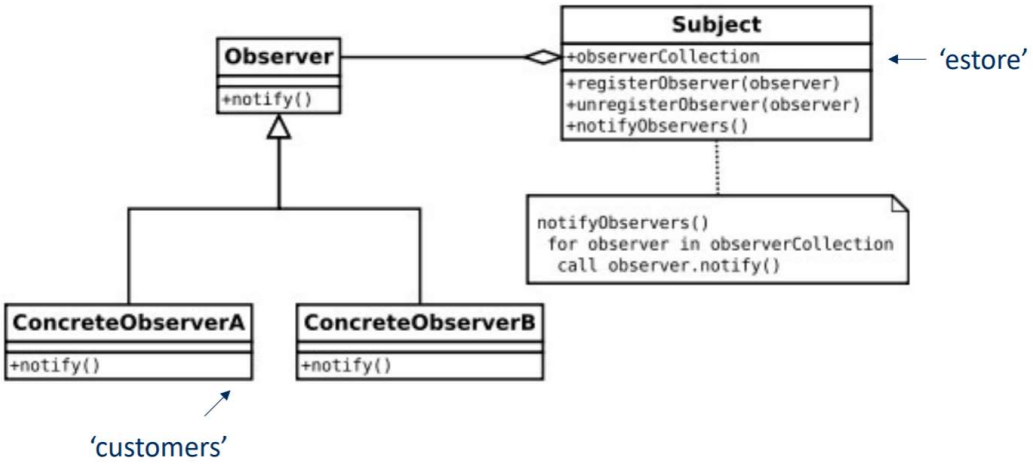
Behavioural Patterns:

- Behavioural patterns are responsible for the efficient and safe communication of behaviours between the program's objects

Observer Pattern:

- Lets you define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically e.g. magazine subscriptions
- The problem…
    - Imagine there are two objects – customer and estore
    - Customers could visit the estore each day to check the availability of a product
    - Many of these visits would be for nothing
    - The estore could send lots of messages to all its customers when the product is available
    - Either the customer wastes resources, or the estore does
- The solution…

- The customer ‘subscribes’ to the estore to get product updates
- The estore then has a list of customer subscribers for certain product updates
- Only those subscribing customers are informed (updated) when the product details/availability changes
- The customers have no access to the data in the estore, they are dependent on the ‘publisher’ to provide them data



Summary:

- Factory pattern:
  - When we want to create different objects
  - Object creation code is not needed by the client
- Façade pattern:
  - When we want to hide the inner workings (of an API for example)
  - Façade interface masks complex behaviour from a client
- Observer pattern:
  - Subscribing clients are updated when a system changes state

Software Testing

Bugs:

A software bug occurs when at least one of these is true…

- The software does not do something that the specification says it should do
- The software does something that the specification says it should not do
- The software does something that the specification does not mention
- The software does not do something that the product specification does not mention, but should

Testing Goals:

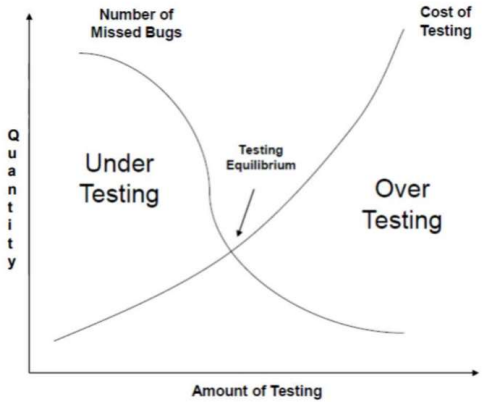
- Validation Testing: to demonstrate to the developers and the customer that the software meets its requirements
    - Success: system operates as expected
    - At least one test for every requirement
    - Tests each feature as well as combinations of features
- Defect Testing: to discover situations in which the behaviour of the software is incorrect, undesirable or does not conform to its specification
    - Success: system performs incorrectly
    - Discover undesirable system behaviour such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption

Software testing axioms (statements taken to be true):

- It is impossible to test a program completely
- Testing cannot show to absence of bugs
- Not all bugs found will be fixed
- It is difficult to say when a bug is indeed a bug
- Specifications are never final
- Software testers are not the most popular members of a project

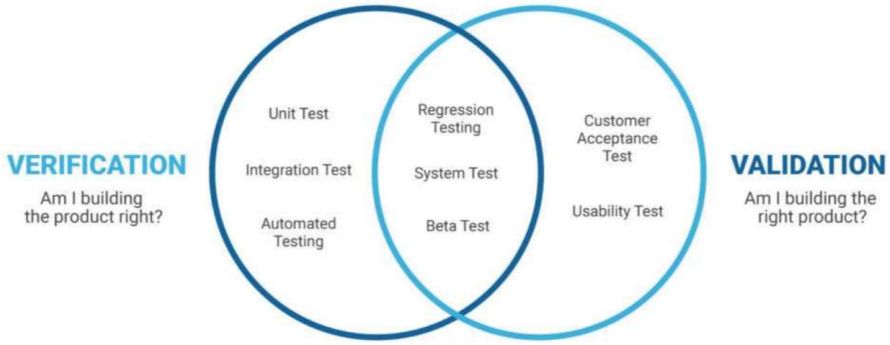
Is it possible to test a program completely?

The only way to be absolutely sure software works is to run it against all possible inputs and observe all of its outputs



Verifications vs Validation:

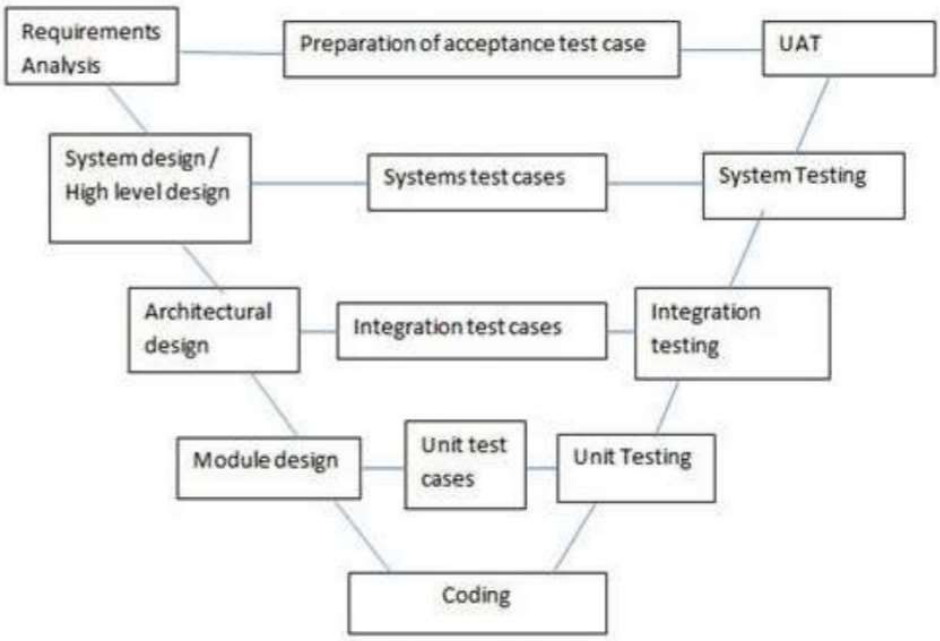
- Verification: are we building the product right? Check against specification
- Validation: are we building the right product? Check against user requirements

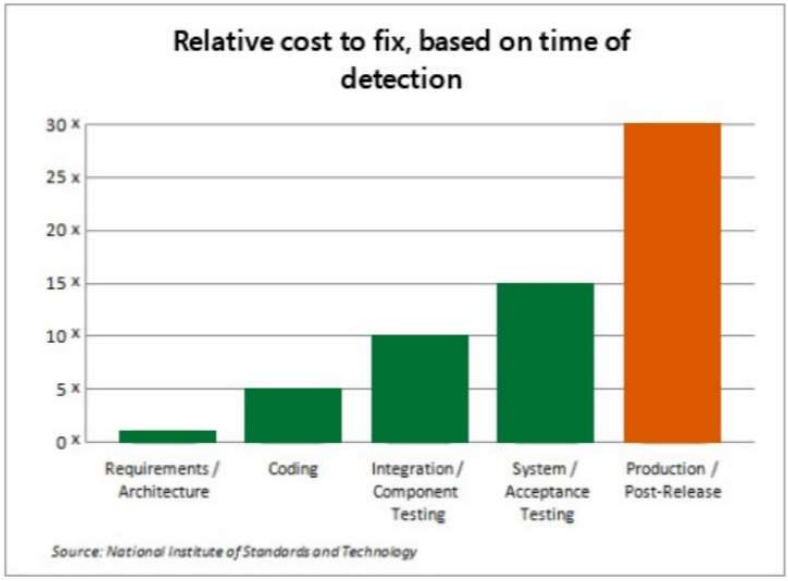


Verification & Validation Confidence:

- Aim of V&V is to show a system is 'fit for purpose'
- 'fit for purpose' means different things in different contexts
- Depends on the purpose of the software, the expectations of the user and the marketing context
    - Is it a critical software system?
    - Does the user have high/low expectations for the software?
    - Does the marketing context mean there is pressure to release?

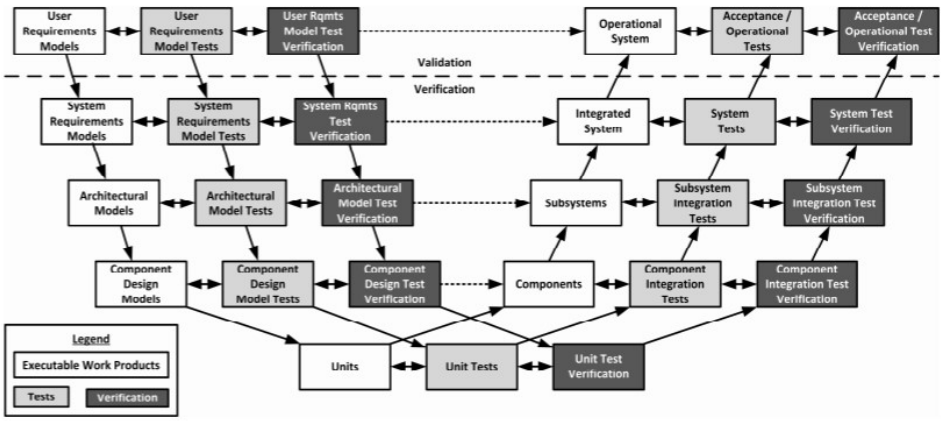
The V-Model:





The costs of fixing a defect increases across the development life cycle. The earlier in life cycle a defect is detected, the cheaper it is to fix it.

- The V-model of testing was developed where for every phase in the software development life cycle, there is a corresponding testing phase
- The v-model is an extension of 'plan-based' development methodologies, such as the waterfall model
    - Unlike the waterfall model, in the V-model, there is a corresponding testing phase for each development phase
    - At each stage of the software development life cycle, test cases are produced to be applied to the software
- Multiple layers can be added to the v-model
- When model-based design is used, these model designs need to be verified
- Verifying the tests



V-model vs Agile Development:

- Very difficult to apply the v-model to an agile methodology

- However, some principles are common between the two, e.g.…
    - Kill bugs at their source – agile best practise is to address the bugs in the iteration they are discovered, 'phase containment' in v-model removes defects at their point of introduction
    - Acceptance criteria – user stories (agile) ad requirements (v-model) both include acceptance criteria i.e. what will be tested, and how can they be successful

Types of Testing:

Unit Testing:

- The process of testing individual components in isolation
- Test driven developments creates tests before the code
- 'units' could be individual functions, classes, or components with well-defined interfaces
- Purpose is to validate that each unit of software performs as it was designed
- 'linter' – type of tool that assess code form bugs, errors, inconsistencies, etc. before it is run/compiled (static code checks) – useful for interpreted languages e.g. Python
- Catch is a test framework for C/C++ etc.
- Complete test coverage of a class (for instance) involves…
    - Testing all operations associated with an object
    - Setting and interrogating all object attributes
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localised – parent/child class operations may need to be considered
- Test normal operation of the object/function/component
- Test abnormal input to see if it is handled correctly

Equivalence Partitioning/Boundary Value Analysis:

- EP: a 'black box' testing technique
- Can be applied to all levels of testing
- Reduce range of inputs to 'ranges' or 'classes'
- One test case is chosen from each class
- Reduces total number of test cases to be more manageable
- E.g. software accepts inputs between 1-1000 so divide into 3 classes, above below, within
- BVA: many errors occur at boundaries or extremes of input values

Integration Testing:

- Test to determine whether independently developed 'units' or components work together when connected

System testing:

- Complete and integrated software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements

Acceptance testing:

- Process of verifying that a created solution/software works for the user
- Does the software conform to the business requirements?

Performance/Stress Testing:

- Part of release testing may involve testing the emergent properties of a system, such as performance and reliability
- Performance tests usually inolve planning a series of tests where the load steadily increases until the system performance becomes unacceptable
    - Speed – how it changes with load
    - Scalability – maximum user load
    - Stability – how it changes with load
- Stress testing is a form of testing where the system is deliberately overloaded to test its failure behaviour
- Also: spike testing, endurance testing, etc.

Summary:

- Test Driven Development (TDD) creates tests before the code is written
- Different tests address different stages of the Software Development Life Cycle (SDLC)
- Difficult to apply the V-model of testing to Agile methodologies
- Use ‘linter’s to inspect code before it is compiled etc.
- Use automated test frameworks, such as Catch

Software Reliability

"The probability of failure-free software operation for a specified period of time in a specified environment"

Software will not change over time unless intentionally changed or upgraded…

- Is the specification/requirements correct?
- Is the software development methodology rigorous?
- Has it been tested adequately?

Probability of Failure on Demand (POFOD):

- Likelihood a system will fail when a request for service is made
- E.g. a POFOD of 0.0001 means 1 in 10,000 requests may result in failure
- Relevant measure for safety critical systems

Rate of Occurrence of Failure (ROCOF):

- Frequency of occurrence of failures
- ROCOF of 0.005 means 5 failures are likely to happen in each 1000 time units
- Relevant measure for banking/financial systems

Mean Time Between Failures (MTBF):

- Measure of time between recoverable failures over time lifetime of the product
- A MTBF of 100 means that failures may occur between every 100 time units
- Relevant measure for 'long duration transaction' systems such as database operations, word processors, etc.
- MTBF should be longer than the typical transaction length

Mean Time to Failure (MTTF):

- Measure of time to fail (non-recoverable systems)
- E.g. lightbulbs etc.

Availability (AVAIL):

- Measure of how long a system is available for use
- An availability of 0.995 means a system will likely be available for 995 out of 1000 time units
- Relevant measure for continuous systems such as communication/broadband

Software Failure Consequences:

- Not just the number of system failures that matter, the consequences are more important
- E.g. failure to load a webpage, and failure of a 'break by wire' system – both are software failures but can have very different consequences

- Amazon example scenario…
    - If Amazon MTBF (mean time between failures) was 3 years, but MTTR (mean time to recover) was 1 hour, then customers would notice
    - However, if MTBF was 1 day and MTTR was less than 1 second, customers would not notice
    - Backup systems/redundancy try to reduce the MTTR

Reliability Specifications:

- Reliability is dependent on MTTR, MTBF, AVAIL, failure consequences for example
- For reliability specifications…
    - Quantitative statements of the reliability requirement
    - Usage environment description
    - What constitutes a failure?
- Example:
    - E.g. MTBF of a Printer should be greater than 2000 hours, based on…
    - 8 hours per day, 168 hours per month usage
    - 40 chars per second average, 3 million per month
    - MTBF = total operating hours / total number of failures
    - Failure = inability of printer to perform its function (within operation limits, excluding operator error etc.)