COMP 1531
Software Engineering Fundamentals

Summary of Lectures 1-3

Aarthi Natarajan

What is software?

- A ***software*** is a program or sequence of instructions that tells the computer what tasks it needs to perform and how to perform them
- Software can be distinguished into:

  - ***application software*** such as a database, spreadsheet or word-processing program, a web browser, a console game etc.

  - ***system software*** that deals with operating the computer or devices connected to the computer (e.g., operating system such as Microsoft Windows, Mac OS, Unix or device drivers) or utility software (e.g., anti-virus programs)
- Software is everywhere and the economies of ALL developed nations are dependent on software
- A ***software product*** is often used to refer to a collective set of entities that includes software program, documentation, data…

What is Software Engineering?

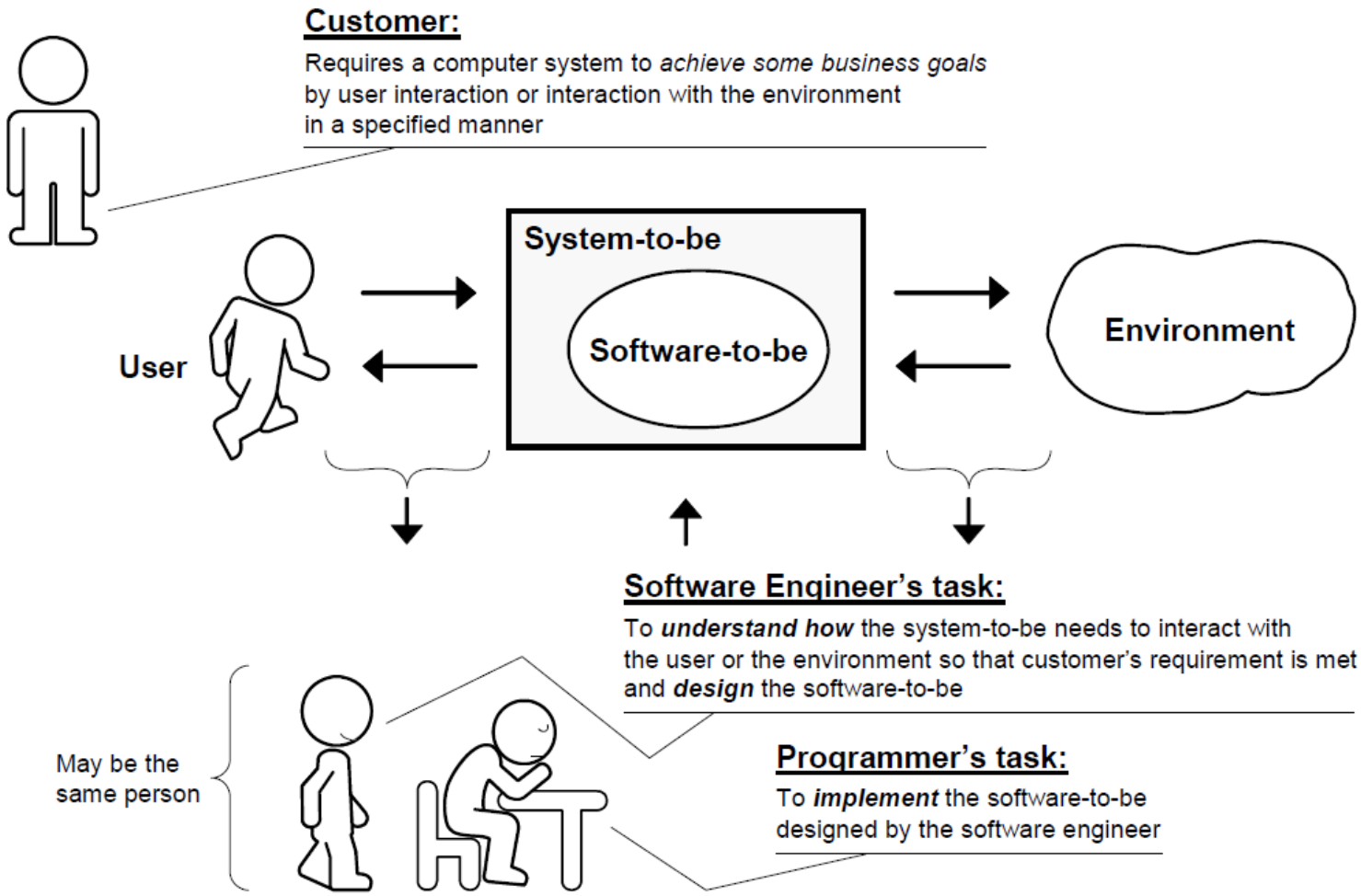
- "Software Engineering" is a discipline that enables customers achieve business goals through *designing* and *developing* software-based systems to solve their business problems e.g., develop a patient-record software in a doctor's surgery, a software to manage inventory and involves:
  - Understanding of the business problem definition
  - Creative formulation of ideas to solve the problem based on this understanding and designing the "blueprint" or architecture of the solution
  - Implementing the "blue-print"
- Software engineering requires great emphasis on *methodology* or the *method* for managing this design and development process in addition to great skills with tools and techniques.

Software Engineering is not Programming

- Software engineering is often *confused* for programming
- A software engineer's focus is on *understanding* the interaction between the system-to-be and its users and the environment, and *designing* the software-to-be based on this understanding
- A programmer's focus is on the program code and ensuring that the code *faithfully* implements the design

Role of a software engineer

- A software engineer is willing to understand the problem-domain and promises to ***deliver*** value to the customer



Software Engineering Life-Cycle

- We described software engineering as a complex, organised process with a great emphasis on *methodology*. This organised process can be broken into the following phases:
  - Analysis and Specification
  - Design
  - Implementation
  - Testing
  - Release & Maintenance
- Each of the above phases can be accompanied by an artifact or deliverable to be achieved at the completion of this phase
- The life-cycle usually comprises peripheral activities such as feasibility studies, software maintenance, software configuration management etc.

Software Engineering Life-Cycle

1. Analysis and Specification:
   - A process of knowledge-discovery about the "system-to-be" and list of features, where software engineers need to:
     - understand the problem definition (delimit its scope, elaborate the system's services (*behavioural characteristics)*
     - abstract the problem to define a domain model (*structural characteristics*)
   - Comprises both functional (inputs and outputs) and non-functional requirements (performance, security, quality, maintainability, extensibility)
   - Popular techniques include use-case modelling, user-stories…
2. Design:
   - A problem-solving activity that involves a "Creative process of searching **how to implement all of the customer's requirements"** and generating software engineering blue-prints

Software Engineering Life-Cycle

3. Implementation:
   - The process of encoding the design in a programming language to deliver a software product
4. Testing:
   - A process of verification that our system works correctly and realises the goals
   - Testing process encompasses unit tests (individual components are tested), integration tests (the whole system is testing), user acceptance tests (the system achieves the customer requirements)
5. Operation & Maintenance:
   - Running the system; Fixing defects, adding new funtionality

Software Engineering Life-Cycle

However, software development is unlike any other product development in these aspects:

- software is *intangible* and hard to visualize.
- software is probably the most *complex* artifact—a large software product consists of so many bits and pieces as well as their relationships
- software is probably the most *flexible* artifact—it can be easily and radically modified at any stage of the development process, so it can quickly respond to changes in customer requirements Hence, a linear order of understanding the problem, designing a solution, implementing and deploying the solution, does not produce best results.

Incremental and Iterative Methods

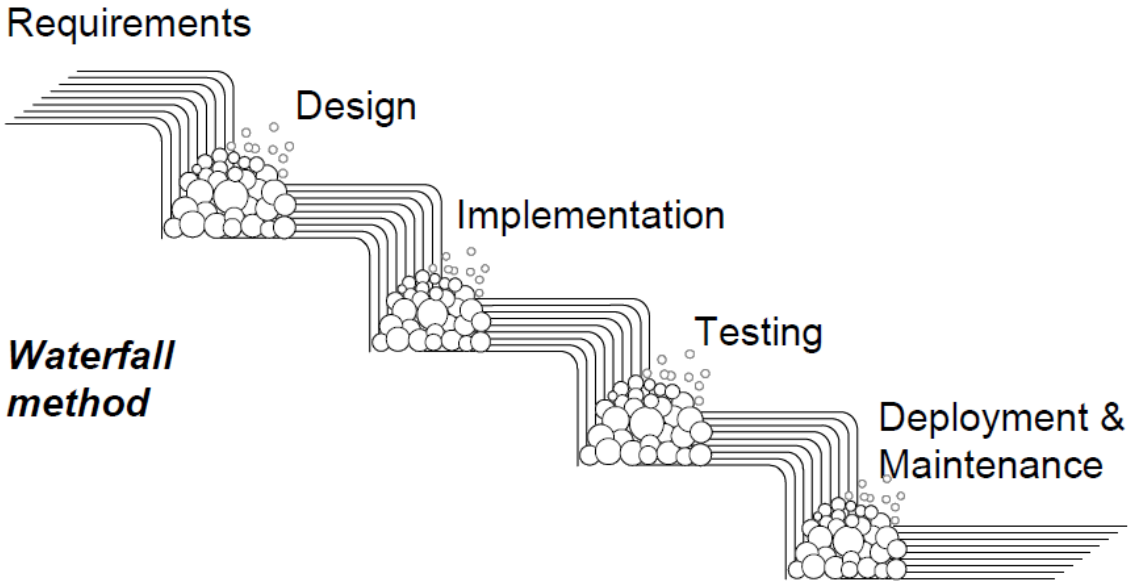
- These insights led to adopting ***incremental*** and ***iterative*** development methods, which are characterized by:
  1. Break the big problem down into smaller pieces (increments) and prioritize them.
  2. In each iteration progress through the development in more depth.
  3. Seek the customer feedback and change course based on improved understanding.
- An incremental and iterative process
  - seeks to get to a working instance as soon as possible.
  - progressively deepen the understanding or "visualization" of the target product, by both advancing and retracting to earlier activities to rediscover more of its features.

Software Development Methodologies

- A **software development method** lays out a prescriptive process by mandating a *sequence of development tasks*. Can be broken into:
- *Elaborate processes* with rigid, plan-driven, documentation heavy methodologies
  - Waterfall: Unidirectional, finish this step before moving to the next
- *Iterative & Incremental processes* which develop increments of functionality and repeat in a feedback loop
  - Rational Unified Process [Jacobson et al., 1999]
  - Agile methods (e.g., SCRUM, XP):
    - Methods that are more aggressive in terms of short iterations
    - Heavy customer involvement, user feedback essential; feedback loops on several levels of granularity; customer is continuously asked to prioritize the remaining work items and provide feedback about the delivered increments of software.

Waterfall Model (1970's)

- Traditional, linear, sequential life cycle model also known as *plan-driven development model* with **detailed planning**
  - Detailed planning – problem is identified, documented and designed
  - Implementation tasks identified, scoped and scheduled
  - Development cycle followed by testing cycle
- Simple to understand and manage due to *project visibility* i.e. better control over all the processes in the project because of clear visibility in all the phases
- Suitable for simple, risk-free projects with **stable** product statement, clear, well-known requirements with no ambiguities, technical requirements clear and resources ample or mission-critical applications (e.g., NASA)

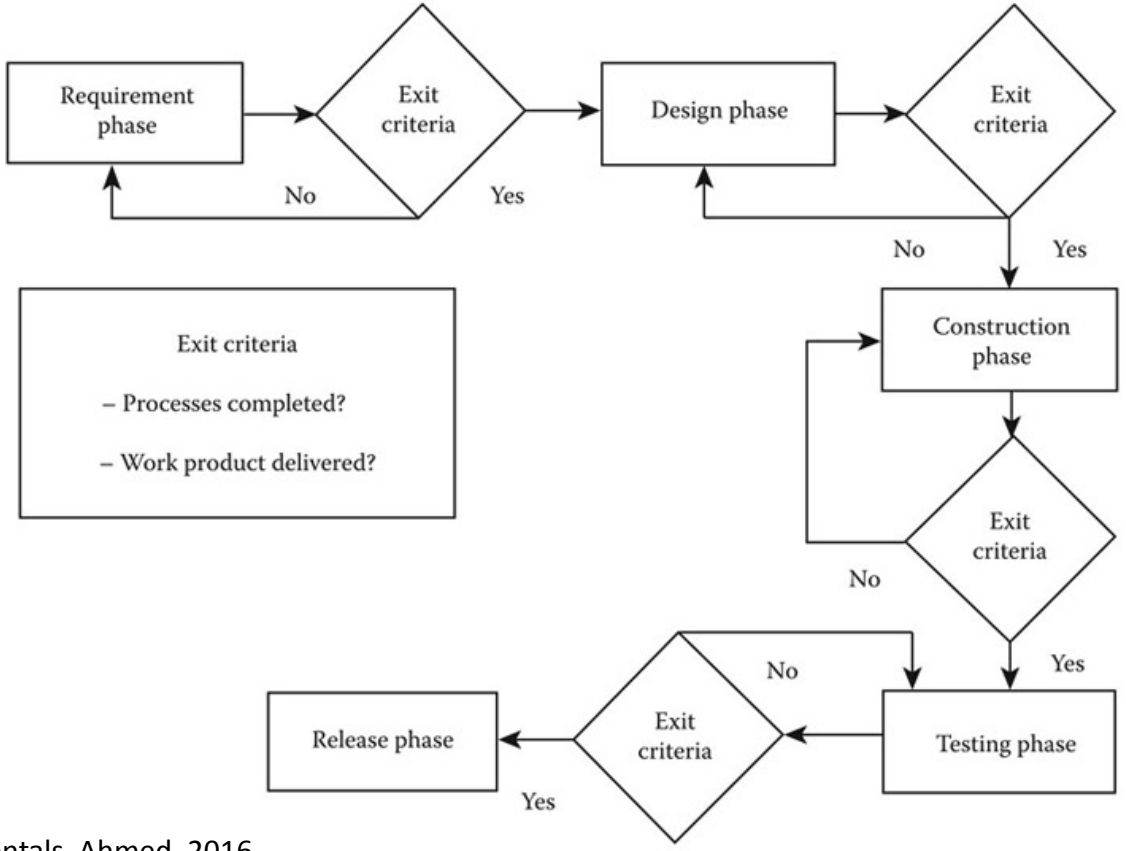


Waterfall process for software development.

Waterfall Model Variation

A **waterfall model variation** that implements a quality gate system

- A process model that ensures that quality is maintained throughout the life-cycle
- A ***completion criteria check*** is done that does a quality assurance check to see if all the necessary artifacts are generated for that phase and the artifacts meet the quality standards



Waterfall Model Drawbacks

- No working software is produced until late into the software life-cycle
- Rigid and not very flexible
  - Does not support fine-tuning or refinement of customer requirements through the cycle
  - Good ideas need to be identified upfront
  - As typically all requirements are frozen at the end of the requirements phase, once the application is implemented and in the "testing" phase, it is difficult to retract and change something that was not "well-thought out" in the concept phase or design phase
  - A great idea in the release cycle is a threat
- Heavy documentation (typically model based artifacts, UML)
- Not suitable for projects where requirements are at a moderate risk of changing
- Typically incurs a large management overhead

Rational Unified Process (RUP)

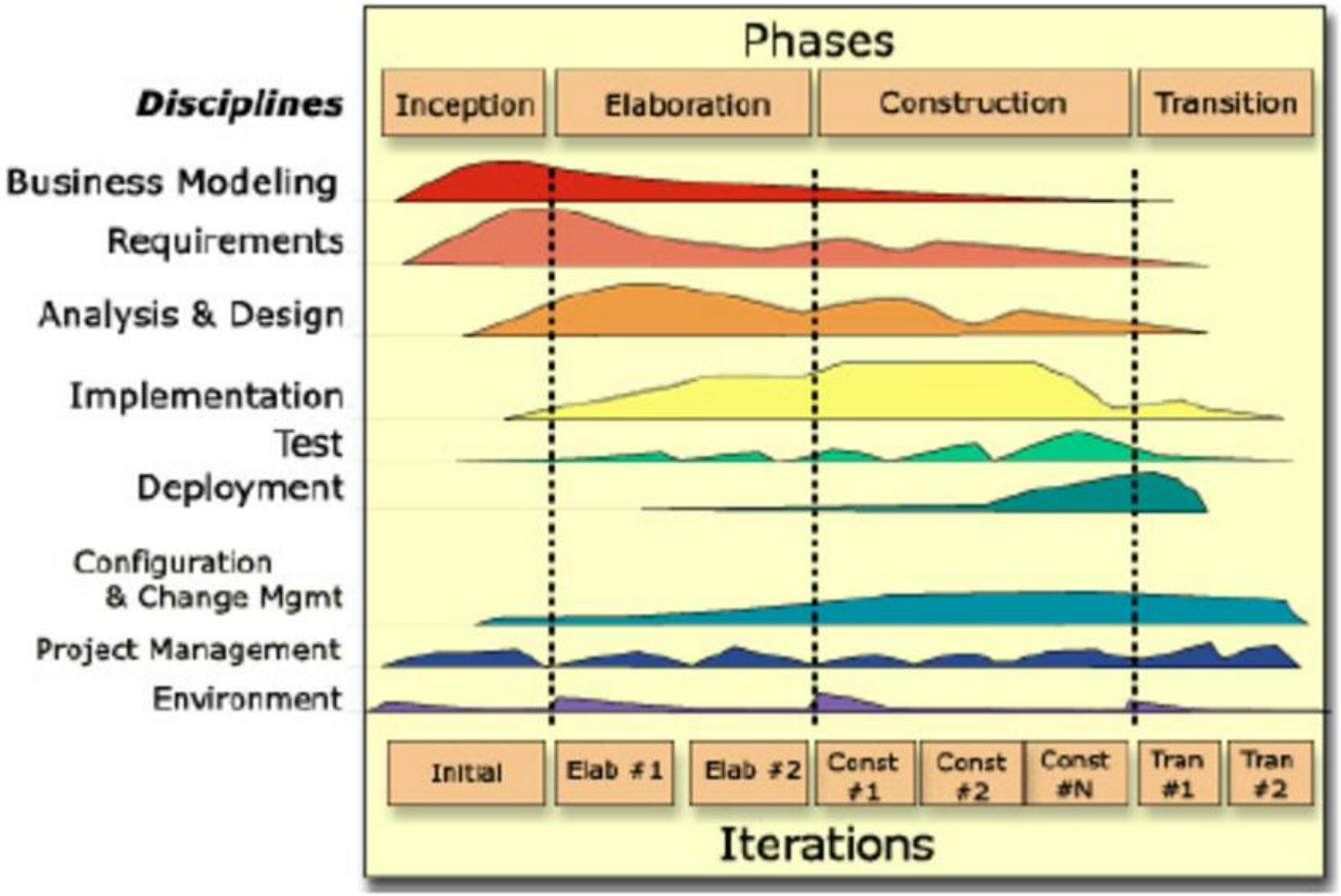
- An iterative software development process developed by Ivar Jacobson, Grady Booch and James Rumbaugh which has a series of four phases:
  - **Inception** – scope the project, identify major players, what resources are required, architecture and risks, estimate costs
  - **Elaboration –** understand problem domain, analysis, evaluate in detail required architecture and resources
  - **Construction –** design, build and test software
  - **Transition –** release software to production
- RUP is ***serial in the large*** and ***iterative in the small***
  - The four phases occur in a serial manner over time (hmm…sounds like sequential), however…
  - Work in an iterative manner on a day-to-day basis (some modelling, some implementation, some testing, some management…)

Rational Unified Process (RUP)

- All work in RUP organised into ***disciplines*** (previously workflows) :
  - **Development disciplines**
    - Business Modelling: understanding domain, develop a high-level requirements model (use-case model)
    - Requirements: Identify, model and document vision and requirements (use-case model, , domain model (class or data diagram), a business process model (a data flow diagram, activity diagram)
    - Analysis & Design: Engineer the blue-print
    - Implementation: Encode the design
    - Test: Testing throughout the project
    - Deployment: Product releases, software delivery
  - **Support disciplines**
    - Configuration and Change Management, Project Management, Environment
- **UP is not inherently documentation centric**.

Rational Unified Process (RUP)

❖



Agile Manifesto (Agile Alliance, 2001)

We are uncovering better ways of developing software by doing it and helping others do it.
Through this work we have come to value:

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

That is, while there is value in the items on
the right, we value the items on the left more.

Agile drawbacks

- Daily stand up meetings and close collaboration makes it difficult to adapt the process to development outsourcing, clients and developers separated geographically, or business clients who simply don't have the manpower, resources or interest to spare.
- Emphasis on modularity, incremental development, and adaptability doesn't suit it easily to clients who want contracts with firm estimates and timetables
- Reliance on small self-organized teams makes it difficult to adapt to large software projects with many stakeholders with different needs and neglects to take into account the need for leadership while team members get used to working together.
- Lack of comprehensive documentation can make it difficult to maintain or add to the software after members of the original team turn over, and can lead to modules with inconsistent features and interfaces.
- Lastly, more than with Waterfall and RUP, Agile development typically depends on the ability to recruit very experienced software engineers who know how to work independently and interface effectively with business users.

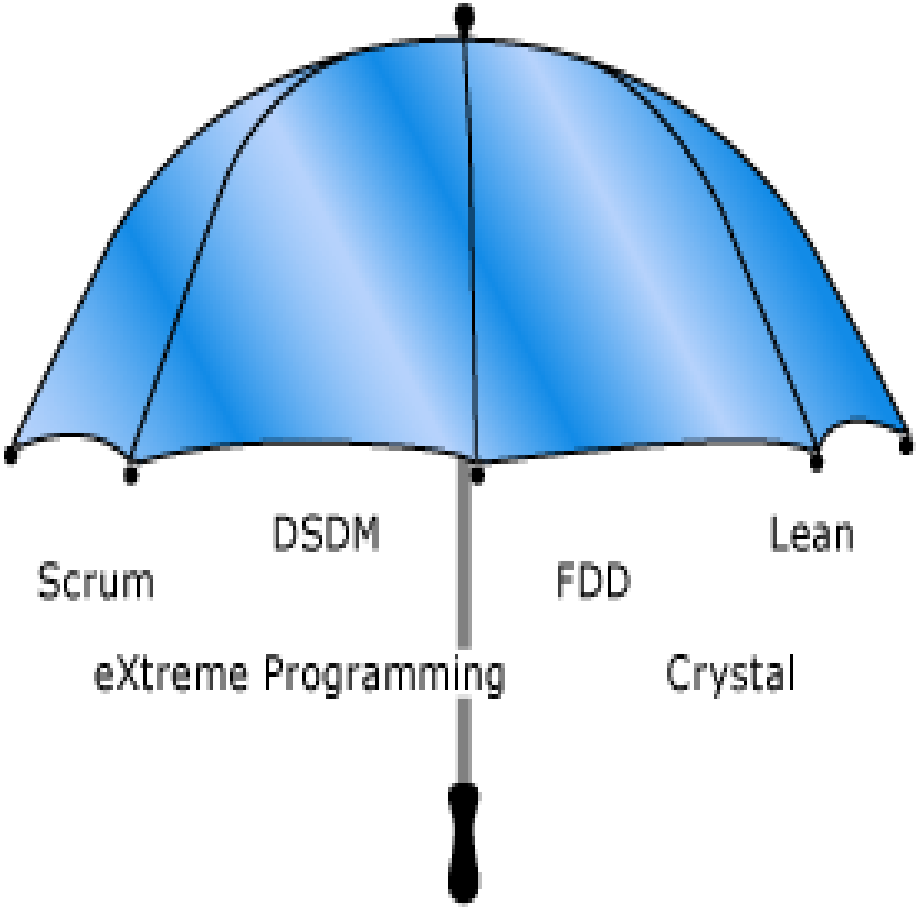
Which methodology?

- What does the customer want?

  - need software yesterday with the most advanced features at the lowest possible cost !

- No one methodology is the best fit
- secret to successful software development is to understand all three processes in depth and take the parts of each that are most suited to your particular product and environment.
- Stay agile in your approach through constant re-evaluation and revising the development process
- SaaS (Software as a Service) and Web 2.0 applications that require moderate adaptability are likely to be suited to agile style
- Mission-critical applications such as military, medical that require a high degree of predictability are more suited to waterfall

The Agile umbrella



Extreme Programming (XP)

A prominent agile software engineering methodology that:

- focuses on providing the highest value for the customer in the fastest possible way
- acknowledges changes to requirements as a *natural* and *inescapable* aspect of software development
- places higher value on ***adaptability*** (to changing requirements) over ***predictability*** (defining all requirements at the beginning of the project) – being able to adapt is a more realistic and cost-effective approach
- aims to lower the cost of change by introducing a set of basic values, principles and practices to bring more flexibility to changes

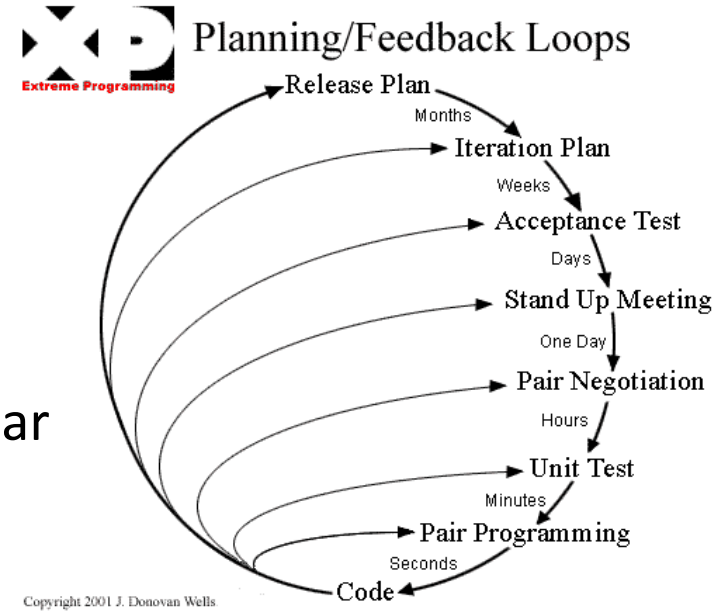
XP Principles

(1) XP Principle - Whole Team:

- Managers, developers and developers collaborate work closely as a single team, being aware of one another's problems and collaborating to solve these problems.
- Customer of an XP team (who defines and prioritises features) can be a group of BA or marketing specialists but must be a member of the team and available to the team at all times.

(2) XP Principle - Continuous Feedback:

- Developers - feedback by working in pairs, constant testing and continuous integration
- XP team- daily feedback on progress and obstacles through daily stand-up meetings
- Customers - feedback on progress with regular iteration demos
- XP developers produce working software progressively at a "steady heartbeat" and receive regular customer feedback and changes



(3) **XP Principle:** ***User Stories***

- a mnemonic token of an ongoing conversation about a requirement and are at the heart of the planning of XP

(4) **XP Principle: Short Development Cycles**

Based on the above user stories, an XP team create a:

Project Release plan

- Typically, a three months' worth of work, representing a major delivery and consists of prioritized collections of user stories (chosen by user) mapped out to the next six or so iterations
- Business determines the order in which the stories will be implemented in the release. If the team so desires, it can map out the first few iterations of the release by showing which stories will be completed in which iterations.

Iteration Plan:

- XP project works in **short cycles,** delivering a collection of user-stories (chosen by customer) every two weeks.
- Business cannot change the definition or priority of the user-stories once an iteration has been started.

(5) **XP Principle: High Quality**

Pair programming

- Code written by pairs of programmers working together intensely at the same workstation, where one member of the pair "codes" and the other "reviews". Ensures spread of knowledge through team and roles can change frequently.
- No **collective ownership**, pair can check out any module and improve it

Continuous Integration

- Programmers check their code in and integrate several times per day
- First one to check in wins, every one else merges

Sustainable pace – Team must run at a moderate, steady pace to conserve energy and alertness (a marathon, not a sprint) and **not** allowed to work overtime

Open Workspace – Teams work in an open room, typically with tables of 2 or 3 workstations, walls covered with status charts, UML artifacts…

Refactoring – Practice of making a series of tiny transformations to improve structure of the system without affecting the behaviour to prevent software degradation

(5) XP Principle: High Quality (contd …)

Test-driven Development **(TDD)** – Detailed test coverage at two levels:

Unit Testing

- A complete body of test-cases evolve along with the code.
- When you write code that passes a unit test, that code by definition is "testable".
- Unit tests encourages developers to "decouple" modules, so that they can be test independently (**Low coupling**) and facilitate "refactoring"

User Acceptance Testing

- Written by customers or business analysts
- Every detail about every feature is described in the acceptance test
- Serve as the final authority to determine the correctness of the implementation.
- Once an acceptance test passes, it is added to the body of passing acceptance tests and is never allowed to fail again.
- Growing body of acceptance tests run several times and every time the system is integrated and built (regression testing).

XP Principle: Simple Design

- An XP team focuses on the stories in the current iteration and keeps the designs simple and expressive.
- Migrate the design of the project from iteration to iteration to be the best design for the set of stories currently implemented
- Spike solutions, prototypes, CRC cards are popular techniques during design
- Three design mantras for developers:
  - *Consider the simplest possible design* for the current batch of user stories (e.g., if the current iteration can work with flat file, then don't use a database)
  - *You aren't going to need it* – Resist the temptation to add the infrastructure before it is needed (e.g., "Yeah, we know we're going to need that database one day, so we need put the hook in for it?)
  - *Once and only once* – XP developers don't tolerate duplication of code, wherever they find it, they eliminate it

XP Principle - System Metaphor:

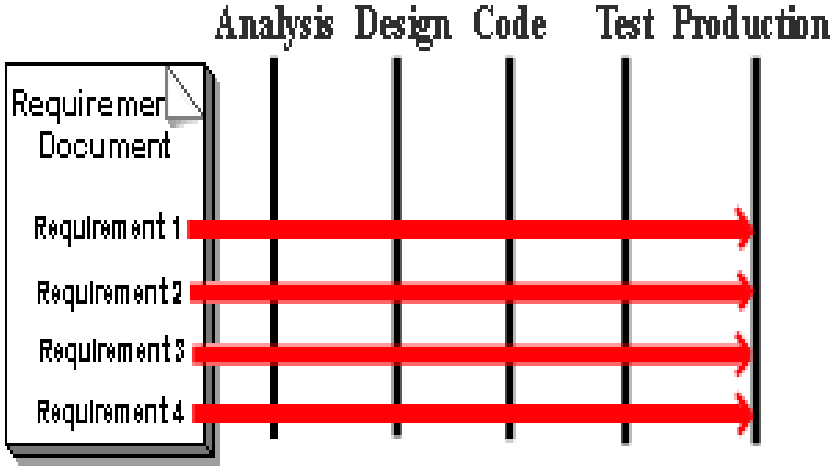
One of the trickiest, poorly understood principles of XP, yet one of the most important practices of all

A system metaphor in necessary for:

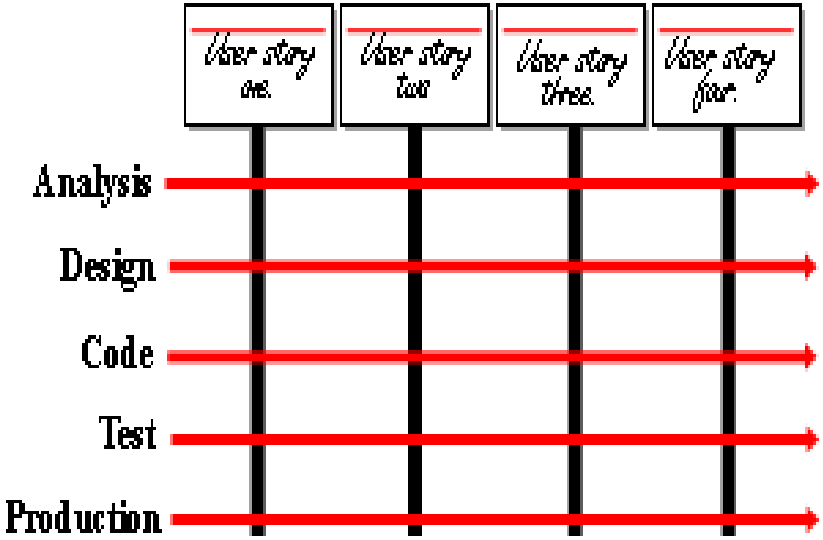
- ***A common vision:*** Makes it easier to understand what the system is (or what it could be) and enables everyone to agree on how the system works. The metaphor suggests the key structure of how the problem and the solution are perceived.
- ***Shared vocabulary:*** Often, helps to suggest a common system of names, which provide a vocabulary for elements in the system and helps define relationships between them (a powerful, specialised, shorthand vocabulary for the team)
- ***Architecture:*** Shapes the system, by identifying key objects and suggesting aspects of their interfaces. It supports the static and dynamic object models of the system.
- ***Generativity:*** The analogies of a metaphor can suggest new ideas about the system (both problems and solutions)

XP Programming Life-Cycle

Traditional software development is linear, with each stage of the lifecycle requiring completion of the previous stage.

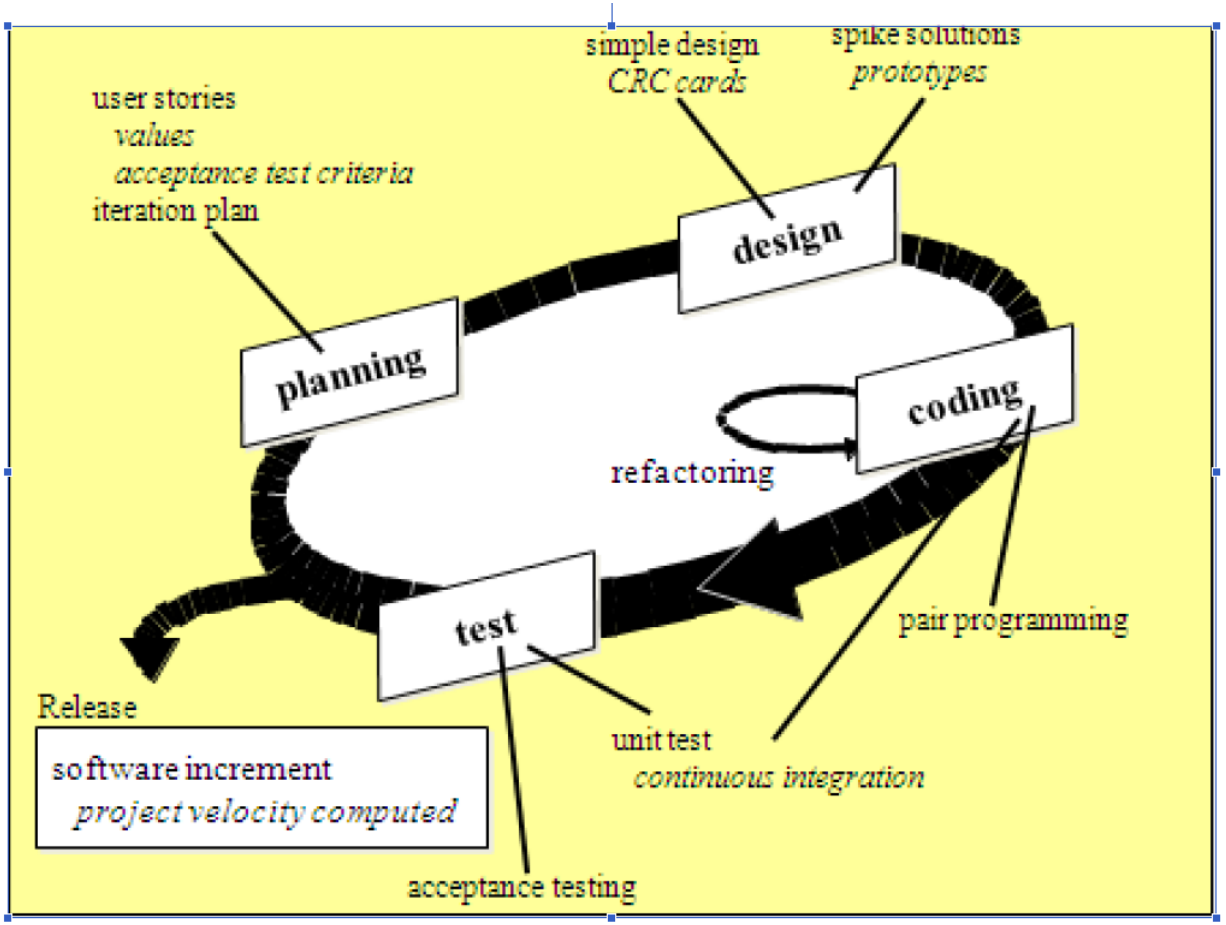


Extreme Programming (XP) turns the traditional software development process sideways, flipping the axis of the previous chart, where we visualise the activities, keeping the process itself a constant



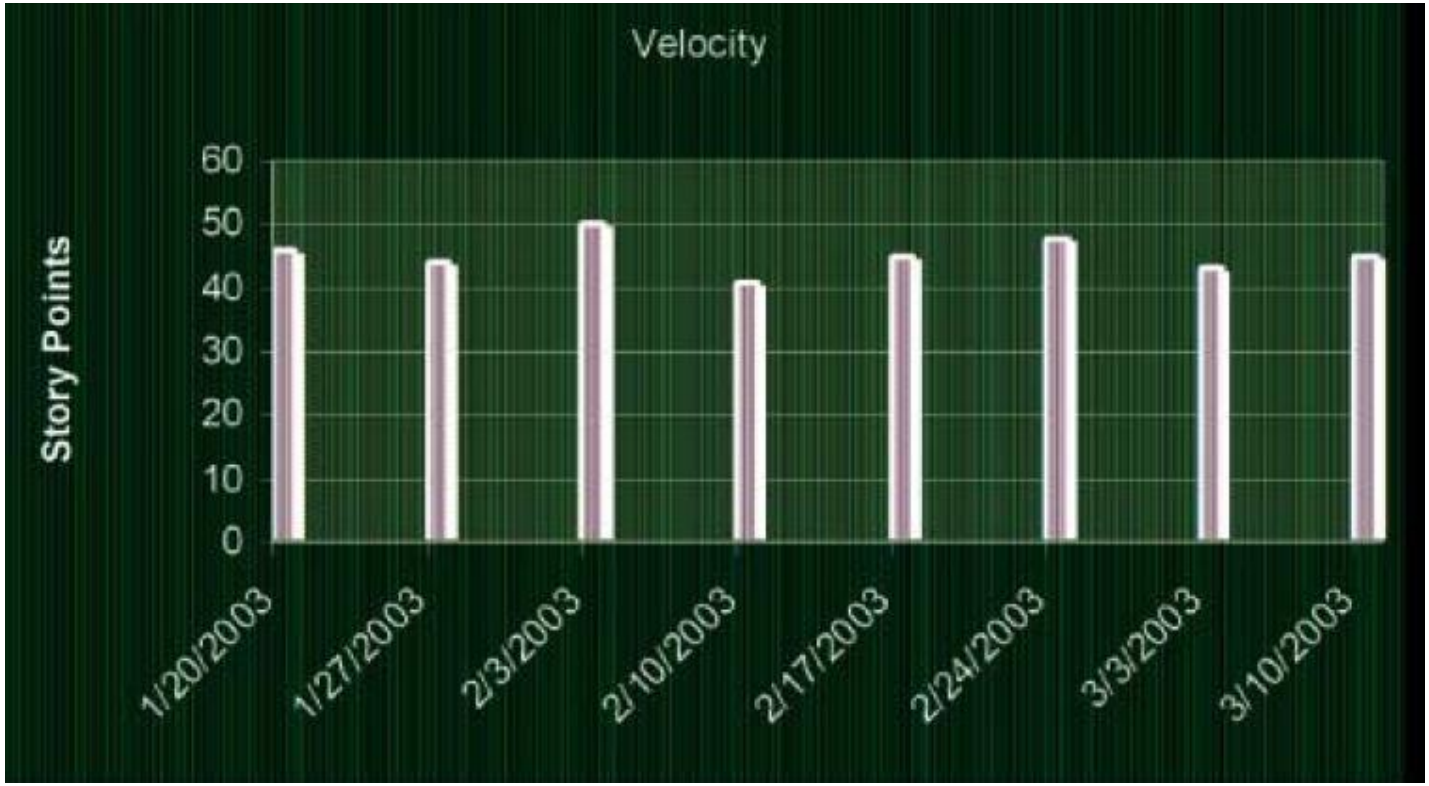
XP Programming Life-Cycle

The Extreme Programming software development process starts with **planning**, and all iterations consist of four basic phases in its life cycle: designing, coding, testing, and listening.



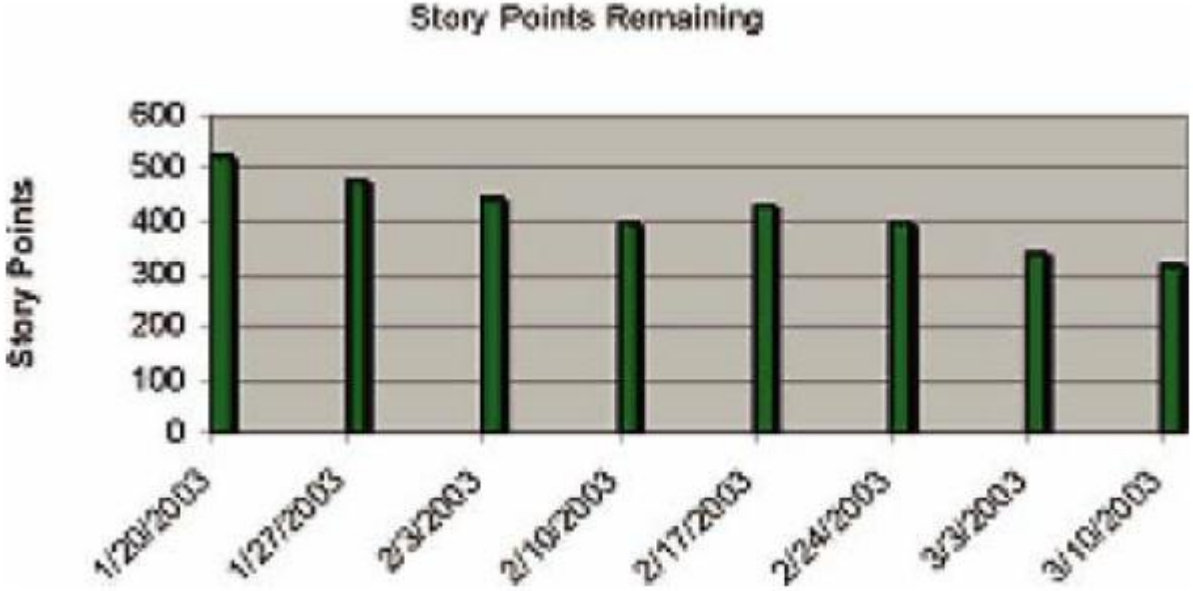
Project Tracking

- Project tracking – Recording the results of each iteration and use those results to predict what will happen in the next iteration
  - Tracking the total amount of work done during each iteration is the key to keep the project running at a **sustainable, steady pace**
  - XP teams use a ***velocity chart*** to track the project velocity which shows how many story points were completed (passed the user acceptance tests)
  - Average project velocity in this example is approximately 42 story points



Project Tracking

- XP Teams also use a *burn-down* chart to show the week-by-week progress
  - The slope of the chart is a reasonable predictor of the end-date
  - The difference between the bars in the burn-down chart does not equal the height of the bars in the velocity chart as new stories are being added to the project. (may also indicate that the developers have re-estimated the stories)



- Together, the velocity chart and the *burn-down* charts provide a reliable project management information for XP teams

When should XP be used?

- Useful for problem domains where requirements change
- When your customers do not have a firm idea of what they want or your system functionality is expected to change every few months
- XP was set up to address project risk. XP practices mitigate risk and increase the likelihood of success (e.g. a new challenge for a software group to be delivered by a specific date)
- XP ideally suitable for project group sizes of 2-12
- XP requires an extended development team comprising managers, developers and customers all collaborating closely
- XP also places great emphasis on *testability* and stresses creating automated unit and acceptance tests
- XP projects deliver greater productivity, although this was not aimed as the goal of XP

Requirements Engineering: The Agile Way

Requirements Engineering

- The key task of requirements engineering is formulating a *well-defined* problem to solve

- A *well-defined problem* includes:
  - A set of criteria ("requirements") according to which proposed solutions either definitely solve the problem or fail to solve it
  - The description of the resources and components at disposal to solve the problem.

- Requirements Engineering involves different stakeholders with varying stakes:
  - End users interested in the requested functionality
  - Business owner interested in cost and time-lines
  - Architects and developers interested in how well the functionality is implemented

What is a problem?

❖ **Is the storm a problem?**

- Only if you want to play outdoors
- Otherwise a blessing to a drought-stricken area

❖ Problem or not depending upon the goal we have

Traditional Phases of Requirements Engineering

Requirements Gathering

- Elicit requirements to enable developers to understand the business context through meetings, questionaries etc.
- Customers articulate what is required, what is to be accomplished, how the system will fit into the context of their business… the problem statement is rarely precise

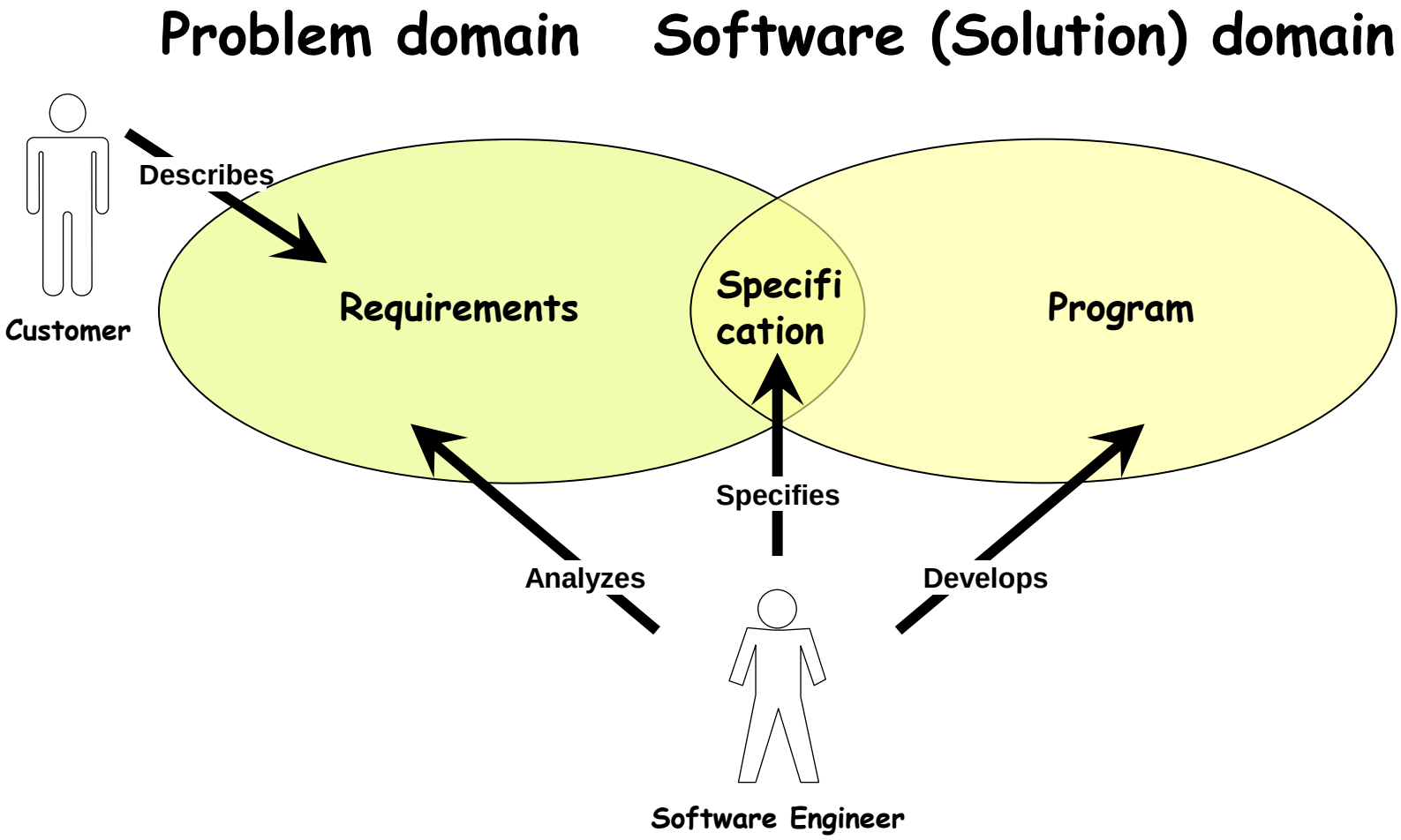
Requirements Analysis

- Refining and reason about the requirements gathered
- Identify dependencies, conflicts and risk
- Scope the project, negotiate with the customer to determine the priorities - what is important, what is realistic
- Ensure developer's understanding of the problem matches the customer's expectations

Requirements Specification

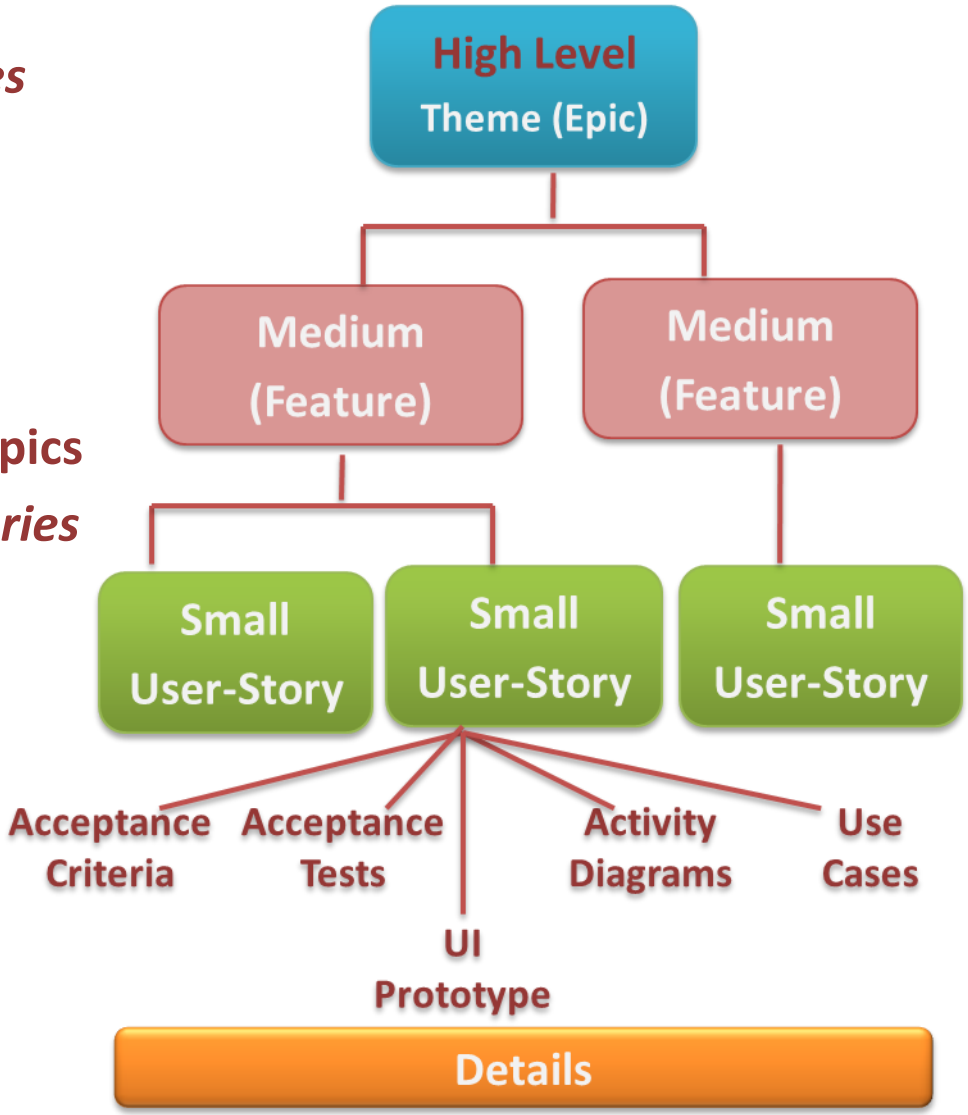
- Document the function, quality and constraints of software-to-be using formal, structured notations or graphical representation to ensure clarity, consistency and completeness
- Popular tool is **UML use-cases** (use-case scenarios describe how the end user will interact with the system)
- A formal **SRS** (System Requirements Specification) document is often produced in a typical traditional prescriptive process

Requirements and Specification



Agile Requirements Engineering

- Start with ***visioning***
  - To identify the ***Theme*** or ***Epic Stories***
  - What are the key features?
  - Who are the target users
  - What are the selling points of this product? (3-5 selling objectives)
- ***Brainstorm*** to identify features of the **epics**
- ***Breakdown*** features into small ***user-stories***
- ***Detail*** user-stories to yield iteration deliverables
  - Acceptance criteria
  - Screen sketches, Use case diagrams
  - UI Prototypes, Wire Frames (UI focussed projects)
  - Activity Diagrams (Process centred projects)
- A final list of user-stories or ***product backlog*** of user-stories is created



User Story

- **User Stories –** short, simple descriptions of a feature narrated from the perspective of the person who desires that capability

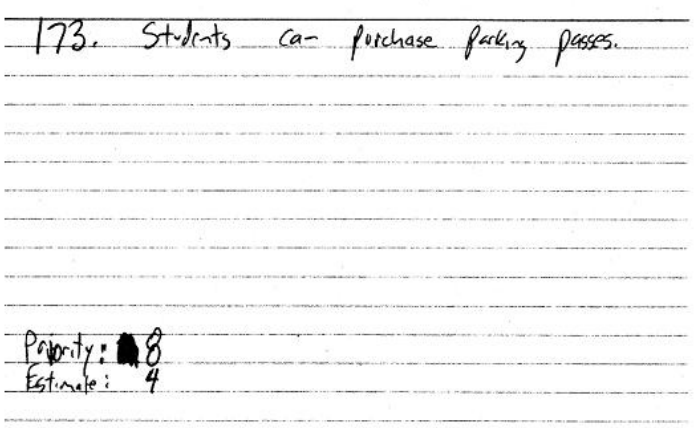
  As a < type of user >, I want < some goal > so that < some reason >

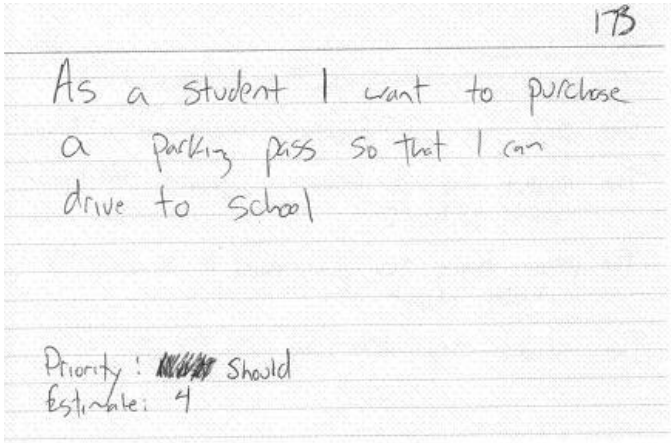
- Often written on index cards or sticky notes and arranged on walls or tables which helps to shift focus from writing about features to **planning** and **discussion (**more important than written text)

- Can be written at varying levels of details.
  - Epic user stories – covers large amount of functionality and generally to large for an agile team to complete in one iteration, e.g., a student can purchase a monthly or annual parking pass with my credit-card OR print student transcripts online
  - Split an epic user story into multiple smaller user stories e.g,
    - As a student, I can purchase a monthly or annual parking pass with my credit-card
    - As a student, I can purchase a monthly or annual parking pass with PayPal
    - As a student, I can order my official transcript online
    - As an admin, I can order an official transcript for any student
  - Or By adding “conditions of satisfaction”
- Anyone can write user stories (team member, and product owner’s responsibility to make sure a **product backlog of agile user** stories exist

Important considerations for User Stories

- Anyone can write user stories (team member, but product owner's responsibility to make sure a **product backlog of agile user** stories exist)
- Use the simplest tool - often written on index cards





- Assign a unique identifier e.g., US – 12
- Remember non-functional requirements (e.g., the *Students can purchase parking passes online* user story is a usage requirement similar to a use case whereas the *Transcripts will be available online via a standard browser* is closer to a *technical requirement*)
- Assign each user story a unique story ID (perhaps a story name)
- **Indicate the estimated size** - one way to estimate is to assign user story points to each card, a relative indication of how long it will take a pair of programmers to implement the story. (e.g., if a user-story point = 2.5 hours, then the user story in the above picture will take around 10 hours to implement)
- **Indicate the priority** (e.g., on a scale of 1 – 10)

Techniques to write a User Story (1)

- **Role-Feature-Reason** template or RGB (Role, Goal, Benefit), developed by Mike Cohn of Mountain Goat Software, 1991

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

  - e.g., As a *librarian*, I want *to have the facility to search for a book by different criteria such as author, title and ISBN* so that *I will save time to serve our customer*.
  - As a *student*, I'd like to *be able to search the course offerings*, so that *I'll be able to find an offering that most interests me.*

- This structure keeps the focus on the *who*, *what* and *why*
  - **The role (who):** describes *who* will be benefited by the feature, must clearly identify the specific type of user e.g., a manager, administrator, librarian, trainer, student etc.
  - **The feature (what):** describes ***what*** the user wants from the perspective of the user and **not** from the perspective of the developer who will be coding it e.g., feature = "search for a book", "search the course offering"
  - **The reason (why):** states why the user wants this feature. What benefit the user will get out of this feature? e.g. reason = "improve customer service", "find an offering that interests me". (If the value or benefit can't be articulated, it might be something that's not necessary)

Techniques to write a User Story (2)

- **The three C's model** – card, conversation and confirmation

  – developed by Ron Jeffries in 2001 to distinguish "social" user stories from "documentary" requirements practices such as use cases

- This formula captures the components of a User Story:
  - a ***"Card"*** (or often a Post-It note), a physical token giving tangible and durable form to what would otherwise only be an abstraction:
  - a ***"Conversation"*** taking place at different time and places during a project between the various people concerned by a given feature of a software product: customers, users, developers, testers; this conversation is largely verbal but most often supplemented by documentation;
  - the ***"Confirmation"***, finally, the more formal the better, developers need to get *confirmation* regarding the acceptance criteria from the product owner (*have a clear understanding how the feature will work in different situations, including error conditions)*

Techniques to write a User Story (3)

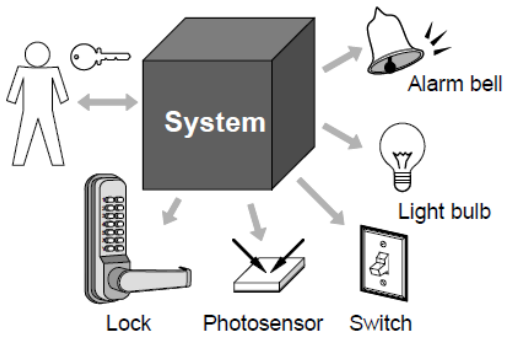
- **INVEST** – an acronym that helps evaluate whether you have a high quality user story
  - I = Independent: user story could be developed independently and delivered separately
  - N = Negotiable: user story should be discussable further
  - V = Valuable: the product owner should be "clear" on the "why" of the original statement (value of the user story)
  - E = Estimable: user story should be understandable enough so could be divided into task and could get estimated
  - S = Small: user story should be small, deliverable within an iteration
  - T = Testable: user story should be defined with clear *acceptance criteria*, both the correct functionality and the error conditions

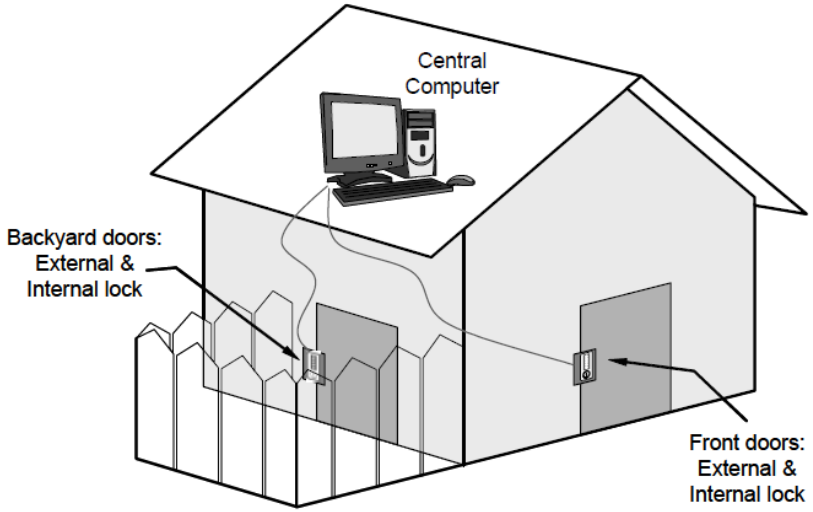
An agile functional specification

- An agile functional specification requires more than user-stories for a successful user experience
  - User stories are excellent for capturing product functionality and serve as a useful planning tool, but do not express user journey or visual design well
- Complement user stories with other techniques:
  - Use case diagrams
  - Activity diagrams,
  - Screen sketches, mock-ups, prototypes, wire frames

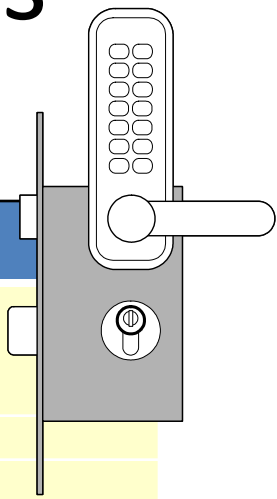
Home Access Case Study

- A home access control system for several functions such as door lock control, lighting control, intrusion detection
- First iteration – Support basic door unlocking and locking functions





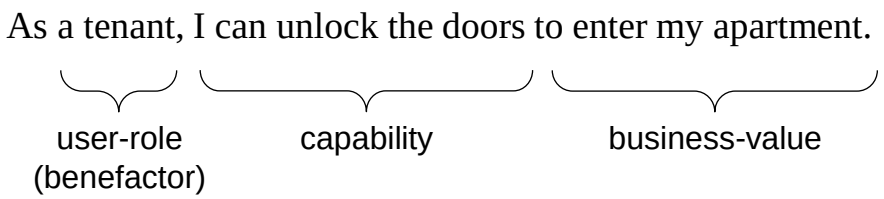
Example System Requirements



Identifier	Priority	Requirement
REQ1	5	The system shall keep the door locked at all times, unless commanded otherwise by authorized user. When the lock is disarmed, a countdown shall be initiated at the end of which the lock shall be automatically armed (if still disarmed).
REQ2	2	The system shall lock the door when commanded by pressing a dedicated button.
REQ3	5	The system shall, given a valid key code, unlock the door and activate other devices.
REQ4	4	The system should allow mistakes while entering the key code. However, to resist "dictionary attacks," the number of allowed failed attempts shall be small, say three, after which the system will block and the alarm bell shall be sounded.
REQ5	2	The system shall maintain a history log of all attempted accesses for later review.
REQ6	2	The system should allow adding new authorized persons at runtime or removing existing ones.
REQ7	2	The system shall allow configuring the preferences for device activation when the user provides a valid key code, as well as when a burglary attempt is detected.
REQ8	1	The system should allow searching the history log by specifying one or more of these parameters: the time frame, the actor role, the door location, or the event type (unlock, lock, power failure, etc.). This function shall be available over the Web by pointing a browser to a specified URL.
REQ9	1	The system should allow filing inquiries about "suspicious" accesses. This function shall be available over the Web.

- Problem: Requirements prioritization.
- See how solved in **agile methods**.

User Stories For Home Access Control



- Similar to system requirements, but focus on the user benefits, instead on system features.
- Preferred tool in **agile methods**.

Example User Stories

Identifier	User Story	Size
ST-1	As an authorized person (tenant or landlord), I can keep the doors locked at all times.	**4 points**
ST-2	As an authorized person (tenant or landlord), I can lock the doors on demand.	**3 pts**
ST-3	The lock should be automatically locked after a defined period of time.	**6 pts**
ST-4	As an authorized person (tenant or landlord), I can unlock the doors. (Test: Allow a small number of mistakes, say three.)	**9 points**
ST-5	As a landlord, I can at runtime manage authorized persons.	**10 pts**
ST-6	As an authorized person (tenant or landlord), I can view past accesses.	**6 pts**
ST-7	As a tenant, I can configure the preferences for activation of various devices.	**6 pts**
ST-8	As a tenant, I can file complaint about “suspicious” accesses.	**6 pts**

- Story priority is given by its order of appearance on the work backlog (described next)
- Size (Effort) estimated in story points (last column)

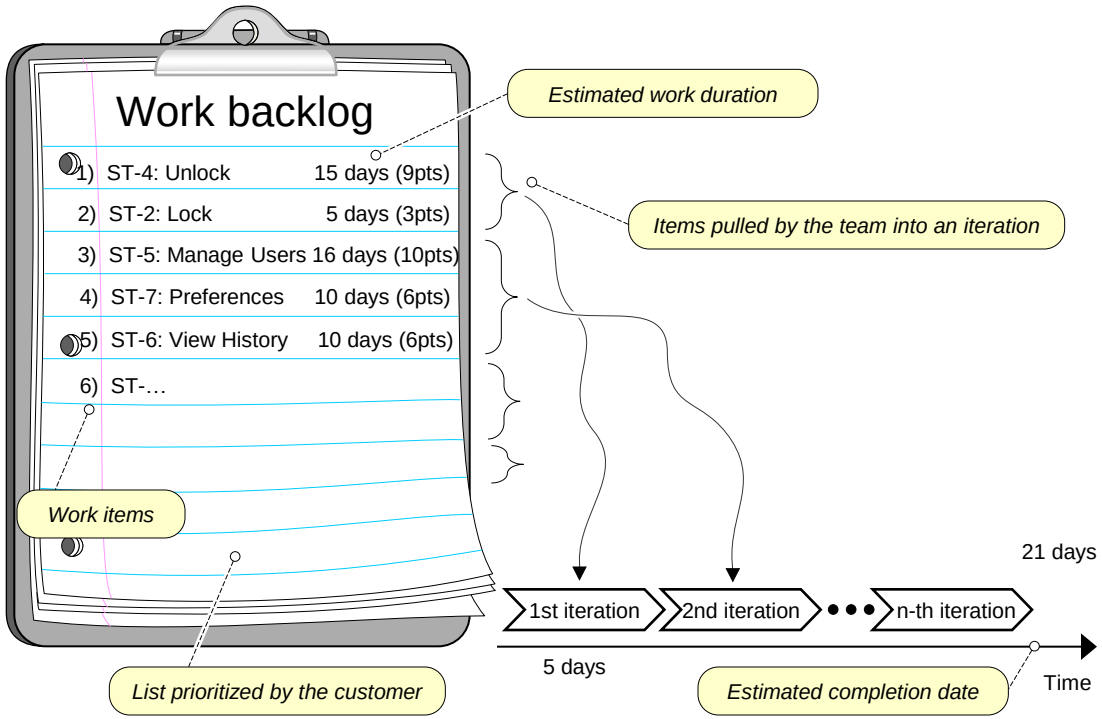
Project Estimation using User Story Points

- Size points assigned to each user story
- Total work size estimate:
  - Total size = $\sum$ (points-for-story ***i***), $i = 1..N$
- Velocity (= team's productivity)
  - estimated from experience (Number of user-story points that the team can complete per single iteration)

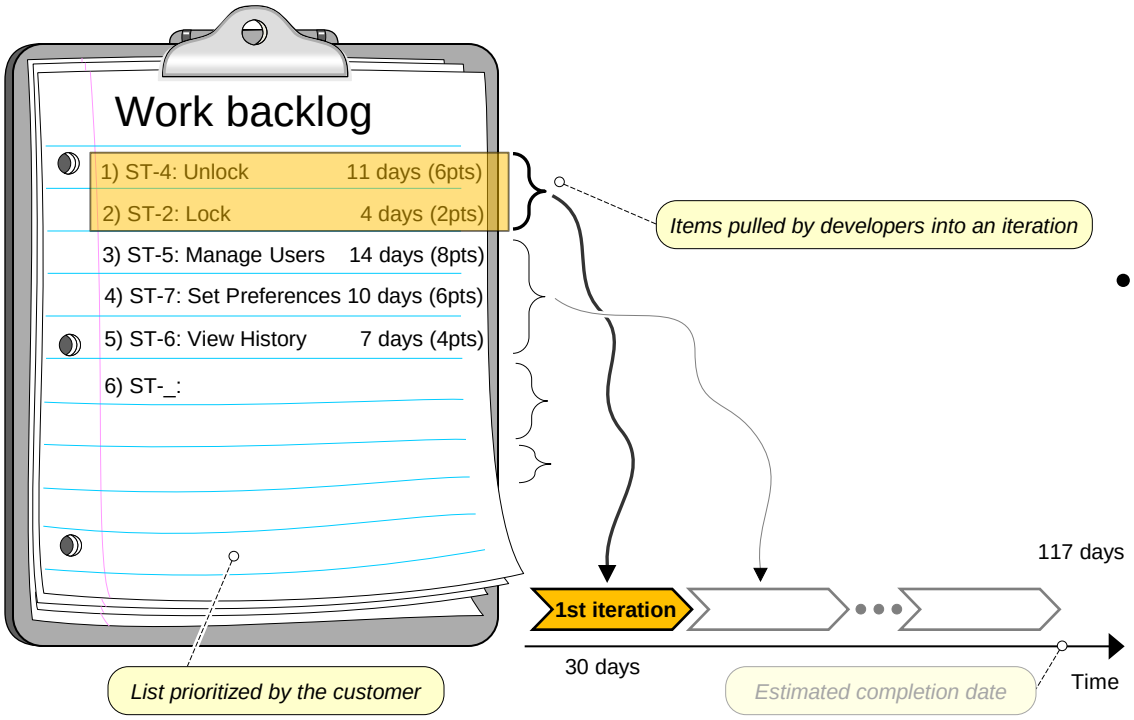
- Estimate the work duration

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

Agile Estimation of Project Effort

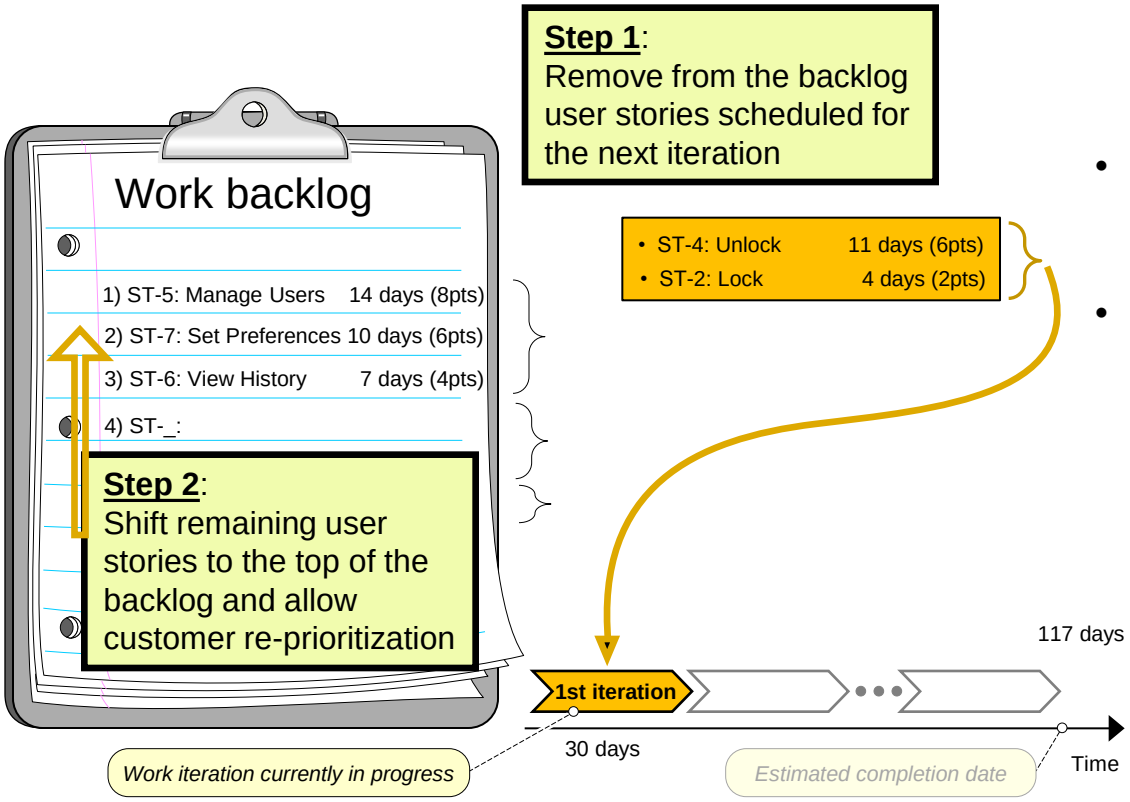


Agile Prioritization of Work



- 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

Types of Requirements

- Functional Requirements
- Non-functional requirements (or quality requirements)
  - FURPS+
  - Functionality (security), Usability, Reliability, Performance , Supportability
- On-screen appearance requirements

Acceptance Tests

- Means of assessing that the requirements are met as expected
- Conducted by the customer
- An acceptance test describes whether the system will pass or fail the test, given specific input values
- Cannot ever guarantee 100% coverage of all usage scenarios, but *systematic approach* can increase the degree of coverage

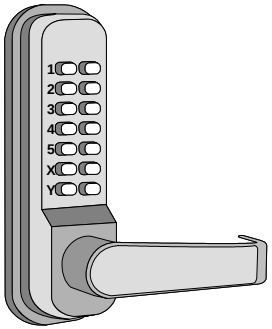
Use Case Modelling

Use Cases

- Used for Functional Requirements Analysis and Specification
- A ***use case*** is a step-by-step description of how a user will use the system-to-be to accomplish business goals
  - Detailed use cases are usually written as *usage scenarios* or *scripts*, showing an envisioned sequence of actions and interactions between the external actors and the system-to-be

Deriving Use Cases from System Requirements

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



Actor	Actor's Goal (what the actor intends to accomplish)	Use Case Name
Landlord	To disarm the lock and enter, and get space lighted up.	**Unlock (UC-1)**
Landlord	To lock the door & shut the lights (sometimes?).	**Lock (UC-2)**
Landlord	To create a new user account and allow access to home.	**AddUser (UC-3)**
Landlord	To retire an existing user account and disable access.	**RemoveUser (UC-4)**
Tenant	To find out who accessed the home in a given interval of time and potentially file complaints.	**InspectAccessHistory (UC-5)**
Tenant	To disarm the lock and enter, and get space lighted up.	**Unlock (UC-1)**
Tenant	To lock the door & shut the lights (sometimes?).	**Lock (UC-2)**
Tenant	To configure the device activation preferences.	**SetDevicePrefs (UC-6)**
LockDevice	To control the physical lock mechanism.	**UC-1, UC-2**
LightSwitch	To control the lightbulb.	**UC-1, UC-2**
[to be identified]	To auto-lock the door if it is left unlocked for a given interval of time.	**AutoLock (UC-2)**

(Actors already given if working from user stories instead of system requirements)

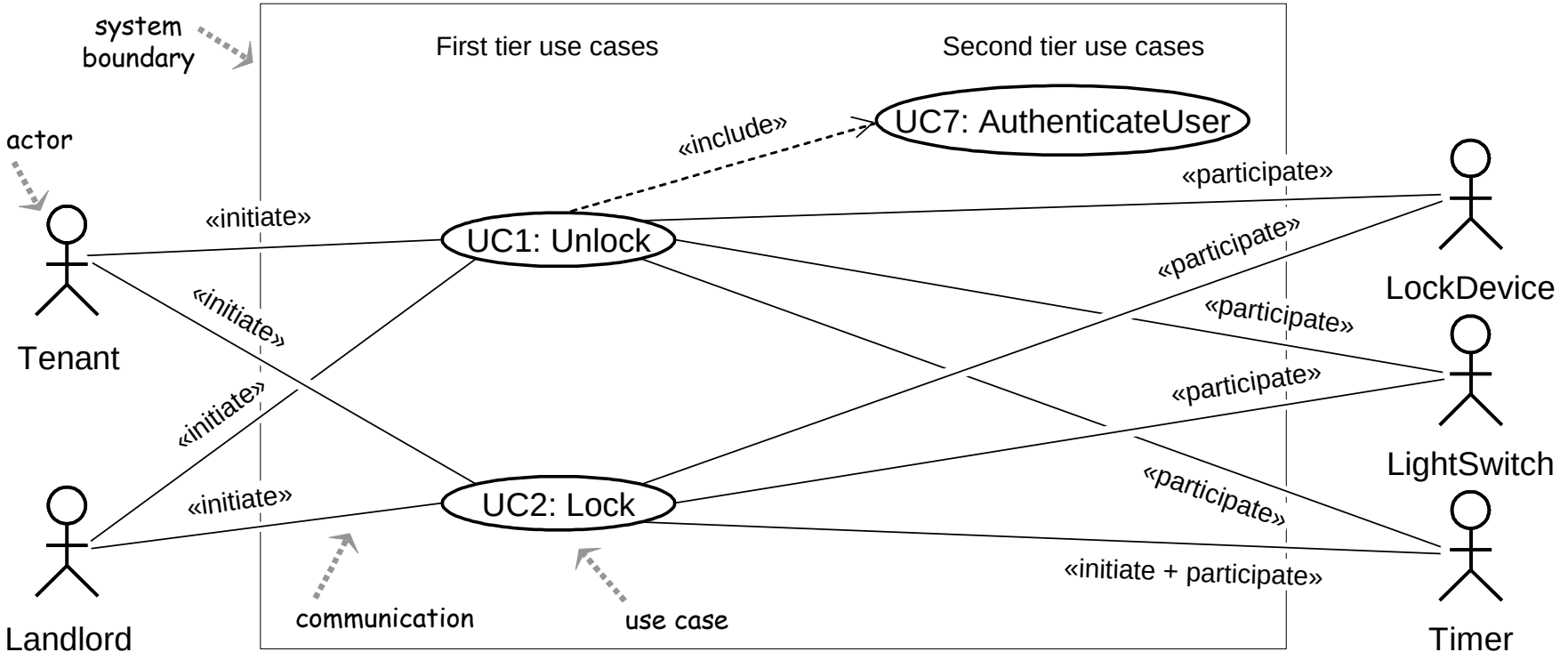
Types of Actors

- ***Initiating actor*** (also called *primary actor* or simply "user"): initiates the use case to achieve a goal

- ***Participating actor*** (also called *secondary actor*): participates in the use case but does not initiate it. Subtypes of participating actors:
  - ***Supporting actor***: helps the system-to-be to complete the use case
  - ***Offstage actor***: passively participates in the use case, i.e., neither initiates nor helps complete the use case, but may be notified about some aspect of it (e.g., for keeping records)

Use Case Diagram: Device Control

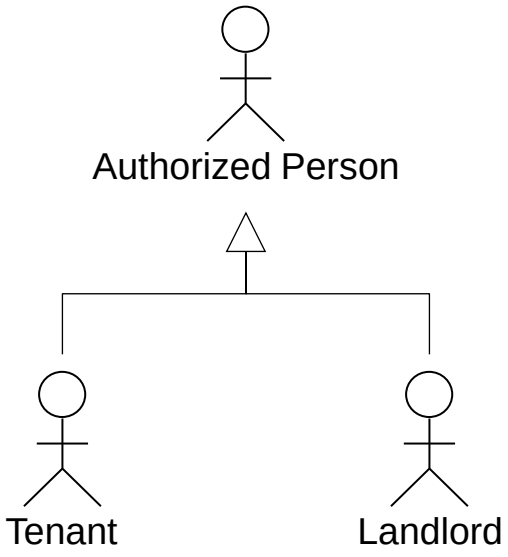
UC1: Unlock
UC2: Lock
UC3: AddUser
UC4: RemoveUser
UC5: InspectAccessHistory
UC6: SetDevicePrefs
UC7: AuthenticateUser
UC8: Login



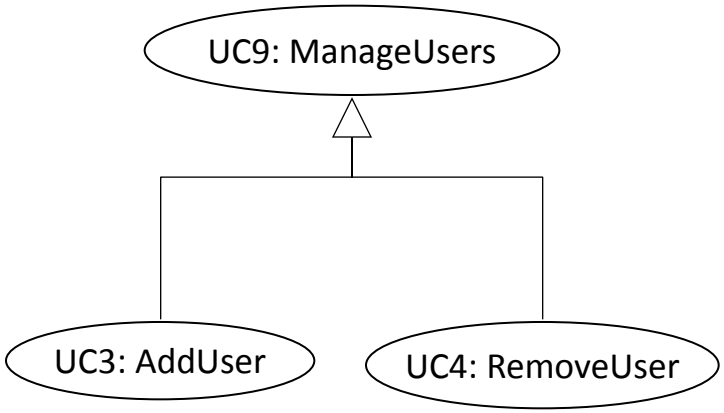
Use Case Generalizations

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

- More abstract representations can be derived from particular representations



Actor Generalization

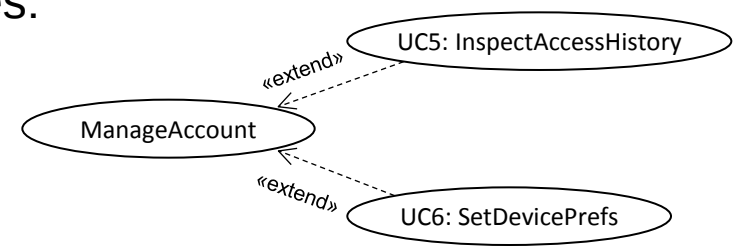


Use Case Generalization

Optional Use Cases: «extend»

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

Example optional use cases:



Key differences between **«include»** and **«extend»** relationships

	Included use case	Extending use case
Is this use case optional?	No	Yes
Is the base use case complete without this use case?	No	Yes
Is the execution of this use case conditional?	No	Yes
Does this use case change the behavior of the base use case?	No	Yes

[Source: Robert Maksimchuk & Eric Naiburg: ***UML for Mere Mortals***, Addison-Wesley, 2005.]

Traceability Matrix

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

Continued for domain model, design diagrams, …

Schema for Detailed Use Cases

Use Case UC-#:	**Name / Identifier [verb phrase]**
Related Requirements:	List of the requirements that are addressed by this use case
Initiating Actor:	Actor who initiates interaction with the system to accomplish a goal
Actor's Goal:	Informal description of the initiating actor's goal
Participating Actors:	Actors that will help achieve the goal or need to know about the outcome
Preconditions:	What is assumed about the state of the system before the interaction starts
Postconditions:	What are the results after the goal is achieved or abandoned; i.e., what must be true about the system at the time the execution of this use case is completed

Flow of Events for Main Success Scenario:		
→	1.	The initiating actor delivers an action or stimulus to the system (the arrow indicates the direction of interaction, to- or from the system)
←	2.	The system's reaction or response to the stimulus; the system can also send a message to a participating actor, if any
→	3.	…

Flow of Events for Extensions (Alternate Scenarios): What could go wrong? List the exceptions to the routine and describe how they are handled		
→	1a.	For example, actor enters invalid data
←	2a.	For example, power outage, network failure, or requested data unavailable
		…

The arrows on the left indicate the direction of interaction: → Actor's action; ← System's reaction

Use Case 1: Unlock

Use Case UC-1:	**Unlock**
Related Requirem'ts:	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]

Subroutine «include» Use Case

Use Case UC-7:	**AuthenticateUser** (sub-use case)
Related Requirements:	REQ3, REQ4 stated in Table 2-1
Initiating Actor:	Any of: Tenant, Landlord
Actor's Goal:	To be positively identified by the system (at the door interface).
Participating Actors:	AlarmBell, Police
Preconditions:	• The set of valid keys stored in the system database is non-empty. • The counter of authentication attempts equals zero.
Postconditions:	None worth mentioning.

Flow of Events for Main Success Scenario:

← 1. **System** prompts the actor for identification, e.g., alphanumeric key

→ 2. **Tenant**/**Landlord** supplies a valid identification key

← 3. **System** (a) verifies that the key is valid, and (b) signals to the actor the key validity

Flow of Events for Extensions (Alternate Scenarios):

2a. **Tenant**/**Landlord** enters an invalid identification key

← 1. **System** (a) detects error, (b) marks a failed attempt, and (c) signals to the actor

← 1a. **System** (a) detects that the count of failed attempts exceeds the maximum allowed number, (b) signals to sound **AlarmBell**, and (c) notifies the **Police** actor of a possible break-in

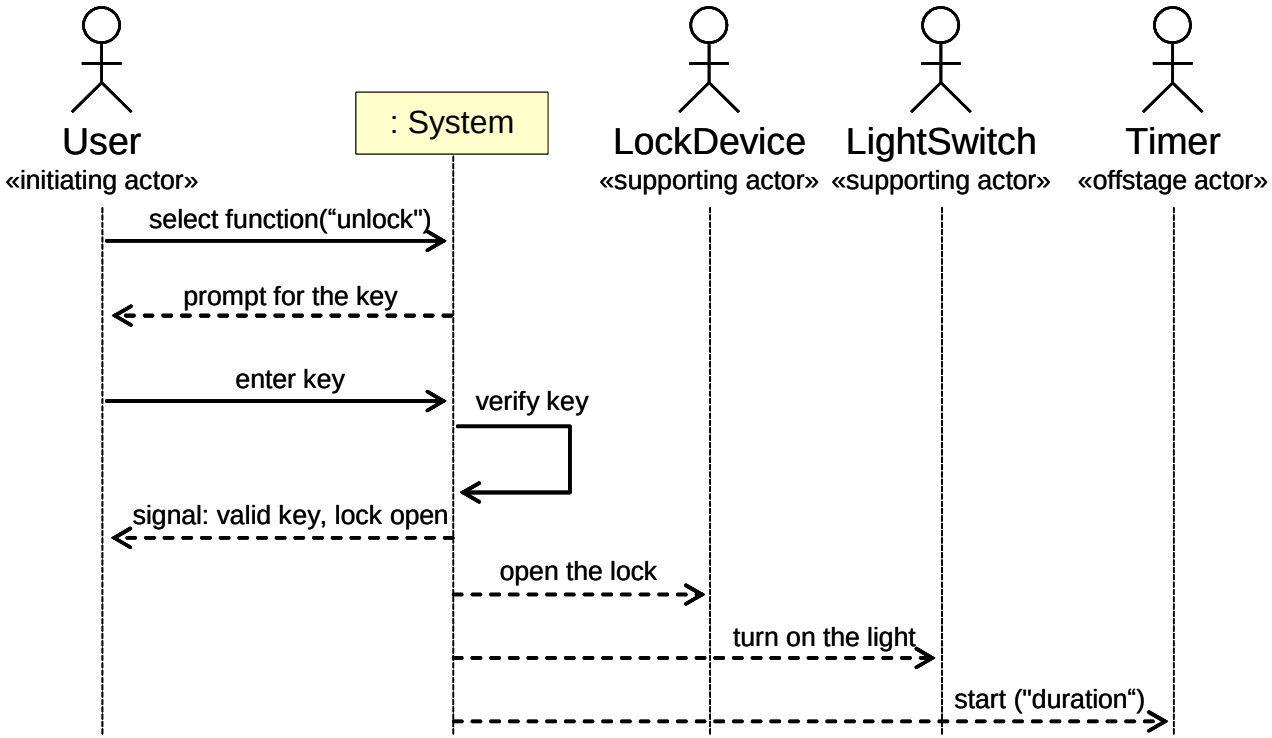
→ 2. **Tenant**/**Landlord** supplies a valid identification key

3. Same as in Step 3 above

System Sequence Diagram

[Modeling System Workflows]

Use case: Unlock



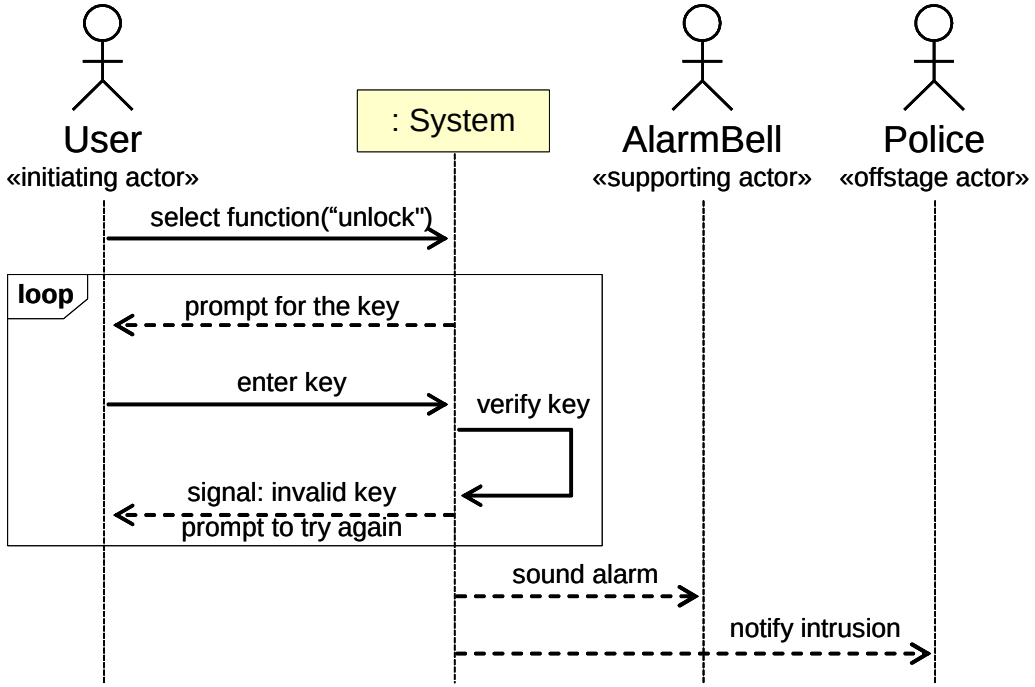
Main success scenario

Similar to UML sequence diagrams,
but for *actor interactions* instead of software *object interactions*

System Sequence Diagram

[Modeling System Workflows]

Use case: Unlock



Alternate scenario (burglary attempt)