COMP 1531
Software Engineering Fundamentals

Summary of Lectures 01, 02 and 05

Aarthi Natarajan

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
- A software engineer starts with a *problem definition* and applies tools and techniques to obtain a *problem solution*. However…
- Software engineering requires great emphasis on ***methodology*** or the *method* for managing the development process in addition to great skills with tools and techniques.

Software Engineering is not Programming

- Software engineering:
  - \- Understanding the business problem (system-to-be, interaction between this system and its users and environment)
  - Creative formulation of ideas to solve the problem based on this understanding
  - Designing the “blueprint” or architecture of the solution
- Programming:
  - Craft of *implementing* the given “blueprint”

Role of a software engineer

- Software engineering ***delivers*** value to the customer

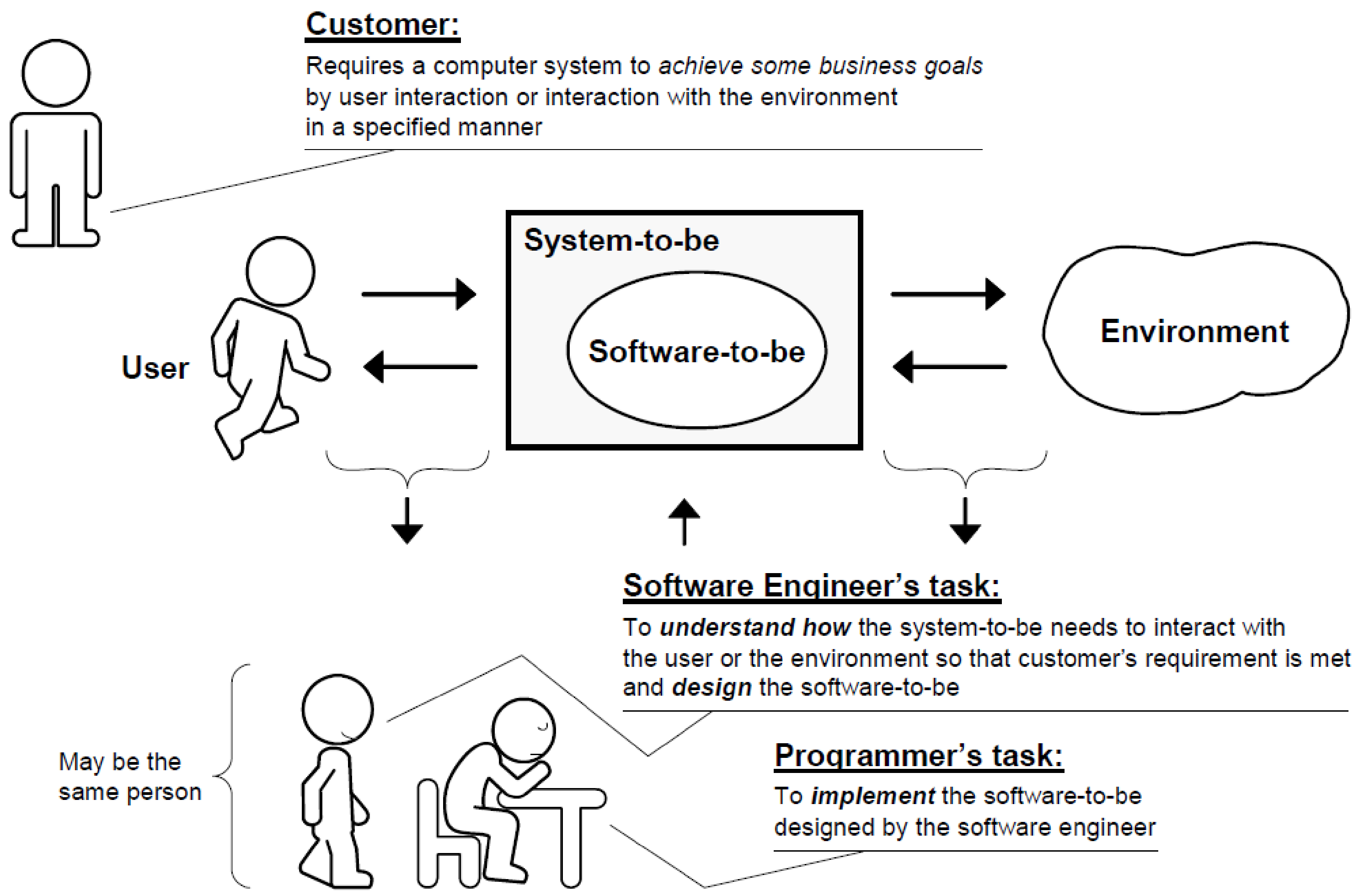


Figure 1: Role of Software Engineering

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:

- A process of knowledge-discovery about the “system-to-be” and list of features, where software engineers need to:
  - understand the problem definition – identify 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…

Software Engineering Life-Cycle

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 (design artifacts e.g., domain model)

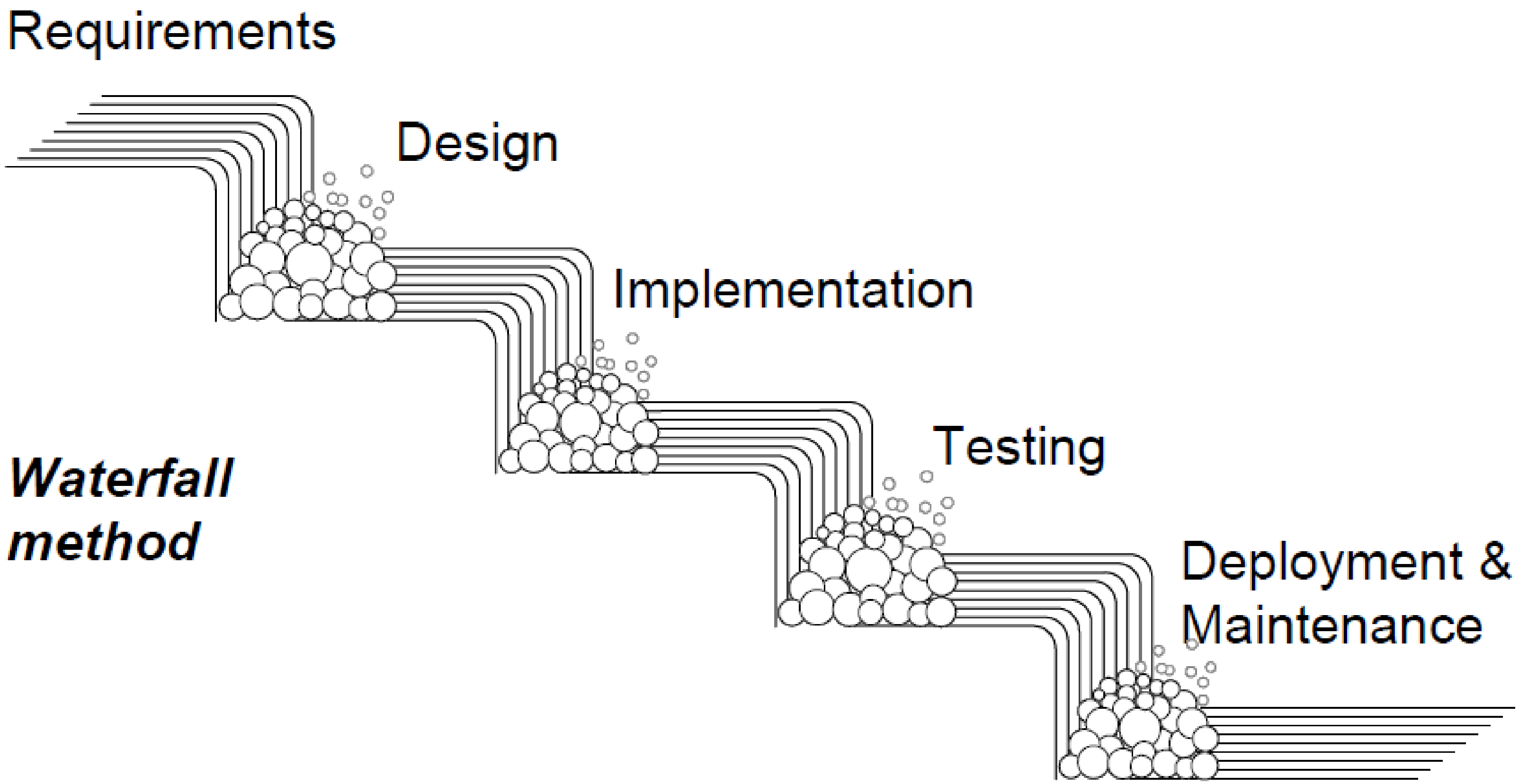
Software Engineering Life-Cycle

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 Development Methodologies

- Elaborate processes:
  - Rigid, plan-driven documentation heavy methodologies e.g., Waterfall model
  - 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

Waterfall Model (1970's)



Waterfall process for software development.

Waterfall Model (1970's)

- Linear, sequential project life-cycle (*plan-driven development model*) characterized by detailed planning:
  - Problem is identified, documented and designed
  - Implementation tasks identified, scoped and scheduled
  - Development cycle followed by testing cycle
- Each phase must be fully completed, documented and signed off before moving on to the next phase
- Simple to understand and manage due to *project visibility*
- Suitable for 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 Model Drawbacks

- No working software produced until late into the software life-cycle
- Rigid and not very flexible
  - No support for fine-tuning of requirements through the cycle
  - Good ideas need to identified upfront; a great idea in the release cycle is a threat!
  - All requirements 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
- Heavy documentation (typically model based artifacts, UML)
- Incurs a large management overhead
- Not suitable for projects where requirements are at a moderate risk of changing

Software Development Challenges

- Software is:
  - – probably, the most complex artifact
  - – intangible and hard to visualise
  - - the most flexible artifact – radically modified at any stage of software development
- - Linear order of understanding the problem domain, designing a solution, implementing and deploying the solution, does not <u>always</u> produce best results
- - Easier to understand a complex problem by implementing and evaluating pilot solutions.

Incremental and Iterative Project Life-Cycle

- Incremental and iterative approaches:
  1. Break the big problem down into smaller pieces and prioritize them.
  2. An “iteration” refers to a step in the life-cycle
  3. Each “iteration” results in an “increment” or progress through the overall project
  4. Seek the customer feedback and change course based on improved understanding at the end of each iteration
- 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

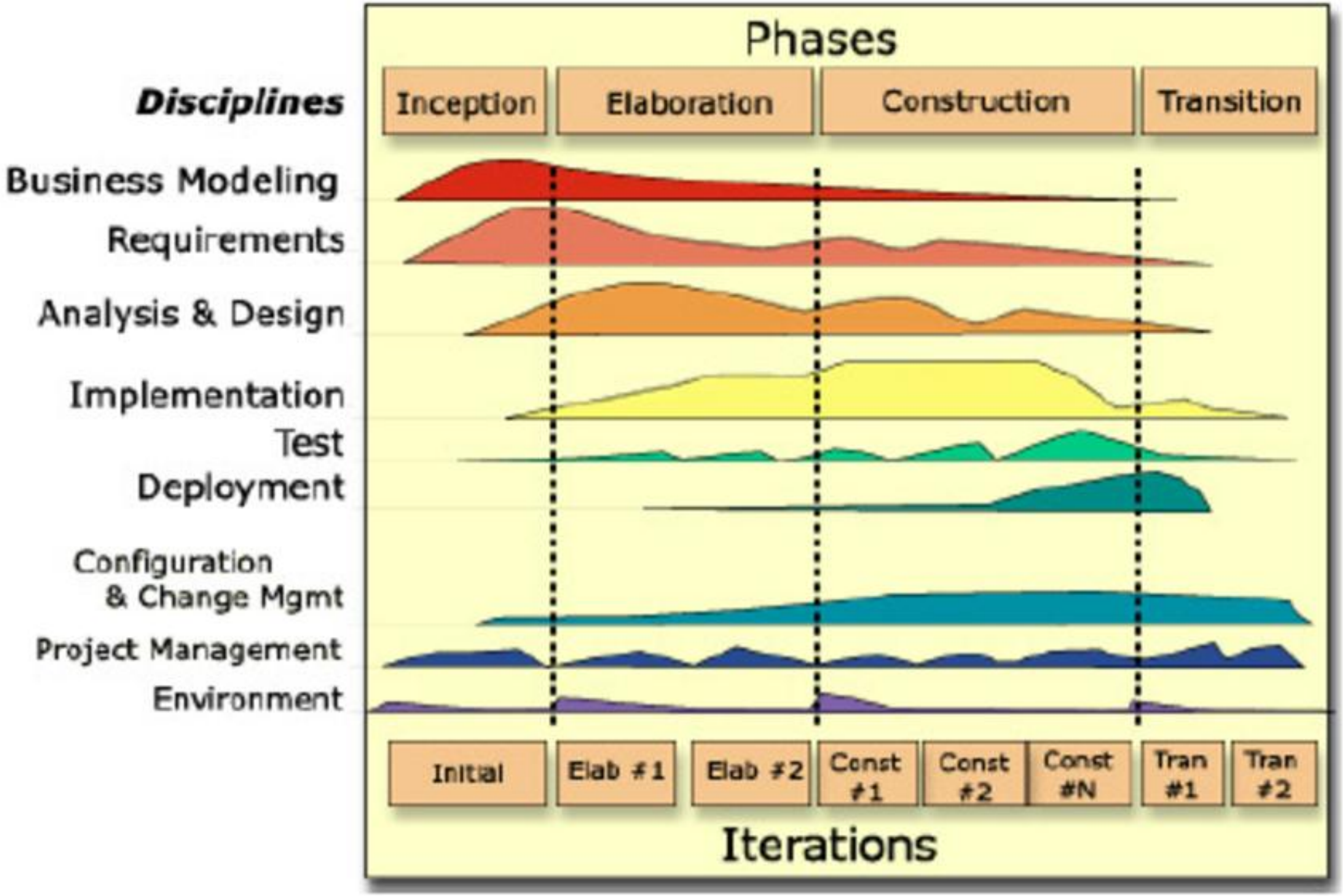
Rational Unified Process (RUP)

- An iterative software development process developed by Ivar Jacobson, Grady Booch and James Rumbaugh and consists of four major 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

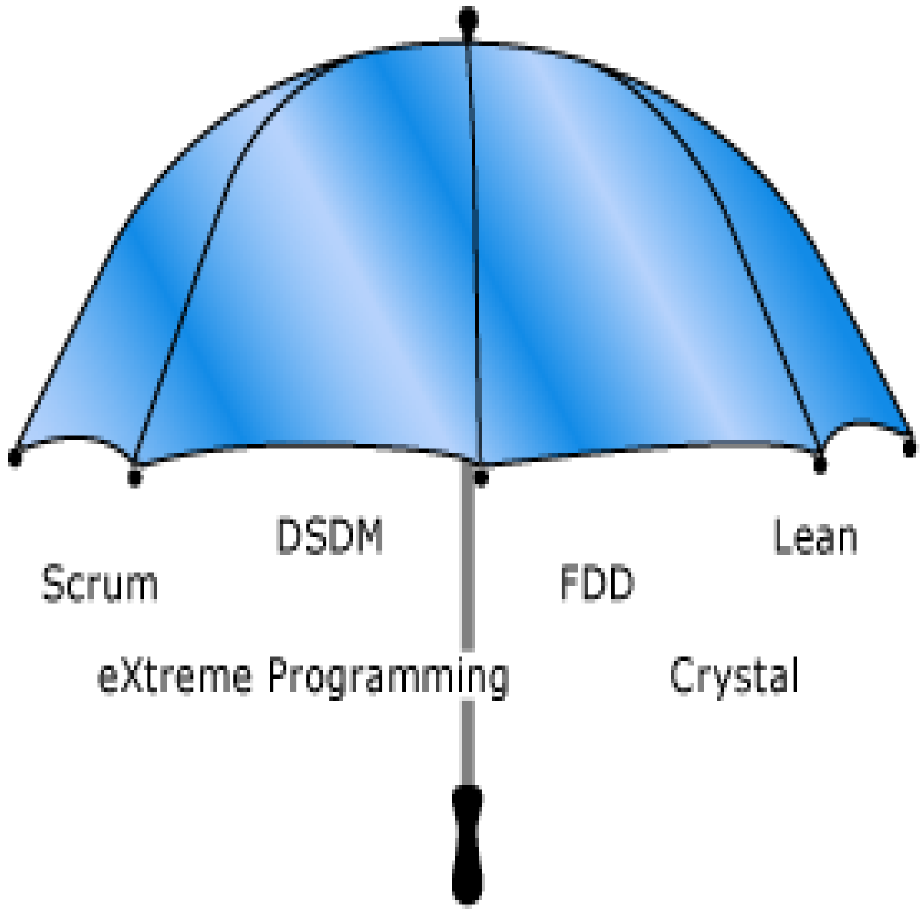
Rational Unified Process (RUP)

- Work in RUP in each phase is organized into **disciplines**
  - **Development disciplines**
    - Business Modelling & Requirements:
    - Understanding domain, develop high-level requirements, 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 and Implementation
    - Test: Testing throughout the project
    - Deployment: Product releases, software delivery
  - **Support disciplines**
    - Configuration and Change Management, Project Management, Environment
- RUP appears serial in the large (sequential phases), but iterative in small (work across all disciplines in each iteration)
- UP is not inherently documentation centric.

RUP: Serial in the large, iterative in the small



Agile Software Development Methodologies



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, close collaboration – not ideal for development outsourcing, clients and developers separated geographically, or business clients who simply don't have the manpower, resources
- Emphasis on modularity, incremental development, and adaptability – not suited to clients desiring contracts with firm estimates and schedule
- 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
- Agile development – Need highly experienced software engineers who know how to both 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
- Successful software development - 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

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 principles (high quality, simple design and continuous feedback) and practices to bring more flexibility to changes

XP Core Principles: High Quality (1)

- 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”.
- Continuous Integration - Programmers check their code in and integrate several times per day
- Sustainable pace - Moderate, steady pace
- Open Workspace – Open environment
- Refactoring – Series of tiny transformations to improve the structure of the system
- Test-Driven Development - Unit-testing and User Acceptance Testing

XP Core Principles: Simple Design (2)

- Focus 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

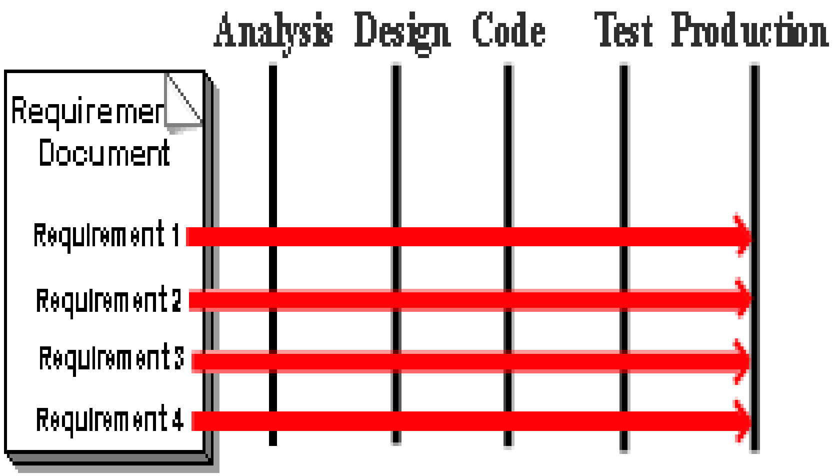
XP Core Principles: Continuous Feedback (3)

An XP team receives intense feedback in many ways, in many levels (developers, team and customer)

- Developers receive constant feedback by working in pairs, constant testing and continuous integration
- XP team receives daily feedback on progress and obstacles through daily stand-up meetings
- Customers get feedback on progress with user acceptance scores and demonstrations at the end of each iteration
- XP developers deliver value to the customer through producing working software progressively at a “steady heartbeat” and receive customer feedback and changes that are “gladly” accepted.

XP vs Waterfall Life-Cycle

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



Waterfall Life Cycle

XP Life Cycle

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

© http://www.agile-process.org/

What happens in XP Planning ?

Key steps in XP Planning Game:

- Initial Exploration
- Release Plan
- Iteration Planning
- Task Planning

The XP Planning Game: Initial Exploration

- Developer and customer have ***conversations*** about the system-to-be and identify significant features (not “all” features…customers will discover more)
- Each feature broken into one or more ***user stories***
- Developers *estimate* the user story in *user story point*s based on team’s velocity (becomes more accurate through iterations)
  - Stories that are too large or too small are difficult to estimate. An ***epic*** story should be split into pieces that aren't too big.
  - Developers complete a certain number of stories each week. Sum of the estimates of the completed stories is a metric known as ***velocity***
  - Developers have a more accurate idea of ***average velocity*** after 3 or 4 weeks, which is used to provide better estimates for ongoing iterations.

The XP Planning Game: Release Plan

- Negotiate a **release date** (6 or 12 or 24 months in the future)
  - Customers specify which user stories are needed and the order for the planned date (business decisions)
  - Customers can't choose more user stories than will fit according to the current project velocity
  - Selection is crude, as initial velocity is inaccurate. RP can be adjusted as velocity becomes more accurate
- Use the project velocity to plan
  - by **time: compute** #user stories that can be implemented before a given date (multiply number of iterations by the project velocity
  - by **scope:** how long a set of stories will take to finish divide the total weeks of estimated user stories by the project velocity

The XP Planning Game: Iteration Planning

- Use the release plan to create **iteration plans**
- Developers and customers choose an iteration size: typically 1 or 2 weeks
- Customers prioritise user stories from the release plan in the first iteration, but must fit the current velocity
- Customers cannot change the stories in the iteration once it has begun (can change or reorder any story in the project except the ones in the current iteration
- The iteration ends on the specified date, even if all the stories aren't **done**. Estimates for all the completed stories are totalled, and velocity for that iteration is calculated

  - *The planned velocity for each iteration is the measured velocity of the previous iteration.*
- Defining “done” - **A story is not done until *all* its acceptance tests pass**

The XP Planning Game: Task Planning

- Developers and customers arrange an **iteration planning meeting** at the beginning of each iteration
  - Customers choose user stories for the iteration from the release plan but **must** fit the current project velocity
  - User stories are broken down into programming tasks and order of implementation of user stories within the iteration is determined ***(technical decision)***
  - Developers may sign up for any kind of tasks and then estimate how long task will take to complete (*developer's budget – from previous iteration experience*)
  - Each task estimated as 1, 2, 3 (or even ½) days of ideal programming days. Tasks < 1 day grouped together, tasks > 3 days broken down
  - Project velocity is used again to determine if the iteration is over-booked or not
  - Time estimates in ideal programming days of the tasks are **summed** up, and this must not exceed the project velocity (initial or from the last iteration).
    - *If the iteration has too much - the customer must choose user stories to be put off until a later iteration (snow plowing). If the iteration has too little then another story can be accepted.*
  - Team holds a meeting halfway through iteration to track progress

Project Velocity

- Size points assigned to each user story
- Total work size estimate:
  - Total size = Σ (points-for-story ***i***), *i* = 1..N
- Estimate project velocity (= team's productivity) - estimated from the number of user-story points that the team can complete in a particular iteration (enables customers to obtain an idea of the cost of each story, its business value and priority)

  (e.g., if 42 points' worth of stories are completed during the previous week, the velocity is 42)
- Estimate the project duration
  - Project duration = Total Work Size / Project Velocity

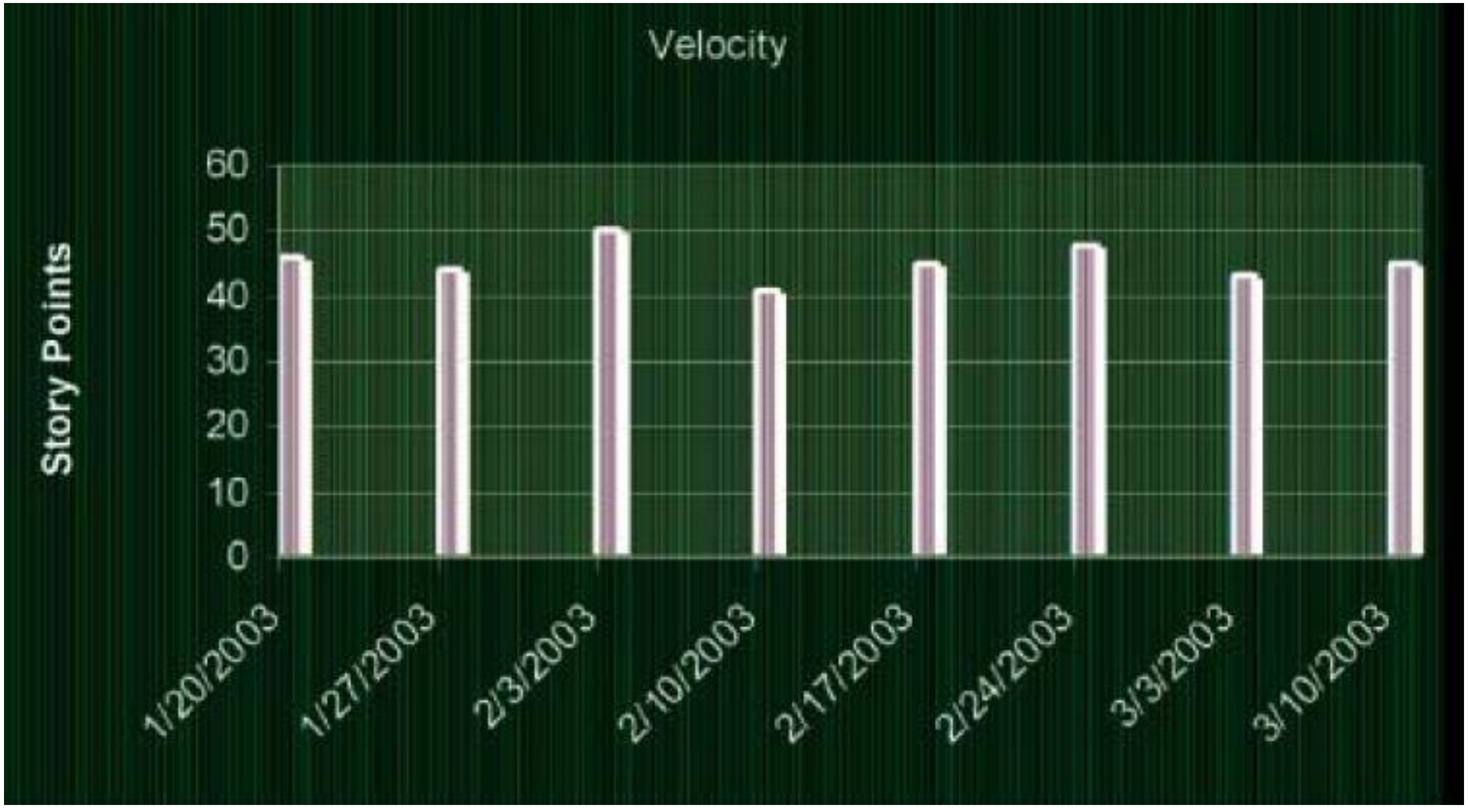
Project Tracking

The 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 or burn-down chart to track the project velocity which shows how many story points were completed (passed the user acceptance tests)
- These tools provide a reliable project management information for XP teams

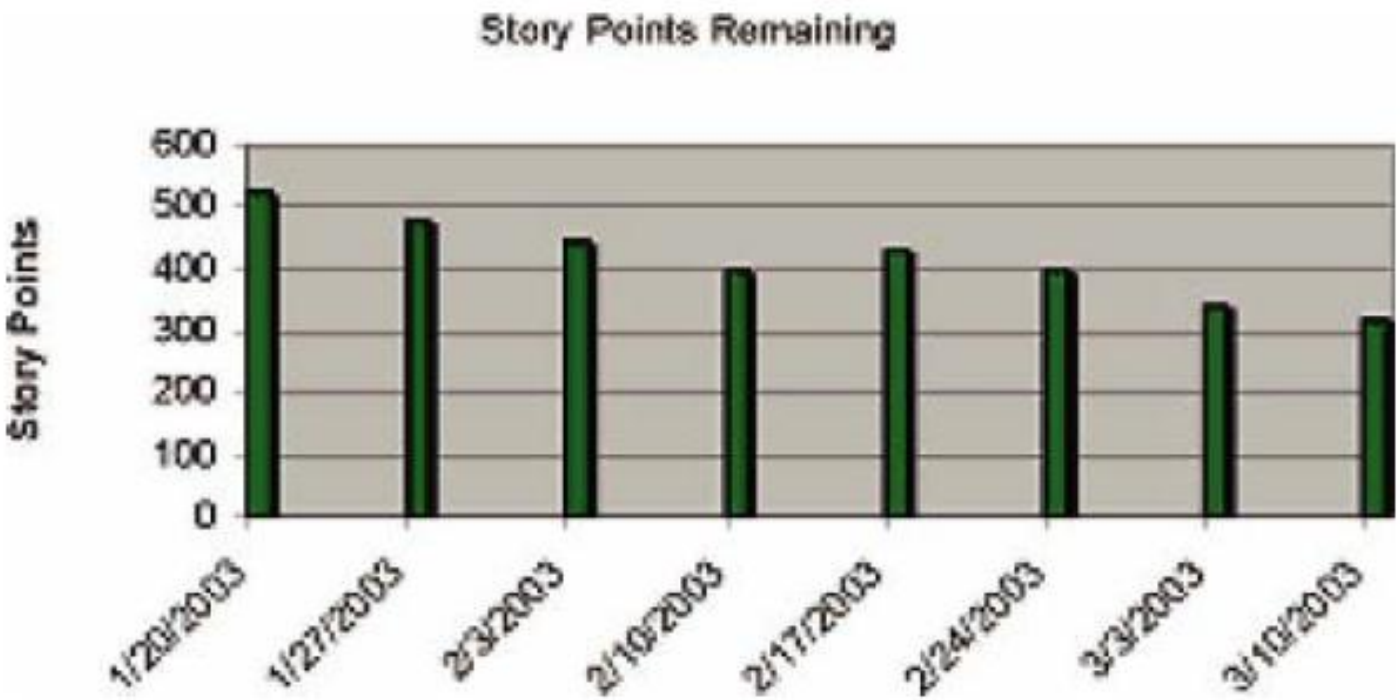
Project Tracking using Velocity Chart

- 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 Using Burn-Down Chart

- A burn-down chart shows 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)



When should XP be used?

- Useful for problem domains where requirements change, when customers do not have a firm idea of what they want
- 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 – Part I

What is a requirement?

- *"A condition or capability needed by a user to solve a problem or achieve an objective"* [IEEE]

- Simply stated, a requirement is a statement; a short, concise piece of information that:
  - describes an aspect of what the proposed system should do or describe a constraint
  - must help solve the customer's problem
- Set of requirements as a whole represents a negotiated agreement among all stakeholders

Functional vs Non-Functional requirements

Functional Requirements:

Defines the specific functionality that the software system is expected to accomplish i.e the *services* provided by the system and is typically described as:

- What inputs the system should accept and under what conditions
- The behaviour of the system
- What outputs the system must produce and under what conditions

Non-Functional Requirements:

Describe the *quality attributes* of the software system

- The constraints of the functionality provided by the software-system e.g. security, reliability, maintainability, efficiency, portability, scalability

Metrics for Non-Functional requirements

Non-functional requirements are quantifiable and must have a measurable way to assess if the requirement is met (metrics)

- Performance (user response time or network latency measured in seconds, transaction rate (#transactions/sec)
- Reliability (MTBW – mean time between failures, downtime probability, failure rate, availability)
- Usability (training time, number of clicks)
- Portability (% of non-portable code)

Requirements Engineering

Key task of requirements engineering:

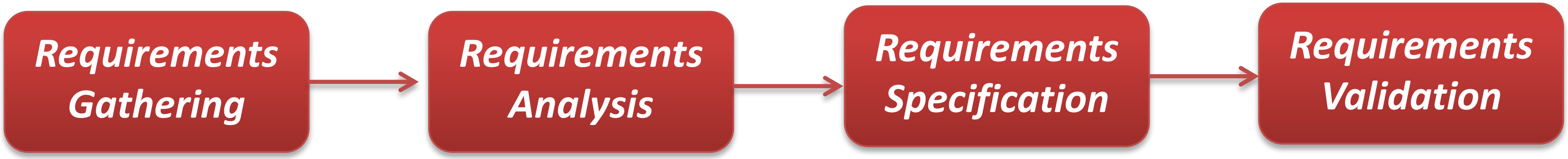
- Formulate a *well-defined problem* to solve
- A well-defined problem consists of:
  - a set of criteria (“requirements”) according to which proposed solutions either definitely solve the problem or fail to solve it
  - The description of the resources and components at disposal to solve the problem

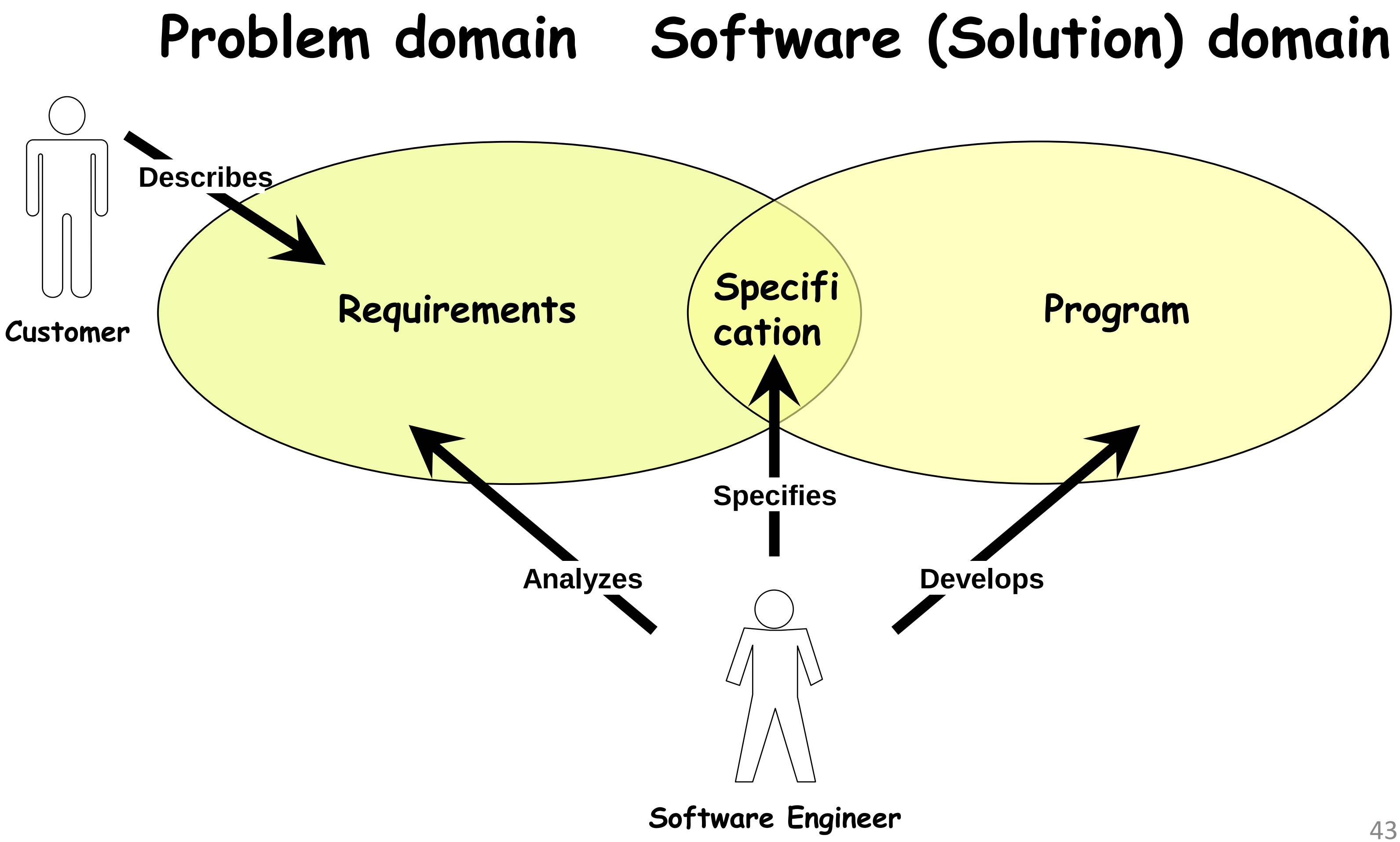
Who is involved in requirements engineering?

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

Phases of Requirements Engineering





Phases of Requirements Engineering (1)

Requirements Gathering: A process through which customers, buyers and end-users articulate, discover and understand their requirements

- Customers specify:
  - What is required?
  - How will the intended system fit into the context of their business
  - How the system will be used on a day-to-day basis?
- Developers to understand the business context through meetings (what if this? What if that?), market analysis, questionnaires etc.
- The problem statement is rarely precise

Phases of Requirements Engineering (2)

Requirements Analysis

- Refining and reasoning about the requirements elicited
- Scope the project, negotiate with the customer to determine the priorities - what is important, what is realistic
- Identify dependencies, conflicts and risk
- Apply user-case modelling – elaborate user-scenarios that describe interaction of user with the system; to ensure developer's understanding of the problem matches the customer's expectations or develop user-stories (Agile requirements analysis)

Phases of Requirements Engineering (2)

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

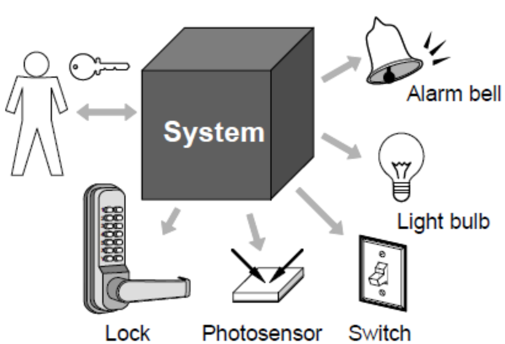
  *(**Use-cases, User-stories, prototypes, formal mathematical models or a combination of these… OR a formal SRS (System Requirements Specification)***

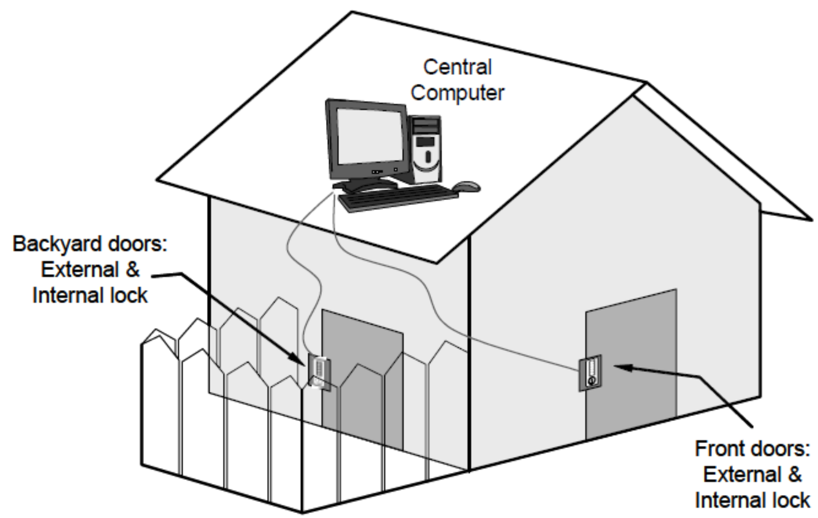
Requirements Validation

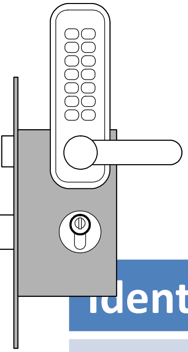
- The process of confirming with the customer or user of the software that the specified requirements are valid, correct, and complete

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 instructed otherwise by an authorised 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) and lights turned off
REQ2	4	The system shall lock the door when commanded by pressing a dedicated button and shut the lights
REQ3	5	The system shall, given a valid key code unlock the door and turn light on
REQ4	3	The system shall permit three failed attempts. However, to resist “dictionary attacks”, after the allowable number of failed attempts, the system will block and an alarm is activated
REQ5	1	The system shall maintain a history log of all attempted accesses for later review
REQ6	2	The system should allow adding new authorised users or removing existing users at run-time

Granularity of requirements

- Some of the requirements in our previous table are relatively complex or compound requirements. Consider REQ2:
  - *The system shall keep the door locked at all times, unless instructed otherwise by an authorised 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)*
- Suppose, testing this requirement fails (the door was found unlocked when it should have been locked)
  - *Did the system accidentally disarm the lock*
  - *Did the auto-lock feature fail?*

(Difficult to tell)

Granularity of requirements

- TDD (Test-driven-development) stipulates writing requirements such that they can be individually tested

- REQ1 can be split into:
  - REQ1a: *The system shall keep the doors locked at all times, unless commanded otherwise by authorized user.*
  - REQ1b: *When the lock is disarmed, a countdown shall be initiated at the end of which the lock shall be automatically armed (if still disarmed).*

- Choice of granularity is subject to judgment and experience

Test-Driven Development

- Write User Acceptance Tests (UAT) for requirements during the requirements analysis:
  - capture the customer's assumptions about how the requirement will work and what could go wrong
  - are defined by the customer or developer in collaboration with the customer
- For example, test-cases for REQ1:
  - *Test with the valid key of a current tenant on his or her apartment (pass)*
  - *Test with the valid key of a current tenant on someone else's apartment (fail)*
  - *Test with an invalid key on any apartment (fail)*
  - *Test with the key of a removed tenant on his or her previous apartment (fail)*
  - *Test with the valid key of a just-added tenant on his or her apartment (pass)*

What is UML?

UML stands for **Unified Modelling Language** (http://www.uml.org/)

Programming languages not abstract enough for OO design

An open source, graphical language to model software solutions, application structures, system behaviour and business processes

Several uses:

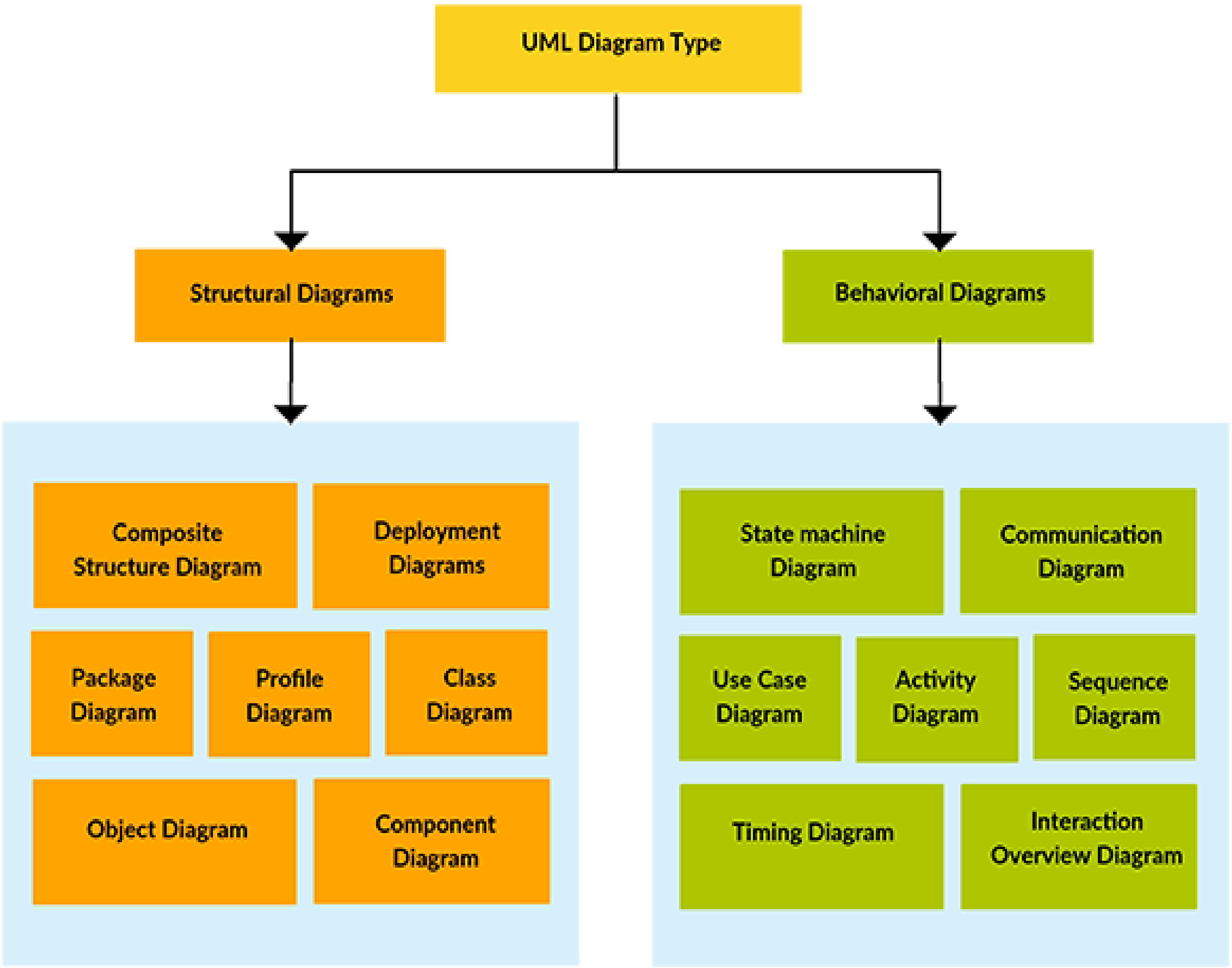
- As a design that communicates aspects of your system
- As a software blue print
- Sometimes, used for auto code-generation

UML diagram categories

- **Structure Diagrams**
  - show the static structure of the system and its parts and how these parts relate to each other
  - they are said to be static as the elements are depicted irrespective of time (e.g. class diagram)

- **Behaviour Diagrams**
  - show the dynamic behaviour of the objects in the system i.e. a series of changes to the system over a period of time (e.g. use case diagram or sequence diagram)
  - a subset of these diagrams are referred to as interaction diagrams that emphasis interaction between objects (e.g., an activity diagram)

UML Diagram Types



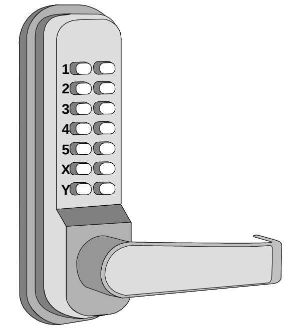
UML Use-Case Diagrams

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
  - Written as *usage scenarios* visualizing a 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



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)**

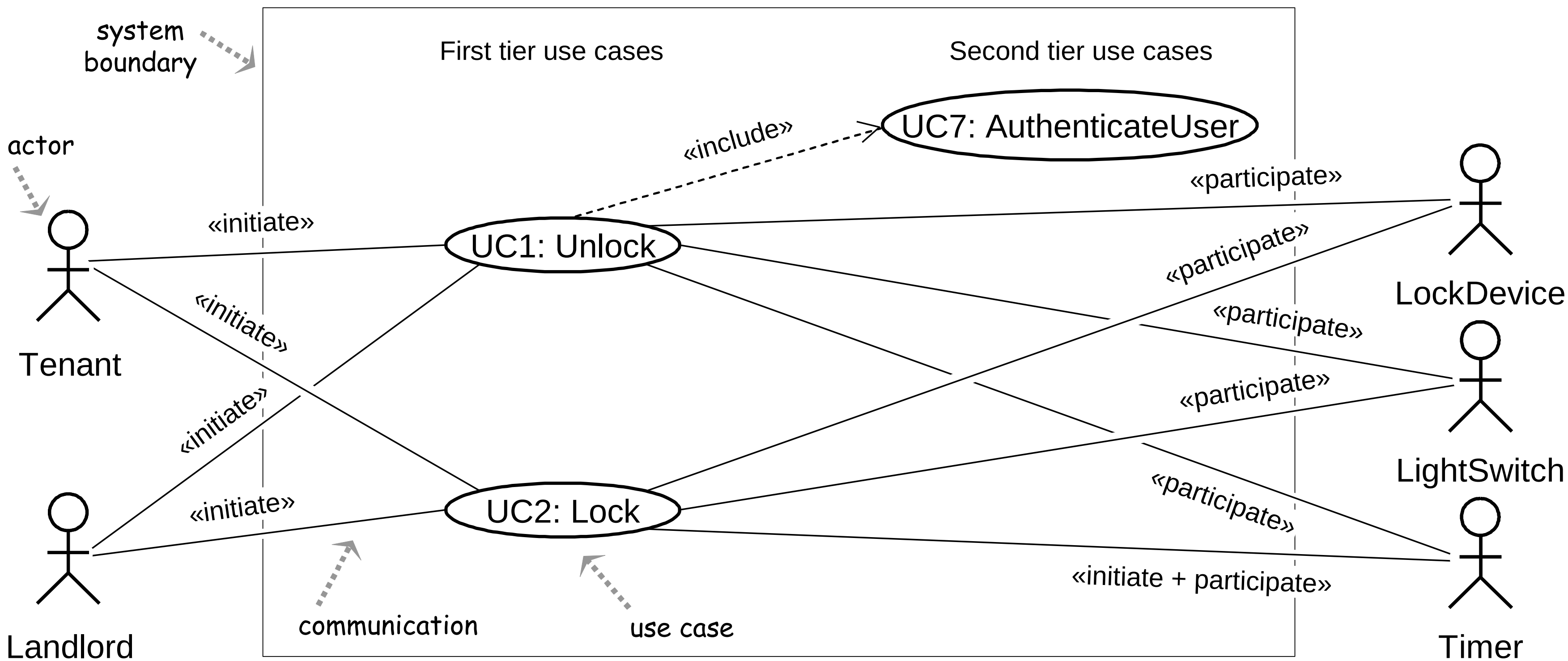
Types of Actors

- ***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.
  - helps the system-to-be to complete the use case

Use Case Diagram: Device Control

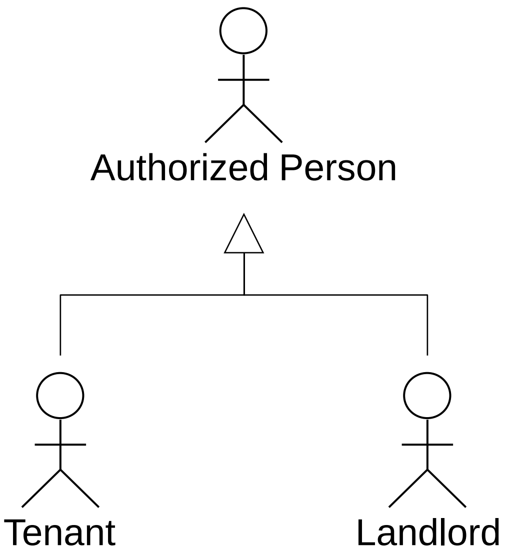
UC1: Unlock
UC2: Lock
UC3: AddUser
UC4: RemoveUser
UC5: InspectAccessHistory
UC6: SetDevicePrefs
UC7: AuthenticateUser
UC8: Login



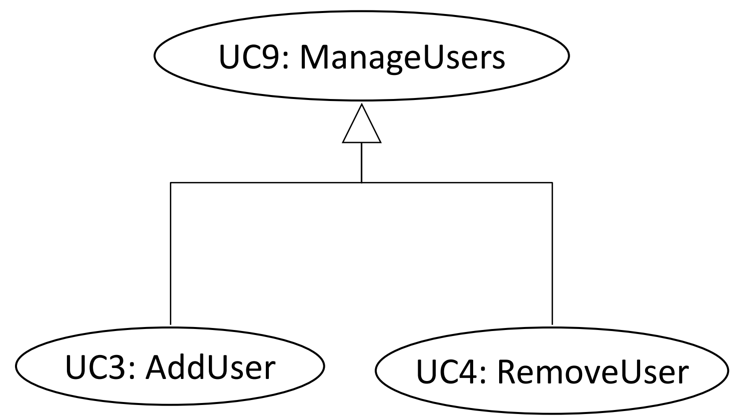
Use Case Generalizations

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



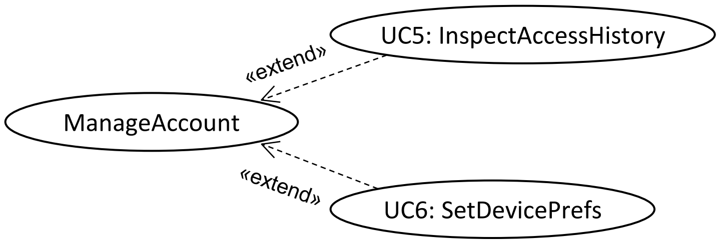
Actor Generalization



Use Case Generalization

Optional Use Cases: «extend»

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



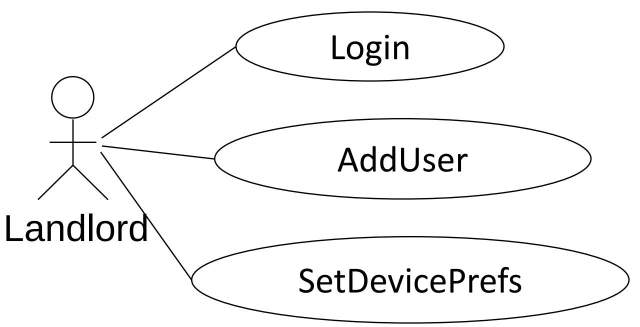
Key differences between **«include»** and **«extend»** relationships

	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

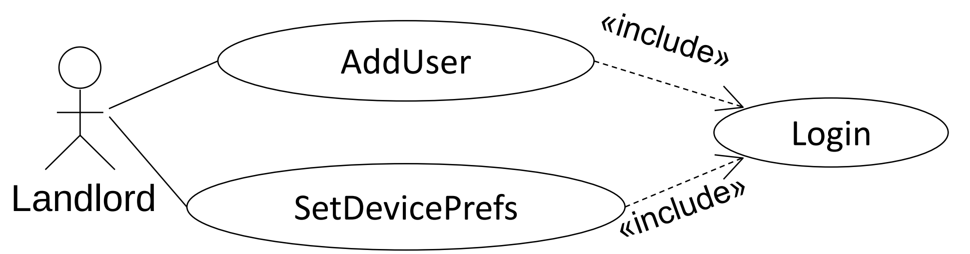
[Source: Robert Maksimchuk & Eric Naiburg: ***UML for Mere Mortals***, Addison-Wesley, 2005.]

Login Use Case?

BAD:



GOOD:



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

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, …

Traceability Matrix Purpose

- **Traceability** refers to the property of a software artefact (use-case, a class etc) of being *traceable* to the original requirement that motivated its existence
- Traceability matrices are continued through the domain model, design diagrams etc…
- In the context of use-cases, the matrix serves to:
  - To check that all requirements are covered by the use cases
  - To check that none of the use cases is introduced without a reason (i.e., created not in response to any requirement)
  - To prioritize the work on use cases

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	

Acceptance Test Case for UC-7 Authenticate User

Test-case Identifier:	TC-1
Use Case Tested:	UC-1, main success scenario, and UC-7
Pass/fail Criteria:	The test passes if the user enters a key that is contained in the database, with less than a maximum allowed number of unsuccessful attempts
Input Data:	Numeric keycode, door identifier
Test Procedure:	**Expected Result:**
Step 1. Type in an incorrect keycode and a valid door identifier	System beeps to indicate failure; records unsuccessful attempt in the database; prompts the user to try again
Step 2. Type in the correct keycode and door identifier	System flashes a green light to indicate success; records successful access in the database; disarms the lock device

Use Case 2: Lock

Use Case UC-2:	**Lock**
Related Requirements:	REQ1, REQ2, and REQ5 stated in Table 2-1
Initiating Actor:	Any of: Tenant, Landlord, or Timer
Actor's Goal:	To lock the door & get the lights shut automatically (?)
Participating Actors:	LockDevice, LightSwitch, Timer
Preconditions:	The system always displays the menu of available functions.
Postconditions:	The door is closed and lock armed & the auto-lock timer is reset.

Flow of Events for Main Success Scenario:

→	1.	**Tenant**/**Landlord** selects the menu item "Lock"
←	2.	**System** (a) signals affirmation, e.g., "lock armed," (b) signals to **LockDevice** to arm the lock (if not already armed), (c) signal to **Timer** to reset the auto-lock counter, and (d) signals to **LightSwitch** to turn the light off (?)

Flow of Events for Extensions (Alternate Scenarios):

2a. System senses that the door is not closed, so the lock cannot be armed

←	1.	**System** (a) signals a warning that the door is open, and (b) signal to **Timer** to start the alarm counter
→	2.	**Tenant**/**Landlord** closes the door
←	3.	**System** (a) senses the closure, (b) signals affirmation to the **Tenant**/**Landlord**, (c) signals to **LockDevice** to arm the lock, (d) signal to **Timer** to reset the auto-lock counter, and (e) signal to **Timer** to reset the alarm counter

Requirements Engineering - II

Agile Requirements Analysis and Specification using User-Stories

User Story

- One of the primary development artifacts in XP and Scrum
- A short, simple descriptions of a feature or requirement narrated from the perspective of the person who desires that capability
  - Initial User Story (informal)

    e.g., Student can purchase monthly parking passes online
  - Initial User Story (formal): uses a RGB (Role-Goal-Benefit) template

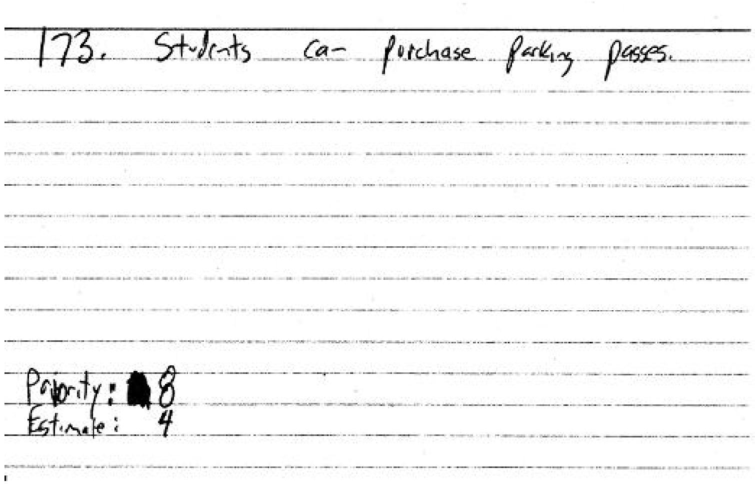
    As a < type of user >, I want < some goal > so that < some reason >

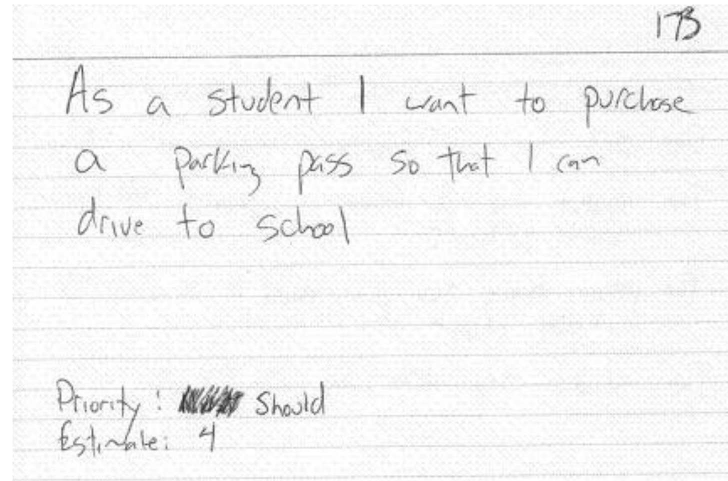
    e.g., As a student, I want to purchase a parking pass so that I can drive to school
- A reminder to have a conversation with your customer
- Anyone can write user stories (team member, and product owner's responsibility to make sure a **product backlog of agile user** stories exist
- Use the simplest tool - Index cards or sticky notes

User Story

- **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 parking pass OR print student transcripts online
- **User Stories** - split an epic user story into multiple smaller atomic stories
  - 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
- **Themes** – A collection of related epic user-stories
  - e.g., possible themes for a university registration system - student enrolment, course management, transcript generation

Important considerations for User Stories

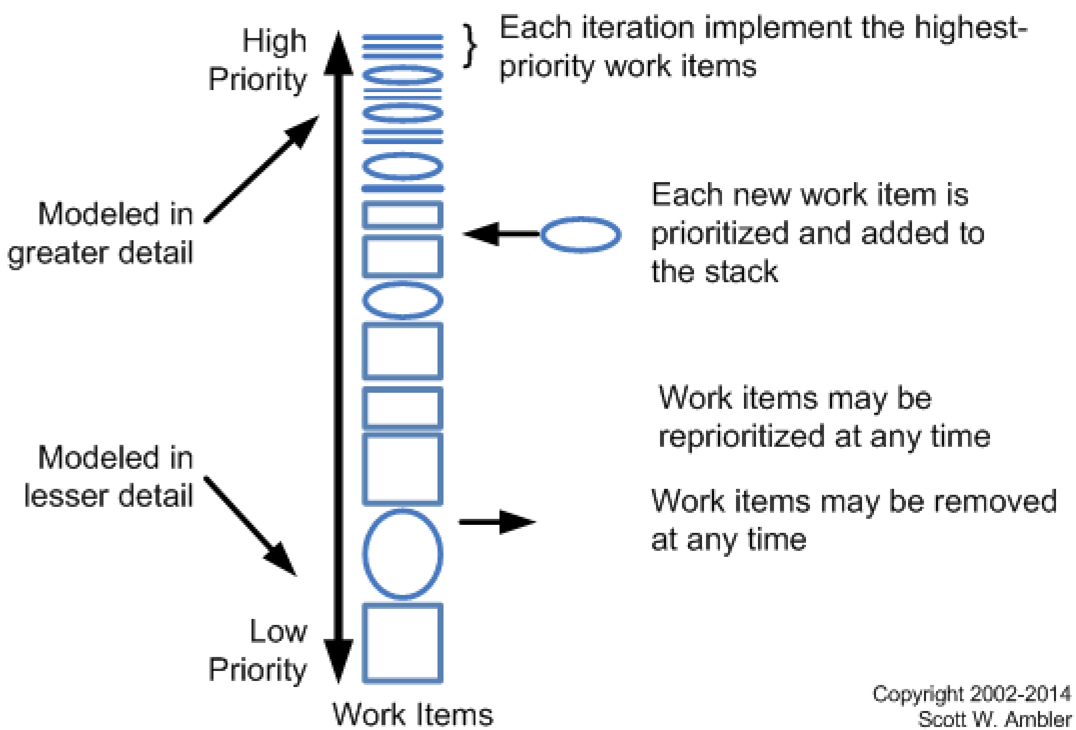




- Assign each user-story a **unique identifier** e.g., US – 12
- Remember non-functional requirements (e.g., the *Students can purchase parking passes online* user story is a usage requirement similar to a use case whereas the *Transcripts will be available online via a standard browser* is closer to a *technical requirement*)
- **Indicate the estimated size** - 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, user story (above) will take around 10 hours)
- **Indicate the priority** (e.g., on a scale of 1 – 10)

User Story and Planning

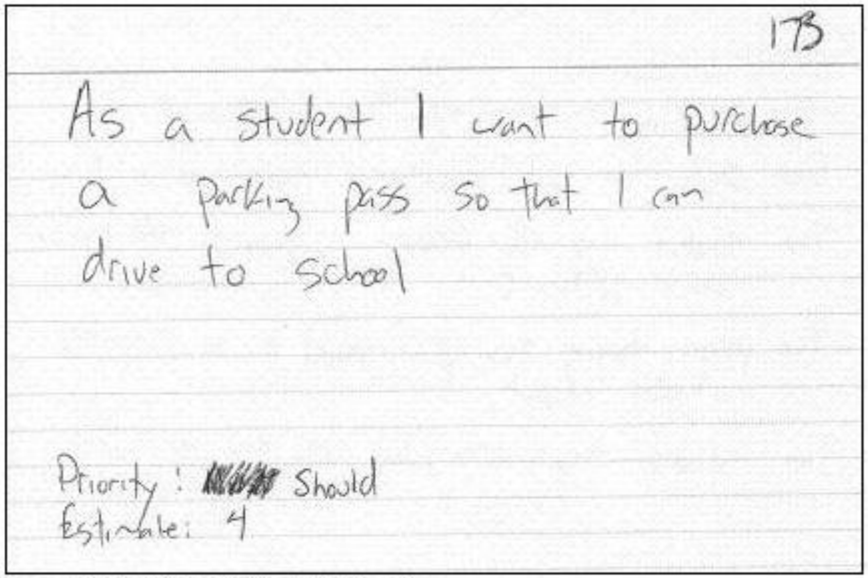
- User-stories affect the planning process in two key areas:
  - Scheduling
  - Estimating



Detailing a user-story

- User-story with confirmations

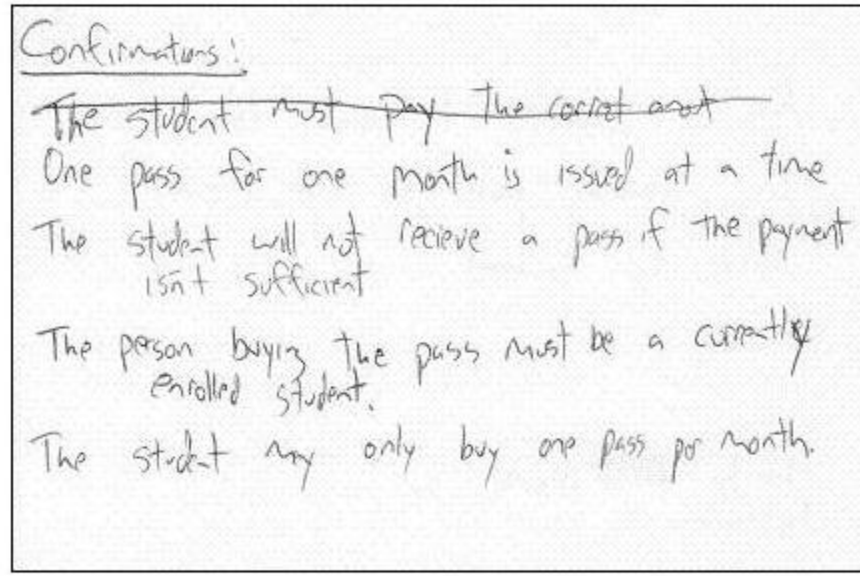
Front of Card


173

As a student I want to purchase a parking pass so that I can drive to school

Priority: Should
Estimate: 4

Back of Card


Confirmations:

~~The student must pay the correct amount~~

One pass for one month is issued at a time

The student will not recieve a pass if the payment isn't sufficient

The person buying the pass must be a currently enrolled student.

The student may only buy one pass per month.

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"

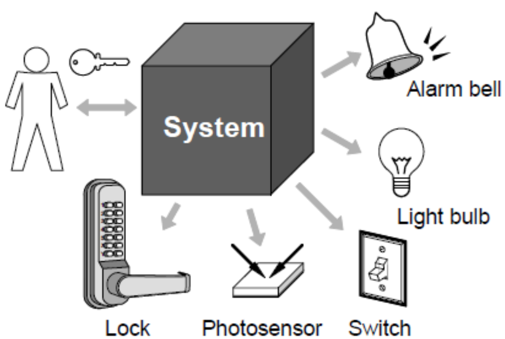
  – **The reason (why):** states why the user wants this feature. What benefit the user will get out of this feature? e.g. reason = "improve customer service", "find an offering that interests me". (If the value or benefit can't be articulated, it might be something that's not necessary)

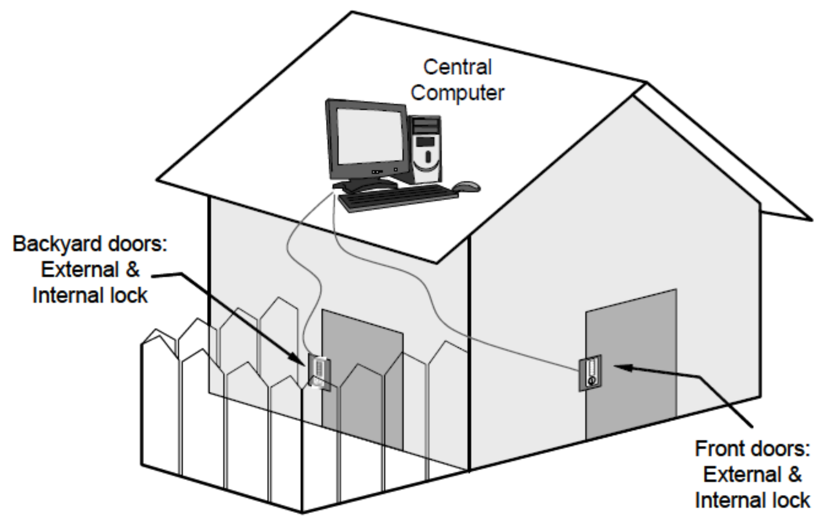
What is a good User Story ?

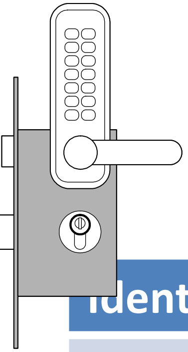
- **INVEST –** an acronym that helps evaluate whether you have a high quality user story
  - I = *Independent*: user story could be developed independently and delivered separately
  - N = *Negotiable*: avoid, too much detail. user story should be discussable further, keep them flexible
  - V = *Valuable*: 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 (i.e., designed, coded and tested within the iteration)
  - T = *Testable*: user story should be defined with clear *acceptance criteria*, both the correct functionality and the error conditions which leads to test-cases

Revisiting - 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 instructed otherwise by an authorised 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) and lights turned off
REQ2	4	The system shall lock the door when commanded by pressing a dedicated button and shut the lights
REQ3	5	The system shall, given a valid key code unlock the door and turn light on
REQ4	3	The system shall permit three failed attempts. However, to resist “dictionary attacks”, after the allowable number of failed attempts, the system will block and an alarm is activated
REQ5	1	The system shall maintain a history log of all attempted accesses for later review
REQ6	2	The system should allow adding new authorised users or removing existing users at run-time

User Stories For Home Access Control

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

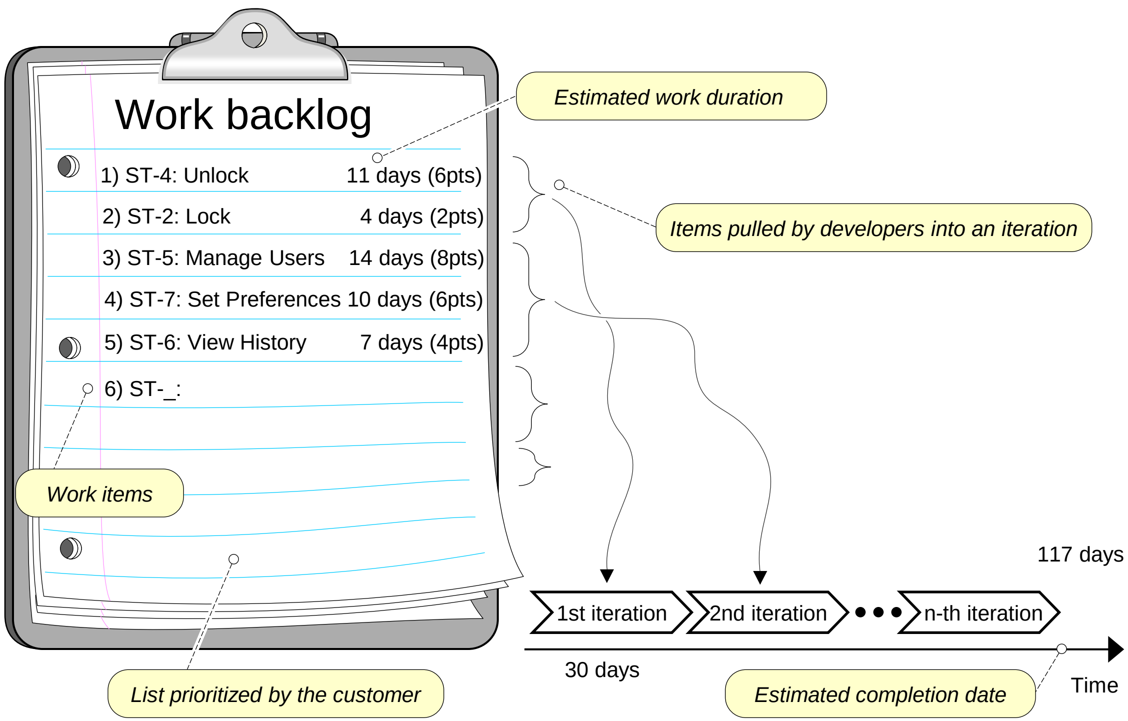
user-role (benefactor) | capability | business-value

- 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**

Example of Agile Estimation of Project Effort for case-study



Agile Prioritization of Work