COMP 1531 Software Engineering Fundamentals

Summary of Week 01 and Week 06 lectures

Course Mark

- Course Work Mark = Lab + Quiz + Group Project (out of 40)
- Mid-Term-Exam-Mark = Mark from mid-term exam(out of 10)
- Final-Exam-Mark = Mark from the 3 hour final exam (out of 50)
- **Exam_OK = (Mid-Term-Exam-Mark + Final-Exam-Mark) >= 30/60**
- **Final Course Mark** (out of 100) = Course Work Mark + Mid-Term-Exam-Mark + Final-Exam-Mark
- Final Grade
  - UF, if !ExamOK (even if final course mark > 50)
  - FL, if Final Course Mark < 50/100
  - PS, if 50/100 ≤ Final Course Mark < 65/100
  - CR, if 65/100 ≤ Final Course Mark < 75/100
  - DN, if 75/100 ≤ Final Course Mark < 85/100
  - HD, if Final Course Mark ≥ 85/100

Supplementary Exam

- Please consult the **UNSW Computing Supplementary Assessment Policy** at https://www.engineering.unsw.edu.au/computer-science-engineering/about-us/organisational-structure/student-services/policies/essential-advice-for-cse-students

- If you are granted a supplementary exam or if you think you are eligible for a supplementary exam, you must make sure that you are available on the scheduled day.

- There will be **only one** supplementary Exam, which is held in the period, scheduled by the exam unit.

COMP 1531 Software Engineering Fundamentals

Introduction to Software Engineering

What is software engineering?

“The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software.” [IEEE]

What is Software Engineering?

- “Software Engineering” is a discipline that enables customers to achieve **business goals** through *developing* software-based systems to solve their **business problems** e.g., develop a course enrolment application or a software to manage inventory

- This discipline places great emphasis on the ***methodology*** or the *method for managing the development process*

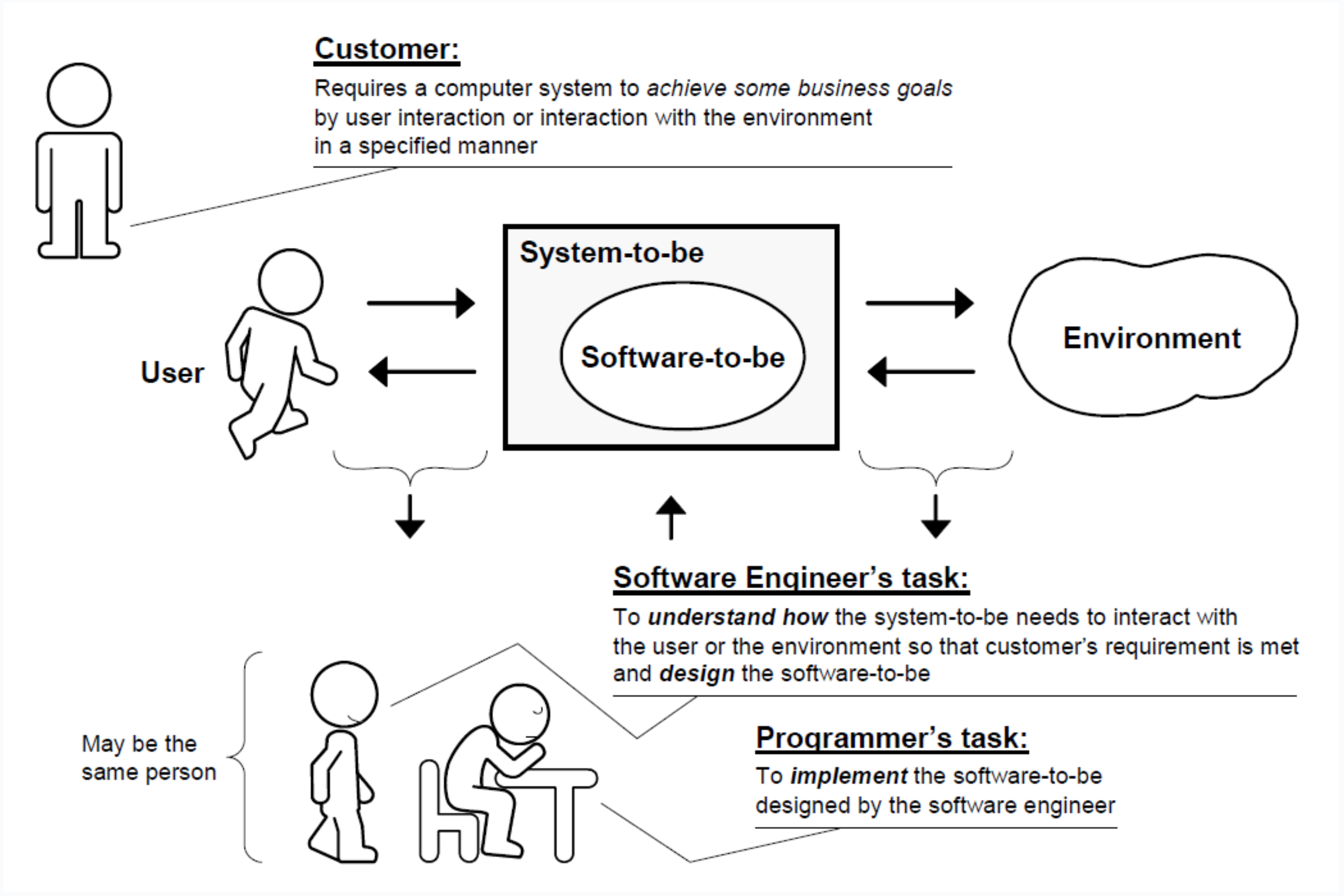
- The methodology is commonly referred to as **Software Development Life-Cycle (SDLC)**

Software Engineering is not Programming

- Software engineering:
  - **Understanding** the business problem (understanding the interaction between the system-to-be, 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:
  - **Implementing** the “blueprint” designed by the software engineer

Role of a software engineer

- Software engineer thus acts as a bridge from customer **needs** (problem domain) to programming **implementation** (solution domain)
- This enables the software engineer to design solutions that accurately target the customer's needs, that is, deliver **value** to the customer


Customer:
Requires a computer system to *achieve some business goals* by user interaction or interaction with the environment in a specified manner
System-to-be
Software-to-be
User
Environment
Software Engineer's task:
To ***understand how*** the system-to-be needs to interact with the user or the environment so that customer's requirement is met and ***design*** the software-to-be
Programmer's task:
To ***implement*** the software-to-be designed by the software engineer
May be the same person

Why do we need SE (1)?

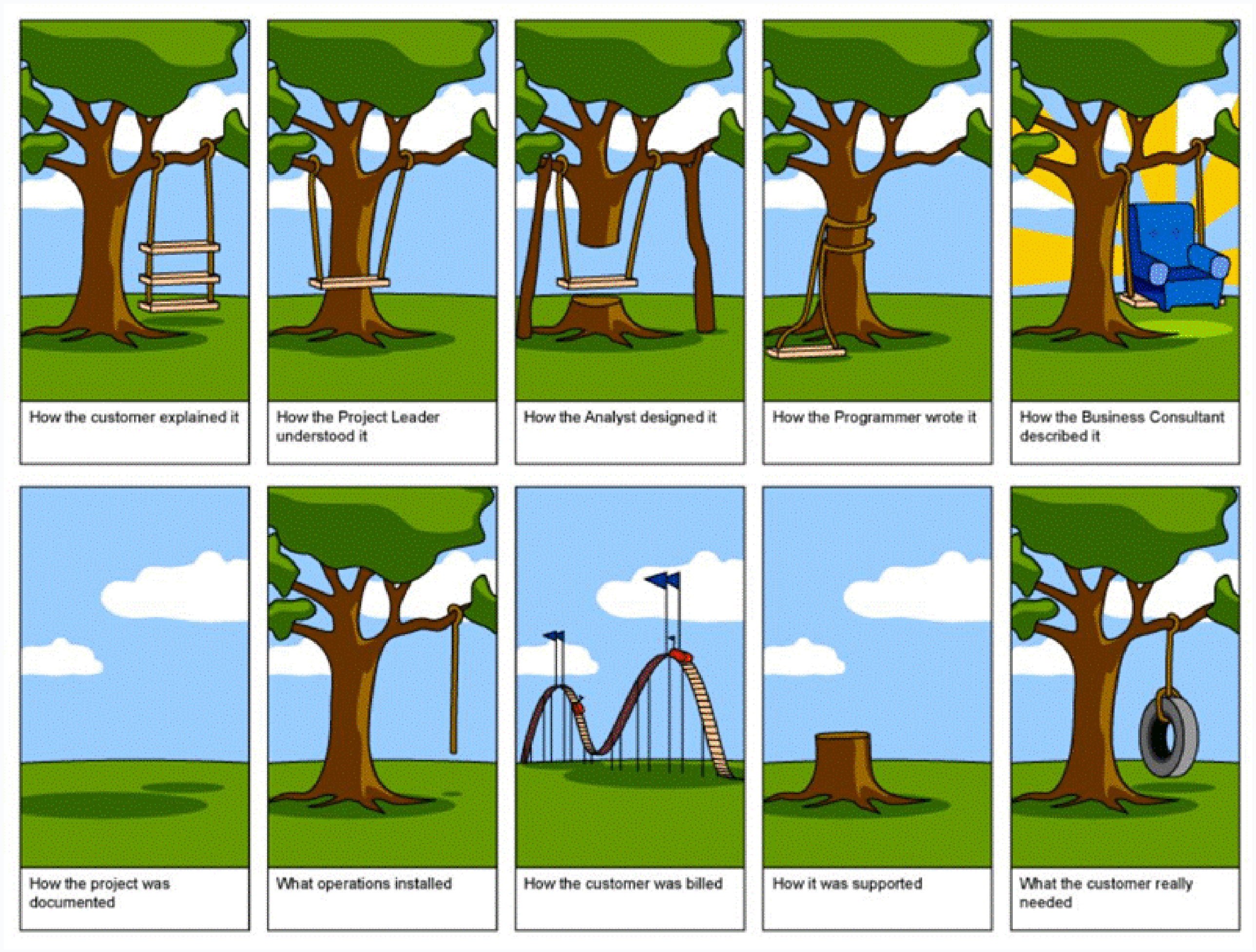
We want to…

- Make sure what we build is what the customer actually wanted
- Deliver the software on time and on budget
- Minimize defects
- Ensure reliability, security, performance, extensibility, usability, maintainability…

To do so, we need a systematic and disciplined approach to software development - and that's what software engineering is about

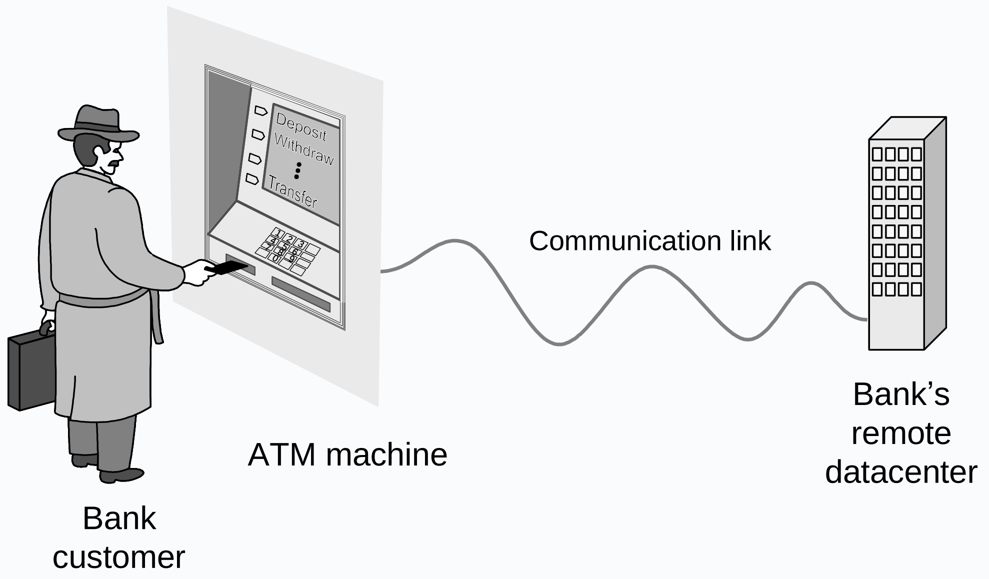
Why do we need SE (2)?

- Software development is a complex process, so building software requires a discipline, to ensure that the product delivered realises customer goals


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

Why do we need SE (3)?

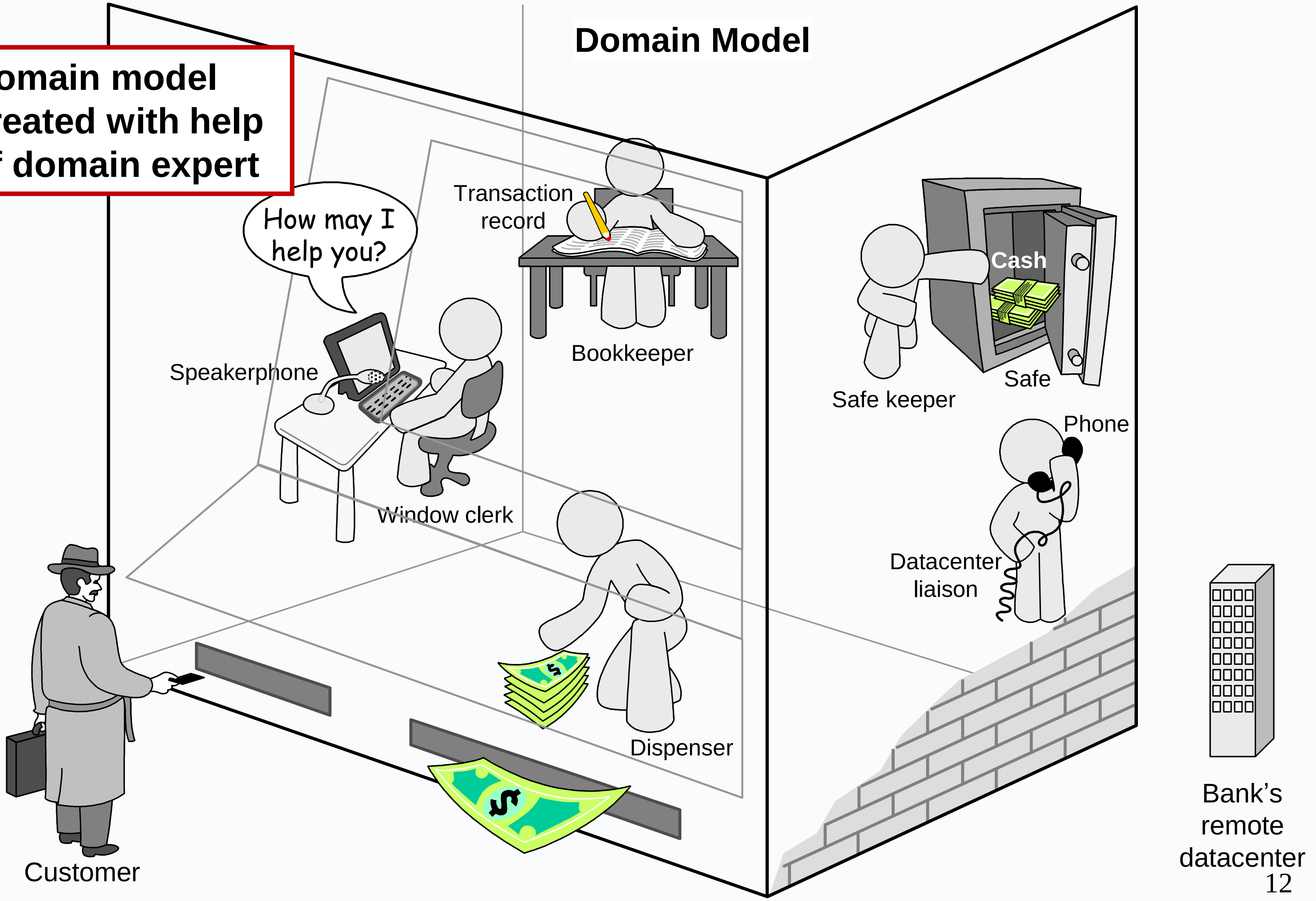
- Software is intangible and software development requires imagination e.g., the ATM Machine



  - Understand both the problem domain and software domain
  - Challenge: find a set of *good abstractions* that is representative of the problem domain and build a conceptual domain model
  - Creative formulation of solutions, design alternatives, trade-offs, difficult to come up with the "correct solution"

How ATM Machine Might Work

Domain model created with help of domain expert



Why do we need SE (4)?

- Software errors, poor design and inadequate testing have led to loss of time, money, human life
  - **Case of the Therac-25:** one of the most well-known killer software bugs in history
  - **Explosion of Ariane-501 in 1996:** blew up 37 seconds after initial launch, cost $7 billion and 10 years of development
  - Or just imagine… if Facebook accidentally leaked out your private information or Amazon started shipping products to wrong customer

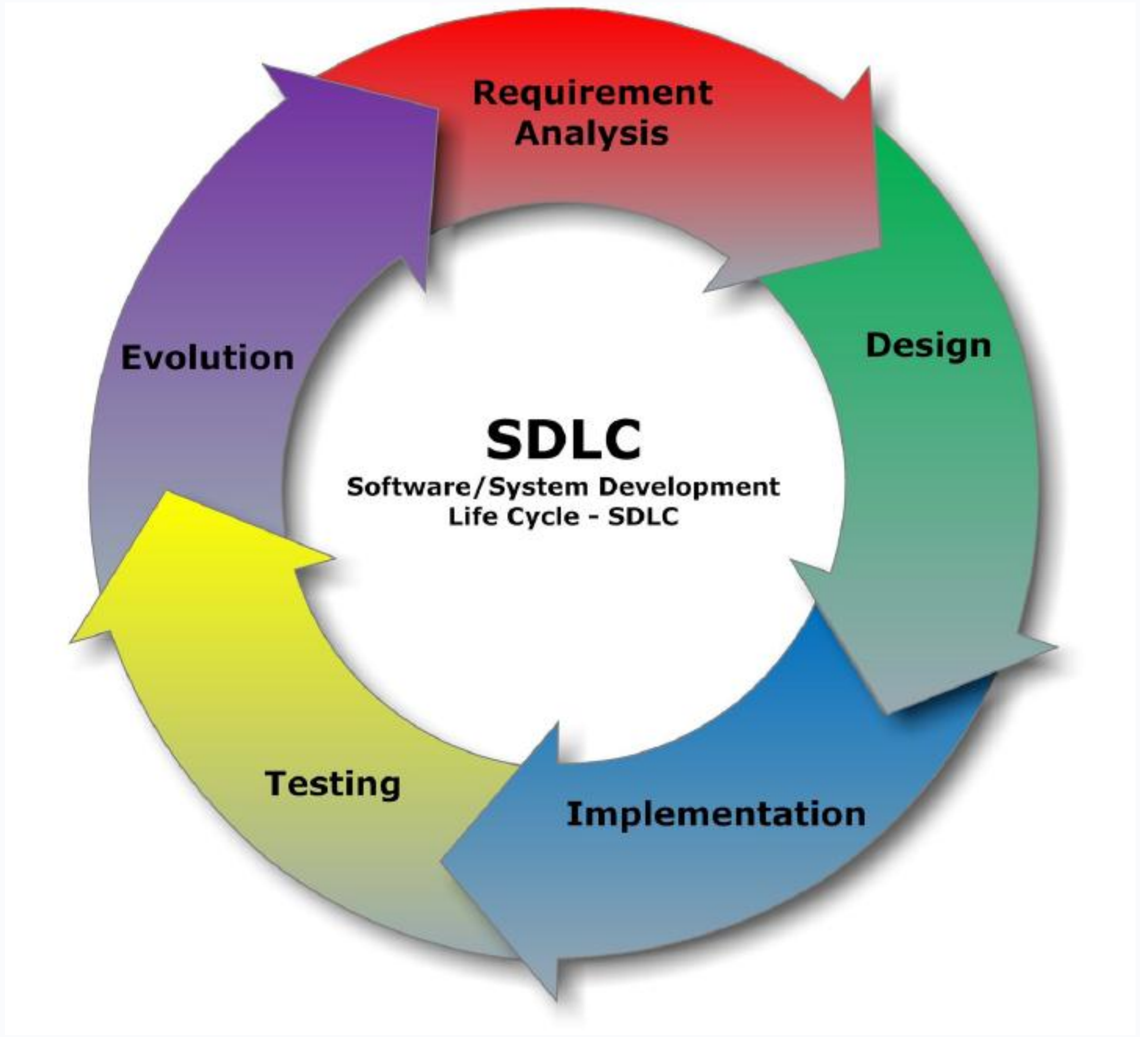
Checkpoint !

Quiz - Software Engineering Basics

Software Development Life-Cycle

- We described software engineering as a complex, organised process with a great emphasis on *methodology*
- This *methodology* is essentially a framework to structure, plan and control the development of the software system and typically consists of 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 phas

Software Development Life-Cycle



SDLC – Requirements Analysis

1. Analysis:

Discover and learn about the **problem domain** and the **“system-to-be”** where software engineers need to:

- Analyse the problem, understand the problem definition – what *features/services* does the system need to provide? ***(behavioural characteristics or external behaviour)***

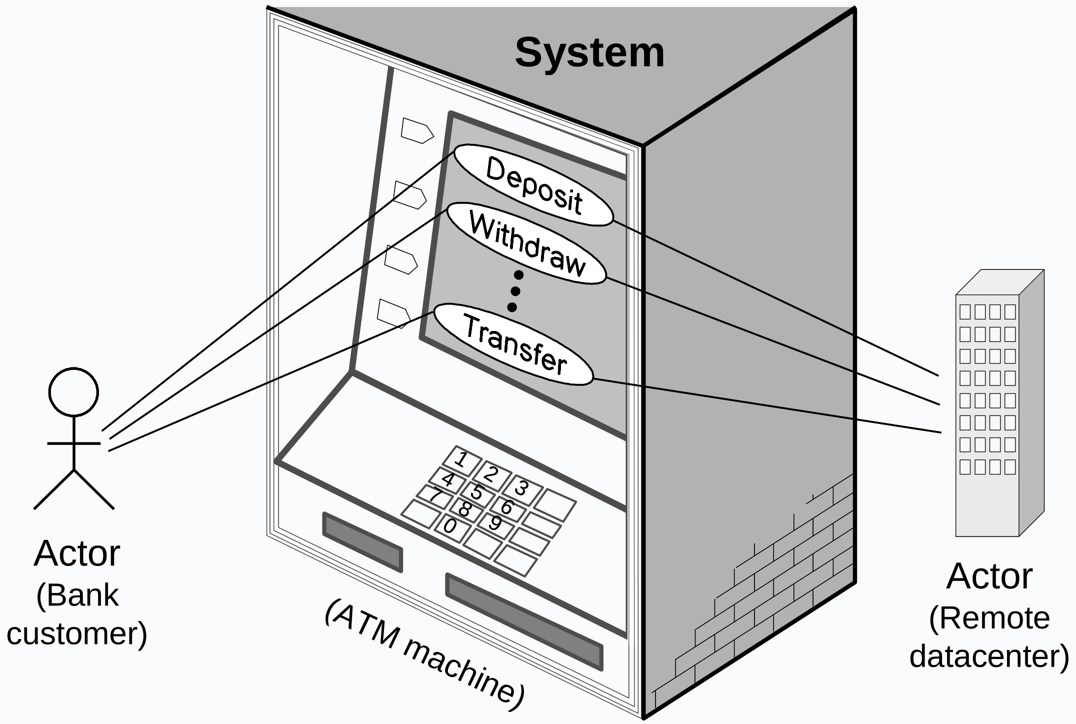
- Determine both functional (inputs and outputs) and non-functional requirements (performance, security, quality, maintainability, extensibility etc.)
- Use-case modelling, user-stories are popular techniques for analysing and documenting the customer requirements

SDLC – Design Phase

2. Design:

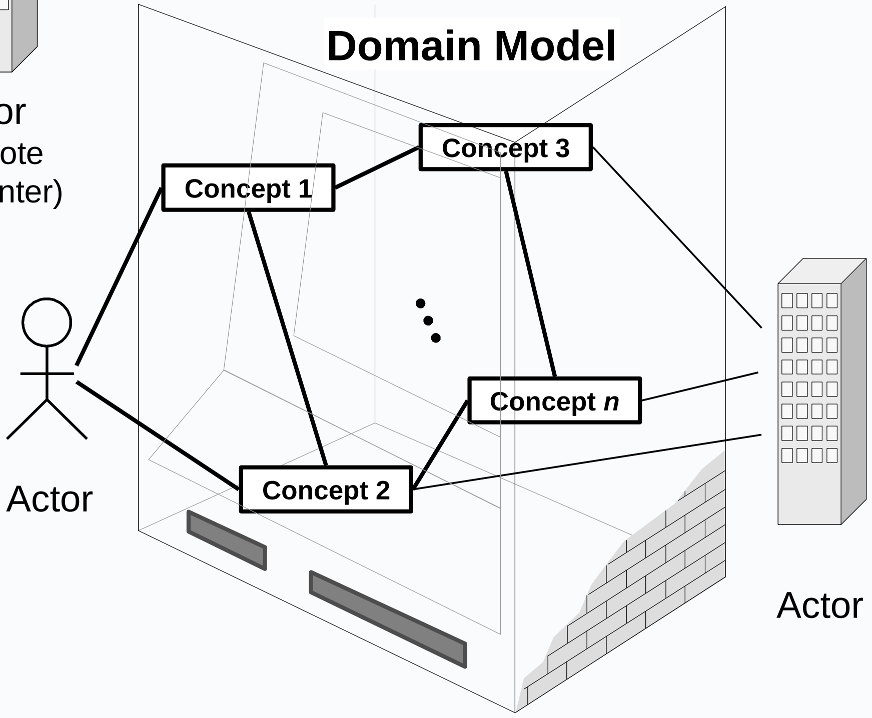
- A problem-solving activity that involves a "creative process of searching how to implement all of the customer's requirements"
- Plan out the system's "internal structure or structural characteristics" that delivers the system's "external behaviour" specified in the previous phase
- Produce software blue-prints (design artifacts e.g., domain model)
- Often the design phase overlaps with previous phase

Requirements Analysis vs Design



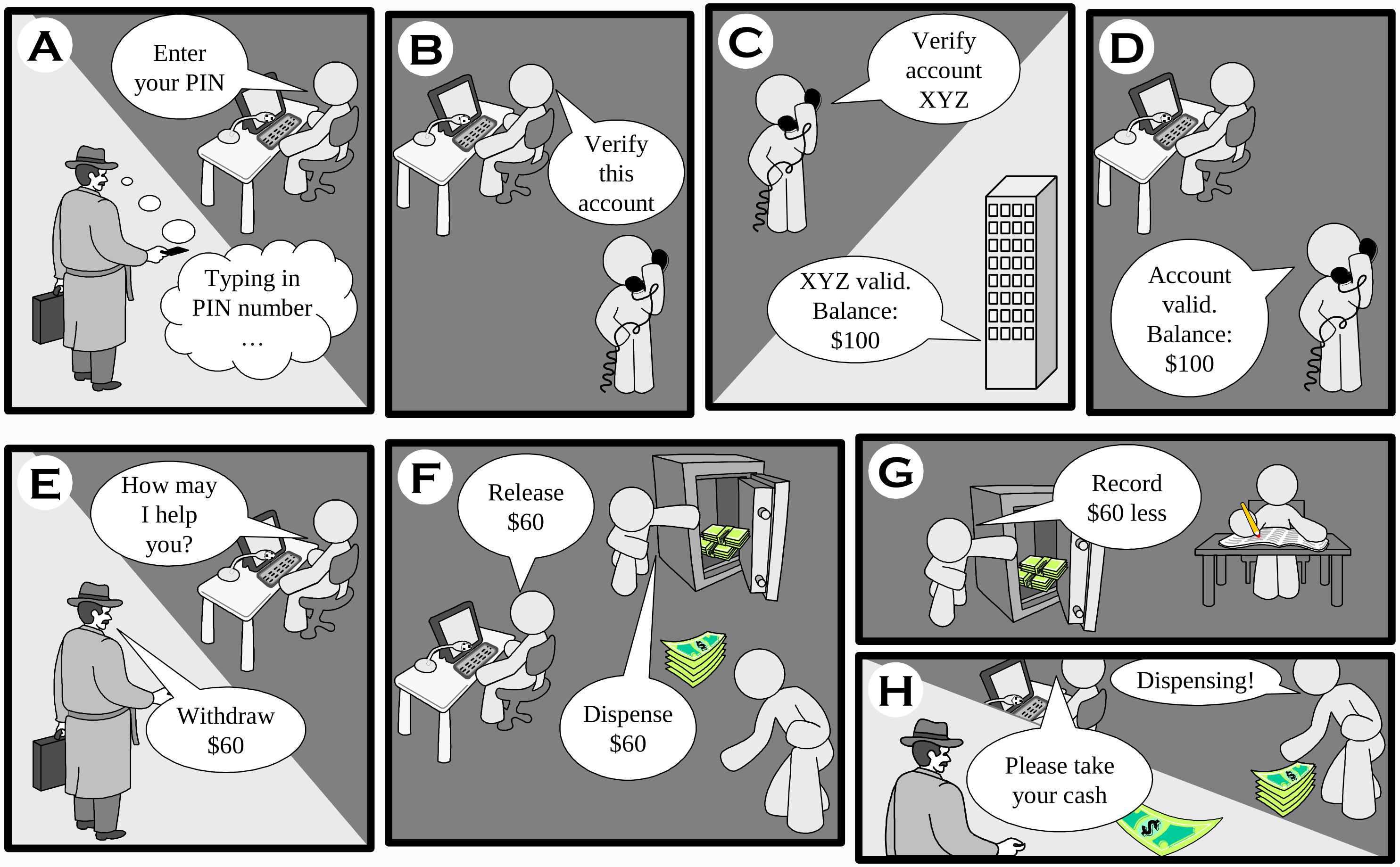
(a)

(b)



SDLC – Design Phase

In generating these "blue-prints"….
Cartoon Strip Writing OR more formal symbols (e.g., UML symbols – use-case diagram, class diagram, component diagram…?



Cartoon Strip: How ATM Machine Works

SDLC Phases

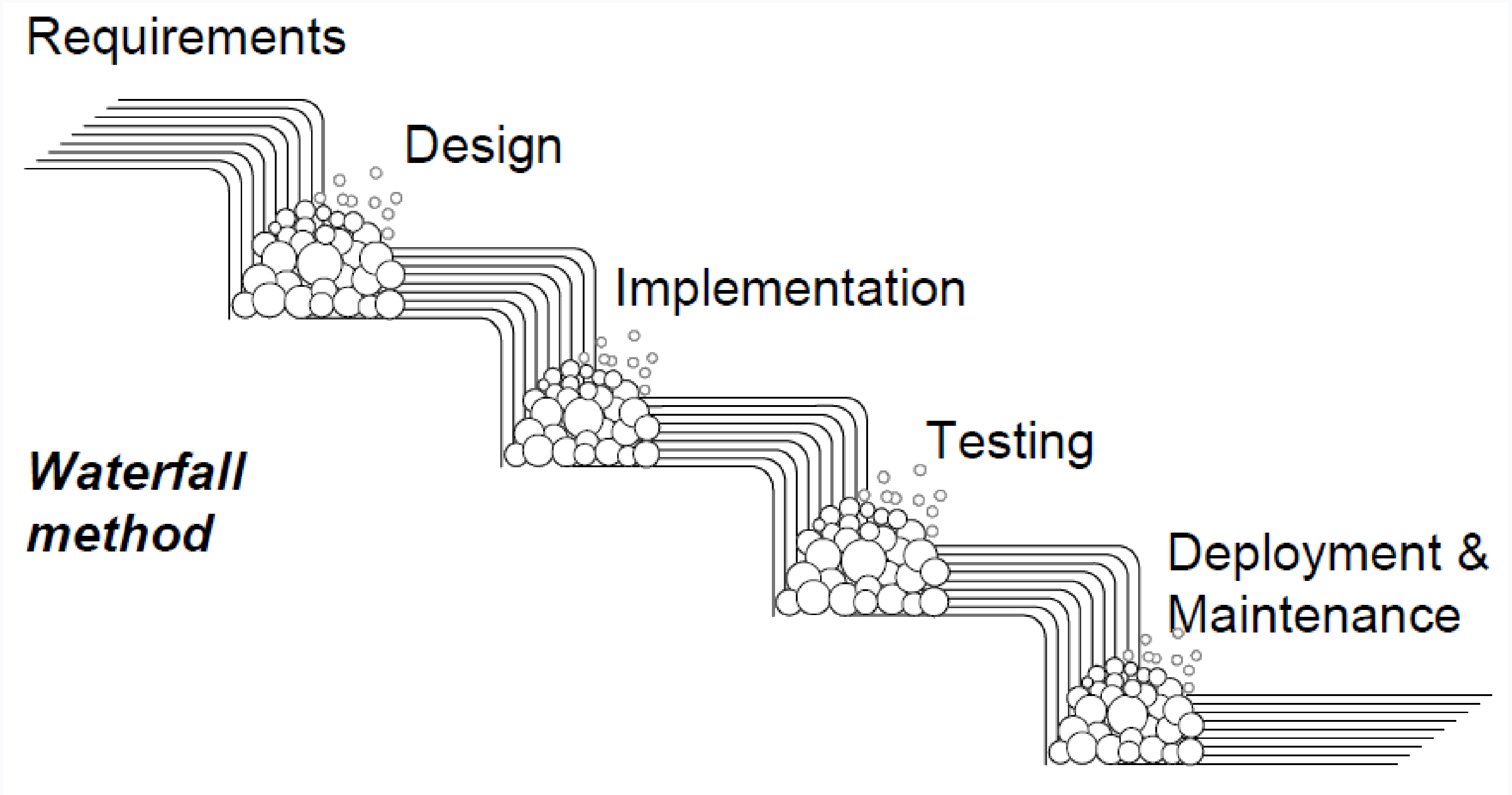
3. Implementation:
   - Encode the design in a programming language to deliver a software product
4. Testing:
   - Verify that the system works correctly and realises the goals
   - Testing process encompasses
     - unit tests (individual components are tested)
     - integration tests (the whole system is testing)
     - user acceptance tests (the system achieves the customer requirements)
5. Release & Maintenance:
   - Deploying the system, fix defects and adding new functionality

Software Development Methodologies

A software development methodology prescribes, how the different phases in SDLC are carried out and can be grouped into two broad categories:

- Waterfall Process:
  - Rigid, plan-driven documentation heavy methodologies
  - A linear process, where the various SDLC phases are carried out in a sequential manner
- Iterative & Incremental processes
  - which develop increments of functionality and repeat in a feedback loop
  - e.g., 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 when customer changes requirements
- Waterfall model prescribes a sequential process, but this linear order does not **always** 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

Incremental and Iterative Models

- Unified Software Development Process (Ivar Jacobson, Grady Booch and James Rumbaugh)
  - an iterative software development process where a system is progressively built and refined through multiple iterations, using feedback and adaptation
  - each iteration will include requirements, analysis, design, and implementation
  - Iterations are **timeboxed**
- Rational Unified Process (A specific adaptation of Unified Process), evolved at IBM
- Agile Methodologies (e.g., XP, SCRUM), that are more aggressive in terms of short iterations

Incremental models explored in detail in week 4

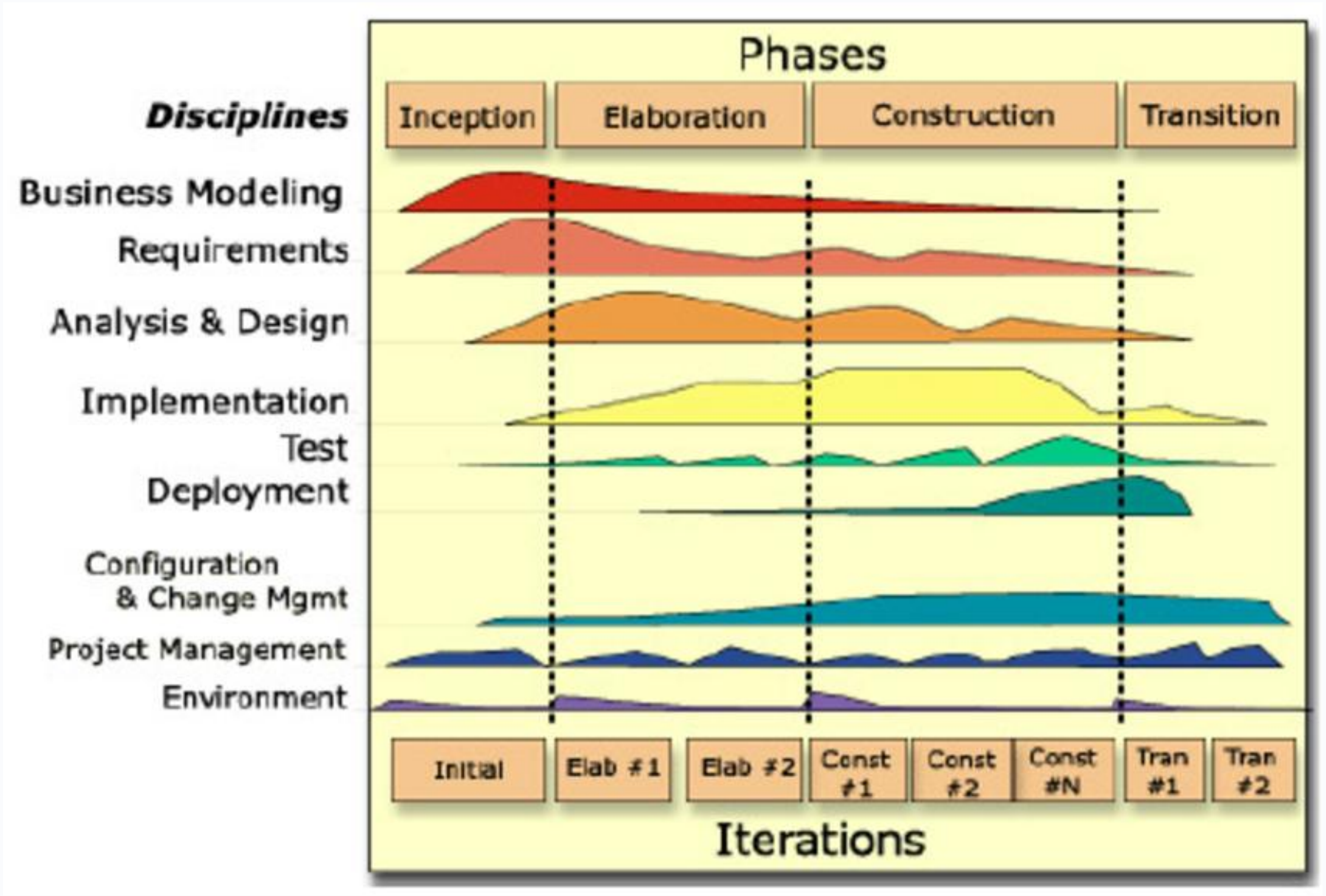
Checkpoint !

Quiz - Software Development Methodologies

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

RUP: Serial in the large, iterative in the small



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.

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

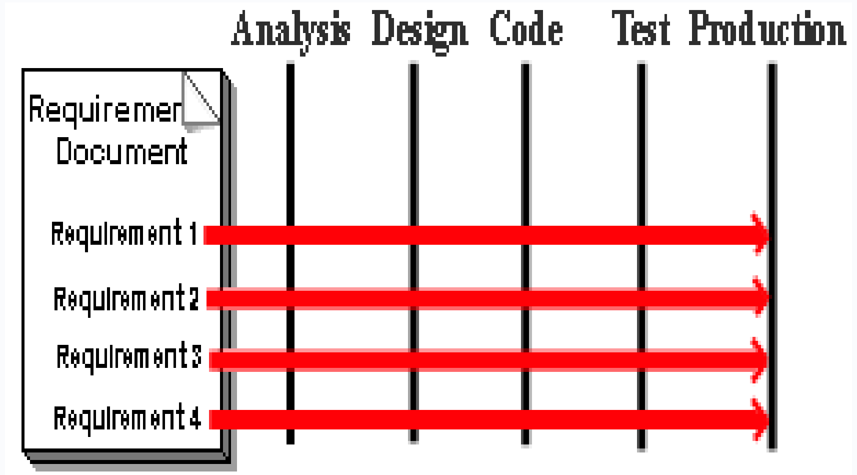
XP Core Principles: Continuous Feedback (3)

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

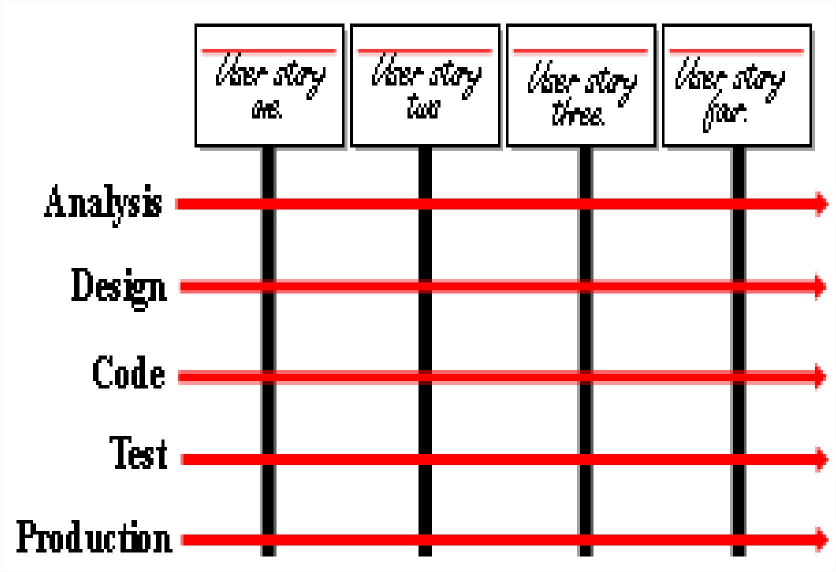
- Developers receive constant feedback by working in pairs, constant testing and continuous integration
- XP team receives daily feedback on progress and obstacles through daily stand-up meetings
- Customers get feedback on progress with user acceptance scores and demonstrations at the end of each iteration
- 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

What happens in XP Planning ?

Key steps in XP Planning Game:

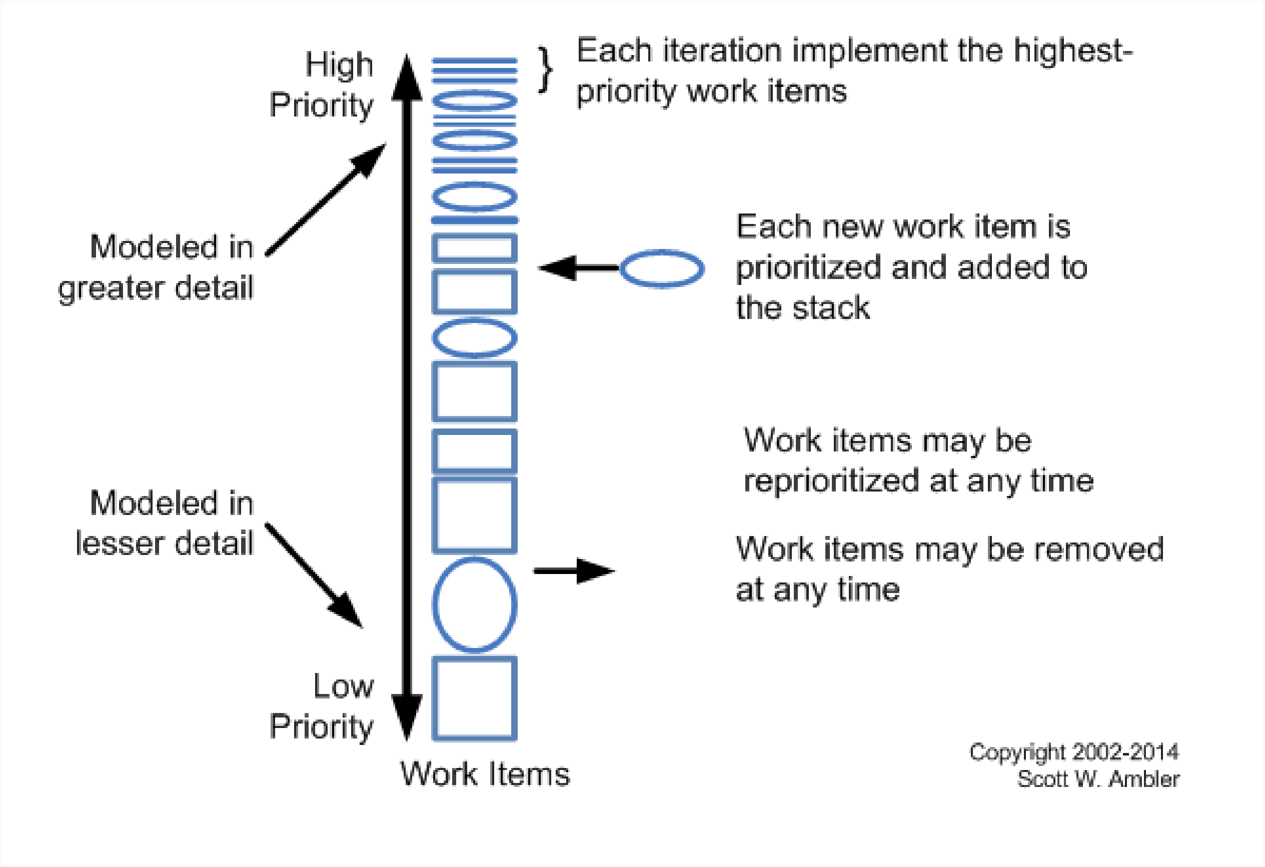
- Initial Exploration
- Release Plan
- Iteration Planning
- Task Planning

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.

User Story and Planning

- User-stories not only capture the user's vision but also impact the planning process in two key areas; estimating and scheduling


High Priority
Modeled in greater detail
Modeled in lesser detail
Low Priority
Work Items
Each iteration implement the highest-priority work items
Each new work item is prioritized and added to the stack
Work items may be reprioritized at any time
Work items may be removed at any time
Copyright 2002-2014 Scott W. Ambler

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.*

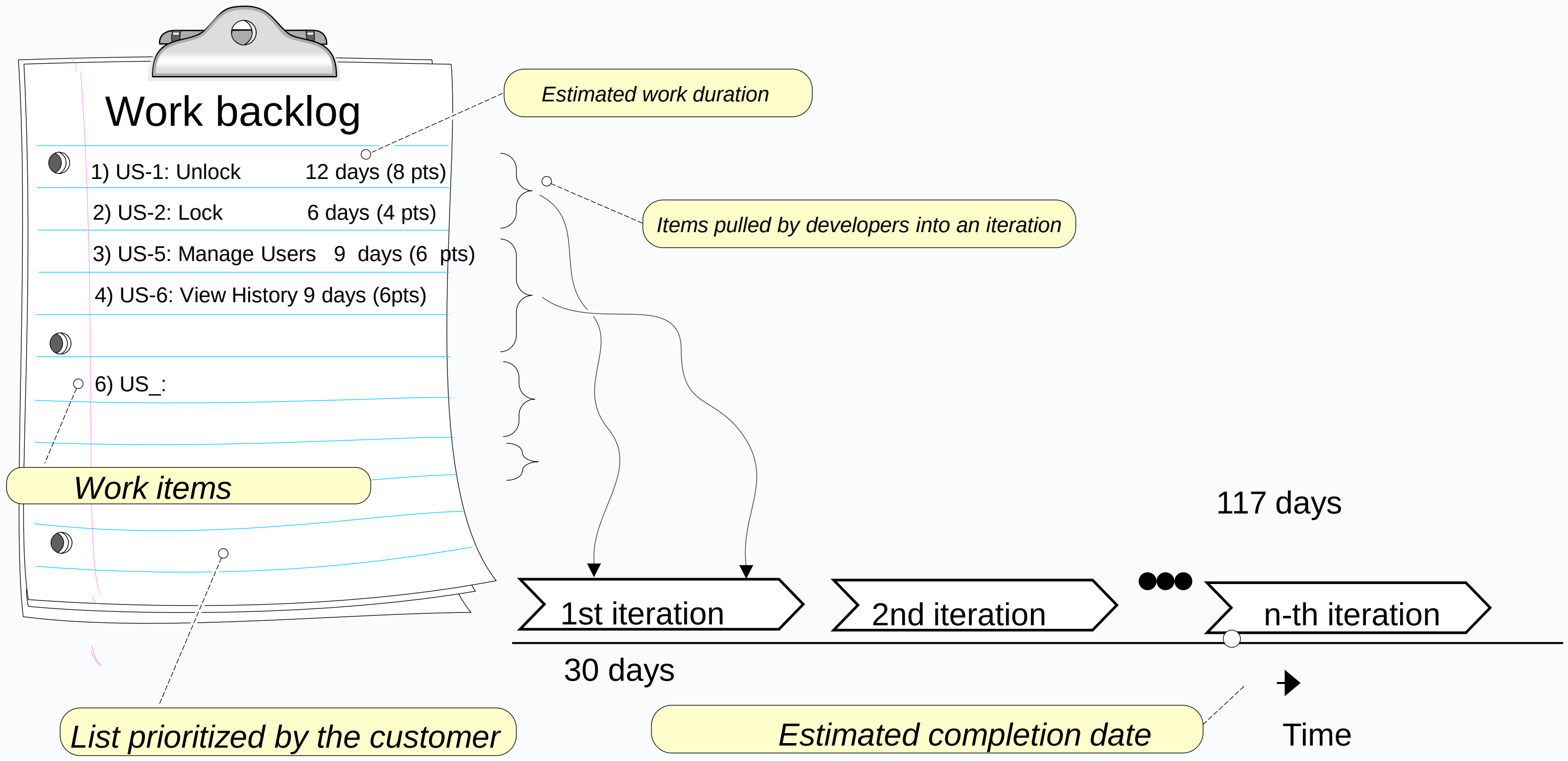
- ❖ Defining "done" - **A story is not done until *all* its acceptance tests pass**

The XP Planning Game: Task Planning

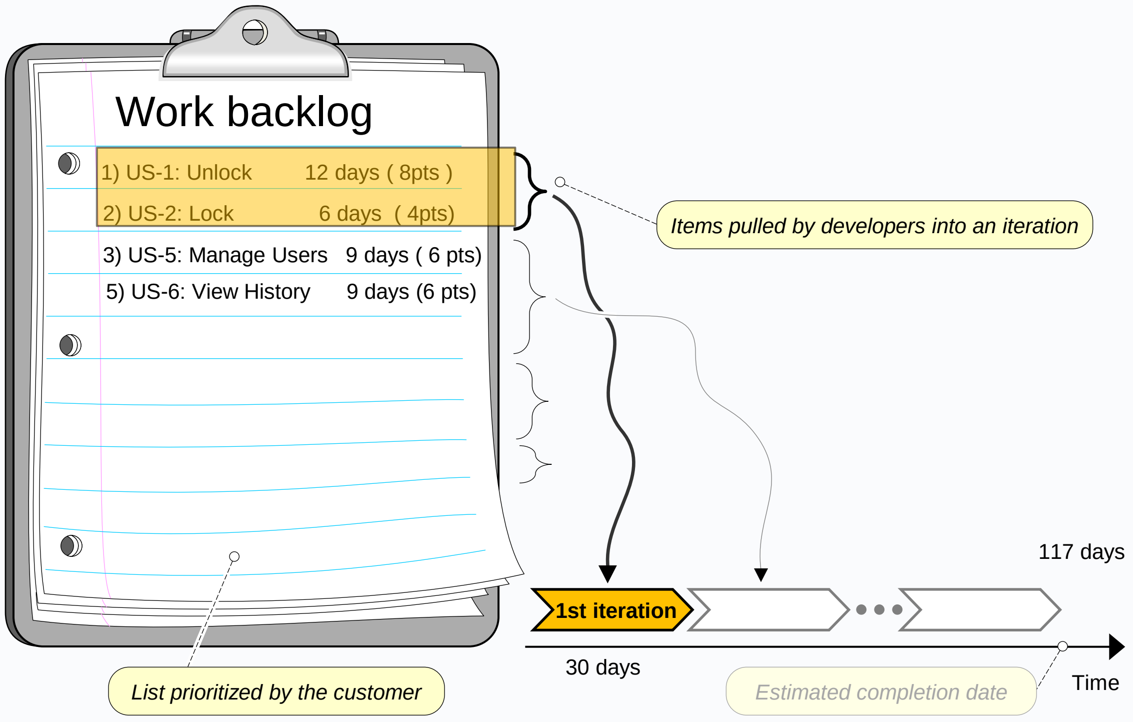
- Developers and customers arrange an **iteration planning meeting** at the beginning of each iteration
  - Customers choose user stories for the iteration from the release plan but **must** fit the current project velocity
  - User stories are broken down into programming tasks and order of implementation 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.*
  - **The velocity in task days (iteration planning) overrides the velocity in story weeks (release planning) as it is more accurate.**
  - Team holds a meeting halfway through iteration to track progress

Agile Project Effort Estimation for case-study

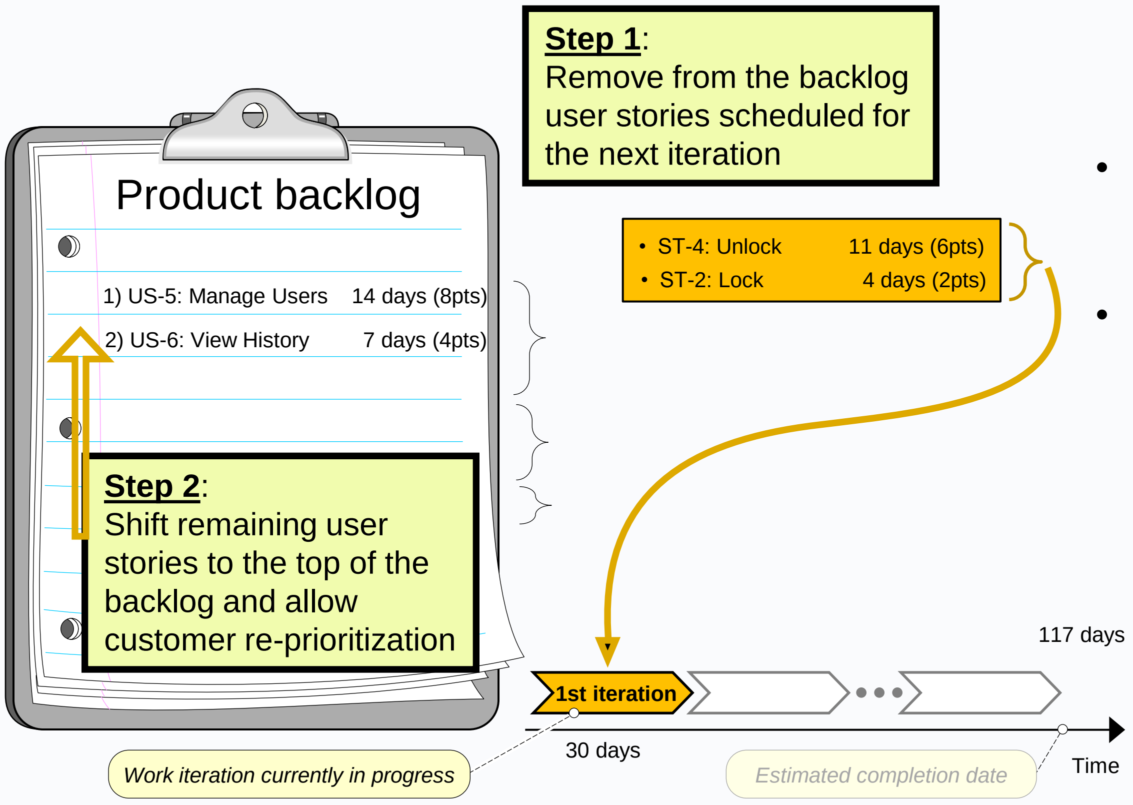
Total project effort is estimated based on the cumulative story points of all user-stories


Work backlog
1) US-1: Unlock 12 days (8 pts)
2) US-2: Lock 6 days (4 pts)
3) US-5: Manage Users 9 days (6 pts)
4) US-6: View History 9 days (6pts)
6) US_:
Estimated work duration
Items pulled by developers into an iteration
Work items
List prioritized by the customer
1st iteration
2nd iteration
n-th iteration
30 days
117 days
Estimated completion date
Time

Agile Prioritization of Work


Work backlog
1) US-1: Unlock 12 days (8pts)
2) US-2: Lock 6 days (4pts)
3) US-5: Manage Users 9 days (6 pts)
5) US-6: View History 9 days (6 pts)
Items pulled by developers into an iteration
117 days
1st iteration
30 days
Time
Estimated completion date
List prioritized by the customer

Update Product Backlog between iterations


Step 1:
Remove from the backlog user stories scheduled for the next iteration
Product backlog
1) US-5: Manage Users 14 days (8pts)
2) US-6: View History 7 days (4pts)
• ST-4: Unlock 11 days (6pts)
• ST-2: Lock 4 days (2pts)
Step 2:
Shift remaining user stories to the top of the backlog and allow customer re-prioritization
117 days
1st iteration
30 days
Time
Work iteration currently in progress
Estimated completion date

- 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

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

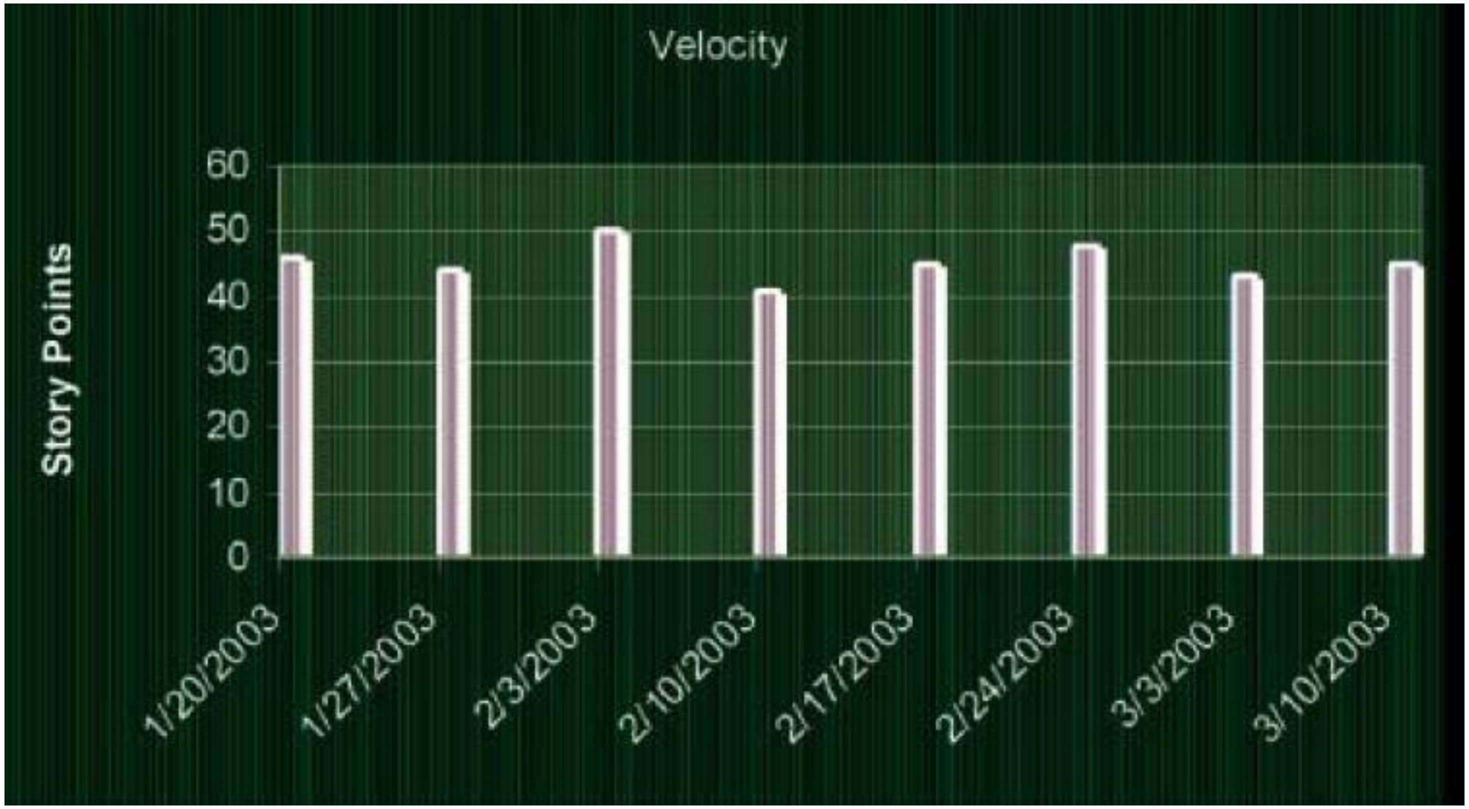
Project Tracking

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

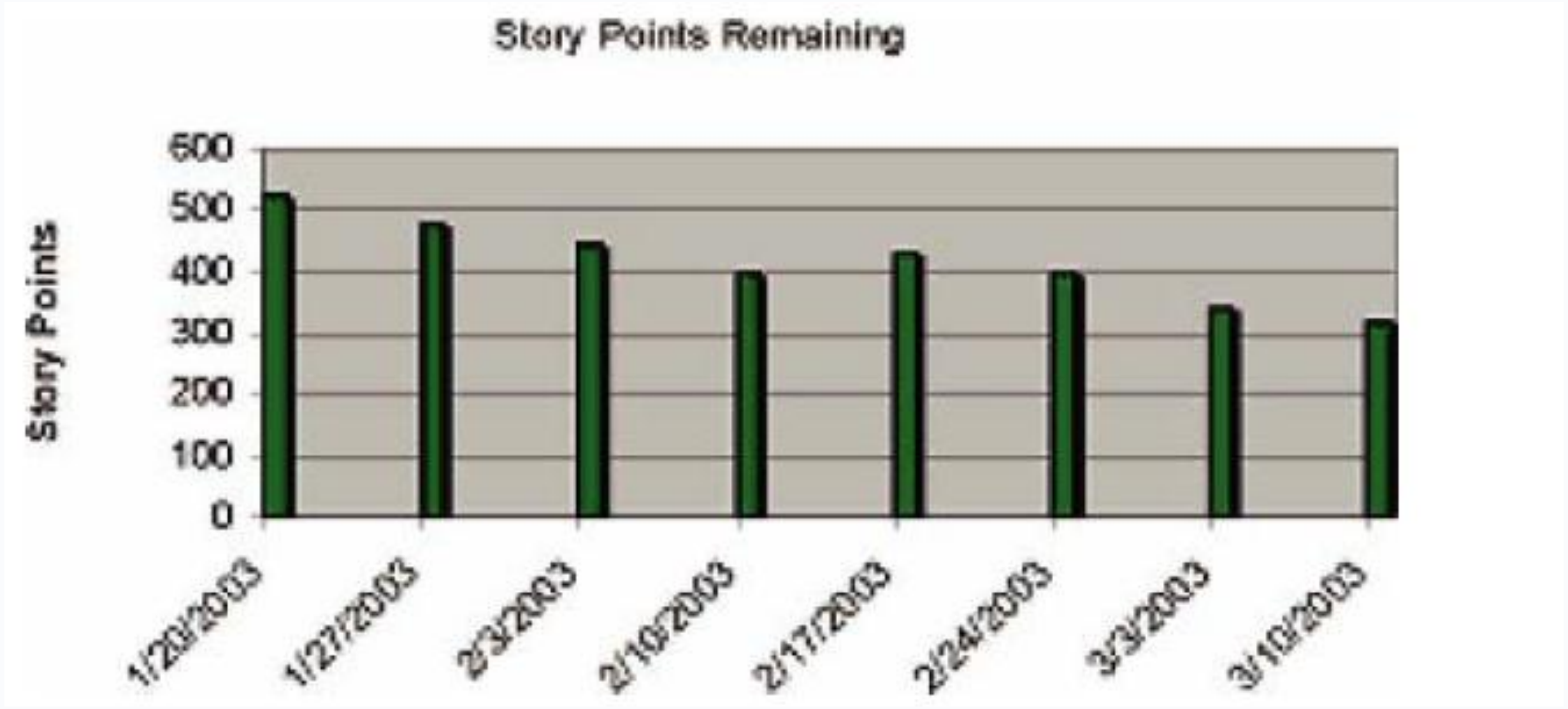
Project Tracking using Velocity Chart

- 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)



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.

When should XP be used?

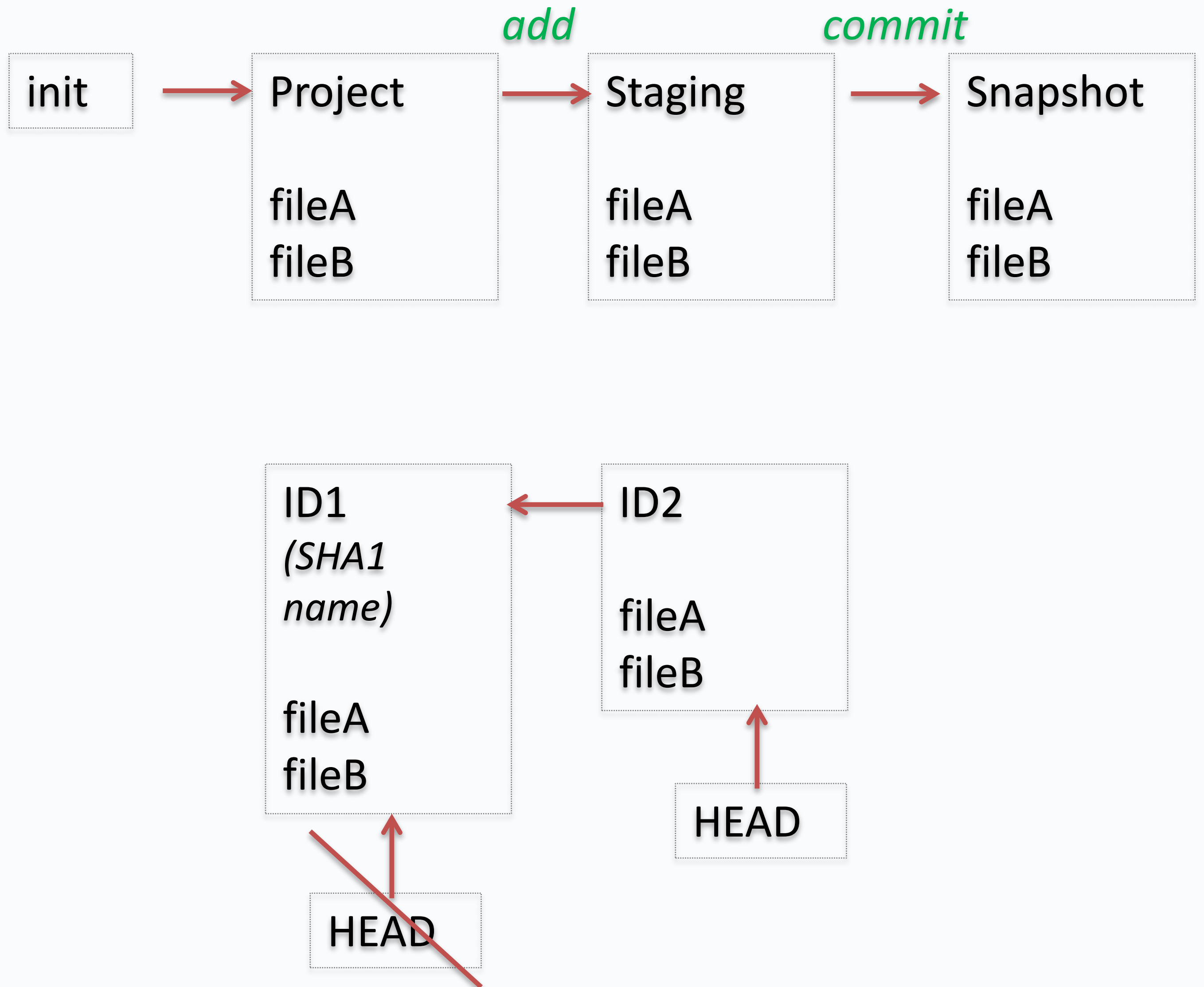
- Useful for problem domains where requirements change, when customers do not have a firm idea of what they want
- XP was set up to address project risk. XP practices mitigate risk and increase the likelihood of success (e.g. a new challenge for a software group to be delivered by a specific date)
- 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

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

Lecture Demo: Git & GitHub

Git Basics – Creating Snapshots



Checkpoint !

Check your Git understanding by watching the series of Git Videos and answering the following questions

1. Quiz - Git Basics
2. Quiz - Branching
3. Quiz - Merging Branches
4. Quiz - Remote Repositories