

# **MCA20303: AGILE TECHNOLOGIES**

## **Unit I: Software Engineering**

**Dr. M. Durairaj**  
**Associate Professor**  
**School of Computer Science, Engineering and Applications,**  
**Bharathidasan University,**  
**Tiruchirappalli - 620 023**

# Software Engineering

## Unit I : Introduction to Software Engineering

- **Introduction** : Nature of software, Software Process, Software Engineering Practice, Software Myths, Generic Process Model
- **Process Models** : Waterfall Model, Incremental Models, Evolutionary Models, Concurrent Process Model, Specialized process Models
- **Personal & Team Process Models**
- **Agile process Models** : Agile Process, Extreme Programming (XP)

- ❖ What is Software?
- ❖ What are the characteristics of Software?
- ❖ Nature of Software
- ❖ Software Myths
- ❖ Software Engineering Practice
- ❖ Framework Activities
- ❖ Umbrella Activities

## What is Software?

- Set of programs
- Data Structures
- Documentation :

## ❓ What are the characteristics of Software?

- ❖ Logical rather than Physical
- ❖ Software is developed or engineered, it is not manufactured (Quality, People, Process, Cost)
- ❖ Software does not wear out (Bathtub Curve)
- ❖ Moving towards component based construction, however most of the software is custom built

# Hardware vs. Software

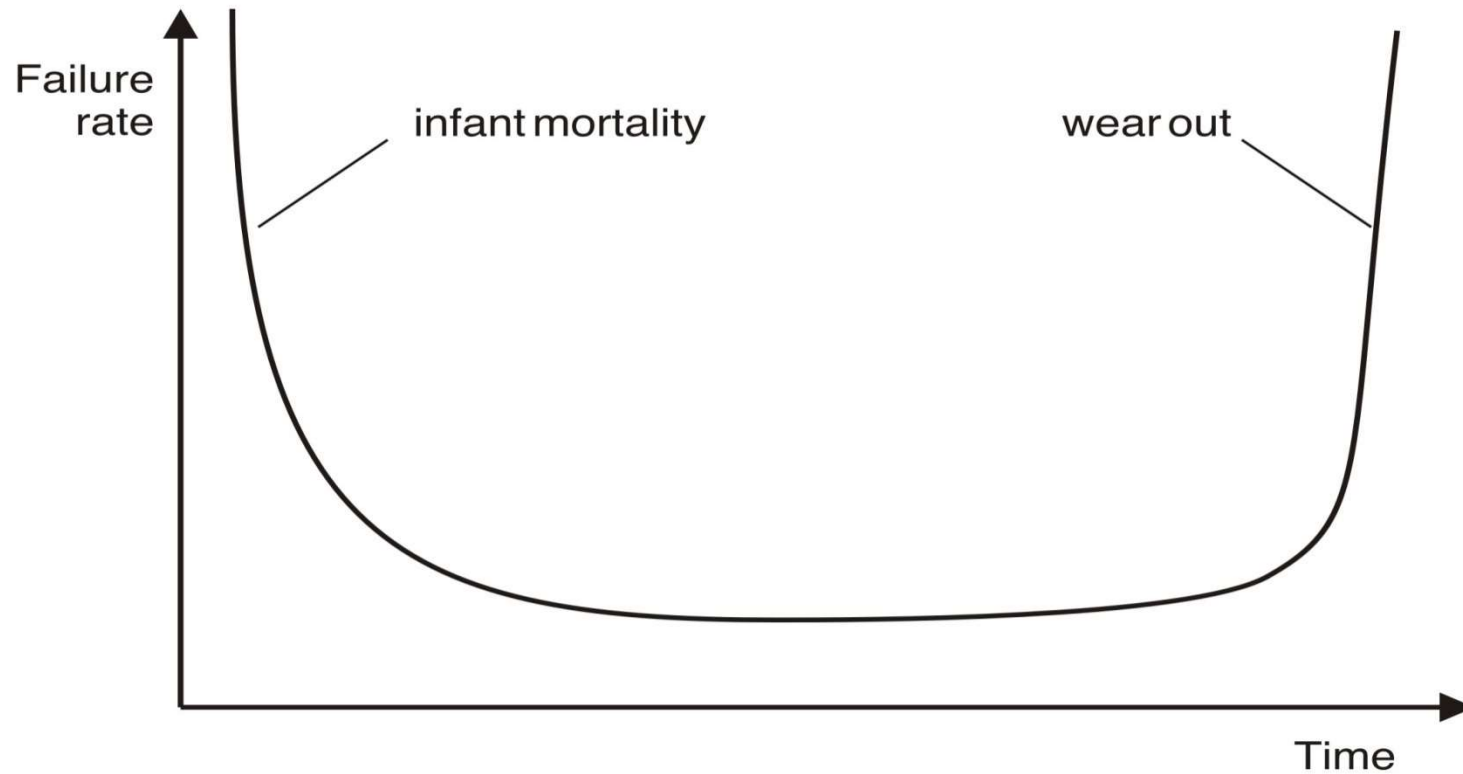
Hardware	Software
<ul style="list-style-type: none"><li>▪ Manufactured</li><li>▪ Wears out</li><li>▪ Built using components</li><li>▪ Relatively simple</li></ul>	<ul style="list-style-type: none"><li>▪ Developed/Engineered</li><li>▪ Deteriorates</li><li>▪ Custom built</li><li>▪ Complex</li></ul>

# Manufacturing vs. Development

- Once a hardware product has been manufactured, it is difficult or impossible to modify. In contrast, software products are routinely modified and upgraded.
- In hardware, hiring more people allows you to accomplish more work, but the same does not necessarily hold true in software engineering.
- Unlike hardware, software costs are concentrated in design rather than production.

# Wear vs. Deterioration

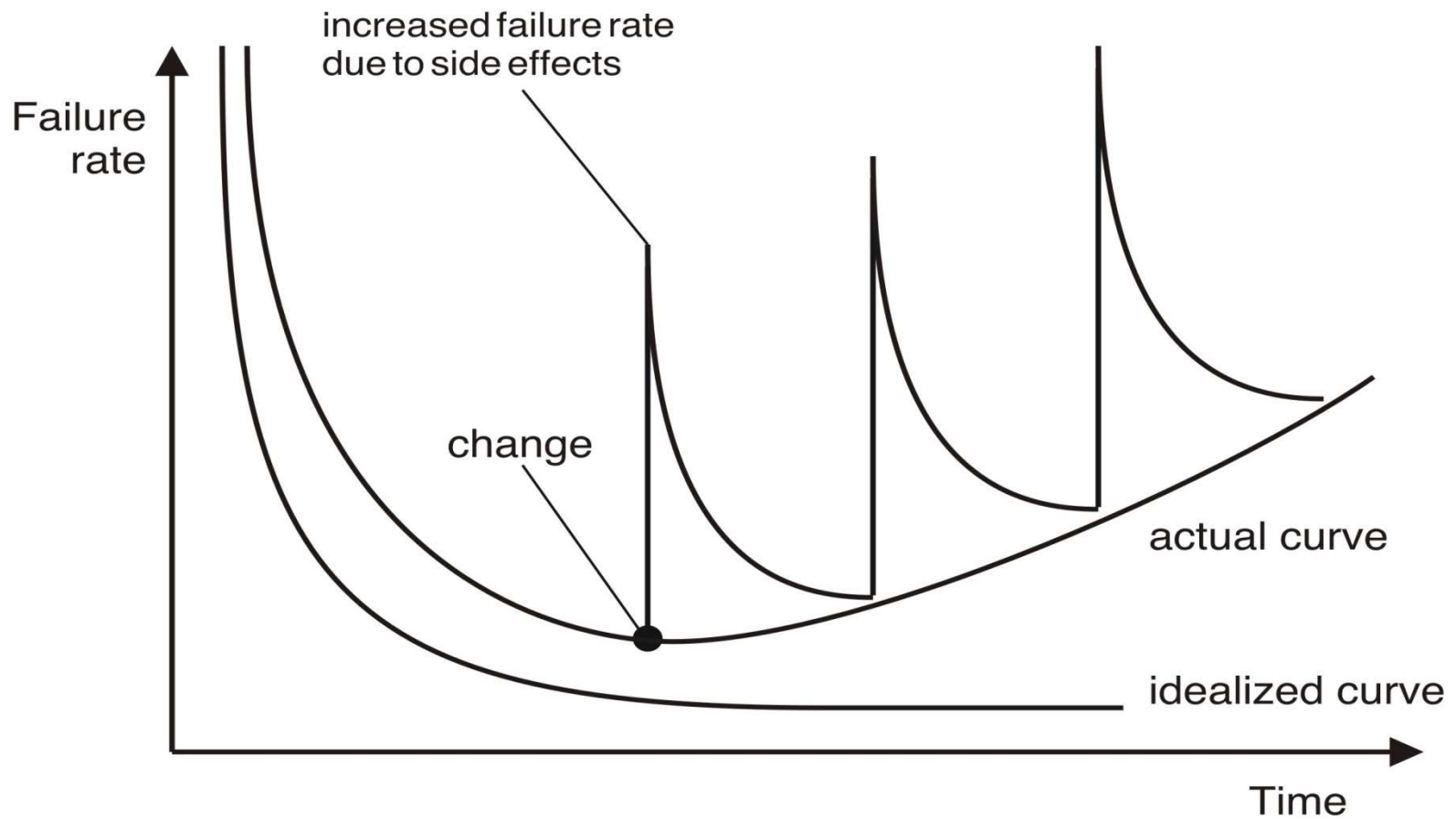
Hardware wears out over time





# Wear vs. Deterioration

Software deteriorates over time



# Software Complexity

“I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation”.

If this is true, building software will always be hard. There is inherently no silver bullet.

- Fred Brooks, “No Silver Bullet”

<http://www.computer.org/computer/homepage/misc/Brooks/>

## **❓ Nature of Software**

- ❖ Systems Software
- ❖ Applications Software
- ❖ Engineering / Scientific software
- ❖ Embedded Software
- ❖ Product Line Software
- ❖ Web Applications
- ❖ Artificial Intelligence Software
- ❖ Ubiquitous Software
- ❖ Net sourcing
- ❖ Open Source

## **❑ Software Myths**

❑ Management Myths

❑ Customer Myths

❑ Practitioners Myth

# Software Myths

## ❓ Management Myths

❓ We are already having a book that is full of Standards & Procedures for building software. It will provide my people everything they need to know.

❓ If we get behind schedule, we can add more developers & catch up.

❓ We can outsource the software project to a third party. I can just relax & let that firm build it.

# Software Myths

## ❑ Customer Myths

- A general statement of objectives is sufficient to begin writing programs.
- We can fill in the details later
- Project requirements continually change, but the change can be easily accommodated because software is flexible

# Software Myths

## ❓ Practitioners Myth

- Once we write the program and get it to work our job is done
- Until I get the program running, I have no way of accessing its quality
- The only deliverable work product for a successful project is the working program
- Software Engineering will make us create voluminous & unnecessary documentation & invariably slow us down

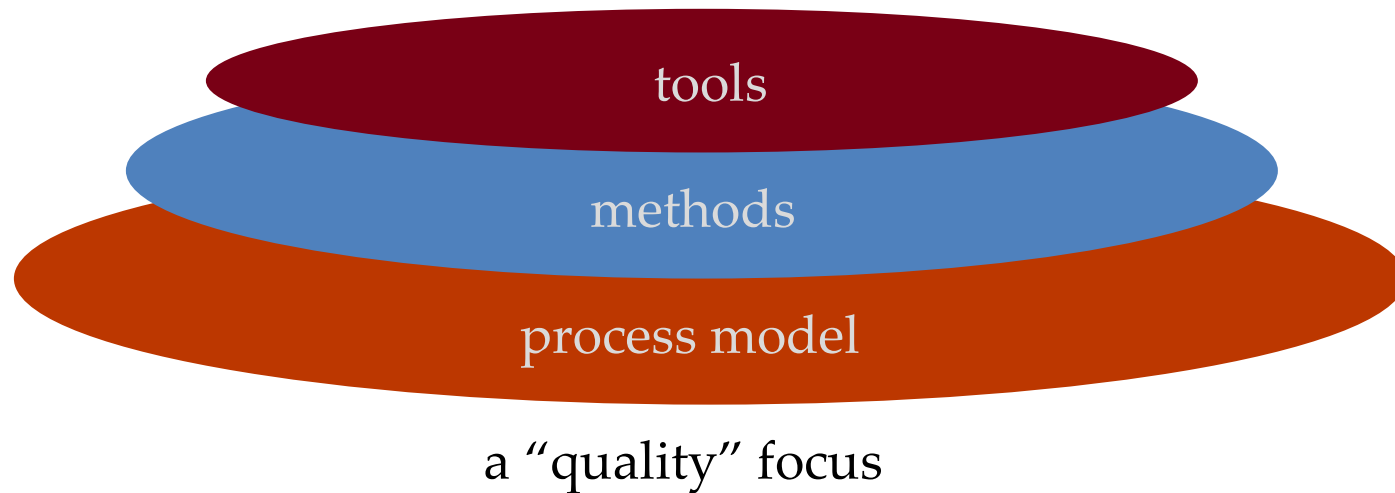
## ❑ Software Engineering – A layered Technology

- **Software Engineering** is the application of systematic, disciplined, quantifiable approach to the development, operation and maintenance of software i.e. the application of engineering to software [ IEEE definition ]



# A Layered Technology

Software Engineering



# Software Engineering Layers

**Quality Focus** : Organizational commitment to quality ( TQM, Six Sigma )

**Process** : Defines a framework (Foundation for Software Engineering)

**Methods** : “How to”s for building software (Tasks)

**Tools** : Automated or semi-automated support (Rational Rose, CASE tools)

## Framework Activities

Communication

Planning

Modeling

Construction

Deployment

# A Process Framework

Software process

Process framework

Umbrella activities

framework activity #1

SE action #1.1

task  
set  
s

work  
tasks  
work  
products  
QA points  
milestone  
s

SE action #1.2

task  
set  
s

work  
tasks  
work  
products  
QA points  
milestone  
s

framework activity #2

SE action #2.1

task  
set  
s

work  
tasks  
work  
products  
QA points  
milestone  
s

SE action #2.2

task  
set  
s

work  
tasks  
work  
products  
QA points  
milestone  
s

# **Umbrella Activities**

- Software Project Tracking & Control
- Risk Management
- Software Quality Assurance
- Formal Technical Reviews
- Measurement
- Software Configuration Management
- Reusability Management

## **Software Engineering Practice**

- **Practice is collection of Concepts, Principles, Methods & Tools that a software engineer calls upon on a daily basis.**
- **Practice allows Managers to manage software projects & Software Engineers to build computer programs.**

# The Essence of SE Practice

1. Understand the Problem – Communication & Analysis
2. Plan a Solution – Modeling & Software Design
3. Carry out the Plan – Code generation
4. Examine the Results – Testing & QA

## Core Principles

- 1.The Reason it all Exists
- 2.Keep It Simple, Stupid (KISS!)
- 3.Maintain the Vision
- 4.What you Produce, others will Consume
- 5.Be Open to Future
- 6.Plan Ahead for Reuse
- 7.THINK



# Software process model

- **Attempt to organize the software life cycle by**
  - defining activities involved in software production
  - order of activities and their relationships
- **Goals of a software process**
  - standardization, predictability, productivity, high product quality, ability to plan time and budget requirements

# Code & Fix

## The earliest approach

- Write code
- Fix it to eliminate any errors that have been detected, to enhance existing functionality, or to add new features
- Source of difficulties and deficiencies
  - impossible to predict
  - impossible to manage

# Process Models

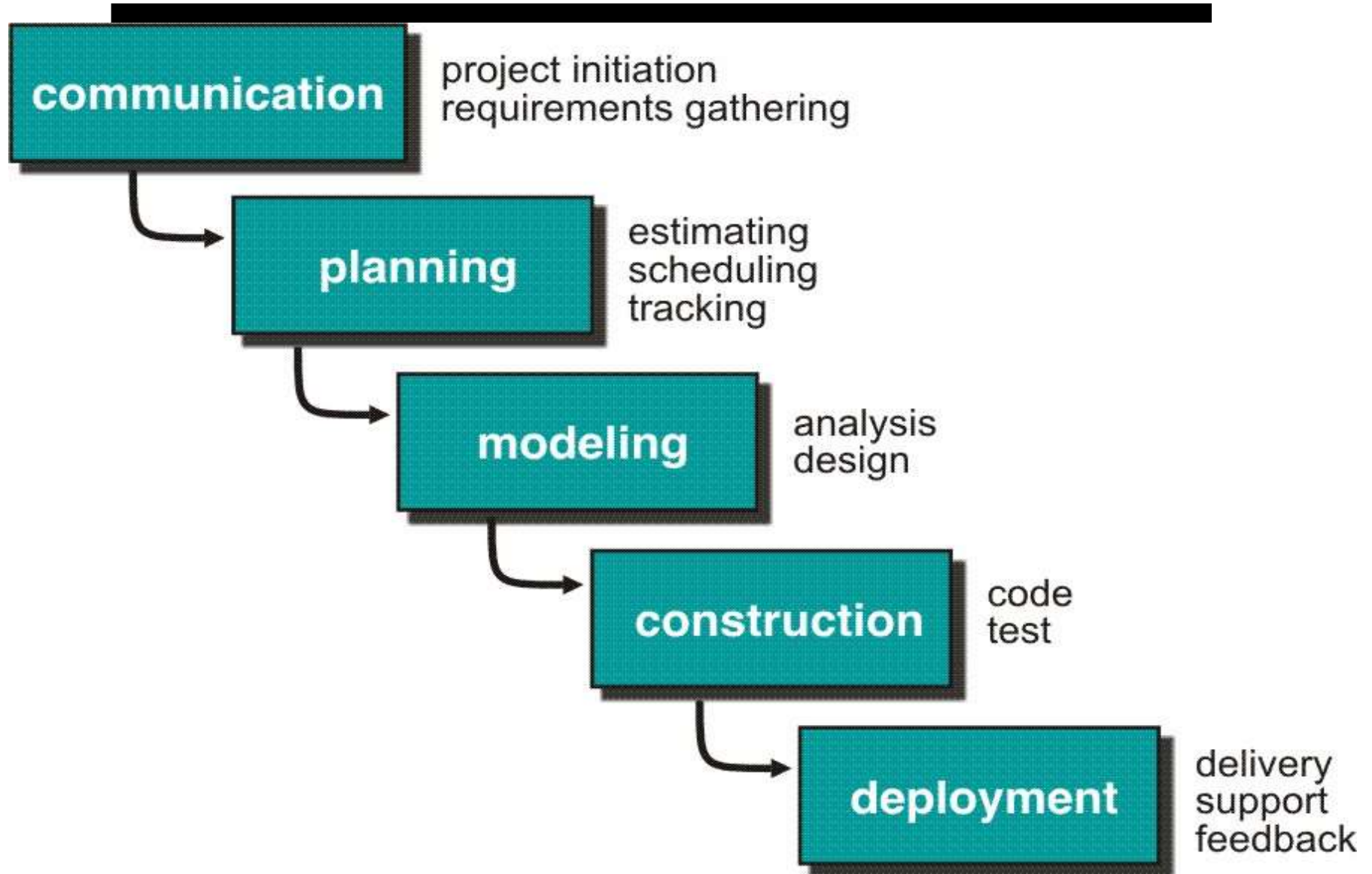
## Prescriptive Process Models

- The Waterfall Model
- Incremental Model
- The RAD Model

## Evolutionary Process Models

- The Prototyping Model
- The Spiral Model
- The Concurrent Development Model

# The Waterfall Model



# **The Waterfall Model: (Payroll System)**

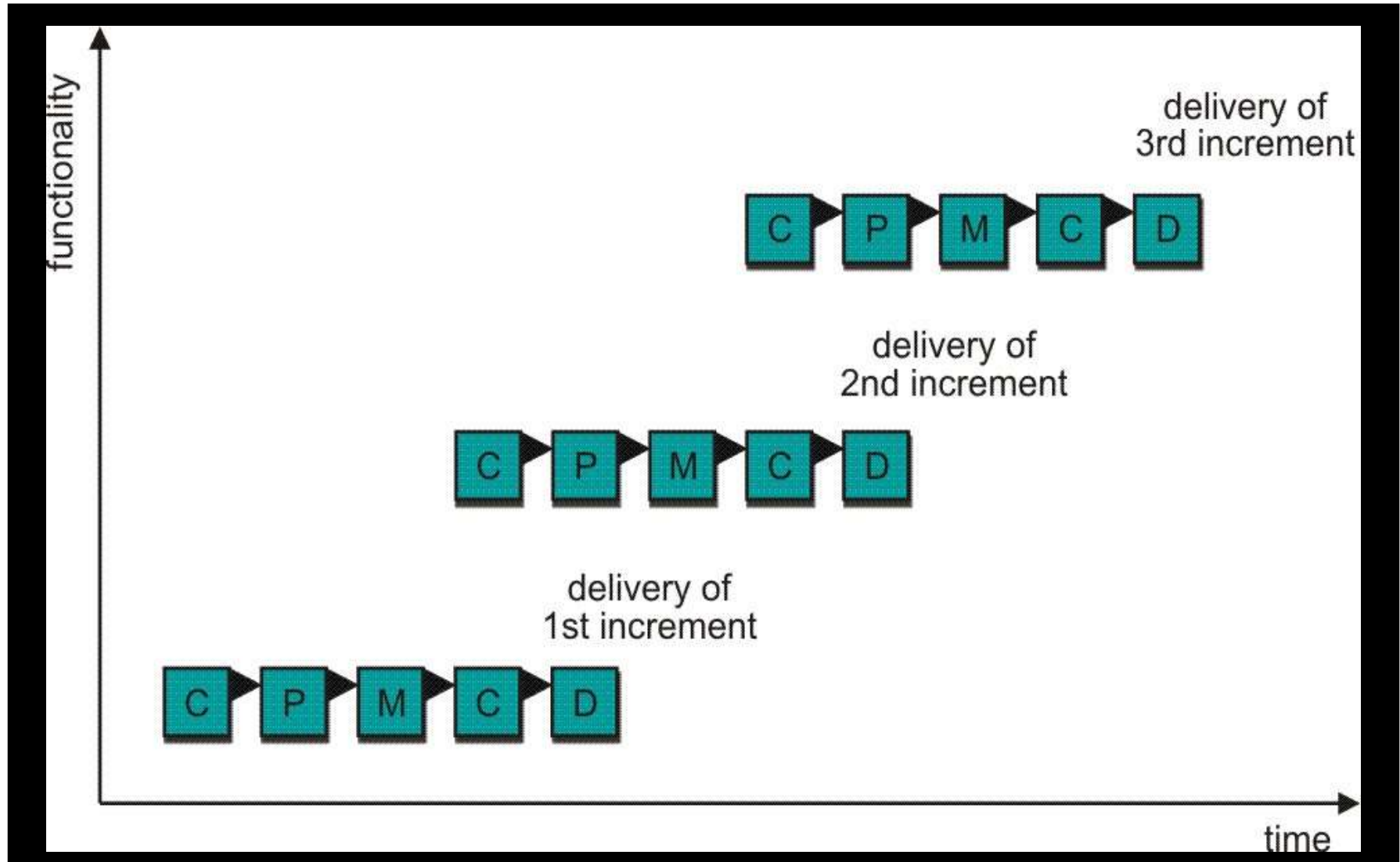
## **Merits :**

- **It is systematic sequential approach for Software Development**

## **Demerits**

- **All Customer Requirements at the start of project may be difficult**
- **Problems remain uncovered until testing phase**
- **Customer patience is needed, working version of the software is delivered too late.**

# Incremental Models: Incremental



# Incremental Model: (Word Processor)

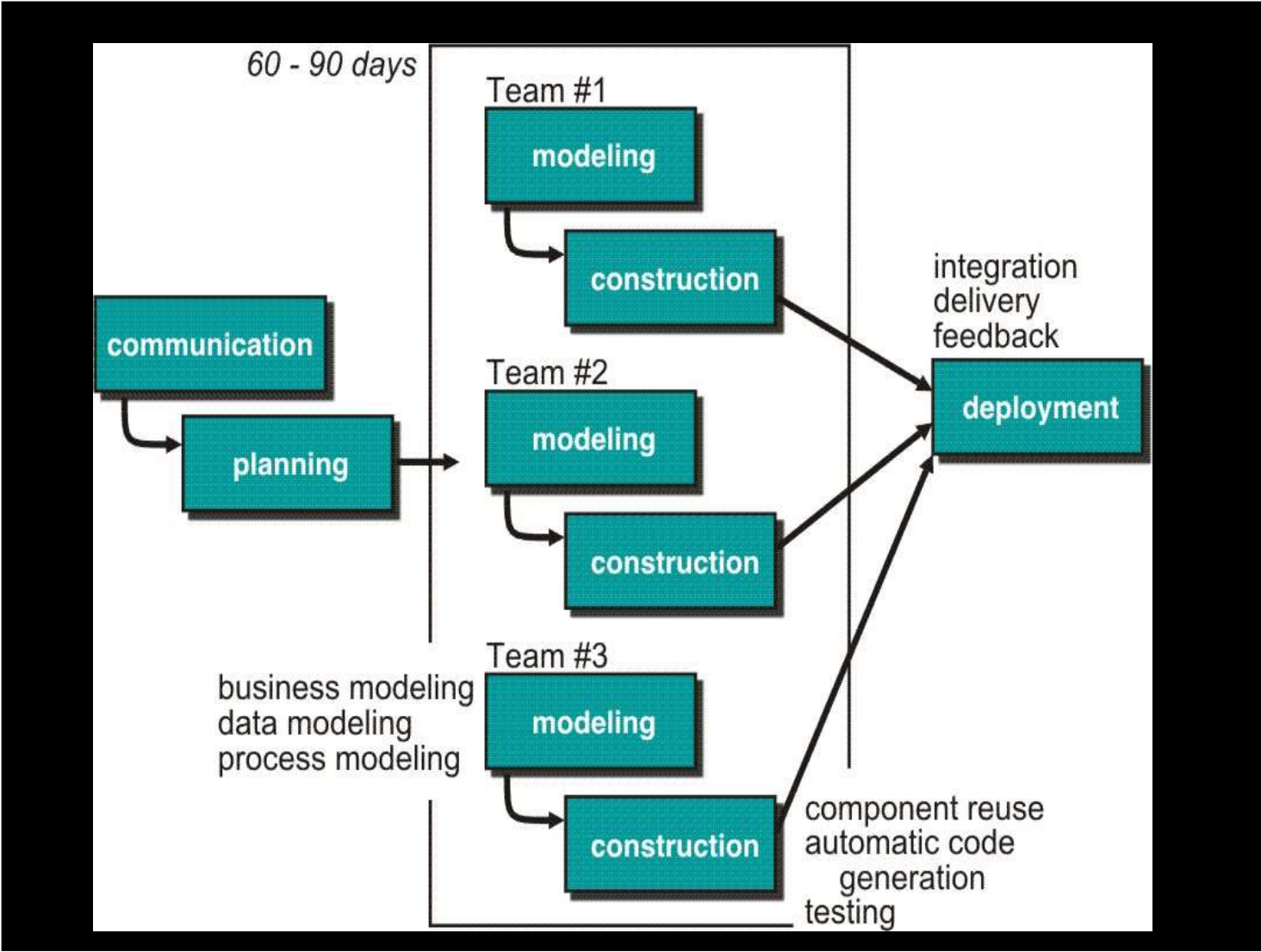
## Merits :

- Less number of developers required
- All the requirements need not be known at the beginning of the project
- Technical risks can be managed

## Demerits :

- Problems remain uncovered until testing phase
- Customer patience is needed, working version of the software is delivered too late.

# Incremental Models: RAD Model





# The RAD Model : (Very Large Projects)

## Merits :

- Project cycle time is reduced

## Demerits :

- All Customer Requirements at the start of project may be difficult
- For large projects high human resources are required
- Risk of project failure if teams are not committed to rapid fire action
- Problems due to improper modularization of system
- RAD approach may not work if high is an issue
- RAD may not be appropriate if technical risks are very high

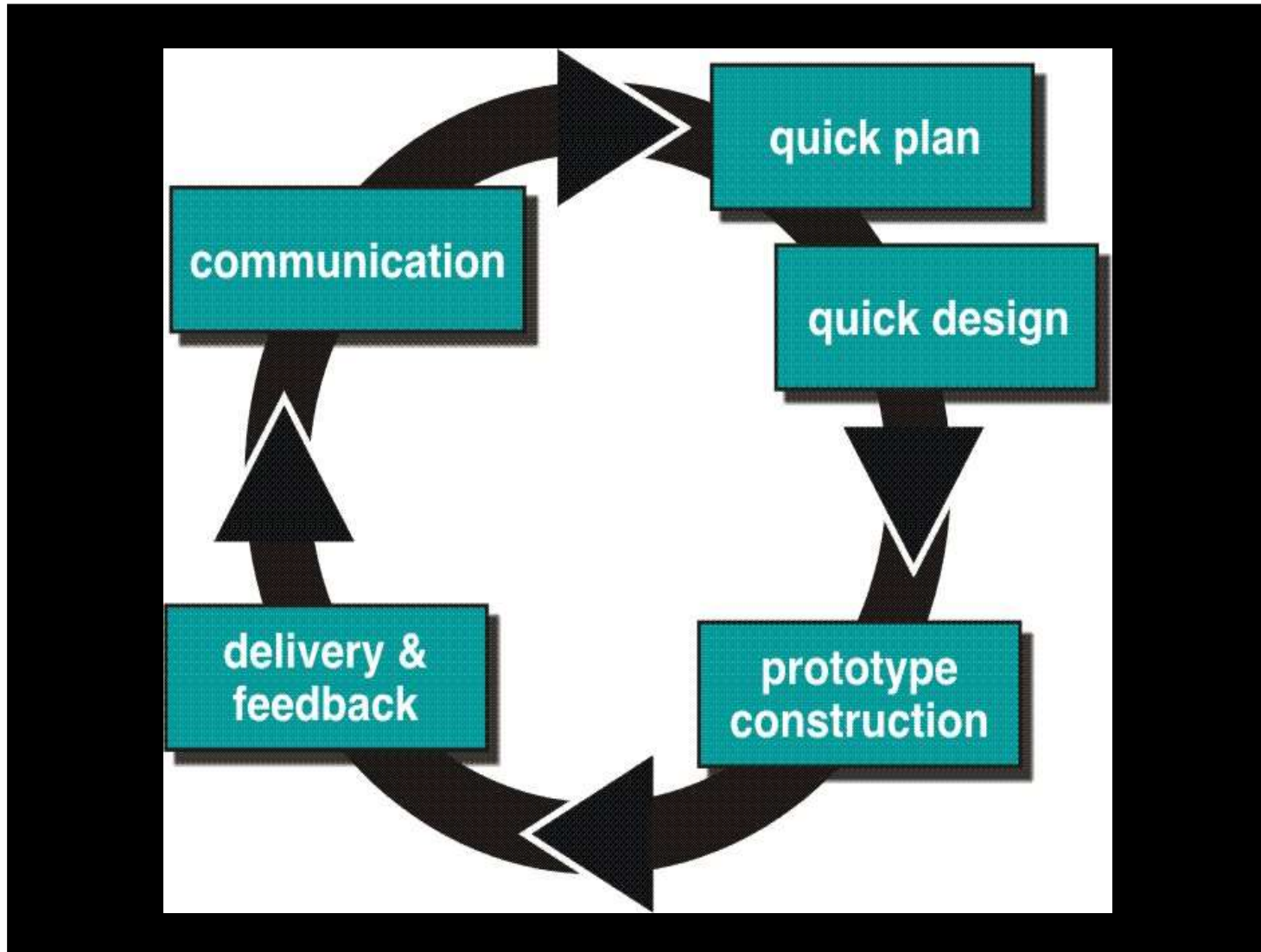
## **Evolutionary Process Models: Need**

- Software like all complex systems evolves over a period of time.**
- Target market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure.**
- Some of the core product or system requirements are well understood but the details of product or system extensions have yet to be defined.**
- Solution is to adapt Evolutionary Model which is Iterative**

## **The Prototyping Model: (Need)**

- **Prototyping is used when customers requirements are fuzzy.**
- **OR the developer may not be sure of the efficiency of algorithm, the adaptability of an Operating System or the form that Human Computer interaction should take**
- **But we have to throw away the prototype once the customer requirements are clear & met for better quality. The product must be rebuilt using software engineering practices for long term quality.**

# Evolutionary Models: Prototyping



# The Prototyping Model:

## Merits:

- Prototyping helps in requirement gathering & can be applied at any stage of the project.

## Demerits:

- Customer insists to convert prototype in working version by applying “few fixes”
- Developer may become comfortable with the compromises done. “The-less-than-ideal-choice” may become integral part of the system

# Evolutionary Models: Spiral

**Spiral Model is an evolutionary software process model that couples the iterative nature of Prototyping with controlled & systematic aspects of the Waterfall Model**

## **Merits:**

- Risk is considered as each iteration is made
- Spiral Model can be applied throughout the life of the computer software.

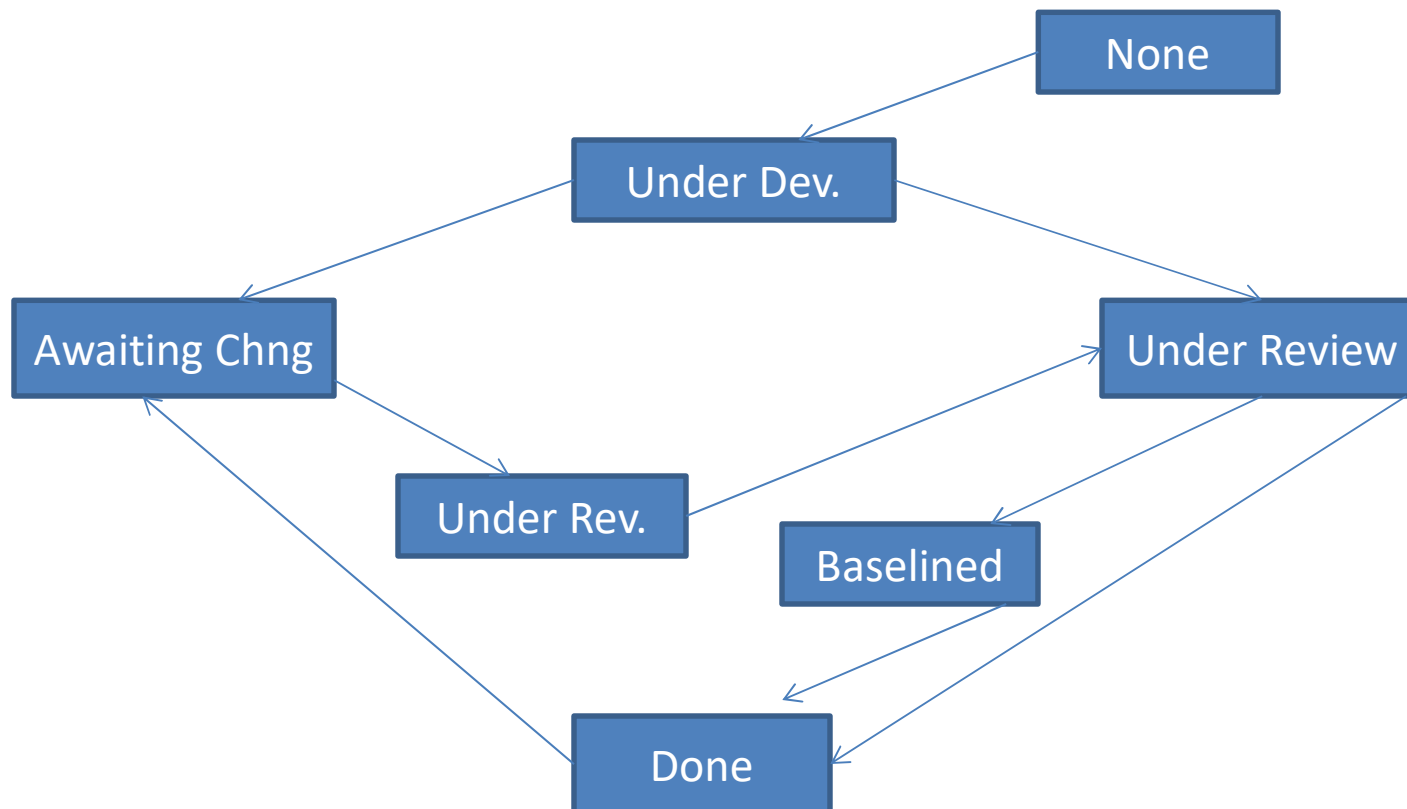
## **Demerits:**

- It is difficult to convince customers that the evolutionary approach is controllable
- Considerable risk assessment expertise required
- If major risk is uncovered, problems will occur

## Concurrent Development Model:

The concurrent Development Model, sometimes called concurrent engineering can be represented schematically as a series of framework activities, software engineering actions and tasks, & their associated states. All activities exist concurrently.

**Modeling activity (Example) :**



# Concurrent Development Model: Contd...

## Merits:

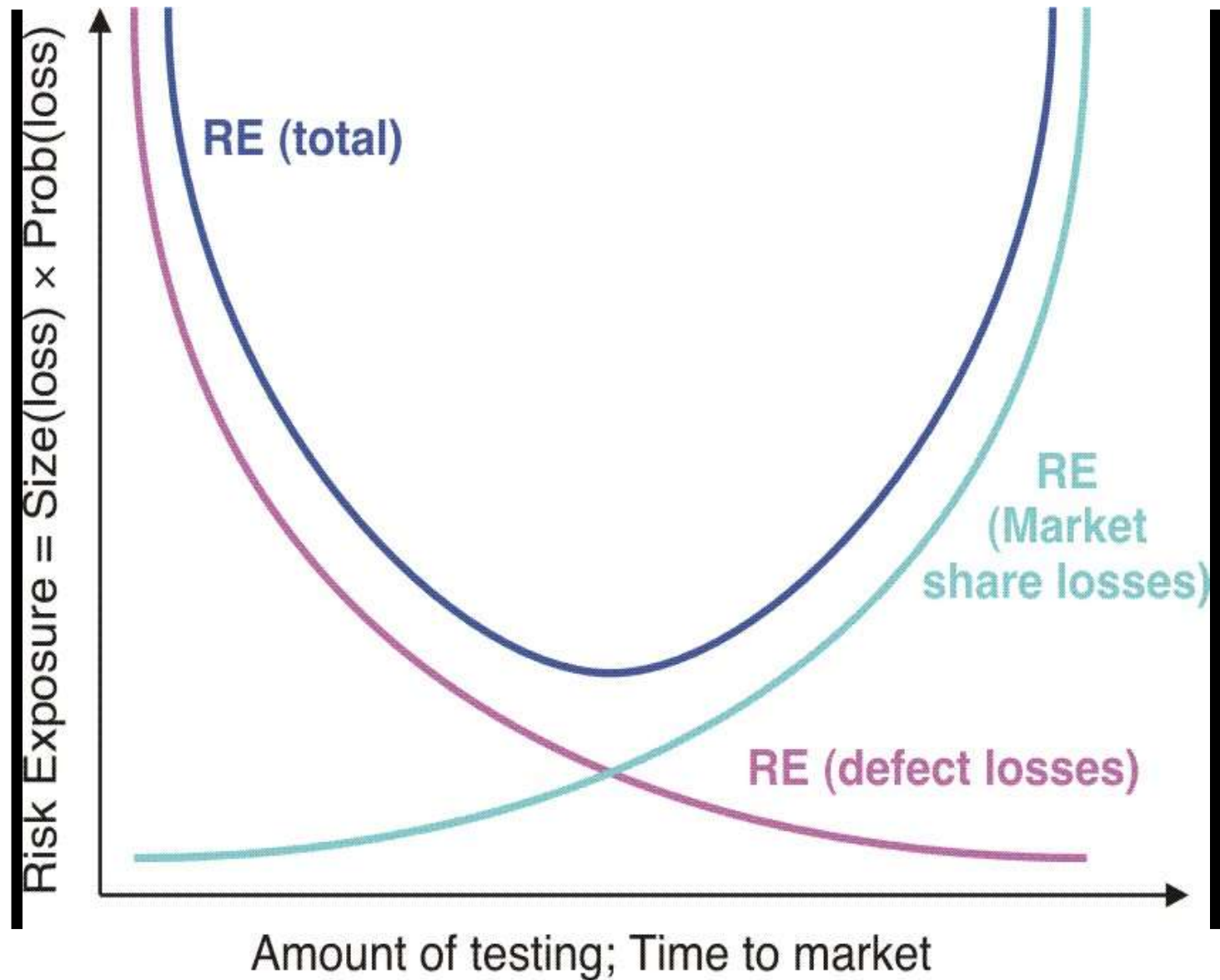
- Applicable to all types of S/W development & provides an accurate picture of the current state of the project.

## Demerits:

- Problem to Project planning. How many No of iterations are to be planned? Uncertainty...
- Process may fall in chaos if the evolutions occurs too fast without a period of relaxation. On the other hand if the speed is too slow productivity could be affected.
- S/W processes are focussed on flexibility & extendability, rather than on high quality.



# Risk Exposure



## **Specialized Process Models:**

**Specialized process models use many of the characteristics of one or more of the conventional models presented so far, however they tend to be applied when a narrowly defined software engineering approach is chosen. They include,**

- Components based development**
- The Formal Methods Model**
- Aspect oriented software development**

# Components Based Development :

In this approach, Commercial Off-The-Shelf (COTS) S/W components, developed by vendors who offer them as products are used in the development of software. Characteristics resemble to spiral model.

## Merits:

- Leads to software reuse, which provides number of benefits
  - 70% reduction in development cycle time
  - 84 % reduction in project cost
  - Productivity index goes up to 26.2 ( Norm : 16.9)

## Demerits:

- Component Library must be robust.
- Performance may degrade

## **The Formal Methods Model:**

- The formal methods model encompasses a set of activities that lead to formal mathematical specification of computer software.
- It consists of specifications, development & verification by applying rigorous mathematical notation. Example, Clean Room S/W Engineering (CRSE)

### **Merits:**

- Removes many of the problems that are difficult to remove using other S/W Engg. Paradigms.
- Ambiguity, Incompleteness & Inconsistency can be discovered & corrected easily by using formal methods of mathematical analysis.

### **Demerits:**

- Development is time consuming & expensive
- Extensive training is required
- Difficult to use with technically unsophisticated customers

## Aspect Oriented Software Development (AOSD):

- A set of localized features, functions & information contents are used while building complex software.
- These localized s/w characteristics are modeled as components (e.g. OO classes) & then constructed within the context of a system architecture.
- Certain “concerns” (Customer required properties or areas of technical interest) span the entire architecture i.e. Cross cutting Concerns like system security, fault tolerance etc.

### Merits:

- It is similar to component based development for aspects

### Demerits:

- Component Library must be robust.
- Performance may degrade

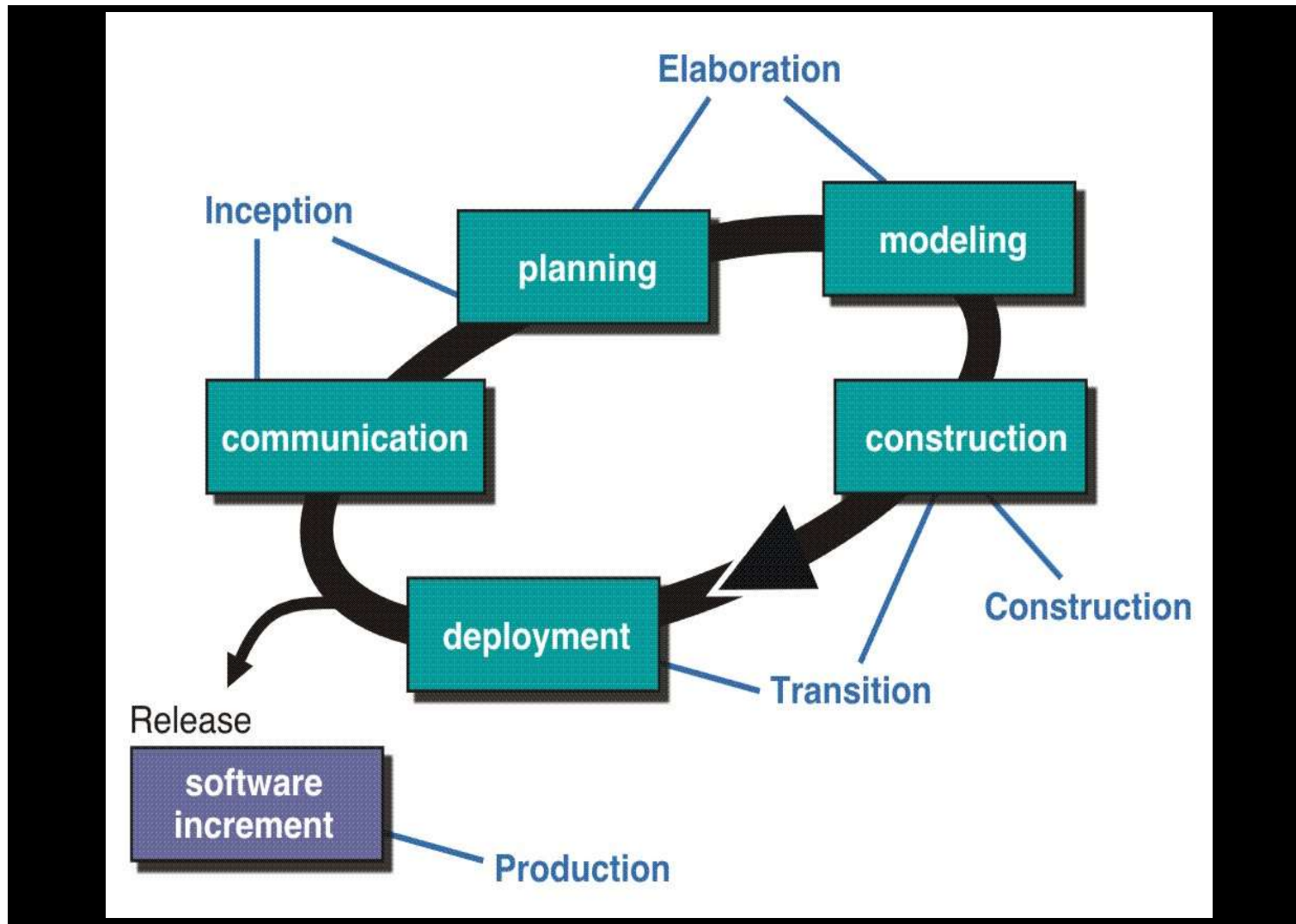
# Unified Process Model

A software process that is:

- use-case driven
- architecture-centric
- iterative and incremental

Closely aligned with the  
Unified Modeling Language (UML)

# The Unified Process (UP)



# UP Work Products

## Inception Phase

- Vision document
- Initial use-case model
- Initial project glossary
- Initial business case
- Initial risk assessment
- Project plan phases and iterations
- Business model if necessary
- One or more prototypes

## Elaboration Phase

- Use-case model
- Supplementary requirements including non-functional
- Analysis model
- Software architecture description
- Executable architectural prototype
- Preliminary design model
- Revised risk list
- Project plan including
  - iteration plan
  - adapted workflows
  - milestones
  - technical work products
- Preliminary user manual

## Construction Phase

- Design model
- Software components
- Integrated software increment
- Test plan and procedure
- Test cases
- Support documentation
  - user manuals
  - installation manuals
  - description of current increment

## Transition Phase

- Delivered software increment
- Beta test reports
- General user feedback



# Common Fears for Developers

- The project will produce the wrong product.
- The project will produce a product of inferior quality.
- The project will be late.
- We'll have to work 80 hour weeks.
- We'll have to break commitments.
- We won't be having fun.

# The Manifesto for Agile Software Development

“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.”

-- *Kent Beck et al.*

# What is “Agility”?

- Effective (rapid and adaptive) response to change
- Effective communication among all stakeholders
- Drawing the customer onto the team
- Organizing a team so that it is in control of the work performed

*Yielding ...*

- Rapid, incremental delivery of software

# An Agile Process

- Is driven by customer descriptions of what is required (scenarios)
- Recognizes that plans are short-lived
- Develops software iteratively with a heavy emphasis on construction activities
- Delivers multiple 'software increments'
- Adapts as changes occur

# Principles of Agility

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.
- Business people and developers must work together daily throughout the project.

# Principles of Agility

- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

# Principles of Agility

- Continuous attention to technical excellence and good design enhances agility.
- Simplicity - the art of maximizing the amount of work not done - is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

# Extreme Programming (XP)

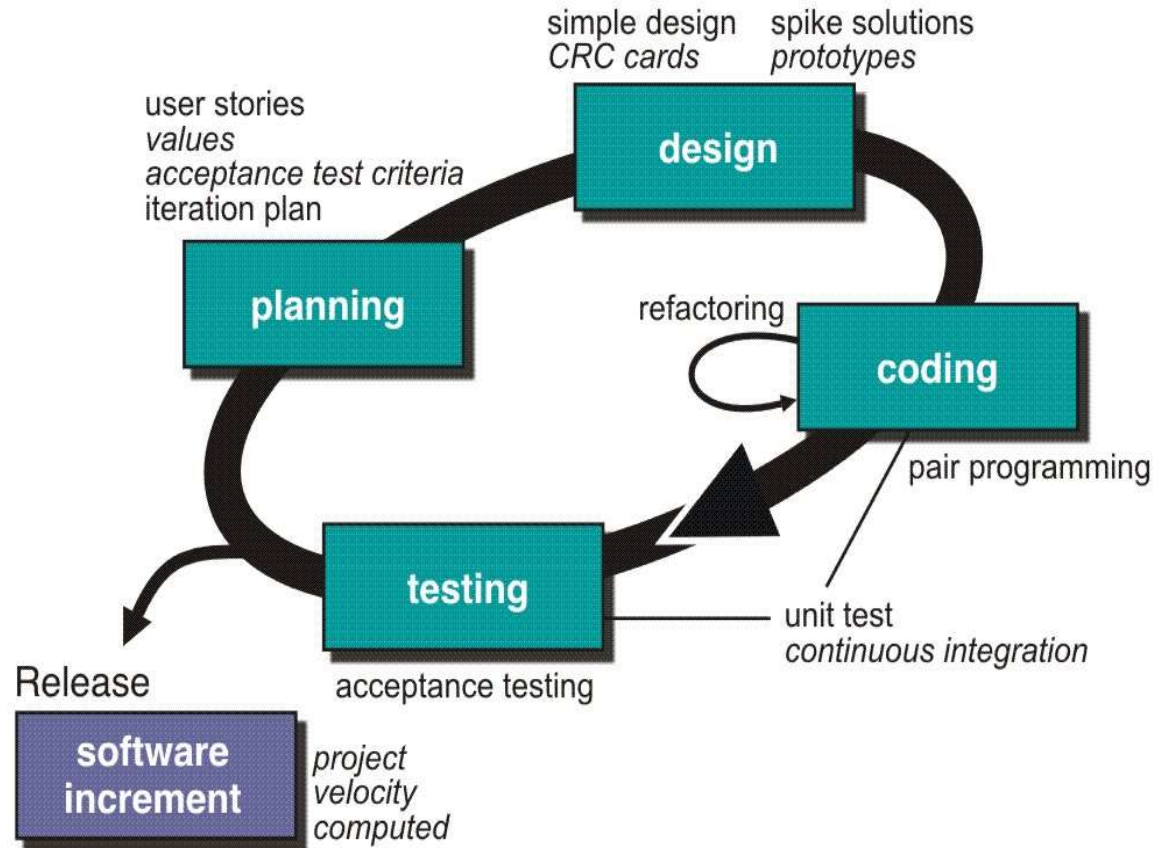
- The most widely used agile process, originally proposed by Kent Beck
- XP Planning
  - Begins with the creation of **user stories**
  - Agile team assesses each story and assigns a **cost**
  - Stories are grouped to for a **deliverable increment**
  - A **commitment** is made on delivery date
  - After the first increment **project velocity** is used to help define subsequent delivery dates for other increments



# Extreme Programming (XP)

- XP Design
  - Follows the **KIS principle**
  - Encourage the use of **CRC cards** (see Chapter 8)
  - For difficult design problems, suggests the creation of **spike solutions** — a design prototype
  - Encourages **refactoring** — an iterative refinement of the internal program design
- XP Coding
  - Recommends **the construction of a unit test** for a store *before* coding commences
  - Encourages **pair programming**
- XP Testing
  - All **unit tests are executed daily**
  - Acceptance tests** are defined by the customer and executed to assess customer visible functionality

# Extreme Programming (XP)



# Other Agile Processes

- Adaptive Software Development (ASD)
- Dynamic Systems Development Method (DSDM)
- Scrum
- Crystal
- Feature Driven Development
- Agile Modeling (AM)

## PSP and TSP

- PSP is a high-maturity process framework for individuals
- TSP addresses high-maturity practices for teams of PSP-trained engineers
- PSP & TSP provide a set of “Hows” to the CMM’s “Whats”
- PSP & TSP get individuals and teams more involved in process improvement
- PSP & TSP are sometimes referred to as Level 5 processes for individuals and team

# THE PSP PHILOSOPHY

- Use effective methods
- Recognize strengths and weaknesses
- Practice, practice, practice
- Learn from history
- Find and learn new methods
- Practice software development as an engineering discipline rather than craft

## What is the PSP?

- The PSP is a set of practices that engineers can apply to most structured personal tasks to improve predictability, quality, & productivity
- The PSP as taught contains *one set of methods that* can be effective for many
  - An excellent starting point, but not expected to be a “one size fits all” process
- Currently, few engineers practice the best available methods, negatively impacting chances of project success—PSP addresses this

# What does PSP Developer DO

- Tracks basic development process data
  - Size, time, defects, and task completion
  - Time & defects are tracked by phase, e.g., planning, design, code, personal reviews, test, postmortem
- *Uses data derived from the basic data for process management and improvement*
- Plans using historical data and tracks progress
  - “PROBE” (PROxy Based Estimating) estimating
  - “Earned Value” scheduling & tracking
  - Quality planning
- “Builds in” Quality
  - Produces verifiable designs
  - Conducts structured personal design and code reviews
- Improves development process using data

# What is the TSP?

The TSP strategy is to improve performance from the bottom up. This strategy starts with PSP training.

## **Team Member Skills**

- Process discipline
- Performance measures
- Estimating and planning skills
- Quality management skills

## **Team Building**

- Goal setting
- Role assignment
- Tailored team process
- Detailed and balanced plans

## **Team Management**

- Team communication
- Team coordination
- Project tracking
- Risk analysis
- PSP TSP

**Make CMM Level 5 behavior normal and expected**



## What does TSP Developer DO

- Developers use PSP practices for their personal work
- For each development phase (2-4 months), the team
  - Conducts a team “launch” to come to a common understanding of the project & to develop detailed plans
  - Tracks progress against schedule and quality weekly, adjusts plans, and takes immediate action if necessary to ensure commitments will be met
  - Uses team-level data the same way as developers use their individual data to assess schedule and quality
    - Quality data, Software inspections, Time on Task, Earned value , etc.
  - Conducts Postmortems to improve development process

## Process Patterns

- A Process pattern provides us with a template which provides a consistent method for describing an important characteristics of the software process
- Thus software process can be defined as a collection of patterns that define a set of activities, actions, work tasks, work products and related behaviors required to develop computer software.
- By combining patterns, a software team can construct a process that best meets the needs of a project.
- Patterns can be defined at any level of abstraction – Complete process, framework Activity, Umbrella activity, SE action or a task

# Process Patterns Template

<b>Pattern Name</b>	Meaningful Name of pattern
<b>Intent</b>	Objective of pattern
<b>Type</b>	Task, SE action, Framework activity, Umbrella activity etc
<b>Initial Context</b>	Applicable Pre-requisites
<b>Problem</b>	Problem Definition
<b>Solution</b>	Solution to the problem / process
<b>Resulting Context</b>	S/W Engg. Info. & Project info. Generated after completion
<b>Related Processes</b>	A list of other related process patterns
<b>Known Uses</b>	Where it is useful & Examples where it has been used

# Software Requirement Analysis

## Unit II

**Dr M. Durairaj**

**Associate Professor**

**School of Computer Science, Engineering and Applications**

**Bharathidasan University**

# Software Requirements

- *Software requirements* are documentation that completely describes the behavior that is required of the software-before the software is designed built and tested.
  - Requirements analysts (or business analysts) build software requirements specifications through *requirements elicitation*.
    - Interviews with the users, stakeholders and anyone else whose perspective needs to be taken into account during the design, development and testing of the software
    - Observation of the users at work
    - Distribution of discussion summaries to verify the data gathered in interviews

# Discussion Summary

- A requirements analyst can use a *discussion summary* to summarize information gathered during elicitation and validate it through a review.
- Notes gathered during the elicitation should fit into the discussion summary template
- The discussion summary outline can serve as a guide for a novice requirements analyst in leading interviews and meetings

## Discussion Summary outline

1. Project background
  - a) Purpose of project
  - b) Scope of project
  - c) Other background information
2. Perspectives
  - a) Who will use the system?
  - b) Who can provide input about the system?
3. Project Objectives
  - a) Known business rules
  - b) System information and/or diagrams
  - c) Assumptions and dependencies
  - d) Design and implementation constraints
4. Risks
5. Known future enhancements
6. References
7. Open, unresolved or TBD issues

# Use Cases

- A *use case* is a description of a specific interaction that a user may have with the system.
- Use cases are deceptively simple tools for describing the functionality of the software.
  - Use cases do not describe any internal workings of the software, nor do they explain how that software will be implemented.
  - They simply show how the steps that the user follows to use the software to do his work.
  - All of the ways that the users interact with the software can be described in this manner.

# Functional Requirements

- *Functional requirements* define the outward behavior required of the software project.
  - The goal of the requirement is to communicate the needed behavior in as clear and unambiguous a manner as possible.
  - The behavior in the requirement can contain lists, bullets, equations, pictures, references to external documents, and any other material that will help the reader understand what needs to be implemented.



# Nonfunctional Requirements

- *Nonfunctional requirements* define characteristics of the software which do not change its behavior.
  - Users have implicit expectations about how well the software will work.
  - These characteristics include how easy the software is to use, how quickly it executes, how reliable it is, and how well it behaves when unexpected conditions arise.
  - The nonfunctional requirements define these aspects about the system.
    - The nonfunctional requirements are sometimes referred to as “non-behavioral requirements” or “software quality attributes”

# Software Requirements Specification

- The *software requirements specification* (SRS) represents a complete description of the behavior of the software to be developed.
- The SRS includes:
  - A set of use cases that describe all of the interactions that the users will have with the software.
  - All of the functional requirements necessary to define the internal workings of the software: calculations, technical details, data manipulation and processing, and other specific functionality that shows how the use cases are to be satisfied
  - Nonfunctional requirements, which impose constraints on the design or implementation (such as performance requirements, quality standards or design constraints).

# Requirements vs. Design

- Many people have difficulty understanding the difference between scope, requirements and design.
  - Scope demonstrates the needs of the organization, and is documented in a vision and scope document
  - Requirements document the behavior of the software that will satisfy those needs
  - Design shows how those requirements will be implemented technically

# Change Control

- *Change control* is a method for implementing only those changes that are worth pursuing, and for preventing unnecessary or overly costly changes from derailing the project.
  - Change control is an agreement between the project team and the managers that are responsible for decision-making on the project to evaluate the impact of a change before implementing it.
  - Many changes that initially sound like good ideas will get thrown out once the true cost of the change is known.

# Change Control

- *A change control board (CCB)* is made up of the decision-makers, project manager, stakeholder or user representatives, and selected team members.
  - The CCB analyzes the impact of all requested changes to the software and has the authority to approve or deny any change requests once development is underway.
  - Before the project begins, the list of CCB members should be written down and agreed upon, and each CCB member should understand why the change control process is needed and what their role will be in it.

# Change Control

- Whenever a change is needed, the CCB follows the *change control process* to evaluate the change:
  - The potential benefit of the change is written down, and the project manager works with the team to estimate the potential impact that the change will have on the project.
  - If the benefit of the change is worth the cost, the project manager updates the plan to reflect the new estimates. Otherwise, the change is thrown out and the team continues with the original plan.
  - The CCB either accepts or rejects the change.



# UNIT-III: UNDERSTANDING XP

Dr. M. Durairaj

Associate Professor, Bharathidasan University



# Introduction/Course Description



- Introduction
  - The XP Lifecycle
  - The XP Team
  - XP Concepts
  - Adopting XP:
    - Is XP Right for US? Go!
    - Assess Your Agility




# The XP Team

Working solo on your own project—“scratching your own itch”—can be a lot of fun. There are no questions about which features to work on, how things ought to work, if the software works correctly, or whether stakeholders are happy. All the answers are right there in one brain.

Team software development is different. The same information is spread out among many members of the team. Different people know:

- • How to design and program the software (programmers, designers, and architects)
- • Why the software is important (product manager)
- • The rules the software should follow (domain experts)
- • How the software should behave (interaction designers)
- • How the user interface should look (graphic designers)
- • Where defects are likely to hide (testers)

- 
- How to interact with the rest of the company (project manager)
  - • Where to improve work habits (coach)

All of this knowledge is necessary for success.

# The Whole Team

- ❑ XP teams sit together in an open workspace.
- ❑ At the beginning of each iteration, the team meets for a series of activities: an iteration demo, a retrospective, and iteration planning.
- ❑ These typically take two to four hours in total. The team also meets for daily stand-up meetings, which usually take five to ten minutes each.
- ❑ Other than these scheduled activities, everyone on the team plans his own work.
- ❑ That doesn't mean everybody works independently; they just aren't on an explicit schedule.
- ❑ Team members work out the details of each meeting when they need to.
- ❑ Sometimes it's as informal as somebody standing up and announcing across the shared workspace that he would like to discuss an issue.
- ❑ This self-organization is a hallmark of agile teams.

# XP Concepts

- As with any specialized field, XP has its own vocabulary. This vocabulary distills several important concepts into snappy descriptions. Any serious discussion of XP (and of agile in general) uses this vocabulary. Some of the most common ideas follow.
- Prefer better to bigger.
- Fractional assignment is dreadfully counterproductive.

# Refactoring



- ❑ There are multiple ways of expressing the same concept in source code.
- ❑ Some are better than others.
- ❑ Refactoring is the process of changing the structure of code—rephrasing it—without changing its meaning or behavior.
- ❑ It's used to improve code quality, to fight off software's unavoidable entropy, and to ease adding new features.

# Technical Debt

- Technical debt is the total amount of less-than-perfect design and implementation decisions in your project.
- This includes quick and dirty hacks intended just to get something working right now! and design decisions that may no longer apply due to business changes.
- Technical debt can even come from development practices such as an unwieldy build process or incomplete test coverage.
- It lurks in gigantic methods filled with commented-out code and “TODO: not sure why this works” comments.
- These dark corners of poor formatting, unintelligible control flow, and insufficient testing breed bugs like mad.

# Timeboxing

- Some activities invariably stretch to fill the available time.
- There's always a bit more polish you can put on a program or a bit more design you can discuss in a meeting.
- Yet at some point you need to make a decision.
- At some point you've identified as many options as you ever will.
- Recognizing the point at which you have enough information is not easy.
- If you use timeboxing, you set aside a specific block of time for your research or discussion and stop when your time is up, regardless of your progress.
- This is both difficult and valuable.
- It's difficult to stop working on a problem when the solution may be seconds away.
- However, recognizing when you've made as much progress as possible is an important time-management skill.
- Timeboxing meetings, for example, can reduce wasted discussion.

# The Last Responsible Moment

- XP views a potential change as an opportunity to exploit; it's the chance to learn something significant.
- This is why XP teams delay commitment until the last responsible moment.\*
- Note that the phrase is the last responsible moment, not the last possible moment.
- As [Poppendieck & Poppendieck] says, make decisions at “the moment at which failing to make a decision eliminates an important alternative.
- If commitments are delayed beyond the last responsible moment, then decisions are made by default, which is generally not a good approach to making decisions.”
- By delaying decisions until this crucial point, you increase the accuracy of your decisions, decrease your workload, and decrease the impact of changes.
- Why? A delay gives you time to increase the amount of information you have when you make a decision, which increases the likelihood it is a correct decision.
- That, in turn, decreases your workload by reducing the amount of rework that results from incorrect decisions.
- Changes are easier because they are less likely to invalidate decisions or incur additional



# Stories

- ❑ Stories represent self-contained, individual elements of the project.
- ❑ They tend to correspond to individual features and typically represent one or two days of work.
- ❑ Stories are customer-centric, describing the results in terms of business results.
- ❑ They're not implementation details, nor are they full requirements specifications.
- ❑ They are traditionally just an index card's worth of information used for scheduling purposes.

# Iterations

---

- An iteration is the full cycle of design-code-verify-release practiced by XP teams.
- It's a timebox that is usually one to three weeks long.
- Each iteration begins with the customer selecting which stories the team will implement during the iteration, and it ends with the team producing software that the customer can install and use.

# Velocity



- ❑ In well-designed systems, programmer estimates of effort tend to be consistent but not accurate.
- ❑ Programmers also experience interruptions that prevent effort estimates from corresponding to calendar time.
- ❑ Velocity is a simple way of mapping estimates to the calendar.
- ❑ It's the total of the estimates for the stories finished in an iteration.

# Theory of Constraints

- ❑ [Goldratt 1992]'s Theory of Constraints says, in part, that every system has a single constraint
- ❑ that determines the overall throughput of the system.
- ❑ This book assumes that programmers are the constraint on your team.
- ❑ Regardless of how much work testers and customers do, many software teams can only complete their projects as quickly as the programmers can program them.
- ❑ If the rest of the team outpaces the programmers, the work piles up, falls out of date and needs reworking, and slows the programmers further.
- ❑ Therefore, the programmers set the pace, and their estimates are used for planning.
- ❑ As long as the programmers are the constraint, the customers and testers will have more slack in their schedules, and they'll have enough time to get their work done before the programmers need it.

# Mindfulness

- Agility—the ability to respond effectively to change—requires that everyone pay attention to the process and practices of development.
- This is mindfulness.
- Sometimes pending changes can be subtle.
- You may realize your technical debt is starting to grow when adding a new feature becomes more difficult this week than last week.
- You may notice the amount and tone of feedback you receive from your customers change.
- XP offers plenty of opportunities to collect feedback from the code, from your coworkers, and from every activity you perform.
- Take advantage of these.
- Pay attention.
- See what changes and what doesn't and discuss the results frequently

# Adopting XP

- “I can see how XP would work for IT projects, but product development is different.” —a product development team
- “I can see how XP would work for product development, but IT projects are different.” —an in-house IT development team
- Before adopting XP, you need to decide whether it’s appropriate for your situation.
- Often, people’s default reaction to hearing about XP is to say, “Well, of course that works for other teams, but it couldn’t possibly work for us.”

# Is XP Right for Us?

- ❑ You can adopt XP in many different conditions, although the practices you use will vary depending on your situation.
- ❑ The practices in this book were chosen to give you the greatest chance of success.
- ❑ That leads to some prerequisites and recommendations about your team's environment.
- ❑ You don't have to meet these criteria exactly, but it's worth trying to change your environment so that you do.
- ❑ This will give you the best chance of succeeding.

# UNIT-IV

# PRACTICING XP

---

Dr. M. Durairaj  
Bharathidasan University



# Practicing XP

- A. Thinking
- B. Collaborating
- C. Releasing
- D. Planning
- E. Developing

# COLLABORATING

1. Trust
2. Sit Together
3. Real Customer
4. Ubiquitous Language
5. Stand-Up Meetings
6. Coding Standards
7. Iteration Demo
8. Reporting

# Trust

- Team Strategy 1 – Customer-Programmer Empathy
- Team Strategy 2 – Programmer-Tester Empathy
- Team Strategy 3 – Eat Together
- Team Strategy 4 – Team Continuity
- Impressions
- Organizational Strategy 1 :
  - Show Some Hustle
- Organizational Strategy 2 :
  - Deliver on Commitments

- Organizational Strategy 3 :
- Manage Problems
- Organizational Strategy 4 :
- Respect Customer Goals
- Organizational Strategy 5 :
- Promote the Team
- Organizational Strategy 6 :
- Be Honest

# Sit Together

- Accommodating Poor Communication
- A Better Way
- Exploiting Great Communication
- Secrets of Sitting Together
- Making Room
- Designing Your Workspace
- Sample Workspaces
- Adopting an Open Workspace

# Real Customer Involvement

- Personal Development
- In-House Custom Development
- Vertical-Market Software
- Horizontal-Market Software

# Ubiquitous Language

You need a ubiquitous language to explain business logic to a nonprogrammer domain expert.

- The Domain Expertise Conundrum
- Two Languages
- How to Speak the Same Languages
- Ubiquitous Language in Code
- Refining the Ubiquitous Language

# Stand-Up Meetings

- We know what our teammates are doing.
- I have a special antipathy for status meetings.
- You know—a manager reads a list of tasks and asks about each one in turn.
- They seem to go on forever, although my part in them is typically only five minutes.
- I learn something new in perhaps 10 of the other minutes.
- The remaining 45 minutes are pure waste.
- There's a good reason that organizations hold status meetings: people need to know what's going on.
- XP projects have a more effective mechanism: informative workspaces and the daily stand-up meeting.
- How to Hold a Daily Stand-Up Meeting
- Be Brief



# Coding Standards

- Beyond Formatting
- How to Create a Coding Standard
- Dealing with Disagreement
- Adhering to the Standard

# Iteration Demo

It is a powerful way to show off your work, and stakeholders are happy to see product.

- How to Conduct an Iteration Demo
- Two Key Questions
- Weekly Deployment Is Essential

# Reporting

- Types of Reports
- Progress Reports to Provide
  - Vision Statement
  - Weekly demo
  - Release and iteration plans
  - Burn-up chart
- Progress Reports to Consider
  - Roadmap
  - Status email
- Management Reports to Consider
  - Productivity
  - Throughput
  - Defects
  - Time usage
- Reports to Avoid
  - Source lines of code (SLOC) and function points
  - Number of stories
  - Velocity
  - Code quality

# RELEASING

**What is the value of code? Agile developers value “working software over comprehensive documentation.”\* Does that mean a requirements document has no value? Does it mean unfinished code has no value?**

1. “Done Done”
2. No Bugs
3. Version Control
4. Ten-Minute Build
5. Continuous Integration
6. Collective Code Ownership
7. Documentation

1. **"done done"** ensures that completed work is ready to release.
2. **No bugs** allows you to release your software without a separate testing phase.
3. **Version control** allows team members to work together without stepping on each other's toes.
4. **A ten-minute** build builds a tested release package in under 10 minutes.
5. **Continuous integration** prevents a long, risky integration phase.
6. **Collective code ownership** allows the team to solve problems no matter where they may lie.
7. **Post-hoc documentation** decreases the cost of documentation and increases its accuracy.

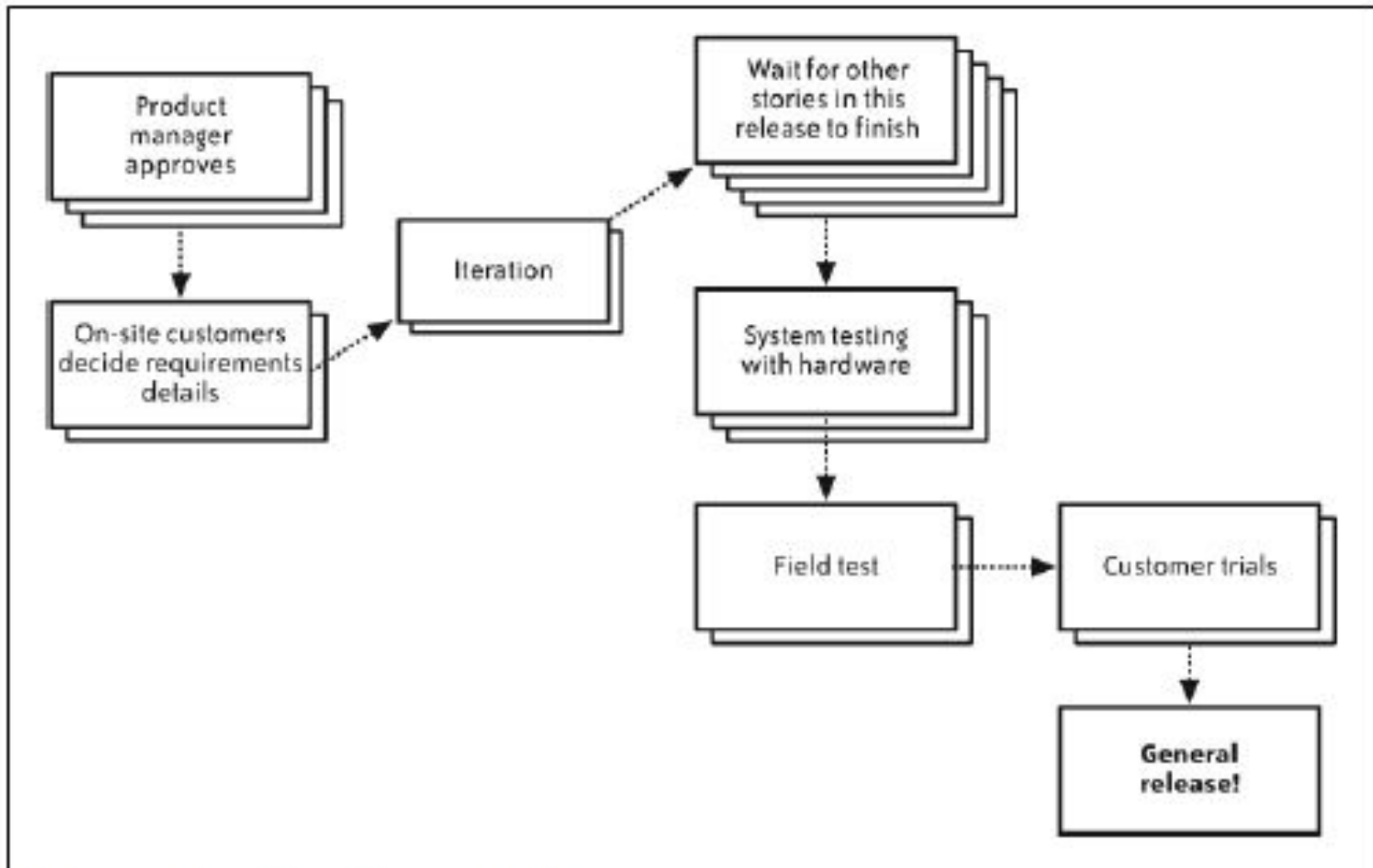


Figure 7-2. A sample value stream map

# PLANNING

1. Vision
2. Release Planning
3. The Planning Game
4. Risk Management
5. Iteration Planning
6. Slack
7. Stories
8. Estimating

1. **Vision** reveals where the project is going and why it's going there.
2. **Release Planning** provides a roadmap for reaching your destination.
3. **The Planning Game** combines the expertise of the whole team to create achievable plans.
4. **Risk Management** allows the team to make and meet long-term commitments.
5. **Iteration Planning** provides structure to the team's daily activities.
6. **Slack** allows the team to reliably deliver results every iteration.
7. **Stories** form the line items in the team's plan.
8. **Estimating** enables the team to predict how long its work will take.



# DEVELOPING

1. **Incremental Requirements** allows the team to get started while customers work out requirements details.
2. **Customer Tests** help communicate tricky domain rules.
3. **Test-Driven Development** allows programmers to be confident that their code does what they think it should.
4. **Refactoring** enables programmers to improve code quality without changing its behavior.
5. **Simple Design** allows the design to change to support any feature request, no matter how surprising.

# Contd..

8. **Incremental Design and Architecture** allows programmers to work on features in parallel with technical infrastructure.
9. **Spike Solutions** use controlled experiments to provide information.
10. **Performance Optimization** uses hard data to drive optimization efforts.
11. **Exploratory Testing** enables testers to identify gaps in the team's thought processes.

# Unit – V

## Deliver Value



The Art of

# Agile Development



**DR. M. DURAIRAJ**  
**BHARATHIDASAN UNIVERSITY**

# Unit V -Deliver Value



## Deliver Value :

- Exploit Your Agility,
- Only Releasable Code Has Value,
- Deliver Business Results,
- Deliver frequently,

## Seek Technical Excellence :

- Software Doesn't Exist,
- Design Is for Understanding,
- Design Trade-offs,
- Quality with a Name,
- Great Design,
- Universal Design Principles,
- Principles in Practice,
- Pursue Mastery

## **Deliver Value**



- Your software only begins to have real value when it reaches users.
- Only at that point do you start to generate trust, to get the most important kinds of feedback, and to demonstrate a useful return on investment.
- That's why successful agile projects deliver value early, often, and repeatedly.

## ○ Exploit Your Agility



- Simplicity of code and process are aesthetically pleasing.
- Yet there's a more important reason why agility helps you create great software:
  - it improves your ability to recognize and take advantage of new opportunities.
- If you could predict to the hour how long your project would take, know what risks would and wouldn't happen, and completely eliminate all surprises, you wouldn't need agility
  - —you would succeed with any development method.



- **In Practice**

- XP exploits agility by removing the time between taking an action and observing its results, which improves your ability to learn from this feedback.
- This is especially apparent when the whole team sits together.

- **Beyond Practices**

# ○ Only Releasable Code Has Value



- Having the best, most beautiful code in the world matters very little unless it does what the customer wants.
- It's also true that having code that meets customer needs perfectly has little value unless the customer can actually use it.
- Until your software reaches the people who need it, it has only potential value.
- Delivering actual value means delivering real software.
- Un-releasable code has no value.
- Working software is the primary measure of your progress.
- At every point, it should be possible to stop the project and have actual value proportional to your investment in producing the software





- **In Practice**

- The most important practice is that of “done done,” where work is either complete or incomplete.
- This unambiguous measure of progress immediately lets you know where you stand.
- Practicing “no bugs” is a good reminder that deferring important and specific decisions decreases the project’s value—especially if those decisions come directly from real customer feedback.

- **Beyond Practices**

# ○ Deliver Business Results



- What if you could best meet your customer's need without writing any software? Would you do it? Could you do it?
- Someday that may happen to you.
- It may not be as dramatic as telling a recurring customer that he'll get better results if you don't write software, but you may have to choose between delivering code and delivering business results.
- Value isn't really about software, after all.
- Your goal is to deliver something useful for the customer.
- The software is merely how you do that.
- The single most essential criterion for your success is the fitness of the project for its business purposes.
- Everything else is secondary—not useless by any means, but of lesser importance.



- **In Practice**

- XP encourages close involvement with actual customers by bringing them into the team, so they can measure progress and make decisions based on business value every day.
- Real customer involvement allows the on-site customer to review these values with end-users and keep the plan on track.
- Their vision provides answers to the questions most important to the project.

- **Beyond Practices**

# ○ Deliver frequently



- If you have a business problem, a solution to that problem today is much more valuable than a solution to that problem in six months—especially if the solution will be the same then as it is now.
- Value is more than just doing what the customer needs.
- It's doing what the customer needs when the customer needs it.
- Delivering working, valuable software frequently makes your software more valuable.
- This is especially true when a real customer promotes the most valuable stories to the start of the project.
- Delivering working software as fast as possible enables two important feedback loops.
- One is from actual customers to the developers, where the customers use the software and communicate how well it meets their needs.
- The other is from the team to the customers, where the team communicates by demonstrating how trustworthy and capable it is.



- **In Practice**

- Once you've identified what the customer really needs and what makes the software valuable, XP's technical practices help you achieve fast and frequent releases.
- Short iterations keep the schedule light and manageable by dividing the whole project into week-long cycles, culminating in a deliverable project demonstrated in the iteration demo.
- This allows you to deliver once a week, if not sooner.

- **Beyond Practices**

# Seek Technical Excellence



- I like logical frameworks and structures.
- When I think about technical excellence, I can't help but wonder: "What's the intellectual basis for design? What does it mean to have a good design?"
- Unfortunately, many discussions of "good" design focus on specific techniques.
- These discussions often involve assumptions that one particular technology is better than another, or that rich object-oriented domain models or stored procedures or service-oriented architectures are obviously good.
- With so many conflicting points of view about what's obviously good, only one thing is clear:
- good isn't obvious.

# Seek Technical Excellence



- Software Doesn't Exist
- Design Is for Understanding
- Design Trade-offs
- Quality with a Name
- Great Design
- Universal Design Principles
- Principles in Practice
- Pursue Mastery

# • Software Doesn't Exist



- Let me digress for a moment: software doesn't exist. OK, I exaggerate—but only slightly.
- When you run a program, your computer loads a long series of magnetic fields from your hard drive and translates them into capacitances in RAM.
- Transistors in the CPU interpret those charges, sending the results out to peripherals such as your video card. More transistors in your monitor selectively allow light to shine through colored dots onto your screen.
- Yet none of that is software. Software isn't even ones and zeros; it's magnets, electricity, and light.
- The only way to create software is to toggle electrical switches up and down—or to use existing software to create it for you.
- You write software, though, don't you?
- Actually, you write a very detailed specification for a program that writes the software for you.
- This special program translates your specification into machine instructions, then directs the computer's operating system to save those instructions as magnetic fields on the hard drive.
- Once they're there, you can run your program, copy it, share it, or whatever.



# • Design Is for Understanding



- If source code is design, then what is design?
- Why do we bother with all these UML diagrams and CRC cards and discussions around a whiteboard?
- All these things are abstractions—even source code.
- The reality of software's billions of evanescent electrical charges is inconceivably complex, so we create simplified models that we can understand.
- Some of these models, like source code, are machine-translatable.
- Others, like UML, are not—at least not yet.
- Early source code was assembly language: a very thin abstraction over the hardware.
- Programs were much simpler back then, but assembly language was hard to understand.
- Programmers drew flow charts to visualize the design.
- Why don't we use flow charts anymore?
- Our programming languages are so much more expressive that we don't need them!
- You can read a method and see the flow of control.

# • Design Trade-offs



- When the engineers at Boeing design a passenger airplane, they constantly have to trade off safety, fuel efficiency, passenger capacity, and production cost.
- Programmers rarely have to make those kinds of decisions these days.
- The assembly programmers of yesteryear had tough decisions between using lots of memory (space) or making the software fast (speed).
- Now, we almost never face such speed/space trade-offs. Our machines are so fast and have so much RAM that once-beloved hand optimizations rarely matter.
- In fact, our computers are so fast that modern languages actually waste computing resources.
- With an optimizing compiler, C is just as good as assembly language.
- C++ adds virtual method lookups—requiring more memory and an extra level of indirection.
- Java and C# add a complete intermediate language that runs in a virtual machine atop the normal machine.
- Ruby\* interprets the entire program on every invocation!
- How wasteful.
- So why is Ruby on Rails so popular? How is it possible that Java and C# succeed?
- What do they provide that makes their waste worthwhile?
- Why aren't we all programming in C?

# • Quality with a Name



- A good airplane design balances the trade-offs of safety, carrying capacity, fuel consumption, and manufacturing costs.
- A great airplane design gives you better safety, and more people, for less fuel, at a cheaper price than the competition.
- What about software? If we're not balancing speed/space trade-offs, what are we doing?
- Actually, there is one trade-off that we make over and over again. Java, C#, and Ruby demonstrate that we are often willing to sacrifice computer time in order to save programmer time and effort.
- Some programmers flinch at the thought of wasting computer time and making “slow” programs.
- However, wasting cheap computer time to save programmer resources is a wise design decision.
- Programmers are often the most expensive component in software development.
- If good design is the art of maximizing the benefits of our trade-offs—and if software design's only real trade-off is between machine performance and programmer time—then the definition of “good software design” becomes crystal clear:
- A good software design minimizes the time required to create, modify, and maintain the software while achieving acceptable runtime performance.

# • Great Design



- Equating good design with the ease of maintenance is not a new idea, but stating it this way leads to some interesting conclusions:
- **1. Design quality is people-sensitive.**
  - Programmers, even those of equivalent competence, have varying levels of expertise.
  - A design that assumes Java idioms may be incomprehensible to a programmer who's only familiar with Perl, and vice versa.
  - Because design quality relies so heavily on programmer time, it's very sensitive to which programmers are doing the work. A good design takes this into account.
- **2. Design quality is change-specific.**
  - Software is often designed to be easy to change in specific ways.
  - This can make other changes difficult.
  - A design that's good for some changes may be bad in others.
  - A genuinely good design correctly anticipates the changes that actually occur.
- **3. Modification and maintenance time are more important than creation time.**
  - It bears repeating that most software spends far more time in maintenance than in initial development.
  - When you consider that even unreleased software often requires modifications to its design, the importance of creation time shrinks even further.
  - A good design focuses on minimizing modification and maintenance time over minimizing creation time.



- **4. Design quality is unpredictable.**
  - If a good design minimizes programmer time, and it varies depending on the people doing the work and the changes required, then there's no way to predict the quality of a design.
  - You can have an informed opinion, but ultimately the proof of a good design is in how it deals with change.
  
- **Furthermore, great designs:**
  - Are easy to modify by the people who most frequently work within them
  - Easily support unexpected changes
  - Are easy to maintain
  - Prove their value by becoming steadily easier to modify over years of changes and upgrades

# • Universal Design Principles



- In the absence of design quality measurements, there is no objective way to prove that one design approach is better than another.
- Still, there are a few universal principles—which seem to apply to any programming language or platform—that point the way.
- None of these ideas are my invention.
- How could they be? They're all old, worn, well-loved principles.
- They're so old you may have lost track of them amidst the incessant drum-beating over new fads. Here's a reminder.



- The Source Code Is the (Final) Design
- Don't Report Yourself (DRY)
- Be Cohesive
- Decouple
- Clarify, Simplify, and Refine
- Fail Fast
- Optimize from Measurements
- Eliminate Technical Debt

# • Principles in Practice



- These universal design principles provide good guidance, but they don't help with specific languages or platforms.
- Every design decision occurs in the context of the whole design—the problem domain, other design decisions, the time schedule, other team members' capabilities, etc.
- Context makes every piece of specific design advice suspect.
- Yes, you should listen to it—there's a lot of wisdom out there—but exercise healthy skepticism.
- Ask yourself, “When is this not true?” and “What is the author assuming?”
- Consider the simple and popular “instance variables must be private” design rule.
- As one of the most widely repeated design rules, it often gets applied without real thought.
- That's a shame because without context, the rule is meaningless and easily misused.



## • Pursue Mastery



- A good software design minimizes the time required to create, modify, and maintain the software while achieving acceptable runtime performance.
- This definition, and the conclusions it leads to keep in mind when considering a design.
- Follow some core design principles, and have some techniques that are useful for the languages you work with.
- However, have a willing to throw away even the design principles if they get in the way of reducing programmer time and, most importantly, solving real customer problems.
- The same is true of agile software development.
- Ultimately, what matters is success, however you define it.



- The practices, principles, and values are merely guides along the way.
- Start by following the practices rigorously.
- Learn what the principles mean.
- Break the rules, experiment, see what works, and learn some more.
- Share your insights and passion, and learn even more.
- Over time, with discipline and success, even the principles will seem less important.
- When doing the right thing is instinct and intuition, finely honed by experience, it's time to leave rules and principles behind.
- When you produce great software for a valuable purpose and pass your wisdom on to the next generation of projects, you will have mastered the art of successful software development.

# End of Unit V



The Art of

# Agile Development

## Unit V

# Mastering Agility

Dr. M. Durairaj

Bharathidasan University

# Values and Principles

- **Commonalities**

- Can any set of principles really represent agile development?
- After all, agility is just an umbrella term for a variety of methods, most of which came about long before the term “agile” was coined.
- The answer is yes: agile methods do share common values and principles.
- Five themes:
  1. Improve the Process,
  2. Rely on People,
  3. Eliminate Waste,
  4. Deliver Value, and
  5. Seek Technical Excellence.
- Each is compatible with any of the specific agile methods.

# Mastering Agility

- **Values and Principles:**
  - ❖ Commonalities,
  - ❖ About Values, Principles, and Practices,
  - ❖ Further Reading,
- **Improve the Process:**
  - ❖ Understand Your Project,
  - ❖ Tune and Adapt,
  - ❖ Break the Rules,
- **Rely on People :**
  - ❖ Build Effective Relationships,
  - ❖ Let the Right People Do the Right Things,
  - ❖ Build the Process for the People,
- **Eliminate Waste :**
  - ❖ Work in Small,
  - ❖ Reversible Steps,
  - ❖ Fail Fast,
  - ❖ Maximize Work Not Done,
  - ❖ Pursue Throughput

# • Values and Principles:

- No process is perfect. Every approach to development has some potential for improvement.
- Ultimately, your goal is to remove every barrier between your team and the success of your project, and fluidly adapt your approach as conditions change. **That is agility.**

❖ Commonalities,

❖ About Values, Principles, and Practices,

❖ Further Reading,

# • Improve the Process:

- Agile methods are more than a list of practices to follow.
- When your team has learned how to perform them effectively, you can become a great team by using the practices to modify your process.

❖ Understand Your Project,

❖ Tune and Adapt,

❖ Break the Rules,



# • Rely on People :

- Agile methods put people and their interactions at the center of all decisions.
- How can we best work together?
- How can we communicate effectively?
- Successful software projects must address these questions.
  - ❖ Build Effective Relationships,
  - ❖ Let the Right People Do the Right Things,
  - ❖ Build the Process for the People,

# • Eliminate Waste :

Agility requires flexibility and a lean process, stripped to its essentials. Anything more is wasteful. Eliminate it! The less you have to do, the less time your work will take, the less it will cost, and the more quickly you will deliver.

## ❖ Work in Small, Reversible Steps

- ❖ The easiest way to reduce waste is to reduce the amount of work you may have to throw away.

## ❖ Fail Fast

- Failure is another source of waste. Unfortunately, the only way to avoid failure entirely is to avoid doing anything worthwhile.
- That's no way to excel.

## ❖ Maximize Work Not Done

- Simplicity is the art of maximizing the work not done.”
- This idea is central to eliminating waste. To make your process more agile, do less.

## ❖ Pursue Throughput

- ❖ To minimize partially done work and wasted effort, maximize your throughput.

END OF THE UNIT V