VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

SWE 2029 – Agile Development Process

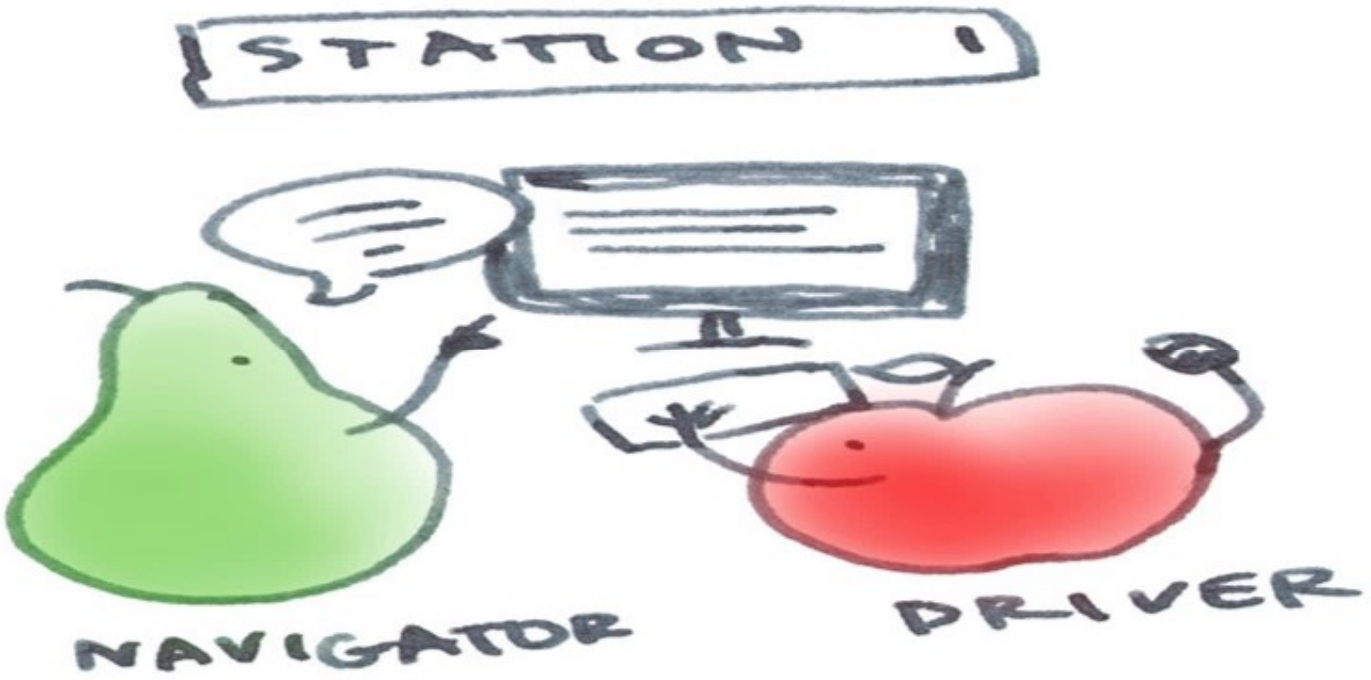
Module – IV

AGILE METHODOLOGIES

Dr. Rajesh M
School of Computer Science and Engineering
Vellore Institute of Technology [VIT]
Chennai, India.

Module	Topics	L Hrs
IV	AGILE METHODOLOGIES: Pair Programming – Refactoring – Dynamic Systems Development (DSD) – Feature Driven Development (FDD) – Test Driven Development (TDD), Agile Unified Process(AUP) – Agile Failure Models - Various reasons why agile fails?	8

Pair Programming

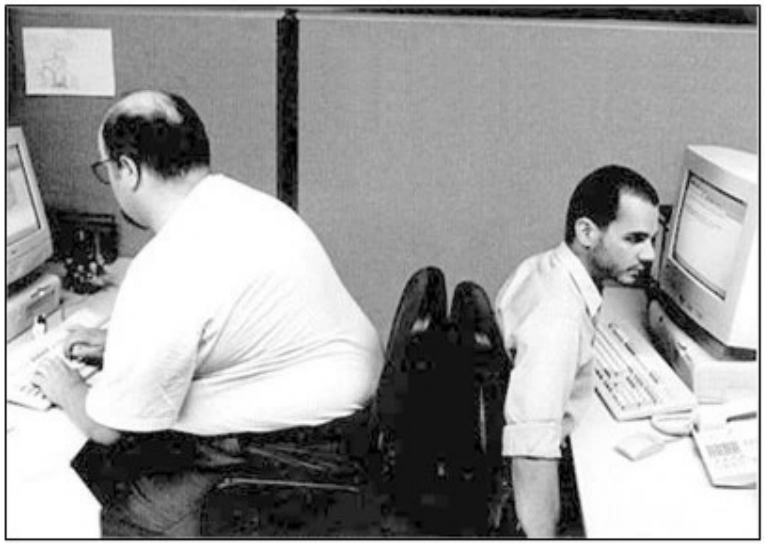


Pair Programming

- Two software engineers work on one task at one computer
  - **The driver** has control of the keyboard and mouse and *creates the implementation*
  - **The navigator** *watches* the driver's implementation
    - Identifies defects and participates in on-demand brainstorming
  - The *roles* of driver and observer are *periodically rotated*

Pair Programming

- Pairs produce *higher quality code*
- Pairs *complete their tasks faster*
- Pairs enjoy their work more
- Pairs *feel more confident* in their work



How It Works for Me?

- Pair programming is great for complex and critical logic
  - When developers *need good concentration*
  - Where *quality is really important*
  - Especially during design
  - *Reduces time wasting*

- Trivial tasks can be done alone

- Peer reviews instead pair programming is often alternative

Pair Programming

- Two Developers, One monitor, One Keyboard
- One "drives" and the other thinks
- Switch roles as needed

Pair Programming – Advantages

- Two heads are better than one
- **Focus**
- Two people are more likely to answer the following questions:
  - Is this whole approach going to work?
  - What are some test cases that may not work yet?
  - Is there a way to simplify this?

Pair Programming – Disadvantages

- *Many tasks really don't require* two programmers
- A hard sell to the customers
- Not for everyone

Refactoring

Refactoring

- *Improve the design of existing code without changing its functionality*
  - Relies on unit testing to ensure the code is not broken

- Bad smells in code:
  - *Long method / class*
  - *Duplicate code*
  - Methods does several different things (bad cohesion)
  - *Too much dependencies* (bad coupling)
  - *Complex / unreadable code*

How It Works for Me?

- *Delivering working software faster* is important!
  - You can write the code to run somehow
    - With *simple design*
    - With *less effort*
  - Later you can refactor the code if necessary
- Refactoring is not a reason to intentionally write bad code!
  - *Good coding style* is always important!

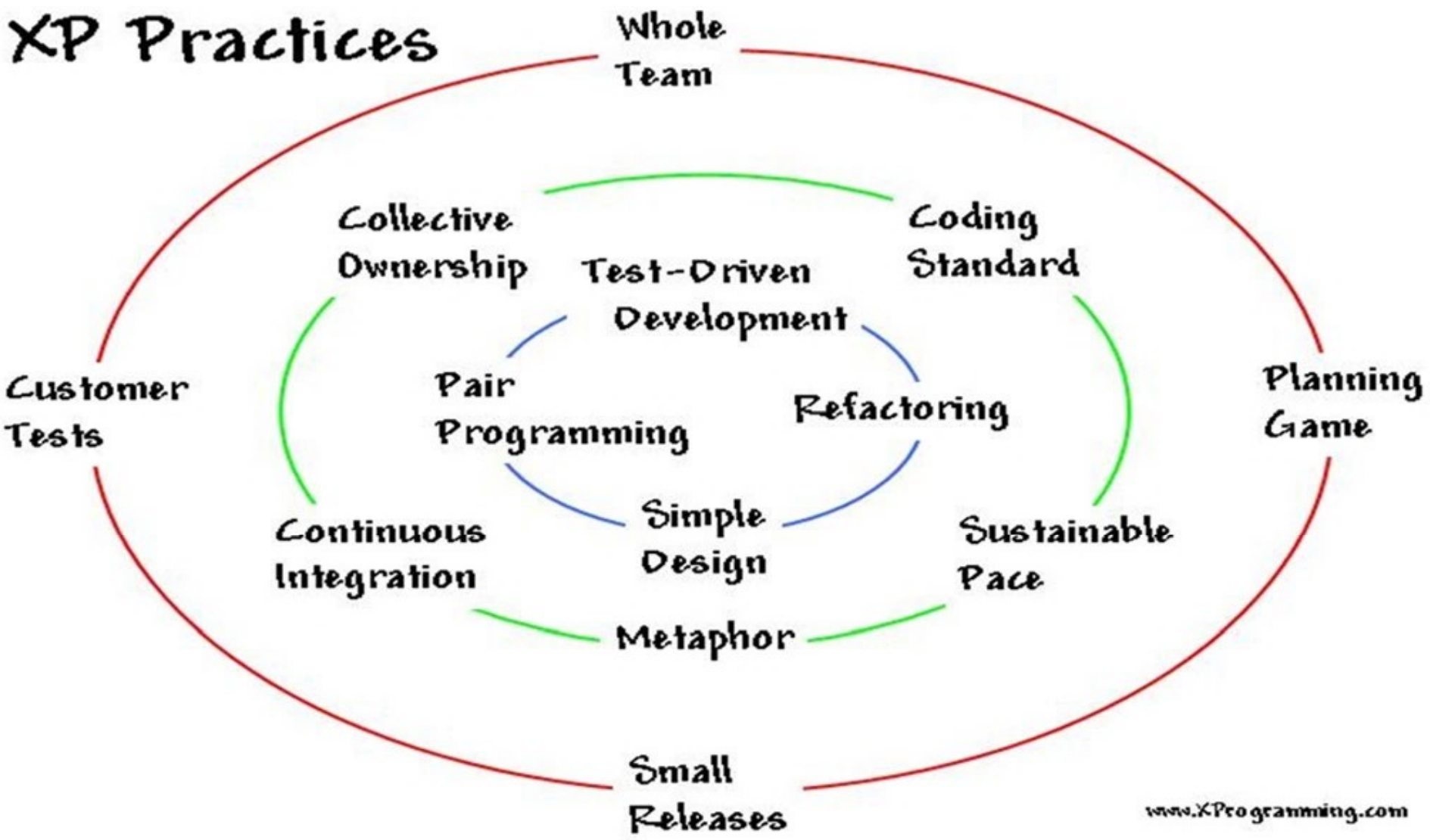
XP Practices: Whole Team

- *All contributors* to an XP project are *one team*
- Must include a business representative--the **'Customer'**
  - **Provides requirements**
  - **Sets priorities**
  - **Steers project**
- Team members are **programmers, testers, analysts, coach, manager**
- Best XP teams have **no specialists**

The 12 Key Practices

- The Planning Game
- Small Releases
- Metaphor
- Simple Design
- Test-Driven Development
- Refactoring
- Pair Programming
- Collective Ownership
- Continuous Integration
- 40-Hour Workweek
- On-site Customer
- Coding Standards

XP Practices



XP Rules

Extreme Programming (XP):

- Most popular agile method, next to SCRUM;

12 rules of Extreme programming(XP) are:

- ***The planning game***: *Start of each iteration*, customers, managers, and developers meet to *flesh out, estimate, and prioritize requirements for the next release*. The requirements are called "user stories" and are captured on "story cards".

XP – Rule's – contd..

- ***Small releases***: An *initial version* of the system is put into production *after the first few iterations* from every few days to few weeks.

- ***Metaphor***: *Customers, managers, and developers construct a metaphor*, or set of metaphors after which to model the system.

- ***Simple design***: Developers are urged to keep design *as simple as possible*, "say everything once and only once".

XP – Rule's – contd..

- ***Tests***: *Developers* work test-first; that is, they *write acceptance tests* for their code before they write the code itself. *Customers* write *functional tests* for each iteration and at the end of each iteration, all tests should run.

- ***Refactoring***: As developers work, the design should be evolved to keep it as simple as possible.

XP – Rule's – contd..

- ***Pair programming***: Two developers sitting at the same machine write all code.

- ***Continuous integration***: Developers integrate new code into the system as often as possible.

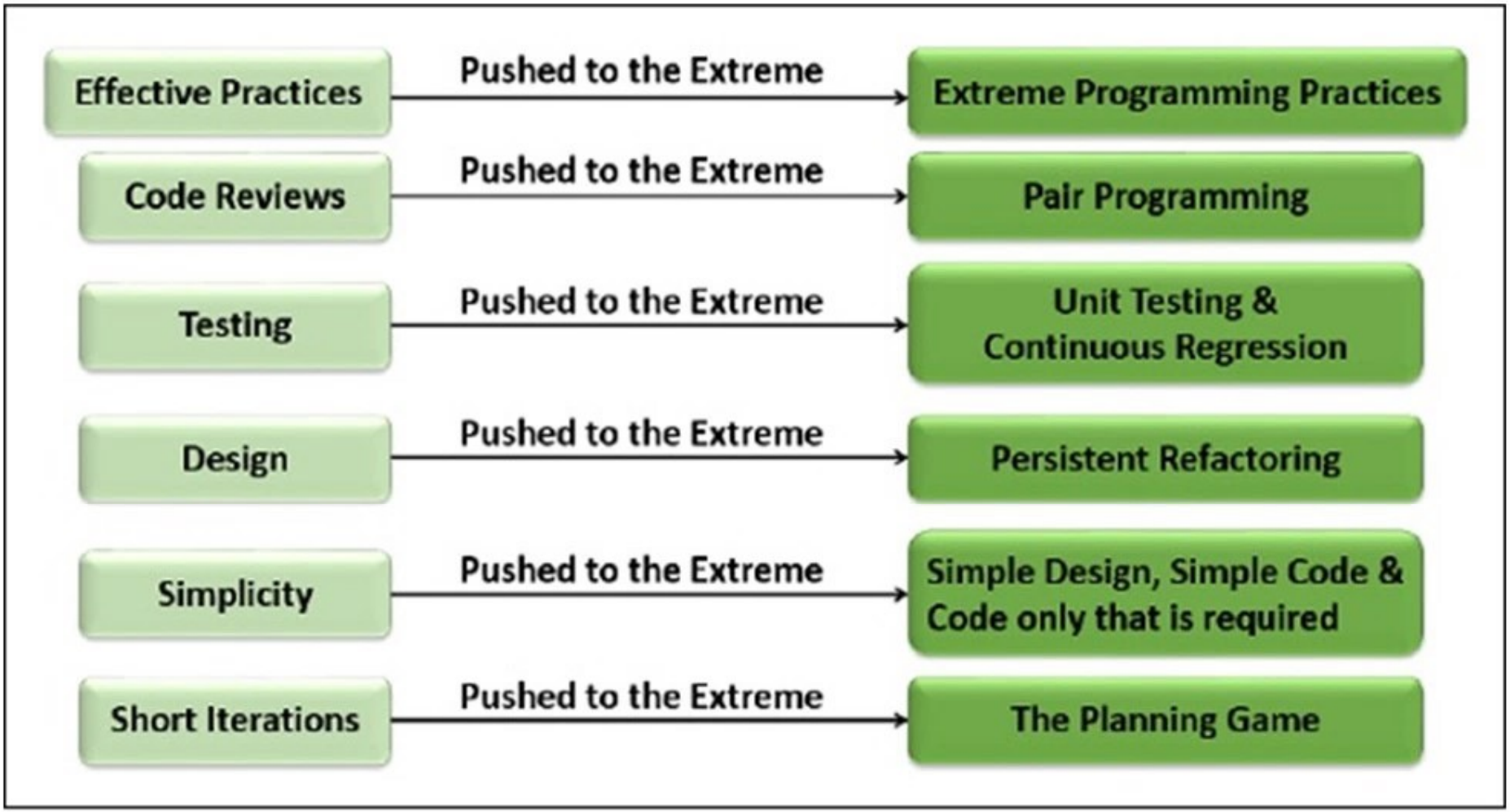
- ***Collective ownership***: The code is owned by all developers, and they may make changes anywhere in the code at anytime they feel necessary.

XP – Rule's – contd..

- ***On-site customer***: A customer works with the development team at all times to answer questions, perform acceptance tests, and **ensure that development is progressing as expected**.

- ***40-hour weeks***: Requirements should be selected for each iteration such that developers do not need to put in overtime.

Why is it called "Extreme?"



1 - The Planning Game

- *Planning* for the *upcoming iteration*
- *User stories provided by the customer*
- Technical persons determine *schedules, estimates, costs,* etc
- A result of *collaboration between the customer and the developers*

The Planning Game
– Advantages / Limitations

Advantages:

- *Reduction in time* wasted on useless features
- *Greater customer appreciation* of the cost of a feature
- *Less guesswork* in planning

Limitations:

- *Customer availability*
- Is *planning* this *often necessary*?

2- Small Releases

- *Small* in terms of *functionality*
- *Less functionality* means *releases happen more frequently*
- Support the planning game

Small Releases
– Advantages / Limitations

Advantages:

- *Frequent feedback*
- *Tracking*
- *Reduce chance of* overall *project slippage*

Limitations:

- *Not easy for all projects*
- *Not needed* for all projects
- Versioning issues

3 – Metaphor

- The *oral architecture of the system*
- A common set of terminology

Advantages:

- *Encourages a common set of terms* for the system
- *Reduction of buzz words* and jargon
- *A quick and easy way to explain* the system

Limitations:

- Often the **metaphor** is the **system**
- Another opportunity for *miscommunication*
- *The system is often not well understood* as a metaphor

4 – Simple Design

- *Do as little as needed*, nothing more;

Advantages:

- *Time is not wasted* adding superfluous functionality
- *Easier to understand* what is going on
- *Refactoring* and *collective ownership* is made possible
- *Helps* keeps *programmers on track*

Limitations:

- What is "simple?"
- *Simple isn't always best*

6 – Testing

- *Unit testing*
- *Test-first design*
- *All automated*

Advantages:

- Unit testing promote testing completeness
- Test-first gives developers a goal
- Automation gives a suite of regression test

Limitations:

- Automated unit testing isn't for everything
- A test result is only as good as the test itself

7 – Pair Programming

- *Two Developers*, One monitor, One Keyboard
- *One "drives"* and the *other thinks;*
- *Switch roles* as needed.

Pair Programming – Advantages

- *Two heads are better than one*
- *Focus*
- Two people are more likely to answer the following questions:
  - Is this whole approach going to work?
  - What are some test cases that may not work yet?
  - Is there a way to simplify this?

Pair Programming – Disadvantages

- *Many tasks really don't require* two programmers
- A hard sell to the customers
- *Not for everyone*

8 – Collective Ownership

- The idea that *all developers own all of the code*
- Enables refactoring

Collective Ownership – Advantages

- *Helps mitigate the loss of a team member* leaving
- *Promotes developers to take responsibility for the system as a whole* rather then parts of the system

Collective Ownership - Disadvantages

- *Loss of accountability*
- *Limitation to* how much of a large system that an *individual can practically "own"*

9 – Continuous Integration

- New features and changes are worked into the system immediately
- Continuous Integration (CI) is a **development practice** that **requires developers to integrate code into a shared repository several times a day.**
- **Each check-in is then verified** by an automated build, **allowing teams to detect problems early.**

Continuous Integration - Advantages

- *Reduces the lengthy process*,
- *Risk is reduced*, Able to *catch bugs quickly*,
- **Enables the Small Releases practice**

Continuous Integration – Disadvantages

- The *one day limit is not always practical*
- *Reduces the importance* of a well-thought-out architecture

10 – 40-Hour Week

- The *work week* should be *limited to 40 hours*
- Regular overtime is a symptom of a problem and not a long term solution

40-Hour Week – Advantage

- *Most developers lose effectiveness* past 40-Hours
- Value is placed on the developers well-being
- *Management is forced to find real solutions*

40-Hour Week
- Disadvantages

- The *underlying principle is flawed*
- 40-Hours is a magic number
- *Some may like to work more than 40-Hours*

11 – On-Site Customer

- Just like the title says!
- **Acts to "steer"** the project
- *Gives quick and continuous feedback to the development team*

On-Site Customer – Advantages

- *Can give quick and knowledgeable answers* to real development questions
- *Makes sure* that *what is developed is what is needed*
- *Functionality is prioritized* correctly

On-Site Customer – Disadvantages

- *Difficult to get an On-Site Customer*
- The On-Site customer that is given *may not be fully knowledgeable about the project*
- *May not have authority to make many decisions*
- *Loss of work* to the customer's company

12 – Coding Standards

- *All code should look the same*

- It should not possible to determine who coded what based on the code itself.

Coding Standards
– Advantages / Limitations

Advantages:

- *Reduces the amount of time developers spend* reformatting other peoples' code
- *Reduces the need for internal commenting*
- Call for clear, **unambiguous code**

Limitations:

- *Practical implementation* is a *challenge*,

XP - Values

- The five values of XP are communication, simplicity, feedback, courage, and respect and are described in more detail below.

Communication:

- Software development is inherently a team sport that relies on communication to transfer knowledge from one team member to everyone else on the team. XP stresses the importance of the appropriate kind of communication – face to face discussion with the aid of a white board or other drawing mechanism.

XP - Values

Simplicity

- Simplicity means "what is the simplest thing that will work?" The purpose of this is to avoid waste and do only absolutely necessary things such as keep the design of the system as simple as possible so that it is easier to maintain, support, and revise. Simplicity also means address only the requirements that you know about; don't try to predict the future.

Feedback

- Through constant feedback about their previous efforts, teams can identify areas for improvement and revise their practices. Feedback also supports simple design. Your team builds something, gathers feedback on your design and implementation, and then adjust your product going forward.

XP - Values

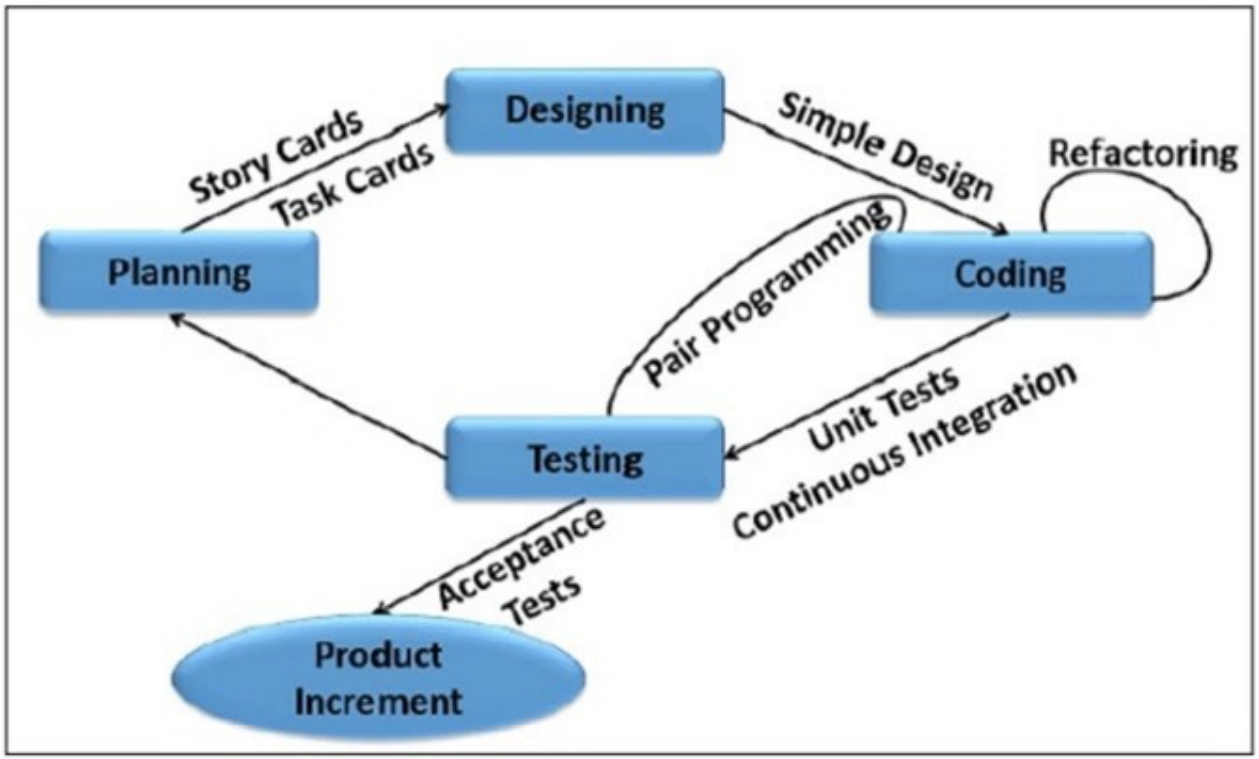
Courage

- Kent Beck defined courage as "effective action in the face of fear" (Extreme Programming Explained P. 20). This definition shows a preference for action based on other principles so that the results aren't harmful to the team. You need courage to raise organizational issues that reduce your team's effectiveness. You need courage to stop doing something that doesn't work and try something else. You need courage to accept and act on feedback, even when it's difficult to accept.

Respect

- The members of your team need to respect each other in order to communicate with each other, provide and accept feedback that honors your relationship, and to work together to identify simple designs and solutions.

XP – Advantages



- Built-In **Quality**
- Overall **Simplicity**
- **Customer & Team oriented**

XP – Advantages

- Extreme Programming solves the following problems often faced in the software development projects –
- **Slipped schedules** – and achievable development cycles ensure timely deliveries.
- **Cancelled projects** – Focus on continuous customer involvement ensures transparency with the customer and immediate resolution of any issues.
- **Costs incurred in changes** – Extensive and ongoing testing makes sure the changes do not break the existing functionality. A running working system always ensures sufficient time for accommodating changes such that the current operations are not affected.

XP – Advantages

- Extreme Programming solves the following problems often faced in the software development projects –
- **Production and post-delivery defects:** Emphasis is on – the unit tests to detect and fix the defects early.
- **Misunderstanding the business and/or domain** – Making the customer a part of the team ensures constant communication and clarifications.
- **Business changes** – Changes are considered to be inevitable and are accommodated at any point of time.
- **Staff turnover** – Intensive team collaboration ensures enthusiasm and good will. Cohesion of multi-disciplines fosters the team spirit.

XP – Controversial Aspects

- *Requirements are expressed as automated acceptance* tests rather than specification documents.
- *Requirements are defined incrementally*, rather than trying to get them all in advance.
- Software developers are usually *required to work in pairs*.

XP – Controversial Aspects – contd..

- There is no Big Design Up Front. Most of the design activity takes place on the fly and incrementally, *starting with "the simplest thing that could possibly work" and adding complexity only when it's required.*

- Critics fear this will result in more re-design effort than only re-designing when requirements change.

XP – Controversial Aspects – contd..

- A *customer representative* is attached to the project.
- This role can become a *single-point-of- failure* for the project.
- Also, there is the danger of *micro-management* by a non-technical representative trying to dictate the use of technical software features and architecture.

XP – Controversial Aspects – contd..

- Is **only as effective as the people involved**
- **Lack of structure** and necessary documentation
- **Only works with senior-level developers**
- Incorporates insufficient software design
- **Requires meetings at frequent intervals** at enormous expense to customers
- **Requires too much cultural change to adopt**
- Can increase the risk of scope creep due to the lack of detailed requirements documentation

Agile Methods:
Dynamic Systems Development Method (DSDM)

Dynamic Systems Development Method (DSDM)

Dynamic Systems Development Method:

- Arising in early 1990's, it is a formalization of RAD practices;

DSDM lifecycle has six stages:

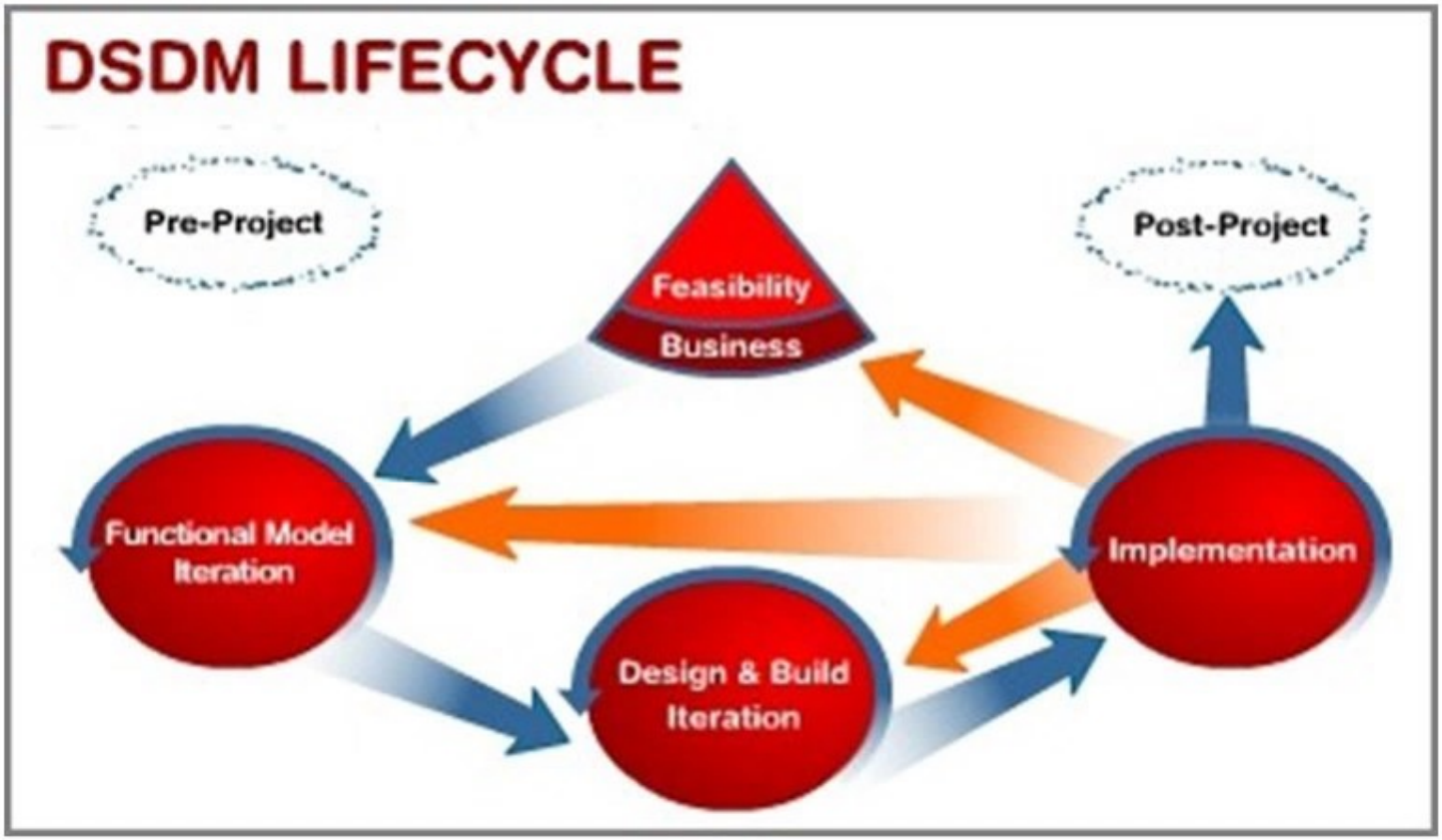
- Pre-project, Feasibility Study, Business Study, Functional Model Iteration, Design and Build Iteration, Implementation and Post-project.

Dynamic Systems Development Method (DSDM)

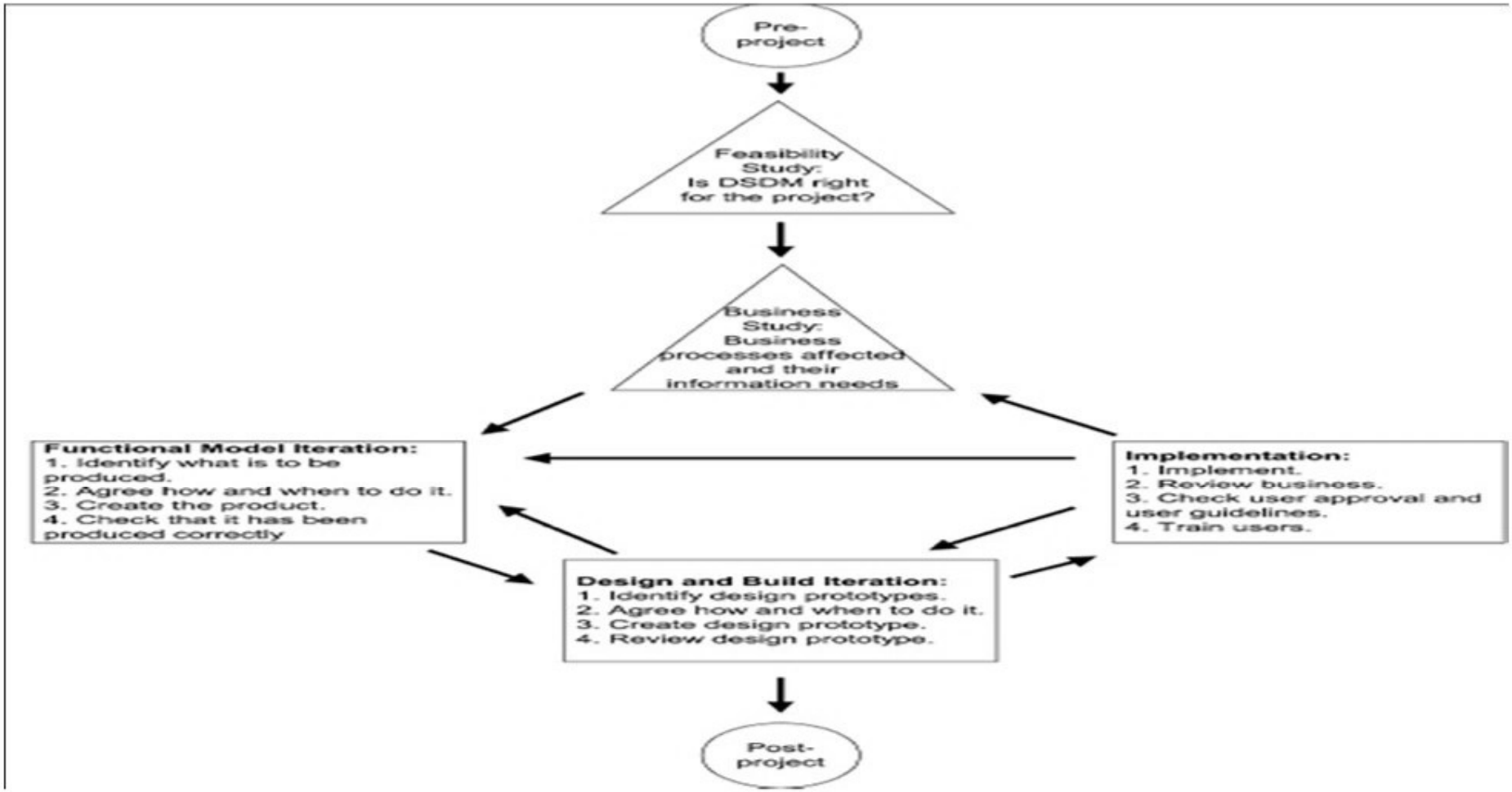
The eight Principles of DSDM :

- Focus on the business need
- Deliver on time
- Collaborate
- Never compromise quality
- Build incrementally from firm foundations
- Develop iteratively
- Communicate continuously and clearly
- Demonstrate control

Dynamic Systems Development Method (DSDM)



DSDM Lifecycle



DSDM Lifecycle

- **Pre-project:** The pre-project phase establishes that *the project is ready to begin*, funding is available, and that *everything is in place to commence* a successful project.

- **Feasibility study:** DSDM stresses that the feasibility study should be short, no more than a few weeks. Along with the usual feasibility activities, this phase should *determine whether DSDM is the right approach for the project*.

- **Business study:** The business study phase is "*strongly collaborative*, using a series of facilitated workshops attended by knowledgeable and empowered staff who can quickly pool their knowledge and gain consensus as to the priorities of the development".

- The result of this phase is the Business Area Definition, which identifies users, markets, and business processes affected by the system.

- **Functional model iteration:** Aims to build on the *high-level requirements identified* in the business study.
- The DSDM framework works by *building a number of prototypes based on risk* and evolves these prototypes into the complete system.

- This phase and the design and build phase have a common process:

(1) Identify what is to be produced.

(2) Agree how and when to do it.

(3) Create the product.

(4) Check that it has been produced correctly (by reviewing documents, demonstrating a prototype or testing part of the system).

- **Design and build iteration:** The prototypes from the functional model iteration are completed, combined, and tested and a working system is delivered to the users.

- **Implementation:**
  - The system is transitioned into use.
  - An Increment Review Document is created that discusses the state of the system.

- Either the ***system found to meet all requirements*** and can be considered ***complete***, or there is missing functionality.

- ***If there is still work to be done on the system***, the functional model, design and build, and ***implementation phases are repeated until the system is complete***.

- ***Post-project.*** This phase includes normal post- project clean-up, as well as on-going maintenance.

- Team size,
- Iteration lengths,
- Distribution, or
- system criticality ????

DSDM advocates the use of **several proven practices**, including:

- Facilitated Workshops
- Modelling and Iterative Development
- MoSCoW Prioritisation
- Time boxing

- **Workshop:** brings project stakeholders together to discuss requirements, functionalities and mutual understanding.

- **Modeling:** helps visualize a business domain and improve understanding. Produces a diagrammatic representation of specific aspects of the system or business area that is being developed.

- **MoSCoW Prioritisation:** The term MoSCoW itself is an acronym derived from the first letter of each of four prioritization categories (Must have, Should have, Could have, and Won't have), with the interstitial Os added to make the word pronounceable. While the Os are usually in lower-case to indicate that they do not stand for anything, the all-capitals MOSCOW is also used.

MoSCoW Prioritisation:

- **Must have**
- Requirements labelled as Must have are critical to the current delivery timebox in order for it to be a success. If even one Must have requirement is not included, the project delivery should be considered a failure (note: requirements can be downgraded from Must have, by agreement with all relevant stakeholders; for example, when new requirements are deemed more important). MUST can also be considered an acronym for the Minimum Usable Subset.

- **Should have**
- Requirements labelled as Should have are important but not necessary for delivery in the current delivery timebox. While Should have requirements can be as important as Must have, they are often not as time-critical or there may be another way to satisfy the requirement so that it can be held back until a future delivery timebox.

- **MoSCoW Prioritisation:**
  - **Could have**
  - Requirements labelled as Could have are desirable but not necessary and could improve the user experience or customer satisfaction for a little development cost. These will typically be included if time and resources permit.

  - **Won't have (this time)**
  - Requirements labelled as Won't have, have been agreed by stakeholders as the least-critical, lowest-payback items, or not appropriate at that time. As a result, Won't have requirements are not planned into the schedule for the next delivery timebox. Won't have requirements are either dropped or reconsidered for inclusion in a later timebox. (Note: occasionally the term Would like to have is used; however, that usage is incorrect, as this last priority is clearly stating something is outside the scope of delivery).

- **Timeboxing:** is the approach for completing the project incrementally by breaking it down into splitting the project in portions, each with a fixed budget and a delivery date.
- For each portion a number of requirements are prioritised and selected. Because time and budget are fixed, the only remaining variables are the requirements.
- So if a project is running out of time or money the requirements with the lowest priority are omitted.
- This does not mean that an unfinished product is delivered, because of the Pareto Principle that 80% of the project comes from 20% of the system requirements, so as long as those most important 20% of requirements are implemented into the system, the system therefore meets the business needs and that no system is built perfectly in the first try.

Agile Methods:
Feature-Driven Development (FDD)

Feature-Driven Development (FDD)

- Based on expressing the ***requirements in terms of user valued pieces of functionality*** called **Features.**

- **Not a full-lifecycle methodology:**

1. ***Starts when feasibility study*** and ***overall project planning have already been done***, a business case has been established, and permission has been granted by the sponsors to go on with the development.

2. ***Excludes post-implementation activities*** such as system-wide verification and validation, and the ultimate system deployment and maintenance.

FDD: Features

- Each feature is a relatively ***fine-grained function*** of the system expressed in client-valued terms.
- **Conforming to the general template:**

  <action> <result> <object>
- **For example:** ***"calculate the total value of a shipment"*** or ***"check the availability of seats on a flight"***
- The granularity of each feature should be such that it would take ***not more than two weeks to develop***,
- ***otherwise it will be broken down into smaller features***.

FDD: Feature Sets

- 1. ***Each feature is identified as a Step*** in one or more Activities (also called Feature Sets), which are expressed as:

  <action><-ing> a(n) <object>

  - **For example:** "reserving a seat"

- 2. Activities in turn belong to Areas (or Major Feature Sets) which are expressed using the general template:

  <object> management

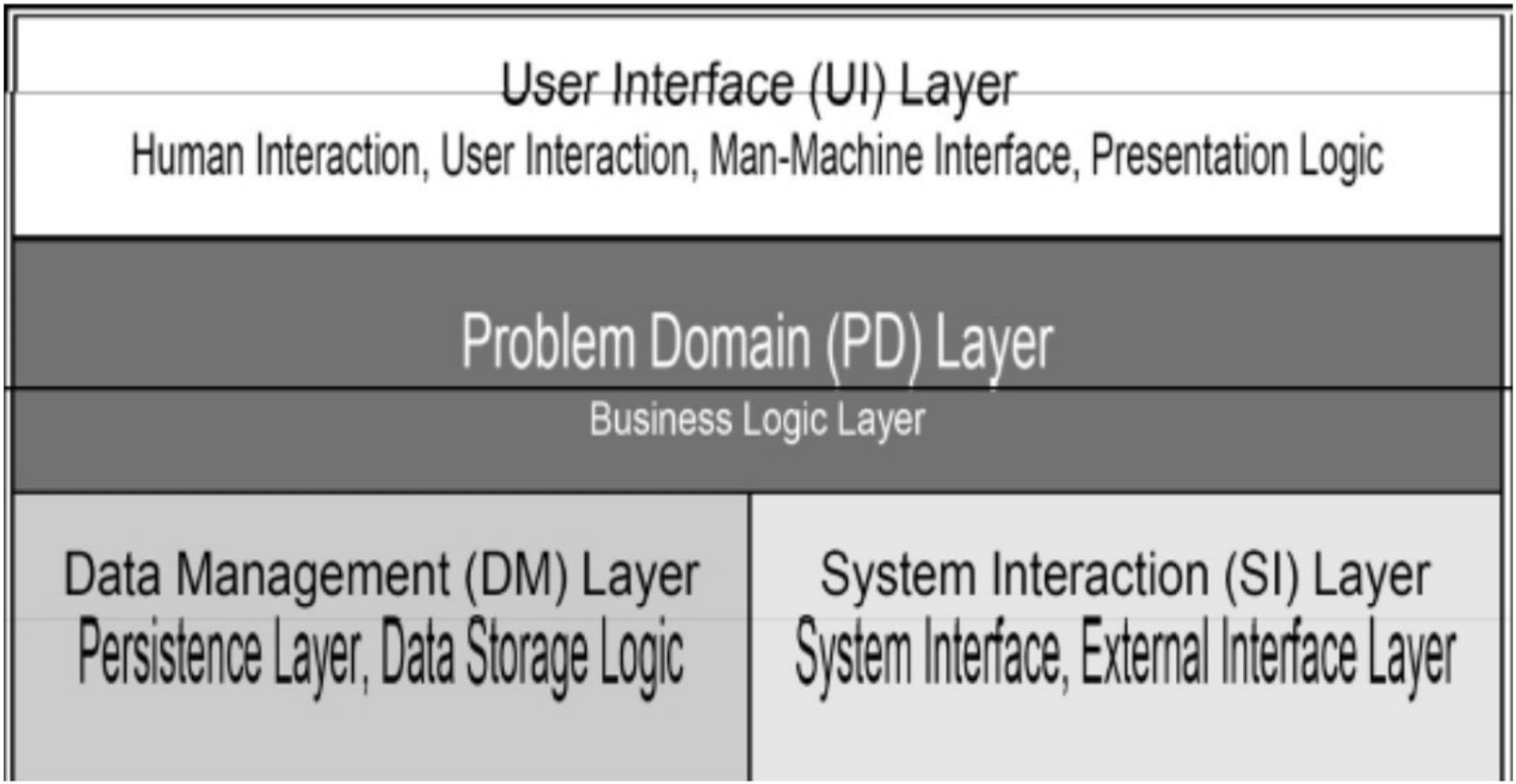
  - **For example:** "reservations management"

FDD: Feature Sets

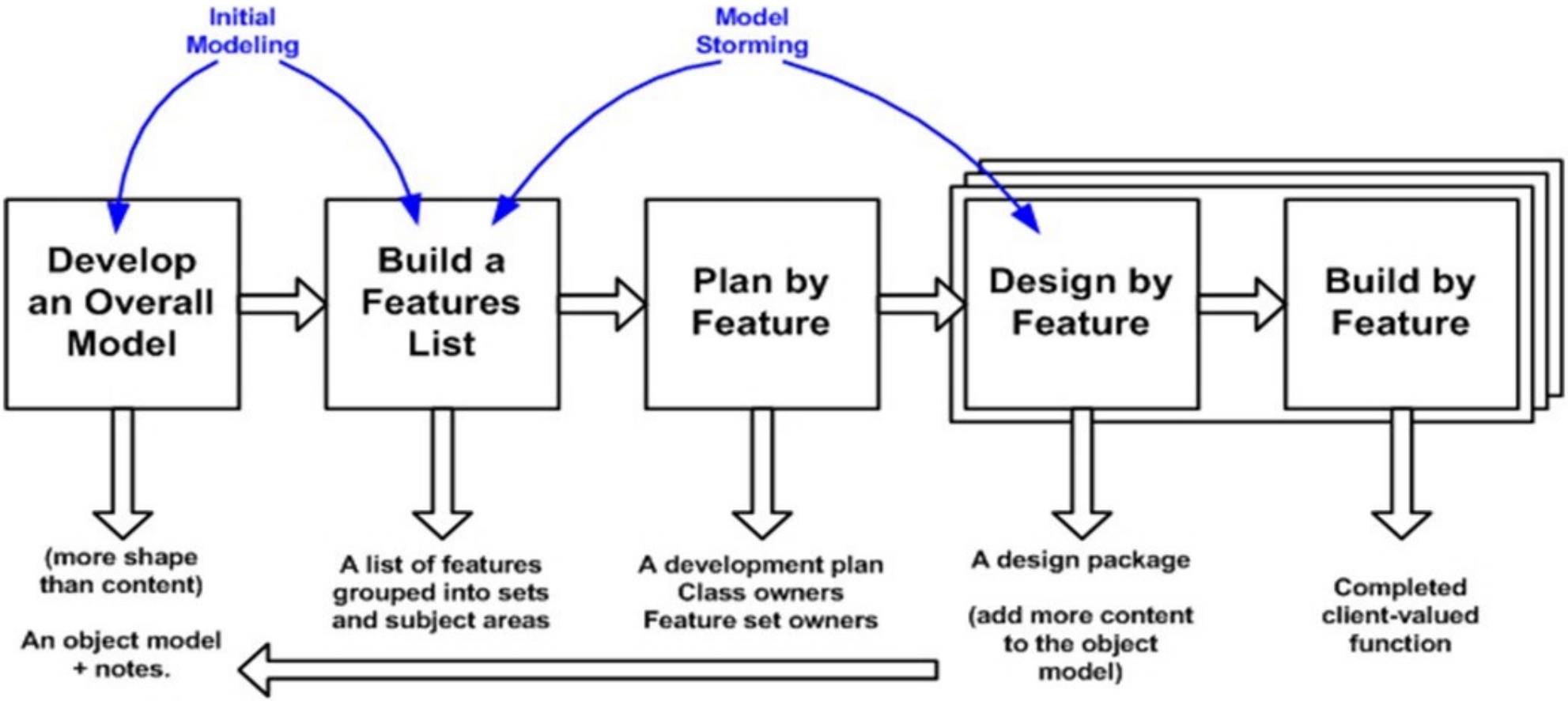
- The ***three-layered architecture*** of Areas-Activities-Steps allows the developers to adequately manage the complexity of the requirements.

- **Features can also be partitioned** according to the **architectural layer** to which they belong.

FDD: System Architecture



FDD: Process



FDD: Sequential Sub processes

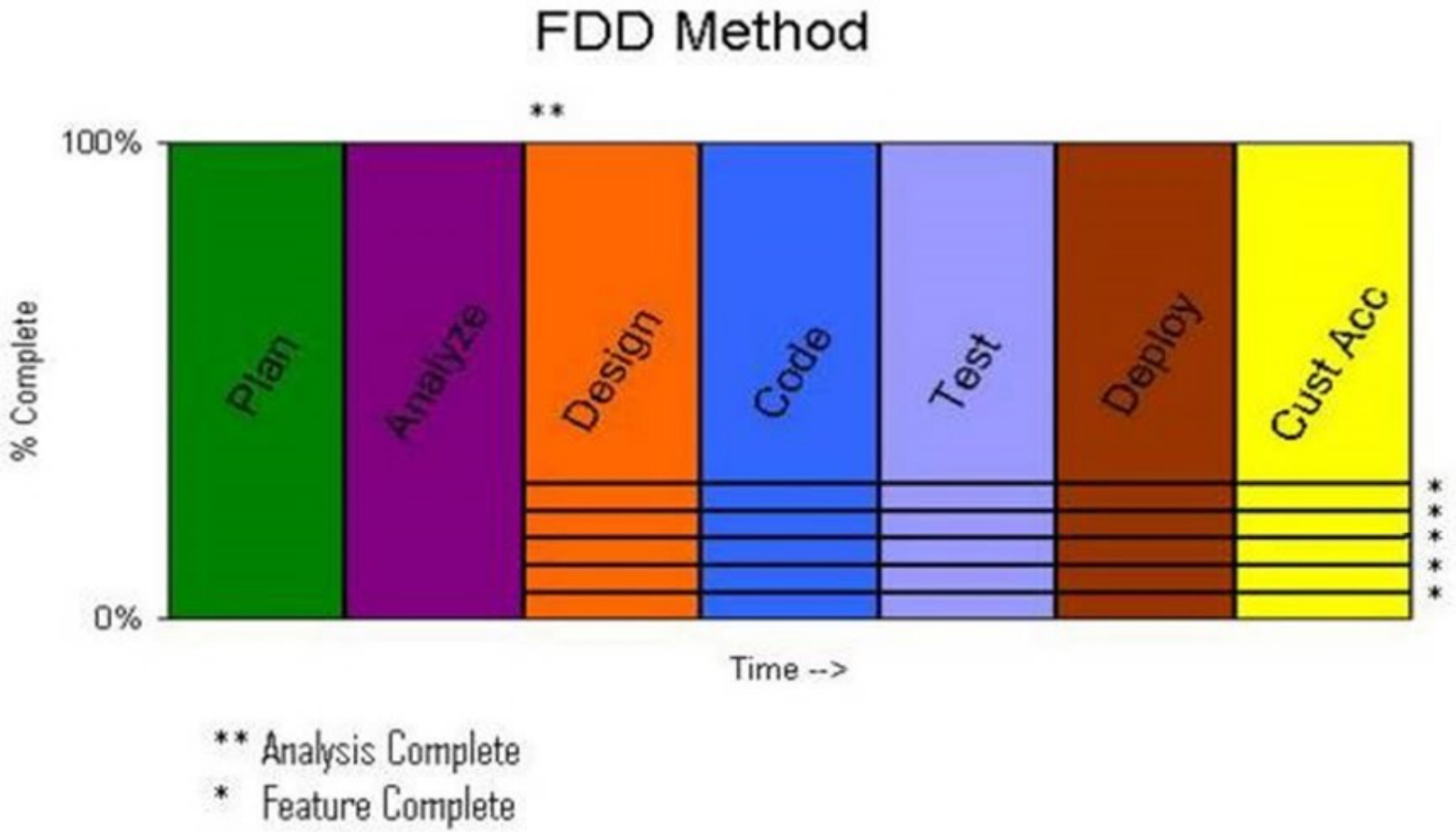
- **1. Develop an Overall Model:** building a mainly structural model of the problem domain called the Object Model.
- **2. Build a Features List:** ***identifying the required functionality of the system***, expressed as features and feature sets.
- **3. Plan by Feature:** ***scheduling the features for development***, and assigning the feature sets (activities), and the classes in the object model, to developers.

FDD: Sequential Sub processes

- **4. Design by Feature:** determining ***how the features should be realized at run-time*** by interactions among objects.

- **5. Build by Feature:** ***coding and unit-testing*** the necessary items for realization of the features. Implemented items are then promoted to the main build.

FDD: Sequential Sub processes



FDD: Strengths and Weaknesses

Strengths:

- ***Iterative-incremental process***
- Based on **layered architecture**
- Based on system requirements captured as Features
- ***Traceability*** implemented through features
- ***Simple and straightforward*** process
- **Design-based development**
- ***Continuous validation***
- ***Simple five-step process allows for more rapid development***
- ***Allows larger teams to move products forward with continuous success***
- Frequent deliveries once the iterations start
- Complexity management at the features level through layering

FDD: Strengths and Weaknesses

Weaknesses:

- ***Does not cover post-implementation activities*** and preliminary analysis.
- **Does not work efficiently for smaller projects**
- **Less written documentation**, which can **lead to confusion**
- **Highly dependent** on lead developers or programmers
- ***Lacks adaptability due to inexistence*** of iteration level planning, reviewing and revision.
- Intensive project supervision is essential.

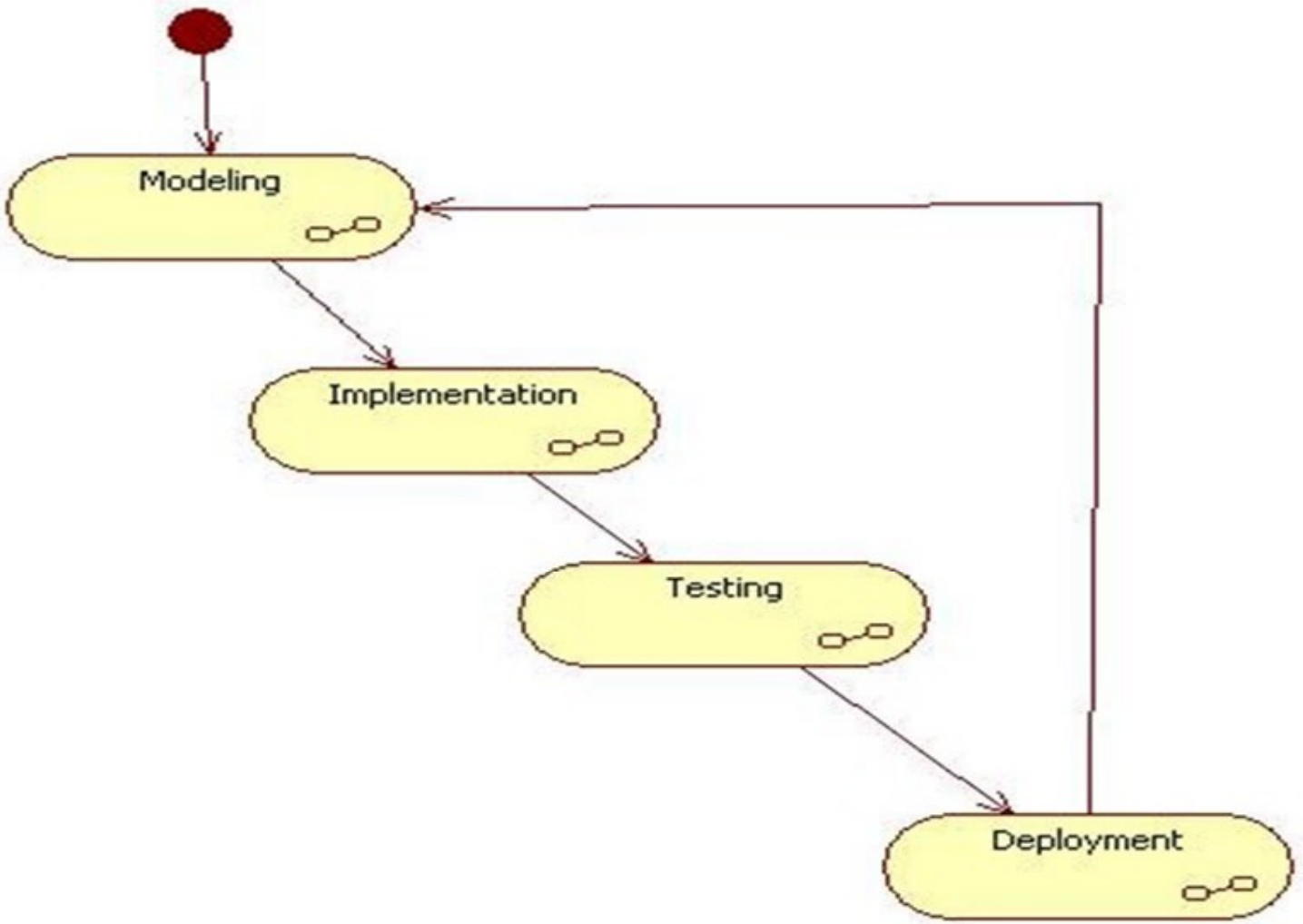
Agile Methods:

Agile Unified Process (AUP)

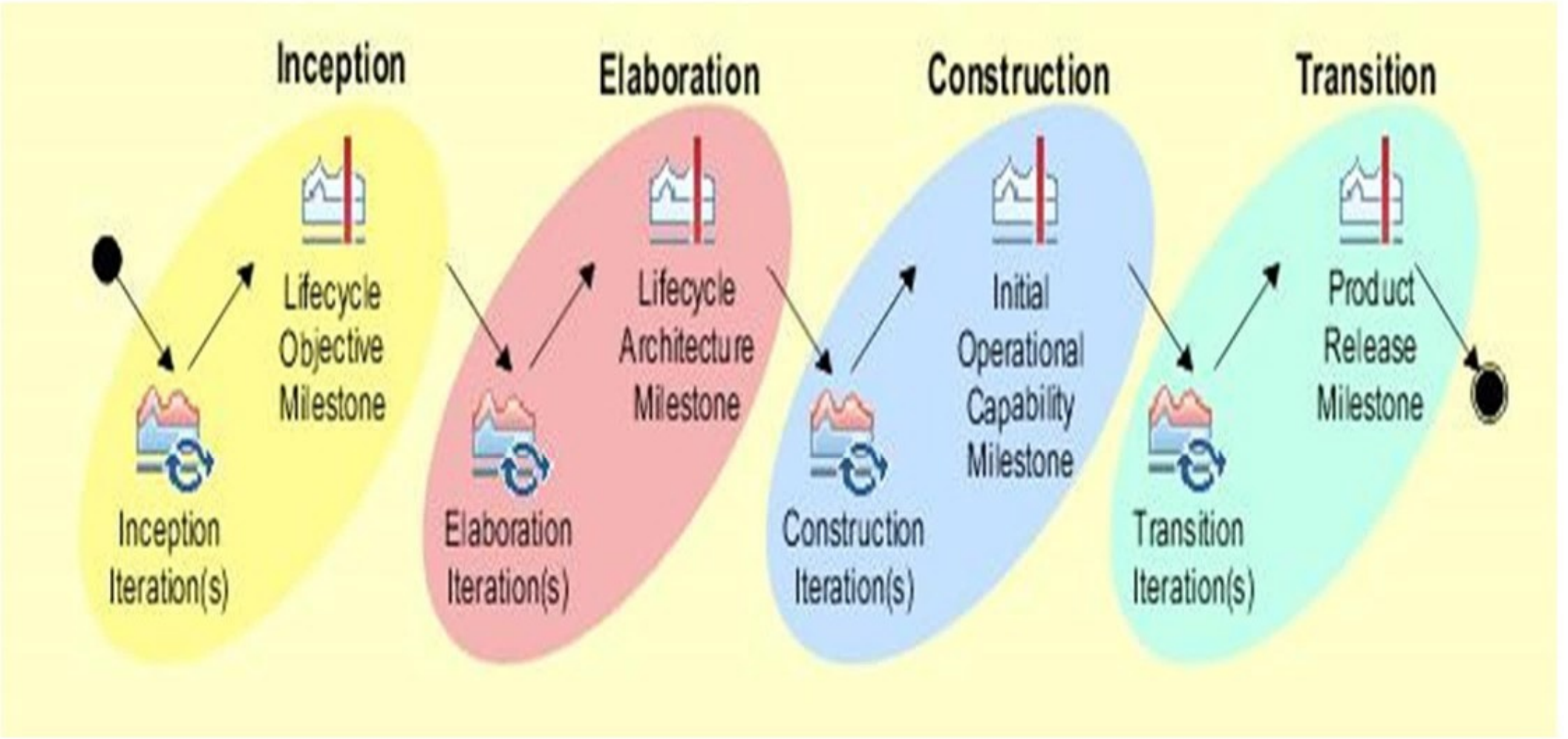
Agile Unified Process (AUP)

- UML is the main modeling language, but ***AUP is not restricted to UML.***
- Use case driven.
- ***Iterative-incremental.***
- Architecture-centric.
- ***Covering the full generic lifecycle.***

Agile Unified Process (AUP)



AUP: Process – Phases

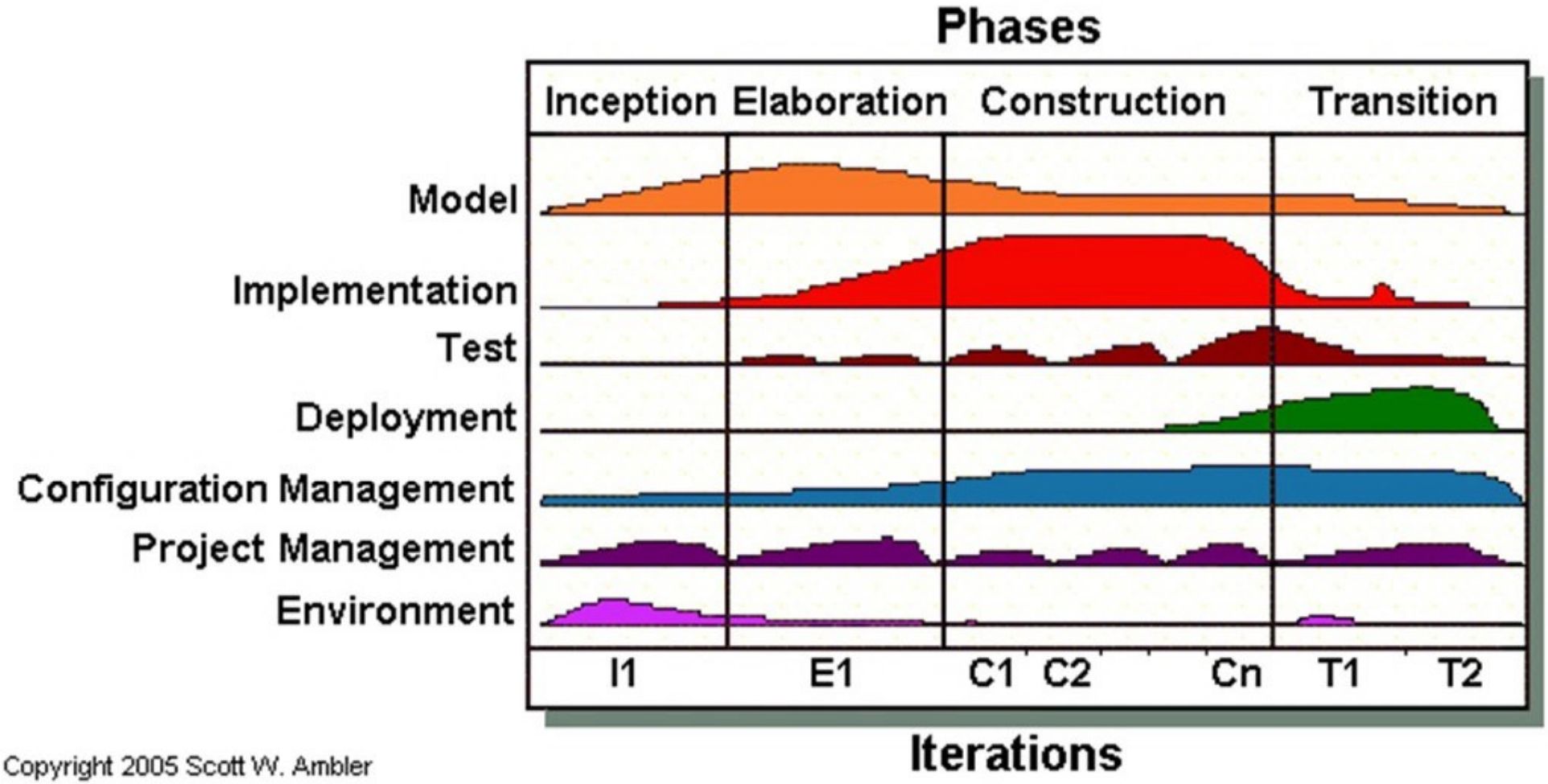


AUP: Process – Phases

Overall development cycle consists of four phases:

- **Inception:** The goal is to ***identify the initial scope*** of the project, a potential architecture for the system, and to obtain initial project funding and stakeholder acceptance.
- **Elaboration:** The goal is to ***develop the system architecture***.
- **Construction:** The goal is to ***build working software*** on a regular, incremental basis which meets the highest-priority needs of project stakeholders.
- **Transition:** The goal is to ***validate and deploy the system*** into the production environment.

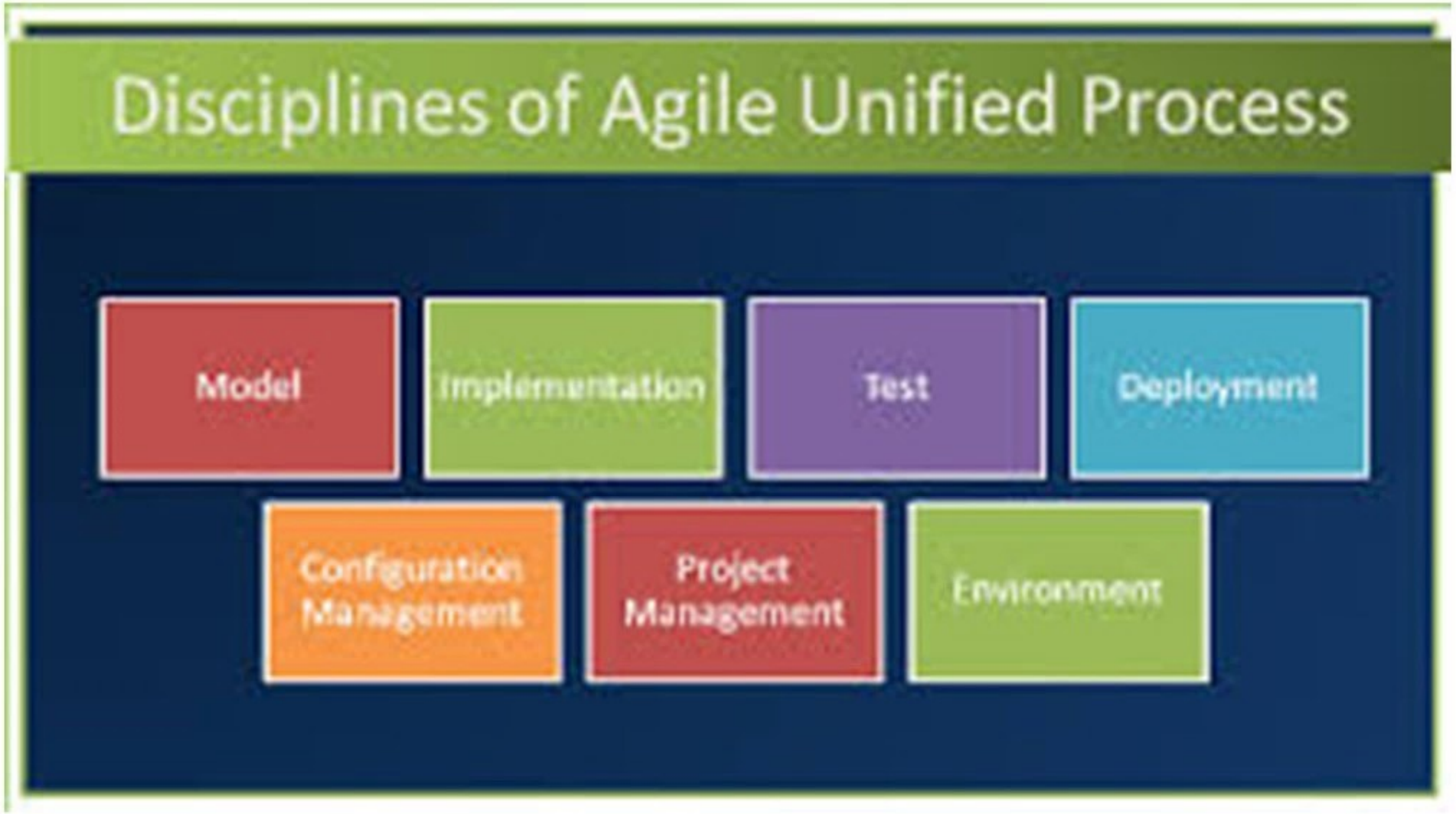
AUP: Process – Phases



AUP Process
– Iterations and Disciplines

- ***Each phase*** can be further ***broken down into iterations***.
- An ***iteration is a complete development loop*** resulting in a release of an executable increment to the system.
- ***Each iteration consists of seven work areas*** (disciplines) performed during the iteration.

AUP: Process – Disciplines



AUP: Process – Disciplines

- **1. Model:** The goal is to ***understand the business of the organization***, the **problem domain**, and to identify a viable solution to address the problem domain.

- **2. Implementation:** The goal is to ***transform the model(s) into executable code*** and to perform a basic level of testing, in particular **unit testing.**

- **3. Test:** The goal is to perform an objective ***evaluation to ensure quality***. This includes finding defects, **validating** that the system works as designed, and **verifying that the requirements are met**.

AUP: Process – Disciplines

- **4. Deployment.** The goal is to ***plan for the delivery of the system*** and to execute the plan to make the system available to end users.

- **5. Configuration Management.** The goal is to ***manage access to the project artifacts***. This includes not only tracking artifact versions over time but also ***controlling and managing changes*** to them.

- **6. Project Management.** The goal is to ***direct the activities that take place on the project***. This includes ***managing risks, directing people, and coordinating with people*** and systems outside the scope of the project to be sure that it is delivered on time and within budget.

- **7. Environment.** The goal is to ***support the rest of the effort*** by ensuring that the ***proper process, guidance*** (standards and guidelines), and ***tools (hardware, software, etc.)*** are available for the team as needed.

AUP: Strengths and Weaknesses

Strengths:

- ***Iterative-incremental process***
- Based on **system architecture**
- Based on **system functionality** captured in use cases
- ***Traceability*** to requirements through use-cases
- **Design-based** development
- ***Risk-based process***
- Formal features can be added via UML

AUP: Strengths and Weaknesses

Weaknesses

- **Modeling may jeopardize agility** if limits are not strictly observed;
- ***Chances of becoming traditional approach.***

Most Suited Projects for Agile

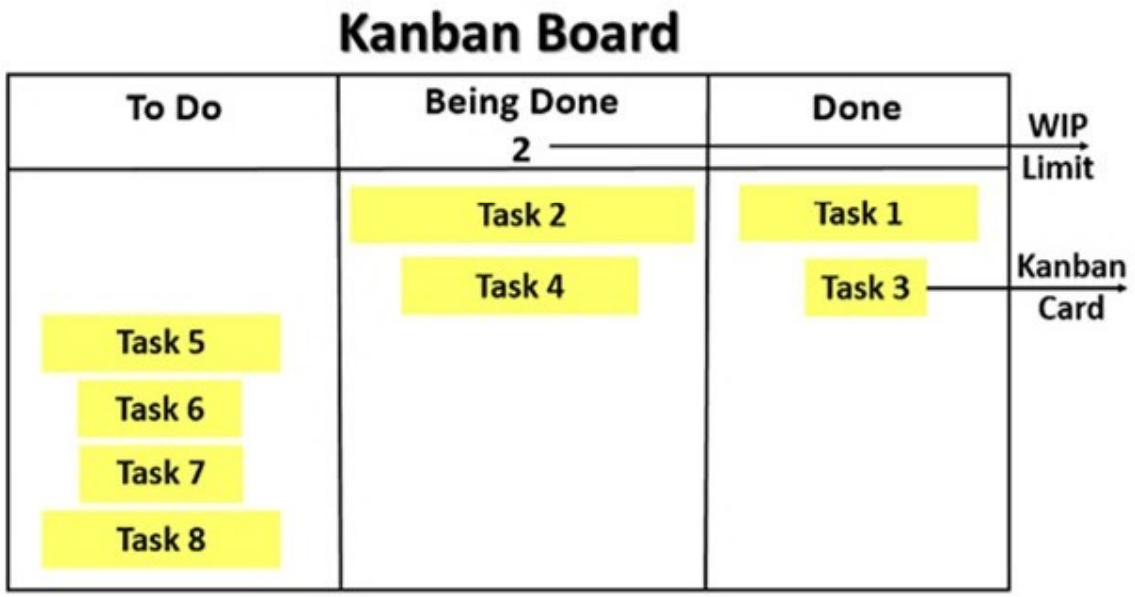
Appropriate projects are ones with:

- Aggressive deadlines,
- High degree of complexity &
- High degree of uniqueness.

- ***Use agile when we are doing something new***.
  - Example: Manufactoring

- ***Urgency*** is another factor;

Kanban Methodology



Kanban Methodology

What is kanban?

- Kanban term came into existence using the flavors of "visual card," "signboard," or "billboard", "signaling system" to indicate a workflow that limits **Work In Progress (WIP).** Kanban has been used in Lean Production for over half-century.
- Kanban is a **popular framework** used to implement agile software development.
- It **requires real-time communication** of capacity and full transparency of work.
- Work items are represented visually on a **kanban board**, allowing team members to see the state of every piece of work at any time.

Kanban Methodology

Kanban Board:

- Kanban board is **used to depict the flow of tasks** across the value stream.

- The Kanban board –
  - **Provides easy access** to everyone involved in the project.
  - **Facilitates communication** as and when necessary.
  - **Progress of the tasks** are visually displayed.
  - **Bottlenecks are visible** as soon as they occur.

Kanban - Values

Values:

Teams applying Kanban to improve the services they deliver embrace the following values:

- **Transparency** – sharing information openly using clear and straightforward language improves the flow of business value.

- **Balance** – different aspects, viewpoints, and capabilities must be balanced in order to achieve effectiveness.

- **Collaboration** – Kanban was created to improve the way people work together.

Kanban - Values

- **Customer Focus** – Kanban systems aim to optimize the flow of value to customers that are external from the system but may be internal or external to the organization in which the system exists.

- **Flow** – Work is a continuous or episodic flow of value.

- **Leadership** – Leadership (the ability to inspire others to act via example, words, and reflection) is needed at all levels in order to realize continuous improvement and deliver value.

Kanban - Values

- **Understanding** – Individual and organizational self-knowledge of the starting point is necessary to move forward and improve.
- **Agreement** – Everyone involved with a system are committed to improvement and agree to jointly move toward goals while respecting and accommodating differences of opinion and approach.
- **Respect** – Value, understand, and show consideration for people.

Kanban Methodology - Principles

Kanban is based on 3 basic principles:

- Visualize what you do today (**workflow**): seeing all the items in context of each other can be very informative.

- Limit the amount of work in progress (**WIP**): this helps balance the flow-based approach so teams don't start and commit to too much work at once.

- **Enhance flow:** when something is finished, the next highest thing from the backlog is pulled into play.

Kanban Methodology - Benefits

Benefits:

- **Shorter cycle times** can deliver features faster.
- **Responsiveness to Change:**
- **When priorities change very frequently**, Kanban is ideal.
- Balancing demand against throughput guarantees that most the customer-centric features are always being worked.
- Requires fewer organization / room set-up changes to get started
- **Reducing waste** and removing activities that don't add value to the team/department/organization
- **Rapid feedback** loops improve the chances of more motivated, empowered and higher-performing team members

Kanban Vs Scrum

Kanban	Scrum
No prescribed roles	Pre-defined roles of Scrum master, Product owner and team member
Continuous Delivery	Timeboxed sprints
Work is pulled through the system (single piece flow)	Work is pulled through the system in batches (the sprint backlog)
Changes can be made at any time	No changes allowed mid-sprint
Cycle time	Velocity
More appropriate in operational environments with a high degree of variability in priority	More appropriate in situations where work can be prioritized in batches that can be left alone

Agile Failure Models

Agile Failure At Different Levels

I. Agile Failure At Organizational Level
II. Agile Failure At Team Level
III. Agile Failure At Process Level
IV. Agile Failure At Facility Level

Failure Modes of Agile

I. **Agile Failure At Organizational Level:**

- ***Not having a (product) vision in the first place***. If you don't know where you are going, any road will get you there.
- The fallacy of "We know what we need to build". ***There is no need for product discovery or hypotheses testing***, the senior management can define what is relevant for the product backlog.
- A perceived ***loss of control at management level*** leads to micro-management.
- ***The organization is not transparent*** with regard to vision and strategy hence the teams are hindered to become self-organizing.
- **There is no culture of failure:** Teams therefore do not move out of their comfort zones, but instead play safe. (Mathgeek on HN suggests to replace "culture of failure" with "**culture of learning to get back up again after falling.**")
- ***The organization is not optimized for a rapid build-test-learn culture*** and thus departments are moving at different speed levels. The resulting friction caused is likely to equalize previous Agile gains.

Failure Modes of Agile

I. Agile Failure At Organizational Level:

- ***Senior management is not participating in Agile processes***, e.g. sprint demos, despite being a role model. But they do expect a different form of (push) reporting.
- ***Not making organizational flaws visible:*** The good thing about Agile is that it will identify all organizational problems sooner or later. "When you put problem in a computer, box hide answer. Problem must be visible!"
- ***Product management is not perceived as the "problem solver and domain expert" within the organization***, but as the guys who turn requirements into deliverables, aka "Jira monkeys".
- ***Other departments fail to involve product management from the start*** A typical behavior in larger organizations is a kind of silo thinking, featured by local optimization efforts without regard to the overall company strategy, often driven by individual incentives, e.g. bonuses. (Personal agendas are not always aligned with the company strategy.)
- ***Core responsibilities of product management are covered by other departments***, e.g. tracking, thus leaving product dependent on others for data-driven decisions.
- ***Product managers w/o a dedicated team can be problem***, if the product management team is oversized by comparison to the size of the engineering team.

Failure Modes of Agile

II. Agile Failure At Team Level:

- ***There are too many junior engineers on an engineering team***. They tend to appreciate micro-management as part of their training. Usually, they have no or little experience with Agile methodologies, hence they hardly can live up to processes, particularly they fail to say "No".

- ***Engineers with an open source coding mentality:*** Tasks are discussed on pull requests once they're finished, but not in advance during grooming or sprint planning sessions. (They tend to operate in distributed team mentality.)

- ***Teams are too small and hence not cross-functional.*** Example: A team is only working on frontend issues and lacks backend competence. That team will always rely on an other team delivering functionality to build upon.

Failure Modes of Agile

II. Agile Failure At Team Level:

- ***Teams are not adequately staffed,*** e.g. Scrum Master positions are not filled and product owners have to serve two roles at the same time.
- ***Team members reject Agile methodologies openly***, as they do not want to be pushed out of their comfort zones.
- ***Teams are not self-organizing***. That would require to accept responsibility for the team's performance and a sense of urgency for delivery and value creation. A "team" in this mode is actually more acting like a group—e.g. people waiting at a bus-stop—: They are at the same time in the same place for the same purpose, but that does not result in forming a team. (The norming, storming, forming, performing cycle seems to be missing.)
- ***Even worse, team members abandon Agile quietly,*** believing it is a management fad that will go away sooner or later.
- Faux Agile: ***Teams follow the "Agile rules" mechanically without understanding why those are defined in the first place***. This level of adoption often results often in a phenomenon called "Peak Scrum": There is no improvement over the previous process, despite all Agile rules are being followed to the letter. Or even worse: morale and productivity go down, as the enthusiasm after the initial agile trainings wears off quickly in the trenches.
- ***Moving people among teams upon short notice***. Even if required for technical reasons, this has a negative impact on a team's performance & morale.

Failure Modes of Agile

III. Agile Failure At Process Level:

- Agile processes are either bent or ignored whenever it seems appropriate. There is a ***lack of process discipline***.
- Agile processes are tempered with, e.g. the Scrum Product Owner role is reduced to a project manager role. Often this is done so by assigning the task of backlog ownership to a different entity at management level. ("Scrum" without a Product Owner actually makes a great Waterfall 2.0 process.)
- ***Stakeholders are bypassing product management to get things done*** and get away with it in the eyes of the senior management, as they would show initiative.
- ***The organization is not spending enough time on team communication*** and workshops to create a shared understanding on what is to be built.

Failure Modes of Agile

IV. Agile Failure At Facility Level:

- ***A team is not co-located, not working in the same room***, but scattered across different floors, or worse, different locations.
- The work environment is ***lacking*** spots for formal and – more important – ***informal communication***: cafeterias, tea kitchens, sofas etc.
- ***The work environment is lacking whiteboards***: Actually, the absence of whiteboards on each and every available wall within the office should be questionable, not having them.
- ***Agile requires suitable offices to further collaboration***: spacy, with plenty of air and light. But they should not be a mere open space, which tends to get too noisy, particularly when several Scrum team are having stand-ups at the same time.

Various reasons why agile fails?

Failure Modes of Agile

Eight Failure Modes of Agile:

- 1. No Test Harness
- 2. System Integration Not Included in "Done"
- 3. ***No Involvement***
- 4. ***Little Thought*** to Architecture
- 5. ***Building the Wrong Thing***
- 6. Product Owner as Handover
- 7. ***Lack of competency*** Leadership
- 8. Failure to Constantly Improve

Failure Modes of Agile Transformation

12 failure modes in Agile Transformation:

- ***Lack of*** Executive ***Sponsorship***
- ***Failure to Transform*** Leader Behaviors
- ***No Change*** to the ***Organizational Infrastructure***
- ***No Business View*** of the Value Stream
- ***Failure to Decentralize*** Control
- Unwillingness to Address Illusions Around Distributed Teams

Failure Modes of Agile

12 failure modes in agile transformation:

- Lack of a Transformational Product Manager
- ***Failure*** to Create ***Fast Feedback***
- Short-changing Collaboration and Facilitation
- ***Ineffective Plan*** for Transforming Beyond IT
- ***Viewing Transformation Solely*** as Process and Structure
- ***Ignoring the Path of Individual,*** Team and Organizational Transitions

References

- K.S. Rubin, Essential Scrum: A Practical Guide to the Most Popular Agile Process, Addison-Wesley, 2012.
- M. Cohn, Succeeding with Agile: Software Development Using Scrum, Addison-Wesley, 2009
- S.W. Ambler, M. Lines, Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise, IBM Press, 2012.
- Chetankumar Patel, Muthu Ramachandran, Story Card Maturity Model (SMM): A Process Improvement Framework for Agile Requirements Engineering Practices, Journal of Software,Academy Publishers, Vol 4, No 5 (2009), 422-435, Jul 2009.
- Kevin C. Desouza, Agile information systems: conceptualization, construction, and management, Butterworth-Heinemann, 2007
- K. Beck, C. Andres, Extreme Programming Explained: Embrace Change, 2nd Edition, Addison-Wesley, 2004.

Thank You!...