# Software Engineering

Bernardo Ramalho

9 January 2021

# Contents

# 1 Introduction

## 1.1 What is Software Engineering?

The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

Software engineering is concerned with the cost-effective and timely development of high-quality software in a predictable way, particularly for large scale systems.

## 1.2 The Software Process

A **software process** is a structured set of activities required to develop a software system. There are many different software processes but all involve the following **process activities**:

- **Specification** - defining what the system should do;
- **Design and Implementation** - defining the organization of the system and implementing the system;
- **Validation** - checking that it does what the customer wants;
- **Evolution** - changing the system in response to changing customer needs.

## 1.3 Why do we need (defined) processes?

### 1.3.1 Efficiency

- Incorporates best practices;
- Structures and guides your work;
- Keeps you focused on what needs to be done now.

### 1.3.2 Consistency

- Results likely to be similar;
- Work likely to become predictable.

### 1.3.3 Basis for Improvement

- Gathering data on your work helps determine which steps are the most time consuming, ineffective or troublesome;
- This is useful determine opportunities for improvement.

## 1.4 Plan-driven and agile processes

**Plan-driven** processes are processes where **all** of the process activities are **planned in advance** and progress is measured against this plan.

In **agile** processes, **planning is incremental** and it is easier to change the process to reflect changing customer requirements.

In practice, most practical processes include elements of both plan-driven and agile approaches.

There are no right or wrong software processes.

# 2 Process Activities

## 2.1 Requirements engineering

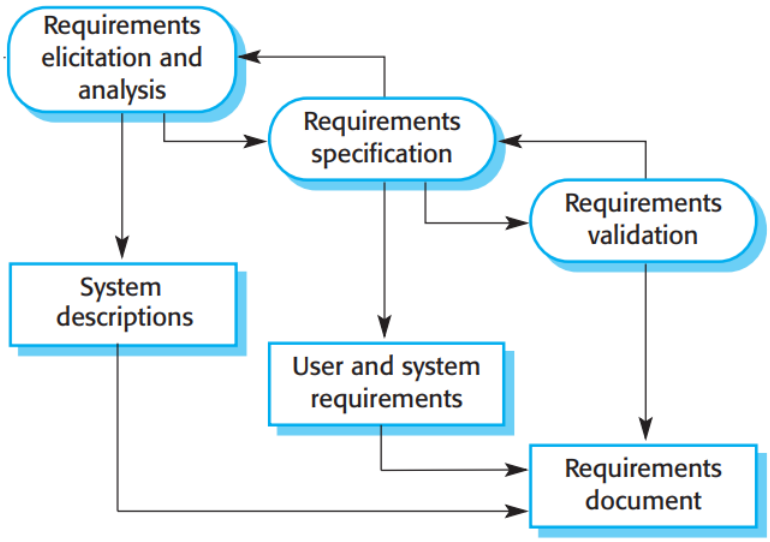


Figure 1: Graph demonstrating Requirements Engineering.

## 2.2 Software Design and Implementation

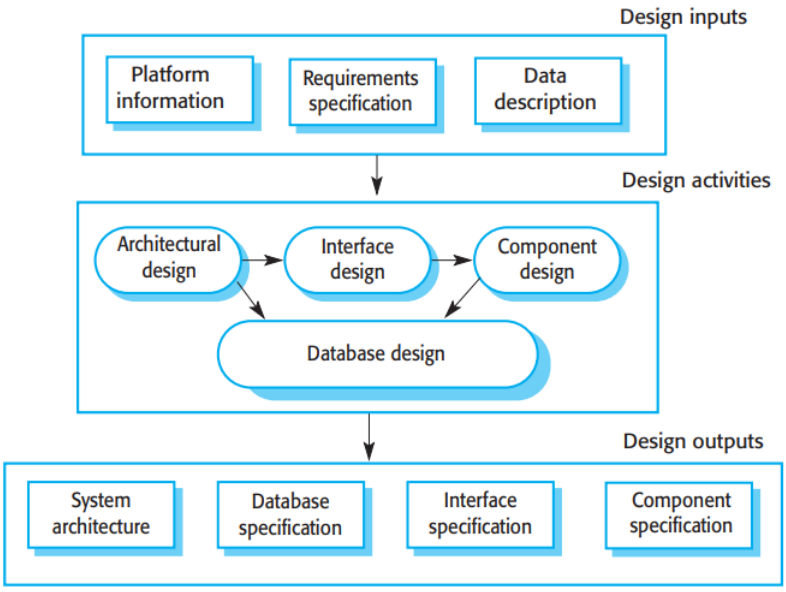


Figure 2: Graph demonstrating Software Design.

## 2.3 Software verification and validation

Verification and validation is intended to show that the system conforms to its specification (verification) and meets the requirements and customer needs (validation).

Performed mainly through testing, reviews and inspections. Typical testing stages (or levels) are:

- **Unit (or component testing)** - Individual components are tested usually by their developers;
- **Integration Testing** - Performed as components are integrated, to find integration problems;
- **System Testing** - The system as a whole is tested usually by an independent test team, with a focus on emergent properties (performance, usability, etc...);

- **Acceptance Testing** - The system is tested (under the customer responsibility) with customer data to check that customer's needs are met.

## 2.4 Software evolution (or maintenance)

Software is inherently flexible and can change. As requirements change through changing business circumstances, the software that supports the business must also evolve and change.

Although there has been a demarcation between development and evolution, this is increasingly irrelevant as fewer and fewer systems are completely new.

Type of maintenance activities:

- **Corrective** - Bug fixing;
- **Adaptive** - Adapt to new platforms, technologies;
- **Perfective** - New Functionalities.

# 3 Software Process Models

## 3.1 Model Types

- **The Waterfall Model** - Plan-driven model. Separate and distinct phases of specification and development;
- **Incremental Development and Delivery** - Specification, development and validation are interleaved. May be plan-driven or agile.
- **Integration and configuration** - The system is assembled from existing configurable components. May be plan-driven or agile.
- **Software prototyping** - Not actually a model but an approach to cope with uncertainty.

In practice, most large systems are developed using a process that incorporates elements from all of these models.

## 3.2 Waterfall

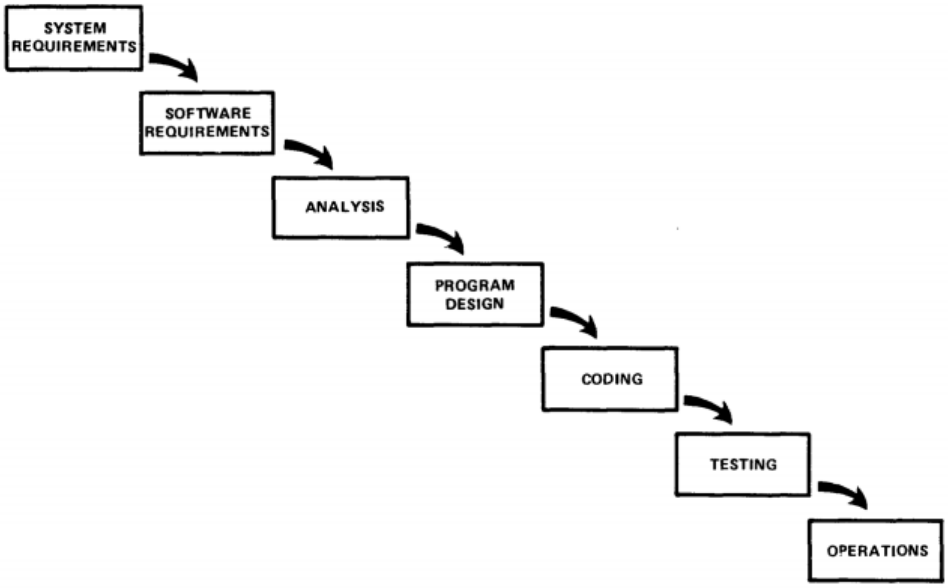


Figure 3: Implementation steps to develop a large computer program for delivery to a customer.

The waterfall model is Plan-driven. Consists of separate and distinct phases of specification, development and validation. In principle, a phase has to be complete before moving to the next phase.

In terms of **applicability**, the waterfall models has inflexible partitioning of the project into distinct stages, which makes it difficult to respond to changing customer requirements. Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process. Few business systems have stable requirements.

The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites. In those circumstances, the plan-driven (predictive) nature of the waterfall model helps coordinate the work.

## 3.3 Incremental development and delivery

The system is developed in increments and each increment is evaluated (or even delivered to customers) before proceeding to the development of the

next increment. Specification, development and validation may be interleaved. May be plan-driven or agile.

**Benefits:**

- The cost of accommodating changing customer requirements is reduced. There is less documentation to change and unstable requirements can be left for later stages of development;
- More frequent and early customer feedback reduces risk of failure;
- Customer value can be delivered with each increment so system functionality is available earlier;
- Early increments act as a prototype to help elicit requirements for later increments.

**Problems:**

- System structure tends to degrade as new increments are added. Unless time and money is spent on **refactoring** to improve the software, regular change tends to corrupt its structure and incorporating further changes becomes increasingly difficult and costly;
- It can be hard to identify upfront common facilities that are needed by all increments, so level of **reuse may be sub-optimal**;
- Incremental delivery may not be possible for **replacement systems** as increments have less functionality than the system being replaced;
- The nature of incremental development of the specification together with the software may not be adequate for establishing a development **contract** at the beginning.

## 3.4 Integration and Configuration

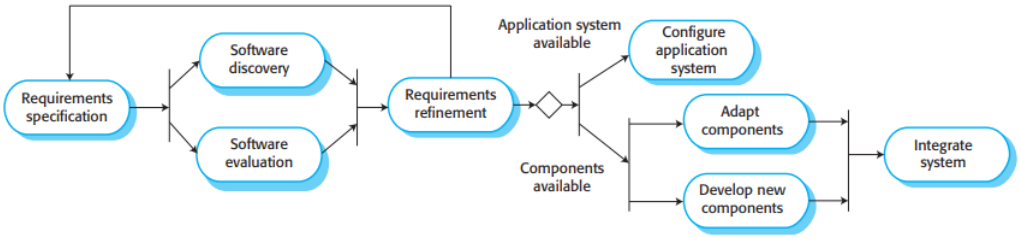


Figure 4: Graph showing how integration and configuration works.

Integration and Configuration has reduced costs and risks as less software is developed from scratch. It also has faster delivery and deployment of system. But requirements compromises are inevitable so system may not meet real needs of users. Another disadvantage is the loss of control over evolution of reused system elements.

# 4 Rational Unified Process (RUT)

## 4.1 Key Characteristics

- Iterative and Incremental Process;
- Rational Unified Process was developed hand-in-hand with UML;
- Driven by Use Cases (the identification of use cases and typical usage scenarios is the activity that drives all development process, from requirements analysis to final system testing);
- Architecture-centric (promotes the initial definition of a robust software architecture that helps the development parallelization, reuse, and maintenance.

## 4.2 Inception Phase

- Establishing the project's software scope and boundary conditions, including an operational vision, acceptance criteria and what is intended to be in the product and what is not;

- Discriminating the critical use cases of the system, the primary scenarios of operation that will drive the major design trade offs;
- Exhibiting, and maybe demonstrating, at least one candidate architecture against some of the primary scenarios;
- Estimating the overall cost and schedule for the entire project (and more detailed estimates for the elaboration phase that will immediately follow);
- Estimating potential risks (the sources of unpredictability;
- Preparing the supporting environment for the project.

## 4.3 Elaboration Phase

- Defining, validating and baselining the architecture as rapidly as practical;
- Refining the Vision, based on new information obtained during the phase, establishing a solid understanding of the most critical use cases that drive the architectural and planning decisions;
- Creating and baselining detailed iteration plans for the construction phase;
- Refining the development case and putting in place the development environment, including the process, tools and automation support required to support the construction team;
- Refining the architecture and selecting components, Potential components are evaluated and the make/buy/reuse decisions sufficiently understood to determine the construction phase cost and schedule with confidence. The selected architectural components are integrated and assessed against the primary scenarios.

## 4.4 Construction Phase

- Resource management, control and process optimization;
- Complete component development and testing against the defined evaluation criteria;
- Assessment of product releases against acceptance criteria for the vision.

## 4.5 Transition Phase

- Executing deployment plans;
- Finalizing end-user support material;
- Testing the deliverable product at the development site;
- Creating a product release;
- Getting user feedback;
- Fine-tuning the product based on feedback;
- Making the product available to end users.

# 5 eXtreme Programming (XP)

## 5.1 Introduction

### 5.1.1 Agile Methods

Dissatisfaction with the **overheads** involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods focus on the code rather than the design, are based on an iterative approach to software development and are intended to deliver working software quickly and evolve this quickly to meet changing requirements.

The aim of agile methods is to reduce overheads in the software process and to be able to **respond quickly to changing requirements** without excessive rework.


Figure 5: Agile Manifest.

### 5.1.2 eXtreme Programming

One of the first agile methods developed by Kent Beck in the late 90's. XP is a "light-weight methodology for **small to medium-sized teams** developing software in the face of **vague or rapidly changing requirements**". It is an alternative to "heavy-weight" software development models (which deal poorly with changes).

### 5.1.3 XP motto: Embrace Change

In traditional software life cycle models, the cost of changing a program rises exponentially over time. A key assumption of XP is that the cost of changing a program can be hold mostly constant over time.

Hence XP is a lightweight (agile) process:

- Instead of lots of documentation nailing down what customer wants up front, XP emphasizes plenty of feedback;
- Embrace change: iterate often, design and redesign, code and test frequently, keep the customer involved;
- Deliver software to the customer in short iterations;
- Eliminate defects early, this reducing costs.

## 5.2 XP Values

### 5.2.1 Communication

XP emphasizes value of communication (namely face to face communication) in many of its practices. XP employs a coach whose job is noticing when people aren't communicating and reintroduce them.

### 5.2.2 Simplicity

"Do the simplest thing that could possibly work" (DTSTTCPW) principle, elsewhere know as KISS (Keep it Simple, Stupid). A coach may say DTSTTPCW when he sees an XP developer doing something needlessly complicated.

"You ain't gonna need it" (YAGNI) principle.

### 5.2.3 Feedback

- Feedback at different time scale;
- Unit tests tell programmers status of the system;
- When customers write new user stories, programmers estimate time required to deliver changes;
- Programmers produce new releases every 2-4 weeks for customers to review.

### 5.2.4 Courage

- The courage to communicate and accept feedback;
- The courage to throw code away (prototypes);
- The courage to refactor the architecture of a system.

## 5.3 XP Practices

### 5.3.1 The Planning Game

Customer comes up with a list of desired features. Each feature is written out as a user story. A user story describes in broad strokes what the feature requires (typically written in 2-3 sentences on 4x6 story cards).

Developers estimate 'size' of each story (effort to implement). The total size done per iteration (= Project Velocity) is important to select the user stories for each iteration. Given developer estimates the project velocity and the customer prioritizes which stories to implement.

The customer decides the scope, priorities, release content and delivery dates. The developers decide the (effort) estimates for each story, the consequences, the process and the detailed (task) schedule.

It's a two planning process. Customers and developers decide about release content (user stories) and date for the next releases (**Release planning**). The developers decide about development tasks for the next iteration (possible iterations per release).

### 5.3.2 Small Releases

Start with the smallest useful feature set. Release early and often, adding a few features each time. Releases can be date driven or user story driven.

### 5.3.3 System Metaphor

The system metaphor is a story that everyone - customers, programmers and managers - can tell about how the system works. Role somewhat similar to architectural styles/patterns.

### 5.3.4 Simple Design

Use the simplest possible design that gets the job done. The requirements will change tomorrow, so only do what's needed to meet today's requirements. Avoid big up-front design.

### 5.3.5 Test-driven Development

**Test first**: before adding a feature, write a test for it. If code has no automated tests, it's assumed it doesn't work. Create tests for bugs discovered, before fixing them.

**Unit tests** (or **developer tests**): testing of small pieces of functionality as developers write them. Usually for testing single methods, classes or scenarios. They are usually automated with a unit testing framework, like JUnit. Experiments show that TDD reduces debugging time.

**Acceptance Tests** (or **customer tests**): specified by the customer to check overall system functioning. A user story is complete when all its acceptance tests pass. Usually specified as scripts of UI actions and expected results. Ideally automated with a UT or AT framework, like FIT.

### 5.3.6 Refactoring

Recently updated to "Design Improvement", refactoring is the act of improving the structure of the code without changing externally visible behavior.

### 5.3.7 Pair Programming

**How it works:**

- Two programmers work together at one machine;
- Driver enter code, while navigator critiques it;
- Periodically switch roles and pairs;
- Requires proximity in lab or work environment.

**Advantages:**

- Serves as an informal review process;
- Helps developing collective ownership and spread knowledge;
- Improves quality (less defects, better design), whilst maintaining (or improving) productivity.

### 5.3.8 Collective Code Ownership

No single person "owns" a module but any developer can work on any part of the code base at any time.

The advantages of this are: no islands of expertise develop, all the developers take responsibility for all of the code, pressure to create better quality code and change of team members is less of a problem.

### 5.3.9 Continuous Integration

All changes are integrated into the code base at least daily as opposed to "big-bang integration". Tests have to always run before and after integration. This enables frequent releases.

### 5.3.10 Sustainable Pace

Also know as "40 hours work a weed". Programmers are supposed to go home on time. This makes them "fresh and eager every morning but tired and satisfied every night". In crunch mode, up to one week of overtime is allowed. More than that and there's something wrong with the process.

### 5.3.11 On-Site Customer

Recently updated to "Whole Team". Development team has continuous access to a real live **customer**, that is, someone who will actually be using the system or a proxy (in Scrum: product owner).

### 5.3.12 Coding Standards

Everyone codes to the same standards. Codding standards typically cover: naming conventions, file organization, indentation, comments, declaration, statements, white space, etc...

Ideally, you shouldn't be able to tell by looking at it who on the team has touched a specific piece of code.

Coding standards are important because:

- 80% of the lifetime of a piece of software goes to maintenance;
- Hardly any software is maintained for its whole life by the original author;
- Coding standards improve software readability, allowing engineers to understand new code more quickly and thoroughly.

## 5.4 Conclusions

Extreme programming is a well-known **agile method** that integrates a range of good **engineering and management practices** such as frequent releases of the software, continuous design improvement and customer participation in the development team.

A particular strength of extreme programming is the development of **automated tests** before a program feature is created. All tests must successfully execute when an increment is integrated into a system.

# 6 Requirements Engineering

## 6.1 Scope, Importance and Challenges

### 6.1.1 What is requirements engineering

Requirements engineering (RE) is the process of studying customer and user needs to arrive at a definition of **system**, hardware, or **software** requirements.

**Software Requirement** is a property which must be exhibited by software developed or adapted to solve a particular problem.

### 6.1.2 Main problems of RE

- Requirements (mis)communication and (mis)understanding;
- Evolving requirements: Changing and Growing;
- Requirements creep: uncontrolled changes or continuous growth in a project's requirements.

## 6.2 Classification of Software Requirements

### 6.2.1 Levels of Requirements

- **Business requirements/needs** represent high-level objectives of the organization or **customer** who requests the system. (Can be captures in vision and scope document)
- **User requirements/needs** describe user goals or tasks that the **user** must be able to perform with the product. (Can be captured in use case models)
- **System requirements** are the requirements for the system as a whole (which frequently include hardware and software components). (Can be captured in a system requirements specification document)
- **Software requirements** are derived from system requirements (by allocating system req.'s to software components and detailing them).

(Can be captured in a software requirements specification (SRS) document)

### 6.2.2 Types of Requirements

- Functional requirements describe the functions that the software is to execute, also known as capabilities;
- Nonfunctional requirements are the ones that act to constrain the solution. Most of them are quality requirements (can be defined on quality characteristic and quality metrics) but can also include development process requirements.

### 6.2.3 ISO/IEC 25010: Product Quality Characteristics

- **Functionality suitability** - degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions;
- **Perfomance efficiency** - performance relative to the amount of resources used under stated conditions;
- **Reliability** - degree to which a system, product or component performs specified functions under specified conditions for a specified period of time;
- **Usability** - degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use;
- **Compatibility** - degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment;
- **Maintainability** - degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers;
- **Portability** - degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another;
- **Security** - degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization.

## 6.3 Requirements Engineering in Some Well-Known Processes

### 6.3.1 Agile Methods and Requirements

Many agile methods argue that producing detailed system requirements is a waste of time as requirements change so quickly. Requirements documents are therefore always out of date.

Agile methods usually use incremental requirements engineering and may express requirements as user stories. This is practical for business systems but may be problematic for systems that require pre-delivery analysis or systems developed by several teams.

### 6.3.2 User Stories - INVEST

- **I**ndependents;
- **N**egotiable;
- **V**aluable;
- **E**stimable;
- **S**mall (Sized appropriately);
- **T**estable.

## 6.4 Requirements Engineering Process

### 6.4.1 Requirements Elicitation(or discovery)

The goal is to interact with stakeholders and other sources (documents, existing systems, etc...) to collect/discover their requirements.

The various techniques used are:

- Interviews;
- Facilitated meetings;
- Questionnarires;
- Goal Analyses;
- Social Observation and analysis (of how people actually work);
- (User-Interface) Prototypes;

- Scenarios, user stories, use cases (real-life usage examples);
- Social Media Analysis (including customer reviews).

### 6.4.2 Stakeholders

The stakeholders are people who will be affected by the system and who have a direct or indirect influence of the elaboration of requirements. Some example of stakeholders are: customers, end users, managers, people responsible for maintaining the system, clients of the organization that may use the system, regulatory and certification bodies, etc...

### 6.4.3 Requirements Analysis and Negotiation

The goal is to detect and resolve problems with the requirements (conflicts, omissions, ambiguity, etc..) by grouping related requirements and organize them in clusters. In the end, we should arrive at a list of agreed requirements.

The techniques used are:

- Checklists - helps discovering recurring problems;
- Modeling - formalization helps discovering inconsistencies;
- Requirements classification and prioritization.

### 6.4.4 Requirements Analysis Checklist

| Characteristic | What to consider |
|---|---|
| **Completeness** | Is anything missing or forgotten? Is it thorough? Does it include everything necessary to make it stand alone? |
| **Consistency** | Are there any requirements conflicts? |
| **Unambiguity** | Is there a single interpretation? Or is it vague? |
| **Verifiability** | Is enough information provided to create acceptance tests to check requirements implementation? |
| **Necessity** | Is the feature within the system scope? Is it traceable to an original customer need? Does the specification stick with defining the product and not the underlying design? |
| **Feasibility (Realism)** | Can the features be implemented with the available technology & budget? |

Figure 6: Requirements Analysis Checklist.

### 6.4.5 Requirements Specification

Produce a Software Requirements Specification (SRS) document. It is often accompanied by other artifacts such as:

- **Other Documents** - Preliminary user manual (sometimes created instead of the SRS) or a glossary (business and technical terms);
- Tables and Matrices - Requirements attributes tables and traceability matrices (requirements to user needs, req.s to test cases, etc..);
- Prototypes - User-Interface prototypes (may be embedded in preliminary user manual);
- Models - Use Case models + Domain Models (in UML), or Data-Flow diagrams (DFD's) + Entity-Relationship Diagrams (ERD's), or formal models (for critical systems).

### 6.4.6 Requirements Validation

The goal is to demonstrate that the requirements define the system that the customer really wants. This is important because requirements error costs are high so validation is very important. Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

The techniques used are: requirements reviews and inspections, prototyping, acceptance test case generation and model validation.

## 6.5 Requirements Elicitation Techniques

### 6.5.1 Interview

Mostly widely used requirements elicitation technique. There are 3 types of interviews:

- **Open/Unstructured** - Various issues are explored with stakeholders. It is better for initial exploration and for developing new/innovative requirements;
- **Closed/structure** - based on pre-determined list of questions. Better for filling knowledge gaps;
- **Mixed** - most often in practice.

Both individual or group interviews are possible. Some activities involved in an interview are:

- **Preparation** - goals, participants, location, questions, background, info;
- **Execution** - opening, questions, finalisation;
- **Follow-up** - analyse results, ask interviewees to confirm results.

### 6.5.2 Brainstorming

Useful to elicit new and innovative requirements. In each requirements brainstorming sessions there are one moderator (usually a requirements analyst) and 4-8 people with different/multiple perspectives on the product.

There are two phases in brainstorming sessions. The **Idea Generation** phase in which participants are encouraged to come up with as many ideas as possible, without discussion of the merits of the ideas. This phase as some rules:

- Quantity over quality (as many ideas as possible;
- Free Association and visionary thinking are explicitly desired;
- Take on and combine expressed ideas;
- Do not criticize;
- Questions for clarification of ideas;
- Do not abort the brainstorming at the first deadlock, make a short break;
- Let the brainstorming come to a natural end.

The other phase is the **Consolidation** phase where ideas are discussed, revised and organized.

### 6.5.3 Questionnaires (surveys)

Well-suited for confirming/prioritizing previously identified candidate requirements. A set of questions are sent to a (potentially large) number of stakeholders. Very limited suitability for developing new and innovative requirements.

Steps:

- **Preparation** - select questions and target participants; prepare (Web) form;
- **Execution** - contact participants, remind deadlines, thank answers;
- **Follow-up** - check data quality, compute statistics, inform participants about the results.

### 6.5.4 Goal Analysis

Hierarchical decomposition of stakeholder goals to derive system requirements. A goal is a desired state while a requirement is a desired property of a system.

Benefits of focusing on the notion of goals in RE:

- Helping identifying requirements (ask why, how several times);
- Helping justifying the presence of requirements;
- Helping detecting and resolving requirements conflicts.

### 6.5.5 Social Observation and Analysis

Requirements can be derived from the external observation of the routine way and tactics of work. Many systems are developed to support people work.

People often find it difficult to tell how they perform routine tasks and work with others. When tasks become routine and people don't think much about them consciously, it is hard to verbalize how the work is done.

### 6.5.6 Prototyping

A prototype is an initial/primitive version of a system (cheaper, easier and faster to develop than than the real system but with limited functionality).

User interface prototypes give an early preview of what the final system will look and work like, and are used in RE to address areas of higher uncertainty and risks of misunderstandings and validate previously identified requirements and identify new ones.

There are 2 types of prototyping. There is the throw-away prototypes (paper or computer based) that ensure focus on requirements rather than implementation constraints. And there are the evolutionary prototypes that are appropriate for rapid, iterative, application development with strong end user involvement.

**Paper prototyping** is quick, easy and cheap to develop but it has low fidelity. Usually the preferred approach for requirements elicitation.

**Computer-Based Prototypes** require more time, skills and cost to develop but they have a higher fidelity. They are functional, evolutionary prototype or non-functional, throwaway drawings and mock-ups.

# 7 Use Case Modeling

## 7.1 System Models in Requirements Engineering

A system model is a simplified representation of a system (as-is or to-be) from a certain perspective. Models are used in many fields of engineering to tackle complexity through abstraction. Semi-formal models also help removing the ambiguity and lack of structure inherent to natural language descriptions.

The following are helpful in requirements engineering:

- **Use case model** - for organizing functional requirements (in a way closer the structure of final system);
- **Domain model** - for organizing the vocabulary and information requirements.

## 7.2 Use Case Diagrams

Use case model is a use case diagram(s) plus associated documentation. It shows the **Actors**, the user roles or external systems, the **Use case**, system functionalities or services as perceived by users (types of interactions between actors and the system). It also shows the relationships between Actors and Use cases.

The purpose of this diagrams is to show the system purpose and usefulness. It also captures functional requirements (through the use cases) and specifies the system context (through the actors).

This diagrams are not applicable only to software systems.

## 7.3 Actors

Actors are the user role or external systems. An actor (in relation to a system) is a role that someone or something of the surrounding environment plays when interacting with the system. An actor is not an individual since the same individual can interact with the system in various roles.

## 7.4 Use Cases

Use cases are functionalities or services as perceived by users. They are a type of interaction between actors and the system. They can also be a sequence of actions, including variants, resulting in an observable result with value for an actor. The name of a Use Case must show purpose and it should be given from the perspective of the actor.

## 7.5 Relationships: Generalization

### 7.5.1 Actors

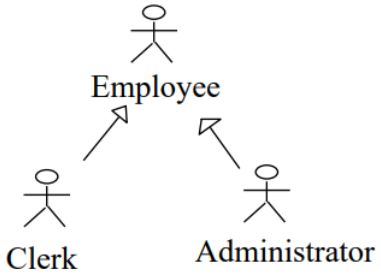


Figure 7: Example of Actors Generalization.

Generalization relationship between a more general concept and a more specialized concept. It should be possible to read "is a (special case of)". Specialized actors inherit use cases of more generic actors. Allow simplifying and structuring diagrams.

### 7.5.2 Use cases

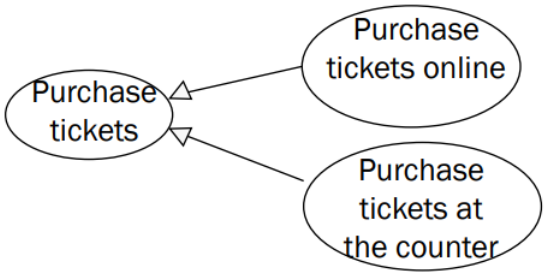


Figure 8: Example of Use Cases Generalization.

The specialized use case inherits the behavior, meaning and actors from the generic use case, and may add behavior. It should be possible to read "is a (special case of)".

### 7.5.3 Extend

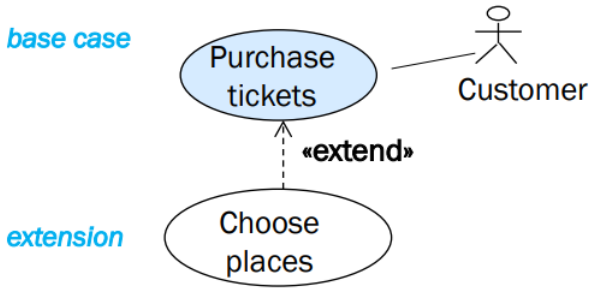


Figure 9: Example of Extend Generalization.

Extensions to base cases indicate conditionally added behaviors. they allow to highlight optional features and distinguish what is mandatory or essential from what is optional or exceptional. Actors interact with the base case, which should make sense alone.

### 7.5.4 Include

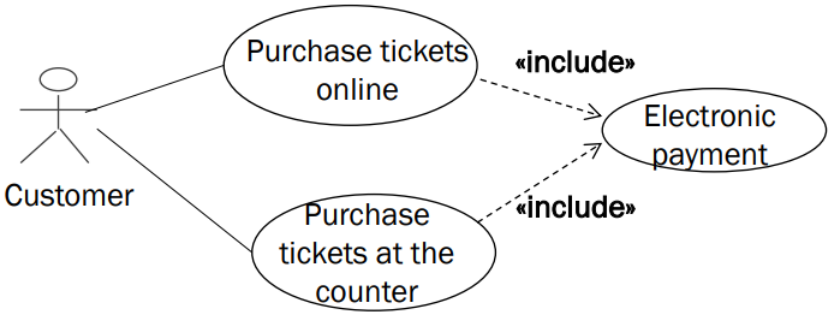


Figure 10: Example of Include Generalization.

When several use cases share some common behavior, that common behavior can be separated and described in a new use case which is included by the first ones. Inclusion is mandatory. In the textual description, write: include(Electronic payment).

# 8 User Stories

## 8.1 What Stories Are

### 8.1.1 What problem do stories address?

Software requirements is a communication problem. Those who want the software must communicate with those who will built it.

### 8.1.2 Balance is Critical

If either side dominates, the business loses.

If the business side dominates functionality and dates are mandated with little regard for reality or whether the developers understand the requirements.

If the developers dominate technical jargon replaces the language of the business and developers lose the opportunity to learn from listening.

### 8.1.3 Resource Allocation

We need a way of working together so that resource allocation becomes a shared problem. Project fails when the problem of resource allocation falls too far to one side.

If developers are responsible they may trade quality for additional features. They also may only partially implement a feature or may solely make decisions that should involve the business.

If the business is responsible they will make lengthy upfront requirements negotiation and sign off. Features will be progressively dropped as the deadlines nears.

### 8.1.4 Imperfect Schedules

We cannot perfectly predict a software schedule. As users see the software, they come up with new ideas. There are also too many intangible and developers have a notoriously hard time estimating. If we can't perfectly predict a schedule, we can't perfectly say what will be delivered.

In order to combat that, we make decisions based on information we have and we do it often. Rather than making one encompassing set of decisions we spread decision-making across the project. This is where user stories come in.

### 8.1.5 Three Cs

- **Card** - stories are traditionally written on note cards. Cards may be annotated with estimates, notes, etc..
- **Conversation** - Details behind the story come out during conversations with product owner.
- **Confirmation** - Acceptance tests confirm a story was coded correctly.

## 8.2 Why User Stories

User stories shift the focus from writing to talking. So stories are very understandable. This is, developers and customer understand them. Also, people are better able to remember events if they are organized into stories.

User stories support and encourage iterative development. We can easily start with epics and desegregate closer to development time.

Stories are the right size for planning and support opportunistic development. We design solutions by moving opportunistically between top-down and bottom-up approaches. Stories also support participatory design.

In the end, don't forget, the story text we write on cards is less important than the conversations we have.

# 9 Architectural Design

## 9.1 The Role of Software Architecture

Software architecture has various roles:

- To control overall system complexity (technological and scale);
- To ensure system integrity;
- To ensure required quality attributes: scalability, interoperability, usability, performance, cost, schedule, modularity, etc...;
- To improve development predictability;
- To establish trade offs that influence system development and future evolution;
- To ease the collaboration between different development teams;
- The Key Challenge: "to architect and design timeless and (ultra) large-scale software..".

## 9.2 Software Architecture

Software Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

## 9.3 Architectural Knowledge: Reusing

To architect solutions from scratch is hard and not effective in most cases.

Good architects reuse solutions that worked well before and adapt them smoothly and incrementally to new situations.

Therefore it is very importance:

- To know well proven solutions;
- To reuse generic solutions;
- To adapt existing solutions;
- And then to add innovation over them to fit the needs.

## 9.4 Architectural Patterns

### 9.4.1 Model-View-Controller

Used for interactive processing. Separates presentation (V) and interaction (C) from the application data/state (M), by structuring the system into three logical parts that interact with each other. It is commonly used in GUI and Web Development frameworks.

### 9.4.2 Pipes and Filters (or Data Flow)

Used for batch processing. Organizes the system as a set of data processing components (filters), connected so that data flows between components for processing (as in a pipe). The name comes from Unix.

### 9.4.3 Layered Architecture

Used for complex systems with functionalities at different levels of abstraction. Organizes the system into a set of layers, each of which groups related functionality and provides services to the layer above.

An example can be **Strict** or **Relaxed**. **Strict** means that each layer can only interact with the layer directly below (default). **Relaxed** mean each layer can interact with any lower layers (to avoid).

### 9.4.4 Repository Architecture (data centric

Used for accessing and manipulating shared data by multiple subsystems. All data in a system is managed in a central repository that is accessible to all system components or subsystems. Components or subsystems do not interact directly, only through the repository.

There are two variants: **Passive repository**, passively accepts requests from components, **Active repository**, changes to the repository trigger component execution.

## 9.5 Typical Outputs of Architectural Design

**Architecture description document:**

- Architecturally significant requirements;
- Architectural Views;
- Components' responsibilities and interfaces;
- Common Mechanisms;
- Technologies used and rational;
- Other significant decisions.

**Architectural models:**

- UML;

**Architectural prototypes:**

- For validating architectural design decisions;

**Architectural test scenarios:**

- Clarify interactions among components;

## 9.6 4+1 View Model of Software Architecture

The **logical view** shows the key abstractions in the system as objects, object classes, or their packages (UML package diagrams). The **Implementation view** shows how the software is decomposed (into software components) for development (UML deployment diagrams). The **process view** shows how, at run-time, the system is composed of interacting processes (UML activity diagrams). The **Use Case view** (+1) relates the other views.

## 9.7 Component Diagrams

### 9.7.1 Components

A **component** represents a modular part of a system that encapsulates its contents and whose manifestation in replaceable with its environment. Components are the central concept in **Component-based Software Engineering** (CBSE) as units of **reuse**.

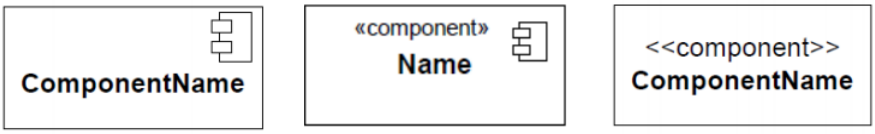


Figure 11: Example of Components.

### 9.7.2 Interfaces

A component defines its behavior in terms of **Interfaces provided** (or **realized**), and **Interfaces required** (or **used**). Components with the same interfaces are interchangeable.

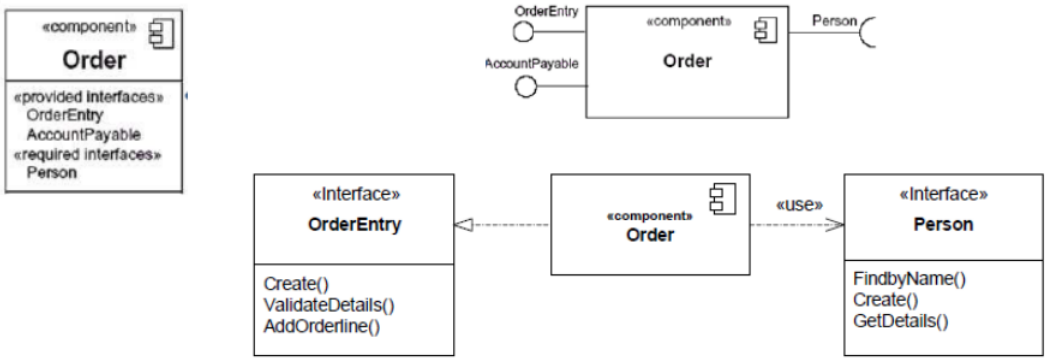


Figure 12: Example of Interfaces.

### 9.7.3 Dependencies

To promote interchangeability, components shouldn't depend directly on other components but rather on interfaces (that are implemented by other components).

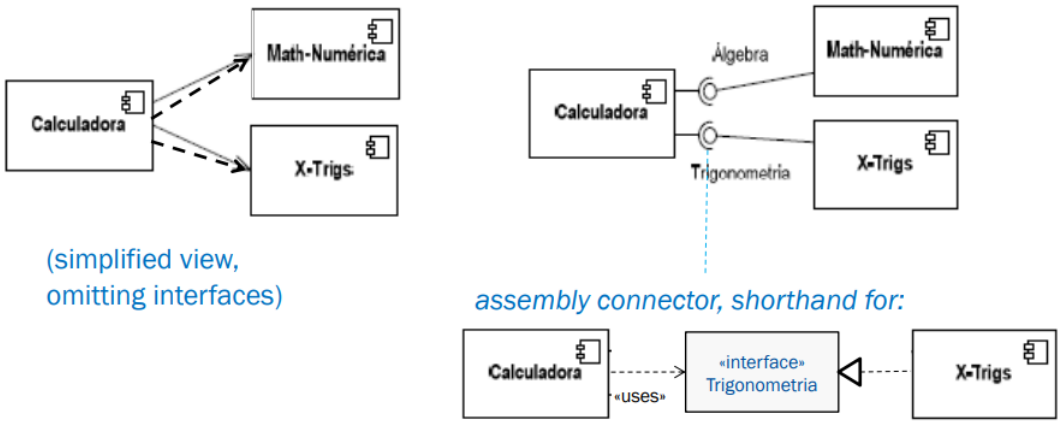


Figure 13: Example of Dependencies.

### 9.7.4 Components and Classes

The behavior of a component is usually realized (implemented by its internal classes.

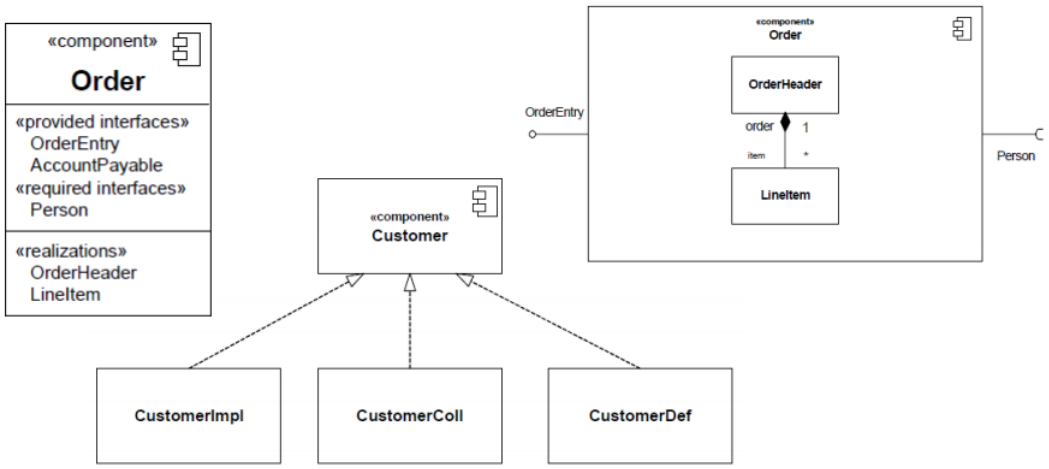


Figure 14: Example of Components and Classes.

### 9.7.5 Components and Artifacts

Components manifest physically as artifacts (that may be deployed in hardware nodes).

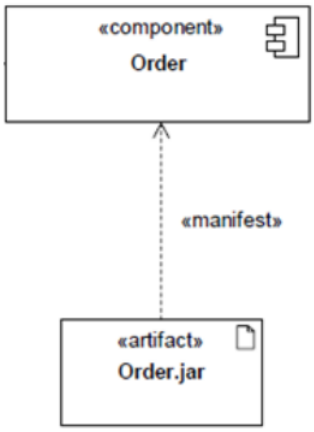


Figure 15: Example of Components and Artifacts.

## 9.8 Deployment Diagrams

### 9.8.1 Nodes

Nodes are computational resources where artifacts may be deployed. They are connected by communication associations and may be represented as types or instances. they may have attributes, methods and stereotypes (as text or icon).

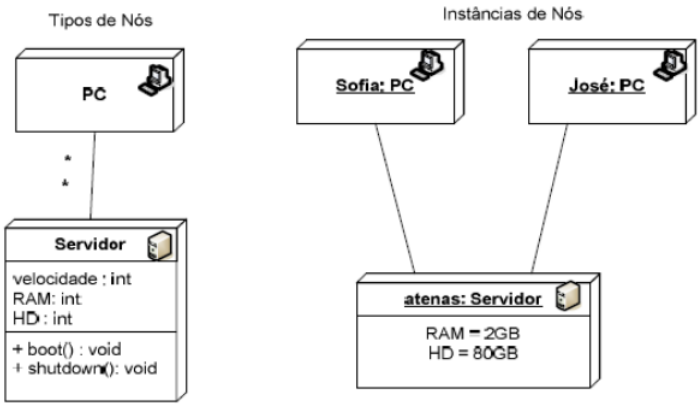


Figure 16: Example of Nodes.

### 9.8.2 Artifacts

Artifacts are physical information elements used or produced by a software development process or by the installation or operation of a system, like for example model files, source code files, executable files, scripts, etc...

Artifacts are deployed on hardware nodes as types or instances. It is possible to indicate dependencies between artifacts.

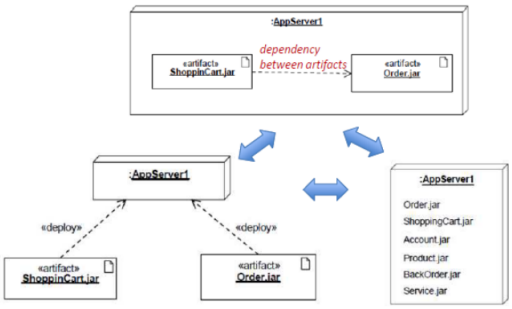


Figure 17: Example of Artifacts.

## 9.9 Package Diagrams

Packages are a grouping mechanism in UML. They may group elements of any type (even other packages). For the logical architecture, packages typically group classes. May have stereotypes as "system", "subsystem", "layer", etc...

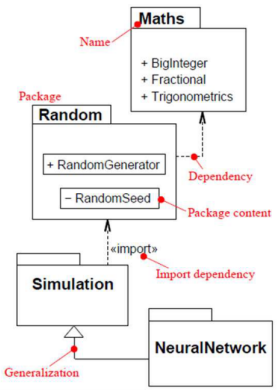


Figure 18: Example of a Package Diagram.

# 10 Project Management

## 10.1 Controlling Software Projects

Four control variables require balancing: Resources, Time, Scope and Quality. It is not advisable to set a priori the value of all variables simultaneously, if we want a successful project. "scope" is often the most important variable to control.

## 10.2 The Resource Variable

Staffing is usually the **least effective** variable to adjust.

- Staffing increases have long lead times;
- Increased intensity has diminishing returns;
- Team culture requires some degree of stability.

Tools and technology can provide benefits.

- Effective tools provide continuing benefits;
- Front-end costs need to be carefully amortized;
- The wrong tools and technology increase friction.

## 10.3 The Time Variable

Can be the most **most painful** variable to adjust.

- Early commitments are usually date-based;
- Target dates are often the most important objective.
- Within a date boundary, there's a only so much time.

## 10.4 The Scope Variable

Can be the **most effective** variable to adjust.

- Can adjust scope breadth - what's included;
- Can adjust scope depth - refinement;
- Partial scope can often generate immediate returns;
- It is often preferable to reach a date with partial scope completely finished, rather than complete scope partially finished.

## 10.5 Iterative and Incremental

### 10.5.1 Iterative:

- Repeatedly executing nested process cycles;
- Iterations provide synchronizing points;
- Iterations provide feedback points.

### 10.5.2 Incremental:

- System is built in progressive stages;
- Iterations add features and refinements;
- Increments are working systems.

## 10.6 Parallel and Concurrent Activities

### 10.6.1 Phased Approach:

- Gathers similar activity types together;
- Preference towards serial completion;
- Ultimate in phased approach is waterfall.

### 10.6.2 Concurrent and Parallel:

- Activities occur opportunistically;
- Activities of all types happening at the same time;
- Partial completion considered the norm.

## 10.7 Predictive vs Agile Planning

### 10.7.1 Predictive Planning:

- Creation of comprehensive activity-based plans;
- Execution of defined activities to follow plan;
- Management by controlling activities to conform to plan.

### 10.7.2 Agile Planning:

- Creation of prioritized set of deliverables;
- Opportunistic execution of activities to create deliverables;
- Management via feedback and adaptation.

## 10.8 Project Balance in an Agile Process

### 10.8.1 Sustainable Resource Management

- Stable Teams;
- Steady Pace;
- Favor High ROI tools and technology.

### 10.8.2 Fixed time management

- Timed-boxed development cycles.

### 10.8.3 Adaptive scope management

- Feedback-based scope adjustments.

## 10.9 Heroic vs Collaborative

### 10.9.1 Heroic development

- Heroic Development emphasizes individuals
- Activities assigned to individuals;
- Project results heavily dependent on individual performance;
- Increases "keyhole" risks.

### 10.9.2 Collaborative development

- Collaborative Development emphasizes teams
- Teams self-organize activities to meet goals;
- Teams leverage diverse skills;
- Teams mitigate keyhole risks.

## 10.10 Management by Facilitation

### 10.10.1 Traditional "Command and Control Strategy":

- Decisions made by central authorities;
- Activities delegated;
- Manager controls activities.

### 10.10.2 Replaced by "Facilitation and Empowerment Strategy"

- Decisions made by those with the most info;
- Activities accepted;
- Team self-manages and adapts;
- Organization ensures supportive environment.

## 10.11 Iteration 0

Some people think agile development gives developers license to dive in and build, spending little or no time on early requirements gathering or architectural issues.

Projects in which months and months of planning, requirements specification, and architectural philosophizing occur before they deliver any customer value are not positively evaluated.

Iteration 0 is a practice that can help teams find that middle ground and balance anticipation with adaptation.

The "0" means that this iteration has nothing useful to the customer-features, in other words-gets delivered in this time period. However, the fact that we have designated an iteration implies that the work is useful to the project team: architecture work, technological training, requirements document to base a contract signing.

## 10.12 Iterations 1-N

Iteration planning is assigning features to iterations for the duration of the project. This helps getting a feel for the project flow and determination of completion dates, staffing, costs, and other project planning information.

The activities that belong to the iteration planning are:

- Determining how identified risks will influence iteration planning;
- Identifying the schedule target;
- Establishing the milestone and iteration periods;
- Developing a theme for each iteration (or milestone);
- Assigning feature cards to each iteration, balancing customer priorities, risks, resources, and dependencies as necessary.
- Summarizing the plan in some combination of a feature-level spreadsheet plan, a feature card layout (usually on a wall), or a project parking lot;
- Calculating an initial project schedule from staff availability and total feature effort estimates;
- Adjusting the completed plan as necessary.

## 10.13 Three Types of Iteration Plans

### 10.13.1 Complete

- A complete plan with features assigned to every iteration.

### 10.13.2 Two-iteration plan

- A two-iteration plan utilizing only a next iteration and then everything after.

### 10.13.3 One Iteration

- An iteration-by-iteration plan.

### 10.13.4 Best Type of Plan

The best type of plan depends on the nature of the project and the expectations of customer and stakeholders. High exploration-factor projects suggest "one-iteration" plan that selects features for the first iteration, and continue only with a vague idea of the rest of the project.

## 10.14 Next Iteration Plan

### 10.14.1 Activities

- To construct the list of activities to implement each feature, recording it on the back of the card.

### 10.14.2 Re-estimation

- The team re-estimates the work effort based on the more detailed assessment and adjusts the features planned for the iteration if necessary.

### 10.14.3 Assignment

- Team members sign up for features or activities based on their skills and/or desires;
- Taking on the responsibility for getting the work done reinforces each individual's commitment to the project and thereby contributes to building a self-organizing team..

## 10.15 Estimation

### 10.15.1 How to estimate by features rather than activity?

- Estimate requirements gathering on a feature-by-feature basis, instead of for the overall project.

### 10.15.2 How to do progressive estimation?

- Bottom up and top down, comparisons to similar projects, and using estimating tools, can help teams arrive at better overall project estimates, but they can't make up for uncertainty;
- Multiple techniques can provide a better estimate for the entire project;
- Team member estimates should be used for the next iteration plan.

### 10.16 Scope Evolution

- Some scope changes are inexpensive but valuable;
- Some scope changes are extensive and expensive but crucial to delivering customer value ;
- In general, scope changes incorporated to meet evolving customer requirements and undertaken with an understanding, and approval, of their impact on the project increase the probability of project success;
- Agile development encourages change that arises from evolving knowledge, while at the same time it discourages the gold plating and requirements bloat that often occur in traditional up-front requirements gathering;
- Agile development is about focus and balance—focusing on the project's key vision and goals and forcing hard trade off decisions that bring balance to the product;
- Agile development plans by feature, in customer terminology, thereby concentrating the planning process on something the customer can relate to and prioritize easily;
- Because plans are adjusted each iteration based on actual development experience, not someone's guesses or wishes, nice-to-have features are pushed into later iterations and are often eliminated completely;
- A product's scope should be driven by customer value, technical feasibility and cost, the impact on a product's adaptability, and critical business schedule needs. It should not be held hostage to a plan developed when our product and project knowledge was still in its infancy.

# 11 Behaviour-Driven Development (BDD)

## 11.1 Definition

"BDD is a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters" - Dan North

"It's using examples to talk through how an application behaves... And having conversations about those examples." - Liz Keogh

Focused on finding the places where there's a lack of mismatch of understanding.

Deliberate discovery: assume that we don't know what we're doing.

Uses examples to promote conversations.

Requires tools but the main focus are the conversations, not the tools.

### 11.2 Gherkin

Language used to capture examples of scenarios. Designed to be non-technical and human readable. It is also designed to promote Behavior-Driven Development practices for the whole team. Initially created for Cucumber (BDD tool for Ruby) but now supported for many different languages.

## 12 Software Evolution

### 12.1 Software Change

Software change is inevitable because:

- New requirements emerge when the software is used;
- The business environment changes;
- Errors must be repaired;
- New computers and equipment is added to the system;
- The performance or reliability of the system may have to be improved.

A key problem for all organizations is implementing and managing change to their existing software systems.

### 12.2 Importance of Evolution

Organisations have huge investments in their software systems - they are critical business assets. To maintain the value of these assets to the business, they must be changed and updated. The majority of the software budget in large companies is devoted to changing and evolving existing software rather than developing new software.

## 12.3 Evolution and Servicing

**Evolution** is the stage in a software system's life cycle where it is in operational use and is evolving as new requirements are proposed and implemented in the system.

**Servicing** is the stage where the software remains useful but the only changes made are those required to keep it operational i.e. bug fixers and changes to reflect changes in the software's environment. No new functionality is added.

**Phase-out** is the stage where the software may still be used but no further changes are made to it.

## 12.4 Evolution Processes

### 12.4.1 Change Identification and Evolution processes

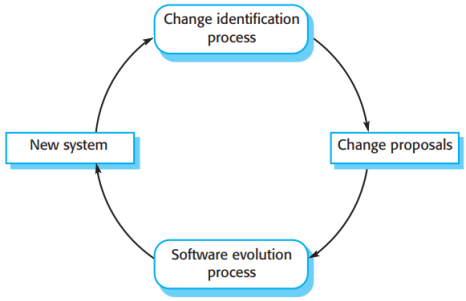


Figure 19: Change Identification Diagram.

### 12.4.2 The software evolution process

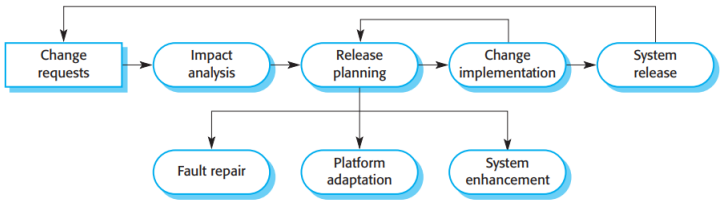


Figure 20: Evolution Process Diagram.

### 12.4.3 Change implementation

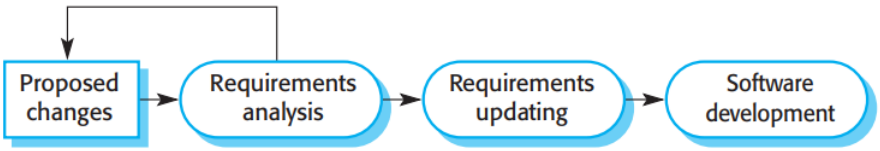


Figure 21: Change implementation Diagram.

### 12.4.4 Agile Methods and Evolution

Agile methods are based on incremental development so the transition from development to evolution is a seamless one. Evolution is simply a continuation of the development process based on frequent system releases.

Automated regression testing is particularly valuable when changes are made to a system. Changes may be expressed as additional user stories.

### 12.4.5 Handover problems

Where the development team have used an agile approach but the evolution team is unfamiliar with agile methods and prefer a plan-based approach. The evolution team may expect detailed documentation to support evolution and this is not produced in agile processes.

Where a plan-based approach has been used for development but the evolution team prefer to use agile methods. The evolution team may have to start from scratch developing automated tests and the code in the system may not have been refactored and simplified as is expected in agile development.

## 12.5 Software Maintenance

### 12.5.1 Definition

Software Maintenance is modifying a program after it has been put into use. The term is mostly used for changing custom software. Generic software products are said to evolve to create new versions.

Maintenance does not normally involve major changes to the system's architecture. Changes are implemented by modifying existing components and adding new components to the system.

### 12.5.2 Types of Maintenance

**Fault Repairs** is about changing a system to fix bugs/vulnerabilities and correct deficiencies in the way meets its requirements.

**Environmental Adaptation** is a maintenance to adapt software to a different operating environment. Changing a system so that it operates in a different environment from its initial implementation.

**Functionality addition or modification** is about modifying the system to satisfy new requirements.

### 12.5.3 Re-engineering Process Activities

**Source code translation** is when you convert code to a new language.

**Reverse Engineering** is when you analyse a program in order to understand it.

**Program structure improvement** is when you restructure a program for understandability.

**Program modularisation** is when you reorganise the program structure;

**Data re-engineering** is when you clean-up and restructure system data.

## 12.6 Key Points

Software development and evolution can be thought of as integrated, iterative process that can be represented using a spiral model.

For custom system, the costs of software maintenance usually exceed the software development costs.

The process of software evolution is driven by requests for changes and includes change impact analysis, release planning and change implementation.

Legacy systems are older software systems, developed using obsolete software and hardware technologies, that remain useful for a business. It is often cheaper and less risky to maintain a legacy system than to develop a replacement system using modern technology. The business value of a legacy system and the quality of the application should be assessed to help decide if a system should be replaced, transformed or maintained.

There are 3 types of software maintenance, namely bug fixing, modifying software to work in a new environment, and implementing new or changed requirements.

Software re-engineering is concerned with re-structuring and re-documenting software to make it easier to understand and change.

Refactoring, making program changes that preserve functionality, is a form of preventative maintenance.