CO2

2. ESTABLISHING THE GROUNDWORK

In a perfect world, software engineers and stakeholders would be on the same team. In these situations, requirements engineering is merely a matter of having fruitful discussions with coworkers who are well-known team members.

We go over the actions necessary to get the project started in a way that will keep it moving ahead toward a successful solution—to provide the framework for a comprehension of software needs.

a) Identifying Stakeholders:

A stakeholder is defined as "someone who directly or indirectly benefits from the system being designed." The typical stakeholders include business operations managers, product managers, marketers, end users, internal and external clients, consultants, product engineers, software engineers, and support and maintenance engineers. Each stakeholder sees the system differently, benefits differently when the system is successfully constructed, and is exposed to various risks if the project should fail.

b) Recognizing Multiple Viewpoints:

The requirements of the system will be examined from a variety of angles because there are numerous distinct stakeholders. The requirements engineering process will benefit from the knowledge that each of these constituencies will provide. The requirements that emerge as information from various perspectives is gathered may be contradictory or at odds with one another. All stakeholder data should be organised into categories that make it possible for decision-makers to select a set of system needs that are internally consistent.

c) ELICITING REQUIREMENTS

Problem-solving, elaboration, negotiation, and specification are all included in requirements elicitation.

1.Collaborative Requirements Gathering:

Collaborative requirements gathering has been approached in a variety of ways.

• Software engineers and other stakeholders convene and participate in meetings.

• There are specified guidelines for participation and preparedness.

• A proposed agenda is official enough to cover all pertinent topics yet informal enough to promote discussion.

• The gathering is run by a "facilitator."

• The employment of a "definition mechanism" (worksheets, flip charts, etc.).

The objective is to identify the issue, put forth components of a solution, discuss various options, and outline a preliminary set of solution criteria in an environment that supports achieving the objective.

Each participant is requested to compile a list of items that are a part of the system's environment, other objects that the system will produce, and objects that it will utilise to carry out its functions while evaluating the product request in the days leading up to the meeting. All interested parties are given the mini-specs for debate. There are modifications, deletions, and further explanations.

2.Quality Function Deployment (QFD):

QFD is a quality management technique that converts client requests into technical specifications for software. To maximise customer satisfaction from the software engineering process, QFD "concentrates on." Three categories of needs are identified by QFD:

Normal prerequisites.

The aims and objectives that are discussed for a system or product during client meetings. The customer is satisfied if these conditions are met. Examples of normal requirements include the types of graphical displays that are needed, particular system functions, and predetermined performance levels.

Expected specifications.

These specifications are built into the product or system and could be so basic that the client does not declare them directly. There will be a great deal of disappointment in their absence. Expected needs include things like simple machine-human interface, general operational accuracy and dependability, and simple software installation.

Exciting requirements.

When present, these elements go above and beyond the customer's expectations and show to be very satisfying. For instance, a new mobile phone's software has typical functionalities along with a number of unexpected features that excite every user of the device (such as a multitouch screen and visible voice mail).

3.Usage Scenarios:

As requirements are obtained, a generalised idea of the features and functions of the system starts to take shape. The utilisation of these features and functions by various end user classes must first be understood before moving on to more technical software engineering tasks. To do this, users and developers can come up with a series of scenarios that specify a general pattern of usage for the system that will be built.

4.Elicitation Work Products:

Depending on the size of the system or product to be constructed, several work products will be generated as a result of the requirements elicitation process. The work products for the majority of systems comprise a statement of need and feasibility.

• A limited statement of the system's or product's scope.

• A list of the clients, users, and other parties involved in the requirements elicitation.

• A description of the system's technical environment.

• A list of prerequisites and the relevant domain restrictions for each.

• A collection of usage scenarios that shed light on how the system or product is used in various operating environments.

• Any prototypes created to more precisely specify needs.

BUILDING THE REQUIREMENTS MODEL

A description of the informational, functional, and behavioural domains necessary for a computer-based system is what the analysis model aims to do. As you gain more knowledge about the system that has to be constructed and as other stakeholders gain a better understanding of what they actually need, the model adapts dynamically. The analysis model serves as a snapshot of needs at any given time as a result.

Elements of the Requirements Model:

The requirements for a computer-based system can be seen from a variety of angles. You are forced to analyse requirements from many angles as a result of the various ways of representation, which increases the likelihood that omissions, inconsistencies, and ambiguities will be found.

1.Scenario-based elements:

Utilizing a scenario-based approach, the system is explained from the perspective of the user. For instance, simple use cases and the use-case diagrams that go with them develop into more complex template-based use cases. The initial portion of the requirements model to be built is frequently its scenario-based components. There are three degrees of elaboration displayed, with a scenario-based portrayal as the capstone.

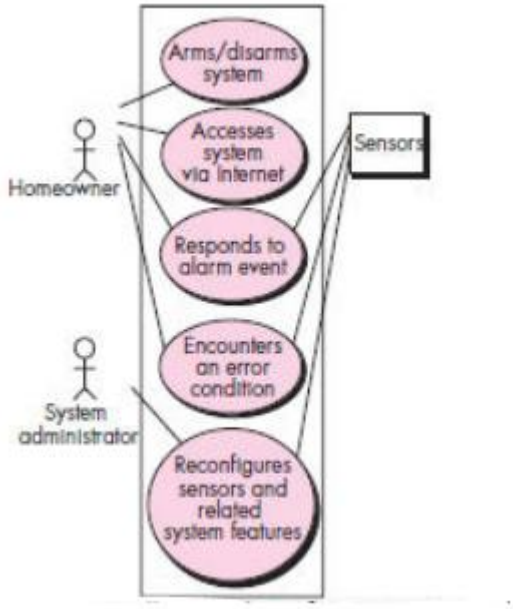


Figure 1: Use case diagram for safe home security function

2.Class-based elements.:

A set of objects that are altered as an actor engages with the system is implied by each usage scenario. These things are grouped into classes, which are groups of entities with shared characteristics and behavior. Figure 1 provides an example.

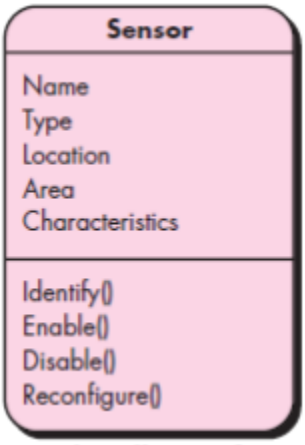


Figure 2: Class diagram for sensor

3.Behavioral elements :

The design that is selected and the implementation strategy that is used can both be significantly impacted by a computer-based system's behavior. The requirements model must therefore include modeling components that represent behavior. One way to portray a system's behavior is through a state diagram, which shows the system's states as well as the events that lead to state changes. Any

outwardly observable style of conduct is a state. The state diagram also shows the actions that have been taken as a result of a certain event.

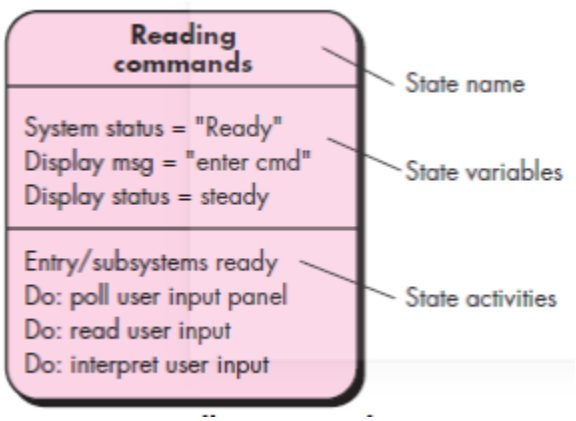


Figure 3: UML state diagram notation

4.Flow-oriented elements:

As data passes through a computer-based system, it is altered. The system accepts input in a variety of formats, transforms it using various functions, and outputs in a variety of formats. A control signal sent by a transducer, a string of numbers entered by a person, a packet of data sent over a network link, or a sizable data file retrieved from secondary storage are all examples of input. A single logical comparison, a complicated mathematical formula, or a rule-inference strategy used by an expert system could make up the transform(s).

NEGOTIATING REQUIREMENTS

The inception, elicitation, and elaboration tasks in an ideal requirements engineering scenario discover client needs in sufficient depth to move on to following software engineering activities. With one or more stakeholders, you might need to negotiate. In most cases, stakeholders are expected to weigh cost and time-to-market against functionality, performance, and other product or system features. The goal of this negotiation is to create a project plan that satisfies stakeholder needs while also taking into account the limitations that have been imposed on the software team in the real world. The most effective negotiations aim for a "win-win" outcome.

In other words, stakeholders benefit when a system or product meets the majority of their needs, and you benefit when you adhere to realistic, doable schedules and budgets.

Each software process iteration starts with a series of negotiating actions that Boehm [Boe98] describes. The following actions are defined rather than just one customer communication activity:

1. Identifying the major stakeholders in the system or subsystem.

2. Establishing the "win conditions" for the stakeholders.

3. Conciliation of the stakeholders' win conditions into a set of win-win circumstances that benefit everyone.

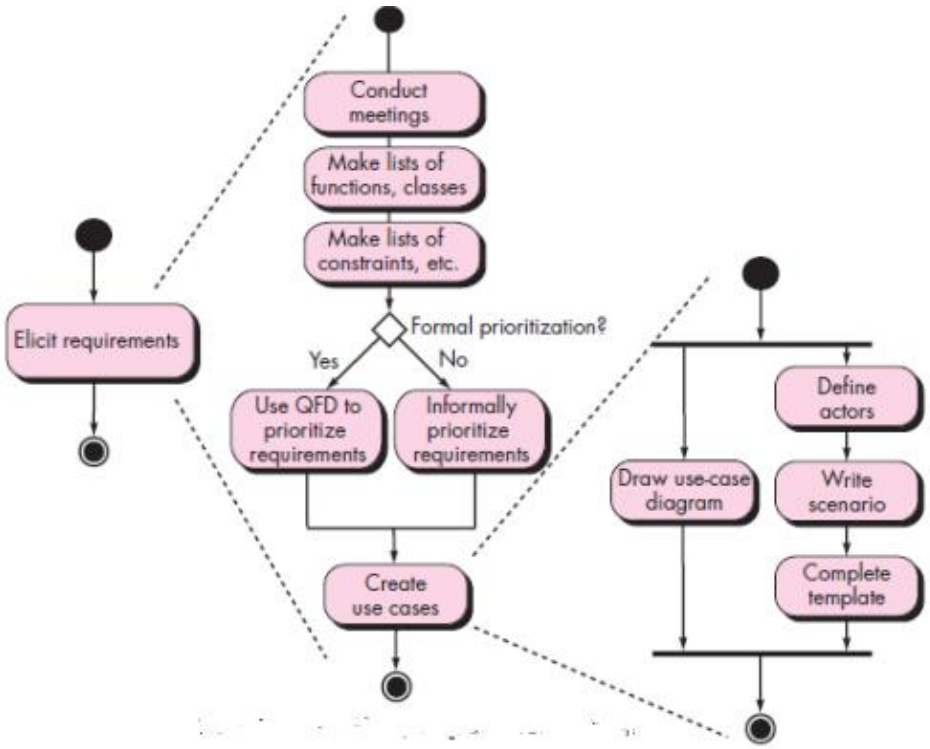


Figure 4: UML Activity diagram for Eliciting

VALIDATING REQUIREMENTS

As the requirements model is being developed, each component is checked for consistency, omissions, and ambiguity. The stakeholders have prioritised and organised the needs represented by the model into packages that will be implemented as software increments. The following inquiries are addressed by a study of the requirements model:

• Is each demand in line with the system's or product's overarching goals?

• Have all specifications been made at the appropriate level of abstraction? Do certain criteria offer a level of technical information that is improper at this point, in other words?

• Is the need actually required, or is it only an optional feature that might not be crucial to the system's goal?

• Is each demand well-defined and bounded?

• Are there any requirements that contradict with one another?

• Is each demand feasible in the system or product's intended technical environment?

• Can each criteria be tested after it is put into practice?

• Does the system-to-be-information, build's function, and behavior accurately mirror the requirements model?

• Has the requirements model been "partitioned" so that gradually more in-depth details about the system are revealed?

• Has the requirements model been made simpler by the usage of requirements patterns? Have all patterns received the necessary validation? Are all patterns in accordance with the demands of the client?

• Is each demand well-defined and bounded?

• Are there any requirements that contradict with one another?

SRS VS USER STORIES

SRS:

A document known as a software requirements specification (SRS) contains a detailed description of how the system is supposed to function. At the conclusion of the requirements engineering phase, it is often approved.

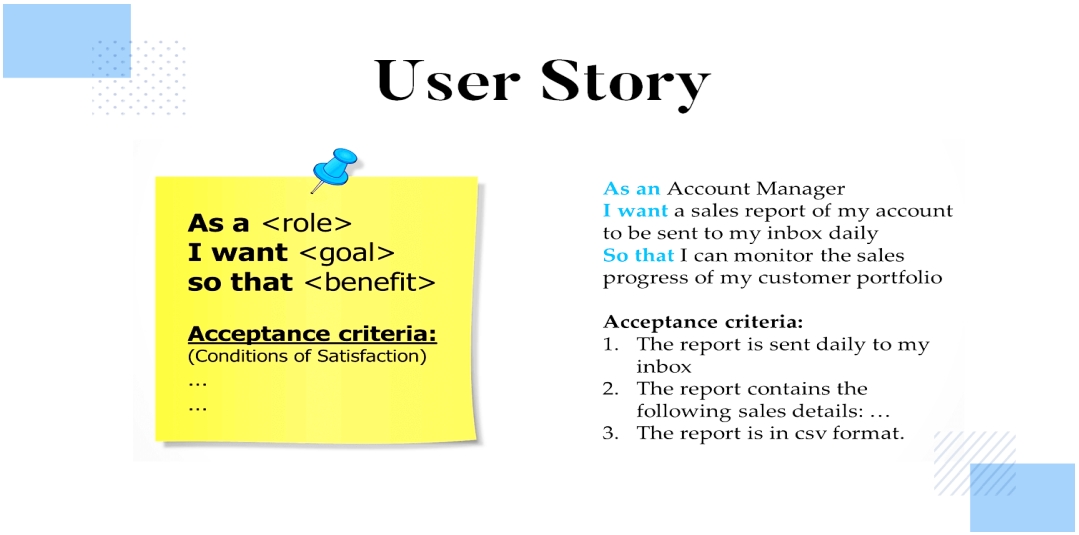
Qualities of SRS:

- Correct
- Unambiguous
- Complete
- Consistent
- Ranked for importance and/or stability
- Verifiable

- Modifiable
- Traceable

USER STORIES

A user story is a casual, all-inclusive description of a software feature written from the viewpoint of the client or end user. A user story's objective is to describe how a piece of work will provide the customer with a specific value.



SRS Vs User Stories

User Stories

- Provide a small scale and easy to use presentation of information. Are generally formulated in the everyday language of the user and contain little detail, thus remaining open to interpretation. They should help the reader understand what the software should accomplish.
- Must be accompanied by acceptance testing procedures for clarification of behavior where stories appear ambiguous.

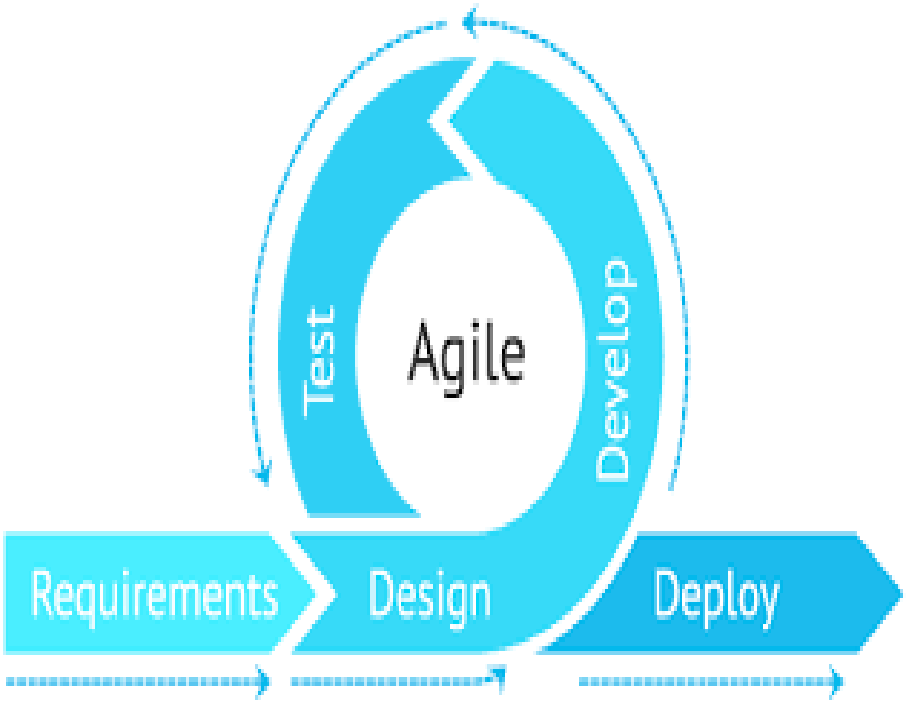
SRS

- Software requirements specification establishes the basis for an agreement between customers and contractors or suppliers on how the software product should function.

AGILE MODELING

The agile software development life cycle (SDLC) model combines iterative and incremental process models with a focus on process adaptability and customer satisfaction through quick delivery of

functional software. The product is divided into smaller incremental builds using agile methods. Iterations of these builds are supplied.



1. **Requirements gathering:** In this phase, you must define the requirements. You should explain business opportunities and plan the time and effort needed to build the project. Based on this information, you can evaluate technical and economic feasibility.

2. **Design the requirements:** When you have identified the project, work with stakeholders to define requirements. You can use the user flow diagram or the high-level UML diagram to show the work of new features and show how it will apply to your existing system.

3. **Construction/ iteration:** When the team defines the requirements, the work begins. Designers and developers start working on their project, which aims to deploy a working product. The product will undergo various stages of improvement, so it includes simple, minimal functionality.

4. **Testing:** In this phase, the Quality Assurance team examines the product's performance and looks for the bug.

5. **Deployment:** In this phase, the team issues a product for the user's work environment.

6. **Feedback:** After releasing the product, the last step is feedback. In this, the team receives feedback about the product and works through the feedback.

Following are the Agile Manifesto principles –

1. **Individuals and interactions** − In Agile development, self-organization and motivation are important, as are interactions like co-location and pair programming.
2. **Working software** − Demo working software is considered the best means of communication with the customers to understand their requirements, instead of just depending on documentation.
3. **Customer collaboration** − As the requirements cannot be gathered completely in the beginning of the project due to various factors, continuous customer interaction is very important to get proper product requirements.
4. **Responding to change** − Agile Development is focused on quick responses to change and continuous development.

Agile Principles:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4. Business people and developers must work together daily throughout the project.

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

6. The most efficient and effective method of conveying information to and within a development team is face–to–face conversation

Working software is the primary measure of progress.

8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

9. Continuous attention to technical excellence and good design enhances agility.

10. Simplicity – the art of maximizing the amount of work not done – is essential.

11. The best architectures, requirements, and designs emerge from self–organizing teams.

12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Extreme Programming

Extreme Programming (XP) is a well-known agile method; it emphasizes collaboration, quick and early software creation, and skillful development practices.

XP is a lightweight, efficient, low-risk, flexible, predictable, scientific, and fun way to develop a software.

Extreme Programming is one of the Agile software development methodologies. It provides values and principles to guide the team behavior. The team is expected to self-organize. Extreme Programming provides specific core practices where −

- Each practice is simple and self-complete.
- Combination of practices produces more complex and emergent behavior.

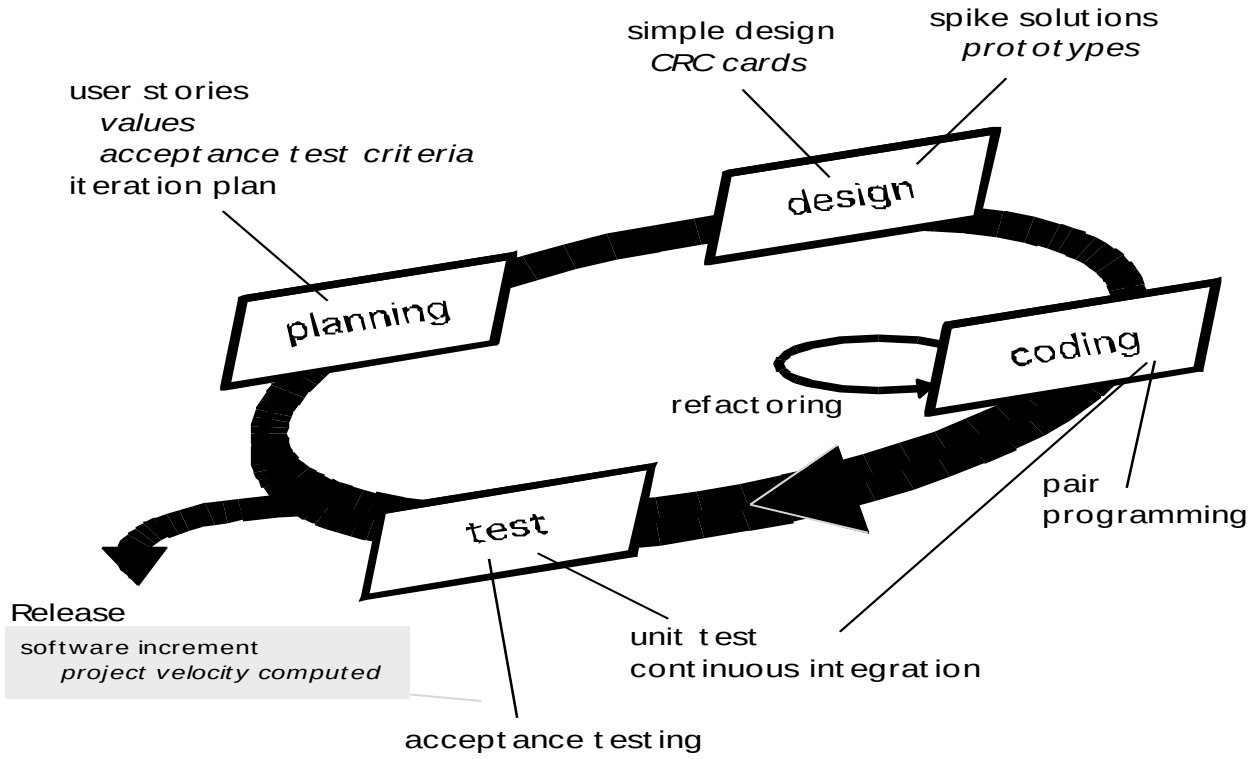
XP founded on four values:

- **communication, simplicity, feedback, and courage.**

In addition to IID, it recommends **12 core practices**:

- **1. Planning Game**
- **2. small, frequent releases**
- **3. system metaphors**
- **4. simple design**
- **7. pair programming**
- **8. team code ownership**
- **9. continuous integration**
- **10. sustainable pace**

- **5. testing** **11. whole team together**
- **6. frequent refactoring** **12. coding standards**



- XP Planning
  - Begins with the listening, leads to creation of "user stories" that describes required output, features, and functionality. Customer assigns a value(i.e., a priority) to each story.
  - Agile team assesses each story and assigns a cost (development weeks. If more than 3 weeks, customer asked to split into smaller stories)
  - Working together, stories are grouped for a deliverable increment next release.
  - A commitment (stories to be included, delivery date and other project matters) is made. Three ways: 1. Either all stories will be implemented in a few weeks, 2. high priority stories first, or 3. the riskiest stories will be implemented first.
  - After the first increment "project velocity", namely number of stories implemented during the first release is used to help define subsequent delivery dates for other

increments. Customers can add stories, delete existing stories, change values of an existing story, split stories as development work proceeds

XP Design

- Follows the KIS principle (keep it simple) Nothing more nothing less than the story.
- Encourage the use of CRC (class-responsibility-collaborator) cards in an object-oriented context. The only design work product of XP. They identify and organize the classes that are relevant to the current software increment. (see Chapter 8)
- For difficult design problems, suggests the creation of “spike solutions”—a design prototype for that portion is implemented and evaluated.
- Encourages “refactoring”—an iterative refinement of the internal program design. Does not alter the external behavior yet improve the internal structure. Minimize chances of bugs. More efficient, easy to read XP Coding

XP Coding

- Recommends the construction of a unit test for a story *before* coding commences. So implementer can focus on what must be implemented to pass the test.
- Encourages “pair programming”. Two people work together at one workstation. Real time problem solving, real time review for quality assurance. Take slightly different roles.

XP Testing

- All unit tests are executed daily and ideally should be automated. Regression tests are conducted to test current and previous components.
- “Acceptance tests” are defined by the customer and executed to assess customer visible functionality