Software development process

- A software development process is a structure imposed on the development of a software product

Software development activities

- **Requirements analysis**

  The important task in creating a software product is extracting the requirements or requirements analysis. **Customers typically have an abstract idea** of what they want as an end result, but not what software should do. Incomplete, ambiguous, or even contradictory requirements are recognized by **skilled and experienced software engineers** at this point. Frequently demonstrating live code may help reduce the risk that the requirements are incorrect

Software development activities

- **Specification**

  Specification is the task of **precisely describing the software to be written**, possibly in a rigorous way. Specifications are most important for **external interfaces** that must remain stable. A good way to determine whether the specifications are sufficiently precise is to have a **third party review the documents** making sure that the requirements and Use Cases are logically sound
- **Architecture**

  The architecture of a software system or software architecture refers to an abstract representation of that system. Architecture is concerned with making sure the software system will meet the requirements of the product, as well as ensuring that future requirements can be addressed

Software development activities

The architecture step also addresses interfaces between the software system and other software products, as well as the underlying hardware or the host operating system

- **Design, implementation and testing**

  Implementation is the part of the process where software engineers actually program the code for the project.

  Software testing is an integral and important part of the software development process. This part of the process ensures that bugs are recognized as early as possible.

  Documenting the internal design of software for the purpose of future maintenance and enhancement is done throughout development. This may also include the authoring of an API, be it external or internal.

Software development activities

- **Deployment and maintenance**

  Deployment starts after the code is appropriately tested, is approved for release and sold or otherwise distributed into a production environment.

  Software Training and Support is important because a large percentage of software projects fail because the developers fail to realize that it doesn't matter how much time and planning a development team puts into creating software if nobody in an organization ends up using it. People are often resistant to change and avoid venturing into an unfamiliar area, so as a part of the deployment phase, it is very important to have training classes for new clients of your software.

Software development activities

Maintenance and enhancing software to cope with newly discovered problems or new requirements can take far more time than the initial development of the software. It may be necessary to add code that does not fit the original design to correct an unforeseen problem or it may be that a customer is requesting more functionality and code can be added to accommodate their requests. It is during this phase that customer calls come in and you see whether your testing was extensive enough to uncover the problems before customers do.

Bug Tracking System tools are often deployed at this stage of the process to allow development teams to interface with customer/field teams testing the software to identify any real or perceived issues. These software tools both open source and commercially licensed provide a customizable process to acquire, review, acknowledge, and respond to reported issues.

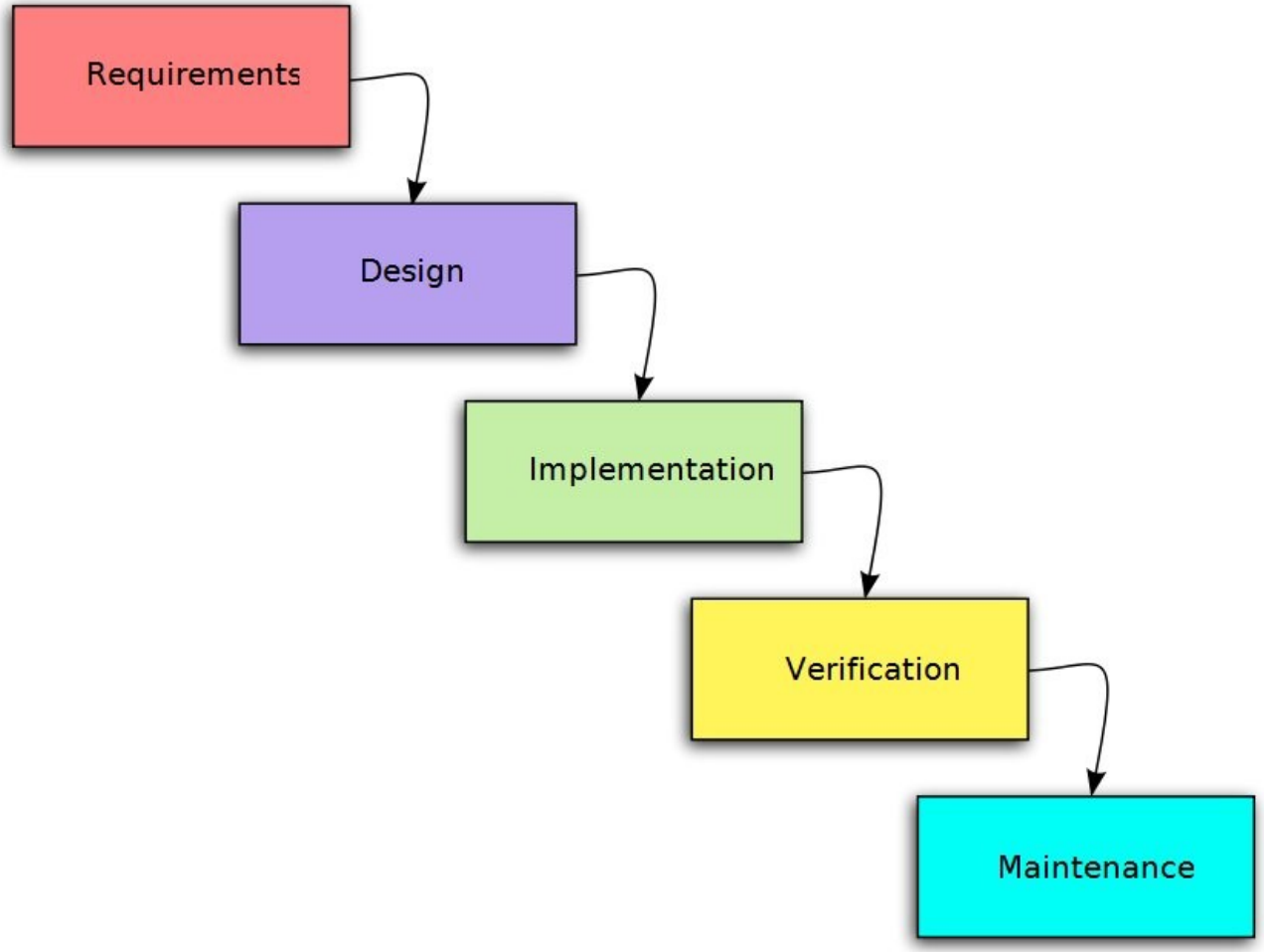
Software development methodologies

- **Waterfall model**

  The waterfall model is a **sequential development process**, in which development is seen as flowing steadily downwards (like a waterfall) through the phases of requirements analysis, design, implementation, testing (validation), integration, and maintenance. Basic principles of the waterfall model are:

  - Project is divided into sequential phases, with some overlap and splashback acceptable between phases
  - Emphasis is on planning, time schedules, target dates, budgets and implementation of an entire system at one time
  - Tight control is maintained over the life of the project through the use of extensive written documentation, as well as through formal reviews and approval/signoff by the user and information technology management

Software development methodologies



Software development methodologies

- **Prototyping**

  Software prototyping, is the framework of activities during software development of creating prototypes, i.e., incomplete versions of the software program being developed. Basic principles of prototyping are:

  - Not a standalone, complete development methodology, but rather an approach to **handling selected portions** of a larger, more traditional development methodology (i.e. Incremental, Spiral, or Rapid Application Development (RAD))
  - Attempts to **reduce inherent project risk** by breaking a project into smaller segments and providing more ease-of-change during the development process
  - **User is involved throughout the process**, which increases the likelihood of user acceptance of the final implementation

Software development methodologies

- **Small-scale mock-ups** of the system are developed following an iterative modification process until the prototype evolves to meet the users' requirements
- While most prototypes are developed with the expectation that they will be discarded, it is possible in some cases to **evolve from prototype to working system**
- A basic understanding of the fundamental business problem is necessary to avoid solving the wrong problem

- **Incremental**

  Various methods are acceptable for combining linear and iterative systems development methodologies, with the primary objective of each being to reduce inherent project risk by **breaking a project into smaller segments** and providing more ease-of-change during the development process

Software development methodologies

Basic principles of incremental development are:

- A **series of mini-Waterfalls** are performed, where all phases of the Waterfall development model are completed for a small part of the systems, before proceeding to the next incremental
- Overall requirements are defined before proceeding to evolutionary, mini-Waterfall development of individual increments of the system
- The initial software concept, requirements analysis, and design of architecture and system core are defined using the Waterfall approach, followed by iterative Prototyping, which culminates in installation of the final prototype

- **Spiral**

  The spiral model is a software development process combining elements of both **design and prototyping-in-stages**, in an effort to combine advantages of **top-down and bottom-up** concepts

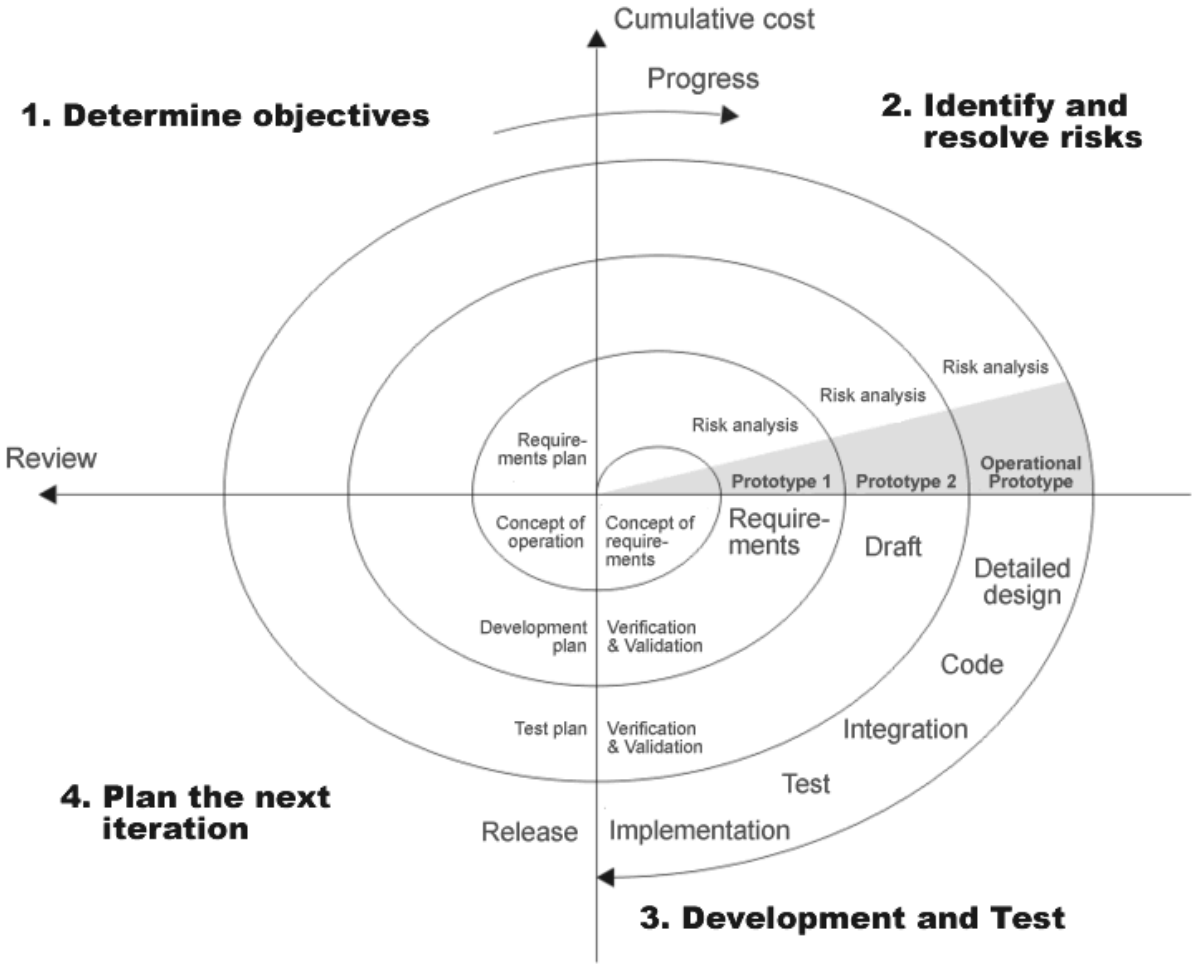
Software development methodologies

Basic principles:

- Focus is on **risk assessment** and on minimizing project risk by breaking a project into smaller segments and providing more ease-of-change during the development process, as well as providing the opportunity to evaluate risks and weigh consideration of project continuation throughout the life cycle
- Each cycle involves a progression through the **same sequence of steps**, for each portion of the product and for each of its levels of elaboration, from an overall concept-of-operation document down to the coding of each individual program
- Each trip around the spiral traverses four basic quadarants: (1) determine objectives, alternatives, and constrainst of the iteration; (2) evaluate alternatives; identify and resolve risks; (3) develop and verify deliverables from the iteration; and (4) plan the next iteration

Software development methodologies

- Begin each cycle with an identification of stakeholders and their win conditions, and end each cycle with review and commitment



Software development methodologies

- **Rapid Application Development**

  Rapid Application Development (RAD) is a software development methodology, which involves **iterative development and the construction of prototypes**.

  Basic principles:

  - Key objective is for **fast development** and delivery of a **high quality system at a relatively low investment cost**
  - Attempts to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process
  - Aims to produce high quality systems quickly, primarily through the use of iterative Prototyping (at any stage of development), **active user involvement**, and **computerized development tools**.

Software development methodologies

These tools may include Graphical User Interface (GUI) builders, **Computer Aided Software Engineering** (CASE) tools, Database Management Systems (DBMS), fourth-generation programming languages, **code generators**, and object-oriented techniques

- Key emphasis is on **fulfilling the business need**, while technological or engineering excellence is of lesser importance
- Project control involves prioritizing development and defining delivery deadlines or “timeboxes”. If the project starts to slip, emphasis is on **reducing requirements to fit the timebox, not in increasing the deadline**
- Generally includes Joint Application Development (JAD), where **users are intensely involved in system design**, either through consensus building in structured workshops, or through electronically facilitated interaction

Software development methodologies

- Active user involvement is imperative
- **Iteratively produces production software**, as opposed to a throwaway prototype
- Produces documentation necessary to facilitate future development and maintenance
- Standard systems analysis and design techniques can be fitted into this framework

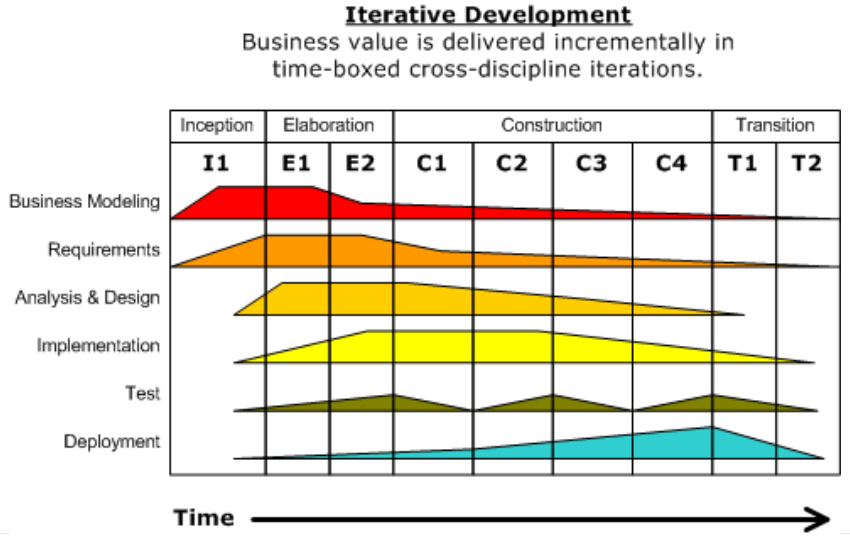
Unified Process

- The Unified Software Development Process or Unified Process is a popular **iterative and incremental software development process framework**. The best-known and extensively documented refinement of the Unified Process is the **Rational Unified Process** (RUP)
- The Unified Process is not simply a process, but rather an **extensible framework** which should be customized for specific organizations or projects
- The first book to describe the process was titled The Unified Software Development Process and published in 1999 by Ivar Jacobson, Grady Booch and James Rumbaugh
- **Iterative and Incremental**

  The Unified Process is an **iterative and incremental** development process.

Unified Process

The Elaboration, Construction and Transition phases are divided into a series of **timeboxed iterations**. (The Inception phase may also be divided into iterations for a large project). Each iteration results in an increment, which is a release of the system that contains added or improved functionality compared with the previous release. Although most iterations will include work in most of the **process disciplines** (e.g. Requirements, Design, Implementation, Testing) the relative effort and emphasis will change over the course of the project



Unified Process

- **Use Case Driven**

  In the Unified Process, use cases are used to capture the functional requirements and to define the contents of the iterations. Each iteration takes a set of use cases or scenarios from requirements all the way through implementation, test and deployment
- **Architecture Centric**

  The Unified Process insists that architecture sit at the heart of the project team's efforts to shape the system. Since no single model is sufficient to cover all aspects of a system, the Unified Process supports multiple architectural models and views. One of the most important deliverables of the process is the executable architecture baseline which is created during the Elaboration phase. This partial implementation of the system serves to validate the architecture and act as a foundation for remaining development

Unified Process

- **Risk Focused**

  The Unified Process requires the project team to focus on addressing the **most critical risks early** in the project life cycle. The deliverables of each iteration, especially in the Elaboration phase, must be selected in order to ensure that the greatest risks are addressed first
- Refinements of the Unified Process vary from each other in how they categorize the project disciplines or workflows. The Rational Unified Process defines nine disciplines: Business Modeling, Requirements, Analysis and Design, Implementation, Test, Deployment, Configuration and Change Management, Project Management, and Environment. Agile refinements of UP such as OpenUP/Basic and the Agile Unified Process simplify RUP by reducing the number of disciplines and the number of expected artifacts

Agile software development

- Agile software development refers to a group of software development methodologies based on **iterative development**, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams
- **Manifesto for Agile Software Development**

  We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
  - ***Individuals and interactions*** *over processes and tools*
  - ***Working software*** *over comprehensive documentation*
  - ***Customer collaboration*** *over contract negotiation*
  - ***Responding to change*** *over following a plan*

Agile software development

- Agile methods generally promote a project management process that encourages **frequent inspection and adaptation**, a leadership philosophy that encourages **teamwork**, **self-organization and accountability**, a set of engineering best practices that allow for **rapid delivery of high-quality software**, and a **business approach that aligns development with customer needs and company goals**
- Agile methods break tasks into **small increments with minimal planning**, and don't directly involve long-term planning
- Iterations are **short time frames** ('timeboxes') that typically last from one to four weeks. Each iteration is worked on by a team through a **full software development cycle**, including planning, requirements analysis, design, coding, unit testing, and acceptance testing when a working product is demonstrated to stakeholders

Agile software development

- An iteration may not add enough functionality to warrant a market release, but the goal is to have an available release (with minimal bugs) at the end of each iteration. Multiple iterations may be required to release a product or new features
- Team members normally take responsibility for tasks that deliver the functionality an iteration requires. They decide individually how to meet an iteration's requirements
- Agile methods emphasize **face-to-face communication over written documents** when the team is all in the same location. When a team works in different locations, they maintain **daily contact through videoconferencing, voice, e-mail, etc**
- Most agile teams work in a single open office, which facilitates such communication. **Team size is typically small** (5-9 people) to help make team communication and team collaboration easier

Agile software development

- No matter what development disciplines are required, **each agile team will contain a customer representative**. This person is appointed by stakeholders to act on their behalf and **makes a personal commitment to being available for developers to answer mid-iteration problem-domain questions**. At the end of each iteration, stakeholders and the customer representative **review progress and re-evaluate priorities** with a view to optimizing the return on investment and ensuring **alignment with customer needs and company goals**
- Agile emphasizes **working software as the primary measure of progress**. This, combined with the preference for face-to-face communication, produces **less written documentation than other methods** - though, in an agile project, documentation and other artifacts rank equally with working product

Agile software development

- Specific tools and techniques such as **continuous integration**, automated or xUnit test, **pair programming**, **test driven development**, **design patterns**, **domain-driven design**, **code refactoring** and other techniques are often used to improve quality and enhance project agility
- **Principles behind the Agile Manifesto**
  - Our highest priority is to **satisfy the customer** through early and **continuous delivery of valuable software**
  - **Welcome changing requirements**, even late in development. Agile processes harness change for the customer's competitive advantage
  - **Deliver working software frequently**, from a couple of weeks to a couple of months, with a preference to the shorter timescale
  - **Business people and developers must work together** daily throughout the project

Agile software development

- **Build projects around motivated individuals**. Give them the environment and support they need, and trust them to get the job done
- The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**
- **Working software is the primary measure of progress**
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely
- Continuous attention to technical excellence and good design enhances agility
- **Simplicity** - the art of maximizing the amount of work not done - is essential
- The best architectures, requirements, and designs emerge from **self-organizing teams**
- At regular intervals, the **team reflects on how to become more effective**, then tunes and adjusts its behavior accordingly

Agile software development

- **Agile methods**
  - Agile Modeling
  - Agile Unified Process (AUP)
  - Agile Data Method
  - Dynamic Systems Development Method (DSDM)
  - Essential Unified Process (EssUP)
  - Extreme programming (XP)
  - Feature Driven Development (FDD)
  - Getting Real
  - Open Unified Process (OpenUP)
  - Scrum

Agile Unified Process (AUP)

- Agile Unified Process (AUP) is a **simplified version** of the IBM Rational Unified Process (RUP)
- It describes a simple, easy to understand approach to developing business application software using **agile techniques and concepts** yet **still remaining** true to the RUP
- Unlike the RUP, the AUP only has **seven disciplines:**
  - **Model** - Understand the business of the organization, the problem domain being addressed by the project, and identify a viable solution to address the problem domain
  - **Implementation** - Transform model(s) into executable code and perform a basic level of testing, in particular unit testing

Agile Unified Process (AUP)

- **Test** - 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
- **Deployment** - Plan for the delivery of the system and to execute the plan to make the system available to end users
- **Configuration Management** - Manage access to project artifacts. This includes not only tracking artifact versions over time but also controlling and managing changes to them
- **Project Management** - Direct the activities that takes place within the project. This includes managing risks, directing people (assigning tasks, tracking progress, etc.), and coordinating with people and systems outside the scope of the project to be sure that it is delivered on time and within budget

Agile Unified Process (AUP)

  - **Environment** - 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
- The Agile UP is based on the following **philosophies:**
  - **Your staff knows what they're doing** - People are not going to read detailed process documentation, but they will want some high-level guidance and/or training from time to time. The AUP product provides links to many of the details, if you are interested, but doesn't force them upon you
  - **Simplicity** - Everything is described concisely using a handful of pages, not thousands of them
  - **Agility** - The Agile UP conforms to the values and principles of the agile software development and the Agile Alliance

Agile Unified Process (AUP)

  - **Focus on high-value activities** - The focus is on the activities which actually count, not every possible thing that could happen to you on a project
  - **Tool independence** - You can use any toolset that you want with the Agile UP. The recommendation is that you use the tools which are best suited for the job, which are often simple tools
  - **You'll want to tailor the AUP to meet your own needs**
- The Agile Unified Process distinguishes between **two types of iterations**. A **Development Release Iteration** results in a deployment to the Quality Assurance and/or Demo area. A **Production Release Iteration** results in a deployment to the Production area

Extreme Programming (XP)

- Proponents of Extreme Programming and agile methodologies in general regard **ongoing changes to requirements as a natural, inescapable and desirable aspect of software development projects**; they believe that adaptability to changing requirements at any point during the project life is a **more realistic** and better approach than attempting to **define all requirements at the beginning** of a project and then expending effort to control changes to the requirements
- XP sets out to reduce the cost of change by introducing basic **values**, **principles** and **practices**. By applying XP, a system development project should be **more flexible** with respect to changes
- XP describes **four basic activities** that are performed within the software development process:

Extreme Programming (XP)

- **Coding** - The advocates of XP argue that the **only truly important product** of the system development process **is code** (a concept to which they give a somewhat broader definition than might be given by others). Without code you have nothing. Coding can be drawing diagrams that will generate code, scripting a web-based system or coding a program that needs to be compiled. Coding can be used to **figure out the most suitable solution**. Coding can **help to communicate** thoughts about programming problems. Code is always **clear and concise** and cannot be interpreted in more than one way
- **Testing** - One cannot be certain of anything unless one has tested it. Testing is not a perceived, primary need for the customer. A lot of software is shipped without proper testing and still works. In software development, XP says this means that **one cannot be certain that a function works unless one tests it**

Extreme Programming (XP)

You can be uncertain whether **what you coded is what you meant**. To test this uncertainty, XP uses **Unit Tests**. These are automated tests that test the code. You can be uncertain whether **what you meant is what you should have meant**. To test this uncertainty, XP uses **acceptance tests** based on the requirements given by the customer in the exploration phase of release planning

- **Listening** - Programmers do not necessarily know anything about the business side of the system under development. **The function of the system is determined by the business side**. For the **programmers** to find what the functionality of the system should be, they **have to listen to business**. They have to try to understand the business problem, and to give the customer feedback about his or her problem, to improve the customer's own understanding of his or her problem

Extreme Programming (XP)

- **Designing** - From the point of view of simplicity, one could say that **system development doesn't need more than coding, testing and listening**. If those activities are performed well, the result should always be a system that works. In practice, **this will not work**. One can come a long way without designing but at a given time one will get stuck. The **system becomes too complex and the dependencies within the system cease to be clear**. One can avoid this by creating a **design structure that organizes the logic in the system**. Good design will avoid lots of dependencies within a system; this means that changing one part of the system will not affect other parts of the system

Extreme Programming (XP)

- Extreme Programming recognize **five values**:
  - **Communication** - Building software systems requires communicating system requirements to the developers of the system. In formal software development methodologies, this task is accomplished through documentation. Extreme Programming techniques can be viewed as methods for rapidly **building and disseminating institutional knowledge among members of a development team**. The goal is to give all developers a **shared view of the system** which matches the view held by the users of the system
  - **Simplicity** - Extreme Programming encourages starting with the **simplest solution**. Extra functionality can then be added later. The difference between this approach and more conventional system development methods is the focus on **designing and coding for the needs of today** instead of those of tomorrow, next week, or next month.

Extreme Programming (XP)

Proponents of XP acknowledge the disadvantage that this can sometimes entail more effort tomorrow to change the system; their claim is that this is **more than compensated for by the advantage of not investing in possible future requirements that might change before they become relevant**. Related to the "communication" value, simplicity in design and coding should improve the quality of communication.

Extreme Programming (XP)

- **Feedback**

  Feedback from the system - by writing **unit tests**, or running periodic **integration tests**, the programmers have direct feedback from the state of the system after implementing changes

  Feedback from the customer - The **functional tests** (aka **acceptance tests**) **are written by the customer and the testers**. They will get concrete feedback about the current state of their system. This review is planned once in every two or three weeks so the customer can easily steer the development

  Feedback from the team - When customers come up with **new requirements** in the planning game the team directly gives an estimation of the time that it will take to implement

Extreme Programming (XP)

- **Courage** - One is the commandment to always design and code for today and not for tomorrow. This is an effort to avoid getting bogged down in design and requiring a lot of effort to implement anything else. Courage enables developers **to feel comfortable with refactoring their code when necessary**. This means reviewing the existing system and modifying it so that future changes can be implemented more easily. **Courage is knowing when to throw code away**: courage to remove source code that is obsolete, no matter how much effort was used to create that source code. **Courage means persistence**: a programmer might be stuck on a complex problem for an entire day, then solve the problem quickly the next day, if only they are persistent

Extreme Programming (XP)

- **Respect** - In Extreme Programming, team members respect each other because **programmers should never commit changes that break compilation**, that make existing unit-tests fail, or that otherwise delay the work of their peers. Members respect their work by **always striving for high quality and seeking for the best design for the solution** at hand through refactoring. Nobody on the team should feel unappreciated or ignored. This ensures **high level of motivation** and encourages loyalty toward the team, and the goal of the project. This value is very dependent upon the other values, and is very much oriented toward people in a team

Extreme Programming (XP)

- Extreme Programming has **12 practices**, grouped into **four areas**, derived from the best practices of software engineering:
  - **Fine scale feedback**
    - **Pair programming**

      Pair programming means that all code is produced by two people programming on one task on one workstation. One programmer has control over the workstation and is thinking mostly about the **coding in detail**. The other programmer is more focused on the **big picture**, and is continually reviewing the code that is being produced by the first programmer. Programmers trade roles regularly. The pairs are not fixed: it's recommended that **programmers try to mix** as much as possible, so that everyone knows what everyone is doing, and everybody can become familiar with the whole system

Extreme Programming (XP)

- **Planning game**

  The main planning process within Extreme Programming is called the planning game. The game is **a meeting that occurs once per iteration**, typically once a week. The planning process is divided into two parts:

  - **Release Planning** - This is focused on determining **what requirements** are included in **which near-term releases**, and **when** they should be delivered. **The customers and developers are both part of this**. Release Planning consists of three phases:

    Exploration Phase - In this phase the customer will provide a short list of high-value requirements for the system. These will be written down on **user story cards** (write a story, estimate a story, split a story)

Extreme Programming (XP)

Commitment Phase - Within the commitment phase business and developers will commit themselves to the functionality that will be included and the date of the next release (sort by value, sort by risk, set velocity, choose scope)

Steering Phase - In the steering phase the plan can be adjusted, new requirements can be added and/or existing requirements can be changed or removed

- **Iteration Planning** - This plans the activities and tasks of the developers. In this process **the customer is not involved**. Iteration Planning also consists of three phases:

  Exploration Phase - Within this phase the requirement will be translated to different tasks. The tasks are recorded on **task cards** (translate the requirement to tasks, combine/split task, estimate task)

Extreme Programming (XP)

Commitment Phase - The tasks will be assigned to the programmers and the time it takes to complete will be estimated (a programmer accepts a task, programmer estimates the task, set load factor, balancing)

Steering Phase - The tasks are performed and the end result is matched with the original user story (get a task card, find a partner, design the task, write unit test, write code, run test, refactor, run functional test)

- **Test driven development**

  Unit tests are automated tests that test the functionality of pieces of the code (e.g. classes, methods). Within XP, **unit tests are written before the eventual code is coded**. This approach is intended to stimulate the programmer to think about conditions in which his or her code could fail.

Extreme Programming (XP)

XP says that the programmer is finished with a certain piece of code when he or she cannot come up with any further condition on which the code may fail

- **Whole team**

  Within XP, **the "customer" is not the one who pays the bill, but the one who really uses the system**. XP says that the customer should be on hand at all times and available for questions

- **Continuous process**
  - **Continuous integration**

    The development team should always be working on the **latest version** of the software.

Extreme Programming (XP)

Since different team members may have versions saved locally with various changes and improvements, they should try to upload their current version to the code repository **every few hours**, or when a significant break presents itself. Continuous integration will avoid delays later on in the project cycle, caused by integration problems

- **Design improvement**

  Because XP doctrine advocates programming only **what is needed today**, and implementing it **as simply as possible**, at times this may result in a system that is stuck. One of the symptoms of this is the need for dual (or multiple) maintenance: functional changes start requiring changes to **multiple copies of the same (or similar) code**. Another symptom is that changes in one part of the code affect lots of other parts.

Extreme Programming (XP)

XP doctrine says that when this occurs, the system is telling you to **refactor your code by changing the architecture, making it simpler and more generic**

- **Small releases**

  The delivery of the software is done in **predetermined releases** (sometimes called 'Builds'). The release plan is determined when initiating the project. Usually each release will carry a **small segment of the total software**, which **can run without depending on components that will be built in the future**. The small releases help the customer to gain confidence in the progress of the project. The small releases are **only alpha releases and are not intended to go live**. This helps maintain the concept of the whole team as the customer can now come up with his suggestions on the project

Extreme Programming (XP)

- **Shared understanding**
  - **Coding standard**

    Coding standard is an agreed upon **set of rules** that the entire development **team agree to adhere** to throughout the project. The standard specifies a **consistent style and format for source code**, within the chosen programming language, as well as various **programming constructs and patterns** that should be avoided in order to reduce the probability of defects. The coding standard may be a **standard conventions** specified by the language vendor (e.g The Code Conventions for the Java Programming Language, recommended by Sun), or **custom defined** by the development team

Extreme Programming (XP)

- **Collective code ownership**

  Collective code ownership means that **everyone is responsible for all the code**; this, in turn, means that **everybody is allowed to change any part of the code**. Pair programming contributes to this practice: by working in different pairs, **all the programmers get to see all the parts of the code**. A major advantage claimed for collective ownership is that it speeds up the development process, because **if an error occurs in the code any programmer may fix it**. By giving every programmer the right to change the code, there is **risk of errors** being introduced by programmers who think they know what they are doing, but do not foresee certain dependencies. Sufficiently **well defined unit tests** address this problem: if unforeseen dependencies create errors, then when unit tests are run, they will show failures

Extreme Programming (XP)

- **Simple design**

  Programmers should take a "**simple is best**" approach to software design. Whenever a new piece of code is written, the author should ask themselves '**is there a simpler way to introduce the same functionality?**'. If the answer is yes, the simpler course should be chosen. Refactoring should also be used, to make complex code simpler

- **System metaphor**

  The system metaphor is a story that everyone - customers, programmers, and managers - can tell about how the system works. It's a **naming concept for classes and methods** that should make it easy for a team member to **guess the functionality of a particular class/method, from its name only**. For each class or operation the functionality is obvious to the entire team

Extreme Programming (XP)

- **Programmer welfare**
  - **Sustainable pace**

    The concept is that programmers or software developers **should not work more than 40 hour weeks**, and if there is overtime one week, that the next week should not include more overtime. Since the development cycles are **short cycles of continuous integration**, and **full development (release) cycles are more frequent**, the projects in XP do not follow the typical crunch time that other projects require (requiring overtime). Also, included in this concept is that **people perform best and most creatively if they are rested**. A key enabler to achieve sustainable pace is **frequent code-merge and always executable & test covered high quality code**.

Extreme Programming (XP)

The **constant refactoring way** of working enforces team members with fresh and alert minds. The **intense collaborative way** of working within the team drives a need to recharge over weekends. Well-tested, continuously integrated, frequently deployed code and environments also **minimize the frequency of unexpected production problems and outages**, and the associated after-hours nights and weekends work that is required