Advanced Programing Practices

Software development models

Predictive and agile models

Software development

- At its core, software development aims at producing code.
- However, if one want to produce large, complex, high quality applications, other activities are to be added, based on additional quality concerns:
  - **Are we building the right software? Do we really know what the client needs?**
    - If not, we may be building the wrong software features, and missing important features.

Software development

- At its core, software development aims at producing code.
- However, if one want to produce large, complex, high quality applications, other activities are to be added, based on additional quality concerns:
  - **Do we have a solid general plan of action for the design of our entire system?**
    - If not, later additions will be requiring major redesigns.

Software development

- At its core, software development aims at producing code.
- However, if one want to produce large, complex, high quality applications, other activities are to be added, based on additional quality concerns:
  - **Is our produced software properly tested before it is delivered?**
    - If not, the resulting software will fail, with disastrous consequences to our client and our reputation.

Software development

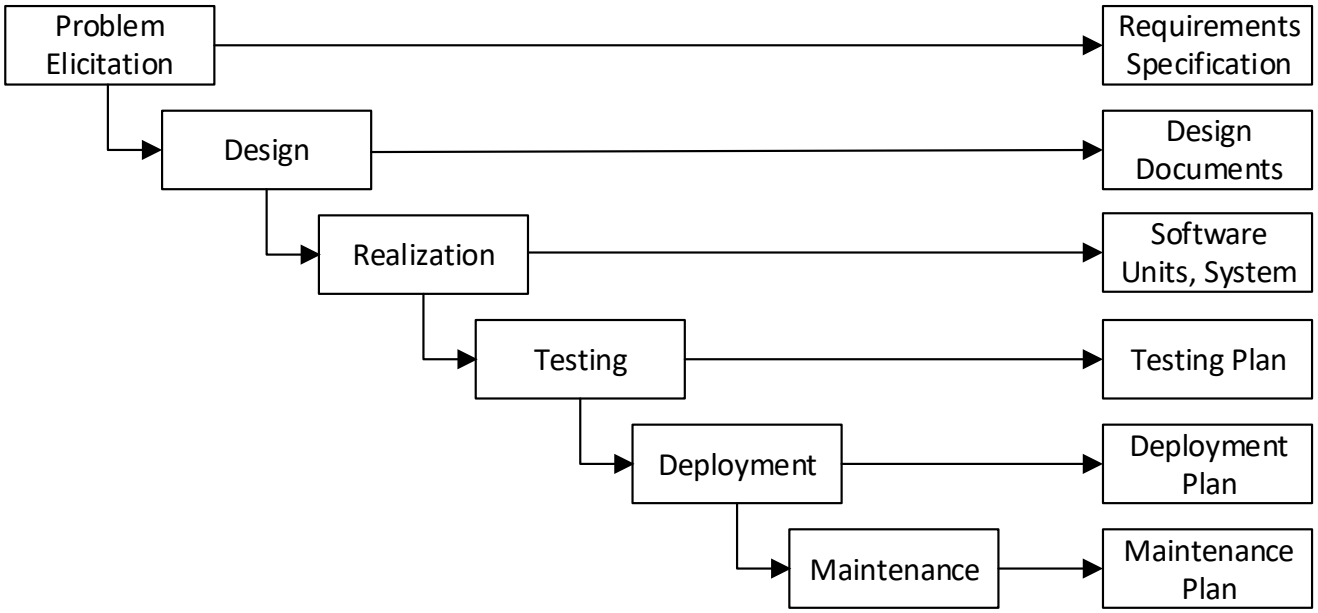
- At its core, software development aims at producing code.
- However, if one want to produce large, complex, high quality applications, other activities are to be added, based on additional quality concerns:
  - **How do we develop the system now so that its structure will sustain further development before deployment, or maintenance after deployment?**
    - If not, our system will become exponentially harder to develop/maintain, until ultimately it needs to be redone from scratch.

Software development

- At its core, software development aims at producing code.
- However, if one want to produce large, complex, high quality applications, other activities are to be added, based on additional quality concerns.
- Software development is a complex activity that requires many more activities and concerns than the core production of software artifacts through coding.

Software development phases: the waterfall model

- One of the earlier software development models was the waterfall model, in which the following phases are followed in order, producing some artifacts :



Software development phases: the waterfall model

- The original waterfall model maintains that one should move to a phase only when its preceding phase is reviewed and verified, and that going back to a previous phase is not possible, or prohibitively costly.
- Developed from traditional Engineering processes, where physical artifacts are produced and can hardly be changed as they are designed, produced and used.
  - However, software is a malleable artifact, i.e. it can be changed at any time during its lifetime.

Software development models

- A software development model is a definition of a group of related precepts, tasks, or artifacts, that are deemed necessary for the production of software.
- There are numerous examples of software development models, who emphasize different important factors and methods to take into consideration while developing software.

Software development models

- A software development model:
  - **Prototyping**: emphasizes the early development of prototype software to elicit the problem statement and develop early solutions to get feedback.
  - **Iterative and incremental development**: emphasizes the structured use of iterations during software development to bring focus on a few development issues at a time.
  - **Spiral development**: emphasizes on risks associated with a particular problem/solution and to minimize risks.

Software development models

- A software development model:
  - **Rapid application development**: emphasizes on productivity of software artifacts rather than the strict following of an elaborated process.
  - **Extreme programming**: emphasizes on precepts to be followed in order to achieve productivity while controlling potentially chaotic aspects of software development.

Predictive vs. adaptive models

- Software development models can be categorized as either **predictive** or **adaptive**:
  - **Predictive model**:
    - Based on the notion that all activities involved in software development can be predicted and documented along the way, and that further development is based on the information accumulated in previous phases of the development.
    - Such models tend to be descriptive models, i.e. to define all the roles, activities, and artifacts involved in a clearly defined process.
    - Predictive models focus on being able to plan the future in detail.

Predictive vs. adaptive models

- Software development models can be categorized as either **predictive** or **adaptive**:
  - **Predictive model:**
    - A predictive team can report exactly what features and tasks are planned for the entire length of the development process.
    - Predictive teams have difficulty changing direction. The plan is typically optimized for the original destination and changing direction can cause completed work to be thrown away and done over differently.
    - Predictive teams will often institute a *change control board* to ensure that only the most valuable changes are considered.

Predictive vs. adaptive models

- **Adaptive model**:
- Based on the notion that software development is characterized by changing information as the development proceeds, and thus that a software development model should be made to cope with change.
- Such models tend to be prescriptive models, i.e. to define a set of precepts to be followed, without an exact definition of a process.
  - Adaptive models focus on being able to adapt quickly to changing realities.
  - When the needs of a project change, an adaptive team changes with it.

Predictive vs. adaptive models

- **Adaptive model**:
  - An adaptive team will have difficulty describing exactly what will happen in the future.
  - The further away a date is, the more vague an adaptive method will be about what will happen on that date.
  - An adaptive team can report exactly what tasks are being done next week, but only which features are planned for next month.
  - When asked about a release six months from now, an adaptive team may only be able to report the mission statement for the release.

Predictive software development models

Predictive models: software development process

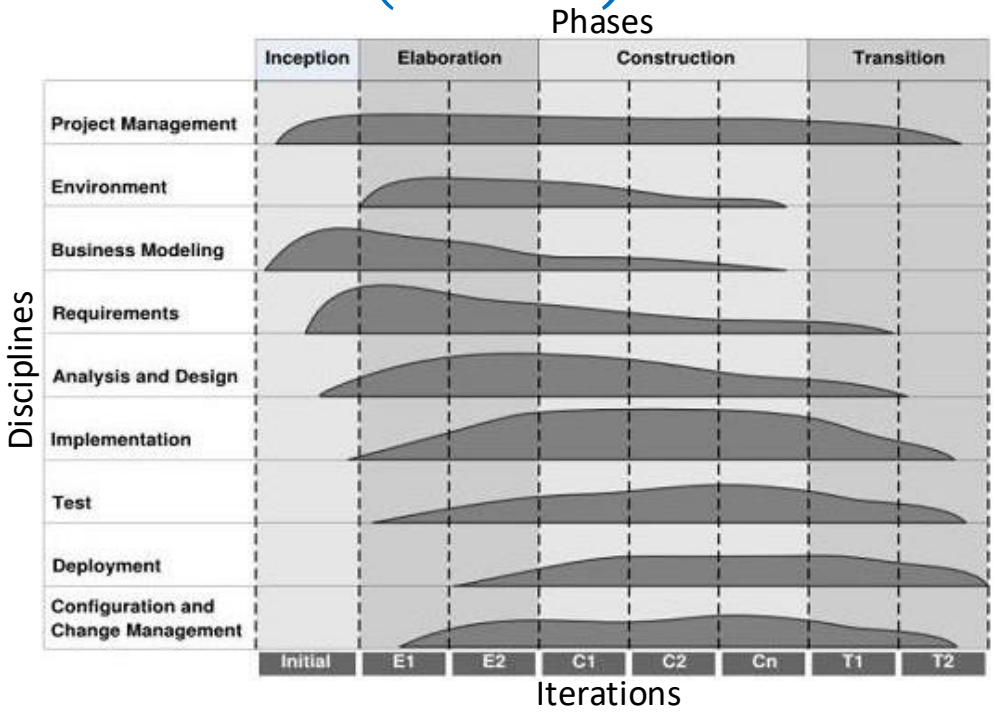
- Generally speaking, a software development process is a formally defined software development process that defines in details the who, what and how of everything that needs to be done in order to produce software.

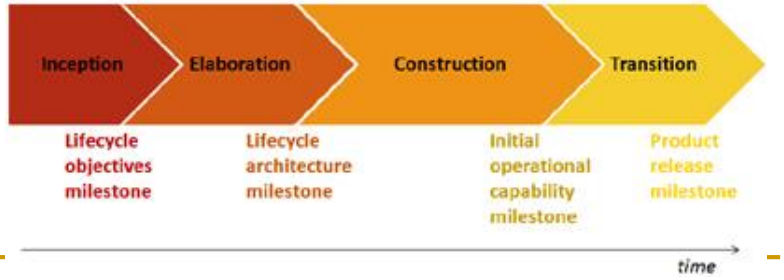
Predictive models: software development process

- A software development process defines the following entities that all play a role in the development of software:
  - **Actor**: defines a set of skills and responsibilities that are necessary for the achievement of tasks and the production of artifacts in the process.
  - **Artifact**: defines a product resulting from the achievement of a task, which is then used as input for further tasks in the process.
  - **Task**: defines a unit of work that aims at producing one or more artifacts, using certain tools and techniques.

Software development process example: Rational Unified Process (RUP)

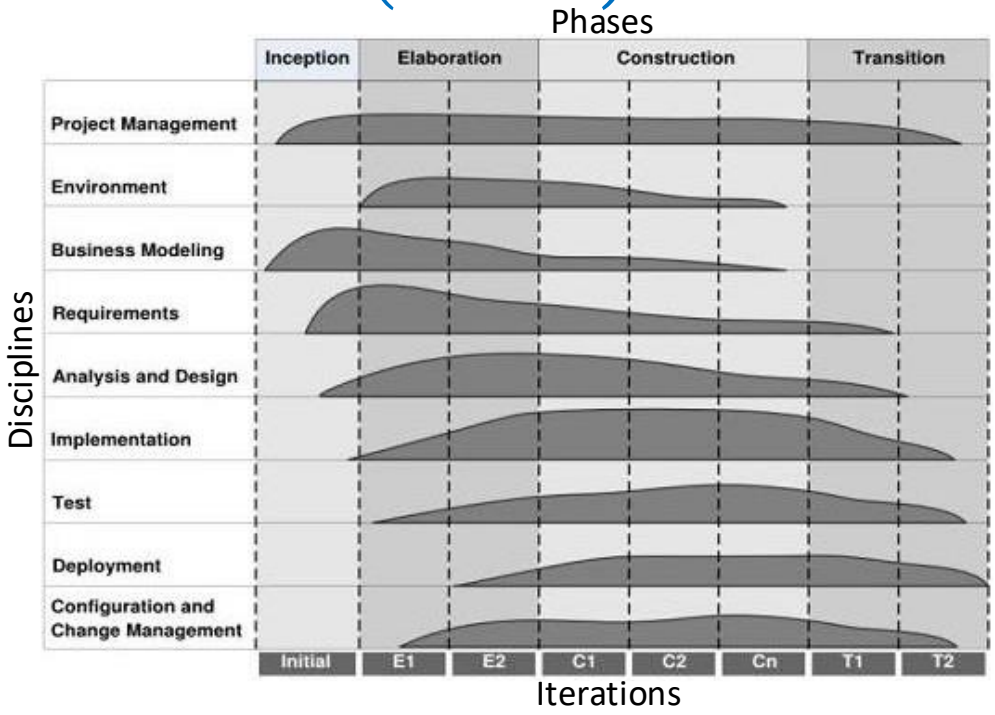
- A good example of a software development process is IBM's Rational Unified Process (RUP).

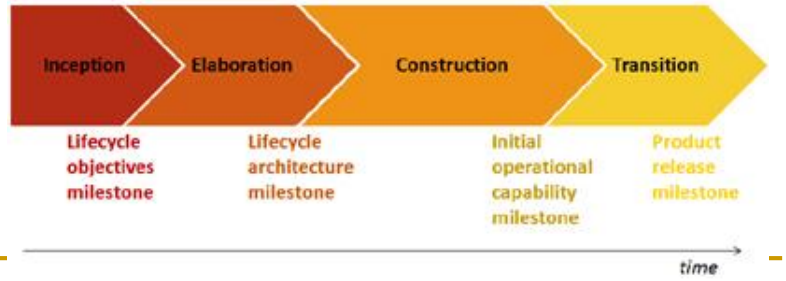




Software development process example: Rational Unified Process (RUP)

- Process that defines:
  - **Disciplines**: major areas of concern in software development
  - **Phases**: plan of action for each discipline, ranging from abstract thinking to concrete development to deployment.
  - **Iterations**: any number of iteration is allowed in each phase in order to reach for the set goals of the phase.





Software development process example: Rational Unified Process (RUP)

- The RUP uses the notion of iterative development
- **Iterative development** is a design methodology based on a cyclic process of prototyping, testing, analyzing, and refining a product.
  - Based on the results of testing the most recent iteration of a design, changes and refinements are made.
  - This process is intended to ultimately improve the quality and functionality of a design.

Software development process example: Rational Unified Process (RUP)

- The RUP uses the notion of iterative development
- **Iterative development** is a design methodology based on a cyclic process of prototyping, testing, analyzing, and refining a product.
  - In iterative design, interaction with the designed system is used as a form of research for informing and evolving a project, as successive versions, or iterations of a design are implemented.

Adaptive software development models

Adaptive software development models: The Agile Manifesto

- Often also called “agile” methods.
- The “Agile Manifesto” (2001) was a statement against predictive methods.

Adaptive software development models: The Agile Manifesto

- It proposed the following principles that are more realistic than what can be achieved by predictive methods in many software development projects:
  - Customer satisfaction by rapid delivery of useful software
  - Welcome changing requirements, even late in development
  - Working software is delivered frequently (weeks rather than months)
  - Close, daily cooperation between business people and developers
  - Projects are built around motivated individuals, who should be trusted
  - Face-to-face conversation is the best form of communication (co-location)
  - Working software is the principal measure of progress

Adaptive software development models: The Agile Manifesto

- It proposed the following principles that are more realistic than what can be achieved by predictive methods in many software development projects:
  - Sustainable development, i.e. able to maintain a constant pace
  - Continuous attention to technical excellence and good design
  - Simplicity—the art of maximizing the amount of work done—is essential
  - Self-organizing teams
  - Regular adaptation to changing circumstances

Adaptive software development models: Concepts

- Adaptive methods assume that software development is inherently about managing change, assuming that both the problem and the solution change during development:
  - The problem statement is refined and changes as the system is developed as the client sees the solution being developed.
  - The details of the designed solution are constantly changing during development.

Adaptive software development models: Concepts

- Most adaptive methods attempt to minimize risks and manage changes by developing software in short time-boxes, called **builds**, which typically last one to four weeks.
- Each build is like a miniature software project of its own, and includes all the tasks necessary to release the mini-increment of new functionality: planning, requirements analysis, design, coding, testing, and documentation.

Adaptive software development models: Concepts

- Capable of releasing new software at the end of every build.
- At the end of each build, the team re-evaluates project priorities.

Adaptive software development models: Concepts

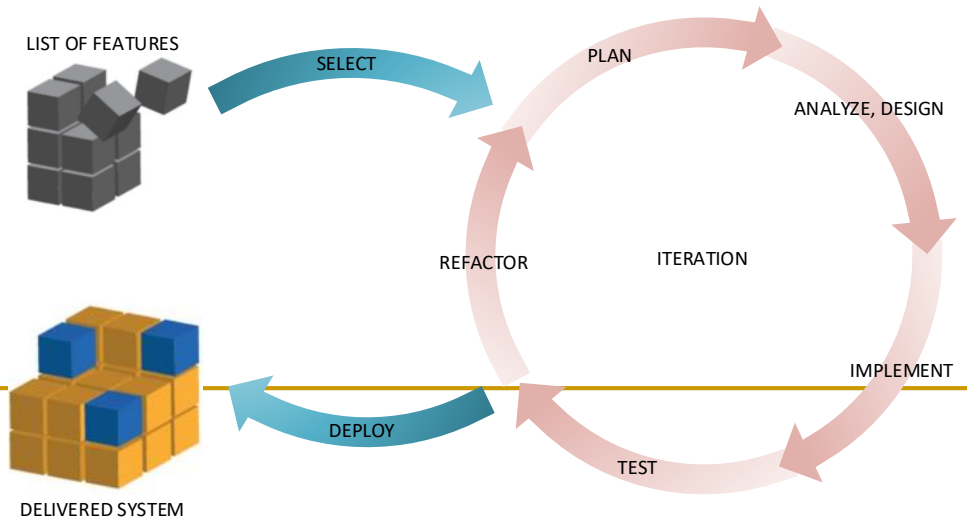
- Adaptive methods emphasize real-time communication, preferably face-to-face, over written documents, as documents accumulate information that is costly to write, verify and change.
- Adaptive methods emphasize working software as the primary measure of progress.

Adaptive software development models: Concepts

- Adaptive methods produce very little written documentation relative to other methods.
- The only artifacts being produced are directly related to the efficient and sustainable production of implementation code.

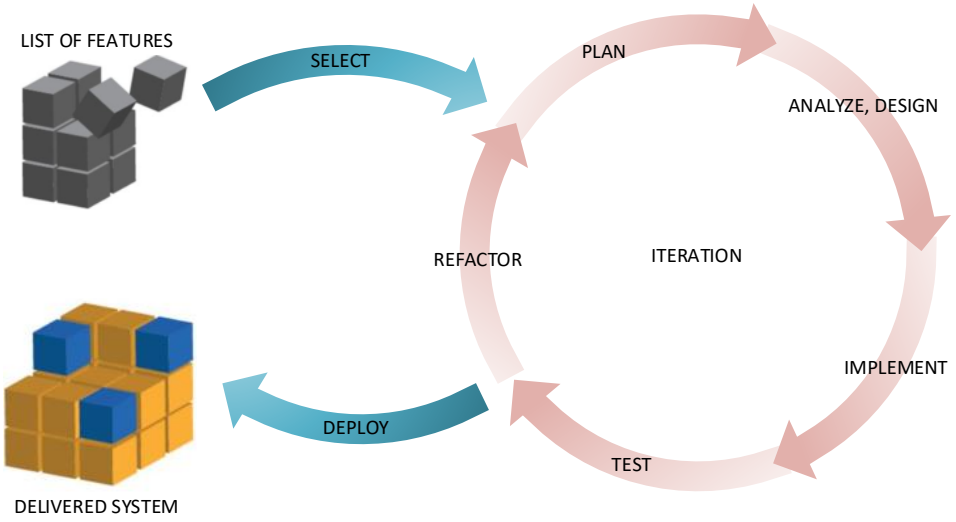
Adaptive software development models: Incremental development

- Most adaptive methods develop software following an incremental model, where the software is produced in a series of “builds” that aim at the production of a solution for a small portion of the problem.



Adaptive software development models: Incremental development

- **Incremental development** is a method of software development where the software is incrementally designed, implemented and tested until the product is finished.
  - During each increment, a set of features is selected for development, which are then analyzed, designed, implemented, tested, and deployed.
  - Each increment builds upon the accumulated system implementation.



Adaptive software development models: Coping with change

- One of the main advantages of the incremental models is their ability to **cope with change** during the development of the system.
- Predictive models rely on careful review of artifacts to avoid errors. Once a phase has been completed, there is limited provision for stepping back to fix/add something uncovered later.

Adaptive software development models: Coping with change

- It is difficult to verify artifacts precisely and this is a weakness of the predictive models.
- As an example, consider an **error in the requirements**:
  - With the waterfall model, the error may not be noticed until acceptance testing, when it is probably too late to correct it.
  - The error may be a requirements error, but it is very tedious to verify requirements statements before they become operational, especially when buried in hundreds of other requirements statements.
  - The real problem of finding a requirements error at the end of the production phase is that a change in one requirement very often induces a "**ripple effect**" of changes in other requirements and to other following artifacts that are based on it (e.g design, code, tests, etc).

Adaptive software development models: Coping with change

- Thus, uncovering such a mistake toward the end of the production is likely to require many other changes.
  - The uncovering of many of such mistakes at the end of the production leads to a dramatic situation that may put the whole project in jeopardy.
- On the other hand, in the incremental model, there is a good chance that a requirements error will be recognized as soon as the corresponding software is incorporated into the system.
- As software is developed then validated in short time boxes and for a reduced number of implemented features, errors uncovered are likely to have **lesser magnitude** in the ripple effect of changes that they induce.

Adaptive software development models: Distribution of feedback

- One of the main reasons why predictive models are not appropriate in many cases is the accumulation of unstable information at all stages.
  - For example, a list of 500 requirements is extremely likely to change, no matter how confident is the client on the quality of these requirements at this point.

Adaptive software development models: Distribution of feedback

- Inevitably, the following design and implementation phases will uncover flaws in these requirements, raising the need for the update and re-verification of the requirements and their subsequent artifacts each time a flaw is uncovered.

- A better approach is thus to limit the accumulation of unstable information by concentrating on the definition, implementation and validation of only a subset of the requirements at a time.

Adaptive software development models: Distribution of feedback

- Such an approach has the benefit of **distributing the feedback** on the quality of the accumulated information.
- In the Waterfall model, most of the relevant feedback is received at the end of the development cycle, where the programming and testing are concentrated.
  - Such a model is evidently likely to lead to failure in later stages.
- By distributing the development and validation efforts throughout the development cycle, incremental models achieve distribution of feedback, thus increasing the **sustainability of further development**.

Adaptive software development models: Advantages

- Delivers an operational quality product at each stage, but one that satisfies only a subset of the clients requirements.
- A relatively small number of developers may be used.
- From the delivery of the first build, the client is able to perform useful work, providing early return on investment (ROI), an important economic factor.
- Reduces the traumatic effect of imposing a completely new product on the client organization by providing a gradual introduction.
- There is a working system at all times.

Adaptive software development models: Advantages

- Clients can see the system and provide feedback.
- Progress is visible, rather than being buried in documents.
- Breaks down the problem into sub-problems, dealing with reduced complexity, and reducing the ripple effect of changes by reducing the scope to only a part of the problem at a time.
- Distributes feedback throughout the whole development cycle, leading to more stable artifacts and sustainable development and maintenance.

Adaptive software development models: Disadvantages

- Each additional build has somehow to be incorporated into the existing structure without degrading the quality of what has been build to date.
- Addition of succeeding builds must be easy and straightforward.
- The more the succeeding builds are the source of unexpected problems, the more the existing structure has to be reorganized, leading to inefficiency and degrading internal quality and degrading maintainability.

Adaptive software development models: Disadvantages

- The incremental models can easily degenerate into the build and fix approach.
- Design errors become part of the system and are hard to remove.
- Clients see possibilities and want to change requirements.

Adaptive software development models: Dangers and Solutions

- **Planning**
  - The main danger of using incremental models is to proceed too much in an ad-hoc manner.
  - Initially determining a global plan of action is of prime importance to ensure the success of use of incremental models.

Adaptive software development models: Dangers and Solutions

- **Planning**
  - The early stages of development must include a preliminary analysis phase that determines the scope of the project, tries to determine the highest risks in the project, define a more or less complete list of important features and constraints, in order to establish a **build plan**, i.e. a plan determining the nature of each build, and in what order the features are implemented.
  - Such a plan should be made in order to foresee upcoming issues in future builds, and develop the current build in light of these issues and make their eventual integration easier.

Adaptive software development models: Dangers and Solutions

- **Structural quality control**
  - The incremental model, like the build-and-fix model, is likely to result to the gradual degrading of internal structural quality of the software.
  - In order to minimize the potentially harmful effect of this on the project, certain quality control mechanisms have to be implemented, such as refactoring. Refactoring is about increasing the quality of the internal structure of the software without affecting its external behavior.

Adaptive software development models: Dangers and Solutions

- **Structural quality control**
  - The net effect of a refactoring operation is to make the software more easy to understand and change, thus easing the implementation of the future builds, i.e. to achieve sustainability of development.
  - How often a refactoring operation needs to be done depends on the current quality degradation of the software.

Adaptive software development models: Dangers and Solutions

- **Structural quality control**
  - Note that planning also has a similar effect by enabling to foresee further necessary changes and developing more flexible solutions in light of the knowledge of what needs to be done in the future.

Adaptive software development models: Dangers and Solutions

- **Architectural baseline**
  - One of the reasons for the degradation of internal structural quality of the system through increments is often associated with a lack of a well-defined overall architectural design.
  - Predictive methods advocate the early definition of the architecture of the system, or early identification and design of the system core.

Adaptive software development models: Dangers and Solutions

- **Architectural baseline**
  - Such a practice has the effect of easing the grafting of new parts on the system throughout increments, and minimizing the magnitude of changes to be applied upon grafting of new parts of the builds.
  - Achieving an architectural design is advisable when writing a project plan. The architecture can also help building a clear plan that developers can relate to.

Adaptive software development models: Dangers and Solutions

- **Architectural baseline**
  - Achieving an architectural design will help control the structural quality of the system by providing a framework for the entire application helping the developers to see the big picture of the system, as they are working on individual parts during de development of the different builds.
  - Also, refactoring operations normally have a result of conforming, or further defining or refining the architecture of the system.

Adaptive software development models: Dangers and Solutions

- **Parallel builds**
  - Various builds could be performed simultaneously by different teams.
    - For example, after the coding phase of build one is started, another team is already starting with the design the second build.
  - The risk is that the resulting builds will not fit together. Each build inevitably has some intersection with other builds.

Adaptive software development models: Dangers and Solutions

- **Parallel builds**
  - Good coordination and communication is important to make sure that teams that have intersecting builds are agreeing on the nature and implementation of their common intersection.
  - The more builds are done concurrently, the more this problem is growing exponentially.
  - Also, larger number of software developers is necessary compared to linear incremental development.

Adaptive software development models: Applicability

- Adaptive development has been widely documented as working well for small (<10 developers) co-located teams.
- Adaptive development is particularly indicated for teams facing unpredictable or rapidly changing requirements.

Adaptive software development models: Applicability

- Adaptive development is less applicable in the following scenarios:
  - Large scale development efforts (>20 developers)
  - Distributed development efforts (non-co-located teams)
  - Mission- and life-critical efforts
  - Command-and-control company cultures
  - Low requirements change
  - Junior developers

Adaptive software development models: Applicability

- Agile home ground:
  - Low criticality
  - Senior developers
  - High requirements change
  - Small number of developers
  - Culture that thrives on chaos

References

References

- Craig Larman and Victor Basili. *Iterative and Incremental Development: A Brief History*. Computer 36 (6): 47–56. June 2003. doi:10.1109/MC.2003.1204375.
- Boehm, B. and Turner, R., *Balancing Agility and Discipline: A Guide for the Perplexed,* Addison-Wesley, Boston. 2004. ISBN-13: 978-0321186126
- Beck, et. al., *Manifesto for Agile Software Development*. http://agilemanifesto.org/
- Kenneth S. Rubin. *Essential Scrum: A Practical Guide to the Most Popular Agile Process.* Addison-Wesley Professional, 2012. ISBN-13: 978-0137043293
- Alistair Cockburn. *Agile Software Development.* Addison-Wesley, 2001. ISBN-13: 978-0201699692
- Alistair Cockburn. *Agile Software Development: The Cooperative Game*. Addison-Wesley, 2006. ISBN-13: 978-0321482754

Extreme Programming

Extreme programming

- Extreme Programming (XP) is a method or approach to software engineering and a precursor of several agile software development methodologies.
- Formulated by Kent Beck, Ward Cunningham, and Ron Jeffries. Kent Beck wrote the first book on the topic, Extreme programming explained: Embrace change, published in 1999.

Extreme programming

- The twelve key features of XP, outlined below in Beck's words, are:

  - **The Planning Game:** Quickly determine the scope of the next release by combining business priorities and technical estimates. As reality overtakes, update the plan.
  - **Small Releases:** Put a simple system into production quickly, then release new versions on a very short cycle.
  - **System Metaphor:** Guide all development with a simple shared story of how the whole system works.

Extreme programming

- The twelve key features of XP, outlined below in Beck's words, are:

  - **Simple Design:** The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered.
  - **Testing:** Programmers continually write unit tests, which must run flawlessly for development to continue. Customers write user stories for desired features that the system must demonstrate to expose.
  - **Refactoring:** Programmers restructure the system without changing its behaviour to remove duplication, improve communication, simplify, or add flexibility.

Extreme programming

- The twelve key features of XP, outlined below in Beck's words, are:

  - **Pair Programming:** All production code is written with two programmers at one workstation.
  - **Collective Ownership:** Anyone can change code anywhere in the system at any time.
  - **Continuous Integration:** Integrate and build the system many times a day, every time a task is completed.
  - **Sustainable Pace:** Work no more than 40 hours a week as a rule. Never allow overtime for the second week in a row.
  - **On-site Customer:** Include a real, live customer on the team, available full-time to answer questions.

Extreme programming

- The twelve key features of XP, outlined below in Beck's words, are:
  - **Coding Standards:** Programmers write all code in accordance with rules emphasizing communication throughout the code.
- These ideas are not new. They have been tried before and there have been many reports of failure. The point of XP is that, taken together, these techniques do constitute a workable methodology.

Key features of Extreme Programming

The Planning Game

- Description
  - The long term build plan determines the general goals of each successive builds. It is not detailed, and it can be changed as required. It can be changed by either the customer or developers, depending on the situation.
  - The short term detailed plan determines what exactly needs to be done in next few days. It is re-evaluated every day as development evolves.
- Potential drawbacks
  - A rough plan is not a sufficient basis for detailed development. Constantly updating the plan may be inefficient and may confuse customers and developers.

The Planning Game

- Why it works in XP
  - The build plan is sufficient to give the clients a vision of what you will achieve.
  - Small releases enables the team to concentrate on immediate goals in the detailed plan. The build plan determines the goals of each successive small release.
- Project
  - Make a build plan that determines what will be the goals of each successive builds.
  - As each build starts, make a detailed plan for the current build, assign tasks.
  - Meet regularly and update the plans according to the latest developments.
  - Make the updated plans available to everybody in the team.

Small Releases

- Description
  - A release is a working version of the software. Between releases, the software may be in an inconsistent state. “Small” releases mean obtaining a working version every week, or every month, rather than every six months, or every year.
- Potential drawbacks
  - Small releases mean that time is spent on getting the releases to work perfectly.
  - May not be necessary if the client does not need/want the intermediate builds to be delivered and used operationally.

Small Releases

- Why it works in XP
  - Planning focuses immediate attention on the most important parts of the system, so even small releases are useful to customers.
  - With continuous integration, assembling a release does not take much effort.
  - Frequent testing reduces the defect rate and release testing time.
  - The design is simple, thus easier/faster to implement but may be elaborated later.
- Project
  - Each predefined build is a small release.

System Metaphor

- Description
  - The system metaphor is a “story” about the system. It provides a framework for discussing the system and deciding whether features are appropriate. A well-known simple example of a metaphor is the Xerox “desktop” metaphor for user-interface design. Another is the “spreadsheet” metaphor for accounting. Games are their own metaphor: knowledge of the game helps to define the program.
- Potential drawbacks
  - A metaphor may not have enough detail. It might be misleading or even become wrong if it is not updated.

System Metaphor

- Why it works in XP
  - Small releases provide quick feedback from real code to support the metaphor.
  - Clients know the metaphor and can use it as a basis for discussion.
  - Frequent refactoring are made within the practical implications of the metaphor.
- Project
  - The initial project description is the system metaphor.

Simple Design

- Description
  - A simple design is an outline for a small portion of the solution to be implemented.
  - Has the smallest number of features that meet the requirements of current phase and does not incorporate solutions to the requirements of the upcoming phases.
  - Generally, overly complicated designs end up having unused features that become hindrance.
- Potential drawbacks
  - A simple design may have faults and omissions.
  - Implementing an overly simple design might turn out to be too simple eventually.
  - Components with simple designs might not integrate correctly into the system.

Simple Design

- Why it works in XP
  - Refactoring allows you to correct design errors and omissions.
  - The metaphor helps to keep the design process on track with the overall picture.
  - Pair programming helps to avoid silly mistakes and to anticipate design problems.
- Project
  - Every time a solution is proposed, it should be debated as to whether it is the simplest solution that can meet the required features.
  - Overly complex designs should be avoided as a team principle.

Testing

- Description
  - Write large numbers of simple tests. Provide a fully automated testing process.
  - Unit tests are automated tests that test the functionality of methods.
  - Unit tests are written before the eventual code is coded. This approach stimulates the programmer to think about conditions in which their code could fail. The programmer is finished with a certain piece of code when they cannot come up with any further condition in which the code may fail.

Testing

- Potential drawbacks
  - Writing tests is time consuming. Time spent on testing must be justified.
  - In larger projects/companies, programmers don't write tests — testing teams do.
- Why it works in XP
  - Simple design implies that the tests should be simple too.
  - With pair programming, one partner can think of tests while the other is coding. Seeing tests work is good for morale.
  - Clients like seeing tests working.
  - There are many tests and most of them are run automatically.
- Project
  - Unit tests must be delivered with each build.

Refactoring

- Description
  - Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.
- Potential drawbacks
  - Refactoring takes time, may be hard to control, and may be error-prone.

Refactoring

- Why it works in XP
  - Collective ownership makes refactoring easier as anybody can change any code.
  - Coding standards facilitates the task of refactoring by making code easier to understand.
  - Pair programming makes refactoring less risky and based more on consensus.
  - You have a set of tests that you can run at any time during the refactoring process.
  - Continuous integration gives rapid feedback about refactoring problems.
- Project
  - After each build is delivered, have a meeting and decide what parts need to be cleaned up before development continues on the next build.

Pair Programming

- Description
  - Pair Programming means that all code is produced by two people programming on one task as a team. One programmer has control over the programming workstation and is thinking mostly about the coding in detail.
  - The other programmer is more focused on the big picture, continually reviewing the code that is being produced, as well as researching solutions.
  - The pairs are not fixed: it's recommended that programmers try to mix as much as possible, so that everybody can become familiar with the whole system.

Pair Programming

- Potential drawbacks
  - Pair programming if not done properly may be inefficient.
- Why it works in XP
  - Coding standards avoids trivial arguments.
  - Simple design, refactoring, and writing tests together helps to avoid misunderstanding and make decisions based on consensus.
  - Both members of the pair are familiar with the metaphor.
  - If one partner knows a lot more than the other, the second person learns quickly.
- Project
  - Work in pairs and make sure both individuals know their responsibilities.

Collective Ownership

- Description
  - Anyone can make changes to any part of the code.
  - This contrasts with traditional processes, in which each piece of code was owned by an individual or a small team who has complete control over it and access to it.
  - Speeds up the development process, because if an error occurs in the code, any programmer may fix it rather than wait for it to be fixed and/or have arguments about how/why to fix the code.
- Potential drawbacks
  - Problematic if change are applied without caution.

Collective Ownership

- Why it works in XP
  - Continuous integration avoids large scale code breakdowns.
  - Continuously writing and running tests, warns about breakdowns.
  - Pair programmers are less likely to break code than individual programmers.
  - Coding standards avoid trivial arguments.
  - Knowing that other people are reading your code makes you work better.
  - Complex components are simplified as people understand them better.
- Project
  - Setup a software repository and enforce that it is used as frequently as possible.

Continuous Integration

- Description
  - The system is assembled very frequently, perhaps several times a day.
  - Not to be confused with short releases, in which a new version with new features is built and delivered.
  - In order to validate integration, newly integrated system can be compiled and tested.
- Potential drawbacks
  - Each integration can be difficult if different programmers are going in different directions or changing existing code without consulting other programmers.

Continuous Integration

- Why it works in XP
  - Tests are run automatically and quickly, so that errors introduced by integration are detected quickly.
  - Refactoring maintains good structure and reduces the chance of conflicts in integration.
  - Simple designs can be integrated quickly.
- Project
  - Enforce the practice of frequent commits.
  - Enforce that any code committed actually compiles and passes all tests.

Sustainable Pace

- Description
  - Many software companies require large amounts of overtime: programmers work late in the evening and during weekends.
  - They get over-tired, make silly mistakes, get irritable, and waste time in petty arguments, and eventually are more likely to fall sick or go away.
  - This XP policy ensures that no one works too hard.
- Potential drawbacks
  - 40 hours a week is often not enough to obtain the productivity required for competitive software development.

Sustainable Pace

- Why it works in XP
  - Good planning increases the value per hour of the work performed; there is less wasted time.
  - Planning and testing reduces the frequency of unexpected surprises that lead to complex problems to be solved that requires many hours of work.
  - XP as a whole helps the team to work more rapidly and efficiently.
- Project
  - Distribute work evenly. Do not wait until the last few days to work day and night.
  - Implement practices that ensure efficient usage of time.

On-site Customer

- Description
  - A representative of the client's company works at the developer's site all the time.
  - The client is available all the time to consult with developers and monitor the development of the software.
- Potential drawbacks
  - The representative would be more valuable working at the client's company.

On-site Customer

- Why it works in XP
  - Clients can contribute, e.g. by writing user stories and contributing/commenting on tests.
  - Rapid feedback for programmer questions is valuable.
  - XP focuses on efficiency, which includes efficient of communication with the client.
- Project
  - Discussions about the project during lectures.
  - Contact the instructor any time for clarifications.

Coding Standards

- Description
  - All code written must follow defined conventions for layout, variable names, file structure, documentation, etc. The team agrees to and adopts a group of coding conventions, then ensures that they are followed by everyone.
- Potential drawbacks
  - Programmers can be individualists and refuse to be told how to write their code.
  - Can be overdone and thus be wasteful of time.

Coding Standards

- Why it works in XP
  - Coding standards lead to more understandable code, which is required for pair programming, continuous integration, testing, and productivity in general.
  - Refactoring can be used to enforce conformance to coding standards between builds.
- Project
  - Use a predefined set of coding conventions.
  - Keep focused on a simple, reduced set of conventions.
  - Use a documentation generation software (e.g. Javadoc).

Values of Extreme Programming

Communication

- Building software systems requires, for example:
  - communicating system requirements to the developers of the system
  - communicating the software interfaces (APIs) to fellow developers.
- In formal software development methodologies, this task is accomplished through precise and standard documentation.
- Extreme programming techniques can be viewed as methods for efficiently building and disseminating institutional knowledge among members of a development team without relying on heavy documentation.

Communication

- The goal is to give all developers a shared view of the system which matches the view held by the users of the system. The developers and users should come to understand and use each other's terms and language.
- To this end, extreme programming favors simple design, common metaphors, collaboration of users and programmers, frequent verbal communication, and feedback.

Communication

- During coding, automated code documentation tools (e.g. Doxygen, Javadoc) and coding standards can be used to facilitate communication between developers.
- When discussing the project with the instructor, learn and use correct terminology.

Simplicity

- Extreme programming encourages implementing the simplest solution. Extra functionality can then be added later when it becomes a need.
- 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.

Simplicity

- Coding and designing for uncertain future requirements implies the risk of spending resources on something that might not be needed, while perhaps delaying crucial features that are needed now.
- Often summed up as the "You aren't gonna need it" (YAGNI) approach.

Simplicity

- A simple design with very simple and neat code could be easily understood by most programmers in the team, promoting efficiency.
- When many simple short steps are made, the customer and the developers have more control and feedback over the development process and the system that is being developed.

Feedback

- **Feedback from the system**: by writing unit tests, or running tests during continuous integration, the programmers have direct feedback from the state of the system after implementing new code or changes to existing code.
- **Feedback from the customer**: The functional tests are provided 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 during the delivery of each build so the customer can easily steer the development.

Feedback

- **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.
  - Feedback is closely related to communication and simplicity.
  - Flaws in the system are easily communicated by writing a unit test that proves a certain piece of code will not misbehave in a specific case.

Feedback

- **Feedback from the team**:
  - The direct feedback from the system tells programmers to fix this part.
  - A customer is able to test the system periodically according to the functional requirements, simplified as user stories.

Courage

- Several XP practices require courage.
- Courage to:
  - Change one's habits
  - Admit one's own mistakes or shortcomings
  - Have one's work constantly actively questioned
- One is to always design and code for today and not for tomorrow. This is an effort to avoid getting bogged down in overly complicated design and concentrate on what is required now.

Courage

- 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.
- Continuous integration forces all individuals to confront their own code with the main body of code, which might uncover design flaws or omissions.

Courage

- Pair programming forces individuals to uncover their lack of knowledge or erroneous code to tier peers as they are working in pairs.
- Courage is required when code needs to be thrown away: courage to remove source code that is obsolete, no matter how much effort was used to write it.

Respect

- The respect value includes respect for others as well as self-respect.
- 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 own work by always striving for high quality and seeking for the best design for the solution at hand through refactoring, and to follow coding standards.

Respect

- Adopting good values leads to respect gained from others in the team. Nobody on the team should feel unappreciated or ignored if they embrace the values common to the team.
- This ensures a high level of motivation and encourages loyalty toward the team and toward the goal of the project.
- This value is very dependent upon the other values, and is very much oriented toward people in a team.

Embracing change

- The principle of embracing change is about not working against changes but embracing them.

- For instance, if at one of the iterative meetings it appears that the customer's requirements have changed dramatically, programmers are to embrace this and plan the new requirements for the next iteration.

Embracing change

- If the design of the system shows significant flaws that are hampering further development, its change should be embraced through redesign and refactoring.
- When encountering unit testing failures or integration problems, one should see this as an opportunity to improve the system.

References

References

- Kent Beck. Extreme programming explained: Embrace change, Addison-Wesley, ISBN 0201616416
- Kent Beck and Martin Fowler. Planning Extreme Programming, Addison-Wesley, ISBN 0201710919
- Martin Fowler. Refactoring: Improving the Design of Existing Code, Addison-Wesley, ISBN 0201485672
- Ken Auer and Roy Miller. Extreme Programming Applied: Playing To Win, Addison-Wesley, ISBN 0201616408
- Ron Jeffries, Ann Anderson and Chet Hendrickson. Extreme Programming Installed, Addison-Wesley, ISBN 0201708426
- Kent Beck. Extreme programming explained: Embrace change, Second Edition, Addison-Wesley, ISBN 0321278658
- Kent Beck, Cynthia Andres. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, 2004. ISBN-13: 978-0321278654
- Matt Stephens and Doug Rosenberg. Extreme Programming Refactored: The Case Against XP, Apress, ISBN 1590590961

References

- Beck, Kent. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston. 1999. ISBN-13: 978-0201616415
- Fowler, Martin. *Is Design Dead?* In *Extreme Programming Explained*, G. Succi and M. Marchesi, ed., Addison-Wesley, Boston. 2001. ISBN 0-201-71040-4
- M. Stephens, D. Rosenberg. *Extreme Programming Refactored: The Case Against XP*. Apress L.P., Berkeley, California. 2003.
- McBreen, P. *Questioning Extreme Programming*. Addison-Wesley, Boston. 2003.
- Riehle, Dirk. *A Comparison of the Value Systems of Adaptive Software Development and Extreme Programming: How Methodologies May Learn From Each Other*. Appeared in *Extreme Programming Explained*, G. Succi and M. Marchesi, ed., Addison-Wesley, Boston. 2001. ISBN 0-201-71040-4