P1L2 Life Cycle Models

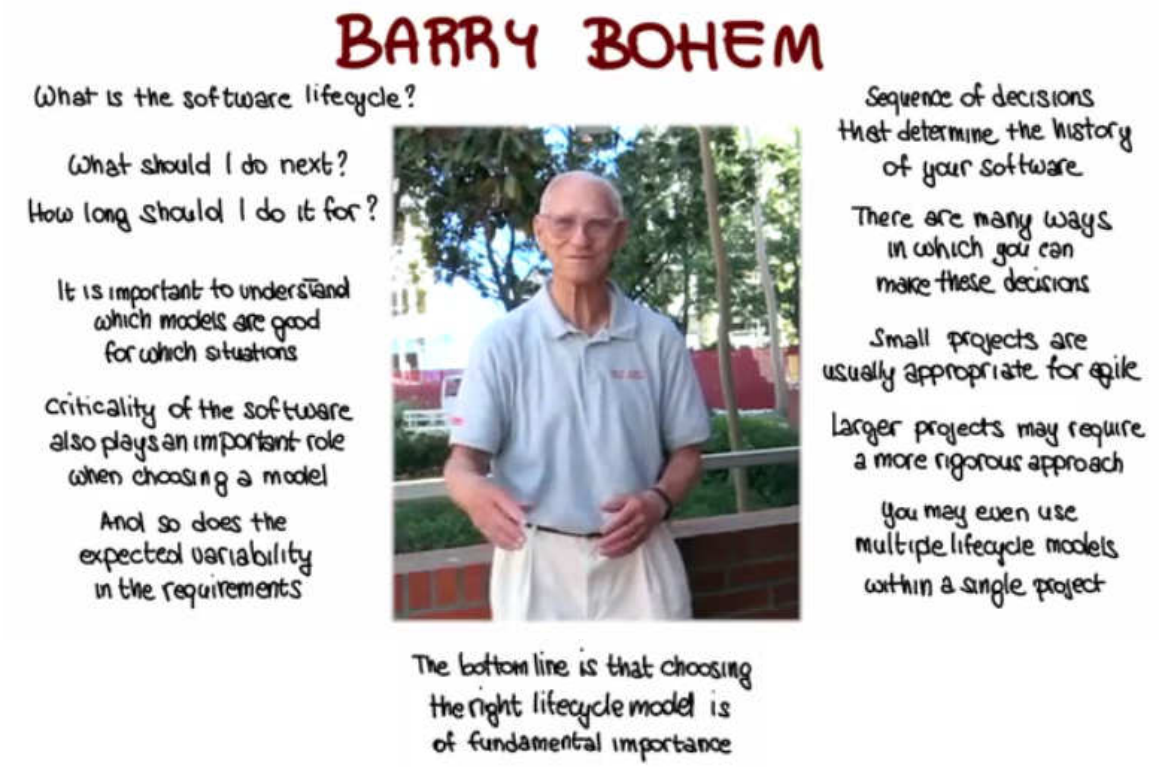
Notes 08/18/2014

Will discuss several traditional software engineering life cycle models.

- *Main advantages and shortcomings.*

Classic mistakes in software engineering

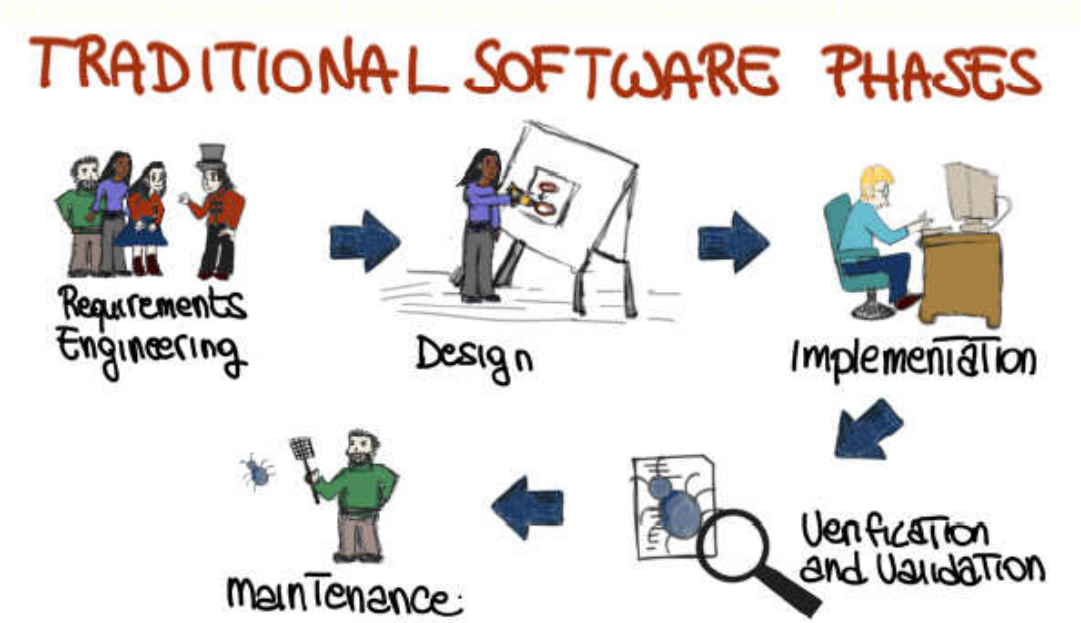
- *Well-known ineffective development practices that when followed, tend to lead to bad results*



Professor Barry Boehm (University of Southern California), one of the fathers of software engineering:

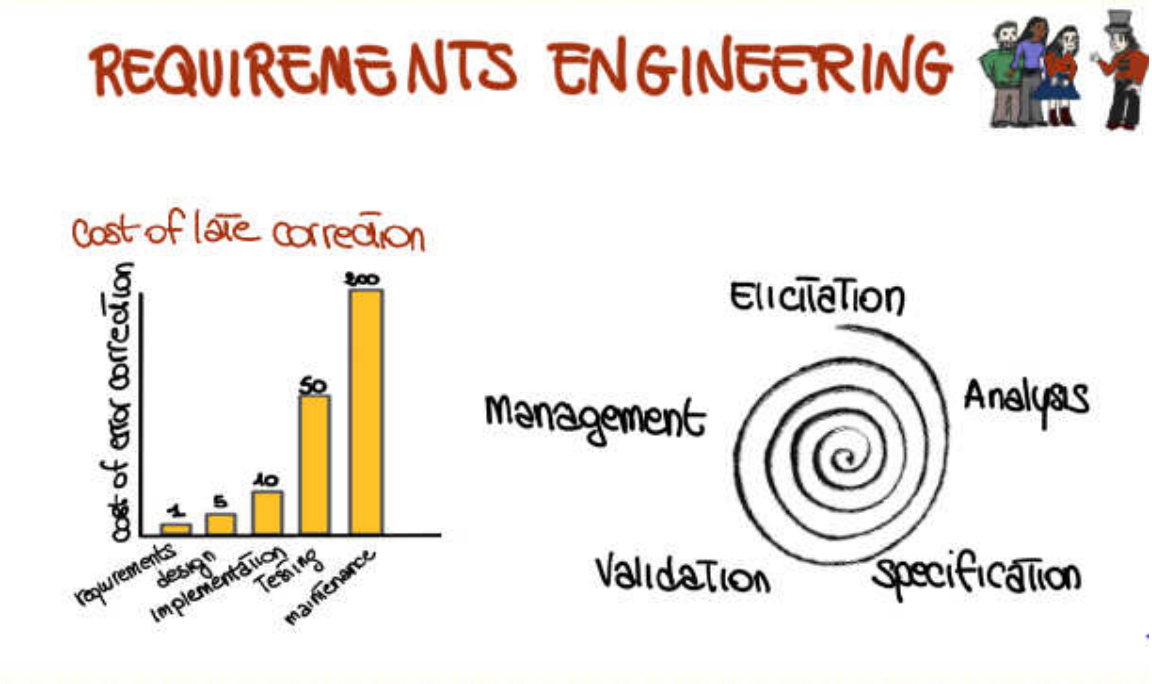
- A software life cycle is a sequence of decisions that you make, and it's fundamentally those decisions that are going to determine the history of the software that you are going to build that other people are going to use.
- Process model is basically answering the question of what do I do next and how long shall I do it for.
    - And again, because there are a lot of different ways you can make that decision
    - Need to figure out which models are good for which particular situations.
- Book: Balancing Agility and Discipline
    - Under what conditions should you use agile methods

    - Under which conditions should you invest more time in analyzing the situation and planning what you're going to do and the like
    - Typically if the project is, is small where it's three to ten people, agile works pretty well. If it's 300 people, then I think we don't want to go that way.
    - If the affect of the defect is loss of comfort or limited funds, then agile is fine, but if it is a loss of life, then you don't.
    - On the other hand if, if you have a situation where you have lot of unpredictable change, you really don't want to spend a lot of time writing plans and lots of documents.
- In some cases you may have a project where you want to do waterfall in some parts and agile in others.
    - These are the kind of things that, that make the choice of life cycle process model very important and very interesting as a subject of concern.



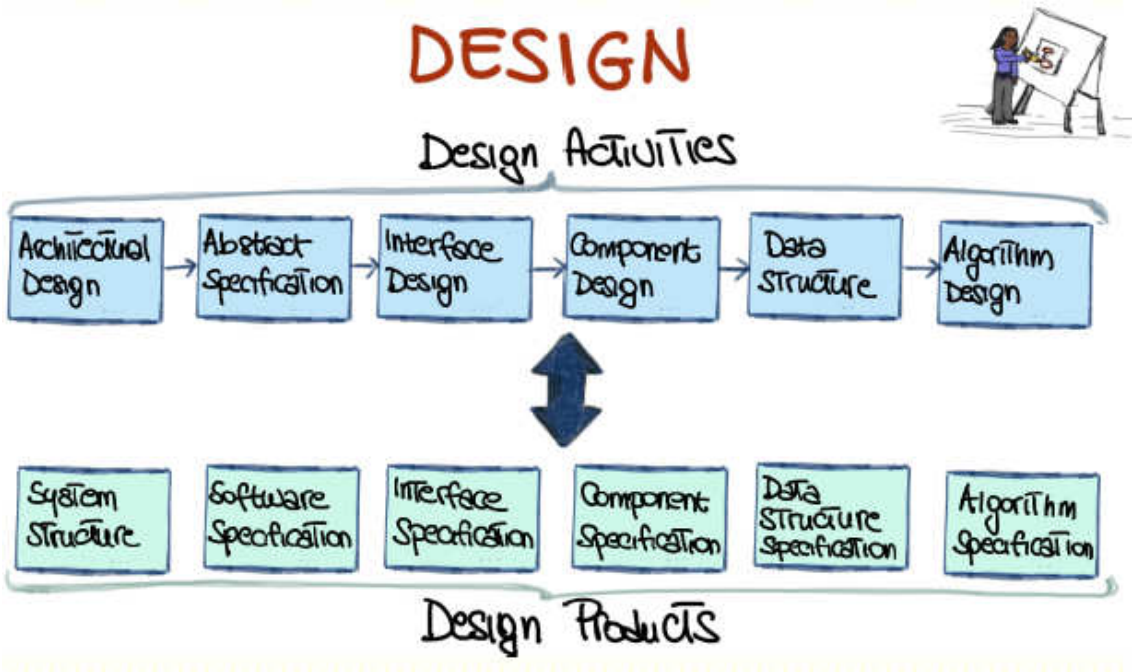
Traditional Software Phases

- Software engineering is an important and critical discipline, concerned with cost effective software development.
- Is based on a systematic approach that uses appropriate tools and techniques, operates under specific development constraints.
- And most importantly, follows a process.
- We start with requirements engineering, followed by design, implementation, verification and validation, and finally maintenance.



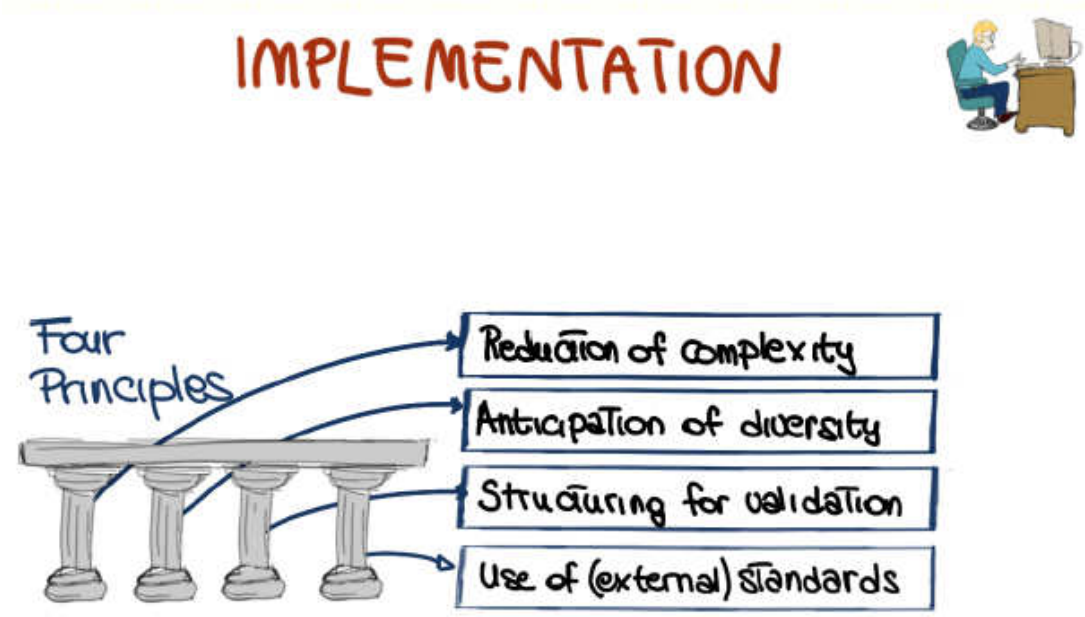
Requirements Engineering

- The field within software engineering that deals with establishing the needs of stakeholders that are to be solved by the software.
- Why is this phase so important?
    - In general, the cost of correcting an error depends on the number of subsequent decisions that are based on it. Therefore, errors made in understanding requirements have the potential for greatest cost because many other design decisions depend on them and many other follow up decisions depend on them.
    - The cost grows dramatically as we go from the requirements phase to the maintenance phase because if we discover a problem later, to correct the error we're left to undo a lot of the decisions that we had made before.
- How can we collect the right requirements? Through a set of steps.
    - Elicitation - the collection of requirements from stake holders and other sources and can be done in a variety of ways
    - Requirement analysis - involves the study and deeper understanding of the collective requirements.
    - Specification of requirements - the collective requirements are suitably represented, organized and saved so that they can be shared
        - There are many ways to do this
    - Validation - make sure that they're complete, consistent, not redundant and so on, so that they've satisfied a set of important properties for requirements.
    - Requirements management - accounts for changes to requirements during the lifetime of the project.
- This is not a linear but an iterative process



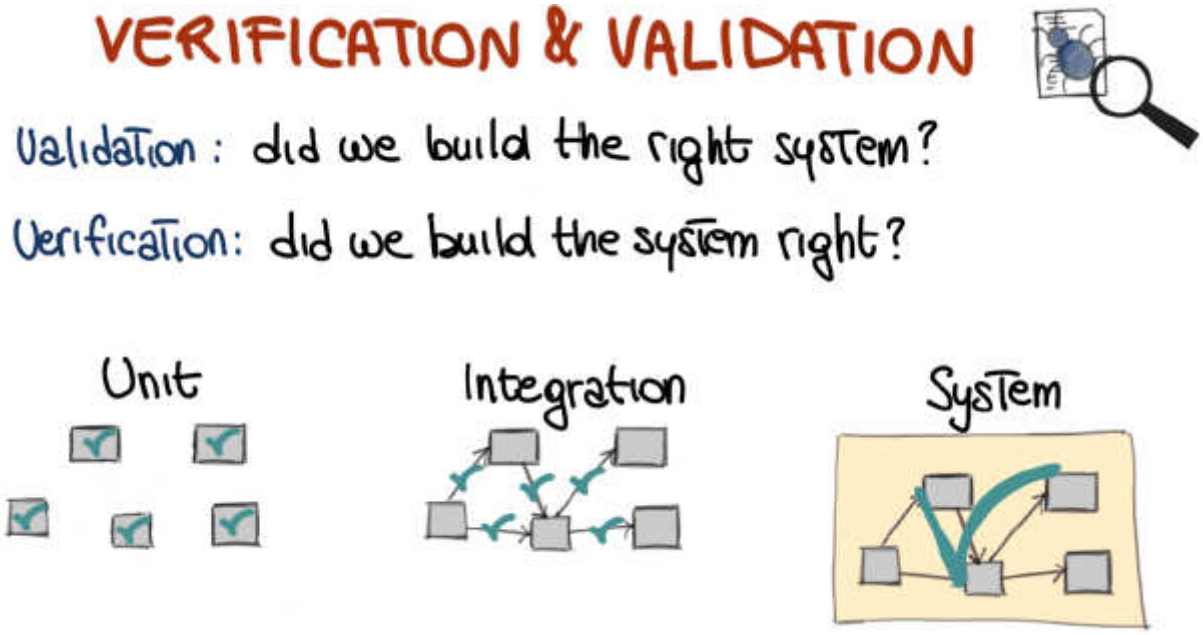
Design

- Phase where software requirements are analyzed in order to produce a description of the internal structure and organization of the system.
- This description will serve as the basis for the construction of the actual system.
- Traditionally, the software design phase consists of a series of design activities, normally consisting of:
    - Architectural design phase
    - Abstract specification
    - Interface design
    - Component design
    - Data structure
    - Algorithm design
- Different books, and different sources, you might find different activities described.
- Core idea/important point is that we go from sort of a high-level view of the system, which is the architectural design, to a low-level view, which is the algorithm design.
- And these activities result in a set of design products, which describe various characteristics of the system.
- They describe the architecture of the system
    - how the system is decomposed and organized into components, the interfaces between these components.
- They also describe these components into a level of details that is suitable for allowing their construction.



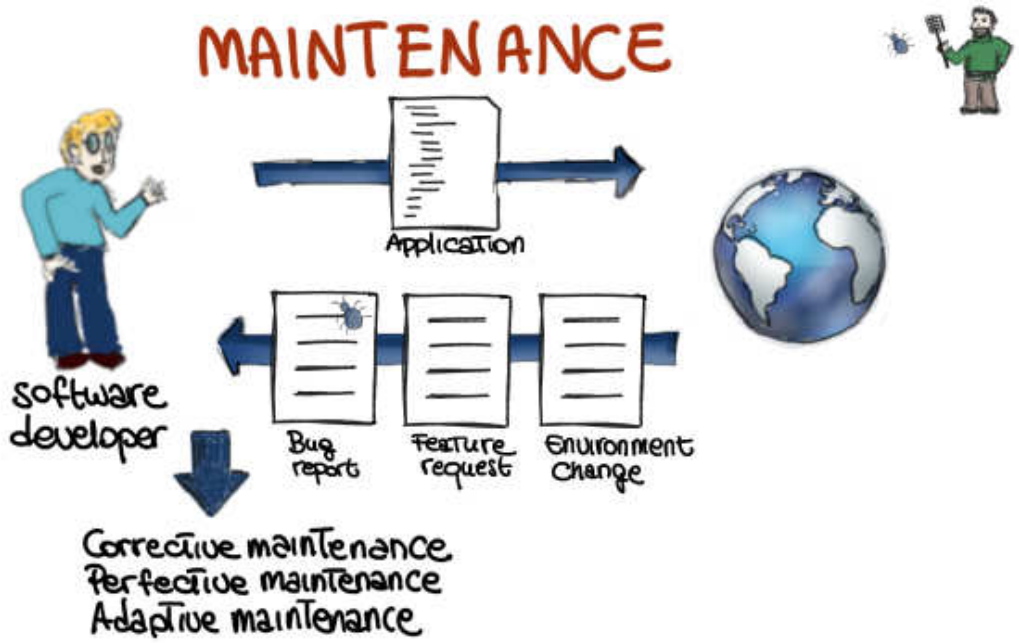
Implementation

- In the implementation phase what we do is basically taking care of realizing the design of the system that we just created and create a natural softer system.
- There are four fundamental principles, four pillars that can affect the way in which software is constructed.
    - Reduction of complexity - aims to build software that is easier to understand and use.
    - Anticipation of diversity - takes into account that software construction might change in various ways over time; the software evolves and often in unexpected ways.
        - Therefore, we have to be able to anticipate some of these changes.
    - Structuring for validation (aka design for testability) - what this means is that we want to build software so that it is easily testable during the subsequent validation and verification activities.
    - Use of standards (internal or external) - important that the software conforms to a set of internal or external standards and this is especially true within specific organizations and or domains
        - Example of internal standards: coding standards within an organization, or naming standards within an organization.
        - Example of external standards: if you are developing some medical software there are some regulations and some standards that you have to adhere to in order for your software to be valid in that domain.



Verification and Validation

- Phase of software development that aims to check that the software system meets its specification and fulfills its intended purpose.
- We can look at verification and validation independently.
    - Validation - answers the question did we build the right system. Did we build the system that the customer wants?
        - That will make the customer happy.
    - Verification - answers did we build the system right?
        - So given a description of the system that is the one that we derived from the customer through the collection of requirements and then design and so on, did we build a system that actually implements the specification that we defined?
        - Many, many ways of doing verification
            - Can be performed at different levels.
            - Unit level - test that the individual units work as a expected.
            - Integration level in which what we test is the interaction between the different units; want to make sure that the different modules talk to each other in the right way
            - System level - test the system as a whole and we want to make sure that all the system, all the different pieces of the system work together in the right way.
                - And this is also the level at which then we will apply validation and some other testing techniques like stress testing or robustness testing and so on.



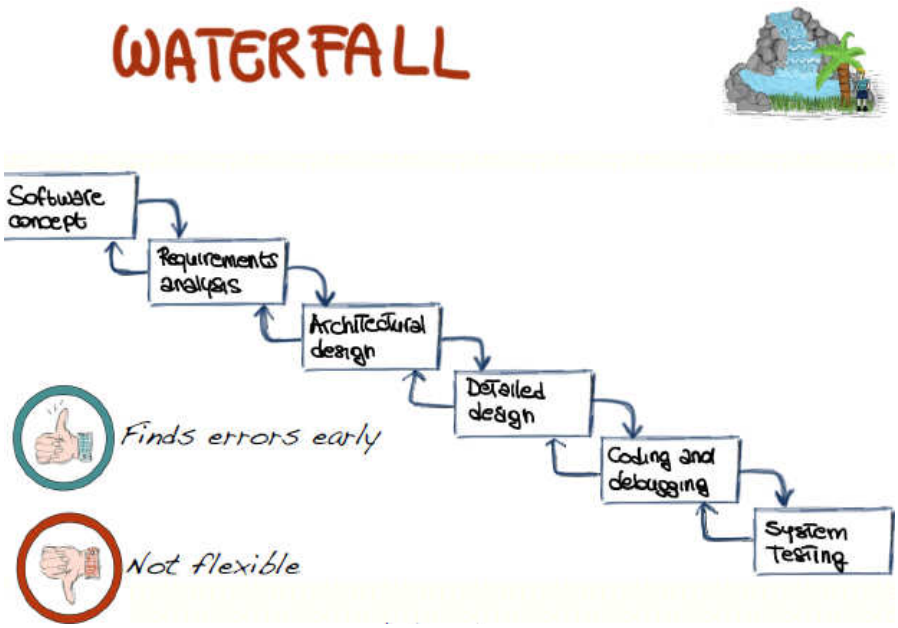
Maintenance

- Software development efforts normally result in the delivery of a software product that satisfies the user requirements.
- Normally our software development organization will release this application to its final users, however, once the software is in operation many things can happen.
    - Examples:
        - Environment might change.
        - Might be new libraries.
        - Might be new systems in which our software has to operate.
        - May be future requests, so the users may find out that they want to do something different with the program that we gave them.
        - One of the most common occurrences: users might find problems with the software and may file bug reports
- These are the reasons why software maintenance is a necessary phase in software development.
- Software maintenance is the activity that sustains the software product as it evolves throughout its life cycle, specifically in response to bug reports, feature requests and environment changes.
- Development organizations perform three kinds of maintenance activities:
    - Corrective maintenance to eliminate problems with the code
    - Perfective maintenance to accommodate feature request, and in some cases just to improve the software, for example, to make it more efficient,
    - Adaptive maintenance, to take care of the environment changes.

- And after this activities have been performed, the software developer will produce a new version of the application, will release it and the cycle will continue throughout the lifetime of the software.
- That's why maintenance is a fundamental activity and a very expensive one.
- One of the reasons why maintenance is expensive is regression testing.
    - Every time you modify your application you have to regression test the application,
        - Regression testing - the activity of retesting software after it has been modified to make sure that the changes you perform to the software work as expected, and that your changes did not introduce any unforeseen effect.
        - Regression testing targets and tries to eliminate regression errors before the new version of the software is released into the world.

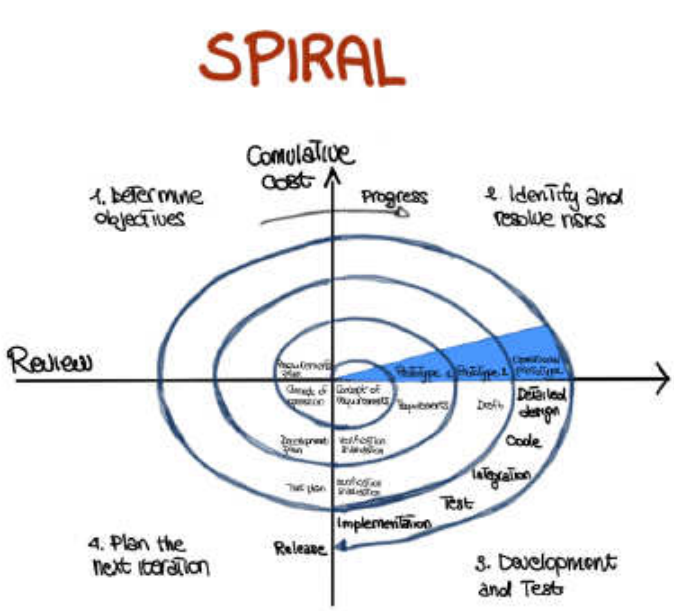
Software Process Model Introduction

- Also called software lifecycle model.
- It is a prescriptive model of what should happen from the very beginning to the very end of a software development process.
- Main function of the life cycle model is to determine the order of the different activities so that we know which activities should come first and which ones should follow.
- Another important function of the life cycle model is to determine the transition criteria between activities.
- The model should describe what should we do next and how long should we continue to do it for each activity in the model.



Waterfall model

- Grandfather of all life cycle models.
- In the waterfall model the project progresses to an orderly sequence of steps, from the initial software concept, down until the final phase of system testing.
- At the end of each phase there will be a review to determine whether the project is ready to advance to the next phase.
- Pure waterfall model performs well for software products in which there is a stable product definition
    - The domain is well known and the technologies involved are well understood.
    - In these kind of domains, the waterfall model helps you to find errors in the early, local stages of the projects.
- The main advantage of the waterfall model is that it allows you to find errors early.
- The main disadvantages of the waterfall model arise from the fact that it is not flexible.
    - Normally, it is difficult to fully specify requirements at the beginning of a project.
    - This lack of flexibility is far from ideal when dealing with project in which requirements change, the developers are not domain experts, or the technology used are new and evolving,

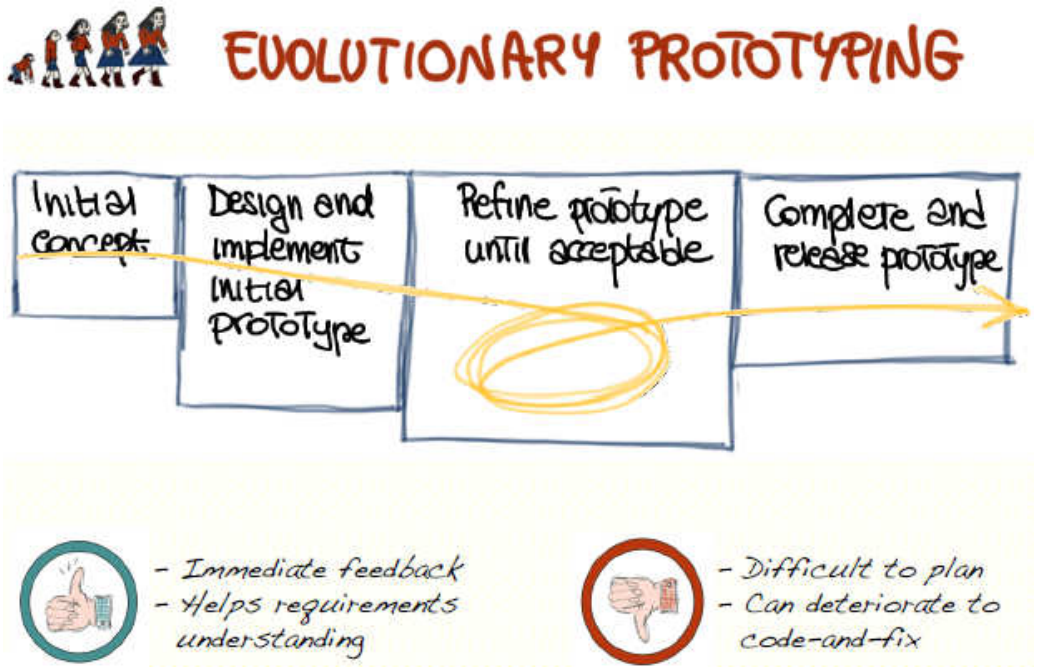


- Risk reduction
- Functionality can be added
- Software produced early

- Specific expertise
- Highly dependent on risk analysis
- Complex

The Spiral Model

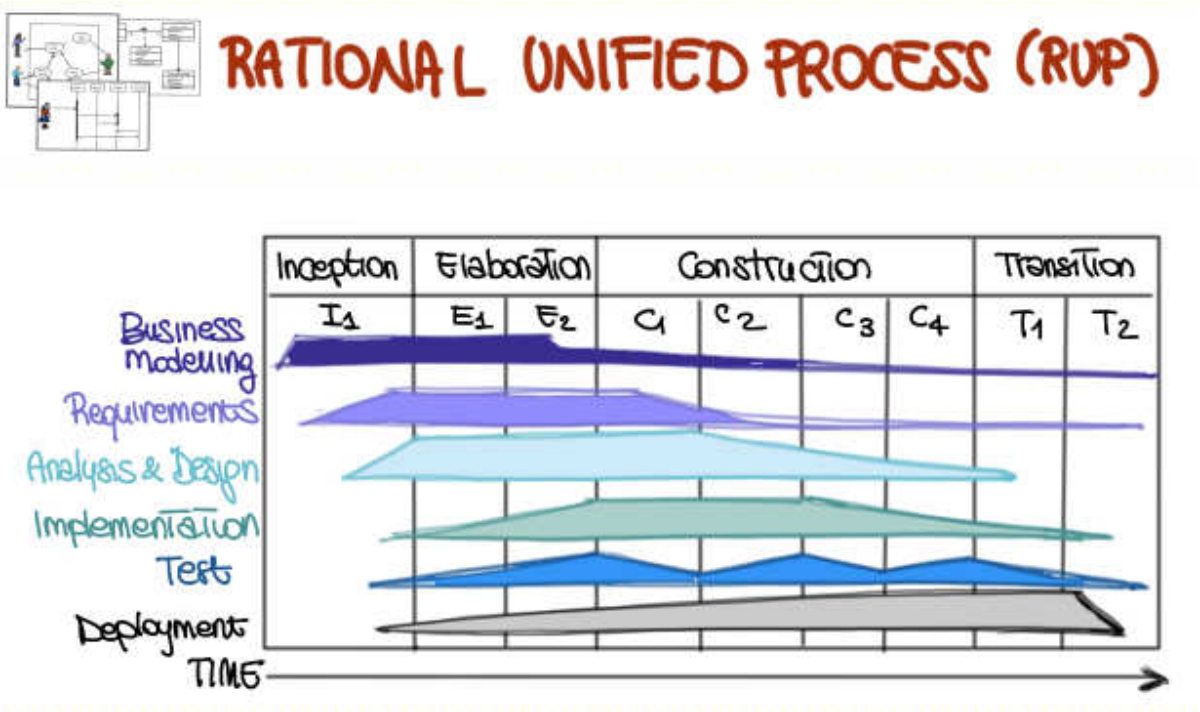
- First described by Barry Boehm in his paper from 1986 that was entitled A Spiral Model of Software Development and Enhancement.
    - One of the main characteristics of that paper is that it was describing the spiral model using a diagram which has become very very popular,
- Spiral model is an incremental risk-oriented lifecycle model that has four main phases:
    - Determine objectives - the requirements will be gathered
    - Identify and resolve risks - the risks and the alternate solutions will be identified, and a prototype will be produced
    - Development and tests – produce software and tests for the software
    - Plan the next iteration - the output of the project, so far, is evaluated, and the next iteration is planned
- A software project will go through these four phases in an iterative way.
- So basically, what the spiral process prescribes is a way of developing software by going through these phases in an iterative way, in which we learn more and more of the software, we identify more and more, and account for, more and more risks and we go more and more towards our final solution, our final release.
- Several advantages:
    - Risk reduction advantage - extensive risk analysis does reduce the chances of the project to fail. So there is a risk reduction advantage.
    - Functionality can be added at a later phase because of the iterative nature of the process.
    - Software is produced early in the software lifecycle so at any iteration we have something to show for our development.
    - Can get early feedback from the customer about what we produced.
- Main disadvantages:
    - Risk analysis requires a highly specific expertise; and since the whole success of the process is highly dependent on risk analysis, the risk analysis has to be done right.
    - Way more complex than other models and therefore it can be costly to implement.



Evolutionary Prototyping

- Works in four main phases.
    - Start from an initial concept
    - Design and implement a prototype based on this initial concept.
    - Refine the prototype until it is acceptable.
    - Complete and release the prototype.
- When developing a system using evolutionary prototyping, the system is continually refined and rebuilt.
- Is an ideal process when not all requirements are well understood, which is a very common situation.
- Developers start by developing the parts of the system that they understand, instead of developing a whole system, including parts that might not be very clear at that stage.
- Partial system is then shown to the customer and the customer feedback is used to drive the next iteration, in which either changes are made to the current features or new features are added.
- Either the current prototype is improved or the prototype is extended.
- When the customer agrees that the prototype is good enough, the developers will complete the remaining work on the system and release the prototype as the final product.
- Main advantages
    - Immediate feedback - developers get feedback immediately as soon as they produce a prototype and they show it to the customer.
        - Risk of implementing the wrong system is minimized.
- Main disadvantages

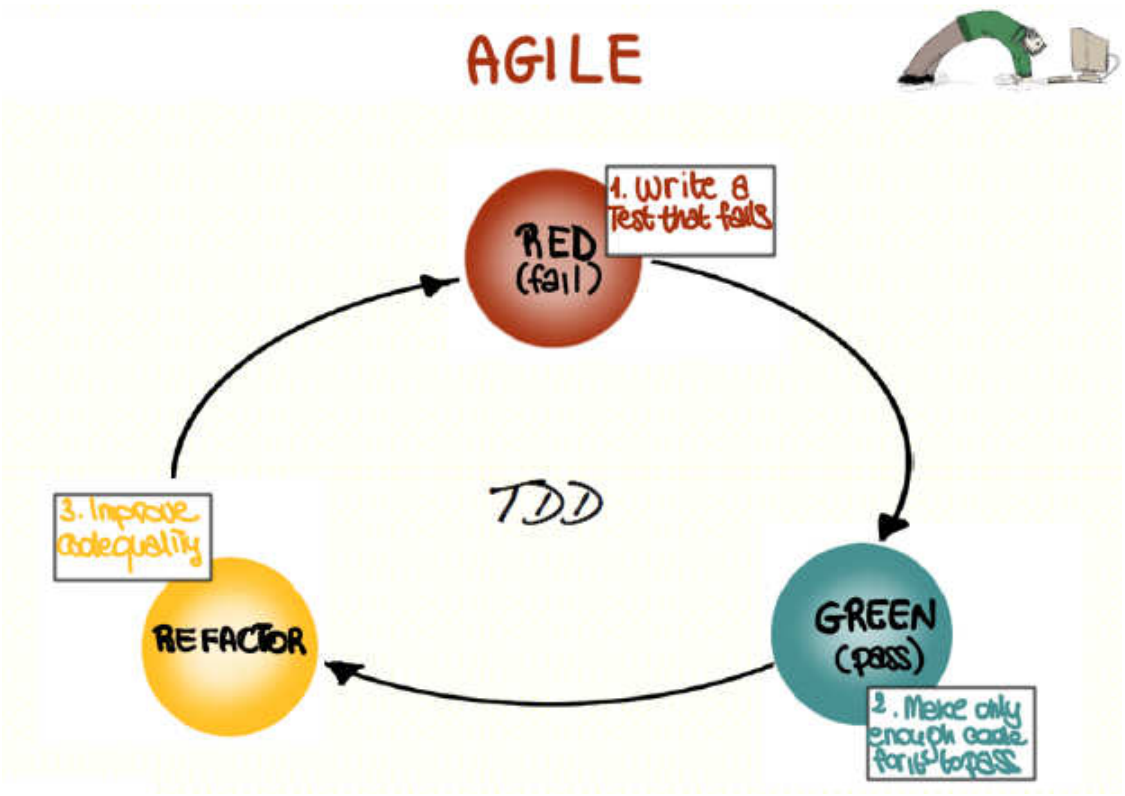
    - Difficult to plan - hard to plan in advance how long the development is going to take because we don't know how many iterations will be needed.
    - Can easily become an excuse to do a kind of code and fix type of approach, in which we hack something together, fix the main issues when the customer gives us feedback, and then continue this way, until the final product is something that is kind of working, but is not really a product of high quality.
- There are many different kinds of prototyping; evolutionary prototyping is just one of them.
- Throw-away prototyping is another kind of prototyping in which the prototype is just used to gather requirements, but is thrown away at the end of the requirements gathering instead of being evolved as it happens here.



Rational Unified Process

- A very popular process based on UML.
- RUP works in a iterative way, which means that it performs different iterations, and at each iteration it performs four phases.
- In each one of these four phases we perform standard software engineering activities and we do them to different extent, based on the phase in which we are.
- In the inception phase, the work is mostly to scope the system.
    - So basically figuring out what is the scope of the work? What is the scope of the project? What is the domain? So that we can be able to perform initial cost, and budget estimates.

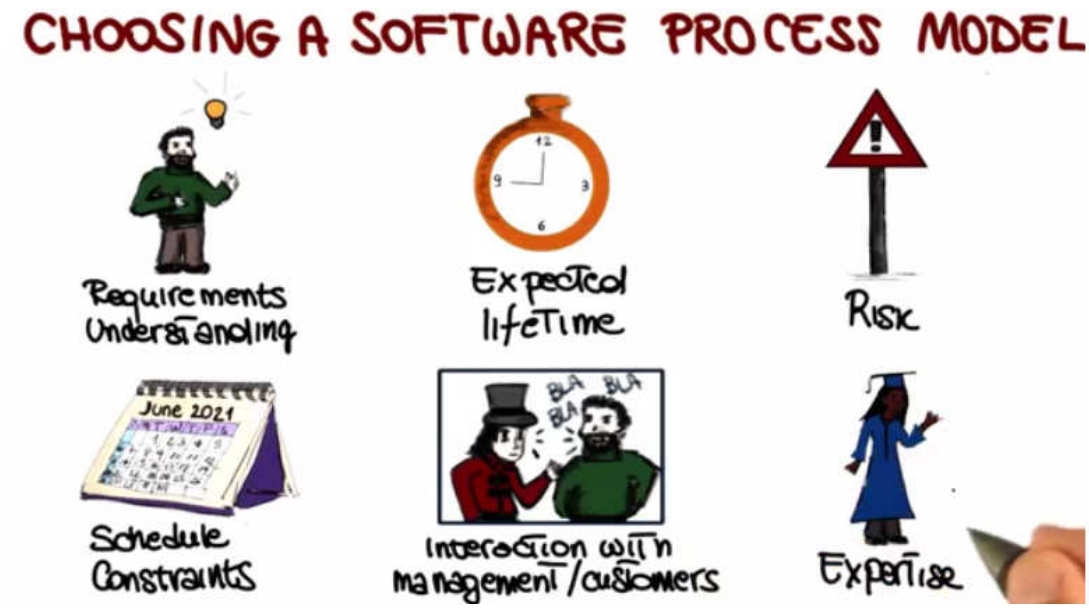
- The elaboration phase is the phase in which we focus on the domain analysis and define the basic architecture for the system.
    - So this is the phase in which analysis and design are particularly prevalent.
- Then there is the construction phase which is where the bulk of the development actually occurs and is where most of the implementation happens.
- The transition phase is the phase in which the system goes from development into production, so that it becomes available to users.
    - This is the phase in which the other activities and software development becomes less relevant and deployment becomes the main one.



Agile

- This is a group of software development benefits that are based on highly iterative and incremental development.
- Test driven development, or TDD, is based on the iteration of three main phases.
    - First, we write test cases that encode our requirements and for which we haven't written code yet, and therefore they will fail (red phase)
    - Second, we move to this phase in which after we test enough code to make the test cases pass we have a set of test cases that are all passing (green phase).

        - We have enough code to make the test cases pass, because the test cases in code are a requirements,
        - We have just written enough code to satisfy our requirements.
        - When we do this over time though, what happens is the structure of the code deteriorates, because we keep adding pieces, so that's why we have the last step, which is refactoring.
    - Third, we have refactoring where we modify the code to make it more readable, more maintainable,
        - We modify to improve the design of the code. And after this phase, we will go back to writing more test cases for new requirements.
- We continue to iterate among these phases.



Choosing a Software Process Model

- The specific process model that we choose has as much influence over a project's success as any other major planning decision that we make.
- It is very important that we pick the appropriate model for our development process.
- Picking an appropriate model can ensure the success of a project whereas if we choose the wrong model, that can be a constant source of problems and ultimately, it can make the project fail.
- So how can we choose the right model for a project? To be able to do so, we have to take into consideration many factors. In particular,
    - Level of understanding we have of the requirements.
        - Do we understand all of them?

        - Are we going to be able to collect all the requirements in advance, or collecting requirements is going to be hard and therefore, we might want to follow a process that is more flexible with that respect?
    - Expected lifetime of the project.
        - Is this a quick project that we are putting together for a specific purpose or something that's going to last for a number of years and that we're going to maintain over all those years?
    - Also, what is the level of risk involved?
        - Do we know the domain very well?
        - Do we know exactly the technologies involved? If so, we might go with a more traditional process. Otherwise, we might want to be more agile, more flexible.
    - Know the schedule constraints.
        - How much time, how many resources do we have for this project?
        - What is the expected interaction with the management and the customer?
            - There are many processes that rely on the fact that there can be a continuous interaction with the customer. If that interaction is not there, there's no way we are going to be able to use these processes.
            - Conversely, there are processes that don't require the presence of the customer at all, except for the initial phase and maybe some checking points and so if the customer is very inaccessible, we might want to follow one of those processes, instead of one of the more demanding ones in terms of customer's time.
    - Level of the expertise of the people involved.
        - Do we have people that know the technologies that we're using?
        - Do we know people that know a specific kind of process?
        - Some processes require some specific expertise and we're not going to be able to follow that process if we don't have the right expertise.
- So we need to take into account all of these aspects, and sometimes more, in order to be able to make the right decision and pick the right software process model for our project.

Life Cycle Documents

- Documenting the activities carried out during the different phases of the softer lifecycle is a very important task.
- The documents we product are used for different purposes such as:
    - Communicate details of the software systems to different stakeholders
    - Ensure the correct implementation of the system
    - Facilitate maintenance and so on.

- There are standardized documents that are provided by IEEE that you can use for this purpose. However, they're kind of heavy-weight.

Now let's talk about well known, ineffective development practices. These practices, when followed, tend to lead to predictably bad results...

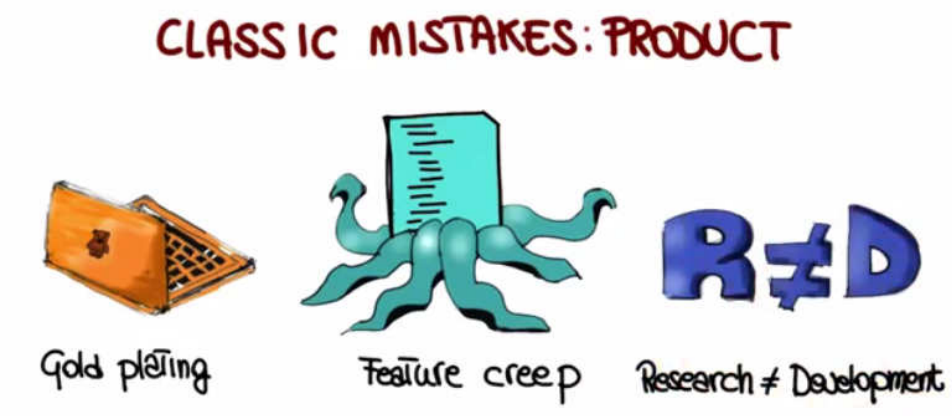
Classic Mistakes: People

- Heroics.
    - Too much emphasis on can do attitudes
    - This idea that one person by himself or by herself can do everything and can make the difference in the whole project.
    - This encourages extreme risk taking and discourages cooperation, which is plain bad for the project.
    - It might force people not to report schedule slips.
    - It might force people to take on on too much responsibility.
    - Normally the final result is a failure.
    - What you want when you're developing a larger project is have teams, have team work, and have cooperation among the different team members, without pointing too much on single individuals.
- Right working environment
    - Strong evidence that the working environments can play a big role in productivity.
    - Evidence that productivity increases when the workplace is nice, quiet, warm, and welcoming.
- Poor people management
    - Lack of leadership, or leadership that is exercised using the wrong means in the wrong way, which can lead to very unhappy personnel and therefore, low productivity, or even people leaving teams.
    - Adding people to a project that is behind schedule, which never works.
        - New people need to be brought up to speed, and that causes further delays rather than improving the situation with the project schedule.

- See Brook's Law in The Mythical Man Month (adding manpower to a late software project makes it later... Example: Nine women can't make a baby in one month)

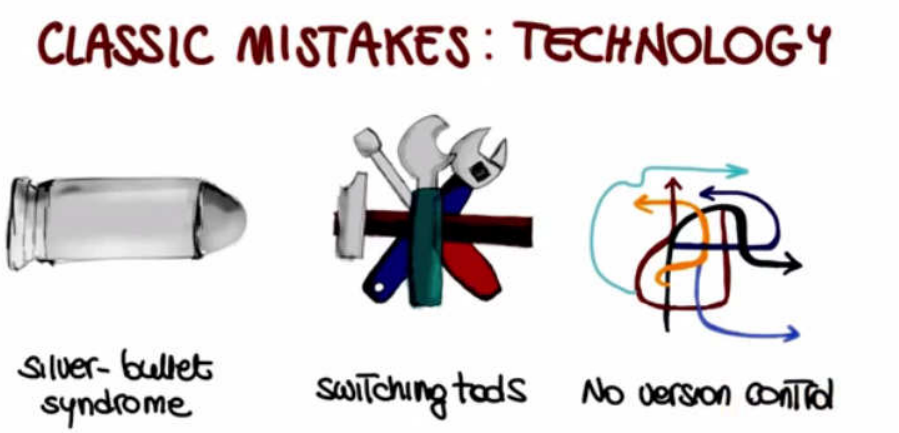
Classic Mistakes: Process

- Many types.
- Scheduling issues - due to the fact of being unable to come up with a realistic schedule.
    - Have an overly optimistic schedule because we underestimate the effort involved in different parts of the project.
    - Because we overestimate the ability of the people involved.
    - Because we overestimate the importance of the use of tools.
    - Result is typically that the projects end up being late, which is a very common situation.
- Mistakes in planning such as insufficient planning or abandoning planning due to pressure, usually lead inexorably to failure.
    - Planning is a fundamental factor in software processes and in software development.
- Failures - often there are unforeseen failures
    - Failures on the constructor's end, for example, that might lead to low quality or late deliverables, which ultimately affects the downstream activities.



Classic Mistakes: Product

- Gold plating of requirements - basically that it's very common for projects to have more requirements than they actually need.
    - For example, marketing might want to add more features than the ones that are actually needed by the users.
    - Having more requirements lengthens the project's schedule in a totally unnecessary way.
- Feature creep is another common mistake and consists in adding more and more features to a product that were not initially planned and are not really needed in most cases.
    - Average project experiences about a 25% growth in the number of features over its lifetime which can clearly highly effect the project schedule.
- Working on a project that strains the limits of computer science
    - For example, you need to develop new algorithms for the project, or you have to use new techniques.
    - The project might be more research than actual development and should be managed accordingly (i.e. take into account that you will have a highly unpredictable schedule).



Classic Mistakes: Technology

- Silver-bullet syndrome - refers to situations in which there is too much reliance on the advertised benefits of some previously unused technology.
    - For example, a new technology.
    - Problem here is that we cannot expect technology alone to solve our software development issues.
    - We should not rely too much on technology alone.
- Switch or add tools in the middle of a project.
    - Sometimes it can make sense to upgrade a tool, but introducing new tools, which can have a steep learning curve, has almost always negative effects.
- Lack of an automated version control system for your code and for your various artifacts.
    - Manual and ad hoc solutions are just not an option.
    - Way too easy to make mistakes, use out of date versions, be unable to find a previous working version, and so on.