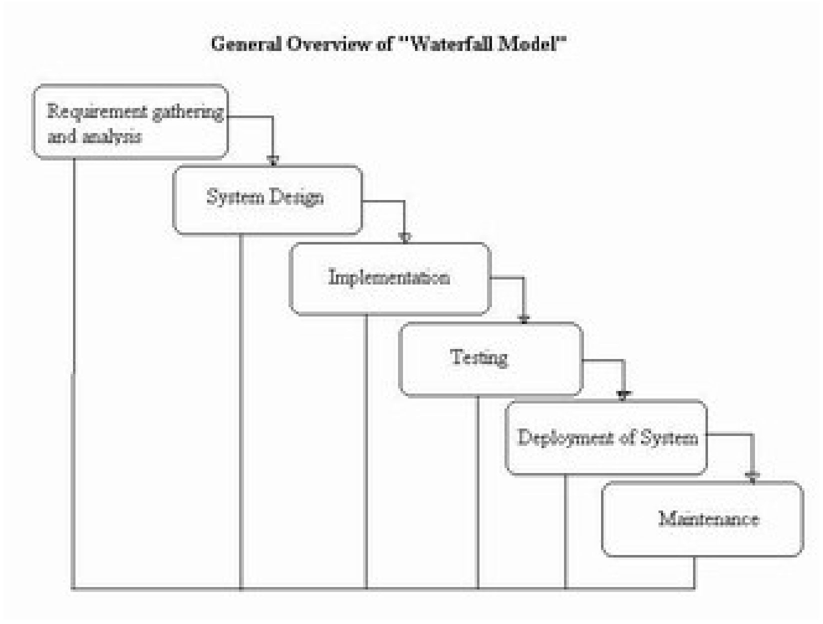
OSQA

SDLC Models

There are various software development approaches defined and designed which are used/employed during development process of software, these approaches are also referred as "Software Development Process Models". Each process model follows a particular life cycle in order to ensure success in process of software development.

Waterfall Model

Waterfall approach was first Process Model to be introduced and followed widely in Software Engineering to ensure success of the project. In "The Waterfall" approach, the whole process of software development is divided into separate process phases. The phases in Waterfall model are: Requirement Specifications phase, Software Design, Implementation and Testing & Maintenance. All these phases are cascaded to each other so that second phase is started as and when defined set of goals are achieved for first phase and it is signed off, so the name "Waterfall Model". All the methods and processes undertaken in Waterfall Model are more visible.


General Overview of "Waterfall Model"
Requirement gathering and analysis
System Design
Implementation
Testing
Deployment of System
Maintenance

The stages of "The Waterfall Model" are:

Requirement Analysis & Definition: All possible requirements of the system to be developed are captured in this phase. Requirements are set of functionalities and constraints that the end-user (who will be using the system) expects from the system. The requirements are gathered from the end-user by consultation, these requirements are analyzed for their validity and the possibility of incorporating the requirements in the system to be development is also studied. Finally, a Requirement Specification document is created which serves the purpose of guideline for the next phase of the model.

System & Software Design: Before a starting for actual coding, it is highly important to understand what we are going to create and what it should look like? The requirement specifications from first phase are studied in this phase and system design is prepared. System Design helps in specifying hardware and system requirements and also helps in defining overall system architecture. The system design specifications serve as input for the next phase of the model.

Implementation & Unit Testing: On receiving system design documents, the works divided in modules/units and actual coding is started. The system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality; this is referred to as Unit Testing. Unit testing mainly verifies if the modules/units meet their specifications.

Integration & System Testing: As specified above, the system is first divided in units which are developed and tested for their functionalities. These units are integrated into a complete system during Integration phase and tested to check if all modules/units coordinate between each other and the system as a whole behaves as per the specifications. After successfully testing the software, it is delivered to the customer.

Operations & Maintenance: This phase of "The Waterfall Model" is virtually never ending phase (Very long). Generally, problems with the system developed (which are not found during the development life cycle) come up after its practical use starts, so the issues related to the system are solved after deployment of the system. Not all the problems come in picture directly but they arise time to time and needs to be solved; hence this process is referred as Maintenance.

Advantages and Disadvantages of Waterfall Model

Advantages

The advantage of waterfall development is that it allows for departmentalization and managerial control. A schedule can be set with deadlines for each stage of development and a product can proceed through the development process like a car in a carwash, and theoretically, be delivered on time. Development moves from concept, through design, implementation, testing, installation, troubleshooting, and ends up at operation and maintenance. Each phase of development proceeds in strict order, without any overlapping or iterative steps.

Disadvantages

The disadvantage of waterfall development is that it does not allow for much reflection or revision. Once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought out in the concept stage. Alternatives to the waterfall model include joint application development (JAD), rapid application development (RAD), synch and stabilize, build and fix, and the spiral model.

Common Errors in Requirements Analysis

In the traditional waterfall model of software development, the first phase of requirements analysis is also the most important one. This is the phase which involves gathering information about the customer's needs and defining, in the clearest possible terms, the problem that the product is expected to solve.

This analysis includes understanding the customer's business context and constraints, the functions the product must perform, the performance levels it must adhere to, and the external systems it must be compatible with. Techniques used to obtain this understanding include customer interviews, use cases, and "shopping lists" of software features. The results of the analysis are typically captured in a formal requirements specification, which serves as input to the next step.

Well, at least that's the way it's supposed to work theoretically. In reality, there are a number of problems with this theoretical model, and these can cause delays and knock-on errors in the rest of the process. This article discusses some of the more common problems that project managers experience during this phase, and suggests possible solutions.

Problem 1: Customers don't (really) know what they want

Possibly the most common problem in the requirements analysis phase is that customers have only a vague idea of what they need, and it's up to you to ask the right questions and perform the analysis necessary to turn this amorphous vision into a formally-documented software requirements specification that can, in turn, be used as the basis for both a project plan and an engineering architecture.

To solve this problem, you should:

- Ensure that you spend sufficient time at the start of the project on understanding the objectives, deliverables and scope of the project.
- Make visible any assumptions that the customer is using, and critically evaluate both the likely end-user benefits and risks of the project.
- Attempt to write a concrete vision statement for the project, which encompasses both the specific functions or user benefits it provides and the overall business problem it is expected to solve.
- Get your customer to read, think about and sign off on the completed software requirements specification, to align expectations and ensure that both parties have a clear understanding of the deliverable.

Problem 2: Requirements change during the course of the project

The second most common problem with software projects is that the requirements defined in the first phase change as the project progresses. This may occur because as development progresses and prototypes are developed, customers are able to more clearly see problems with the original plan and make necessary course corrections; it may also occur because changes in the external environment require reshaping of the original business problem and hence necessitates a different solution than the one originally proposed.

Good project managers are aware of these possibilities and typically already have backup plans in place to deal with these changes.

To solve this problem, you should:

- Have a clearly defined process for receiving, analyzing and incorporating change requests, and make your customer aware of his/her entry point into this process.
- Set milestones for each development phase beyond which certain changes are not permissible -- for example, disallowing major changes once a module reaches 75 percent completion.
- Ensure that change requests (and approvals) are clearly communicated to all stakeholders, together with their rationale, and that the master project plan is updated accordingly.

Problem 3: Customers have unreasonable timelines

- It's quite common to hear a customer say something like "it's an emergency job and we need this project completed in X weeks". A common mistake is to agree to such timelines before actually performing a detailed analysis and understanding both of the scope of the project and the resources necessary to execute it. In accepting an unreasonable timeline without discussion, you are, in fact, doing your customer a disservice: it's quite likely that the project will either get delayed (because it wasn't possible to execute it in time) or suffer from quality defects (because it was rushed through without proper inspection).
- To solve this problem, you should:
- Convert the software requirements specification into a project plan, detailing tasks and resources needed at each stage and modeling best-case, middle-case and worst-case scenarios.
- Ensure that the project plan takes account of available resource constraints and keeps sufficient time for testing and quality inspection.
- Enter into a conversation about deadlines with your customer, using the figures in your draft plan as supporting evidence for your statements. Assuming that your plan is reasonable, it's quite likely that the ensuing negotiation will be both productive and result in a favorable outcome for both

Problem 4: Communication gaps exist between customers, engineers and project managers

- Often, customers and engineers fail to communicate clearly with each other because they come from different worlds and do not understand technical terms in the same way. This can lead to confusion and severe miscommunication, and an important task of a project manager, especially during the requirements analysis phase, is to ensure that both parties have a precise understanding of the deliverable and the tasks needed to achieve it.
- To solve this problem, you should:
- Take notes at every meeting and disseminate these throughout the project team.
- Be consistent in your use of words. Make yourself a glossary of the terms that you're going to use right at the start, ensure all stakeholders have a copy, and stick to them consistently

Problem 5: The development team doesn't understand the politics of the customer's organization

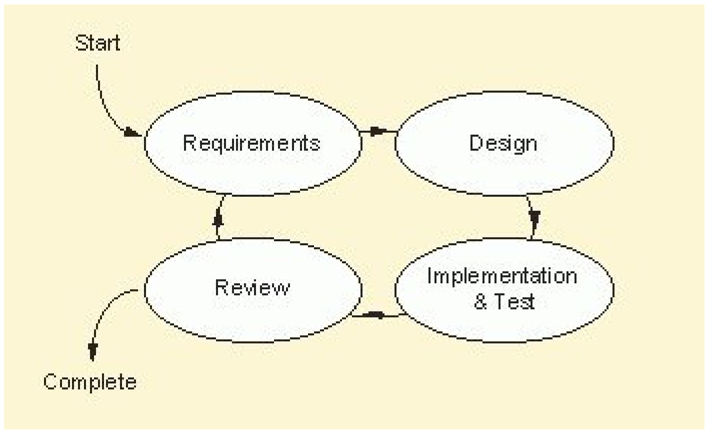
The scholars Bolman and Deal suggest that an effective manager is one who views the organization as a "contested arena" and understands the importance of power, conflict, negotiation and coalitions. Such a manager is not only skilled at operational and functional tasks, but he or she also understands the importance of framing agendas for common purposes, building coalitions that are united in their perspective, and persuading resistant managers of the validity of a particular position.

These skills are critical when dealing with large projects in large organizations, as information is often fragmented and requirements analysis is hence stymied by problems of trust, internal conflicts of interest and information inefficiencies.

To solve this problem, you should:

- Review your existing network and identify both the information you need and who is likely to have it.
- Cultivate allies, build relationships and think systematically about your social capital in the organization.
- Persuade opponents within your customer's organization by framing issues in a way that is relevant to their own experience.
- Use initial points of access/leverage to move your agenda forward.

Iterative Model

- An iterative lifecycle model does not attempt to start with a full specification of requirements. Instead, development begins by specifying and implementing just part of the software, which can then be reviewed in order to identify further requirements. This process is then repeated, producing a new version of the software for each cycle of the model. Consider an iterative lifecycle model which consists of repeating the following four phases in sequence:
- 

Requirements phase, in which the requirements for the software are gathered and analyzed. Iteration should eventually result in a requirements phase that produces a complete and final specification of requirements. –

Design phase, in which a software solution to meet the requirements is designed. This may be a new design, or an extension of an earlier design.

Implementation and Test phase, when the software is coded, integrated and tested.

Review phase, in which the software is evaluated, the current requirements are reviewed, and changes and additions to requirements proposed.

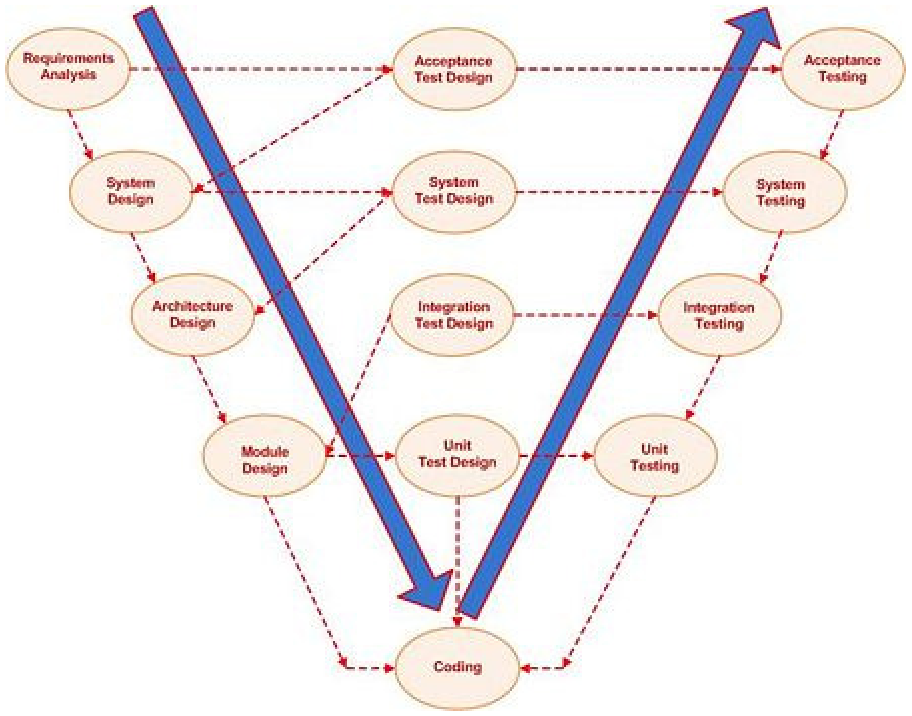
For each cycle of the model, a decision has to be made as to whether the software produced by the cycle will be discarded, or kept as a starting point for the next cycle (sometimes referred to as incremental prototyping). Eventually a point will be reached where the requirements are complete and the software can be delivered, or it becomes impossible to enhance the software as required, and a fresh start has to be made.

The iterative lifecycle model can be likened to producing software by successive approximation. Drawing an analogy with mathematical methods that use successive approximation to arrive at a final solution, the benefit of such methods depends on how rapidly they converge on a solution.

The key to successful use of an iterative software development lifecycle is rigorous validation of requirements, and verification (including testing) of each version of the software against those requirements within each cycle of the model. The first three phases of the example iterative model is in fact an abbreviated form of a sequential V or waterfall lifecycle model. Each cycle of the model produces software that requires testing at the unit level, for software integration, for system integration and for acceptance. As the software evolves through successive cycles, tests have to be repeated and extended to verify each version of the software.

V-Model

The V-model is a software development model which can be presumed to be the extension of the waterfall model. Instead of moving down in a linear way, the process steps are bent upwards after the coding phase, to form the typical V shape. The V-Model demonstrates the relationships between each phase of the development life cycle and its associated phase of testing.



Verification Phases

1. **Requirements analysis**:

   In this phase, the requirements of the proposed system are collected by analyzing the needs of the user(s). This phase is concerned about establishing what the ideal system has to perform. However, it does not determine how the software will be designed or built. Usually, the users are interviewed and a document called the user requirements document is generated. The user requirements document will typically describe the system's functional, physical, interface, performance, data, security requirements etc as expected by the user. It is one which the business analysts use to communicate their understanding of the system back to the users. The users carefully review this document as this document would serve as the guideline for the system designers in the system design phase. The user acceptance tests are designed in this phase.

2. **System Design**:

System engineers analyze and understand the business of the proposed system by studying the user requirements document. They figure out possibilities and techniques by which the user requirements can be implemented. If any of the requirements are not feasible, the user is informed of the issue. A resolution is found and the user requirement document is edited accordingly.

The software specification document which serves as a blueprint for the development phase is generated. This document contains the general system organization, menu structures, data structures etc. It may also hold example business scenarios, sample windows, reports for the better understanding. Other technical documentation like entity diagrams, data dictionary will also be produced in this phase. The document for system testing is prepared in this phase.

3. **Architecture Design**:

This phase can also be called as high-level design. The baseline in selecting the architecture is that it should realize all which typically consists of the list of modules, brief functionality of each module, their interface relationships, dependencies, database tables, architecture diagrams, technology details etc. The integration testing design is carried out in this phase.

4. **Module Design**:

This phase can also be called as low-level design. The designed system is broken up in to smaller units or modules and each of them is explained so that the programmer can start coding directly. The low level design document or program specifications will contain a detailed functional logic of the module, in pseudo code - database tables, with all elements, including their type and size - all interface details with complete API references- all dependency issues- error message listings- complete input and outputs for a module. The unit test design is developed in this stage.

Spiral Model

History

The spiral model was defined by Barry Boehm in his 1988 article A Spiral Model of Software Development and Enhancement. This model was not the first model to discuss iterative development, but it was the first model to explain why the iteration matters. As originally envisioned, the iterations were typically 6 months to 2 years long. Each phase starts with a design goal and ends with the client (who may be internal) reviewing the progress thus far. Analysis and engineering efforts are applied at each phase of the project, with an eye toward the end goal of the project.

The Spiral Model

The spiral model, also known as the spiral lifecycle model, is a systems development method (SDM) used in information technology (IT). This model of development combines the features of the prototyping model and the waterfall model. The spiral model is intended for large, expensive, and complicated projects.

The steps in the spiral model can be generalized as follows:

1. The new system requirements are defined in as much detail as possible. This usually involves interviewing a number of users representing all the external or internal users and other aspects of the existing system.

2. A preliminary design is created for the new system.
3. A first prototype of the new system is constructed from the preliminary design. This is usually a scaled-down system, and represents an approximation of the characteristics of the final product.

4. A second prototype is evolved by a fourfold procedure: (1) evaluating the first prototype in terms of its strengths, weaknesses, and risks; (2) defining the requirements of the second prototype; (3) planning and designing the second prototype; (4) constructing and testing the second prototype.

5. At the customer's option, the entire project can be aborted if the risk is deemed too great. Risk factors might involve development cost overruns, operating-cost miscalculation, or any other factor that could, in the customer's judgment, result in a less-than-satisfactory final product.

6. The existing prototype is evaluated in the same manner as was the previous prototype, and, if necessary, another prototype is developed from it according to the fourfold procedure outlined above.

7. The preceding steps are iterated until the customer is satisfied that the refined prototype represents the final product desired.

8. The final system is constructed, based on the refined prototype.

9. The final system is thoroughly evaluated and tested. Routine maintenance is carried out on a continuing basis to prevent large-scale failures and to minimize downtime.

Applications

For a typical shrink-wrap application, the spiral model might mean that you have a rough-cut of user elements (without the polished / pretty graphics) as an operable application, add features in phases, and, at some point, add the final graphics. The spiral model is used most often in large projects. For smaller projects, the concept of agile software development is becoming a viable alternative.

Advantages

1. Estimates (i.e. budget, schedule, etc.) become more realistic as work progresses, because important issues are discovered earlier.

2. It is more able to cope with the (nearly inevitable) changes that software development generally entails.

3. Software engineers (who can get restless with protracted design processes) can get their hands in and start working on a project earlier.

Disadvantages

1. Highly customized limiting re-usability

2. Applied differently for each application

3. Risk of not meeting budget or schedule

4. Risk of not meeting budget or schedule

The Big Bang Model

Bin – Bang Model
The Big- Bang Model is the one in which we put huge amount of matter (people or money) is put together, a lot of energy is expended – often violently – and out comes the perfect software product or it doesn't.
The beauty of this model is that it's simple. There is little planning, scheduling, or Formal development process. All the effort is spent developing the software and writing the code. It's and ideal process if the product requirements aren't well understood and the final release date is flexible. It's also important to have flexible customers, too, because they won't know what they're getting until the very end.

RAD Model

What is RAD?

RAD (rapid application development) is a concept that products can be developed faster and of higher quality through:

- Gathering requirements using workshops or focus groups
- Prototyping and early, reiterative user testing of designs
- The re-use of software components
- A rigidly paced schedule that defers design improvements to the next product version
- Less formality in reviews and other team communication

Some companies offer products that provide some or all of the tools for RAD software development. (The concept can be applied to hardware development as well.) These products include requirements gathering tools, prototyping tools, computer-aided software engineering tools, language development environments such as those for the Java platform, groupware for communication among development members, and testing tools. RAD usually embraces object-oriented programming methodology, which inherently fosters software re-use. The most popular object-oriented programming languages, C++ and Java, are offered in visual programming packages often described as providing rapid application development.

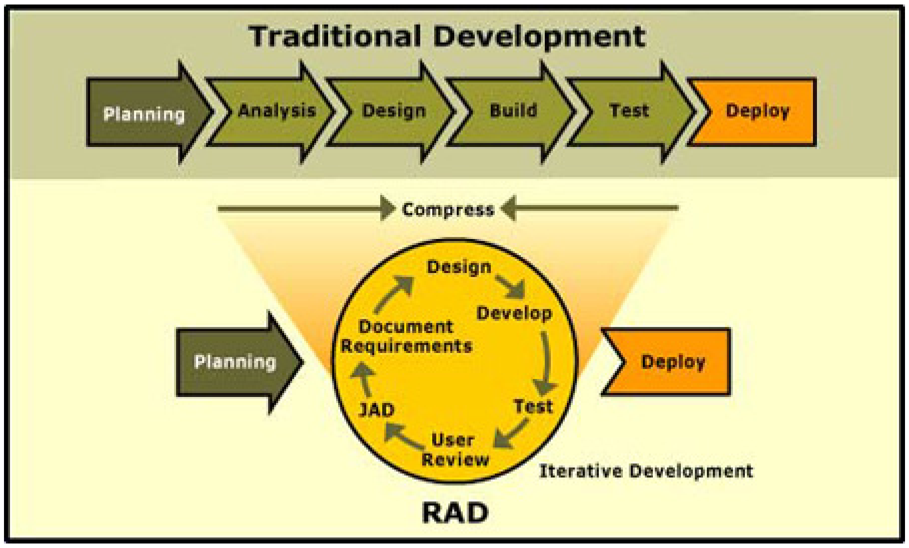
Development Methodology

The traditional software development cycle follows a rigid sequence of steps with a formal sign-off at the completion of each. A complete, detailed requirements analysis is done that attempts to capture the system requirements in a Requirements Specification. Users are forced to "sign-off" on the specification before development proceeds to the next step. This is followed by a complete system design and then development and testing.

But, what if the design phase uncovers requirements that are technically unfeasible, or extremely expensive to implement? What if errors in the design are encountered during the build phase? The elapsed time between the initial analysis and testing is usually a period of several months. What if business requirements or priorities change or the users realize they overlooked critical needs during the analysis phase?

These are many of the reasons why software development projects either fail or don't meet the user's expectations when delivered.

RAD is a methodology for compressing the analysis, design, build, and test phases into a series of short, iterative development cycles. This has a number of distinct advantages over the traditional sequential development model.



RAD projects are typically staffed with small integrated teams comprised of developers, end users, and IT technical resources. A small team, combined with short, iterative development cycles optimizes speed, unity of vision and purpose, effective informal communication and simple project management.

RAD Model Phases

RAD model has the following phases:

1. **Business Modeling:**

   The information flow among business functions is defined by answering questions like what information drives the business process, what information is generated, who generates it, where does the information go, who process it and so on.

2. **Data Modeling:**

   The information collected from business modeling is refined into a set of data objects (entities) that are needed to support the business. The attributes (character of each entity) are identified and the relation between these data objects (entities) is defined.

3. **Process Modeling:**

   The data object defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting or retrieving a data object.

4. **Application Generation:**

   Automated tools are used to facilitate construction of the software; even they use the 4th GL techniques.

5. **Testing and Turn over:**

   Many of the programming components have already been tested since RAD emphasis reuse. This reduces overall testing time. But new components must be tested and all interfaces must be fully exercised.

Advantages and Disadvantages

RAD reduces the development time and reusability of components help to speed up development. All functions are modularized so it is easy to work with.

For large projects RAD require highly skilled engineers in the team. Both end customer and developer should be committed to complete the system in a much abbreviated time frame. If commitment is lacking RAD will fail. RAD is based on

Object Oriented approach and if it is difficult to modularize the project the RAD may not work well

Prototyping Model

A prototype is a working model that is functionally equivalent to a component of the product.

In many instances the client only has a general view of what is expected from the software product. In such a scenario where there is an absence of detailed information regarding the input to the system, the processing needs and the output requirements, the prototyping model may be employed.

This model reflects an attempt to increase the flexibility of the development process by allowing the client to interact and experiment with a working representation of the product. The developmental process only continues once the client is satisfied with the functioning of the prototype. At that stage the developer determines the specifications of the client’s real needs.

Software prototyping

Software prototyping, a possible activity during software development, is the creation of prototypes, i.e., incomplete versions of the software program being developed.

A prototype typically implements only a small subset of the features of the eventual program, and the implementation may be completely different from that of the eventual product.

The purpose of a prototype is to allow users of the software to evaluate proposals for the design of the eventual product by actually trying them out, rather than having to interpret and evaluate the design based on descriptions.

Prototyping has several benefits: The software designer and implementer can obtain feedback from the users early in the project. The client and the contractor can compare if the software made matches the software specification, according to which the software program is built. It also allows the software engineer some insight into the accuracy of initial project estimates and whether the deadlines and milestones proposed can be successfully met. The degree of completeness and the techniques used in the prototyping have been in development and debate since its proposal in the early 1970's.

This process is in contrast with the 1960s and 1970s monolithic development cycle of building the entire program first and then working out any inconsistencies between design and implementation, which led to higher software costs and poor estimates of time and cost. The monolithic approach has been dubbed the "Slaying the (software)Dragon" technique, since it assumes that the software designer and developer is a single hero who has to slay the entire dragon alone. Prototyping can also avoid the great expense and difficulty of changing a finished software product.

Overview

The process of prototyping involves the following steps

1. Identify basic requirements

   Determine basic requirements including the input and output information desired. Details, such as security, can typically be ignored.

2. Develop Initial Prototype

   The initial prototype is developed that includes only user interfaces.

3. Review

   The customers, including end-users, examine the prototype and provide feedback on additions or changes.

4. Revise and Enhancing the Prototype

   Using the feedback both the specifications and the prototype can be improved. Negotiation about what is within the scope of the contract/product may be necessary. If changes are introduced then a repeat of steps #3 ands #4 may be needed.

Versions

There are two main versions of prototyping model:

Version I: Prototyping is used as a requirements technique.

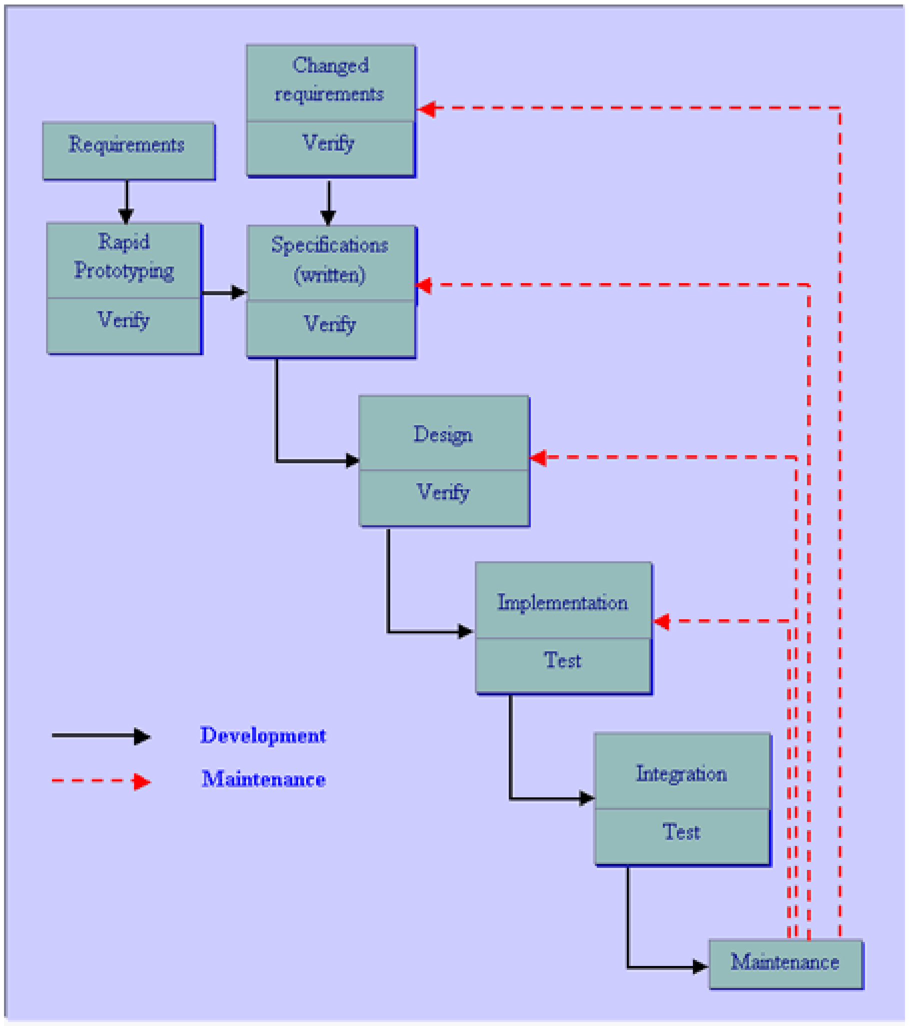
Version II: Prototype is used as the specifications or a major part thereof.

Version One

This approach, uses the prototype as a means of quickly determining the needs of the client; it is discarded once the specifications have been agreed on. The emphasis of the prototype is on representing those aspects of the software that will be visible to the client/user (e.g. input approaches and output formats). Thus it does not matter if the prototype hardly works.

Note that if the first version of the prototype does not meet the client's needs, then it must be rapidly converted into a second version.

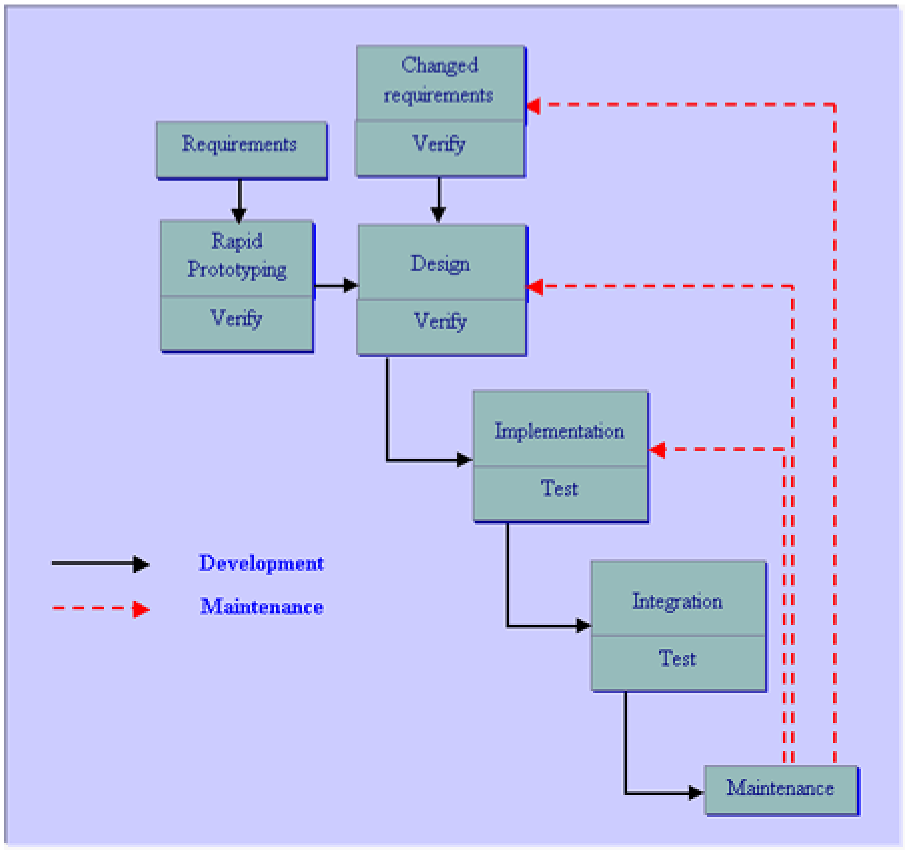
OSQA



- Prototyping Model Version 1

Version Second

In this approach, the prototype is actually used as the specifications for the design phase. This advantage of this approach is speed and accuracy, as not time is spent on drawing up written specifications. The inherent difficulties associated with that phase (i.e. incompleteness, contradictions and ambiguities) are then avoided.



Prototyping Model Version II

Types of Prototyping

Software prototyping has many variants. However, all the methods are in some way based on two major types of prototyping: Throwaway Prototyping and Evolutionary Prototyping.

Throwaway prototyping

Throwaway or Rapid Prototyping refers to the creation of a model that will eventually be discarded rather than becoming part of the finally delivered software. After preliminary requirements gathering is accomplished, a simple working model of the

system is constructed to visually show the users what their requirements may look like when they are implemented into a finished system.
Rapid Prototyping involved creating a working model of various parts of the system at a very early stage, after a relatively short investigation. The method used in building it is usually quite informal, the most important factor being the speed with which the model is provided. The model then becomes the starting point from which users can re-examine their expectations and clarify their requirements. When this has been achieved, the prototype model is 'thrown away', and the system is formally developed based on the identified requirements.
The most obvious reason for using Throwaway Prototyping is that it can be done quickly. If the users can get quick feedback on their requirements, they may be able to refine them early in the development of the software. Making changes early in the development lifecycle is extremely cost effective since there is nothing at that point to redo. If a project is changed after a considerable work has been done then small changes could require large efforts to implement since software systems have many dependencies. Speed is crucial in implementing a throwaway prototype, since with a limited budget of time and money little can be expended on a prototype that will be discarded.
Another strength of Throwaway Prototyping is its ability to construct interfaces that the users can test. The user interface is what the user sees as the system, and by seeing it in front of them, it is much easier to grasp how the system will work.
…it is asserted that revolutionary rapid prototyping is a more effective manner in which to deal with user requirements-related issues, and therefore a greater enhancement to software productivity overall. Requirements can be identified, simulated, and tested far more quickly and cheaply when issues of evolvability, maintainability, and software structure are ignored. This, in turn, leads to the accurate specification of requirements, and the subsequent construction of a valid and usable system from the user's perspective via conventional software development models.
Prototypes can be classified according to the fidelity with which they resemble the actual product in terms of appearance, interaction and timing. One method of creating a low fidelity Throwaway Prototype is Paper Prototyping. The prototype is implemented using paper and pencil, and thus mimics the function of the actual product, but does not look at all like it. Another method to easily build high fidelity Throwaway Prototypes is to use a GUI Builder and create a click dummy, a prototype that looks like the goal system, but does not provide any functionality.
Not exactly the same as Throwaway Prototyping, but certainly in the same family, is the usage of storyboards, animatics or drawings. These are non-functional implementations but show how the system will look.

Evolutionary Prototyping

Evolutionary Prototyping (also known as breadboard prototyping) is quite different from Throwaway Prototyping. The main goal when using Evolutionary Prototyping is to build a very robust prototype in a structured manner and constantly refine it. "The reason for this is that the Evolutionary prototype, when built, forms the heart of the new system, and the improvements and further requirements will be built
When developing a system using Evolutionary Prototyping, the system is continually refined and rebuilt.
"…evolutionary prototyping acknowledges that we do not understand all the requirements and builds only those that are well understood."
This technique allows the development team to add features, or make changes that couldn't be conceived during the requirements and design phase.
For a system to be useful, it must evolve through use in its intended operational environment. A product is never "done;" it is always maturing as the usage environment changes…we often try to define a system using our most familiar frame of reference---where we are now. We make assumptions about the way business will be conducted and the technology base on which the business will be implemented. A plan is enacted to develop the capability, and, sooner or later, something resembling the envisioned system is delivered.
Evolutionary Prototyping have an advantage over Throwaway Prototyping in that they are functional systems. Although they may not have all the features the users have planned, they may be used on an interim basis until the final system is delivered.
"It is not unusual within a prototyping environment for the user to put an initial prototype to practical use while waiting for a more developed version…The user may decide that a 'flawed' system is better than no system at all."
In Evolutionary Prototyping, developers can focus themselves to develop parts of the system that they understand instead of working on developing a whole system.
To minimize risk, the developer does not implement poorly understood features. The partial system is sent to customer sites. As users work with the system, they detect opportunities for new features and give requests for these features to developers. Developers then take these enhancement requests along with their own and use sound configuration-management practices to change the software-requirements specification, update the design, recode and retest.

Incremental Prototyping

The final product is built as separate prototypes. At the end the separate prototypes are being merged in an overall design.

Advantages of Prototyping

There are many advantages to using prototyping in software development, some tangible some abstract.

Reduced time and costs: Prototyping can improve the quality of requirements and specifications provided to developers. Because changes cost exponentially more to implement as they are detected later in development, the early determination of what the user really wants can result in faster and less expensive software.

Improved and increased user involvement: Prototyping requires user involvement and allows them to see and interact with a prototype allowing them to provide better and more complete feedback and specifications. The presence of the prototype being examined by the user prevents many misunderstandings and miscommunications that occur when each side believe the other understands what they said. Since users know the problem domain better than anyone on the development team does, increased interaction can result in final product that has greater tangible and intangible quality. The final product is more likely to satisfy the users desire for look, feel and performance.

Disadvantages of Prototyping

Using, or perhaps misusing, prototyping can also have disadvantages.

Insufficient analysis: The focus on a limited prototype can distract developers from properly analyzing the complete project. This can lead to overlooking better solutions, preparation of incomplete specifications or the conversion of limited prototypes into poorly engineered final projects that are hard to maintain. Further, since a prototype is limited in functionality it may not scale well if the prototype is used as the basis of a final deliverable, which may not be noticed if developers are too focused on building a prototype as a model.

User confusion of prototype and finished system: Users can begin to think that a prototype, intended to be thrown away, is actually a final system that merely needs to be finished or polished. (They are, for example, often unaware of the effort needed to add error-checking and security features which a prototype may not have.) This can lead them to expect the prototype to accurately model the performance of the final system when this is not the intent of the developers. Users can also become attached to features that were included in a prototype for consideration and then removed from the specification for a final system. If users are able to require all proposed features be included in the final system this can lead to feature creep.

Developer attachment to prototype: Developers can also become attached to prototypes they have spent a great deal of effort producing; this can lead to problems like attempting to convert a limited prototype into a final system when it does not have an appropriate underlying architecture. (This may suggest that throwaway prototyping, rather than evolutionary prototyping, should be used.)

Excessive development time of the prototype: A key property to prototyping is the fact that it is supposed to be done quickly. If the developers lose sight of this fact, they very well may try to develop a prototype that is too complex. When the prototype is thrown away the precisely developed requirements that it provides may not yield a sufficient increase in productivity to make up for the time spent developing the prototype. Users can become stuck in debates over details of the prototype, holding up the development team and delaying the final product.

Expense of implementing prototyping: the start up costs for building a development team focused on prototyping may be high. Many companies have development methodologies in place, and changing them can mean retraining, retooling, or both. Many companies tend to just jump into the prototyping without bothering to retrain their workers as much as they should.

> A common problem with adopting prototyping technology is high expectations for productivity with insufficient effort behind the learning curve. In addition to training for the use of a prototyping technique, there is an often overlooked need for developing corporate and project specific underlying structure to support the technology. When this underlying structure is omitted, lower productivity can often result.

Best projects to use Prototyping

It has been argued that prototyping, in some form or another, should be used all the time. However, prototyping is most beneficial in systems that will have many interactions with the users.

It has been found that prototyping is very effective in the analysis and design of on-line systems, especially for transaction processing, where the use of screen dialogs is much more in evidence. The greater the interaction between the computer and the user, the greater the benefit is that can be obtained from building a quick system and letting the user play with it.

Systems with little user interaction, such as batch processing or systems that mostly do calculations, benefit little from prototyping. Sometimes, the coding needed to perform the system functions may be too intensive and the potential gains that prototyping could provide are too small.

Prototyping is especially good for designing good human-computer interfaces. "One of the most productive uses of rapid prototyping to date has been as a tool for iterative user requirements engineering and human-computer interface design."

Methods

There are few formal prototyping methodologies even though most Agile Methods rely heavily upon prototyping techniques.

Dynamic systems development method

Dynamic Systems Development Method (DSDM) is a framework for delivering business solutions that relies heavily upon prototyping as a core technique, and is itself ISO 9001 approved. It expands upon most understood definitions of a prototype. According to DSDM the prototype may be a diagram, a business process, or even a system placed into production. DSDM prototypes are intended to be incremental, evolving from simple forms into more comprehensive ones.

DSDM prototypes may be throwaway or evolutionary. Evolutionary prototypes may be evolved horizontally (breadth then depth) or vertically (each section is built in detail with additional iterations detailing subsequent sections). Evolutionary prototypes can eventually evolve into final systems.

The four categories of prototypes as recommended by DSDM are:

Business prototypes – used to design and demonstrates the business processes being automated.

Usability prototypes – used to define, refine, and demonstrate user interface usability, accessibility, look and feel.

Performance and capacity prototypes - used to define, demonstrate, and predict how systems will perform under peak loads as well as to demonstrate and evaluate other non-functional aspects of the system (transaction rates, data storage volume, response time, etc.)

Capability/technique prototypes – used to develop, demonstrate, and evaluate a design approach or concept.

The DSDM lifecycle of a prototype is to:

Identify prototype

Agree to a plan

Create the prototype

Review the prototype

Tools

Efficiently using prototyping requires that an organization have proper tools and a staff trained to use those tools. Tools used in prototyping can vary from individual tools like 4th generation programming languages used for rapid prototyping to complex integrated CASE tools.

4th generation programming languages like Visual Basic are frequently used since they are cheap, well known and relatively easy and fast to use. CASE tools, like the Requirements Engineering Environment are often developed or selected by the military or large organizations. Object oriented tools are also being developed like LYMB from the GE Research and Development Center.