SOFTWARE DEVELOPMENT PROJECTS

Programs ***versus*** Products

usually small in size and support limited functionalities.

author of a program is usually the sole user of the software and himself maintains the code.

lack good user-interface and proper documentation.

poor maintainability, efficiency, and reliability.

do not have any supporting documents such as users' manual, maintenance manual, design document, test documents, etc.

Types of Software Development Projects

A software development company is typically structured into a large number of teams that handle various types of software development projects.

Software products

generic software products

- determines the features or functionalities that would be useful to a large cross section of users. Based on these, the development team draws up the product specification on its own.

Software services

A software service usually involves either development of a *customized software* or development of some specific part of a software in an outsourced mode.

A *customised software* is developed according to the specification drawn up by one or at most a few customers

The developing company may tailor one of its existing software products that it might have developed in the past for some other client.

In a customised software development project, a large part of the software is reused from the code of related software that the company might have already developed.

Usually, only a small part of the software that is specific to some client is developed.

For example, suppose a software development organisation has developed an academic automation software that automates the student registration, grading, Establishment, hostel and other aspects of an academic institution. When a new educational institution requests for developing a software for automation of its activities, a large part of the existing software would be reused

Another type of software service is *outsourced software*

For example, a company that has developed a generic software product usually gets an uninterrupted stream of revenue that is spread over several years.

However, this entails substantial upfront investment in developing the software and any return on this investment is subject to the risk of customer acceptance.

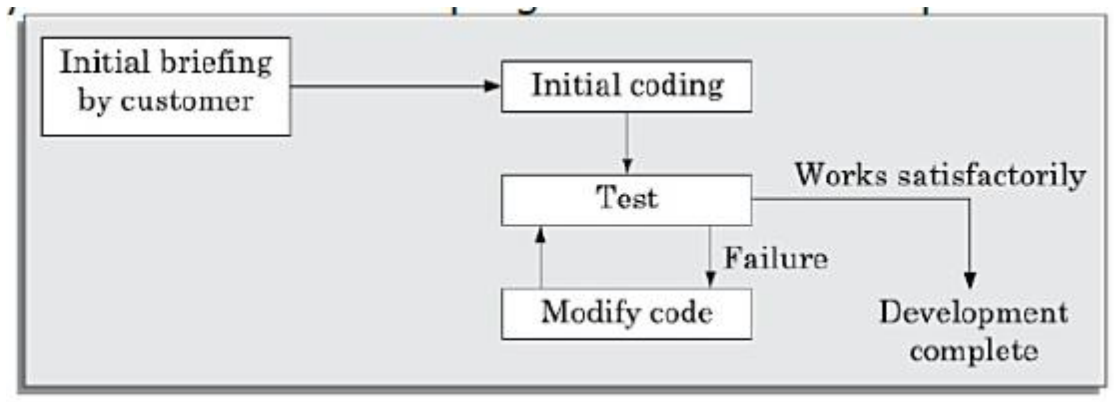
On the other hand, outsourced projects are usually less risky, but fetch only one time revenue to the developing company.

EXPLORATORY STYLE OF SOFTWARE DEVELOPMENT

Informal development style where the programmer makes use of his own intuition to develop a program.

The exploratory development style gives complete freedom to the programmer to choose the activities using which to develop software.

A schematic of this work sequence is a build and fix style.



What is wrong with the exploratory style of software development?

Effort and time required to develop a professional software increases with the increase in program size.

For large problems, it would take too long and cost too much to be practically meaningful to develop the program.

Shortcomings of the exploratory style

The foremost difficulty is the exponential growth of development time and effort with problem size and large-sized software becomes almost impossible using this style of development.

The exploratory style usually results in unmaintainable code. The reason for this is that any code developed without proper design would result in highly unstructured and poor quality code.

It becomes very difficult to use the exploratory style in a team development environment. It becomes very difficult to meaningfully partition the work among a set of developers who can work concurrently.

Besides poor quality code, lack of proper documentation makes any later maintenance of the code very difficult.

Why study software engineering?

Develop the skill to participate in development of large software after learning the systematic techniques that are being used in the industry.

Effectively handle complexity in a software development problem. In particular, you would learn how to apply the principles of abstraction and decomposition to handle complexity during various stages in software development such as specification, design, construction, and testing.

Early Computer Programming

Early commercial computers were very slow and too elementary

Those programs were usually written in assembly languages.

Program lengths were typically limited to about a few hundreds of lines of monolithic assembly code.

Every programmer developed his own individualistic style of writing programs according to his intuition and used this style *ad hoc* while writing different programs.

Build and fix (or the *exploratory programming*) style.

High-level Language Programming

High-level languages such as FORTRAN, ALGOL, and COBOL were introduced. This considerably reduced the effort required to develop software and helped programmers to write larger programs

Typical programs were limited to sizes of around a few thousands of lines of source code.

Control Flow-based Design

As the size and complexity of programs kept on increasing, the exploratory programming style proved to be insufficient.

Programmers found it increasingly difficult not only to write cost-effective and correct programs, but also to understand and maintain programs written by others.

Focus on the design of a program’s *control flow structure*.

A program’s control flow structure indicates the sequence in which the program’s instructions are executed.

In order to help develop programs having good control flow structures, the *flow charting technique* was developed.

Structured programming

The need to restrict the use of GO TO statements was recognised by everybody.

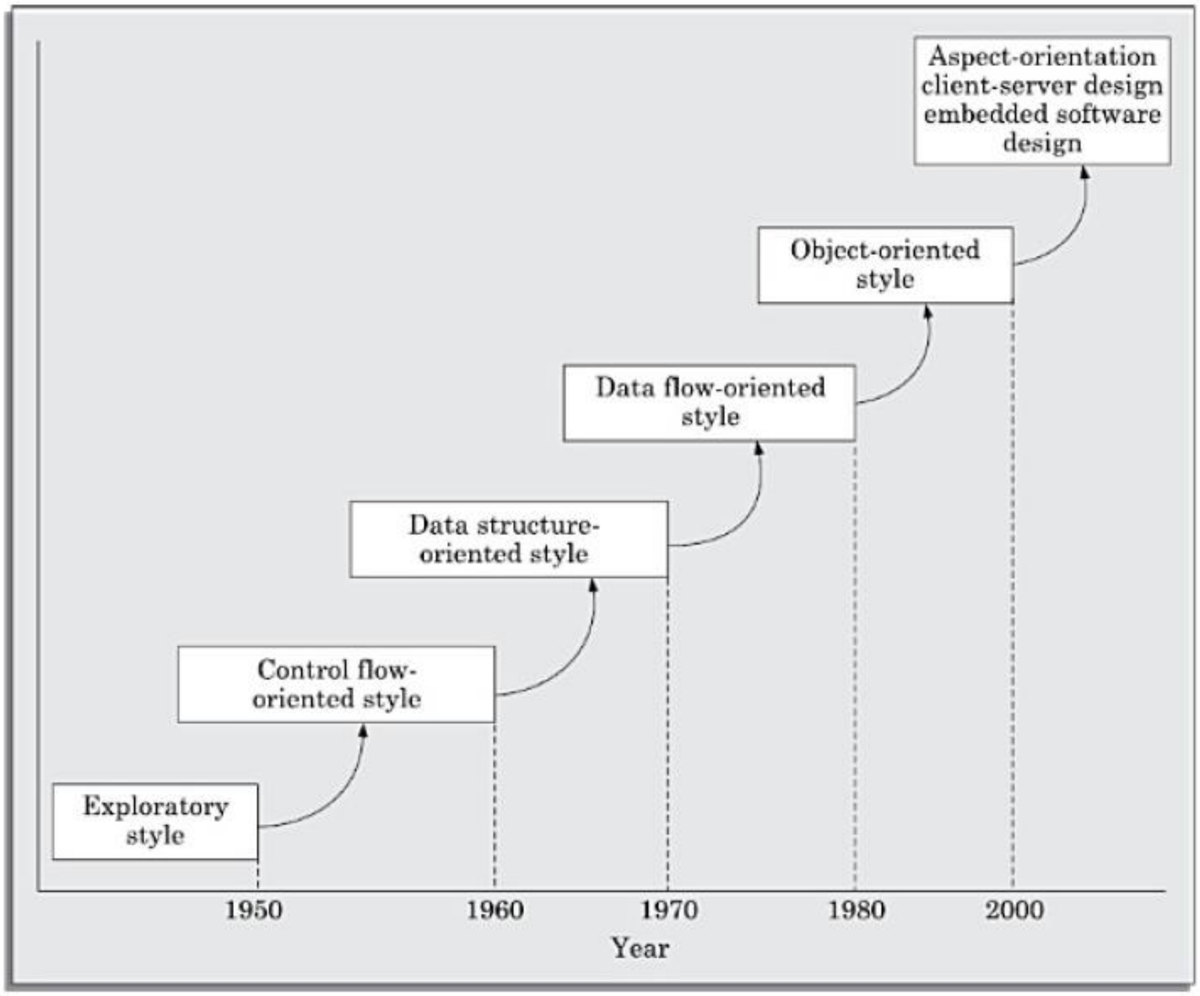
JUMP instructions are frequently used for program branching in assembly languages.

Only three programming constructs—sequence, selection, and iteration—were sufficient to express any programming logic.

This formed the basis of the structured programming methodology.

A program is called structured when it uses only the sequence, selection, and iteration types of constructs and is modular.

Abstraction (modelling)
Decomposition (Divide and conquer)


Aspect-orientation client-server design embedded software design
Object-oriented style
Data flow-oriented style
Data structure-oriented style
Control flow-oriented style
Exploratory style
1950
1960
1970
1980
2000
Year

CHANGES IN SOFTWARE DEVELOPMENT PRACTICES

Exploratory software development style is based on *error correction (build and fix)* while the software engineering techniques are based on the principles of *error prevention*.

Inherent in the software engineering principles is the realisation that it is much more cost-effective to prevent errors from occurring than to correct them as and when they are detected.

In the exploratory style, coding was considered synonymous with software development - developing a working system as quickly as possible and then successively modifying it until it performed satisfactorily.

Exploratory programming not only turns out to be prohibitively costly for non-trivial problems, but also produces hard-to-maintain programs. In the modern software development style, coding is regarded as only a small part of the overall software development activities.

A lot of attention is now being paid to requirements specification.

Unless the requirements specification is able to correctly capture the exact customer requirements, large number of rework would be necessary at a later stage.

There is a distinct design phase where standard design techniques are employed to yield coherent and complete design models.

Periodic reviews are being carried out during all stages of the development process. The main objective of carrying out reviews is *phase containment of errors*, i.e. detect and correct errors as soon as possible.

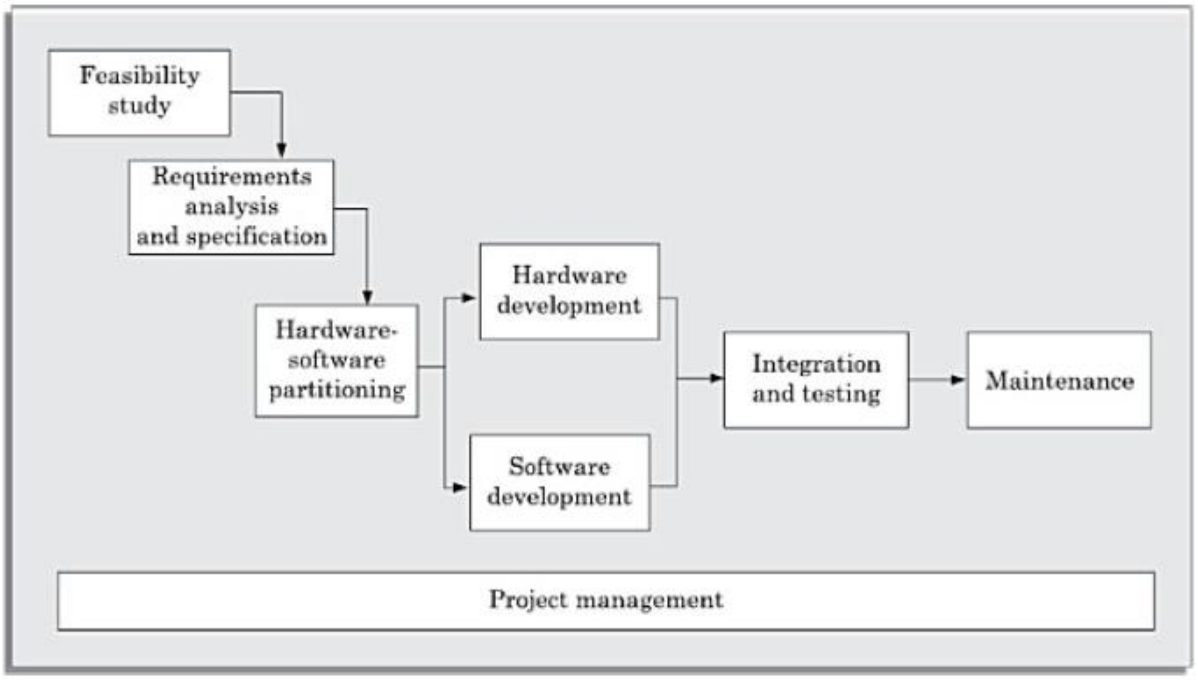
Visibility of the software is improved - production of good quality, consistent and peer reviewed documents at the end of every software development

In the exploratory style, the design and test activities were not documented satisfactorily. Today, consciously good quality documents are being developed during software development. This has made fault diagnosis and maintenance far more smoother.

Now, projects are being thoroughly planned. The primary objective of project planning is to ensure that the various development activities take place at the correct time and no activity is halted due to the want of some resource.

Project planning normally includes preparation of various types of estimates, resource scheduling, and development of project tracking plans.

COMPUTER SYSTEMS ENGINEERING



SOFTWARE LIFE CYCLE MODELS (SDLC)

Software life cycle

The term *software life cycle* has been defined to imply the different stages (or phases) over which a software evolves from an initial customer request for it, to a fully developed software, and finally to a stage where it is no longer useful to any user, and then it is discarded.

The stage where the customer feels a need for the software and forms rough ideas about the required features is the *inception* stage.

Once installed and made available for use, the users start to use the software. This signals the start of the operation (also called *maintenance*) phase.

As the users use the software, not only do they request for fixing any failures that they might encounter, but they also continually suggest several improvements and modifications to the software.

Thus, the maintenance phase usually involves continually making changes to the software to accommodate the bug-fix and change requests from the user.

The operation phase is usually the longest of all phases and constitutes the useful life of a software.

Finally the software is retired, when the users do not find it any longer useful due to reasons such as changed business scenario, availability of a new software having improved features and working, changed computing platforms, etc.

The life cycle of a software represents the series of identifiable stages through which it evolves during its life time.

Software development organisations have realised that adherence to a suitable life cycle model helps to produce good quality software and that helps minimise the chances of time and cost overruns.

Why document a development process?

Let us consider that a development organisation does not document its development process.

Developers develop only an informal understanding of the development process.

A documented process model ensures that every activity in the life cycle is accurately defined.

Without documentation, the activities and their ordering tend to be loosely defined, leading to confusion and misinterpretation by different teams in the organisation.

Another difficulty is that for loosely defined activities, the developers tend to use their subjective judgments.

An undocumented process gives a clear indication to the members of the development teams about the lack of seriousness on the part of the management of the organisation about following the process.

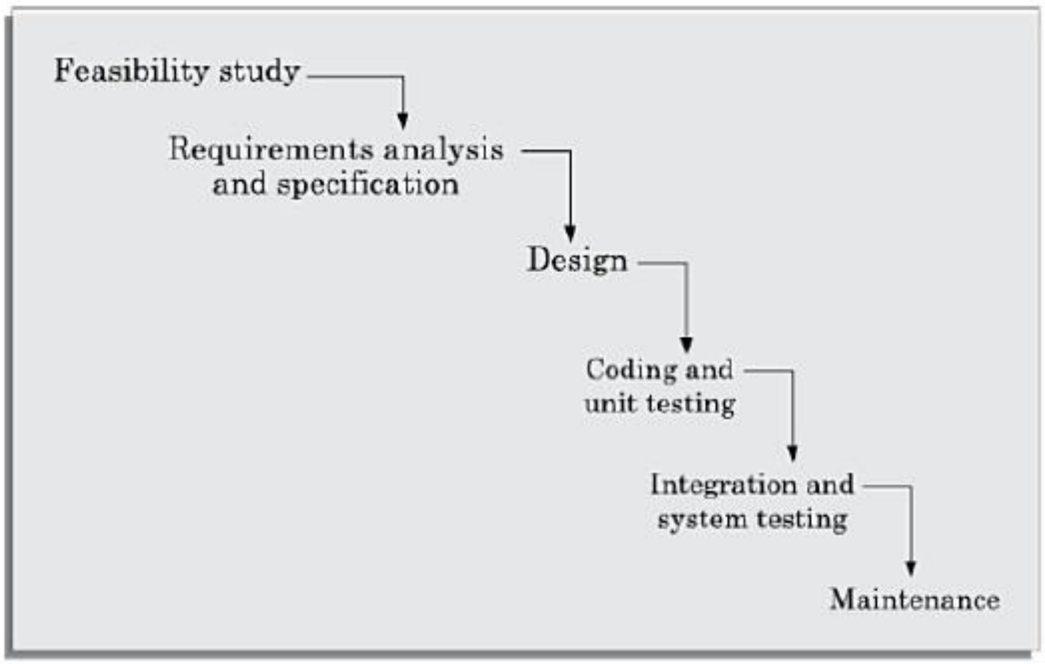
WATERFALL MODEL AND ITS EXTENSIONS

The waterfall model and its derivatives were extremely popular in the 1970s and still are heavily being used across many development projects.

The waterfall model is possibly the most obvious and intuitive way in which software can be developed through team effort.

Classical Waterfall Model

Classical waterfall model is intuitively the most obvious way to develop software.


Feasibility study
Requirements analysis and specification
Design
Coding and unit testing
Integration and system testing
Maintenance

An activity that spans all phases of software development is *project management*. Since it spans the entire project duration, no specific phase is named after it. Project management, nevertheless, is an important activity in the life cycle and deals with managing the software development and maintenance activities.

Feasibility study

The main focus of the feasibility study stage is to determine whether it would be *financially* and *technically feasible* to develop the software.

The feasibility study involves collection of basic information relating to the software such as the different data items that would be input to the system, the processing required to be carried out on these data, the output data required to be produced by the system, as well as various constraints on the development.

Development of an overall understanding of the problem: It is necessary to first develop an overall understanding of what the customer requires to be developed.

Formulation of the various possible strategies for solving the problem: In this activity, various possible high-level solution schemes to the problem are determined. For example, solution in a client-server framework and a standalone application framework may be explored.

Evaluation of the different solution strategies: The different identified solution schemes are analysed to evaluate their benefits and shortcomings.

Such evaluation often requires making approximate estimates of the resources required, cost of development, and development time required.

The different solutions are compared based on the estimations and once the best solution is identified, all activities in the later phases are carried out as per this solution.

At this stage, it may also be determined that none of the solutions is feasible due to high cost, resource constraints, or some technical reasons. This scenario would, of course, require the project to be abandoned.

Go/No-Go

Requirements analysis and specification

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly.

This phase consists of two distinct activities, namely requirements gathering and analysis, and requirements specification.

Requirements gathering and analysis: First, requirements are gathered from the customer and then the gathered requirements are analysed. The goal of the requirements analysis activity is to weed out the incompleteness and inconsistencies in these gathered requirements.

An *inconsistent* requirement is one in which some part of the requirement contradicts with some other part.

An *incomplete* requirement is one in which some parts of the actual requirements have been omitted.

Requirements specification: After the requirement gathering and analysis activities are complete, the identified requirements are documented. This is called a *software requirements specification* (SRS) document. The SRS document should be understandable to the customer.

The SRS document normally serves as a contract between the development team and the customer.

The SRS document is therefore an important document which must be thoroughly understood by the development team, and reviewed jointly with the customer.

The SRS document not only forms the basis for carrying out all the development activities, but several documents such as users' manuals, system test plan, etc. are prepared directly based on it.

Design

The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language.

In technical terms, during the design phase the *software architecture* is derived from the SRS document.

Two distinctly different design approaches are popularly being used at present—the functional/procedural and object-oriented design approaches.

Coding and Testing

Verification (SRS) and Validation (Test Case) difference

Verification process includes checking of documents, design, code and program whereas Validation process includes testing and validation of the actual product.

- Verification does not involve code execution while Validation involves code execution.

- Verification uses methods like reviews, walkthroughs, inspections and desk-checking whereas Validation uses methods like black box testing, white box testing and non-functional testing.
- Verification checks whether the software confirms a specification whereas Validation checks whether the software meets the requirements and expectations.
- Verification finds the bugs early in the development cycle whereas Validation finds the bugs that verification cannot catch.
- Verification process targets on software architecture, design, database, etc. while Validation process targets the actual software product.
- Verification is done by the QA team while Validation is done by the involvement of testing team with QA team.
- Verification process comes before validation.

Unit testing, Integration testing are carried out to verify that the interfaces among different units are working satisfactorily. On the other hand, the goal of system testing is to ensure that the developed system conforms to the requirements that have been laid out in the SRS document.

Shortcomings of the classical waterfall model

No feedback paths: In classical waterfall model, the evolution of a software from one phase to the next is analogous to a waterfall.

Once a phase is complete, the activities carried out in it and any artifacts produced in this phase are considered to be final and are closed for any rework.

Incorporates no mechanism for error correction.

Difficult to accommodate change requests: This model assumes that all customer requirements can be completely and correctly defined at the beginning of the project.

There is much emphasis on creating an unambiguous and complete set of requirements. But, customers' requirements usually keep on changing with time.

Inefficient error corrections: This model defers integration of code and testing tasks until it is very late when the problems are harder to resolve.

No overlapping of phases: This model recommends that the phases be carried out sequentially—new phase can start only after the previous one completes.

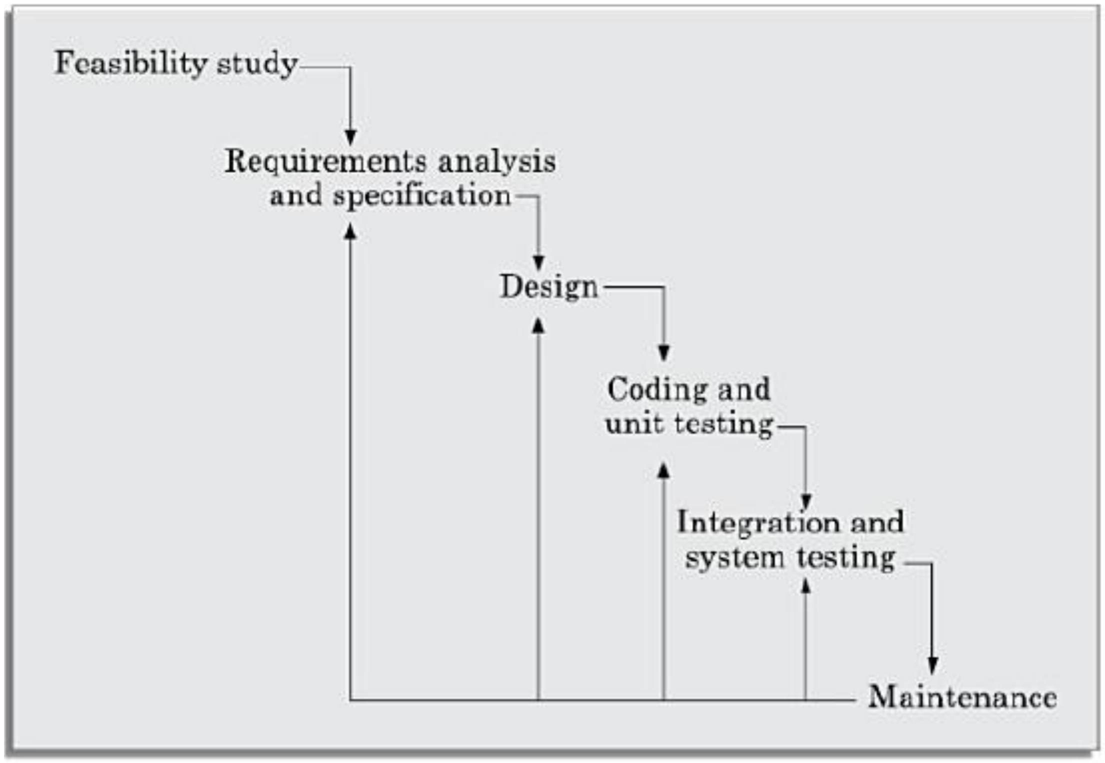
This leads to a large number of team members to idle for extended periods.

Irrespective of the life cycle model that is actually followed for a product development, the final documents are always written to reflect a classical waterfall model of development, so that comprehension of the documents becomes easier for anyone reading the document.

Iterative Waterfall Model

The iterative waterfall model can be thought of as incorporating the necessary changes to the classical waterfall model to make it usable in practical software development projects.

The main change brought about by the iterative waterfall model to the classical waterfall model is in the form of providing feedback paths from every phase to its preceding phases.


Feasibility study
Requirements analysis and specification
Design
Coding and unit testing
Integration and system testing
Maintenance

The principle of detecting errors as close to their points of commitment as possible is known as *phase containment of errors*.

Shortcomings of the iterative waterfall model

Difficult to accommodate change requests: A major problem with the waterfall model is that the requirements need to be frozen before the development starts.

Incremental delivery not supported: In the iterative waterfall model, the full software is completely developed and tested before it is delivered to the customer.

Phase overlap not supported: For most real life projects, it becomes difficult to follow the rigid phase sequence prescribed by the waterfall model.

By the term *a rigid phase sequence*, we mean that a phase can start only after the previous phase is complete in all respects.

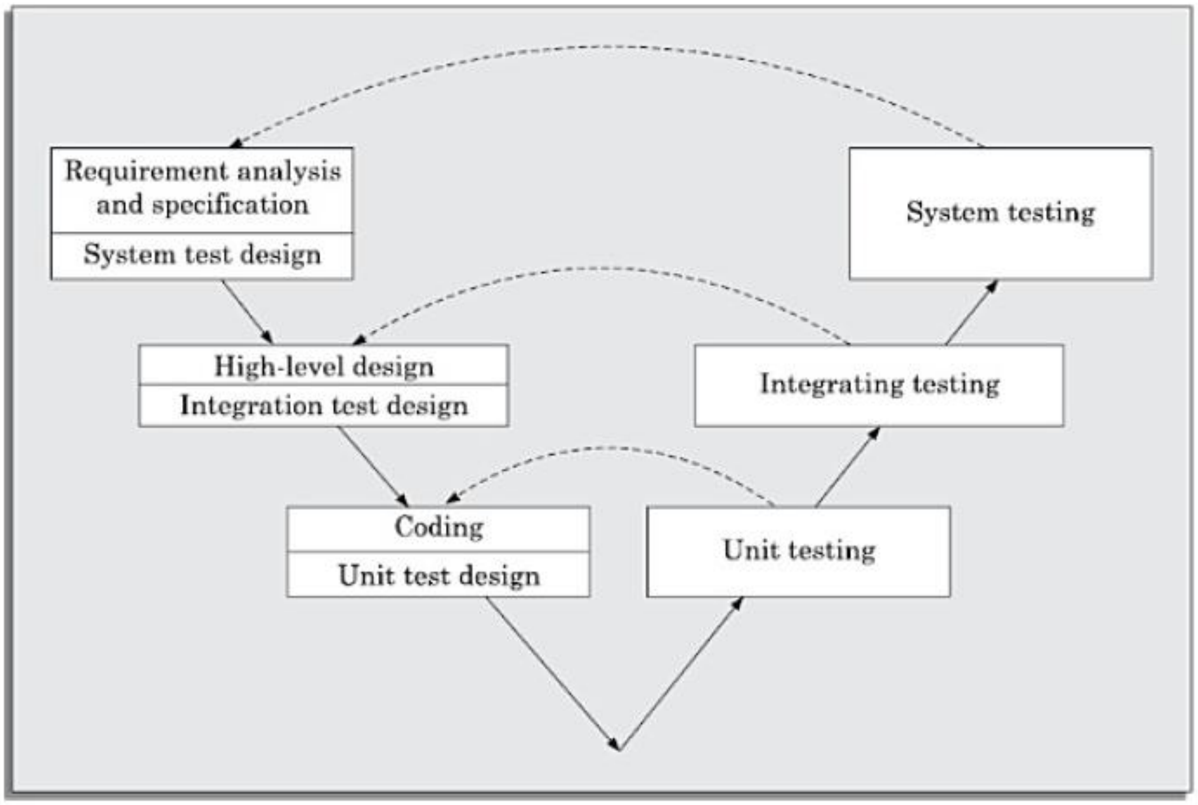
Error correction unduly expensive: The defects that are noticed at the time of validation incur expensive rework and result in cost escalation and delayed delivery.

Limited customer interactions: This model supports very limited customer interactions. It is generally accepted that software developed in isolation from the customer is the cause of many problems.

Heavy weight: The waterfall model over-emphasises documentation.

No support for risk handling and code reuse: It becomes difficult to use the waterfall model in projects that are susceptible to various types of risks, or those involving significant reuse of existing development artifacts.

V-Model



There are two main phases—development and validation phases. The left half of the model comprises the development phases and the right half comprises the validation phases.

Advantages of V-model

Much o f the testing activities (test case design, test planning, etc.) are carried out in parallel with the development activities.

This model usually leads to a shorter testing phase and an overall faster product development as compared to the iterative model.

Since test cases are designed when the schedule pressure has not built up, the quality of the test cases are usually better.

The test team is reasonably kept occupied throughout the development cycle in contrast to the waterfall model where the testers are active only during the testing phase. This leads to more efficient manpower utilisation.

In the V-model, the test team is associated with the project from the beginning. Therefore they build up a good understanding of the development artifacts, and this in turn, helps them to carry out effective testing of the software.

Disadvantages of V-model

Being a derivative of the classical waterfall model, this model inherits most of the weaknesses of the waterfall model.

Prototyping Model

The prototyping model can be considered to be an extension of the waterfall model.

This model suggests building a working *prototype* of the system, before development of the actual software.

It has limited functional capabilities, low reliability, or inefficient performance as compared to the actual software. A prototype can be built very quickly by using several shortcuts.

The shortcut implementation of a function, for example, may produce the desired results by using a table look-up rather than by performing the actual computations.

Necessity of the prototyping model

The prototyping model is advantageous to use for specific types of projects.

Development of the *graphical user interface* (GUI) part of an application. It becomes easier to illustrate the input data formats, messages, reports, and the interactive dialogs to the customer.

The prototyping model is especially useful when the exact technical solutions are unclear to the development team.

The prototyping model can be deployed when development of highly optimised and efficient software is required.

Life cycle activities of prototyping model

Prototype development: Prototype development starts with an initial requirements gathering phase. A quick design is carried out and a prototype is built.

The developed prototype is submitted to the customer for evaluation.

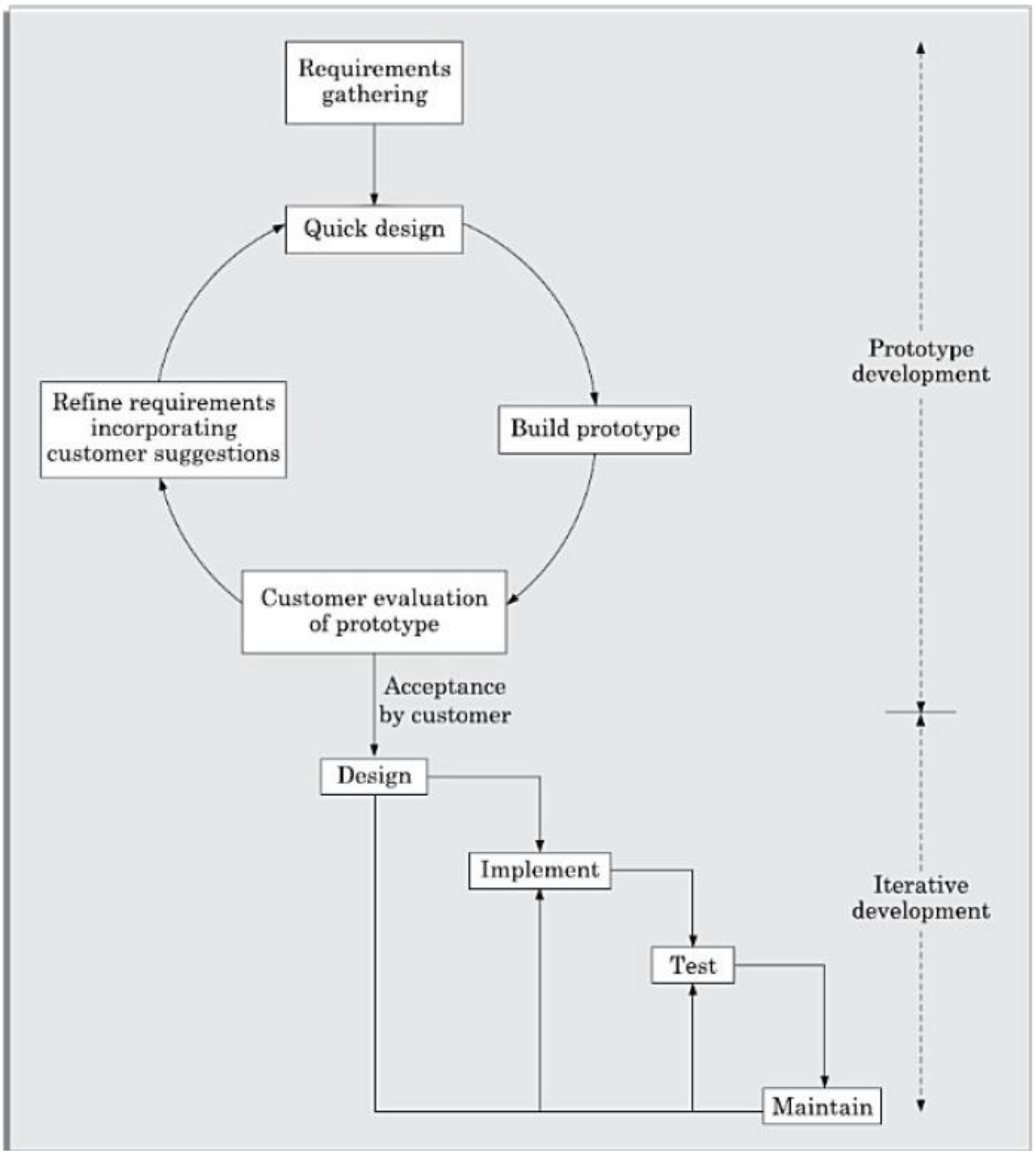
Based on the customer feedback, the requirements are refined and the prototype is suitably modified. This cycle of obtaining customer feedback and modifying the prototype continues till the customer approves the prototype.

Iterative development: Once the customer approves the prototype, the actual software is developed using the iterative waterfall approach.

The SRS document is usually needed to be developed since the SRS document is invaluable for carrying out traceability analysis, verification, and test case design during later phases.

However, for GUI parts, the requirements analysis and specification phase becomes redundant since the working prototype that has been approved by the customer serves as an animated requirements specification.

The code for the prototype is usually thrown away. However, the experience gathered from developing the prototype helps a great deal in developing the actual system.



Even though the construction of a throwaway prototype might involve incurring additional cost, for systems with unclear customer requirements and for systems with unresolved technical issues, the overall development cost usually turns out to be lower compared to an equivalent system developed using the iterative waterfall model.

Strengths of the prototyping model

This model is the most appropriate for projects that suffer from technical and requirements risks. A constructed prototype helps overcome these risks.

Weaknesses of the prototyping model

The prototype model can increase the cost of development for projects that are routine development work and do not suffer from any significant risks.

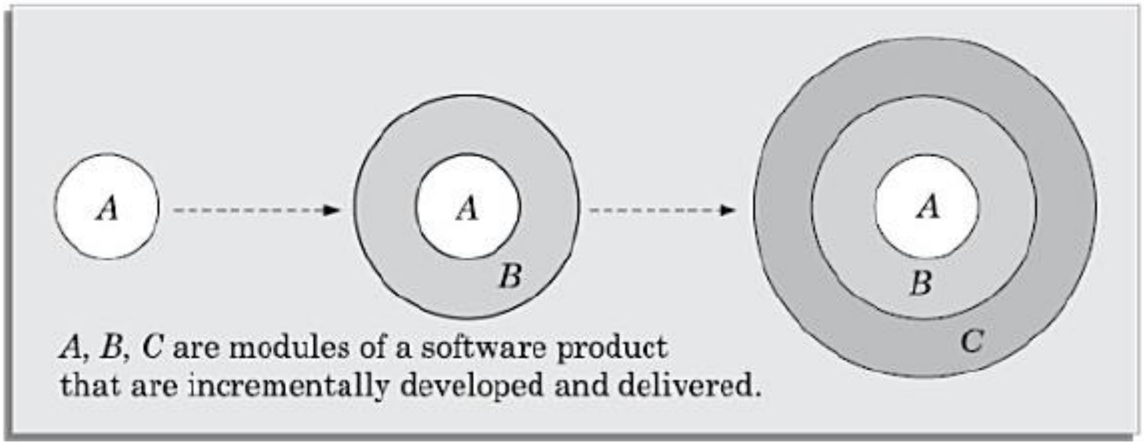
Even when a project is susceptible to risks, the prototyping model is effective only for those projects for which the risks can be identified upfront before the development starts.

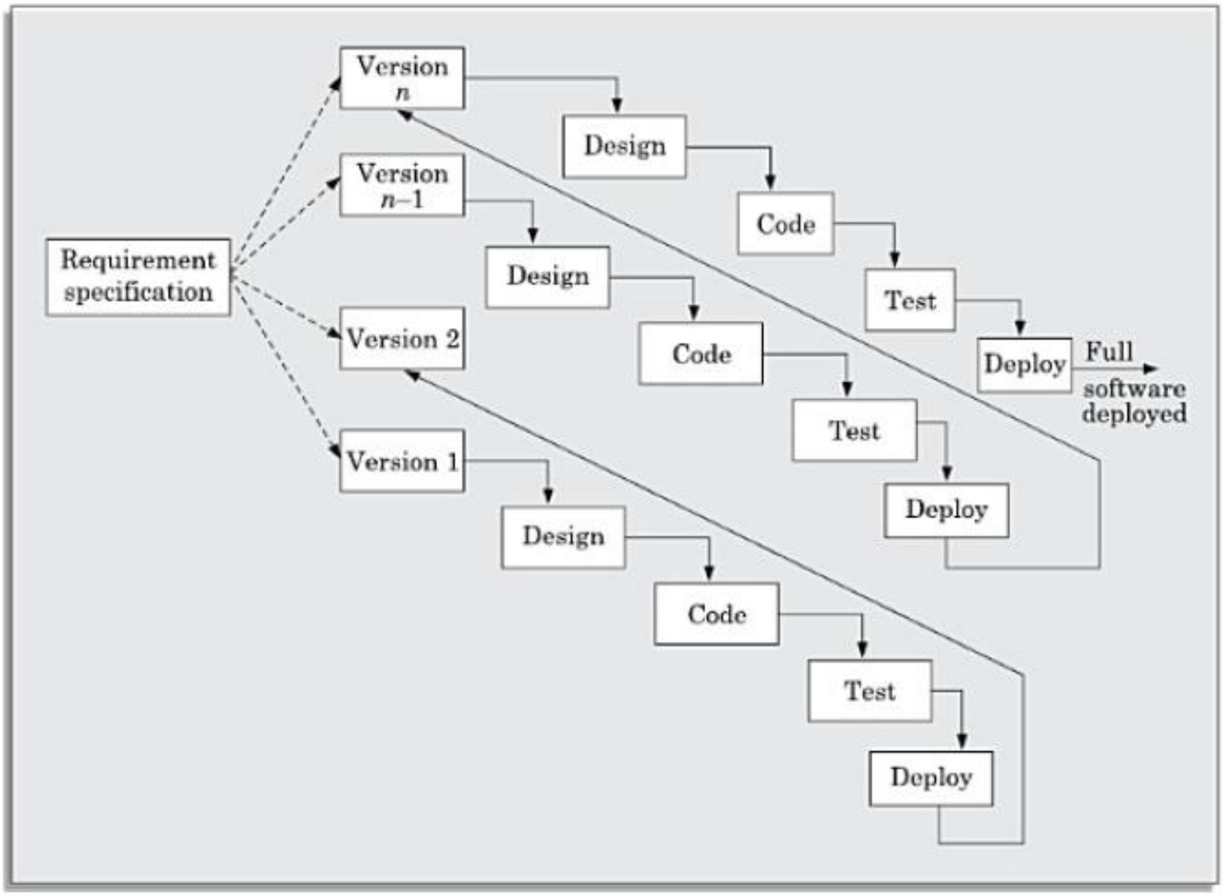
Since the prototype is constructed only at the start of the project, the prototyping model is ineffective for risks identified later during the development cycle.

Incremental Development Model

This life cycle model is sometimes referred to as the *successive versions* model and sometimes as the incremental model. In this life cycle model, first a simple working system implementing only a few basic features is built and delivered to the customer.

Over many successive iterations successive versions are implemented and delivered to the customer until the desired system is realised.


A
A
B
A
B
C
A, B, C are modules of a software product that are incrementally developed and delivered.


Requirement specification
Version n
Version n–1
Version 2
Version 1
Design
Code
Test
Deploy
Design
Code
Test
Deploy
Design
Code
Test
Deploy
Full software deployed

Advantages

Error reduction: The core modules are used by the customer from the beginning and therefore these get tested thoroughly. This reduces chances of errors in the core modules of the final product, leading to greater reliability of the software.

Incremental resource deployment: This model obviates the need for the customer to commit large resources at one go for development of the system. It also saves the developing organisation from deploying large resources and manpower for a project in one go.

Evolutionary Model

As in case of the incremental model, the software is developed over a number of increments.

At each increment, a concept (feature) is implemented and is deployed at the client site. The software is successively refined and feature-enriched until the full software is realised.

The principal idea behind the evolutionary life cycle model is conveyed by its name.

In the incremental development model, complete requirements are first developed and the SRS document prepared. In contrast, in the evolutionary model, the requirements, plan, estimates, and solution evolve over the iterations.

Such evolution is consistent with the pattern of unpredictable feature discovery and feature changes that take place in new product development.

Though the evolutionary model can also be viewed as an extension of the waterfall model, but it incorporates a major paradigm shift that has been widely adopted in many recent life cycle models.

The evolutionary software development process is sometimes referred to as *design a little, build a little, test a little, deploy a little* model.

Advantages

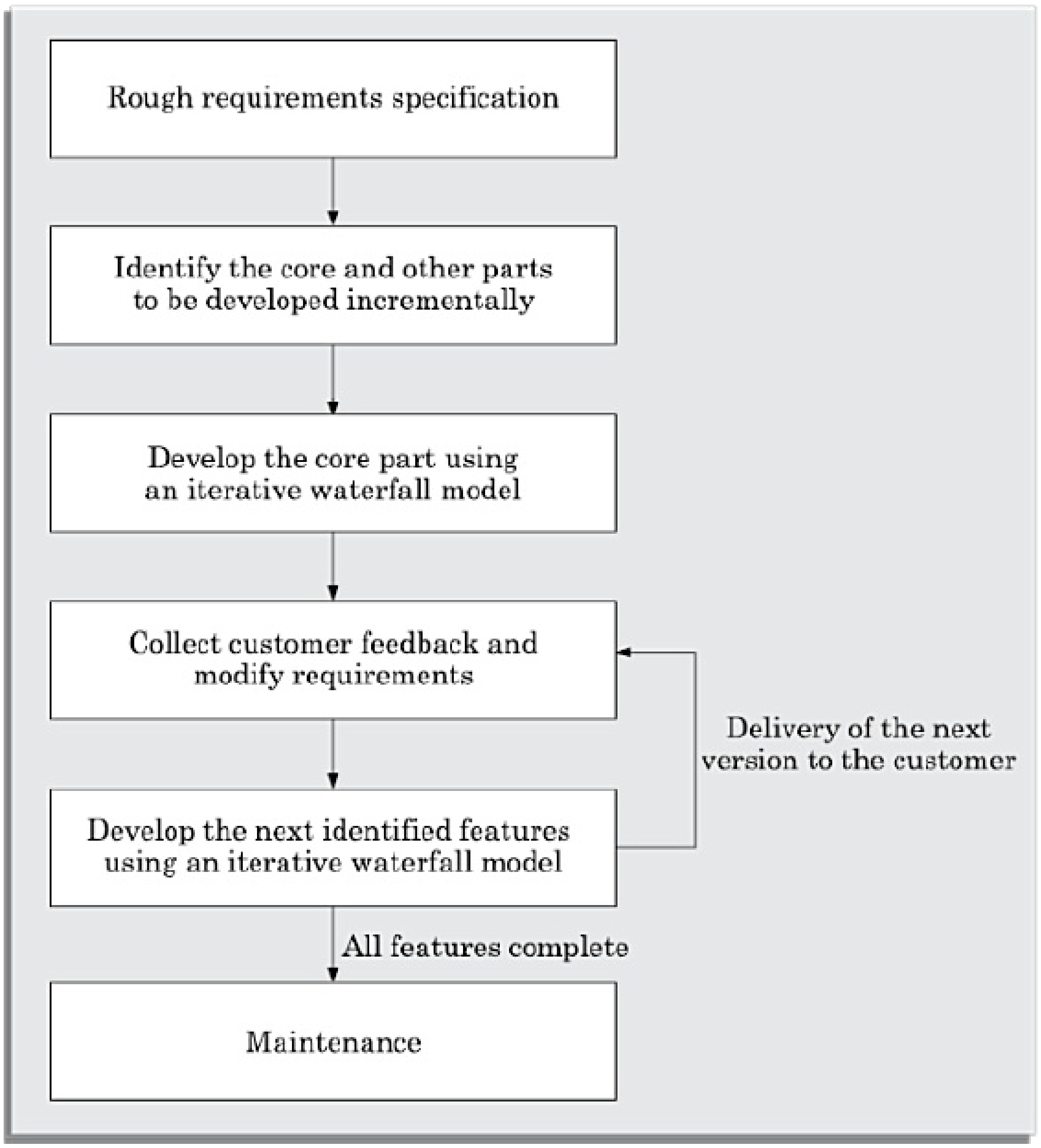
Effective elicitation of actual customer requirements: In this model, the user gets a chance to experiment with a partially developed software much before the complete requirements are developed.

Therefore, the evolutionary model helps to accurately elicit user requirements with the help of feedback obtained on the delivery of different versions of the software.

Easy handling change requests: In this model, handling change requests is easier as no long term plans are made. Consequently, reworks required due to change requests are normally much smaller compared to the sequential models.

Disadvantages

The evolutionary model is well-suited to use in object-oriented software development projects.


Rough requirements specification
Identify the core and other parts to be developed incrementally
Develop the core part using an iterative waterfall model
Collect customer feedback and modify requirements
Delivery of the next version to the customer
Develop the next identified features using an iterative waterfall model
All features complete
Maintenance

RAPID APPLICATION DEVELOPMENT (RAD)

The *rapid application development* (RAD) model was proposed in the early nineties in an attempt to overcome the rigidity of the waterfall model (and its derivatives) that makes it difficult to accommodate any change requests from the customer.

This model has the features of both prototyping and evolutionary models.

It deploys an evolutionary delivery model to obtain and incorporate the customer feedbacks on incrementally delivered versions.
In this model prototypes are constructed, and incrementally the features are developed and delivered to the customer.

But unlike the prototyping model, the prototypes are not thrown away but are enhanced and used in the software construction

The major goals of the RAD model are as follows:

To decrease the time taken and the cost incurred to develop software systems.

To limit the costs of accommodating change requests.

To reduce the communication gap between the customer and the developers.

Working of RAD

In the RAD model, development takes place in a series of short cycles or iterations. At any time, the development team focuses on the present iteration only, and therefore plans are made for one increment at a time.

The time planned for each iteration is called a *time box*. Each iteration is planned to enhance the implemented functionality of the application by only a small amount.

During each time box, a quick-and-dirty prototype-style software for some functionality is developed. The customer evaluates the prototype and gives feedback on the specific improvements that may be necessary.The prototype is refined based on the customer feedback.

Please note that the prototype is not meant to be released
to the customer for regular use though.

The development team almost always includes a customer representative to clarify the requirements.

The development team usually consists of about five to six members, including a customer representative.

RAD model emphasises code reuse as an important means for completing a project faster.

The adopters of the RAD model were the earliest to embrace object-oriented languages and practices.

RAD advocates use of specialised tools to facilitate fast creation of working prototypes. These specialised tools usually support Visual style of development and Use of reusable components.

Applicability of RAD Model

Customised software: As already pointed out a customised software is developed for one or two customers only by adapting an existing software.

In customised software development projects, substantial reuse is usually made of code from pre-existing software.

Non-critical software: The RAD model suggests that a quick and dirty software should first be developed and later this should be refined into the final software for delivery.

Therefore, the developed product is usually far from being optimal in performance and reliability. In this regard, for well understood development projects and where the scope of reuse is rather restricted, the Iterative waterfall model may provide
a better solution.

Highly constrained project schedule: RAD aims to reduce development time at the expense of good documentation, performance, and reliability. Naturally, for projects with very aggressive time schedules, RAD model should be preferred.

Large software: Only for software supporting many features (large software) can incremental development and delivery be meaningfully carried out.

Application characteristics that render RAD unsuitable

Generic products (wide distribution): Software products are generic in nature and usually have wide distribution. For such systems, optimal performance and reliability are imperative in a competitive market.

Requirement of optimal performance and/or reliability: For certain categories of products, optimal performance or reliability is required. Examples of such systems include an operating system (high reliability required) and a flight simulator software (high performance required).

Lack of similar products: If a company has not developed similar software, then it would hardly be able to reuse much of the existing artifacts.

RAD ***versus*** prototyping model

In the prototyping model, the developed prototype is primarily used by the development team to gain insights into the problem, choose between alternatives, and elicit customer feedback. The code developed during prototype construction is usually thrown away.

In contrast, in RAD it is the developed prototype that evolves into the deliverable software.

Though RAD is expected to lead to faster software development compared to the traditional models (such as the prototyping model), the quality and reliability would be inferior.

RAD ***versus*** iterative waterfall model

In the iterative waterfall model, all the functionalities of a software are developed together. On the other hand, in the RAD model the product functionalities are developed incrementally through heavy code and design reuse.

Further, in the RAD model customer feedback is obtained
on the developed prototype after each iteration and based on this the prototype is refined. The iterative waterfall model does not support any mechanism to accommodate any requirement change requests.

The iterative waterfall model does have some important advantages:

Use of the iterative waterfall model leads to production of good quality documentation which can help during software maintenance.

Also, the developed software usually has better quality and reliability.

RAD ***versus*** evolutionary model

Incremental development is the hallmark of both evolutionary and RAD models. However, in RAD each increment results in essentially a quick and dirty prototype, whereas in the evolutionary model each increment is systematically developed using the iterative waterfall model.

Also in the RAD model, software is developed in much shorter increments compared the evolutionary model. In other words, the incremental functionalities that are developed are of fairly larger granularity in the evolutionary model.

AGILE DEVELOPMENT MODELS

As already pointed out, though the iterative waterfall model has been very popular during the 1970s and 1980s, developers face several problems while using it on present day software projects.

The main difficulties included handling change requests from customers during product development, and the unreasonably high cost and time that is incurred while developing customised applications.

Research involving 800 real-life software development projects concluded that on the average 40 per cent of the requirements is arrived after the development has already begun.

Over the last two decade or so, several life cycle models have been proposed to overcome the important shortcomings of the waterfall based models that become conspicuous when used in modern software development projects.

Two changes that are becoming noticeable are rapid shift from development of software products to development of customised software and the increased emphasis and scope for reuse.

The agile software development model was proposed in the mid-1990s to overcome the serious shortcomings of the waterfall model of development.

The agile model was primarily designed to help a project to adapt to change requests quickly.

A few popular agile SDLC models are the following:

- Crystal
- Atern (formerly DSDM)
- Feature-driven development
- Scrum
- Extreme programming (XP)
- Lean development
- Unified process

In the agile model, the requirements are decomposed into many small parts that can be incrementally developed. The agile model adopts an iterative approach.

Each incremental part is developed over an iteration.

Each iteration is intended to be small and easily manageable and lasting for a couple of weeks only. At a time, only one increment is planned, developed, and then deployed at the customer site.

The time to complete an iteration is called a *time box*. The implication of the term *time box* is that the end date for an iteration does not change. That is, the delivery date is considered sacrosanct.

The development team can, however, decide to reduce the delivered functionality during a time box if necessary.

A central principle of the agile model is the delivery of an increment to the customer after each time box.

- Working software over comprehensive documentation.

- Frequent delivery of incremental versions of the software to the customer in intervals of few weeks.

- Requirement change requests from the customer are encouraged and efficiently incorporated.

- Having competent team members and enhancing interactions among them is considered much more important than issues such as usage of sophisticated tools or strict adherence to a documented process.

- It is advocated that enhanced communication among the development team members can be realised through face-to-face communication rather than through exchange of formal documents.

- Continuous interaction with the customer is considered much more important rather than effective contract negotiation. A customer representative is required to be a part of the development team, thus facilitating close, daily co-operation between customers and developers.

- Agile development projects usually deploy pair programming.

  In pair programming, two programmers work together at one work station. One types in code while the other reviews the code as it is typed in. The two programmers switch their roles every hour or so.

Disadvantages of agile methods

Lack of formal documents leaves scope for confusion and important decisions taken during different phases can be misinterpreted at later points of time by different team members.

In the absence of any formal documents, it becomes difficult to get important project decisions such as design decisions to be reviewed by external experts.

When the project completes and the developers disperse, maintenance can become a problem.

Agile ***versus*** Other Models

Agile model ***versus*** iterative waterfall model

The waterfall model is highly structured and systematically steps through requirements-capture, analysis, specification, design, coding, and testing stages in a planned sequence.

Progress is generally measured in terms of the number of completed and reviewed artifacts such as requirement specifications, design documents, test plans, code reviews, etc.

In contrast, while using an agile model, progress is measured in terms of the developed and delivered functionalities.

In agile model, delivery of working versions of a software is made in several increments.

As regards to similarity it can be said that agile teams use the waterfall model on a small scale, repeating the entire waterfall cycle in every iteration.

Agile ***versus*** exploratory programming

Agile development model's frequent re-evaluation of plans, emphasis on face-to-face communication, and relatively sparse use of documentation are similar to that of the exploratory style.

Agile teams, however, do follow defined and disciplined processes and carry out systematic requirements capture, rigorous designs, compared to chaotic coding in exploratory programming.

Agile model ***versus*** RAD model

Agile model does not recommend developing prototypes, but emphasises systematic development of each incremental feature.

In contrast, the central theme of RAD is based on designing quick-and dirty prototypes, which are then refined into production quality code.

Agile projects logically break down the solution into features that are incrementally developed and delivered.

The RAD approach does not recommend this. Instead, developers using the RAD model focus on developing all the features of an application by first doing it badly and then successively improving the code over time.

Agile teams only demonstrate completed work to the customer. In contrast, RAD teams demonstrate to customers screen mock ups, and prototypes, that may be based on simplifications such as table look-ups rather than actual computations.

Extreme Programming Model

Extreme programming (XP) is an important process model under the agile umbrella and was proposed by Kent Beck in 1999.

The name of this model reflects the fact that it recommends taking the *best practices* that have worked well in the past in program development projects to extreme levels.

Good practices that need to be practised to the extreme

Code review: It is good since it helps detect and correct problems most efficiently. It suggests *pair programming* as the way to achieve continuous review.

Testing: Testing code helps to remove bugs and improves its reliability. XP suggests *test-driven development* (TDD) to continually write and execute test cases. In the TDD approach, test cases are written even before any code is written.

Incremental development: Incremental development is good, since it helps to get customer feedback, and extent of features delivered is a reliable indicator of progress. It suggests that the team should come up with new increments every few days.

Simplicity: Simplicity makes it easier to develop good quality code, as well as to test and debug it. Once the simplest thing works, other aspects can be introduced through refactoring.

Design: Since having a good quality design is important to develop a good quality solution, everybody should design daily. This can be achieved through *refactoring*, whereby a working code is improved for efficiency and maintainability.

Integration testing: It is important since it helps identify the bugs at the interfaces of different functionalities. To this end, extreme programming suggests that the developers should achieve continuous integration, by building and performing integration testing several times a day.

Basic idea of extreme programming model

XP is based on frequent releases (called *iteration*), during which the developers implement "user stories". User stories are similar to use cases, but are more informal and are simpler.

A user story is the conversational description by the user about a feature or functionality of the required system. For example, a user story about a library software can be:

A library member can issue a book.
A library member can query about the availability of a book.
A library member should be able to return a borrowed book.

A user story does not mention about finer details such as the different scenarios that can occur, the precondition (state at which the system) to be satisfied before the feature can be invoked, etc.

On the basis of user stories, the project team proposes “metaphors”—a common vision of how the system would work. The development team may decide to construct a *spike* for some feature.

A *spike*, is a very simple program that is constructed to explore the suitability of a solution being proposed. A spike can be considered to be similar to a prototype.

XP prescribes several basic activities to be part of the software development process.

Coding: XP argues that code is the crucial part of any system development process, since without code it is not possible to have a working system.

However, the concept of code as used in XP has a slightly different meaning from what is traditionally understood. For example, coding activity includes drawing diagrams (modelling) that will be transformed to code, scripting a web-based system, and choosing among several alternative solutions.

Testing: XP places high importance on testing and considers it be the primary means for developing a fault-free software.

Listening: The developers need to carefully listen to the customers if they have to develop a good quality software. Programmers may not necessarily be having an in-depth knowledge of the the specific domain of the system under development. On the other hand, customers usually have this domain knowledge.

Designing: A good design should result in elimination of complex dependencies within a system. Thus, effective use of a suitable design technique is emphasised.

Feedback: It espouses the wisdom: “A system staying out of users is trouble waiting to happen”. It recognises the importance of user feedback in understanding the exact customer requirements. The time that elapses between the development of a version and collection of feedback on it is critical to learning and making changes.

Simplicity: A corner-stone of XP is based on the principle: “build something simple that will work today, rather than trying to build something that would take time and yet may never be used”.

XP is in favour of making the solution to a problem as simple as possible. In contrast, the traditional system development methods recommend planning for reusability and future extensibility of code and design at the expense of higher code and design complexity.

Applicability of extreme programming model

Projects involving new technology or research projects: In this case, the requirements change rapidly and unforeseen technical problems need to be resolved.

Small projects: Extreme programming was proposed in the context of small teams as face to face meeting is easier to achieve.

Project characteristics not suited to development using agile models

Stable requirements: Conventional development models are more suited to use in projects characterised by stable requirements.

Therefore, process models such as iterative waterfall model that involve making long-term plans during project initiation can meaningfully be used.

Mission critical or safety critical systems: In the development of such systems, the traditional SDLC models are usually preferred to ensure reliability.

Scrum Model

In the scrum model, a project is divided into small parts of work that can be incrementally developed and delivered over time boxes that are called *sprints*.

The software therefore gets developed over a series of manageable chunks. Each sprint typically takes only a couple of weeks to complete.

At the end of each sprint, stakeholders and team members meet to assess the progress made and the stakeholders suggest to the development team any changes needed to features that have already been developed and any overall improvements that they might feel necessary.

In the scrum model, the team members assume three fundamental roles—
software owner, scrum master, and team member.

The software owner is responsible for communicating the customers vision of the software to the development team.

The scrum master acts as a liaison between the software owner and the team, thereby facilitating the development work.

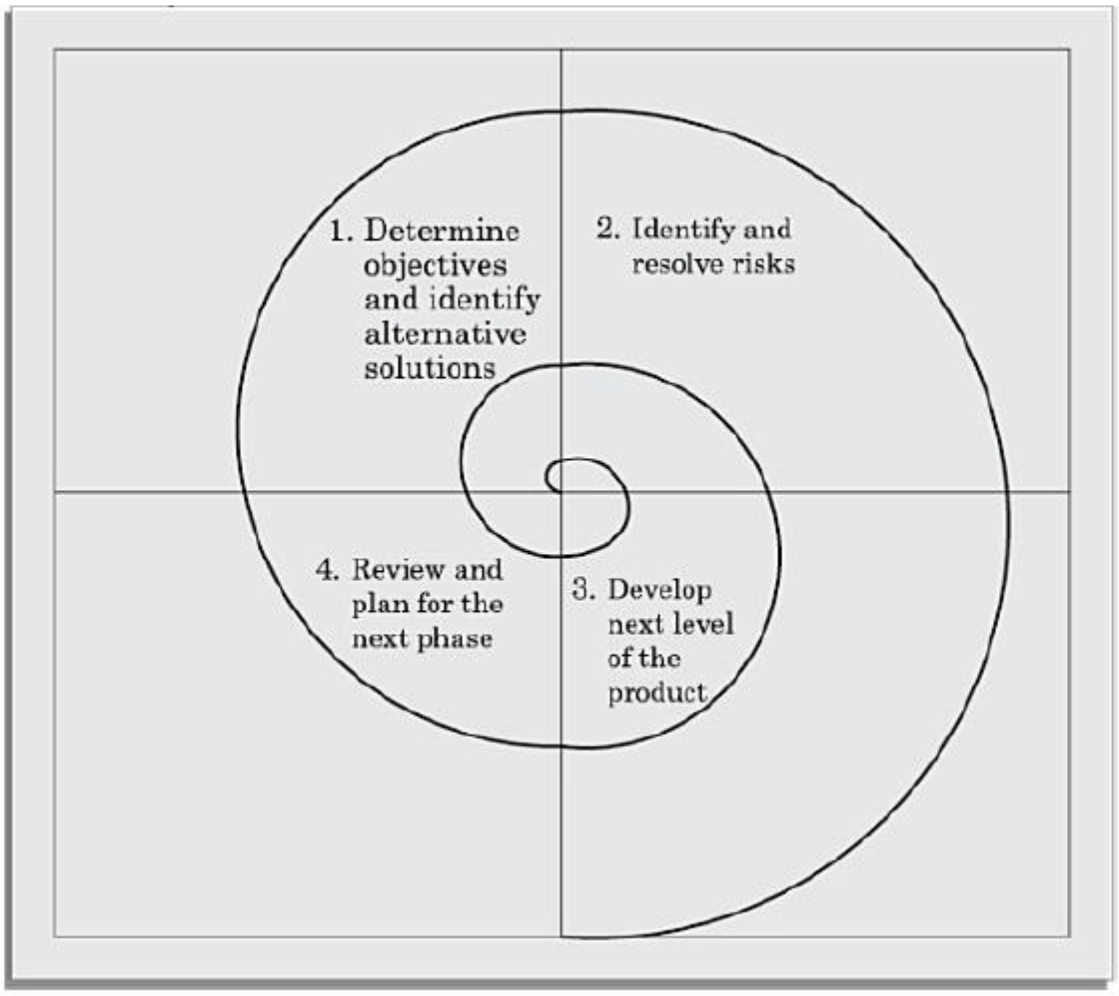
SPIRAL MODEL

The exact number of loops of the spiral is not fixed and can vary from project to project.

Each loop of the spiral is called a *phase* of the software process. The exact number of phases through which the product is developed can be varied by the project manager depending upon the project risks.

A prominent feature of the spiral model is handling unforeseen risks that can show up much after the project has started.

This model incorporates much more flexibility compared to SDLC.


1. Determine objectives and identify alternative solutions
2. Identify and resolve risks
3. Develop next level of the product
4. Review and plan for the next phase

Phases of the Spiral Model

In the first quadrant, a few features of the software are identified to be taken up for immediate development based on how crucial it is to the overall software development.

With each iteration around the spiral (beginning at the center and moving outwards), progressively more complete versions of the software get built. In other words, implementation of the identified features forms a phase.

Quadrant 1: The objectives are investigated, elaborated, and analysed. Based on this, the risks involved in meeting the phase objectives are identified. In this quadrant, alternative solutions possible for the phase under consideration are proposed.

Quadrant 2: During the second quadrant, the alternative solutions are evaluated to select the best possible solution. To be able to do this, the solutions are evaluated by developing an appropriate prototype.

Quadrant 3: Activities during the third quadrant consist of developing and verifying the next level of the software. At the end of the third quadrant, the identified features have been implemented and the next version of the software is available.

Quadrant 4: Activities during the fourth quadrant concern reviewing the results of the stages traversed so far (i.e. the developed version of the software) with the customer and planning the next iteration of the spiral.

The radius of the spiral at any point represents the cost incurred in the project so far, and the angular dimension represents the progress made so far in the current phase.

The project manager plays a crucial role in tuning this model.

Advantages and disadvantages of the Spiral Model

There are a few disadvantages of the spiral model that restrict its use to a only a few types of projects.

To the developers of a project, the spiral model usually appears as a complex model to follow, since it is risk driven and is a more complicated structure than the other models discussed. It would therefore be counterproductive to use, unless there are knowledgeable and experienced staff in the project.

Also, it is not very suitable for use in the development of outsourced projects, since the software risks need to be continually assessed as it is developed.

However, the advantages of the spiral model can outweigh its disadvantages.

For projects having many unknown risks that might show up as the development proceeds, the spiral model would be the most appropriate development model to follow.