Unit 1

Introduction to S.E.

Software Crisis

It was in late 1960’s

•Many software projects failed.

• Many software projects late, over budget, providing unreliable software that is expensive to maintain.

• Many software projects produced software which did not satisfy the requirements of the customer.

•Complexities of software projects increased as hardware capability increased.

•Larger software system is more difficult and expensive to maintain.

•Demand of new software increased faster than ability to generate new software.

All the above attributes of what was called a ‘Software Crisis’. So the term ‘Software Engineering’ first introduced at a conference in late 1960’s to discuss the software crisis.

Why software engineering?

Once the need for software engineering was identified and software engineering recognized as a discipline--

- The late 1970’s saw the widespread evolution of software engineering principles.
- The 1980’s saw the automation of software engineering and growth of CASE (Computer Aided Software Engineering).
- The 1990’s have seen increased emphasis on the ‘management’ aspects of projects and the use of standard quality and ‘process’ models like ISO 9001 and the Software Engineering Institute’s Software Capability Maturity Model (CMM).

These models help organizations put their software development and management processes in place

What is software Engineering?

- In1969 Fritz Bauer defined software eng. as, ‘the establishment and use of sound engineering principles in order to obtain, economically, software that is reliable and works efficiently on real machines’.

- According to Boehm, software engineering involves, ‘the practical application of scientific knowledge to the design and construction of computer programs and the associated documentation required developing, operating and maintaining them’

- IEEE, in its standard 610.12-1990, defines software engineering as:

(i) The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.

(ii) The study of approaches as in (i).

- By combining all the above definition we can define software engineering as, ‘Software engineering is the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates.’

Goal of software engineering

The primary goals of software engineering are:

- To improve the quality of the software products.
- To increase the productivity &
- To give job satisfaction to the software engineers.

Foundation of Software engineering

Software engineering is a technological discipline distinct from, but based on the foundation of the following disciplines:

- Computer Science
- Management Science
- Economics
- System Engineering &
- Communication Skills

The relationship of software engineering with other disciplines

- **Computer Science** gives the scientific foundation to the software as electrical engineering relies on physics.
- **Management Science** provides the foundation for software project management. Since software engineering is labor intensive activity, it requires both technical and managerial control.
- **Economics** provides the foundation for resource estimation and cost control. Since, computing system must be developed and maintained on time and within cost estimates; thus economics plays an important role.
- **System Engineering** is the field concerned with studying complex systems. Software is often a component of a much larger system. For example, the software in a factory monitoring system or the flight software on an airplane; is just the component of more complex system. System engineering techniques can be applied to study of such systems
- Good oral, written and interpersonal **communication skills** are crucial for the software engineers, because software engineering activities occur within an organizational context, and a high degree of communication is required among customers, managers, software engineers, hardware engineers and other technical workers.

Difference of s/w eng. with traditional engineering

- Software is intangible. It has no mass, no volume, no color, no odor--- no physical properties. Source code is merely a static image of computer program, and while the effects produced by a program are often observable, the program itself not.
- Software doesn’t degrade with time as hardware does. Software failures are caused by design and implementation error, not by degradation over time
- There is always an obscurity in the interface between software modules. It is difficult to design a software system so that all the control and interfaces among modules are explicit, and so that the modules do not interact to produce unexpected side effects when they invoked one another.

…. difference

- In classical engineering disciplines, the engineer is equipped with tools and the mathematical maturity to specify the properties of the product separately from those of design.
- The typical software engineering relies much more on experience and judgment rather than mathematical formula. While experience and judgment are necessary, formal analysis are also essential in the practice of engineering.

The role of software engineer

The evolution of software engineering field has defined the role of the software engineer. A software engineer should have the following qualities:

- Should be a good programmer, be well-versed in data structures and algorithms, and be fluent in one or more programming languages.
- Should be familiar with several design approaches, be able to translate vague requirements and desires into precise specifications and be able to converse with the use of a system in terms of applications.
- Needs the ability to move among several levels of abstraction at different stages of the project, from specific application procedures and requirements, to abstraction for software systems, to a specific design for system and finally to the detailed coding level.

The characteristics of software engineer

- Should be able to build and use a model of the application to guide choices of the many trade-offs that he or she will face. The model is used to answer questions about both the behavior of the system and its performance.

- Needs communication skills and interpersonal skills. He also needs the ability to schedule work both of his own and that of others.

What is well engineered software?

If the software system does what the user wants, and can be made to continue to do what the user wants, it is well engineered.

- Any well engineered software system should have the following attributes:
- Be easy to maintain
- Be reliable
- Be efficient
- Provides an appropriate user interface

The development of software must make trade-offs between these attributes.

Distribution of software effort

The typical life-span for a typical software product is 1 to 3 years in development and 5 to 15 years in use. The distribution of effort between development and maintenance has been variously reported depending on the type of software as 40/60, 30/70 and 10/90.

Maintenance:

- **Corrective**: Even with the best quality of software, it is likely that customer will uncover defect in software. Corrective maintenance changes the software to correct the defects.

- **Adaptive:** Over time, the original environment (CPU, OS, business rules, external product character etc.) for which the software was developed may change. Adaptive maintenance results in modification to the software to accommodate the change to its environment.

- **Perfective**: As the software is used, the customer / user will recognize additional function that will provide benefit. Perfective maintenance extends the software beyond its original functional requirements.

The software product

The objective of software engineering is to produce software products. Computer software is the product that software engineers design and built. Software products are software systems delivered to a customer with the documentation which describes how to install and use the system.

- Software products fall into two broad classes:

- **Generic products**: These are stand alone systems which are produced by a software development organizations/firms and sold on the open market to any customer who is able to buy them.

- **Customized products**: These are systems which are commissioned by a particular customer. The software is developed specially for that customer by some developer.

…. Software products

- Until the 1980’s, the vast majority of software systems which were sold that were customized and specially designed systems which run on large computers. They are expensive because all the development cost had to be met by a single client.

- After the development of PCs, this situation has completely changed. The PC market is totally dominated by software products produces by companies such as Microsoft. These account for the vast majority of software sales. These are usually relatively cheap because their development cost is spread across hundred or thousands of different customers.

Difference between generic and customized software

- The generic software product specifications are produced internally by the marketing department of the product company. They reflect what they think will sell. They are usually flexible and non-prescriptive.

- For customized systems are often the basis for the contract between customer and developer. They are usually defined in detail and changes have to be negotiated and carefully costed.

Software product attributes

The attributes of a software product are the characteristics displayed by the product, once it is installed and put in use. They are not the services provided by the product.

Rather, they are concerned with the products dynamic behavior and the use made of the product.

Examples of these attributes are therefore, **efficiency, reliability, maintainability, robustness, portability and so on.**

The relative importance of these characteristics obviously varies from system to system.

… product attributes

Product characteristics	**Description**
Maintainability	It should be possible to evolve software to meet the changing needs of the customer.
Dependability	Software dependability includes a range of characteristics including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycle
Usability	Software should have an appropriate user interface and documentation

..product attributes

Optimizing the above attribute is difficult as some are exclusive.

For example, providing a better user interface may reduce system efficiency.

The relationship between cost and improvement in each of the attribute is not linear one.

Small improvement in any of these attributes may be devoted to optimizing particular attribute.

Software Processes and Models

The software process

- A structured set of activities required to develop a software system.
- Many different software processes but all involve:
  - Specification – defining what the system should do;
  - Design and implementation – defining the organization of the system and implementing the system;
  - Validation – checking that it does what the customer wants;
  - Evolution – changing the system in response to changing customer needs.

- A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.

Software process descriptions

- When we describe and discuss processes, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc. and the ordering of these activities.
- Process descriptions may also include:
  - Products, which are the outcomes of a process activity;
  - Roles, which reflect the responsibilities of the people involved in the process;
  - Pre- and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced.

Plan-driven and agile processes

- Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan.
- In agile processes, planning is incremental and it is easier to change the process to reflect changing customer requirements.
- In practice, most practical processes include elements of both plan-driven and agile approaches.

Software Development Life cycle (SDLC)

- A life cycle model prescribes the different activities that need to be carried out to develop a software product and sequencing of these activities.
- Also referred to as Systems Development Life cycle.
- Every software product starts with a request for the product by the customer.- Production conception.
- The software life cycle can be considered as the business process for software development and therefore is often referred to as a Software process. (SLCM).
- Process models – More detailed and precise life cycle activities.

- **Different stages in a life cycle model:** After **Product conception.** The stages are: **(Life cycle phase)**
  - **Feasibility study stage**
  - **Requirements analysis and specification.**
  - **Design**
  - **Coding**
  - **Testing and**
  - **Maintenance.**
- A SDLC is a series of identifiable stages that a software product undergoes during its lifetime.
- A SDLC is a descriptive and diagrammatic representation of the software life cycle.
- A life cycle model maps the different activities performed on a software product from its beginning to retirement into a set of life cycle phases.

- Why use a life cycle model?

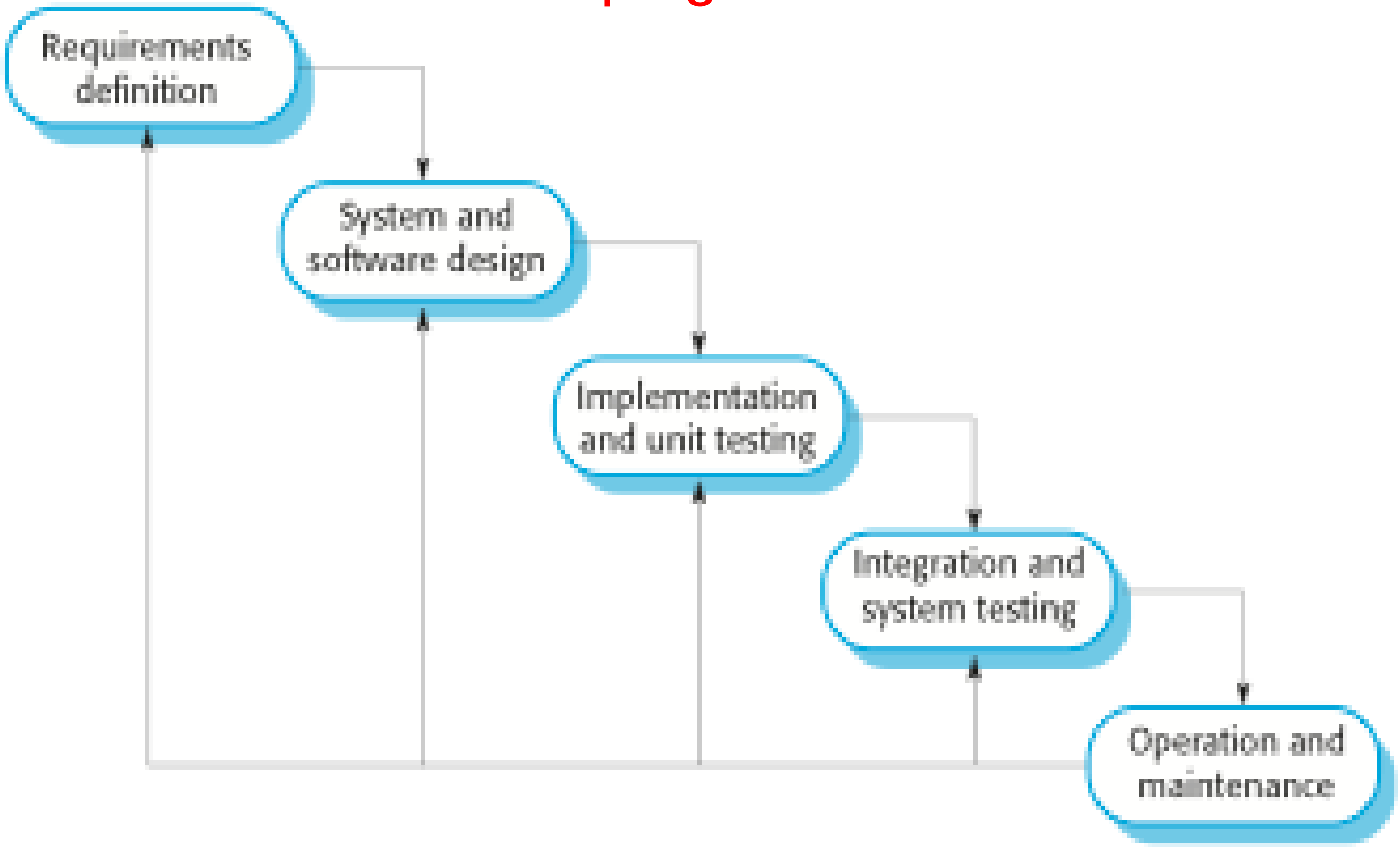
- Encourages development of software in a systematic and disciplined manner
- S.D organisations have realized that adherence to a suitable well-defined life cycle model helps to produce good quality products and that too without time and cost overruns.

- Why document a life cycle model?

- A documented life cycle model, besides preventing misinterpretations that occur when the life cycle model is no adequately documented, also helps to identify inconsistencies, redundancies, and omissions in the development process

1. The classical waterfall model

Basic life cycle model- Theoretical way of developing software.


Requirements definition
System and software design
Implementation and unit testing
Integration and system testing
Operation and maintenance

Classical Waterfall model phases

- There are separate identified phases in the waterfall model:
  - Requirements analysis and definition
  - System and software design
  - Implementation and unit testing
  - Integration and system testing
  - Operation and maintenance
- The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. In principle, a phase has to be complete before moving onto the next phase.

Classical Waterfall model problems

- Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.

  - Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
  - Few business systems have stable requirements.

- The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.
  - In those circumstances, the plan-driven nature of the waterfall model helps coordinate the work.

2. Structured Evolutionary Prototyping Model

- Developers build a prototype during the requirements phase.
- Prototype is evaluated by end users.
- Users give corrective feedback.
- Developers further refine the prototype.
- When the user is satisfied, the prototype code is brought up to the standards needed for a final product.

Structured Evolutionary Prototyping Steps

- A preliminary project plan is developed.
- An partial high-level paper model is created.
- The model is source for a partial requirements specification.
- A prototype is built with basic and critical attributes
- The designer builds
  - the database
  - user interface
  - algorithmic functions
- The designer demonstrates the prototype, the user evaluates for problems and suggests improvements.
- This loop continues until the user is satisfied.

Structured Evolutionary Prototyping Strengths

- Customers can “see” the system requirements as they are being gathered.
- Developers learn from customers.
- A more accurate end product.
- Unexpected requirements accommodated.
- Allows for flexible design and development.
- Steady, visible signs of progress produced.
- Interaction with the prototype stimulates awareness of additional needed functionality.

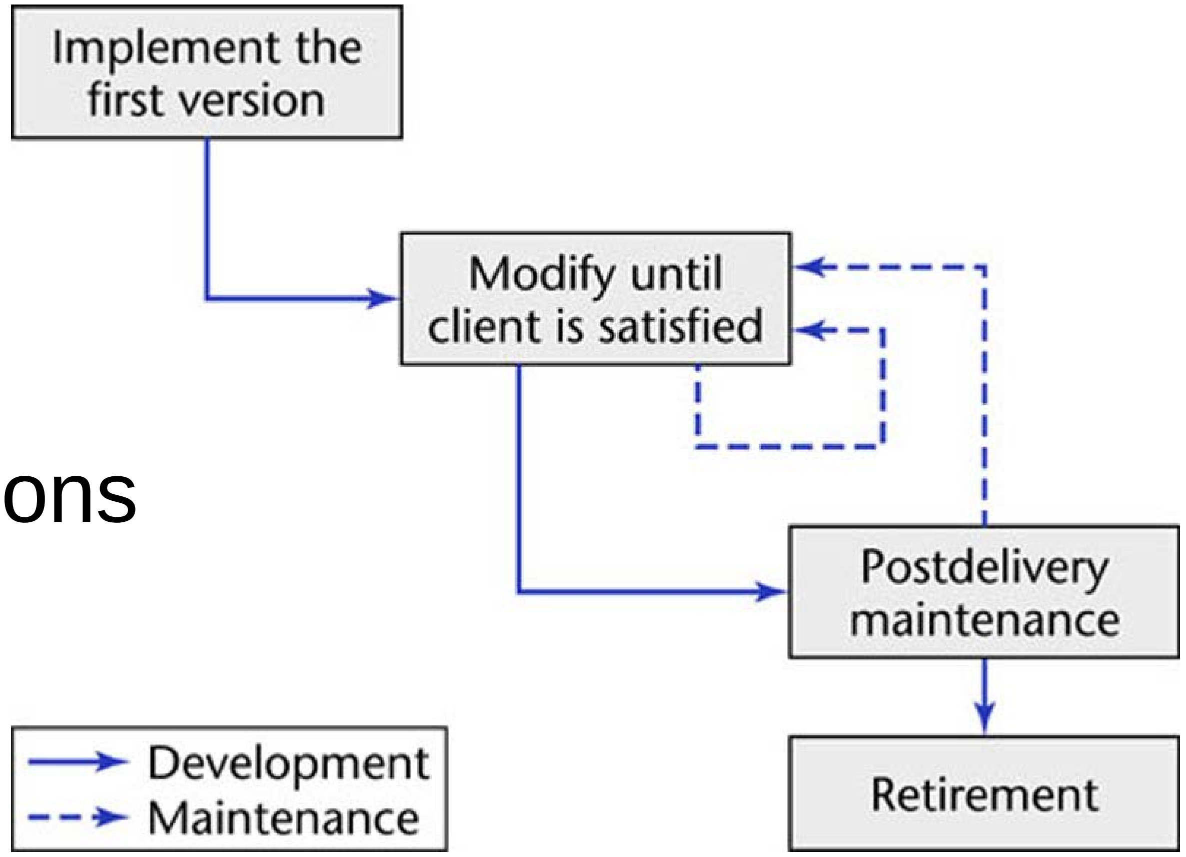
Structured Evolutionary Prototyping Weaknesses

- Tendency to abandon structured program development for “code-and-fix” development.
- Bad reputation for “quick-and-dirty” methods
- Overall maintainability may be overlooked.
- The customer may want the prototype delivered.
- Process may continue forever (scope creep).

When to use
Structured Evolutionary Prototyping

- Requirements are unstable or have to be clarified.
- As the requirements clarification stage of a waterfall model.
- Develop user interfaces.
- Short-lived demonstrations.
- New, original development.
- With the analysis and design portions of object-oriented development.

➢ Code-and-Fix Life-Cycle Model



- No design
- No specifications

The easiest way to develop software
The most expensive way for maintenance
(i.e., maintenance nightmare)

Code-and-Fix Life-Cycle Model (Cont.)

- The product is implemented without requirements or specifications, or any attempt at design.
- The developers simply throw code together and rework it as many times as necessary to satisfy the client.
- It is used in small project and is totally unsatisfactory for products of any reasonable size.

3. Spiral Model

Since end-user requirements are hard to obtain/define, it is natural to develop software in an ***experimental*** way: e.g.

1. Build some software
2. See if it meets customer requirements
3. If no go to 1 else stop.

This loop approach gives rise to structured

<u>iterative lifecycle models.</u>

In 1988 Boehm developed the spiral model as an iterative model which includes ***risk analysis*** and ***risk management***.

<u>Key idea</u>: on each iteration identify and solve the sub-problems with the ***highest risk***.

Spiral model

- Spiral with many loops.
- Each loop of the spiral is called the phase of a software process.

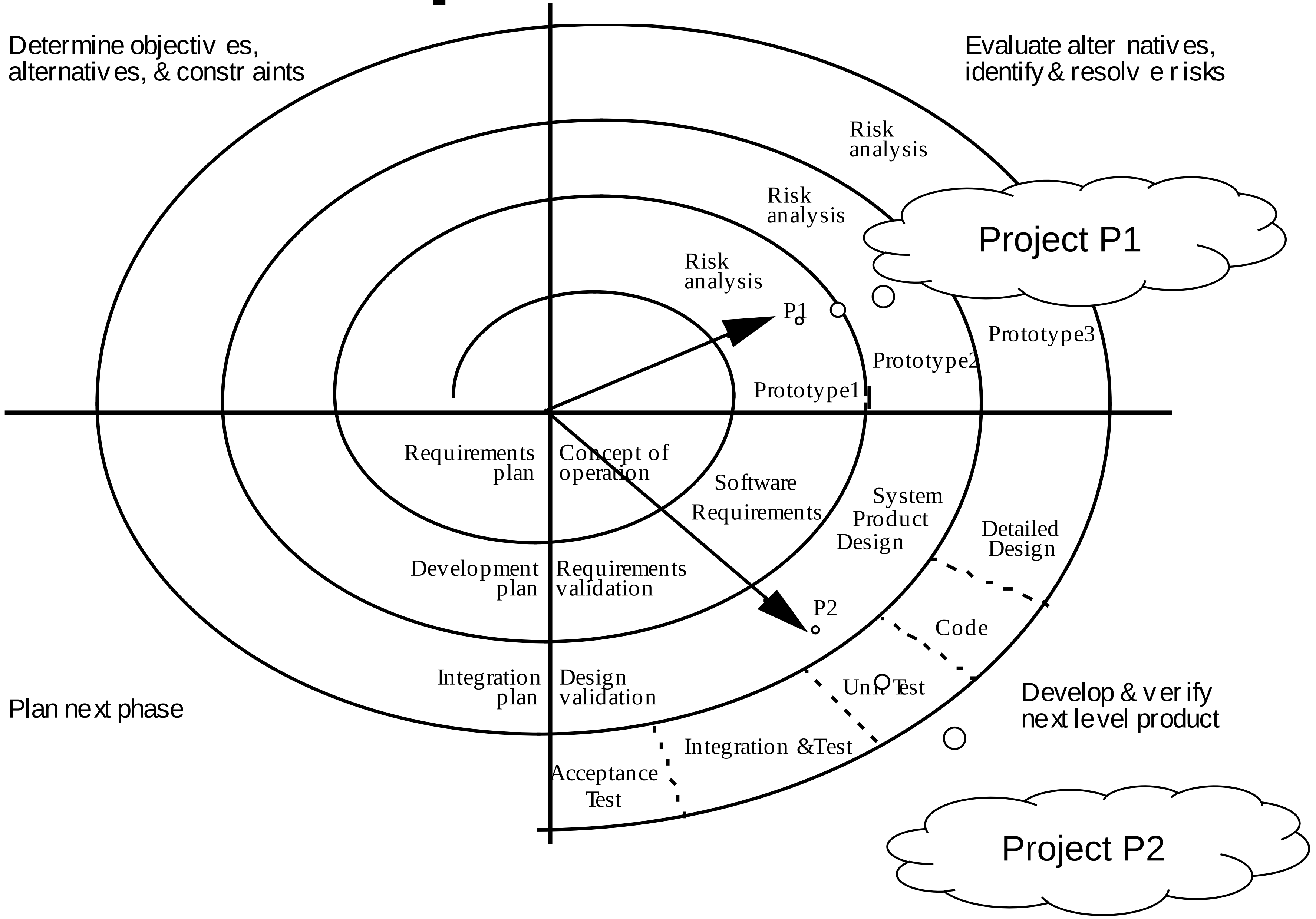
- Over each loop, one or more features of the product are elaborated and analysed and risks at that point of time are identified and resolved through prototyping. Based on this, the identified features are implemented.

4 Quadrants of spiral model

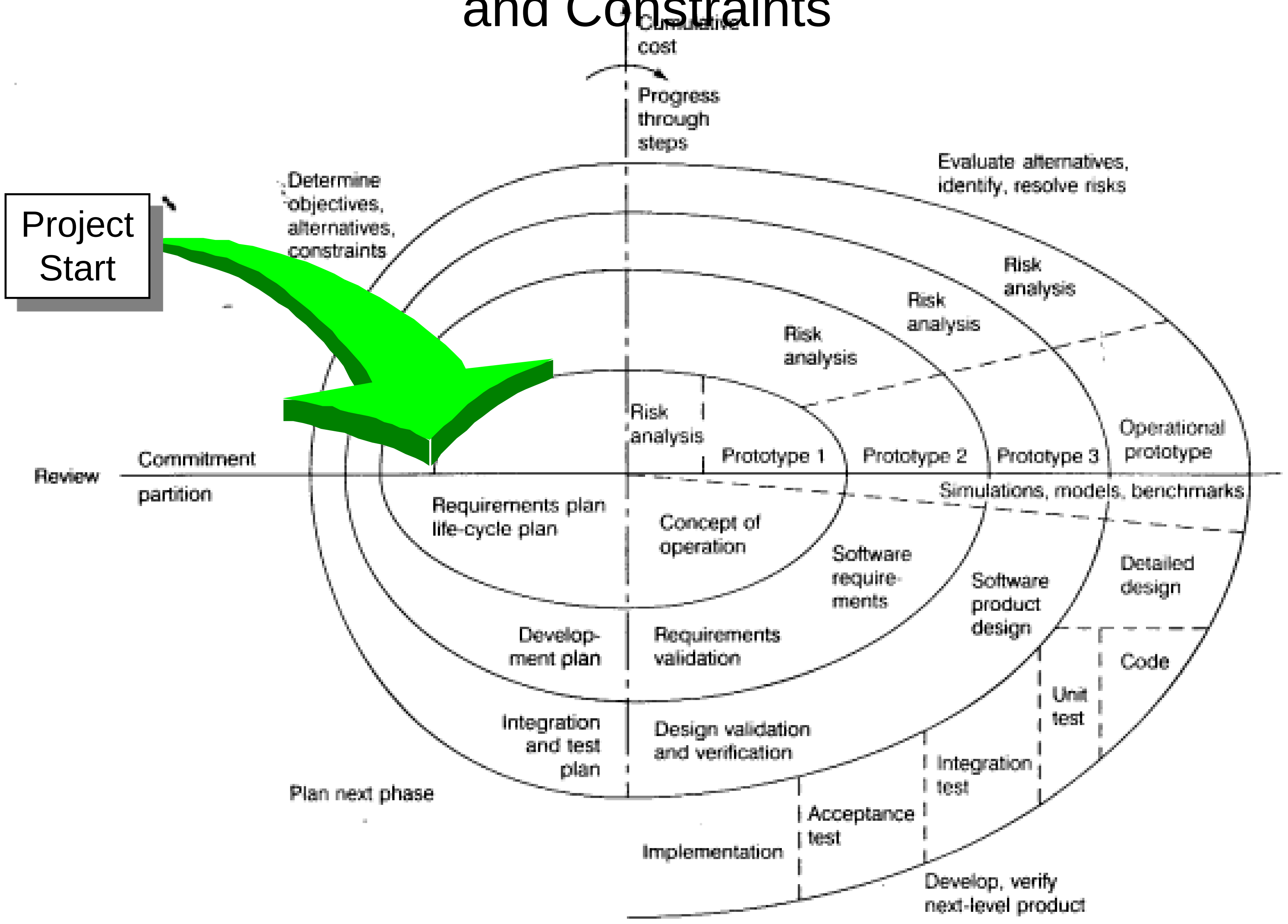
- 1st Quadrant:
  - The objectives are investigated, elaborated and analysed.
  - Risks are also identified.
  - Alternative Solutions are proposed
- 2nd Quadrant:
  - Alternative solutions are evaluated to select best.
- 3rd Quadrant:
  - Developing and verifying the next level of the product
- 4th Quadrant:
  - Reviewing the results of the stages traversed so far with the customer.
  - Planning the next iteration around the spiral.

:

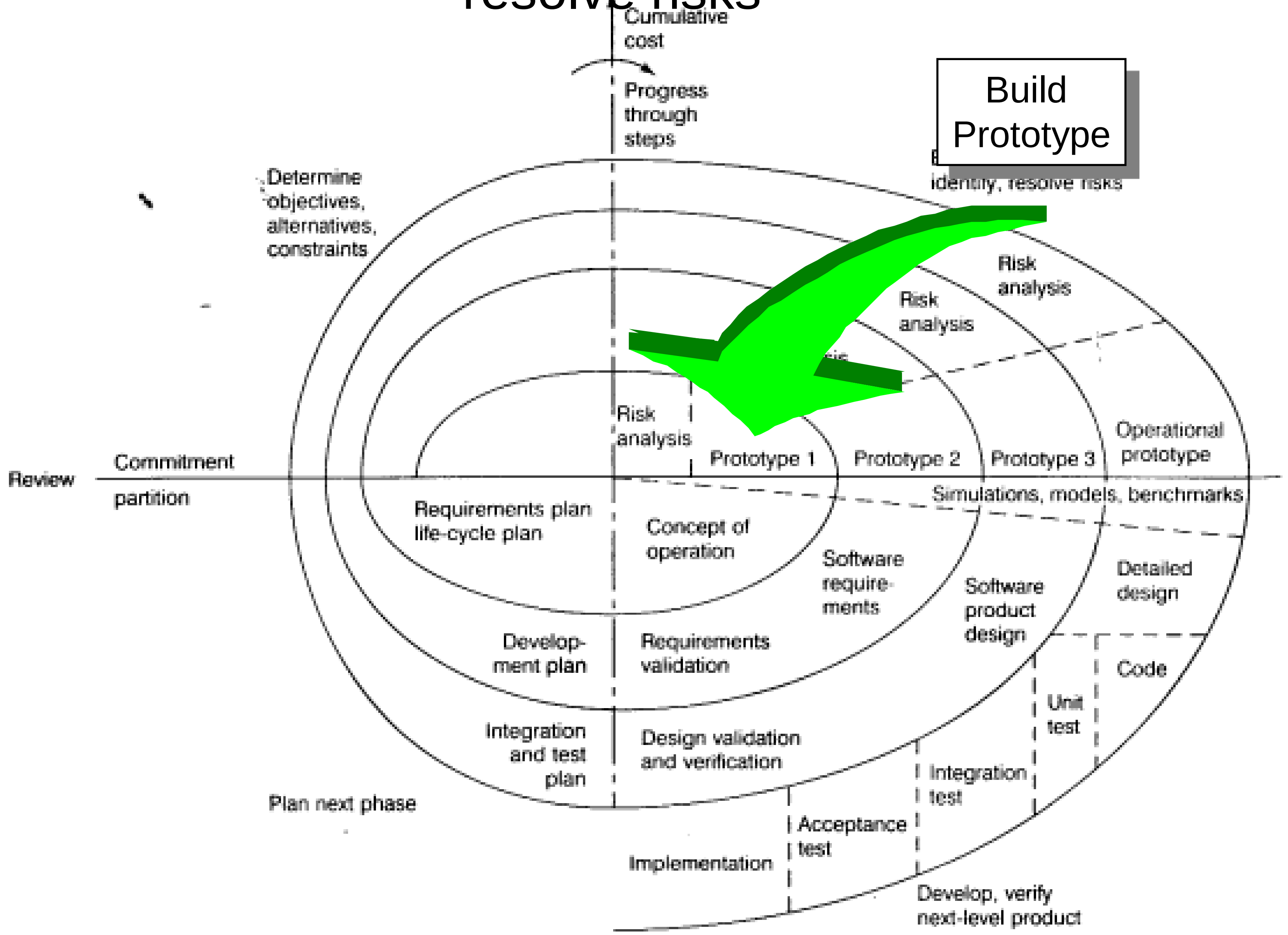
Spiral Model



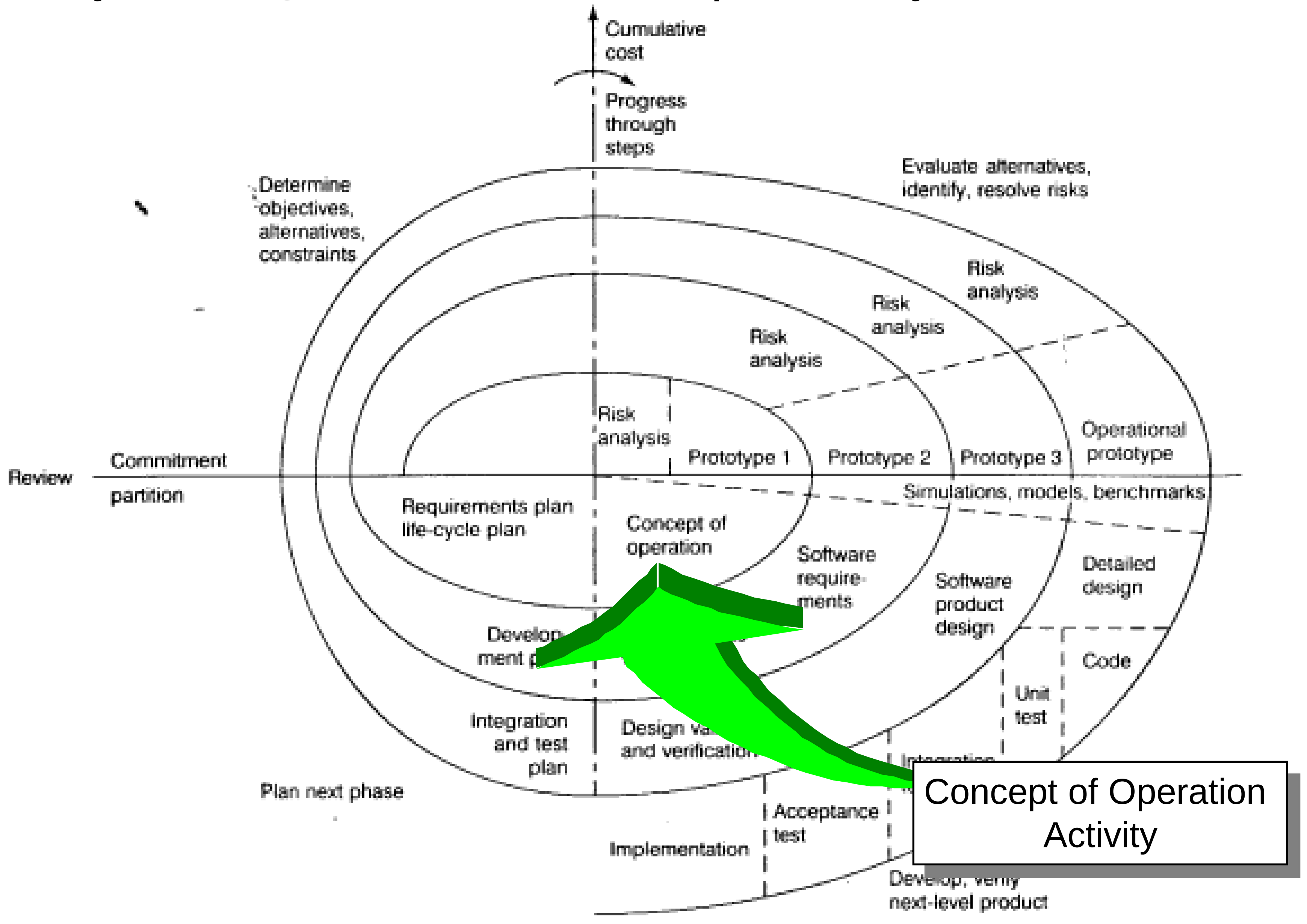
Cycle 1, Quadrant IV: Determine Objectives, Alternatives and Constraints



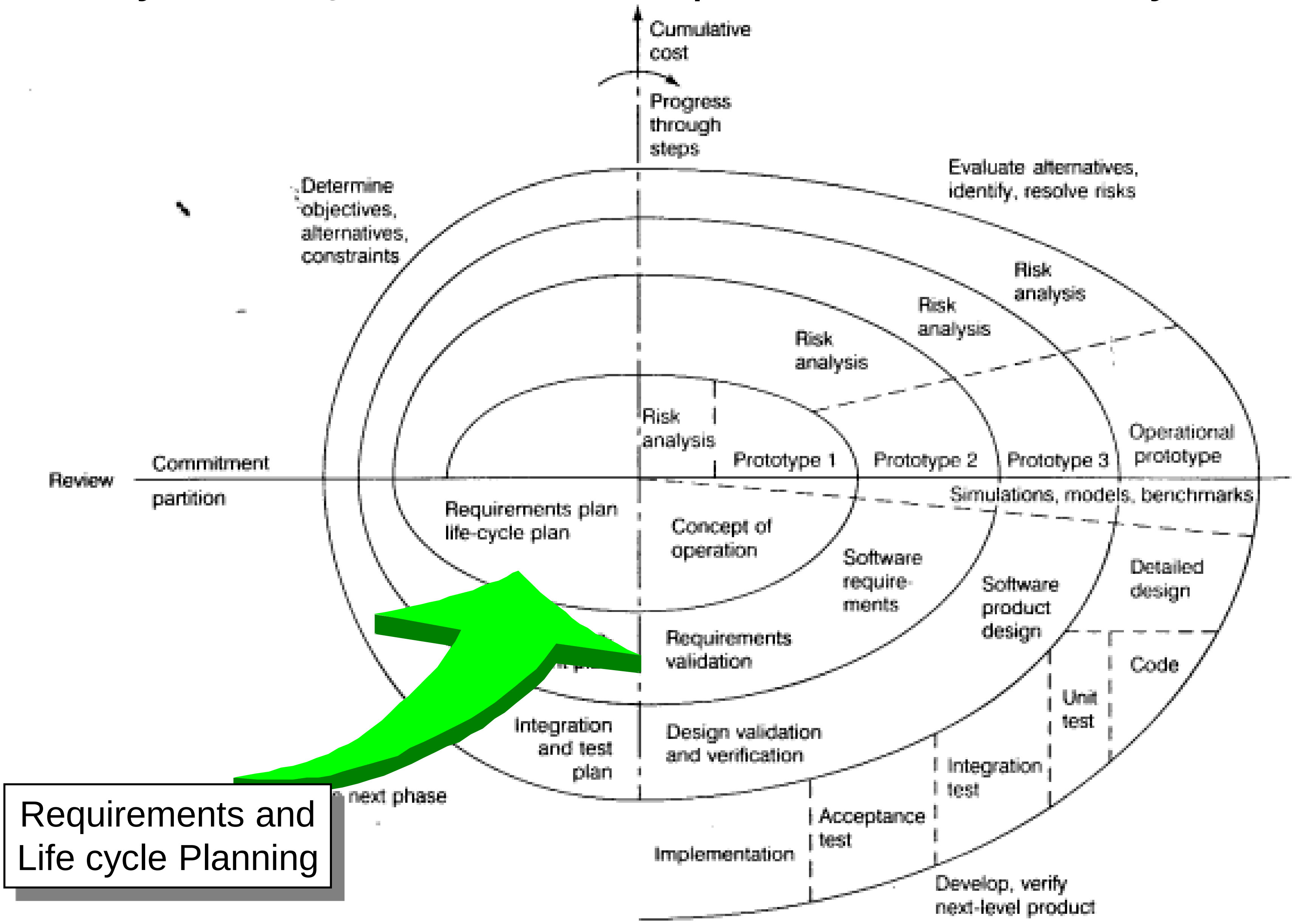
Cycle 1, Quadrant I: Evaluate Alternatives, Identify, resolve risks



Cycle 1, Quadrant II: Develop & Verify Product

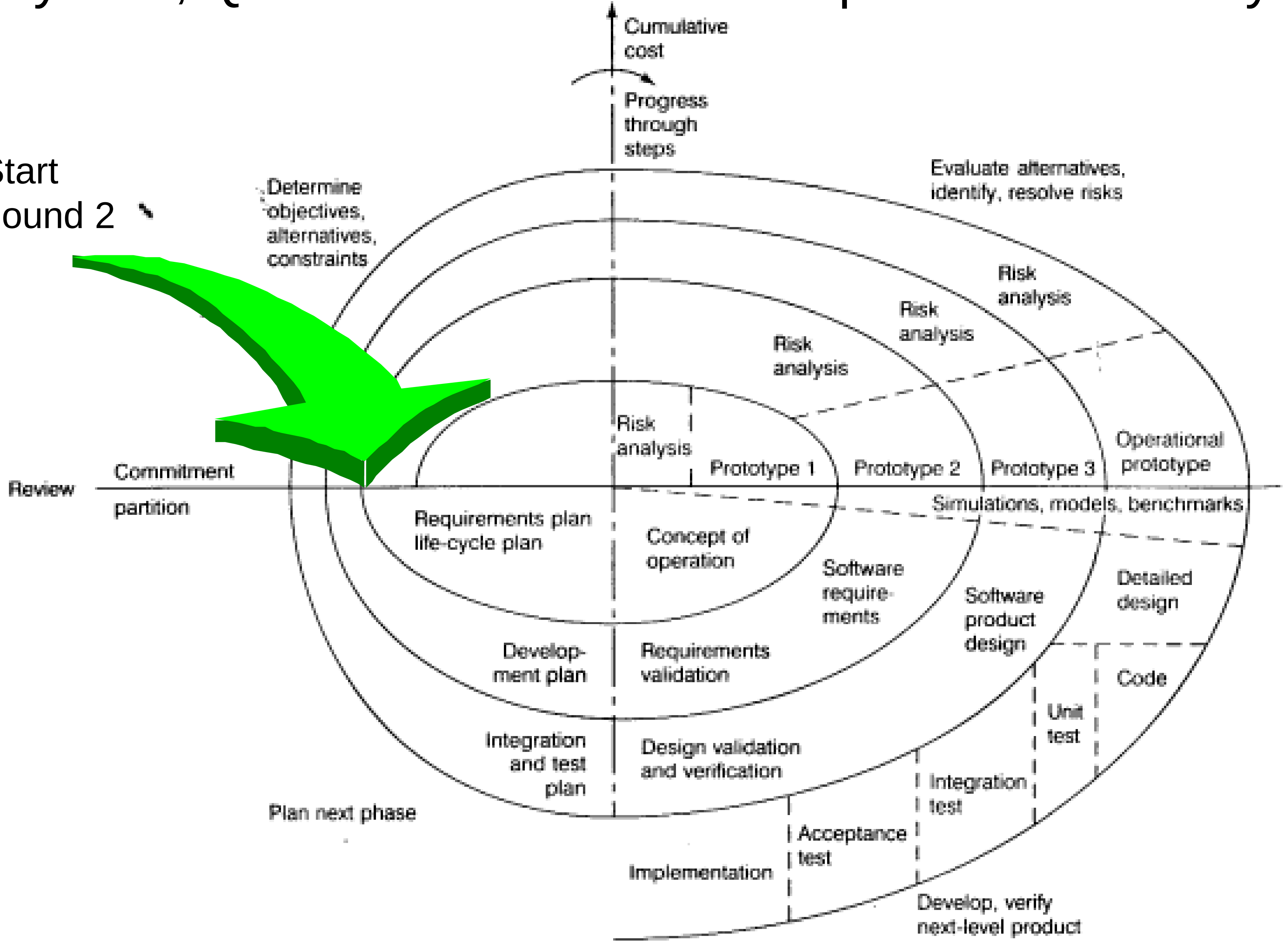


Cycle 1, Quadrant III: Prepare for Next Activity



Cycle 2, Quadrant IV: Software Requirements Activity

Start
of Round 2



Spiral Model Strengths

- Provides early indication of insurmountable risks, without much cost.
- Users see the system early because of rapid prototyping tools.
- Critical high-risk functions are developed first.
- The design does not have to be perfect.
- Users can be closely tied to all lifecycle steps.
- Early and frequent feedback from users.
- Cumulative costs assessed frequently.

Spiral Model Weaknesses

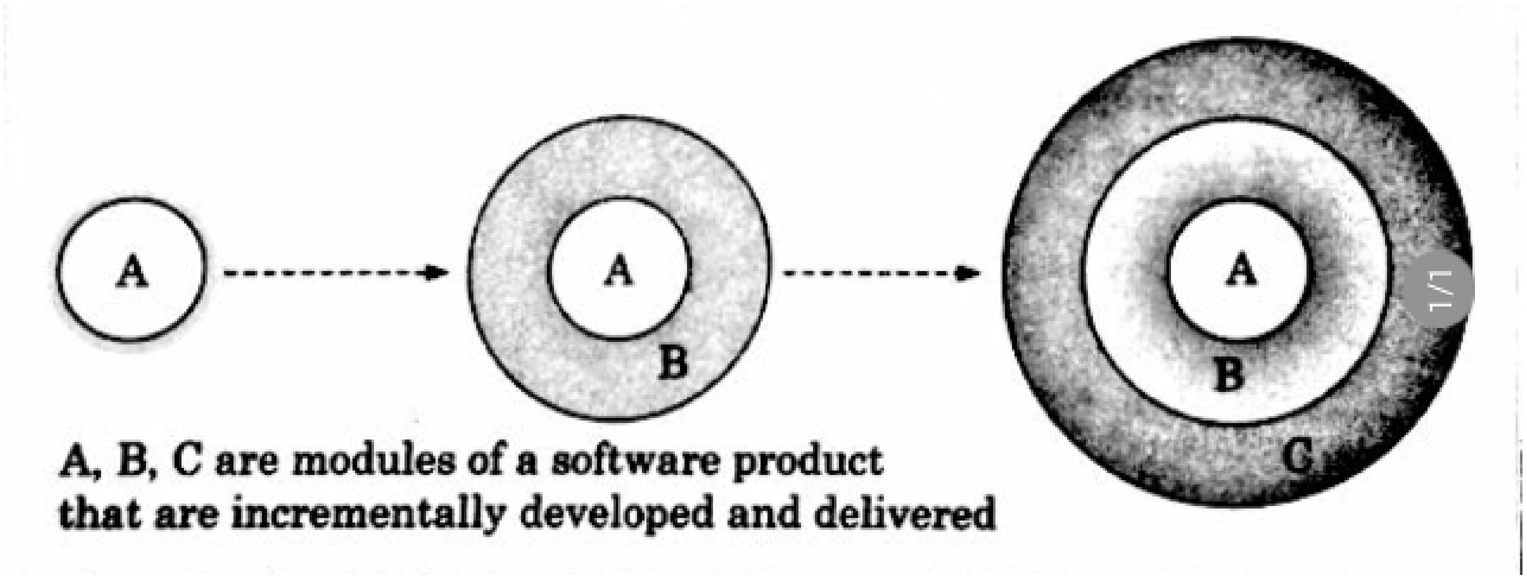
- Time spent for evaluating risks too large for small or low-risk projects.
- Time spent planning, resetting objectives, doing risk analysis and prototyping may be excessive.
- The model is complex.
- Risk assessment expertise is required.
- Spiral may continue indefinitely.
- Developers must be reassigned during non-development phase activities.
- May be hard to define objective, verifiable milestones that indicate readiness to proceed through the next iteration.

4. Evolutionary Model

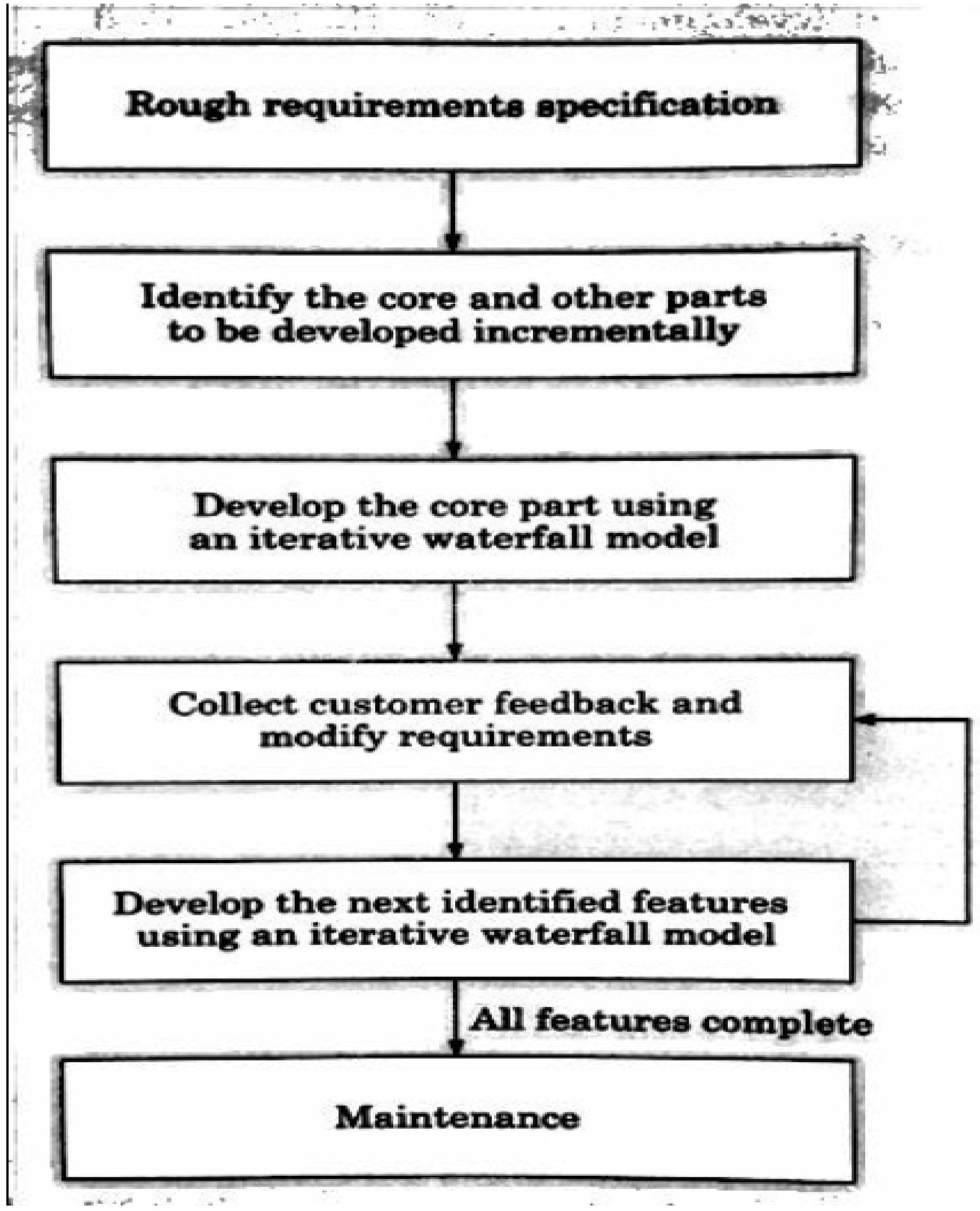
- Successive versions model.
- Incremental model.
- A simple working system is built, which subsequently undergoes many functionality improvements and additions until the desired system is realized .
- Also sometimes referred to as design a little, build a little, test a little, deploy a little model.
- That is once the requirements have been specified, the design, build, test and deployment activities are interleaved.

- The software requirements is first broken down into several modules(functional units) that can be incrementally constructed and delivered.
- The development team first develop the core modules of the system.
- The Core modules are those that don’t need services from the other modules.
- The initial product Skelton is refined into increasing levels of capability by adding new functionalities in the successive versions.

Life cycle activities



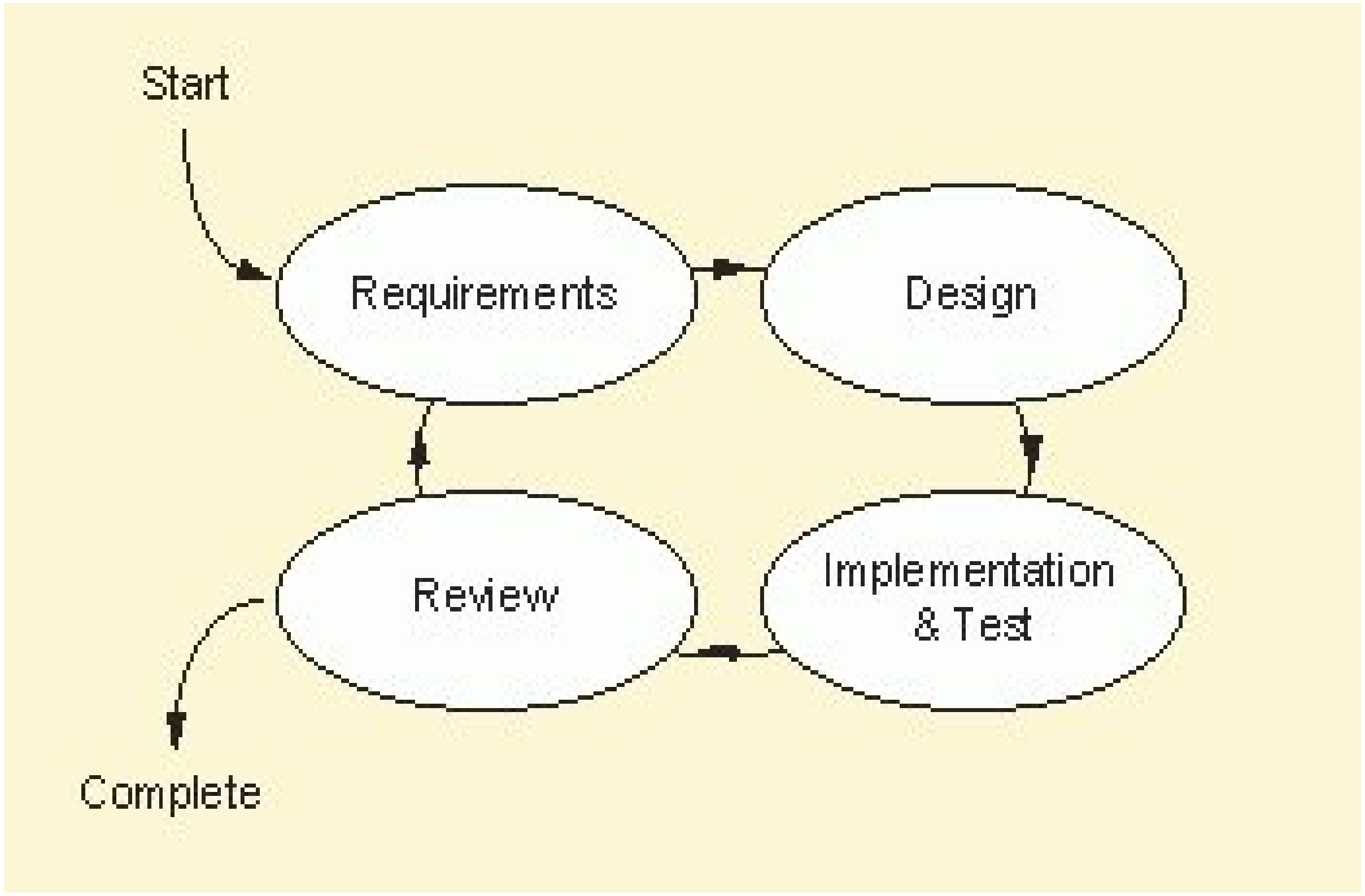
A, B, C are modules of a software product that are incrementally developed and delivered


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
Develop the next identified features using an iterative waterfall model
All features complete
Maintenance

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:

Iterative Model



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

- A ***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.

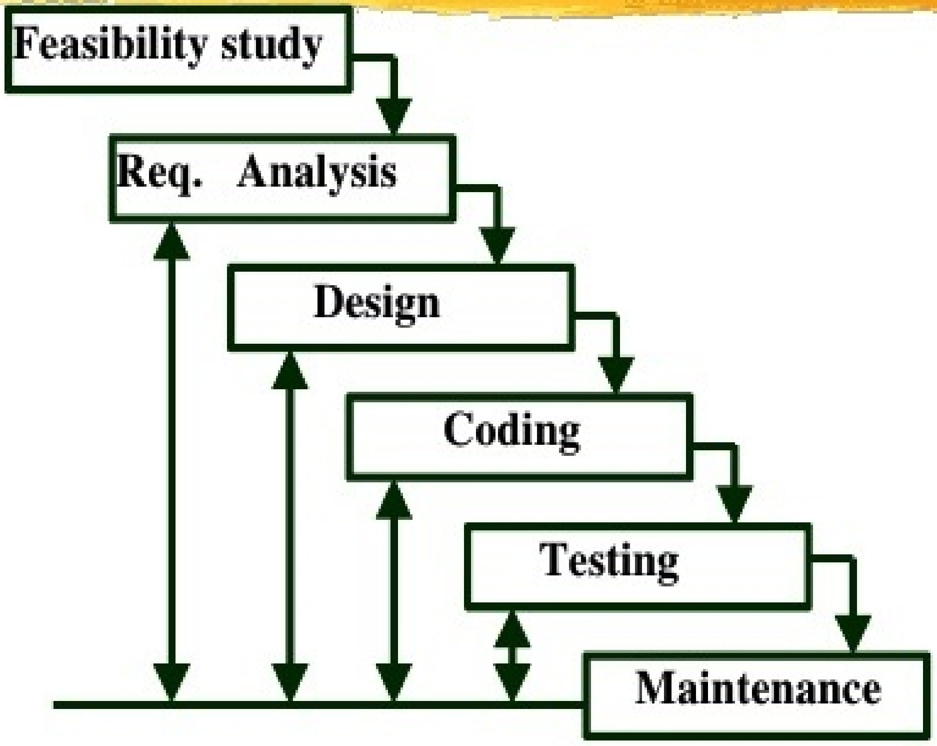
- An ***Implementation and Test*** phase, when the software is coded, integrated and tested.

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

Iterative Waterfall model

- The iterative waterfall model is classical waterfall model with necessary changes so that it becomes applicable to practical software development projects.
- The main change to the classical waterfall model is in the form of providing feedback paths from every phase to its preceding phase.
- The feedback paths allows for correction of errors committed during a phase, as and when these are detected in a later phase.
- The principle of detecting errors as close to their points of introduction as possible is known as phase containment of errors.

Iterative Waterfall Model
(CONT.)



Comparison of Life-Cycle Models

- Different life-cycle models have been presented
  - Each with its own strengths and weaknesses
- Criteria for deciding on a model include:
  - The organization
  - Its management
  - The skills of the employees
  - The nature of the product
- Best suggestion
  - "Mix-and-match" life-cycle model

Features	Original water fall	Iterative water fall	Prototyping model	Spiral model
Requirement Specification	Beginning	Beginning	Frequently Changed	Beginning
Understanding Requirements	Well Understood	Not Well understood	Not Well understood	Well Understood
Cost	Low	Low	High	Expensive
Availability of reuseable component	No	Yes	yes	yes
Complexity of system	Simple	simple	complex	complex
Risk Analysis	Only at beginning	No Risk Analysis	No Risk Analysis	yes
User Involvement in all phases of SDLC	Only at beginning	Intermediate	High	High
Guarantee of Success	Less	High	Good	High
Overlapping Phases	No overlapping	No Overlapping	Yes Overlapping	Yes Overlapping
Implementation time	long	Less	Less	Depends on project
Flexibility	Rigid	Less Flexible	Highly Flexible	Flexible