Software Development Models

Lecturer: Ngo Huy Bien
Software Engineering Department
Faculty of Information Technology
VNUHCM - University of Science
Ho Chi Minh City, Vietnam
nhbien@fit.hcmus.edu.vn

Objectives

- To present *Waterfall* model
- To present *modified Waterfall* models
- To present *Iterative and Incremental development* (IID) model
- To present *Spiral* model
- To present *Vee* model

References

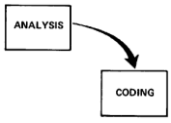
1. Herbert D. Bennington. Production of Large Computer Programs. 1956.
2. Winston Royce. Managing The Development of Large Software Systems. 1970.
3. Steve Mcconnell, *Rapid Development - Taming Wild Software Schedules*. 1996.
4. Craig Larman, *Agile and Iterative Development: A Manager's Guide*. 2003.
5. Barry Boehm. A Spiral Model of Software Development and Enhancement. 1988.
6. Kevin Forsberg and Harold Mooz. The Relationship of System Engineering to the Project Cycle. 1991.
7. Nayan B. Ruparelia. Software Development Lifecycle. 2010.

Why Development Models?

- Which step should we do *next*?
- How *long* will it take?
- *How* to perform the step?
- Which *artifacts* will it use and produce?
- *Who* is responsible for doing the step?

Analysis and Coding [1] [2]

- There are two essential steps common to all computer program developments, *regardless of size or complexity*. There is first an *analysis* step, followed second by a *coding* step.



- It's the development effort for which most customers are *happy to pay*, since both steps involve genuinely creative work which directly contributes to *the usefulness* of the final product.

Code and Fix

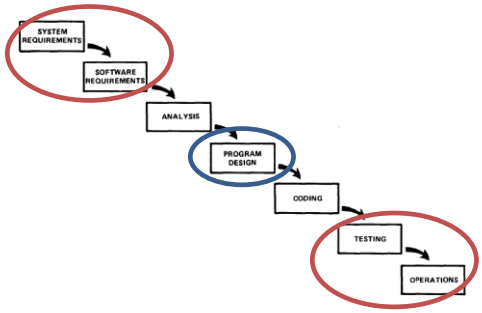
Code and Fix is the development method in which you write some code and then fix the problems in the code.

System specification (Optional) → Analyze, Code and Fix → Release (Maybe)

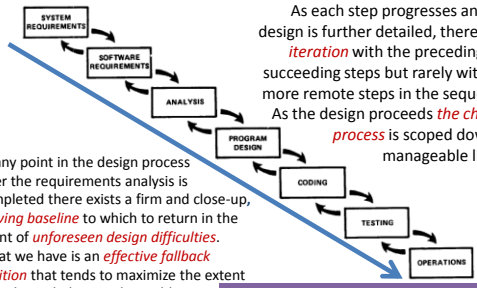
✓ No Overhead (No planning, documentation, QA, standards enforcing, etc., just coding.)
✓ Requires no expertise; anybody can do this.

❖ No way of identifying risk
❖ Poor match to user's need
❖ Poor structure
❖ No way of accessing quality
❖ Expensive fix
❖ No way of accessing progress

A More Grandiose Approach



Iterative Relationship Between Successive Phases



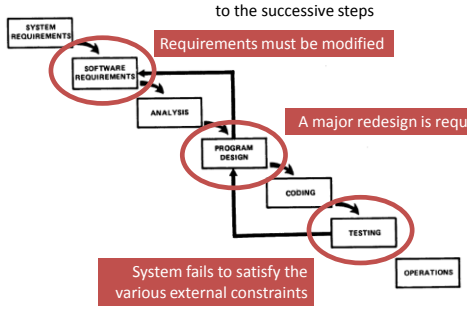
As each step progresses and the design is further detailed, there is *an iteration* with the preceding and succeeding steps but rarely with the more remote steps in the sequence. As the design proceeds *the change process* is scoped down to manageable limits.

At any point in the design process after the requirements analysis is completed there exists a firm and close-up, *moving baseline* to which to return in the event of *unforeseen design difficulties*. What we have is an *effective fallback position* that tends to maximize the extent of early work that is salvageable and preserved.

Hopefully, the iterative interaction between the various phases is confined to successive steps.

Development Risk

Unfortunately, the design iterations are *never confined* to the successive steps



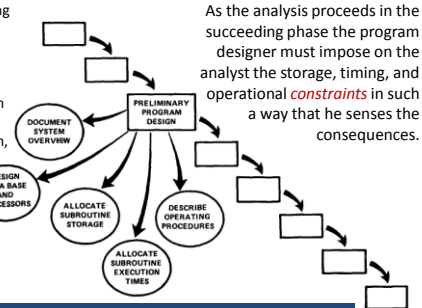
Requirements must be modified

A major redesign is required

System fails to satisfy the various external constraints

Preliminary Program Design

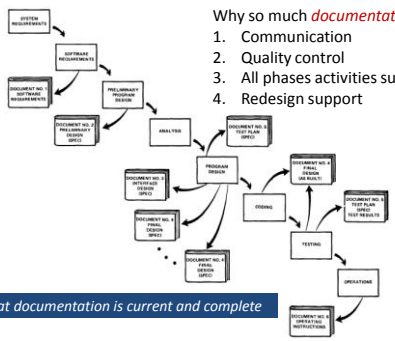
If something wrong is recognized at this earlier stage, the iteration with requirements and preliminary design can be *redone* before final design, coding and test commences.



As the analysis proceeds in the succeeding phase the program designer must impose on the analyst the storage, timing, and operational *constraints* in such a way that he senses the consequences.

Insure that a preliminary program design is complete before analysis begins

Document The Design

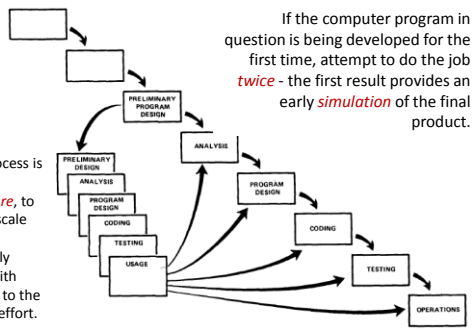


Why so much *documentation*?

1. Communication
2. Quality control
3. All phases activities support
4. Redesign support

Insure that documentation is current and complete

Do It Twice

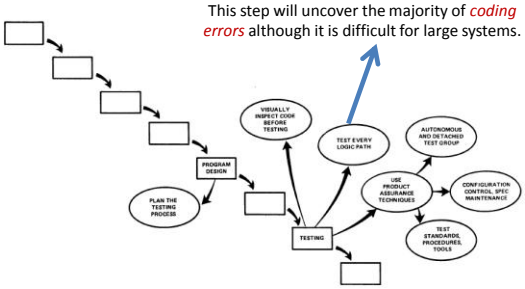


If the computer program in question is being developed for the first time, attempt to do the job *twice* - the first result provides an early *simulation* of the final product.

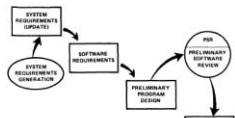
This process is done in *miniature*, to a time scale that is relatively small with respect to the overall effort.

Plan, Control and Monitor Testing

This step will uncover the majority of *coding errors* although it is difficult for large systems.

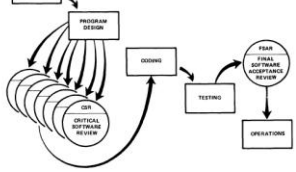


Involve the Customer



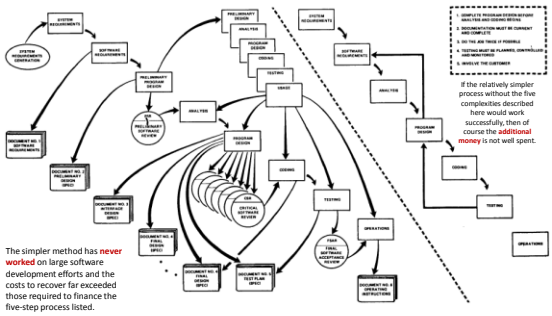
To give the contractor *free rein* between requirement definition and operation is inviting trouble.

The insight, *judgment, and commitment* of the customer can bolster the development effort.



The involvement should be formal, in-depth and continuing.

Managing the Development of Large Software Systems



If the relatively simpler process without the five complexities described here would work successfully, then of course the **additional money** is not well spent.

The simpler method has **never worked** on large software development efforts and the costs to recover far exceeded those required to finance the five-step process listed.

(Winston Royce's) Waterfall Model

- The (Winston Royce's) successful waterfall model is a SDLC model that has the following characteristics:
  - It consists of definite phases that are *executed in sequence*.
  - There are *tangible deliverables produced* at the end of each phase.
  - The phases *be revisited* but the overall cycle is completely executed no more than *2 times*. If executed more than once, the first cycle is for a simulation.
  - Once design begins, a formal *change-control process* is used.
  - The process is *document-driven*.

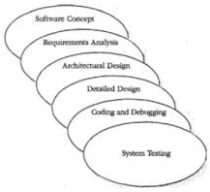
Waterfall Model Pros & Cons

- *Limit risk* to an affordable sum.
- *Minimize effort* spent on unproductive or undesired directions.
- Avoid the possibility of *losing control* of a project.

- Its emphasis on *fully elaborated documents* as completion criteria for early requirements and design phases.
- Document-driven standards have pushed many projects to write *elaborate specifications* of poorly understood user interfaces and decision support functions, followed by the design and development of *large quantities of unusable code*.
- In real life there is a need to initiate software design and coding, and hardware modeling, earlier in the project cycle to ensure that user requirements are understood and to *prove technical feasibility*.

Sashimi Model [3]

- Most of the weaknesses in the pure waterfall model arise not from problems with these activities but from the treatment of these activities as *disjoint, sequential phases*.
- The traditional waterfall model allows for *minimal overlapping* between phases at the end-of-phase review.
- This model suggests a *stronger degree of overlap*.

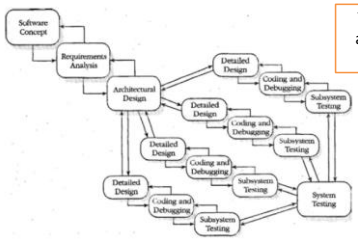


- Milestones are *more ambiguous*. It's harder to track progress accurately.
- Performing activities in parallel can lead to *miscommunication*, *mistaken assumptions*, and inefficiency.

Waterfall with Subprojects

Why delay the implementation of the areas that are easy to design just because we're *waiting for* the design of a difficult area? If the architecture has broken the system into *logically independent subsystems*, you can spin off separate projects, each of which can proceed at its own pace.

The main risk with this approach is *unforeseen interdependences*.



Waterfall with Risk-Reduction

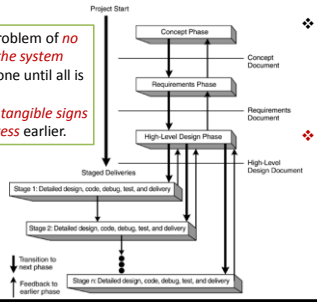
It requires more effort.

1. Develop a user-interface prototype
2. Use system storyboarding
3. Conduct user interviews
4. Videotape users interacting with an older system

Staged Delivery

The staged-delivery is a model in which you show software to the customer in *successively refined stages*.

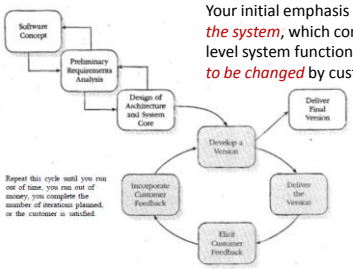
- ✓ Avoid problem of *no part of the system* being done until all is done.
- ✓ Provide *tangible signs of progress* earlier.



- ❖ Must account for all *technical dependencies* between different components of the product.
- ❖ *Integration* cost.

Evolutionary Delivery

Evolutionary delivery is a lifecycle model in which you develop a version of your product, show it to your customer, and refine the product based on customer feedback.



Your initial emphasis is on *the core of the system*, which consists of lower level system functions that are *unlikely to be changed* by customer feedback.

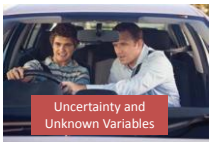
Requirements Gathering

I can't tell you what I want, but I'll know it when I see it.

Software Changes

Software Development is more like New Product Development than Manufacturing

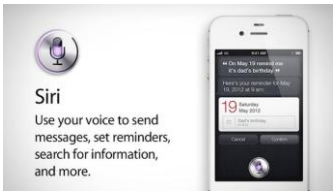
- ❑ The requirements change.
- ❑ The design changes.
- ❑ The business changes.
- ❑ The technology changes.
- ❑ The team changes.
- ❑ The team members change.



The problem isn't change, because change is going to happen; the problem, rather, is our inability to cope with change.

Customer Confidence

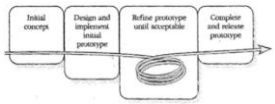
- ❑ When will I receive my product?
- ❑ How will it look like?
- ❑ Will it be usable?



Deliver business values to client fast and repeatedly.
Gives customers a chance to "try software" periodically and provide feedback.

Evolutionary Prototyping Model

Evolutionary prototyping is a lifecycle model whose stages consist of expanding increments of an operational software product, with the directions of evolution being determined by operational experience.

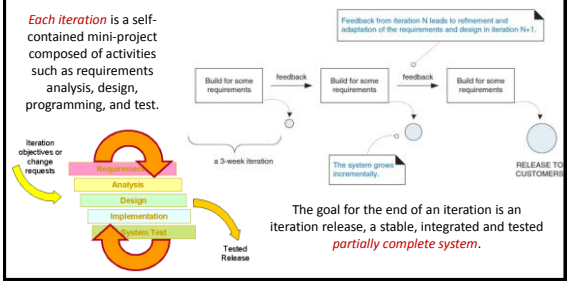


- Begin by developing the *most visible aspects* of the system.
- Demonstrate that part of the system to the customer and then continue to develop die prototype based on *the feedback* you receive.
- At some point, you and the customer agree that the prototype is "*good enough*."
- At that point, you *complete any remaining work* on the system and release the prototype as the final product.

Iterative and Incremental Development [4]

Iterative and incremental development is an approach to building software in which the overall lifecycle is composed of *several iterations* in sequence and system functionality are sliced into *increments* (portions).

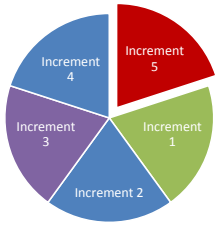
Each iteration is a self-contained mini-project composed of activities such as requirements analysis, design, programming, and test.



The goal for the end of an iteration is an iteration release, a stable, integrated and tested *partially complete system*.

Incremental Delivery

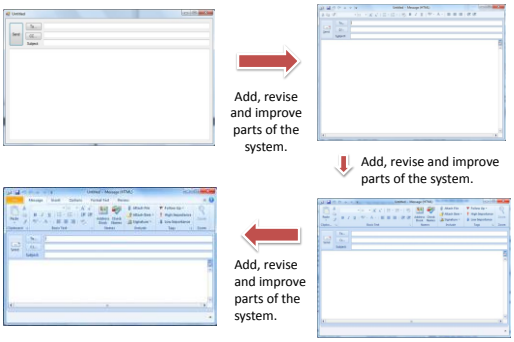
Incremental delivery is the practice of repeatedly delivering a system into production (or the marketplace) in a series of expanding capabilities.



Evolutionary Delivery

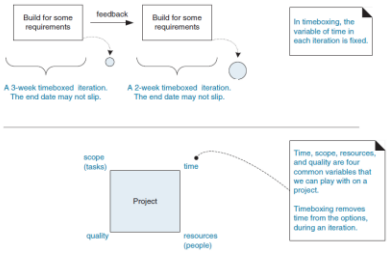
Evolutionary delivery is a refinement of the practice of incremental delivery in which there is a vigorous attempt to capture feedback regarding the installed product, and use this to guide the next delivery.

Incremental Evolutionary Delivery



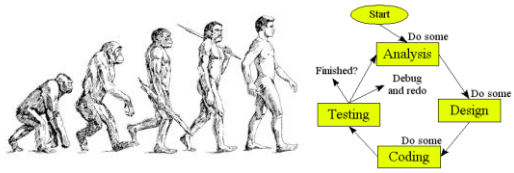
Time-boxed Iterative Development

Iteration time-boxing is the practice of fixing the iteration end date and not allowing it to change.



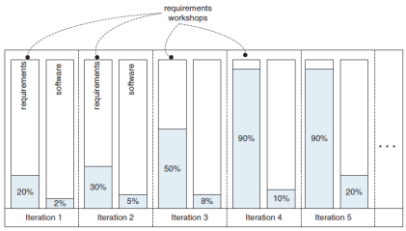
Evolutionary Iterative Development

Evolutionary iterative development implies that the requirements, plan, estimates, and solution evolve or are refined over the course of the iterations, rather than fully defined and "frozen" in a major up-front specification effort before the development iterations begin.



Evolutionary And Iterative Requirements

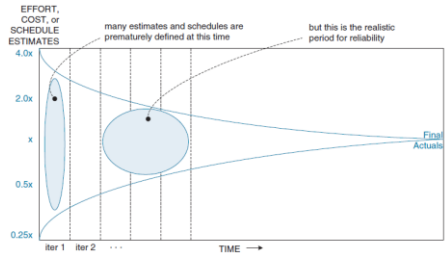
Are the requirements *forever unbounded* or *always changing* at a high rate?



20-iteration project. Most requirements will be discovered and refined within the *first four iterations*

Evolutionary and Adaptive Planning

Are the estimates and schedules are *forever unbounded or unknown*?



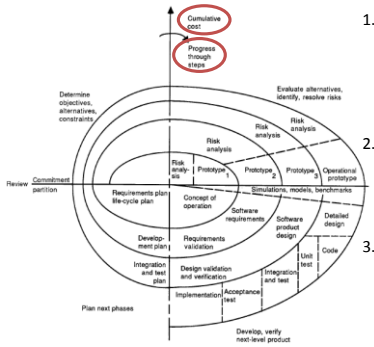
Cone of Uncertainty

Pros & Cons of IID

- ✓ *Requirements* are changing rapidly.
- ✓ Customer is reluctant to commit to a set of requirements.
- ✓ Neither you nor your customer understands the *application area* well.
- ✓ Developers are unsure of the optimal *architecture* or *algorithms* to use.
- ✓ Steady, visible signs of *progress*.

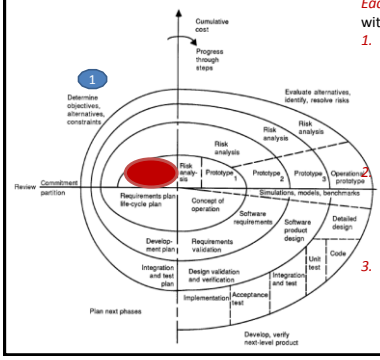
- ❖ It's impossible to know at the outset of the project *how long it will take* to create an acceptable product.
- ❖ It's hard for several independently evolved applications to be *integrated*.
- ❖ It's easy to evolve a lot of *hard-to-change code* before addressing long-range architectural and usage considerations.

Spiral Model [5] [7]



1. The *radial dimension* represents the *cumulative cost* incurred in accomplishing the steps to date.
2. The *angular dimension* represents the *progress* made in completing each *cycle of the spiral*.
3. The risks are categorized as *performance* (or user interface) related risks or *development* risks.

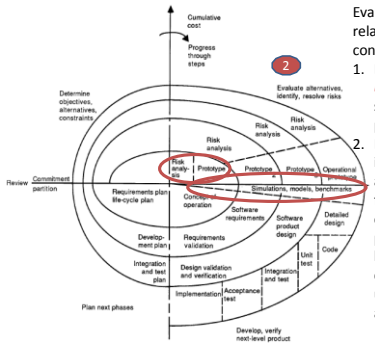
Start



Each cycle of the spiral begins with the identification of

1. *the objectives* of the portion of the product being elaborated (performance, functionality, ability to accommodate change, etc.);
2. *the alternative means* of implementing this portion of the product (design A , design B, reuse, buy, etc.); and
3. *the constraints* imposed on the application of the alternatives (cost, schedule, inter-face, etc.).

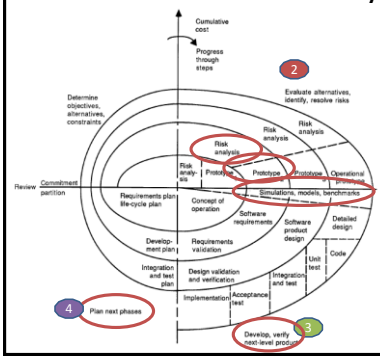
Risk Analysis



Evaluate the *alternatives* relative to the objectives and constraints.

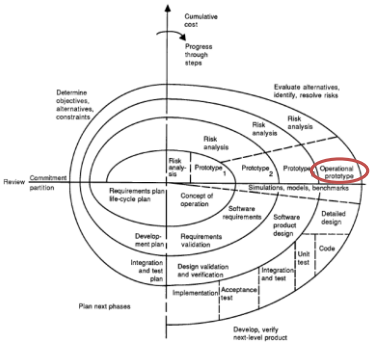
1. Identify areas of *uncertainty* that are significant sources of project risk.
2. If so, the next step should involve the formulation of a cost-effective *strategy* for resolving the sources of risk. This may involve prototyping, simulation, benchmarking, reference checking, administering user questionnaires, analytic modelling.

Another Prototype



If *performance or user-interface risks* strongly dominate program development or internal interface-control risks, the next step may be an *evolutionary development one*: a minimal effort to specify the overall nature of the product, a plan for the next level of prototyping, and the development of a *more detailed prototype* to continue to resolve the major risk issues.

Series of Evolutionary Prototypes



If this prototype is operationally useful and *robust enough* to serve as a low-risk base for future product evolution, the subsequent risk-driven steps would be the evolving *series of evolutionary prototypes* going toward the right of the figure.

In this case, the option of writing *specifications would be addressed but not exercised*. Thus, risk considerations can lead to a project implementing only *a subset* of all the potential steps in the model.

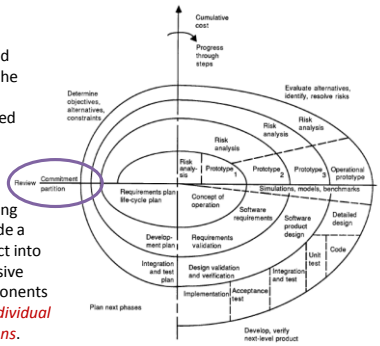
Waterfall Approach

On the other hand, if previous prototyping efforts have already resolved all of the performance or user-interface risks, and *program development or interface-control risks* dominate, the next step follows the basic *waterfall* approach (concept of operation, soft-ware requirements, preliminary design, etc.), modified as appropriate to incorporate incremental development. Each level of software specification in the figure is then followed by *a validation step* and the preparation of plans for the succeeding cycle.

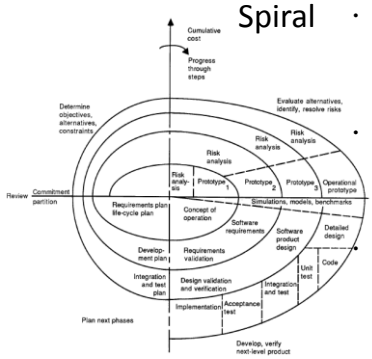
Review

Each cycle is completed by a review involving the primary *people* or organizations concerned with the product.

The plans for succeeding phases may also include a partition of the product into increments for successive development or components to be developed by *individual organizations or persons*.

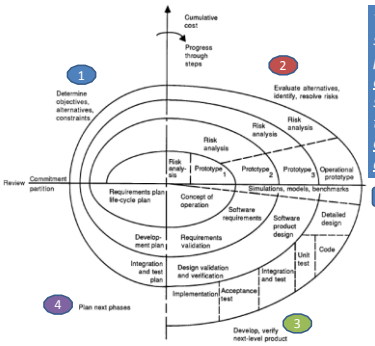


Initiating and Terminating the Spiral



- The spiral gets started by a hypothesis that a particular *operational mission* (or set of missions) could be improved by a software effort.
- The spiral process then involves a test of this hypothesis: at any time, if the *hypothesis fails* the test the spiral is terminated.
- Otherwise, it terminates with the installation of *new on modified software*, and the hypothesis is tested by observing the effect on the operational mission.

Spiral Model



The spiral model is a software development process that uses a risk-driven approach to the software process rather than a primarily document-driven or code-driven process.

Risk resolution

Barry Boehm

Pros & Cons of Spiral Model

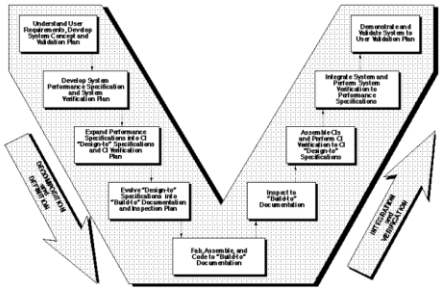
- Very *flexible*
- It is more able to cope with the (nearly inevitable) *changes*
- Takes a *pro-active stance* on risks with explicit risk analysis assessment and resolving stage

- Complex and time consuming
- Only intended for internal projects (inside a company), because risk is assessed as the project is developed.
- Spiral model is risk driven. Therefore it requires *knowledgeable* staff.
- Suitable for only large scale software development. It does not make sense if the *cost of risk analysis* is a major part of the overall project cost.

Vee-Model Introduction [6]

- In real life there is a need to initiate software design and coding, and hardware modeling, earlier in the project cycle to ensure that User Requirements are understood and to prove *technical feasibility*.
- Spiral model attempts to resolve the above deficiency by addressing the need for *early feasibility modeling* ("prototyping") to identify risks and define appropriate action. The *system engineering role* is still obscured.

Vee-Model Overview

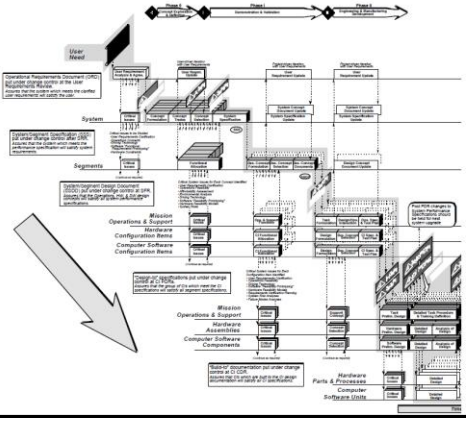


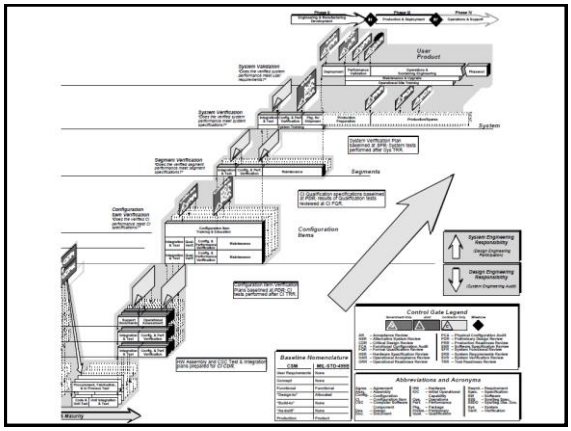
Nine Phases

1. Understand user requirements, develop system concept and validation plan
2. Develop system performance specification and system verification plan
3. Expand system performance specification into CI *"Design-to" specifications* and CI verification plan
4. Evolve "Design-to" specifications into *"Build-to" documentation* and inspection plan
5. Fabricate, assemble and code to "Build-to" documentation
6. Inspect to "Build-to" documentation
7. Assemble CIs and perform CI verification to CI "Design-to" specifications
8. Integrate system and perform system verification to performance specification
9. Demonstrate and validate system to user validation plan

The "Vee" Chart

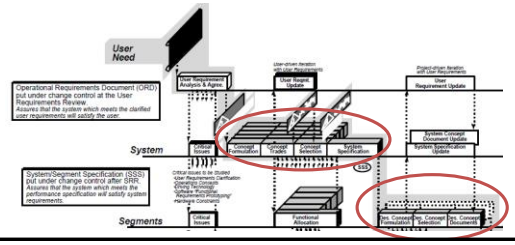
- V-model is a software development model in which the *technical aspect* of the project cycle is envisioned as a "Vee," starting with User needs on the upper left and ending with a User-validated system on the upper right.
- On the left side of the chart, *Decomposition and Definition* descends as in the waterfall model.
- However, *Integration and Verification* flows up and to the right as successively higher levels of assemblies, units, components, and subsystems are verified, culminating at the system level.
- The substantial advance in visualization of the technical aspect of the project cycle, and the role of system engineering, is gained by understanding the *comprehensive "Vee" chart*.





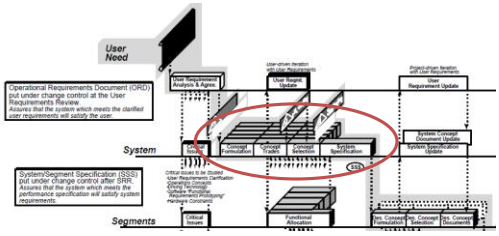
Segments and Configuration Items

- At each level, moving into the depth of the paper (perpendicular to the surface) there are a number of *parallel boxes* illustrating that there may be many *Segments or Configuration Items* that make up the system at that level of decomposition.



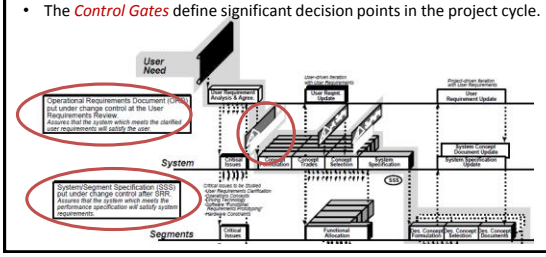
Alternate Concepts

- Also at the System level, on the left of the chart, the number of parallel boxes illustrates that *alternate concepts* should be evaluated to determine the best solution for the User's needs. At the System Requirements Review (SRR), the choice is approved and a single concept is base-lined for further definition.



Baselines and Control Gates

- As project development progresses, a series of *six baselines* are established to systematically manage cohesive system development. Each of the baselines is put under formal Configuration Management at the time they are approved.
- The *Control Gates* define significant decision points in the project cycle.

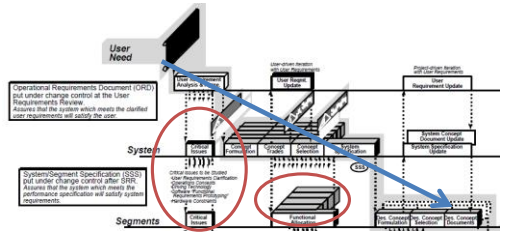


Six Baselines

- "User Requirements Baseline" established by the *System Requirement* Document approved and put under Configuration Management prior to the System Requirements Review (SRR).
- "Concept Baseline" established by the *Concept Definition* section of the Integrated Program Summary document at the SRR.
- "System Performance Baseline" (or Development Baseline) established by the *System Performance Specification* at the System Design Review (SDR).
- "'Design-To' Baseline" (or Allocated Baseline) established at the series of Preliminary Design Reviews (PDRs).
- "'Build-To' Baseline" (or preliminary Product Baseline) established at the series of Critical Design Reviews (CDRs).
- "'As-Built' Baseline" (or Production Baseline) established at the series of Formal Qualification Reviews (FQRs).

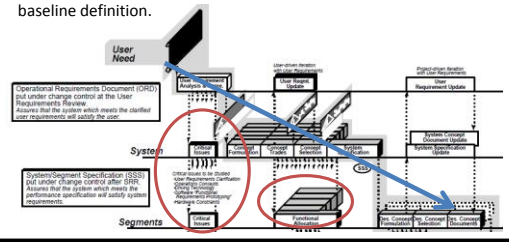
Off-core Activities

- The left side of *the core of the "Vee"* (the shaded area) follows the well-established waterfall model for the project cycle.
- As the project progresses, detailed analyses, risk identification, and risk reduction modeling continues. This is shown on the chart by the vertical and descending *off-core activities*.



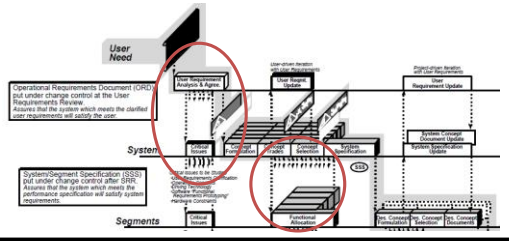
Off-core Activities Repeat

- While technical feasibility decisions are made in the off-core activities only *decisions* at the core-level are put under Configuration Management at the various Control Gates.
- The off-core work is not formally controlled, and will be *repeated* at the appropriate level to prepare justification for introduction into the baseline definition.



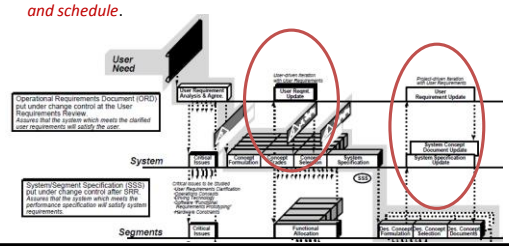
Downward Iteration

- The multiple arrows descending from the bottom of the left side of the core of the "Vee" indicate that there can, and should be, sufficient *iteration* downward to establish feasibility and to identify and quantify risks.



Upward Iteration

- Upward iteration with User Requirements (and levels leading to them) is permitted, but should be kept to a *minimum* unless the user is still generating requirements. The User needs to be cautioned that changes in requirements during the development process will cause positive or negative changes in the predicted *cost and schedule*.

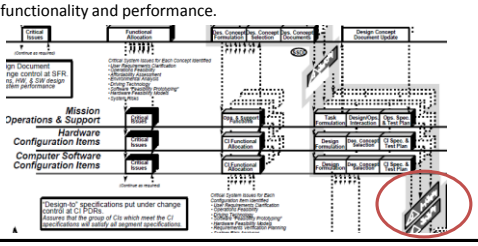


Modification of User Requirements

- Often in software projects upward confirmation of solutions with the User is necessary because User Requirements cannot be adequately *defined* at the start.
- Iteration with User Requirements should be *stopped* at PDR.
- Modification of User Requirements after PDR should be held for the next model or release. If *significant changes* to User Requirements must absolutely be made after PDR, then the project should be *stopped and restarted* at the start of a new "Vee," reinitiating the entire process.
- The repeat of the process may be *quicker* because of the lessons learned, but all steps must be redone.

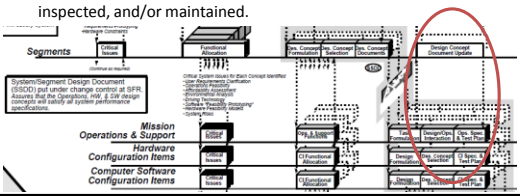
Incremental Development

- If the User Requirements are *too vague* to permit final definition at PDR, one approach is to develop the project in predetermined incremental releases.
- The first release is focused on meeting a *minimum* set of User Requirements, with subsequent releases providing added functionality and performance.



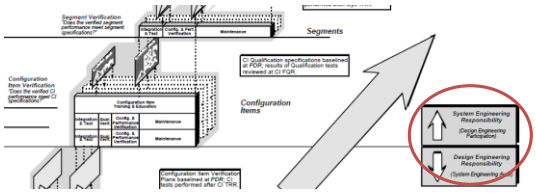
Concurrent Engineering

- If *high iteration* with User Requirements is required after the System Design Review (SDR), it is probable that the project has passed early Control Gates prematurely, and it is not sufficiently defined.
- One cause of premature advance is that the appropriate *technical experts* were not involved at early stages, resulting in acceptance of requirements and design concepts which cannot be built, inspected, and/or maintained.



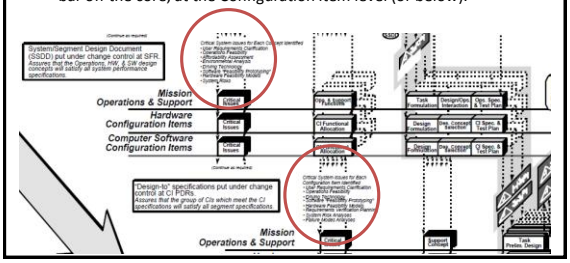
Role of System Engineering

- Above the line *System Engineering* is responsible, and Design Engineering provides technical assistance. Below the line *Design Engineering* is responsible, and System Engineering performs technical audit.
- Note that System Engineering is influential throughout the *entire project* life cycle, from User Requirements development to system decommissioning.



Technology Insertion

- *Technology development* can be done in parallel with the project evolution, and inserted as late as Preliminary Design Review.
- The technology development would be represented by a horizontal bar off the core, at the Configuration Item level (or below).

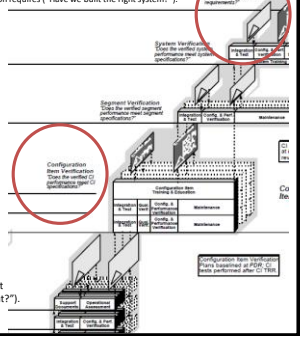


Integration and Verification

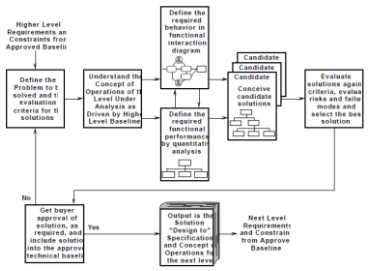
Validation is the process of demonstrating (as opposed to proving) that the product satisfies the *User Needs*, "regardless" of what the system specification requires ("Have we built the right system?").

- Ascending the right side of the "Vee" is the process of *Integration and Verification*.
- At each level there is a direct *correspondence* between activities on the left and right sides of the chart. This is deliberate. The method of verification must be determined as the requirements are developed and documented at each level.

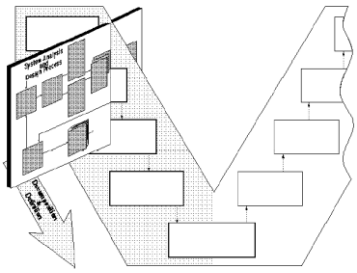
Verification is the process of proving that each product meets its *specification* ("Have we built the system right?").



System Analysis and Design Process

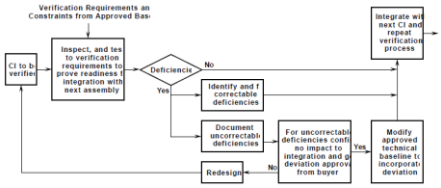


Application of the System Analysis and Design Process to the Technical Aspect of the Project Cycle

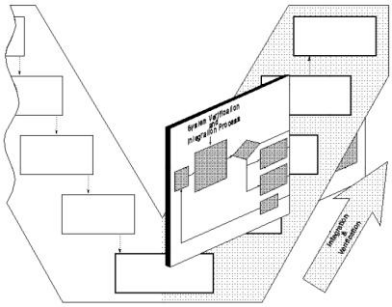


The system engineering process is *repeated* at every level of the cycle, and may be repeated many times within a phase.

System Verification and Integration Process



Application of the System Verification and Integration Process to the Technical Aspect of the Project Cycle



System Engineering

- *System Engineering* can now be more accurately defined as the application of the System Analysis and Design Process and the Integration and Verification Process to the logical sequence of the Technical Aspect of the Project Cycle.
- Emphasize *baseline management* and configuration control that is an essential discipline to good system management.

Thank You For Your Time