LIFE CYCLE MODELS

The Waterfall Software Development Life Cycle Model

Strengths of the Waterfall Model

We can see that the waterfall model has many strengths when applied to a project for which it is well suited. Some of these strengths are:

- The model is well known by non software customers and end-users (it is often used by other organizations to track nonsoftware projects).
- It tackles complexity in an orderly way, working well for projects that are well understood but still complex.
- It is easy to understand, with a simple goal to complete required activities.
- It is easy to use as development proceeds one phase after another.
- It provides structure to a technically weak or inexperienced staff.
- It provides requirements stability.
- It provides a template into which methods for analysis, design, code, test and support can be placed.
- It works well when quality requirements dominate cost and schedule requirements.
- It allows for tight control by project management.
- When correctly applied, defects may be found early, when they are relatively inexpensive to fix.

- Its milestones are well understood.
- It is easy to track the progress of the project using a timeline or Gantt chart the completion of each phase is used as a milestone
- It is easy for the project manager to plan and staff.
- It allows staff who have completed their phase activities to be freed up for other projects.
- It defines quality control procedures. Each deliverable is reviewed as it is completed. The team uses procedure to determine the quality of the system.

Weaknesses of the Waterfall Model

We can also note weakness of the model when it is applied to a project for which it is not well suited:

- It has an inherently linear sequential nature any attempt to go back two or more phases to correct a problem or deficiency results in major increases in cost and schedule.
- It does not handle the reality of iterations among phases that are so common in software development because it is modeled after a conventional hardware engineering cycle.
- It doesn't reflect the problem-solving nature of software development.
- Phases are tied rigidly to activities, not how people or teams really work.
- It can present a false impression of status and progress "35 percent done‟ is a meaningless metric for the project manager.
- Integration happens in one big bang at the end. With a single pass through the process, integration problems usually surface too late. Previously undetected errors or design deficiencies will emerge, adding risk with little time to recover.

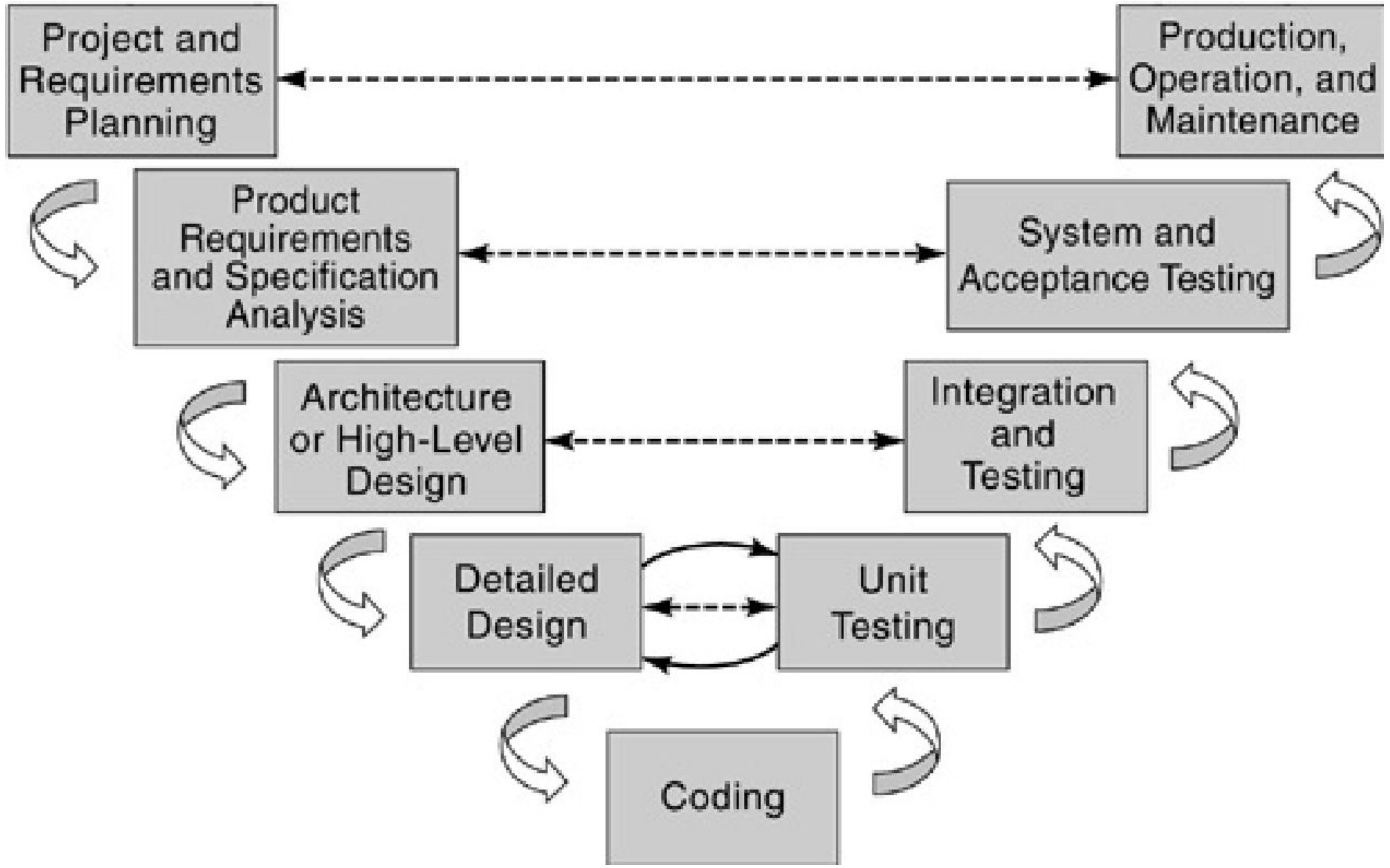
- There is insufficient opportunity for a customer to preview the system until very late in the life cycle. There are no tangible interim deliverables for the customer; user responses cannot be fed back to developers. Because a completed product is not available until the end of the process, the user is involved only in the beginning, while gathering requirements, and at the end, during acceptance testing.
- Users can't see quality until the end. They can't appreciate quality if the finished product can't be seen.
- It isn't possible for the user to get used to the system gradually. All training must occur at the end of the life cycle, when the software is running.

- It is possible for a project to go through the disciplined waterfall process, meet written requirements, but still not be operational.
- Each phase is a prerequisite for succeeding activities, making this method a risky choice for unprecedented systems because it inhibits flexibility.
- Deliverables are created for each phase and are considered frozen that is, they should not be changed later in the life cycle of the product. If the deliverable of a phase changes, which often happens, the project will suffer schedule problems because the model did not accommodate, nor was the plan based on managing a change later in the cycle.
- All requirements must be known at the beginning of the life cycle, yet customers can rarely state all explicit requirements at that time. The model is not equipped to handle dynamic changes in requirements over the life cycle, as deliverables are "frozen." The model can be very costly to use if requirements are not well known or are dynamically changing during the course of the life cycle.
- Tight management and control is needed because there is no provision for revising the requirements.
- It is document-driven, and the amount of documentation can be excessive.
- The entire software product is being worked on at one time. There is no way to partition the system for delivery of pieces of the system. Budget problems can occur because of commitments to develop an entire systemat one time. Large sums of money are allocated, with little flexibility to reallocate the funds without destroying the project in the process.
- There is no way to account for behind-the-scenes rework and iterations.

When to Use the Waterfall Model

- When requirements and the implementation of those requirements are very well understood.
- The waterfall model performs well for product cycles with a stable product definition and well-understood technical methodologies.
- If a company has experience in building a certain genre of system accounting,payroll, controllers, compilers, manufacturing then a project to build another of the same type of product, perhaps even based on existing designs, could make efficient use of the waterfall model.
- Another example of appropriate use is the creation and release of a new version of an existing product, if the changes are well defined and controlled.
- Porting an existing product to a new platform is often cited as an ideal project for use of the waterfall.
- Although the modified waterfall is much more flexible than the classic, it is still not the best choice for rapid development projects.
- Waterfall models have historically been used on large projects with multiple teams and team members.

The V-Shaped Software Development Life Cycle Model


Project and Requirements Planning
Production, Operation, and Maintenance
Product Requirements and Specification Analysis
System and Acceptance Testing
Architecture or High-Level Design
Integration and Testing
Detailed Design
Unit Testing
Coding

STRENGTHS

- The model emphasizes planning for verification and validation of the product in the early stages of product development. Emphasis is placed on testing by matching the test phase or process with the development process. The unit testing phase validates detailed design. The integration and testing phases validate architectural or high-level design. The system testing phase validates the product requirements and specification phase.
- The model encourages verification and validation of all internal and external deliverables, not just the software product.
- The V-shaped model encourages definition of the requirements before designing the system, and it encourages designing the software before building the components.
- It defines the products that the development process should generate; each deliverable must be testable.
- It enables project management to track progress accurately; the progress of the project follows a timeline, and the completion of each phase is a milestone.
- It is easy to use (when applied to a project for which it is suited).

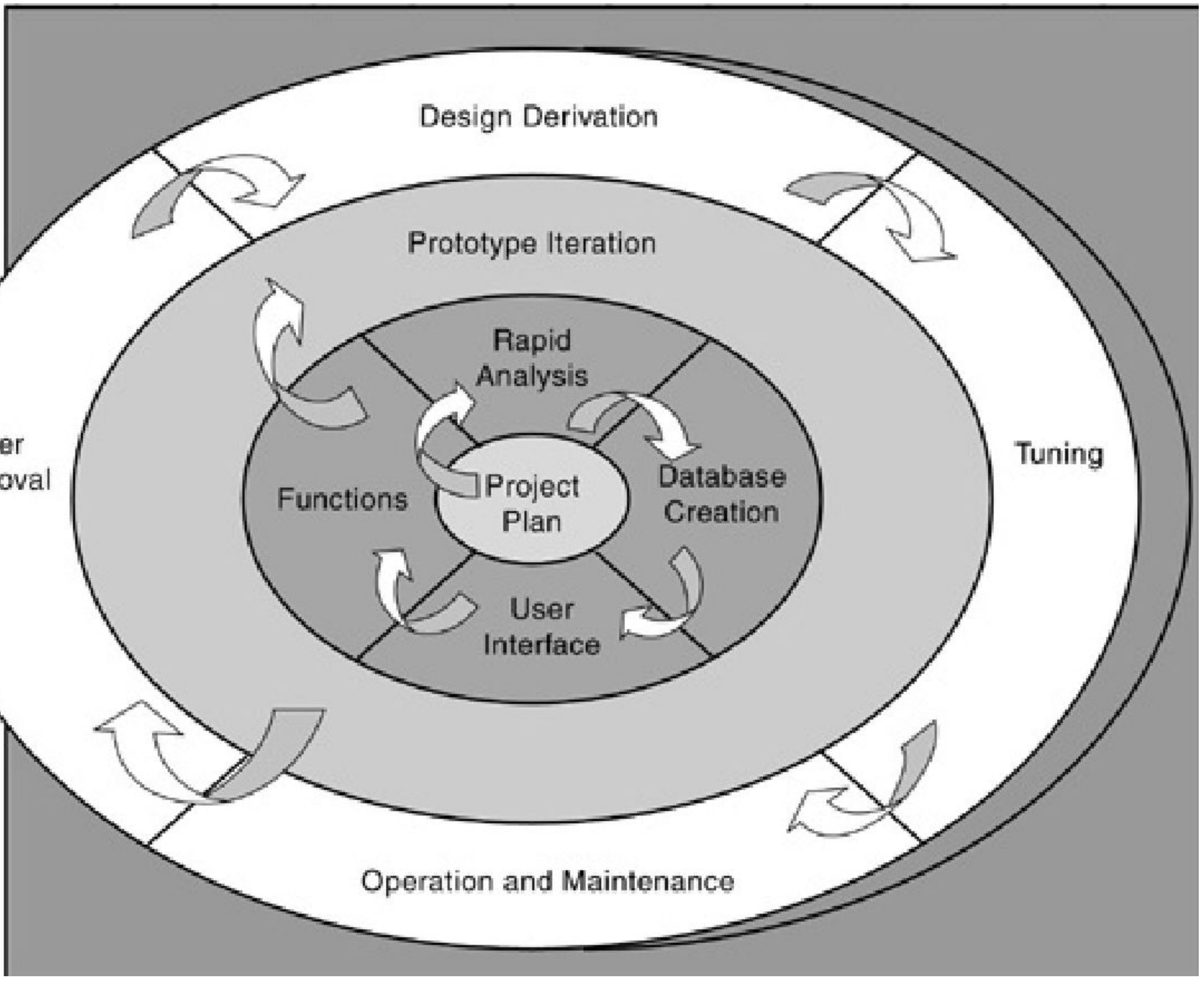
Weaknesses

- When applied to a project for which it is *not* well suited, the weaknesses of the V-shaped model are evident:
- It does not easily handle concurrent events.
- It does not handle iterations of phases.
- The model is not equipped to handle dynamic changes in requirements throughout the life cycle.
- The requirements are tested too late in the cycle to make changes without affecting the schedule for the project.
- The model does not contain risk analysis activities.

When to Use the V-Shaped Model

- Like its predecessor, the waterfall model, the V-shaped model works best when all knowledge of requirements is available up-front. A common modification to the V-shaped model, to overcome weaknesses, includes the addition of iteration loops to handle the changing of requirements beyond the analysis phase.
- It works well when knowledge of how to implement the solution is available, technology is available, and staff have proficiency and experience with the technology.
- The V-shaped model is an excellent choice for systems that require high reliability, such as hospital patient control applications and embedded software for air-bag chip controllers in automobiles.

The Prototype Software Development Life Cycle Model



Strengths of the Structured Evolutionary Rapid Prototyping Model

- The end user can "see" the system requirements as they are being gathered by the project team customers get early interaction with system.
- Developers learn from customers' reactions to demonstrations of one or more facets of system behavior, thereby reducing requirements uncertainties.
- There is less room for confusion, miscommunication, or misunderstanding in the definition of the system requirements, leading to a more accurate end product customer and developer have a baseline to work against.
- New or unexpected user requirements can be accommodated, which is necessary because reality can be different from conceptualization.

- It provides a formal specification embodied in an operating replica.
- The model allows for flexible design and development, including multiple iterations through life cycle phases.
- Steady, visible signs of progress are produced, making customers secure.
- Communications issues between customers and developers are minimized.
- Quality is built in with early user involvement.
- The opportunity to view a function in operation stimulates a perceived need for additional functionality.
- Development costs are saved through less rework.
- Costs are limited by understanding the problem before committing more resources.
- Risk control is provided.
- Documentation focuses on the end product, not the evolution of the product.
- Users tend to be more satisfied when involved throughout the life cycle

Weaknesses of the Structured Evolutionary Rapid Prototyping Model

- The model may not be accepted due to a reputation among conservatives as a "quick-and-dirty" method.
- Quick-and-dirty prototypes, in contrast to evolutionary rapid prototypes, suffer from inadequate or missing documentation.
- If the prototype objectives are not agreed upon in advance, the process can turn into an exercise in hacking code.
- In the rush to create a working prototype, overall software quality or longterm maintainability may be overlooked.
- Sometimes a system with poor performance is produced, especially if the tuning stage is skipped.
- There may be a tendency for difficult problems to be pushed to the future causing the initial promise of the prototype to not be met by subsequent products.

- If the users cannot be involved during the rapid prototype iteration phase of the life cycle, the final product may suffer adverse effects, including quality issues.
- The rapid prototype is a partial system during the prototype iteration phase. If the project is cancelled, the end user will be left with only a partial system.
- The customer may expect the exact "look and feel" of the prototype. In fact, it may have to be ported to a different platform, with different tools to accommodate size or performance issues, resulting in a different user interface.
- The customer may want to have the prototype delivered rather than waiting for full, well-engineered version.
- If the prototyping language or environment is not consistent with the production language or environment, there can be delays in full implementation of the production system.

- Prototyping is habit-forming and may go on too long. Undisciplined developers may fall into a code-and-fix cycle, leading to expensive, unplanned prototype iterations.
- Developers and users don't always understand that when a prototype is evolved into a final product, traditional documentation is still necessary. If it is not present, a later retrofit can be more expensive than throwing away the prototype.
- When customers, satisfied with a prototype, demand immediate delivery, it is tempting for the software development project manager to relent.
- Customers may have a difficult time knowing the difference between a prototype and a fully developed system that is ready for implementation.
- Customers may become frustrated without the knowledge of the exact number of iterations that will be necessary.
- A system may become overevolved; the iterative process of prototype demonstration and revision can continue forever without proper management. As users see success in requirements being met, they may have a tendency to add to the list of items to be prototyped until the scope of the project far exceeds the feasibility study.

- Developers may make less-than-ideal choices in prototyping tools (operating systems, languages, and inefficient algorithms) just to demonstrate capability.
- Structured techniques are abandoned in the name of analysis paralysis avoidance. The same "real" requirements analysis, design, and attention to quality for maintainable code is necessary with prototyping, as with any other life cycle model (although they may be produced in smaller increments).

When to Use the Structured Evolutionary Rapid Prototyping Model

- When requirements are not known up-front;
- When requirements are unstable or may be misunderstood or poorly communicated;
- For requirements clarification;
- When developing user interfaces;
- For proof-of-concept;
- For short-lived demonstrations;
- When structured, evolutionary rapid prototyping may be used successfully on large systems where some modules are prototyped and some are developed in a more traditional fashion;

- On new, original development (as opposed to maintenance on an existing system);
- When there is a need to reduce requirements uncertainty reduces risk of producing a system that has no value to the customer;
- When requirements are changing rapidly, when the customer is reluctant to commit to a set of requirements, or when application not well understood;
- When developers are unsure of the optimal architecture or algorithms to use;
- When algorithms or system interfaces are complex;
- To demonstrate technical feasibility when the technical risk is high;
- On high-technology software-intensive systems where requirements beyond the core capability can be generally but not specifically identified;
- During software acquisition, especially on medium- to high-risk programs;
- In combination with the waterfall model the front end of the project uses prototyping, and the back end uses waterfall phases for system operational efficiency and quality;
- Prototyping should always be used with the analysis and design portions of object-oriented development.

Table 4-1. Selecting a Life Cycle Model Based on Characteristics of Requirements

Requirements	Waterfall	V-Shaped	Prototype	Spiral	RAD	Incremental
Are the requirements easily defined and/or well known?	Yes	Yes	No	No	Yes	No
Can the requirements be defined early in the cycle?	Yes	Yes	No	No	Yes	Yes
Will the requirements change often in the cycle?	No	No	Yes	Yes	No	No
Is there a need to demonstrate the require-ments to achieve definition?	No	No	Yes	Yes	Yes	No
Is a proof of concept required to demonstrate capability?	No	No	Yes	Yes	Yes	No
Do the require-ments indicate a complex system?	No	No	Yes	Yes	No	Yes
Is early functionality a requirement?	No	No	Yes	Yes	Yes	Yes

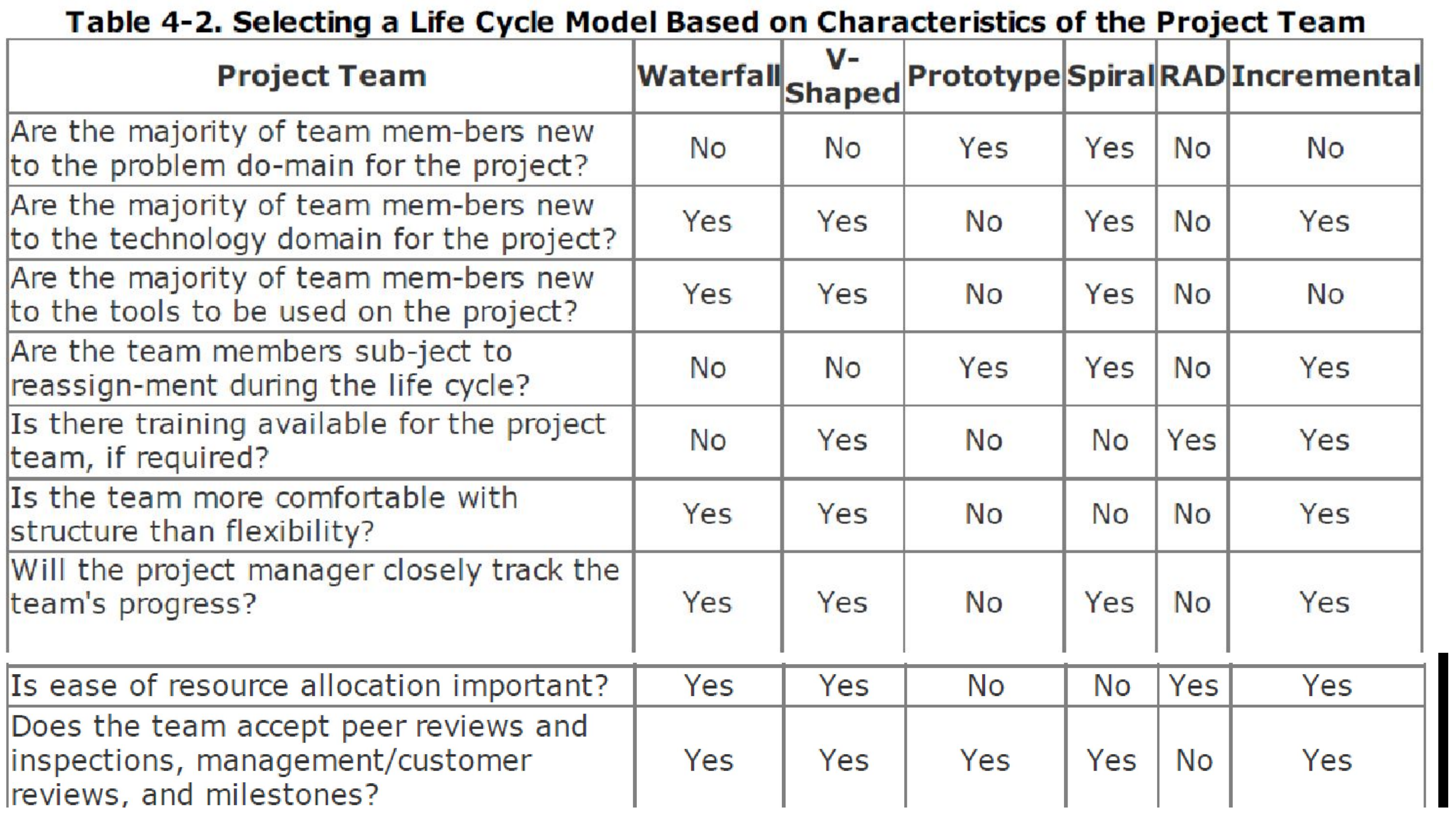


Table 4-2. Selecting a Life Cycle Model Based on Characteristics of the Project Team

Project Team	Waterfall	V-Shaped	Prototype	Spiral	RAD	Incremental
Are the majority of team mem-bers new to the problem do-main for the project?	No	No	Yes	Yes	No	No
Are the majority of team mem-bers new to the technology domain for the project?	Yes	Yes	No	Yes	No	Yes
Are the majority of team mem-bers new to the tools to be used on the project?	Yes	Yes	No	Yes	No	No
Are the team members sub-ject to reassign-ment during the life cycle?	No	No	Yes	Yes	No	Yes
Is there training available for the project team, if required?	No	Yes	No	No	Yes	Yes
Is the team more comfortable with structure than flexibility?	Yes	Yes	No	No	No	Yes
Will the project manager closely track the team's progress?	Yes	Yes	No	Yes	No	Yes
Is ease of resource allocation important?	Yes	Yes	No	No	Yes	Yes
Does the team accept peer reviews and inspections, management/customer reviews, and milestones?	Yes	Yes	Yes	Yes	No	Yes

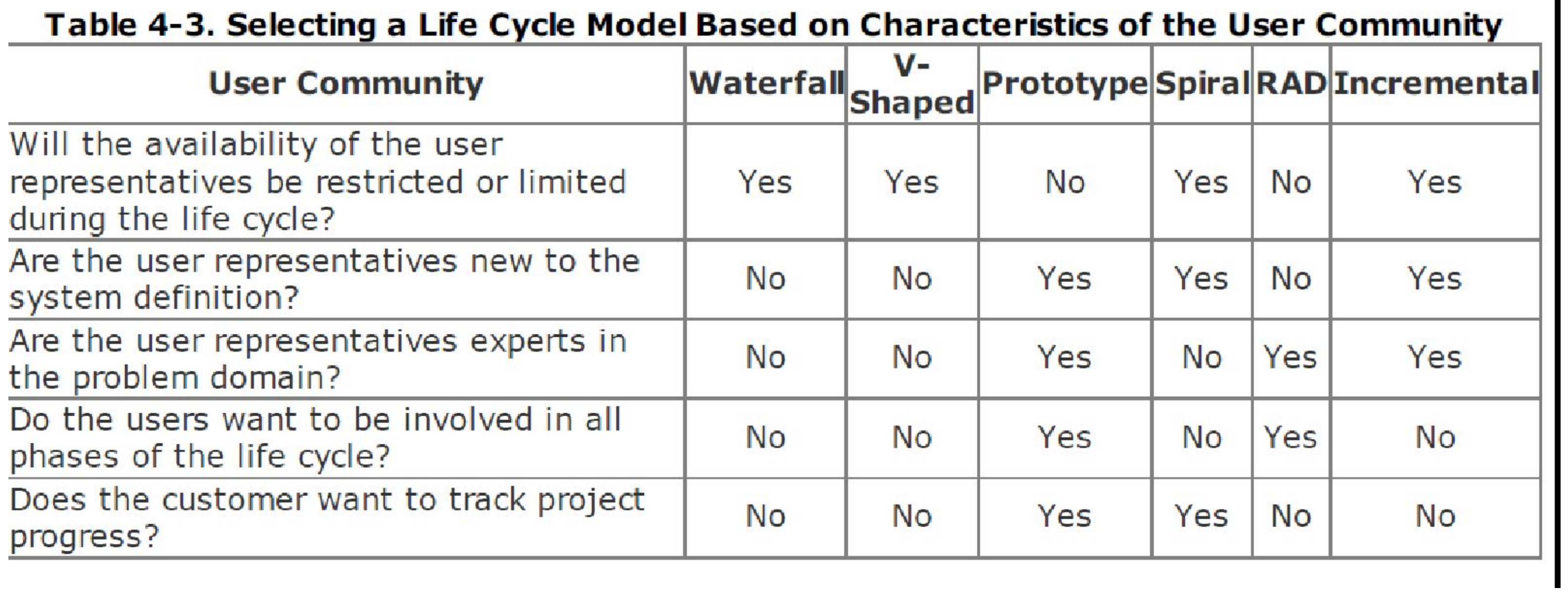


Table 4-3. Selecting a Life Cycle Model Based on Characteristics of the User Community

User Community	Waterfall	V-Shaped	Prototype	Spiral	RAD	Incremental
Will the availability of the user representatives be restricted or limited during the life cycle?	Yes	Yes	No	Yes	No	Yes
Are the user representatives new to the system definition?	No	No	Yes	Yes	No	Yes
Are the user representatives experts in the problem domain?	No	No	Yes	No	Yes	Yes
Do the users want to be involved in all phases of the life cycle?	No	No	Yes	No	Yes	No
Does the customer want to track project progress?	No	No	Yes	Yes	No	No

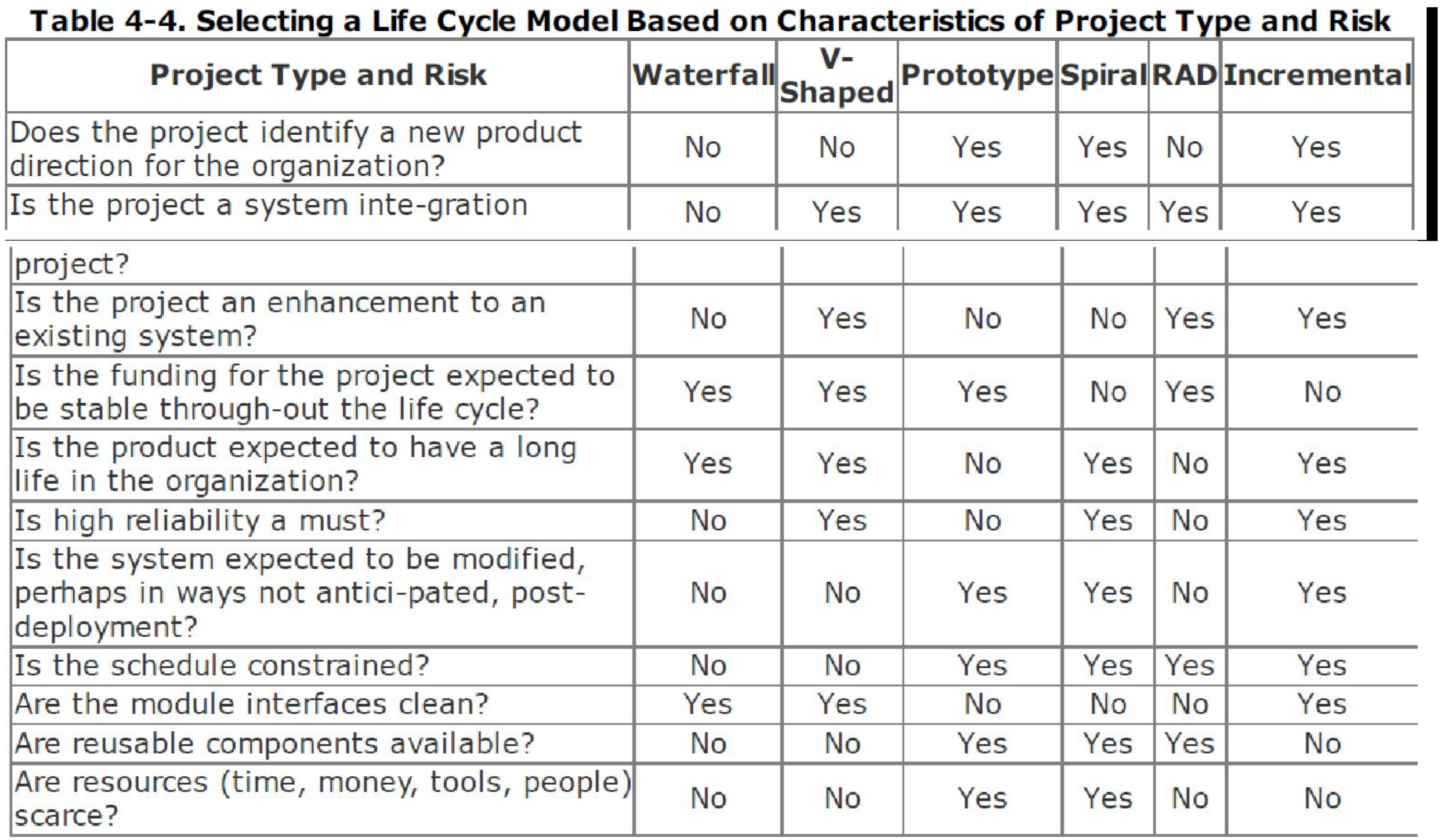


Table 4-4. Selecting a Life Cycle Model Based on Characteristics of Project Type and Risk

Project Type and Risk	Waterfall	V-Shaped	Prototype	Spiral	RAD	Incremental
Does the project identify a new product direction for the organization?	No	No	Yes	Yes	No	Yes
Is the project a system inte-gration project?	No	Yes	Yes	Yes	Yes	Yes
Is the project an enhancement to an existing system?	No	Yes	No	No	Yes	Yes
Is the funding for the project expected to be stable through-out the life cycle?	Yes	Yes	Yes	No	Yes	No
Is the product expected to have a long life in the organization?	Yes	Yes	No	Yes	No	Yes
Is high reliability a must?	No	Yes	No	Yes	No	Yes
Is the system expected to be modified, perhaps in ways not antici-pated, post-deployment?	No	No	Yes	Yes	No	Yes
Is the schedule constrained?	No	No	Yes	Yes	Yes	Yes
Are the module interfaces clean?	Yes	Yes	No	No	No	Yes
Are reusable components available?	No	No	Yes	Yes	Yes	No
Are resources (time, money, tools, people) scarce?	No	No	Yes	Yes	No	No