The Design Phase

Table 2.2: The Design Phase: What are the plans?

Phase	Deliverable
Design	• Architecture Document
	• Implementation Plan
	• Critical Priority Analysis
	• Performance Analysis
	• Test Plan

In the **design phase** the **architecture** is established. This phase starts with the requirement document delivered by the requirement phase and maps the requirements into an architecture. The **architecture** defines the components, their interfaces and behaviors. The deliverable design document is the architecture. The design document describes a plan to implement the requirements. This phase represents the ``how'' phase. Details on computer programming languages and environments, machines, packages, application architecture, distributed architecture layering, memory size, platform, algorithms, data structures, global type definitions, interfaces, and many other engineering details are established. The design may include the usage of existing components. The design phase is summarized in Table 2.2 on page .

The architectural team can now expand upon the information established in the requirement document. Using the typical and atypical scenarios provided from the requirement document, performance trade-offs can be accomplished as well as complexity of implementation trade-offs.

Obviously, if an action is done many times, it needs to be done correctly and efficiently. A seldom used action needs to be implemented correctly, but it is not obvious what level of performance is required. The requirement document must guide this decision process. An example of a seldom used action which must be done with high performance is the emergency shutdown of a nuclear reactor.

Analyzing the trade-offs of necessary complexity allows for many things to remain simple which, in turn, will eventually lead to a higher quality product. The architecture team also converts the typical scenarios into a test plan.

In our approach, the team, given a complete requirement document, must also indicate **critical priorities** for the implementation team. A critical implementation priority leads to a task that has to be done right. If it fails, the product fails. If it succeeds, the product might succeed. At the very least, the confidence level of the team producing a successful product will increase. This will keep the implementation team focused. Exactly how this information is conveyed is a skill based on experience more than a science based on fundamental foundations.

The importance of priority setting will become evident in the theory chapter presented later.

Ronald LeRoi Burback

1998-12-14

Architecture

The architecture defines the components, interfaces, and behaviors of the system.

The components are the building blocks for the system. These components may be built from scratch or re-used from an existing component library. The components refine and capture the meaning of details from the requirement document.

The components are composed with other components using their interfaces. An interface forms a common boundary of two components. The interface is the architectural surface where independent components meet and communicate with each other. Over the interface, components interact and affect each other.

The interface defines a behavior where one component responds to the stimuli of another component's actions.

Ronald LeRoi Burback
1998-12-14

Implementation Plan

The **implementation plan** establishes the schedule and needed resources. It defines implementation details including programming languages, platforms, programming environments, debuggers, and many more.

The implementation plan could be considered as part of the design, which is the position taken here, or it could be considered as the first accomplishment in the implementation phase. One of the goals of the design phase is to establish a plan to complete the system. Thus it is very natural to include the implementation plan. Also, the trade-offs between alternative architectures can be influenced by differences in their implementation plans.

Ronald LeRoi Burback
1998-12-14

Critical Priority Analysis

The **critical priority analysis** generates a list of **critical tasks**. It is absolutely necessary to successfully accomplish a critical task. The project will succeed or fail based on the outcome of these tasks. Some projects may have more than one critical task.

There are two major categories of critical tasks. One category of tasks are associated with the building of the system. These are the critical tasks that the teams must accomplish well. An example might be a high-quality implementation of a critical section of code in the system.

The other category of critical tasks are associated with the system itself. These are the critical tasks that the system, once built, must accomplish well. An example might be the successful flying of an airplane under automatic pilot.

It is absolutely necessary to successfully accomplish both categories of critical tasks.

Not all methodologies have critical priority analysis as a well defined task. Later in the thesis it will be shown that the setting of priorities will play a significant role in methodology's performance characteristics. Critical priority analysis is one of the key features of the WaterSluice software engineering methodology.

Ronald LeRoi Burback
1998-12-14

Performance Analysis

Once given the typical scenarios from the requirement document, the system can be designed to meet performance objectives. Different system architectures will yield different predicted performance characteristics for each typical scenario. Depending on the usage frequency of the scenarios in the system, each architecture will have benefits and drawbacks with advantages and disadvantages. The trade-offs are then weighted to establish the system architecture. Frequently a system is designed to give fast response to an action initiated by a human customer at the expense of having to do more complex systems work such as including indexes, cache management, and predictive pre-calculations.

Ronald LeRoi Burback
1998-12-14

Test Plan

The **test plan** defines the testing necessary to establish quality for the system. If the system passes all tests in the test plan, then it is declared to be complete. If the system does pass all test then it is considered to be of high quality. The more complete the coverage of the system, the higher is the confidence in the system: hence the system's quality rises.

The test plan could be considered as part of the design, which is the position taken here, or it could be considered as the first accomplishment in the testing phase. One of the goals of the design phase, is to establish a plan to complete the system, thus it is very natural to include the test plan. Also the trade-offs between alternative architectures can be influenced by differences in their test plans.

One single test will exercise only a portion of the system. The coverage of the test is the percentage of the system exercised by the test. The coverage of a suite of tests is the union of the coverage of each individual test in the suite.

Ideally, 100 percent test coverage of the entire system would be nice, but this is seldom achieved. Creating a test suite that covers 90 percent of the entire system is usually simple. Getting the last 10 percent requires significant amount of development time.

For an example, consider the Basic Input/Output System (BIOS) built by IBM in the early 1980s as the foundation of the Disk Operating System (DOS) built by Microsoft. For performance reasons, the BIOS needed to be placed in a Read Only Memory (ROM) chip. Because the BIOS would be placed on a ROM, error patches would be nearly impossible. Thus 100% test coverage of the BIOS was dictated. The BIOS code itself was small, only a few thousand lines. Because the BIOS is asynchronous in nature, creating a test would first require an asynchronous environment to bring the system to a desired state, and then an event would be needed to trigger a single test. Quickly, the test suite grew much larger than the BIOS. This introduced the problem of doing quality assurance on the test suite itself. Eventually, 100% coverage was reached but at a high cost. A more cost effective approach would be to place the BIOS on a Electronic Programmable ROM (EPROM) and ROM combination. Most of the BIOS would be on the ROM with error patches being placed on the EPROM. This is the approach that Apple took on the Macintosh.

Usually, it is sufficient that the test suite includes all of the typical and atypical scenarios and need not cover the entire system. This gives reasonable quality for the investment of resources. All the typical and atypical scenarios need to be covered, but in doing so, not all threads of execution within the system may be covered. The system may contain internal branches, errors, or interrupts that will lead to untested threads of execution. Tools exists to measure code coverage.

Systems are full of undiscovered bugs. The customer becomes a logical member of the testing team and bug fixes are pushed off to the next release.

Ronald LeRoi Burback
1998-12-14