Software Paradigms (Lesson 1)

Introduction & Procedural Programming Paradigm

Uddrag fra
https://coronet.iicm.tugraz.at/sa/scripts/lesson01.pdf

Table of Contents

2 Software Engineering and Software Paradigms

The term "software engineering" was coined in about 1969 to mean "the establishment and use of sound engineering principles in order to economically obtain software that is reliable and works efficiently on real machines".

This view opposed uniqueness and "magic" of programming in an effort to move the development of software from "magic" (which only a select few can do) to "art" (which the talented can do) to "science" (which supposedly anyone can do!). There have been numerous definitions given for software engineering (including that above and below).

Software Engineering is not a discipline; it is an aspiration, as yet unachieved. Many approaches have been proposed including reusable components, formal methods, structured methods and architectural studies. These approaches chiefly emphasize the engineering product; the solution rather than the problem it solves.

Software Development current situation:

- People developing systems were consistently wrong in their estimates of time, effort, and costs
- Reliability and maintainability were difficult to achieve
- Delivered systems frequently did not work
    - 1979 study of a small number of government projects showed that:
    - 2% worked
    - 3% could work after some corrections
    - 45% delivered but never successfully used
    - 20% used but extensively reworked or abandoned
    - 30% paid and undelivered
- Fixing bugs in delivered software produced more bugs
- Increase in size of software systems
    - NASA
    - StarWars Defense Initiative
    - Social Security Administration
    - financial transaction systems
- Changes in the ratio of hardware to software costs
    - early 60's - 80% hardware costs
    - middle 60's - 40-50% software costs
    - today - less than 20% hardware costs
- Increasingly important role of maintenance
    - Fixing errors, modification, adding options
    - Cost is often twice that of developing the software
- Advances in hardware (lower costs)
- Advances in software techniques (e.g., users interaction)
- Increased demands for software
    - Medicine, Manufacturing, Entertainment, Publishing
- Demand for larger and more complex software systems

- Airplanes (crashes), NASA (aborted space shuttle launches),
- "ghost" trains, runaway missiles,
- ATM machines (have you had your card "swallowed"?), life-support systems, car systems, etc.
- US National security and day-to-day operations are highly dependent on computerized systems.

Manufacturing software can be characterized by a series of steps ranging from concept exploration to final retirement; this series of steps is generally referred to as a *software lifecycle*.

Steps or phases in a software lifecycle fall generally into these categories:

- Requirements (Relative Cost 2%)
- Specification (analysis) (Relative Cost 5%)
- Design (Relative Cost 6%)
- Implementation (Relative Cost 5%)
- Testing (Relative Cost 7%)
- Integration (Relative Cost 8%)
- Maintenance (Relative Cost 67%)
- Retirement

Software engineering employs a variety of methods, tools, and paradigms.

Paradigms refer to particular approaches or philosophies for designing, building and maintaining software. Different paradigms each have their own advantages and disadvantages which make one more appropriate in a given situation than perhaps another (!).

A method (also referred to as a technique) is heavily depended on a selected paradigm and may be seen as a procedure for producing some result. Methods generally involve some formal notation and process(es).

Tools are automated systems implementing a particular method.

Thus, the following phases are heavily affected by selected software paradigms

- Design
- Implementation
- Integration
- Maintenance

The software development cycle involves the activities in the production of a software system. Generally the software development cycle can be divided into the following phases:

- Requirements analysis and specification
- Design
  - Preliminary design
  - Detailed design
- Implementation
  - Component Implementation
  - Component Integration
  - System Documenting

- Testing
  - Unit testing
  - Integration testing
  - System testing
- Installation and Acceptance Testing
- Maintenance
  - Bug Reporting and Fixing
  - Change requirements and software upgrading

Software lifecycles that will be briefly reviewed include:

- Build and Fix model
- Waterfall and Modified Waterfall models
- Rapid Prototyping
- Boehm's spiral model

2.1 Build and Fix model

This works OK for small, simple systems, but is completely unsatisfactory for software systems of any size. It has been shown empirically that the cost of changing a software product is relatively small if the change is made at the requirements or design phases but grows large at later phases.

The cost of this process model is actually far greater than the cost of a properly specified and designed project. Maintenance can also be problematic in a software system developed under this scenario.

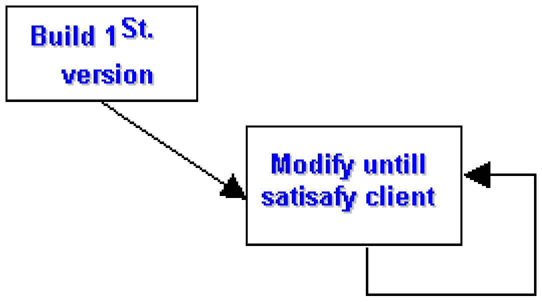

Figure: Build and Fix model

2.2 Waterfall and Modified Waterfall models

2.2.1 Waterfall Model

Derived from other engineering processes in 1970. Offered a means of making the development process more structured. Expresses the interaction between subsequent phases.

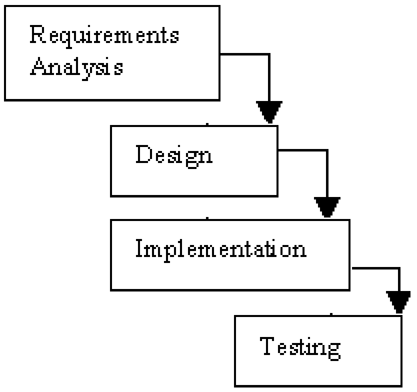

Figure: Waterfall model

Each phase cascades into the next phase. In the original waterfall model, a strict sequentially was at least implied. This meant that one phase had to be completed before the next phase was begun.

It also did not provide for feedback between phases or for updating/re-definition of earlier phases. Implies that there are definite breaks between phases, i.e., that each phase has a strict, non-overlapping start and finish and is carried out sequentially.

Critical point is that no phase is complete until the documentation and/or other products associated with that phase are completed.

2.2.2 Modified Waterfall Model

Needed to provide for overlap and feedback between phases. Rather than being a simple linear model, it needed to be an iterative model. To facilitate the completion of the goals, milestones, and tasks, it is normal to freeze parts of the development after a certain point in the iteration. Verification and validation are added. Verification checks that the system is correct (building the system right). Validation checks that the system meets the users desires (building the right system).

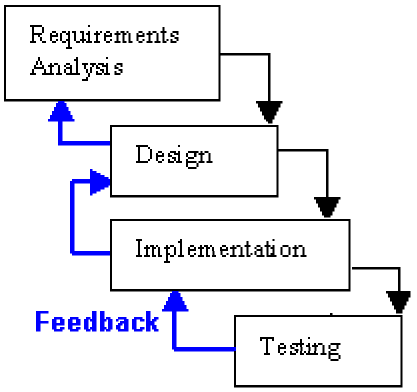

Figure: Modified Waterfall model

The waterfall model (and modified waterfall model) are inflexible in the partitioning of the project into distinct phases. However, they generally reflect engineering practice.

Considerable emphasis must be placed on discerning users' needs and requirements prior to the system being built. The identification of users' requirements as early as possible, and the agreement between user and developer with respect to those requirements, often is the deciding factor in the success or failure of a software project. These requirements are documented in the

requirements specification, which is used to verify whether subsequent phases are complying with the requirements. Unfortunately specifying users' requirements is very much an art, and as such is extremely difficult. Validation feedback can be used to prevent the appearance of a strong divergence between the system under development and the users' expectations for the delivered system.
Unfortunately, the waterfall lifecycle (and the modified waterfall lifecycle) are inadequate for realistic validation activities. They are exclusively document driven models. The resulting design reality is that only 50% of the design effort occurs during the actual design phase with 1/3 of the design effort occurring during the coding activity! This is topped by the fact that over 16% of the design effort occurs after the system is supposed to be completed! In general the behavior of many individuals in this type of process is opportunistic. The boundaries of phases are indiscriminately crossed with deadlines being somewhat arbitrary.

2.3 Rapid Prototyping

Prototyping also referred to as evolutionary development, prototyping aims to enhance the accuracy of the designer's perception of the user's requirements. Prototyping is based on the idea of developing an initial implementation for user feedback, and then refining this prototype through many versions until an satisfactory system emerges. The specification, development and validation activities are carried out concurrently with rapid feedback across the activities. Generally, prototyping is characterized by the use of very high-level languages, which probably will not be used in the final software implementation but which allow rapid development, and the development of a system with less functionality with respect to quality attributes such as robustness, speed, etc.

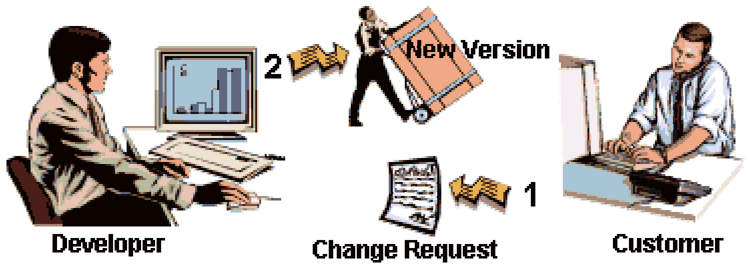

Figure: Rapid Prototyping model

Prototyping allows the clarification of users requirements through, particularly, the early development of the user interface. The user can then try out the system, albeit a (sub) system of what will be the final product. This allows the user to provide feedback before a large investment has been made in the development of the wrong system.

There are two types of prototypes:

- Exploratory programming: Objective is to work with the user to explore their requirements and deliver a final system. Starts with the parts of the system which are understood, and then evolves as the user proposes new features.
- Throw-away prototyping: Objective is to understand the users' requirements and develop a better requirements definition for the system. Concentrates on poorly understood components.

Experiments with prototyping showed that this approach took 40% less time and resulted in 45% less code; however, it produced code which was not as robust, and therefore more difficult to maintain. Documentation was often sacrificed or done incompletely. The schedule expectations of users and managers tended to be unrealistic especially with respect to throw-away prototypes.