ECSE 429 Study guide

Francis Piché

September 23, 2019

Contents

1 License Information

These notes are curated from Professor Kataryzna Radecka lectures at McGill University. They are for study purposes only. They are not to be used for monetary gain.

Part I
Introduction

2 Idealized Tests

There is no single testing method to cover all situations. This would be an ideal test, that is, one that:

- Detects any and all defects
- Pass all good devices

The challenges when trying to design idealized tests are that there is a very large number of defects to be tested, of different types, and not all defects are testable (due to the way the system interfaces are designed, dependence on external factors etc).

There are external dependencies to almost any program, whether that be the hardware, the OS the program is running in, applications yours depends on, interacts with, or libraries your program utilizes. An ideal test would have to cover cases of failure in ALL of these dependencies, which with modern systems is infeasible.

3 What Does Testing Do?

With increased reliance on software systems, we must be able to have confidence in mission-critical pieces of software that govern our lives. Improper or lack of testing can lead to massive losses.

Traditionally and in hardware, testing has the following roles:

- Determine if the device has faults
- Diagnose which faults occurred
- Determine and correct errors in design or testing

- Failure mode analysis: What could have caused the defects in the devices

More specifically for software, testing has 3 responsibilities:

1. Does the software satisfy the requirements
2. Does the software perform adequately
3. Does the software have defects?

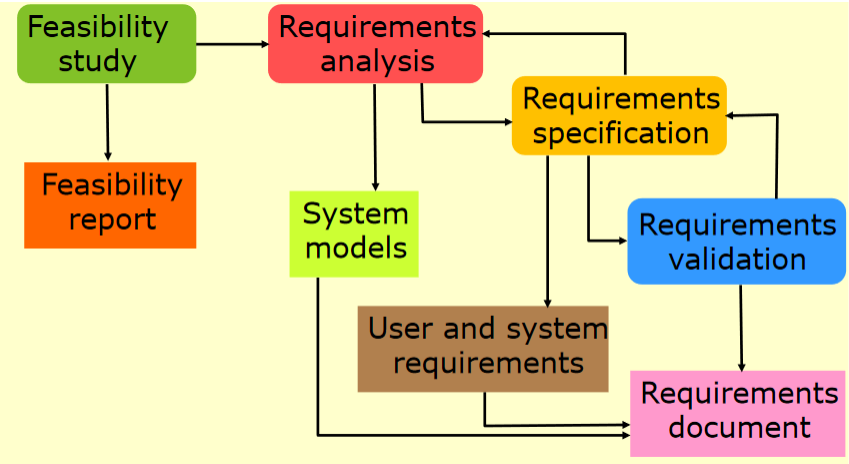
4 Software Development Processes

A software development process is a structured plan or set of activities to develop a software system. It has the task of specifying requirements, designing, validating and evolving the system further.

4.1 Specification Phase

The specification phase of software development is where we must establish what the system must be able to do, the constraints under which the system is to be built and operated. To determine this, there are several phases involved:

- Determine if the requirements are feasible
- Analyze requirements
- Formally specify requirements
- Validate requirement specification matches true requirements



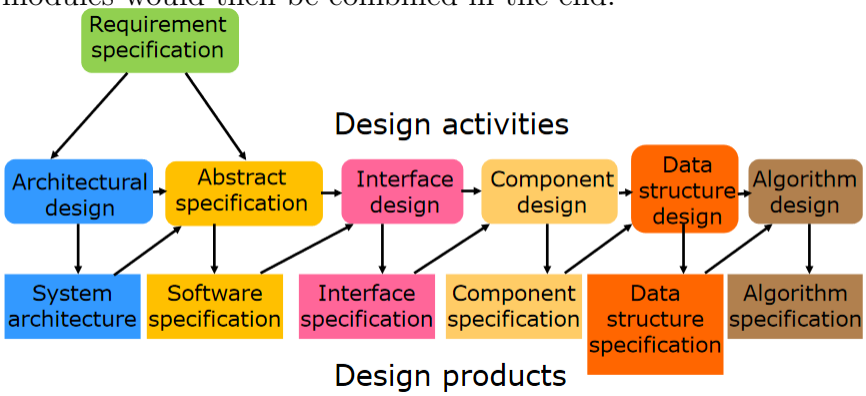
4.2 Design Phase

The design phase is where the system is built "in-theory" to a level of detail which proves it can match the requirements determined in the specification phase.

Phases:

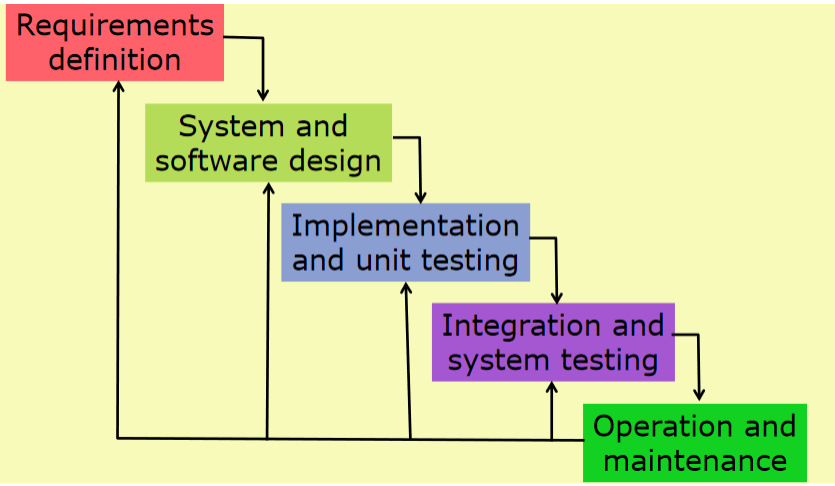
- Data design : How the data will be structured
- Architectural Design : How the structure of the application will be laid out
- Interface Design : How the components will communicate
- Procedural Design : Specify how the exection flow will work
- Algorithm Design : Low-level design of individual algorithms used

To accomplish this there are two common approaches: *refinement* in which the entire system is described in increasing levels of detail, and *Modularity* where the software is divided into conceptual (and functional) modules which can be developed independently. These modules would then be combined in the end.



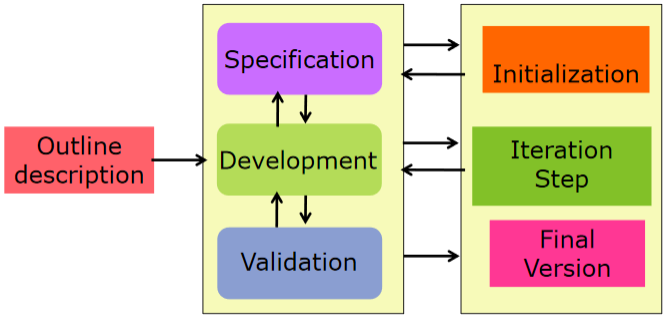
4.3 Generic Software Process Models

There are several generified models for designing software. The classic is the *Waterfall Model*. In this method, the phases are distinct and disjoint. First specify requirements, then implement, test etc. There is never any "backtracking" to previous phases.



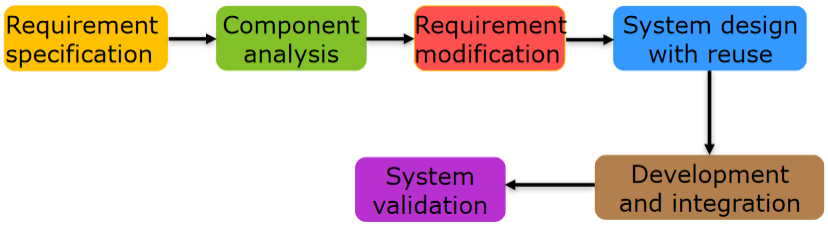
The advantage to this model is that if a design cannot be implemented for any reason, this will be detected and corrected far before time and money has gone into the implementation. This was especially useful for hardware, where manufacturing is also involved. The disadvantage is that its very difficult to say a phase is "completed" and never return, as this requires full-knowledge of the requirements, limitations and changes to the requirements are limited or non-existent.

In the *evolutionary* model interleaves the phases and iteratively improves the software by adding more and more requirements.



The advantage to this is that problems can be addressed early, requirements are more flexible, and feedback from the stakeholders can be leveraged to make course-corrections. However, the disadvantage is that it can involve a lot of re-writing of "poorly-designed" components, and often it is difficult to make the early stages "generic" enough to be flexible for later changes. This strategy is best used for small to medium sized interactive systems, or for parts of a larger system.

In *component* based model the software is built by plugging several existing components together.



. Here the advantage is that often it can be assumed that the components built can be trusted, and so their inner-workings do not need to be tested. Instead, only interactions between components is to be tested. Using "off-the-shelf" solutions also saves development time, and (depending on the cost of the module) costs.

5 Quality

Software quality is the degree to which a system meets specific requirements, customer needs and expectations.

The environment in which the software is developed brings many challenges for quality assurance. The features requested, budgeting, timeline, customer relation, team dynamics, etc.

Software Quality Factors:

- Usability: How is the user experience?
- Maintainability: Find and fix bugs
- Flexibility: ability to add new features
- Testability
- Portability: adaptation to execution environments
- Reusability of components
- Interoperability with other components/systems
- Robustness, performance, understandability etc.

5.1 Quality Assurance and Control

Quality Assurance aims to plan and minimize cost of guarenteeing quality, while *quality control* is a set of activities to implement quality assurance principles.

The cost of independent contractors for QA is often high. However, forming QA teams within a company is often done to minimize cost and maximize effectiveness, since the team will form a deeper knowledge of your software. These teams aim to know what the software is doing, what it should be doing, and how to measure the difference. This is accomplished

through being involved with development teams, business teams, and using methods to measure software performance:

- Formal methods: mathematically verification
- Testing: explicit input/ouput to the software and check results
- Inspections: Human interaction to check requirements, design, code etc
- Metrics: data generated about the program to analyze quality

Part II
Testing and Verification Terminology

6 Validation vs Verification vs Testing

Validation is the process of evaluating software to ensure compliance with intended usage. "Did we build the *right* thing?"

Verification is the process of determining if products of a given phase of the software development cycle fulfills the requirements of the previous phase. "Did we build the thing *correctly*?"

Validation is more than just testing. It must demonstrate reliability, ensure compliance, generate knowledge about the application, and establish future requirements.

There are several techniques of validation:

- Formal Methods: Math based, often not possible
- Fault Injection: Intentional, methodical injection of faults to see how the system react and verify it matches the expected behavior.
- Dependency Analysis: Identification of hazards and proposing methods of how to get around them.

7 Testing vs Debugging

Testing is finding the existence of bugs, failures and defects. This job is handled by a QA team.

Debugging is finding the cause, analyzing and correcting the bugs. Done mostly by development team.

8 Faults, Failures and Errors

Often it is not possible to establish that software is correct under all conditions due to application complexity. Therefore we must rely on showing that software fails under more specific condition. The problem then lies in which conditions we simulate with the system to target the majority of all possible cases.

8.1 Errors

An **error** is defined as an incorrect internal state that is the manifestation of some fault.

This fault must change the operation to be performed by a statement without changing the execution flow, OR, one that changed the execution flow.

However the software may still execute correctly even if it is Erroneous. (Redundant code, dead code etc.)

8.2 Faults

Faults are only design faults when talking about software. Whether it was mistyped or misunderstood, it is a fault. The same is not true for hardware where faults can be caused by outside influences.

A fault is *active* if it causes an error, otherwise it is *dormant*. *Fault activation* is when dormant fault becomes active through some change to the execution inputs or context.

8.2.1 Fault Causes

Faults can often be traced to a single root cause, whether that be software or hardware. This cause is not always logical errors in code or defects in hardware. The causes can come from:

- Failures in Communication
- Improper Resource management
- Inadequate Verification
- Bad Risk management
- Faulty Assumptions

8.3 Failures

An error which causes incorrect behavior with respect to the requirements specification. Software fails due to two things:

- Differ from specification
- Specification does not describe intended behavior.

Failure of some software component causes a **permanent fault** in software that uses that component. Failure of the whole software leads to a **permanent external fault**.

8.4 Defects

A defect is anything that produces wrong results causing failure. A single defect can cause a wide range of failures, but not all defects result in failure. Kind of like a fault.

So all in all, faults activate errors which propagate to failures. Failures are caused ultimately by faults.

There are 9 major causes of defects:

1. Faulty requirements definition
2. Client-developer communication failure
3. Deliberate deviation (in development or usage of system)
4. Design logic errors
5. Coding errors
6. Non-compliance with documentation of code
7. Improper testing
8. Documentation errors
9. Procedure errors (misusing a UI etc)

9 The 7 Testing Principles

9.1 Bug Presence over Absence

The purpose of testing is not to show the absence of buggs, but rather to show the presence of bugs.

If no bugs are found this does not mean that the software is correct.

Of course this doesn't mean the software is bad no matter what, its just that we did not stretch the software to its limit during testing.

9.2 Exhaustive Testing is Impossible

There is often very large number of possible inputs, outputs, states, execution paths for a piece of software, which makes it practically impossible to exhaustively test. Further, specifications may be ambiguous or subjective, leading to multiple "correct" implementations.

Clearly, if the function execution time is long, then it is infeasible to do many inputs.

Example:

```
for (int i=0; i < n; i++){
   if (some cond){
      do one thing
   }else{
      do another thing
   }
}
```

For each loop, there is 2 possible paths. So for n=1 there are 3 paths, n=2 5 paths, and so on. So there are $2^n + 1$ possible paths which quickly grows impossible to exhaustively test.

9.3 Defect cost is exponential

The later defects are caught and fixed, the more they cost. It turns out this is generally exponential.

9.4 Defects Cluster Together

Generally defects are not distributed evenly across the application. This means that the tester can focus on specific key areas.

9.5 Software Becomes Resistant to Testing

After several test iterations the software will be revised to make the tests pass. Thus the tests must always also be revised and improved to try and find more bugs.

9.6 Testing is Content Dependent

Different systems are tested differently. This means that something mission-critical like a piece of military software must be more rigorously tested than something like a cat picture web scraper.

9.7 Absence of errors is a Fallacy

Finding and fixing defects will not make your application "good" if its unusable or doesn't fullfil the users requirements/expectations, it's still bad software.

10 Four Types of Testing Activities

- Test Design: Select appropriate test method based on characteristics of the software
- Test Automation: Tools for making testing feasible and repeatable
- Test Execution: Run the test cases
- Test Evaluation: Check test coverage criteria to estimate the quality of the test and reliability of the results.

11 IQ, QQ, PQ

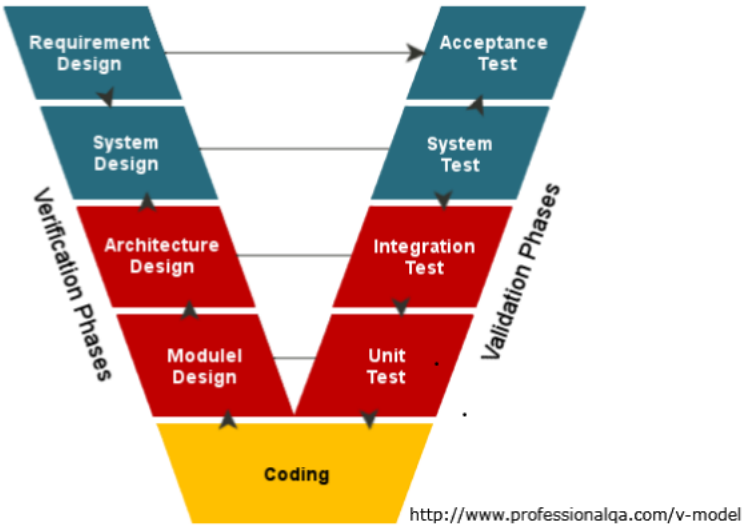
TODO

- IQ: Installation Qualification
- OQ: Operational Qualification
- PQ: Performance Qualification

12 V Model

The V-model is is a verification and validation model that extends to the waterfall design model.

It matches each design phase with a verification/validation counterpart.


Requirement Design
Acceptance Test
System Design
System Test
Architecture Design
Integration Test
Modulel Design
Unit Test
Coding
Verification Phases
Validation Phases
http://www.professionalqa.com/v-model