Examples of spectacular software failure

The last three examples from *Software Engineering*, by Hans Van Vliet.

- Blackout Summer 2003.

  The big northeast blackout from the 2003 summer was caused in part by a software bug in an energy management system sold by General Electric, according to a story on SecurityFocus.

  The bug meant that a computerized alarm that should have been triggered never went off, hindering FirstEnergy's response to the train of events that lead to the cascading blackout.

  Investigators found the bug in an intensive code audit following the outage, and a patch is now available.

- Ariane 5, Flight 501.

  The maiden flight of the Ariane 5 launcher took place on June 4, 1996. After about 40 sec, at an altitude of 4 km, the launcher broke up and exploded.

  This $500 million loss was ultimately caused by an overflow in the conversion from a 64-bit floating point number to a 16-bit signed integer.

  There are several levels at which the Ariane 5 failure can be understood and explained: (i) It was a software failure which could have been revealed with more extensive testing/verification, (ii) the failure was caused by reusing a flawed component, and (iii) the failure was caused by a flaw in the design.

- The Therac-25 is a computer-controlled radiation machine.

  One of its malfunctions has become known as "Malfunction 54." The operator would type 'x' (for X-ray) instead of 'e' (for electron mode) at the console.

  The machine would issue the message "Malfunction 54" indicating a wrong dose (too high or too low). Since the machine had "quirks" the operator would just proceed.

  It was later estimated that the patient received 16,000–25,000 rad on a very small surface, instead of the intended dose of 180 rad.

  The patient died five months later after becoming seriously ill.

  Bad software was the single point of failure.

- London Ambulance Service

  LAS handles the ambulance traffic in greater London. Carries 5000 patients per day in 750 vehicles.

  A computer-aided dispatch (CAD) system was ordered to handle this. The system was delivered in January 1992, still far from being complete.

  In November, the system crashed. An investigation was started.

  Not only was the program badly and naïvely designed, it allocated a small amount of memory every time a vehicle mobilization was generated by the system. This memory was not deallocated ...

  The system had other serious flaws: it would allocate the nearest ambulance regardless of the originating station, so it would happen that crews would operate further and further from their home base and in unfamiliar territory, taking wrong turns ...

Desirable Program Characteristics

This, and the remaining sections, come from *Java Software Solutions*, by Lewis and Loftus.

- Integrity
- Clarity
- Simplicity
- Efficiency
- Modularity
- Generality

Integrity. This refers to the accuracy of the calculations. It should be clear that all other program enhancements will be meaningless if the calculations are not carried out correctly. Thus, the integrity of the calculations is an absolute necessity in any computer program.

Eg. 1 See Ariane 5 on the previous page.

Eg. 2 Kruskal's Algorithm *must* compute a (i) spanning tree of (ii) minimal cost.

Clarity refers to the overall readability of the program, with particular emphasis on its underlying logic. If a program is clearly written, it should be possible for another programmer to follow the program's logic without undue effort.

One of the objectives in the design of C is the development of clear, readable program through an orderly and disciplined approach to programming.

Simplicity. The clarity and accuracy of a program are usually enhanced by keeping things as simple as possible, consistent with the overall program objective. In fact, it may be desirable to sacrifice a certain amount of computational efficiency in order to maintain a relatively simple, straightforward program structure.

Efficiency is concerned with execution speed and efficient memory utilization. These are generally important goals, though they should not be obtained at the expense of clarity or simplicity.

Eg. We should use the greedy algorithm for finding the minimal cost spanning tree, rather than, say, a brute force search algorithm.

Modularity. Many programs can be broken down into a series of identifiable subtasks. It is good programming practice to implement each of these subtasks as a separate program module. In C such modules are written as functions. The use of a modular programming structure enhances the accuracy and clarity of a program, and it facilitates future program alterations.

Generality. Usually we want a program to be as general as possible, within reasonable limits.

Eg. we want to input `N`, and assign an array of appropriate size, rather than define `N` to be a particular value.

Developing Programs

Four basic activities of any software development effort:

1. establishing the requirements
2. **creating a design**
3. implementing the code
4. testing the implementation

These activities overlap and interact.

Requirements specify what a program must accomplish—they indicate the tasks that a program should perform, but not how to perform them.

An initial set of requirements are often provided by the person or group who want the software developed. However, these initial requirements are often incomplete or ambiguous.

The software developer must work to refine the requirements until all key decisions have been addressed.

For example, what type of "malformed" inputs to test for?

Design specifies how a program will accomplish its requirements.

One of the most fundamental design issues is defining the algorithm to be used in the program. An algorithm is often described using pseudo-code.

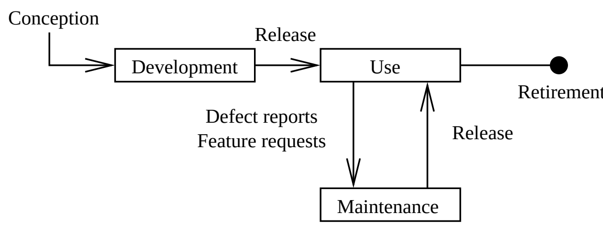
For example, Kruskal's algorithm in our design for the program computing minimum cost spanning trees.

Implementation is the process of writing source code: it is the translation of design into a particular programming language (C in this course).

Too many programmers focus on implementation exclusively when it actually should be the *least creative* of all activities.

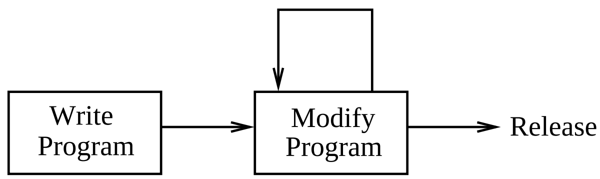
Testing a program involves running it multiple times with various inputs and observing the results. The goal of testing is to find errors.

Software Life Cycle



Development Process Models

The build-and-fix approach

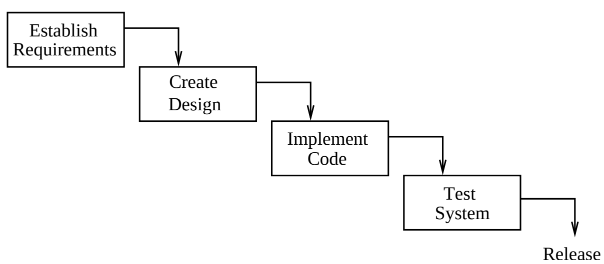


Programmer creates an initial version of the program, then continually modifies it until it has reached some level of acceptance.

Programmer is simply reacting to problems, as opposed to creating something worthy in the first place (i.e., *reactive* rather than *proactive* thinking).

Bad, *ad hoc*[1], reckless .

The (unrealistic) waterfall model



First offered in the early 1970s.

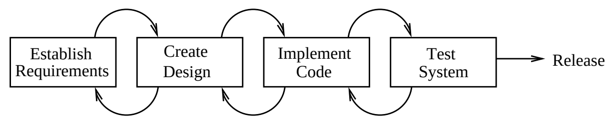
Linear: it does not allow for revisiting earlier stages.

Tremendous effort must go into each stage to ensure its completeness: unrealistic.

Does not take into account that the developmental activities are overlapping.

[1] Works for a special case only, without general application.

An iterative process

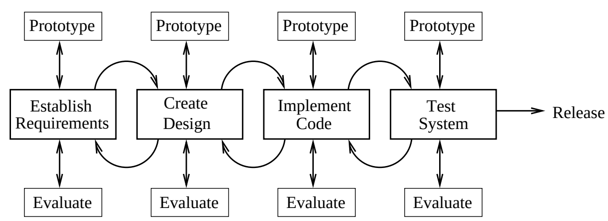


Earlier stages can be revisited (formally): waterfall model with backtracking.

Danger: rely on backtracking too much!

Backtracking should be used primarily to correct problems uncovered in later stages.

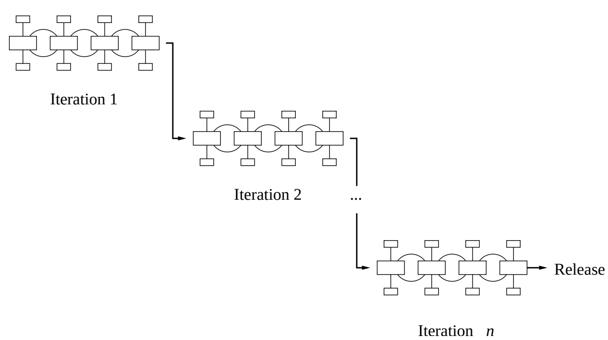
Iterative process with Prototyping



A prototype is a program that is created to explore a particular concept.

Sometimes a programmer doesn't know how to accomplish a particular task, or weather certain requirements are feasible, or the user interface acceptable to the client. Prototypes can be used to explore all these issues.

Iteration of the entire process



The entire model can be used to develop a program iteratively.

One iteration of the model is used to build a specific part of a program.

Testing Concepts

- Defect testing
- Black box testing
- Equivalence category
- Glass box testing

Defect testing is aimed at uncovering problems in a program.

The goal is to find errors.

Black box testing assumes no knowledge of how a given program is implemented. Test cases are often derived from the requirements.

An **equivalence category** is a collection of inputs that are expected to produce similar outputs. Should select test cases from different equivalence categories.

Glass box testing assumes knowledge of the internals of the program, and different test cases trace the different paths through the code.