Object-Oriented Software Engineering

WCB/McGraw-Hill, 2008
Stephen R. Schach
srs@vuse.vanderbilt.edu

Modifications by Jules White

Why Do We Study Software Engineering?

- What is the difference between SE and coding?
- Why don't we just study more algorithms or design techniques instead?
- Isn't SE a technical problem?

Historical Aspects

- 1968 NATO Conference, Garmisch, Germany
- Aim: To solve the *software crisis*
- Software is delivered
  - Late
  - Over budget
  - With residual faults

SE in 1968

- "One survey lists *nearly* 500 organizations concerned with producing or selling software.."
- Software still came free with the hardware
- Other than Fortran, high-level languages were only used on mainframes
- State of the art languages: Fortran, Algol, Cobol, PL/I
- Whether or not high-level languages could be used for systems programming was still being debated
- "…the number of systems labeled time-sharing has increased from around 5 experimental systems to around 30 commercial systems.."
- First paper mentioning the "process theory of programming"

- Randell, B. 1979. Software engineering in 1968. In *Proceedings of the 4th international Conference on Software Engineering* (Munich, Germany, September 17 - 19, 1979). International Conference on Software Engineering. IEEE Press, Piscataway, NJ, 1-10.

Where Are We Now?

- 2004, in just Mass., 2,781 software companies with 118,000 employees
- Last I checked, Adobe Creative Suite costs ~1,000$ .. doesn't come with hardware
- Assembly programming is a punishment only inflicted on undergrads
- State of the art languages: Java, C#, PHP, Perl, Python
- You will be laughed at if you argue that your company should implement its next web service in assembly
- If we count Web 2.0 apps as time-sharing…I can't count all the time-sharing systems
- Even my mom knows how to kill a "process" in Windows

- Randell, B. 1979. Software engineering in 1968. In *Proceedings of the 4th international Conference on Software Engineering* (Munich, Germany, September 17 - 19, 1979). International Conference on Software Engineering. IEEE Press, Piscataway, NJ, 1-10.

How Have We Done Since Garmish?

- Successes
  - Huge scale
    - » GMail has 5,000,000+ accounts at 7057mb per account…isn't this a massive distributed time-sharing system?
  - With Java, C++, C#, OO is the standard
    - » The new debate is on more advanced language features, such as closures
  - Higher levels of abstraction
    - » Domain-specific languages (DSLs)
  - Software everywhere
    - » High-end BMWs have ~80 small CPUs
    - » Every cell phone
  - Distribution the norm –> ***the web***
    - » Even cars use distributed systems

How Have We Done Since Garmish?

- We can certainly build bigger and more complex systems
  - Some high-end TVs have ~1,000,000 lines of code
  - This is more than the first moon landing
    - Allan Klump was one of about 300 people who designed the LM's software over a 7 year period for 46 million 1967-era dollars. He did his work as a graduate student at the MIT Draper Lab during the Apollo years.
    - Compiled the software BY HAND
    - Software had to fit into 36K
  - So…why isn't Samsung writing the software for the next moon lander?

How is Complexity Affecting Us?

- All the new technologies have helped us build bigger systems – but they haven't improved the chance of project success
- We may be worse off than before:
- The belief that complex systems require armies of designers and programmers is wrong. A system that is not understood in its entirety, or at least to a significant degree of detail by a single individual, should probably not be built. –Niklaus Wirth
- The price of reliability is the pursuit of the utmost simplicity. It is a price which the very rich may find hard to pay. –C.A.R Hoare
- There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.--*C.A.R. Hoare*

Standish Group Data

- Data on 9236 projects completed in 2004

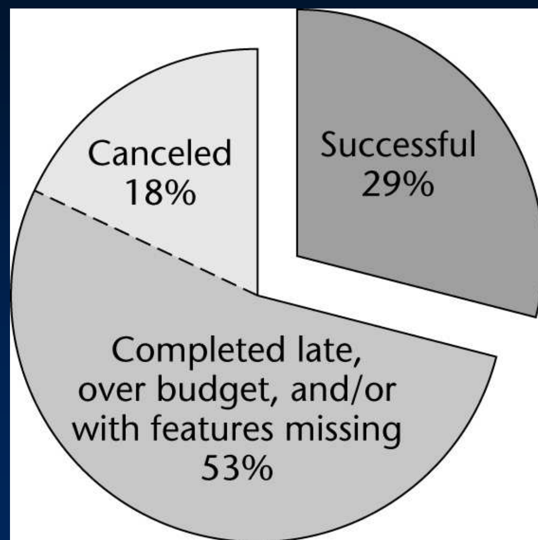


Figure 1.1

What this Means for You

Chance that a year or more of your life is wasted on coding something that is thrown away

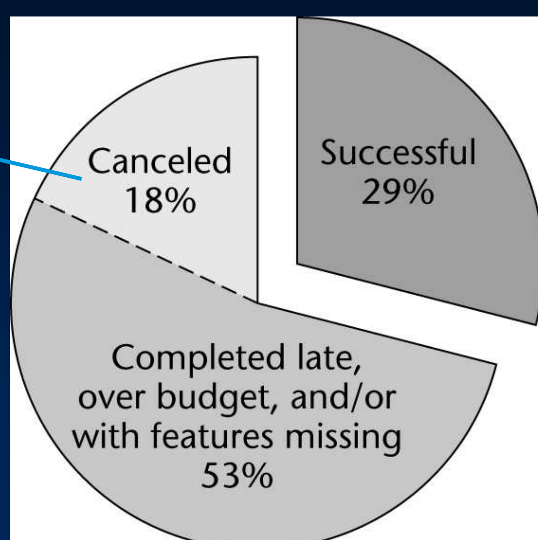


Figure 1.1

What this Means for You

Chance that you will be forced to work long hours, weekends, and that moral will be low

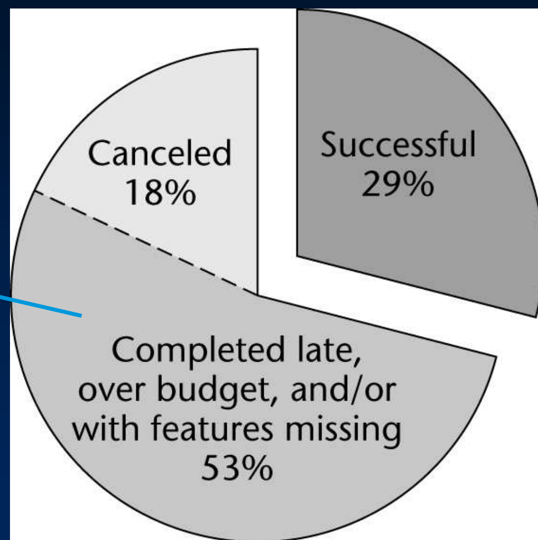


Figure 1.1

What this Means for You

IBM only certifies architects that have worked on a project considered “troubled”

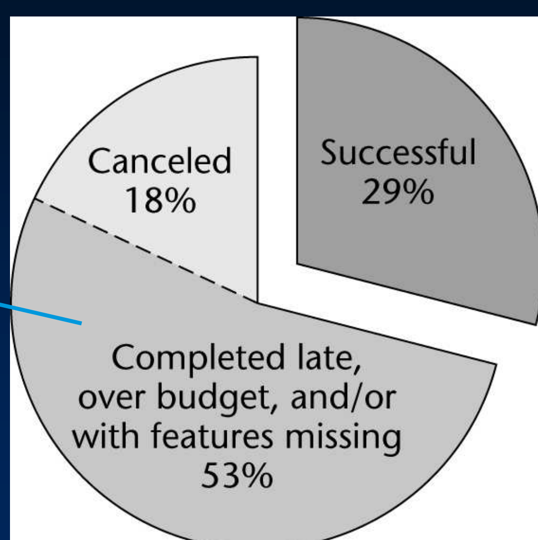


Figure 1.1

What this Means for You

Chance that things end up ok – the project may not be ok along the way

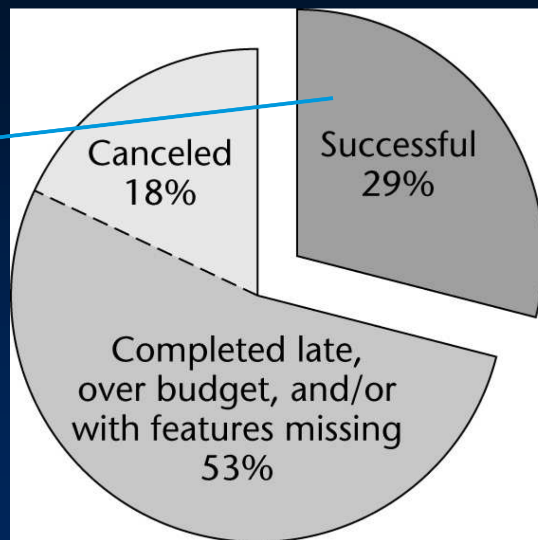


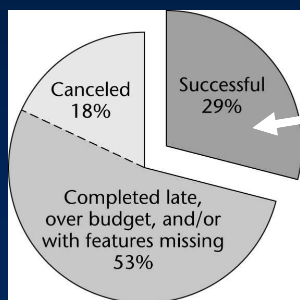
Figure 1.1

We Are Still In Bad Shape

- We know about as much about software quality problems as they knew about the Black Plague in the 1600s. We've seen the victims' agonies and helped burn the corpses. We don't know what causes it; we don't really know if there is only one disease. We just suffer -- and keep pouring our sewage into our water supply. -- Tom Van Vleck
- Considering the current sad state of our computer programs, software development is clearly still a black art, and cannot yet be called an engineering discipline. -- Bill Clinton

Software Mortality Compare to the Plague

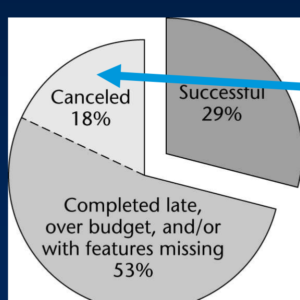
- …. Bubonic plague has a 1-15% mortality rate in treated cases and a 40-60% mortality rate in untreated cases.
  http://www.emedicine.com/emerg/TOPIC428.HTM



If these are the survivors, the software mortality rate is ~71%

How is Software Mortality Compare to the Plague

- …. Bubonic plague has a 1-15% mortality rate in treated cases and a 40-60% mortality rate in untreated cases.
  http://www.emedicine.com/emerg/TOPIC428.HTM



If this is the software mortality rate, it is still worse than treated cases of the plague

Why Do We Study Software Engineering

- The Bubonic plague mortality is 40%-60% when untreated
- A coder who hasn't studied software engineering would fall into the "untreated" category
- By teaching software engineering, we are attempting to lower the software mortality rate
- Software engineering can help prevent the spread of the disease…but probably won't eliminate it
- The sooner a diseased project is treated, the better the chance of recovery (although the recovery may be painful)

History of Fixing Mistakes

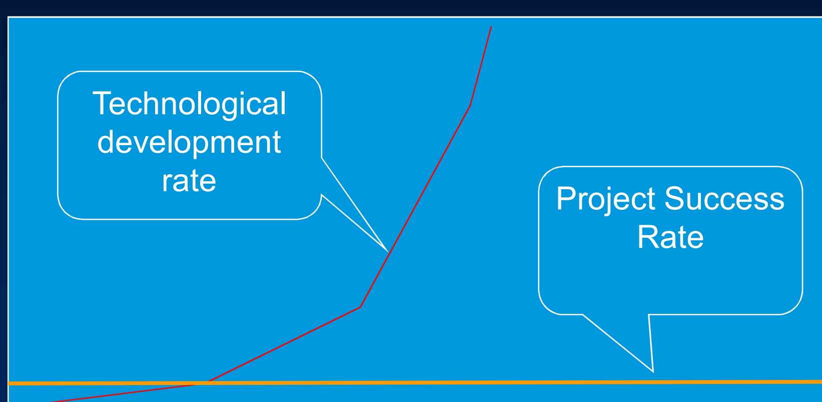
- How have we done historically at realizing what we are doing wrong?
- 18th & 19th Century Medicine Case-study (Venkat Subramaniam)
  - Blood-letting, vomiting, and starvation thought to restore health
  - Barbers and surgeons performed blood-letting
  - Took ~200yrs to figure out that it was a mistake
  - Most patients died of infection after surgery
  - Not until 1800 did Lister invent germ theory

History of Solving Huge Problems

- Flight case-study (Venkat Subramaniam)
- Chinese flew the first kite in 400BC
- Shortly after, people began trying to fly
- 1903 Wright brothers finally fly for 12 seconds, 120ft, 10ft altitude
- 1939 Pan Am makes first commercial trans-atlantic flight
- 2005….Virgin Galactic starts booking suborbital flights
- Once we get over the initial hurdle, we improve fast

SE is More than Technology

- Software isn't just a technical problem – it's also a people problem
- Technology is improving rapidly but SE success isn't
- Why don't the trends match?



Is a Lack of Skilled Labor the Problem?

- Is the problem a shortage of skilled people?
  - McKinsey & Co. predicts that over the next three decades the demand for experienced IT professionals between the ages of 35 and 45 will increase by 25%, while the supply will decrease by 15%.
  - Note: this means you should go into CS
  - Would an unlimited supply of good coders fix the problem?
- Modern systems are far too big to be developed by a single good coder
- Developer skill level is not the problem
  - A good developer solving the wrong problem produces a failure

Team Behavior Isn't Predictable

- All major software development projects require a team of programmers, designers, testers, etc.
- One of the chief problems with software development is that we can't predict how a given team will perform
  - If we could predict team behavior, sports would be boring
- Multiple people create emergent behavior
  - Emergent behavior = non-linear behavior that cannot be predicted by examining the individual parts
  - Emergent property: the testers and the developers hate each other…don't communicate properly

½ Social Problem

- SE is partly a social problem and partly a technical problem
- People don't communicate
- People miscommunicate
- People have egos
- People leave projects
- People get tired
- People make mistakes
- People don't get along

Solving the Social Problem

- The social problem, coordinating people, is harder…just look at governments
- US government case-study
  - State constitutions first
  - Articles of confederation adopted in 1777
    - » 1787 Shay's Rebellion
  - Constitution adopted in 1787
  - Still, we had to work out the details and get things right
    - » 1861 Civil War
    - » ~1955-1968 Civil Rights Movement

One Role of Technology in SE

- Social problems almost always will occur
- Technical things can help to reduce the impact of the social aspects
- We want to use technology to:
  - Identify when social problems are occurring (are tests failing, is the project on schedule)
  - Identify who is causing the problem (what was the last code change that broke the build)
  - Avoid miscommunication (use precise technical specifications)
  - Facilitate communication
  - Etc.

Cutter Consortium Data

- 2002 survey of information technology organizations
  - 78% have been involved in disputes ending in litigation
- For the organizations that entered into litigation:
  - In 67% of the disputes, the functionality of the information system as delivered did not meet up to the claims of the developers
  - In 56% of the disputes, the promised delivery date slipped several times
  - In 45% of the disputes, the defects were so severe that the information system was unusable

Paths to Failure

- Four key types of problems that occur:
  - A system is created that does the wrong thing
    - In 67% of the disputes, the functionality of the information system as delivered did not meet up to the claims of the developers
  - The system takes too long to develop or longer than expected
    - In 56% of the disputes, the promised delivery date slipped several times
  - The system has a poor quality implementation
    - In 45% of the disputes, the defects were so severe that the information system was unusable
  - The system isn't maintainable

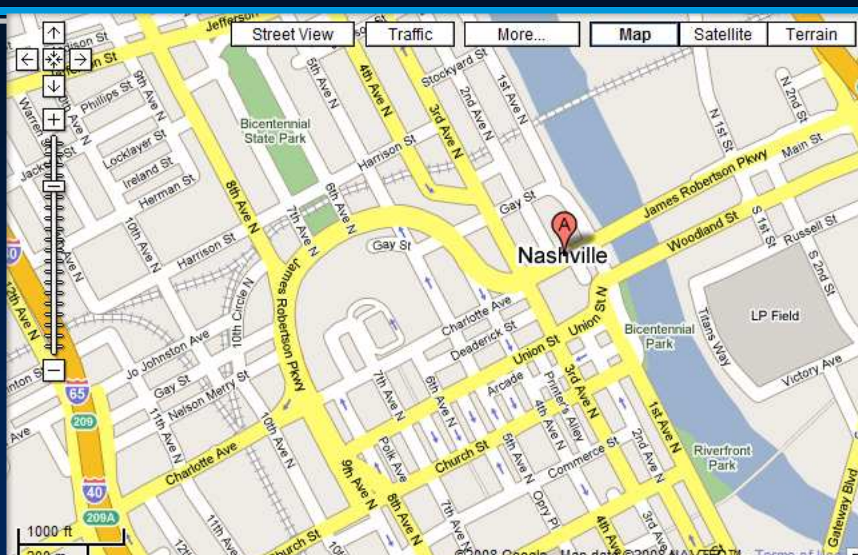
Paths to Failure

- Four key types of problems that occur:
  - A system is created that does the wrong thing
    - Even good developers can build a great system to solve the wrong problem

Requirements Failure Case-Study

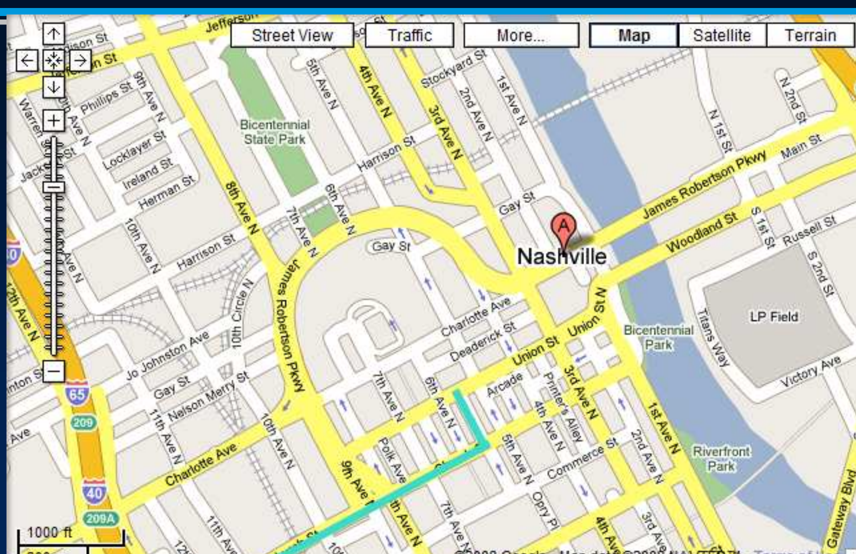
- System for automated scheduling of limousines and drivers to reservations
- Extensive web portal “dashboard” created to manage day to day operations
  - 10+ developers, 5,000,000+$
- Created a complex system to encode rules, such as “this executive leaves the limo messy”
- Complex web-based mapping system to track limos via GPS
  - Track every vehicle at *all times*

Example HCI Expert Requirement



Example HCI Expert Requirement

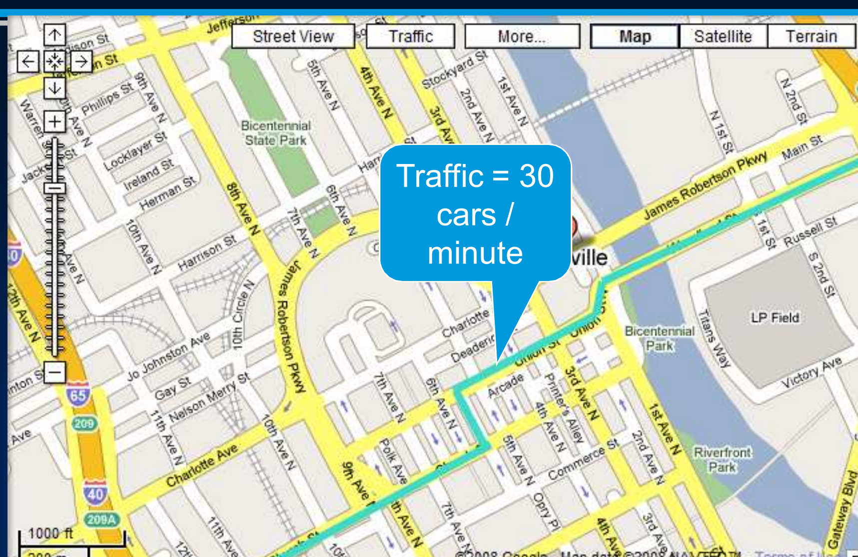
Example HCI Expert Requirement



Copyright © 2008 by The McGraw-Hill Companies, Inc. All rights reserved.

Example HCI Expert Requirement

Example HCI Expert Requirement



Copyright © 2008 by The McGraw-Hill Companies, Inc. All rights reserved.

Case-Study Result

- Extensive web portal "dashboard" replaced by an Excel sheet and macros written by one developer in a week
  - Limo executives like Excel, don't want to use a browser
- Created a complex system to encode rules, such as "this executive leaves the limo messy"
  - Limo executives can't code/understand the implications of rules….had to hire a Ph.D. to manage the rules (they never change)
- Complex web-based mapping system to track limos via GPS
  - Search for closest driver pulls up cars parked in the parking lot – parked cars not removed from map

Paths to Failure

- Four key types of problems that occur:
  - A system is created that does the wrong thing
  - The system takes too long to develop or longer than expected
    - » Someone grossly underestimated how long it would take to develop
      - – Probably got their sales commission anyways
      - – Some salesman are like shady mortgage originators
        - » Yes, we can do it in 2 weeks for a fixed cost of 10,000$ dollars – reality – 1yr, 1,000,000$
    - » What should have been a simple project became a nightmare when the developers couldn't communicate.
    - » There was a huge design flaw
    - » The requirements changed (*requirements creep*)

NPfIT Case-Study

- **The British government budgeted close to $12 billion to transform its health-care system with information technology. The result: possibly the biggest and most complex technology project in the world --** http://www.baselinemag.com/c/a/Projects-Management/UK-Dept-of-Health-Prescription-for-Disaster/
- Originally bid at 12 billion
- Current estimates are that it will cost 28.4 billion
- Another key health-care software maker, iSoft, is some two years behind schedule in delivering a new electronic health-care system called Lorenzo, according to British newspaper The Guardian.
- A 2005 report issued by the NPfIT stated that the migration of data from computers in health-care practices into systems that complied with a new national health-care records system would take far longer than the five years originally projected by the NHS' Connecting for Health (CfH), the unit overseeing the NPfIT project.
- Key software systems have little chance of ever working properly

NPfIT Case-Study

- **What happened?**
  - No one knows for sure
  - There was certainly a lot of emergent behavior that was observed
  - Emergent behavior = we didn't expect that
  - We didn't expect that = our design didn't account for that
  - Our design didn't account for that = we have to change our design
  - We have to change our design …. could mean a cheap change….
    - But, they didn't expect the behavior, so that means they found it after the system was implemented
    - We have to change our design = we have to change the implementation
  - We have to change our implementation = things are going to take a LOT longer than expected (usually)

Paths to Failure

- Four key types of problems that occur:
  - A system is created that does the wrong thing
  - The system takes too long to develop or longer than expected
  - The system has a poor quality implementation
    - » The system is so buggy it isn't worth using
    - » We didn't test it properly
    - » I could name a new OS implementation here
    - » This is probably the problem that we are best at fixing
      - Extensive research and practice on testing techniques

Paths to Failure

- Four key types of problems that occur:
  - A system is created that does the wrong thing
  - The system takes too long to develop or longer than expected
  - The system has a poor quality implementation
  - The system isn't maintainable
    - » More money is spent in maintenance
    - » Technology changes force maintenance
      - New JVM version fixes bug X, we have to migrate
    - » Competitive forces stimulate maintenance
      - Facebook just released feature X, MySpace better duplicate feature X to compete
    - » Etc.

Maintenance is the Future

- *We should all be concerned about the future because we* will have to *spend the rest* of our lives there – Charles Kettering

- We could rephrase this as:
  - *We should all be concerned about the future of this software because we* will have to *spend the rest* of our lives using it

If the MySpace Implementation Wasn't Maintainable...

- MySpace growth

from http://www.mychurch.org/blog/3201/myspace-viral-growth-numbers

Conclusion

- The software crisis has not been solved
- Perhaps it should be called the *software depression*
  - Long duration
  - Poor prognosis

How Do We Fix Things?

- Why don't we just apply the latest greatest technology X to this problem
- Service-Oriented Architectures (SOA) are all the rage
- Everyone wants SOA
- A Google search for "service oriented architectures" pulls up **7,880,000** results

SOA Legends (http://soafacts.com)

- SOA is the only thing Chuck Norris can't kill.
- SOA invented the internet, and the internet was invented for SOA.
- SOA is not complex. You are just dumb.
- In the last year, SOA increased Turkey's GDP by a factor of 10.
- One person successfully described SOA completely, and immediately died.
- Another person successfully described SOA completely, and was immediately outsourced.
- Larry Ellison once died in a terrible accident, but was quickly given SOA. He came back to life, built a multibillion dollar software company, and now flies fighter jets.

Reality: Economic Aspects

- Coding method CM_{new} is 10% faster than currently used method CM_{old}. Should it be used?
- Common sense answer
  - Of course!
- Software Engineering answer
  - Consider the cost of training
  - Consider the impact of introducing a new technology
  - Consider the effect of CM_{new} on maintenance

Other Aspects

- Coding method CM_{new} allows my system to do X. Should it be used?
- Common sense answer
  - Of course!
- Software Engineering answer
  - Can we still guarantee that it will work correctly?
    - » Example: The Department of Defense (DoD) requires all systems to be verified. The DoD doesn't use techniques that it can't 100% guarantee will always work properly. SOA isn't running any missile launchers yet.

SE Shouldn't be Like Dieting

- Coding method CM_{new} allows my system to do X. Should it be used?
- Another way of looking at this: the diet industry
  - Should we immediately trust every dieting fad?
  - Do we really know what diet X is going to do to our bodies in 10yrs?
  - Do I really want to take the latest supplement of ground femur from small animal Y?
- Let's not follow the dieting movement and arbitrarily apply the latest fads
- SE problems should be attacked with sound well-tested principles

Building Software is Like Raising a Child

- To understand a child, you have to know the phases that they go through
- We teach children based on their developmental stage
  - We don’t try to teach kids Calculus before arthimetic
- Understanding the stages of human development are important for maintenance
  - Eating greasy fast food may make me grow faster but what will it do to me 20yrs down the road….how will it make me grow

Maintenance Aspects

- Life-cycle model
  - The steps (*phases*) to follow when building software
  - A theoretical description of what should be done
  - Just like a child, we need to know what to do or teach the child in each stage
- Life cycle
  - The actual steps performed on a specific product

Waterfall Life-Cycle Model

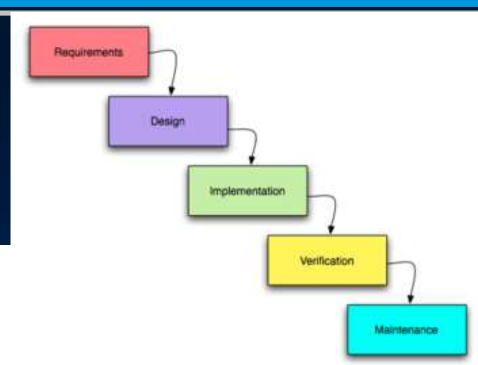
- Classical model (1970)

1. Requirements phase
2. Analysis (specification) phase
3. Design phase
4. Implementation phase
5. Postdelivery maintenance
6. Retirement

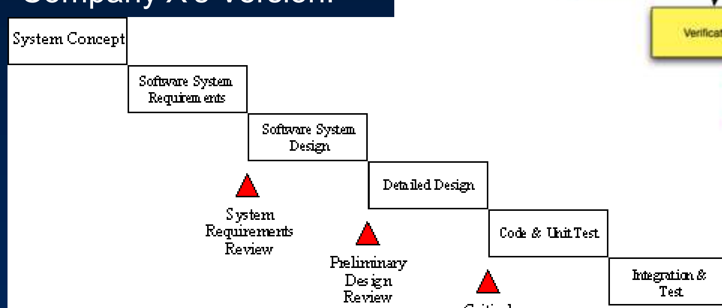
Figure 1.2

Waterfall Life-Cycle Steps Vary

Wikipedia Version:



Company X's Version:



Everything done in stages and Big Design Up Front (BDUF) – Get the design right before you do anything else!!

Typical Classical Phases

- Requirements phase
  - Explore the concept
  - Elicit the client’s requirements
    - » Use-cases
    - » Don’t create the web dashboard
- Analysis (specification) phase
  - Analyze the client’s requirements
  - Draw up the specification document
  - Draw up the software project management plan
  - “What the product is supposed to do”
  - IBM’s mantra is “under promise and over deliver”

Typical Classical Phases (contd)

- Design phase
  - Architectural design, followed by
  - Detailed design
    - » “How the product does it”
- Implementation phase
  - Coding
  - Unit testing
    - » Test the pieces
  - Integration
    - » Test the whole
  - Acceptance testing
    - » Can the customer actually use it

Typical Classical Phases (contd)

- Implementation phase
  - Unit testing
    - » Discover code problems
  - Integration
    - » Discover design flaws
  - Acceptance testing
    - » Discover requirement flaws
- Implementation phase
  - Requirements/Analysis
  - Design
  - Implementation

We identify problems in the opposite order of our development stages – the problems we identify last are the most expensive types to fix

Typical Classical Phases (contd)

- Postdelivery maintenance
  - Corrective maintenance
    - » The banking software is rounding off the cents and putting them into an offshore account, let's fix that
  - Perfective maintenance
    - » Firefox tunes their javascript engine to increase performance
  - Adaptive maintenance
    - » JDK 1.X doesn't support feature Y that we use, let's update the software
- Retirement

Maintenance is Important

- The U.S. Strategic Air Command's 465L Command System, even after being operational for 12 years, still averaged one software failure per day. From G. J. Myers, *Software Reliability: Principles & Practice,* p. 25.

- On June 3, 1980, the North American Aerospace Defense Command (NORAD) reported that the U.S. was under missile attack. The report was traced to a faulty computer circuit that generated incorrect signals. If the developers of the software responsible for processing these signals had taken into account the possibility that the circuit could fail, the false alert might not have occurred. From "The development of software for ballistic-missile defense," by H. Lin, *Scientific American*, vol. 253, no. 6 (Dec. 1985), p. 48.

The Modern View of Maintenance

- Classical maintenance
  - Development-then-maintenance model
- This is a temporal definition
  - Classification as development or maintenance depends on when an activity is performed

Classical Maintenance Defn — Consequence 1

- A fault is detected and corrected one day after the software product was installed
  - Classical maintenance
- The identical fault is detected and corrected one day before installation
  - Classical development

Classical Maintenance Defn — Consequence 2

- A software product has been installed
- The client wants its functionality to be increased
  - Classical (perfective) maintenance
- The client wants the identical change to be made just before installation (“moving target problem”)
  - Classical development

Classical Maintenance Definition

- The reason for these and similar unexpected consequences
  - Classically, maintenance is defined in terms of the time at which the activity is performed
- Another problem:
  - Development (building software from scratch) is rare today
  - Reuse is widespread

Modern Maintenance Definition

- In 1995, the International Standards Organization and International Electrotechnical Commission defined maintenance *operationally*
- Maintenance is nowadays defined as
  - The process that occurs when a software artifact is modified because of a problem or because of a need for improvement or adaptation

Modern Maintenance Definition (contd)

- In terms of the ISO/IEC definition
  - Maintenance occurs whenever software is modified
  - Regardless of whether this takes place before or after installation of the software product
- The ISO/IEC definition has also been adopted by IEEE and EIA

Maintenance Terminology in This Book

- *Postdelivery maintenance*
  - Changes after delivery and installation [IEEE 1990]
- *Modern maintenance* (or just *maintenance*)
  - Corrective, perfective, or adaptive maintenance performed at any time [ISO/IEC 1995, IEEE/EIA 1998]

The Importance of Postdelivery Maintenance

- Bad software is discarded
- Good software is maintained, for 10, 20 years or more
- Software is a model of reality, which is constantly changing

Time (= Cost) of Postdelivery Maintenance

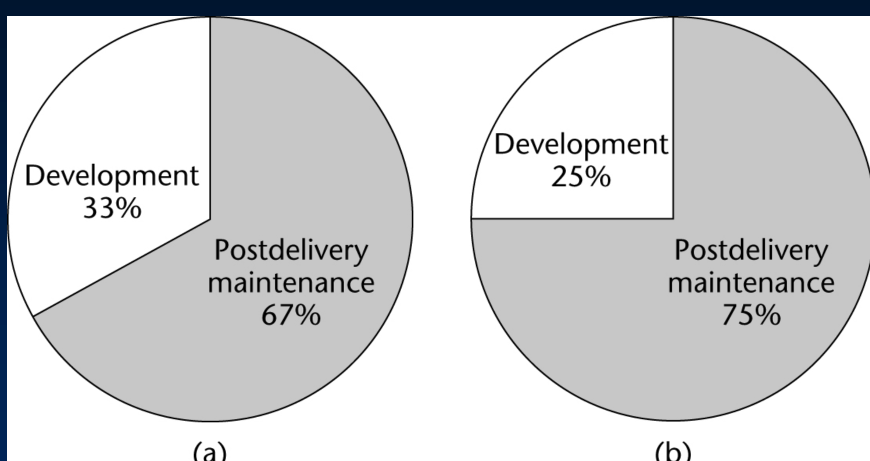


Figure 1.3

(a) Between 1976 and 1981

(b) Between 1992 and 1998

Google Maintenance

- Initially, Google was run out of a dorm room
- Used whatever older servers they could get
- 1998, handled 10,000 searches per day
- 1999, 500,000 searches a day
- 2000, 100,000,000 searches a day
- Clearly, Google had some serious maintenance and probably a few software rewrites along the way

Handling Change in Life

- When we are young, it is easy to change
  - We can pickup a foreign language easily
  - We know less, so we are constantly improving
  - We know less, so we have less biases to get in the way of new ideas
- When we get older, we tend to only learn new things if we have to (in comparison to a child)
  - Adults seek out learning experiences in order to cope with specific life-changing events--e.g., marriage, divorce, a new job, a promotion, being fired, retiring, losing a loved one, moving to a new city. --Ron and Susan Zemke

Requirements, Analysis, and Design Aspects

- Changing software is similar to human learning
- The earlier we detect and correct a fault, the less it costs us
  - Less resistance to the change in early stages
  - Later stages people have become invested in the current structure
  - In later stages, we have to change existing code
  - Harder to predict how the change is going to affect the system when it is bigger

Requirements, Analysis, and Design Aspects (contd)

- The cost of detecting and correcting a fault at each phase

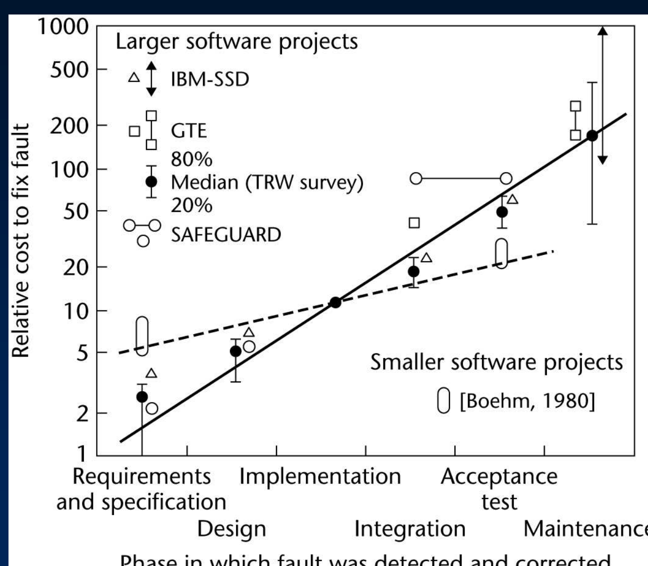


Figure 1.4

Requirements, Analysis, and Design Aspects (contd)

- The previous figure redrawn on a linear scale

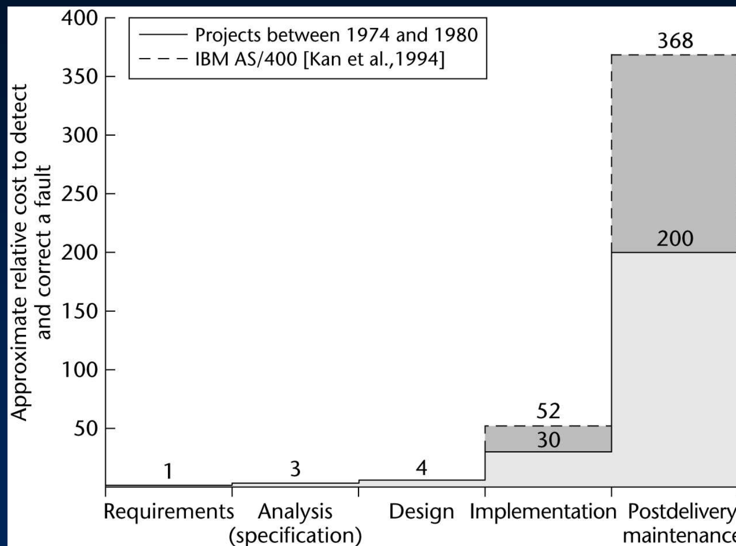


Figure 1.5

Requirements, Analysis, and Design Aspects (contd)

- To correct a fault early in the life cycle
  - Usually just a document needs to be changed
- To correct a fault late in the life cycle
  - Change the code and the documentation
  - Test the change itself
  - Perform regression testing
    - » Regression testing -> did we break something else by making the change?
  - Reinstall the product on the client's computer(s)

Requirements, Analysis, and Design Aspects (contd)

- Between 60 and 70% of all faults in large-scale products are requirements, analysis, and design faults
- Example: Jet Propulsion Laboratory inspections
  - 1.9 faults per page of specifications
  - 0.9 per page of design
  - 0.3 per page of code

Conclusion

- It is vital to improve our requirements, analysis, and design techniques
  - To find faults as early as possible
  - To reduce the overall number of faults (and, hence, the overall cost)

Team Programming Aspects

- Hardware is cheap
  - We can build products that are too large to be written by one person in the available time
- Software is built by teams
  - Interfacing problems between modules
  - Communication problems among team members

Why There Is No Planning Phase

- We cannot plan at the beginning of the project — we do not yet know exactly what is to be built

Planning Activities of the Waterfall Model

- Preliminary planning of the requirements and analysis phases at the start of the project
- The software project management plan is drawn up when the specifications have been signed off by the client
- Management needs to monitor the SPMP throughout the rest of the project

Conclusion

- Planning activities are carried out throughout the life cycle
- There is no separate planning phase

Why There Is No Testing Phase

- It is far too late to test after development and before delivery
- One very popular development approach is “test-driven development”
  - Test cases are written first
  - Software is written to the spec defined by the test cases
- One test is worth a thousand opinions.—Anonymous
- Program testing can be used to show the presence of bugs, but never to show their absence!—Dijkstra

Testing Activities of the Waterfall Model

- Verification
  - Testing at the end of each phase (too late)
- Validation
  - Testing at the end of the project (far too late)
- The later we detect a problem, the more expensive it is to fix
- As soon as we can test something, we should do it

Conclusion

- Continual testing activities must be carried out throughout the life cycle
- This testing is the responsibility of
  - Every software professional, and
  - The software quality assurance group
    - The 3 guys dedicated to finding every mistake you made and documenting it
- There is no separate testing phase

Why There Is No Documentation Phase

- It is far too late to document after development and before delivery

Documentation Must Always be Current

- Key individuals may leave before the documentation is complete
- We cannot perform a phase without having the documentation of the previous phase
- We cannot test without documentation
- We cannot maintain without documentation

Conclusion

- Documentation activities must be performed in parallel with all other development and maintenance activities
- There is no separate documentation phase

1.9 The Object-Oriented Paradigm

- The structured paradigm was successful initially
  - It started to fail with larger products (> 50,000 LOC)
- Postdelivery maintenance problems (today, 70 to 80% of total effort)
- Reason: Classical methods are
  - Action oriented; or
  - Data oriented;
  - But not both

The Object-Oriented Paradigm (contd)

- Both data and actions are of equal importance
- Object:
  - A software component that incorporates both data and the actions that are performed on that data
- Example:
  - Bank account
    - » Data: account balance
    - » Actions: deposit, withdraw, determine balance

Strengths of the Object-Oriented Paradigm

- 1. With information hiding, postdelivery maintenance is safer
  - The chances of a regression fault are reduced
- 2. Development is easier
  - Objects generally have physical counterparts
  - This simplifies modeling (a key aspect of the object-oriented paradigm)

Strengths of the Object-Oriented Paradigm (contd)

- 3. Well-designed objects are independent units
  - Everything that relates to the real-world item being modeled is in the corresponding object — *encapsulation*
  - Communication is by sending *messages*
  - This independence is enhanced by *responsibility-driven design*

Weaknesses of the Object-Oriented Paradigm

- 1. The object-oriented paradigm has to be used correctly
  - All paradigms are easy to misuse
- 2. When used correctly, the object-oriented paradigm can solve some (but not all) of the problems of the classical paradigm

Weaknesses of the Object-Oriented Paradigm (contd)

- 3. The object-oriented paradigm has problems of its own
- 4. The object-oriented paradigm is the current standard
- 5. Other important/current paradigms:
  - Component-based
  - Service-based

1.10 Terminology

- Client, developer, user
- Internal software
- Contract software
- Commercial off-the-shelf (COTS) software
- Open-source software
  – Linus's Law

Terminology (contd)

- Software
- Program, system, product
- Methodology, paradigm
  – Object-oriented paradigm
  – Classical (traditional) paradigm
- Technique

Terminology (contd)

- Mistake, fault, failure, error
- Defect
- Bug
  - “A bug crept into the code”
    instead of
  - “I made a mistake”

Object-Oriented Terminology

- Data component of an object
  - State variable
  - Instance variable (Java)
  - Field (C++)
  - Attribute (generic)
- Action component of an object
  - Member function (C++)
  - Method (generic)

Object-Oriented Terminology (contd)

- C++: A member is either an
  - Attribute (“field”), or a
  - Method (“member function”)
- Java: A field is either an
  - Attribute (“instance variable”), or a
  - Method

Definition of Object-Oriented Software Engineering

- Software engineering
  - A discipline whose aims are
    - » The production of fault-free software,
    - » Delivered on time and within budget,
    - » That satisfies the client’s needs
    - » Furthermore, the software must be easy to modify when the client’s needs change
- Object-oriented software engineering
  - A discipline that utilizes the object-oriented paradigm to achieve the aims of software engineering

1.11 Ethical Issues

- Developers and maintainers need to be
  - Hard working
  - Intelligent
  - Sensible
  - Up to date and, above all,
  - Ethical
- IEEE-CS ACM Software Engineering Code of Ethics and Professional Practice
  www.acm.org/serving/se/code.htm