SOFTWARE MAINTENANCE & EVOLUTION

STUDENT HANDOUT

Jeff Offutt

TABLE OF CONTENTS

A HISTORICAL INTRODUCTION TO SOFTWARE MAINTENANCE & EVOLUTION [1]

This essay prints a short historical overview of how software maintenance has changed over the years and how the term “evolution” is not more appropriate than “maintenance.”

TRADITIONAL QUALITY ATTRIBUTES (1980S)

When I was in school in the 1980s, most software projects only cared about two quality attributes:

1. Efficiency of process (time-to-market)
2. Efficiency of execution (performance)

These are still taught as the most important attributes to computer science students. It is usually taught implicitly, as that is how assignments are graded. It might have been true in 1980, but is very seldom true in the 21st century.

A few years ago, I surveyed about a dozen project managers and asked them what the most important quality attributes ("ilities") were to their projects. Every single manager listed the first three in the same order:

1. Reliability
2. Usability
3. Security

The next four varied in order, but were reasonably consistent:

4. Availability
5. Scalability
6. Maintainability
7. Performance & Time-to-market

Note that performance and time-to-market were still in the list, but pretty far down. Why do we have such a mis-match in what we teach and what industry needs? I think a historical view of how software projects have grown is instructive.

SOFTWARE PROJECTS ACROSS THE DECADES

In the 1960s, programmers built "tiny log cabins." Most projects were built by a single-programmer and were not very complex. The engineering process used was not very important, largely because the design could be kept in one programmer's short-term memory.

In the 1970s, programmers started to build houses. Most teams were small and there was a strong focus on algorithms and programming. The software was larger and more complex, and programmers had to start thinking harder. The lack of emphasis on engineering process led to disasters where projects were not completed, completed late, or failed spectacularly after they were completed. A very important point is that, for most of the industry, the quality of the software did not affect profits. There was very little competition, almost nobody built high-quality software, and expectations were extremely low. But the costs were starting to increase.

Parts of the industry saw these changes sooner than most of the industry. Space systems, aircraft systems, weapon systems, and some other safety-critical applications were building software that was larger, more complex, and had higher reliability needs. In fact, the NATO Science Committee sponsored conferences in 1968 and 1969 that introduced the terms “software crisis” and “software engineering” [2].

In the 1980s, teams of programmers were building office buildings. We needed significant teamwork and communication, including clear requirements and design that could be shared and archived. The software was much more complex and relied heavily on data abstraction. Unfortunately, the use of poor processes and ignorance of the need for process created many spectacular software failures. It was very clear that we no longer had the skills and knowledge for successful software engineering.

This process accelerated in the 1990s, when large teams were creating skyscrapers. Industry needed more than teamwork, communication, and cooperation; we needed totally new technologies, including languages, modeling techniques, and engineering processes. Software development changed completely in the 1990s. New languages (Java, UML, etc) led to revolutionary procedures, and new tools such as IDEs and deployment engines dramatically increased productivity. Sadly, more productivity often meant programmers could simply make more mistakes, faster. Our education fell further and further behind as universities continued to focus on teaching algorithms and theory, using outdated languages, and emphasize speed to the exclusion of all else—performance and time-to-completion. Computer science education became more and more competitive, and students who valued cooperation, communication, and quality moved out of the field.

Two huge waves started in the 1990s that dramatically changed the nature of software engineering. The first was subtle and fundamental—software became competitive. More users and more uses for software led to more software companies, which in turn gave software purchasers choice. The obvious, and probably most significant innovation in the 1990s was, of course, the web. As they say, “the web changes everything,” especially software engineering.

By the 2000s, software companies were using large teams to build integrated collections of continuously evolving cities. Algorithm design and programming was no longer the primary focus of software development, modifying and expanding existing software was much more important. Component-based software, “glue software,” and continuous evolution became essential. New applications, led by web software, made quality attributes such as reliability, usability, and security critical to the success of software. CS education fell so far behind it started to look almost obsolete. Professional engineers often learn more from training courses after college than they did in college. By the 2000s, we had relatively little new software development. Most professional engineers join projects after they start and spend most of their career modifying existing code bases.

The changes have, if anything, accelerated in the 2010s. In addition to web applications moving from novel to standard, software has transformed our mobile telephones into incredibly powerful pocket computers, giving us the ability to shop, register for classes, pay taxes, monitor current events, be entertained, and not get lost. And as a bonus, we can use these computers to communicate. Software is also invading our homes in many ways (IoT), including our thermostats, garage doors, ovens, washing machines, and everything else that uses electricity. In the 2010s, software is ubiquitous and diverse. Software looks like metropolises and small gadgets. Perhaps most importantly, users expect the software to be designed and work better than ever before, and to continuously evolve.

As someone who has seen five decades of software engineering, the pace of change has been exhilarating. We have gone from building log cabins to houses to office buildings to skyscrapers to building the most complicated engineering systems in human history in just half a life-time. Civil engineers took thousands of years to make this much progress, and the most complicated civil engineering products pale in comparison to the complexity of modern software systems. Electrical engineers took a couple of centuries to reach this much complexity. There is simply no way that humans could have kept up. Among other things, that means that all software engineers must be continuous lifelong learners.

THEORY, EDUCATION, AND PRACTICE

What do students learn in college? Primarily how to build houses. Most universities have one general software engineering course that introduces a few concepts about buildings. But the way we build software has changed dramatically since the CS curriculum stabilized in the 1980s.

What can you do? As a professional software engineer, some things turn out to be simple. Program very neatly so others can change your software later. Design to make future changes easy. Follow processes that make changes easy. At a higher professional level, learn from your colleagues, teach your colleagues, and never stop learning. Take as many training courses as you can, and if you don’t have one already, go back to school and start that master’s degree.

OVERVIEW OF SOFTWARE MAINTENANCE AND EVOLUTION [3]

This overview of software maintenance is drawn from multiple sources. We are at a relatively strange period in software engineering where maintenance and evolution activities account for much, if not most, of software costs, yet most of our understanding is based on studies that are decades out of date.

Sommerville defines software maintenance as: “When the transition from development to evolution is not seamless, the process of changing the software after delivery is often called software maintenance” [4].

More generally, maintenance involves modifying a program after it has been put into use. We usually do not expect maintenance to involve major changes to the system's architecture. Rather, changes are made by modifying existing components and adding new components to the system. From a programmer's perspective, the key issue is that the programmer must understand the program and its structure.

Maintenance is important because software is crucial to company's success and because software is very complicated. In today's world, most of the software budget is devoted to modifying existing software rather than developing new software

Any successful software product is in use much longer than in development, making maintenance even more crucial. If software does not continue to adapt to changes in needs and environment, it becomes progressively less useful. Changes are inevitable, and occur for several reasons:

- New requirements emerge when the software is used
- The business environment changes
- Faults must be repaired
- New computers or equipment are added to the system
- Growing user base makes the performance or reliability insufficient.

Some changes are due to the tight coupling with the environment. When software is installed, it changes that environment, in effect changing the software requirements

MYTHS RELATED TO SOFTWARE MAINTENANCE AND EVOLUTION

In practice, many software engineers and managers have falsely been convinced of several things that are simply not true. Some of these myths are explained and contradicted below.

Myth: "We already have a book that's full of standards and procedures for building software, won't that provide my people with everything they need to know?"
Reality: The book of standards may very well exist, but is it used? In many cases, the answers to the following questions are "no."

1. Are software practitioners aware of its existence?
2. Does it reflect modern software engineering practice?
3. Is it complete?
4. Is it streamlined to improve time to delivery while still maintaining a focus on quality?

Myth: If we get behind schedule, we can add more programmers and catch up.
Reality: Software development is not a mechanistic process like manufacturing. As Brooks said: "adding people to a late software project makes it later."

Myth: If I outsource the software project to a third party, I can just relax and let that firm build it.
Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

Myth: General objectives are enough to start programming--we can fill in the details later.
Reality: A poor up-front definition is a major cause of failed software efforts. If you don't know what you want at the beginning, you won't get what you want.

Myth: Project requirements continually change, but change can be easily accommodated because software is flexible.
Reality: It is true that software requirements change, but the impact of a change depends on when it is introduced.

Myth: Once we write the program and get it to work, our job is done
Reality: Someone once said that “the sooner you begin ‘writing code,’ the longer it’ll take you to get done.” Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer.

Myth: Until I get the program “running” I have no way to assess its quality
Reality: One of the most effective software quality assurance mechanisms can be applied from the beginning of a project—the formal technical review. Software reviews are more effective than testing for finding certain classes of software defects.

Myth: The only deliverable work product for a successful project is the working program
Reality: A working program is only one part of a software configuration that includes many elements. Documentation provides a foundation for successful engineering and, more important, guidance for software support.

Myth: Software engineering will make us create voluminous and unnecessary documentation and will always slow us down
Reality: Software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

TRADITIONAL TYPES OF MAINTENANCE

Maintenance has traditionally been divided into four types. The percentages given in textbooks are quite similar but are based on studies from the early 1980s. It seems unlikely the numbers still hold, so take these with a grain of salt. They are all we have.

1. *Perfective maintenance* (50%): Enhancements where new operations and refinements are added to existing functions.
2. *Adaptive maintenance* (25%): Modifying the application to meet new operational circumstances.
3. *Corrective maintenance* (21%): Eliminating errors in the program’s functionality.
4. *Preventive maintenance* (4%): Modifying a program to improve its future maintainability.

Some authors will consider *emergency maintenance* as being a type of corrective maintenance that is not scheduled.

A major change in the last 20 years in the way software is maintained is that companies often release an intentionally partial version, then add to it over time. It is not clear to me whether this is properly considered to be perfective, adaptive, or something else. But it is clear that the term *evolution* is more appropriate than maintenance. This is possible when software is easy to update. For example, software in our mobile devices can be updated every day, as opposed to software in submarines, which can normally only be updated when the submarine visits port. Indeed, this is a major driver behind the Internet of Things movement. Fixing a software problem in my washing machine requires an expensive visit by a technician to replace a circuit board, whereas if my washing machine is on the internet, the company can download new software for almost no cost.

COSTS OF SOFTWARE MAINTENANCE AND EVOLUTION

Modifying software, no matter how or when, is difficult and costly. When software and hardware is tightly integrated, software is also often looked at as the easiest part to change. But just because the manufacturing cost is so slow does not mean the design and implementation cost of making changes is free.

Large software programs are extremely complicated, and very difficult to understand. Yet they must be understood before they can be changed. Many changes are rushed, so the modifications are often poorly designed and implemented to make things worse, changing the software almost invariably injects new faults that must be repaired later.

Sommerville [4] claims that 90% of all software costs are related to software evolution. Even if that's overstated, few would doubt that more than half of software costs are accrued [5] after first deployment. The costs are due to both technical and non-technical factors. Most engineers agree that the cost of making the same type of change goes up as the software ages. This is due to many factors, including loss of memory about the initial design and construction, changes in technologies and languages, and the difficulty of understanding previous (often sloppy) changes.

Some cost factors typically listed are:

1. **Team stability**: Maintenance costs are greater if the original developers are not available.
2. **Contractual responsibility**: If the original developers were not expected to be responsible for future changes, they had very little incentive to design for change.
3. **Staff skills**: Maintenance tasks are sometimes given to entry-level engineers, summer interns, or poorly educated low-skilled programmer. This means they may take longer and their changes may be less reliable.
4. **Program age and structure**: As programs age, changes start to degrade their structure, making them harder to understand and change.

SOFTWARE MAINTENANCE TERMS

Most of these definitions are taken from IEEE standards [5]or textbooks (which probably derived them from the IEEE).

- *Maintainability*: The ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment. Maintainability cannot be measured directly, but is affected by factors such as age, size, programming language, architecture and design, and documentation and formatting. Maintainability is also quite subjective, since the engineer's ability to understand the software is heavily influenced by prior knowledge and experience with other programs.

- *Ripple Effect*: Changes in one software location can impact other components. Ripple effects cannot be fully understood with static analysis of the source, so dynamic analysis must be used. That is, we have to run the program.

- *Impact Analysis*: The process of identifying potential consequences of a change in terms of how the change will affect the rest of the system [6]. This is a difficult analysis that is done to estimate the cost of a change, to choose which fault repair is most cost-effective, to plan for regression testing, or to understand what resources will be needed to make the change.

- *Traceability*: The degree to which a relationship can be established between two or more products of the development process, such as the requirements and code, or design documentation and tests [6].

- *Legacy systems*: A software system that is still in use but the development team is no longer available. Legacy systems have often been inherited, is still valuable, and is very hard to change.

REALITY CHECK

As said above, most of the available literature on maintenance and regression is 30 years old! The IEEE standards are from 1990, and only changed modestly from their 1983 versions. The major textbooks have been updated regularly since the 1980s, the information on maintenance has not changed much.

The literature is also conflicted over the use of maintenance and evolution. A possible distinction is:

- *Software Maintenance*: The activities required to keep a software system operational and responsive after it is deployed.
- *Software Evolution*: A continuous change from a lesser, simpler, or worse state to a higher or better state.

I have also heard them distinguished as software maintenance being about fixing the software so that it is close to working as intended (adaptive, corrective, and preventive), whereas evolution is about changing its intended behavior (perfective).

LEHMAN’S LAWS OF SOFTWARE EVOLUTION

Before concluding, any overview of maintenance or evolution must have a nod to Lehman’s laws [7]. They were first proposed around 1980, and still seem to be relevant today.

1. **Law of Continuing Change**: Software that is used in a real-world environment must change or become less and less useful in that environment.
2. **Law of Increasing Complexity**: As an evolving program changes, its structure becomes more complex, unless active efforts are made to avoid this phenomenon (hence refactoring).
3. **Law of Self-Regulation**: Program evolution is a self-regulating process. System attributes such as size, time between releases, and the number of reported errors are approximately invariant for each system release.
4. **Law of Conservation of Organizational Stability**: Over a program’s lifetime, its rate of development is approximately

constant and independent of the resources devoted to system development.

5. **Law of Conservation of Familiarity**: Over the lifetime of a system, the incremental system change in each release is approximately constant.

6. **The Law of Continuing Growth**: The functionality offered by systems has to continually increase to maintain user satisfaction.

7. **The Law of Declining Quality**: The quality of a system will appear to be declining unless it is adapted to changes in its operational environment.

8. **The Feedback System Law**: Evolution processes incorporate multi-agent, multi-loop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

HOW THE WEB RESUSCITATED EVOLUTIONARY DESIGN [8]

One of my favorite oldies, *The Design of Everyday Things* [9], discusses evolutionary design. It caused me to consider what this concept means to software design, development, and testing. I want to start with cost. All technological artifacts, hardware and software, come with costs. Not being an economist or systems engineer, I may leave some out, but at least four types of costs help us understand a major trend in software:

1. **Design**
2. **Production**
3. **Distribution**
4. **Support**

The relative amounts of these four costs are continuously changing. Back “in the day,” we evolved our designs very slowly. For centuries before the industrial evolution, most things were crafted by hand. A carpenter took days or weeks to build a rocking chair with hand tools. After each chair was built, the carpenter considered ways to make the next one better. The design evolved over time, but each new chair was a little bit better than the last.

Hand-crafting incurred very little design costs, but had very high production costs. Craftsmen worked for days, weeks, or months on each product. The distribution cost was fairly low—most craftsmen before about 1850 sold directly to customers, usually in the same village or town. Support cost was also low. Buyers used the product until it wore out; usually years if not decades.

Then manufacturing and the industrial revolution changed the equation. Assembly lines allow many people to work on one product, letting the same team produce many products quickly. The same design was put into hundreds or thousands of products, bringing production costs down dramatically. But to ensure quality, designs had to be created more carefully, and encoded into a factory to allow the assembly line to operate. Thus, design costs went up. Because more products could be created in the same factory, products were distributed more widely, increasing distribution costs. More complicated technological objects, in turn, meant that support costs started increasing. They were often outsourced to people like car mechanics or plumbers, so that customers bore the support costs.

We developed automated manufacturing in the 1900s. Automation increased the speed and efficiency of production, so production cost continued to decline. Design costs now included creating expensive automation hardware, including robots. At the same time, distribution continued to expand, increasing costs.

After World War II, we globalized our manufacturing. In particular, cheap energy, large ships, and global air freight dramatically increased our ability to distribute. As automation increased with electronics and software, production costs continued to decline, and in turn, design costs continued to increase. As costs decreased, support also decreased because people replaced instead of maintained.

By the end of the 20th century, free trade accelerated globalization. Oil became cheaper and shipping costs declined. Ultimately, design became the dominant cost of many engineering products. In other words:

MANUFACTURING DEFEATED EVOLUTIONARY DESIGN!

Automation and globalization allow people to buy thousands of incredibly cheap products, but many are low quality and built to last a few months instead of decades. Instead of evolution, we have replacement. We have lost something precious and wonderful: **craftsmanship**.

What does this have to do with software?

Traditional software development such as the waterfall process has very low production cost, and substantial distribution cost that includes marketing, sales, and shipping. Support costs escalated as software became larger and more complicated. Design activities split into software design and software implementation. That is, what software engineers call implementation is more akin to design in hardware, and software production is simply compiling and copying files. In software engineering, both design and implementation is very expensive.

In the 1990s, having millions of customers skewed costs to the back end, where support costs and distribution costs steadily increased. New versions of major software products shipped every five to ten years, which meant that software problems affected users for years. This forced a mentality that software had to be "*perfect out of the box*." Both design and implementation became very expensive—including testing, which was 50% or more of the total cost. As a result, software evolved very slowly.

The need to be “perfect out of the box” has heavily influenced decades of software engineering research, creating such goals as describing software formally, modeling large systems, creating processes that led to perfection, testing products after completion, and looking at maintenance in terms of years. Until recently, most of our research focus and results have assumed that design costs, implementation costs, support costs, and distribution costs are high.

But a countercurrent had been slowly but steadily forming as storage technology drove software distribution costs down. We moved from shipping large disk packs to floppy disks, to diskettes, to CDs, to thumb drives. At the same time, engineers responded to high support costs by increasing usability, which decreases the need for customer support.

Then the web changed everything, creating a tipping point: First, the web created a new way to deploy and distribute software. Second, the web rearranged the importance of quality criteria, making usability and reliability crucial as competitive advantages.

The web is used to distribute desktop software nearly instantaneously and with near zero-cost, allowing more frequent updates. Even more impressive, software that runs on the web (web applications) are not distributed in any meaningful sense. Web applications reside on servers where updates can be made weekly, daily, hourly, or even continuously! Even more extreme, applications on mobile devices allow the craftsman to come into your “home” to improve that rocking chair at any time.

Almost magically, near-zero production costs ... immediate distribution ... and near-zero support costs ... resuscitated evolutionary design!

In summary, before the web, design was expensive, development was expensive, new versions were deployed infrequently, and evolution was very slow. Post-web software engineers design and develop “pretty good” initial versions, then gradually make it bigger and better. Evolution is faster as we make immediate changes to web applications, automatic updates to desktop applications, push out software upgrades to mobile devices, and replace chips in cars during oil changes. This changes all of software engineering!

As researchers, we have the opportunity to look at many novel and interesting problems. Software is not so much designed and built; it *grows*. Testing must focus on evolution, not new software. The waterfall process is now, finally, thankfully, completely dead. Yes, the web really does change EVERYTHING.

These changes affect software testing in deep and fundamental ways. Test-driven design uses tests to drive requirements—every step is evolutionary. We must stop thinking of regression testing as something special done “late in the process,” and think of virtually all testing as regression testing. Model-based testing is so widely studied and used because it allows test design to quickly and easily adapt to changes. Test automation is essential to running tests as often as software changes occur.

This new world brings many new questions. How do we translate from test models to automated tests? What is our best strategy for creating test oracles? How do we test continuously, when requirements evolve too often to track? The meta-question that all software testing researchers should ask is: Does our research support evolutionary design?

CODING FOR CHANGE [10]

Programmers can do lots of small things to improve the maintainability of their software. If you don't do these things already, there will be a modest learning cost. But if you develop these to habits, there will be a significant payoff, not just in maintainability, but in debugging and reliability.

CHANGE OVERVIEW

A fourth-year student in a computing major has probably figured out some of these points. Others will be familiar, but not yet clear. And still others won’t be apparent until you get more experience. And when software engineers say “experience,” we usually mean something bad happened.

Changing software, just like changing your house, is quite different from the initial development. Adding a new room costs more than it would have cost to build that room initially. Changes are constrained by the goals, style, and implementation of the existing system and require changes to existing parts of the system.

Another challenge is that we have to understand an existing system to change it. How can the existing code base accommodate the change? What are the potential ripple effects? What skills and knowledge are required to make these changes?

Whether you do it consciously or unconsciously, you will go through several activities every time you change existing software.

1. **Identify** what to change and why the change is needed.
2. **Manage** the process in terms of what resources are needed (including people, materials, money, and time).
3. **Understand** the program, learn how to change the program, and estimate ripple effects of the change.
4. **Make** the change.
5. **Test** the change.
6. **Document** and record the change through configuration management.

Note that most well organized software companies will have processes and procedures that must be followed. If not, that could be a sign that projects are not well organized and quality is not valued.

UNDERSTANDING THE PROGRAM

Students usually make simple changes to small programs, often to programs, they wrote themselves. This makes understanding the code very simple. But the elements of understanding are still the same, whether the program is large or small. First, the programmer must develop domain knowledge about software’s behavior. Domain knowledge can be gleaned from documentation, end-users, or as a last resort, from reading the program source. Next, the programmer must understand the execution behavior, including the external behavior and how the internal algorithms work. After that, it is essential to learn how different parts of the software affect and depend on each other. Finally, maintenance programmers must learn how the product interacts with the environment.

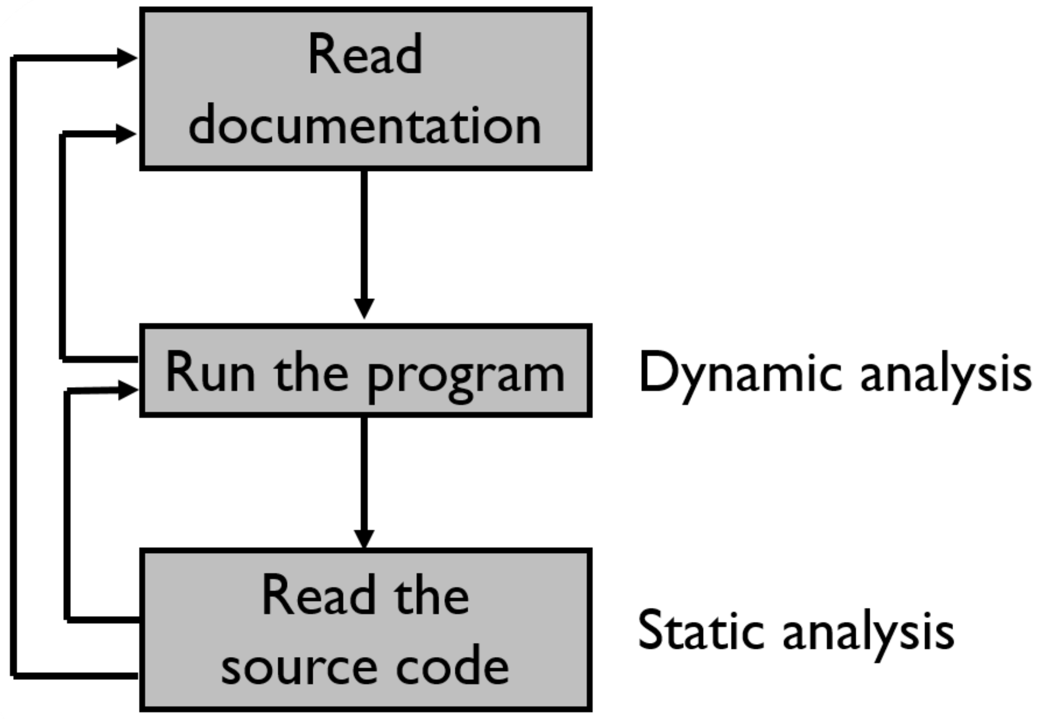

Figure 1: Comprehension Process

Figure 1 illustrates a typical process used to understand software. Reading the documentation can save enormous time over reading the source, but it’s important to recognize that the documentation could be out of date. If the software was built with an agile process, automated tests might be available that document the behavior. They can be very helpful—in fact, the primary goal of most agile processes is to make change easier.

The challenge of understanding an existing program is influenced by many things, including:

- **Expertise:** Domain knowledge and programming skills
- **Program structure:** Modularity, level of nesting, shared data stores, and communication paths
- **Documentation:** Is it readable, accurate, and up-to-date?
- **Coding conventions:** Naming style and small design patterns
- **Comments:** Accuracy, clarity, and usefulness
- **Program presentation:** Indentation and spacing

PROGRAMMING FOR CHANGE

In 1980 the most important programming concerns were about speed and size. Computer hardware was slow, memory was expensive, and screens were small. Programming tools were primitive and often expensive.

Today, however, computers are faster, memory is cheap and plentiful, screens are huge, and programming tools get better and cheaper every day. The overriding concern is to make it easier to change the program later.

This means **readability** is very important. Even debugging your own code can be painful if you were not tidy to begin with. (By the way, the best programmers usually try to be tidy in real life too.) Readable code is also more secure and reliable because problems are easier to spot. This is not to say that speed and size is never important, it's just less important than in the past. Real-time and embedded software must run fast. On the other hand, a good optimizing compiler can do far more than a human can and can do more with well-structured software.

Another essential habit to support maintenance is to avoid unnecessary fancy tricks. It's fun to learn nifty tricks with pointer arithmetic, but those tricks have very little benefit, but create a huge amount of maintenance debt. Write for humans, not for compilers:

1. *Fully parenthesize all expressions*
2. *Use optimizations only if you are sure they will help*
3. *Use variable names that others will understand*
4. *If your algorithm gets complicated, explain what's going on in comments*
5. *Don't be afraid to break up complicated expressions by creating a few temporary variables—they are cheap and an optimizing compiler will remove them anyway*

In 1980, the control flow of individual functions dominated the running time of a program, which is exactly why undergraduate CS programs emphasize analysis of algorithms so much. Today however, the architecture of the system usually washes out any effect of optimizing individual methods. I once worked on a project where a colleague spent six weeks to optimize methods that accessed data from files—saving almost 4% of execution time. I then looked at his overall design and found that he was reading the entire file into memory each time through an outer loop! Four hours to make one architectural change saved over 40% of execution time. Moreover, nobody could ever understand his micro-optimizations ... even him.

Back "in the day," student programs were graded based on comments and formatting. They developed skills that became good habits. We grade automatically these days and often don't even look at students' code. That debt is paid eventually, usually by future colleagues. This begs the question: when should you write comments?

1. *Always include header blocks for each method*
2. *Document all assumptions*
3. *Add a comment every time you have to stop and think*
4. *Document all variables that can be overridden by child methods*
5. *Document reliance on default and superclass constructors*
6. *Write pseudocode as comments, then write the code and keep the comments*

The last habit will also help you finish faster, result in more reliable software, and leave you with ready-made, free, documentation.

“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.”
— C.A.R. Hoare

TIPS FOR WRITING MAINTAINABLE JAVA

Developing the following habits will make you more popular among your teammates, more respected by your team leads, and more valuable to your company. Developing these habits in college will make you more efficient, faster, and help you get better grades.

1. *Be precise— Sloppy style looks like sloppy thinking*
2. *Test software pieces continually—the “build and test” cycle is essential to well-engineered software*
3. *KISS: Keep It Simple and Stupid*
4. *Do not optimize until you know what needs to be optimized, and keep the un-optimized version in comments for documentation*
5. *Use design patterns when they fit, but don’t bang on the jigsaw puzzle pieces to make them fit*
6. *Don’t test for error conditions you don’t know how to handle; let them propagate to someone who does*

Out-dated documentation can be very dangerous because it can mislead future maintainers. One school of thought says that it’s better to have no information than incorrect information. One solution is to avoid documentation. Another is to be disciplined about updating documentation. Which is better depends on the situation and the people involved. But leaving incorrect document is unprofessional—selfish, lazy, and short-sighted.

This also means we should not over-document. There is no point in documenting the obvious:

```
setList (List list); // This method sets a list
```

This comment makes the code worse, not better. But it might be better to describe the list and how it's being set. This could be done in comments ... or even better, with self-documenting names:

```
readNamesFromFile (List nameList);
```

Using your language well can also improve the readability. Here are a few Java-specific tips.

Always implement both or neither equals() or hashCode()

Always override toString() to produce a human-readable description of the object

If o1.equals (o2) is true, o1.toString() should equal o2.toString()

If equals() is called on the wrong type, return false, not an exception

If your class is cloneable, use super.clone(), not new() (new() will break if another programmer inherits from your class)

Don't keep two copies of the same data

Threads are hard to get right and harder to modify; use only if necessary, and if you do, check them by hand and with careful testing

Don't add error checking that the VM already does

Also remember that immutable objects are your friends in Java. They are simpler, safer, and more reliable. They sacrifice some speed, but not much. Basic types such as keys should always be immutable. After all, their values cannot change. Immutable objects should be declared final so that they can be used just as elementary types. And last but not least, immutable objects are especially useful in concurrent software because they cannot be corrupted by thread interference

Another really good habit is to increase modularity and reduce coupling as much as possible. The goal of reducing coupling led to most major programming advances in the last 40 years, including macro assemblers, high level languages, structured programming, ADTs, data hiding, inheritance, polymorphism, CASE tools, UML, JavaBeans, XML, the web, J2EE, etc. The most common theme is to increase modular components such as methods, classes, files, and packages. It must be possible to describe each component in a very concise way, for example, "this is a set of allowable prices." If not, nobody will ever be able to maintain or reuse it.

Think hard about which modules refer to which other modules. It should always be possible to change the implementation without having to change other modules. That is, assume the implementation changes regularly, but the API rarely changes. A very effective way to test the design is to make a few changes early, before the development team breaks up or moves on.

Identity also matters. Think about what an object is, not what the class does. "This is a library book" is clearer and simpler than "this class stores names, prices, and a count of books, and provides access to the information about the books." A quick rule of thumb is that most class names should be vowels, not verbs. Remember that an object is defined by its state, and the class defines its behavior.

Look for classes that have lots of switch statements; that sometimes means they are trying to do too many things. Use inheritance, a base class and children classes, and use type parameterization (generics). They are in the language to make your programs simpler. And when you do that, don't confuse inheritance and aggregation. Inheritance should implement "is-a." Aggregation should implement "has-a."

Finally, keep it simple and stupid. Long names are simple, short names are complicated. Long methods are never simple. Good programmers write less code, not more, and bad designs lead to more and longer methods. Don't generalize unless you need it. The best programmers can accomplish the same task in half the time, with a quarter of the code, and 10 times more reliably than the worst programmer. And don’t be too proud. The best programmers in college cannot compare to the best programmers with 10 or 15 years of experience. Make sure that’s your future self.

PROGRAM STYLE

Finally, we can develop lots of simple habits to make our programs more readable. Many of these are commonly included in style guides, and lots of organizations have their own style guides. They may be required or optional. They key is to pick a style and stick with it. The details of the conventions usually are not important but being consistent is.

The basics are pretty simple. A study all the way back in the 1960s asked “how far should we indent,” and found very strong evidence that two to four characters is ideal. Fewer than two is hard to see, and more than four makes programs too wide. Never use tabs; they look different in every editor and every printer, so what you see will not be what others see. Even worse is to mix tabs and spaces. Use plenty of white space. That makes names easier to find and read and is a habit that pays dividends down the road as your eyes age. And never put more than one statement per line. (By the way, I fully realize that most IDEs love tabs. Tools suck.)

A good style guide should cover several things:

- *Case for names ... including variables, objects, methods, classes, packages, and files*
- *Guidelines for choosing names*
- *Width, special characters, and splitting lines*
- *Location of statements*
- *Organization of methods and use of types*
- *Use of variables*
- *Control structures*
- *Proper spacing and white space*

- *Comments—where and how stylistically*
- *Location of left curly braces*
- *Location of else statements*

And don’t forget to mention language. I will never forget a graduate student who left me a program that was very well commented. In Korean.

To summarize, the little things that programmers do have a major impact on readability. In turn, readability has a major impact on maintainability, and maintainability is a primary determining factor in the long-term cost of the system. The minor decisions that you make as a programmer determines how much money your company makes. That is what engineering means!

Be tidy, my friends.

DESIGNING FOR CHANGE [11]

The first level of engineering software for maintainability is coding. As I said before, tidy programs are easier to understand and modify. The second level is during design and integration. Well organized, simple, and clean interfaces among components make future changes easier.

INTEGRATING SOFTWARE COMPONENTS

In the 20th century, many programs were single stand-alone systems. 21st century software, however, is often collections of integrated software systems that live in an eco-system, not in isolation. They interact with other applications, they use shared libraries, they communicate with related applications on the internet and the cloud, and they share data with multiple computing devices. This increase in couplings among software systems is very powerful but makes careful design more important. In effect, distributed computing has become the norm, not the exception.

Many factors make this kind of integration-heavy software difficult. Networks are notoriously unreliable and much slower than the devices that use them. Programs are diverse in terms of language, operating system, data formats, and many other characteristics. And of course, change is inevitable and continuous. The eco-system changes every time hardware or software is updated and when new applications are brought on board.

Back "in the day," software components were coupled through function calls and shared, non-local, variables. Now software components are coupled through networks, messages, the cloud, databases, and other convenient mechanisms. The notion of coupling has been extended:

- *Tight Coupling:* Dependencies among the methods are encoded in their logic. Changes in method **A** may require changing the logic in a coupled module **B**. This was common in the 1980s and often indicated a functional design of software.
- *Loose Coupling:* Dependencies among modules are encoded in the structure and data flows. Changes in module **A** may require changing data uses in module **B**. This was the original goal of data abstraction and object-oriented concepts [12], which are now embedded firmly in early programming courses such as data structures and in object-oriented programming languages.
- *Extremely Loose Coupling (ELC):* Dependencies are encoded only in the data contents. Changes in **A** only affect the **contents** of **B**'s data. This is the primary motivating goal for most advances in distributed software and web applications over the last 20 years.

eXtensible MARKUP LANGUAGE (XML)

ELC leads directly to XML as a simple but powerful way to support it. Passing data from one software component to another has always been difficult. The two components must agree on format, types, and organization. In method calls, this agreement is syntactically hard-coded into method signatures. But we don't have the same level of type checking in distributed software (such as web applications), and we have additional requirements:

- The software must support extremely loose coupling
- Software components may be integrated dynamically (during run time)
- The software must adapt to frequent changes

I examine this through a common form of coupling: One program, **P2**, needs to use data created by another program, **P1**. In the 1970s, before the emphasis on data abstraction, **P1** would write the data into a file. To save space, files were saved in very compact forms (binary, not text), and with rigidly structured records that were often not documented. If **P2** is written **after** the source of **P1** is no longer available, the structure of the file would have to be deduced by a slow trial-and-error process of reading bytes into memory, and printing them in different formats to see if they legible.

By the 1980s, the concept of data abstraction led to the file being controlled by a "wrapper module" that could read and write the file. The wrapper was shared by both P1 and P2. This was still slow, and since the wrapper module was shared among multiple programs, it was very difficult to change it or the structure of the file. Adding a single field could disrupt dozens of programs.

The modern solution is to use file formats that are free-form, textual, and self-documenting. That is, XML. XML files take a lot more space:

```
<book>

      <title>Don’t Make Me Think, Revisited: A Common Sense Approach to
Web Usability</title>

      <author>Steve Krug</author>

      <year>2014</year>

</book>
```

but make programs much more maintainable.

Sharing Data and Message Passing

A major factor in the maintainability of software is how data is shared and information is passed among software components. Four major styles are:

1. *File storage:* This is a traditional method where one program writes to a file that another later reads. Both programs need to agree on several things, including the file name and location, the format of the file, when the file can be read and written to, and who will delete the file.

2. *Shared database:* A more robust method is to replace a file with a database. This allows most decisions to be encapsulated in the table design, and offloads much of the effort onto a database package.

3. *Remote Procedure Calls (RPC):* This is similar to an old-fashioned method call, except the procedure operates in a different memory space, runs inside a different owning process, and may be on a different computer. Communication is real-time and synchronous, and needs to be managed by robust and sophisticated software. RPCs typically expect the caller to pause execution until the callee completes, and returns some result.

4. *Message passing:* This is more asynchronous than RPCs. One module sends a message to a common *message channel*, and other modules read the messages. The sender does not necessarily wait for the receiver to respond, and the receiver does not necessarily read the message immediately. Both modules must agree on the channel and message format, usually without built-in type checking. XML is commonly used to encode messages because of its flexibility and self-documenting nature.

The distinction between *synchronous* and *asynchronous* messages is very important. A telephone call is synchronous, because both parties need to be on the call at the same time. Synchronous communication has greater bandwidth but introduces two restrictions. First, both parties have to be available at the same time, and second, communication must be in real-time. Voice mail is asynchronous. We leave messages for later retrieval, so the real-time aspects are less important. In web applications, the traditional request-response cycle imposes a synchronous model, but Ajax introduces asynchronous calls.

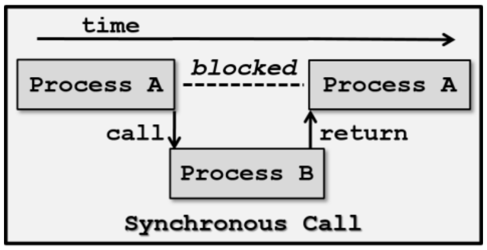

Figure 1: A synchronous call

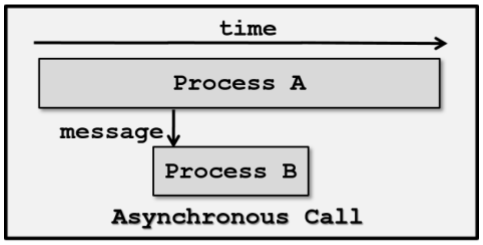

Figure 2: An asynchronous call

Asynchronous messaging architectures are very powerful, but we have to change how we design and develop software. We have decades of deep knowledge about using files, shared databases, and remote procedure calls, but we teach college students relatively little about asynchronous software engineering.

Advantages and Disadvantages of Message Passing

Data encapsulation is stronger than with databases and file storage. RPCs have reliability problems because any glitch in the network will disrupt the communication, whereas messages simply wait until the network recovers. Message passing reduces dependencies, making it less likely that changes will cause problems elsewhere in the system. That is, fewer ripple effects. This improves maintainability, as well as reliability, security, and scalability.

On the other hand, the lack of deep knowledge for how to write asynchronous software makes them less likely to be reliable, and harder to understand (a negative for maintainability).

The programming model is different and complex. Logic is distributed across several software components, which does not match how we teach topics like algorithms. Many universities do not teach event-driven software at all.

The sequencing of software tasks is harder. Message systems do not guarantee when the message will arrive, so messages sent in one sequence may arrive in a different sequence. In fact, many applications that could use asynchronous events intentionally do not because of the engineering challenges.

Using Design Patterns to Integrate

Enterprise systems contain hundreds, sometimes thousands, of separate applications. They are a mix of custom-built components, third party vendors, and legacy systems. They are often designed with multiple tiers that run on different computers and different operating systems. Many companies depend on large enterprise systems that encapsulate the operation of many aspects of the business. Patriot Web is an example at my university. That and Blackboard, which is used widely to support teaching, are both unreliable, hard to modify, and have extreme usability problems. Although universities suffer more than many companies, problems with enterprise systems are quite common. Many actually grew from multiple smaller software components, just like small towns grow together, slowly integrating to form cities. This type of organic growth invariably creates maintenance debt and confusion—just think of the last time you saw a street change its name without warning.

Integrating diverse applications into a coherent enterprise application will be an important task for years to come. It's not easy, but understanding important goals like maintainability and usability help. Lots of frameworks and integration platforms are available. One of the most important thing to understand is their set of basic assumptions. Some assume that the data never changes, but new functions will be continually added. Thus, APIs should be strong and clear, although the central database may be very hard to change. Others assume that the functions will remain constant, but they will be adapted to new hardware platforms and to new users. Thus, new features will be hard to add, but the UI should be easy to change.

These are about tradeoffs. When systems are integrated, we usually can’t support maintainability in all aspects, so a crucial early decision is which types of changes should be planned for. If an organization’s software architect gets that wrong, the entire organization will suffer for years. One of the hardest part about making these decisions is that the planning team must be able to look 5 or 10 years into the future. This is a rare ability.

To summarize, reducing coupling is a key goal to ensuring maintainability at any level. Software engineers have known about the importance of coupling since the 1970s, although the specifics change with each generation of language, hardware, and software engineering technologies. The primary goal of coupling is to reduce the assumptions that two software components have to make when exchanging data. Loose coupling means fewer assumptions. Local method calls have very tight coupling, as do remote procedure calls. Worse, RPCs come with the complexity of distributed processing. Message passing, however, has extremely loose coupling and is thus a strong way to increase maintainability.

BIBLIOGRAPHY

[1] J. Offutt, "A Historical Introduction to Software Maintenance & Evolution," January 2018. [Online]. Available: https://cs.gmu.edu/~offutt/classes/437/maintlectures/maintIntroduction.html.

[2] B. Randell, 2001. [Online]. Available: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/.

[3] J. Offutt, "Overview of Software Maintenance and Evolution," January 2018. [Online]. Available: https://cs.gmu.edu/~offutt/classes/437/maintlectures/maintEvolutionOverview.html.

[4] Sommerville, Software Engineering, edition 10, Addison-Wesley Publishing Company Inc., 2015.

[5] IEEE, "Standard Glossary of Software Engineering Terminology," *Institute of Electrical and Electronic Engineers,* no. 1990, pp. ANSI/IEEE Std 610.12-1990.

[6] Wikipedia, "Change impact analysis," [Online]. Available: https://en.wikipedia.org/wiki/Change_impact_analysis. [Accessed January 2018].

[7] Wikipedia, "Lehman's laws of software evolution," [Online]. Available: https://en.wikipedia.org/wiki/Lehman%27s_laws_of_software_evolution. [Accessed January 2018].

[8] J. Offutt, "How the Web Resuscitated Evolutionary Design," October 2015. [Online]. Available: https://cs.gmu.edu/~offutt/classes/437/maintlectures/evolutionaryWeb.html.

[9] D. Norman, The Design of Everyday Things, Basic Books, 1988.

[10] J. Offutt, "Coding for Change," January 2018. [Online]. Available: https://cs.gmu.edu/~offutt/classes/437/maintlectures/codingForChange.html.

[11] J. Offutt, "Designing for Change," January 2018. [Online]. Available: https://cs.gmu.edu/~offutt/classes/437/maintlectures/designForChange.html.

[12] D. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM,* pp. 15(12):1053-1058, December 1972.