

Maintenance and Evolution Overview

Brittany Johnson

Adapted from slides by Paul Ammann & Jeff Offutt

Software Maintenance

"When the transition from development to evolution is not seamless, the process of changing the software after delivery is often called *software maintenance*."

- **Sommerville, 2004**

Modifying a program after it has been put into use

Maintenance does not normally involve major changes to software architecture

Changes are implemented by modifying existing components and adding new components to the system

Maintenance requires program understanding

Importance of maintenance

Organizations have **huge investments** in their software systems – they are critical business assets

To maintain the value of these assets to the business, they must be **changed** and **updated**

Much of the **software budget** in large companies is for modifying existing software

WHEN YOU HEAR THIS:



Software changes are inevitable

We cannot avoid changing software

- new **requirements** emerge when software is used
- The **business environment** changes
- **Faults** must be repaired
- **New computers** and equipment is added to the system
- The **performance** or **reliability** may have to be improved

Software is **tightly coupled** with the environment.

A key problem for organizations is **implementing** and **managing change** to their existing software.



Management Myths

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:** Book of standards may exist, but is it used?
Are software practitioners aware of its existence?
Does it reflect modern SE practice?
Is it complete?
Is it streamlined to improve time to delivery while still maintaining focus on quality?



Management Myths

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 decide to outsource the software project to a third party, I can just relax and let them build it.

- **Reality:** If an organization does not understand how to manage and control software projects internally, it won't be able to outsource effectively.

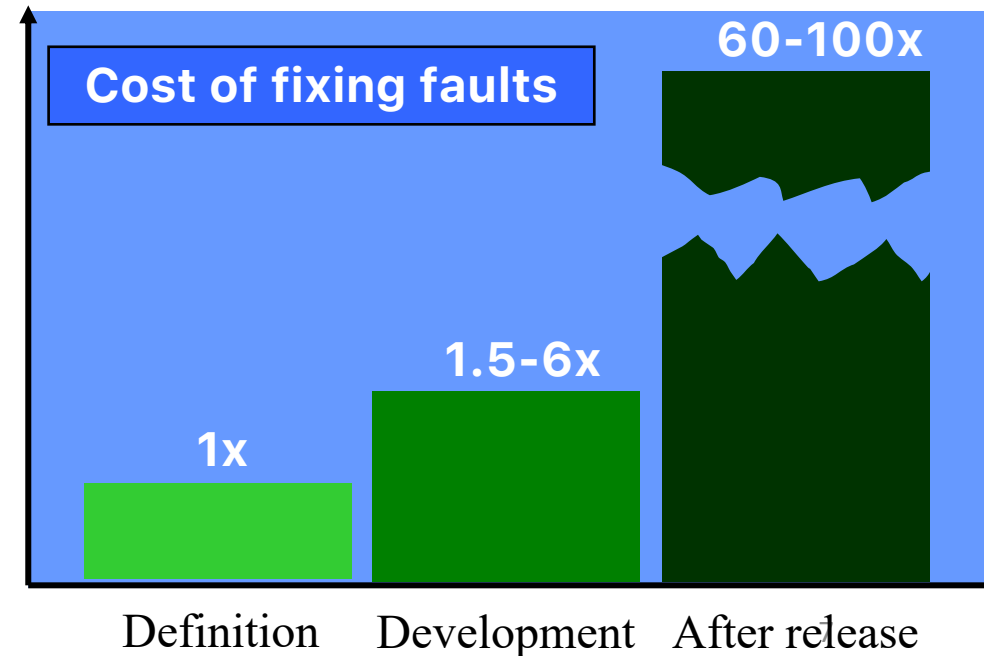
Customer Myths

Myth: A general statement of objectives is enough to start writing programs – we can fill in the details later.

- **Reality:** A poor upfront definition is the major cause of failed software efforts. If you don't know what you want at the beginning, you won't get it

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 change varies with the time at which it is introduced



Practitioner's Myths

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 60-80% 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 inception of a project – the formal technical review. Software reviews are more effective than testing for finding certain classes of software defects.

Practitioner's Myths

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.



Software maintenance problems

Most computer systems are difficult and expensive to maintain

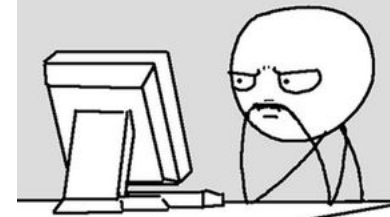
Software changes are poorly designed and implemented

The repair and enhancement of software often injects new faults that must be repaired

99 little bugs in the code,
99 little bugs.



Take one down, patch it around...
127 little bugs in the code!



Maintenance costs

Usually greater than development costs (2 – 100 times depending on the application)

Affected by both technical and non-technical factors

Increases as software evolves

- Maintenance corrupts the software structure, making further maintenance more difficult

Aging software can have high support costs (old languages, compilers, etc.)

Maintenance cost factors

Team stability

- Maintenance costs are lower if the same staff stay involved

Contractual responsibility

- If the developers of a system are not responsible for maintenance, there is no incentive to design for future change

Staff skills

- Maintenance staff are often inexperienced and don't have much domain knowledge

Program age and structure

- As programs age, changes degrade the code, design, and structure and they become harder to understand and change

Additional maintenance terms

Maintainability: The ease with which software can be modified

Impact analysis: Understanding how changes in one software component can impact other components

Ripple effect: How changes transfer through the system, primarily through data and control flow connections

Traceability: The degree to which a relationship can be established between two or more software artifacts

Legacy systems: A software system that is still in use but the development team is no longer active

Reality check!

Sorry to say, but...

All the previous data comes from publications in the 1990s...

Based on knowledge from the 1980s...

When our software was "single-building size"!

**How out of date is this information for building
software of today??**

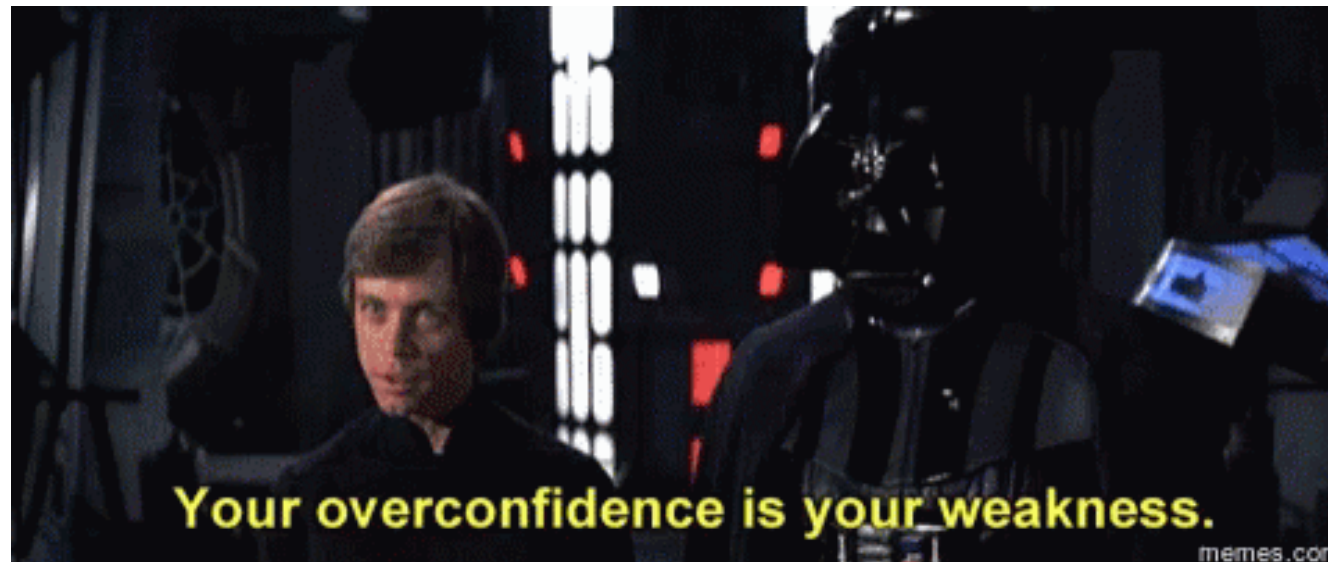
VERY!!

(Over) confidence

Knowing we're right

Arrogance: based on hopes and dreams

Confidence: based on experience, knowledge, and ability



Maintenance vs. Evolution

Software **Maintenance**

- Activities required to keep a software system operational after it is deployed

Software **Evolution**

- Continuous changes from a lesser, simpler, or worse system to a higher or better system



Software Evolution

“Software development does not stop when a system is delivered but continues throughout the lifetime of the system”

- **Sommerville, 2004**

The system changes related to **changing needs** – business and user

The system **evolves continuously** throughout its lifetime

Modern agile processes emphasize getting a few functionalities running, then adding new behaviors over time

Lehman's Laws of Software Evolution

1. Law of **Continuing Change** (1974)

- 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** (1974)

- As evolving program changes, its structure becomes more complex, unless active efforts are made to avoid this phenomenon

3. Law of **Self Regulation** (1974)

- 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

Lehman's Laws of Software Evolution

4. Law of **Conservation of Organizational Stability** (1980)

- 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** (1980)

- Over the lifetime of a system, the incremental system change in each release is approximately constant

6. The Law of **Continuing Growth** (1980)

- The functionality offered by systems has to continually increase to maintain user satisfaction

Lehman's Laws of Software Evolution

7. The Law of **Declining Quality** (1996)

- The quality of systems will appear to be declining unless they are adapted to changes in their operational environment

8. The **Feedback System Law** (1996)

- Evolution processes incorporate multi-agent, multi-loop feedback systems and you have to treat them as feedback systems to achieve significant product improvement

The pace of change is increasing

Hardware advances leads to **new, bigger software**

The **rate of change** (that is, new features) is **increasing**



How can we deal with the spiraling need to handle change??