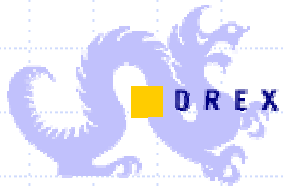

"Software Aging"

D. L. Parnas

Software Aging

"Programs, like people, get old. We can't prevent aging, but we can understand its causes, take steps to limit its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable. ... (We must) lose our preoccupation with the first release and focus on the long term health of our products."

D.L. Parnas

Software "Aging"?

- "It does not make sense to talk about software aging!"
  - Software is a mathematical product, mathematics does not decay with time.
  - If a theorem was correct 200 years ago, it will be correct tomorrow.
  - If a program is correct today, it will be correct 100 years from now.
  - If a program is wrong 100 years from now, it must have been wrong when it was written.
- All of the above statements are true, but not really relevant.

Software Does Age

- Software aging is gaining in significance because:
  - of the growing economic importance of software,
  - software is the "capital" of many high-tech firms.

Software Does Age

- The authors and owners of new software products often look at aging software with disdain.
- "If only the software had been designed using today's languages and techniques …"
- Like a young jogger scoffing at an 86 year old man (ex-champion swimmer) and saying that he should have exercised more in his youth!

The Causes of Software Aging

- There are two types of software aging:
  - **Lack of Movement:** Aging caused by the failure of the product's owners to modify it to meet changing needs.
  - **Ignorant Surgery:** Aging caused as a result of changes that are made.
- This "one-two punch" can lead to rapid decline in the value of a software product.

Lack of Movement

- Unless software is frequently updated, its user's will become dissatisfied and change to a new product.
- Excellent software developed in the 60's would work perfectly well today, but nobody would use it.
- That software has aged even though nobody has touched it.
- Actually, it has aged because nobody bothered to touch it.

Ignorant Surgery

- One must upgrade software to prevent aging.
- Changing software can cause aging too.
- Changes are made by people who do not understand the software.
  - Hence, software structure degrades.

Ignorant Surgery (Cont'd)

- After many such changes *nobody* understands the software:
  - the original designers no longer understand the modified software,
  - those who made the modification still do not understand the software.
- Changes take longer and introduce new bugs.
- Inconsistent and inaccurate documentation makes changing the software harder to do.

The Cost of Software Failure

- Inability to keep up,
- reduced performance,
- decreasing reliability.

Inability To Keep Up

- **As software ages, it grows bigger.**
- "*Weight gain*" is a result of the fact that the easiest way to add a feature is to add new code.
- Changes become more difficult as the size of the software increases because:
  - There is more code to change,
  - it is more difficult to find the routines that must be changed.
- **Result:** Customers switch to a "younger" product to get the new features.

Reduced Performance

- *As the size of the program grows, it places more demands on the computer memory*.
- Customers must upgrade their computers to get acceptable response.
- Performance decreases because of poor design that has resulted from long-term ad hoc maintenance.
- A "younger" product will run faster and use less memory because it was designed to support the new features.

Decreasing Reliability

- *As the software is maintained, errors are introduced.*
- Many studies have shown that each time an attempt is made to decrease the failure rate of a system, the failure rate got worse!
- That means that, on average, more than one error is introduced for every repaired error.

Decreasing Reliability (Cont'd)

- Often the choice is to either:
  - abandon the project
  - stop fixing bugs
- For a commercial product, Parnas was once told that the list of known unrepaired bugs exceeded 2,000.

Reducing the Cost of SW Aging

- *We should be looking far beyond the first release to the time when the product is old.*
- Inexperienced programmers get a "rush" after the first successful compile or demonstration.
- Experienced programmers realize that this is only the beginning ...

Reducing the Cost of SW Aging (Cont'd)

- Responsible, professional, organizations realize that more work is invested between the time after the first successful run and the first release than is required to get the first successful run.
- Extensive testing and rigorous reviews are necessary.

Preventive Medicine

- Design for success
- Keep records (documentation)
- Seek second opinions (reviews)

Design for Success

- *Design for change.*
- This principle is known by various names:
  - information hiding
  - abstraction
  - separation of concerns
  - data hiding
  - object-orientation

Design for Change

- To apply this principle one begins by trying to characterize the changes that are likely to occur over the "lifetime" of a product.
- Since actual changes cannot be predicted, predictions will be about classes of changes:
  - changes in the UI
  - change to a new windowing system
  - changes to data representation
  - porting to a new operating system ...

Design for Change (Cont'd)

- Since it is impossible to make everything equally easy to change, it is important to:
  - estimate the probabilities of each type of change
  - organize the software so that the items that are most likely to change are "confined" to a small amount of code

Why is Design for Change Ignored?

- Textbooks fail to discuss the process of estimating the probability of change for various classes of changes.
- Programmers are impatient because they are too eager to get the first version working.
- Designs that result from this principle are different from the "natural" designs of the programmer's intuition.

Why is Design for Change Ignored? (Cont'd)

- Few good examples of the application of the principle. Designers tend to mimic other designs they have seen.
- Programmers tend to confuse design principles with languages.
- Many practitioners lack training in software development.

Keeping Records (Documentation)

- Even when software is well designed, it is often not documented.
- When documentation *is* present it is often:
  - poorly organized
  - inconsistent
  - incomplete
  - written by people who do not understand the system

Documentation

- Hence, documentation is ignored by maintainers.
- Worse, documentation is ignored by managers because it does not speed up the initial release.

Second Opinions (Reviews)

- In engineering, as in medicine, the need for reviews by other professionals is never questioned.
- In designing a building, ship, aircraft, there is always a series of design documents that are carefully reviewed by others.

Reviews

- This is not true in the software industry:
  - Many programmers have no professional training in software at all.
  - Emphasis of CS degrees on mathematics and science; professional discipline is not a topic for a "liberal" education.
  - Difficult to find people who can serve as quality reviewers; no money to hire outsiders.
  - Time pressure misleads designers into thinking that they have no time for proper reviews.
  - Many programmers resent the idea of being reviewed.

Reviews

- *Every design should be reviewed and approved by someone whose responsibilities are for the long-term future of the product.*

Why is Software Aging Inevitable?

- *Our ability to design for change depends on our ability to predict the future.*
- We can do so only approximately and imperfectly.
- Over a period of years:
  - changes that violate original assumptions will be made
  - documentation will never be perfect
  - reviewers are bound to miss flaws ...

Why is Software Aging Inevitable? (Cont'd)

- Preventive measures are worthwhile but anyone who thinks that this will eliminate aging is living in a dream world.

Software Geriatrics

- **Retroactive Documentation:**
  - A major step in slowing the age of older software, and often rejuvenating it, is to upgrade the quality of the documentation.
- **Retroactive Modularization:**
  - Change structure so that each module hides a design decision that is likely to change.

Software Geriatrics (Cont'd)

- **Amputation:**
  - A section of code has been modified so often, and so thoughtlessly, that it is not worth saving.
- **Major Surgery (Restructuring):**
  - Identify and eliminate redundant components and gratuitous dependencies.

Planning Ahead

- It's time to stop acting as if *"getting it to run"* was the only thing that matters.
- Designs and changes have to be documented and carefully reviewed.
- If it's not documented, it's not done.
- In other areas of engineering, product obsolescence is recognized and included in design and marketing plans.
- The same should be done for software engineering.