# Software Aging

*David Lorge Parnas*

McMaster University, Hamilton, Ontario, Canada L8S 4K1

Programs, like people, get old. We can't prevent aging, but we can understand its causes, take steps to limits its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable. A sign that the Software Engineering profession has matured will be that we lose our preoccupation with the next release and focus on the long term health of our products. Researchers and practitioners must change their perception of the problems of software development. Only then will Software Engineering deserve to be called Engineering.

# Structural Aging

Buildings, 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 building is no longer viable. A sign that the Structural Engineering profession has matured will be that we lose our preoccupation with the next project and focus on the long term health of buildings. Researchers and practitioners must change their perception of the problems of architecture. Only then will Structural Engineering deserve to be called Engineering.

# Software Aging is Nonsense!

- Software is a mathematical product.
- Mathematics doesn't 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 it is wrong 100 years from now, it must have been wrong when it was written.
- It makes no sense to talk about software aging.

## True, but not really relevant.

Software Does Get Old and it costs more in old age than it did at birth.

# Software Products Do Exhibit Aging!

It may not be aging but it sure feels like aging.

- Old software has begun to cripple its once-proud owners.
- Old products are considered a burdensome legacy
- Obesity often accompanies aging.
- Increasing effort required to support legacy software.
- Like human aging, software aging is inevitable, but
- there are things that we can do to slow down the process.
- Sometimes, we can reverse its effects (temporarily).

Software aging is not a new phenomenon.

Software aging is gaining in significance.

Owners of new software products often look at aging software with disdain.

"*The new products won't age. They are written in Object C-dula ++" and using xx.oriented techniques.*"

Some of those old products were once proud youth, and written using the latest fad.

There is no fountain of youth.

# What Causes Software Aging?

## Two basic types of software aging:

- Failure to keep up with changing environment,
- Tissue damage resulting from maintenance.

A “one-two punch”!

Damned if you do; damned if you don’t.

Both lead to a decline in the value of a software product.

# Causes of Software Aging

## Lack of movement

- Our expectations have changed
- We are no longer willing to use the clumsy software interfaces of yesterday.
- We demand interactive access; don't even think about batch systems.
- Old software would still do its job if used, but nobody would use it.
- Old software can be modified and extended to meet modern expectations.
- Old software must compete with young products

When changes are **not** made, it seems as if the software has aged!

A product that was considered great a few years ago, is not useful although it hasn't changed.

# More Causes of Software Aging

## Ignorant surgery

Designers of software had simple concept in mind when writing the program.

Understanding that concept allows one to find sections to be altered.

Understanding that concept means understanding the interfaces.

Changes made without understanding design concept almost always invalidate it.

Changed code is inconsistent with the original concept.

After changes, one must know both the original design concept, and the exceptions.

After such changes, the original designers no longer understand the product.

***Nobody*** **understands the modified product.**

continued.....

# More Causes of Software Aging

## Ignorant surgery (continued)

When nobody understands the product,

- Changes take longer.
- Changes are more likely to introduce “bugs”.

The problem is exacerbated when maintainers do not have time to update the documentation.

The documentation becomes increasingly inaccurate thereby making future changes even more difficult.

# More causes of software aging

## Cancerous Growth

We live in a world that requires many versions of a software product.

Many companies allow the product line to split, with the second line inheriting the properties of the first.

This policy leads to explosive, difficult to constrain, growth.

Changes must be made to many versions instead of to one.

The cost of maintenance gets very high.

It is no longer possible to keep up with the market on all the versions.

The diversity grows and is hard to stop.

# More Causes of Software Aging

## Aging and Disappearing Programmers

Programmers move on to other projects.

Programmers get promoted to managers.

Programmers forget what they did and why they did it.

Programmers change their “style”.

# More Causes of Software Aging

## Bad Documentation

- Information not available.
- Information not correct.
- Information hard to find.
- Information must be verified.
- Information is imprecise, easily misinterpreted.

Documents are not kept “alive”.

The situation keeps getting worse.

Movement is visibly slower

- It takes time to find information
- It all has to be verified.

That is why you will find so much emphasis on precise and clear documentation in this class and others.

# Pseudo Software Aging

## Kidney failure

Often *confused* with software aging!

Failure to release allocated memory.

Files require pruning.

Swap and file space are diminished.

Performance degrades.

Often a congenital design failure.

Can strike at any age.

May be the result of ignorant surgery

May be exacerbated by changing usage patterns.

Perceived as “leaking”.

## More easily cured than aging!

- Dialysis: cleans up the file system and memory
- Kidney transplant: new allocation routines, garbage routines

# The costs of software aging

## Rapidly Rising Maintenance Costs

It’s harder to know where to make changes.

Changes are harder to make.

There are more errors.

Testing becomes more of a burden.

Documentation takes longer to update.

# The Costs of Software Aging

## Inability to keep up with others

Younger products have desired features.

Younger products can be adapted more quickly.

Younger products win more market share.

Obese products consume more resources

Older products starve to death because they cannot catch new “prey” and they need more “food”.

# The Costs of Software Aging

## Reduced performance

The size of the program grows.

More demands on the computer memory.

More delays caused by swapping.

Performance decreases because of poor (poorly understood) design.

Customers must upgrade their hardware to get acceptable response.

Some customers switch to younger products to obtain better performance.

# The Costs of Software Aging

## Decreasing reliability

Errors are introduced.

Each error corrected may introduce more than one error.

Improvements can make things worse.

Often a product must be abandoned.

Some products that we use daily have thousands of *known* bugs.

Large groups are devoted to customer requested repairs.

**Nobody is shocked by such statements**

# Software Aging in the Telecom Industry

**Nowhere is software aging more evident than in the telecom industry!**

- There is a great demand for change
- There is a lot of competition
- Interoperability increases the demand for change
- Interoperability increases the competitive pressure
- The need to move into markets fast has led to cancerous growth
- Rapid change has led to obesity and bad structure.
- Documentation is often postponed until slower times.
- Rapid growth has led to “ignorant” surgery.

# Reducing the costs of Software Aging Neo-Natal Phase

Outgrow “ran the first time” elation.

- It is not “right the first time” that matters.
- “Clean compile” means even less.

“Configuration Management” while vitally important, is a losing battle with increasing entropy. It’s not the solution.

Controlling software aging, requires good design, not just good management.

*Software geriatrics begins with neo-natal care!*

# Reducing the Costs of Software Aging Neo-Natal Phase -Preventive Medicine

## Design for success!

The only software that doesn't change is software that isn't used. Designing for change is designing for success.

Apply:

- "information hiding",
- "abstraction",
- "separation of concerns",
- "data hiding", or
- "object orientation".

Begin by characterising the likely classes of changes.

Estimate the probabilities of each type of change.

Organise the software to confine likely changes to a small amount of code.

Provide an "abstract interface" that abstracts from the changes.

Implement "objects" that hide changeable data structures.

# Preventive medicine

## These are old ideas. Have they failed?

When I examine industry code, I do not see consistent application of the principle.

- Many textbooks on software mention this technique, but they cover it in a superficial way. The principle is simple; applying it requires a lot of thought.
- Many programmers are too impatient. They find the design of abstract interfaces boring.
- Deadlines do not allow time to design for change.
- Designs that result from a careful application of information are not natural for many program designers.
- Designers mimic older designs.
- Many confuse design principles with choice of language.
- Many software designers are self-taught (or worse)!
- Researchers are “preaching to the choir” ignoring old unsolved problems and unused solutions because they want to innovate.

The principle hasn’t failed; we have failed to apply it.

# Don't Depend on Miracle Drugs

Our field is filled with snake-oil salesmen:

- Structured Design
- Knowledge-Based Software Engineering
- O-O-O-O Technology
- Dr. Somebody's miracle drug

They promise easy solutions, but there are none.

Good design is ***hard***. It requires:

- careful consideration of possible changes,
- careful examination of constraints,
- careful design of interfaces
- careful review
- qualified people

# Don’t Depend on Languages

Programming Languages have been touted as the miracle solution since their first appearance.

Nobody wants to go back to assembler, but ...

We must recognise that software problems never went away in spite of the introduction of FORTRAN, ALGOL, PL/I, ADA and PROTEL.

You can do bad design in the best languages.

You can do good design in the worst languages.

# Preventive Medicine

## Medical records - documentation

Design concepts and decisions are not recorded.

Documentation is neglected by researchers.

Documentation is neglected by practitioners.

Some believe that the code is its own documentation;

- they have never read other people's code or
- their own code 10 months later.

Documentation is usually poorly organised, incomplete and imprecise.

Often the coverage is random, written when time and interest are there.

Where documentation is a contractual requirement, it is often done by people who need the documentation.

Some projects keep two sets of books.

Documentation that seems clear today, may be impenetrable tomorrow.

# Preventive Medicine

## Medical records - documentation

Documentation is not an “attractive” research topic.

- Researchers yawn,
- Everyone confuses it with proofs. Documentation must be “blah blah”.
- Developers ask if it will speed up their next release.
- Students try to do as little as possible.

Although there is never enough time to do documentation using design, there is always enough time to hunt for the information when it is needed.

It is your responsibility as an engineer to produce accurate design documents as you work on a design and to keep them up-to-date.

# Preventive Medicine

**We must start taking documentation more seriously.**

As in other kinds of engineering documentation, software documentation must be based on mathematics.

- Each document will be a representation of one or more mathematical relations.
- The notation must be concise.
- The notation must be precise.
- The notation must be formally defined.

The notation must be based on engineering mathematics, not theorem proving logic.

Logic is a descriptive tool, not just a reasoning tool.

# Preventive Medicine

## Second opinions - reviews

Reviews are standard practice in Engineering

They are *not* standard practice among programmers. Why?

- Not engineers, “fallen” scientists and mathematicians.
- Professionalism not taught in “liberal” education.
- Lack of professional documentation to review against. Review becomes “show and tell” followed by a chat.
- Cottage industry - no reviewers.
- Time-pressure leads to a “short cut” to a long road.
- This is an “art”; I do it my way.

Designers naturally focused on short-term goals.

Reviewers must represent long-term interests of the manufacturer or client.

Review for ease of change, must become standard.

# Why Conventional Reviews are Ineffective

(1) The reviewers are swamped with information,.

(2) Most reviewers are not familiar with design goals.

(3) There are no clear individual responsibilities.

(4) Reviewers can avoid potential embarrassment by saying nothing.

(5) The review is conducted as a large meeting where detailed discussions are difficult.

(6) Presence of managers silences criticism.

(7) Presence of uninformed may turn the review into a tutorial.

(8) Specialists are asked general questions.

(9) There is no systematic review procedure.

(10) Unstated assumptions (subtle design errors) may go unnoticed.

# Effective Reviews are Active Reviews

A dilemma:

- Errors in design documents should be found before the documents/systems are used.
- Errors in documents are usually found when the documents are used.

Another dilemma:

- Everyone's work requires review
- It's easier to say "OK" than to find subtle errors
- Reviewer's work is not reviewed.

One more dilemma:

- No individual knows enough to review all aspects of a design.
- When working in a group, people tend to relax in the belief that others are working the problem.

**Solutions:**

- Make the reviewers use the documents.
- Make the reviewers document their analysis.
- Have specialised reviews. Ask the reviewer about things that they know.
- Make the reviewers provide specifics not "ok".

# Reviewing Design Documents

Design the review process based on the nature of the document.

Begin by identifying desired properties.

Prepare questionnaires for the reviewers. Ask them questions that:

- make them use the document.
- make them demonstrate that the desired properties are present.
- ask for sources of information to support the answers to other questions.

# Inspecting Programs

It is the code that “hits the road”.

Getting the requirements right, the structure right, the interfaces right, the structure right, etc. are all important but we have to check the code.

The same review principles apply.

- Make the reviewers use the material they review.
- Make the reviewers answer questions.
- Ask the reviewer about things that they know.
- Make the reviewers provide specifics.

We want to compare the completed programs with previously reviewed specifications.

We ask the reviewers to produce descriptions.

We then show that the descriptions match the specifications.

It is hard work but it produces results.

- We get good documentation for future use
- We find errors in the best industrial code - programs that were considered correct.

# Software aging is inevitable

Our ability to design for change depends on our ability to predict the future.

We will make changes that violate our original assumptions.

Documentation, even if formal and precise, will never be perfect.

Reviews will overlook some issues.

Preventive measures are worthwhile, but we cannot eliminate aging.

We have to think about how to care for aging software.

# Software geriatrics

## Stopping the deterioration

- The first step: slow the deterioration.
- Introducing or recreating, structure every time that changes are made.
- Design principles can guide maintenance.
- Revised data structure or algorithm can be hidden.
- Careful reviews must insure that each change is consistent with the intent of the original designers,

Easier to say than to do.

- It is hard to practice caution in the good times.
- “Cancerous growths” (new versions) often accepted by people in a hurry.
- New versions must be examined and a good family structure found.
- All versions must fit in that mould.
- New documents must be created and reviewed.

Nobody wants to take the time in good times.

“We have more important things to do.” is one of the biggest lies in the industry.

# Software Geriatrics

## Retroactive documentation

To slow the aging upgrade the documentation!

Documentation too often neglected in maintenance

- "There is no time".
- "It's already so bad, we can't fix it."
- "Let's just put a memo in the file".

Correcting the documentation can be a major project, but worth undertaking.

Redoing the documentation often leads to improvements in the software.

Writing the documentation is a way to study and understand the software.

It makes the next person's job easier too.

However, if there is a large family created by cancerous growth, restructuring may be a only practical prerequisite for documentation.

# A small sample of precise documentation

***Specification***

| **UseColr** | external variables: v, r, w, b, colr, colw |
|---|---|

$\mathbf{R}_5$(,) = *partial_flag*(‘v, ‘r, ‘w, ‘b) $\wedge$ (‘colr = ‘v‘$_w$) $\wedge$
(‘r $\leq$ ‘w) $\wedge$ ((‘r < ‘w) $\Rightarrow$ (‘colr $\neq$ red))
$\Rightarrow$
*partial_flag*(v’, r’, w’, b’) $\wedge$ *same_colors*(‘v,v’) $\wedge$

| | ‘colr = | | | |
|---|---|---|---|---|
| | red | white | blue | |
| r’ \| | r’=‘r + 1 | NC(r) | NC(r) | |
| w’ \| | NC(w) | w’ =‘w–1 | w’= ‘w –1 | |
| b’ \| | NC(b) | NC(b) | b’ =‘b –1 | $\wedge$ NC (colr,colw) |

Tables have a formally defined meaning.

Done without specifying proof rules.

Documentation is not verification.

# Software geriatrics

## Plastic surgery

New modules can be introduced when making necessary changes.

Revised decisions can be hidden.

Often, saves work when change is made.

Will save work in the future.

Must be done throughout product when it is done.

Must be documented.

# Software geriatrics

## Amputation

Some code has been modified so often, and so thoughtlessly, that it is not worth saving.

Often it should never have been written at all.

The authors and owners are hesitant to see it go.

Short-term pain for long-term gain.

# Software geriatrics

## Major surgery - restructuring

First step - reduce the size of the program family.

Examine the various versions to determine why they differ.

Chose between parameterisation and hiding.

- Hide differences wherever that is possible.
- In other cases, make the difference a parameter.
- Replace many versions with one - abandoning the family lines that are replaced.
- Pay with “run time” and “system-construction time” rather than programming time.

It often pays to ship the same code to everyone - even those who have not paid for all the features.

Such a program is called “general” and the features can be “turned off”.

“Flexible Programs” are distinct from “General Programs”. They are easily changed.

“Parameterisation” is one form of “flexibility”.

# Taking Responsibilities Seriously

## Professional Engineers have responsibilities:

- Not to do shoddy work even if the boss says so
- Not to release work without adequate review
- Not to release work without good documentation
- Not to release work without adequate testing
- Not to release work without adequate inspection.
- To keep up with the latest in their profession.

## Managers have responsibilities too!

- Don’t just think about this product
- Don’t just think about today
- Think about the whole product line and future products.
- Give your employees the time to do it right
- Make sure it is done right. Reward quality
- Read Dilbert and think about it.