

## **What Software Engineers Should Do For You**

*David Lorge Parnas, P.Eng, Ph.D, Dr.h.c., Dr.h.c., FRSC, FACM, FCAE*

SFI Fellow, Professor of Software Engineering  
Director of the Software Quality Research Laboratory (SQRL)  
Department of Computer Science and Information Systems  
Faculty of Informatics and Electronics, University of Limerick

### **Abstract**

Software that is written for use in one country is often difficult to revise for use in another culture. One of the basic precepts of software engineering is, “Design Software for Change - You will have to”. Localisation is a very predictable type of change and good software design should make it easier. This talk reviews some basic software design principles and explores the ways that they can be used to make localisation easier.

• ***SOFTWARE QUALITY RESEARCH LABORATORY*** •



## The Software Quality Research Laboratory at UL

Newly established (about 1 year in the building)

**Staff:** Director, 5 senior researchers, 5 graduate students, ....

**Goal:** Tools and methods to build high quality software, *and know it.*

**Emphasis:** SQA (Software Quality Assessment)

**Key Approach:** Well structured, precise documentation used in design, design review, inspection, and testing.

Building on experience in safety-critical applications

Intent to extend approach to mission critical applications.

Ultimate goal: Software as professional as hardware.



## **Some Observations About Software Development**

The “rush to code” was wrong 35 years ago and is still wrong.

Design documents have to make real decisions or they are useless.

There are proven software design principles; we should use them.

We don't get what we aim for, but we get closer than if we don't aim at all.

It is easier to extend than revise.

- Using someone's code is usually easier than understanding and changing it.
- Adding programs that use what's there is easier than changing what is there.
- Using other people's programs is only reasonable if there are well designed and well documented interfaces that are designed with such use in mind.



## **Some Observations About Translation**

I can often identify someone's mother tongue by their English.

The same is true for any other language (except that I can't always do it).

Reading a translation you often identify the source language.

A description in one language often leads you astray when writing a description in another.

It is often easier to write than to translate.

The trick to producing good "translations" is to understand, forget what you read and write your own explanation.

Unfortunately, forgetting is difficult and we often do not trust our own understanding so we remain close to the original.

Localisation (much more than translation) should not be translation at all.



## **Some Observations About Software Development Projects**

The early workers often leave a mess for the later workers.

Short-sightedness is endemic; I'll get my job done and leave yours to you.

User interfaces are often restricted by low level decisions that assume that the next interface will be essentially the same.

Localisers inherit it all.



## Five Software Engineering Principles

### The Program Family Model

- First Proposed in 1976
- Intended to provide “what to do first” guidance
- Now studied under the “product line” banner.
- Really more general than that

Information Hiding - easing change by anticipation and encapsulation.

Designing for Extension and Contraction - consciously designed uses relation,

Interface Design and Specification - distinguishing what from how

Formal (language free) Documentation long a dream, now in sight.



## Design and Development of Program Families

A Program Family is a set of programs that have so much in common that it pays to study their common characteristics before looking at the differences and making choices among them.

You have a program family even if you only sell one member at a time

If a firm is developing a program family, it can pay to do so consciously.

Widely recognized that early design decisions costliest to change.

- Later design decisions are usually based on earlier ones and would have to be reconsidered if the earlier ones are reversed.
- Make the shared decisions first - they won't have to be reversed later.

Identify commonalities and parameters of variation.

Distinguish “family development” from individual product development.

“Product line” community does this at requirements level.

Commonalities may be in shared “engine”.



## Implications of the Program Family Concept for Localisation

Do not develop a complete member of the family and revise to get others.

Develop an incomplete “framework” and extend it to get the first member.

Go back to the framework for each subsequent member.

Add to the framework when you discover additional commonalities.

The framework or base may be usable by nobody, but you make it useful for some people by adding to it, not by modifying it.

The framework is not likely to be right the first time, but ...

Maintain the distinction and make the common changes in the framework not individual members.

The framework interface should be completely and precisely documented.



## The Information-Hiding Principle

Perhaps my most subversive idea!

Common belief in universally accessible detailed design documents.

Suggested:

- Divide the software according to design decisions, not control flow.
- Providing interfaces that “hide” (abstract from) these decisions
- Publish only the interface specifications, not the hidden decisions.

Two reviews (10 years apart).

“Nobody does it that way”

“Wrote down what all good programmers did anyway.”



## Dividing Systems Into Modules

### What do we mean by “module”?

- A work assignment for a programmer or programmer team.
- No other meaning and no criteria in this definition.

### What would make a module structure good?

- Parts can be designed independently.
- Parts can be tested independently.
- Parts can be changed independently.
- No decision made in two places.
- Integration goes smoothly.

### What’s the information-hiding principle?

- Each module’s implementation is a “secret”.
- Each module’s interface is abstractly and precisely described.
- No two modules share a secret. (revised decision affects only one)
- No module has two independently-changeable secrets.

• *SOFTWARE QUALITY RESEARCH LABORATORY* •



## What is a secret?

A design decision made in one module that need not be known to demonstrate the correctness of any other module.

If it's shared, it isn't a secret!

Secrets may be:

- data structures
- device interfaces
- user interfaces
- algorithms
- communication protocols
- the user's language
- type of interface
- any of the things that localisers have to change

• *SOFTWARE QUALITY RESEARCH LABORATORY* •



## How Do You Keep A Secret?

Make the secret uninteresting to the developers of other modules

Provide a precise interface specification.

Provide a complete interface specification.

Provide an accurate interface specification.

If other programmers can learn everything that they need to know from the specification, they won't care about the secret and won't ask to know it.

One should inspect code to make sure that correctness does not depend on the secrets of other modules.

Subtle assumptions such as direction of writing may be shared unintentionally.

Hiding this information requires experience, awareness, and determination.



## **Information Hiding and Program Families**

Another view of the same principle.

Hidden decisions are the ones that are likely to differentiate family members.

Knowing what to hide is much harder than hiding it.

Information hiding sounds easy but it is not. It is rarely done correctly and most people, even advocates, do not realise its implications.



## **Designing for Extension and Contraction**

Any program can be extended, but

- Can you do it without modifying existing code and still get exactly what you want?
- Can you do it without paying a performance penalty?

Programs have to be designed so that they can be easily extended and contracted. It does not happen without planning.

The trick is to use the right hierarchical (uses) structure.

Its not enough to have levels, you have to have the right levels in the right order.

Consider the ISO 7 layer model.

See the work by Courtois on “Near Complete Decomposibility”



## Hierarchical Structure

**Definition:** A structure is a description with parts and relations between them.

**Definition:** A hierarchical structure is a structure with no loops in its relation's graph.

Before you know what someone means by a hierarchical structure, you must ask what the parts are and what relation they mean.

Hierarchies are not necessarily trees.



## The Uses Hierarchy

**Parts:** programs

**Relation:** uses

**Definition:** uses:

Given program A with specification S and program B, we say that A *uses* B if A cannot satisfy S unless B is present and functioning correctly.

A diagram of the uses relation shows what subsets you can have.

Having the uses structure be hierarchical allows a virtual-machine analogy.

Found in Dijkstra's T.H.E. and in many examples of structured programming.

If there are no localised parts at the low levels, you can build localised family members by extension.

However, some low level services must be localised (e.g. communication); this must be well encapsulated.



## **Interface Design**

Low level interfaces should be “transparent”, i.e. not prevent any functions that you might want to perform.

Each level removes capability - make sure you only remove what you will not need, (e.g., do not remove error reporting capability).

Low level interfaces should be global, i.e. with language, culture or locality bias.

“Interfaces are “first class objects”.

The interface between two programs is the set of assumptions that each makes about the other. Are your assumptions explicit, conscious?

Design it, document it, review it, prototype using it, revise it.

The interface document must consider all possibilities including things that should not happen.



Interface Documentation: 12 Element Queue

<u>Program Name</u>	<u>Value</u>	<u>Arg#1</u>
ADD		<integer>
REMOVE		
FRONT	<integer>	

Canonical representation:  $(\text{rep} = \langle [a_i]_{i=1}^n \rangle) \wedge (0 \leq n \leq 12)$  Initial:  $n = 0$ . (sequence of up to 12 elements)

ADD([rep],a)  $\equiv$

<u>conditions</u>	<u>new rep</u>	<u>extension class</u>
$n = 12$	rep	%full%
$n < 12$	rep.a	

REMOVE([rep])  $\equiv$

<u>conditions</u>	<u>new rep</u>	<u>extension class</u>
$\text{rep} = \_$	rep	%empty%
$\text{rep} \neq \_$	$\langle [a_i]_{i=2}^n \rangle$	

FRONT([rep])  $\equiv$

<u>conditions</u>	<u>new rep</u>	<u>extension class</u>	<u>Value returned</u>
$\text{rep} = \_$	rep	%empty%	
$\text{rep} \neq \_$	rep		$a_1$



## **Should Localisers be Testers?**

The bugs should be gone before you get it.

SQRL has testing and inspection procedures that should be applied before you start your work.

Programs are hierarchically decomposed.

Documentation allows programs to be inspected in small parts.

Documents can be tested by comparison with program behaviour.

Thorough inspection and testing will save a lot of time and frustration for the localisers.



## A Rational Software Design Process

1. Establish and document system requirements (black-box view).
2. Select hardware components and document the system design.
3. Document the required software behaviour.
4. Design and document the module structure.
5. Design and document the module interfaces.
6. Design and document the uses hierarchy.
7. Establish and document process structure guidelines.
8. For each module that is too big to throw away, repeat 4...8, then
  - Document the module's internal design.
  - Review the design (using the design documents).
  - Code in accordance with the design document.
  - Review and test the code using the design documents.

**Documentation is key to a better process.**

• *SOFTWARE QUALITY RESEARCH LABORATORY* •



## **What is Rational About This Process**

You determine what you need before you design/develop it.

After the first step, design decisions can be based on previous documents.

After the first step, decisions can be reviewed against earlier documents.

Each designer (document author) has information that is needed to make a rational decision.

This process is often denigrated as “the old waterfall model” but it is still a rational way to design software.



## **What is Different if the Software must be Localised?**

The requirements should be designed for the localisers, not the ultimate users.

Separate requirements should be drawn up for each locality.

These can be structured into commonalities and differences.

The exercise will pay off when the localisation extensions are written.



## **The Power of Parameterisation**

When designing software consider parameterising anything that might vary from version to version.

If the parameters can be set at compile time, there is little run-time penalty.

If the performance is not an issue, you get more flexibility by using run-time assignment of values to parameters.

This will allow a single piece of software to serve many users.

This will reduce the maintenance load on your company.

Even language can be (often is) a run-time parameter but speed, type of interface and many other things can be handled in this way.



## Summary

Localisation is usually hard because the software was badly designed.

Insist on the application of the “family principle”.

Insist on a modular structure with clean and documented interfaces.

Insist on a consciously designed uses hierarchy.\

YOU, as localiser, should write the requirements.

Don't forget the error handling. This can kill the best design, especially if some errors generate messages to the screen.

