Revisiting "Good" Software Design Principles To Shape Undone Computer Science Topics[1]

Florence Maraninchi
www-verimag.imag.fr/~maraninx
orcid.org/0000-0003-0783-9178

Verimag /Grenoble INP - UGA / Ensimag

Undone Computer Science Conference, Nantes, Feb 5th, 2024

[1] ALDIWO – This project has received financial support from the CNRS through the MITI interdisciplinary programs

Summary Of Previous Episodes: Anti-Limits in Digital Systems

COMMUNICATIONS OF THE ACM

HOME | CURRENT ISSUE | NEWS | BLOGS | OPINION | RESEARCH | PR

Home / Magazine Archive / September 2022 (Vol. 65, No. 9) / September 2022 (Vol. 65, No. 9) / Full Text

VIEWPOINT

Let Us Not Put All Our Eggs in One Basket

By Florence Maraninchi
Communications of the ACM, September 2022, Vol. 65 No. 9, Pages 35-37
10.1145/3528088
Comments

VIEW AS: SHARE:

Our colleagues at the Intergovernmental Panel on Climate Change (IPCC[a]) and the Intergovernmental Science-Policy Platform on Biodiversity and Ecosystem Services (IPBES[b]) have been telling us for years the situation is serious. Last year saw both the publication of the sixth IPPC report, and dramatic illustrations of the impacts of climate change. Researchers and teachers in all disciplines face the question: What can

An anti-limit is both a promise and a deliberate hypothesis that resources will grow as needed.

- Requires an increasing amount of resources globally (unlimited number of cryptocurrencies relying on proof-of-work, space, or bandwidth, ...)
- Promises immediate service delivery, whatever the number of clients and usages (most of the cloud services)
- Promises unlimited storage in both space and time (Gmail)
- Assumes availability of some hardware, software and vendor cloud forever (some home automation devices)
- Is designed to allow for unlimited functional extensions
- Bets on the availability of a more efficient machine, soon
- Needs more users or an increased usage per user to be profitable

This Paper

What's the responsibility of the *"design-for-extensibility" principle* in the fact that we do not know how to stay within limits?

This Talk

1. Is It Our Fault? Questioning Extensibility/Generalization
2. System Design vs SW Design for Open Systems. Case-Study: Memory Hierarchy
3. Scenarios For Future Uses, Open/Closed Systems, Extensible/Shrinkable SW
4. This Is Not A Conclusion

What Do We Teach (Implicitly)?

When confronted with a student's solution to a basic algorithm/programming exercice, what kind of implicit assumptions do we have in mind?

— General is better than purely *ad hoc*
— Extensible is better than purely *ad hoc*
— No Redundancy is better (although, in some cases...)
— ...

Generality (and Spurious Generalizations): Example

Question: compute the max of a non-empty array of positive integers
Candidate generalizations:

[2] http://www.cs.yale.edu/homes/perlis-alan/quotes.html

Generality (and Spurious Generalizations): Example

Question: compute the max of a non-empty array of positive integers
Candidate generalizations:
— A possibly-empty array of general integers
— A possibly-empty array of TypeX values, use a max-like function on TypeX values
— What if we also need the min of the array?

[2] http://www.cs.yale.edu/homes/perlis-alan/quotes.html

Generality (and Spurious Generalizations): Example

Question: compute the max of a non-empty array of positive integers
Candidate generalizations:
— A possibly-empty array of general integers
— A possibly-empty array of TypeX values, use a max-like function on TypeX values
— What if we also need the min of the array?

\+ a more general component/function is more reusable.
\+ it covers more cases, so less misuse errors (but we could use defensive code)
− it often involves additional initial complexity. How much should we accept?

[2] http://www.cs.yale.edu/homes/perlis-alan/quotes.html

Generality (and Spurious Generalizations): Example

Question: compute the max of a non-empty array of positive integers
Candidate generalizations:
— A possibly-empty array of general integers
— A possibly-empty array of TypeX values, use a max-like function on TypeX values
— What if we also need the min of the array?

+ a more general component/function is more reusable.
+ it covers more cases, so less misuse errors (but we could use defensive code)
− it often involves additional initial complexity. How much should we accept?

Epigrams in programming (Alan Perlis)[2]: In programming, everything we do is a special case of something more general – and often we know it too quickly.

[2] http://www.cs.yale.edu/homes/perlis-alan/quotes.html

When asked to write a function that returns the max of a non-empty array of positive integers, the average student will often deliver a more general version.

Probably because we expect them to do so!

The Butcher's Approach To Writing Software

When asked to write a function that returns the max of a non-empty array of positive integers, the average student will often deliver a more general version.

Probably because we expect them to do so!

Two Definitions Found On The Web

Extensibility is a software engineering and systems design principle that provides for future growth. Extensibility is a measure of the ability to extend a system and the level of effort required to implement the extension[3].

[3] https://en.wikipedia.org/wiki/Extensibility

[4] https://www.codium.ai/glossary/software-extensibility/

Two Definitions Found On The Web

Extensibility is a software engineering and systems design principle that provides for future growth. Extensibility is a measure of the ability to extend a system and the level of effort required to implement the extension[3].

Software extensibility encapsulates the software's innate ability to absorb fresh features, capabilities, or alterations, all without requiring an extensive reconstruction of its core architecture. Think of this as building with a "future-proof" mindset (...)[4].

[3] https://en.wikipedia.org/wiki/Extensibility

[4] https://www.codium.ai/glossary/software-extensibility/

Two Definitions Found On The Web

Extensibility is a software engineering and systems design principle that provides for future growth. Extensibility is a measure of the ability to extend a system and the level of effort required to implement the extension[3].

Software extensibility encapsulates the software's innate ability to absorb fresh features, capabilities, or alterations, all without requiring an extensive reconstruction of its core architecture. Think of this as building with a "future-proof" mindset (...)[4].

This is always considered a desirable property. But for whom? And how much should a piece of SW be extensible? What futures do we have in mind? What amount of added initial complexity should we accept?

[3] https://en.wikipedia.org/wiki/Extensibility

[4] https://www.codium.ai/glossary/software-extensibility/

You design a car with the initial *driving-on-roads* function

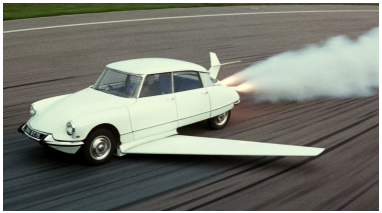
You design a car with the initial *driving-on-roads* function

But you make it extensible enough so it won't be too difficult to add the "flying" function later

The Fantomas Approach to Writing Software[5]

You design a car with the initial *driving-on-roads* function

But you make it extensible enough so it won't be too difficult to add the "flying" function later



[5] https://fr.wikipedia.org/wiki/Fantomas_(film,_1964)

What Kind Of Lifecycle Do We Have In Mind?

More general / extensible... ok but for what purposes? how much? and what amount of added initial complexity can we afford?

What hypotheses do we make – implicitly – on the future of the system?

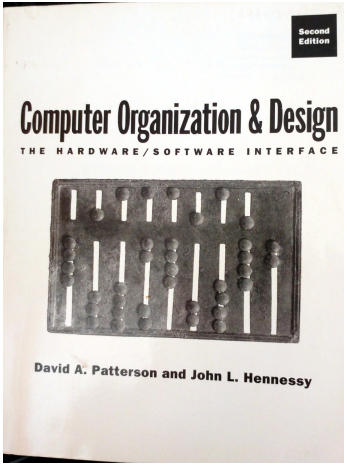
Do we prefer general/extensible programs because we really expect them to grow, or just because we think they are more elegant?

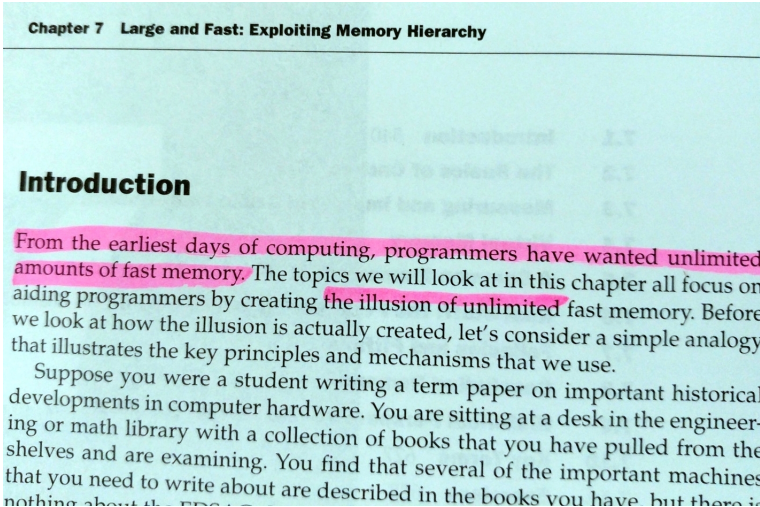
Towards a System View: Norms for Software In Civil Avionics

Traceability in the DO178B:
— Each system-level requirement should be associated with some lines of object code
— Each line of object code should be traceable to some system-level requirement

Notions of:
— Dead code: will never be executed (and is not always removed by compilers)
— Deactivated code: code that could be executed in another (static) configuration, or for a **planned extension**.
while extensibility is preparing for unplanned extensions.

Creating The Illusion of Unlimited Fast Memory





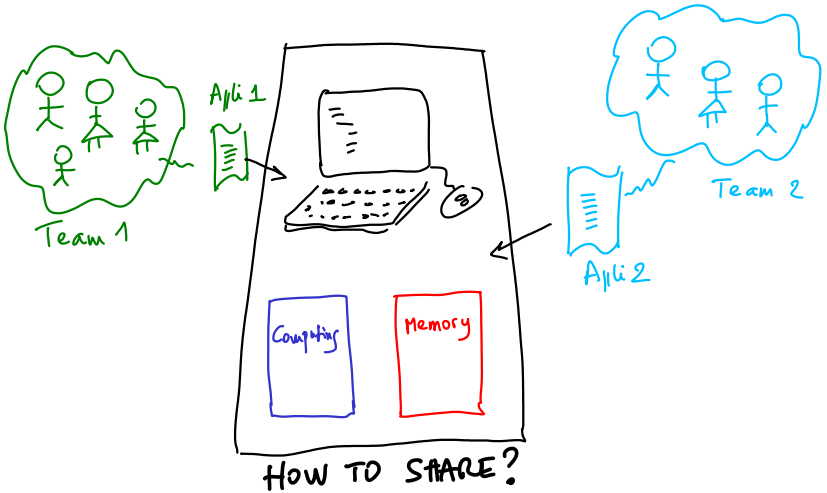
Chapter 7 Large and Fast: Exploiting Memory Hierarchy

Introduction

From the earliest days of computing, programmers have wanted unlimited amounts of fast memory. The topics we will look at in this chapter all focus on aiding programmers by creating the illusion of unlimited fast memory. Before we look at how the illusion is actually created, let's consider a simple analogy that illustrates the key principles and mechanisms that we use.

Suppose you were a student writing a term paper on important historical developments in computer hardware. You are sitting at a desk in the engineering or math library with a collection of books that you have pulled from the shelves and are examining. You find that several of the important machines that you need to write about are described in the books you have, but there is

Principles for Resource Sharing

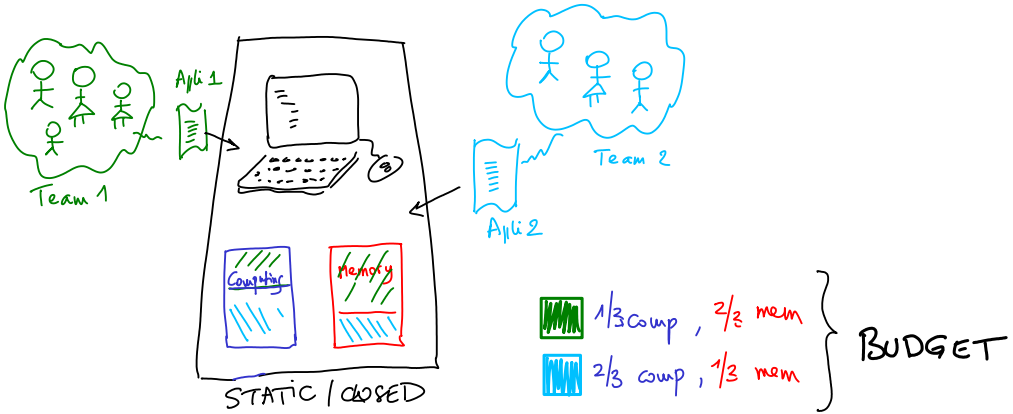


Is There a System Integrator?

- If we know in advance all the applications to be developed and run on a given machine, and also their precise needs in memory and computing power, a system design approach can be used, establishing the **budget** of each of them. Not extensible but simple.

Can be the appropriate approach for critical systems
(*You have to avoid "sorry, no more memory!" in the middle of a flight, anyway*).

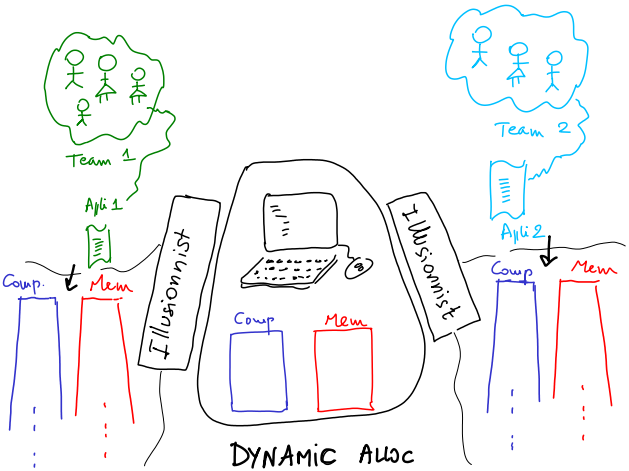
System View, Static Allocation, Closed System



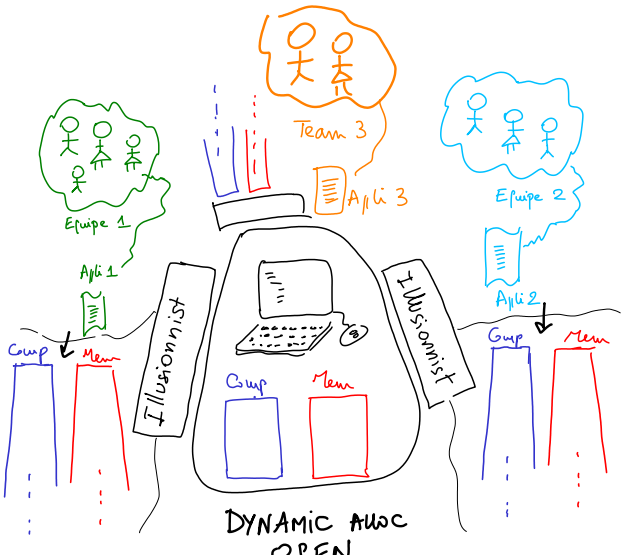
No System View

- If we know in advance all the applications to be developed and run on a given machine, and also all precise their needs in memory and computing power, a system design approach can be used, establishings the budget of each. Not extensible, simple.
- If we do not know all the applications in advance, or we want to add new ones later, or their computing and memory needs are not known precisely, SW designers should be allowed to work anyway, ignoring the needs and constraints imposed by others.

No System View, Dynamic Allocation, Open Systems



No System View, Dynamic Allocation, Open Systems



No System View

When the limits are reached anyway, what can you do?
None of the actors has sufficient control over the system.

The end-user has no choice but to buy a new smartphone, or augment the memory of their computer (if possible).

Open/Extensible Systems Are Meant For Growth Scenarios

This is clear in the definition of general/extensible systems

But we observe quite counter-intuitive effects (in digital systems):

- There are perfectly ad-hoc digital systems that have been running unchanged for more than 30 years (examples in nuclear power-plants)
- But the most versatile HW/SW object ever (the smartphone) has to be replaced every 2-5 years

Open/Extensible Systems Are Meant For Growth Scenarios

This is clear in the definition of general/extensible systems

But we observe quite counter-intuitive effects (in digital systems):

- There are perfectly ad-hoc digital systems that have been running unchanged for more than 30 years (examples in nuclear power-plants)
- But the most versatile HW/SW object ever (the smartphone) has to be replaced every 2-5 years

About "technological" objects in general:

- SW is expected to make things extensible/reusable/... but 30 years is already considered very long!
- 300-year-old Stradivarius violins are still usable

To Stay Within Limits We Need Closed/Shrinkable Systems

Closed (Ad-Hoc) Systems:
We could design digital systems from early precise specifications and a few *planned extensions*, not for unexpected extensions.

We could design a smartphone like a washing machine (in which there are SW components, but for well-defined functions).

To Stay Within Limits We Need Closed/Shrinkable Systems

Closed (Ad-Hoc) Systems:
We could design digital systems from early precise specifications and a few *planned extensions*, not for unexpected extensions.

We could design a smartphone like a washing machine (in which there are SW components, but for well-defined functions).

Shrinkable: If you do not know what the future will be, plan for smaller (instead of bigger) systems. Hence design for shrinkability rather than extensibility.

Tentative Definitions of Shrinkability

Shrinkability is a software engineering and systems design principle that provides for future degrowth. It is a measure of the ability to reduce or reconfigure the functionalities if the resources available decrease, and the level of effort required to implement this functional reduction.

Tentative Definitions of Shrinkability

Shrinkability is a software engineering and systems design principle that provides for future degrowth. It is a measure of the ability to reduce or reconfigure the functionalities if the resources available decrease, and the level of effort required to implement this functional reduction.

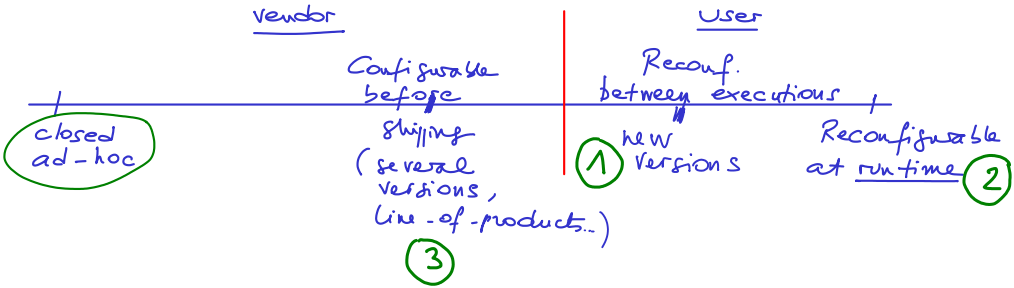
Software shrinkability encapsulates the software's innate ability to be simplified by removing features or capabilities, all without requiring an extensive reconstruction of its core architecture. Think of this as building with a "future-proof" mindset.

Reviewers's Questions and Suggestions

1. Optimizations aimed at getting rid of the fluff
2. Multi-mode SW, depending on resources available (dynamic switching?)
3. Dynamic/static Shrinkage (ex. in small OSes)
4. *How to make this shrinking evolution desirable/sexy for developers and clients?*

Reviewers's Questions and Suggestions

1. Optimizations aimed at getting rid of the fluff
2. Multi-mode SW, depending on resources available (dynamic switching?)
3. Dynamic/static Shrinkage (ex. in small OSes)
4. *How to make this shrinking evolution desirable/sexy for developers and clients?*



Degrowth Scenarios and The Beauty of Ad-Hoc Systems

- Decide beforehand on the use scenario for your system; why could'nt it be a degrowth scenario?
- Invent (and teach) shrinkability properties to replace extensibility

Degrowth Scenarios and The Beauty of Ad-Hoc Systems

- Decide beforehand on the use scenario for your system; why could'nt it be a degrowth scenario?
- Invent (and teach) shrinkability properties to replace extensibility
- Change the way we think about HW/SW objects: Preserving existing objects, or designing new ones for centuries, should be considered as noble as maintaining a Stradivarius (instead of mass-producing plastic copies!)

Min-Jin Kym plays her antique Stradivarius violin. Photograph: National News and Pictures

The End. Thank you.
Questions ?

www-verimag.imag.fr/~maraninx