IT University of Copenhagen

Software Architecture Reconstruction

III: Evolutionary Analysis

Mircea Lungu (mlun@itu.dk)

github.com/mircealungu/reconstruction

Figure 1: 400

> No man ever steps in the same river twice, for it is not the same river and he is not the same man.
>
> – Heraclitus

Why *Must* Software Systems Evolve?

> An e-type program that is used in a real-world environment must change, or become progressively less useful in that environment. – M. Lehman, *The Law of Continuing Change*

What might be referring to when he talks about an e-type system?

In the terminology of Lehman, "e" stands for embedded. - an e-type system is *embedded* in the real world - and since the world changes, so the system must change

The *Real-world Context* Changes

To think about: *Do you have good examples of systems that had to change because the real world changed?* Some examples: - the software that computes taxes in Denmark - more?

To think about: *What other kinds of systems are there then? Can you think about another type?* Are there programs that are not impacted by the change in the world around them?

It seems that one could even argue that this is the difference between algorithms and software systems: algorithms don't have to change with the world, while systems have to.

The Technical Ecosystem Changes

Think about the `npm` ecosystem. Every day you can execute `npm audit` in your React-based web application to find out that a dozen of the packages you depend on have new versions. Do you upgrade? Do you stay like this for a while? What's the best strategy for managing this portfolio of dependencies?

But this is not only about the `npm` ecosystem. The same situations emerges when a developer builds a system on top of a programming language. Does their code still depend on Python 2.7? Some of the libraries their code depends on, might have dropped support for that version of Python. So they either upgrade, or they are left behind. If a developer des not want to be left behind, they have to keep up with all their *upstream* dependencies.

Developing software is a little bit like living in Wonderland where the Red Queen is telling Alice: "*In this place you have to run to stay in one place*".

Software Evolution

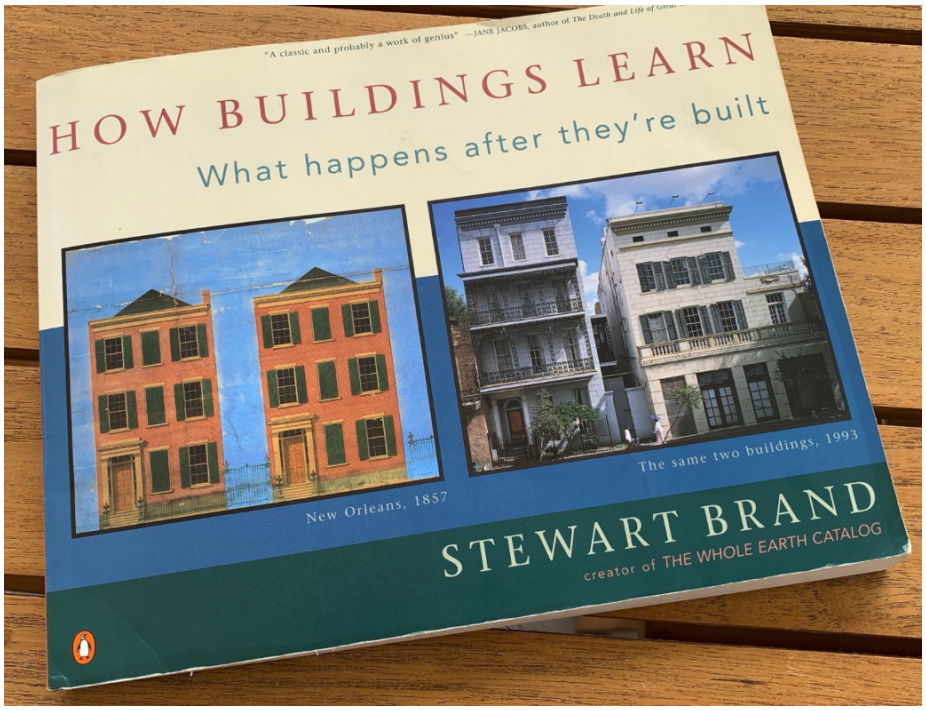
Software evolution is the continual development of a piece of software after its initial release to address changing stakeholder requirements.

Observations: - It used to be called *software maintenance* - Nowadays evolution is the preferred term because it highlights the fact that a software system is never *finished*

Does Architecture Evolve?

From this POV, the **architecture metaphor might not be the best** - because we normally think about architecture as unchanging.

This is where Stewart Brand – one of my favorite non-fiction authors – comes to help. He wrote a whole book about how buildings evolve over time exactly with the goal of making it clear that architecture is also changing.



I personally think a ***garden*** **would have been a better metaphor** for a software system. You must constantly tend to your garden if you want to maintain it.

Version Repositories Can Help Re-trace Evolution

One of the benefits of the widespread adoption of tools like git is the fact that the meta-data captured in the version control system captures relevant information about the evolution of the system.

Types of Evolutionary Information Relevant for AR available in VCS

Code Ownership: which developers are responsible with which parts of the code?

This can help find an expert for a given problem.

One example: Git-Truck

```
git clone git@github.com:zeeguu/api.git
npx -y git-truck-beta@latest
```

Why is it named git-truck?

The truck-factor To think about - Is a truck factor good if high or if low? - How could you devise a team-strategy or company strategy to improve it?

Logical Coupling: the parts of the system that always change together

This is called logical coupling. When two entities always change together, even if there is no explicit dependency between them, this information can be inferred from the version control.

Temporally-distributed documentation

One of the beautiful insights I recently had is the importance of the git messages associated to commits as documentation.

Indeed, even if there is no separate documentation, well described commits can serve as an evolving documentation for a software system.

Evolutionary Hotspots: the parts of the system have been most changed over time

> *"The value of anything is proportional to time invested in it."* (M. Lungu)

Process Metric: Code Churn

= a metric that indicates how often a given piece of code—e.g., a file, a class, a function—gets edited.

- process metric (*as opposed to?*)
- can suggest relevance for the architecture (*in wjhich way?*)
- can be detected with **language independent analysis** (which is good for polyglot systems)

Why would places in the system with high-code churn be relevant?

Practically:

- places in the code with high code churn are likely to be most important parts of the code
- studies observe correlation between *code churn* and complexity metrics
- high *code churn* predicts bugs better than size
- it's likely that they'll require more effort in the future (e.g. yesterday's weather [Girba et al.])

Evolutionary Hotspots Viewpoint

Evolutionary Hotspots – **an architectural viewpoint that highlights those code entities where most commits are made**

Notebook: Computing Evolutionary Hotspots with PyDriller

Challenges when computing an Evolutionary Hotspots viewpoint: - Taking into account developer styles - the micro-commits developer vs. the large chunk commiter - Removing irellevant files that change frequently (`README.md`, or `LICENSE.md`) - Combine with static complexity metrics - Manual investigation - Selecting the appropriate time-interval for the analysis - Weighting towards recency (discarding past changes more) - Tracking file renames over the course of a system's history - Sometimes git loses track of file history: e.g. if you rename and make changes at the same time