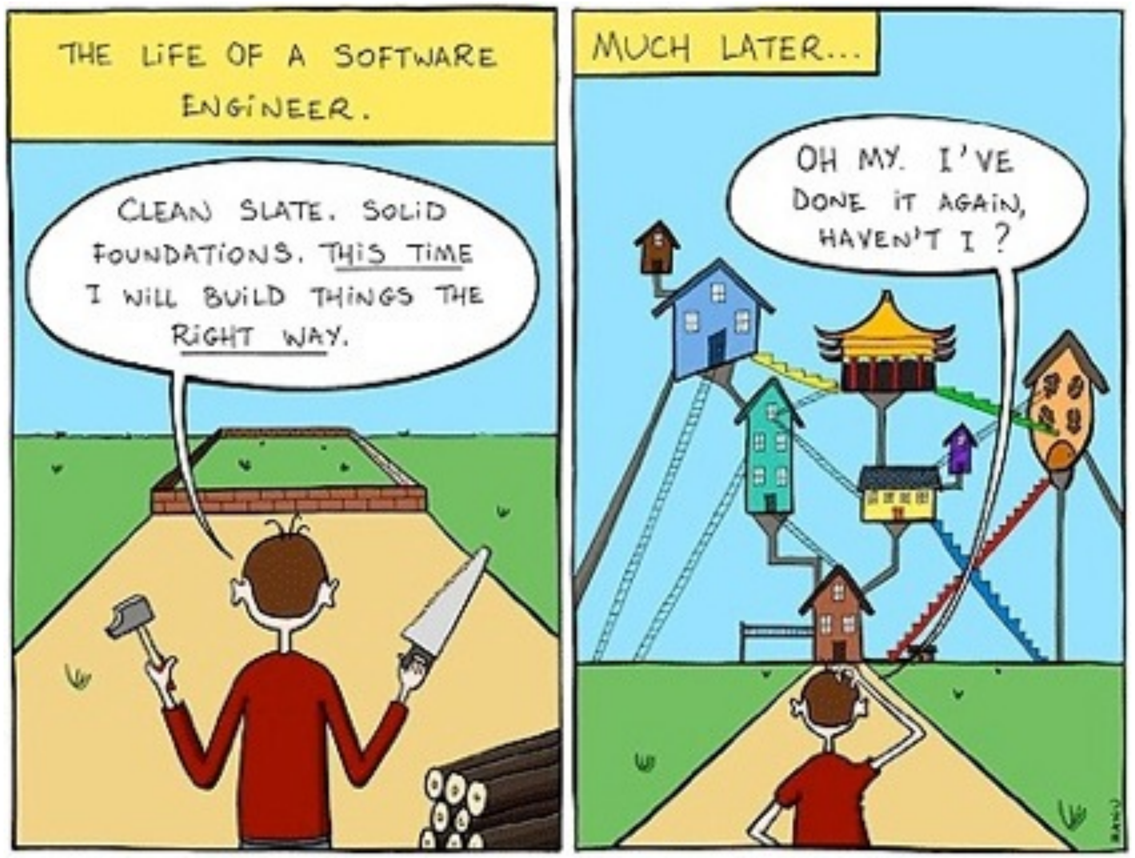
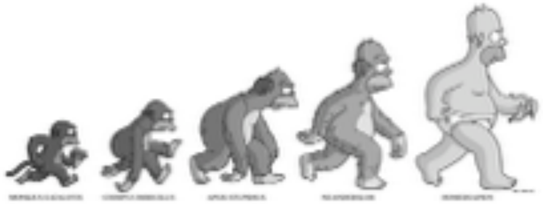

THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?

Introduction to Software Engineering

11. Software Evolution

Based on a lecture by Oscar Nierstrasz and the **SDE Course** at the University of Bern.



Laws of Software Evolution

Reverse and Reengineering

Mining Software Evolution

Laws of Software Evolution

Reverse and Reengineering

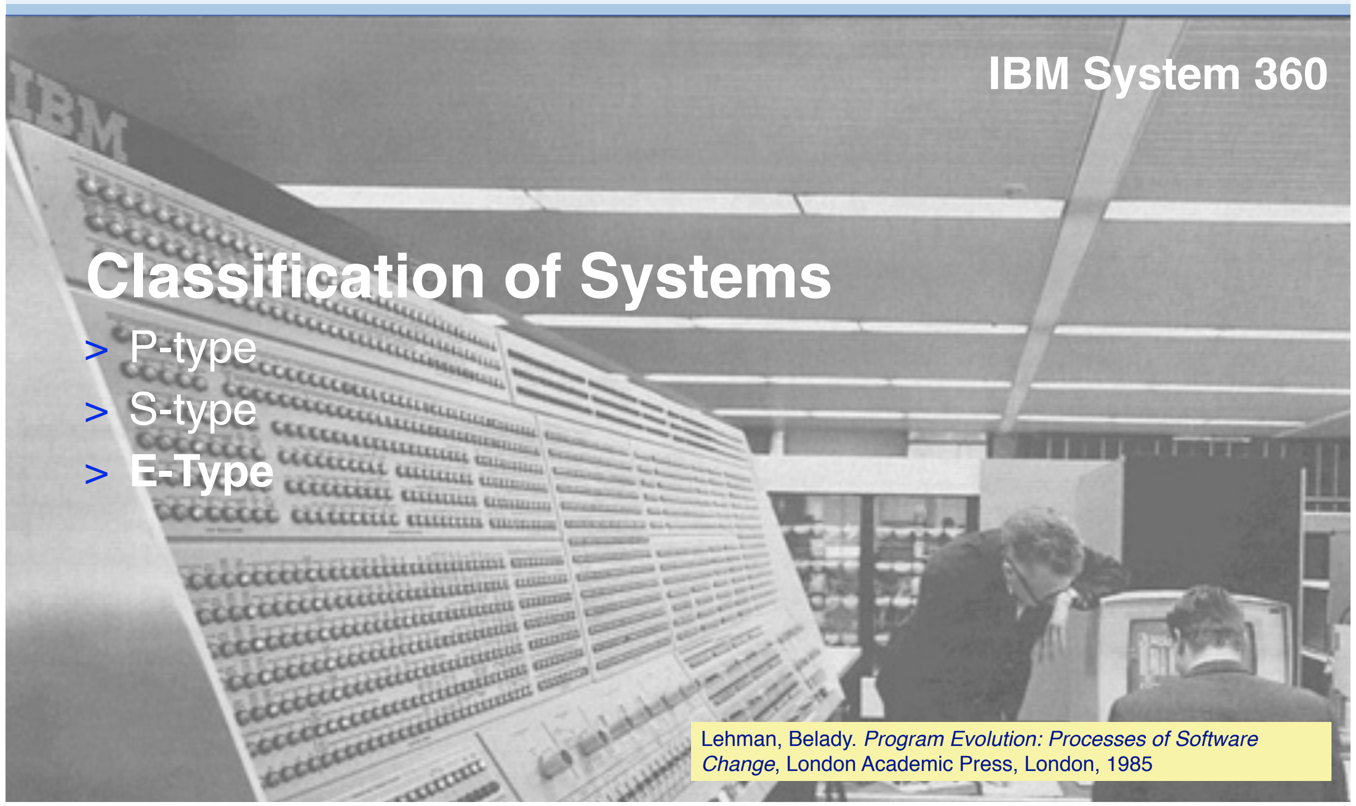
Mining Software Evolution

Lehman's Law of Continuing Change

—A program that is used in a real-world environment *must change*, or become progressively less useful in that environment.

Lehman, Belady. *Program Evolution: Processes of Software Change*, London Academic Press, London, 1985

Lehman’s Laws of Software Evolution



IBM System 360

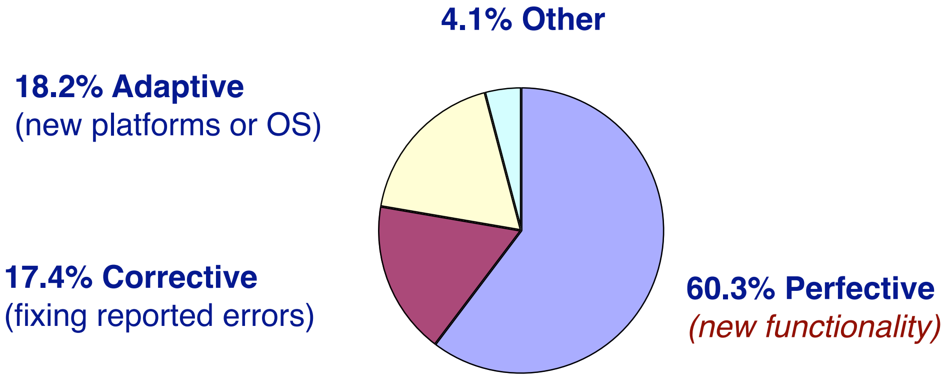
Classification of Systems

- P-type
- S-type
- **E-Type**

Lehman, Belady. *Program Evolution: Processes of Software Change*, London Academic Press, London, 1985

S–type — specification based
P–type — algorithms. chess playing system.
E–type — real world activity. integrated in the environment.

Continuous Development



The bulk of the maintenance/evolution cost is due to *new functionality*
⇒ even with better requirements, it is hard to predict new functions

data from [Lien78a]

Well, better requirements engineering indeed helped to identify stable ground in the muddle of CHANGING REQUIREMENTS.

Unfortunately, an empirical survey done by Lientz and Swanson revealed that the majority of changes requested have to do with EXTRA FUNCTIONALITY.

That’s why modern requirements engineering tries to define SCENARIOs FOR FUTURE EXTENSIONS so that the designers can accommodate their designs. That’s why we try to define good DOMAIN MODELS, so that most of the changes will fit our design.

Unfortunately, no matter how good our requirements engineering, we will never be able TO PREDICT THE FUTURE

Therefore, there will always be changes that DO NOT FIT OUR ORIGINAL DESIGN, and depending on our problem domain this may be quite a lot.

By the way, that’s one of the reasons why AGILE DEVELOPMENT METHODS are so popular nowadays. They observed that for certain problem domains you cannot make good predictions about the 60% perfective maintenance here (POINT TO SLIDE). When that’s the case, its useless to invest in a good design because you don’t know which variation points to build into your design. Half of the times, you will guess wrong and then you

Lehman's Law of Increasing Entropy

—As a program evolves, it becomes *more complex*, and extra resources are needed to preserve and simplify its structure.

Lehman, Belady. *Program Evolution: Processes of Software Change*, London Academic Press, London, 1985

Lehman’s Law of …

this slide is intentionally left blank.

Software Ageing = Increasing Entropy

Lack of Knowledge

- > *obsolete* or no documentation
- > *departure* of the original developers or users
- > *limited understanding* of entire system (*missing tests?)*

How can you ASSESS WHETHER A SYSTEM NEEDS REENGINEERING (i.e., when is it time to take the second path ?)

In fact, there are QUITE A LOT OF SYMPTOMS as you can see on this slide

I won't go into detail for each of them,
RATHER, I will point you to the ones I ASK FOR EACH TIME I FACE A NEW SYSTEM
⇒missing tests, simple changes take too long. big build times

The reason I looked for those is because they are MEASURABLE.
This is especially important, because during a reengineering project, there will ALWAYS COME A TIME WHEN YOU WILL BE CHALLENGED. Opponents will try to cancel you project and start again from scratch.
Therefore, its is good to have some QUANTIFYABLE GOALS, to show that you are making progress, EVEN IF THE REENGINEERING IS NOT FINISHED

Common Symptoms of Software Aging (2)

Process Failures

> *too long* to turn things over to production

> *constant bug fixes*

> *simple changes take too long*

Code symptoms

- *duplicated* code
- *code smells*

Common Problems

Architectural Problems

- insufficient *documentation*
  = non-existent or out-of-date
- improper *layering*
  = too few or too many layers
- lack of *modularity*
  = strong coupling
- *duplicated code*
  = copy, paste & edit code
- duplicated *functionality*
  = by separate teams

Refactoring opportunities

- *misuse* of inheritance
  = code reuse vs polymorphism
- *missing* inheritance
  = duplication, case-statements
- *misplaced* operations
  = operations outside classes

We have been facing quite a lot of systems that NEEDED REENGINEERING.
We learned that all of them have different MOTIVATIONS,
however there seems to be a COMMON SET OF PROBLEMS THAT REAPPEAR IN ALL projects
... READ SLIDE ...

So what to do?

Before:

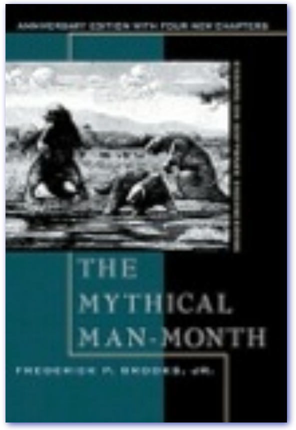
Design for change. Build a family of solutions

If software is bound to change, plan for change. Don’t build a solution, build a family of solutions.
Open-Closed Principle.

So what to do?

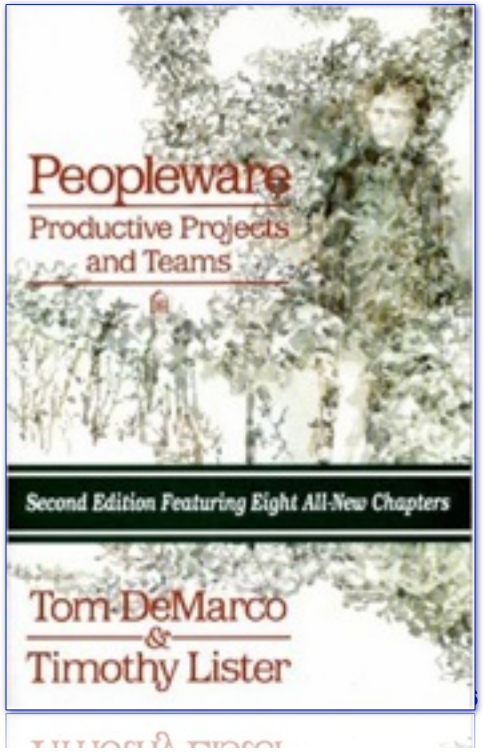
After:
  Reengineer.
  Refactor.

Never try to rewrite the system. There is too much knowledge encoded in the running system.



People Matter: *Peopleware*

- Project management
- Work environment
- The concept of “flow”



[Bonus Topic] The Concept of *Flow*

Mihaly Csikszentmihalyi: Flow, the secret to happines

“You know that what you need to do is possible to do, even though difficult, and sense of time disappears. You forget yourself. You feel part of something larger.”

Mihaly Csikszentmihalyi on experiencing ‘flow’

Night on Earth, Jim Jarmush

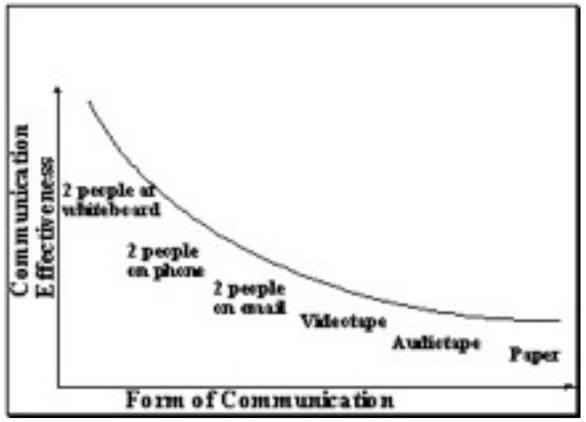
Anecdotal evidence of distance-communication still working.

Image from “Night on Earth”. Jim Jarmush.
Some software shops work around the clock.
But is this efficient?

Laws about people

Characterizing people as non-linear, first-order components in software development, **Alistair Cockburn**

- — Communication degradation
- — Inconsistency of people
- — Good citizenship
- — Diversity of people

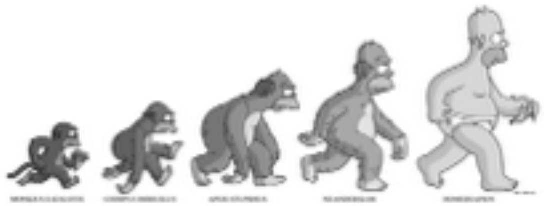


Good citizenship. GitHub.
Inconsistency. People need external factors.
People like proximity.

Gamification: “People like to see progress”

Erwann Wernli @wrnli
It feels great to move post-it! pic.twitter.com/0dtxmVIOMP

On Gamification.



Laws of Software Evolution

Reverse and Reengineering

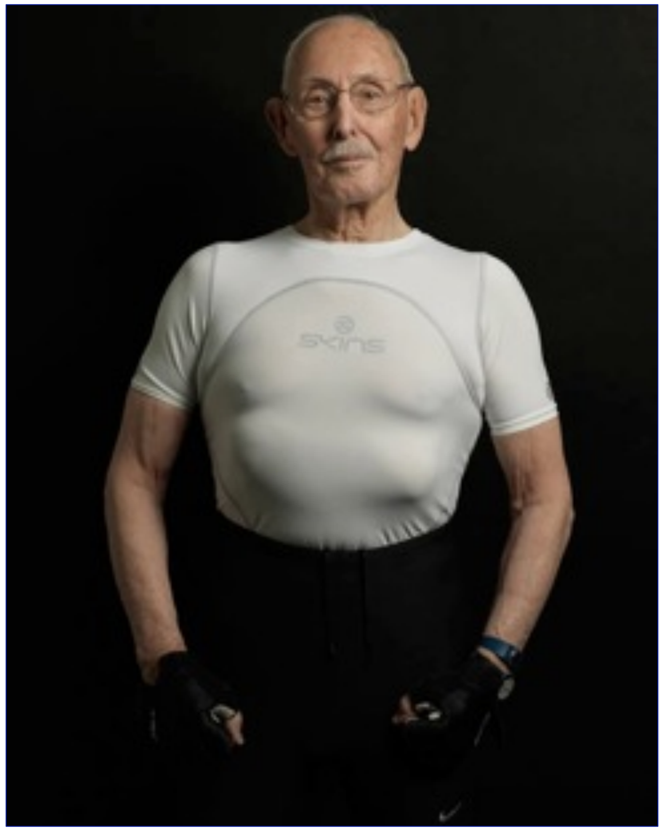
Mining Software Evolution

Reengineering ... is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.

"Forward Engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system."

"Reverse Engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction."

— Chikofsky and Cross [in Arnold, 1993]



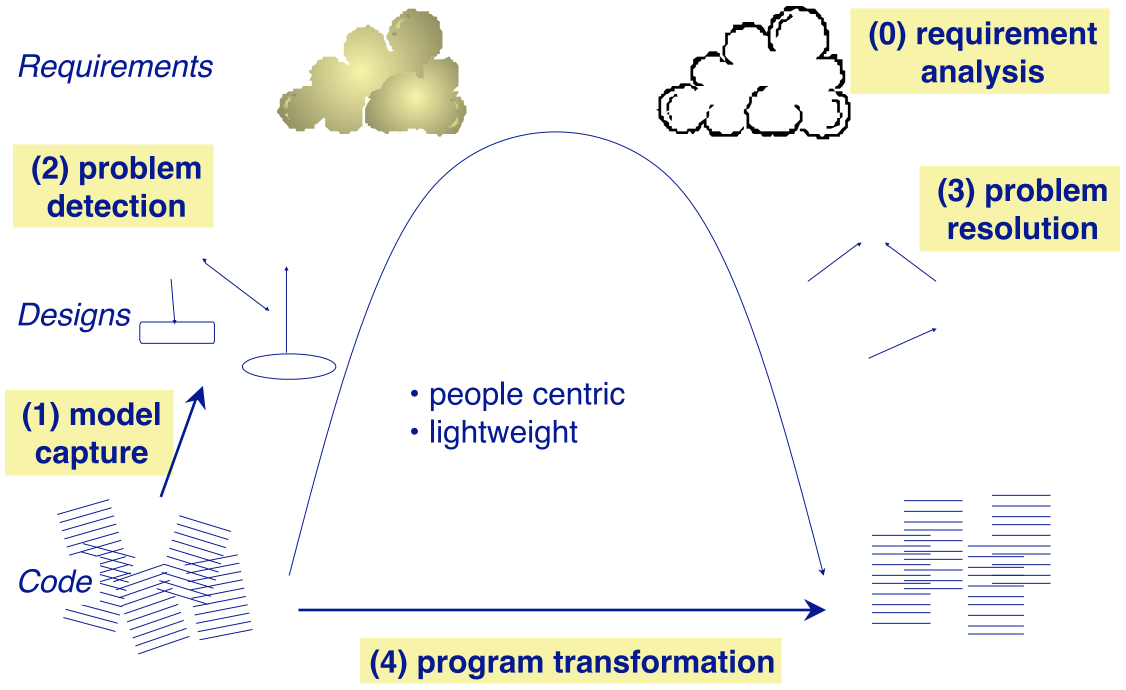
Charles Eugster @ TEDxZurich

Born in 1919. Still goes to the gym.
How to get this?
With explicit effort.

Goals of Reengineering

> *Untangling*
>
> —split a monolithic system into parts that can be separately marketed
>
> —increase the understandability of the code
>
> *Performance*
>
> —"first do it, then do it right, then do it fast" — experience shows this is the right sequence!
>
> *Porting*
>
> —the architecture must distinguish the platform dependent modules
>
> *Design extraction*
>
> —to improve maintainability, portability, etc.
>
> Exploitation of *New Technology*
>
> —i.e., new language features, standards, libraries, etc.

The Reengineering Life-Cycle



To tackle these problems, you need some kind of GENERIC REENGINEERING PROCESS

Here is the one that we propose, WHICH WE WILL USE TROUGHOUT THIS TALK

Note that you should see this as a way to describe the various activities that take place during a project, but not necessarily a STRICT ORDER ON when these activities must take place.

(0) Requirement analysis: analyse on WHICH PARTS OF YOUR REQUIREMENTS HAVE CHANGED

(1)Model capture: REVERSE ENGINEER from the source-code into a MORE ABSTRACT FORM, typically some form of a design model. How abstract depends on the kind of problem you want to solve

(2)problem detection: IDENTIFY DESIGN PROBLEMS in that abstract model

(3)problem resolution: PROPOSE AN ALTERNATIVE DESIGN that will solve the identified problem

(4)program transformations: MAKE THE NECESSARY CHANGES TO THE CODE, so that it adheres to the new design YET PRESERVES ALL THE REQUIED FUNCTIONALITY
Here TESTING will play an important role

In the REMAINDER OF THE TALK, we will use this picture to ILLUSTRATE WHERE the various techniques and tools FIT IN.

While doing that, we will emphasize the role of THE HUMAN IN THE LOOP, because we believe that reengineering

Goals of Reverse Engineering

- Facilitate *reuse*
  - detect candidate reusable artifacts and components
- Generate *alternative views*
  - automatically generate different ways to view systems
- Synthesize *higher abstractions*
  - identify latent abstractions in software

- Cope with *complexity*
  - need techniques to understand large, complex systems
- Recover *lost information*
  - extract what changes have been made and why
- Detect *side effects*
  - help understand ramifications of changes

— Chikofsky and Cross [in Arnold, 1993]

Reverse Engineering Techniques

> *Re-documentation*
—diagram generators
—cross-reference listing generators

> *Design recovery*
—software metrics
—browsers, visualization tools
—static analyzers
—dynamic (trace) analyzers

CS - they pay good money for a decent cross-referencer

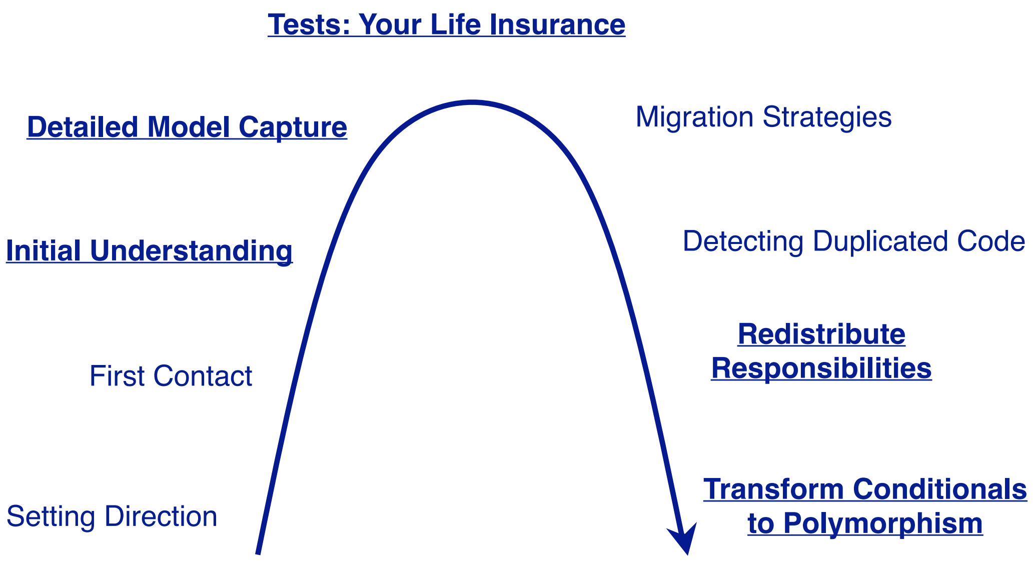
Reengineering Patterns

Reverse engineering patterns *encode expertise and trade-offs* in

— *extracting design* from source code, running systems and people.

—*transforming legacy code* to resolve problems that have emerged.

A Map of Reengineering Patterns



Tests: Your Life Insurance

Detailed Model Capture

Initial Understanding

First Contact

Setting Direction

Migration Strategies

Detecting Duplicated Code

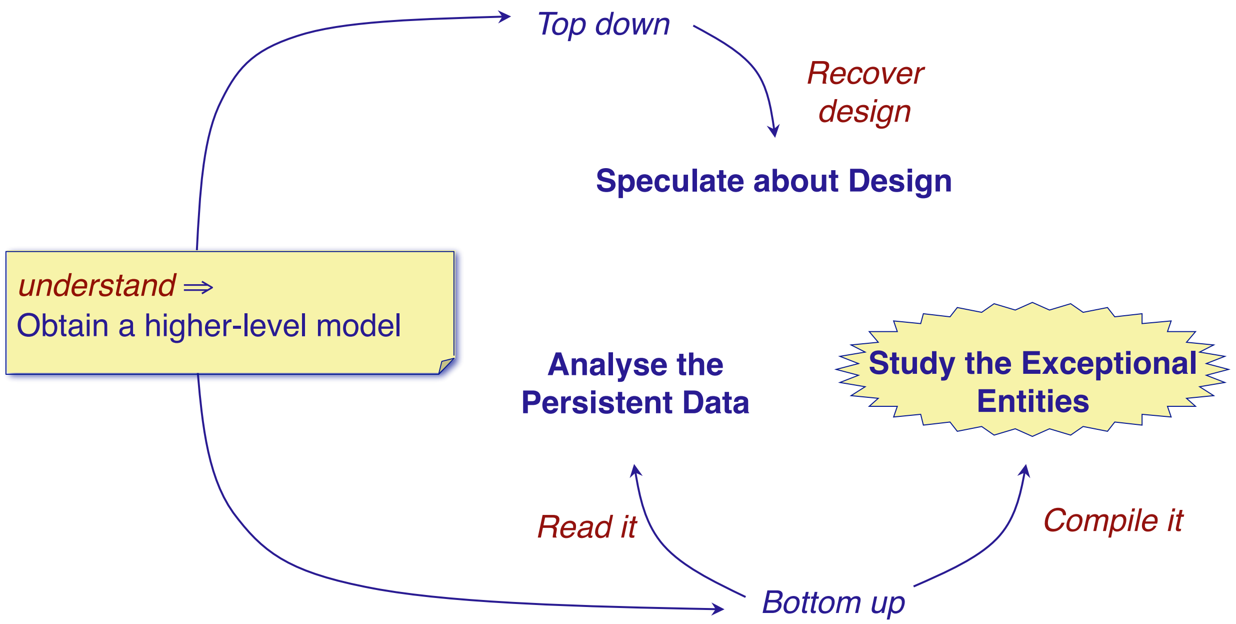
Redistribute Responsibilities

Transform Conditionals to Polymorphism

We documented most of our techniques in the form of REENGINEERING PATTERNS

Here is a map of these patterns, as they appeared in ou book

Initial Understanding



understand ⇒
Obtain a higher-level model

Top down

Recover design

Speculate about Design

Analyse the Persistent Data

Study the Exceptional Entities

Read it

Compile it

Bottom up

Pattern: Study the Exceptional Entities

Problem

—**How can you quickly gain insight into complex software?**

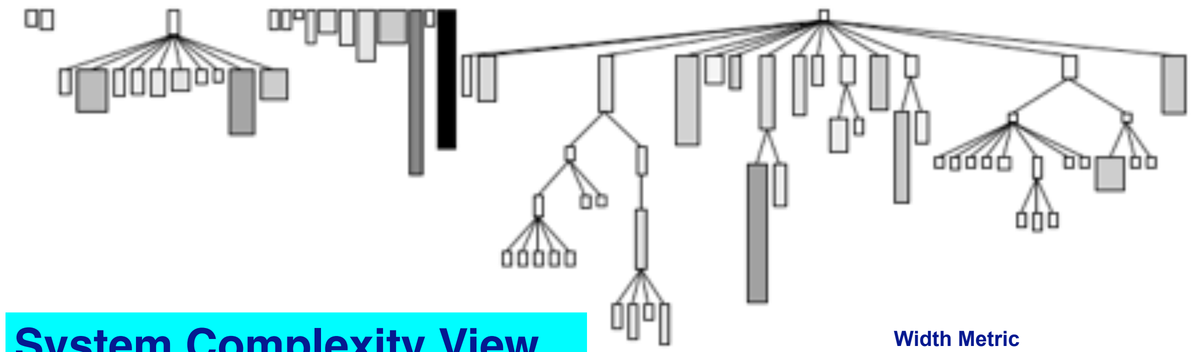
Solution

—*Measure* software entities and *study the anomalous ones*

Steps

- —Use simple metrics
- —Visualize metrics to get an overview
- —Browse the code to get insight into the anomalies

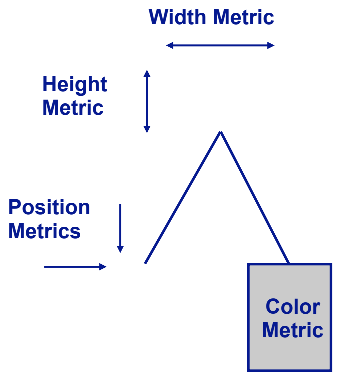
System Complexity View



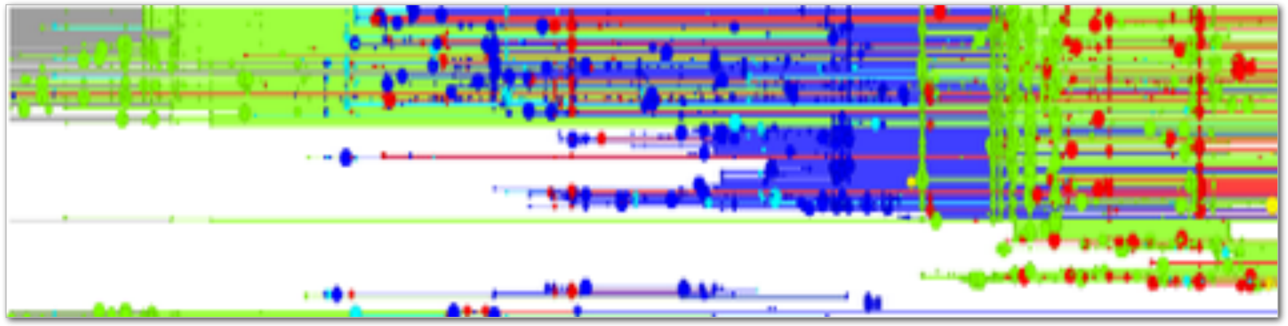
System Complexity View

Nodes = Classes
Edges = Inheritance Relationships

Width = Number of Attributes
Height = Number of Methods
Color = Number of Lines of Code

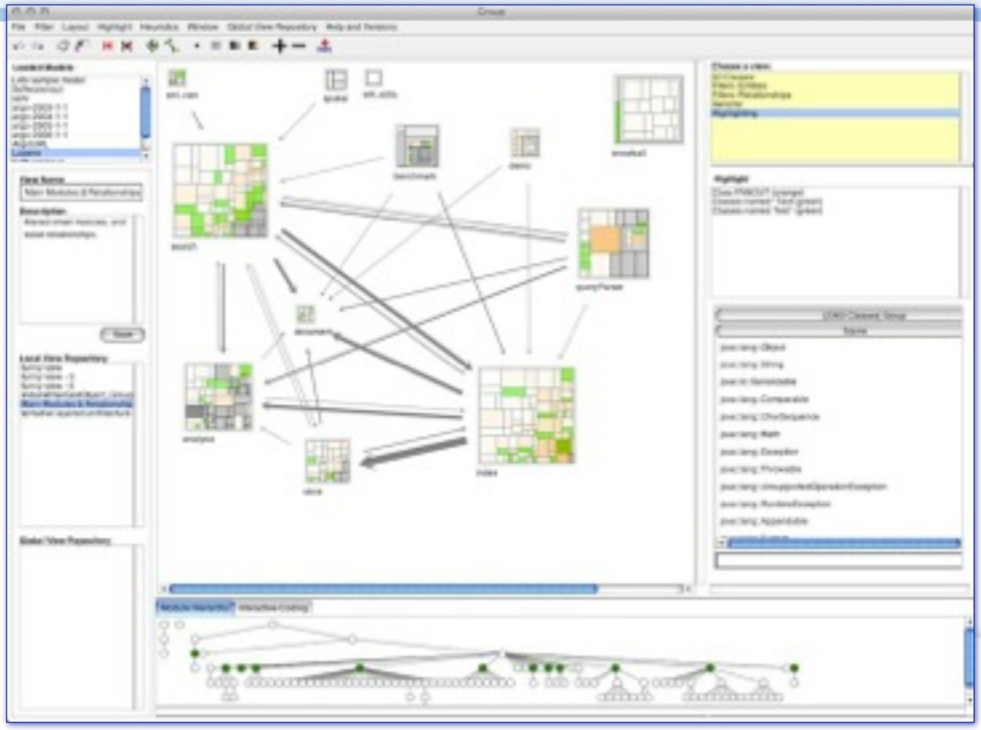


Code Ownership View



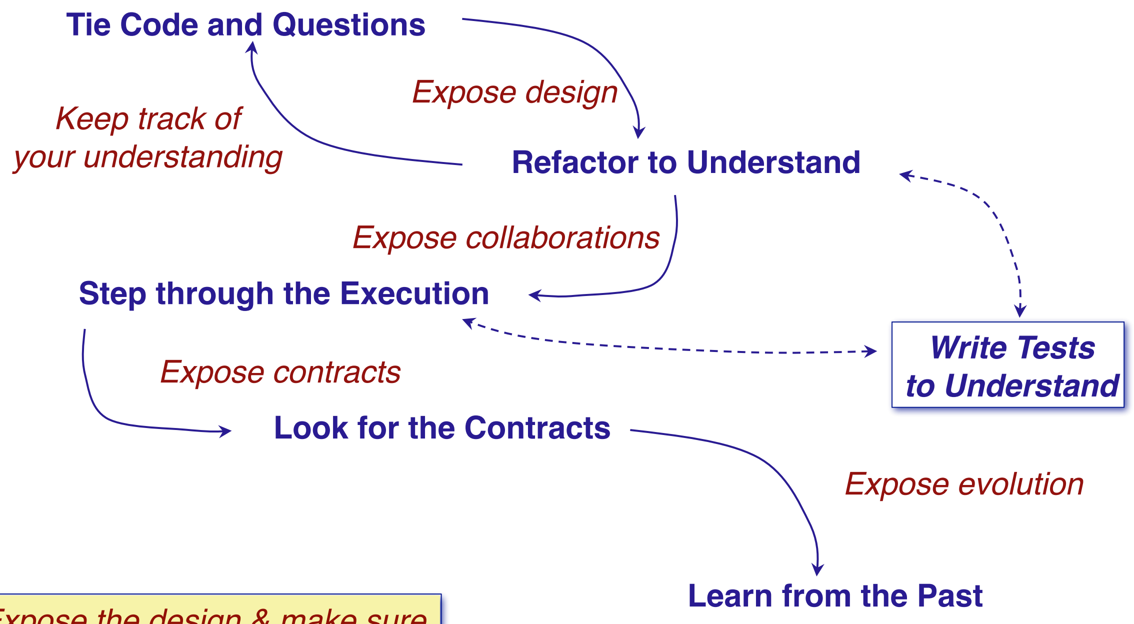
System complexity - Clone evolution view
Class blueprint - Topic Correlation Matrix - Distribution Map for topics spread over classes in packages
Hierarchy Evolution view - Ownership Map

High-Level Dependency View



http://scg.unibe.ch/softwarenaut

Detailed Model Capture



Tie Code and Questions

Expose design

Keep track of your understanding

Refactor to Understand

Expose collaborations

Step through the Execution

Write Tests to Understand

Expose contracts

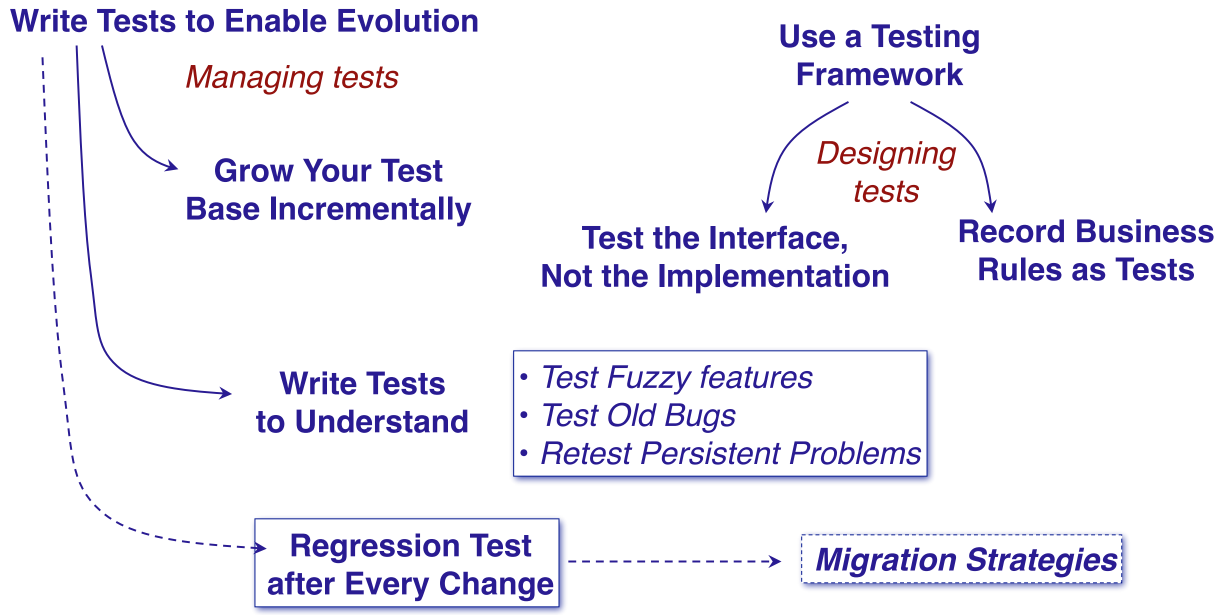
Look for the Contracts

Expose evolution

Learn from the Past

Expose the design & make sure it remains exposed

Tests: Your Life Insurance



Write Tests to Enable Evolution

Managing tests

Grow Your Test Base Incrementally

Write Tests to Understand

- *Test Fuzzy features*
- *Test Old Bugs*
- *Retest Persistent Problems*

Regression Test after Every Change

Migration Strategies

Use a Testing Framework

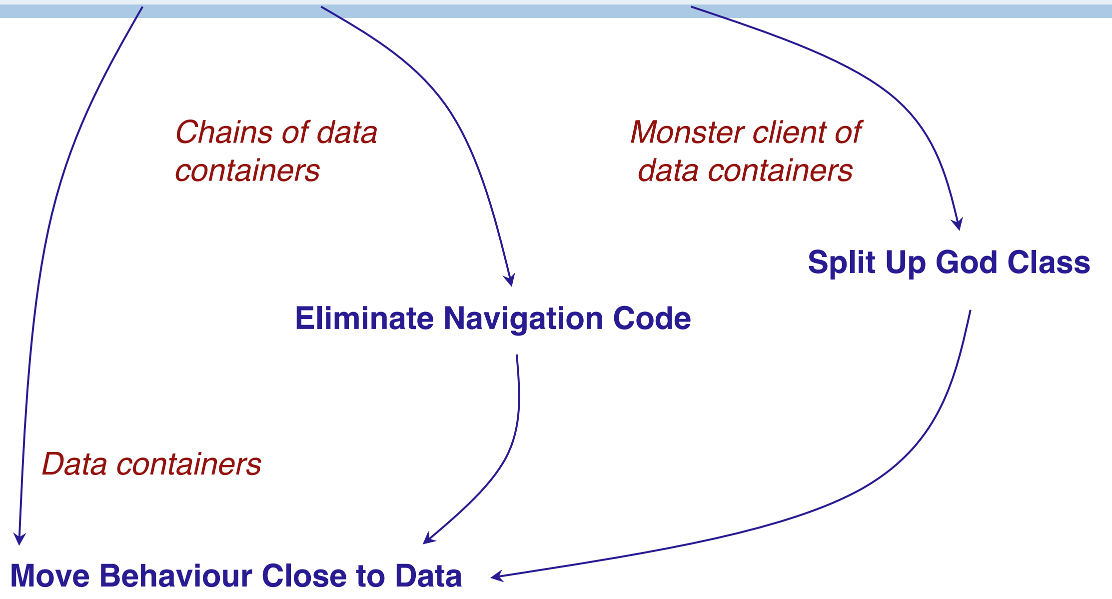
Designing tests

Test the Interface, Not the Implementation

Record Business Rules as Tests

Heads up to Cucumber

Redistribute Responsibilities



Chains of data containers

Monster client of data containers

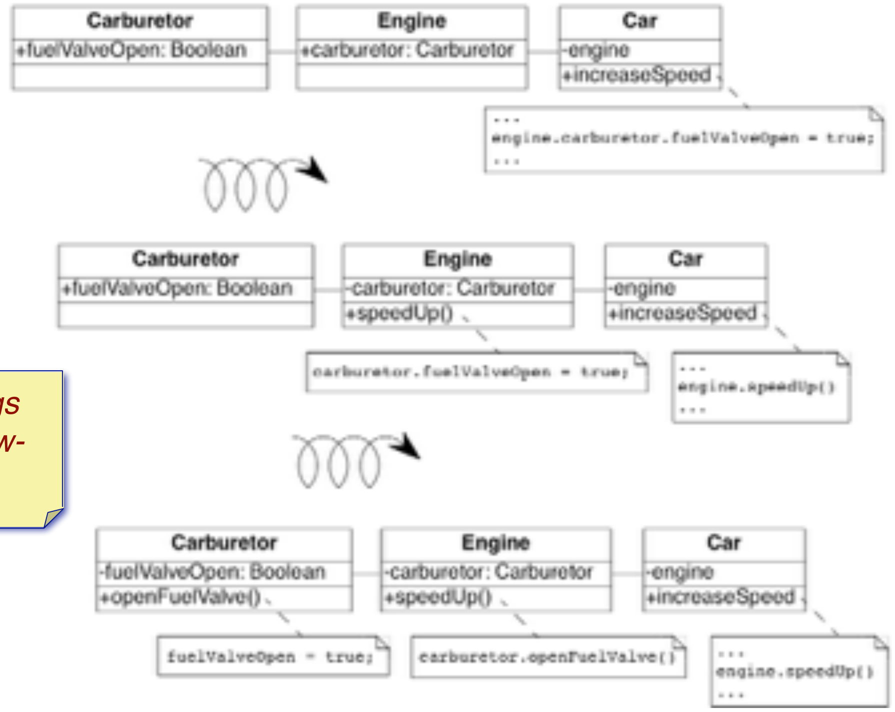
Split Up God Class

Eliminate Navigation Code

Data containers

Move Behaviour Close to Data

High-level refactorings



```
...
engine.carburetor.fuelValveOpen = true;
...
```

```
carburetor.fuelValveOpen = true;
```

```
...
engine.speedUp()
...
```

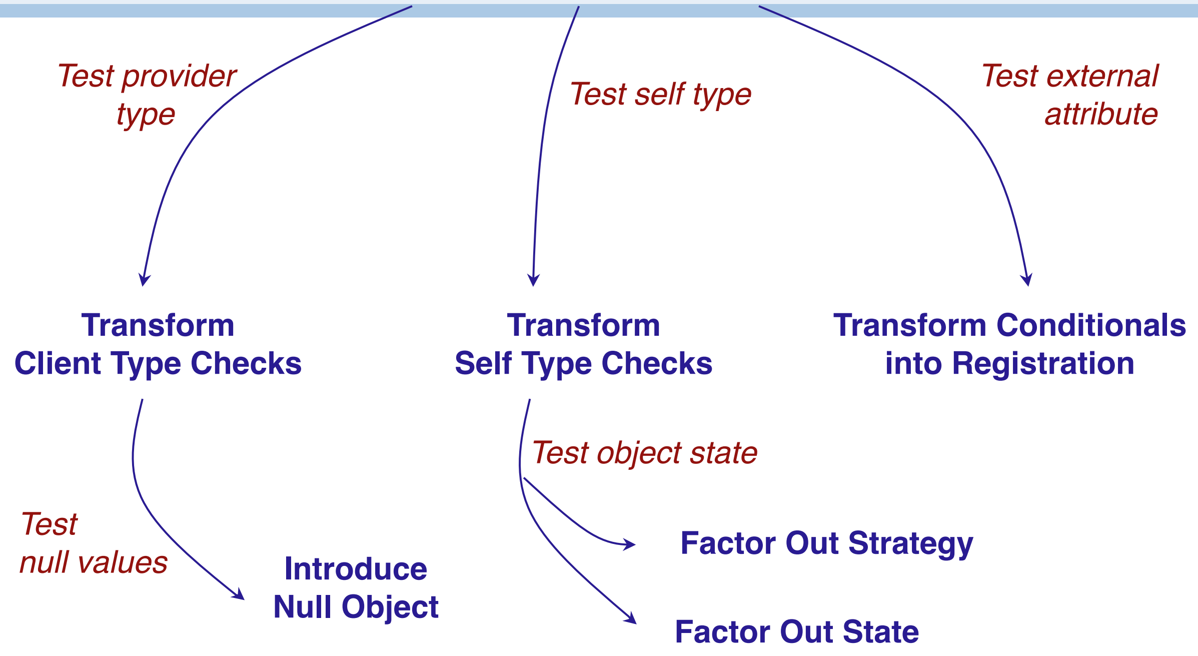
```
fuelValveOpen = true;
```

```
carburetor.openFuelValve()
```

```
...
engine.speedUp()
...
```

High-level refactorings make use of many low-level refactorings

Transform Conditionals to Polymorphism



Test provider type

Test self type

Test external attribute

Transform Client Type Checks

Transform Self Type Checks

Transform Conditionals into Registration

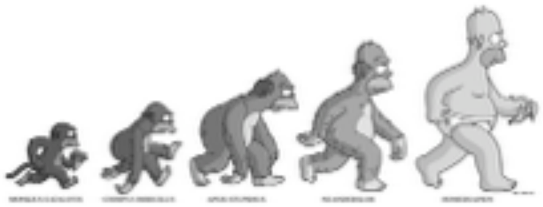
Test null values

Introduce Null Object

Test object state

Factor Out Strategy

Factor Out State

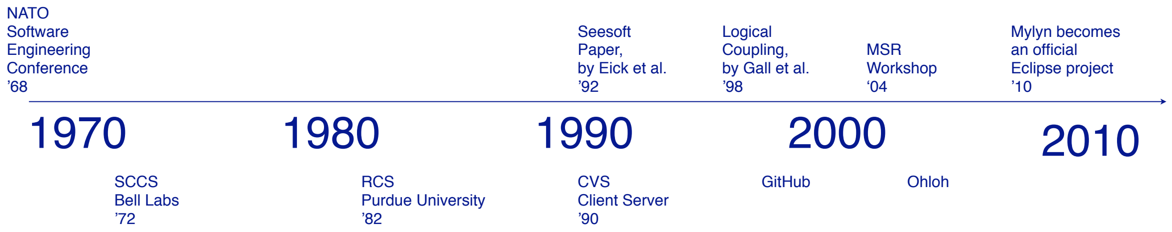


Laws of Software Evolution

Reverse and Reengineering

Mining Software Evolution

Versioning Systems and Applications



NATO Software Engineering Conference ’68

Seesoft Paper, by Eick et al. ’92

Logical Coupling, by Gall et al. ’98

MSR Workshop ‘04

Mylyn becomes an official Eclipse project ’10

1970 1980 1990 2000 2010

SCCS Bell Labs ’72

RCS Purdue University ’82

CVS Client Server ’90

GitHub

Ohloh

20 years of **VCS** before people start doing research in **analyzing software repositories.**

20 more years until software evolution research results are integrated in the IDE.

WCRE, ICSE – constant research is being generated
No way we can cover everything.
Goal is to present several highlights so you can have a feeling on the general directions of the field.

2,683,002 people hosting over 4,502,148 repositories

git /git/

git-hub

Plans, Pricing and Signup

Team management

Code review

Reliable code hosting

Open source collaboration

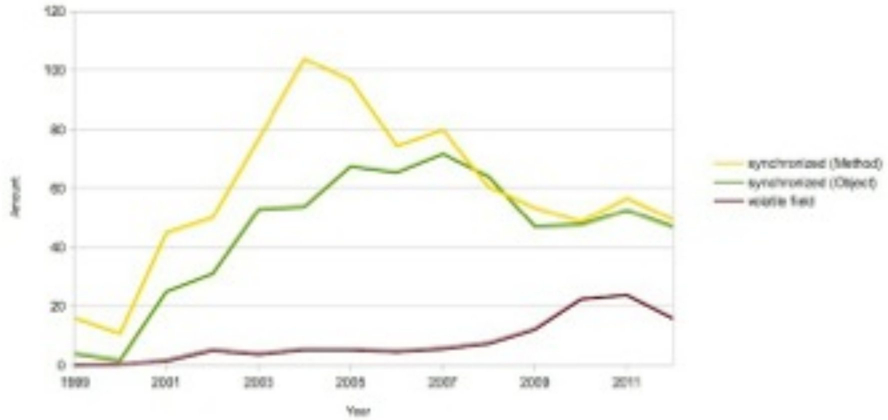
Learn about how people build software

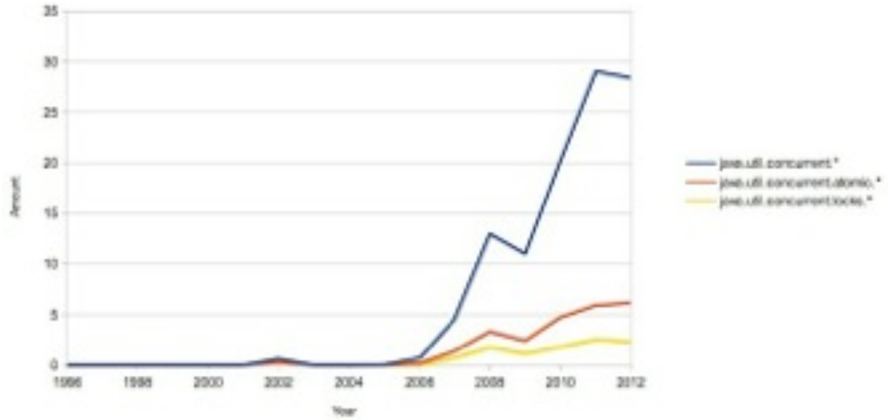
- detecting the source of a given piece of software
- validate hypotheses: are people using metaprogramming?
-

SC Seminar Project
by Julian S. & Roger K.

Are people migrating to the **new** concurrency libraries of **Java?**

Source: GitHub
Projects: 880
Size: 16,5 GB





Can we learn from developer behavior?

Inspired by Your Browsing History

You viewed

Customers who viewed this also viewed

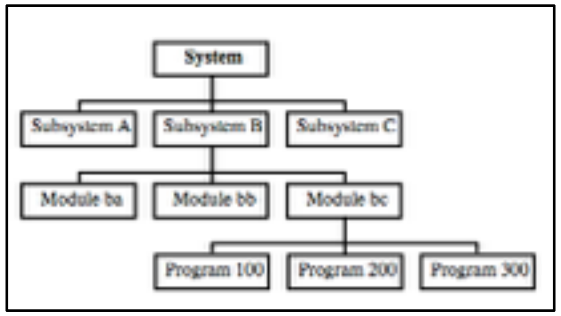
A Fire Upon The Deep (Zones of Thought)
› Vernor Vinge
Paperback
★★★★☆ (293)
~~$14.99~~ $10.19

A Deepness in the Sky
› Vernor Vinge
Mass Market Paperback
★★★★½ (245)
$8.99

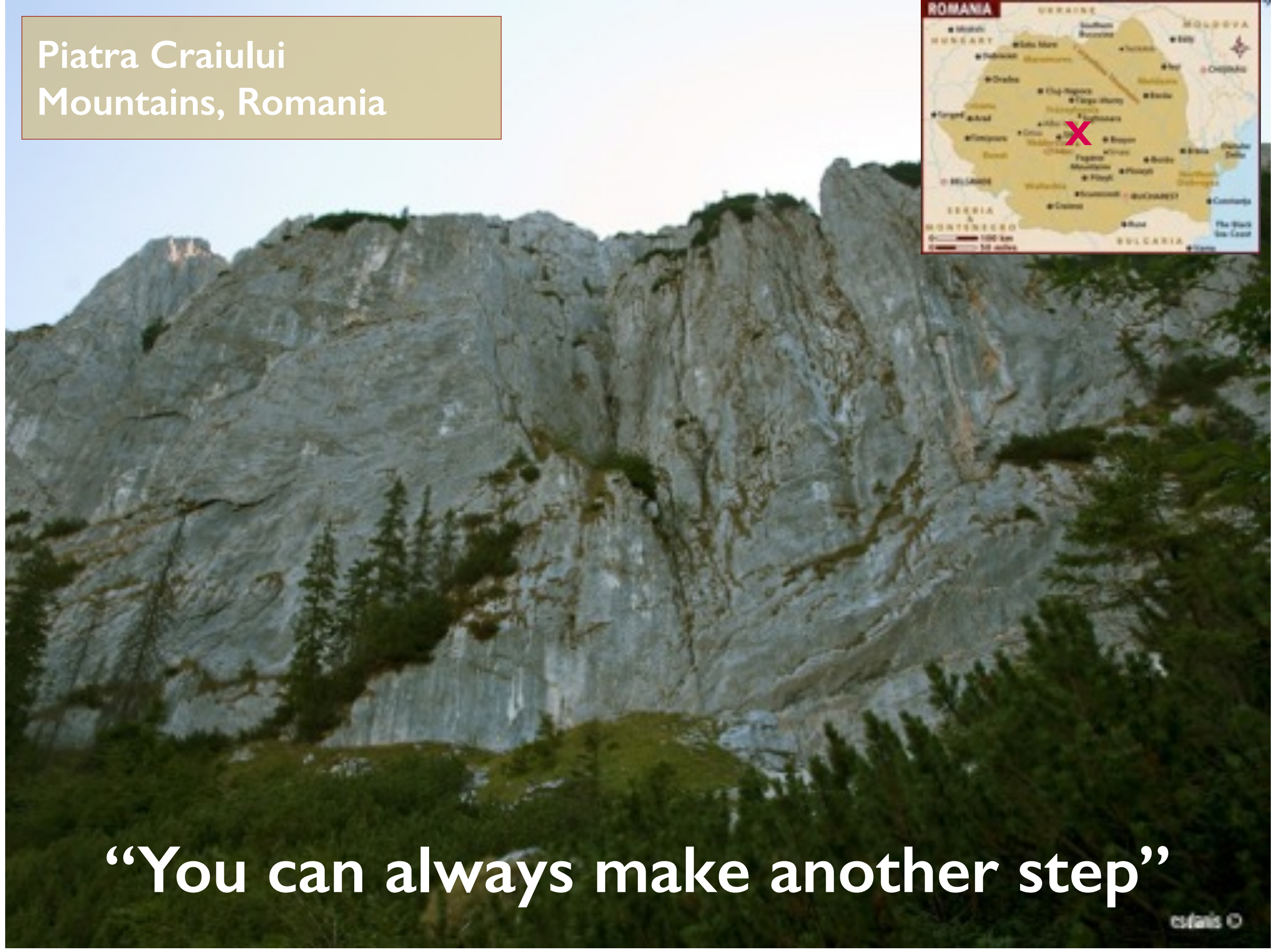
The Forever War
› Joe Haldeman
Paperback
★★★★☆ (545)
~~$14.95~~ $8.59

Invisible Dependencies

- Logical Coupling
  - by Gall et al.
  - things that changed together might change again together
  - advantages over static analysis
    - *can detect IO-based dependencies*
    - *works beyond source code*



Piatra Craiului
Mountains, Romania



“You can always make another step”

esdanis ©

“You can always make another step”


Piatra Craiului Mountains, Romania

“You can always take another step”

(wins in googlefight against “make”)

We are not that original.
Challenge: can we use this lack of originality when writing source code to learn from others?

Clone analysis
for querying

14% - 17% of methods in SqueakSource are clones

On how often is code cloned across repositories
Schwarz et al.

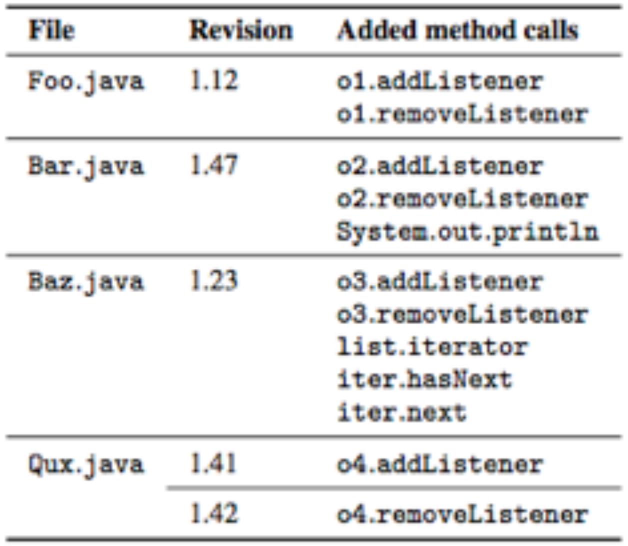
Build better development tools

API Specification Mining

- Data mining reveals frequent patterns
  - Matching Method Pairs
  - State Machines

Principles

1. Mines from history
2. API specific errors
3. Co-addition is a pattern
4. Small commits are fixes



File	Revision	Added method calls
`Foo.java`	1.12	`o1.addListener` `o1.removeListener`

`Bar.java`	1.47	`o2.addListener`
		`o2.removeListener`
		`System.out.println`

`Baz.java`	1.23	`o3.addListener`
		`o3.removeListener`
		`list.iterator`
		`iter.hasNext`
		`iter.next`

`Qux.java`	1.41	`o4.addListener`
	1.42	`o4.removeListener`

some patterns are incomplete, sometimes mistakes, sometimes fixes
– furthermore – you can mine Framework changes. The JUnit example.

Recording IDE Interactions

- Kersten & Murphy ‘05
  - Mylin
  - Task-Focused Interface
  - Degree of Interest ranking

Mylyn

Downloads

Support

Get Started

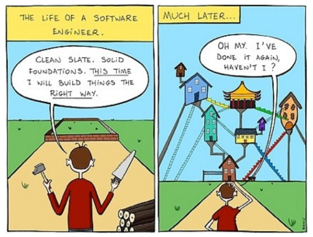
How to filter the large amount of information available in the IDE?

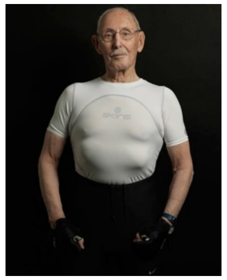
What if we recorded and REPLAYED our development sessions?

+ Correctness (10%)
+ Completion time (6%)

Replaying past changes in multi-developer projects
Lile Hattori et al.

Summary





2,683,002 people hosting over 4,502,148 repositories

git /git/

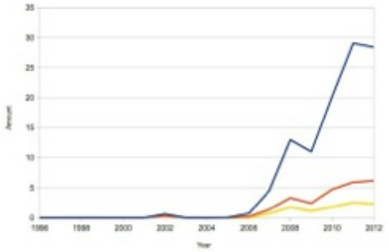
git-hub /git,hub/

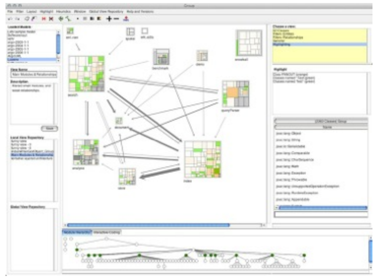
Team management

Code review

Reliable code hosting

Open source collaboration





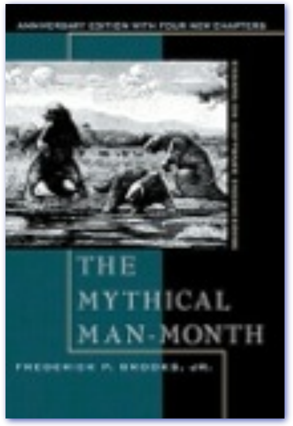
What you should know!

- Three of Lehman & Belady’s Laws of Software Evolution
- Why do software systems become more complex over time
- When should one keep an older version of a system rather than rewrite?
- What is meant by “reverse engineering”?
- How to approach a new software system for reengineering it

Can you answer the following questions?

- How would you ensure that documentation stays in sync with implementation?
- How can you use the history of a system to improve development tools?
- What approach would you take to reengineer a large legacy Java system?

References



http://scg.unibe.ch/download/oorp/

http://www.joelonsoftware.com/