Software Re-Engineering

- → Why software evolves continuously
- → Costs of Software Evolution
- → Challenges of Design Recovery
- → What reverse engineering tools can and can't do

The Altimeter Example

```
IF not-read1(V1) GOTO DEF1;
display (V1);
GOTO C;
DEF1: IF not-read2(V2) GOTO DEF2;
display(V2);
GOTO C;
DEF2: display(3000);
C:
```

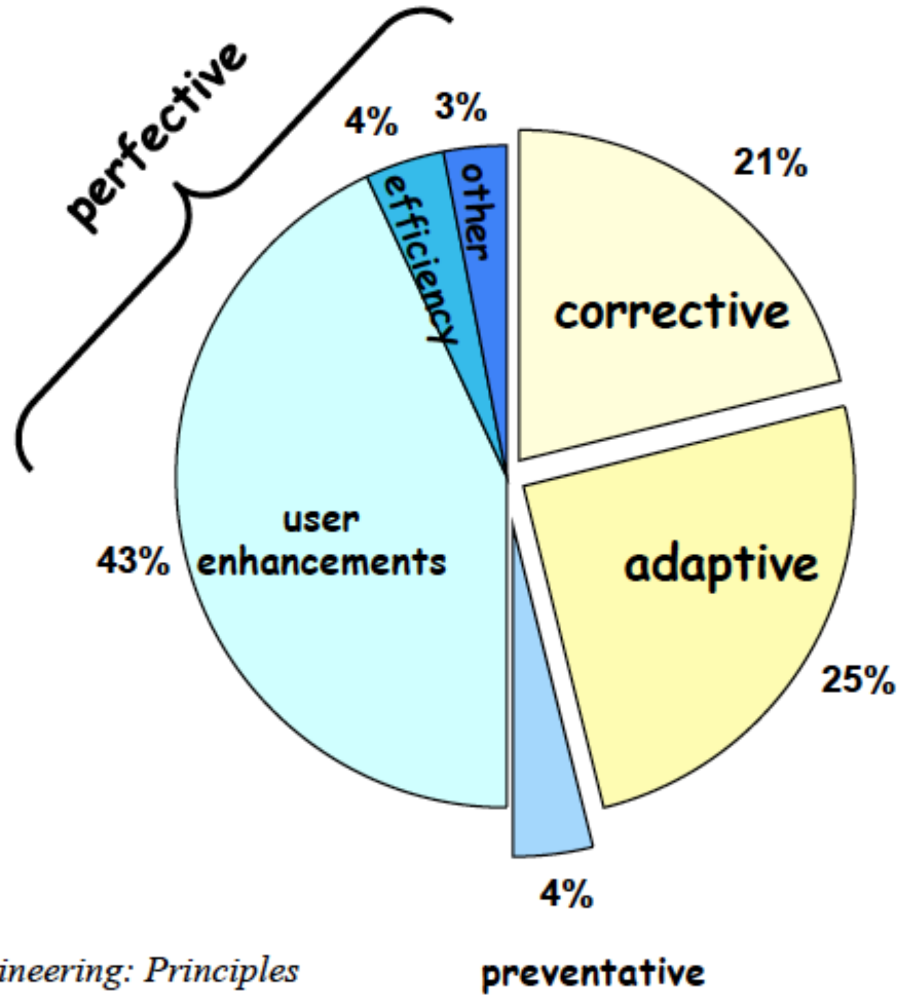
```
if (read-meter1(V1))
  display(V1);
else {
  if (read-meter2(V2))
    display(V2);
  else
    display(3000);
}
```

Questions:

- Should you refactor this code?
- Should you fix the default value?

***Source:** Adapted from van Vliet 1999*

Software Evolves Continuously



Data from:
van Vliet, H., *Software Engineering: Principles and Practices*, Wiley 1999, p449

Program Types

Source: Adapted from Lehman 1980, pp1061-1063

S-type Programs ("Specifiable")

- problem can be stated formally and completely
- acceptance: Is the program correct according to its specification?
- "evolution" not relevant
  - A new specification defines a new problem

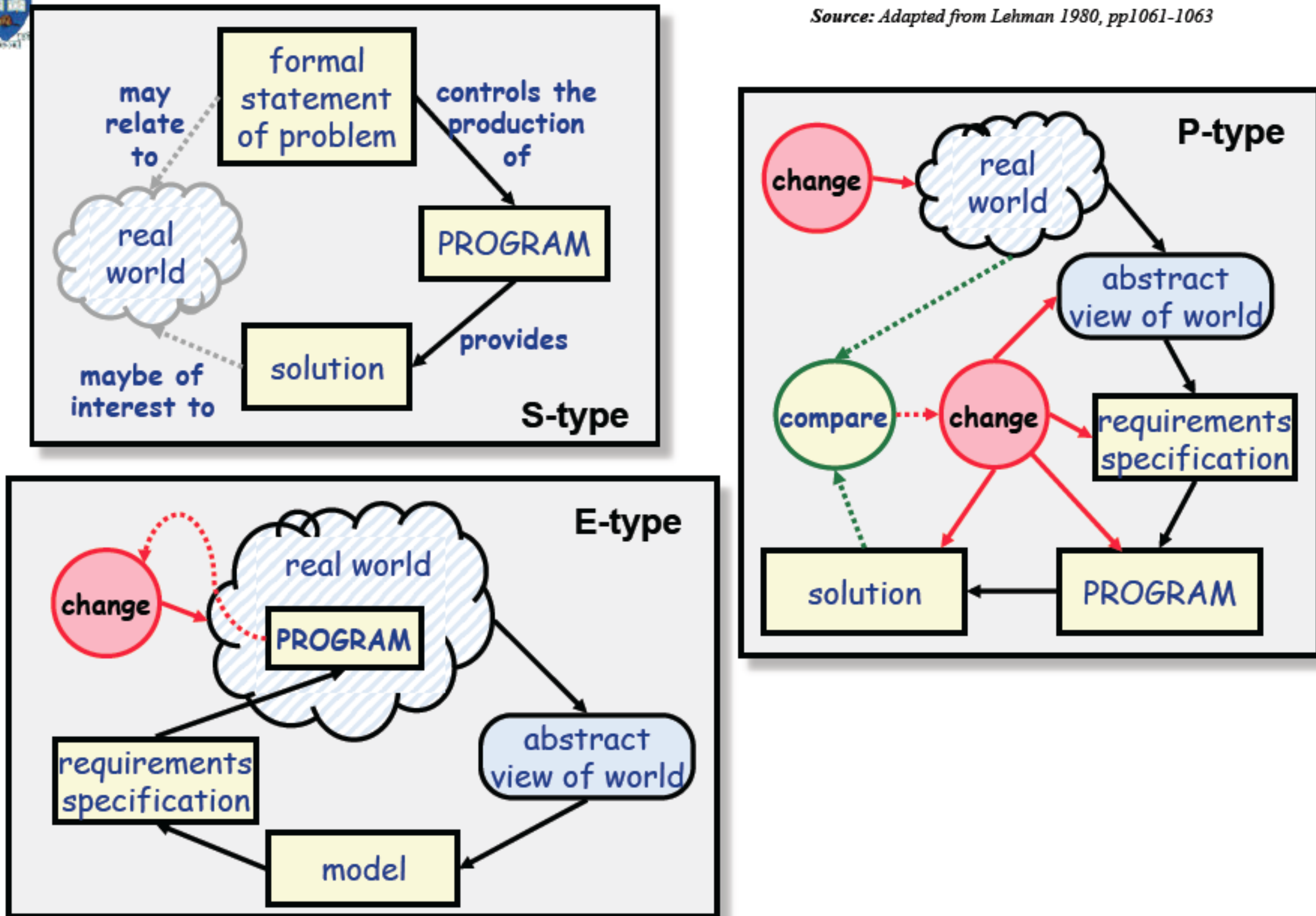
P-type Programs ("Problem-solving")

- imprecise statement of a real-world problem
- acceptance: Is the program an acceptable solution to the problem?
- This software may evolve continuously
  - the solution is never perfect, and can be improved
  - the real-world changes and hence the problem changes

E-type Programs ("Embedded")

- software that becomes part of the world that it models
- acceptance: depends entirely on opinion and judgment
- This software is inherently evolutionary
  - changes in the software and the world affect each other

Source: Adapted from Lehman 1980, pp1061-1063



Laws of Program Evolution

Source: Adapted from Lehman 1980, pp1061-1063

Continuing Change

Any software that reflects some external reality undergoes continual change or becomes progressively less useful

change continues until it is judged more cost effective to replace the system

Increasing Complexity

As software evolves, its complexity increases…

…unless steps are taken to control it.

Fundamental Law of Program Evolution

Software evolution is self-regulating

…with statistically determinable trends and invariants

Conservation of Organizational Stability

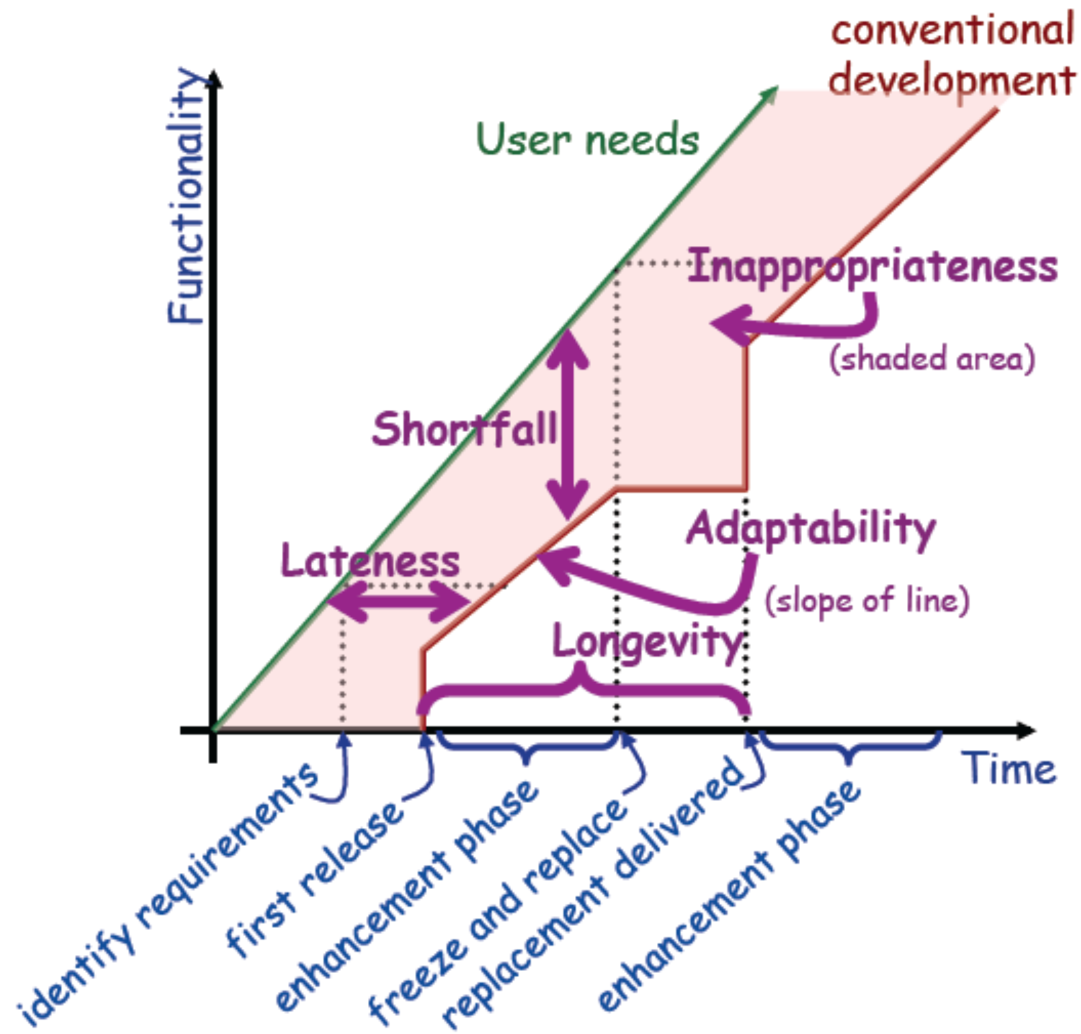
During the active life of a software system, the work output of a development project is roughly constant (regardless of resources!)

Conservation of Familiarity

The amount of change in successive releases is roughly constant

No.	Brief Name	Law
I 1974	Continuing Change	*E*-type systems must be continually adapted else they become progressively less satisfactory.
II 1974	Increasing Complexity	As an *E*-type system evolves its complexity increases unless work is done to maintain or reduce it.
III 1974	Self Regulation	*E*-type system evolution process is self regulating with distribution of product and process measures close to normal.
IV 1980	Conservation of Organisational Stability (invariant work rate)	The average effective global activity rate in an evolving *E*-type system is invariant over product lifetime.
V 1980	Conservation of Familiarity	As an *E*-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behaviour [leh80a] to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
VI 1980	Continuing Growth	The functional content of *E*-type systems must be continually increased to maintain user satisfaction over their lifetime.
VII 1996	Declining Quality	The quality of *E*-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
VIII 1996	Feedback System (first stated 1974, formalised as law 1996)	*E*-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

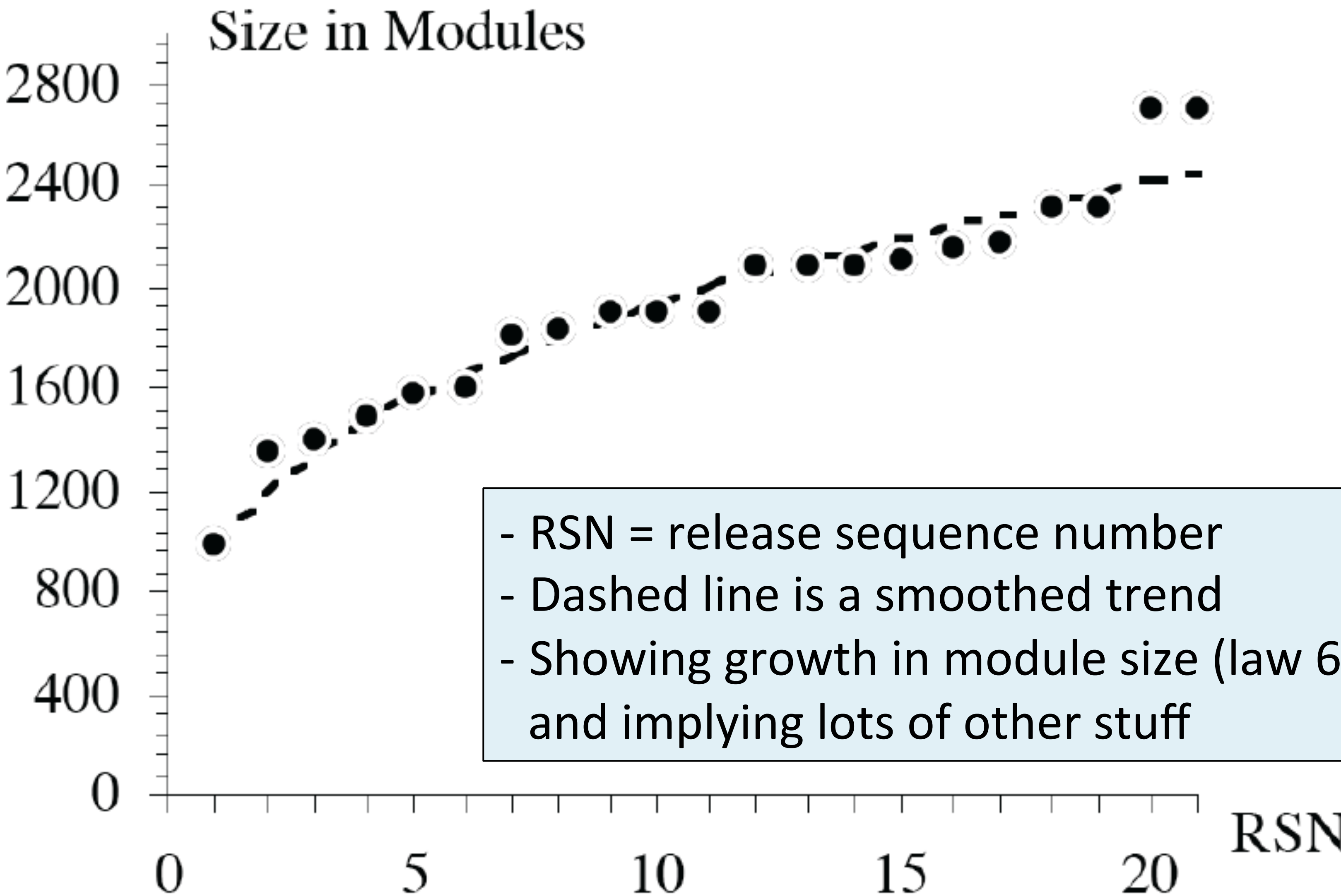
User requirements always grow



Source: Adapted from Davis 1988, pp1453-1455

E.g. Logica Financial Software

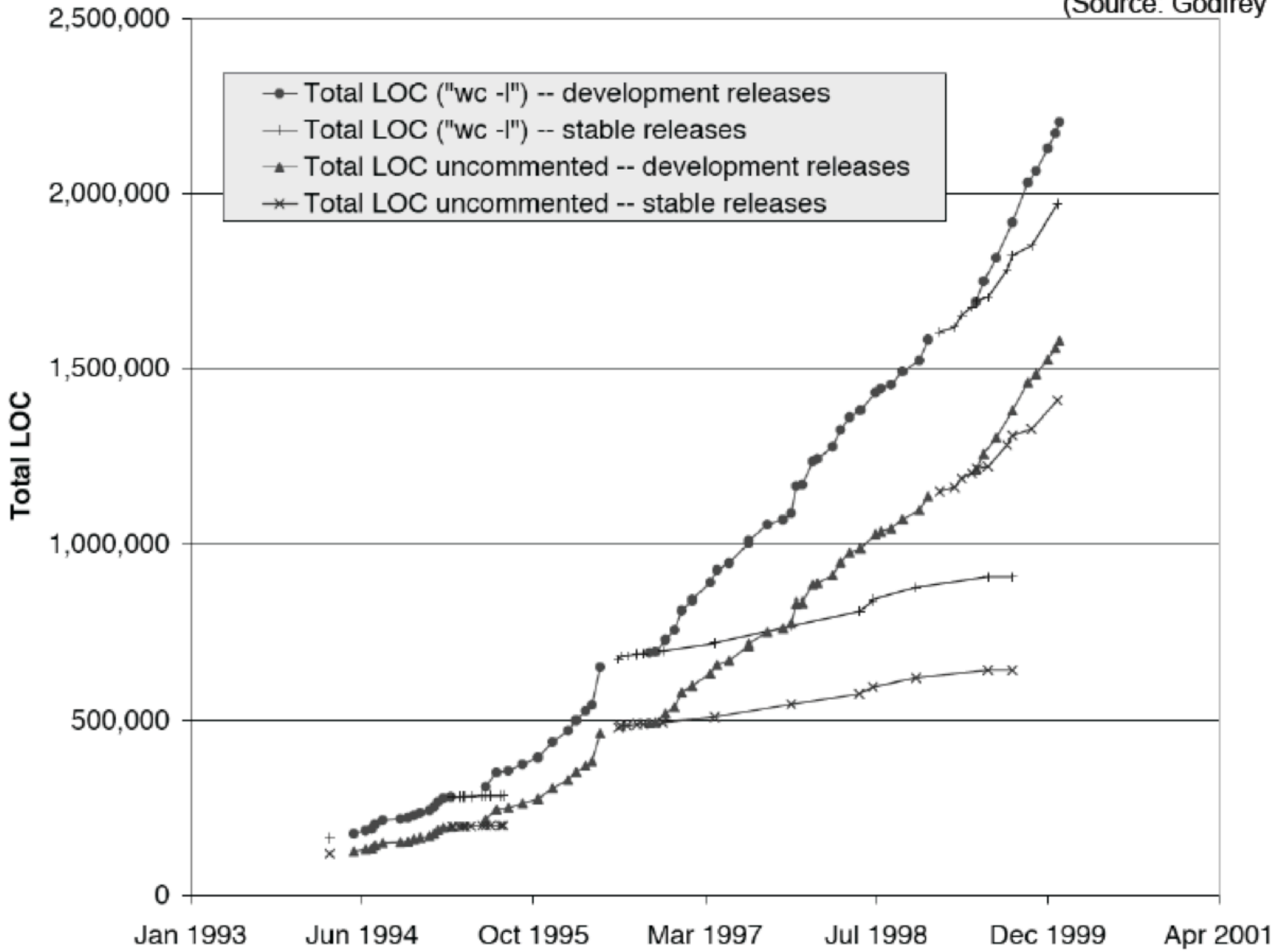
(Source: Lehman et al, 2000)



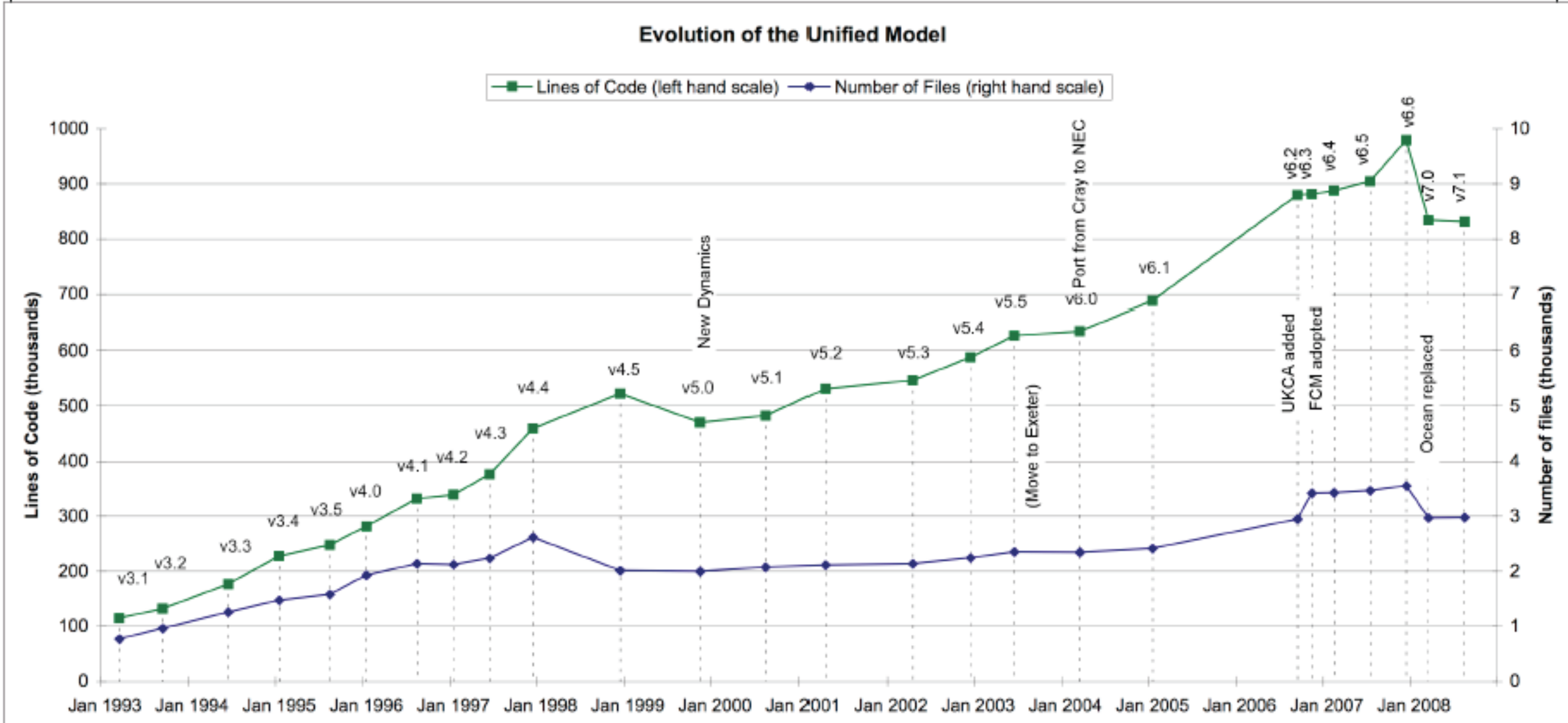
- RSN = release sequence number
- Dashed line is a smoothed trend
- Showing growth in module size (law 6) and implying lots of other stuff

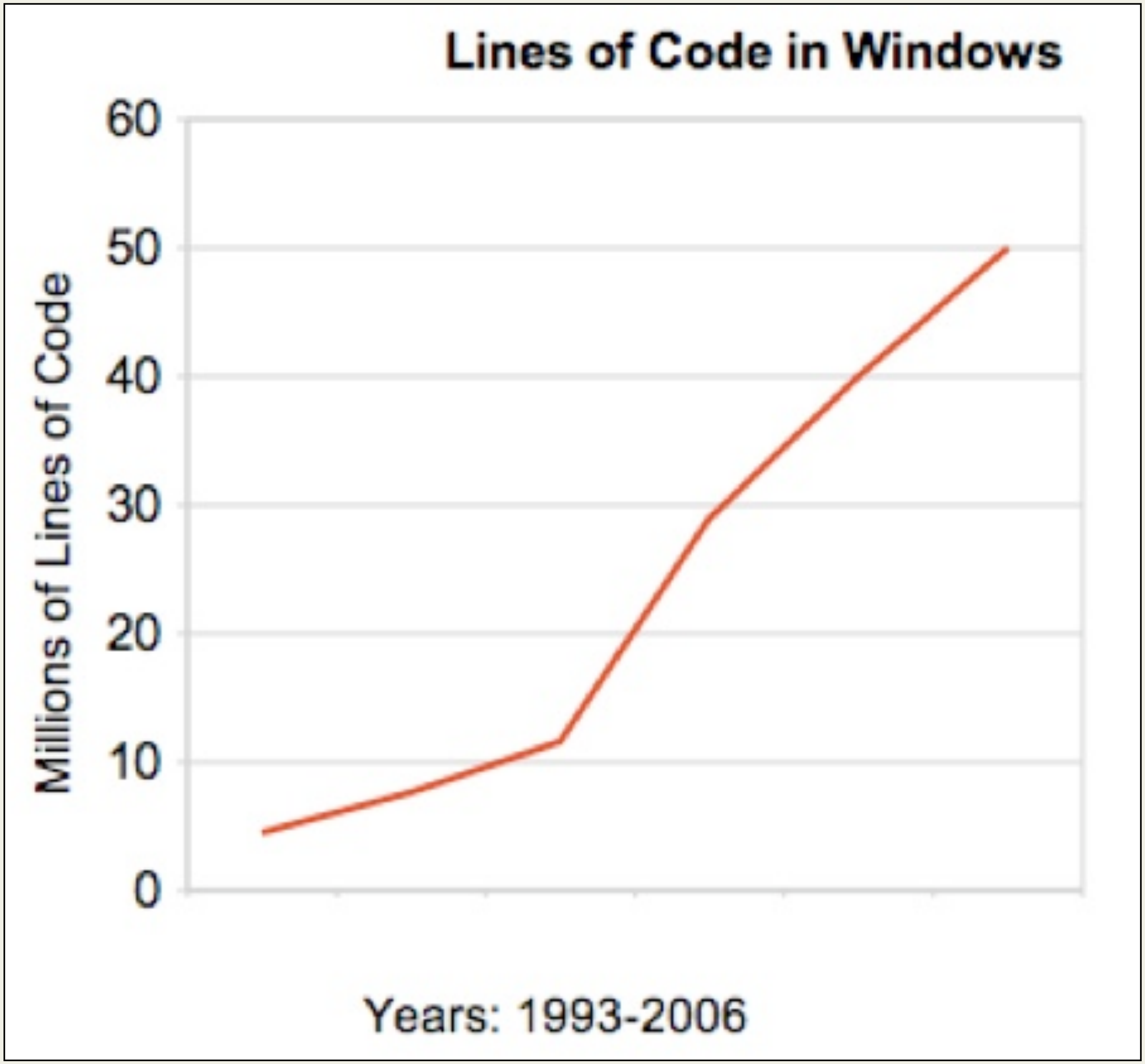
E.g. Linux Kernel

(Source: Godfrey & Tu, 2000)



E.g. Hadley Centre Climate Model





Lines of Code in Windows
Millions of Lines of Code
60
50
40
30
20
10
0
Years: 1993-2006

Software Geriatrics

***Source:** Adapted from Parnas, "Software Aging" 1996*

Causes of Software Aging

- Failure to update the software to meet changing needs
  - Customers switch to a new product if benefits outweigh switching costs
- Changes to software tend to reduce coherence & increase complexity

Costs of Software Aging

- Owners of aging software find it hard to keep up with the marketplace
- Deterioration in space/time performance due to deteriorating structure
- Aging software gets more buggy
  - Each "bug fix" introduces more errors than it fixes

Ways of Increasing Longevity

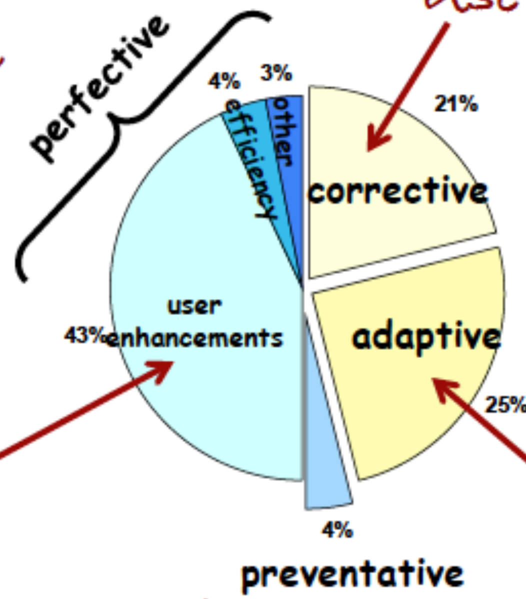
- Design for change
- Document the software carefully
- Requirements and designs should be reviewed by those responsible for its maintenance
- Software Rejuvenation…

Reducing Maintenance Costs

General

Modular structure
Comprehensibility
Good documentation

Higher quality code
Better testing (verification)
Use of standards



Platform independence
Design for change
Good architecture

Better requirements analysis
prototyping, iterative development
Design for change

Why maintenance is hard

Poor code quality

- opaque code
- poorly structured code
- dead code

Lack of knowledge of the application domain

- understanding the implications of change

Lack of documentation

- code is often the only resource
- missing rationale for design decisions

Lack of glamour

***Source:** Adapted from van Vliet 1999*

Rejuvenation

Reverse Engineering

- Re-documentation (same level of abstraction)
- Design Recovery (higher levels of abstraction)

Restructuring

- Refactoring (no changes to functionality)
- Revamping (only the user interface is changed)

Re-Engineering

- Real changes made to the code
- Usually done as round trip:
  design recovery -> design improvement -> re-implementation

Source: *Adapted from van Vliet 1999*

Program Comprehension

During maintenance:

- programmers study the code about 1.5 times as long as the documentation
- programmers spend as much time reading code as editing it

Experts have many knowledge chunks:

- programming plans
- beacons
- design patterns

Experts follow dependency links

- ...while novices read sequentially

Much knowledge comes from outside the code

***Source:** Adapted from van Vliet 1999*

Example 1

What does this do?

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    if (A[i,j]) {
      for (k=0; k<n; k++) {
        if (A[j,k])
          A[i,k]=true;
      }
    }
  }
}
```

***Source:** Adapted from van Vliet 1999*

Example 2

```
procedure A(var x: w);
begin
  b(y, n1);
  b(x, n2);
  m(w[x]);
  y := x;
  r(p[x]);
end;
```

```
procedure change_window(var nw: window);
begin
  border(current_window, no_highlight);
  border(nw, highlight);
  move_cursor(w[nw]);
  current_window := nw;
  resume(process[nw]);
end;
```

***Source:** Adapted from van Vliet 1999*

What tools can do

Reformatters / documentation generators

- Make the code more readable
- Add comments automatically

Improve Code Browsing

- E.g visualize and traverse a dependency graph

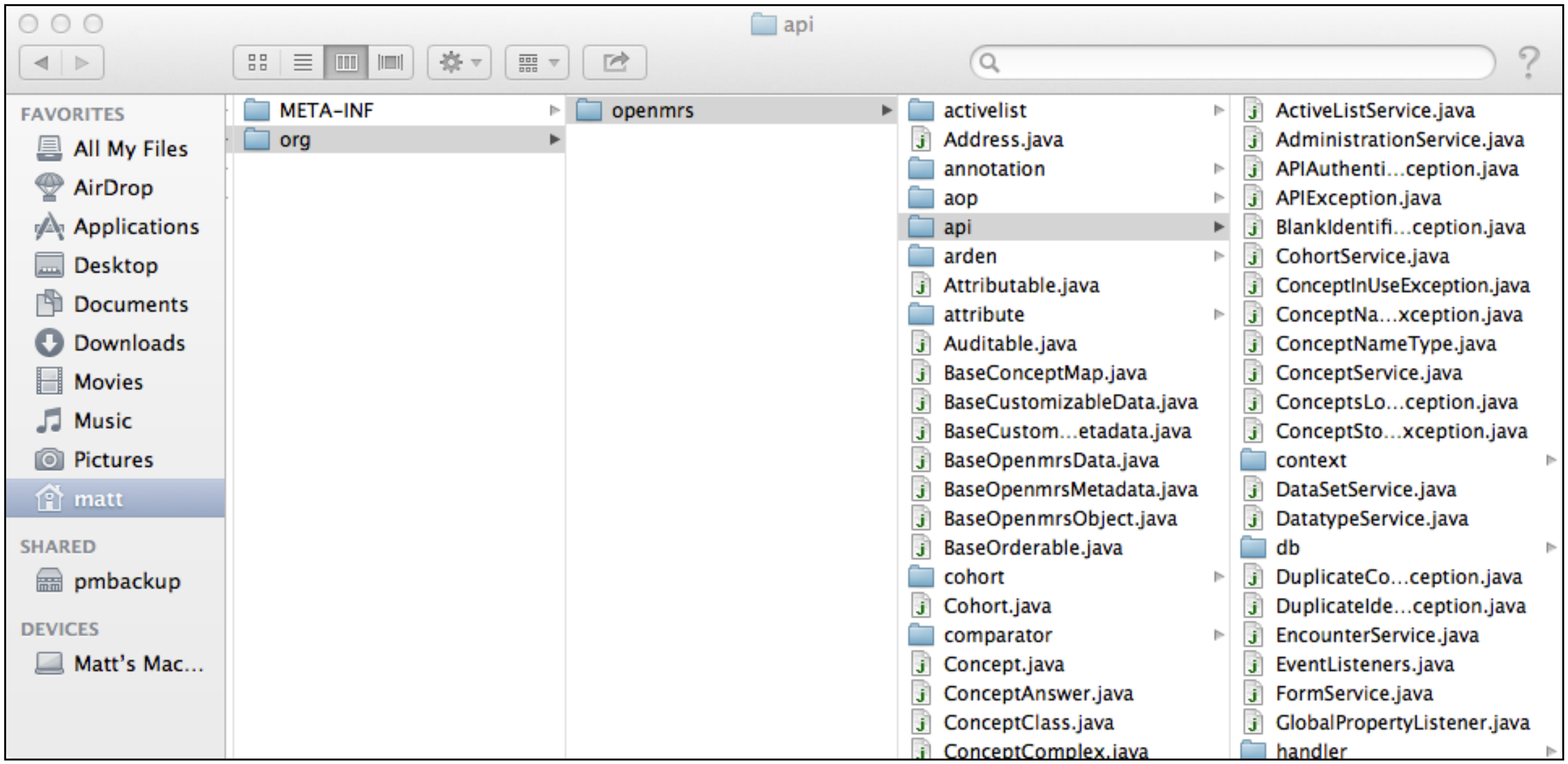
(simple) Code transformation

- E.g. Refactoring class browsers
- E.g. Clone detectors

(simple) Design Recovery

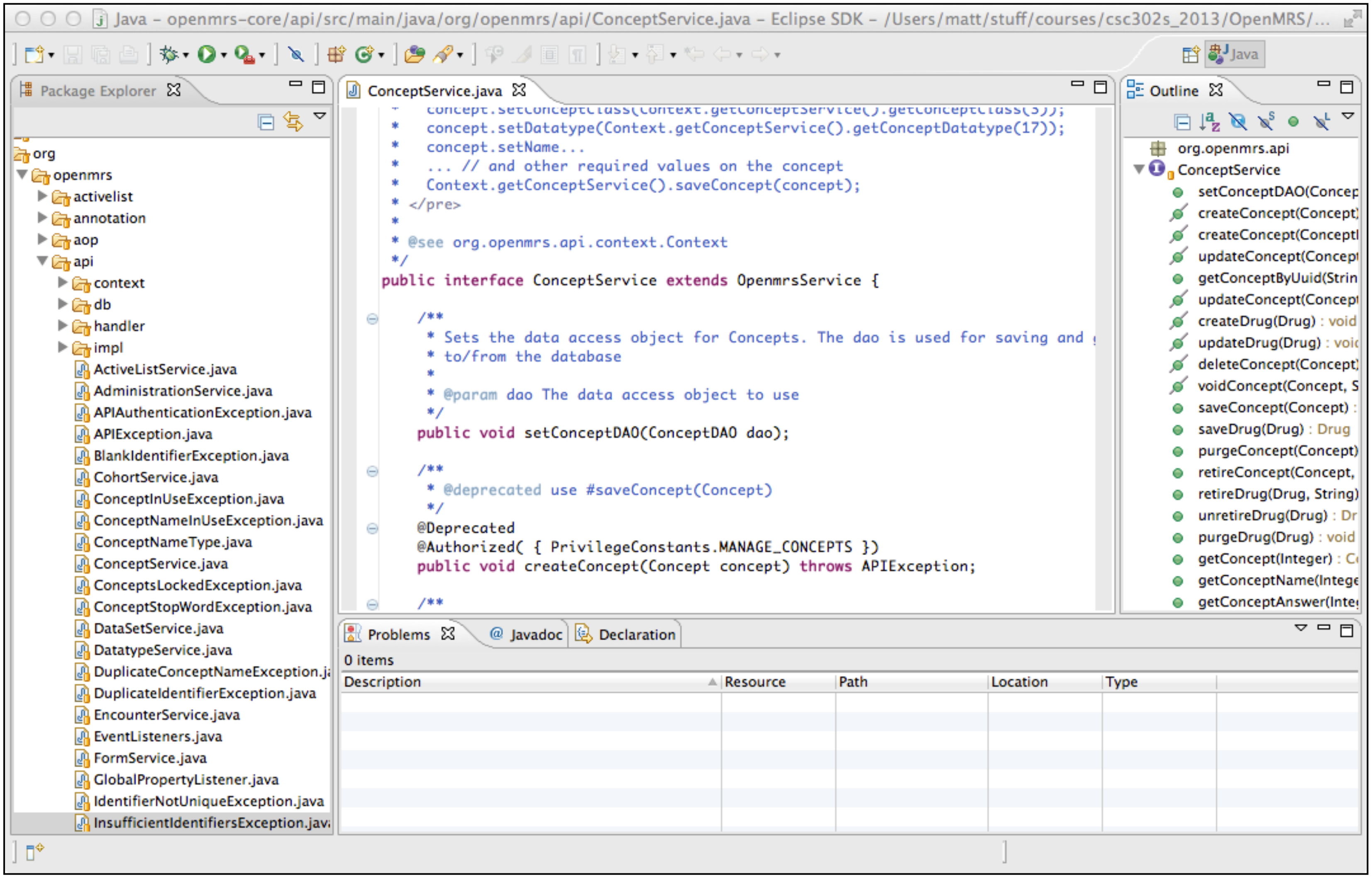
- E.g. build a basic class diagram
- E.g. use program traces to build sequence diagrams

design discovery tools



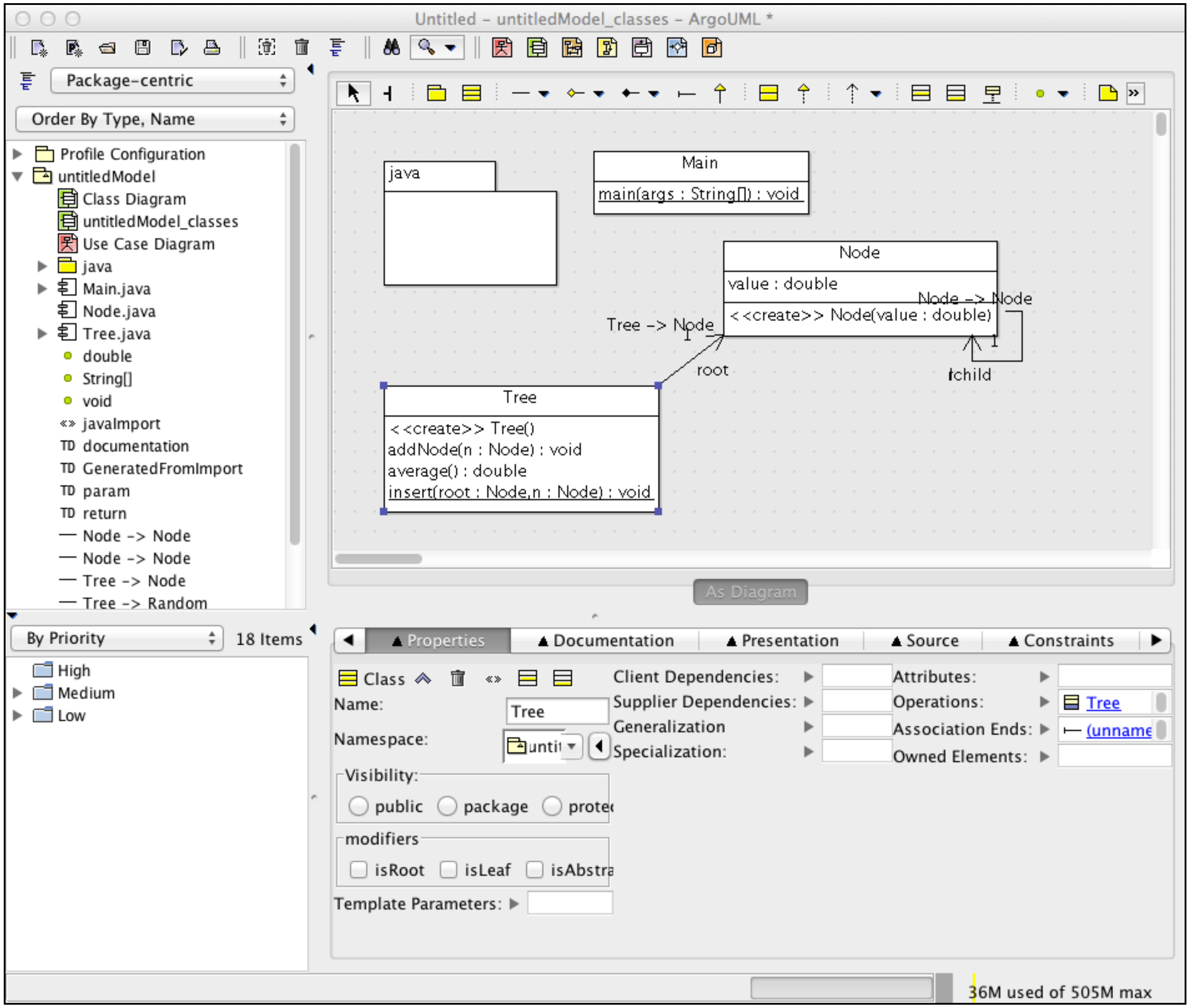
Source: Adapted from van Vliet 1999

design discovery tools (2)



Source: Adapted from van Vliet 1999

design discovery tools (3)



Source: Adapted from van Vliet 1999