

CSE584: Software Engineering

Lecture 8 (November 24, 1998)

David Notkin
Dept. of Computer Science & Engineering
University of Washington
www.cs.washington.edu/education/courses/584/CurrentQtr/

This week and next

- Tools and analysis
- In some ways, at least the analysis part might be close to the specification topics covered last week
- But the focus will be much, much closer to the source code

Notkin (c) 1997, 1998

2

What's a tool?

- Merriam-Webster
 - a handheld device that aids in accomplishing a task; the cutting or shaping part in a machine or machine tool; a machine for shaping metal
- From Jargon File (4.0.0/24 July 1996)
 - A program used primarily to create, manipulate, modify, or analyze other programs, such as a compiler or an editor of a cross-referencing program
 - For these lectures, I will omit "intellectual" tools, focusing instead on this kind of tool

Notkin (c) 1997, 1998

3

What's analysis?

- Merriam-Webster
 - an examination of a complex, its elements, and their relations
- What do I mean (for these lectures)?
 - Computer-supported "reasoning" that is intended to aid a programmer in understanding and improving a software system

- One thing I learned is that psychoanalysts don't have much of a sense of humor
 - You try to find a picture of a person laying on a couch talking to a psychoanalyst somewhere on the web!

Notkin (c) 1997, 1998

4

Tools you probably use

- Compilers
- Editors
- Debuggers
 - How often?
- Profilers
 - How often?
- Configuration management and version control tools

Notkin (c) 1997, 1998

5

What other tools do you use?

- GUI builders
 - What kinds, how frequently?
- Testing tools
 - What kinds, how frequently?
- What else?

Notkin (c) 1997, 1998

6

Some context

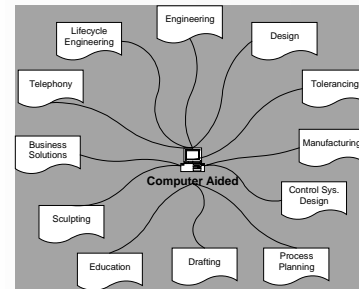
- Environments and CASE

Notkin (c) 1997, 1998

7

CASE

- Computer Aided Software Engineering
- Using computers to help people engineer software



Notkin (c) 1997, 1998

8

Environments vs. tools

- Integrated
 - User interacts with a single environment
 - Environment is responsible for managing consistency
- Standalone
 - User interacts with each tool separately
 - User must apply tools appropriate to ensure consistency
- Sharing of representation
- Independent representations

Notkin (c) 1997, 1998

9

Why environments?

- "During the past decade there has been a growing realization [that software tools] have by and large failed to reduce cost and improve quality. ... [T]he essence of an environment is that it attempts to redress the failures of individual software tools through synergistic integration."
 - Osterweil, 1981

Notkin (c) 1997, 1998

10

Why environments?

- "Current software development environments often help programmers solve their programming problems by supplying tools such as editors, compilers, and linkers, but rarely do these environments help projects solve their system composition or management problems."
 - Notkin & Habermann, 1979

Notkin (c) 1997, 1998

11

What are environments?

- "A software development environment consists of a set of techniques to assist the developers of software systems, supported by some (possibly automated) tools, along with an organizational structure to manage the process of software development. Historically, these facilities have been poorly integrated."
 - Wasserman, 1981

Notkin (c) 1997, 1998

12

Computer help?

- Interaction and rich user interfaces
- Translation of high-level descriptions
- Maintaining consistency among large and complex representations
- Encoding knowledge about an activity, organization or process
- Broader availability of pertinent information
- Communication medium

Notkin (c) 1997, 1998

13

The "promise" of CASE

Notkin (c) 1997, 1998

14

CASE is ...

- "The CASE technology is a combination of software tools and methodologies. ... Spanning all phases of the software lifecycle, not just on implementation, CASE is the most complete software technology yet."
- McClure, 1989

Notkin (c) 1997, 1998

15

CASE is ...

- "To be truly successful, a CASE product must literally support application development from womb to tomb, from the initial conception of an application through its long-term maintenance."
- Towner, 1989

Notkin (c) 1997, 1998

16

CASE is

- "CASE provides the rigorous automated support required to build flexible, high-quality information systems quickly."
- Mylles, 1994

Notkin (c) 1997, 1998

17

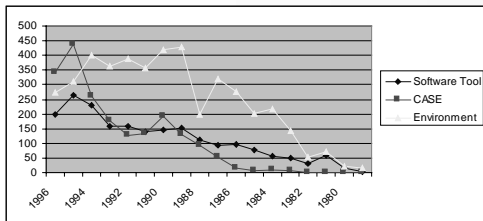
CASE confusion

- Environments in academia, CASE in industry
- CASE in information systems & information technology
 - The MIS world of CIOs
 - Application Development (AD)
 - Information Engineering (IE)
 - Methodology plays a key role
- Upper CASE vs. lower CASE

Notkin (c) 1997, 1998

18

Appearances in INSPEC



- Programming *environment*, program development *environment*, software development *environment*, software engineering *environment*, integrated program support *environment*

Notkin (c) 1997, 1998

19

(A few) classic environments

- Interlisp
- Smalltalk-80
- Unix
- Cedar

Notkin (c) 1997, 1998

20

Interlisp (Xerox PARC)

- Teitelman & Masinter, 1981
- Language-centered environment
- Very fast turnaround for code changes
- Monolithic address space
 - Environment, tools, application code commingled
- Code and data share common representation

Notkin (c) 1997, 1998

21

Smalltalk-80 (Xerox PARC)

- Goldberg, 1984
- Language-centered environment (OO)
 - Classes as first-class objects, inheritance, etc.
- Environment structured around language features (class browsers, protocols, etc.)
- Rich libraries (data structures, UI, etc.)

Notkin (c) 1997, 1998

22

Unix (Bell Labs)

- Toolkit-based environment
- Simple integration mechanism
 - Convenient user-level syntax for composition
- Standard shared representation
- Language-independent (although biased)
- Efficient for systems' programming

Notkin (c) 1997, 1998

23

Cedar (Xerox PARC)

- Teitelman, 1984
- Intended to mix best features of Interlisp, Smalltalk-80, and Mesa
- Primarily was an improvement on Mesa
 - Language-centered environment
 - Abstract data type language
 - Strong language and environment support for interfaces
 - Key addition: garbage collection

Notkin (c) 1997, 1998

24

Commercialization

- 22 companies matched "CASE" in Company Profiles database
 - About 10,000 matched "software"
 - 23 matched "application development"
- 3 Yahoo CASE categories
 - 55-60 registered CASE pages in Yahoo
 - (35 Java categories, thousands of pages)
- (As of spring 1997)

Notkin (c) 1997, 1998

25

The business of CASE

- IDE (Software through Pictures)
 - Founded 1983
 - Acquired by Thomson-CSF 1996
 - ~\$10M annual sales
- Rational
 - Founded 1982
 - Recently purchase Pure Atria
 - ~\$91M annual sales (+ ~\$132M annual from Pure Atria)

Notkin (c) 1997, 1998

26

The business of CASE

- Popkin
 - Founded 1986
 - ~\$15M annual sales
- Cayenne Software, Inc. (1996)
 - Merger of Bachman (1983) and CADRE (1982)
 - ~\$14M annual sales
- StructSoft (TurboCASE/Sys)
 - Formed 1984
 - ~\$6M annual sales

Notkin (c) 1997, 1998

27

The business of CASE

- I-Logix
 - Founded 1987
 - ~\$10M annual sales
- Reasoning Systems
 - Founded 1984
 - ~\$20M annual sales

Notkin (c) 1997, 1998

28

Some context

- SAP AG
 - R/3 Business Process (Re)Engineering
 - Founded 1972
 - >\$2B annual revenue
- Largest "application development" company in Company Profiles
 - Progress Software, 1981
 - ~\$180M annual revenue

Notkin (c) 1997, 1998

29

CASE quotation I

- "Despite the many grand predictions of the trade press over the past decade, computer-assisted software engineering (CASE) tools failed to emerge as the promised 'silver bullet.'"
 - Guinan, Coopriider, Sawyer; IBM Systems Journal, 1997

Notkin (c) 1997, 1998

30

CASE quotation II

- "CASE tools are sometimes excessively rigid in forcing the user to input too much information before giving usable results back. CASE tools also typically don't adapt to multiple or in-house methodologies..."
- www.confluent.com; 1997

Notkin (c) 1997, 1998

31

Myth #1 of CASE

- *Integration is job #1*
- Integrating tools helps, but only if the tools are the "right" tools
- That is, integration is a second order effect, not a first order effect

Notkin (c) 1997, 1998

32

Myth #2 of CASE

- *Graphics inherently dominate text*
- "A picture is worth a thousand words"
- This is a complicated issue
 - Screen real estate
 - Sharing with other tools
 - Sharing with other people

Notkin (c) 1997, 1998

33

Myth #3 of CASE

- *Software tools are more important than intellectual tools*
- False
 - Software tools are important but are generally a second order effect
 - Sometimes software tools can qualitatively change the world, although usually indirectly

Notkin (c) 1997, 1998

34

Myth #4 of CASE

- *Tool adoption is a consumer problem not a producer problem*
- False

Notkin (c) 1997, 1998

35

Organizational issues (Orlikowski)

- "CASE Tools as Organizational Change: Investigating Incremental and Radical Changes in Systems Development"
- MIS Quarterly Best Paper, 1993
- "[To] account for the experiences and outcomes associated with CASE tools, researchers should consider the social context of systems development, the intentions and actions of key players, and the implementation process following by the organization."

Notkin (c) 1997, 1998

36

Myth #5 of CASE

- *Goal should be to change how software engineering is done*
- No, it should be to enhance how people are doing software engineering

Notkin (c) 1997, 1998

37

Myth #6 of CASE

- *The tools can handle creative aspects of software engineering*
- Tools frequently fail to be useful because they make poor judgments about what the human does well and what the computer does well

Notkin (c) 1997, 1998

38

Tools

- For at least a while, the pendulum seems to have moved back towards tools rather than environments/CASE
 - This isn't uniformly true, but I think it is true on the whole
- Undoubtedly, at some point the pendulum will swing back again

Notkin (c) 1997, 1998

39

News Flash! Notkin's wrong!

- From the Miller Freeman, Inc. newswire from November 23, 1998
- 'Better, faster, cheaper,' the mantra of technology vendors still drives product development and marketing. But now a new word must be added: integrated. This year's Intelligent Enterprise Dozen offers testament to the importance of application integration. Today, vendors find that customers have the parts; now they want the whole. Best-of-breed products must adhere to industry standards; packages must work together; and productivity tools, such as decision support, must become closed-loop solutions.

Notkin (c) 1997, 1998

40

Programming languages

- Many (researchers, especially) claim that with better programming languages, we'd have no need (or significantly reduced need) for software tools
- How many of you took CSE583?
 - Scheme, ML, Haskell, etc.
- Assuming you could/would build most commercial products with such languages, which tools would you give up?
 - Probably none

Notkin (c) 1997, 1998

41

However

- The underlying premises and implementation structures for many tools and language implementations are closely related
- Examples include:
 - The program dependence graph representation is heavily used in program optimization and parallelization, as well as in software engineering tools
 - Type inference is being used increasingly broadly as the basis for some software engineering tools

Notkin (c) 1997, 1998

42

Types

- Type systems fit into the definition of analysis we're using
- These systems are intended to eliminate some classes of errors in programs
 - Programs in a language are analyzed and ones that don't pass type checking are considered illegal programs (static typing)
 - During execution checks are made to ensure that bit representations are never misinterpreted (dynamic typing)

Notkin (c) 1997, 1998

43

Type inferencing

- A downside of type systems is that the programmer has to write more "stuff"
- Type inferencing has the compiler compute what the types of the expressions should be
 - The programmer writes less down
 - The programmer has less to change when the program is modified
 - The programmer gets almost all the benefits of static typing

Notkin (c) 1997, 1998

44

A simple taxonomy

- Due to Mike Ernst
 - Based primarily on pointer-based issues, but more broadly applicable
- Draws one common distinction and one less common distinction
- Shows desirability of combining static and dynamic analyses
 - Now largely (if not entirely) ad hoc

	Bug Finding	Performance Improvement
Dynamic	Purify	Profiling
Static	LCLint	Alias Anal.

Notkin (c) 1997, 1998

45

LCLint [Evans et al.]

- [Material taken in part from a talk by S. Garland]
- Add some partial specification information to C code to
 - Detect potential bugs
 - Enforce coding style
- Versatile and lightweight
 - Incremental gain for incremental effort
 - Fits in with other tools

Notkin (c) 1997, 1998

46

Detects potential bugs

- Specifications enable more accurate checks, messages
- Memory management a particular problem in the C language

Notkin (c) 1997, 1998

47

Enforces coding style

- Abstraction boundaries
- Use of mutable and immutable types

Notkin (c) 1997, 1998

48

LCLint Does Not

- Encourage programmer to write
 - Contorted code
 - Inefficient code
- Report only actual errors
- Report all errors
- Insist on reporting a fixed set of potential errors
 - Many options and control flags

Notkin (c) 1997, 1998

49

Ex: Definition before Use

- Sample code
 - if (setVal(n, &buffer)) ...
- **Must** buffer be defined before calling setVal?
 - **Yes:** bool setVal(int d, char *val);
 - **No:** bool setVal(int d, out char *val);
- **Is** buffer defined afterwards?
 - **Yes:** bool setVal(...); {modifies *val;}
- **Maybe:** bool setVal(...); {modifies nothing;}
- **NO!** bool setVal(...); {ensures
trashed(val);}

Notkin (c) 1997, 1998

50

More Accurate Checks

- Conventional lint tools report
 - Too many spurious errors
 - Too few actual errors
- **Because**
 - Code does not reveal the programmer's intent
 - Fast checks require simplifying assumptions
- Specifications give good simplifying assumptions

Notkin (c) 1997, 1998

51

Abstraction Boundaries

- Client code should rely only on specifications
- Types can be specified as abstract
 - **immutable type** date;
 - date nextDay(date d); { }
 - **mutable type** set;
 - void merge(set s, set t); {modifies s;}
- LCLint detects
 - Inappropriate access to representation
 - Including use of ==
 - Inappropriate choice of representation
 - E.g., for meaning of = (sharing)

Notkin (c) 1997, 1998

52

Checking Abstract Types

- **Specification:** set.lcl contains the single line
 - **mutable type** set;
- **Client code**
 - #include "set.h"
 - bool f(set s, set t) {
if (s->size > 0) return (s == t);
...}
- **> lclint set client.c**
 - client.c:4,7: Arrow access field of abstract type (set): s->size
 - client.c:5,13: Operands of == are abstract type (set): s == t

Notkin (c) 1997, 1998

53

Checking Side Effects

- **Specification:**

```
void set_insert (set s, int e)
{ modifies s; }
void set_union(set s, set t)
{ modifies s; }
```
- **Code (in set.c):**

```
void set_union (set s, set t) {
int i;
for (i = 0; i < s->size; i++)
set_insert(t, s->elements[i]);
}
```
- **Message:**
 - set.c:35, 27: Called procedure set_insert may modify t:
 - set_insert(t, s->elements[i])

Notkin (c) 1997, 1998

54

Checking Use of Memory

- Specifications
 - only char *gname;
 - . . .
 - void setName (temp char *pname) char *gname;
- Code
 - void setName (char *pname) {
 gname = pname;
}
- LCLint error messages
 - sample.c:2:3: Only storage gname not released before assignment: gname = pname
 - sample.c:2:3: Temp storage assigned to only: gname = pname

Notkin (c) 1997, 1998

55

If C Were Better...

- Would LCLint still help?
- Yes, because specifications
 - contain information not in code
 - contain information that is hard to infer from code
 - are usable with legacy code, existing compilers
 - can be written faster than languages can be changed
 - are important even with better languages

Notkin (c) 1997, 1998

56

Experience with LCLint

- Reliable and efficient
 - Runs at compiler speed
- Used on both new and legacy code
 - 1,000-200,000 line programs
 - Over 500 users have sent e-mail to MIT
- Tested with varying amounts of specification
 - Lots to almost none
 - LCLint approximates missing specifications
- Results encouraging

Notkin (c) 1997, 1998

57

Understanding Legacy Code

- Analyzed interpreter (quake) built at DEC SRC
- Discovered latent bugs (ordinary lint can do this)
- Discovered programming conventions
 - Documented use of built-in types (int, char, bool)
 - Identified (and repaired) (nearly) abstract types
- Documented action of procedures
 - Use of global information, side-effects
- Enhanced documentation a common thread
 - Easier to read and write because formulaic
 - More trustworthy because checked

Notkin (c) 1997, 1998

58

Fundamental benefit

- Partial specifications
- Low entry cost
- You get what you pay for (or maybe a bit more)

Notkin (c) 1997, 1998

59

Purify

- Dynamic (commercial) tool for detecting memory leaks and access errors
 - <http://www.rational.com/products/purify/>
 - In some sense, a dynamic analog to the memory checking aspects of LCLint
- Trapping every memory access would be overly expensive
 - Purify inserts function calls before loads and stores to maintain a table that holds a 2-bit state for each byte in the heap and stack
 - Requires working with malloc/free, too

Notkin (c) 1997, 1998

60

Memory State Transitions

- Writing to memory with any bytes that are unwriteable prints a diagnostic
- Same with reading unreadable bytes



Notkin (c) 1997, 1998

61

Other violations

- **Array Bound Violations**
 - Allocate a "red-zone" around malloc blocks, recording them as unwriteable and unreadable
- **Uninitialized variables**
 - Initialize them to allocated-but-uninitialized state
- **Accessing freed memory**
 - Do not reallocate memory until it has aged
 - Aging can be specified by the user in terms of number of calls to free

Notkin (c) 1997, 1998

62

Overhead

- 2 bits per state; about 25% memory overhead during development
- Run-time no worse than 5.5 times optimized C code (and usually no worse than compiling with debugging on)

Notkin (c) 1997, 1998

63

Next week

- Type inference (Lackwit)
- Program representations for tool support

Notkin (c) 1997, 1998

64