Software evolution then and now: The research impact of David Notkin

Michael Ernst
UW CSE
Notkinfest
February 1, 2013
Software engineering

A disciplined approach to the design, development, and **maintenance** of software

Software evolution: understanding, managing, and reducing the **cost of change**
Software engineering has been wildly successful

- Software pervades every aspect of our lives
  If you don’t like it, try living a day without it!
- Software provides the value in our gadgets
- Huge economic impact
- Increasingly sophisticated functionality is possible
  500 lines of code ⇒ 50,000,000 lines of code
  We always want to do more!

David Notkin has contributed to the success of software engineering
Software engineering is challenging

Mathematics:
• Modeling
• Analysis

A non-ideal component in the system: People!

Both aspects are intellectually deep

This talk:
• A few challenges
• How the field (and David Notkin) have addressed them
David Notkin’s contributions to software engineering

• Refactoring
• Implicit invocation
• Environments and integration
• Mediators
• Parallel programming
• Software reflexion models
• Model checking
• Specification mining
• Architectural conformance
• Testing
• Detecting behavioral anomalies
• Mining software repositories
• Speculative analysis
Outline of the talk

A few specific projects:
• Refactoring
• Architecture
• Code clones
• Model checking
• Specification mining
• Structure-oriented computing

For each project:
• The topic
• How it was viewed, Before David
• Research contribution
• How it is viewed, After David
Refactoring

• Semantics-preserving program transformation
  – Not fixing a bug nor adding a feature
• Improves “ilities”
  – Testability
  – Maintainability
Before David

Refactoring did not exist
The design of an extended system is different than if it had been developed from scratch.

– David Notkin
(paraphrase)

All possible designs of a program can be reached by algebraic transformation

Key observations

Extension and Software Development*

David Notkin
William G. Griswold
Department of Computer Science, FR 35
University of Washington, Seattle, WA 98195

Abstract

Enhancement is the most costly phase of the software development lifecycle. By developing an extension, the mechanism consists of an arbitrator, which maps names of functions to implementations, a dynamic linker, which allows for incremental incorporation of program parts at run-time, and translation support that automatically translates one solution to another.

The extension encourages an incremental definition of functions that directly addresses it portion of the enhancement process.

The laws given in this article can be regarded as a completely formal algebraic specification of our programming language. The laws are strong enough to permit each program, not involving recursion, to be reduced to a normal form. A natural partial ordering is defined for normal forms, and programs can be identified with ideals in this partial ordering. The ideal con-
A meaning-preserving restructuring tool

• Concept:
  – The programmer starts the change
  – The tool completes it

• Represent program semantics with the Program Dependence Graph (PDG)
Contributions and impact

• Meaning-preserving restructuring (= “refactoring”)
  – A confluence of needs, ideas, and technologies

• Every IDE supports basic refactorings

• Application of PDG, with transformation rules
  – Aids reasoning, efficient transformation tool

• Visualizations
Software architecture

- Architecture expresses the high-level design of software: components and connections

Parallel application

<table>
<thead>
<tr>
<th></th>
<th>GM</th>
<th>FM</th>
<th>TCP / IP</th>
<th>Myrinet</th>
<th>Fast Ethernet</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPICH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Legend:
- Component
- Connector
- Communication Link
Architectural conformance

Does the code match the intended design?

• Example: communication integrity

In 2002, key challenges remained unsolved:

• Bridging the abstraction gap between architecture and code
• Guarding against unintentional architectural violations
• Managing features of modern object-oriented languages
public component class Transform {
    public port in {
        provides void draw(Shape s);
    }
    public port out {
        requires void draw(Shape s);
    }
    void draw(Shape s) {
        currentTransform.apply(s);
        out.draw(s);
    }
}...

public component class GraphicsPipeline {
    protected Generate generate = ... ;
    protected Transform transform = ... ;
    protected Rasterize rasterize = ... ;
    connect generate.out, transform.in;
    connect transform.out, rasterize.in;
}

Typing rules enforce the architecture

with Jonathan Aldrich, Craig Chambers, and others

Architectural interfaces

Connections link components

Ordinary Java code
Case study

Architectural visualization based on ArchJava

Architectural drawing by developer
Contributions

• Connection between architecture and code
• Real architectures differ from conceptual models
• Machine-enforced architectures are possible
• ArchJava reveals coding problems
Code clones

Code clone = duplicate source code
usually due to cut-and-paste

Problems:
• code bulk affects comprehension
• inconsistent updating
  – fix bug in one place, same bug remains elsewhere
2004 view of clones

• Clones are a serious problem

• Solution:
  – Prevent creation
  – Proactively refactor to eliminate clones
  – Avoid the duplication & potential for error

Might there be good reasons to have clones?
Empirical study of code clone genealogies

• Most clones are short-lived, diverging clones
• Most long-lived clones cannot be eliminated using standard refactoring techniques

Conclusion:
Proactive refactoring can be counterproductive
**Code Matching for Longitudinal Program Analysis**

“How do we automatically match corresponding code elements between two program versions?”

- Automatically reconstruct the history of code clones from a source code repository
- Indicate if change is consistent or inconsistent
- Change-rules can concisely describe a set of related API-level changes.
  - Can be automatically inferred
Concise description of changes:

- “Moved 3 classes between directories”
- Previous approach: 10,000 lines of diff output

“Replace all calls to SQL.exec with SafeSQL.exec”
deprecatedCalls(m, "SQL.exec") => addedCalls(m, "SafeSQL.exec")
Impact

• David proposed analyzing multiple versions of a program rather than the single latest version
• Promoted research in mining software repositories
• Helped direct the field toward relevant problems
Model checking

- Model-checking = brute-force search
- Can you get to Jail without passing Go?
  - Try every possibility to find out
Traditional model checking

• In late 1990s, model-checking was applied to:
  – Hardware (finite-state)
  – Low-level cache coherency protocols

• Software is too hard:
  – Greater complexity
  – Infinite state
Model checking of software specifications

- Scaled symbolic model checking to MITRE’s TCAS collision avoidance specification
- Found errors:
  - Non-determinism
  - Inconsistency in definition by cases
  - Inconsistent outputs (“climb”, and “go to lower altitude”)
  - Bad outputs (“descend” when at low altitude)
- Specifications from Boeing
  - Good results

with William Chan, Paul Beame, Richard Anderson, and others
Technical challenges

• Non-linear constraints
  – Combine constraint solving and symbolic model checking to solve non-linear constraints
  – Represent each constraint as a BDD variable
    • Example constraint: $\forall i: x_i = x'_i$

• Large & complex control
  – Model-checking optimizations: eliminate infeasible constraints
  – Intersect constraints with feasible set
  – Filter constraints: remove infeasible paths from BDD on the fly
Impact on model checking

• The largest software model-checking at the time
• By the early 2000s, model-checking became an accepted software verification technique
  – David’s work in the 1990s helped lead to this shift

• Today, model checking is one of the most important program analyses
  – Whole conferences are devoted to it
  – Used daily at Microsoft & elsewhere
Program specification

A specification is a formal description of behavior

Program:

// Sum array $b$ of length $n$ into variable $s$.
$i := 0; s := 0;$
while $i \neq n$ do
  $\{ s := s + b[i]; \ i := i+1 \}$

Specification:

Precondition: $n \geq 0$
Postcondition: $s = (\sum j: 0 \leq j < n : b[j])$
Loop invariant: $0 \leq i \leq n$ and $s = (\sum j: 0 \leq j < i : b[j])$
Traditional view of specifications

Around 1997, many papers start:

“Assuming a specification exists, we give a technique for
{ testing, verification, refactoring, modeling, upgrades, ...}”

Few specifications exist
- Programmers do not write specifications
- No practical way to recover invariants from existing programs

Researchers were divided into two camps:
- Verification and static analysis
- Testing and dynamic analysis
The Daikon invariant detector

1. Run the program
2. Examine the computed values
3. Use machine learning to generalize

Output:
- \( x > \text{abs}(y) \)
- \( x == 16*y + 4*z + 3 \)
- For every node \( n \), \( n.\text{prev}.\text{next} == n \)
- Array \( a \) contains no duplicates
- Graph \( g \) is acyclic
Contributions

• Mined specifications are **unsound, incomplete, ... and useful!**
  – Helps programmers, helps other analyses
  – **Accurate** in practice

• Combine static and dynamic analyses
  – Look across area boundaries

• Previous AI techniques were not applicable
  – Now, machine learning is commonplace
Dissertation: Interactive Structure-Oriented Computing
Writing the wrong program

Problems:

• Broken programs due to
  – dangling else
  – mistyped keywords
  – ...

• Inability to perform IDE analyses on broken programs

```plaintext
if (door-is-open)
  if (resident_is_visible)
    say "Hello!"
else
  wring doorbell
```
Structure-Oriented Computing or Syntax-Directed Editing

Idea: Make these mistakes impossible

- Programmer does not manipulate raw text
- Programmer manipulates trees and graphs
- Program remains well-formed

Analogy: problems in writing English text
Using a structured editor

The cursor, representing the current node of the tree, is displayed in dashed boxes. Unfilled-in nodes, called meta-nodes, are displayed as $CLASS where CLASS defines the language constructs that may replace the node.
“the remainder of my thesis does not directly champion the benefits of structure-oriented editing. Instead, I assume that structure-oriented editors are ... interesting [and] deserve further study.”
What we can learn

• Goal is now embraced by IDEs:
  – They always maintain a parsed structure
  – This enables powerful analyses and transformations

• Aim high

• Don’t give up after a failure
  – There may be other ways to attack the problem; or,
  – Find new problems and collaborators

• Judge people by their merits, not one result
David Notkin’s impact

• Significant technical success
  – We’ve seen a small slice of his projects
  – Contributed to the success of software engineering

• Another big effect: mentoring
  – See the amazing productivity of his students
    and others he affected