Research in Program Comprehension

Susan Elliott Sim
ses@ics.uci.edu
November 7, 2006

Overview

- Program Comprehension
- People Comprehending Programs
  - Strategies
  - Models
  - Factors
- Tools to Help People Comprehend Programs
  - Generic Pipeline

Program Comprehension

- Help programmers understand code more quickly
  - Improve general understanding
  - To complete a task
- Examples of program comprehension tools?

- Related fields: reverse engineering, source code analysis, software visualization

People Comprehending Programs

- Models
- Strategies
- Factors

Studies of Program Comprehension

- Cognitive models
  - A single programmer working alone to understand a program
    - Mental model is the representation created by a programmer during the process of understanding
- Social and distributed cognition
  - A programmer working as part of a team to develop or maintain a software system
- Frameworks for designing program comprehension tools

Cognitive Models

- Bottom-up
- Top-down
- Hybrid
  - Systematic and As-Needed
  - Knowledge-based
  - Integrated

Bottom-Up Models

- Build larger abstractions from details
  - *Chunking* is the process of building larger units
- Allows programmer to hold more information in short-term memory
  - Miller's magic number: 7 ± 2
- Larger chunks corresponds to more meaningful understanding of larger parts of the program

B. Shneiderman and R. Mayer, "Syntactic/semantic Interactions in Programmer Behavior: A Model and Experimental Results," *International Journal of Computer and Information Sciences*, vol. 8, no. 3, pp. 219-238, 1979.

  - Lexical -> Syntactic -> Semantic

N. Pennington, "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs," Cognitive Psychology, vol. 19, pp. 295-341, 1987.

  - Program model -> situation model

Top-Down Models

- Features in the program are mapped onto expectations
  - Hypotheses are formed at higher levels and direct investigation of details
  - Expectations are called schemas or plans

Ruven Brooks, "Towards a Theory of the Cognitive Processes in Computer Programming," Interntional Journlal of Man-Machine Studies, vol. 18, pp. 543-554, 1983.

  - Programmers look for *beacons* in the code

E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," IEEE Transactions on Software Engineering, vol. SE-10, no. 5, pp. 595-609, September, 1984.

  - Two types of programming knowledge: programming plans and rules of programming discourse

Hybrid Models

- Programmers don't use just one approach all the time.
  - Can choose an appropriate one for the task
  - Switch between them during a single task

Systematic and As-Needed Models

- Systematic = read the code methodically, tracing control flow and data flow
  - Provides static (structural) and causal information
- As-needed = focus only on the code relevant to immediate task
  - Also called Just In Time Comprehension
  - Provides only static information (weaker mental model)

Merging Systematic and As-Needed

- The two strategies were later merged into a single model

E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert, "Designing Documentation to Compensate for Delocalized Plans," Communications of the ACM, vol. 31, no. 11, pp. 1259-1267, 1988.

- Micro-strategies
  - Inquiry episodes: read, question, conjecture, search
- Macro-strategies
  - Systematic
  - As-Needed

Knowledge-Based Models

- Programmers are *opportunistic processors*; they use what is available to get the job done.

J. Koenemann and S.P. Robertson, "Expert Problem Solving Strategies for Program Comprehension," presented at Human Factors in Computer Systems, Conference Proceedings of CHI'91, New Orleans, LA, pp. 125-130, April.

- Comprehension is a goal-oriented, hypothesis-driven, problem solving process

S. Letovsky, "Cognitive Processes in Program Comprehension," presented at Empirical Studies of Programmers, pp. 58-79.

- Inquiries = asking questions, conjecturing answers, verifying answers
- Three Components in Model: Knowledge Base, Assimilation Process, Mental Model

Integrated

- The Integrated Metamodel

A. von Mayrhauser and A.M. Vans, “Program Comprehension During Software Maintenance,” IEEE Computer, pp. 44-55, August, 1995.

Sources of Variation

- Aside from the issue of how comprehension occurs, researchers agree that programmer performance is affected by a number of factors

- Maintainer Characteristics
- Program Characteristics
- Task Characteristics

Maintainer Characteristics

- Familiarity with code base
- Application domain knowledge
- Programming language knowledge
- Programming expertise
- Tool expertise
- Individual differences

Program Characteristics

- Application domain
- Programming domain
- Quality of problem to be understood
- Program size and complexity
- Availability of documentation

Task Characteristics

- Task type
  - Experimental: recall, modification
  - Perfective, corrective, adaptive, reuse, code leverage
- Task size and complexity
- Time constraints
- Environmental factors

Social and Distributed Cognition

- On typical industrial projects, software developers:
  - Work in teams
  - Have access to documentation
  - Can ask people questions
  - Can surf the web for answers
  - Can go home and think about the problem
  - Learn!
- Work on cognitive models don't really address program comprehension as a open system

Distributed Cognition

- Cognition is an interactive process between a person and the environment
  - Plans are incomplete
  - External world is used to help cognitive process
  - Examples: wayfinding, GUI desktops, to do lists
  - See work in HCI by Hutchins and by Suchman

Andrew Walenstein, Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework. Ph.D. Thesis, School of Computing Science, Simon Fraser University, 2002.

- Developed a distributed cognition model for program comprehension
- Created a framework for designing and evaluating tools

Teams

Susan Elliott Sim and Richard C. Holt, "The Ramp-Up Problem in Software Projects: A Case Study of How Software Immigrants Naturalize," presented at Twentieth International Conference on Software Engineering, Kyoto, Japan, pp. 361-370, 19-25 April, 1998.

- On large software projects, it takes many months to become a contributing team member

Audris Mockus and Jim D. Herbsleb, "Expertise Browser: A Quantitative Approach to Identifying Expertise," presented at Twenty-fourth International Conference on Software Engineering (ICSE), Orlando, FL, pp. 503-512, 19-25 May 2002.

- A significant part of learning about the system is identifying expertise and ownership

J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An Examination of Software Engineering Work Practices," presented at Centre for Advanced Studies Conference (CASCON'97), Toronto, Canada, pp. 209-223.

- While comprehension is a critical part of programming, software engineers actually spend a minority of their time coding

Work Practices

- An approach more than a framework
- Study work practices in order to tailoring tools (or methods) to local organization and environment
- Example: source code searching with grep
  - Why do people use it?
  - What does it do well? (Keep these features)
  - What does it not do well? (Ideas for improvements)
  - Result: SEE (Software Exploration Environment)
  - Does the resulting tool fit with work practices?

Janice Singer and Timothy Lethbridge, "Studying Work Practices to Assist Tool Design in Software Engineering," presented at International Workshop on Program Comprehension (IWPC'98), Ischia, Italy, pp. 173-179.

Research Opportunities

- New research is needed on larger systems, with modern architectures, involving multiple programmers
  - Past research conducted on small programs
    - A few hundred or thousands of lines of code
  - Single programmer
- Past studies did not use right mix of psychology and software engineering
  - Example: use of recognition and recall as dependent variable
  - Example: Integrated metamodel

Recent Research

- Navigation through source code
  - Information foraging

Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung, "An Exploratory Study of How Developerrs Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks" IEEE Transactions on Software Engineering, Vol. 32, No. 12, December, 2006.

- Developers spent on average 35% of their time performing the mechanics of finding information

  - Graph-based model of traversal

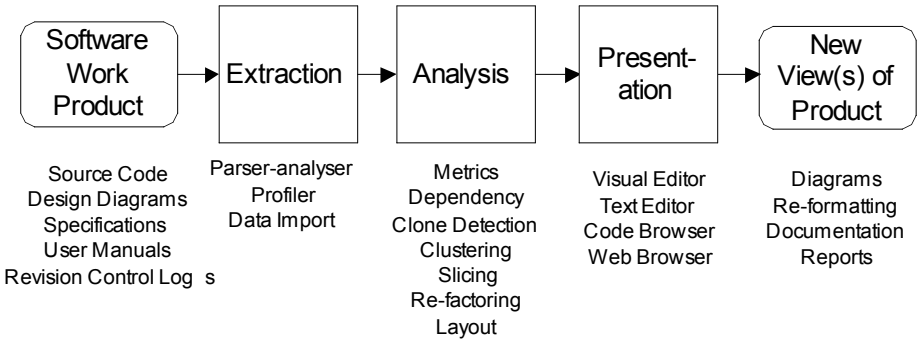
Susan Elliott Sim, Sukanya Ratanotayanon, and Leyna Cotran, "Using Graphs to Characterize How Developers Navigate During Program Comprehension" *in submission to* International Conference on Software Engineering, 2008.

- In a web application, the layered architecture graph is an effective model for developer's navigation behavior
- Developers who successfully completed a perfective maintenance task made fewer traversals and were more likely to make traversals of distance 1

Tools to Help People Comprehend

- Generic Pipeline
- Environments

Generic Pipeline



Analyses in Reverse Engineering

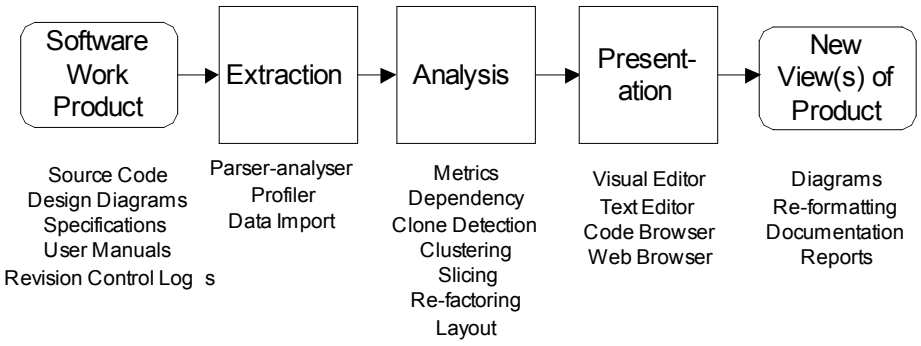
- Assumption: Legacy code

- Redocumentation
- Design Recovery
- Restructuring
- Reengineering

Things to Notice

- Architecture influenced by UNIX and compilers
- Graph-based representation
  - Take a graph theory course
- Tools, analyses, and representations that are useful for other areas of software engineering
- Each element in the diagram (including the arrows) is a research problem

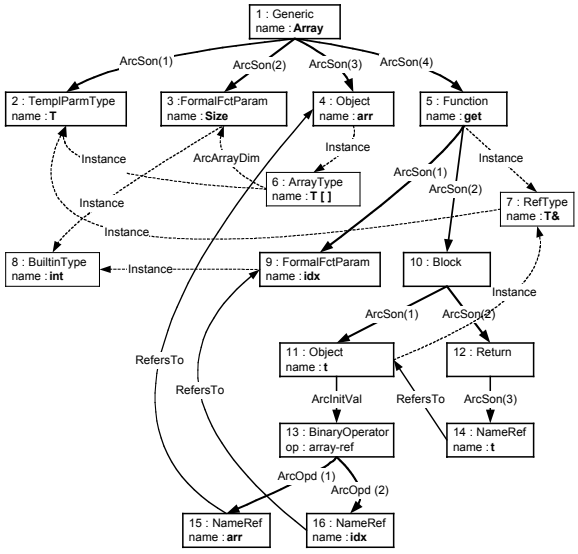
Generic Pipeline



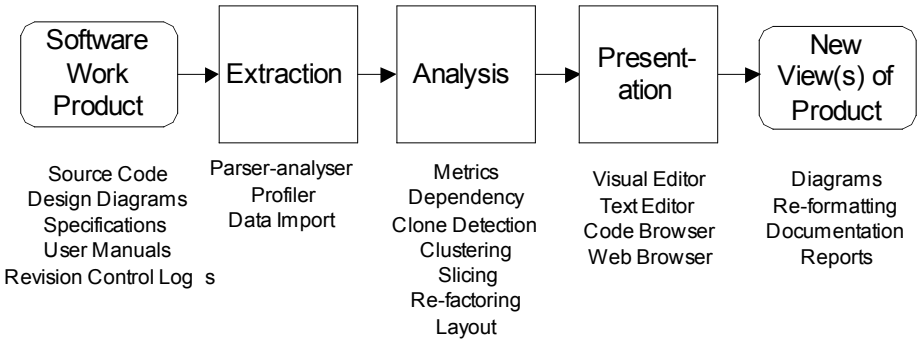
Fact Extraction for C++

```
 1  template <class T, int Size>
 2  class Array
 3  {
 4    T arr[Size];
 5  public:
 6    virtual const T& get(int idx)
 7    {
 8      T& t = arr[idx];
 9      return t;
10    }
11    /* set()...*/
12  };
```


Decorated AST



Generic Pipeline



Analysis Tools

- Clustering
- Program summaries, fingerprints
- Design pattern detection
- Slicing

Slicing

- Program analysis technique for reducing a program
  - Eliminates all lines that are not currently of interest
- Choose a variable, choose a line of code, slice forward (or backward)

Backward Slice

```
void main () {
    int i = 1;
    int sum = 0;
    while (i<11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("sum = %d\n", sum);
    printf("i = %d\n", i);
}
static int add(int a, int b) {
   return(a+b);
}
```

Backward slice from printf("i = %d\n", i);

Backward Slice

```
void main () {
    int i = 1;
    int sum = 0;
    while (i<11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("sum = %d\n", sum);
    printf("i = %d\n", i);
}
static int add(int a, int b) {
   return(a+b);
}
```

Backward slice from printf("i = %d\n", i);

Forward Slice

```
void main() {
    int i = 1;
    int sum = 0;
    while (i<11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("sum = %d\n",sum);
    printf("i = %d\n", i);
}
static int add(int a, int b){
   return(a+b);
}
```

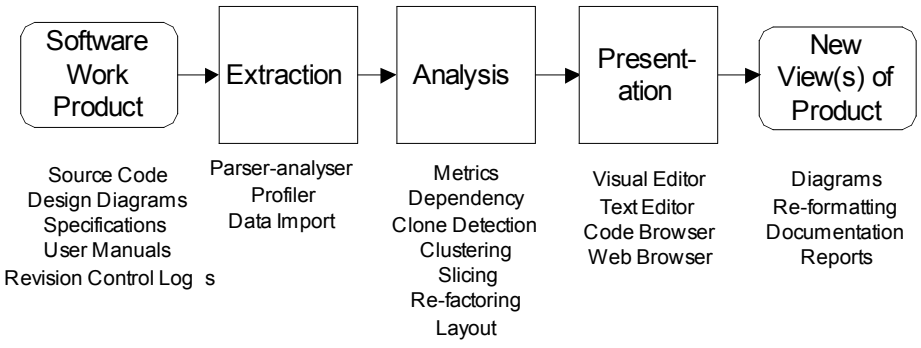
Forward slice from sum = 0;

Forward Slice

```
void main() {
    int i = 1;
    int sum = 0;
    while (i<11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("sum = %d\n",sum);
    printf("i = %d\n", i);
}
static int add(int a, int b){
   return(a+b);
}
```

Forward slice from sum = 0;

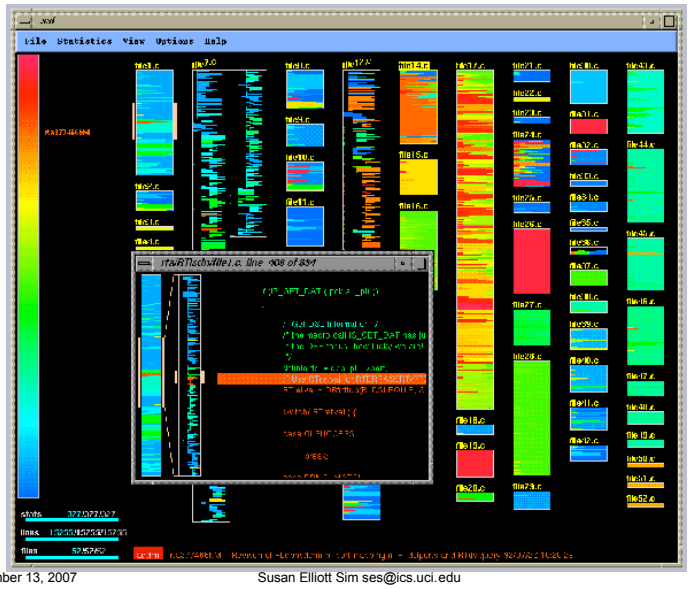
Generic Pipeline



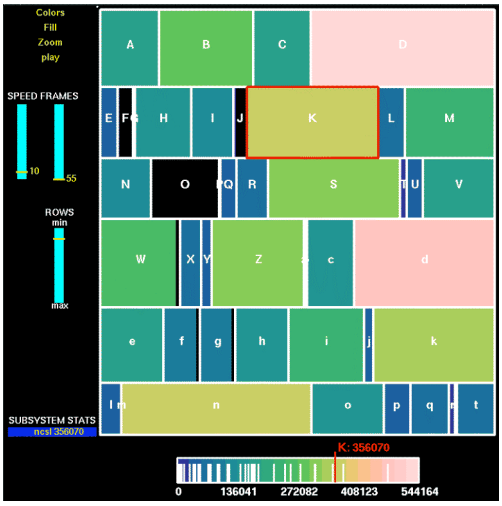
Presentation Tools

- SNiFF+
- Seesoft™
- SHriMP/Creole
- PBS (Portable Bookshelf)

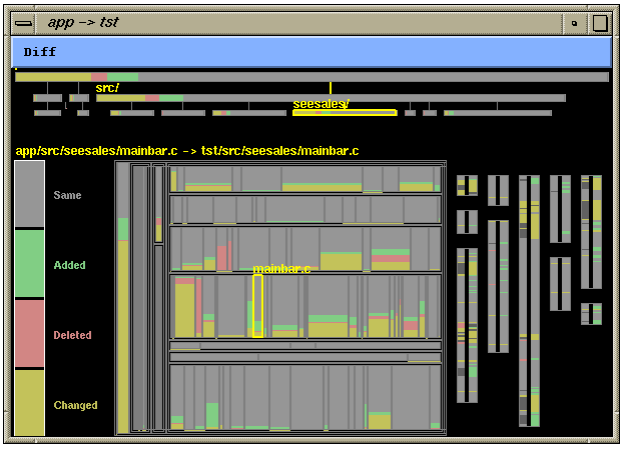
Seesoft™



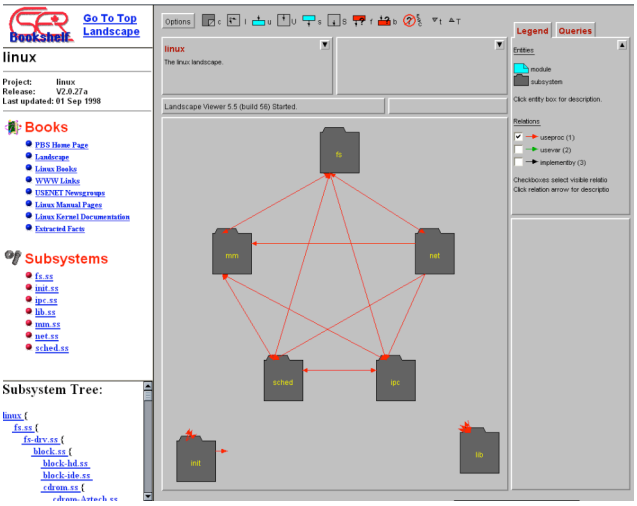
Seesoft™



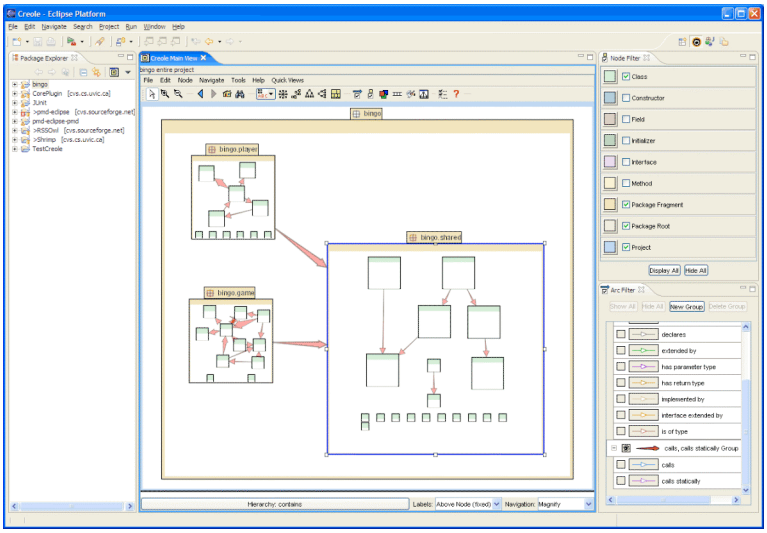
Seesoft™



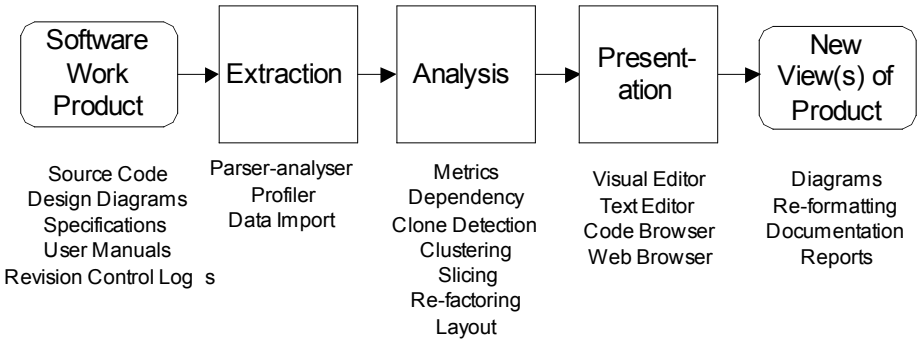
PBS


Go To Top
Landscape
linux
Project: linux
Release: V2.0.27a
Last updated: 01 Sep 1998
Books
PBS Home Page
Landscape
Linux Books
WWW Links
USENET Newsgroups
Linux Manual Pages
Linux Kernel Documentation
Extracted Facts
Subsystems
fs.ss
init.ss
ipc.ss
lib.ss
mm.ss
net.ss
sched.ss
Subsystem Tree:
linux {
fs.ss {
fs-drv.ss {
block.ss {
block-hd.ss
block-ide.ss
cdrom.ss {
Options
Legend
Queries
Landscape Viewer 5.5 (build 56) Started.
fs
mm
net
sched
ipc
init
lib

Creole


Creole - Eclipse Platform
Package Explorer
Creole Main View
Node Filter
Arc Filter
bingo
bingo.player
bingo.shires
bingo.game

Generic Pipeline



Standard Exchange Format

- GXL (Graph eXchange Language)
  - An XML sub-language for exchanging graphs
    - Joint work by Sim, Andreas Winter, Ric Holt, Andy Schürr
  - Mechanism for data interoperability between reverse engineering, reengineering, and graph transformation tools
  - In use by ~40 research groups around the world
  - Similar to MDA currently promoted by OMG

GXL Features

- Labeled, typed graph model
  - Nodes, (directed) edges, hyperedges, attributes
  - Unique identifiers on nodes, optional identifiers on others
- Transmit schema along with data
  - Use single DTD for both
- Provides a common syntax for data interchange, so problem shifts to schema interchange

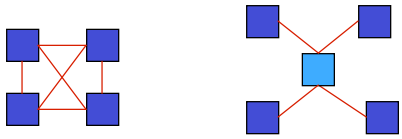
```
<?xml version="1.0"?>
<!DOCTYPE gxl SYSTEM "gxl.dtd">
<gxl>

<graph id ="schema">

...

</graph>

<graph id ="instance">

...

</graph>



</gxl>
```


Schemas

- Converting from one syntax to another is relatively easy
- Mapping between schemas is the most difficult aspect of conversion
  - GXL helps by making them explicit
  - More tools are needed to support schema exchange
- Related to research into model management in databases
  - Idea: translating data is isomorphic to translating between their respective schemas

Reference Schemas

- A canonical schema for a particular domain or task
  - Examples: graph drawing, C++ fact extraction
- Contains an agreed-upon set of modeling entities and relations
  - Instead of writing mappings between every local schema, write a mapping between local and reference schema



Research Opportunities

- Programming Environments
  - UNIX pipelines vs. Eclipse
  - New tools, new opportunities
- Languages
  - Previous: COBOL, PL/I
  - Current emphasis: Java
  - Future? Scripting languages, JSP, Visual Basic, C#
- Systems
  - Web applications, plug-ins, games
- Mining Software Repositories
- Applications of Reverse Engineering