Welcome to SENG 371 Software Evolution Spring 2013

A Core Course of the BSEng Program

Hausi A. Müller, PhD PEng
Professor, Department of Computer Science
Associate Dean Research, Faculty of Engineering
University of Victoria

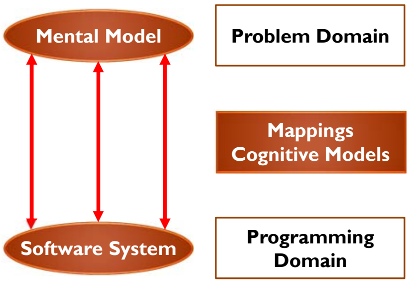
Announcements

- Lab attendance
  - Has been a problem as of late — needs to change
  - Several questions on labs on final exam
- Final exam
  - Sat, April 13 — 7:00 -10:00 pm
- Teaching evaluations
  - Next week
- Marking
  - A2 should be graded this week
  - Midterm and A1 graded
  - Marks posted
- Course website
  - http://www.engr.uvic.ca/~seng371
  - Lecture notes posted
  - Lab slides and activities are posted
- Assignment 3
  - Due Thu, April 4
  - Part I — Define software evolution terms
  - Part II — Investigate two AntiPatterns — Vendor-Lock-In — Analysis Paralysis
  - Part III — Refactoring in IBM Eclipse and MS Visual Studio and Blob AntiPattern
  - Cite your sources
  - Submit by e-mail to seng371@uvic.ca

Reading Assignment

- Murphy, Notkin, Lan: An empirical study of static call graph extractors, *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7(2):158-191 (1998)
  - http://dl.acm.org/citation.cfm?id=279314
- Müller, Jahnke, Smith, Storey, Tilley, Wong: Reverse Engineering: A Roadmap, in *The Future of Software Engineering*, pp. 47-60 (2000)
  - http://dl.acm.org/citation.cfm?id=336526
- Storey: Theories, tools and research methods in program comprehension: past, present and future, *Software Quality Journal* 14:187-208 (2006)
  - http://webhome.cs.uvic.ca/~chisel/pubs/storey-pc-journal.pdf
- Brown, Malveau, McCormick III, Mowbray: *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis,* John Wiley (1998)
- AntiPatterns Tutorial and Website
  - http://www.antipatterns.com/briefing/index.htm
  - http://www.antipatterns.com

Developing mental models using cognitive models

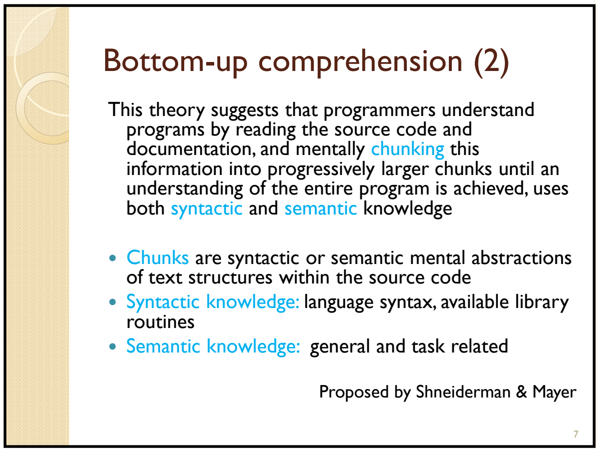


Cognitive models of program comprehension

- Bottom-up comprehension
- Top-down comprehension
- Knowledge based understanding model
- Systematic and as-needed strategies
- Integrated meta-model of program comprehension

Bottom-up comprehension (1)

- Starts understanding from the source code, constructing higher level abstractions using chunking and concept assignment
  - Shneiderman and Mayer 79
  - Pennington 87
  - Biggerstaff, et al. 93
- Chunking creates new higher level abstractions from lower level structures
- When higher level structures are recognized, they replace more detailed lower level ones
- This helps to overcome the limitations of the human memory when confronted with too many pieces of information

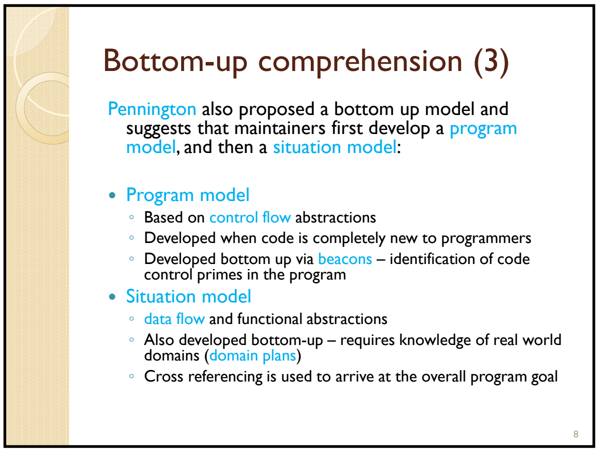


Bottom-up comprehension (2)

This theory suggests that programmers understand programs by reading the source code and documentation, and mentally chunking this information into progressively larger chunks until an understanding of the entire program is achieved, uses both syntactic and semantic knowledge

- Chunks are syntactic or semantic mental abstractions of text structures within the source code
- Syntactic knowledge: language syntax, available library routines
- Semantic knowledge: general and task related

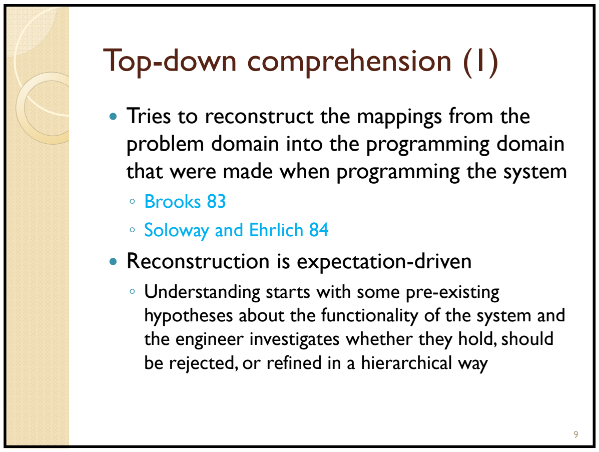
Proposed by Shneiderman & Mayer



Bottom-up comprehension (3)

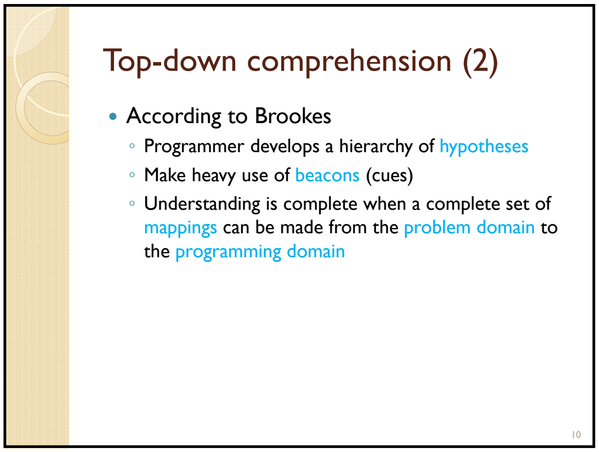
Pennington also proposed a bottom up model and suggests that maintainers first develop a program model, and then a situation model:

- Program model
  - Based on control flow abstractions
  - Developed when code is completely new to programmers
  - Developed bottom up via beacons – identification of code control primes in the program
- Situation model
  - data flow and functional abstractions
  - Also developed bottom-up – requires knowledge of real world domains (domain plans)
  - Cross referencing is used to arrive at the overall program goal



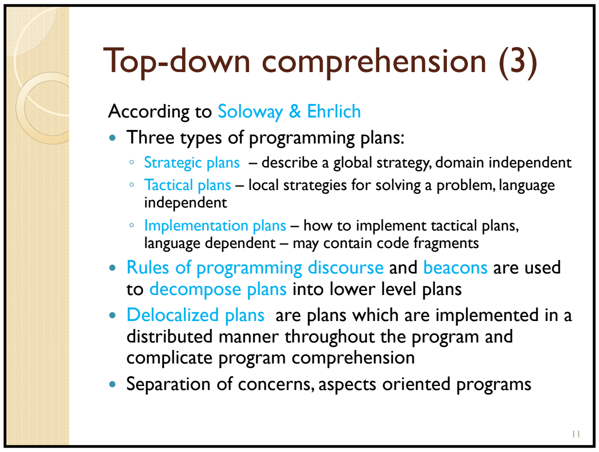
Top-down comprehension (1)

- Tries to reconstruct the mappings from the problem domain into the programming domain that were made when programming the system
  - Brooks 83
  - Soloway and Ehrlich 84
- Reconstruction is expectation-driven
  - Understanding starts with some pre-existing hypotheses about the functionality of the system and the engineer investigates whether they hold, should be rejected, or refined in a hierarchical way



Top-down comprehension (2)

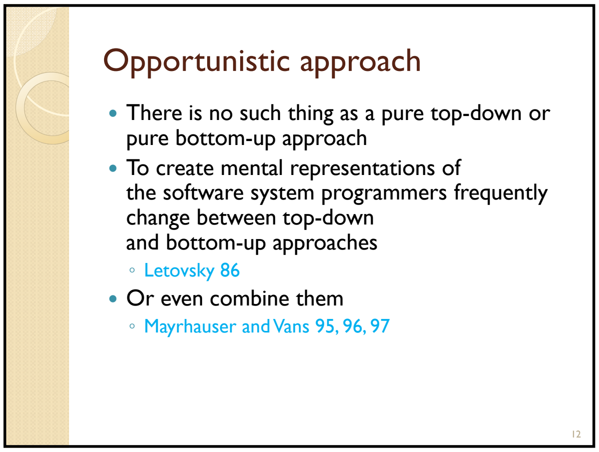
- According to Brookes
  - Programmer develops a hierarchy of hypotheses
  - Make heavy use of beacons (cues)
  - Understanding is complete when a complete set of mappings can be made from the problem domain to the programming domain



Top-down comprehension (3)

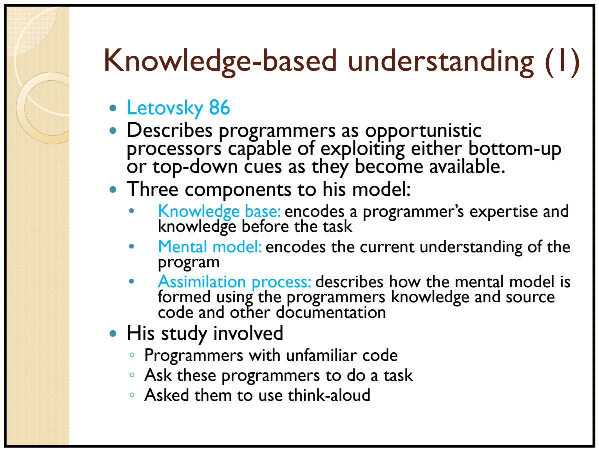
According to Soloway & Ehrlich

- Three types of programming plans:
  - Strategic plans – describe a global strategy, domain independent
  - Tactical plans – local strategies for solving a problem, language independent
  - Implementation plans – how to implement tactical plans, language dependent – may contain code fragments
- Rules of programming discourse and beacons are used to decompose plans into lower level plans
- Delocalized plans are plans which are implemented in a distributed manner throughout the program and complicate program comprehension
- Separation of concerns, aspects oriented programs



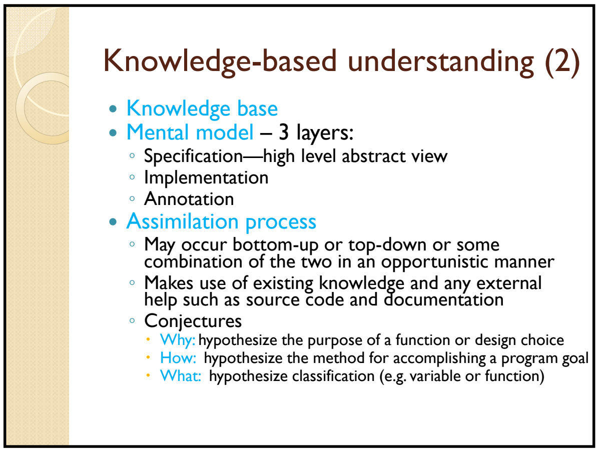
Opportunistic approach

- There is no such thing as a pure top-down or pure bottom-up approach
- To create mental representations of the software system programmers frequently change between top-down and bottom-up approaches
  - Letovsky 86
- Or even combine them
  - Mayrhauser and Vans 95, 96, 97



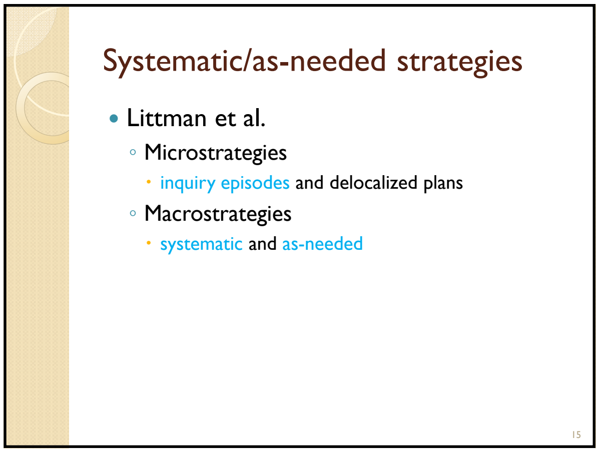
Knowledge-based understanding (1)

- Letovsky 86
- Describes programmers as opportunistic processors capable of exploiting either bottom-up or top-down cues as they become available.
- Three components to his model:
  - Knowledge base: encodes a programmer's expertise and knowledge before the task
  - Mental model: encodes the current understanding of the program
  - Assimilation process: describes how the mental model is formed using the programmers knowledge and source code and other documentation
- His study involved
  - Programmers with unfamiliar code
  - Ask these programmers to do a task
  - Asked them to use think-aloud



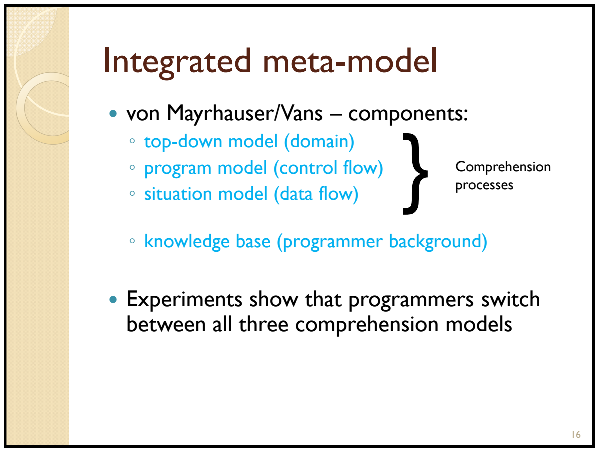
Knowledge-based understanding (2)

- Knowledge base
- Mental model – 3 layers:
  - Specification—high level abstract view
  - Implementation
  - Annotation
- Assimilation process
  - May occur bottom-up or top-down or some combination of the two in an opportunistic manner
  - Makes use of existing knowledge and any external help such as source code and documentation
  - Conjectures
    - Why: hypothesize the purpose of a function or design choice
    - How: hypothesize the method for accomplishing a program goal
    - What: hypothesize classification (e.g. variable or function)



Systematic/as-needed strategies

- Littman et al.
  - Microstrategies
    - inquiry episodes and delocalized plans
  - Macrostrategies
    - systematic and as-needed

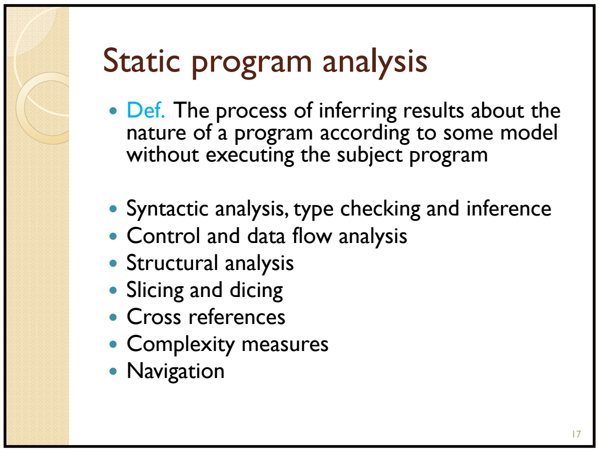


Integrated meta-model

- von Mayrhauser/Vans – components:
  - top-down model (domain)
  - program model (control flow)
  - situation model (data flow)

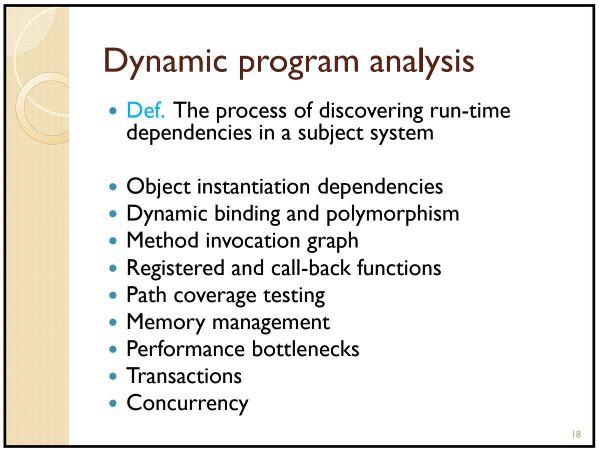
  } Comprehension processes

  - knowledge base (programmer background)
- Experiments show that programmers switch between all three comprehension models



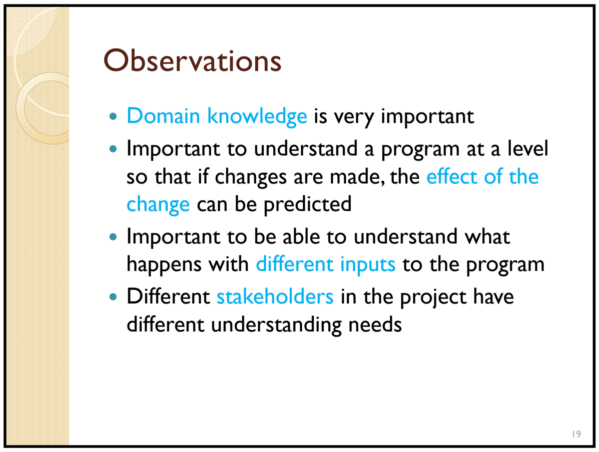
Static program analysis

- Def. The process of inferring results about the nature of a program according to some model without executing the subject program
- Syntactic analysis, type checking and inference
- Control and data flow analysis
- Structural analysis
- Slicing and dicing
- Cross references
- Complexity measures
- Navigation



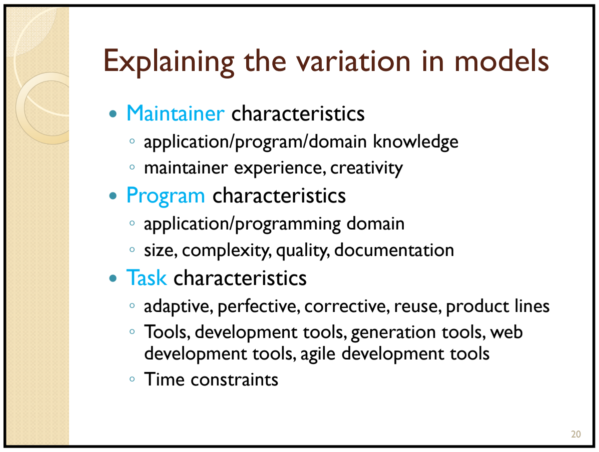
Dynamic program analysis

- Def. The process of discovering run-time dependencies in a subject system
- Object instantiation dependencies
- Dynamic binding and polymorphism
- Method invocation graph
- Registered and call-back functions
- Path coverage testing
- Memory management
- Performance bottlenecks
- Transactions
- Concurrency



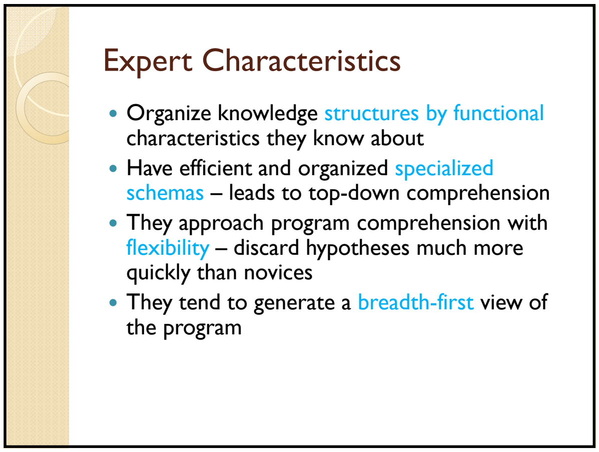
Observations

- Domain knowledge is very important
- Important to understand a program at a level so that if changes are made, the effect of the change can be predicted
- Important to be able to understand what happens with different inputs to the program
- Different stakeholders in the project have different understanding needs



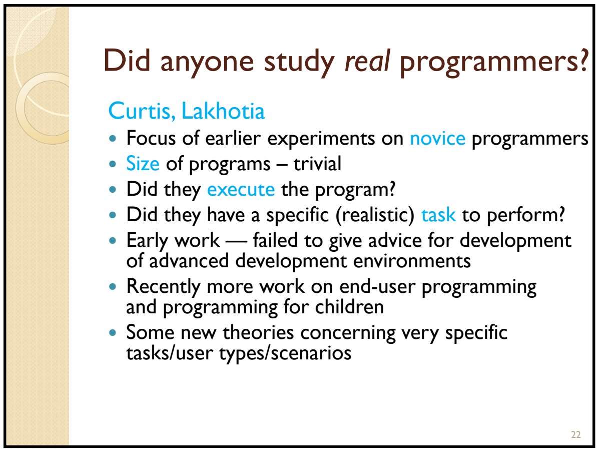
Explaining the variation in models

- Maintainer characteristics
  - application/program/domain knowledge
  - maintainer experience, creativity
- Program characteristics
  - application/programming domain
  - size, complexity, quality, documentation
- Task characteristics
  - adaptive, perfective, corrective, reuse, product lines
  - Tools, development tools, generation tools, web development tools, agile development tools
  - Time constraints



Expert Characteristics

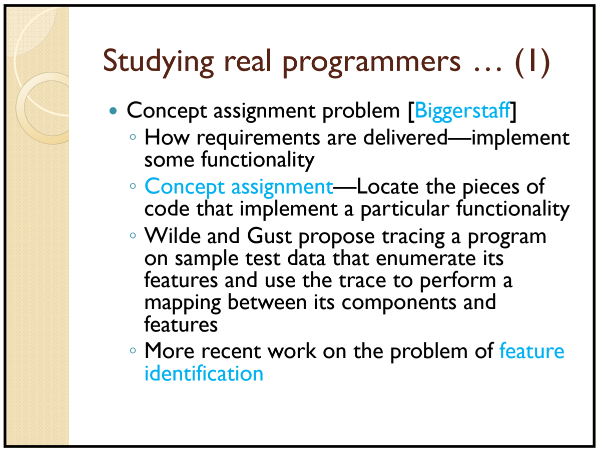
- Organize knowledge structures by functional characteristics they know about
- Have efficient and organized specialized schemas – leads to top-down comprehension
- They approach program comprehension with flexibility – discard hypotheses much more quickly than novices
- They tend to generate a breadth-first view of the program



Did anyone study *real* programmers?

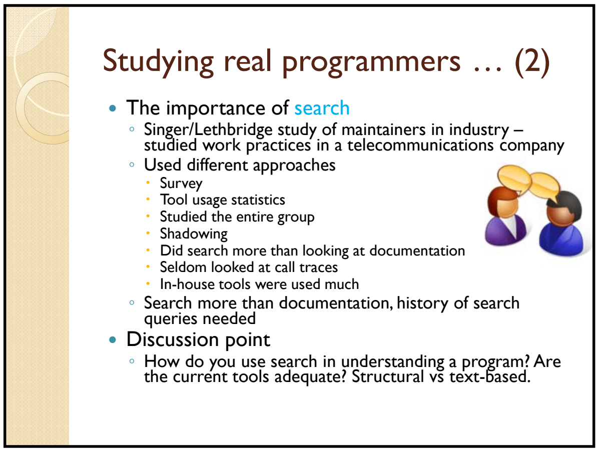
Curtis, Lakhotia

- Focus of earlier experiments on novice programmers
- Size of programs – trivial
- Did they execute the program?
- Did they have a specific (realistic) task to perform?
- Early work — failed to give advice for development of advanced development environments
- Recently more work on end-user programming and programming for children
- Some new theories concerning very specific tasks/user types/scenarios



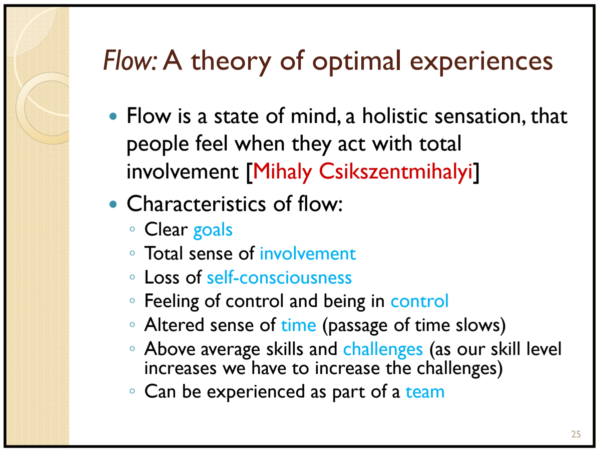
Studying real programmers … (1)

- Concept assignment problem [Biggerstaff]
  - How requirements are delivered—implement some functionality
  - Concept assignment—Locate the pieces of code that implement a particular functionality
  - Wilde and Gust propose tracing a program on sample test data that enumerate its features and use the trace to perform a mapping between its components and features
  - More recent work on the problem of feature identification



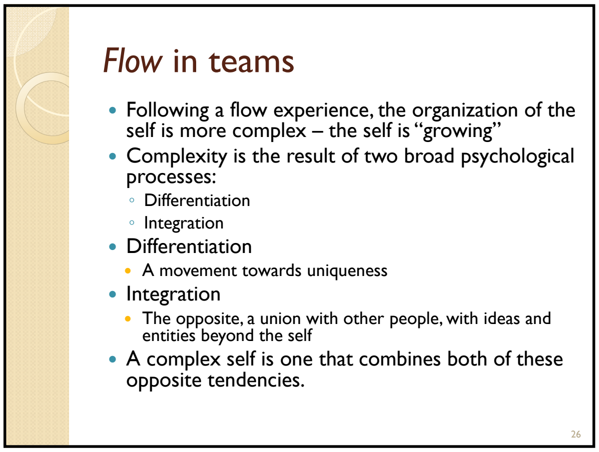
Studying real programmers … (2)

- The importance of search
  - Singer/Lethbridge study of maintainers in industry – studied work practices in a telecommunications company
  - Used different approaches
    - Survey
    - Tool usage statistics
    - Studied the entire group
    - Shadowing
    - Did search more than looking at documentation
    - Seldom looked at call traces
    - In-house tools were used much
  - Search more than documentation, history of search queries needed
- Discussion point
  - How do you use search in understanding a program? Are the current tools adequate? Structural vs text-based.



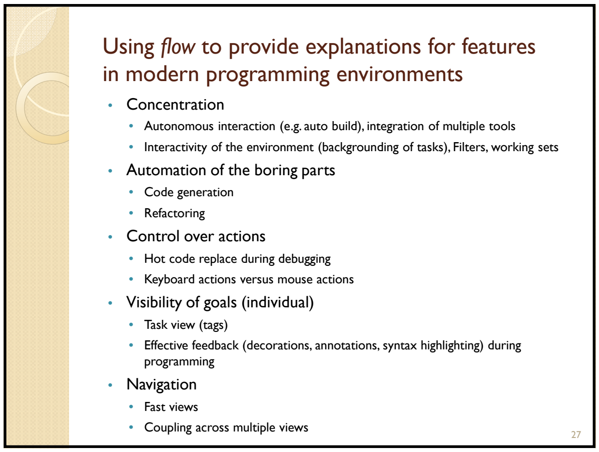
Flow: A theory of optimal experiences

- Flow is a state of mind, a holistic sensation, that people feel when they act with total involvement [Mihaly Csikszentmihalyi]
- Characteristics of flow:
  - Clear goals
  - Total sense of involvement
  - Loss of self-consciousness
  - Feeling of control and being in control
  - Altered sense of time (passage of time slows)
  - Above average skills and challenges (as our skill level increases we have to increase the challenges)
  - Can be experienced as part of a team



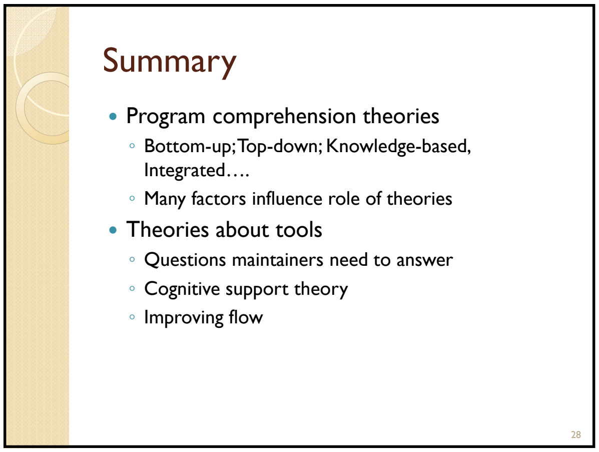
Flow in teams

- Following a flow experience, the organization of the self is more complex – the self is "growing"
- Complexity is the result of two broad psychological processes:
  - Differentiation
  - Integration
- Differentiation
  - A movement towards uniqueness
- Integration
  - The opposite, a union with other people, with ideas and entities beyond the self
- A complex self is one that combines both of these opposite tendencies.



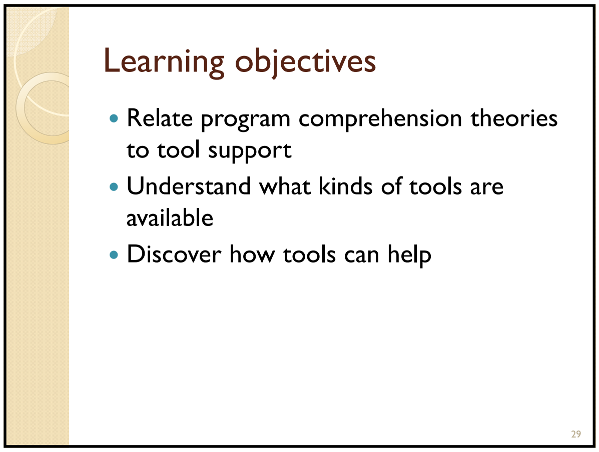
Using *flow* to provide explanations for features in modern programming environments

- Concentration
  - Autonomous interaction (e.g. auto build), integration of multiple tools
  - Interactivity of the environment (backgrounding of tasks), Filters, working sets
- Automation of the boring parts
  - Code generation
  - Refactoring
- Control over actions
  - Hot code replace during debugging
  - Keyboard actions versus mouse actions
- Visibility of goals (individual)
  - Task view (tags)
  - Effective feedback (decorations, annotations, syntax highlighting) during programming
- Navigation
  - Fast views
  - Coupling across multiple views



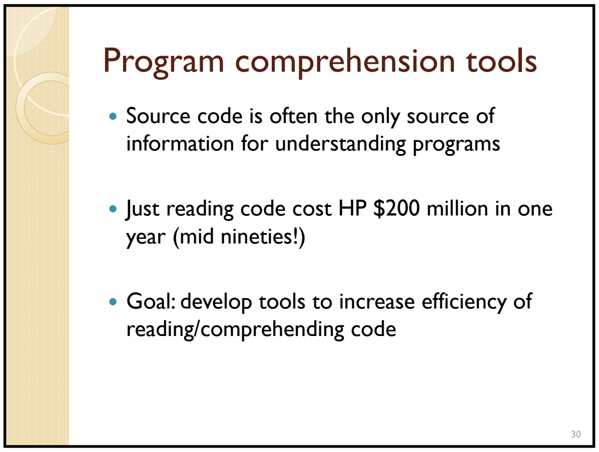
Summary

- Program comprehension theories
  - Bottom-up; Top-down; Knowledge-based, Integrated….
  - Many factors influence role of theories
- Theories about tools
  - Questions maintainers need to answer
  - Cognitive support theory
  - Improving flow



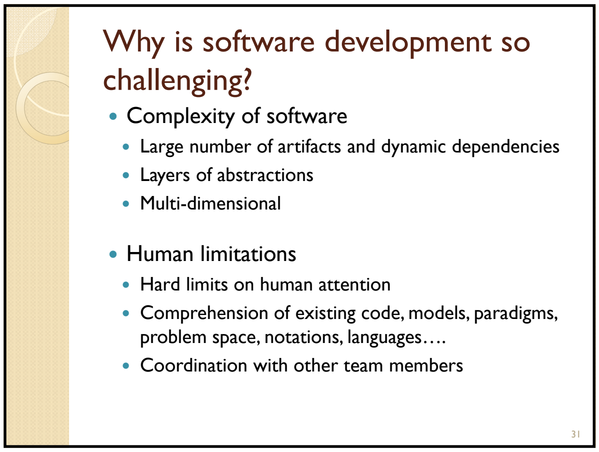
Learning objectives

- Relate program comprehension theories to tool support
- Understand what kinds of tools are available
- Discover how tools can help



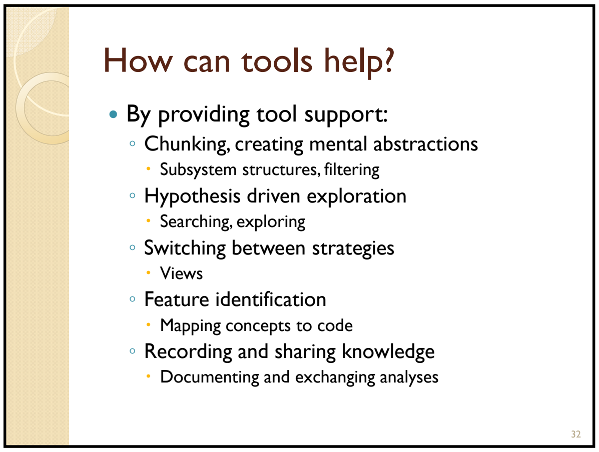
Program comprehension tools

- Source code is often the only source of information for understanding programs
- Just reading code cost HP $200 million in one year (mid nineties!)
- Goal: develop tools to increase efficiency of reading/comprehending code



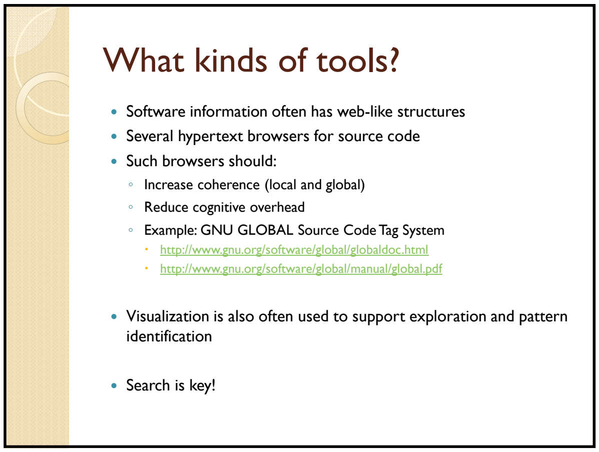
Why is software development so challenging?

- Complexity of software
  - Large number of artifacts and dynamic dependencies
  - Layers of abstractions
  - Multi-dimensional
- Human limitations
  - Hard limits on human attention
  - Comprehension of existing code, models, paradigms, problem space, notations, languages….
  - Coordination with other team members



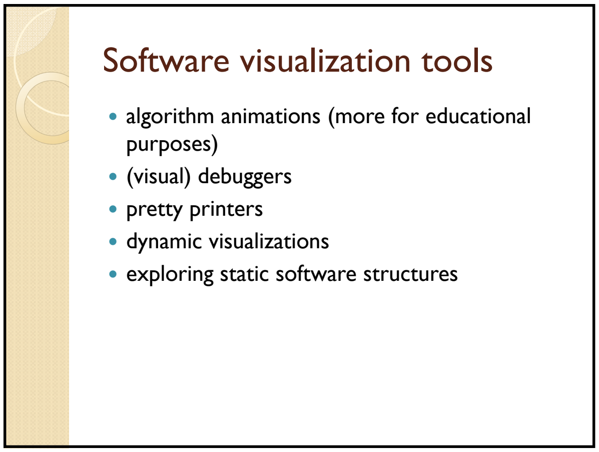
How can tools help?

- By providing tool support:
  - Chunking, creating mental abstractions
    - Subsystem structures, filtering
  - Hypothesis driven exploration
    - Searching, exploring
  - Switching between strategies
    - Views
  - Feature identification
    - Mapping concepts to code
  - Recording and sharing knowledge
    - Documenting and exchanging analyses



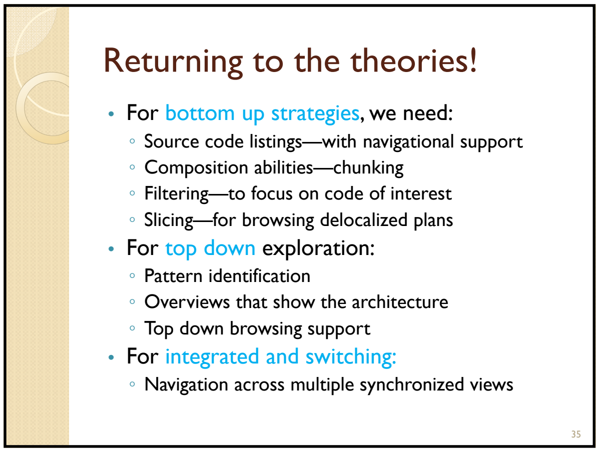
What kinds of tools?

- Software information often has web-like structures
- Several hypertext browsers for source code
- Such browsers should:
  - Increase coherence (local and global)
  - Reduce cognitive overhead
  - Example: GNU GLOBAL Source Code Tag System
    - http://www.gnu.org/software/global/globaldoc.html
    - http://www.gnu.org/software/global/manual/global.pdf
- Visualization is also often used to support exploration and pattern identification
- Search is key!



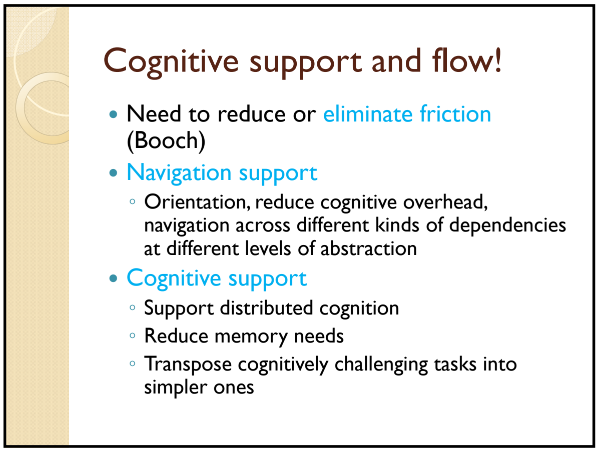
Software visualization tools

- algorithm animations (more for educational purposes)
- (visual) debuggers
- pretty printers
- dynamic visualizations
- exploring static software structures



Returning to the theories!

- For bottom up strategies, we need:
  - Source code listings—with navigational support
  - Composition abilities—chunking
  - Filtering—to focus on code of interest
  - Slicing—for browsing delocalized plans
- For top down exploration:
  - Pattern identification
  - Overviews that show the architecture
  - Top down browsing support
- For integrated and switching:
  - Navigation across multiple synchronized views



Cognitive support and flow!

- Need to reduce or eliminate friction (Booch)
- Navigation support
  - Orientation, reduce cognitive overhead, navigation across different kinds of dependencies at different levels of abstraction
- Cognitive support
  - Support distributed cognition
  - Reduce memory needs
  - Transpose cognitively challenging tasks into simpler ones

Navigation support

- Hypertext is one approach
- Navigation views
  - Package, outline, cross reference, bookmarks, bread crumbs, history views, …
- Graph navigation
  - Follow typed dependencies
- Search
  - Structured vs. unstructured search
  - Integrated Structured and unstructured search
  - Search can replace navigation

The promise of software tools

- Makes human cognition easier or better
- Redistribution: Cognitive resources or cognitive processes that are in the head can be moved outside of the head
- Perceptual substitution: Artifacts can transform a task into one that ca be done more quickly and more easily
- Ends-means reification: Solving a problem can be considered a search f a solution – parts of the problem space can be "reified" – supporting display-based problem solving
- Tools are an extension of a programmer's mind:
  - By providing a mechanism to externalize cognitive processes (e.g., scripts)
  - By providing alternate representations of information (e.g., models, views)
  - By supporting the manipulation of artifacts to facilitate cognitive tracing and experimentation (e.g., filtering, exploration, elision)
  - By helping bridge the gap from artifacts to abstractions (e.g., chunking support, building subsystem structures, pattern identification)

38

In particular… visualization

How can visualization improve program comprehension during software development, maintenance and evolution?

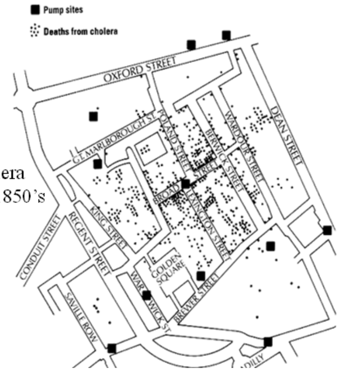
Information visualization

- People have used external aids for centuries to amplify cognition
  - Paper, slide rule, diagrams, charts
- Visualize: to form a mental image or vision
- Visualization is done by humans, it is not done by a computer
  - But the use of computer supported, interactive, visual representations of abstract data can amplify cognition
- Visualizations can help us gain insight about data

40

Gaining Insight from Data….

An outbreak of cholera by Dr. John Snow, 1850's



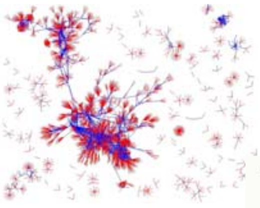
Software visualization tools…

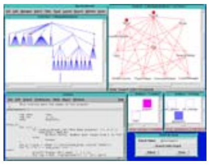
- Structure of information – architecture
- How the program works – demonstrating the virtual machine
- Presenting attributes of large software systems
- Changing the perspective – multi-dimensional views
- Display-based reasoning

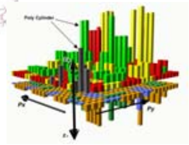
Software exploration tools provide graphical views of static and dynamic software structures linked to textual views of the source code and documentation – reveal multi-dimensional views of the software

42

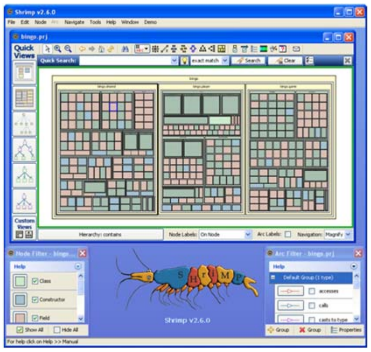
Example Software Visualization Tools







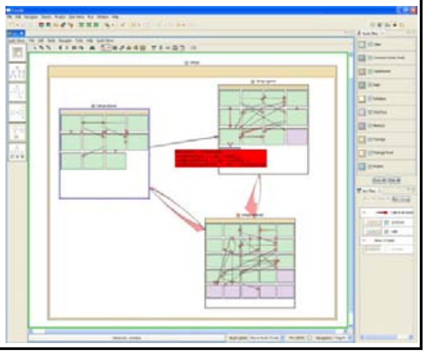
SHriMP, Creole, www.thechiselgroup.org/creole
Margaret-Anne Storey, University of Victoria



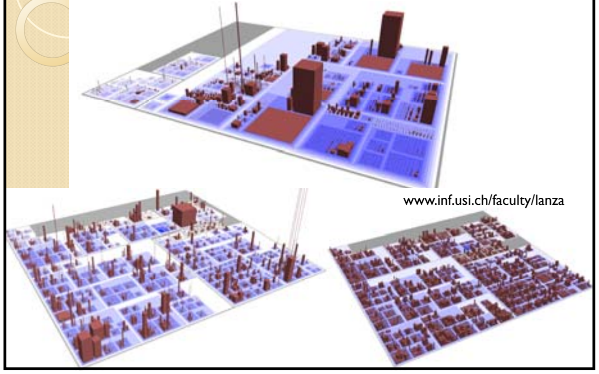
44

SHriMP, Creole, www.thechiselgroup.org/creole
Margaret-Anne Storey, University of Victoria

Integrated with Eclipse

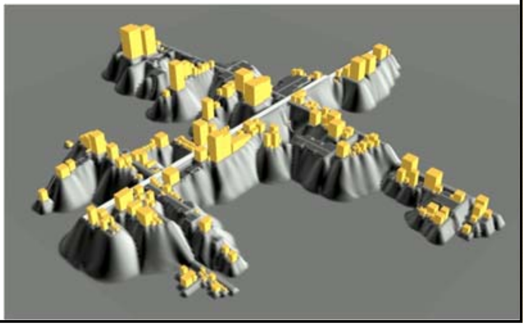


Visualizing Software Systems as Cities
Michele Lanza, University of Lugano



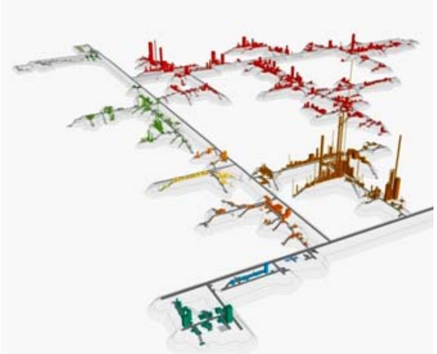
www.inf.usi.ch/faculty/lanza

Software Cities and Landscapes, software-cities.org
Claus Lewerentz, University of Technology Cottbus



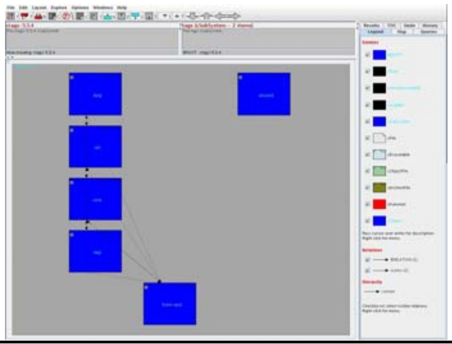
Software Cities and Landscapes, software-cities.org
Claus Lewerentz, University of Technology Cottbus

Eclipse
JDK

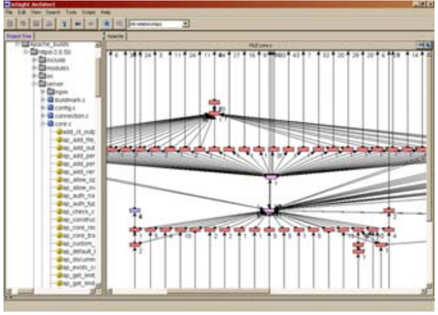


Software Cities and Landscapes, software-cities.org
Claus Lewerentz, University of Technology Cottbus

PBC, LSEdit,
www.swag.uwaterloo.ca/lsedit/index.html
Ric Holt, SWAG, Waterloo



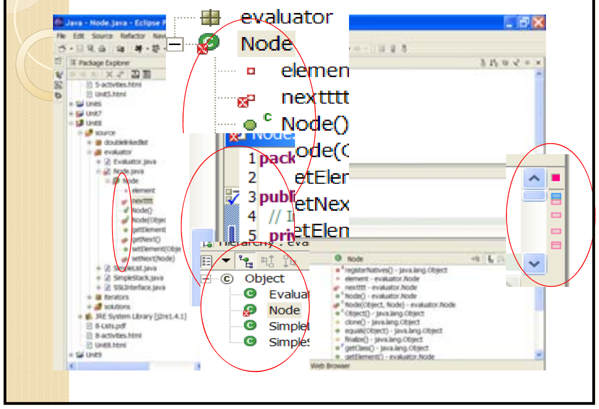
Klocwork insight architect
www.klocwork.com/products/insight/architect-code-visualization/index.php



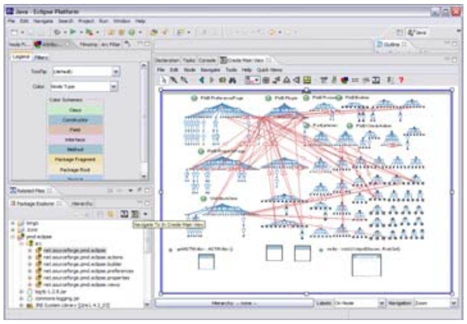
State-of-the-art IDE: eclipse

- Customizable
- Extensible (plug-in architecture)
- Open source
- Powerful features:
  - Refactoring, code assist, code folding, hot code debugging, version control, task view, team support, ….
  - User interface: perspectives and views for managing tools/features, excellent user interface capabilities
  - Many plug-ins, e.g. UML plug-ins
- *Usability and cognitive support are key*

Eclipse and visualization



Creole: Integration of SHriMP with Eclipse



Imagix 4D

Supports:

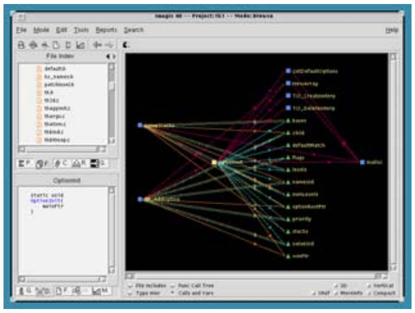
- Reverse engineering
- Code quality metrics
- Documentation

Features

- Graph window
- Integrated source code browing

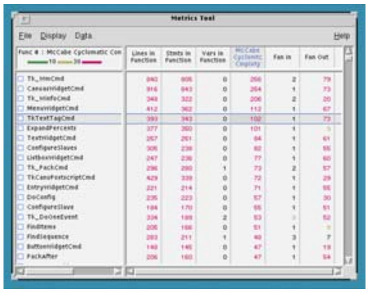
(Rigi like)

Imagix – Graph Window



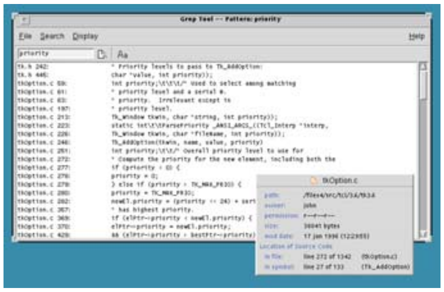
Imagix – integrated source code browsing

Imagix – Metrics Tool



	Lines in Function	Stmts in Function	Vars in Function	McCabe Cyclomatic Complxty	Fan in	Fan Out
□ Tk_WmCmd	840	806	0	266	2	79
□ CanvasWidgetCmd	916	843	0	264	1	73
□ Tk_WinfoCmd	348	322	0	206	2	20
□ MenuWidgetCmd	412	362	0	112	1	67
□ TkTextTagCmd	393	343	0	102	1	73
□ ExpandPercents	377	360	0	101	1	5
□ TextWidgetCmd	257	251	0	84	1	61
□ ConfigureSlaves	305	238	0	82	1	55
□ ListboxWidgetCmd	247	236	0	77	1	60
□ Tk_PackCmd	296	280	1	73	2	57
□ TkCanvPostscriptCmd	429	339	0	72	1	29
□ EntryWidgetCmd	221	214	0	71	1	55
□ DoConfig	235	223	0	57	1	30
□ ConfigureSlave	184	170	0	55	1	51
□ Tk_DoOneEvent	334	189	2	53	[illegible]	52
□ FindItems	205	166	0	51	1	[illegible]
□ FindSequence	283	211	1	49	3	7
□ ButtonWidgetCmd	148	145	0	47	1	19
□ PackAfter	206	160	0	47	1	54

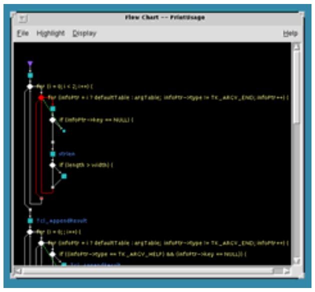
Imagix – Grep Tool

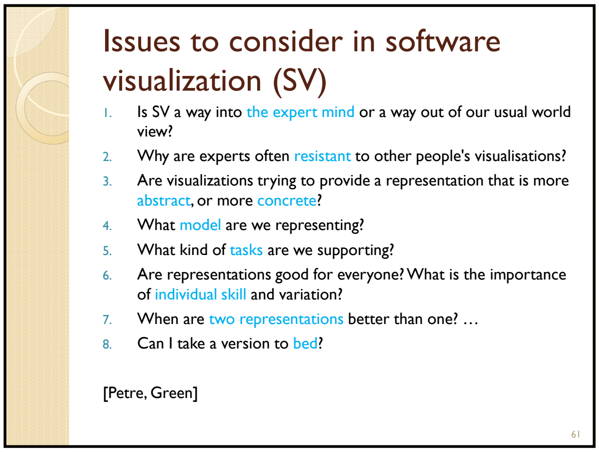


```
tk.h 242:         * Priority levels to pass to Tk_AddOption:
tk.h 446:         char *value, int priority));
tkOption.c 59:    int priority;\t\t\t/* Used to select among matching
tkOption.c 61:    * priority level and a serial #.
tkOption.c 63:    * priority.  Irrelevant except to
tkOption.c 197:   * priority level.
tkOption.c 213:   Tk_Window tkwin, char *string, int priority));
tkOption.c 223:   static int\t\tParsePriority _ANSI_ARGS_((Tcl_Interp *interp,
tkOption.c 226:   Tk_Window tkwin, char *fileName, int priority));
tkOption.c 246:   Tk_AddOption(tkwin, name, value, priority)
tkOption.c 251:   int priority;\t\t/* Overall priority level to use for
tkOption.c 272:   * Compute the priority for the new element, including both the
tkOption.c 277:   if (priority < 0) {
tkOption.c 278:   priority = 0;
tkOption.c 279:   } else if (priority > TK_MAX_PRIO) {
tkOption.c 280:   priority = TK_MAX_PRIO;
tkOption.c 282:   newEl.priority = (priority << 24) + seri
tkOption.c 367:   * has highest priority.
tkOption.c 369:   if (elPtr->priority < newEl.priority) {
tkOption.c 370:   elPtr->priority = newEl.priority;
tkOption.c 429:   && (elPtr->priority > bestPtr->priority)
```

tkOption.c
path: /files4/src/tcl/3.6/tk3.6
owner: john
permission: r--r--r--
size: 36041 bytes
mod date: 17 Jan 1996 (12:29:55)
Location of Source Code
in file: line 272 of 1342 (tkOption.c)
in symbol: line 27 of 133 (Tk_AddOption)

Imagix – Flow Chart





Issues to consider in software visualization (SV)

1. Is SV a way into the expert mind or a way out of our usual world view?
2. Why are experts often resistant to other people's visualisations?
3. Are visualizations trying to provide a representation that is more abstract, or more concrete?
4. What model are we representing?
5. What kind of tasks are we supporting?
6. Are representations good for everyone? What is the importance of individual skill and variation?
7. When are two representations better than one? …
8. Can I take a version to bed?

[Petre, Green]