Software Maintenance

- Key problems/issues
- Historical models and data
- Program comprehension

Software maintenance

Defined in IEEE Standard 1219 as:
“The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.”

Related term: Software evolution

Maintenance activities

Four distinct categories:
- Adaptive: accommodates changes in the software environment
- Perfective: incorporates new user requirements
- Corrective: fixes errors
- Preventive: aims to prevent future problems

Studies of prevalence:
- Around 75% of maintenance is adaptive or perfective; 21% corrective [Lientz and Swanson]
- Incorporation of new requirements is a core problem for software maintenance

Fundamental problems

If changes can be anticipated at design time, they can be built in by some form of parameterization

Fundamental problem: Many modifications involve changes the original designers could not even conceive of

Maintenance important to understand because:
- It consumes a large portion of overall lifecycle costs
- Inability to change software quickly and reliably means business opportunities are lost
- Concerns have led to new and competing lifecycle models

Topics

Empirical model of dynamic evolution of software products [Lehman and Belady]
- Three laws of software evolution dynamics

Maintenance of legacy systems

Program comprehension

Lehman/Belady model of evolution

Classic empirical study of the evolution of a real software system over many releases
- System: OS/360
- Releases: over 20
- Metrics:
  - # of modules added (per release)
  - # of modules handled (per release)
  - others
Laws of evolution dynamics

1. Law of Continuing Change: A system that is used undergoes continuing change until it is judged more cost effective to freeze and recreate it
2. Law of Increasing Entropy: The entropy of a system (its unstructuredness) increases with time, unless specific work is executed to maintain or reduce it
3. Law of statistically smooth growth. Growth trend measures of global system attributes may appear to be stochastic locally in time and space, but, statistically, they are cyclically self regulating, with well-defined long-range trends

Other consequences of model

Analyses indicate a cyclical pattern of growth
- Modules handled per release fluctuates by release within a 400-module window
- This pattern is invariant under changes to languages, programming staff, machines, etc
Possible to budget in time to restructure code (i.e., preventive maintenance) to reduce the rate of growth of complexity; however rate will always be positive

Issues with legacy systems

Legacy systems

Defn: System of value that is being maintained by staff that were not part of its original development
Typically:
- Large (MLOC), complex, old (> 10 years)
- Critical corporate assets
  - E.g., telecom switching systems, banking systems, defense systems
- Highly reliable

Legacy systems (continued)

Maintenance consumes up to 90% of company’s IT budget
Running on legacy platforms, and/or written in legacy languages
Poorly documented:
- documentation out of date
- too much documentation
- original source may not even be available
Coping with legacy systems

No obvious solution to problem:
– Too large to rewrite from scratch
– Can’t shift staff resources w/o risking critical losses

Approaches to date:
– Legacy-platform simulators on new platforms
– Port compilers for legacy languages to new platforms
– Reverse engineering

Program comprehension

Greatest part of maintenance process devoted to understanding the system being maintained
– 47% - 62%
– I.e., reading documentation, scanning code, trying to understand the changes to be made

Idea: To improve software development, one must improve maintenance process; to improve maintenance, one must improve program comprehension

Gaps complicating prog comp

Problem in application domain vs solution in a programming language
Concrete world of programs/machines vs abstract world of high-level design descriptions
Coherent, highly structured description of original system vs actual system, whose structure may have disintegrated over time
Hierarchical nature of programs vs. associational nature of human cognition
Bottom-up analysis of the source code vs. top-down synthesis of an application description

Application domain vs program

Program = solution to problem in some application domain
However, program may contain no hints about the particular problem it solves
– Hints found in comments, variable names
– These are informal and nearly always out of date
Consequently, most automatic comprehension tools work with only program text

Concrete programs; abstract descriptions

Programs incredibly detailed:
– Not every detail is relevant to a particular comprehension need,
– but how can you tell which ones are relevant?
Abstraction is the process of sifting away the unimportant details; however abstraction is not linear!
Many lines of program code result from the interleaving of multiple purposes or program plans
Hierarchical programs; associative cognition

Programs are highly structured syntactically
Human cognition is associative:
  - Raw data are perceived
  - Patterns in data are detected
  - Abstractions are constructed that relate them
  - Understanding controlled by setting up and verifying expectations that follow from these associative suggestions and abstractions
Program is understood to extent that reader can build up correct chunks from the details in the code

Program analysis; model synthesis

Process of analyzing a program proceeds "bottom-up"
Process of synthesizing a model of program behavior proceeds "top-down" from an expectation of the program’s behavior
Problem: Bottom-up analysis and top-down synthesis must be synchronized

Humans in the loop

Program comprehension currently a highly manual task
Software psychology attempts to describe human limitations in interacting with computers
Program comprehension studied via experiments involving memorization/reconstruction of programs

Experiments in software psychology (Schneiderman)

Utility of comments not substantiated
  - Shown to impede understanding of small programs
  - Incorrect or out of date
  - Larger programs not studied
  - Same is true for indentation
Mnemonic variable names contribute to comprehension:
  - IF carry semantic (rather than syntactic) meaning
  - Reduces reader’s short-term memory load
  - Mnemonics likely have different meanings to different readers:
    • Would be useful if reader could easily and systematically rename identifiers to reflect understanding relative to a given task

Models of program comprehension

Two major models:
  - Top-down approach, due to Ruven Brooks
  - Bottom-up approach, due to Elliot Soloway
Knowledge of these models can help one manage one’s own comprehension tasks
Can also serve as the basis for support tools

Brooks model of comprehension

Key ideas:
  - Programming process is construction of mappings from a task domain through one or more intermediates into the programming domain
  - Comprehension is process of reconstructing these mappings
  - Reconstruction process is expectation-driven
    • Hypotheses formed to connect domains
    • Reconstruction driven by creation, confirmation, and refinement of these hypotheses
Brooks model (continued)

Understander begins with initial hypothesis:
– global, abstract description of what understander thinks the program does
– even before looking at a single line of code
– often based on the name or description of program
Initial hypothesis then refined into a cascade of sub-hypotheses, which are explored depth first
Eventually sub-hypothesis is sufficiently refined that it can be confirmed by looking at code

Beacons

Defn: Program details that confirm (or lend credence to) an hypothesis
Example: "sort" hypothesis confirmed by finding a pair of loops whose body contains a comparison and exchange between elements
Beacons are first link between hypotheses and program text
Existence verified experimentally
– Experts more efficient at locating beacons than are novices

Abilities of understander

Knowledge about the various domains affects understanding at all levels
– Task-domain knowledge affects quality of primary and subsidiary hypotheses
– Programming-domain knowledge affects lower-level bindings and beacon location process

Soloway model of comprehension

Key idea: To understand a program is to uncover the intention behind the code
– Goals denote intentions
– Plans denote techniques for realizing intentions
– Plans work as rewrite rules, which convert goals into subgoals and finally into code
Comprehension involves recognizing plans in code, then abstracting them to subgoals and eventually higher-level goals

Soloway model continued

Existence of plans in programs confirmed experimentally
Delocalized plans:
– Plans that manifest as non-contiguous source code, which may even cross module boundaries
– Key source of complexity in program comprehension
Interleaving problem
Plan calculus: A model of plan composition

Supporting program comprehension

Automated cliché recognition
– Build a knowledge base of common programming patterns (i.e., clichés)
  • e.g., pattern of loops and conditionals to perform binary search
– Uncover clichés in existing code to support bottom-up comprehension
Interleaving detection
– Identify instances of optimizations that capitalize on some form of resource sharing
– Automatically undo the optimizations to simplify comprehension
Designing for comprehension

Reify and maintain representations of intentions rather than just results of refining intentions (code)
  – Knuth’s *literate programming*
  – Intentional programming
Automated refinement of programs from specifications
  – Specifications often separate concerns better than code
  – So have tools synthesize program code from specs
Aspect-oriented programming

Domain analysis

Domain: problem area, such as weather modeling, accounting, report writing, etc.
Typically many application programs exist to solve problems in a single domain
Domain analysis: Engineering approach to codifying domain knowledge
  – Looks at wealth of programs in same domain
  – Teases out and then explicitly documents useful "domain abstractions"
  – Yields "domain books"