Restructuring, design recovery, and reengineering

- Taxonomy of concepts from reverse engineering and design recovery
- Concept assignment problem
- Domain models
- Tool support for these complex tasks

Recall from last time...

Second law of software evolutionary dynamics:

*Law of Increasing Entropy:* The entropy of a system (its unstructuredness) increases with time, unless specific work is executed to maintain or reduce it.

“Specific work” executed to maintain or reduce entropy drives many activities:
- Code restructuring
- Reverse engineering/reengineering
- Design recovery

Signs of entropy

Structure of a system can be observed and improved without requiring deep understanding of the system

Systems should exhibit good “modular structure”, but how can we determine “goodness” of a modular structure?

Two metrics: coupling and cohesion

Sound structures exhibit highly cohesive, loosely coupled modules

Cohesion

**Defn:** Extent to which a module (e.g., procedure) has unity of purpose

Modules with high cohesion are more reusable, reliable, understandable, and modifiable

Example of low cohesion:
- Procedure with 10 parameters, the first of which acts as a switch to indicate which of the subsequent 9 are to be used
- Sign of multiple procedures hiding inside one monolith

Desirable for modules to be highly cohesive

Coupling

**Defn:** Degree to which program modules rely on the implementations of others

Loose coupling means that changes to the implementation of one module will not affect the other and vice versa

Example of tight coupling:
- Module responsible for monitoring a sensor exposes some of the details of the hardware API, such as that the data from the sensor is stored in a bitfield of width 8
- Modules using the monitor perform bit arithmetic on this exposed bitfield
- When system modified to use a new sensor, all modules will need to be modified

Desirable for modules to be loosely coupled

Code restructuring

Combats entropy by replacing structurally-eroded system with an “equivalent” system whose structure is more sound

Examples of “weak” structures
- Procedures with too many parameters
- Procedures that are too long or that lack cohesion
- Code with unstructured control flow
  - e.g., jumps into loops

Tool support:
- Automated restructuring (for intra-module improvement)
- Automated analysis of coupling/cohesion
Refactoring

Form of restructuring that is popular in OO systems
Couched in terms of "rewrite rules" that transform a poorly structured class hierarchy or method into a more "well-factored" version
Example refactoring:
- Pull up method
- Change value to reference
- Extract hierarchy
- Remove parameter
- Many more...

Example: Extract Method

Inputs: Source method, sequence of lines
Outputs: New source and target methods
Procedure:
1. Create new target method, name by intention (what it does, not by how it does it)
2. Copy line sequence from source method into new target method
3. Scan the extracted code for references to variables local to source method
   - if variable used only within extracted code, then move declaration from source to target method
   - else declare parameter to target method …

Refactoring (continued)

Rationale: Code changes often erode the design; erosion has a cumulative effect
Analyst may choose a set of classes and/or methods and then apply a refactoring
- Supported by automated tools
- Correctness of refactoring judged by testing
Possible for highly structured designs to emerge where none existed before
Often used in agile development methods, which place less emphasis on up-front design in favor of incremental development and early testing

Tool support for refactoring

SmallTalk refactoring browser:
- SmallTalk development environment written in smallTalk
XRefactory:
- Enables standard refactorings for C++

Design is more than just structure

Code restructuring can help to combat entropy, but it’s not capable of combating a deeper problem inherent to legacy systems
Problem: Loss of design intent as system evolves
- As old programmers retire (or move to new projects) they take a lot of the design of a system with them
- New programmers, who lack the big picture, adopt a black-box approach to extension and maintenance

Reverse engineering

Defn: Process of analyzing a subject system to:
- identify the system’s components and their interrelationships, and
- create representations of the system in another form or at a higher level of abstraction.
Has roots in reverse engineering of hardware systems
Redocumentation

Simplest and oldest form of reverse engineering
Produces a semantically equivalent representation of a system at the same level of abstraction:
- E.g., producing structure charts from PL/1 code or class diagrams from OO code
- Does not involve the encoding of deep domain abstractions
Aims to "recover" documentation that should have existed

Design recovery

Subset of reverse engineering
Takes as input information other than what appears directly in the code itself
- E.g., domain knowledge, documentation, etc
Typified by use of fuzzy recognition and plausible inference from accumulated evidence rather than formal deduction
Key problem: Assignment of domain concepts to program entities

Concept assignment problem

Defn: Problem of assigning concepts that are meaningful to the human analyst to their implementation instances within a program
- E.g., "reserve an airplane seat" vs.
  if ((seat = request(flight)) && available(seat))
  then reserve(seat, customer)
Design recovery is all about establishing concept assignments

Assigning concepts to code

Two general tasks:
1. identify what few entities and relationships are really important out of the mass of entities and relationships that appear in the code
2. assign them to known domain concepts and relationships
First task is more generic, requiring only knowledge of programming language; second requires deep knowledge of the domain
Suggests that a priori knowledge of the domain is required
- i.e., can't throw new programmers onto a design recovery problem and expect to get meaningful results

Example of concept assignment tasks

Example taken from multi-tasking window system
Two tasks:
1. Given code, what kind of information may be plausibly inferred?
2. Given a domain model, what kind of additional information may be plausibly inferred?

Example

Example taken from multi-tasking window system
Two tasks:
1. Given code, what kind of information may be plausibly inferred?
2. Given a domain model, what kind of additional information may be plausibly inferred?
Generic knowledge tells us...

Statements related in a non-casual way because:
- grouped together (proximity)
- bracketed by empty lines
- share a common surface structure
- many of the entities are coupled by common parameters (e.g., MAXPROCS, MAXBREAKS)

Conclusion: We can assign this code to the generic concept data-group
- instance of some (currently unknown) domain concept
- likely a composite concept, comprising sub-concepts just based on the number of elements and their slight differences in naming

Task 2

Assign this data-group (and subcomponents) to some domain-specific concept
- requires a priori domain-specific knowledge
- in this case, a model of how debugging tools handle breakpoints

Rep. of domain knowledge

Target program

Set breakpoints command
locations of breakpoints
code bytes from breakpoints
number of breakpoints
Interrupt service routine
Save code bytes and store
interrupt 3
Restore bytes at locations

User input
Break event

Question

How might analyst relate this model to specific instances of concepts in the program under analysis?
What program features might he or she use to help in making the assignments?

Example (continued)

Consider the Location of Breakpoints concept

Features in the program:
- natural language token meanings
  - e.g., breakpoint, breakpoints, brkpt
- occurrence of closely related concepts
  - e.g., references to registers, instructions, etc
- individual relations paralleling those in the model
  - used_by relations between these data items and previously assigned breakpoint functions in the code correspond to arrows in the model
- overall pattern of relationships in the model

Concept assignment process

Akin to solving a crossword puzzle:
- initial concept assignments may be generic or tentative until they can be supported by the evidence of further concept assignments
- may occur in any order
  - E.g., may have assigned concepts to functions bpint3, set_breaks, and restore_breaks prior to trying to assign concepts to the data-group, or vice versa

Question: How might automated tools support concept assignment?
Design recovery tools

Example: DESIRE system

Provides:
- naïve assistants to help with generic assignment
- intelligent assistants to help with domain-specific assignment
- graphical browsers to analyze structure of code
  - call graphs
  - slicers
- interactive “agenda” for recording where in the process one is
- support for recording concept assignments

Domain models

Largely held that domain knowledge is the asset that leads to good designs and orderly evolution of large software systems
- i.e., “knowledge” contained in the human-oriented model of the breakpoints handler is much more valuable than the actual code in any one system
- Thus there is value in domain models, which record these key representations and enable new programmers to access them

Domain engineering is the construction of good domain models
- Usually performed via domain analysis of existing systems
- Results useful for documentation and also reuse

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”

Domain-specific languages

Domain analysis can often lead to custom languages for selecting or parameterizing domain abstractions and then automating the generation of code
Examples of DSLs:
- Telecom switching logic generated from state diagrams, some of which were reverse engineered from legacy code
- Logistics scheduling algorithms synthesized from declarative specifications of constraints
- etc.

Reengineering

Defined as a process of reverse engineering or design recovery followed by a systematic “forward engineering”
Applications:
- Migration from use of flat files to relational databases
- Migration from legacy terminal screens to GUIs

Business process reengineering

Reengineering incorporates new technologies:
- E.g., movement from batch systems to interactive web-based systems
- Often technologies can serve as business enablers
- So how should one reengineer business rules and processes that were tied to the old technologies?

BPR is the idea that reengineering involves not only code reengineering but also business rule rethinking and rewriting