Program Comprehension

Idaho State University | Computer Science

Isaac Griffith

CS 4423 and CS 5523
Department of Computer Science
Idaho State University

ROAR

Idaho State University
Computer Science

Outcomes

After today's lecture you will be able to:

- Understand and describe the general ideas related to program comprehension
- Understand and describe the type of knowledge to be gained from software artifacts
- Understand and describe the different mental models used in program comprehension

ROAR

Program Comprehension

CS 4423/5523

ROAR

General Idea

- It is important to comprehend a complex system to be able to maintain it.
- Modification of software with inaccurate and incomplete understanding is likely to degrade its performance and reliability.
- Good program comprehension is key to providing effective software maintenance and effective evolution of software.

General Idea

- To understand the role of program comprehension, consider five kinds of tasks associated with program maintenance (Table 8.1.)
- Understanding the system or problem is common to all maintenance and evolution tasks.
- Understanding of a system is a cognitive issue and a number of cognitive models have been developed (Table 8.1.)

Basic Terms

Model	Maintenance Activity	Authors
Control-flow	Understand	Pennington
Functional	Understand	Pennington
Top-down	Understand	Soloway, Adelson, and Ehrlich
Integrated	Understand, Corrective, Adaptive, and Perfective	Von Mayrhauser and Vans
Other	Enhancement, Understand	Letovsky, Brooks, Shneiderman, and Mayer

- To understand these cognition models, the following set of terms form the background material:
  - Goal of code cognition
  - Knowledge
  - Mental model

General Idea

Maintenance Tasks	Activities
Adaptive	Understand system Define adaptation requirements Develop preliminary and detailed adaptation design Code changes Debug Regression tests
Perfective	Understand system Diagnosis and requirements definition for improvements Develop preliminary and detailed perfective design Code changes/additions Debug Regression tests
Corrective	Understand system Generate/evaluate hypotheses concerning problem Repair code Regression tests

General Idea

Reuse	Understand problem Find solution based on close fit with reusable components Locate components Integrate components
Code coverage	Understand problem Find solution based on predefined components Reconfigure solution to increase likelihood of using predefined components Obtain and modify predefined components Integrate modified components

Goal of Code Cognition

- A code maintainer tries to understand a program with a specific goal in mind.
  - Example 1: Debugging a program to detect the cause of a known failure
  - Example 2: Adding a new function to the existing program
- Identifying the goal can help in defining the scope of program comprehension.
- Scope of program comprehension: complete program or part of a program
- A program comprehension process is a sequence of activities that use **existing knowledge** about the program to generate **new knowledge** about it.
- Thus, program comprehension is a process of knowledge acquisition.

Knowledge

- Programmers possess two kinds of knowledge
  - General knowledge
  - Software-specific knowledge
- General knowledge: This covers a broad range of topics in computer systems and software.
  - Algorithms and data structures
  - Operating systems
  - Programming principles
  - Programming languages
  - Software architecture and design
  - Testing and debugging techniques

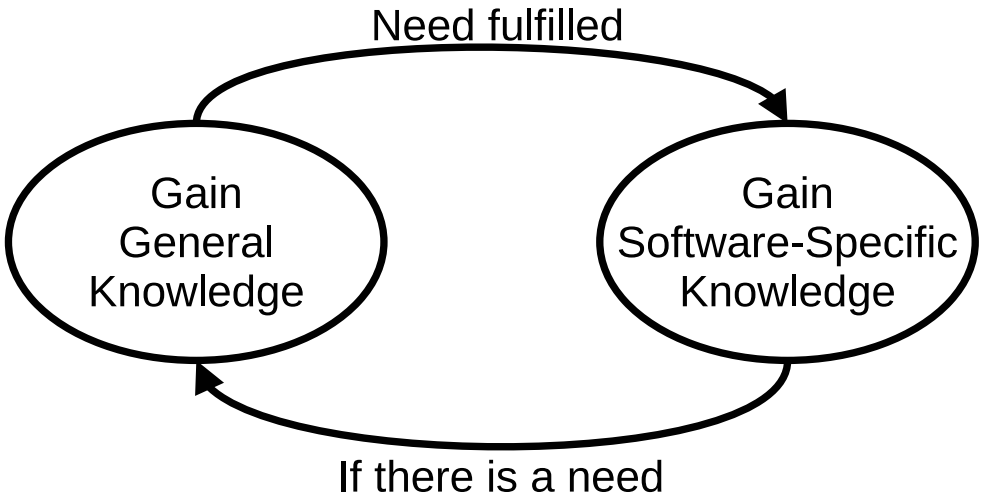
Knowledge

- Software-specific knowledge: This represents a detailed understanding of the software to be modified.
- Some examples of software-specific knowledge are
  - The software system has implemented public-key cryptography for data encryption.
  - The software system has been structured as a three-tier client-server system.
  - Module `x` implements a location server.
  - A certain `for` loop in method `y` may execute for a random number of times.
  - Variable `mcount` keeps track of the number of times module `z` is invoked.

Knowledge

- In the process of gaining new knowledge about a software system, a programmer learns the details of the following aspects:
  - Functionality
  - Software architecture
  - Control flow and data flow
  - Exception handling
  - Stable storage
  - Implementation details
- A programmer goes back and forth between acquiring general knowledge and software-specific knowledge, as illustrated in Fig. 8.1.

Knowledge


Need fulfilled
Gain General Knowledge
Gain Software-Specific Knowledge
If there is a need

Mental Model

- A mental model describes a programmer's mental representation of the program being comprehended.
- A mental model of a program is not unique; different programmers view and interpret a programmer in different ways.
- A programmer develops a mental model by identifying both **static** and **dynamic** elements of the program.
- Examples of program elements
  - for loop
  - A TCP (Transmission Control Protocol) connection
  - Over all control flow

Mental Model

- Static elements of a mental model
  - Text-structures
  - Chunks
  - Schemas
  - Plans
  - Hypotheses
- Dynamic elements of a mental model
  - Chunking
  - Cross-referencing
  - Strategies

Text-structures

- Code and its structure are known as text-structures.
- It is useful in gaining control flow knowledge in program understanding.
- A programmer can easily identify the following text-structures:
  - Loop constructs: for, while, and until
  - Sequences
  - Conditional statements: if-then-else
  - Variable definitions and initializations
  - Calling hierarchies within and among modules
  - Definitions of module parameters
- Understanding text-structures is the beginning of program comprehension.

Chunk

- A program chunk is a block of related code segment.
- Chunks enable programmers to create higher level abstractions from lower-level abstractions.
- Examples
  - A code block initializing a module's parameters tells the programmer about the nature of the parameters and their value ranges.
  - Understanding a for loop enables a programmer to create an abstraction of an internal functional step performed by the program.

Schema

- Schemas are generic knowledge structures that guide the programmer's interpretations, inferences, expectations, and attentions when passages are comprehended.
- The concept of programming plans correspond to the notion of schemas.

Plans

- A knowledge element is anything that is useful in understanding a program.
- Plans are broad kinds of knowledge elements used by programmers.
- Examples of knowledge elements
  - If the name of a function gives an indication of the activity performed by the function, then the function identifier is a knowledge element.
  - A block of comments describing a for loop
  - A `for` loop itself is a knowledge element.
  - A description of the problem domain of the program

Plans

- Example of plan
  - A doubly-linked list is an example of a plan; a designer has planned to implement certain concepts with this data structure.
- A plan is a kind of schema with two parts:
  - Slot type
    - Slot types describe generic objects.
    - Example: A tree data structure is a generic slot type.
  - Slot filler
    - Slot fillers are customized to hold elements of particular types.
    - Example: A code segment, such as a `for` loop's code is a slot filler.
- The programmer links the slot-type and slot-filler structures by means of the kind-of and is-a modeling relationships.

Plans

- There are two broad kinds of plans.
  - Domain plans
  - Programming plans
- Domain plans
  - These include knowledge about the real world problem, including the program's environment.
  - Example: If the software is for numerical analysis application, plans will include schemas for different aspects of linear algebra, such as matrix multiplication and matrix inversion.
  - Domain plans help programmers understand the code.

Programming Plans

- Programming plans are program fragments representing action sequences that programmers repeatedly apply while coding.
- Example: A programmer may design a for loop to search an item in a data set and repeatedly use the loop in many places in the program.
- Such a for loop is an example of a programming plan to implement the system.
- Programming plans differ in their granularities to support low level or high level tasks.

Hypotheses

- As programmers start reading code and the related documents, they start developing an understanding of the program to varying degrees.
- Programmers can test the results of their understanding as conjectures (aka hypotheses.)
  - Why: Why conjectures hypothesize the purpose of a program element.
    - Verification of a why conjecture enables a programmer to have a good understanding of the program element.
  - How: How conjectures hypothesize the method for realizing a program goal.
    - Given a program goal, the programmer needs to know how that goal has been implemented.
  - What: What conjectures enable programmers to classify program elements.
- A conjecture may not be completely correct.
- By continuously formulating and verifying conjectures, the programmer understands more and more code.

Dynamic Elements

- Chunking
- Cross-referencing
- Strategies

Chunking

- In a program, the **lowest level of chunks** are code segments.
- To understand a program in terms of its higher level functionalities, a programmer creates higher level abstraction structures by combining lower level chunks.
- This process of creating higher level chunks is called chunking.
- The process of chunking is repeatedly applied to create increasingly higher levels of abstractions.
- When a block of code is recognized, it is replaced by the programmer with a label representing the functionality of the code block.
- A block of lower level labels can be replaced with one higher level label representing a higher level functionality.

Cross-referencing

- Cross-referencing means being able to link elements of different abstraction levels.
- This helps in building a mental model of the program under study.
- Example:
  - Control flow and data-flow can be program elements at a lower level, whereas functionalities are higher level program elements.
  - There is a need to cross-reference between control-flow and data-flow elements and program functionalities.

Strategies

- A strategy is a planned sequence of actions to reach a specific goal.
- A strategy is formulated by identifying actions to achieve a goal.
- Example: if the goal is to understand the code representing a function, one can define a strategy as follows:
  - Understand the overall computational functionality of the function by reading its specification, if it exists.
  - Understand all the input parameters to the function.
  - Read all code line by line.
  - Identify chunks of related code.
  - Create a higher level model of the function in terms of the chunks.
- Strategies guide the two dynamic elements, namely, chunking and cross-referencing, to produce higher-level abstraction structures.

Understanding Code

- Two key factors influencing code understanding are:
  - Acquiring knowledge from code
    - Code is a rich source of information.
  - The level of expertise of the code reader
    - The level of expertise determines how quickly the code is understood.

Idaho State University
Computer Science

Acquiring knowledge from code

Several concepts can be applied while reading code in order to gain a high-level understanding of programs.

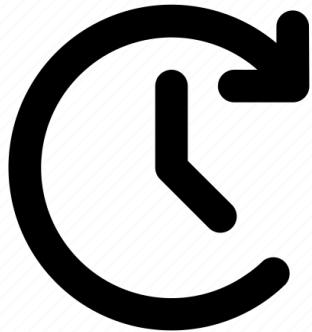
- Beacons
  - A beacon is code text that gives a cue to the computation being performed in a code block.
    - Example `swap()`, `sort()`, `select()`, `startTimer()`.
  - Code with good quality beacons are easier to understand.
- Rules of programming discourse
  - Rules of programming discourse specify the conventions, also called "rules," that programmers follow while writing code.
  - Some examples of rules are:
    - Function name: The function name agrees with what the function does.
    - Variable name: Choose meaningful names for variables and constants.
  - The rules set up expectations in the minds of a reader about what should be in the program.

Levels of expertise of code readers

- Expert programmers tend to possess the following characteristics:
  - **Organization of knowledge by functional characteristics**
    - Novice programmers tend to organize program knowledge in terms of program syntax.
    - Experts tend to organize knowledge in terms of algorithms and functionalities.
  - **Comprehension with flexibility**
    - Experts tend to generate a breadth-first view of the program, and keep adding useful details as more information is available.
  - **Development of specialized design schemas**
    - Design schemas are used to organize complex entities into constituents.

For Next Time

- Review EVO Chapter 8.1 - 8.2
- Read EVO Chapter 8.3 - 8.4
- Watch Lecture 21



Are there any questions?