Chapter 6

Concepts and Concept Location

Objectives

When programmers decide to implement a specific software change, the first step is to find which part of the code needs to be modified. Concept location is the technique that allows one to identify the code that needs to change. After you have read this chapter, you will know:

- The concepts and their importance in software change
- The concept name, intension, and extension
- Significant concepts of a change request
- Concept location by grep and by dependency search

***

To find what to change in small and familiar programs is easy, but it can be a formidable task in large software systems, which sometimes consist of millions of lines of code. *Concept location* is the phase that finds a code snippet that the programmers will modify; it follows the change initiation (see Figure 6.1).

To explain concept location, we have to understand that in this phase, the programmers deal with two separate systems: One is the *program*, and the other is the *discourse* about the program. This discourse consists of everything that stakeholders say or write about the program, and change requests belong to this discourse. The stakeholders are programmers, users, managers, investors, and other interested

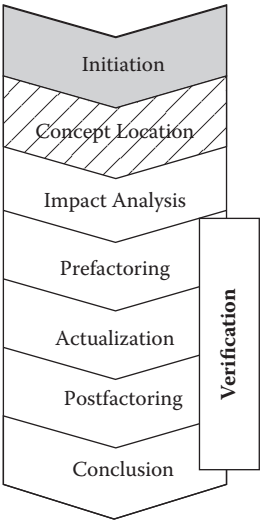

Figure 6.1 Concept location as part of software change.

parties. The bridge between these two worlds, the program on one hand and the discourse on the other hand, consists of *concepts*.

6.1 Concepts

The notion of a concept is used in linguistics, mathematics, education, philosophy, and other fields; it applies to the discourse about the software as well. Each concept has three facets, depicted symbolically by a triangle in Figure 6.2. One facet is a *name*, represented by the top vertex of the concept triangle; some concepts may have a single-word name like "payment," and other concepts have a more complex name that consists of several words, like "credit card payment." The use of concept

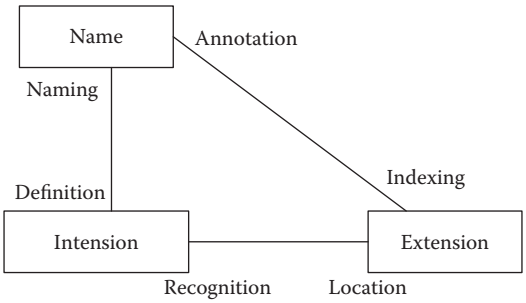

Figure 6.2 Concept triangle and directions of comprehension processes.

names allows people to communicate about the concepts of the world, of the program, or of the program domain.

The lower left vertex of the concept triangle represents the *intension* of the concept; it is the list of the concept properties and relations. For example, the intension of the concept named "dog" is, according to a Merriam-Webster dictionary, "a highly variable domestic mammal (*Canis familiaris*) closely related to the gray wolf."[1] Another name sometimes used for intension is "definition" or "account."

Spelling, Watchmaker Anecdote

The word "intension" is not misspelled; according to the Merriam-Webster dictionary:

> **Intension** \in-'ten(t)-shən\ synonym CONNOTATION, (a) the suggesting of a meaning by a word apart from the thing it explicitly names or describes, (b) something suggested by a word or thing — W. R. Inge> an essential property or group of properties of a thing named by a term in logic.[1]

The more common word with the same pronunciation but slightly different spelling has a different, although overlapping meaning:

> **Intention** \in-'ten(t)-shən\ synonyms INTENT, PURPOSE, DESIGN, AIM, END, OBJECT, OBJECTIVE, GOAL mean what one intends to accomplish or attain. INTENTION implies little more than what one has in mind to do or bring about <announced his intention to marry>. INTENT suggests clearer formulation or greater deliberateness <the clear intent of the statute>. PURPOSE suggests a more settled determination.[1]

Concept location is shorthand for "location of a significant concept extension in the code." The following anecdote illustrates the importance of concept location in a different setting that also deals with complexity:

> A customer asks a watchmaker to fix a watch. The watchmaker opens the watch, looks at the tiny wheels and screws with magnifying glass, then takes one of the small screwdrivers, tightens one of the tiny screws, and hands back the repaired watch. The customer asks: "How much is that going to cost?" The watchmaker answers: "$20." The customer starts complaining that to tighten one small screw cannot cost that much, to which the watchmaker answers: "To tighten the screw costs $1, but to know which screw to tighten costs an additional $19!"
>
> Many software changes are similar to the repair of the watch in this anecdote, and concept location is their most important part.

The *extensions* of a concept are things in the real world that fit the concept intension. Examples of an extension of a dog are real dogs like Fido, the pictures of dogs in family photos, Lassie from the TV series and movies, Buck in Jack London's book *Call of the Wild*, and so forth.

The relations among the vertices of the concept triangle are many-to-many. A name can mean several different intensions (homonyms), or the same intension can be named by several names (synonyms). Each intension or concept name can have several extensions, and each thing in the world may fit several intensions or several concept names.

The software code also contains the concept extensions; these extensions simulate the concepts of the domain. For example, the concept named "payment"

---

[1] With permission. From *Merriam-Webster's Collegiate® Dictionary, 11th Edition* ©2011 by Merriam-Webster, Inc. (www.merriam-webster.com).

belongs to the domain of point-of-sale; in the software system, it has an extension in the form of a statement `int paymt`. This variable `paymt` contains the amount a customer pays for the merchandise, and it is an extension of the concept "payment" the same way as a photograph of Fido is an extension of the concept "dog."

Figure 6.2 depicts not only the three facets of the concept, but also six comprehension processes. They are:

- *Location*: intension → extension
- *Recognition*: extension → intension
- *Naming*: intension → name
- *Definition* (or *account*): name → intension
- *Annotation*: extension → name
- *Indexing*: name → extension

Location is one of these six processes, and the purpose of the concept location is for a given concept intension to find the corresponding extension in the code. It is a phase of the software change; the modification of this extension will be the start of the change. Concept location is a prerequisite for the rest of the software change.

The other comprehension processes also may occasionally appear in a software engineering context. *Recognition* is the opposite of location, and it recognizes an extension in the code ("this code does credit card payment!") and finds the corresponding intension. *Naming* gives a name to an intension. The opposite process is *definition* that, for a given name, finds the corresponding intension. The relation between names and intensions is many-to-many; therefore, the naming and definition have to deal with homonyms and synonyms. Next, there is *annotation* that recognizes an extension and gives it a name. The opposite is *indexing* that for a given name finds corresponding extensions. All of these six processes are important when a programmer tries to understand the code, but concept location is the most important of them when dealing with the software change.

6.2 Concept Location Is a Search

The concept extensions in the program appear as variables, classes, methods, parts of a method body, or other code fragments. The programmers must find these code fragments. It can be easy in small programs or in the programs that the programmer knows well, or in special situations when the concept extension is obvious in the code, for example, when a concept extension is implemented as a whole class and the class has a name that corresponds to the name of the concept. However it can be hard when the program is large, the programmers are not familiar with it, or the concept extensions are implemented in an obscure way. For some software changes, it can be the hardest part of the change, a proverbial search for a "needle in a haystack."

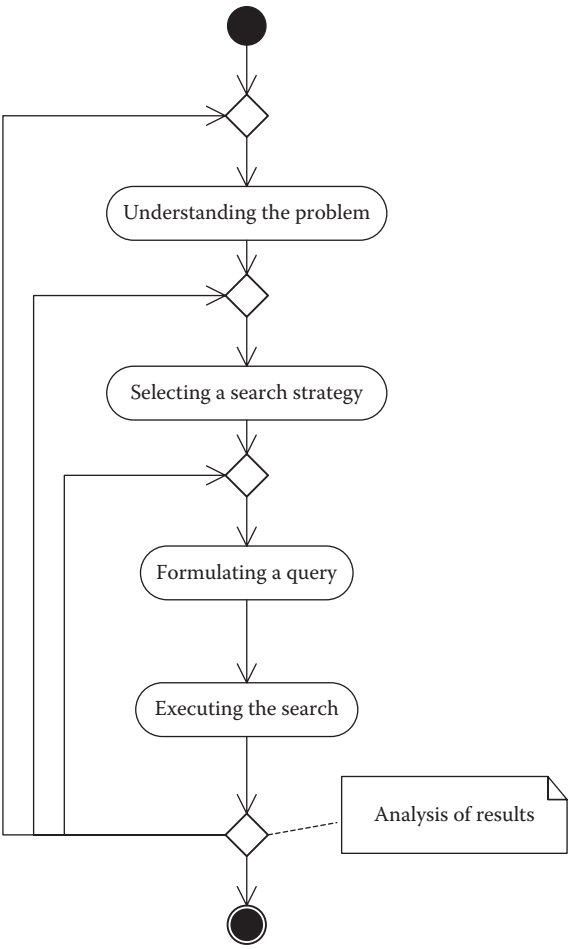

Figure 6.3 Search for concept location. From Rajlich, V. (2009). Intensions are a key to program comprehension. In Proceedings of IEEE International Conference on Program Comprehension (pp. 1–9). Washington, DC: IEEE Computer Society Press. Copyright 2009 IEEE. Reprinted with permission.

There are several concept-location techniques, and they share a common characteristic: The concept location is an interactive search process in which the search target is the code snippet that the programmers will modify in response to the change request. It is an interactive process, in which the programmers make decisions about how to conduct the search, and the tools play a supportive role. Searching for concepts in the source code is in many ways similar to searching for information on the Internet. The process of the search consists of the actions of Figure 6.3. The actions are:

- *Understanding the problem.* The programmers must understand the terminology used in the change request.
- *Selecting a search strategy.* The programmers choose the appropriate search strategy from the available assortment.
- *Formulating a query.* Each search requires a query that uses names of the concepts related to the change request.
- *Executing the search.* Programmers conduct the search with the help of supporting tools.
- *Analysis of results*. If the programmers find the concept location, the search ends successfully. However, if the search was unsuccessful, the programmers analyze the results and learn additional facts about the software system they are searching. They will use this new understanding to choose a better search strategy or to formulate a better query.

The programmers rarely find the significant concept extension with a single search; they often have to repeat the search several times.

The search starts with the concept intensions and ends with the corresponding concept extensions in the program. Figure 6.2 suggests two possible strategies of the search: Some strategies go directly from intensions to extensions, and the programmers use their understanding of concept intensions; others take a detour and identify the concept names first, and then look for these names in the code. The search space also may differ. The strategies search the static code, or external documentation, or preprocessed code, or execution traces.

The concept extensions that the programmers search for are either *explicit* or *implicit. Explicit concept extensions* directly appear in the code as fields, methods, code snippets, or classes. For example, the page count of a word processing document is explicitly present in the code as an integer, and the concept location is a search for this integer.

Implicit concept extensions, on the other hand, are only implied by the code, but there is no specific code snippet that implements them. Using the same word processor example, most word processors authorize any user to access any word processor file. The authorization to open files is an implicit concept extension that the source code does not explicitly implement; it is present in the code as an assumption that underlies the part of the code that is opening word processing files. To locate such implicit concept extensions, programmers must find related explicit concepts extensions that are close to these desired implicit concept extensions, or they have to find the code snippet that makes that implicit assumption. Some concept-location strategies are suitable for finding implicit concept extensions, while others are not.

6.3 Extraction of Significant Concepts (ESC)

A gateway to the search process is to formulate the query that will lead to the location of the concept. When programmers receive a change request, they analyze the change request and identify the appropriate concepts and their names. Concept names may appear in the change requests as nouns, verbs, or clauses. The change request can have many such nouns, verbs, or clauses. The programmers extract significant concepts (ESC) through the following steps:

- Extract the set of concepts used in the change request.
- Delete the *irrelevant concepts* that are intended for the communication with the programmers and do not deal with the requested functionality.
- Delete the *external concepts* that are unlikely to be implemented in the code, like concepts related to the things that are outside of the scope of the program or concepts that are to be implemented from scratch.
- Rank the remaining concepts—the *relevant concepts*—by the likelihood that they can be located in the code. The highest ranked concept is the *significant concept*.

As an example, suppose that the programmers deal with the point-of-sale system, and the change request is, "Implement a credit card payment." We can identify the following concepts: "Implement," "Credit card," "Payment."

The concept "Implement" is the communication with the programmers, unrelated to the code; therefore, it is not relevant. The concept "Credit card" is the new concept to be implemented from scratch; therefore, it is unlikely to be present in the current code. The relevant concept is the concept "Payment"; it is likely that some form of payment already exists in the old program, and this implementation is to be found in the program. It is the only relevant concept and, therefore, it is also a significant concept; its extension in the code is where the change will start. The software change will expand this extension to also include payment by credit card. The ESC process is presented in Figure 6.4.

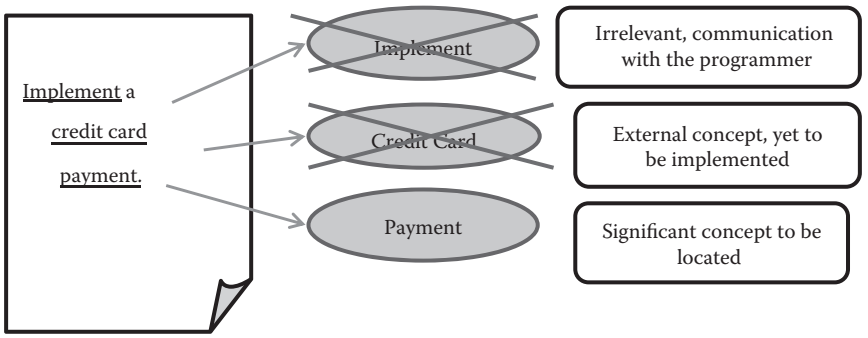

Figure 6.4 Extraction of significant concepts.

The process sometimes has to be expanded by the consideration of homonyms and synonyms. For example, the concept "Payment" can be called in the code by a synonym "Expense," "Imbursement," or various abbreviations like "Pmt" or "Paymt," and the query formulation must consider this. The "Payment" can also be a homonym for several intensions; the programmers look for the intension "Payment as an expense for goods or services" rather than "Payment as a punishment for a misdeed."

6.4 Concept Location by Grep

A popular concept location technique uses the tool "grep." It assumes that the concept extensions in the code are labeled by concept names; the concept name may be used as an identifier, or as a part of an identifier, or as a part of a code comment. The tool grep finds this concept name in the code, so in terms of Figure 6.2, the grep process follows the path intension → name → extension, and combines naming of the concept with the indexing of the name in the code.

The word "grep" is originally an acronym of the original "global/regular expression/print." The word "grep" outgrew its acronym status and programmers widely use it now both as a noun and as a verb.

Using grep, programmers formulate queries (i.e., regular expressions or "patterns"), then query the source code of a software system and investigate the results. If a line of code contains the specified pattern, it is called a "match." For example, when looking for the concept "payment," programmers may choose as a pattern to look for the original word ("payment") or various abbreviations and variants (pmt, paymt, pay_1, and so forth).

The grep tool produces a list of the code lines and the files where the specified pattern appears. There may be several matches; the programmers read the code that surrounds these matches and look for evidence of the presence of the concept extension. Based on this reading, they determine where the concept is located.

The query can fail in several different ways. One possible failure is that the set of matches is empty, and this indicates that the sought word is not used in the code. In another situation, the word is used in the code with a different meaning (as a homonym), and its occurrence does not indicate the significant concept location. In both of these cases, the programmers have to use a different query and repeat the search. In formulating such a query, they often use the new knowledge they learned while they inspected the results of the previous unsuccessful queries.

Sometimes the query produces too many matches, and it is not practical to go through all of them. In this situation, the programmers can formulate an additional query and do another search, this time searching only the set of the earlier matches. In this way, they get a set-theoretical intersection of the results of the two queries, and this resulting set of matches is smaller than the earlier large set.

The grep tool is quick and easy to use and, as such, it is very often used as the first strategy for concept location. However, a grep search often fails, and if the

query does not produce a result, it may not be clear how to continue. In addition, grep often fails in a search for implicit concepts; their names usually do not appear in the code because there is no code, identifier, or comment that indicates the presence of the concept extension. In the situations where grep does not work, programmers must use other concept location techniques.

The programmers, when writing the code, can make the life of future programmers easier by selecting good identifiers and comments in the code that will lead to the concept extensions; the naming conventions can greatly enhance or hinder the grep search. Selection of the appropriate naming conventions is one of the most important early decisions of the project team.

6.5 Concept Location by Dependency Search

Dependency search is another technique of concept location. It uses local and combined responsibility of program modules to guide the search. Section 4.3 explained the local and combined responsibility, and we will revisit them briefly here.

Program modules implement concept extensions, for example, the class `Pay` implements concept "pay by cash" and possibly some other concept extensions. All concept extensions that the class `Pay` implements are part of the *local responsibility* of class `Pay`. When the programmers search for the concept "pay by cash," they want to find the class that is responsible for its extension.

The direction of the search is guided by *combined responsibility*. Combined responsibility is the complete responsibility that the module assumes on behalf of its clients, and it includes not only the concept extensions implemented locally by class `Pay`, but also all concept extensions implemented in the classes of the supplier slice. Figure 6.5 shows the concepts that are a part of the local and combined responsibility of module X. In the figure, the extension of concept A is present in both local and combined responsibility of X, while the extension of concept B is present only in the combined responsibility. Still, the module X delivers both extensions of concept A and concept B to its clients.

The top module of the program has the whole program as its supplier slice. The client of the top module is the user, and all modules of the program are supplying various responsibilities to the top module. Hence, the top module is responsible for all concepts of the whole program, and they are part of its combined responsibility. However, the top module does not implement all these concepts, but rather delegates most of them to its suppliers. These suppliers in turn delegate them further to other suppliers deeper down in the class dependency graph, and so forth.

The programmer who searches for a significant concept needs to recognize the presence of the concept extensions in both local and combined responsibility. Figure 6.6 contains the activity diagram of the concept location by dependency search.

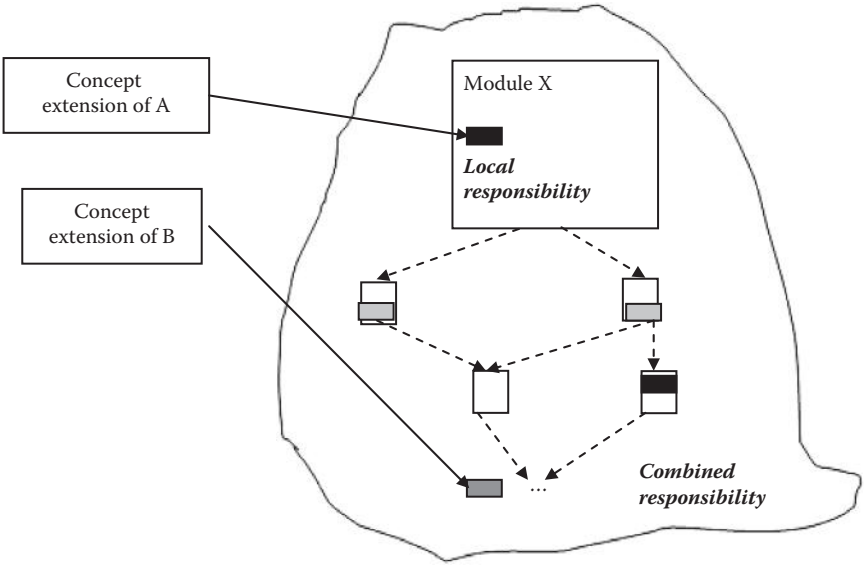

Figure 6.5 Concept extensions in the local and combined responsibility.

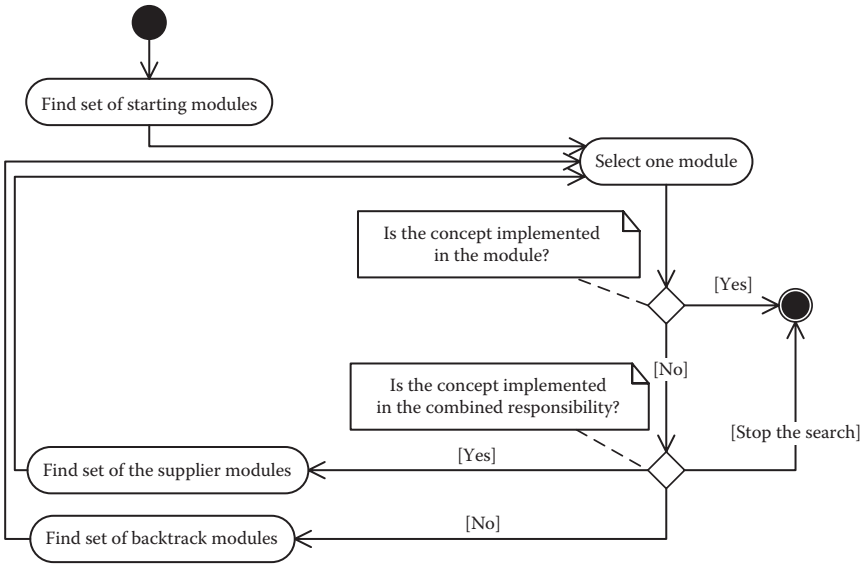

Figure 6.6 Concept location by dependency search. From Rajlich, V. (2009). Intensions are a key to program comprehension. In Proceedings of IEEE International Conference on Program Comprehension (pp. 1–9). Washington, DC: IEEE Computer Society Press. Copyright 2009 IEEE. Reprinted with permission.

The search starts with a set of possible starting modules. This could be the single top module, but in some situations, there are several top modules in the program, and the programmer selects the most relevant one.

Then the programmer decides whether the selected module implements the significant concept as its local responsibility. If this is the case, this class is the location of the concept, and the search ends successfully.

If the concept is not a part of the local responsibility, the programmer has to determine whether the concept is implemented in the combined responsibility. For that, the programmer uses various clues like the identifiers, comments, relations to other concepts, and so forth. Usually, there is no need to read the details of the code of the whole supplier slice; it is sufficient to make a rough guess. If the guess turns out to be wrong, it will lead to a later backtrack. This backtrack does not invalidate the search; it only makes it a little bit longer.

If the programmer concludes that the concept is implemented in the combined responsibility of the module, then the programmer selects the most likely supplier module and checks whether it contains the concept, and the search continues through the supplier slice of this module. However, if the programmer concludes that the concept is not present in the combined responsibility, it means that a wrong turn has been taken sometime before. The programmer then must backtrack to a previously visited client and take a different direction. If the programmer concludes that the search is leading nowhere, the search terminates unsuccessfully. The search must restart with another significant concept or another search technique.

Concept location by dependency search uses class dependency graphs; however, programmers often use the Unified Modeling Language (UML) class diagrams as a substitute. Strictly speaking, UML class diagrams contain ambiguities that can complicate the search. Nevertheless, because the search techniques allow the possibilities of backtracks that correct various errors, UML class diagrams often provide a sufficient support for dependency search. The next example of Violet illustrates a dependency search. An additional example appears in chapter 17.

6.5.1 Example Violet

Violet is an open-source UML editor that supports drawing several UML diagrams, including class diagrams. It contains approximately 60 classes and 10,000 lines of code (Horstmann, 2010). An example of the user interface of Violet is shown in Figure 6.7.

The change request is: “Record the author for each class symbol in the class diagrams.” After this change, Violet will provide information about the authors who created a class symbol to everyone who needs it.

The change request contains the concepts “record,” “author,” “class symbol,” and “class diagram.” Of these concepts, “record” is a communication with the programmer and is therefore, irrelevant; “author” is an implicit concept that is to be implemented from scratch and is unlikely to be present in the current code; “class

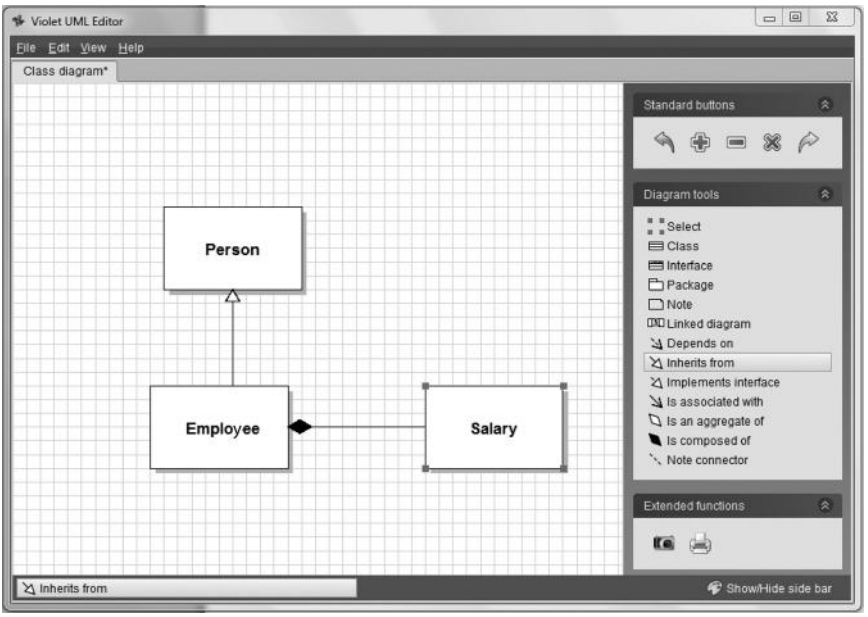

Figure 6.7 User interface of Violet.

diagram" is a background concept, but it is not the specific concept that needs to be changed. The "class symbol" is the significant concept the programmers will try to locate. A dependency graph of the top modules (classes) of the Violet code is in Figure 6.8.

The search starts at the top module `UMLEditor`. The inspection of this module revealed that it does not contain the significant concept within the local responsibility.

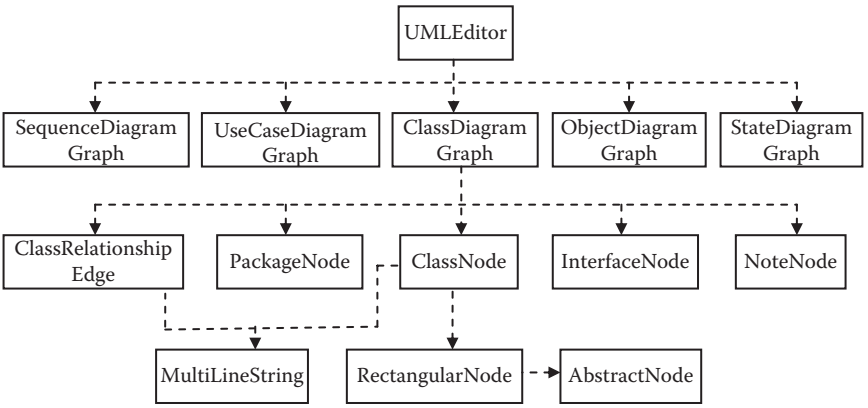

Figure 6.8 Partial class dependency graph of Violet.

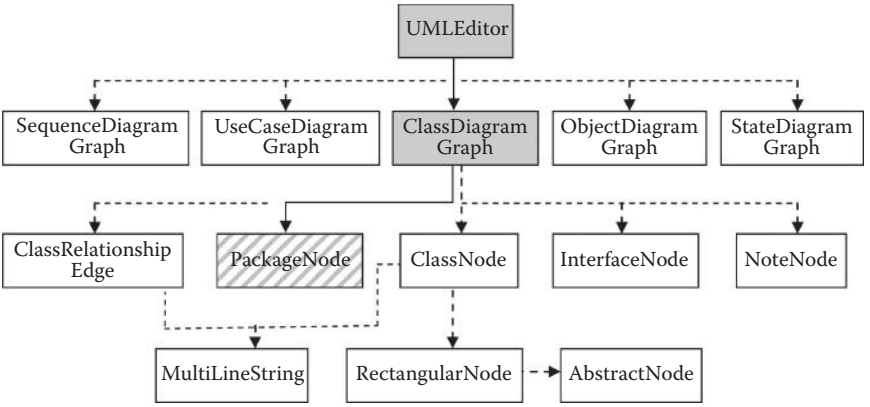

Figure 6.9 Wrong turn in dependency search.

There are several suppliers of this module, including `SequenceDiagramGraph`, `UseCaseDiagramGraph`, `ClassDiagramGraph`, and so forth. Of them, `ClassDiagramGraph` is an obvious choice for the inspection because it is most likely that the concept "class symbol" is in its combined responsibility.

The suppliers of `ClassDiagramGraph` are `ClassRelationshipEdge`, `PackageNode`, `ClassNode`, and several additional modules, some of them represented in Figure 6.8. The programmers inspected the module `PackageNode` but determined that the concept is not in its combined responsibility. Figure 6.9 shows the current state of the dependency search: Inspected modules where the programmer believes that the concept is in combined responsibility are gray; the module where the concept is not in combined responsibility is hatched.

The next step in the search is a backtrack to a previously inspected client that has the concept in its combined responsibility, in this case `ClassDiagramGraph`, and select a different supplier for the continuation of the search. The programmers selected `ClassNode` as the next supplier and determined that the concept is in both its combined and local responsibility; hence, the concept location has been found (see Figure 6.10). In it, the concept location is `ClassNode`, and previously inspected modules are filled with gray; an unsuccessful visit and inspection is represented by a hatched rectangle.

At this point, the search is complete and can stop. The significant concept has been located, and the change can be completed by adding the properties of an author to the module `ClassNode`. However, the class `ClassNode` inherits from class `RectangularNode`, which further inherits from `AbstractNode`. If the concept "author" is implemented there, the authorship of not only class symbols, but also package symbols, interface symbols, and so forth, will be recorded. This exceeds the change request and makes the change even more powerful and useful

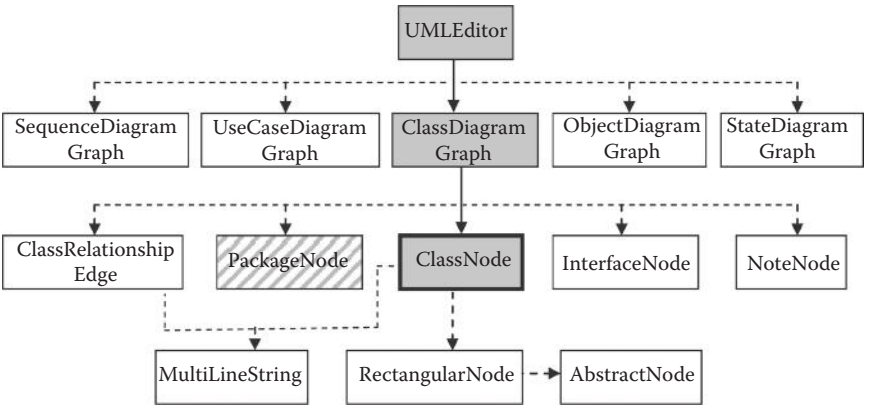

Figure 6.10 Location of concept "class."

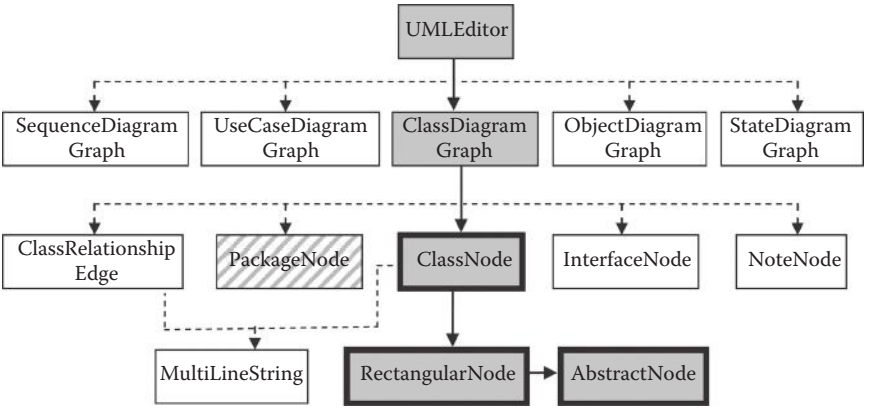

Figure 6.11 Alternative locations of the more abstract concept.

than the original change request. The final figure for this search with three options for the concept location is in Figure 6.11.

6.5.2 An Interactive Tool Supporting Dependency Search

An interactive tool takes over a part of this process. The UML activity diagram in Figure 6.12 contains two swim lanes: one for the tool (computer) and one for the programmer. It represents the cooperation of the computer and the programmer and captures their complementary roles. The computer finds all top modules of the program, finds the set of supplier modules, and keeps track of all previously inspected modules for a possible backtrack. The programmer decides the direction of the search.

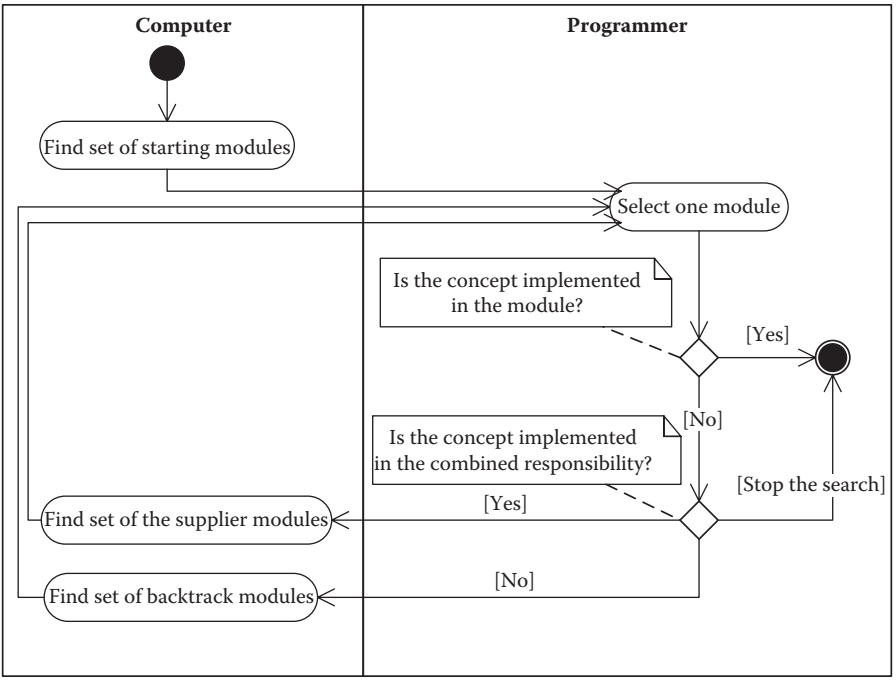

Figure 6.12 Concept location by dependency search using analysis tool.

Summary

Concepts are bridges between discourse about the program (that includes a change request) and the program itself. Concepts have three facets: name, intension, and extension. When a change request arrives, the programmers must find the significant concept and then its extension in the code; that is where the code modification will start. This process is called concept location. Different search strategies are available for concept location. The grep search strategy seeks the name of the concept in the identifiers and comments of the code and considers them as an indication of the presence of the concept extension. Dependency search utilizes local and combined responsibilities of the program modules and searches the code based on these responsibilities.

Further Reading and Topics

Classic literature that deals with concepts includes the work of Wittgenstein (1953), de Saussure (2006), and Frege (1964). The notions of extensions and intension used in this chapter were coined by Frege. How concepts apply to the issues of program

comprehension and concept location are discussed by Rajlich (2009) and by Rajlich and Wilde (2002).

Searches in an electronic environment have been discussed by Marchionini (1997). A grep search is described by Petrenko, Rajlich, and Vanciu (2008), and the original description of the dependency search was provided by Chen and Rajlich (2000). These two concept-location techniques, described in this chapter, do not require any special preprocessing of the code; therefore, they are easy to use, even when the code is incomplete. There is a large set of other techniques that do require code preprocessing; see, for example, information retrieval techniques similar to the ones that are used to search in a natural language text or on the Internet (Marcus, Sergeyev, Rajlich, & Maletic, 2004).

In the software engineering literature, some authors replace the word *concept* with the word *feature* and use these two words as synonyms. However, there is a subtle difference that was discussed by Rajlich and Wilde (2002): Features are special concepts that are controlled by the software users, who can choose whether or not to execute them. An example of a feature is "cut and paste" in a word processor. The users can choose to use it but do not have to; it is their decision whether this feature is executed. On the other hand, an example of a concept that is not a feature is the opening window of the word processor. Whenever the users run the word processor, the windows appears, so the users do not have control over this, and hence, it is not a feature. The difference between concepts and features is an important consideration when using the dynamic concept location that uses execution traces to locate a concept (Wilde & Scully, 1995).

Indexing links play a similar role as an index in a book (Antoniol, Canfora, Casazza, De Lucia, and Merlo, 2002): They list the concept names, and for each name, they have an address of the corresponding concept extension in the code. When indexing links are available, they make the process of concept location much easier. In the literature, indexing is often called traceability.

Information about the extraction of significant concepts from the requirements appears in the literature in many contexts. Abbott (1983) wrote an early paper on the topic about the use of extraction in a context of software specifications and design. However, for some changes, concepts related but not actually present in the change request must be used in the query, and hence, a broader context of the change request must be considered (Petrenko et al., 2008).

Note that the concept location does not have to identify complete concept extension in the code; it is sufficient to identify only a part. The rest of the concept extension and possible secondary code modifications are found by impact analysis, which is described in the next chapter. Also note that some authors deal with "concerns" that have a substantial overlap with notion of "concept" (Robillard & Murphy, 2002).

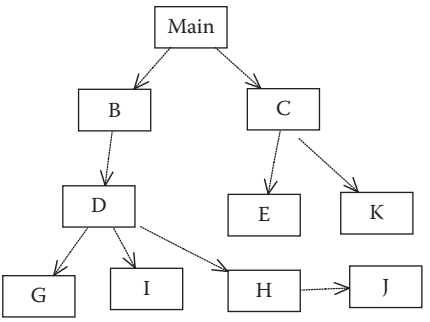

Figure 6.13 A class dependency graph.

Exercises

6.1 Find a significant concept in the following change request: "To the color palette in your application, add 'amber.'"

6.2 For the concept named "color amber," find intension and some real-world extensions.

6.3 What are the basic steps of the search for concept location? Draw the diagram.

6.4 Extract the significant concepts from the following change request: "The application allows the users to draw only two figures: circles and rectangles. Allow the user to draw triangles." Justify your answer.

6.5 What is the difference between grep and dependency search in concept location? What are the advantages and disadvantages of the two techniques?

6.6 Describe a situation when a grep search fails. What would you do if this happened to you?

6.7 Explain the difference between local and combined responsibility.

6.8 Which parts of the concept location by dependency search are done by the programmer, and which parts are done by the computer?

6.9 Suppose that your program has the class dependency graph of Figure 6.13. What is the first class you have to inspect during concept location by dependency search? Suppose that after inspecting a class, you always decide correctly which is the next inspected class. What is the maximum number of classes to be inspected?

References

Abbott, R. J. (1983). Program design by informal English descriptions. *Communications of the ACM, 26*, 882–894.

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., & Merlo, E. (2002). Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering, 28*, 970–983.

Chen, K., & Rajlich, V. (2000). Case study of feature location using dependence graph. In *Proceedings of International Workshop on Program Comprehension* (pp. 241–249). Washington, DC: IEEE Computer Society Press.

de Saussure, F. (2006). *Writings in general linguistics*. Oxford, UK: Oxford University Press.

Frege, G. (1964). *The basic laws of arithmetic: Exposition of the system*. Berkeley: University of California Press.

Horstmann, C. (2010). *Violet UML editor*. Retrieved on June 17, 2011, from http://sourceforge.net/projects/violet/

Marchionini, G. (1997). *Information seeking in electronic environments*. Cambridge, England: Cambridge University Press.

Marcus, A., Sergeyev, A., Rajlich, V., & Maletic, J. (2004). An information retrieval approach to concept location in source code. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering* (pp. 214–223). Washington, DC: IEEE Computer Society Press.

Petrenko, M., Rajlich, V., & Vanciu, R. (2008). Partial domain comprehension in software evolution and maintenance. In *Proceedings of IEEE International Conference on Program Comprehension* (pp. 13–22). Washington, DC: IEEE Computer Society Press.

Rajlich, V. (2009). Intensions are a key to program comprehension. In *Proceedings of IEEE International Conference on Program Comprehension* (pp. 1–9). Washington, DC: IEEE Computer Society Press.

Rajlich, V., & Wilde, N. (2002). The role of concepts in program comprehension. In *Proceedings of International Workshop on Program Comprehension* (pp. 271–278). Washington, DC: IEEE Computer Society Press.

Robillard, M. P., & Murphy, G. C. (2002). Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of International Conference on Software Engineering* (pp. 406–416). New York, NY: ACM.

Wilde, N., & Scully, M. (1995). Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice, 7*, 49–62.

Wittgenstein, L. (1953). *Philosophical investigations*. New York, NY: Macmillan.