The C++ SoftBench Class Editor

The C++ SoftBench class editor adds automatic code generation capabilities to the class graph of the SoftBench static analyzer. Novice C++ programmers can concentrate on their software designs and have the computer handle C++'s esoteric syntax. Experienced C++ programmers benefit from smart batch editing functionality and by having the computer quickly generate the program skeleton.

by Julie B. Wilson

The C++ SoftBench class editor allows the programmer to edit the class constructs in a C++ program using the SoftBench static analyzer's graphical interface. Using the class editor, the programmer can create and modify class hierarchies and edit class components.

Since the class editor is part of the static analyzer, let's look first at the functionality provided by the static analyzer. The static analyzer helps the programmer better understand the code. Through static queries, the programmer can understand a program's structure, assess the impact of changes, and change the architecture of the code when necessary. The static analyzer presents a wide variety of information about the code, including information about variables, classes, functions, and files. Through queries, the programmer can answer questions such as, "What functions and classes call this function?" or "What code accesses any element of this class?" The results of the queries can be displayed either textually or graphically. From either display, a simple double click takes the programmer directly to the source code that supports the displayed information.

To use the static analyzer, the programmer must first generate static information about the application. The default compile mode in the SoftBench program builder generates the static database (the Static.sadb file). When the programmer builds the application, the compiler places the static database in the directory in which the programmer compiled the code. All static queries rely on the information stored in this database.

Benefits of the Class Editor

SoftBench 5.0 adds editing capabilities to the class graph provided by the static analyzer. With the class editor, a novice C++ programmer can concentrate on software design, class hierarchy, data members, and member functions, not on C++ syntax. After each edit request, the class editor automatically generates the specified C++ code with correct syntax. The class editor also checks the work and doesn't let the programmer make typical beginner's mistakes like using the same class name twice.

Expert C++ programmers also benefit from the class editor. In addition to the program visualization capabilities of the graph, experts can quickly generate a program skeleton or make changes to an existing program's structure. Even more useful are the powerful, static-assisted edits that the class editor supports. Using the class editor, the programmer can change the name of a class or class member and all the appropriate changes are made in the source code. These changes can span many files. Because of the underlying static database, if the programmer changes the name of a member function x, the class editor knows exactly which instances of x are relevant and which instances are not.

Controlling Complexity

Fig. 1 shows an example of a C++ program with the classes and inheritance relationships displayed. The class editor provides the ability to display many relationships in addition to inheritance, such as friends, containment, and accesses by members of other classes.

Large C++ applications tend to have many classes and many relationships among the classes. The class editor provides several features to help control the complexity of what is displayed:

- Filters make it possible to display only the type of data in which the programmer is interested. For example, if the programmer only wants to see inheritance relationships, all other types of relationships can be filtered so they are not displayed on the graph.
- The programmer can reduce the complexity of the graph by hiding nodes that are not currently of interest.
- The programmer can add nodes to the graph directly by name or indirectly by querying about relationships with nodes already displayed on the graph.
- The programmer can expand and contract class nodes to show the data members and member functions in the node.

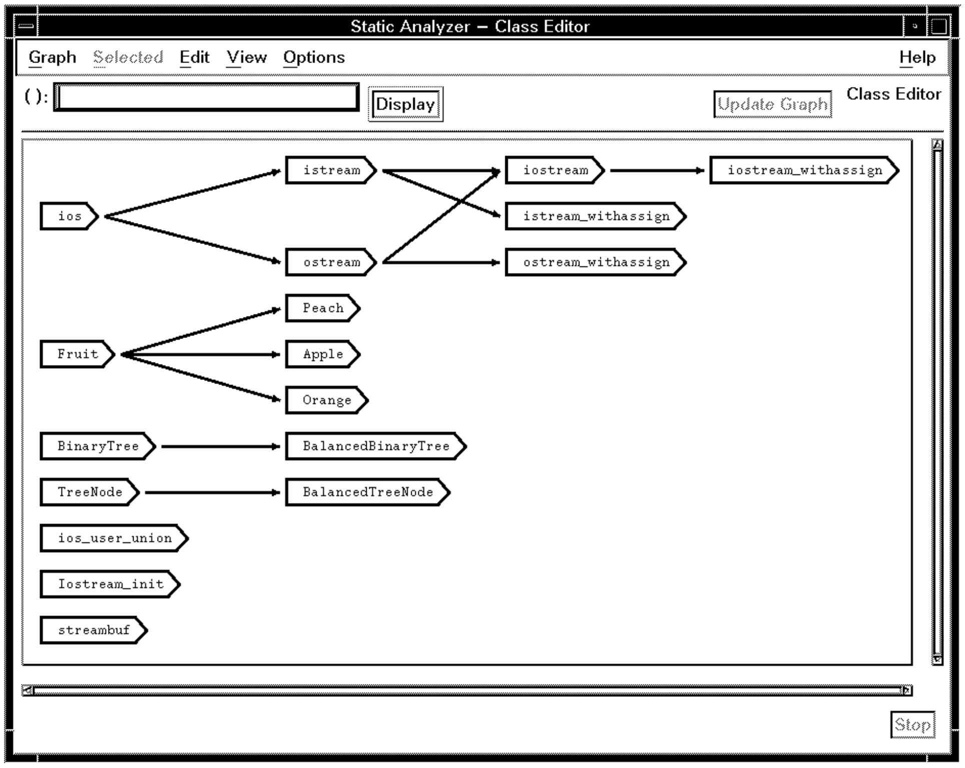

Fig. 1. *Class graph with all classes and inheritance relationships.*

Fig. 2 shows the same program that was represented in Fig. 1, but this time the visual display has been changed by filtering out all the classes from library header files. Additionally, two of the nodes have been expanded to show the member functions.

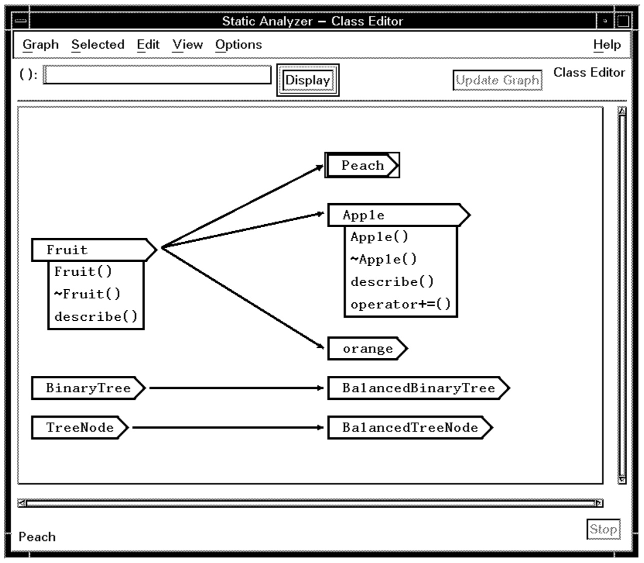

Fig. 2. *Simplified graph with only classes immediately under the programmer's control displayed and two nodes expanded to show the member functions.*

Changing the Class Hierarchy

Like any editor, the class editor allows the programmer to add, modify, and delete edited objects. For example, the programmer can add classes, inheritance relationships, member functions, and data members. Once these C++ structures exist, they can be modified or deleted. For example, the programmer can change an inheritance relationship from public to private or delete the relationship entirely.

If the programmer finds it necessary to restructure relationships by removing a class in the middle of an inheritance structure, the class editor makes the necessary edits to maintain the remaining relationships, as shown in Fig. 3. In this example, A is the base class of B, and B is the base class of C and D. Because the program architecture has been changed, the programmer no longer wants the B class. When B is deleted, the class editor automatically maintains the inheritance relationships so that A becomes the base class of C and D.

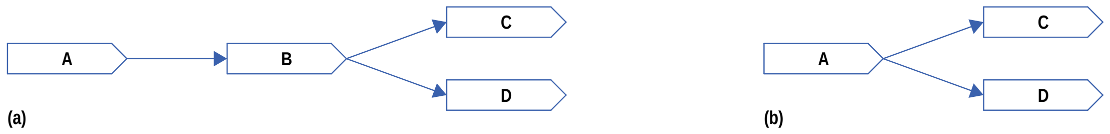

Fig. 3. *If the programmer removes a class in the middle of an inheritance structure, the class editor makes the necessary edits to maintain the remaining relationships. (a) Before the B class is deleted. (b) After the B class is deleted.*

Recovering from Editing Mistakes

The class editor remembers edit requests so that the programmer can undo them in reverse order. For example, if the programmer adds a base class relationship and then reconsiders, the Undo menu command on the Edit menu reads Undo Adding Inheritance.

Keeping the Static Database Up-to-Date

In SoftBench, compilations that produce static information are implemented with two parallel, independent build processes (see Fig. 4). The standard compiler, a cfront-based compiler, produces the error log and object (.o) files. The –y compiler option triggers the sbparse command, which is a subset of HP's ANSI C++ compiler. The sbparse command produces the static database, Static.sadb.

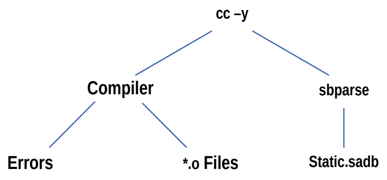

Fig. 4. *In SoftBench, compilations that produce static information are implemented with two parallel, independent build processes. The standard compiler produces the error log and object (.o) files. The sbparse command produces the static database, Static.sadb.*

The –nocode compiler option tells SoftBench not to run the cfront-based compiler. Since everything that the static analyzer knows depends on the underlying static database, each class editor edit request needs to update the static database. When the programmer requests an edit in the class editor, the class editor executes a compile with the –nocode –y compiler options, updating the static database without checking syntax and without producing .o files.

Using the Class Editor with a SoftBench Text Editor

The class editor saves after every logical edit. For example, if the programmer creates a new class, the underlying source code file changes when the programmer makes the request, and the class editor sends a FILE-MODIFIED message to let other tools know that the file changed.

If the programmer has a SoftBench text editor open while working in the class editor, the FILE-MODIFIED message causes the text editor to refresh the display of the file and the programmer can see the immediate propagation of the new source code.

Fig. 5 shows the sequence of events that occurs when the programmer makes an edit using the class editor:

1. The class editor performs pre-edit checks to make sure that the edit makes sense. Assuming that the request passes the pre-edit checks, the edit is displayed on the graph.

2. The class editor updates the underlying files that are impacted by the request.

3. The class editor sends a FILE-MODIFIED message to notify other tools that the edit took place.

4. The class editor executes a compile with –nocode –y options, which updates the Static.sadb file.

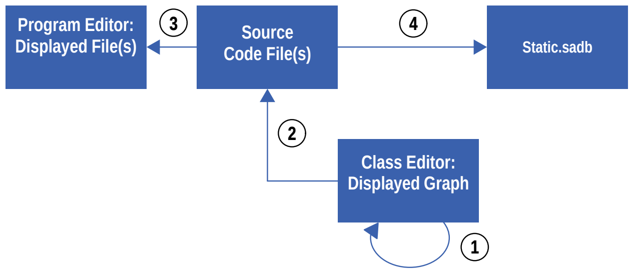
Program Editor: Displayed File(s)
3
Source Code File(s)
4
Static.sadb
2
Class Editor: Displayed Graph
1

Fig. 5. *Sequence of completing a class editor edit. ① Edit displayed on graph. ② Files updated. ③ FILE-MODIFIED message results in redisplay of file in editor. ④ A compile with –nocode –y options updates the database.*

If the programmer chooses to make edits in the text editor, the sequence of events is slightly different (see Fig. 6):

1. When the programmer saves the file, the text editor updates the underlying file and sends a FILE-MODIFIED message.
2. The class editor receives the FILE-MODIFIED message and posts an information dialog box stating Undo disabled due to external edit. The class editor then erases the undo stack, since the external edits may have made the undo actions invalid.
3. The code changes in the text editor are not immediately propagated back into the class editor. The programmer must initiate the action that updates the static database and the graphical display. To update the static database, the programmer chooses the File: Analyze File Set menu command on the main static analyzer window. This menu command executes a –nocode –y compile.
4. After updating the static database, the programmer needs to select the Update Graph button in the class editor to display the code changes made in the text editor.

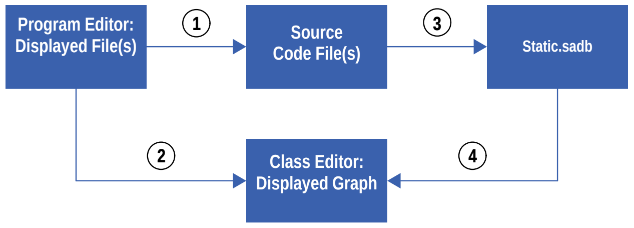
Program Editor: Displayed File(s)
1
Source Code File(s)
3
Static.sadb
2
Class Editor: Displayed Graph
4

Fig. 6. *Sequence of updating the class editor after an external edit. ① Files updated. ② FILE-MODIFIED message disables the undo stack. ③ An Analyze File Set menu command triggers a compile that updates the database. ④ An Update Graph command displays the external edits on the graph.*

Working with Configuration Management

Edits in the class editor have the potential to change many files. For example, if the programmer changes the name of a class, several files may need to change. With the powerful, static-assisted editing, the programmer may not be aware of which files are changing. Consequently, the programmer can attempt to initiate edits on files that do not have write permission.

When the class editor runs into a problem with file permissions, it posts a dialog box giving the programmer three choices:

- Let the class editor check out the necessary files. This option is only valid if the files are under configuration management and available for checkout. The class editor completes the checkout process by sending a VERSION-CHECK-OUT message.
- Resolve the problem manually, then select Retry on the dialog box.
- Cancel the edit.

Fixing Compile Errors

The class editor does not introduce compile errors when it creates code. However, it is possible for the programmer to introduce compile errors. For example, the programmer might reference a function before creating it, make a typing error on a variable name or type when adding a data member, or make a syntax error in the body of a member function. Neither the class editor nor sbparse catches syntax errors of this type.

At first this model may appear surprising, but it actually works to the user's advantage. When the programmer uses a traditional text editor, code is not always compilable as it is being developed. The programmer may frequently create code objects out of order, mentally keeping track of what still needs to be done. The class editor functions in much the same way. If it detected every compilation problem, work would soon grind to a halt. Instead, the programmer can complete the code development tasks and let the compiler catch the syntax errors later.

Preserving White Space and Comments when Editing

The algorithm for completing an edit allows the class editor to preserve spaces, tabs, and comments in the code being edited. When the programmer specifies an edit, the static database provides the class editor with the specific positions in the source that need to be edited. The source code is then searched for "landmarks" to ensure that the right part of the code is being changed. Only minimal additions, substitutions, and deletions are done to the source file. For example, when a class is renamed, each reference is replaced by the new name, leaving any user-added comments or white space intact.

When more complicated things are changed, like the return type of a function, several consecutive tokens may be replaced with new text. In this case, any comments that are between the old tokens for that type are lost.

Troubleshooting

The error Unable to update the database is fairly common. It tends to occur with existing code that has compile errors, and it usually indicates a missing include file. To avoid this error, the programmer should make sure that existing code compiles without errors before starting to use the class editor.

Much more rarely, timing problems are encountered. When the programmer requests an edit, the first step is to make the edit visible on the graph, and the last step is to update the database (see previous discussion under "Using the Class Editor with a SoftBench Text Editor"). Because the class editor allows the programmer to begin the next edit as soon as the previous edit is visible on the graph, it is possible to experience a race condition. If the database is not yet up-to-date when the class editor attempts to complete its pre-edit checks for the next edit, the programmer will get an error message. For example, if the programmer creates a class, then attempts to add a member to the class before the create class edit is complete, the error Class <class name> not found will be issued. To resolve this error, the programmer should wait a moment and try again.

Conclusion

The static analyzer and the class editor together offer the C++ programmer a powerful program visualization and editing tool. The editing capabilities of the class editor facilitate program construction and editing. The code generation capabilities of the class editor facilitate program correctness and consistency. Code generated by the class editor is syntactically correct and consistently formatted. When the programmer makes a mistake using the the class editor, one or more edits can easily be backed out using the Edit: Undo menu command.

The filtering capabilities of the static analyzer allow the programmer to control the complexity of what is displayed and to conceal irrelevant details easily. The visualization capabilities of the static analyzer aid program comprehension. The programmer can choose to investigate many types of relationships in the code, and can easily access the underlying source code when more detail is needed.

Acknowledgments

The author wishes to acknowledge Wade Satterfield, the R&D engineer who developed the class editor, for his technical input and review of this article. The author also wishes to thank Carolyn Beiser, Jack Walicki, and Jerry Boortz for reviewing this paper and providing helpful suggestions.

Reference

1. F. Wales, "Theme 4 Discussion Report," *User-Centered Requirements for Software Engineering Environments*, Springer-Verlag, Nato Scientific Affairs Division, 1994, pp. 335-341. This article presents tasks to be facilitated. The tasks mentioned in the conclusion above are based on this task list.

▶ Go to Next Article
▶ Go to Journal Home Page