Incremental Change in Object-Oriented Programming

Václav Rajlich and Prashant Gosavi, *Wayne State University*

Incremental change adds new functionality and new properties to software. It's an essential part of software processes such as maintenance, evolution, iterative development and enhancements, and agile development. Because of that, it plays an important role in practical software engineering.

In this article, we concentrate on IC in the context of object-oriented Java programs and study selected IC activities. Because change requests are formulated in terms of program concepts—chunks of knowledge about the program and application domain—the resulting IC activities also focus on program concepts. Moreover, most change requests originate from the end user, so the end user's view of the program is the source of most concepts that govern IC. Additional concepts might originate from other programmers and deal with the program architecture or algorithms.

Program dependencies also play a crucial role in IC; if a program component changes, other dependent components might also have to change. This is true even for well-designed object-oriented programs, as this article's example demonstrates.

Incremental changes introduce new functionality and new properties to software. Here, an example presents IC activities in which program concepts and dependencies play a key role.

IC at work

We illustrate IC in an example involving Drawlets (www.rolemodelsoft.com/drawlets), an open source framework that adds graphics to a host application. (See the "Previous Work" sidebar for background information and motivation.) Drawlets supports a drawing canvas that holds figures and lets users interact with them. The figures include lines, freehand lines, rectangles, rounded rectangles, triangles, pentagons, polygons, ellipses, and text boxes. Tools modify the figure attributes, such as size and location. The host application provides an instance of the drawing canvas, toolbars, and tool buttons.

0740-7459/04/$20.00 © 2004 IEEE

Drawlets is well-designed, with more than 100 classes, 35 interfaces, and 40,000 lines of code and documentation. Kent Beck and Ward Cunningham developed the original framework, named HotDraw, in Smalltalk, and RoleModel Software later ported it into Java and renamed it Drawlets.

In this example, we used a host application called SimpleApplet. Class `SimpleApplet` implements this host application and is a part of the Drawlets library. SimpleApplet runs in any browser that supports Java. It displays the drawing canvas in a browser window and arranges the toolbars and tool buttons around it. These tool buttons activate tools that create or modify figures. Figure 1 shows the SimpleApplet canvas with a selection of different shapes.

For this example, the change request is to implement an *owner* for each figure. An owner is the user who put the figure onto the canvas, and only the owner should be allowed to modify a figure. A user inputs his or her ID and password at the beginning of a session. Any attempt to change a figure must validate that the figure owner and the current session owner are the same.

This change will make SimpleApplet more versatile and will let several users work collaboratively on a single document. IC design is our response to the change request.

IC design

IC design responds to the change request by performing certain activities—*concept extraction*, *location*, and *impact analysis*—before the actual change starts. Both programmers and project managers participate in the design to better understand the planned IC and required resources.

Concept extraction

As mentioned earlier, concepts are chunks of knowledge about the program and application domain. In the natural language used in change requests, they appear as nouns, verbs, or short clauses. Concepts let programmers and users communicate about the program and are also present in the code itself. An identifiable code fragment, very often a class, might implement them. In other situations, they appear implicitly as simplifications and assumptions reflected in certain code segments. Change requests typically contain several concepts, so a programmer

Previous Work

A complete version of the Drawlets example appears elsewhere.[1]

The article, "The Staged Model of the Software Lifecycle," summarizes the role of evolution in the software lifecycle.[2] Iterative change and refactoring (change that doesn't modify the program's behavior but modifies the program structure and makes it more suitable for future evolution) are software evolution's main tasks. Martin Fowler has given a comprehensive list of refactoring transformations.[3]

Evolving interoperation graphs[4] are a theoretical model of change propagation and are the basis for the figures that appear in this article.

Concept location, impact analysis, and change propagation all require knowledge of class dependencies. Program analysis is the discipline that develops algorithms that extract class dependencies from a program. You can access a summary of research on impact analysis and analysis of program dependencies elsewhere.[5]

A sizable literature exists on software design that anticipates future changes and prepares software for them.[6] However, frequently, for various reasons, the IC isn't anticipated and affects many classes. Unanticipated IC, despite being a very common software process, hasn't attracted sufficient study. As a result, it's not part of the textbooks, lacks an established terminology and tool support, and is largely a self-taught art that successful programmers must discover on their own.

References

1. P. Gosavi, *Example of Change Propagation*, master's thesis, Dept. Computer Science, Wayne State Univ., 2003; www.cs.wayne.edu/~gosavip/thesis/thesis_index.html.
2. V.T. Rajlich and K.H. Bennett, "The Staged Model of the Software Lifecycle," *Computer*, July 2000, pp. 66–71.
3. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
4. V. Rajlich, "Modeling Software Evolution by Evolving Interoperation Graphs," *Annals of Software Eng.*, vol. 9, 2000, pp. 235–248.
5. S.A. Bohner and R.S. Arnold, *Software Change Impact Analysis*, IEEE CS Press, 1996.
6. D.L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Comm. ACM*, vol. 29, 1972, pp. 1053–1058.

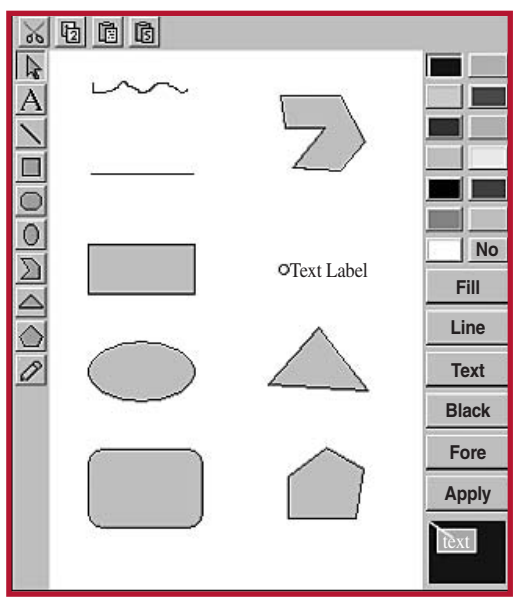


Figure 1. Screen shot of SimpleApplet.

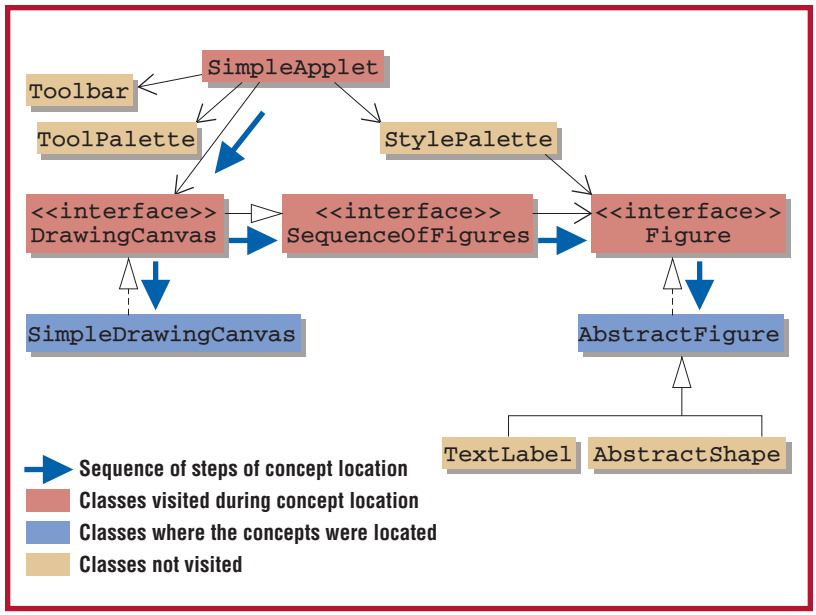


Figure 2. Summary of concept location.

must extract the ones that best characterize the change.

Coming back to the Drawlets example, the change request contains the concepts User, Figure, Canvas, Modify Figure, User ID, User Password, Beginning of a Session, Validate Owner, and so forth. The concepts most relevant to our IC are Figure Owner and Current Session Owner. Both of these concepts are implicit in the old code, which assumes that just one owner owns both sessions and figures.

Location

The next IC activity is to locate the relevant concepts in the code.

Concept location has been the subject of previous studies and several methodologies are available, including string pattern matching that finds relevant identifiers or comments,[1] a dynamic testing technique,[2] and static code search.[3] Each technique has its area of applicability (a discussion of these techniques' advantages and disadvantages appears elsewhere[4]).

In our example, we located the concepts by static code search. We used this technique rather than the others because both of the sought concepts are implicit, so no identifiers exist to point to their locations and you can't identify them by testing the current code. For implicit concepts, static code search is the only suitable technique. The search space is the class diagram, and the programmer conducts the search depth-first by going top-down through control flow dependencies.

The programmer first visits the top class (containing method `init` or `main`), which summarizes the program's functionality. This top class doesn't do everything; it delegates parts of its functionality to other classes. Hence, if the top class doesn't implement the sought concept, the classes it calls must implement the concept. Because these classes are specialized, it's usually easy to decide which ones do and don't lead toward the sought concept. Moving down the call graph toward increasingly specialized classes, the programmer eventually finds the classes that participate in the concept. If more than one concept is involved in the change, the programmer repeats the search until he or she locates all concepts.

Figure 2 gives a UML class diagram[5] of selected top classes of Drawlets that we searched for the Figure Owner and Current Session Owner concepts. Bold blue arrows indicate the search's process, which starts in the top class `SimpleApplet`.

The class `SimpleApplet` is responsible for the whole program. However, after briefly reading the class's code, we determined that it doesn't contain any session or figure attributes, and thus contains neither Current Session Owner nor Figure Owner. We concluded that these concepts must be located in the classes `SimpleApplet` uses. Class `StylePalette` supports a selection of different styles for figures, and we concluded that it couldn't contain the concepts. Similarly, the concepts couldn't be located in `Toolbar` or `ToolPalette`; thus, they must be located in the classes implementing the interface `DrawingCanvas`.

`DrawingCanvas` is an interface and the class `SimpleDrawingCanvas` implements it. There is one and only one instance of `SimpleDrawingCanvas` for each session of Drawlets; hence, most of the session attributes are concentrated in this class and it's the logical location of the implicit concept of Current Session Owner.

We then searched for the Figure Owner concept's location, repeating the same original steps until we reached the interface `DrawingCanvas`. The interface `DrawingCanvas` contains many figures but doesn't contain properties of individual figures; so we visited the interface `SequenceOfFigures`. This interface is still only a collection of Figure objects and also doesn't contain individual figure properties. So, we next visited the interface `Figure`,

implemented by class `AbstractFigure`.

Class `AbstractFigure` holds most of a figure's attributes. It's also the topmost class in the Figure hierarchy. We concluded that the implicit concept of Figure Owner is located in this class.

Impact analysis

Impact analysis determines a change's extent. It determines the set of classes IC affects (the *impact set*) and presents an opportunity to select a strategy for the change. It also gives the programmer an opportunity to understand the relevant code and dependencies that propagate the change.

In an ideal situation, IC is limited to one class only, and the information hiding and abstraction that the class implements are barriers to propagating the IC to other classes. However, even in well-designed systems, some ICs cross these barriers. Our explanation is that it's impractical to encapsulate all imaginable program concepts because there are simply too many. This is particularly true about implicit concepts; the code is based on many assumptions and it's impractical and unlikely that they all will be encapsulated. If a change request is based on an unencapsulated concept, it's likely to affect several classes that participate in that concept.

Impact analysis starts with the classes found in concept location. These are the first classes that the programmer adds to the impact set. For every new class added to the impact set, the programmer visits all classes interacting with it. If the IC will change these classes, they're also added to the impact set. The programmer continues this process recursively until he or she identifies all classes of the impact set.

Different kinds of class interactions have different likelihoods of change propagation. For example, a change in a derived class is less likely to trigger a change in the base class, while a change in a base class is more likely to require changes in derived classes. However, a safe view is that all classes that have a nonzero probability of change must be visited and either added to the impact set or certified that they don't need change.

In some cases, information passes from class A to class B through an intermediary class X. In this case, a change in A might trigger a change in B while the intermediary class X remains unchanged. We call this *hidden propagation*.[5] The existence of hidden propagation has the following consequence: if during impact analysis we encounter a class that doesn't change, we still must investigate whether there's a potential for hidden propagation. If there is, we must visit the neighbors of this class even if the class itself didn't change.

In our example of Drawlets, we knew that `SimpleDrawingCanvas` and `AbstractFigure` would change, so we put them into the impact set. We then iteratively visited the classes and interfaces that interact with these two classes (see Figure 3).

The only interface that interacts with `SimpleDrawingCanvas` is the interface `DrawingCanvas`. We determined that it doesn't need any change, but it could propagate the changes to the classes and interfaces that interact with it. Of these, the class `CanvasTool` and the interface `SequenceOfFigures` don't need modifications and don't propagate this change any further. However, the class `SimpleApplet` needs a change and so we added it to the impact set.

Next, we visited the classes that interact with `SimpleApplet` (`Toolbar`, `ToolPalette` and `StylePalette`), but none of them required a change and there was no possibility of propagation.

Starting from `AbstractFigure`, we visited the interface `Figure`, which interacts with the class `AbstractFigure`, and added it to the impact set. We visited the classes `AbstractShape` and `TextLabel`, added them to the impact set, and propagated the change to the classes beyond them as far as necessary. Figure 3 illustrates the complete impact set and the extent of the change.

IC implementation

Implementing IC follows the design phase and involves *actualization*, *incorporation*, and *change propagation* activities.

Actualization

During actualization, the programmer writes the code that will implement the new concept. For a small change, the new code might be part of an existing class. A larger change could require implementing a completely new class or a new subsystem consisting of several new classes.

In our Drawlets example, we implemented the

Different kinds of class interactions have different likelihoods of change propagation.

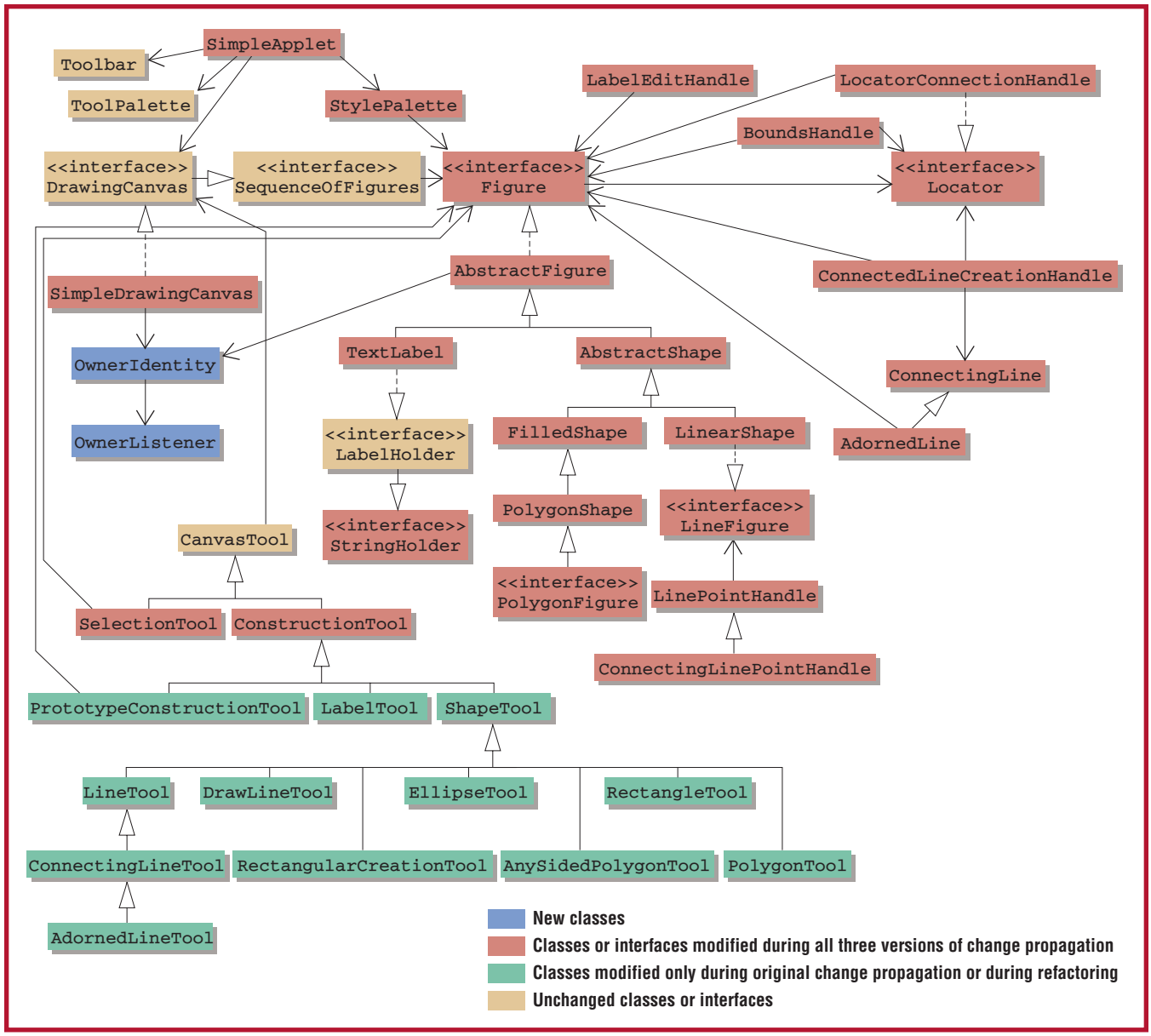


Figure 3. Drawlets classes involved in incremental change (IC).

concept Owner in the new class `OwnerIdentity`. It holds basic owner data, such as owner ID and a password. A dialog box lets users change the current session owner. It also contains a listener that notifies the program when the user clicks on a button in this box. We implemented this listener in the class `OwnerListener`.

Incorporation

Incorporation interconnects the new code with the old. If the new code is a class, we can achieve incorporation by declaring instances of the new class in the old code or vice versa.

In our Drawlets example, we incorporated the newly created classes `OwnerIdentity` and `OwnerListener` into the old code by creating an instance of `OwnerIdentity` in both `AbstractFigure` and `SimpleDrawingCanvas`. That required changes in these classes' methods.

In the class `AbstractFigure`, we added a new argument for figure owner to several existing public methods. After the modifications, these methods first compared the identities of current session owner and figure owner before making changes to the figures. We also modified several other methods in this class that called the above methods.

In the class `SimpleDrawingCanvas`, we modified the function `cutSelections(...)`

to prevent figures not belonging to the current session owner from being cut from the canvas. We implemented several new methods to allow access to the current session owner information stored in class `SimpleDrawingCanvas`.

Figure 4 shows the situation after incorporation; the new classes `OwnerIdentity` and `OwnerListener` have been added, and we've modified the classes `SimpleDrawingCanvas` and `AbstractFigure`. Bold blue arrows indicate inconsistencies in the program and the direction in which they propagate.

Change propagation

This activity parallels impact analysis, but this time it involves real code changes. An interesting case study discovered that programmers often underestimate the impact set.[6] So, programmers should be open to the possibility that they might have missed some propagations during impact analysis. Also, new unanticipated effects of the actual changes could emerge during this phase.

In our Drawlets example, incorporation changed the classes `SimpleDrawingCanvas` and `AbstractFigure`. So, they are the starting points for the change propagation. Because change propagation parallels impact analysis, we don't repeat the individual steps here. The complete set of classes changed in the change propagation is equivalent to the impact set (see Figure 3).

Shorter propagation

Change propagation is a long, difficult, and error-prone IC activity. A change in any class requires a thorough retesting of the class and increases the risk that the code will unexpectedly break. Here, we present two separate and mutually independent techniques that shorten change propagation. We applied them before the incorporation and change propagation took place.

Refactoring

To restructure the program architecture without changing the program's functionality we applied *refactoring*. A comprehensive overview of refactoring techniques appears elsewhere.[7] Here, we apply several of these techniques.

During impact analysis, the programmer might find misplaced code that unnecessarily extends the impact set. In this case, he or she can improve the program architecture by moving the misplaced code into its proper place by refactoring. This results in shortened change propagation.

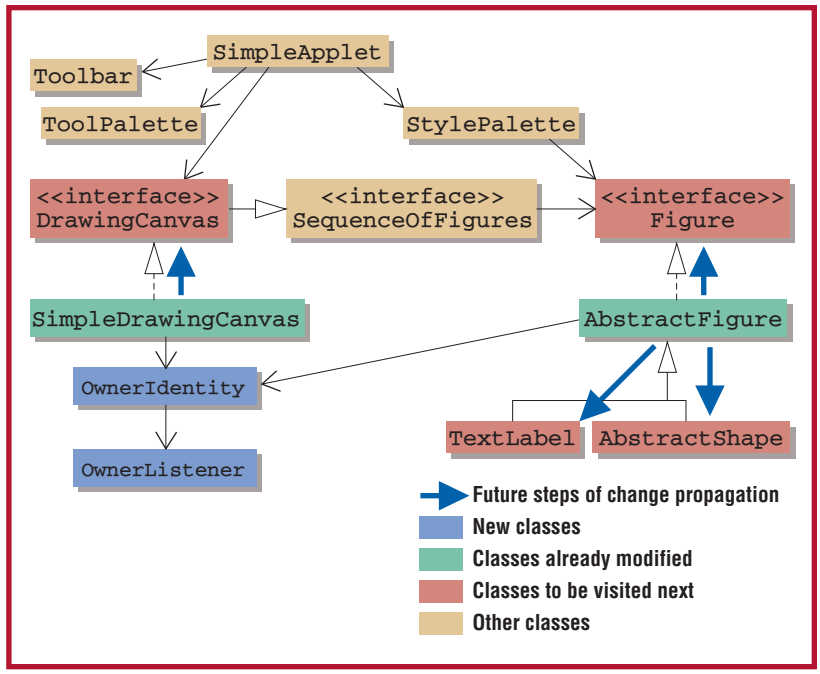


Figure 4. The program after actualization and incorporation.

In our Drawlets example, we noticed that several classes that create new figures have the method `basicNewFigure(...)`. This method has two main parts:

- Create a new figure.
- If the figure was not created at its intended location, move it by calling the method `move(...)`.

Drawlets creates the figures Ellipse, Rectangle, and several other figures in this way. When the method `move(...)` changed, the classes that create these figures also had to change.

We could improve the software architecture by placing this functionality in a base class, as it's shared among several derived classes. This would prevent unnecessary code repetition and would also shorten change propagation. To accomplish this, we used the following refactoring steps:[7]

- *Extract Method*. First, we extracted the call to method `move(...)` from `basicNewFigure(...)` and put in a new method `move_new_figure(...)`.
- *Pull up Method*. Next, we pulled up method `move_new_figure(...)` into the base class ConstructionTool.

Table 1

The number of classes and interfaces modified by incremental change

Technique used in IC	Classes modified	Interfaces modified
Incorporation and propagation only	36	5
Refactoring, incorporation, and propagation	36	5
Refactoring	12	0
Incorporation and propagation after refactoring	25	5
Splitting, incorporation, and propagation	24	5
Splitting	5	0
Incorporation and propagation after splitting	20	5

- *Inline Method*. Then, we merged method `move_new_figure(...)` into method `ConstructionTool::newFigure(...)`. This refactoring improved the code by removing redundancies and shortening the change propagation, as the data in Table 1 and Figure 3 show.

Splitting the roles

Another technique that can shorten change propagation is *splitting the roles*. We can apply this when the code uses the same class or same method in two slightly different roles. Because the roles are similar, the same code can do both jobs.

The propagating change, however, can highlight the differences in these two roles. Only one of them might need updating while the other one remains the same. Splitting the two roles and updating only one of them shortens change propagation.

In our example, we observed that the method `move(...)` is used in Drawlets in two different ways: to move the figure as requested by the user or as part of the algorithm for creating new figures. The method's dual role partially causes the extended change propagation.

It's obvious that a user-initiated move must check the user identity, while the creation of new figures need not. By splitting these two roles into two functions—a new method `secureMove(...)` and the old method `move(...)`—we substantially shortened the change propagation. The new method `secureMove(...)` first checks the user's identity and then invokes the old method `move(...)`.

Observations

In the example, we observed that our IC required large change propagation. The data in Table 1 show the change's extent and the impact of the techniques that limit change propagation. Of the three techniques used, splitting led to changes in the smallest number of classes, while refactoring improved program architecture and, thus, future maintainability. Simple propagation was the easiest technique to employ.

In our example, we chose to work on the granularity of classes; the typical step was a visit to a class. We could have presented IC on other granularities as well, such as the granularity of class members (methods and variables) or even the finer granularity of statements. The granularity of classes is the granularity at which the program concepts are most clearly expressed; many of them are implemented as classes. Classes are also natural units for code editors and software tools. Compared to that, smaller granularities provide information that might be more accurate but can also be too detailed and voluminous. As such, it might be appropriate only for small changes. In our example, the granularity of classes worked well and was a natural granularity for IC activities.

In this article, we didn't explicitly discuss testing, updates of manuals and other documents, or similar IC activities, but we believe that the activities presented here are a compatible subset. You can apply change propagation to other documents, and the change propagates through the dependencies among the documents.

The Drawlets example illustrated a large and delocalized change. In changes that are either small or localized, some activities might become trivial or unnecessary. For example, in a small or familiar program, concept location is immediate and doesn't require any explicit activity.

Localized changes are contained within a single class and avoid change propagation. For small changes it might be reasonable to merge actualization and incorporation and change the old code directly without separating these two activities. Changes that retract functionality might still require location and propagation, but deleting obsolete code replaces actualization and incorporation.

To address the universality of the activities presented here, we plan to study a larger set of ICs in future case

studies. We also plan to address how the activities and techniques described in this article impact other software artifacts such as tests and documentation.

We believe that it would be beneficial to extend the work presented here to other software technologies, including databases, networks, and the like, and to other technologies in general.

We've observed that concept location, change impact analysis, and change propagation are overhead in the IC, while actualization and incorporation constitute IC's core. Small changes lead to multiple repetitions of the overhead, and we speculate that the result is decreased programming efficiency and an increased likelihood of errors. However, changes that are too big and deal with too many issues at once could overload the programmers' cognitive capabilities. Hence, we speculate that there's an optimal IC size, which we'll try to find. Although programmers receive change requests from the user community and don't have control over their size, they can often subdivide or aggregate change requests into blocks of the optimal size.

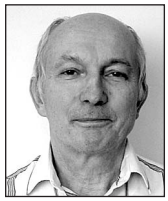
The activities of concept location, change impact analysis, and change propagation employ program dependencies that program analysis discovers. We plan to implement a tool that would extract dependencies from the code and track the progress of the IC activities.

We expect that research on IC will improve the current practice and will contribute to the accumulation of knowledge in this important field.

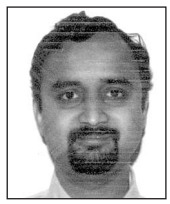
References

1. T.J. Biggerstaff, B.G. Mitbander, and D.E. Webster, "Program Understanding and the Concept Assignment Problem," *Comm. ACM*, May 1994, pp. 72–78.
2. N. Wilde and M.C. Scully, "Software Reconnaissance: Mapping Features to Code," *J. Software Maintenance*, 1995, vol. 7, pp. 49–62.
3. K. Chen and V. Rajlich, "RIPPLES: Tool for Change in Legacy Software," *Proc. IEEE Int'l Conf. Software Maintenance*, IEEE CS Press, 2001, pp. 230–239.
4. N. Wilde et al., "A Comparison of Methods for Locating Features in Legacy Software," *J. Systems and Software*, vol. 65, no. 2, 2003, pp. 105–114.
5. G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, 1998.
6. Z. Yu and V. Rajlich, "Hidden Dependencies in Program Comprehension and Change Propagation," *Proc. 9th Int'l Workshop Program Comprehension*, IEEE CS Press, 2001, pp. 293–299.
7. M. Lindvall and K. Sandahl, "How Well do Experienced Software Developers Predict Software Change?", *J. Systems and Software*, vol. 43, no. 1, 1998, pp. 19–27.
8. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

About the Authors



Vaclav T. Rajlich is a full professor and former chair in the Department of Computer Science at Wayne State University. His research interests include the areas of software evolution and comprehension. He received a PhD in mathematics from Case Western Reserve University. He is a member of the IEEE Computer Society and ACM. Contact him at vtr@cs.wayne.edu.



Prashant Gosavi is a software developer and product manager with Dimensional Control Systems. His research interest is software evolution. He received an MS in computer science from Wayne State University. Contact him at prashant@3dcs.com.