Features and Object Capabilities

Reconciling Two Visions of Modularity

Salman Saghafi
WPI
salmans@cs.wpi.edu

Kathi Fisler
WPI
kfisler@cs.wpi.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Abstract

The prevelance of threats and attacks in modern systems demands programming techniques that help developers maintain security and privacy. In particular, frameworks for composing components written by multiple parties must enable the authors of each component to erect safeguards against intrusion by other components. Object-capability systems have been particularly prominent for enabling encapsulation in such contexts.

We describe the program structures dictated by object capabilities and compare these against those that ensue from feature-oriented programming. We argue that the scalability offered by the latter appears to clash with the precision of authority designation demanded by the former. In addition to presenting this position from first principles, we illustrate it with a case study. We then offer a vision of how this conflict might be reconciled, and discuss some of the issues that need to be considered in bridging this mismatch. Our findings suggest a significant avenue for research at the intersection of software engineering and security.

Categories and Subject Descriptors D.2.11 [*Software Architectures*]: Modules

General Terms Design, Languages, Security

Keywords Feature-oriented programming, object capabilities, modularity

1. Introduction

There are many competing visions of what constitutes the content of a module: phrases like "separation of concerns" still permit a wide variety of interpretations, as the past several decades of research demonstrates.

AOSD MV '12 March 2012, Potsdam, Germany

Given the increasing importance of security, privacy, and related properties, one useful criterion for decomposing systems into modules is to consider what impact the decomposition would have on enabling reasoning about these properties. In particular, when developers compose code written by multiple (perhaps mutually-untrusting) parties, there is a danger that code from one module might inadvertently or maliciously damage the behavior of another. Third-party composition takes place in many traditional component-based systems as well as newer platforms such as extensible Web browsers ("extensions"), client-side Web applications ("mashups"), and mobile phone operating systems ("apps").

Due to the dangers of unfettered code combination, platforms are moving towards circumscribing the set of system resources to which a component is given access, rather than letting the component get the full power of the user running it. This makes it possible to bound the damage that hostile or faulty modules can inflict. The mechanisms found in many of these systems either are, or resemble, *capabilities* [**?**]: e.g., when your mobile phone operating system lists a collection of system resources (location settings, local storage, etc.) that an application demands, your granting these privileges indicates you find this set of demands reasonable, and furthermore informs the operating system to not allow the application access to any other resources.

Capabilities have been a sideline in operating systems research for many years. Over time, they begat a particular form of programming mechanism called the *object capability* [**?**] (OCap), which uses programming language objects to represent capabilities. In an OCap framework, there are no global privileged resources—whether systems resources such as disks and networks, or program-defined ones such as confidential data structures. Instead, all access is provided through explicit granting of authority in the form of a capability object, whose methods represent operations on the privileged resource. This property is called the *absence of ambient authority* [3]; most programming languages violate it by providing static and global variables, through which they expose traditional systems resources (though in some languages, auxiliary mechanisms such as class-loaders can be used to obtain some degree of confinement). Run-time systems can also expose resources through the manufacture

of object references; if this is disabled, the only way for one object to become aware of another is for the former to be given a reference to the latter. Objects can become aware of one another through a handful of mechanisms such as parameter-passing, lexical closure, inheritance, and so on. We abstract over these different mechanisms and (with slight abuse of standard OCaps language) refer to all these processes collectively as *introduction*. An object's net authority is thus the transitive closure of the (methods of the) capabilities to which it has been introduced.

Building on this linguistic foundation, OCap programmers expect modules to obey the *principle of least authority* (POLA) [**?**]. That is, a module must expect no more authority than it requires to accomplish its task. Because a module cannot obtain capabilities through other means (per the OCap assumptions), the user of the module can confidently bound what privileges it may use (and hence, potentially, abuse). OCaps are furthermore effective at thwarting *confused deputy* [**?**] attacks, because the capability is both a designator of an object and the authority to use it; attacks occur when these tasks are separated.

Curiously, though OCaps derive from a principle (POLA) usually associated with security, and their use addresses certain security problems, the reasoning that leads to OCaps is independent of security per se. Instead, POLA can be viewed as analogous to Parnas's principle of information hiding [**?**]: just as information hiding can be regarded as "need to know", Crockford labels POLA as handing out authority only on a "need to do" basis [**?** , page 72]. A module that demands more capabilities than its user expects is either being sloppy or mis-representing its purpose; in either case, a user should proceed with caution or reject the module.

In this paper, we examine how contemporary modularity methods interact with capability-based design, as embodied by POLA. Inherent to the notion of capability-style security (and indeed to most security mechanisms) is the ability to "draw boxes", and then argue how the content of the box cannot harm, or be harmed by, what lies outside it. Modern "boxes" include modularity methods such as aspect-oriented programming, feature-oriented programming, and other responses to weaknesses of traditional object-oriented programming. It is therefore instructive to consider how they fare in a POLA light.

From a POLA perspective, is easy to dismiss of many typical aspect-oriented programming mechanisms [**?**]. The point of weaving [**?**] is to run a program fragment in a context other than that in which it was defined. The woven program thus inherits the power of the location where it was injected. This directly violates the abstraction boundaries of the point of injection, and lets the aspect access capabilities that arguably were not granted to it explicitly (indeed, this is virtually the definition of a confused deputy attack!). Interfaces for aspect-orientation might help in this regard [**? ? ?**].

	`if`	`printf`	`open-tcp`	`···`
Interpreter		screen	network	
Type checker				
Pretty printer	screen	screen	screen	
...				

Figure 1. Features and capabilities in a suite of programming language tools

Feature-oriented programming [**?**] (FOP), in contrast, explicitly defines and composes "boxes". A feature is commonly regarded as a piece of system functionality that a user can identify. In FOP, the unit of modularity is the *feature*; each module implements, roughly, a feature, and the collection of modules corresponds to the features expected by the system's requirements. This alignment of modules with requirements makes it possible to easily customize an application by composing just those features that a particular user desires. Each composition then results in a different system, so FOP naturally leads to product-lines of systems. This connection to product lines underlies FOPs claims to support scalable software development [**?**].

2. An Inherent Conflict?

To concretize features and capabilities, consider a suite of tools that implement a programming language. Suppose each tool is a feature. Each tool must handle (as appropriate) each construct that exists in the language. Figure 1 shows the capabilities that would be needed across common language constructs and tools. For purposes of this paper, the key observations from this table are

- *Different features require different capabilities.* The pretty-printer only requires access to the screen; the interpreter needs both the screen and the network; the typechecker doesn't need any external resource.
- *Different tools require different capabilities across constructs*. The interpreter only needs write access to the screen to implement `printf`; that capability is irrelevant for the other constructs shown.

This table captures the essence of the different views of modularity between features and capabilities. FOP roughly views each row of this table as a module. As the cells within each row may require different capabilities, however, OCaps could demand finer-grained modules, perhaps as small-grained as individual cells. That FOP modularizes around rows as opposed to columns of the table is not relevant here (since different cells in the same column also have different capabilities). The point is that OCaps seeks a modular structure with strong access-control guarantees, FOP seeks a modular structure that enables large-scale, plug-and-play system construction, and these two would frame modules somewhat differently in this example.

Different Perspectives on Modularity Features and OCaps modularize systems differently because they view modularity as solving different problems. Features view modules as a way to enable flexible and scalable construction of families of products. OCaps view modules as a way to ensure that a composed program has no violations of least privilege. A key goal in FOP is to allow third parties to add new features to existing products without modifying existing code, while OCaps enable code from untrusting providers to compose without either party being able to infiltrate the other beyond the bounds specified in the interface. These goals are not identical, and indeed the FOP view is an expansive one while the OCap view is inherently untrusting.

Incompatibility Are these two styles compatible? Perhaps not: this example already illustrates a tension. To ensure least authority, the individual portions of the interpreter would each consume just the capabilities (including none) they require. However, that requires the client linking the features to deal with a potentially large number of small components. By packaging these up into a single component (namely the interpreter), the client linking the features gains scalability by having fewer and higher-level components to content with. In Batory's phrase [**?**], this packaging improves scalability. In return, however, the client must now grant all the capabilities to the interpreter as a whole and hope that it demultiplexes these properly without granting excess authority to any one fragment. Miller refers to this as the "nested platforms" problem [**?** , §22.1].

In short, just as in social structures, *scalability appears to demand delegation and diffusion of authority*. That is, while having lots of little components makes it possible to assign precisely the least authority to each one, this over-burdens the programmer who is responsible for linking them together. By "chunking" the linking, the programmer can deal with significantly fewer, higher-level pieces (the features); but these now require the aggregate of authority required by the components contained inside them. The programmer has to make an unsavory choice between an excessive number of modules with reasonable authority each or a reasonable number of modules with excessive authority each.

3. OCaps Discipline in a FOP System: A Case Study

Is this incompatbility a problem in practice? To gauge the extent to which FOP developers already follow an OCaps discipline, we studied an exemplary application built using FeatureHouse [1]. FeatureHouse is a cross-language framework for FOP; in FeatureHouse, program artifacts are represented as structured trees, and composition is the act of merging trees.

We hasten to note that the point of this section (and indeed of the whole paper) is not to study FeatureHouse in detail, but rather to ask how well existing programs already meet the POLA principle. If they do well, perhaps the theoretical incompatibility is only that, and not a problem in practice. If they do poorly, however, there remains the open question of whether simple refactoring would address the problem, or whether the topic demands more foundational research.

3.1 Assessing Adherence to OCaps Discipline

We first present a simple classification we have found useful when evaluating FOP software for its adherence to OCaps. We classify each feature/capability pair thus:

- Required Capabilities: A capability c is required for a feature f (denoted Req(f,c)) if f is expected to receive c according to the program requirements.
- Given Capabilities: A capability c is given to feature f (denoted Given(f,c)) in a program if some part of f has capability c within the program's implementation.
- Used Capabilities: A capability c is used in feature f (denoted Used(f,c)) if some part of f actually uses c within the implementation.

Unlike given and used capabilities, which can be computed from the source code, required capabilities are hard to pinpoint: a program may have different implementation options, each of which depends on different capabilities. We choose to consider the maximal plausible requirements: this will make least-privilege violations out of given capabilities that have no plausible use in a feature.

The following table characterizes combinations of the labels relative to the goals of capability systems. We do not include the two combinations in which Used(f,c) is true but Given(f,c) is false because it is not possible for a system to use a capability that is not available to it.

Category	Req(f,c)	Given(f,c)	Used(f,c)
No error	false	false	false
No error	true	true	true
LPV	false	true	true/false
Diff Impl	true	true/false	false

Cases in which all labels have the same value do not flag any errors relative to capabilities. Least-privilege violations, denoted LPV in the table, arise when a feature has been given a capability that it does not plausibly need. Whether a feature uses an unrequired capability is not relevant: just because one implementation of a feature does not exploit an unnecessary capability, another implementation (perhaps by a third party) might. Cases in which a feature does not use a plausible capability are not necessarily errors, but indicate that a programmer may have found a different way to implement the system than envisioned in the requirements. In particular, a programmer might choose to virtualize the implementation (e.g., consuming a capability for the file system but actually using an in-memory store instead of the persistent store).

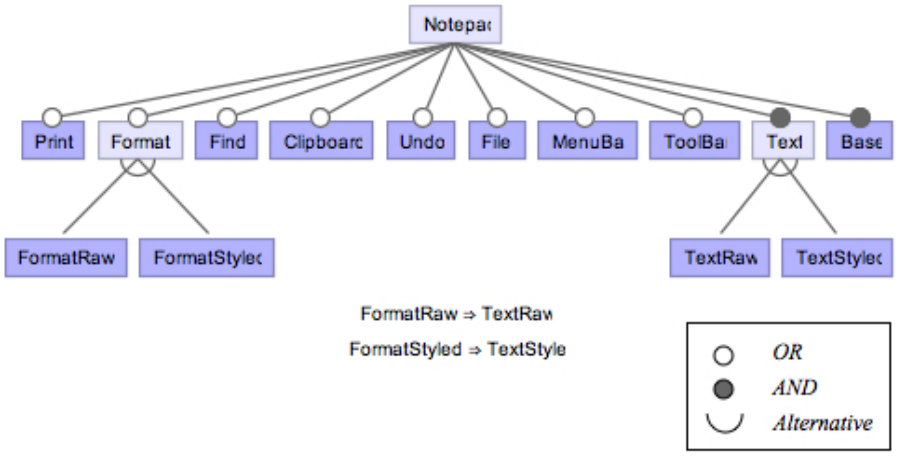

Figure 2. Feature model for the Notepad application. The propositional expressions in the picture shows that FormatRaw and FormatStyled depend on TextRaw and TextStyled respectively. Other dependencies between modules have been suppressed to keep the example small.

3.2 The Case Study

FeatureHouse presents 22 sample projects that have been written in a feature-oriented style in Java (it has additional projects written in other languages, such as Alloy, C, C#, Haskell, JavaCC and UML). We chose a Notepad application, which is both simple enough to immediately understand and rich enough to have interesting resources to protect. Though we studied the entire application, here we focus on a few representative features that illustrate our point.

Notepad implements a canonical note-taking application. Figure 2 shows a feature model of the entire Notepad product family. The primitive application offers a main window as a container for other components of the application. The application requires a Text feature, which provides a text area for editing (but not common file-menu operations). There are also numerous optional features. For this example, we will use features for a Clipboard (cut, copy, and paste), Find (search for text), and Undo. Features that we will not consider include text formatting, menubars, and toolbars.

Our goal is to identify whether Notepad has any least-privilege violations that trace back to its feature-oriented architecture. Recall the term *introduction* (Section 1) for the process by which one object comes to have a reference to another object. As Section 3.1 shows, least-privilege violations in a program arise from introductions that grant more access than necessary. Identifying the source and necessity of the introductions in a program is the first step to checking whether it maintains capability discipline.

For our sample features, we would expect that Find needs to read the contents of the text area and to highlight found terms. Clipboard needs to read, write, and delete from the text area. Undo will read and write to the text area. None of these features need access to each others' data structures, nor does the text area have any need to know about these features.

At the implementation level, the Text feature adds a text area to the notepad, implemented as an instance of Java's `JTextPane` (a standard library class). The feature creates the area, adds it to the main application window, and endows it with a getter. The Text feature also creates an object that holds the callbacks associated with actions a user can take through the Notepad. The Find and Clipboard features add actions, but do not add to the notepad itself. The Undo feature adds an `UndoManager` (a standard Java library object) to the Notepad class rather than use the existing infrastructure for registering actions. For reference, Appendix A shows the portion of the source code for these features that is relevant for this paper (though a reader should be able to follow our arguments without reading the code).

Within this implementation, we find several least-privilege violations. Representative examples of these include:

- *Find has write access to the text area*. Find gets a reference to the text area so it can search the text, but is actually given all of the text area's methods. Under OCaps discipline, Find should have been given a reference to an object that provides only the search method that Find needs.
- *Undo has access to the Actions object*, even though it does not use it. This leakage arises from FeatureHouse's composition technique. Given an ordered list of features

  FeatureHouse Base Text Undo Find Clipboard

  FeatureHouse creates a product by concatenating the contents of identically-named classes across the features.

This approach makes all objects defined in one feature available to all other features in the same product. Since there are no class boundaries between features, even variables marked private in one feature are available to the others.

- *UndoAction has access to the entire Notepad*, even though it only needs access to the Notepad's `UndoManager` and one other specific field that the Undo feature itself added to the Notepad. This leakage arises from a poor architectural decision with the Notepad class. It could easily have been avoided by passing the two required fields rather than the entire Notepad object as arguments.

The first two problems lead to pervasive violation of least privilege. We manually analyzed the entire Notepad feature suite from Figure 2. To perform this analysis, we had to define the required capabilities, which we did by adhering closely to the principle of least privilege. Our analysis found 200 least-privilege violations that trace to the first issue and 373 that trace to the second. The third problem does not show up in required capabilities because the unnecessary object is of a class that is limited to the implementation and does not manifest in the requirements (which perhaps makes its availability even more insidious). Note that every violation is an opportunity for a feature implemented by a potentially untrusted third-party to obtain powers it should not have had.

We find it interesting that the Notepad features introduce relatively few methods of their own. Rather, they rely heavily on methods that are already part of existing Java classes (such as `JTextPane` and `UndoManager`). This is relevant to our discussion because it affects how we should think about required capabilities. We stated that Clipboard needs read and write access to the *text area*. In this implementation, however, Clipboard only uses existing read and write *methods* on the text area—it did not use the text area itself (other than as a route to these methods). Ideally, Notepad should introduce only these methods (or functions)—rather than the entire text area—to the Clipboard.

Wrapper objects are an effective way to limit visibility of existing methods. However, passing entire objects is easy; implementing wrappers is painstaking in Java because of the limitations of the nominal type system. As a result, programmers will be tempted to pass entire objects even if this approach leaks capabilities. The Notepad authors simply followed standard Java practices; unfortunately, those are not always consistent with OCaps discipline (and arguably, by extension, good modularity).

3.3 Feature-Aware Access Modifiers

Acknowledging that FOP may need to limit access to variables across features, some of the researchers behind FeatureHouse have proposed a set of access-modifiers unique to features [2]. In particular, they proposed the following modifiers (usable in the same places as standard Java public/private/etc modifiers):

- `feature`: limits access to the feature in which the variable is defined.
- `subsequent`: limits access to the feature containing the variable and all features composed subsequently (on the command line).
- `program`: the variable is available globally, as with the standard `public` modifier.

These modifiers would have limited value in limiting least privilege. Sometimes, one feature builds on another; these cases would require the `subsequent` modifier, which would leak data to any other features that happened to be included later in a composition. This approach also does little to address problems due to failure to wrap objects in more restricted interfaces.

3.4 Other Approaches to FOP

There are, of course, numerous other approaches to FOP, which are worth studying. We simply note that FOP as represented by tools like FeatureHouse appear to be somewhere in the middle of a spectrum in their ability to incorporate OCaps. Using annotative approaches [**?**] to FOP seems even less likely to be fruitful because of their invasiveness, which would enable feature code to obtain access to ambient capabilities just as with aspects.

While FeatureHouse separates features in its front-end language, its composition technique of syntactically merging code across feature boundaries is a distinct weakness with respect to preserving modular separation. As a result, even with well-designed interfaces, the resulting code still suffers from leakage of authority. However, the critique in our case study does not depend on this property (which is obvious, and might anyway be mitigated by using a different language such as Joe-E [**?**], a capability-safe version of Java).

Unlike FeatureHouse, AHEAD [**?**] (in "mixin", as opposed to "jampack" mode) does erect boundaries between features using subclassing. This in turn limits the scope of private variables to the feature in which they were defined. This obedience to source modularity, which is also found in other systems such as those of Prehofer [**?**] and Findler and Flatt [**?**], is an important step towards preventing unwanted comingling of code and hence the leakage of authority.

4. A Route Forward: Verified Demultiplexing

We repeat the main insight from earlier: scalability appears to demand delegation and diffusion of authority. To use FOP, a programmer is effectively forced to hand over a large set of capabilities to some features, expecting them to be restributed internally using POLA.

This expectation should be captured by an appropriate interface specification that makes clear how the provided capabilities should be distributed internally, and furthermore

(presumably) that the feature code in the interstices is itself not supposed to use any of them. The interface would take the form of access-control rules that dicate how sub-components receive capability objects, but might also include some integrity and information flow rules that dictate flows that are prohibited to avoid attacks through colluding sub-components. Writing such specifications somewhat diminishes the scalability benefits of features, but this appears an unavoidable trade-off, and hopefully these specifications do not need to be written very often.

Given such specifications, a good deal of research in language-based computer security deals with how one can actually enforce these expectations on bodies of code. Enriching a type system with this verification power, for instance, makes it easier to integrate such checking into the development cycle. By using languages or programming systems endowed with such verifiers, the composer of features can thus be confident that the demultiplexing of capabilities is happening in a trustworthy fashion, and that the use of features is not inhibiting the provision of security.

5. Perspective

This paper asks whether increments designed around user-identified functionality, which covers FOP and also aspects, are compatible with capability discipline. Capabilities enforce least privilege. Identifying idioms for programming simultaneously with feature-like constructs and security concerns such as least privilege is an important area for future research. In particular, our work to date raises the following observations and research questions:

- Formal modules, with checked and enforced interfaces, are critical to preventing least-privilege violations. The FOP community is actively debating whether features should be captured in formal modules [**?**]. Our observations demand that those who argue against formal interfaces explain how they can provide security guarantees without them, or why security guarantees do not apply in their context.
- One argument against features as modules is that features sometimes add very small amounts of code (a couple of lines). Formal interfaces on these "micromodules" can seem like overkill. Proponents of micromodules need to weigh the arguments in favor of these fragments against the corresponding leakage of authority.
- The alternative to preserving modularity in the composed system is to obtain its benefits through alternate means. For instance, it may be possible to devise type systems, static analyses, and so on that bless individual feature modules in such a way that even textual composition of such modules will not result in privilege leakage. It is worth noting that such textual analyses will effectively be implementing a modularity mechanism such as language-based sandboxing [**?**].
- We have argued that OCap discipline may be too rigid for features. If we relax that discipline, what other principles can provide design guidelines? One possibility is to incorporate threat modeling and align modules with trust boundaries. The correlation between feature boundaries and trust boundaries has not been studied in the literature, and appears to be a promising direction in the study of modularity.

It is also worth remembering that *least* privilege is itself a relative notion. It must be defined relative to purpose. For instance, an interface might require or provide more than the module's pure functionality seems to demand. These more expansive interfaces might support debugging, performance-tuning, future-proofing, etc. Are these violation of POLA? Not necessarily: it may just mean that the least privilege of the component amounts to more than just its current functionality. In other words, POLA is still a guideline open to multiple interpretations, and these interpretations will depend on the goals of various components.

In conclusion, with features and capabilities each targeting a real, pressing software concern, we cannot ignore the tension between them. We look forward to ongoing discussions about the tradeoffs between these approaches and techniques for making them interoperate.

Acknowledgments

This work is partially supported by the US National Science Foundation. We thank Mark S. Miller for many valuable conversations over the years. We are especially grateful to Kevin Sullivan for his shepherding of this paper. His detailed and thoughtful questions and comments greatly helped us clarify our presentation.

References

[1] Sven Apel, Christian Kastner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 221–231, Washington, DC, USA, 2009. IEEE Computer Society.

[2] Sven Apel, Sergiy Kolesnikov, Jörg Liebig, Christian Kästner, Martin Kuhlemann, and Thomas Leich. Access control in feature-oriented programming. *Science of Computer Programming*, 2010.

[3] Mark Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. Available online at `ttp://srl.cs.ju.edu/pubs/SRL2003-02.pdf`, 2003. Last accessed Sept 23, 2011.

[4] Christian Prehofer. Feature-oriented programming: A fresh look at objects. pages 419–443. Springer, 1997.

Text/Base

```
1  class Notepad {
2    public Actions actions = new Actions(this);
3    private JTextPane textPane;
4
5    public Notepad () {
6      textPane = new JTextPane();
7      getContentPane().add(textPane);
8    }
9
10   public JTextComponent getTextComponent() {
11     return textPane;
12   }
13 }
14
15 class Actions {
16   Notepad n;
17
18   public Actions(Notepad n) {
19     this.n = n;
20   }
21 }
```

Figure 3. Essence of the Text feature in FeatureHouse

Clipboard

```
1  class Actions {
2    public void cut(){
3      n.getTextComponent().cut();
4    }
5
6    public void copy(){
7      n.getTextComponent().copy();
8    }
9
10   public void paste(){
11     n.getTextComponent().paste();
12   }
13 }
```

Figure 4. Essence of the Clipboard feature in FeatureHouse

A. Notepad Feature Code

This appendix presents fragments of the Notepad features that are relevant to our discussion of capabilities along with an example of how they appear in composition. Each feature consists of up to two class definitions. The `Notepad` class contains the structural elements of the notepad application (text areas, menu bars, etc); the `Actions` class contains methods for the callbacks that are executed when the user selects from menus, toolbars, or other GUI elements. Some features, such as Clipboard and Find, add new actions but no new structural elements.

The Text feature (Figure 3) creates a text area (line 6) and adds it to the main application window (line 7). It also adds a getter method for the private `textPane` variable (lines 10-12). This feature does not itself add actions, but merely connects a Notepad instance to an Actions class (lines 17-18).

Clipboard (Figure 4) introduces three actions—cut, copy and paste—that interact with the operating system's clipboard. Each action simply calls the relevant method of the text area object (lines 3, 7 and 11). The Find feature (Figure 5) adds two action methods (`find` and `findnext`) which execute when the user clicks on buttons that Find adds to the toolbar and menubar (these additions are not shown). The `find` method gets the search string from the user through a dialog box (line 3), then passes the search string to the text area's search methods (line 4). Find's actions use the `selectFound` method to highlight the strings found in the text area using the text area's methods (line 14).

Unlike the other features, Undo (Figure 5) defines its own action classes (UndoAction and RedoAction) rather than extend the existing Action class. It also adds an instance of UndoManager, a standard Java class that manages undo/redo operations (line 2). When the user chooses to perform an undo/redo operation via the toolbar or the menubar, the undo/redo actions call the relevant methods of the undo manager instance to handle the operation (line 17).

Figure 6 shows the Notepad class that results from the composition of the Text and Undo features. The comments demark which portions of the class came from each of the features. The rest of the code has been omitted, as it does not add relevant detail to the example.

Find

```
class Actions {
  public void find() {
     findWord = JOptionPane.showInputDialog("Type the word to find");
     findIndex = n.getTextComponent().getText().indexOf(findWord);
     if (findIndex == -1) {
       JOptionPane.showMessageDialog(null,"Word not found",...);
     } else { selectFound(); }
     ...
  }

  public void findNext() { ... }

  private void selectFound() {
    n.getTextComponent().select(findIndex, findIndex + findWord.length());
  }
}
```

Undo

```
class Notepad {
  UndoManager undo = new UndoManager();
  UndoAction undoAction = new UndoAction(this);

  public Notepad() {
    getTextComponent().getDocument().addUndoableEditListener(... undo.addEdit() ...);
  }
}

class UndoAction extends AbstractAction {
  Notepad notepad;

  public UndoAction(Notepad notepad){
    this.notepad = notepad;
  }

  public void actionPerformed(ActionEvent e) {
    notepad.undo.undo();
  }
}
```

Figure 5. Essence of the Find and Undo features in FeatureHouse

```
class Notepad
  //FROM TEXT-BASE
  public Actions actions = new Actions(this);
  private JTextPane textPane;
  //TEXT

  //FROM UNDO
  UndoManager undo = new UndoManager();
  UndoAction undoAction = new UndoAction(this);
  RedoAction redoAction = new RedoAction(this);
  //UNDO

  public Notepad  () {
    //FROM TEXT
    textPane = new JTextPane();
    getContentPane().add(textPane);
    //TEXT

    //FROM UNDO
    getTextComponent().getDocument().addUndoableEditListener(...);
    //UNDO
   ...
   }
  ...
}
```

Figure 6. The Notepad class composed from the Text and Undo features