Diagnosing Java code: Designing extensible applications, Part 3

Examine when, where, and how black box extensibility works best

Level: Introductory

Eric Allen (eallen@cs.rice.edu), Ph.D. candidate, Java programming languages team, Rice University

19 Nov 2001

In contrast to glass box extensibility, which we discussed in the last *Diagnosing Java Code,* black box extensibility refers to the ways in which a software system may be extended when the source code is available neither for viewing nor modifying. Such extensions are made generally through system configuration or the use of an application-specific scripting language. In this installment, Eric Allen discusses when it makes sense to design a system for black box extensibility and offers some ideas for how to do so effectively. After reading this article, you'll know when to use the black box and will have some tips on how to implement it. Share your thoughts on this article with the author and other readers in the discussion forum by clicking **Discuss** at the top or bottom of the article.

I've spoken in earlier articles of the importance of a code-reuse design strategy (due primarily to the growing diversity of information-processing tasks and the associated costs), so if you've decided that system extensibility is your goal, ask yourself, "How extensible should the system be and how extensible can I make it?" Then consider the following:

- The trade-offs for adding extensibility may be decreased performance or testing ability.
- The most testable systems are usually the simplest ones; adding extensibility often adds complexity.
- A key bit of knowledge to planning a successful extensible design is to know how you plan to extend the system at a later date.

In the first article in this series, I outlined the various forms of extensibility that a system may exhibit -- the *black box* design and two *white box* designs, the *glass box* and the *open box*. In the second article, I detailed the uses for and implementation of glass box extensibility, the happy medium between black box and open box design.

This month, I'd like to continue our tour by expanding on black box extensibility.

Feeling your way around the black box

Black box design is the kind of extensibility that involves user configuration to customize the application to perform in ways most useful to a particular context. When extending an application this way, it is not necessary to view the original source code.

We've all worked with applications that provide this kind of extensibility. Two such examples include:

- Netscape's plug-in facility
  Emacs with its endless capacity for configurability with

Don't miss the rest of the "Designing extensible applications" series
Part 1: "Black box, open box, or glass box: which is right and when?" (September 2001)

Part 2: "Examine when, where, and how glass box extensibility works best" (October 2001)

Emacs Lisp

These days, pretty much each new application provides black box extensibility to some degree.

Part 4: "Examine how S-expressions provide a lightweight form of black box extensibility" (December 2001)

How to spot a configuration script

Before continuing, let me try to distinguish configuration scripts from other inputs to a program. Unfortunately, there really is no sharp distinction here. The set of inputs accepted by a program (and the way the program interprets these inputs) can and should be viewed as a language. But configuration scripts have several salient features that you should understand.

Unlike other program inputs, which can change with each use of the application, configuration scripts tend to be more stable. The scripts often have default values that will be set initially by the user and then not reset for a long period. In fact, such scripts are often set when the program is initially installed. Changing the default values of the scripts will tend to significantly affect the behavior of the program on the other inputs. And the values of these inputs are stored permanently, so they can be retrieved on subsequent invocations of the program.

Choose your configurations wisely

Now, the next question you should ask is, "When does it make sense to add this kind of extensibility to an application?"

Adding as much extensibility as possible is not the answer. After all, the end result of such an approach would be a single application that performed every task a user required, given the appropriate script -- sort of an *Überapplikation*.

Arguably, development environments fall into this category, but the "scripts" required by the user/developer can be very long and involved. This extreme example illustrates a fundamental trade-off of black box extensibility -- the more black box extensibility an application provides, the more work a user will have to do to configure it for a particular context.

It is often better to find a narrower, but still popular, demand for an application, and then target this more specific context. As many Extreme Programming teams have demonstrated, narrowing the scope of a project also makes it much more likely that you'll actually succeed in completing the project (and on time, to boot!).

Certain contexts demand black box

Nevertheless, there are contexts in which you can be absolutely certain that there will be a demand for a certain form of black box extensibility, and you will be able to provide this extensibility without overburdening the user. In many cases, it will still pay to implement a non-extensible version of the application with narrower scope and then add the extensibility in another version.

This may strike you as paradoxical. After all, the whole point of adding extensibility is to reduce the cost of implementing new functionality in a system. If you know you'll be extending an application, why not build in the extensibility to begin with? Because it's often much easier and quicker to build the application without the extensibility.

By first providing the simplest version of the application you can, the sooner your customers will be able to use it for at least some of their tasks. And you'll then have revenue from sales of *version 1.0* to support the development of the more extensible version.

Determining complexity

The key point to consider here is whether you expect the extensible version of your application to be significantly more complex. Sometimes, the simplest and most natural way to design an application is to incorporate extensibility.

One of the best examples is a tax-calculating application. Such an application works over sets of tax forms. The particular set of forms used depends on the particular user; therefore, it is quite natural to use a special-purpose configuration language to describe the various forms. It would save no complexity if instead, you hardwired these tax forms into the program (via, say, a class hierarchy of forms with a separate class for each form).

But once you've designed the application in this way, you can easily add new forms (as you'll certainly have to do come next fiscal year) without modifying or even viewing the source code.

There are other examples of applications for which a black box extensible design is simplest. Those that come to mind are Web browsers, television schedule viewers (like those used by digital television services), and automated theorem provers, which rely on a database of axioms.

Language you can be proud of

Now, suppose you've decided to provide a configuration language for some aspect of your application. The most important issue to acknowledge when you decide to do this is to know that you really are designing a language. You must be concerned with the same sorts of issues that concern programming-language designers. For example:

- Your language should have a formal and well-defined syntax and semantics.
- Your interpreter for this language should include a parser that rejects all but syntactically valid specifications in your language.
- You should allow for some context-sensitive checking, such as type checking, when possible.

Note that this is not an exhaustive list; there are plenty of other considerations.

It's tempting to ignore some of these steps, and, say, not worry so much about accepting only syntactically valid code. And it's not uncommon. The usual result is that the many bugs that undoubtedly crop up in the configuration scripts for these languages are difficult to track. (My article "The Saboteur Data bug pattern" discusses this issue; see Resources.)

Unfortunately, bugs are not the only problems that result from ignoring such steps. Some of the most damaging computer security attacks, including the Code Red virus, have resulted from a failure to properly parse input. Ross Anderson's book *Security Engineering* (see Resources) includes a great discussion of such attacks in the context of operating systems.

Advantages to using XML

Fortunately, designing such languages well is not as hard as it used to be. XML is a great tool for this task because there are many off-the-shelf XML parsers and development tools.

Not only can these third-party components make your job easier, but they also make it easier for developers to reuse your configuration scripts for other applications. For example, XML descriptions of molecules could be written for a drug encyclopedia application and then reused for a drug-design development tool (by another party who has no access to the source code of the original application).

By the way, reusing code in this manner is yet another reason to define the semantics of your configuration language well. If the meaning of your programs can be determined only by digging into the details of your interpreter, the cost of reusing the scripts will be much higher. See Resources for links to several good XML tools.

Disadvantages to using XML

There are disadvantages to XML. Scripts written in XML can be extremely verbose, and encoding certain forms of information (such as, say, executable code) can be awkward. Also, the use of XML tools requires some overhead and investment of resources, that, in some cases, may be overkill. In cases such as these, I prefer to use S-expressions as a meta-level syntax for the language.

S-expressions: An alternative to XML

S-expressions are fully parenthesized expressions with only one form of parenthesis. All Scheme programs consist of S-expressions. Scheme and other Lisp-like languages were originally designed by the AI community to facilitate reflective processing of the language by programs in the language itself. But many of the advantages of using S-expressions do not rely on the processing language -- these expressions are also easily processed using any language, including the Java language.

To be continued

Now, availability of the source code shouldn't be too much of an obstacle in crafting extensibility if you remember:

- How to spot configuration scripts
- How to choose the right configurations
- How to recognize which contexts call for the black box
- How to determine the complexity of your extensible version
- When you provide a configuration language, you're actually building a language

Next time, I will demonstrate a simple example of a configuration language consisting of S-expressions, along with a Java interpreter for this language.

Resources

- Participate in the discussion forum.

- *Security Engineering: A Guide to Building Dependable Distributed Systems* (John Wiley & Sons, 2001), by

Ross J Anderson, presents a comprehensive design tutorial that illustrates basic concepts through real-world system design successes and failures.

- Check out the CERT Web site for information about various Internet security attacks and the mechanisms they use.

- The World Wide Web Consortium (W3C) offers an exhaustive source of information on XML.

- Ronald Rivest, professor of electrical engineering and computer science at the Massachusetts Institute of Technology (MIT), discusses S-expressions and their use in various security applications, particularly in use in the SDSI (Simple Distributed Security Infrastructure), a new design for a public-key infrastructure.

- Standard Visitors are discussed in the original Design Patterns book *Design Patterns: Elements of Reusable Object-Oriented Software* , Gamma, et al. (Addison-Wesley 1995). The 1998 update includes a CD with sample code and 23 cut-and-paste patterns.

- Clemens Szyperski offers *Component Software: Beyond Object-Oriented Programming, Second Edition,* (Addison-Wesley, 2002), a good hardcopy source for information on designing systems out of reusable components (including studies of component-oriented tools and languages and in-depth discussions of the potential and challenges of component software).

- Christoph Czernohous's two-part series, "Bank on it: Introduction to J/XFS," (*developerWorks*, August - September 2001) introduces and addresses integrating Extensions for Financial Services for the Java platform (J/XFS) into existing systems.

- Read all of Eric's *Diagnosing Java code* articles. In particular, his article "The Saboteur Data bug pattern" (May 2001) addresses the elusive nature of bugs caused by corrupt data.

- Find more Java technology resources on the *developerWorks* Java technology zone.

About the author

Eric Allen has a bachelor's degree in computer science and mathematics from Cornell University and is a Ph.D. candidate in the Java programming languages team at Rice University. Before returning to Rice to finish his degree, Eric was the lead Java software developer at Cycorp, Inc. He has also moderated the Java Beginner discussion forum at JavaWorld. His research concerns the development of semantic models and static analysis tools for the Java language, both at the source and bytecode levels. Eric has also helped in the development of Rice's compiler for the NextGen programming language, an extension of the Java language with generic run-time types. Contact Eric at eallen@cs.rice.edu.