JAVA TOOLBOX

By Allen Holub, JavaWorld

SEP 5, 2003 12:00 AM PST

Why getter and setter methods are evil

Make your code more maintainable by avoiding accessors

Next, I think of all OO systems as having a procedural boundary layer. The vast majority of OO programs runs on procedural operating systems and talks to procedural databases. The interfaces to these external procedural subsystems are generic by nature. Java Database Connectivity (JDBC) designers don't have a clue about what you'll do with the database, so the class design must be unfocused and highly flexible. Normally, unnecessary flexibility is bad, but in these boundary APIs, the extra flexibility is unavoidable. These boundary-layer classes are loaded with accessor methods simply because the designers have no choice.

In fact, this not-knowing-how-it-will-be-used problem infuses all Java packages. It's difficult to eliminate all the accessors if you can't predict how you will use the class's objects. Given this constraint, Java's designers did a good job hiding as much implementation as they could. This is not to say that the design decisions that went into JDBC and its ilk apply to your code. They don't. We *do* know how we will use the classes, so you don't have to waste time building unnecessary flexibility.

A design strategy

So how do you design without getters and setters?

The OO design process centers on use cases: a user performs standalone tasks that have some useful outcome. (Logging on is not a use case because it lacks a useful outcome in the problem domain. Drawing a paycheck is a use case.) An OO system, then, implements the activities needed to play out the various scenarios that comprise a use case. The runtime objects that play out the use case do so by sending messages to one another. Not all messages are equal, however. You haven't accomplished much if you've just built a procedural program that uses objects and classes.

In 1989, Kent Beck and Ward Cunningham taught classes on OO design, and they had problems getting people to abandon the get/set mentality. They characterized the problem as follows:

> *The most difficult problem in teaching object-oriented programming is getting the learner to give up the global knowledge of control that is possible with procedural programs, and rely on the local knowledge of objects to accomplish their tasks. Novice designs are littered with regressions to global thinking: gratuitous global variables, unnecessary pointers, and inappropriate reliance on the implementation of other objects.*

Cunningham developed a teaching methodology that nicely demonstrates the design process: the CRC (classes, responsibilities, collaboration) card. The basic idea is to make a set of 4x6 index cards, laid out in three sections:

- **Class:** The name of a class of objects.
- **Responsibilities:** What those objects can do. These responsibilities should focus on a single area of expertise.
- **Collaborators:** Other classes of objects that can talk to the current class of objects. This set should be as small as possible.

The initial pass at the CRC card is just guesswork—things will change.

Beck and Cunningham then picked a use case and made a best guess at determining which objects would be required to act out the use case. They typically started with two objects and added others as the scenario played out. They selected people from the

class to represent those objects and handed them a copy of the associated CRC card. If they needed several objects of a given class, then several people represented those objects.

The class then literally acted out the use case following these rules:

- Perform the activities that comprise the use case by talking to one another.
- You can only talk to your collaborators. If you must talk to someone else, you should talk to a collaborator who can talk to the other person. If that isn't possible, add a collaborator to your CRC card.
- You may not ask for the information you need to do something. Rather, you must ask the collaborator who has the information to do the work. It's okay to pass to that collaborator information he needs to do the work, but keep this interaction to a minimum.
- If something needs to be done and nobody can do it, create a new class (and CRC card) or add a responsibility to an existing class (and CRC card).
- If a CRC card gets too full, you must create another class (CRC card) to handle some of the responsibilities. Complexity is limited by what you can fit on a 4x6 index card.

A recording made of the entire conversation is the program's dynamic model. The finished set of CRC cards is the program's static model. With many fits and starts, you can solve just about any problem this way.

The process I just described *is* the OO design process, albeit simplified for a classroom environment. Some people design real programs this way using CRC cards. More often than not, however, designers develop the dynamic and static models in Unified Modeling Language (UML). The point is that an OO system is a conversation between objects. If you think about it for a moment, get/set methods just don't come up when you have a conversation. By the same token, get/set methods won't appear in your code if you design in this manner before you start coding.

Summing up

Let's pull everything together: You shouldn't use accessor methods (getters and setters) unless absolutely necessary because these methods expose information about how a class is implemented and as a consequence make your code harder to maintain. Sometimes get/set methods are unavoidable, but an experienced OO designer could probably eliminate 99 percent of the accessors currently in your code without much difficulty.

Getter/setter methods often make their way in code because the coder was thinking procedurally. The best way to break out of that procedural mindset is to think in terms of a conversation between objects that have well-defined responsibilities. Cunningham's CRC card approach is a great way to get started.

Parts of this article are adapted from my forthcoming book, tentatively titled Holub on Patterns: Learning Design Patterns by Looking at Code, *to be published by Apress (www.apress.com) this fall.*

Allen Holub has worked in the computer industry since 1979. He currently works as a consultant, helping companies not squander money on software by providing advice to executives, training, and design-and-programming services. He's authored eight books, including Taming Java Threads (Apress, 2000) and Compiler Design in C (Pearson Higher Education, 1990), and teaches regularly for the University of California Berkeley Extension. Find more information on his Website (http://www.holub.com).

Learn more about this topic

- A previous **Java Toolbox** article (July 1999) also discussed the what-is-an-object issue, but in the context of client-side UI design
  http://www.javaworld.com/javaworld/jw-07-1999/jw-07-toolbox.html (https://www.javaworld.com/javaworld/jw-07-1999/jw-07-toolbox.html)
- Kent Beck and Ward Cunningham's original paper on CRC cards
  http://c2.com/doc/oopsla89/paper.html (http://c2.com/doc/oopsla89/paper.html)
- *The CRC Card Book*, David Bellin and Susan Suchman Simone (Addison-Wesley Publishing Co., 1997; ISBN0201895358) describes in depth how you might use CRC cards to develop an OO design
  http://www.amazon.com/exec/obidos/ASIN/0201895358/alleiholuasso (http://www.amazon.com/exec/obidos/ASIN/0201895358/alleiholuasso)
- See all of Allen Holub's **Java Toolbox** columns
  http://www.javaworld.com/columns/jw-toolbox-index.shtml (https://www.javaworld.com/columns/jw-toolbox-index.shtml)

- View David Geary's **Java Design Patterns** columns
  http://www.javaworld.com/columns/jw-java-design-patterns-index.shtml
  (https://www.javaworld.com/columns/jw-java-design-patterns-index.shtml)
- *Design Patterns,* Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Publishing Co., 1995; ISBN0201633612)
  http://www.amazon.com/exec/obidos/ASIN/0201633612/javaworld
  (http://www.amazon.com/exec/obidos/ASIN/0201633612/javaworld)
- Browse the **Design Patterns** section of *JavaWorld*'s Topical Index
  http://www.javaworld.com/channel_content/jw-patterns-index.shtml
  (https://www.javaworld.com/channel_content/jw-patterns-index.shtml)
- Browse the **Object-Oriented Design and Programming** section of *JavaWorld*'s Topical Index
  http://www.javaworld.com/channel_content/jw-oop-index.shtml
  (https://www.javaworld.com/channel_content/jw-oop-index.shtml)
- Visit the **Programming Theory & Practice** discussion
  http://www.javaworld.com/javaforums/postlist.php?Cat=&Board=TheoryPractice
  (https://www.javaworld.com/javaforums/postlist.php?Cat=&Board=TheoryPractice)
- Sign up for *JavaWorld*'s free weekly newsletters
  http://www.javaworld.com/subscribe (https://www.javaworld.com/subscribe)

This story, "Why getter and setter methods are evil" was originally published by JavaWorld.