GetterEradicator

You can tell them by the twitch in the left hand side of the mouth when they see a getter method, there's swift pull on their battleaxe and a satisfied cry as another getter is hewn unmercifully from a class which immediately swoons in an ecstasy of gratefulness at the manly Getter Eradicator's feet.

Alright, maybe my return to English beer is affecting me a bit too much, but Chris's gentle tweak struck a pet peevlet of mine. I've often come across people who tell you to avoid having getting methods on classes, treating such things as a violation of encapsulation, Allen Holub's articleis one of the best known.

The general justification is that getters violate encapsulation. If I've got a bowler class with fields for overs, runs and wickets, then adding getters (getOvers, getRuns, getWickets) is little better than just making the fields public.

There's some sense to this argument, and I certainly suggest that you shouldn't write accessors until you really need them, but it also brings in the danger of missing the point of encapsulation. For me, the point of encapsulation isn't really about hiding the data, but in hiding design decisions, particularly in areas where those decisions may have to change. The internal data representation is one example of this, but not the only one and not always the best one. The protocol used to communicate with an external data store is a good example of encapsulation - one that's more about the messages to that store than it is about any data representation. When you are thinking about encapsulation I think it's better to ask yourself "what piece of variability are you hiding and why" rather than "am I exposing data". (Craig Larman wrote a nice column for me on this.)

Although the defense of encapsulation is the common rallying cry for getter eradicators, I think the real motivation for them is rather more reasonable and pragmatic. There's a hell of a lot of code out there in OO languages that is procedural in design. The OO community may have 'won' in the sense that modern languages are dominated by objects, but they are still yet to win in that OO programming is still not widely used. As a result it's still common to see procedures that pull data out of an object to do something, when that behavior would better fit in the object itself - a violation of the pragmatic programmers principle of "Tell Don't Ask". You can only do this kind of procedural programming if you have getters, so telling people to get rid of getters helps push them to move behavior into the right place.

I have a lot of sympathy with this motivation, but I fear that just telling people to avoid getters is a rather blunt tool. There are too many times when objects do need to collaborate by exchanging data, which leads to genuine needs for getters.

If we're looking for a simple rule of thumb, the one I prefer is one that I first heard from Kent Beck, which is to always beware of cases where some code invokes more than one method on the same object. This occurs with accessors and more reasonable commands. If you ask an object for two bits of data, can you replace this with a single request for the bit of data you're calculating? If you tell an object to do two things, can you replace them with a single command? Of course there are plenty of cases where you can't, but it's always worth asking yourself the question.

Another good warning sign of trouble is the Data Class - a class that has only fields and accessors. That's almost always a sign of trouble because it's devoid of behavior. If you see one of those you should always be suspicious. Look for who uses the data and try to see if some of this behavior can be moved into the object. In these cases it can be useful to ask yourself 'can I get rid of this getter?' Even if you can't, asking the question may lead to some good movements of behavior.

Allocation of behavior between objects is the essence of object-oriented design, so like any design, there isn't a hard and fast rule - rather a judging of trade-offs. Putting the behavior in the same class as the data, what Craig Larman calls "Information Expert", is the first choice to consider. But it isn't the only route. Layering often trumps this, many of the Gang of Four patterns separate data from behavior for particular needs. A good rule of thumb is that things that change together should be together. Data and the behavior that uses it often change together, but often you see other groupings that matter more.