Information Hiding and Encapsulation

Information hiding

Information hiding has gotten a bad rap, especially because it's used by governments to hide corruption, fraud, incompetence, and poor operation. It is used to keep people from knowing the truth —and sometimes to keep dictators in power.

But in software design, *information hiding* has been an important design principle ever since it was first discussed in a paper by David Parnas in 1972. (*On the criteria to be used in decomposing systems into modules*, CACM, Dec. 1972).

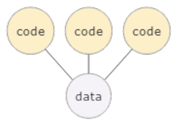
```
97 <= (int) c  &&  (int) c <= 122

'a' <= c  &&  c <= 'z'
```

Hiding implementation details can be used in a very local setting to make a confusing expression understandable. For example, a new, naïve programmer who has learned a tiny bit about Unicode may write the first expression on the right to test whether char variable `c` contains a Latin lowercase letter. In Unicode, `'a'` is represented by 97 and `'z'` by 122; but such "magic numbers" (look that term up in JavaHyperText) should never be used this way. The wise programmer writes the second expression instead, letting Java insert the necessary casts of characters to **int**s.

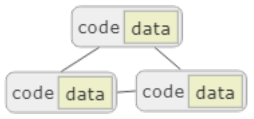
Here's a second example of *information hiding*. A method has two important properties: *what* it does and *how* it does it. The *what* is given in its specification —in Java, usually in a javadoc comment. The *how* is given by the method body. The client, the person who writes calls on the method, looks only the specification. The *how*, the implementation, the method body, is generally unavailable; it is hidden information. Moreover, even if the method body is available, the wise client doesn't look at it.

In fact, the Java API documention is filled with specifications of methods, but the bodies of the methods are not easily available. The clients don't need them.

In his 1972 paper, Parnas was interested in a more global kind of information hiding. He was exploring the problem of designing a program as a collection of *modules* —where a module is just a bunch of stuff that logically seemed to belong together. OOP was unknown at the time. He designed a certain program in two different ways.



The first design was typical at the time, shown to the right, called *procedural programming*. The design focused on the algorithms doing the computation, with the code modules referencing the data in different ways. This kind of design made it awfully difficult to make implementation changes later on. In fact, in Parnas's example problem, a change in the implementation of one data structure required changes in *all* the code modules. That's not good.



Parnas's second design of the program took into account the data and how it could be placed in a module and how the implementation of the data could be hidden from the other modules. He called it *modular programming.* The interaction of each module with others was chosen "to reveal as little as possible about its inner workings". This meant hiding the implementation but providing a way of obtaining and changing data in a way that did not depend on the implementation. Because of this, later, it would be relatively easy to change the inner workings of a module without any affect on the other modules.

It's interesting that Parnas came to this second way of designing programs even though object-oriented programming (OOP) was not known at the time.

Parnas listed the expected benefits of modular programming:

(1) *Managerial*: development time should be shortened because separate groups could work on different modules with little need for communication.

(2) *Product flexibility*: it should be possible to make drastic changes to one module without a need to change others;

(3) *Comprehensibility*: it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood.

OOP makes it easy to implement this principle of information hiding, using the notion of *encapsulation*. Look *encapsulation* up in JavaHyperText.