Del libro:

**Code Complete: A Practical Handbook of Software Construction, Second Edition**

by [Steve McConnell](http://www.cc2e.com/Default.aspx)

Capítulo 6: Working Classes

Contents

6.1 Class Foundations: Abstract Data Types (ADTs)

6.2 Good Class Interfaces

6.3 Design and Implementation Issues

6.4 Reasons to Create a Class

6.5 Language-Specific Issues

6.6 Beyond Classes: Packages

A key to being an effective programmer is maximizing the portion of a program that you can safely ignore while working on any one section of code. Classes are the primary tool for accomplishing that objective.

# 6.1. ADTs

*An abstract data type is realized by writing a special kind of program […] which defines the type in terms of the operations which can be performed on it.*

Barbara Liskov, Programming with Abstract Data Types

## 6.1.1. Class Foundations: Abstract Data Types (ADTs)

An abstract data type is a collection of data and operations that work on that data.

The operations both describe the data to the rest of the program and allow the rest of the program to change the data. The word "data" in "abstract data type" is used loosely. An ADT might be a graphics window with all the operations that affect it, a file and file operations, an insurance-rates table and the operations on it, or something else.

Abstract data types are exciting because you can use them to manipulate real-world entities rather than low-level, implementation entities.

Tap into the power of being able to work in the problem domain rather than at the low-level implementation domain!

Understanding ADTs is essential to understanding object-oriented programming. Without understanding ADTs, programmers create classes that are “classes” in name only—in reality, they are little more than convenient carrying cases for loosely related collections of data and routines.

Here are a few more examples of ADTs:

Suppose you’re writing software that controls the cooling system for a nuclear

reactor. You can treat the cooling system as an abstract data type by defining the

following operations for it:

coolingSystem.Temperature()

coolingSystem.SetCirculationRate( rate )

coolingSystem.OpenValve( valveNumber )

coolingSystem.CloseValve( valveNumber )

The specific environment would determine the code written to implement each of these operations. The rest of the program could deal with the cooling system through these functions and wouldn’t have to worry about internal details of data-structure implementations, data-structure limitations, changes, and so on.

## 6.1.2. Benefits of using ADTs

The problem isn't that the ad hoc approach is bad programming practice. It’s that you can replace the approach with a better programming practice that produces these benefits:

* You can hide implementation details.
* Changes don't affect the whole program.
* You can make the interface more informative.
* The program is more obviously correct.
* The program becomes more self-documenting.
* You don’t have to pass data all over your program..
* You’re able to work with real-world entities rather than with low-level implementation structures.

[Ver más ejemplos de ADTs en página 149]

### Build or use typical low-level data types as ADTs, not as low-level data types

The question you need to ask is, what does this stack, list, or queue represent?

* If a stack represents a set of employees, treat the ADT as employees rather than as a stack.
* If a list represents a set of billing records, treat it as billing records rather than a list.
* If a queue represents cells in a spreadsheet, treat it as a collection of cells rather than a generic item in a queue.

Treat yourself to the highest possible level of abstraction.

### Treat common objects such as files as ADTs

A high-level language protects you from the messy details of generating operating-system calls and manipulating data buffers. It allows you to treat a chunk of disk space as a “file.”

You can layer ADTs similarly. If you want to use an ADT at one level that offers data-structure level operations (like pushing and popping a stack), that’s fine.

You can create another level on top of that one that works at the level of the realworld problem.

### Treat even simple items as ADTs

You don’t have to have a formidable data type to justify using an abstract data type.

One of the ADTs in the example list is a light that supports only two operations—turning it on and turning it off.

Putting the light and its operations into an ADT makes the code more self-documenting and easier to change, confines the potential consequences of changes to the TurnLightOn() and TurnLightOff() routines, and reduces the amount of data you have to pass around.

### Refer to an ADT independently of the medium it’s stored on

Try to make the names of classes and access routines independent of how the data is stored, and refer to the abstract data type, like the insurance-rates table, instead. That would give your class and access routine names like rateTable.Read() or simply rates.Read()

## 6.1.3. Handling Multiple Instances of Data with ADTs in Non-OO Environments

Object-oriented languages provide automatic support for handling multiple instances of an ADT.

If you’re working in a non-object oriented environment such as C, you will have to build support for multiple instances manually.

Example:

The font ADT originally offered these services:

currentFont.SetSize( sizeInPoints )

currentFont.BoldOn()

currentFont.BoldOff()

currentFont.ItalicOn()

currentFont.ItalicOff()

currentFont.SetTypeFace( faceName )

In a non-OO environment, these functions would not be attached to a class, and would look more like this:

SetCurrentFontSize( sizeInPoints )

SetCurrentFontBoldOn()

SetCurrentFontBoldOff()

SetCurrentFontItalicOn()

SetCurrentFontItalicOff()

SetCurrentFontTypeFace( faceName )

If you want in work with more than one font at a time, you’ll need to add services to create and delete font instances—maybe these:

CreateFont( fontId )

DeleteFont( fontId )

SetCurrentFont( fontId )

## ADTs and Classes

Abstract data types form the foundation for the concept of classes.

In languages that support classes, you can implement each abstract data type in its own class.

Classes usually involve the additional concepts of inheritance and polymorphism. One way of thinking of a class is as an abstract data type plus inheritance and polymorphism.

# 6.2. Good Class Interfaces

Creating a good abstraction for the interface to represent and ensuring that the details remain hidden behind the abstraction.

## 6.2.1. Good Abstraction.

The class's interface should offer a group of routines that clearly belong together.

A class that presents a poor abstraction would be one that contained a collection of miscellaneous functions.

C++ Example of a Class Interface that Presents a Good Abstraction

class Employee {

public:

// public construct ors and destructors

Employee();

Employee(

FullName name,

String address,

String workPhone,

String homePhone,

TaxId taxIdNumber,

JobClassification jobClass;

virtual ~Employee();

// public routines

FullName Name();

String Address();

String WorkPhone();

String HomePhone();

TaxId TaxIdNumber();

JobClassification GetJobClassification();

...

private:

…

}

C++ Example of a Class Interface that Presents a Poor Abstraction

class Program {

public:

...

// public routines

void InitializeCommandStack();

void PushCommand( Command &command );

Command PopCommand();

void ShutdownCommandStack();

void InitializeReportFormatting();

void FormatReport( Report &report );

void PrintReport( Report &report );

void InitializeGlobalData();

void ShutdownGlobalData();

…

private:

…

}

The class interface doesn't present a consistent abstraction, so the class has poor cohesion. The routines should be reorganized into more-focused classes, each of which provides a better abstraction in its interface.

* This evaluation of class abstraction is based on the class's collection of public routines
* The routines inside the class don't necessarily present good individual abstractions

### Each class should implement one and only one ADT.

C++ Example of a Class Interface with Mixed Levels of Abstraction

class EmployeeList: public ListContainer {

public:

...

// public routines

void AddEmployee( Employee &employee );

void RemoveEmployee( Employee &employee );

Employee NextItemInList( Employee &employee );

Employee FirstItem( Employee &employee );

Employee LastItem( Employee &employee );

...

private:

…

}

This sort of mixed abstraction commonly arises when a programmer uses a container class or other library classes for implementation and doesn't hide the fact that a library class is used.

* If you think of the class's public routines as an air lock that keeps water from getting into a submarine, inconsistent public routines are leaky panels in the class. In practice, this is what happens when you mix levels of abstraction.

### Provide services in pairs with their opposites.

Most operations have corresponding, equal, and opposite operations. Don’t create an opposite gratuitously, but do check to see whether you need one.

### Move unrelated information to another class.

In some cases, you'll find that half a class's routines work with half the class's data and half the routines work with the other half of the data. In such a case, you really have two classes masquerading one.

### Look for ways to convert semantic interface elements to programmatic interface elements by using Asserts or other techniques.

Las aserciones son una forma de documentación “ejecutable” (diseño por contrato).

### Don't add public members that are inconsistent with the interface abstraction.

Each time you add a routine to a class interface, ask "Is this routine consistent with the abstraction provided by the existing interface?" If not, find a different way to make the modification and preserve the integrity of the abstraction.

### Consider abstraction and cohesion together.

The ideas of abstraction and cohesion are closely related—a class interface that presents a good abstraction usually has strong cohesion.

## 6.2.1. Good Encapsulation.

Abstraction helps to manage complexity by providing models that allow you to ignore implementation details. Encapsulation is the enforcer that prevents you from looking at the details even if you want to.

The two concepts are related because, without encapsulation, abstraction tends to break down. In my experience, either you have both abstraction and encapsulation or you have neither. There is no middle ground.

### Routines: public, private, or protected

If you're wondering whether a specific routine should be public, private, or protected, one school of thought is that you should favor the strictest level of privacy that's workable (Meyers 1998, Bloch 2001).

But the more important guideline is, "What best preserves the integrity of the interface abstraction?" If exposing the routine is consistent with the abstraction, it's probably fine to expose it.

### Don't expose member data in public.

Exposing member data is a violation of encapsulation and limits your control over the abstraction.

Bad Example: a Point class that exposes:

float x;

float y;

float z;

is violating encapsulation

OK:

float GetX();

float GetY();

float GetZ();

void SetX( float x );

void SetY( float y );

void SetZ( float z );

is maintaining perfect encapsulation.

### Avoid putting private implementation details into a class's interface.

### Don't make assumptions about the class's users.

A class should be designed and implemented to adhere to the contract implied by the class interface. It shouldn't make any assumptions about how that interface will or won't be used,

### Avoid friend classes.

In a few circumstances such as the State pattern, friend classes can be used in a disciplined way that contributes to managing complexity (Gamma et al. 1995). But, in general, friend classes violate encapsulation.

### Don't put a routine into the public interface just because it uses only public routines.

### Favor read-time convenience to write-time convenience.

Code is read far more times than it's written, even during initial development.

### Be very, very wary of semantic violations of encapsulation.

At one time I thought that when I learned how to avoid syntax errors I would be home free.

It ain't abstract if you have to look at the underlying implementation to understand what's going on. — P. J. Plauger

Here are some examples of the ways that a user of a class can break encapsulation semantically:

* Not calling the database.Connect() routine before you call employee.Retrieve( database ) because you know that the employee.Retrieve() function will connect to the database if there isn't already a connection.
* Not calling Class A's Terminate() routine because you know that Class A's PerformFinalOperation() routine has already called it.

Anytime you find yourself looking at a class's implementation to figure out how to use the class, you're not programming to the interface; you're programming through the interface to the implementation.

* Once encapsulation starts to break down, abstraction won't be far behind.
* If you can't figure out how to use a class based solely on its interface documentation, the right response is not to pull up the source code and look at the implementation.

### Watch for coupling (acoplamiento) that's too tight.

"Coupling" refers to how tight the connection is between two classes. In general, the looser the connection, the better.

* Coupling goes hand in glove with abstraction and encapsulation. Tight coupling occurs when an abstraction is leaky, or when encapsulation is broken.
* If a class offers an incomplete set of services, other routines might find they need to read or write its internal data directly.

# 6.3 Design and Implementation Issues

Design and Implementation Issues Defining good class interfaces goes a long way toward creating a high-quality program. The internal class design and implementation are also important. This section discusses issues related to containment, inheritance, member functions and data, class coupling, constructors, and value-vs.-reference objects.

## Containment ("has a" Relationships)

Containment is the simple idea that a class contains a primitive data element or object. A lot more is written about inheritance than about containment, but that's because inheritance is more tricky and error-prone, not because it's better. Containment is the work-horse technique in object-oriented programming.

### Implement "has a" through containment.

One way of thinking of containment is as a "has a" relationship. For example, an employee "has a" name, "has a" phone number, "has a" tax ID, and so on. You can usually accomplish this by making the name, phone number, and tax ID member data of the Employee class. Implement "has a" through private inheritance as a last resort.

### Be critical of classes that contain more than about seven data members.

The number "7±2" has been found to be a number of discrete items a person can remember while performing other tasks (Miller 1956). If a class contains more than about seven data members, consider whether the class should be decomposed into multiple smaller classes (Riel 1996). You might err more toward the high end of 7±2 if the data members are primitive data types like integers and strings, more toward the lower end of 7±2 if the data members are complex objects.

## Inheritance ("is a" Relationships)

Inheritance is the idea that one class is a specialization of another class. The purpose of inheritance is to create simpler code by defining a base class that specifies common elements of two or more derived classes. The common elements can be routine interfaces, implementations, data members, or data types. Inheritance helps avoid the need to repeat code and data in multiple locations by centralizing it within a base class.

* For each member routine, will the routine be visible to derived classes? Will it have a default implementation? Will the default implementation be overridable?
* For each data member (including variables, named constants, enumerations, and so on), will the data member be visible to derived classes.

### Implement "is a" through public inheritance.

When a programmer decides to create a new class by inheriting from an existing class, that programmer is saying that the new class "is a" more specialized version of the older class.

public inheritance means "is a." Commit this rule to memory. — Scott Meyers

If the derived class isn't going to adhere completely to the same interface contract defined by the base class, inheritance is not the right implementation technique.

### Design and document for inheritance or prohibit it.

Inheritance adds complexity to a program, and, as such, it's a dangerous technique. As Java guru Joshua Bloch says, "Design and document for inheritance, or prohibit it." If a class isn't designed to be inherited from, make its members non-virtual in C++, final in Java, or non-overridable in Microsoft Visual Basic so that you can't inherit from it.

### Adhere to the Liskov Substitution Principle (LSP).

In one of object-oriented programming's seminal papers, Barbara Liskov argued that you shouldn't inherit from a base class unless the derived class truly "is a" more specific version of the base class (Liskov 1988). Andy Hunt and Dave Thomas summarize LSP like this: "Subclasses must be usable through the base class interface without the need for the user to know the difference" (Hunt and Thomas 2000).

In other words, all the routines defined in the base class should mean the same thing when they're used in each of the derived classes.

If a program has been written so that the Liskov Substitution Principle is true, inheritance is a powerful tool for reducing complexity because a programmer can focus on the generic attributes of an object without worrying about the details.

### Be sure to inherit only what you want to inherit.

A derived class can inherit member routine interfaces, implementations, or both. Table 6-1 shows the variations of how routines can be implemented and overridden.

If you want to use a class's implementation but not its interface, use containment rather than inheritance.

### Move common interfaces, data, and behavior as high as possible in the inheritance tree.

The higher you move interfaces, data, and behavior, the more easily derived classes can use them. How high is too high? Let abstraction be your guide. If you find that moving a routine higher would break the higher object's abstraction, don't do it.

### Be suspicious of classes of which there is only one instance.

A single instance might indicate that the design confuses objects with classes. Consider whether you could just create an object instead of a new class. Can the variation of the derived class be represented in data rather than as a distinct class? The Singleton pattern is one notable exception to this guideline.

### Be suspicious of classes that override a routine and do nothing inside the derived routine.

This typically indicates an error in the design of the base class.

Ejemplo: For instance, suppose you have a class Cat and a routine Scratch()

### Avoid deep inheritance trees

In my experience most people have trouble juggling more than two or three levels of inheritance in their brains at once. The "magic number 7±2" is probably better applied as a limit to the total number of subclasses of a base class rather than the number of levels in an inheritance tree.

Deep inheritance trees increase complexity, which is exactly the opposite of what inheritance should be used to accomplish.

Make sure you're using inheritance to avoid duplicating code and to minimize complexity.

### Prefer polymorphism to extensive type checking.

Frequently repeated case statements sometimes suggest that inheritance might be a better design choice, although this is not always true.

### Make all data private, not protected.

As Joshua Bloch says, "Inheritance breaks encapsulation" (2001). When you inherit from an object, you obtain privileged access to that object's protected routines and data. If the derived class really needs access to the base class's attributes, provide protected accessor functions instead.

## Multiple Inheritance

Multiple Inheritance Inheritance is a power tool. It's like using a chain saw to cut down a tree instead of a manual crosscut saw.

If inheritance is a chain saw, multiple inheritance is a 1950s-era chain saw with no blade guard, no automatic shutoff, and a finicky engine. There are times when such a tool is valuable; mostly, however, you're better off leaving the tool in the garage where it can't do any damage.

Java and Visual Basic recognize the value of mixins by allowing multiple inheritance of interfaces but only single-class inheritance. C++ supports multiple inheritance of both interface and implementation.

## Why Are There So Many Rules for Inheritance?

This section has presented numerous rules for staying out of trouble with inheritance. The underlying message of all these rules is that inheritance tends to work against the primary technical imperative you have as a programmer, which is to manage complexity.

Here's a summary of when to use inheritance and when to use containment:

* If multiple classes share common data but not behavior, create a common object that those classes can contain.
* If multiple classes share common behavior but not data, derive them from a common base class that defines the common routines.
* If multiple classes share common data and behavior, inherit from a common base class that defines the common data and routines.
* Inherit when you want the base class to control your interface; contain when you want to control your interface.

## Member Functions and Data

Here are a few guidelines for implementing member functions and member data effectively.

### Keep the number of routines in a class as small as possible.

strong coupling between classes. Evaluate the tradeoff between minimizing the number of routines and these other factors.

### Disallow implicitly generated member functions and operators you don't want.

The same study found that the more classes a class used, the higher its fault rate tended to be. These concepts are sometimes called "fan out."

### Minimize indirect routine calls to other classes.

Direct connections are hazardous enough. Indirect connections—such as account.ContactPerson().DaytimeContactInfo().PhoneNumber()—tend to be even more hazardous. Researchers have formulated a rule called the "Law of Demeter" (Lieberherr and Holland 1989), which essentially states that Object A can call any of its own routines. If Object A instantiates an Object B, it can call any of Object B's routines. But it should avoid calling routines on objects provided by Object B. In the account example above, that means account.ContactPerson() is OK but account.ContactPerson().DaytimeContactInfo() is not.

### In general, minimize the extent to which a class collaborates with other classes.

Try to minimize all of the following: Number of kinds of objects instantiated Number of different direct routine calls on instantiated objects Number of routine calls on objects returned by other instantiated objects.

## Constructors

Following are some guidelines that apply specifically to constructors. Guidelines for constructors are pretty similar across languages (C++, Java, and Visual Basic, anyway). Destructors vary more,

### Initializing all data members in all constructors is an inexpensive defensive programming practice.

Enforce the singleton property by using a private constructor. If you want to define a class that allows only one object to be instantiated, you can enforce this by hiding all the constructors of the class and then providing a static GetInstance() routine to access the class's single instance.

### Prefer deep copies to shallow copies until proven otherwise.

One of the major decisions you'll make about complex objects is whether to implement deep copies or shallow copies of the object.

# 6.4. Reasons to Create a Class

If you believe everything you read, you might get the idea that the only reason to create a class is to model real-world objects. In practice, classes get created for many more reasons than that.

### Model real-world objects.

Modeling real-world objects might not be the only reason to create a class, but it's still a good reason! Create a class for each real-world object type that your program models. Put the data needed for the object into the class, and then build service routines that model the behavior of the object.

### Model abstract objects.

Another good reason to create a class is to model an abstract object—an object that isn't a concrete, real-world object but that provides an abstraction of other concrete objects. A good example is the classic Shape object. Circle and Square really exist, but Shape is an abstraction of other specific shapes.

### Reduce complexity.

The single most important reason to create a class is to reduce a program's complexity. Create a class to hide information so that you won't need to think about it. Sure, you'll need to think about it when you write the class. But after it's written, you should be able to forget the details and use the class without any knowledge of its internal workings.

If an error does occur, it will be easier to find if it isn't spread through the code but is localized within a class. Changes arising from fixing the error won't affect other code because only one class will have to be fixed—other code won't be touched.

### Hide implementation details. => SRP

Limit effects of changes. Isolate areas that are likely to change so that the effects of changes are limited to the scope of a single class or a few classes.

### Hide global data.

If you need to use global data, you can hide its implementation details behind a class interface. Working with global data through access routines provides several benefits compared to working with global data directly. You can change the structure of the data without changing your program.

### Streamline parameter passing.

If you're passing a parameter among several routines, that might indicate a need to factor those routines into a class that share the parameter as object data.

### Make central points of control.

It's a good idea to control each task in one place. Control assumes many forms. Knowledge of the number of entries in a table is one form. Control of devices—files, database connections, printers, and so on—is another. Using one class to read from and write to a database is a form of centralized control.

### Facilitate reusable code.

Code put into well-factored classes can be reused in other programs more easily than the same code embedded in one larger class.

### Plan for a family of programs.

If you expect a program to be modified, it's a good idea to isolate the parts that you expect to change by putting them into their own classes. You can then modify the classes without affecting the rest of the program, or you can put in completely new classes instead. Thinking through not just what one program will look like but what the whole family of programs might look like is a powerful heuristic for anticipating entire categories of changes (Parnas 1976).

### Package related operations.

In cases in which you can't hide information, share data, or plan for flexibility, you can still package sets of operations into sensible groups, such as trig functions, statistical functions, string-manipulation routines, bit-manipulation routines, graphics routines, and so on. Classes are one means of combining related operations. You could also use packages, namespaces, or header files, depending on the language you're working in.

### Accomplish a specific refactoring.

Many of the specific refactorings described in Chapter 24, result in new classes—including converting one class to two,

## Classes to Avoid

While classes in general are good, you can run into a few gotchas. Here are some classes to avoid.

### Avoid creating god classes.

Avoid creating omniscient classes that are all-knowing and all-powerful.

### Eliminate irrelevant classes.

If a class consists only of data but no behavior, ask yourself whether it's really a class and consider demoting it so that its member data just becomes attributes of one or more other classes.

### Avoid classes named after verbs.

A class that has only behavior but no data is generally not really a class. Consider turning a class like DatabaseInitialization() or String-Builder() into a routine on some other class.

## Summary of Reasons to Create a Class

Here's a summary list of the valid reasons to create a class:

* Model real-world objects
* Model abstract objects
* Reduce complexity
* Isolate complexity
* Hide implementation details
* Limit effects of changes
* Hide global data
* Streamline parameter passing
* Make central points of control
* Facilitate reusable code
* Plan for a family of programs
* Package related operations
* Accomplish a specific refactoring

# CHECKLIST: Class Quality

Abstract Data Types

* Have you thought of the classes in your program as Abstract Data Types and evaluated their interfaces from that point of view?

Abstraction

* Does the class have a central purpose?
* Is the class well named, and does its name describe its central purpose?
* Does the class’s interface present a consistent abstraction?
* Does the class’s interface make obvious how you should use the class?
* Is the class’s interface abstract enough that you don’t have to think about how its services are implemented? Can you treat the class as a black box?
* Are the class’s services complete enough that other classes don’t have to meddle with its internal data?
* Has unrelated information been moved out of the class?
* Have you thought about subdividing the class into component classes, and have you subdivided it as much as you can?
* Are you preserving the integrity of the class’s interface as you modify the class?

Encapsulation

* Does the class minimize accessibility to its members?
* Does the class avoid exposing member data?
* Does the class hide its implementation details from other classes as much as the programming language permits?
* Does the class avoid making assumptions about its users, including its derived classes?
* Is the class independent of other classes? Is it loosely coupled?

Inheritance

* Is inheritance used only to model “is a” relationships?
* Does the class documentation describe the inheritance strategy?
* Do derived classes adhere to the Liskov Substitution Principle?
* Do derived classes avoid “overriding” non overridable routines?
* Are common interfaces, data, and behavior as high as possible in the inheritance tree?
* Are inheritance trees fairly shallow?
* Are all data members in the base class private rather than protected?

Other Implementation Issues

* Does the class contain about seven data members or fewer?
* Does the class minimize direct and indirect routine calls to other classes?
* Does the class collaborate with other classes only to the extent absolutely necessary?
* Is all member data initialized in the constructor?
* Is the class designed to be used as deep copies rather than shallow copies unless there’s a measured reason to create shallow copies?

Language-Specific Issues

* Have you investigated the language-specific issues for classes in your specific programming language?