The Abstract Class Pattern

Bobby Woolf

Knowledge Systems Corp.
4001 Weston Pkwy, Cary, NC 27513-2303
919-677-1119 x541, bwoolf@ksccary.com

ABSTRACT CLASS — Class Behavioral

Intent

Define the interface for a hierarchy of classes while deferring the implementation to subclasses. Abstract Class lets subclasses redefine the implementation of an interface while preserving the polymorphism of those classes.

Also Known As

Liskov Substitution Principle [LW93], Design by Contract [Meyer91], Base Class [Auer95] , Template Class [Woolf97]

Motivation

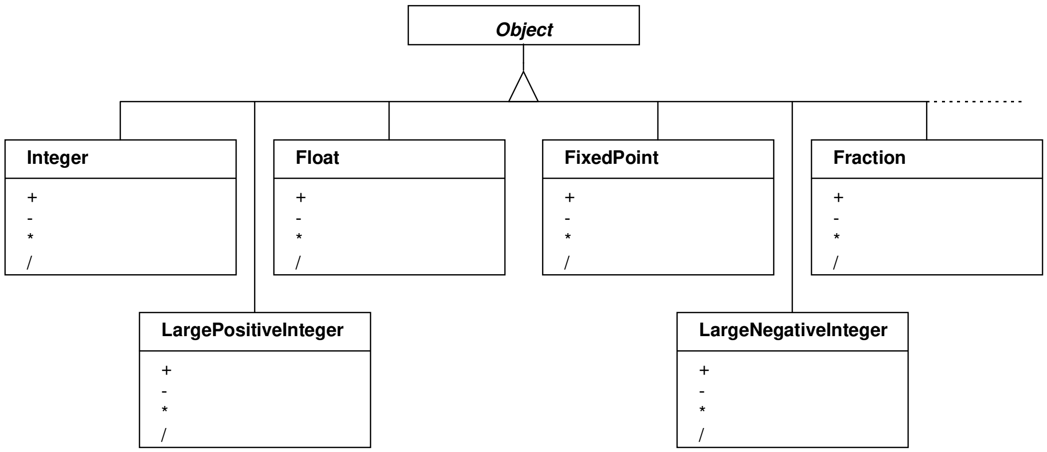
Consider the need to perform simple arithmetic. Every application needs to use simple numbers like integers and floats and to perform simple arithmetic such as addition, subtraction, multiplication, and division.

One obvious way to perform this simple math is to let the CPU do it. Any modern CPU has built in commands to perform simple arithmetic with integers and floats. This is the most efficient way to perform such calculations.

The problem is that not all numerical quantities can be adequately represented as the CPU's integers and floats. Integers have a limited range. Floats have limited precision and loose precision converting between decimal and binary.

The number framework in a robust object-oriented system should take advantage of the CPU's efficiency whenever possible. However, to make the system more robust, the framework should overcome the CPU's limitations whenever possible. It should be able to represent a virtually limitless range of numbers, both really huge numbers and really tiny ones. It should be able to represent a decimal number with complete precision, at least to a specified number of decimal places. It should be able to perform simple arithmetic without any loss of precision. It could even compute complex equations by simplifying them first.

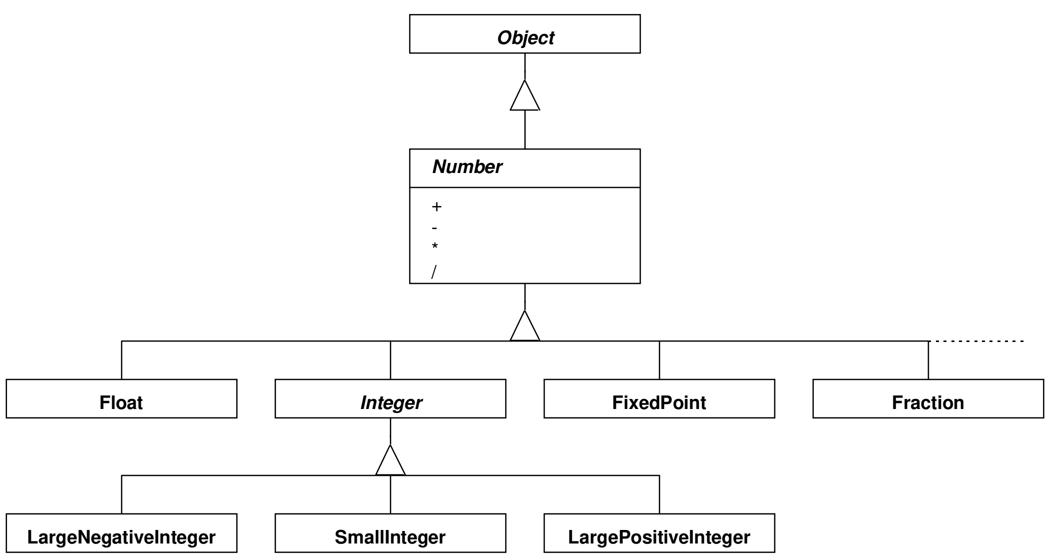
A robust number framework employs various classes to meet these goals: **`Integer`** and **`Float`** for CPU numbers, **`LargePositiveInteger`** and **`LargeNegativeInteger`** for huge integer values, **`FixedPoint`** for complete precision, **`Fraction`** for division without round off, and so on. This way, the framework performs as much computation as possible using the CPU, but also uses other classes to represent numbers that that the CPU cannot. The diagram below shows the classes for this framework.



The problem with all of these number classes is that the rest of the system does not want to be aware of them. To the rest of the system, there are just number objects and they know how to perform arithmetic. When code somewhere in the system has a statement like "x + y," it does not care whether **`x`** is a **`Float`** or **`y`** is a **`Fraction`**. The code just knows that **`x`** and **`y`** are numbers and that numbers know how to perform addition. The implication is that since some numbers know how to perform addition, all numbers must be able to.

Thus the number framework requires more than just these various number classes. It also needs to clearly show which classes are part of the framework. It needs to require that all classes in the framework be able to perform a certain minimal amount of functionality, such as addition. And the framework needs to provide all of this number functionality in a polymorphic way to hide the complexity of the various subclasses from the rest of the system.

The framework will accomplish all of this by using a generalized superclass called **`Number`**. A **`Number`** represents any kind of number, be it an integer, float, or whatever. It defines the minimal functionality that any number must provide, such as addition. It does not define a number's structure, nor does it define the implementation of the functionality. Those details are deferred to subclasses like **`Integer`**, **`Float`**, and so on. Applying **`Number`** as a superclass, and implementing **`Integer`** in a similar manner, leads to the framework of classes shown below.



All number classes are now subclasses of **`Number`**. They are defined to provide basic arithmetic. A client using a couple of number objects knows that numbers can perform basic arithmetic regardless of which subclasses the objects are.

`Number` is an example of the Abstract Class pattern. As described in *Design Patterns*, "An abstract class is one whose main purpose is to define a common interface for its subclasses." [GHJV95, page 15] **`Number`** defines a type that can be implemented several different ways, but all of these implementations will have the same interface so that clients can use them interchangeably. This simplifies client code by causing the code to describe *what* it wants done without specifying *how* it should be done. The client code will even work with unknown, future implementations, as long as those implementations fulfill the interface.

The key to the Abstract Class pattern is a superclass that defines the type for its hierarchy and subclasses that provide various implementations of the type. An abstract class can be implemented as pure interface such that its subclasses must implement all of its messages, but it is more useful still if it also provides a partial implementation suitable for all of the subclasses. In this way, a subclass inherits the partial implementation and only needs to complete it.

The superclass is called "abstract" because its implementation is incomplete so clients do not create instances of it. The subclasses that a client can create instances of are referred to as "concrete." [WWW90, page 27]

Keys

A framework that incorporates the Abstract Class pattern has the following features:

- A superclass that defines a type.
- One or more subclasses that implement the type.
- Polymorphism between the subclasses because they share the interface defined by the superclass.

The framework may also include these variations on the pattern:

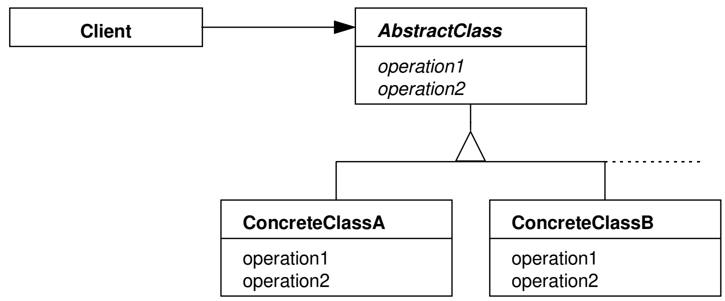
- The superclass may provide a partial but incomplete implementation.
- The superclass may provide a complete implementation that is a default or minimal implementation.
- The superclass may define state as well as interface.
- The subclasses may expand the interface defined by the superclass to include additional functionality. However, that extended interface will not be polymorphic with the other classes.

Applicability

Use the Abstract Class pattern when:

- a framework requires several classes that have the same interface or whose interfaces overlap to form a common, core interface.
- the common interface should be defined in one place so that all of the classes know they should adhere to this interface and so that clients know what interface they can expect.
- a hierarchy should be extensible so that future subclasses can easily be added without having to change the existing superclass or the existing client code.

Structure



Participants

- **AbstractClass** (**`Number`**, **`Integer`**)
  - defines the interface that all of the ConcreteClasses share.
  - does not define state and implementation unless it is common to all concrete classes, including future ones.
  - may itself be a more specific subclass of another AbstractClass.
- **ConcreteClass** (**`FixedPoint`**, **`Float`**, **`Fraction`**, **`LargeNegativeInteger`**, **`LargePositiveInteger`**, **`SmallInteger`**)
  - is a direct or indirect subclass of its AbstractClass.
  - implements the interface inherited from AbstractClass.
  - declares state necessary to implement interface.
- **Client**
  - collaborates with ConcreteClass instances through the AbstractClass interface.

Collaborations

- Clients use the AbstractClass interface to interact with objects that may be any ConcreteClass.
- ConcreteClass relies on AbstractClass to provide the default implementations common to all ConcreteClasses.

Consequences

The advantages of the Abstract Class pattern are:

- *Class polymorphism.* The ConcreteClasses are polymorphic with each other because they all support the common interface defined by the AbstractClass. This means that a client can use any of the ConcreteClasses without regard to which ConcreteClass it is using. Such common interfaces make extensibility easier because client code will be able to use future ConcreteClasses that haven't even been written yet, as long as those future classes adhere to the interface.
- *Algorithm reuse.* If the AbstractClass does contain any implementation, it is usually in the form of Template Methods [GHJV95, page 325]. This is implementation (and in some cases even state) that is common to all of the ConcreteClasses, present and future, and so can be reused by implementing it once in the AbstractClass.

The disadvantages of the Abstract Class pattern are:

- *Abstract vs. concrete.* Clients generally assume that they can create an instance of any class, but this is not the case with abstract classes. Clients should not attempt to instantiate instances of abstract classes, only concrete ones. The superclass is said to be "abstract" because it has an incomplete implementation, so the client cannot or should create instances of it. Each subclass is said to be "concrete" because its implementation is complete, so the client can create instances of it.

- *Single hierarchy.* The pattern forces all of the ConcreteClasses to be gathered together into a single class hierarchy with one common superclass, the AbstractClass. Sometimes a class seems to belong in one hierarchy because of its type but in another hierarchy so that it can inherit some of its implementation. In such a case, it is better to implement the class in its type hierarchy and let it delegate to an instance from its implementation hierarchy.

  Another way this problem can occur is when disparate classes need to implement the same operation. The default implementation for this operation is usually defined in the first superclass they all have in common. This in effect makes that superclass an AbstractClass for those subclasses. However, it also makes the class an AbstractClass for all of the other subclasses, even though they don't need the operation. Thus it becomes clear that inheritance is not the best way to define this operation and reuse its implementation. The classes that need the operation would be better of delegating to an object that has the operation.

- *Overly specific interface.* Sometimes a ConcreteClass is not prepared to implement all of the operations that an AbstractClass specifies. For example, the **`Collection`** abstract class in Smalltalk specifies **`add:`** and **`remove:`** operations, but the **`Array`** subclass cannot implement them. When possible, the AbstractClass should not specify an operation unless all of its ConcreteClasses will be able to implement it. When this cannot be avoided, the ConcreteClass must implement the operation anyway, usually to issue an error.

Implementation

There are several issues to consider when implementing the Abstract Class pattern:

1. *Separate classes.* A object is often implemented using a single class that defines both the object's interface and its implementation. This makes the abstraction that the class represents difficult to reuse. The Abstract Class pattern suggests that the object should be implemented with two classes, an abstract one that defines its interface and a concrete one that implements the interface.

2. *No state.* The AbstractClass usually does not declare any state variables. If it did, all of its ConcreteClasses would be forced to inherit those variables. This would be inefficient for a ConcreteClass whose implementation did not require those variables. However, if all of the ConcreteClasses require a variable, and if future ConcreteClasses would also probably require that variable, then it can be declared in the AbstractClass.

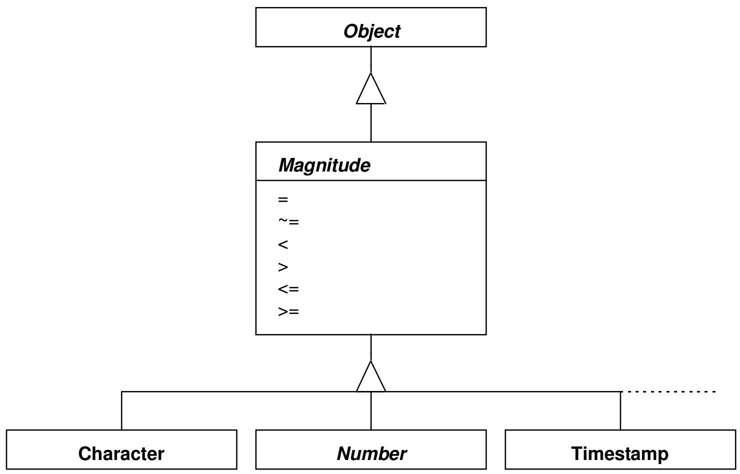
3. *Implementation through Template Methods.* An AbstractClass is usually implemented as a collection of Template Methods [GHJV95, page 325]. An AbstractClass is said to define an interface but leave its implementation to the ConcreteClasses. However, when a message has a default implementation that is appropriate for all ConcreteClasses, that implementation can be made in the AbstractClass. Such an implementation is often either a template method or a primitive operation method.

4. *No private messages.* The AbstractClass defines the hierarchy's interface. These are the public messages that the subclasses will implement. The AbstractClass does not need to define private messages and usually does not do so. However, it may define private messages that are primitive operations of the class' template methods.

Sample Code

The **Magnitude** hierarchy in Smalltalk is an excellent example of the Abstract Class pattern. It includes the **Number** hierarchy discussed in the Motivation because numbers are magnitudes.

A **Magnitude** understands six main messages: equal-to (**=**), not-equal-to (**~=**), less-than (**<**), greater-than (**>**), less-than-or-equal-to (**<=**), and greater-than-or-equal-to (**>=**). Some examples of **Magnitude**s include **Number**, **Timestamp**, and **Character**. All of these are subclasses of **Magnitude** and understand the **Magnitude** messages, as shown below.



Four of the messages are implemented as Template Methods: not-equal-to, greater-than, less-than-or-equal-to, and greater-than-or-equal-to. They are implemented in terms of two primitive operations: equal-to and less-than. The primitive operations are deferred to subclasses.

```
Magnitude>>= aMagnitude
      ^self subclassResponsibility

Magnitude>>~= aMagnitude
      ^(self = aMagnitude) not

Magnitude>>< aMagnitude
      ^self subclassResponsibility

Magnitude>>> aMagnitude
      ^(self <= aMagnitude) not

Magnitude>><= aMagnitude
      ^(self = aMagnitude) or: [self < aMagnitude]

Magnitude>>>= aMagnitude
      ^(self < aMagnitude) not
```

So a subclass of **Magnitude** need only implement the two primitive operations and it gets the other four operations for free. For example, **Character** assumes that characters are ASCII and so sorts them into ASCII order. To do this, it uses a message that returns a character's ASCII value, such as **asciiValue**.

```
Character>>= aCharacter
      ^(self asciiValue) = (aCharacter asciiValue)

Character>>< aCharacter
      ^(self asciiValue) < (aCharacter asciiValue)
```

Thus the AbstractClass, **`Magnitude`**, greatly simplifies the implementation of the various ConcreteClasses. In the sample operations shown here, implementing two messages gives the class four more messages for free.

The AbstractClass also simplifies the interface that client code must understand. The client must know that it is comparing two objects of the same subtype: two **`Character`**s or two **`Number`**s, etc. However, it need not care which subtype they are, because they all behave same. They all understand the same six comparison messages.

One example of why this is useful is the way **`SortedCollection`** works. A **`Sorted-Collection`** is a **`Collection`** that sorts its elements into order. By default, it assumes that its elements are **`Magnitude`**s (of the same subtype) and it uses **`<=`** to sort them into order. Thus it does not care whether the elements are **`Character`**s, **`Number`**s, or **`Timestamp`**s; since they are all **`Magnitude`**s, they will all work correctly.

Known Uses

Abstract Classes are so fundamental that they can be found in almost any multilevel class hierarchy. In such a hierarchy, the superclasses are usually abstract; the leaf classes must be concrete. **`Object`**, the root class for the entire Smalltalk hierarchy, is the ultimate Abstract Class in that language. In Java, **`java.lang.Object`** serves the same purpose. When a class hierarchy is known by the class at the root of the hierarchy (such as Number, Collection, Stream, Window, etc.), that class is almost always an abstract class.

Almost every documented design pattern, such as those in *Design Patterns* [GHJV95], features one or more Abstract Classes. Often the pattern suggests the creation of an Abstract Class if there isn't one already. For example, Composite [GHJV95, page 163] uses the Component abstract class to define the interface for both the Leaf and Composite classes. To apply Proxy [GHJV95, page 207] to a RealSubject class, the developer should use the abstract class Subject to define the interface that the RealSubject and its Proxy will share. When a pattern talks about a participant that "defines an interface" ["State," GHJV95, page 306] or "declares an interface" ["Strategy," GHJV95, page 317] for several subclasses, it is describing an Abstract Class.

Auer discusses how to develop class hierarchies that are reusable and extensible [Auer95]. He suggests using a base class to define and interface and subclasses to implement state.

Related Patterns

Most design-level patterns employ abstract classes. Of the twenty-three patterns in *Design Patterns*, twenty of them suggest implementing abstract classes (Singleton, Facade, and Memento do not). [GHJV95]

An Abstract Class is usually implemented using Template Methods [GHJV95, page 325].

Auer walks the reader through the process of developing a hierarchy whose interface is defined by an abstract class. [Auer95]

References

[Auer95] Ken Auer. "Reusability Through Self-Encapsulation." *Pattern Languages of Program Design*. Edited by James Coplien and Douglas Schmidt. Addison-Wesley, 1995.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[LW93] Barbara Liskov and Jeannette Wing. "A New Definition of the Subtype Relation." *ECOOP '93, Lecture Notes on Computer Science 707*. Berlin, Heidelberg: Springer-Verlag, 1993, pp. 118-141.

[Meyer91] Bertrand Meyer. "Design by Contract." *Advances in Object-Oriented Software Engineering*. Edited by Dino Mandrioli and Bertrand Meyer. Prentice-Hall, 1991, pp. 1-50.

[WWW90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.

[Woolf97] Bobby Woolf. "Polymorphic Hierarchy." *The Smalltalk Report*. January, 1997. 6(4).

Acknowledgments

I would like to thank Dana Anthony and Steve Berzcuk for their help in improving this paper.