Introduction

Object-oriented paradigm presents the most direct and natural medium for modeling automation problems. The linguistic mechanism known as class is a versatile schema for expressing modeling concepts of object-oriented paradigm.

The class schema begins with a keyword, which signifies the overall semantics of the schema. For instance, consider the type-definition mechanisms enumeration, union, class, task and collection. The term task implies threading, while union implies sharing same space. Similarly, enumeration and, its generalization collection, signify ordering of values and the availability of successor and predecessor operators.

Modern mathematics begins with the recognition of the versatility of the schema of abstract space in modeling problems internal to mathematics itself. An abstract space is a set of entities and a set of functions satisfying certain axioms. The axioms of a mathematical space are part of its definition. In somewhat similar manner, it is possible to use Boolean expressions as part of the definition of a class.

The invariants of a class specify conditions that must hold among the members of the class, at all times. Constraints may disallow or reject the execution of methods. In what follows, we cover these concepts in more detail.

Invariants

An invariant is a relationship among members of a class. It specifies a correct state for any object that is an instance of the class. Therefore, violation of an invariant indicates an incorrect state, which must be handled, one way or another.

Two mechanisms are provided to deal with violations of invariants. Either, one chooses to raise an exception, or instead one triggers another method to repair the problem. Following is the syntax for specifying invariants. Note that invariant is a keyword.

```
invariant (Boolean expression) exception-name;
invariant (Boolean expression) method-call;
```

The semantics is that, when the expression becomes false the specified action will take place: either the exception will be raised, or the indicated method will be invoked. The triggered method can raise exceptions if it cannot repair the problem. After repairing the problem all invariants must be true.

The method invoked to repair the violation of an invariant must be none-public, or the compiler will generate error. This is necessary to avoid infinite recursion, as indicated in the next section.

A class can have any number of invariants. Next, we discuss when the invariants are tested, and their relationship with inheritance.

Testing Invariant

Invariants are tested at end of execution of each **public** method, with the exception of a destructor. That is, just before the method returns, all invariants are checked.

When an invariant triggers a method, the call returns to the method that caused the trigger, unless the triggered method raises an exception. Thus, a trigger cannot be public.

Below is an example that shows the declaration of an invariant in context. Note that the default constructor will not raise exception because it sets both members to 0, and the invariant will be true. The invariant will be tested at end of method Modify.

```
class MyClass
      int integer;
      double real;

      invariant( integer <= real ) _someException;

public:

      void Modify(int, double);

end;
```

Constraints

Invariants ensure that objects remain in a desired state at all times during execution. When a method is invoked, we may also have to ensure that the object is in a state to respond to the message. This generally requires tests involving actual arguments to the method. After all, prior to executing the method the object invariants must already hold.

The conditions that must hold so an invoked method will execute are called constraints. Let us illustrate constraints by slightly modifying the previous example.

```
class MyClass
      int integer;
      double real;

      invariant( integer <= real ) _someException;

      void weightProblem(double);

public:

      void Modify(int height, double weight)
      {
          (height > 0 && weight > 0) _negative_Exception;
          (weight < real) weightProblem(weight);
      };

end;
```

The constraints are listed right after the prototype for the method Modify, between the braces. There are two constraints, one raises an exception and the other triggers a private method to repair the problem. The Boolean expressions must use the actual arguments, and may also use the members of the class.

Inheritance

Usually, the set of public methods of a class is referred to as its behavior in the form of an object. Invariants effectively make the state of an object a constituent of its behavior. After all, we refuse to do what we can just because we might be tired, or angry.

Invariants are integral part of an abstraction defined as a class. Therefore, we should specify invariants without regard to the **programming** context in which instances of a class will be used. A pleasant consequence of treating invariants as an abstraction independent of context is that future tools and compiler extensions can use them for program verification.

Invariants go down the inheritance hierarchy. That is, the invariants of a base class become part of invariants of its derived classes. Thus, derivation accumulates invariants. The state of base is part of the state of the derived class. In fact, a derived class may invoke methods of its base, in which case the invariants of the base will enter the picture.

Switching our attention to constraints, they may simply appear as syntactic sugar. One can always test the conditions in the method itself, and raise exceptions. However, separating the specification of constraints from the implementation of a method will allow future tools and compiler extensions to use them for program verification.

The view here is that, **the entire abstract behavior of a class, relatively speaking, must be observable at its definition**. Whether the implementation of methods achieves the intended goals is for testing to determine.

Constraints are not related to inheritance. This is discussed in the section on Constraints and Inheritance.

Contracts

Constraints of a method are conditions related to input to the implementation of the method. Although the implementation may be verified to be mathematically correct, it is generally valid for a certain range of values of the actual arguments.

The use of term contract instead of constraint implies the dependency of a method to the context in which it will be used. It is not possible to foresee all the uses of a class and its methods as a piece of software continues to extend.

A constraint that expresses a relation between an argument and a class member can be viewed as a parameterized constraint. The parameter is the class member used in the

specification of a constraint. Otherwise, the context in which the method is invoked should be oblivious of the state of the object itself.

A global function can only rely on the values of its arguments for any form of validity verification. However, methods of a class operate on the state of their object. Thus, invariants can ensure the correctness of output without the need for the method to verify a set of post-conditions.

Constraints and Inheritance

In Z++ constraints of a method are part of the definition of the method. That implies that a new definition for a method in a derived class does not inherit the constraints of its predecessor. However, if the new implementation invokes the predecessor method, its constraints become part of the new definition.

The use of term contract encourages a designer to apply some form of inheritance to constraints. In a language that supports multiple-inheritance this can result in ambiguities. Consider a base B with a method M. Suppose several classes are derived from B, each redefining M. Now derive a new class C from all derived classes, and redefine the method M of C. The question is how do the constraints of M in C relate to constraints of M in its parents?

Conclusion

Invariants for a class serve a purpose similar to axioms for an abstract space. We can only derive statements that are logical consequences of the axioms for the space. For a class, instead, the invariants ensure that after the execution of a method an object retains its correct state.

However, methods receive input in the form of arguments passed to them. Constraints ensure that an object is in a state to respond, as well as verifying that input is within acceptable range for the code of the method.

The linguistic mechanisms for specifying invariants and constrains facilitate automated verification and the creation of test sets. The presentation of these notions is separated from implementation details. The separation makes invariants and constraints accessible to future tools for automated verification.