Introduction

Z++ modules are true off-the-shelf universal components. This paper illustrates the use of invariants and constraints in conjunction with modules. This unique expressive capability allows the user of a component to control the correctness of its usage.

Z++ provides direct and intuitive support for platform-free component-oriented design and multi-layered architecture. More specifically, a Z++ program is also a reusable component that can be used in composing larger programs.

Unlike Z++ static libraries, a Z++ program must have at least one entry point. A Z++ program can also interact with another standalone program through the interface of its entry points. Thus, the set of entry points can be treated as simply as the Application Programming Interface (API) to a library, or the contract to use the program as a module in a component-oriented development, as well as multi-layered architecture.

Example

In order to illustrate the expressiveness of Z++ in dealing with components, we consider a small program with only three entry points. In practice, this can be any size program, with any number of entry points.

```
// Component.zpp

#include<iostream.h>

// Global declaration, classes, functions etc.

entry void main(void)
    output << "I prefer to be used as a component\n";
end;

entry int FirstEntry(int n)
      output << "FirstEntry : " << n << '\n';
      return 2 * n;
end;

entry double SecondEntry(double d)
      output << "SecondEntry : " << d << '\n';
      return d / 2;
end;
```

The users of this program can only use it as a component via its three entry points. When this program is executed on its own, by default its first entry point, in this example main will be invoked.

Now we write another simple program that uses the above as a component. Note how the component turns into a class, in a straightforward manner. The methods specified as external are the entry points, and the compiler is responsible for generating the code for their bodies. This is true whether the two programs reside on a single machine or not.

```
//CallingProgram.zpp

#include<iostream.h>

class Xinvariants = "Component.zxe" //turn component into class
  int i;

  invariant(i > 5) trigger(); //A simple class invariant

protected:

  void trigger(void);       //trigger used for invariant violation
  void constraint(double);  //trigger used for constraint violation

// These are the entry points of module Component.zpp

  external int FirstEntry(int);
  external double SecondEntry(double);

public:

  Xinvariants(void);

//-------------------------------------------------------------------
// These are the class methods that incorporate invariants
// and constraints, and call the corresponding module entry points.
//-------------------------------------------------------------------

  int XFirstEntry(int);

  double XSecondEntry(double b)
  {
    (b > 50.85) constraint(b);
  };

end;
```

We will omit the definition of bodies of class methods. In Z++ all method specifications are required in the definition of the class, only. As we mentioned, compiler generates the bodies of external methods, and locates and loads the executable for the module. However, this program also needs an entry point, and here we write it.

```
entry void main(void)
  output << "Hello World!\n";

  Xinvariants X;
  X.XFirstEntry(20);
  X.XSecondEntry(21.12);

  output << "Good-bye World!\n";
end;
```

As seen, the object X is an instance of the program Component.zpp. Furthermore, the class `Xinvariants` **controls the use of the component** via its invariants and the constraints that it specifies for its methods.

In the above example, we could have used "task" instead of "class" in the definition of the type `Xinvariants` and endowed its instance X with its own thread. Note that, the component itself could have a large number of threads. The one we are talking about here is the thread in the context of user of the component.

Multi-layered Architecture

We saw how a component can use another component by turning it into a class. This is the Z++ mechanism for composing large programs from smaller ones. Since classes can be derived from one another, one can easily extend modules and take advantage of polymorphism. However, there are other uses for Z++ composition mechanism.

A layer for a project is a set of one or more components (modules). At each layer of a large program, one deals with a different set of concerns and functionality. A plain set of classes does not represent a coherent functionality beyond a class library. On the other hand, each module in a layer cleanly packages a particular aspect or concern using sets of classes, along with global functions, libraries etc. Keep in mind that a module is a standalone program in its own right, lending itself to intrusive testing.

In Z++, modules comprising a layer can be remote or local. This simplicity allows independent development of components at disparate geographic locations. The same principle applies to the development and testing of layers of a large program. The final program is simply a composition of its layers, just as each layer is a composition of each of its components.

Conclusion

The literature on component-oriented development and multi-layered architecture is vast. However, **Z++ is the only language that provides intuitive solution for such illusive and yet significant concepts**. It makes components appear very much like UNIX commands that one pipes together in order to build new commands. In the same vain, one can snap together Z++ programs in order to build larger platform-free programs.

The user of a component can further control the component's usage by specifying its own invariants and constraints to suit its needs. On the other hand, a component can be updated and replaced exactly as required by component-oriented software development.

The simple mechanism of turning a module into a class makes multi-layered architecture straightforward and intuitive. A substantial percentage of the cost of development is in the gluing and other tedious and fragile development activities that are not even part of the actual solution. Z++ simply eliminates this burden allowing an engineer to concentrate on more creative work germane to the solution.