Chapter 4

An Incremental FPGA Integrated Design Environment

To demonstrate the incremental placement algorithm and the guided placement methodology discussed in Chapter 3, a prototype of a JBits-based incremental FPGA integrated design environment (IDE) has been developed. This chapter describes the program structures and the implementation of this design IDE. An example demonstrating this tool is presented as well.

4.1 Requirement Analysis

After developing the incremental placement algorithm and the guided placement methodology, it is necessary to build an infrastructure and a prototype of an incremental FPGA integrated design environment that can be used to demonstrate these algorithms. Design tools play an important role in the FPGA design cycle, but their long design cycle, small resource reuse, and inefficient compilation for engineering changes makes them inadequate for multimillion-gate FPGA designs. Based on the algorithms investigated in Chapter 3, it is highly desirable to develop a design environment that can be used to reduce the FPGA design cycle for a million-gate device.

As examined in Chapter 2, the Xilinx JBits toolkit presents a new way for bitstream generation by directly accessing, modifying, and generating a bitstream file in seconds. Unfortunately, a JBits-based FPGA design tool that can help the designer automatically place, route, and generate the bitstreams does not exist. This IDE is expected to process not only the small design modifications but also the entire design from scratch; it is also anticipated that it will speed up the FPGA design-and-debug cycle, and will exploit the potential function of the JBits toolkit.

The design tool presented in this chapter is expected to provide the following features:

- interpret a design application programmed using JBits APIs and Java language,
- employ the incremental placement algorithm and the guided placement methodology to represent the design,
- use JBits RTPCores as the design library,
- generate a placed-and-routed bistream as the output, and
- provide user-friendly integrated graphics user interface (GUI) to allow the FPGA designers to easily browser the placed designs and to switch between this tool and the debug tool provided by Xilinx.

Using this tool, FPGA designers can build their circuits using the Java language. JBits RTPCores, which serve as the design library, can be easily instantiated. Designers can also expand the design library by developing their own RTPCores, utilizing the object-oriented support in the Java language. After a design is compiled using a Java compiler, it can be loaded into the design tool and be presented by the guided incremental placement algorithms. Placed cores will be shown in the simulated device platform using different colors. A bitstream file will be created depending on the user's options. Designers then can verify their design by linking to the BoardScope software from this design tool environment. To understand the requirements of this integrated design environment, a use case model is developed in Figure 4.1.

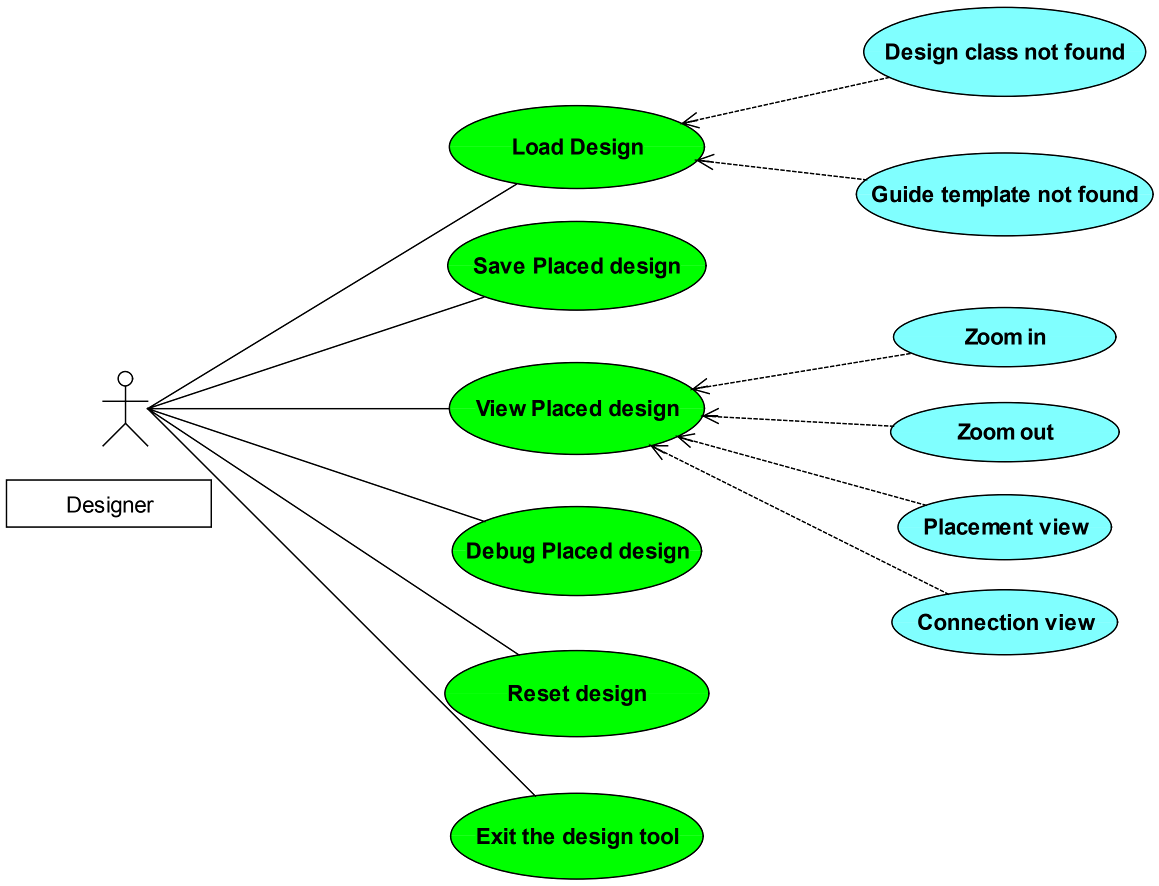

Figure 4.1 Use case model of the incremental FPGA IDE

4.2 Program Structures and Implementations

After analyzing the requirement of the incremental design IDE, its programming structures and implementations are described in this section. The design flow of this IDE is illustrated in Figure 4.2.

Compared with the JBits design flow shown in Figure 2.2, this design tool provides abstract classes beyond the JBits API. These classes register the user's design, find the differences between current and previous designs, and re-place only the changed portions by incorporating the incremental placement algorithm and guided placement methodology. These abstract classes are the basis of this design tool and provide functions to reduce the FPGA development cycle.

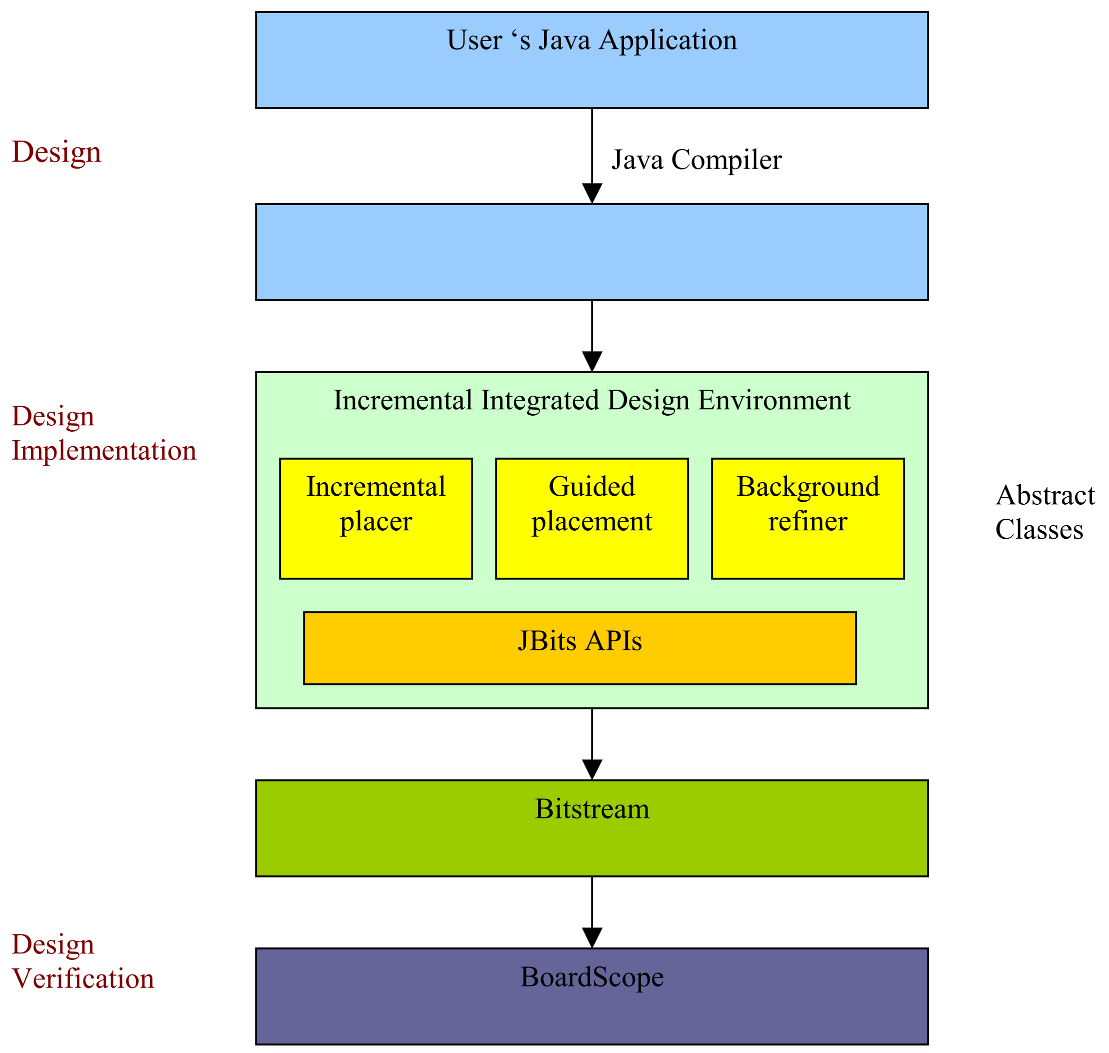

Figure 4.2 Design Flow of the JBits-based incremental FPGA design tool

According to the requirements of this design IDE, these abstract classes are implemented in three packages, namely `DesignIDE`, `DesignArea`, and `DesignTool`. Based on software engineering principles, a good system should provide low coupling and high cohesion [Sou99] between design modules. Coupling means the degree of interactions among modules, while cohesion defines the degree of interactions within each module. Modules in a low coupling system have less dependency on each other. When Module A invokes Module B, a system with the lowest coupling does not pass data between modules. If there has to be some data passed, it is desired that all the data passing from A to B is used by Module B. Passing control signals, such as control flags or function pointers, should be avoided. A system has high cohesion if all of its elements are essential for and directed toward performing one and only one task [Bro01].

To achieve low coupling and high cohesion, the function of each package is distributed as follows. The `DesignIDE` package presents a graphical user interface. This package interacts with users, obtains the user's input data, and invokes the corresponding functions in the other two packages. The `DesignArea` package provides all of the visualization functions including device platform simulation, placed cores display, zoom functions, and placement view and connection view properties. The `DesignTool` package is the key module of the design IDE that consists of classes that register a design, find changed portions, and represent the design using the incremental placement algorithm, the guided placement methodology, and the background refinement mechanism. Figure 4.3 shows the system components and interaction diagram of this design tool.

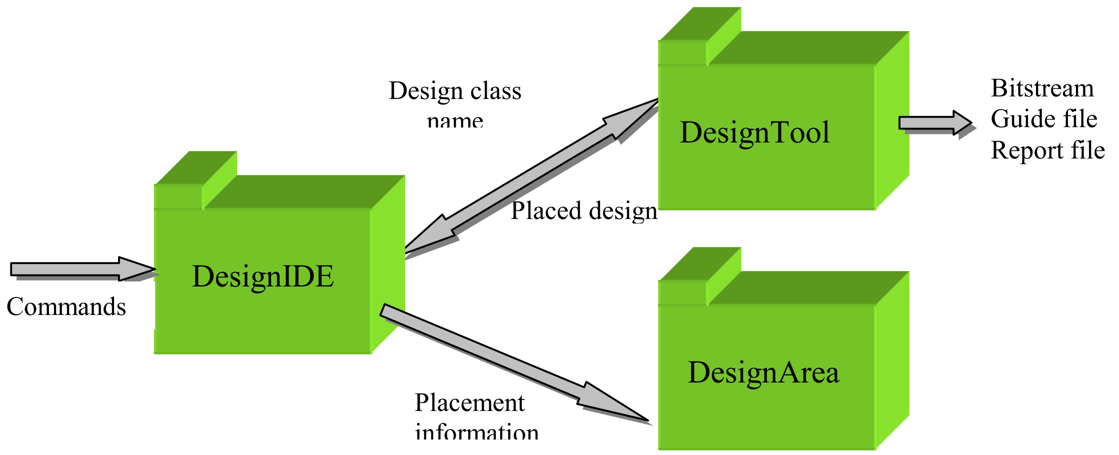

Figure 4.3 System components and interaction

If a designer loads an input design, the design class name and the guide template name (if the designer chooses the guided placement option) read in the `DesignIDE` package are passed to the `DesignTool` package. The Java byte code of the input design is interpreted in the `DesignTool` package and the design is placed using the incremental placement algorithm. Then the placement information is passed to the `DesignArea` package through `DesignIDE`, and the placed cores are displayed in the simulated device platform. A bitstream can be created and saved depending on the user's choice. Thus, the information flowing among packages is only data such as the name of the design class (String) or the placement information (Vector), and all the data passed to another module is used in this module. Low coupling has been achieved in this system. Packages in this design tool system also have high cohesion. Each package has a specific and independent task, and every function in a package is important to fulfill this task. The main functions of each package are described in the following sections.

- **DesignIDE**

The `DesignIDE` package is implemented using Java Swing techniques [Jaw98]. The main function of this package is to provide a user-tool interface. It interacts with a user, reacts the user's command and collects the input data. There are nine commands supported in the `DesignIDE` package namely *File Loading*, *System Reset*, *Design Saving*, *Design Debug*, *Zoom in*, *Zoom out*, *Placement View*, *Connection View,* and *System Exit*. Graphical buttons and system menus are designed for each command to make the user-tool interaction simple. When a user presses a function button, `DesignIDE` will call the corresponding method in `DesignTool` and/or `DesignArea` package to complete the requirement. By clicking the *Load* button, a user can load a design by either typing a Java class name or choosing a file from the `FileChooser` component [Sun99]. The *Save* function stores the placed design into a bitstream file named by the user. The *BoardScope* button dynamically loads the BoardScope software, which offers designers a debug environment to verify the functionality of the generated bitstream. The *Reset* button clears the current design by releasing all resources. The *Zoom* functions provide users with the flexibility to view the design from different scales. *Placement View* displays the placed design on the device simulation platform showing different type of cores in different colors. The *Connection View* allows designers to check the connections among cores.

- **DesignArea**

The `DesignArea` package simulates the device platform and displays the user's design. It is implemented using the Java 2D graphics API [Sun99]. Responding to the calls from the `DesignIDE` package, it shows the placed cores and distinguishes them by displaying different cores in different colors. When a specific core is clicked, it is highlighted and the name and the position of this core are extracted and displayed in the information field. The connectivity among cores is drawn in this package when the *Connection View* bar is clicked. The device platform is simulated using the `JTabbedPane` component provided by Sun [Sun99] to support switching between the placement and connection views. This package also supports zoom functions. When a user clicks the Zoom in/Zoom out button, `DesignIDE` calls the corresponding functions in `DesignArea` and allows the user either to watch the details of the design or to have general information about how the design is placed on the simulated chip.

- **DesignTool**

The `DesignTool` package is the main package in this tool. It implements the incremental placement algorithm and the guided placement methodologies that are running in the foreground thread to provide instant gratification. The implementation of the background refinement mechanism will be discussed in Chapter 5. Figure 4.4 shows the structure diagram of this package. There are four key classes defined in this package. `DesignLink`, `Placement` and `DesignFactory` consist of functions to dynamically link a design to the `DesignTool` package, register the design, find changed portions and represent the design using the incremental placement algorithm and guided placement methodology. `CoreInfo` class presents the data structure interface used to save and transfer the design information.

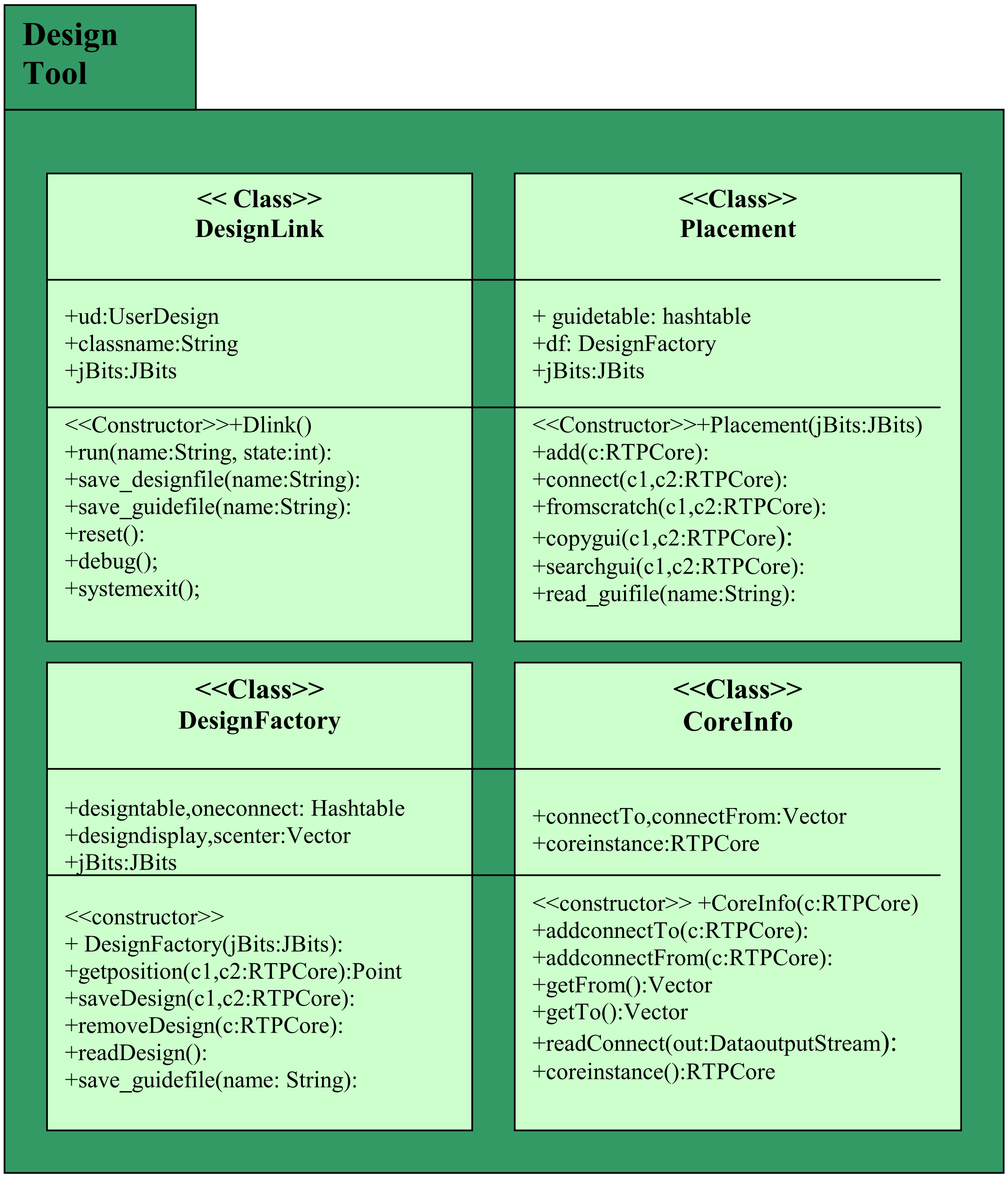
Design Tool
<< Class>>
DesignLink
+ud:UserDesign
+classname:String
+jBits:JBits
<<Constructor>>+Dlink()
+run(name:String, state:int):
+save_designfile(name:String):
+save_guidefile(name:String):
+reset():
+debug();
+systemexit();
<<Class>>
Placement
+ guidetable: hashtable
+df: DesignFactory
+jBits:JBits
<<Constructor>>+Placement(jBits:JBits)
+add(c:RTPCore):
+connect(c1,c2:RTPCore):
+fromscratch(c1,c2:RTPCore):
+copygui(c1,c2:RTPCore):
+searchgui(c1,c2:RTPCore):
+read_guifile(name:String):
<<Class>>
DesignFactory
+designtable,oneconnect: Hashtable
+designdisplay,scenter:Vector
+jBits:JBits
<<constructor>>
+ DesignFactory(jBits:JBits):
+getposition(c1,c2:RTPCore):Point
+saveDesign(c1,c2:RTPCore):
+removeDesign(c:RTPCore):
+readDesign():
+save_guidefile(name: String):
<<Class>>
CoreInfo
+connectTo,connectFrom:Vector
+coreinstance:RTPCore
<<constructor>> +CoreInfo(c:RTPCore)
+addconnectTo(c:RTPCore):
+addconnectFrom(c:RTPCore):
+getFrom():Vector
+getTo():Vector
+readConnect(out:DataoutputStream):
+coreinstance():RTPCore

Figure 4.4 Structure diagram of the DesignTool package

The primary scenario for this system is that a designer loads an application into the design tool and wants to get the design layout in a few seconds. Let's investigate the design sequence of this scenario. When a designer loads a new design, after reading the name of the input class and the guide template, `DesignIDE` class calls the `run()` method in the `DesignLink` class. This method interprets the Java byte code and triggers the `connect()` call in The Placement class. The `connect()` call handles a pair of cores at each run. It decides if guide option is chosen and which core is treated as the target core and which core is a new core. Then the `Placement` class triggers the `getpoistion()` method in the `DesignFactory` class. This function is the key implementations of the incremental placement algorithm and it is this function that finds a position for each newly added core. The found position is returned to the `Placement` class and the design is saved in the database if a valid position is found. After all the components have been added and placed, an `end()` method in the `Placement` class reads the placement information and sends it back to the `DesignIDE` class. Then, the `DesignIDE` class triggers the `show()` method in the `DesignArea` class and the placed design is finally displayed in the simulated device platform. Figure 4.5 presents the sequence diagram of this scenario.

According to the use case diagram shown in Figure 4.1, there are several scenarios in this design IDE system including reset, save, debug and view functions. The sequence diagram for each scenario is drawn during an object-oriented analysis to make sure the designed system can achieve all of the user requirements.

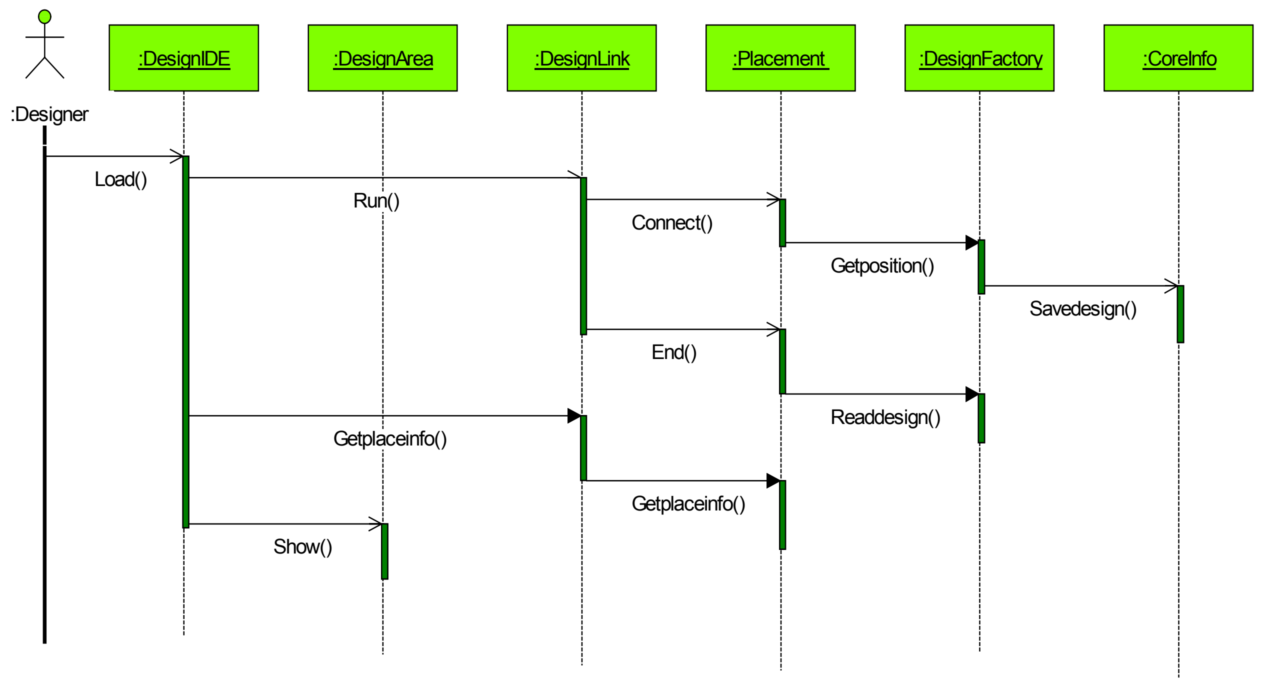

Figure 4.5 Sequence diagram of the load class scenario

Generally in software programming, when Class A wants to trigger a method in Class B, a typical way to fulfill this function is:

```
Class A {
   A()
   {   ….
    B.method();
  }
```

In the `DesignTool` package, even if the `connect` function in the `Placement` class is triggered in the `DesignLink` class, we can not find the `Placement.Connect()` calls in its source code. Instead of making this call in the system support software, it is called by designers in their application files to make sure that a pair of cores is added in the design. Then how can the `DesignLink` class interpret this function and trigger the calls in the `Placement` class? This functionality is implemented using the dynamic linking techniques discussed in the next section.

4.3 Dynamic Linking Techniques

The purpose of the work presented here is to provide a design tool that can be used to process not a specific but a large number of applications. To process those applications, the design tool should have the ability to understand the content of each design before triggering a corresponding algorithm to handle it.

A general way to solve this kind of problem is to write a complier; however, this is not an easy task and it would distract from the central theme of this dissertation. A simple answer is to use the dynamic linking techniques provided by the Java Virtual Machine (JVM). As indicated in its name, when a method or a variable is defined, changed and loaded into the JVM, the dynamic linking technique can bind the changes with the method code at the run time. As explained in [Lin96], a Java Virtual Machine frame records the reference to the constant pool for the type of the current method to support the dynamic linking function. The `class` file code for a method in Java refers to the methods to be invoked and variables to be accessed via symbolic references. To dynamically link methods, the JVM translates method symbolic references into their real reference and then loads related classes to resolve the undefined symbols. For variables, the JVM translates their accesses into offsets in the storage structures associated with their runtime locations. Methods and variables are then bound to the system at runtime to make the system understand the content of a loaded class.

To interpret and dynamically link a design to the placement algorithm, a `UserDesign` class has been developed, and a virtual Java method `design()` has been defined in this class. As illustrated in Figure 4.6, the `design` method in this class is the bridge that links a user's design with the `DesignLink` class.

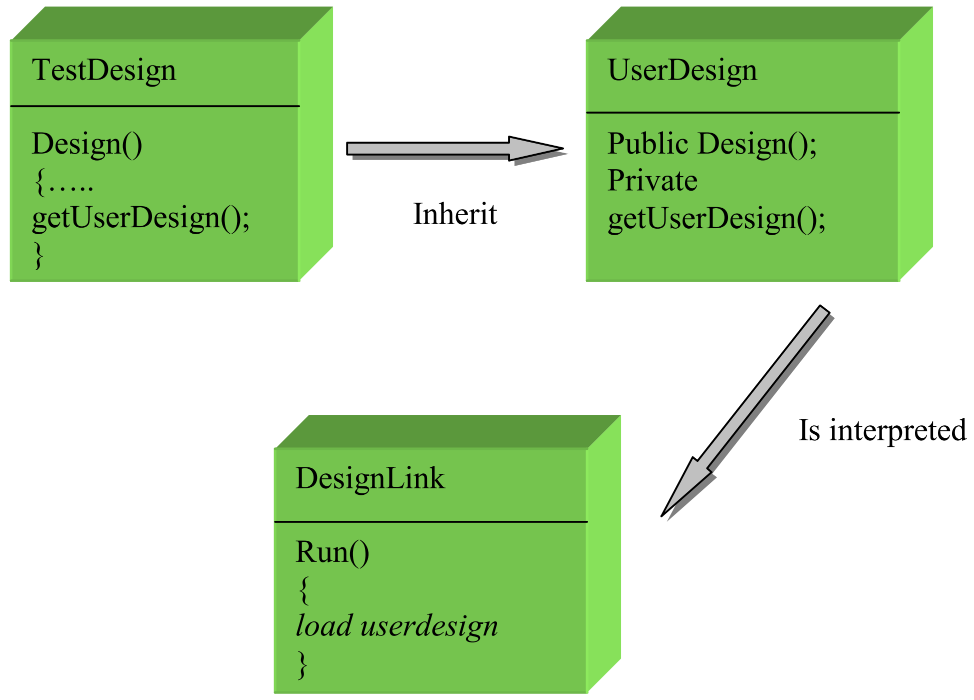

Figure 4.6 Dynamically linking a user's design

On the designer's side, a designer does not need to know much about how the dynamic linking technique works. The only thing he/she has to do is inherit the design application file from the `UserDesign` class and model the design into a method called `design()` to use the incremental placement algorithm. A typical user design class can be found in the following example:

```
public class Tester extends UserDesign
{
public void design ()
        {
        }
}
```

`UserDesign` class contains a `getUserDesign()` method that reads a JBits instance as the input parameter and returns an instance of the `Placement` class. A JBits instance can be easily obtained by instantiating a JBits constructor using the device type as the parameter. To use the incremental placement algorithms to place the design, designers can make the `getUserDesign()` call to obtain the instance of the Placement class in their `design()` method and place their cores using a simple `connect()` call. A typical user `design` method is:

```
public void design ()
{
  …
  Placement p;
  try {
          p= getUserDesign(jBits);
          Counter counter = new Counter("Counter",cp);
          Register reg = new Register("Register1",clk,cout,rout);
          p. connect(reg,counter);
         ….
      }
catch (Exception e){}
```

where `cp` is the property of the counter core, and `Clk`, `cout`, and `rout` are the connection buses [Xil00]. In this example, a Register core and a Counter core are instantiated and automatically placed in the device by making a simple `connect()` call. The entire design can be modeled using the common Java language and can be compiled using the standard Java compiler. The generated Java byte code is then ready to be loaded into the incremental FPGA design tool.

On the software developer's side, a class loader is necessary to get a `Class` object of the newly loaded class. A new instance of the class represented by this `Class` object is then created by using the `newInstance()` function from the `java.lang.Object` class; the class is initialized at the same time. In the above example, after getting the `Class` object of the `Tester` class, it is initialized at run time when the `newInstance()` call is made. Because the Tester class extends the `UserDesign` class, the specific class instance is obtained by converting the type of the instance from a general Java `Object` to `UserDesign` and the user's design is then linked to

`DesignTool` package to implement all functions defined in the user's own `design()` method by calling the `UserDesign.design()` method. The following example shows how to get the instance of the Tester class and how to implement a user's design:

```
try {
            Class t = Class.forName(classname);
            Object o=t.newInstance();
            ud =(UserDesign) o;
            ud.design(jBits);

      }catch (java.lang.ClassnotFoundException e){
           e. printStackTrace();}
```

`forName()` is a static method contained in the `java.lang.Class` [Sun99]. This method takes a string parameter containing the fully qualified class name and returns the instance of a `java.lang.Class`. Making a `forName(classname)`call directly loads a class using the standard class loader defined in the Java Development Kit.

There is a problem when using the standard class loader. After a designer places a design using the incremental design tool and makes changes, and the modified design is reloaded, the standard class loader does not automatically reload the new class but always presents the original design instead. This drawback inconveniences the FPGA designer because it requires restarting the design tool software whenever there is a small change in the design.

To solve this problem, the standard class loader inside the Java Virtual Machine is first investigated. This functions as follows:

1. Check the already loaded class inside JVM by calling `findLoadedClass()` method;
2. If the class not found, then check inside the system/boot start class by calling `findSystemClass()` method;

3. If not found, then load the *.class* file from the local disk using the defined classpath environment variable;
4. If the class not found, then the JVM throws the `ClassNotFoundException;`
5. If found, then resolve the class by loading the parent class and the interfaces.

Following this procedure, when a modified class is reloaded into the design tool, the Java virtual machine will always find the originally loaded class after calling the `findLoadedClass()` method, and even a forced garbage collection call cannot clear the class from the JVM at the run time. Thus, a custom class loader should be developed to force the design tool to load a modified class directly from the hard disk.

The custom class loader extends the `ClassLoader` class from `Java.lang` class and overrides the `loadClass()` method in the `ClassLoader` class. Instead of calling the `findLoadedClass()` method at the very beginning, the custom class loader bypasses the first two steps in the standard class loading procedure to load a modified class file. Because the loaded class, its parent class and the interface of the loaded class are automatically and recursively loaded in a class loader, the custom class loader checks the name of each loaded class before loading. If this name matches the name of the modified class, the custom class loader reads it directly from the hard disk; otherwise it follows the standard loading procedure. A typical custom loader developed in this dissertation can be found in the following example:

```
class myCustomLoader extends ClassLoader {
   public Class loadClass (String name) {
     if (classname.equals( modified class ))
     {read the class from hard disk;
      classobject = defineClass(name, bytes, 0,length);
      }
     else  follow standard procedure;
     resolveClass(classobject);
     return classobject;
                                             }
                                              }
```

4.4 Data Structure

After the user's design class has been dynamically linked to the `DesignLink` class, the entire design is registered and placed in the `DesignTool` package. This section describes the data structures used to save the design information.

A pair of cores is added into the incremental design tool as the design is processed incrementally. At the moment a core is added, its instance is passed to the `DesignFactory` class. The information about a core including its name, height, width, and its position can be obtained from this instance using the Xilinx JBits API. During the placement, the core instance, its connectivity group, and other related information containing the cluster and index number of each core is recorded in a design database. When all of the cores are placed, the information related to the display function consisting of the position, core name, core tag, its index and its cluster number is saved in a vector and returned back to the `DesignIDE` package. At the same time, a bitstream file, a guide template, and a report file are also generated. Figure 4.7 shows the information flow during the design.

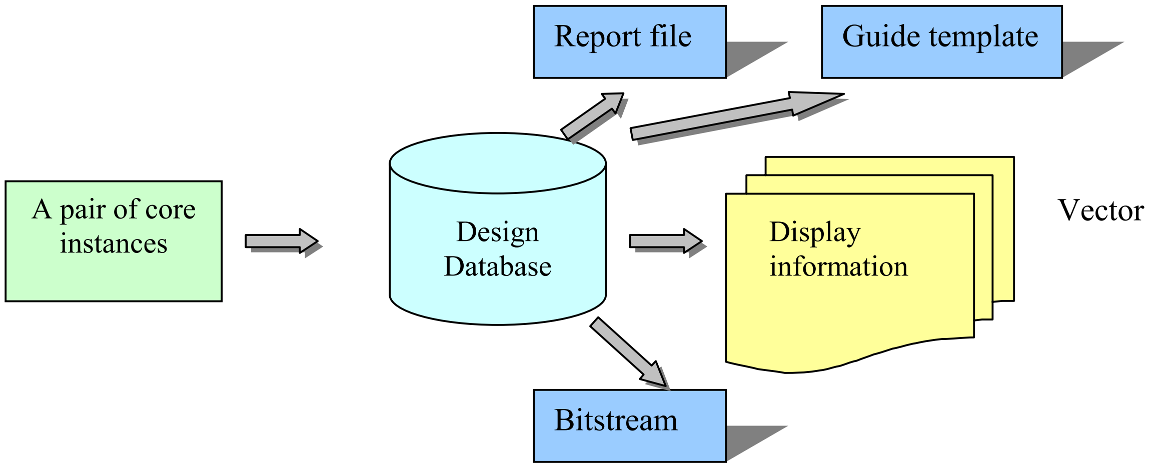

Figure 4.7 Information flow during the design

The design database is implemented using the Hashtable object provided by Java Development Kit. The hash table is a data structure in which keys are mapped to array position by a hash function [Car98]. It is a direct address table that requires a unique key for each stored element. Compared with the binary search tree and its balanced variants, this data structure provides a rapid table search particularly if the number of elements is large. This property makes the hash table structure particularly useful in the design tool development for million-gate FPGAs. Because speed is a fundamental issue in the incremental placement algorithm, and the design database is searched frequently to extract the information of a core such as its position and its connectivity group, a fast access database can help the placement algorithm reduce the processing time. As gate sizes grow, and more and more components are added in a design, the role of the hash table structure becomes more essential.

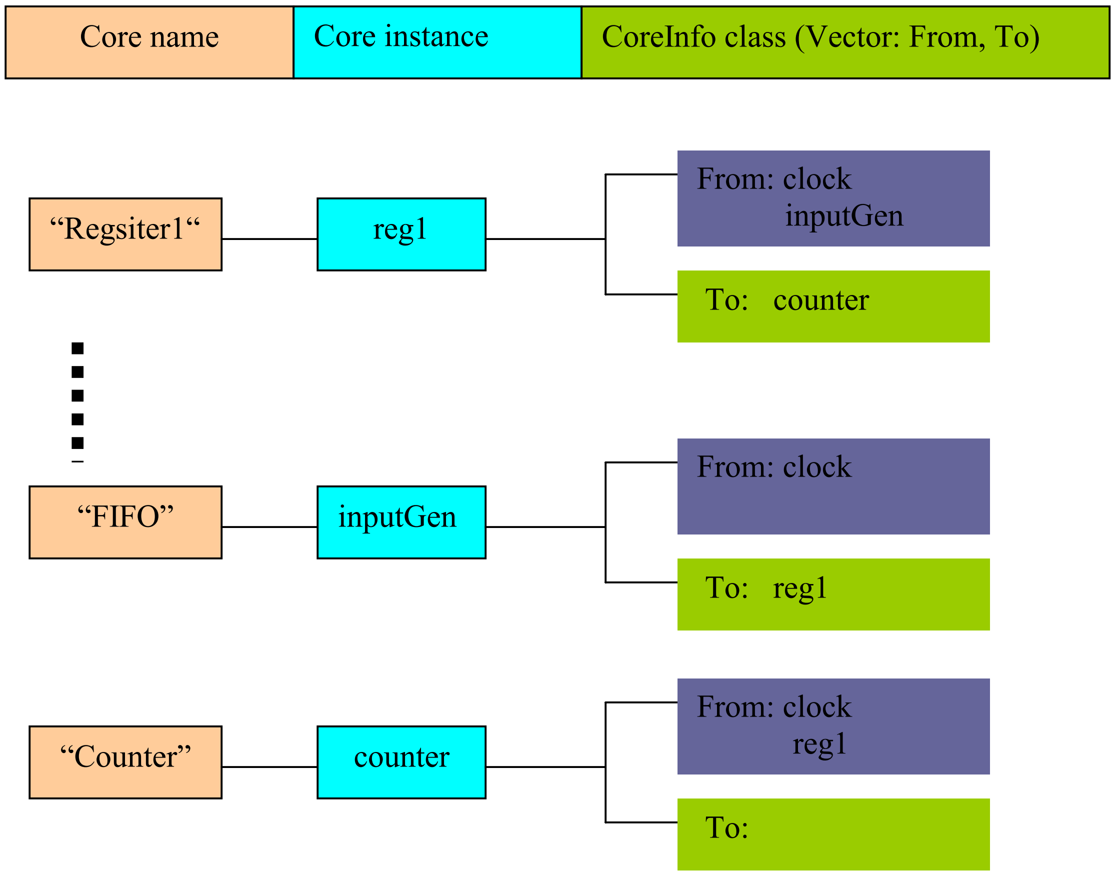

Figure 4.8 Data structure of design registration

The key for the hash table is the name of a core. The contents stored are the core instance, core cluster number, and its connection information. Figure 4.8 illustrates the data structure of the design database. A `CoreInfo` structure is developed to record the connection information for each core. Depending on the direction of connection, the connected cores are divided into two groups: if a core's output is connected to a processed core's input, it is stored in this core's `FROM` group; otherwise it is saved in the `TO` group. Classifying the connectivity group helps the placement algorithm find a target core and calculate the desired position of a newly added component efficiently.

4.5 A Simple Design Example

After describing the implementations and data structures of the incremental design IDE, this section develops a simple JBits-based FPGA design to demonstrate how to use the incremental placement algorithm and the design tool prototype from the designer's point of view.

A simple template matcher is implemented in this example. A few numbers saved in a text file are read from a FIFO and compared with a constant: if they match, a counter is increased by one; otherwise it holds the previous value. The latest value of the counter will be stored in a register. Figure 4.9 shows the diagram of this design.

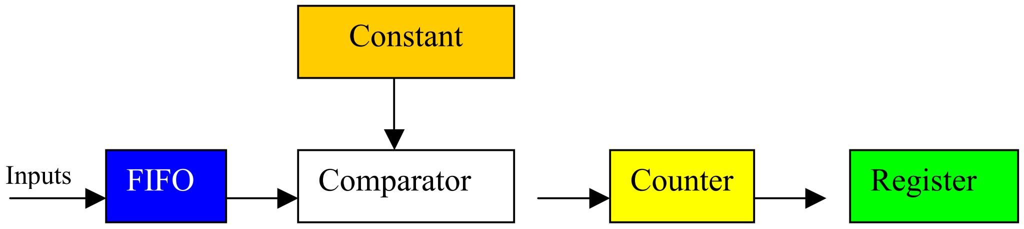

Figure 4.9 Block diagram of a simple design

To implement this circuit, a user needs to build the design using the Java language by instantiating corresponding cores from the Xilinx RTPCore library. After defining the connected bus and initializing these cores with their resultant properties, placement can be flexibly achieved by calling the `connect()` function in the design. The following code example shows how to place these cores by calling the incremental placement algorithm in a user's design.

```
public class Tester extends UserDesign {
public void design ()
{
  Placement p;
  try {
       JBits jBits= new JBits(Devices.XCV1000);
       p= getUserDesign(jBits);
       // defining bus and instantiating the cores
       Bus fifoout = new Bus("fifoout", null, regwidth);
       InputVector fifo = new InputVector("fifo",16,clk,fifoout);
           ……
       //place the cores
       p.add(clock);
       p.connect(fifo,comparator);
       p.connect(constant,comp);
       p.connect(comparator,counter);
       p.connect(counter,register);
       p.end();
       //implementing and routing the cores
        clock.implement(); //implementing
        fifo.implement("fifoinput.txt");
              ……
        BitStream.connect(clkout); //routing
        BitStream.connect(fifout);
              ……
  }
}
```

As discussed in Section 4.3, this design application extends the `UserDesign` class. The entire design is built by overriding the `design()` method. The JBits instance is initialized via reading the device type as the parameter. A `getUserDesign()` call connects the user's design with the incremental design tool. The `Connect()` call takes instances of two connected cores as input parameters, and places these cores by linking them to the incremental placement algorithms. The `end()` function call is made when all of the cores are added into the design. Placed cores can then be implemented and routed using JBits calls `core.implement()` and `Bitstream.connect(corebus)`. In this example, the Clock core is not shown in the diagram, however, it is always needed in a design.

After compiling this design using the standard Java compiler, the successfully generated byte code is ready to be loaded into the design tool. The job remaining for an FPGA designer is to press a few buttons, then the design will be rapidly placed-and-routed and be graphically presented in a simulated device platform.

4.6 Summary

This chapter focused on the software implementation issues of the incremental FPGA integrated design environment. The program structure and the implementation of the design tool have been described after analyzing the system requirements. Dynamic linking techniques have been developed in this chapter to allow designers to build their design using the Java Language and to compile the design using the standard Java compiler. The data structures employed in the tool development have been discussed as well. Finally, a simple design example has been presented to demonstrate the use of the incremental design IDE.