Developing smarter Ada applications using a model-driven approach with UML 2.0 and Ada 2005 interfaces

Fraser Chadburn
March 18, 2010

The Unified Modeling Language (UML) 1.x offered developers the ability to visualize how classes are coupled. A key enhancement to UML 2.x is to improve support for modeling large-scale systems, including the ability to hierarchically decompose systems into parts with clear ports and interfaces. However, since a UML 2.x port can be contracted to provide any number of interfaces in UML 2.x, those developers using Ada 95 have a number of implementation challenges when mapping UML interfaces to code. This article illustrates how these challenges can be resolved utilizing Ada 2005 interfaces. The article will also show how the power of UML 2.x ports and interfaces now supported by the IBM Rational Rhapsody solution and highlight the benefits of using this environment, including the clear separation of components and how components can be re-used in multiple assemblies, leveraging the Rhapsody tool to automatically generate the wiring code.

Introduction

The initial versions of the Object Management Group’s (OMG) Unified Modeling Language 1.x (UML 1.x) originated from three leading object-oriented methods (Booch, OMT, and OOSE), and incorporated a number of best practices from modeling language design, object-oriented programming and architectural description languages. The UML 1.x language is powerful for modeling how classes relate in terms of object-oriented relations such as associations, dependencies, compositions, aggregations and generalizations. Figure 1. shows how classes relate using a Class Diagram.
When building large systems, however, we want to focus not only on how classes interrelate in terms of cohesion, but also how they are separated. Although it was possible in UML 1.x to define interfaces, there was no intrinsic modeling support for structured notation showing how systems are decomposed into subsystems or how subsystems interrelate in such assemblies.

**Figure 1. Class Diagram in UML 1.0**

![Class Diagram in UML 1.0](image)

**Modeling large-scale systems using UML 2.x**

The main enhancement made when developing the latest Unified Modeling Language standard, UML 2.x, was to provide support for modeling large-scale systems. Specifically, UML 2.x seeks to underpin the notion of systems-of-systems, where any system can be composed of subsystems which can by further decomposed into subsystems of their own. To support this UML 2 introduced a new concept called composite structures. In this context a "structure" is a composition of interconnected elements representing run-time instances collaborating over communications links to achieve some common objectives (see resources). A structured class, therefore, is one which contains other classes.

The instances of classes inside a class are known as parts. In Figure 2, for example, Builder1 is a structured class composed of two parts, an instance of a Transmitter class and an instance of a Receiver class. Classes can be composed of classes which themselves have parts, allowing you to hierarchically decompose a system to any arbitrary level.

**Figure 2. Provided and required interfaces with a behavior port**

![Provided and required interfaces with a behavior port](image)
The concept of ports was also introduced in UML 2.x, representing an interaction point between a classifier (e.g. class or actor) instance and its environment or between a classifier instance and instances it may contain, i.e. its parts. The interfaces associated with a port specify the nature of the interactions that may occur over a port.

The example shows that the Transmitter and Receiver are connected via ports. The Receiver has as a behavior port that handles the calls. On a behavior port the concrete operations provided by the interface are implemented by the instance of the class that owns the port. Figure 3. shows the Transmitter invoking asynchronous messages on the Receiver via the connector between the ports. In the case of the Receiver class it simply prints the character it receives to the console window.

**Figure 3. Screenshot from running Builder1 as an application**

![Screenshot from running Builder1 as an application](image)

**Provided and required interfaces using ports**

In UML a classifier (e.g., class or actor) can have any number of ports. Each port is named uniquely with respect to its owning classifier. The operations provided or required via a port can be captured using the lollipop and socket notation.

- "Required interfaces" characterize the requests that may be made from the classifier to its environment through the port. This is denoted as a socket (also known as a cup).
- "Provided interfaces" characterize requests to the classifier that its environment may make through the port. This is denoted as lollipop (also known as a ball).

In the case of a required interface the class can be designed and implemented by invoking the service on its port, rather than directly with the external provider. As such ports conceptually decouple a class from the provider.

Importantly, a different concrete implementation can be used providing that it realizes the same interfaces. This makes ports powerful for component-based design. Using ports a component can therefore be easily moved from one assembly to another. This means that components can be reused in different contexts such as for testing, or in a completely different system altogether.

In Figure 4., for example, the Transmitter is completely unchanged but the Receiver class is replaced by a Banner class that provides the same interface but a completely different implementation of the IReceiver interface.
This figure also illustrates the concept of a non-behavior port. The Banner class does not provide the concrete realization of the IReceiver interface; instead it relays invocations on its port to an internal part called "itsScreen" of class Screen. Non-behavior ports are informally known as either relay or delegation ports as they forward the calls to internal parts.

The figure also illustrates how easy it is using the notation to "plug and play" different components together that have the same interfaces. The output from running Builder2 is shown in Figure 5. It illustrates how the "itsBanner" part displays the characters it received as a series of pixel-based letters in a scrolling banner, rather than simply printing them out. Importantly, it was possible to change the implementation of the class providing the IReceiver interface without impacting the Transmitter class using its services, and illustrates how easy it is to "plug and play" components when using ports.

Figure 5. Screenshot from running Builder2 as an application

A given port can be contracted to provide any number of provided or required interfaces and a classifier may have any number of named ports. The challenge for leading UML tools, such as the IBM Rational® Rhapsody® solution, is how to synthesize the constructs in UML into a target language implementation.

Expressing an object-oriented design in Ada

Before exploring the mapping to ports one must understanding the relationship between Ada and UML. When you are developing software in this unique environment using these languages, how
the key concepts of classes and interfaces in an object-oriented design map to Ada language constructs is of particular concern.

Classes are the building blocks of an object-oriented design and define a description of a set of objects that share the same attributes, operations, relationships and semantics (see Figure 2.). The concept of a class is not native to the Ada language. It is usually synthesized using an Ada record to define the attributes of the class and an Ada package to define the operations that can be invoked on objects of that type by passing a handle to the record as the first parameter in the operation call.

The concept of interfaces in UML is also important. A UML interface is defined as a collection of operations that are used to specify a service of a class or component. An interface will usually consist of more than one service signature. By declaring an interface in UML, you state the desired behavior in an abstract signature independent of the implementation. Interfaces are an important aspect of scaling up to larger systems as you can use interfaces to define the public view of operations "provided" or "required" by a subsystem.

Using the Ada 95 language, it is possible to synthesize interfaces by use of an Ada package and an abstract null record type where the Ada package is used to group subprograms that can operate on objects of a particular type. The problem with Ada 95 is that it only permits a derived type to have one immediate ancestor. Since the record type is used as the basis for operation invocation on an object, this means that the number of "provided" or "required" interfaces on a given port for a class (or composite class) is limited to one, meaning that multiple interfaces for contracts on a port are not allowed.

This leaves the option to either constrain modeling to only a restricted subset of the UML, or to leverage new features found in Ada 2005, namely the introduction of Java-like interfaces that support multiple inheritance.

**Exploiting Ada 2005 interfaces**

In Ada 2005 a new reserved word `interface` was introduced to resolve the limitations of single inheritance. All interface types in Ada 2005 are implicitly tagged types, i.e. they support run-time dispatch (see Figure 3. Error: Reference source not found). Unlike an abstract type, an interface cannot declare components or data; it is used instead as the basis of defining the signatures of subprograms that can be invoked. As such interfaces in Ada 2005 are only permitted to have abstract subprograms and null procedures as operations.

Figure 6., below, shows two interfaces. If an operation is declared as `abstract` then the inheriting class must provide a concrete implementation. If an operation is declared with a `null` keyword, then the inheriting class can choose to override the null procedure.
Figure 6. Abstract subprograms and null procedures in Ada 2005

```ada
package ILetter is
  type ILetter_t is interface;
  procedure Refresh (this : in out ILetter_t) is abstract;
  procedure Setup (this : in out ILetter_t; The_Char : in Character) is abstract;
end ILetter;
package IPrint is
  type IPrint_t is interface;
  procedure Print(This: in out IPrint_t, The_String : in String) is null;
end IPrint;
```

In Ada 95 an abstract type could only have one immediate ancestor, but in Ada 2005 it is possible to derive a new type from multiple interfaces or tagged types. One can refer to the first type as its parent and the others as progenitors.

Figure 7. Multiple inheritance using interfaces

```ada
with ILetter;
with IPrint;
package Banner_Letter is
  type Receiver_t is new ILetter.ILetter_t and IPrint.IPrint_t with private;
  procedure Refresh (this : in out Receiver_t);
  procedure Setup (this : in out Receiver_t; The_Char : in Character);
  procedure Print(This: in out Receiver_t, The_String : in String);
private
  (hidden attributes)
end Banner_Letter;
```

The ability to define an interface means that the caller is abstracted away from the implementation of the operations. In Figure 8, the class diagram shows that Screen has list of zero or more ILetter objects. In this example, the Screen class invokes operations via the ILetter interface, iterating through a collection of ILetters objects calling Refresh. The call on the interface results in the run-time dispatch to the concrete implementations provided in the Banner_Letter class. Using this abstraction a different implementation of ILetter could be used by Screen, provided that it realizes the signatures of the ILetter interface operations.
Figure 8. Interface usage and realization

The code for ports works in a similar way. In the case of a behavior port, calls on provided operations are handled by the class which owns the port. This means that the owning class "realizes" the provided interfaces on the port. When a client invokes an abstract operation on a behavior port, therefore, the call is dispatched to the class.

Automatic synthesis of wiring code

On a non-behavior port, when the client invokes an abstract operation, it is necessary to forward the request to a class that realizes the provided interfaces.

The key thing is that this port wiring code is regular and can be automatically generated, based on the connections defined on the composite structure diagram. This means that the user does not need to write any additional code, they can use the graphical notation to place a part in a given context, and a code generator can synthesize the wiring code based on defined set of rules. Using the Rational Rhapsody environment, for example, you can define that you want to build a Builder2 composite class and it will assemble and instantiate all the necessary parts for you in an auto-generated main, together with the necessary wiring code for forwarding messages between the parts.

This simplifies the integration process enormously and enables developers to easily "plug and play" components using the graphical UML 2.x notation to define how they are connected.

Summary

This article shows you how Ada 2005 can help widen the ability of Ada development teams to exploit advanced UML 2.x techniques previously enjoyed by developers using implementation languages like C++ and Java. A key UML 2 concept supported by the synergy between Ada 2005 interfaces and UML interfaces is the notion of ports. These ports support the definition of composite structures with clear but abstract interfaces that define the services "provided" to, or "required" by your customers.

Defining interfaces and decomposing a system are an important aspect of scaling up to larger systems. Using UML 2.x you can not only decompose a large system into parts but the clear
separation and abstraction of subsystem boundaries enables geographically dispersed teams to work independently in parallel on different system components. It also supports the ability to leverage libraries of re-usable components to reduce your overall development time or effort. Once you've created a class with ports and interfaces, it follows that you can re-use them in multiple contexts.

With support from the modeling tool you can automatically generate the glue code between the components and all the necessary build files to assemble the component. This simplifies bringing separately engineered parts together during integration. Of course, the model also contains many things not synthesized in code that provide a wider context; such as use cases, traceability to requirements, and other key product artifacts. This benefits the developer because you can concentrate on designing the application rather than the integration code that sits around it.