A Graphical Modeling Notation with Similarities to UML

1. Introduction

The notation presented here is a general-purpose graphical format for modeling software artifacts. The notation is general purpose in that it can be used to model both specification- and design-level artifacts. For our purposes, we use two formal textual languages for representing these artifacts. RSL is used to represent requirements and specifications; Java is used to represent design and implementation. The same general-purpose graphical notation maps to both of these languages.

The notation presented here uses concepts from the Unified Modeling Language (UML). The design of UML is a collaborative effort of primarily commercial organizations, led by the Rationale Corporation. The goal for UML is to develop a standard graphical modeling notation. UML is a consolidation of concepts used in the earlier Object Modeling Technique (OMT) and Booch Diagrams.

UML has a number of features that are not used in the notation described here. In addition, UML is missing certain features, notably function and dataflow diagrams. Where a feature is available in UML, the same feature is retained here. Where UML is missing what is considered to be an important notational feature, a previously existing standard is used, in such a way as not to conflict with any existing UML features. That is, if UML has it we'll use it; if UML does not have it we'll use a standard notation that does not conflict with UML.

A complete specification of UML is available at www.omg.org/technology/documents [1]. There are also a number of UML books on the market, none of which is necessary for this class.

2. Class Diagrams

A class diagram represents the composition and inheritance relationships between objects in a specification or classes in a design. The notation described here is largely a subset of UML, with a minor extension added. Elements of class diagrams are the following:

1. Three-part boxes, labeled at the top with a object/class name.
2. Component/data member names, immediately below the class name in a box.
3. Declared operations/function member names, following the data member names in a box.
4. One-part boxes, labeled inside with the object/class name only.
5. Connecting edges between class boxes, with three forms of augmentation:
   a. a hollow triangle, designating an inheritance relation
   b. a hollow diamond, designating a composition relation (same semantics as the second part of a three-part box)
   c. integer or comma-separated integer pair annotation on hollow diamond, indicated multiplicity of composition; the character '*' can be used in place of an integer to represent multiplicity of 0 or more

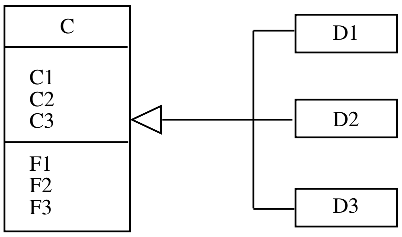
---

[1] The OMG web page changes frequently. If the link given here is not found, go to the main site at www.omg.org and search for "UML Spec" with their search engine.

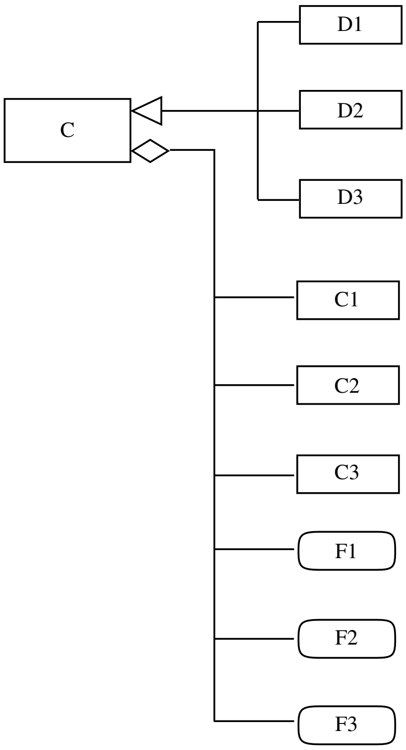
Figure 1 shows the general structure. The diagram on the top of the figure shows data and function membership using the three-part box notation. The diagram on the bottom of the figure, labeled "Alternative Equivalent...", shows the equivalent membership relations using the hollow diamond notation. Note that function members, when shown outside of a three part class box, are drawn as rounded rectangles. This is consistent with their depiction in function diagrams, as discussed below.

Orientation of components in a class diagram is *not significant*. That is, subclasses and members can be shown in any geometric position relative to their parent class. The orientation of the hollow triangle is significant, with the pointed end oriented *towards* the parent class. Also, the positioning of the hollow

Sample Class Diagram:



Alternative Equivalent Sample Class Diagram:



Corresponding RSL:

```
object C is
    components: C1, C2, C3;
    operations: F1, F2, F3;
end C;

object D1 inherits from C is ...;
object D2 inherits from C is ...;
object D3 inherits from C is ...;
```

Corresponding Java Code:

```
class C1 {...}
class C2 {...}
class C3 {...}

class C {
  public void F1(...) {...}
  public void F2(...) {...}
  public void F3(...) {...}
  protected C1 c1;
  protected C2 c2;
  protected C3 c3;
};

class D1 extends C {...}
class D2 extends C {...}
class D2 extends C {...}
```

Corresponding C++ Code:

```
class C1 {...};
class C2 {...};
class C3 {...};

class C {
  public:
    void F1(...);
    void F2(...);
    void F3(...);
  protected:
    C1* c1;
    C2* c2;
    C3* c3;
};

class D1 : public C {...};
class D2 : public C {...};
class D2 : public C {...};
```

Figure 1: General Structure of a Class Diagram.

diamond is significant, in that it is positioned *immediately adjacent* to the contain*ing* class. In this way, two-way containment[2] can be depicted, as shown in Figure 2. Figure 2 also illustrates the use of multiplicity annotations.

Note that the three-part representation of membership versus the hollow diamond representation are two equivalent views that have exactly the same meaning. Membership can be shown in either way separately, or with the two forms combined into a single diagram. When the two forms are used in single diagram, there is redundant information shown -- i.e., a single diagram is showing the same membership relationships in two different ways. The notation does not prohibit such redundancy; it is up to the diagram designer to use the notation as she/he sees fit.

3. Function Diagrams

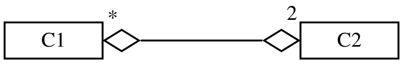
Function diagrams show the calling relationships between functions in a program design. Function diagrams are not applicable at the RSL specification level. Elements of a function diagram are the following:

1. Rounded rectangle nodes, representing functions
2. Edges between nodes, representing function calls
3. Annotated arrows above function nodes, representing input to and output from functions
4. Annotated down-pointing and up-pointing arrows, representing exception handling catch and throw, respectively
5. Labeled doubled lines, representing event invocation
6. Labeled dashed-line boxes around function nodes, representing class membership.

Figure 3 shows the general structure.

The calling relationship shown in a function diagram shows *potential invocation* not actual invocation. Depending on the implemented logic within a calling function, none, some, or all of its potentially called functions may actually be called during program execution.

Sample class diagram:



Corresponding RSL:

```
object C1 is C2 and C2;
object C2 is C1*;
```

Corresponding Java:

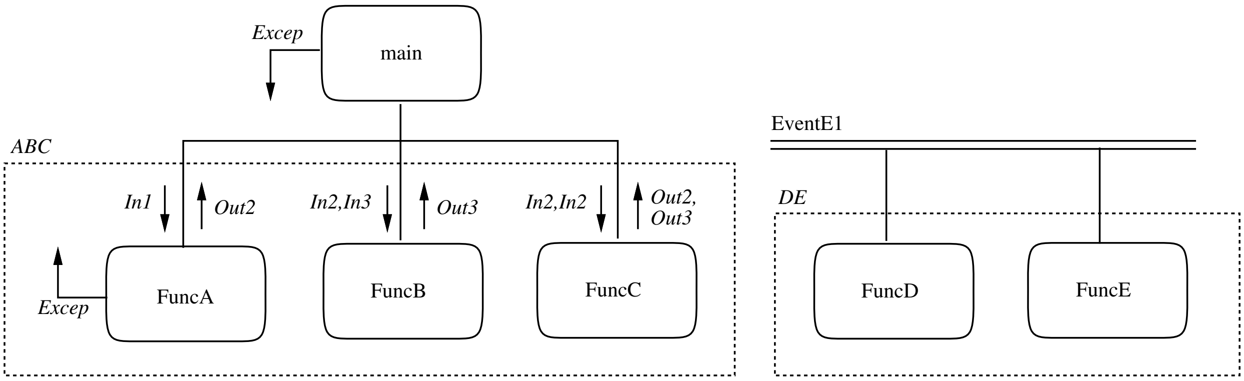
```
class C1 {
    C2 c2a;
    C2 c2b;
};

class C2 {
    C1List C1;
};
```

Figure 2: Two-Way Membership in a Class Diagram.

---

[2] The depiction of two-way containment is prohibited in standard UML, but allowed here since it maps to well-defined semantics in both RSL and Java. In RSL two-way containment maps to mutually-recursive objects, i.e., objects that have each other as components. In Java, two-way containment maps to classes that have references to each other, i.e., classes that declare one another as data members.

Sample Function Diagram:



Corresponding Java Code:

```
class Excep extends Exception {...};

class ABC {
    public Out2 FuncA(In1) {
        ...
        throw (new Excep);
        ...
    }
    public Out3 FuncB(In2, In3) {...}
    public Out2 FuncC(In2, In2, Out3) {...}
};

class DE {
    public void FuncD() {...}
    public void FuncE() {...}
};
```

```
public static void main() {
    static ABC abc = new ABC;
    static DE de = new DE;
    static Out1 out1;
    static Out2 out2a, out2b;
    static Out3 out3;
    ...

    try {
        out2a = abc.FuncA(in1);
    }
    catch (Excep e) {
        ...
    }
    ...
    out3 = abc.funcB(in2, in3);
    ...
    out2b = abc.funcC(in2, In2);
    ...
    bindFunctionToEvent(DE.FuncD, EventE1);
    bindFunctionToEvent(DE.FuncE, EVentE1);
    ...
}
```

Figure 3: General Structure of a Function Diagram.

Orientation of components in a function diagram is *is significant*. The root of a calling tree must be shown above or to the left of the subfunctions that it calls. Similarly, an event invocation line must be shown above or to the left of the functions that may be invoked when the event is triggered.

Strictly speaking, the left-to-right (or top-to-bottom) order of called functions is not significant. However, by convention the left-to-right order in a function diagram will typically be the same as the lexical order of appearance of the functions in the Java code.

4. Package Diagrams

A package diagram is used to show grouping among related classes. A package is a modular software component containing two or more classes. Examples of packages include library, separately launchable executable programs, and other logically related collections of classes.

Elements of a package diagram are the following:

1. Folder-shaped rectangles, representing packages
2. Connecting edges with the same three forms of augmentation as in a class diagram, plus the following:
   a. a solid directed line, representing message passing; the line can be labeled with the name of the operation/function that is invoked to receive the message
   b. a dashed directed line, representing structural dependency; dependencies represent imports in both RSL and Java
3. Package elements may be nested inside each other as an alternate means of showing hollow-diamond-style containment

The message-passing lines *abstractly* represent communication in that the implementation details of the communication are abstracted out. In particular, communication can be by normal procedure call, remote procedure call, subprocess invocation, thread activation, or some other means. This level of detail is *not* specified in the package diagram.

Figure 4 shows the general structure of a package diagram. The diagram depicts three packages in which Subsystem 1 calls (or otherwise invokes) Function 1 in Subsystem 2. Subsystem 2 imports *DataD1* from Subsystem 1. Subsystem 1 imports *DataD2* from Subsystem 3.

In RSL and Java, packages are defined as *modules* and *packages* respectively.

5. Discussion

The shape of a diagram node is unique across all diagrams. Furthermore, the node shape can be traced to a specific semantic construct in RSL and/or Java. Viz.,

- a folder-shaped box is a packaging unit, consisting of multiple RSL objects and operations or multiple Java classes

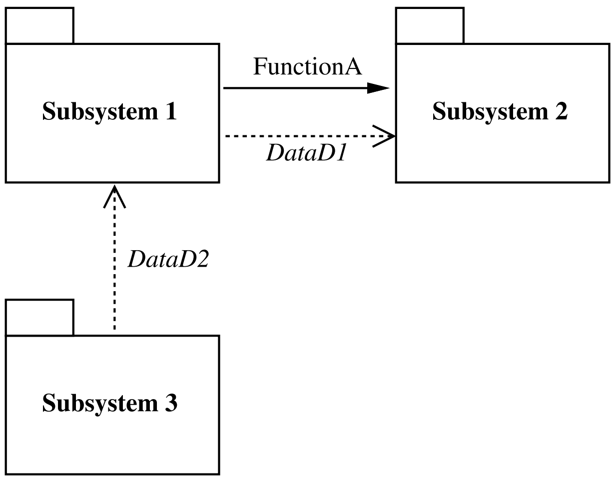


Figure 4: General Structure of a Package Diagram.

- a box is a datatype, which is either an RSL object or a Java class
  - *o* the second and third parts of a three-part box represent the components and operations attributes of an RSL object or the data and function members of a Java class;
  - *o* a hollow triangle connecting boxes abstractly represents inheritance in either RSL or Java;
  - *o* a hollow diamond connecting boxes represents component-of in RSL and membership in Java
- a circle (or ellipse) is an RSL (functional) operation
- a rounded rectangle is a Java (imperative) function (a.k.a., method)

The uniqueness of shapes allows diagram elements to be combined in ways not fully supported in UML, but which is conceptually compatible with UML. In particular, the appearance of rounded rectangles in data diagrams depicts function membership in a form not directly supported in UML.

The current version of UML supports neither dataflow nor function diagrams in the forms defined above. UML does support state-transition diagrams that have similar semantics to dataflow diagrams. In a state-transition, nodes are shown as rounded rectangles. This may cause some confusion with our use of rounded rectangles for functions. However, the edge shape in a UML state transition diagram is different than in a function diagram, so that the two forms of diagram can be distinguished.

It should be noted that all of the above notations except for function diagrams are orientation independent. That is, diagrams can be drawn vertically, horizontally, or any combination of orientations. The restriction for function diagrams is that a calling function must be drawn above or to the left of the functions that it calls.