IDEA GROUP PUBLISHING
701 E. Chocolate Avenue, Hershey PA 17033-1117, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com
ITP4126

Use of UML Stereotypes in Business Models

Daniel Brandon, Jr.
Christian Brothers University, Information Technology Management Department
650 East Parkway South, Memphis, TN, Tel: 901-321-3615, Fax: 901-321-3566, dbrandon@cbu.edu

OVERVIEW

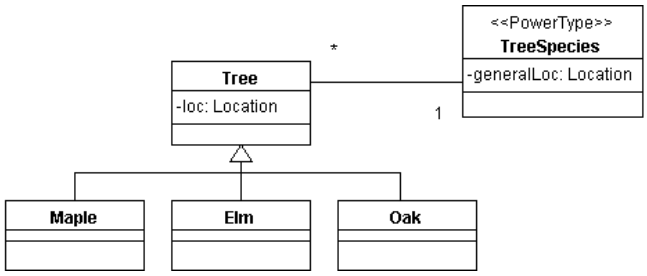
The UML (Unified Modeling language) has become a standard in design of object oriented computer systems. UML provides for the use of stereotypes to extend the utility of its base capabilities. In the design and construction of business systems, we have found some particularly useful stereotypes, and this paper defines and illustrates these.

UML STEROTYPES

"Stereotypes are the core extension mechanism of UML. If you find that you need a modeling construct that isn't in the UML but it is similar to something that is, you treat your construct as a stereotype." [Fowler, 2000] The stereotype is a semantic added to an existing model element and diagrammatically it consists of the stereotype name inside of guillemots (a.k.a. chevrons) within the selected model element. The guillemot looks like a double angle bracket (<< … >>), but it is a single character in extended font libraries. [Brown, 2002] The UML defines about 40 of these stereotypes such as "<<becomes>>", "<<include>>", and "<<signal>". [Scott, 2001] However, these 40 standard stereotypes do not add the meaning necessary for automatic code generation in a UML CASE tool.

One common general use of the stereotype is for a metaclass. A metaclass is a class whose instances are classes, and these are typically used in systems in which one needs to declare classes at run time. [Eriksson, 1998] A similar general use is for powertypes. A powertype is an object type (class) whose instances are subtypes of another object type. Figure 1 shows an example of the use of stereotypes for powertypes. [Martin, 1998]

Figure 1: An example use of stereotypes for powertypes



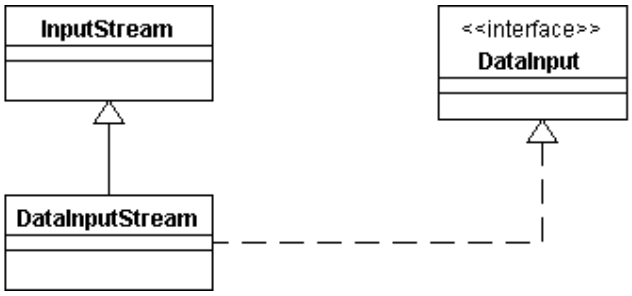
USER DEFINED STEREOTYPES FOR BUSINESS SYSTEMS

In the design of business systems we have found some stereotypes that were useful, and two stereotypes that are extremely useful. When defining stereotypes it is necessary to describe: [Eriksson, 1998]:

1. On which [UML] element the user defined stereotype should be based
2. The new semantics the stereotype adds or refines
3. One or more examples of how to implement the user-defined stereotype

A common use of stereotypes in business systems is for interfaces as found in Java or CORBA; this is shown in Figure 2. An interface typically has public functionality but not data (unless holding data for global constants). The class model element has been modified with the "<<interface>>" notation. This is commonly used for UML CASE products that do not have separate interface symbols or where these symbols do not allow data (i.e. global constants).

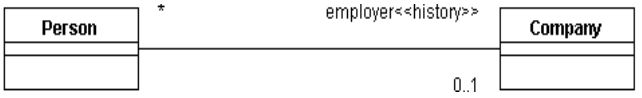
Figure 2: A common use of stereotypes in business systems



Still another common stereotype usage in business systems is to clarify or extend a relationship. Figure 3 shows a stereotype called "history" which implies a "many" cardinality for history purposes, that is, each Person has zero or one current employers but may have many employers in terms of the employee's history. It may imply some common functionality upon code generation such as [Fowler, 2000]:

Company Employee::getCompany(Date);

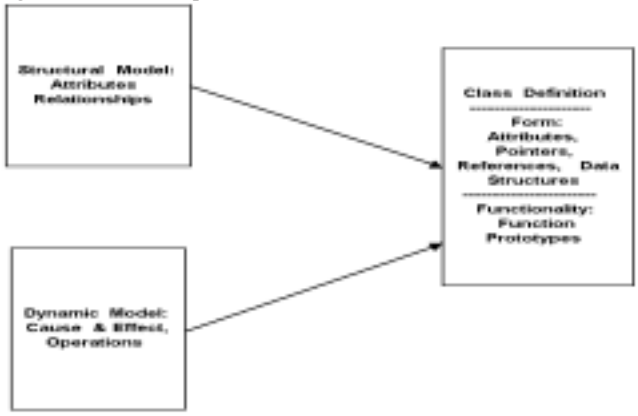
Figure 3: "History" stereotype



CODE WRITING AND GENERATION

Most modern UML CASE (Computer Aided Software Engineering) products can generate "skeleton" classes from the UML class diagrams and possibly other diagrams. For business systems design, we need to write the code for our classes (usually implemented in Java or C++) based on both the Structural Model (UML Class Diagram) and the Dynamic Model (UML Activity Diagram). This process is shown in Figure 4. It is very important that consistency between the two diagrams is achieved.

Figure 4: Code example



This paper appears in *Issues and Trends of Information Technology Management in Contemporary Organizations,* the proceedings of the Information Resources Management Association International Conference.

Many such CASE products allow the user to write their own "class generation scripts" in some proprietary scripting language or in a general scripting language (i.e. Python). With user defined stereotypes, the user can modify the class generation script code to use their stereotypes as needed.

RELATIONSHIP OBJECT TYPES

Often simple relationships (such as basic associations) need to be modeled as object types because these relationships have data content and/or functionality. Figure 5 shows a simple association between two object types representing the relationship "current marriage". If we need to maintain an attribute on each marriage (such as rating), then we can more effectively represent the relationship as an object type as shown in Figure 6. Here we use the "relationship" stereotype to indicate that this object type is a relationship; and the code generation can use a more simplified class representation. Others authors have suggested other notation for relationship object types such as "placeholders" [Martin, 1998], and UML suggests using the dotted line from a standard object type (class) to the relationship line. But implementing these other diagramming techniques in code generation is difficult and has ambiguity problems.

Figure 5: "Current marriage" association

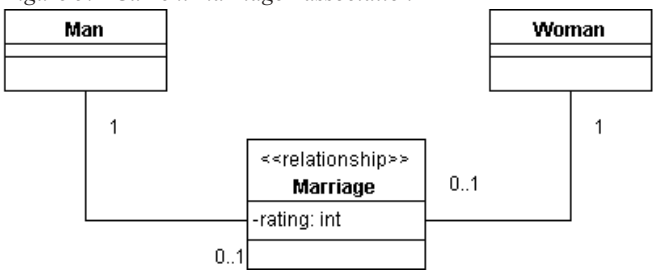
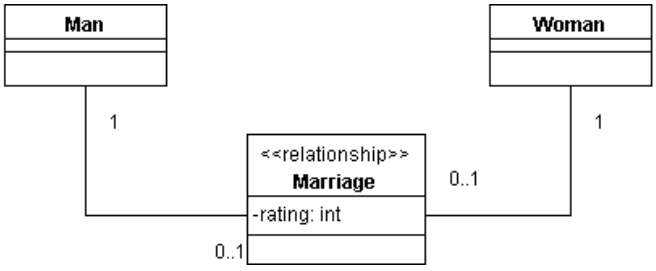


Figure 6: "Current marriage" object type



ACTIVITY DIAGRAMS

A UML Activity Diagram is a state diagram in which most of the states are action states and most of the transitions are triggered by the completion of these action states. This is the case in most models of business systems. Activity Diagrams identify action states, which we call operations [Martin, 1998], and the cause and effect between operations. Each operation needs to belong to an object type, at least for a C++ or Java implementation. Operations may be nested, and at some point in the design the operations need to be defined in terms of methods. The methods are the processing specifications for an operation and can be so specified in lower lever activity diagrams, pseudo code, or language specific code. Note that the term "methods" may cause some confusion here since in programming terminology, a method is a function defined within a class and it is invoked upon an object (unless it is a static method).

Current Drawing Methodology

Figure 7 shows a typical UML activity diagram for a simple ordering process. The operations are represented in the ovals and the arrows show the cause and effect scenario or the "triggers". In this diagram there are two "fork/join" model elements, and the use of "conditional branch states" is also common. Each of the operations must be associated with a particular object type. The standard way to do that in this UML type diagram is to use "swimlanes", and these are the vertical lines shown in Figure 7.

There are two problems with the standard representation as shown in Figure 7. The first problem is that as the system gets more complex (more object types and operations), it is very difficult to draw in swimlanes. The second problem is that code generation is very difficult in UML CASE products since you have to scan the geometry of the drawing to find out which operations lay in which swimlanes.

Figure 7: "Swimlanes"

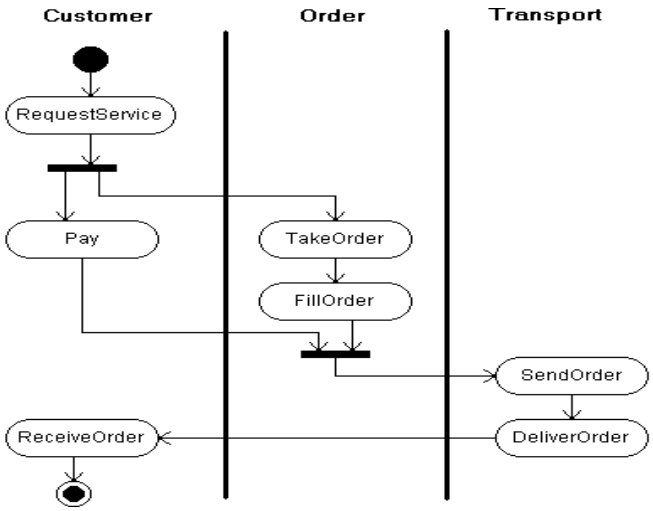
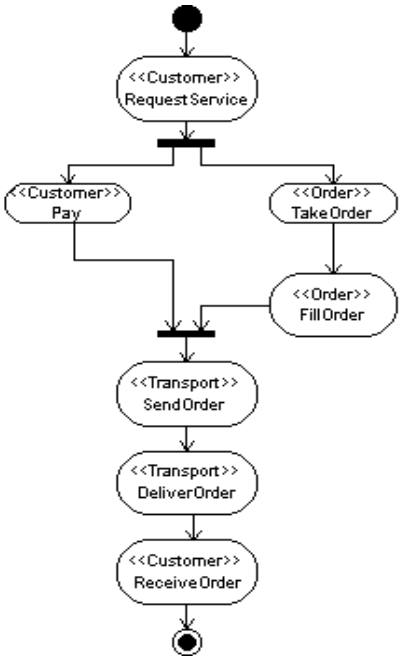


Figure 8: "Operation owner" stereotype



Stereotype Usage for Operations

Our solution to the above problems with standard UML activity diagrams is to use a stereotype for the operation element to indicate the object type (class) owning that operation. Figure 8 shows the same systems as Figure 7 drawn with the "operation owner" stereotype.

Model Consistency

A final business system design will involve several UML diagram types. For example business systems typically have a static Structural Diagram (Class Diagram) and a Dynamic Diagram (UML Activity Diagram). These diagrams must be consistent with one another, in particular:

1. The object types (shown with the operation stereotype notation) that contain the operations in activity diagram must be included on the structural diagram.
2. The operations shown in the activity diagram (along with the object types identified with the stereotype notation) must be included as operations in the same object type on the structural diagram.

EXAMPLE USAGE

This section describes an example of the use of our "operation owner" stereotype on a simple problem. The UML CASE product used was Object Domain Version 2.5 [Object Domain], and the example was implemented in C++.

Design

The business system being modeled is the process of registering students for classes. In this simple model, we have just two object types: Student and Class. The static structure diagram (UML Class Diagram) is shown in Figure 9. Figure 10 shows the activity diagram for the registration process (which is an operation in the Student object type). Figure 11 is the breakdown of that registration operation into other operations, again using our "operation owner" stereotypes.

To express the processing specifications of each operation (the "methods"), we can use pseudo code, specific language code, or lower level activity diagrams. Figures 12 and 13 show processing specifications represented in activity diagrams. Here our "operation owner" stereotype notation uses the object type name and operation name (i.e. Class::addStudent) since the information within the oval is a process description not an operation name. At code generation, the information within these ovals is just added to the code for the operation as a comment (i.e. //add student to class list).

Implementation

An implementation of these diagrams (in C++) is shown in Figures 14 through 17. Figure 14 shows the class definitions. Figure 15 shows the implementation of the operations (C++ functions) of the

Figure 9: UML class diagram

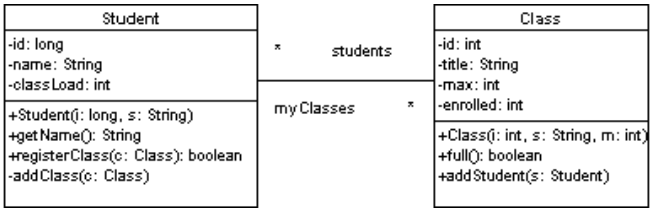


Figure 10: Registration process

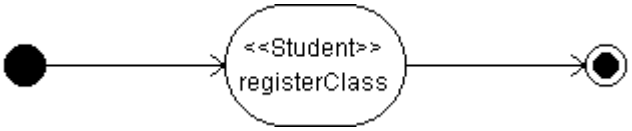


Figure 11: Breakdown of the registration process

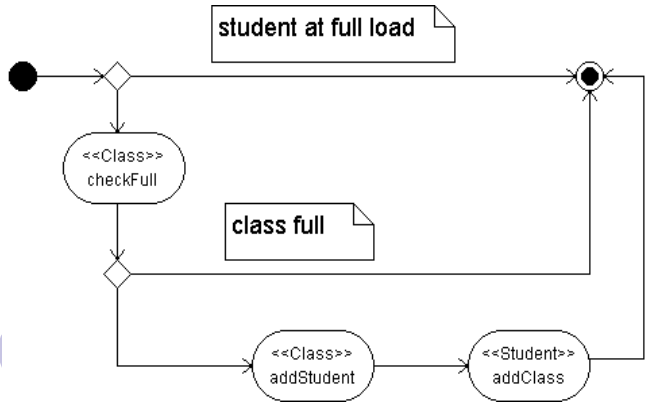


Figure 12: Processing specifications

Figure 13: Processing specifications

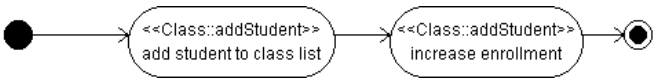


Figure 14: Class definitions

```
#include <iostream.h>
#include <string.h>

#define MAXSTUD 25
#define MAXCLASS 5

enum boolean {FALSE, TRUE};
class Class;

class Student
{
  private:
    long id;
    char name [30];
    int classLoad;
    Class *myClasses[MAXCLASS];
    void addClass(Class *c);
  public:
    Student (long, char*);
    char * getName();
    boolean registerClass(Class *);
};

class Class
{
  private:
    int id;
    char title[20];
    int max;
    int enrolled;
    Student *students[MAXSTUD];
  public:
    Class (int, char *, int);
    boolean checkFull();
    void addStudent(Student *);
};
```

Figure 15: Implementation of the operations of the student class

```
Student::Student (long i, char * s)
{
    id = i;
    strcpy(name, s);
    classLoad = 0;
    for (int j = 0; j < MAXCLASS; j++)
            myClasses[j] = NULL;
    cout << "New student created: " << s << endl;
    return;
}

char * Student::getName()
{
    return name;
}

boolean Student::registerClass(Class *c)
{
    if (classLoad == (MAXCLASS - 1))
    {
            cout << "No more classes for this student" << endl;
            return FALSE;
    }
    if (c->checkFull() == TRUE)
    {
            cout << "Class is full" << endl;
            return FALSE;
    }
    c->addStudent(this);
    addClass(c);
    return TRUE;
}

void Student::addClass(Class *c)
{
    myClasses[classLoad] = c;
    classLoad++;
    return;
}
```

Figure 16: Implementation of the operations of the Class class

```
Class::Class(int i, char * s, int m)
{
    id = i;
    strcpy(title, s);
    max = m;
    enrolled = 0;
    for (int j = 0; j < MAXSTUD; j++)
            students[j] = NULL;
    cout << "New class created: " << s << endl;
    return;
}

boolean Class::checkFull()
{
    if (enrolled == max)
            return TRUE;
    else
            return FALSE;
}

void Class::addStudent(Student *s)
{
    students[enrolled] = s;
    enrolled++;
    cout << "Student: " << s->getName();
    cout << " enrolled in " << title << endl;
{
```

Figure 17: Sample "driver" or C++ main function

```
/* ——— Main Function ——— */
int main()
{
    // Create a student
    Student s1 (123456789, "John Doe");

    // Create a class
    Class c1 (352, "Object Oriented", 20);

    // Register the student in the class
    s1.registerClass(&c1);

    return 0;
}
```

Student class. Figure 16 shows the implementation of the operations (C++ functions) of the Class class. Figure 17 is a sample "driver" or C++ main function.

CONCLUSION

UML stereotypes can be very useful in designing business systems. The use of a "relationship" stereotype is helpful in static structural models (UML Class Diagrams) and the use of an "operation owner" stereotype is most helpful in dynamic models (UML Activity Diagrams). These stereotypes aid in both the design/drawing phase and in the implementation (coding) phase of the overall system construction.

REFERENCES

Brown, David. An Introduction to Object-Oriented Analysis, John Wiley & Sons, 2002

Erikson, Hans-Erik and Penker, Magnus. UML Toolkit, John Wiley & Sons, 1998

Fowler, Martin and Kendall, Scott. UML Distilled, Addison-Wesley, 2000

Martin, James and Odell, James. Object Oriented Methods – A Foundation (UML Edition), Prentice Hall, 1998

Object Domain, Object Domain Systems Inc., [www.objectdomain.com], 2001

Scott, Kendall. UML Explained, Addison-Wesley, 2001