Putting Business Rules into Business Objects

Jill Nicola (jill@streamlinedmodeling.com)
Mark Mayfield (mark@streamlinedmodeling.com)
Mike Abney (mike@streamlinedmodeling.com)

Introduction

Imagine a world without business rules: subways and elevators taking off with open doors; perishable food being loaded into non-refrigerated trucks; automobiles with duplicate license plates, and other people withdrawing money from your checking account. Not a very pretty picture, is it? In a world without business rules, business processes can be initiated by the wrong people, at the wrong times and places, and with the wrong things. Not only does system integrity go right out the window, but dangerous real-world side-effects become very likely.

We rely on software to enforce many of these business rules, but translating rules about business processes into software routines isn't easy. Object modeling techniques help by expressing business processes in representations that map easily into software; however, rigorous techniques for enforcing business rules in object models are rare or nonexistent. This article discusses a practical methodology for enforcing business rules in business objects and shows how to add business rule checking to an object's implementation with examples in Java.

Towards a Business Rules Methodology

An object-oriented system executes a business process by creating and accessing objects that represent the people, places, and things involved in the process and by recording events that occur during the process. A well-designed system puts responsibility for checking business rules in the business objects they govern, making maintenance and extensibility considerably easier. The purpose of a business rules methodology is to eliminate the guesswork in mapping rules to objects and designing rule-checking procedures. This article focuses primarily on how to design rule-checking procedures. Strategies for mapping rules to objects are outlined in-detail in our book, *Streamlined Object Modeling*, Nicola, Mayfield, and Abney[1]. A simple example will help highlight the issues involved.

Consider the simple business process of withdrawing money from an account. The object-oriented version of withdrawing money from an account involves objects for the account (thing), the customer (person), and the withdrawal (event). (See Figure 1.) Business rules regulate the process so that (1) a customer cannot withdraw money from an account he does not own, and (2) a customer cannot make a withdrawal for an amount greater than the balance of the account.

Although domain experts expressed the business rules in terms of limits on the human customer's actions, it would be a mistake to enforce the rules in the customer object. After all, a customer object may be involved in many business processes, so putting all the business rules in it would be unwieldy. An even better reason is that the rules refer to

the properties and collaborations of the thing being acted upon, the account. In general, the object being acted upon enforces business rules that limit actions involving it.

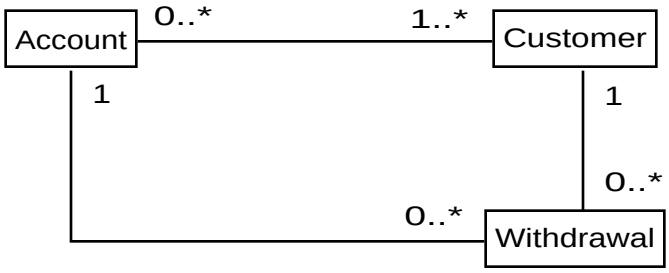

Figure 1. Object model of the account withdrawal business process.

Thinking about business rules in terms of the objects affected makes clear the real reasons for the rules: to protect system integrity by preventing objects from entering invalid states. In the account example, the business rules prevent an account from establishing collaborations with improper withdrawals: (1) an account cannot collaborate with a withdrawal from a customer who does not own the account; (2) an account cannot collaborate with a withdrawal whose amount exceeds the balance of the account.

This simple example yielded three important lessons:

(1) Object models describe which classes of objects collaborate during the business process, and business rules constrain which objects in these classes can legitimately collaborate.

(2) Business rules that regulate how people interact with things in the real world are enforced in the object world by the things acted on.

(3) To prevent invalid system states, business rules should be enforced whenever an object changes state.

These lessons help us formulate our design principles for checking business rules.

Design Principles for Checking Business Rules

Business rule checking should be a fundamental part of every object's implementation, as essential as property accessors, object constructors, and equality operators. Making objects responsible for the rules that govern them is not only good object think, it is also good object design, ensuring robust business services and simplifying rule maintenance and upgrades.

When to Check Business Rules

During the course of a business process, objects change state: order objects transition from pending, to final, to shipped states; account objects have their balances increased or decreased after deposits and withdrawals, respectively; and airplane objects receive assignments to flights on specific dates. In a well-designed system the objects check the business rules before changing state, so that canceled orders don't get shipped, account balances don't become negative, and airplanes don't get conflicting assignments.

In a truly robust system, every change of state by an object is governed by business rules that the object checks before making the state change. Such rule checking sounds difficult to implement. However, thanks to the encapsulated nature of objects, there are really only three ways in which an object changes state: (1) by changing one of its property values, (2) by establishing a collaboration with another object, or (3) by removing an existing collaboration with another object. Actually, the last two ways are pretty much the same, so we can simplify a bit and say that the time to check business rules is when an object changes a property value or collaboration.

When to Bypass Business Rules

Not all object state changes are the same. Restoring an object from persistent storage often means creating an empty object and loading its properties and collaborations using stored values. Running business rules while recreating a stored object is both inefficient and unnecessary. Similarly, certain business scenarios may require bypassing an object's business rules that are checked during other business scenarios. For example, an object may have a business rule preventing the removal of an essential collaboration, but may allow that collaboration to be switched out with another object. To make the switch the affected object checks the new object against business rules and if these succeed bypasses the business rules for removing the existing collaboration. Bypassing business rules should be the exception, but any rule checking design must be flexible enough to permit selectivity in rule checking.

Where to Check Business Rules

Putting rules entirely within an object's implementation limits system pluggability and extensibility. Instead rule logic is typically delegated to policy objects, or housed in external rule databases or method dictionaries that can be updated frequently. A good design for rule checking is flexible enough to support all these implementations. The important thing is that the object affected by the rule initiates the rule logic, either by invoking its own internal services, or by messaging a helper policy object, or by invoking methods stored externally, or by any number of other possibilities.

Logic vs. Business Rules

Not all state transition rules are business rules. Business rules, which come from clients and domain experts, check property values and collaborating objects against acceptable ranges and standards defined within the business domain. Business rules specify that an account balance cannot be negative, a cancelled order cannot collaborate with a shipment, and an airplane must check any new flight assignment against existing flight assignments. Logic rules on the other hand check for invalid program states, such as passing a null pointer instead of an object, or a character value instead of a numeric value. Unlike business rules, logic rules typically don't evolve over time and don't get overridden by a specialization class, so it is a good idea to segregate logic from business rules.

Design Principles

There are five design principles for checking business rules in objects:

1. Business rule checking occurs when objects change state.

2. Business rule checking is invoked by the object changing its state.
3. Business rule checking invokes logic that may or may not reside in the affected object's implementation.
4. Business rule checking must be flexible enough to allow selective bypassing by trusted services.
5. Business rule checking is separate from logic rule checking.

Designing Business Rules Checking

Our rule checking design begins in the object definition, usually called a class implementation. It is common practice and good class design to isolate in separate methods, called write accessors, code that changes the state of the object. We take this a step further and isolate business rule checking and actual state change into separate methods too. These additional methods partition an object's state change into three levels of processing: a public access layer, a business rules layer, and a state change layer. Having this layered approach satisfies our need for the flexibility to bypass, extend, or override business rule checking.

The **public access layer** contains the accessor methods for changing an object's state. Because these methods always invoke business rule checking they are the methods available to object editors, methods of other objects, and internal methods needing business rule checking. When the business rules succeed these accessors delegate the actual state change to the methods in the state change layer. Public accessors should not be overridden by specialization classes, also known as subclasses; however, with this layered approach, specialization classes have the flexibility of overriding or extending the business rules and state change procedures. Also, since null pointers and illogical values are bad at all levels of the hierarchy chain, logic checking also happens here, just prior to checking the business rules.

The **business rules layer** contains the methods that perform the rule checking. These methods can contain the actual rule checking logic, delegate it to helper objects, or invoke logic in external rule databases or method dictionaries. Regardless of the approach, when business rules fail an exception should be raised to signal failure. Specialization classes can implement their own business rules by overriding or extending the methods in the business rules layer.

The **state change layer** contains the methods that actually change the object's state. These methods don't do any rule checking, so they are ideal for reconstructing objects from persistent storage or for internal services that need to bypass rule checking. Specialization classes can implement different representations or state change procedures for an inherited property by overriding or extending these methods.

Figure 2 shows the three-layered design approach for state changing methods. Accessors that change an object's state are typically called set accessors or write accessors.

```
Public set accessor method
   Logic test   // often internal to the public accessor
   ⇒ Business rule test method
   ⇒ State change method
```

Figure 2. Three-layered design approach for methods that change an object's state.

With slight differences in their implementations, the three-layered design works for both property accessors and collaboration accessors.

Property Write Accessors with Business Rule Checking

Property business rules specify the legal values or ranges for properties. For purposes of rule checking there are two kinds of properties: enumerated and continuous. An enumerated property can only take on a value that is within a set of fixed values. A typical enumerated property is a status property that can take on one of the following values: pending, completed, or cancelled. A continuous property is defined by a range of values or is open-ended. Examples of continuous properties are a date property and a name property.

Continuous Property Write Accessors

A continuous property has a single write accesor that accepts a value, a single business rule test method to validate the value, and a single state change method to store the value. (See Figure 3.)

`setX``(aValue)`	Public write accessor to change the value of property X to aValue.
`testSetX``(aValue)`	Business rule method to validate aValue as a value for property X.
`doSetX``(aValue)`	State change method to assign aValue as the value for property X.

Figure 3. Methods for changing the state of continuous property X.

Internal methods and data management methods that can set the continuous property without rule checking call the "doSet" method directly. All others – object editors, methods of other objects, and internal methods needing business rule checking – change the property through the "setX" method.

Enumerated Property Write Accessors

An enumerated property has multiple write accesors, one for each of the enumerated values. Having multiple accessors encapsulates the true representations of the enumerated values. We recommend multiple test methods, too, in place of one test method with a large switch statement. On the other hand, one "doSet" method is usually adequate. (See Figure 4.)

`setXY`	Public write accessor to set the value of property X to Y
`testSetXY`	Business rule method to validate Y as a value

	for property X
doSetX(aValue)	State change method to assign aValue as the value for property X

Figure 4. Methods for changing the state of enumerated property X.

Property Write Accessors – Example

An online commodity brokerage assigns brokers to handle every sale order through the order workflow process. Each sale order has a date indicating when it was entered into the system, a unique identifier, and a status that is either pending, completed, or cancelled.(See Figure 5).

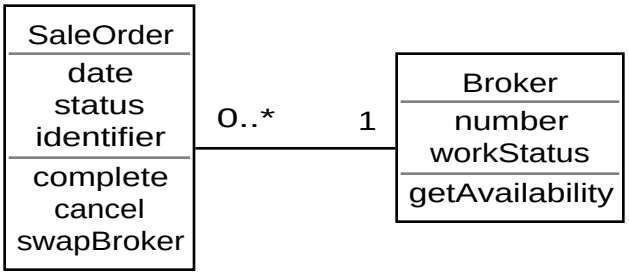

Figure 5. A sale order assigned to a broker.

Business rules dictate how the status property changes:

- If the status is pending it can transition to completed or cancelled.
- If the status is completed it cannot transition to cancelled or pending.
- If the status is cancelled it cannot transition to pending or completed.

A partial listing for the sale order's property write accessors is shown in Figure 6. For the status property only the status-completed methods are shown, but there are similar methods for setting the status to accepted and canceled.

```
/* PUBLIC ACCESS METHODS */

public void setStatusCompleted() throws BusinessRuleException
{
   this.testSetStatusCompleted();
   this.doSetStatus(SaleOrder.STATUS_COMPLETED);
}

public void setIdentifier(String aString) throws
BusinessRuleException
{
   if ((aString == null) || (aString.equals(“”)))
   {
      throw new BusinessRuleException("Identifier cannot be null or
empty.");
   }
   this.testSetIdentifier(aString);
   this.doSetIdentifier(aString);
```

```
}

/* BUSINESS RULE CHECKING METHODS */

public void testSetStatusCompleted() throws BusinessRuleException
{
   if (this.isStatusPending()) return;
   else
   {
      throw new BusinessRuleException("Non-pending sale Order cannot
become completed.”);
   }
}

public void testSetIdentifier (String aString) throws
BusinessRuleException
{
// <snip> code for checking uniqueness of identifier.
}

/* STATE CHANGE METHODS */

public void doSetStatus(NominationStatus aStatus)
{
   this.status = aStatus;
}

public void doSetIdentifier(String aString)
{
   this.identifier = aString;
}
```

Figure 6. Sale Order property acccessors.

Again, although this example shows the business rules directly in the Java code, the test methods could just as well have invoked external procedures. The point is to organize the code in a predictable manner so that business rules are checked at specific times through specific methods. As a later example shows, standardizing business rule checking allows selective application and by-passing of business rules.

Collaboration Accessors with Business Rule Checking

With collaborations, business rule checking gets more complicated because there are two objects involved: the object establishing or removing the collaboration, and the object that is the collaborator. Both objects typically have their own set of business rules checking whether the collaboration can be established or removed.

A collaboration can be either single-valued, as a sale order having only one broker, or multi-valued, as a broker having many sale orders. We use the same implementation pattern for both types of collaborations, because over time a single-valued collaboration can potentially evolve into a multi-valued collaboration, for example when a system decides to keep history about past collaborations.

addX(anObject)	Public write accessor to add anObject as an X collaborator.
testAddX(anObject)	Business rule method to validate anObject can be an X collaborator.
doAddX(anObject)	State change method to assign a reference to anObject as an X collaborator.
removeX(anObject)	Public write accessor to remove anObject as an X collaborator.
testRemoveX(anObject)	Business rule method to validate anObject can be removed as an X collaborator.
doRemoveX(anObject)	State change method to remove a reference to anObject as an X collaborator.

Figure 7. Methods for changing the state of a collaboration X.

Collaboration Write Accessors – Example

In the online commodity system example, business rules dictate whether a broker can be assigned to a sale order. Expressing the rules in terms of collaboration constraints yields:

- A broker on leave or vacation or a broker with too many pending sale orders cannot collaborate with a new sale order.
- A sale order cannot add a broker if it already has one.

These rules are enforced in the testAddSaleOrder method in the Broker class and the testAddBroker method in the SaleOrder class, respectively. Both sets of rules must be called whenever a sale order – broker collaboration is established, from either direction. To reduce code duplication one object delegates responsibility for establishing the collaboration to the other, called the director. For strategies on selecting which object is the director see Nicola, Mayfield, and Abney[1].

```
SaleOrder.java

public void addBroker(Broker aBroker) throws BusinessRuleException
{
   if (aBroker == null)
   {
      throw new BusinessRuleException("Cannot add null broker.");
   }
   this.testAddBroker(aBroker);
   aBroker.testAddSaleOrder(this);
   this.doAddBroker(aBroker);
   aBroker.doAddSaleOrder(this);
}

Broker.java
```

```
public void addSaleOrder(SaleOrder aSaleOrder) throws
BusinessRuleException
{
   if (aSaleOrder == null)
   {
      throw new BusinessRuleException("Cannot add null sale
order.");
   }
   aSaleOrder.addBroker(this);
}
```

Figure 8. Sale Order and Broker write accessors.

This three-layered approach to business rule checking also simplifies writing services that selectively invoke business rules. For example, a swap broker service that allows a sale order to replace its broker with a new one, selectively applies the add rules while bypassing the remove rules. See Figure 9.

```
/* BUSINESS SERVICES */

public void swapBroker(Broker newBroker) throws
BusinessRuleException
{
   if (newBroker == null)
   {
      throw new BusinessRuleException("Cannot have null broker.");
   }
   Broker oldBroker = this.broker;
   this.doRemoveBroker(oldBroker);
   try
   {
      this.testAddBroker(newBroker);
      newBroker.testAddSaleOrder(this);
   }
   catch(BusinessRuleException excptn)
   {
      this.doAddBroker(oldBroker);
      throw excptn;
   }
   oldBroker.doRemoveSaleOrder(this);
   this.doAddBroker(newBroker);
   newBroker.doAddSaleOrder(this);
}
```

Figure 9. Business service selectively bypassing business rules.

Conclusion

To ensure system integrity business rules should be checked whenever an object changes state, and responsibility for checking rules belongs to the object governed by the rule. This article proposes making business rule checking a fundamental part of any object's implementation, and presents a layered approach to writing property and collaboration accessors. This layered approach by isolating business rule checking in standardized methods allows flexibility in applying and relaxing business rules, extending rules in specialization classes, and interfacing with external rule bases.

References

1. Nicola, Jill, Mark Mayfield, and Mike Abney. Streamlined Object Modeling. Upper Saddle River, N.J.: Prentice Hall PTR, 2002.