

# Streamlined Object Modeling

Rules for Managing  
Object Relationships

improving 



Improving is a software consulting firm that provides Applied Training, Certified Consulting, Rural Sourcing, and Recruiting services.

Applied Training—if you take a class from us, it comes with on-site consulting to help you apply what you have learned.

Certified Consulting—personality and behavioral profiles in addition to typical resumé information. Plus, we provide 100% money-back guarantees.

Rural Sourcing—(What I do.) Outsource to our low-cost center in Bryan/College Station for the same high-quality guarantee at a lower price.

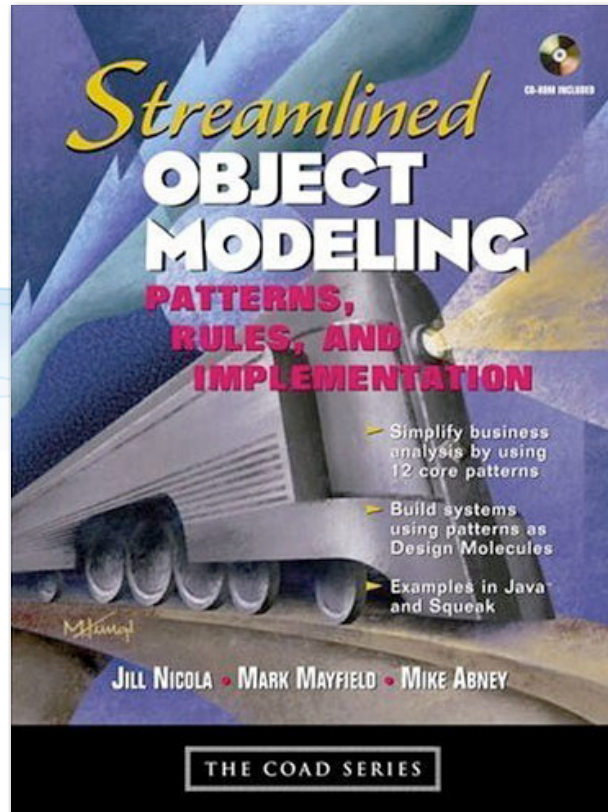
Recruiting—We are experts and finding the best and brightest. So now, we provide that same quality of recruiting to our customers.

# Old Dog New Tricks



Streamlined Object Modeling is more than 10 years old at this point. But nearly all of the high-level patterns and principles still apply just as much today.

However, some of the implementation details have changed. Specifically, this presentation will provide at least one example of using SOM with modern Java and JPA.



SOM came about after Jill and Mark had spent 15 years consulting in various business domains and helping clients to model their businesses. At the time, I had been writing Java for around five years and had helped them to apply some of their ideas to Java development projects.

We spent three years distilling, refining, and applying concepts of domain analysis and our experiences in implementing across finance, document management, retail, services contracts, and other business domains. The result was this book.



improving 

Object-oriented analysis and design is sometimes said to be about “nouns”—the people, places, things, and events a system involves. However, a key concept in SOM is that the focus belongs much more on the relationships between those objects. In fact, thinking in this way helps to discover more objects faster—you can expect relationship patterns to be there.

But there is more to it than structural patterns.

# Knock-knock



My daughter, when she was about four, came up with her own knock-knock joke:

Knock-knock.

Who's there?

Quesadilla.

Quesadilla who?

Quesadilla and beans.

What's wrong with it? It follows the pattern, but it breaks the rules embedded in the relationship between the set-up and the punch line.

It is risky to  
follow patterns  
without knowing  
the rules.



We all do this as we learn, but our goal should be to try to learn the rules.

# Kinds of Rules

- Type
- Multiplicity
- Property
- State
- Conflict



SOM provides a starting point for finding interaction rules. It identifies five kinds or categories of rules.

# Type



Type rules are like zoning. “Perishable food must be loaded onto a refrigerated truck.” If typing is implemented through class structure (static typing), most of these rules may disappear from the actual implementation.

# Multiplicity



Multiplicity rules define how many of its collaborator an object can handle. “An engine doesn’t exist once you remove all its parts, so you can’t remove the last one.”

# Property



Property rules come in validation and comparison flavors. Validation rules are local to the collaborator and, if broken, mean that object is not valid—it should not exist this way. Comparison rules involve checking against the potential partner. “A person must be 13 years old to become a registered user.”

# State



State rules are similar to property validation rules—they are local to the collaborator, but they are more specific to the collaborator’s “readiness” to make or break a connection to a partner. In other words, the object may have the right property values, but not be in the right state. “Cancelled orders cannot be shipped.”

# Conflict



Conflict rules are some of the most interesting. They are rules that can prevent a collaborator from connecting to a potential partner because of that partner's other collaborators.

Collection conflict rules occur when a collaborator asks or is asked if its list of current partners of the same type will allow a new partner to be added. "A student's class schedule cannot add a new class that is at the same time as any of the student's existing classes."

Event conflict rules occur between the set of collaborators involved in an event. "A sales transaction cannot include an alcoholic beverage if the customer is not 21 or older."

Conflict rules can also look like any of the other four types, except they are not applied directly between two collaborators but from one through its partner and into the partner's partner(s). "Perishable food can be loaded into any container, but that container can then only be loaded into a refrigerated truck."

	Actor	Role
Type		✓
Multiplicity		✓
Property		✓
State		✓
Conflict		✓



This is a sample rules table from the book. The check marks indicate where there are most likely to be rules in any domain that has this pattern.

Actor–Role is probably the most familiar collaboration pattern. It exists whenever people are involved in a system. “A person acts as a team member in the system, but to be a team member, the person must be either a consultant or an employee.”

In this case, the Role has all the rules. Why? It represents the Actor in a specific context. Therefore, it knows about most of the Actor’s properties in order to represent them. Also, the specific context of the Role is where the restrictions just naturally lie. “A person may have an SSN/EID or not, however an employee must have a valid one.”

	Outer Place	Place
Type		✓
Multiplicity	✓	✓
Property	✓	✓
State	✓	✓
Conflict		✓



Outer Place–Place (if you can get past the somewhat unwieldy name) is a relationship between a container and its contents when those contents are also containers. An example would be a warehouse containing racks or sections. Here, most of the rules tend to be on the “inner” place because it is the most specific and... localized.

However, in this case, the outer place also has rules. It might have a limit on the number of places it can contain. It may need to compare the contained places’ properties to determine if a new one will fit. It may also be “closed” or in some other state that doesn’t allow places to be added or removed.

There are more of these in the book—twelve in all. For now, let’s move on to...

# Implementation



Back when SOM was written, technologies like JPA didn't exist. So, in the book, rollbacks and other things are implemented in the domain rather than using a library that could take care of them for us.

```

@Entity
public class Person implements PersonProfile {
    @Id
    @GeneratedValue
    private Integer personId;
    private String name;
    private String email;
    @OneToMany(cascade = CascadeType.ALL,
        fetch = FetchType.LAZY, mappedBy = "person")
    private Set<TeamMember> teamMembers;

    public Person() {
        // required by JPA/Hibernate
    }

    public Person(String name) {
        this.setName(name);
        this.teamMembers = new HashSet<TeamMember>();
    }
}

```



Here is the initial definition of the Person class. A few things to note...

- JPA annotations at the field, not property level. This allows us much greater flexibility when defining our accessors.
- Note that the FetchType is LAZY. Many times we will be loading a person because we actually loaded one of its roles. Thus we likely don't want all of its other roles to load until we need them.
- Also note that this is a Set and not a List. Not only is it more appropriate when we don't care about order, but JPA will want an ordering column, especially if we try to load more than one List eagerly.
- We don't really want a default constructor because we don't want to create an invalid object. However, JPA requires it. We could use an isValid method or something similar to help us deal with this.
- Integers work fine for my keys, but if you have more than 2 billion records, you may want to switch to Longs.
- Note the PersonProfile interface. We'll talk more about it in a second. For now, just note that we are linking directly to the TeamMember class rather than to an interface. This is due to a limitation of JPA—it can't automatically figure out the class to instantiate if we use an interface. We would have to use a "targetEntity" attribute to make that work, and that would defeat the purpose of the interface.

```
public interface PersonProfile {  
    // Accessors - Properties  
    Integer getPersonId();  
  
    String getName();  
  
    String getEmail();  
  
    // Determine mine  
    boolean hasValidEmail();  
}
```

The PersonProfile is the set of methods we want to “inherit” in our role classes. That is, the degree to which we want “child” objects to represent their parents. Note that inheritance here means object/state inheritance, not just class/structure inheritance.



**Warning:  
Tangent  
Ahead**

improving 

The concept of object inheritance has been around for a long time. I first wrote an article about it in 2001 or so, but that was far from its inception. Languages such as Ruby and even Javascript are much better suited to its implementation, but Java is where our bread is buttered so....

# Test First Code



SOM focuses on the business rules that appear between collaborating objects. Thus, modification accessors should test those rules first, then do the requested action.

Most of these rules are fairly static, but they can change for subclasses or even because of business decisions. For this reason, it is suggested to extract the test methods from the accessors.

```
public void setEmail(String email) {
    this.testSetEmail(email);
    this.email = email;
}

public void testSetEmail(String email) {
    if (email != null && !EmailAddress.isValid(email)) {
        throw new BusinessException(
            "The email address is not valid.");
    }
}
```



One nice thing about the separate test method is that it doesn't have to be in the same class. It can be extracted into a "business rules engine." That can be a Drools-style or "business intelligence" implementation, but it doesn't have to be that complicated.

At one company, we took advantage of this to actually be able to swap out rules on the fly. It was a great demo, but likely not the safest solution. It depends on the environment.

In this case, we are setting a simple property. Therefore there isn't real "collaboration" with the email String. When there is a true collaboration, a "do" method will need to be pulled out.

We can see this if we look at an actual Actor-Role implementation. But first...

# Bi-Directional Access



... a note on bi-directional access. For a long time, bi-directional links between objects seemed to be discouraged. It was called a cyclic dependency. The main problem with that line of thinking is that object-oriented software works based on collaboration of objects. A Person needs to know about its TeamMember roles so that conflict rules can be checked. A TeamMember needs to know about the Person so that it can successfully represent that person in the context of the team.

When they first came out, my coauthors and I were some of the first to see problems with EJB 1 and 2. We claimed they were not object-oriented frameworks at all because they made these types of relationships difficult at the entity level. With EJB 3 and JPA, we have that capability back, but we have to manage the links ourselves when we want to modify them or the objects in the cache will not be linked properly.

```

// Person.java

public void addTeamMember(TeamMember teamMember) {
    if (teamMember == null) {
        throw new BusinessException(
            "Can only add an actual team member.");
    }
    teamMember.addPerson(this);
}

// TeamMember.java

public void addPerson(Person person) {
    if (person == null) {
        throw new BusinessException(
            "Can only add an actual person.");
    }
    this.testAddPerson(person);
    person.testAddTeamMember(this);
    this.doAddPerson(person);
    person.doAddTeamMember(this);
}

```

Here is the actual bi-directional collaboration example. Note that the Person delegates the action to the TeamMember. Only the TeamMember calls the “test” and “do” methods. Why? The most obvious reason is to avoid an infinite loop if each “add” method only called the other “add” method.

That said, the methods could be written to do a check to prevent that. However, that is a bit messy. We could have the “test” and “do” methods called from both “add” methods, but hopefully you’re familiar with the DRY principle. Finally, by making sure this is done only in one place, we can prevent deadlocks in multithreaded applications since the order will always be the same. (Assume for a second that we add the appropriate synchronization keywords or blocks.)

But how did we decide to put the work in the TeamMember class rather than the Person? Remember that the rule naturally tend to be on the more specific class? That led us to a principle we generalized as...

# Most Specific Carries the Load



..."The most specific carries the load." So, when in doubt, prefer to put things in the most specific class. This allows for the most flexibility. A truly special implementation could change the standard pattern if that makes sense. For example, we could use the same Person class as a parent with a Role that has to work asynchronously or something else.

```

// Person.java

public void testAddTeamMember(TeamMember teamMember) {
    // This is the Actor in Actor-Role. Probably no rules here.
}

public void testAddTeamMemberConflictRules(TeamMember teamMember) {
    for (TeamMember existingTeamMember : this.teamMembers) {
        existingTeamMember.testTeamMemberRoleConflict(teamMember);
    }
}

// TeamMember.java

public void testAddPerson(Person person) {
    if (!StringUtils.hasLength(person.getEmail())) {
        throw new BusinessException(
            "Person must have an e-mail to be a team member.");
    }
    person.testAddTeamMemberConflictRules(this);
}

public void testTeamMemberRoleConflict(TeamMember teamMember) {
    if (this.teamName.equals(teamMember.teamName)) {
        throw new BusinessException(
            "Person is already on that team.");
    }
}

```

Here is a look at the “test” methods. They are fairly trivial at this point. We should also be checking for multiplicity rules, more complex conflict rules, etc. Note the back-and-forth nature of how conflict rules are checked. This is so that the Person retains encapsulation of how to get access to its existing team members while the TeamMember retains the encapsulation of the rules themselves.

The “do” methods are boring and just set the field directly or call the appropriate method on the collection.

# Just Scratched the Surface



There are a lot of other implementation patterns that SOM discusses—way too much to cover here. Some implementations are dated, but the concerns discussed are still current. If you are interested in more, check out my web site. I have an archived copy of the book's web site. A lot of the information and the links on that site are out of date, but there are some sample chapters and an appendix that has the rest of the twelve pattern's rules tables.

Improving Enterprises:

<http://improvingenterprises.com>

Mike Abney (@mikeabney):

<http://mikeabney.com>

Improving Podcasts:

<http://improvingpodcasts.com>

Images used per license.

Source: <http://www.sxc.hu/>

improving 