Natural Creation
A Composite Pattern For Creating Objects

James Noble
Computer Science
Victoria University of Wellington
New Zealand
kjx@mcs.vuw.ac.nz

Context

Use the Natural Creation pattern when you want to create an object, and

- the details of the object to be created depend upon the current configuration of objects in the program.
- the object to be created will collaborate closely with the existing objects.
- you don't want to expose special "creation" objects — such as classes, prototypes, or factories.

Use Natural Creation when it makes sense within the application domain or simplifies the overall design — if it results in fewer objects or simpler interfaces

Problem

Object-oriented programs are communities of communicating objects. As programs run, they need to create new objects to represent new entities in the program's application domain and to support their own internal implementation. The structure of objects' collaborations is almost always more important than the class to which each object belongs, yet most programming languages' object creation mechanisms focus primarily on classes, rather than on object collaborations.

For example, imagine an agricultural information (AI) system to help farmers breed livestock, specifically sheep. To model the breeding process and estimate the characteristics of the resulting livestock, Lamb objects must be created which collaborate with the Ewe and Ram objects that (in the model at least) have created them.

The most common way to solve this problem is to use the primitive object creation mechanisms provided by object-oriented programming languages, such as a call to a constructor or a class method attached to the Lamb class (for example, in Java, `new Lamb()` or Smalltalk `Lamb new`). Primitive creation methods can certainly create objects, but will not initialise those objects correctly without extra work — in this case, the Lamb must be initialised with information from its parent sheep. Most languages allow construction operations to be specialised, so that the new object can be correctly initialised when it is returned, but this would still require the programmer to pass all the required information into

the specialised constructor or class method (for example, in Java `new Lamb(aEwe,aRam)` or Smalltalk `Lamb descendedFrom: aEwe and: aRam`). Language-level construction also requires the program to appeal to concepts which are not directly part of the application domain — in this case the abstract Lamb class.

So, how can you create an object which fits in with the existing objects in the program?

Forces

To solve this problem, we must consider a number of conflicting forces:

- *Abstraction* A program's code should represent the programmer's understanding of the application domain model, rather than the implementation details of objects, methods, or the mechanics of special patterns. Unfortunately, many creational language mechanisms (constructors, class messages) and creational patterns involve a large amount of low level detail.
- *Fidelity to the Domain Model.* A domain model should describe the important entities in the program's application domain. A good program design should try to represent those domain objects as simply and clearly as possible [9]. Many design patterns can complicate programs' domain models. In particular, application domain models should not include the complex creational mechanisms required by the Builder or Abstract Factory patterns, or even language-level constructs like classes or prototypes[1].
- *Marshalling.* Object creation involves three separate stages — marshalling the necessary information, creating the new object, and initialising the new object. Programming languages can create and initialise objects of a given class, and creational patterns like abstract factory or factory method can select the correct class for the object, but neither languages nor patterns support marshalling the required information in the first place.
- *Simplicity.* Language features and creational patterns can have complex interfaces. Often, they require more information than is really necessary to create an object in a particular context, and need to have all this information present explicitly, rather than gathering it from collaborating objects in the program.
- *Coupling.* Because language mechanisms and creational patterns expose the details of the creation mechanisms and present complex interfaces, clients that need to create new objects are tightly coupled to whatever mechanism will create the objects. Changes to the underlying creation mechanism will change the interface presented to the client, necessitating changes to the client code.
- *Explicit Creation.* Object-oriented programs are made up of objects, and objects come from creation mechanisms. To understand a program, you need to understand the objects making it up, which means understanding how they were created. Explicit creation operations (like C++ and Java's "new" statements or Eiffel's creation instructions) make every object creation very explicit in the program text. Unfortunately, creational design patterns such as Factory Method created objects without

[1] Pedantically, an object representing "the class of all Ewes" would be a meta-level object in a domain-specific metamodel, while an individual, real Ewe is an actual object in a domain model.

using explicit creation operations, making it more difficult to understand were the objects have come from.

- *Efficiency.* Object oriented programs can create thousands (or hundreds of thousands) of objects every second, and may create millions of objects during the course of their execution. So, whatever creation mechanisms are used must execute efficiently.

Solution

Create objects following the patterns of interaction of the existing objects in the program.

The Natural Creation pattern places the responsibility for creating new objects onto the existing objects in the program, and requires these objects to collaborate to create the new object.

To use the Natural Creation pattern, first determine the objects with which the newly created object must collaborate. From these objects, choose the one the new object will be closest to, and make this Mother object directly responsible for creating the new object. The Mother object should gather together all the information necessary to create the new object, create it, initialise it, and establish all its necessary collaborations. The Mother's method implementing the natural creation may need arguments to pass collaborators or customisation parameters to the new object.

Of course, *"there ain't no such thing as a free lunch"* [13, p.405], and the new object must be created somehow — typically by the Mother object using language mechanisms or another creational pattern. The point of the Natural Creation pattern is to shield clients of the creation operation from the details of its implementation.

In the AI example, two objects in the program are already involved in creating a new lamb — a ewe and a ram. The natural place to assign the responsibility for creating a new lamb is the ewe object, with the ram with which it is breeding passed as a parameter, say `ewe.breedFrom(ram)` in Java, or `ewe breedFrom: ram` in Smalltalk (see Figure 1). The ewe can actually create a new lamb by calling language level creation operations as necessary.

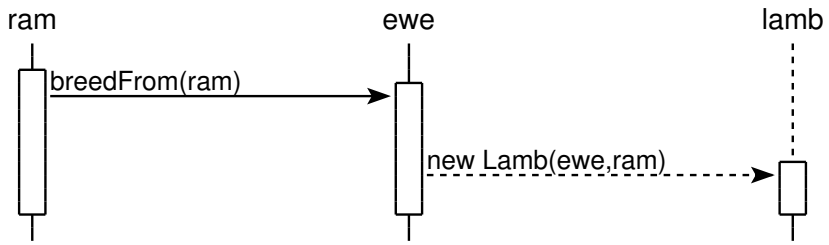

Figure 1. Creating a new Lamb

For this reason, we treat the Natural Creation pattern as a composite pattern [14], in that it describes how one or more lower level patterns can be used to provide a solution to a more complex, higher level, or larger scale design problem than the lower level patterns address directly.

Sketch

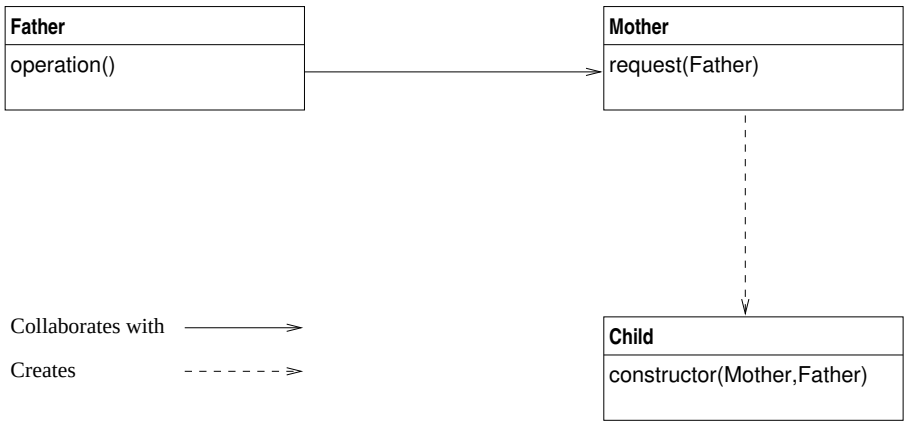

Figure 2. Natural Creation Structure

Resulting Context and Rationale

The Natural Creation pattern resolves a number of forces, that is, it has the following **benefits:**

1. *Creation is Abstract.* The object creation process is hidden from the client code. No special creational objects (like classes, prototypes, factories, or builders) are visible to client programmers — they simply call the natural creation interface, typically just a single method.
2. *Fidelity to the Domain Model.* By abstracting away from the details of the construction mechanism, a program will better reflect its application domain model. The domain model is not polluted by special creational objects, rather, object creation becomes a natural result of the collaborations between preëxisting objects.
3. *Marshalling is handled separately.* The Natural Creation pattern organises creation into three separate stages. First, it gathers the necessary information about the new object to be created. Second, the new object is created — this stage may be trivial if a language level creation operation is used, or very complex if a pattern such as Builder [6] or Product Trader [15] is used. The information gathered in the first stage may be used to provide instructions to the Builder or a specification for a Product Trader. Third, the new object is initialised and its collaborations with preëxisting objects are established, again using the information gathered at the first stage. This separation allows initialisation arguments to be gathered and marshaled independently from the creation and initialisation of the new object.
4. *Simplicity.* The interface provided by the natural creation pattern can be simpler than the interfaces provided by language features or creational patterns. There are two reasons for this: first, the natural creation pattern can be tailored to use only the information required when creating a given kind of object from a particular con-

text, and second, this information can be retrieved from collaborating objects in the program.

5. *Coupling is Reduced.* Because the details of the creation process have been abstracted out from client code, coupling between the creating client's code and the actual code embodying the creation is reduced. This has the advantage that the underlying creation mechanism or design pattern can be redesigned without affecting the interface presented to the client, and consequently, the client code.

The Natural Creation pattern also some leaves forces unresolved (or exposes new forces), that it, it has the following **liabilities:**

1. *Natural Creation is Implicit.* Natural Creation removes direct calls to language-level creation operations from client code. As a result, creation operations will be harder to find in the program text — for example, they will not be able to be located using program cross-reference tools which find constructor calls. Also, being unable to determine which methods create new objects may cause particular problems for languages without garbage collection, such as C++.
2. *Efficiency.* Natural Creation typically requires at least one extra method send whenever it is used to create an object. This may slightly slow the execution of the program, although these methods may often be good candidates for compiler inlining.

Related Patterns

As Natural Creation is a composite creational pattern, one or more creational patterns are typically applied as subordinate patterns whenever the Natural Creation pattern is applied. Figure 3 shows the relationships between the subordinate design patterns in more detail, showing the patterns that use other patterns, and the patterns that conflict (provide alternative solutions to similar problems) [10].

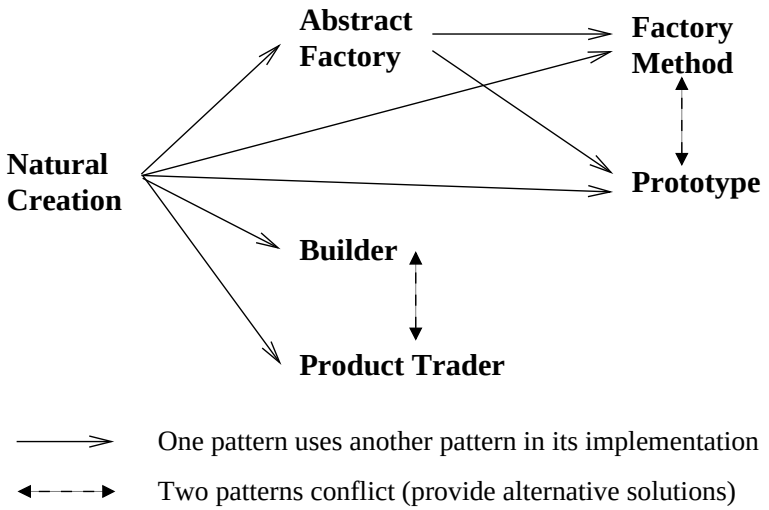

Figure 3. Natural Creation and Subordinate Creational Patterns

Factory Method The Factory Method pattern [6] is one of the most basic creational patterns. The Factory Method pattern allows subclasses to decide what class to instantiate to create an object, rather than hardcoding the class name in a superclass method. The AI example in the Appendix shows how a Factory Method can be used by the Natural Creation pattern — alternatively, an application of the Natural Creation pattern often looks like an application of the Factory Method pattern, except that the method is never overridden.

Considering one of the known uses of Factory Method in *Design Patterns* [6, p.115] can make this distinction clearer. The View class in the Smalltalk-80 MVC framework has a `defaultController` method which returns a controller for the view. This method is defined in terms of a `defaultControllerClass` method which returns the class of the controller to create, and it is this method which is overridden by subclasses. The `defaultControllerClass` method, then, is the Factory Method, while the `defaultController` method is an example of Natural Creation, as this is the method View's clients call.

Prototype The Prototype pattern [6] is an alternative to the Factory Method pattern that allows subclasses (or individual objects) to determine what kind of object to create, by creating that object by cloning a prototype rather than instantiating a class. The Prototype pattern can be used to implement Natural Creation when this flexibility is required, and it can also be used directly where new instances are created by cloning in the application domain [3].

```
Sheep breedFrom(Sheep ram) {
    Sheep clone = this.clone();
    clone.setEwe(this);
    clone.setRam(ram);
    return clone;
}
```

Abstract Factory The Abstract Factory pattern [6] is similar to the Factory Method pattern, in that it allows an object to be created without specifying its concrete class, however, Abstract Factory allows a family of related or dependent objects to be created so that they will work together. If a family of classes needs to be created, an Abstract Factory can be used to implement Natural Creation in much the same way as a Factory Method.

```
Sheep breedFrom(Sheep ram) {
    return sheepAbstractFactory.makeSheep(this, ram);
}
```

Note that an Abstract Factory can be implemented using either language-level object construction or the Prototype pattern, as shown in Figure 3.

Builder The Builder pattern [6] describes how the construction of a complex object can be separated from its representation, so that the same construction process can create different representations. Like an Abstract Factory, a Builder can require quite a complex client interface. The Natural Creation pattern should use a Builder when a complex object needs to be created, and can also be used to present a simpler interface to clients.

```
Sheep breedFrom(Sheep ram) {
    // again, the same creation interface
    return sheepBuilder.makeSheep(this, ram);
}
```

Product Trader The Product Trader pattern [15] allows clients to create objects by providing an abstract specification, and then using a Product Trader object to find, create, and return a new object which matches that specification. The client interface to a Product Trader is complicated by the need for the client to create the specification object, and then pass that to the trader. The Natural Creation pattern can use a Product Trader when the choice of class to instantiate depends upon a very flexible mapping from the marshaled information, and again has the advantage that it encapsulates the trader's complexity, presenting that same simple interface to the ultimate client whichever underlying creational pattern is used.

```
Sheep breedFrom(Sheep ram) {
    ProductSpecification spec =
        new Specification(Sheep, this, ram);
    return sheepProductTrader.createFor(spec);
}
```

Finally, creational programming patterns such as Constructor Method, Explicit Initialisation, and Lazy Initialisation [2] describe how object creation can ultimately be implemented using the facilities of a programming language.

Known Uses

The Natural Creation pattern has been used in quite a large number of systems, although it has not always been recognised as such.

The Smalltalk system, for example, makes quite heavy used of the Natural Creation pattern. For example, Points can be created from numbers using the `@` operator, rectangles from points by using the `corner` and `extent` methods, and fractions from numbers using the `/` operator.

```
point1 := 10 @ 10.
rectangle1 := point1 extent: point1.
rectangle2 := point1 corner: 100@100.
oneThird := 1 / 3.
```

The Java collection classes provide natural creation methods to create iterators (called Enumerations) on collections. For example,

```
Vector vec = new Vector(10);
Enumeration e = vec.elements();
```

Note that this abstracts the name of the concrete `VectorEnumerator` class out of the creation code. Other iterators (including Smalltalk Streams and C++ STL iterators) do not use natural creation methods and require the client to specify the class of the iterator (generally because these libraries have a larger number of iterators to choose from).

Many graphics libraries use Natural Creation to create objects which can draw on windows. For example, VisualWorks uses `GraphicsContext` objects [12], Smalltalk/V uses `Pen` objects [8], and the Java AWT uses `Graphics` objects [4].

In Java, to draw on a window you create a `Graphics` context object for the window and then draw on the context.

```
Graphics g = window.getGraphics();
g.fillRect(0, 0, 100, 100);
g.drawString(``Hello'',10,10);
```

VisualWorks Smalltalk also provides some other good examples of Natural Creation. For example, all Smalltalk objects understand the method `->` to create an association (like a cons cell) out of a pair of objects. This method is a pure example of the Natural Creation pattern (not a Factory Method) because it is never overridden. The method `asValue`, again understood by all objects is another example. The `asValue` method creates a value holder (a sort of decorator) containing the object to which the message is sent [16]. This method is only overridden in one class to which it should not apply.

A final Smalltalk example of the Natural Creation pattern is the way new classes are created in Smalltalk. For example, the Bag class is created as the subclass of the Collection class, adding a single instance variable (or field) called `contents`, as follows:

```
Collection subclass: #Bag
        instanceVariableNames: 'contents '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Collections-Unordered'
```

This is just a Smalltalk message send[2] that is sent to the superclass of the new (as yet nonexistent) subclass. The result of the message send is that a new class is created, ultimately by a language-level operation to create an instance of a system Metaclass. This message is never overridden in the Smalltalk system.

The Driver Duty Constructor pattern for public transport systems [17] uses Natural Creation to create a cascade (a hierarchy of nested obejcts of different classes [5]) of objects representing driver duty allocations, ensuring that the duty objects collaborate correctly.

Appendix

This section presents further implementation details about the Natural Creation pattern.

Participants

Father (Ram)

- Requests that Mother creates a new Child object.

Mother (Ewe)

- Defines an interface by which a child can be created.

[2]The message name in this expression is a real Smalltalk mouthful:
`subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:`

- Selects class of child object to create.
- Marshals initialisation data for child object constructor.
- May collaborate with other objects as necessary to gather initialisation data.
- Takes responsibility for actually creating the child.

Child (Lamb)

- The new object that is created and initialised
- Initialises its internal state based on arguments passed from Mother.

Collaborations

- Father requests object creation
- Mother determines the class of the Child to create
- Mother creates the Child, collecting and passing initialisation arguments.
- Child initialises itself.

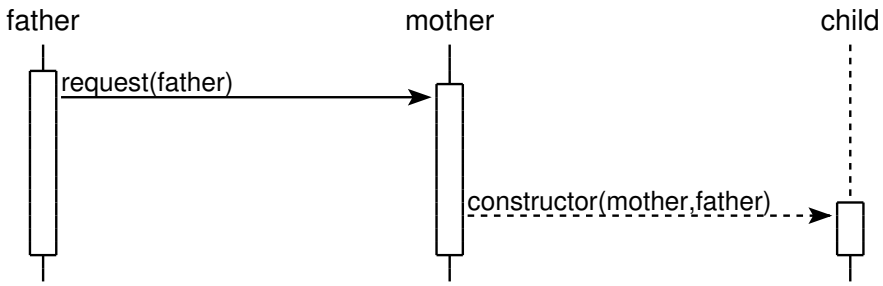

Figure 4. Natural Creation Collaborations

Sample Code

The following Java code shows how the Natural Creation pattern can be used to create Lamb objects for the AI example system. The pattern can be used in several different ways depending on the detailed design of the rest of the system, and other patterns that need to be applied.

The simplest case is where the Ewe, Ram, and Lamb objects are all instances of a Sheep class, but (for flock management reasons) a new Lamb needs to record its parent Sheep. The natural creation method `breedFrom` calls the language level constructor, removing the coupling to the Sheep class from the client.

```
class Sheep {
    Sheep ewe, ram;

    Sheep(Sheep ewe, Sheep ram) {
        this.ewe = ewe;
        this.ram = ram;
    }
```

```
    Sheep breedFrom(Sheep ram) {
        // should be called only on a ewe
        return new Sheep(this, ram);
    }
}
```

If the objects are from different classes (say subclasses of Sheep), then we can use the Factory Method pattern [6] to choose the appropriate subclass, based on the class of the ewe. For example:

```
class Merino extends Sheep {
    Merino(Sheep ewe, Sheep ram) {
        this.ewe = ewe;
        this.ram = ram;
    }

    Sheep breedFrom(Sheep ram) {
        // should be called only on a ewe
        return new Merino(this, ram);
    }
}
```

In this example, the `breedFrom` method is both a Factory Method (because it is overridden in subclasses to choose the class to create) and a Natural Creation method (because it is called from clients to create objects). Note that, as far as the client is concerned, the interface the client needs to call is exactly the same as the earlier example (without a Factory Method) because the natural creation pattern has abstracted the creation operation from the client code.

Implementation

The following implementation issues are relevant to the Natural Creation pattern.

1. *Gathering data from several objects.* Often, information from more than two objects is required to create and initialise a child object. Using the Natural Creation pattern, data can be gathered from other objects which collaborate with the Mother or Father objects, and this can be used to initialise the Child object.
2. *Arguments vs. Collaboration.* The Natural Creation pattern works best when most of the information required to create and initialise the child object come from the existing father and mother objects, and the objects with which they collaborate. Some kinds of objects, however, are primarily initialised with transient information from constants, local variables or arguments; in these cases the Natural Creation pattern provides less benefit. For example, creating a twelve-point bold italic font in Java by

   ```
   Font f = new Font("Times", BOLD|ITALIC, 12);
   ```

   does not require any information from the creating object or its collaborators, and the natural way to create the font is by calling the constructor directly.

3. *The class of the child object.* The class of the object to be created must be determined before the child can be created. If the class is always determined by the client, then it may be more natural to create the object directly by a language level mechanism. If the child's class is not immediately obvious, then one of the other creation patterns can be used to compute the class.
4. *Shallow vs. deep creation.* When creating and initialising an object based on collaborations with other objects in the program, the new object can be built in one of two ways: either by referring directly to the collaborating objects (shallow creation), or by making copies of the collaborating objects and using the copies (deep creation). As in shallow and deep copying [7] shallow creation is more efficient in both space and time, while deep creation ensures that the new object will be completely independent from the objects that create it.
   To be safe, new objects should share only *immutable* data — data that cannot change, and should *copy* any data that could change [1, 11]. As an example of what can go wrong, consider the following Smalltalk code which creates a Rectangle from two points using the `corner:` natural creation method.

   ```
   point1 := 10 @ 10.
   point2 := 20 @ 20.
   rectangle := point1 corner: point2.
   ```

   Now, if we change the x coordinate of the first point, the x coordinate of the rectangle will also be changed, because the `corner:` method uses shallow creation to make the rectangle so the rectangle refers directly to the two points.

   ```
   point1 x: 100.
   rectangle origin x == 100.      "returns true"
   rectangle origin == point1.     "returns true"
   ```

5. *User defined operators.* User defined operators in languages like Smalltalk and C++ can be used to implement the Natural Creation pattern. The meaning of user defined operators is less clear than the meaning of textual message names, so operators should only be used for creating instances of very common classes, or where their meaning is clear in the wider context. For example, Smalltalk's use of the `/` binary operator to define fractions is effective because the operator is used to represent fractions outside programming, fractions are quite common in Smalltalk programs, and the idiom is well known. Kent Beck calls these kind of operators "Shortcut Constructor Methods" and suggests that no more than three be used per application [2].

Acknowledgements

An early version of this paper was presented at TOOLS Pacific 1999. Many thanks to the anonymous reviewers for their comments on this paper, and to Liping Zhao, the KoalaPLOP 2000 shepherd for this paper, who designed the pattern form and insisted that I used it in this version of the paper, and provided comments on various drafts.

References

[1] Henry G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. *OOPS Messenger*, 4(4), October 1993.

[2] Kent Beck. *Smalltalk Best Practice Patterns.* Prentice-Hall, 1997.

[3] K.H.S. Campbell, J. McWhir, W.A. Ritchie, and I. Wilmut. Sheep cloned by nuclear transfer from a cultured cell line. *Nature*, 380:64–66, March 1996.

[4] Patrick Chan and Rosanna Lee. *The Java Class Libraries*, volume 2. Addison-Wesley, second edition, 1998.

[5] Ted Foster and Liping Zhao. Cascade. In *PLOP Proceedings*, 1997.

[6] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1994.

[7] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[8] Wilf Lalonde. *Discovering Smalltalk.* Benjamin/Cummings, 1994.

[9] James Noble. Patterns for finding objects within designs. In *TOOLS Pacific 25*, 1997.

[10] James Noble. Classifying relationships between object-oriented design patterns. In *Australian Software Engineering Conference (ASWEC)*, pages 98–107, 1998.

[11] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP Proceedings*, 1998.

[12] ParcPlace Systems. *VisualWorks Smalltalk User's Guide*, 2.0 edition, 1994.

[13] Eric Raymond and Guy L. Steele. *The New Hacker's Dictionary.* MIT Press, second edition, 1993.

[14] Dirk Riehle. Composite design patterns. In *ECOOP Proceedings*, 1997.

[15] Dirk Riehle. Product trader. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design*, volume 3. Addison-Wesley, 1998.

[16] Bobby Woolf. Understanding and using the ValueModel framework in VisualWorks Smalltalk. In *Pattern Languages of Program Design.* Addison-Wesley, 1994.

[17] Liping Zhao and Ted Forster. Driver duty constructor. *Journal of Object-Oriented Programming*, 12(2):45–51,77, May 1999.