Product Trader

Dirk Bäumer and Dirk Riehle

INTENT[1]

Let clients create objects by naming an abstract superclass and by providing a specification. A Product Trader decouples the client from the product and thereby eases the adaptation, configuration and evolution of class hierarchies, frameworks and applications.

ALSO KNOWN AS

Virtual Constructor, Late Creation

MOTIVATION

Suppose you have designed a class hierarchy of domain value types like AccountNumber, Amount, InterestRate or SocialSecurityNumber. These value types will be used in several applications. For example, one such application will present a form on the screen and let users edit its fields. The value types of the fields correspond to the domain value types. Since each value type has its own semantics and editing constraints, you might want to provide specialized widgets for each of these value types as shown in figure 1 and 2:

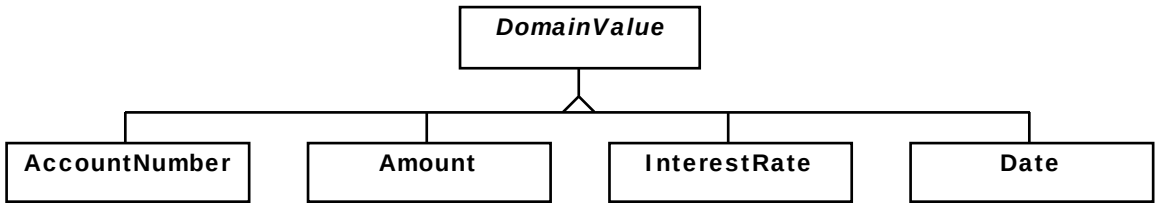

Figure 1: DomainValue class hierarchy

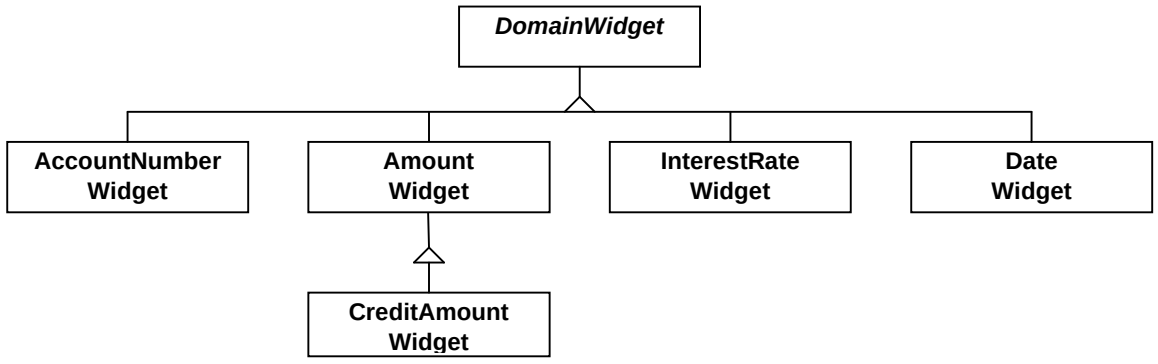

Figure 2: DomainWidget class hierarchy with specialized AmountWidget class

[1] In *Pattern Languages of Program Design 3.* Edited by Robert C. Martin, Dirk Riehle and Frank Buschmann. Reading, MA: Addison-Wesley, 1997. Chapter 3.

Given a Form object with all the fields and associated value types, how do you create the widgets corresponding to the value types?

You might be tempted to write a big case statement for all value types which creates a widget for a given value type, but this cumbersome and might require frequent changes and enhancements. You might use a Factory Method for the value types so that each value type can return a default widget for it. But then you will only be able to provide a single widget type for a value type. Moreover, if the value types are used in a non-interactive environment, like nightly batch runs, you make the batch application link the window system even though it isn't needed. You might also consider using an Abstract Factory. While this hides the creation process from the client, it doesn't clarify what happens behind the factory interface; you might still have to write a big case statement to distinguish between the different value types.

It is much better to go to some object and ask it to return an instance of a DomainWidget sub-class which fits a given value type best. This object might be the abstract DomainWidget class itself, or a factory object or a trader object. We call this pattern Product Trader and it is all about how to do this flexibly and efficiently.

Suppose, DomainWidget offers a class operation that lets you do this. In C++, it looks like

```
static DomainWidget* DomainWidget::createFor(DomainValue*);
```

and in Smalltalk it looks like

```
DomainWidget class>>createFor: aDomainValue.
```

Internally, the widget class might maintain a dictionary of all its subclasses which it indexes with the class representing a given value type. `createFor` can be easily implemented then: The DomainWidget class takes the value type's class as a key for the dictionary, which returns the widget subclass specifically designed for the value type. It then creates an instance of it which it returns to the client. This process is fast and requires only constant time.

The actual class which is to be instantiated is determined only at runtime, and it is accessed using abstract classes only. Thus, the concrete subclasses are hidden from the client. Moreover, the dictionary can be changed and reconfigured at runtime. This lets us easily configure and evolve class hierarchies, frameworks and applications.

APPLICABILITY

Use a Product Trader, if

- you want to make clients fully independent from concrete implementations of the abstract Product class (decoupling argument); or
- you want to dynamically select a product class according to selection criteria available only at runtime (dynamic selection argument); or
- you want to configure the kinds of product classes instantiated for a given selection criterion, either statically or at runtime (flexible configuration argument); or
- you want to change and evolve the product class hierarchy without affecting clients (evolution argument).

Do not use a Product Trader as a replacement for Factory Methods or direct object creation.

STRUCTURE

The structure diagram is shown in figure 3.

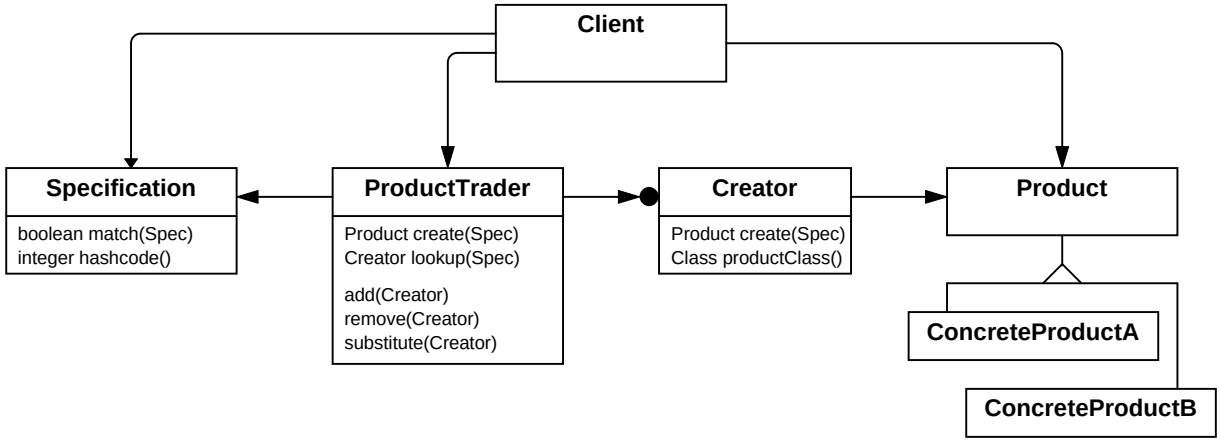

Figure 3: Structure diagram of the Product Trader pattern

PARTICIPANTS

- *Client (Form)*
  - creates a Specification for a ConcreteProduct class;
  - initiates the creation process by providing the Product Trader with a Specification.
- *Product (DomainWidget)*
  - defines the interface to a class hierarchy of which objects are to be instantiated.
- *ConcreteProduct (DateWidget, AmountWidget)*
  - represents a concrete Product class;
  - provides enough information to decide whether it matches a Specification.
- *Product Trader (DomainWidget class object)*
  - provides operations which let a client supply a Specification for a ConcreteProduct class;
  - maps a Specification onto a Creator for a ConcreteProduct class;
  - provides operations to configure the mapping by some configuration mechanism;
  - can be as simple as a hashtable and as complex as a full Object Trader.
- *Creator (ConcreteProduct class objects)*
  - defines the interface to create instances of a ConcreteProduct;
  - knows how to instantiate exactly one ConcreteProduct;
- *Specification (DomainValue subclasses)*
  - represents a specification of a ConcreteProduct class;
  - is used as the lookup argument for a Creator instance;

– can be as lightweight as a string and as heavyweight as a propositional calculus formula.

COLLABORATIONS

Figure 4 shows an interaction diagram of a Product selection and creation process. The following collaborations are involved:

- The Client asks the Product Trader for a Product instance matching a Specification.
- The Product Trader maintains a dictionary of Creator instances.
- The Product Trader looks up a Creator for a given Specification.
- The Product Trader delegates the creation process to a Creator.
- A Creator creates a ConcreteProduct instance.

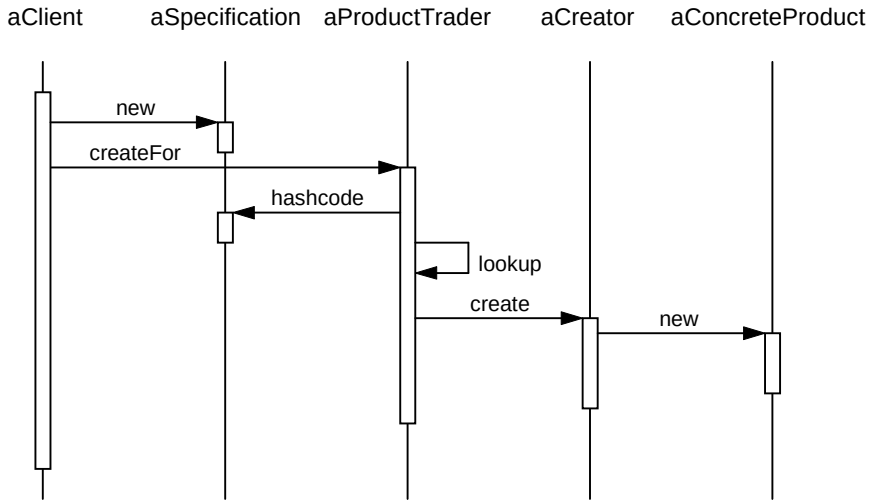

Figure 4: Interaction diagram of Product Trader

CONSEQUENCES

The Product Trader pattern has the following advantages and consequences:

- *Clients become independent of ConcreteProduct classes.* Clients are fully decoupled from the internal structure of the Product class hierarchy. Once you have written the code which creates the Specification instance and calls the createFor operation of the abstract Product class, you are done with the client code.
- *Product classes are determined at runtime.* Sometimes, the criteria which define the class fitting a task best are not known until runtime. Product Trader lets you turn these criteria into a Specification at runtime and use it for ConcreteProduct class lookup. Thus, you get the maximum flexibility for selecting ConcreteProduct classes.
- *Products can be configured for specific domains.* Product Trader lets you configure the available ConcreteProduct classes created via the abstract Product class by adding, substituting and removing Creators for ConcreteProduct classes. This lets you easily configure frameworks and applications for specific domains.

- *Product class hierarchies can be evolved easily.* Since clients are fully decoupled from the internals of the Product class hierarchy, class names, hierarchy structure, and implementation strategies of ConcreteProduct classes can be changed without affecting them (as long as the ConcreteProduct classes preserve the semantics of the abstract Product class).
- *New product classes can be introduced easily*. It is very easy to add new ConcreteProduct classes to an existing framework—it does not require any changes to the framework. New ConcreteProduct classes can be written without having to change existing code. Only configuration code or makefiles have to be changed.
- *Products need not be single classes but can be any complex component.* The Creator might hide potentially complex creation processes, for example by cloning complex Prototype structures.

The Product Trader pattern has the following disadvantages and liabilities:

- *Increased control and dependency complexity.* The pattern turns compile-time dependencies of product creation into runtime dependencies. This makes the resulting system harder to understand, because the dependencies are not explicitly written down as code.
- *Need for advanced configuration mechanisms.* When applying Product Trader, objects of classes can and will be created which are not statically referenced starting with some root class. Thus, a linker might consider those classes as superfluous and decide not to link them. See the Implementation section for techniques of dealing with this problem.
- *Ambiguous specifications.* Depending on the type and implementation of a specification, not only a single class can match it but rather a set of equivalent classes (with respect to the specification). This does not pose problems, since any class matching the requirements will do the job. However, ambiguities might indicate that the specification itself was either imprecise or underdetermined.
- *Special constructor parameters require overhead.* If a ConcreteProduct class requires special constructor parameters during the early object building process, you must supply them in advance. It requires some overhead to do so. You could broaden the creation interface or put the constructor parameters into a generic parameter carrier object. You should be careful, however, not to make any implicit assumptions about specific subclasses being selected by your Specification.

IMPLEMENTATION

The implementation of a Product Trader requires four main dimensions to be considered: the implementation of the mapping from Specification to Creator, the implementation of the Creator itself, the implementation of Specifications and configuration mechanisms.

- *Implementing the mapping from Specification to Creator.* The mapping can be realized by a simple dictionary in the abstract Product class, or by delegating the task to a product specific ProductTrader, or by delegating the task to a global ProductTrader.
  - *Using a simple dictionary* puts the management of the mapping into the abstract Product class. This is only advisable, if Product Trader is applied to the Product hierarchy only, and if the handling of specifications is very specialized for this particular class hierarchy. Otherwise consider using a ProductTrader.

- *Using a ProductTrader* takes the burden of managing specifications, creators, and the mapping from specifications to creators from the Product class. Each class hierarchy can have its own ProductTrader which deals with specifications for this class hierarchy. This solution is both flexible and offers good code reuse.
- *Using a single global ProductTrader* centralizes all specifications and all class semantics in one single place. This goes beyond the class hierarchy specific ProductTrader and should be applied if complex class retrieval and runtime configuration issues must be considered.

- *Implementing Creators.* Creators can be implemented as prototypes, class objects, or specific dedicated Creator objects, for example generated by C++ templates. You will want to avoid introducing a Creator class for every Product class by hand, so consider using one of these variants.
  - *Using prototypes or class objects* amounts to about the same: These objects come for free with any major system and usually provide a clone or a create operation which can carry out the actual creation of the Product instance. The clone operation must be robust with respect to copying.
  - *Using dedicated Creator objects.* In C++, you can use macros or templates to generate the Creator objects. We present a C++ template based version as a type-safe solution. An analysis of the Creator's responsibilities leads us to the following definition of a Creator and ConcreteCreator template:

```
template<class ProductType, class SpecType>
class Creator
{
public:
  Creator(SpecType aSpec) : _aSpecification(aSpec) {}
  SpecType getSpecification() { return _aSpecification; }
  ProductType * create() =0;

private:
  SpecType _aSpecification;
};

template<class ProductType, class ConcreteProductType, class SpecType>
class ConcreteCreator : public Creator<ProductType, SpecType>
{
public:
  ConcreteCreator(SpecType aSpec) : Creator<ProductType, SpecType>(aSpec) {}
  ProductType * create() { return new ConcreteProductType; }
}
```

The Creator template requires two formal arguments, the ProductType and the SpecType (specification type). Usually, the Specification itself is not maintained by the Creator, rather it is the argument for the mapping which leads to the Creator in the first place. However, for management purposes in ProductTraders it is convenient to store a specification with the Creator, given that there is only a single specification which leads to the specific Creator. Eventually, the ConcreteCreator template adds the code for actually creating an instance of a ConcreteProduct class.

A closer look at the templates reveals that they are fairly simple, essentially only defining interfaces and the create operation for concrete product classes. These templates can only be used for product classes where the constructor doesn't need any parameters. In case that parameters must be passed to the Product class constructor, additional templates are needed. A replication of the above two template classes is the result.

- *Implementing Specifications.* Specifications can be realized either directly in the parameters of the creation operation by some primitive value types, or by encapsulating them as classes of their own. This issue affects the implementation of the mapping:
  - *Using built-in value types* like integers, character strings, etc. often requires additional support like free floating operations to compute hash-codes, etc. Essentially, you must define operations which make the value types usable as lookup arguments for the mapping.
  - *Using explicit Specification objects* allows you to define the proper interface for specifications once and reuse it for every new specification. Moreover, ProductTraders can be written based on this explicit interface and reused as well. Such a Specification interface might look like this:

```
class Specification
{
public:
    // returns Product class, that is the root class of classes
    // addressed by this type of specification
  virtual Class* getProductClass() =0;

    // initialize specification to match semantics of a given Product class
  virtual void initForProduct(Class* pc) { this->adaptTo(pc); }

    // initialize specification to match semantics of a given Product class
  virtual void adaptTo(Class*);

    // match two specifications for equivalence
  virtual bool matches(Specification*) =0;
};
```

- *Configuring frameworks and applications.* Product Trader creates objects only indirectly by interpreting specifications. Thus, ConcreteProduct classes are not directly referenced by framework or application code, but only abstract interfaces like the Product class are. As a consequence, you must specify explicitly which classes are to be linked to an application, because linkers might not link classes which are not directly referenced by code reachable from some root object (or main).
  - *Using makefiles* is a straightforward approach. For every application, you explicitly specify the set of ConcreteProduct classes which are to be linked. You can do so by issuing commands to the linker. Since dependencies between ConcreteProduct classes from different class hierarchies might exist (covariant redefinition), be careful not to forget dependent classes.
  - *Using source code to configure systems* is straightforward as well: You might simply write a configuration operation in the system's root class, the main application class, or close to the main operation which references the classes required to be linked once. The linker will take care then that the right classes are linked to the final executable.
  - *Using register objects.* If you want to support Product Trader for all classes linked to the system you should consider using Register Objects. An object of a register class is responsible to add a ConcreteCreator object with a specification to the Product class.
  - *Using configuration scripts.* In large projects, where a lot of applications are developed, it is indispensable that the configuration of the system can be understood easily. When using Product Trader in C++, we recommend the use of configuration scripts. The goals of configuration scripts are:
    * to represent one place that describes the configuration of the system.

* to reference all needed ConcreteProduct classes, so that the linker includes them into the application.

An easy way to satisfy these goals is to create the ConcreteCreator and Specification objects inside a configuration script and add these objects to the Product class depending on the needed configuration of the system. The Sample Code section presents some examples.

- *Providing convenience operations.* You can use the abstract Product class to hide the implementation of a specification as an instance of a dedicated Specification class. The client simply calls the createFor operation with parameters which are native to the specification, and the implementation of createFor in the Product class converts these parameters into a Specification object.
- *Providing initialization parameters.* Sometimes objects require elaborate initialization parameters which must be available during the early object building process. These parameters can be packaged into the Specification objects, or they can be passed along a chain of operation calls which then must provide matching parameter lists (which asks for code generation as discussed above).
- *Consider dynamic link libraries explicitly.* Classes from dynamically linked libraries are not directly accessible, simply because they are not contained in the runtime image. However, the creator objects are usually very lightweight. They are therefore a fine means of representing not yet loaded classes. The creation procedure of a creator for a class in a dynamically linked library might have to load that class first, but this might be exactly what you want. Creators and the ProductTrader can be used effectively to hide the factoring of large applications into DLL's on the expense of having to deal with complex configurations issues.

SAMPLE CODE

Instantiating objects from a stream

As a first example, we will present a simple implementation using prototypes as Creators and a dictionary as the mapping from Specification to ConcreteCreator. The dictionary is maintained by the Product class.

Assume you are reading objects from a stream. A leading string indicates the class of the object to be read. Thus, the class name represents the specification for the class to be instantiated, and we use Product Trader to do so.

`Serializable` is the abstract Product class for all objects that can be read from a stream. `String` is the specification type, and we assume that every subclass of Serializable can provide a prototype.

The Serializable class might then implement `createFor` like this:

```
Serializable* Serializable::createFor(String& className) {
  Serializable* prototype = lookup(className);
  return prototype->clone();
}
```

Lookup simply retrieves the prototype for a given class name from a dictionary maintained by the Product class (a static member variable in C++ or a class variable in Smalltalk):

```
Serializable* Serializable::lookup(String& className) {
  return mapping[className];
}
```

A simple configuration scheme might be to make the prototypes register themselves at the `Serializable` class during system startup time (when the prototypes are created):

```
Customer::Customer(CustomerPrototypeConstructorIndicator* dummy) {
  Serializable::addCreator(this);
}
```

Serializable implements `addCreator` by using the class name to put the prototype into the dictionary:

```
Serializable::addCreator(Serializable* prototype) {
  mapping[prototype->getClassName()] = prototype;
}
```

This implementation uses prototypes as Creators, a simple built-in value type specification, and a static registering and configuration scheme. It is a rather lightweight approach which works well if Product Trader is to be applied to single class hierarchies with only simple specification requirements.

Configuring widgets for value types

As a second example we will use C++ templates to illustrate an implementation of the introducing domain widget and value problem. In this example, specifications are realized by C++ `typeid`'s denoting the class of the `DomainValue`. The configuration scheme of the system is realized by configurations scripts. The mapping from the Specification to the Creator is realized by a simple dictionary.

We use the templates from the Implementation section:

```
DomainWidget* DomainWidget::createFor(DomainValue* aValue) {
  Creator<DomainWidget, type_info&>* aCreator = mapping[typeid(aValue)];
  return aCreator->create();
}

DomainWidget::addCreator(Creator<DomainWidget, type_info&>* aCreator) {
  mapping[aCreator->getSpecification()] = aCreator;
}
```

We use the templates from the implementation section to configure the `DomainWidget` class. It maps the typeid of the DomainValues to the Creator object for the DomainWidgets:

```
StandardDomainWidgetConfiguration() {
  ...
    // map Amount on AmountWidget
  DomainWidget::addCreator(
    new ConcreteCreator<DomainWidget, AmountWidget, type_info&>(
      typeid(Amount)));
    // map Currency on CurrenyWidget
  DomainWidget::addCreator(
    new ConcreteCreator<DomainWidget, CurrencyWidget, type_info&>(
      typeid(Currency)));
  ...
}
```

If we want to offer a new configuration for an application we only have to write an new configuration script to make the required changes. As an example, consider a credit application which uses a special domain widget for the domain value `Amount`. To add the new widget to the application, you have to implement the new domain widget and then use a configuration script to register it. The new configuration script might look like this:

```
CreditConfiguration() {
  ...
    // use the specific CreditAmountWidget for the domain value Amount
  DomainWidget::substituteCreator(
    new ConcreteCreator<DomainWidget, CreditAmountWidget, type_info&>(
      typeid(Amount)));
  ...
}
```

The system's root object, or the main operation, must call the `DomainWidgetConfiguration` and the `CreditConfiguration` operation to install the necessary Creator objects. Only these calls are needed, no further code changes are required.

Using a general ProductTrader

As a final example, we will use Smalltalk to implement a full ProductTrader that can be reused by every class hierarchy that wants to provide Product Trader at its root for any number of different specifications. This example is derived from [Riehle+96] where a more detailed discussion can be found. This implementation uses a general ProductTrader which maintains a picture of the semantics of all classes in the system by means of specifications. The ProductTrader organizes these specifications so that it can easily lookup the classes matching a given specification.

Every Product class can provide convenience operations for Product Trader which are implemented in terms of the system-wide ProductTrader, a Singleton. Classes take over the role of Creators, Specifications are implemented as subclasses of a general Specification class, and the ProductTrader is realized by the ProductTrader just introduced.

The `ProductTrader` provides a mapping from a specification to a class object for every specification type. Specification types are represented by the classes in a `Specification` class hierarchy. The mapping is implemented as a dictionary which takes a Specification instance as a key and returns a collection of all classes that match the specification. Since the specification type is represented as a class, it can be used in a further dictionary to retrieve the dictionary containing the specification to classes mapping.

The ProductTrader has two main lookup methods:

```
ProductTrader>>getSpecificationDictionary: aClass
  "returns dictionary representing a specification to classes mapping"
  ( self specTypeDict includesKey: aClass )
    ifTrue: [ ^self specTypeDict at: aClass ]
    ifFalse: [ ^nil ]

ProductTrader>>getClassCollection: aSpecification
  "returns collection of classes matching aSpecification"
  | specDict |
  specDict := self getSpecificationDictionary: aSpecification class.
  ( specDict includesKey: aSpecification )
    ifTrue: [ ^self specDict at: aSpecification ]
    ifFalse: [ ^nil ]
```

The first method returns the mapping dictionary for a given specification type, and the second method returns a collection of all classes matching the given specification. The implementation of `createFor` becomes a simple two stage retrieval process now. A product class, for example `DomainWidget`, simply does the following:

```
DomainWidget>>createFor: aDomainValue
  "create Specification instance and delegate to ProductTrader"
  | aSpecification |
  aSpecification := DomainWidgetSpecification new: aDomainValue class.
  ^self ProductTrader createFor: aSpecification

ProductTrader>>createFor: aSpecification
  "create object for given specification"
  | classCol |
  classCol := getClassCollection: aSpecification.
  ( classCol notNil )
    ifTrue: [ ^classCol first new ]
    ifFalse: [ ^nil ]
```

The actual problem is how to set up the dictionaries so that the retrieval and creation process can be carried out easily. This is a two step process. In the first step, the ProductTrader collects all classes from the Specification class hierarchy. In the second step, it calculates the mapping for every Specification class; this is done by traversing the product class hierarchy as indicated by the Specification class, and by creating a Specification instance for every Product class.

This process is based on information which has to be provided by the specification classes themselves. Thus, Specification declares the following class and instance methods:

```
Specification class>>getProductClass
  "returns Product class"
  self subclassResponsibility

Specification class>>newFromClient: anObject
  "used by clients which supply the specification information"
  self subclassResponsibility

Specification class>>newFromManager: aClass
  "used by ProductTrader to instantiate spec for a ConcreteProduct class”
  ^super new adaptTo: aClass

Specification>>adaptTo: aClass
  "adapts spec instance to given product class"
  self subclassResponsibility

Specification>>matches: aSpecification
  "matches with another specification"
  ^( self class = aSpecification class )
```

These methods must be overwritten by subclasses in order to fit a certain specification's semantics. For `DomainWidgetSpecification`, this leads to the following code:

```
DomainWidgetSpecification class>>getProductClass
  ^DomainWidget

DomainWidgetSpecification class>>newFromClient: aDomainValueClass
  self domainValueClass: aDomainValueClass

DomainWidgetSpecification>>adaptTo: aDomainWidgetClass
  self domainValueClass: aDomainWidgetClass getDomainValueClass

DomainWidgetSpecification>>matches: aSpecification
  ( super matches: aSpecification ) ifFalse: [ ^false ].
  ^( self domainValueClass = aSpecification domainValueClass )

DomainWidgetSpecification>>hash
```

```
  ^domainValueClass symbol
```

When created by a client or a convenience operation of DomainWidget, the DomainWidgetSpecification instance receives the `DomainValue` class the new DomainWidget instance will have to fit. When created by the ProductTrader, the DomainWidgetSpecification instance is created for a specific DomainWidget class which it asks for the DomainValue class it has been written for. The last method is used to build the mapping of domain values to domain widgets. The ProductTrader traverses the DomainWidget class hierarchy, creates a DomainWidgetSpecification instance for each class, and puts the class into the mapping dictionary with the specification instance as the key. The hash method then maps a client created DomainWidgetSpecification instance on a collection of DomainWidget classes.

This approach is based on the assumption that the specification instance is able to retrieve enough information from the Product class to serve as a key in the mapping. In our example, the `DomainWidget` class provides a `getDomainValueClass` method which returns the domain value class. If this is not an option, the class semantics of Product classes must be specified external to that class, for example in a database.

This approach lets us introduce new specifications simply by introducing a new specification subclass. Implementing Product Trader for object streaming as shown in the first example can be done by creating a `ClassSymbolSpecification` which looks like this:

```
ClassSymbolSpecification class>>getProductClass
  ^Object

ClassSymbolSpecification class>>newFromClient: aSymbol
  self classSymbol: aSymbol

ClassSymbolSpecification>>adaptTo: aClass
  self classSymbol: aClass symbol

ClassSymbolSpecification>>matches: aSpecification
  ( super matches: aSpecification ) ifFalse: [ ^false ].
  ^( self classSymbol = aSpecification classSymbol )

ClassSymbolSpecification>>hash
  ^classSymbol
```

The general ProductTrader is a mighty approach for avoiding the hard-coding of creation dependencies in the class implementations themselves. It has served us well in making our frameworks easier adaptable and configurable.

KNOWN USES

The first example in the Sample Code section, creating objects from a stream, can be found in almost every application framework which provides streaming support. It requires the mapping from some class identifier, for example the class name, to an object which is capable of instantiating this class. You can implement the specification as a string representing the class name, the creator objects by some class object or prototype, and the mapping as a dictionary which maps the class name on the object representing the class. For example, the ClassManager of ET++ [Weinand+94] implements Product Trader by maintaining class objects in a dictionary, and so do the Tools and Materials Metaphor Frameworks [Riehle+95, Riehle+96].

Next to this very frequent but specialized application, Product Trader is often used to select a subclass as a specific implementation of the abstract Product class which fulfills certain criteria,

for example performs very fast or uses only a limited amount of memory resources. Lortz and Shin [Lortz+94] implement Product Trader according to Coplien's generic autonomous exemplar idiom [Coplien92]: The Product class maintains a list of prototypes for each of its subclasses; it uses strings as a specification mechanism; and it traverses the list matching each prototype with the specification. Lortz and Shin use Product Trader to select container classes based on performance requirements in the context of a real-time database system.

The most interesting application of Product Trader, however, is the creation of objects from one hierarchy depending on objects from another hierarchy, as demonstrated by the introducing example. In the Tools and Materials Metaphor frameworks [Bäumer+97, Riehle+96, Riehle+95], we have made extensive use of the pattern to map hierarchies onto each other. We instantiate widgets depending on value types, we instantiate user interface parts depending on domain models, we instantiate tools depending on the materials they have to work on, and we instantiate domain components depending on other domain components.

All these tasks are carried out on the level of abstract classes. This helped us achieve a degree of decoupling that significantly eased system evolution and configuration. Product Trader is an important asset in making the frameworks consisting of more than 2000 classes adaptable, configurable and, eventually, manageable.

RELATED PATTERNS

The global ProductTrader is a Singleton [Gamma+95]; Product Trader is used to create objects from a stream in the Atomizer pattern [Riehle+97]; Convenience operations (used to encapsulate complex Specification objects) are an example of the Convenience Method pattern [Hir96]; Product Trader is used to create ConcreteExtension objects for ConcreteSubject objects in the Extension Objects pattern [Gamma97].

Product Trader serves as a companion to Factory Method [Gamma+95]. Product Trader works well where Factory Method works poorly and vice versa. Factory Methods often introduce cyclic dependencies between the Product and the Creator, in particular if the Product is going to work on the Creator later on. This is the case with applying Factory Method in the introductory example where a value object would create a widget object. This is the case with a container creating its iterators. This is the case with a model object creating its views. If a different product is required, the creator has to be changed. The widget class, the container class and the model class would have to be changed. Furthermore, no two different products can coexist and be selected due to some dynamic selection criteria.

We think of Product Trader as a third fundamental creational pattern next to Factory Method and Prototype.

ACKNOWLEDGMENTS

We wish to thank our shepherd and the participants of the writer's workshop at PLoP '96 for their helpful comments.

REFERENCES

[Bäumer+97] D. Bäumer, G. Gryczan, R. Knoll, C. Lilienthal, C. Riehle, and H. Züllighoven. "The Tools and Materials Metaphor Series of Frameworks." *Submitted for publication.*

[Coplien92] J. O. Coplien. *Advanced C++: Programming Styles and Idioms.* Reading, MA: Addison-Wesley, 1992.

[Gamma97] E. Gamma. "Facet." *This volume.*

[Gamma+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Design.* Reading, MA: Addison-Wesley, 1995.

[Lortz+94] V. B. Lortz and K. G. Shin. "Combining Contracts and Exemplar-Based Programming for Class Hiding and Customization." In *Proceedings of OOPSLA '94, ACM SIGPLAN Notices* 29, 10 (October 1994): Page 453-467.

[Riehle+95] D. Riehle and H. Züllighoven. "A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor." In J. O. Coplien and D. C. Schmidt (eds), *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, 1995. Page 9-42.

[Riehle+96] Dirk Riehle, Bruno Schäffer, and Martin Schnyder. "Design of a Smalltalk Framework for the Tools and Material Metaphor." *Informatik/Informatique* (February 1996). Page 20-22.

[Riehle+97] Dirk Riehle, Wolf Siberski, Dirk Bäumer, Daniel Megert, Heinz Züllighoven. "Serializer." *This volume.*

[Weinand+94] André Weinand and Erich Gamma. "ET++ — a Portable, Homogenous Class Library and Application Framework." In W. R. Bischofberger and H.-P. Frei (eds.), *Proceedings of the Ubilab Conference '94, Zürich.* Konstanz: Universitätsverlag Konstanz, 1994. Page 66-92.

Copyright © 1997 Dirk Bäumer and Dirk Riehle. All Rights Reserved.

Dirk Bäumer works for RWG GmbH, Germany. He can be reached at RWG GmbH, Räpplenstraße 17, 70191 Stuttgart, Germany. He is interested in object-oriented frameworks, software architecture and distributed systems. He welcomes e-mail at Dirk_Baeumer@rwg.e-mail.com or baeumer@informatik.uni-hamburg.de.

Dirk Riehle works at Ubilab, the information technology research laboratory of Union Bank of Switzerland. He is interested in object-oriented frameworks, software architecture and distributed systems. He can be reached at Union Bank of Switzerland, Bahnhofstrasse 45, CH-8021 Zürich. He welcomes e-mail feedback at Dirk.Riehle@ubs.com or riehle@acm.org.