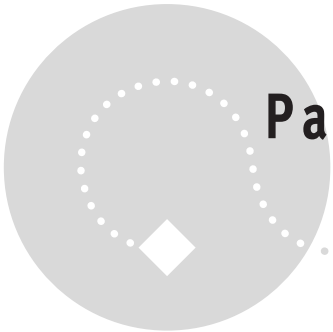


Part 1

Archetype theory, practice, and Model Driven Architecture

Chapter 1
Archetypes and archetype patterns

1.1 Introduction

In this chapter, we introduce and explain the concepts of business archetypes and archetype patterns. We have found these concepts to be exciting and very useful in our own work in object modeling, and we hope that you will too.

Although an understanding of archetypes and archetype patterns was essential to create the archetype patterns presented in the main part of this book, it is not essential to the pragmatic application of these patterns in business systems. If you just want to use this book as a useful pattern catalog, you may safely skim this chapter, which is mainly theoretical. However, you should at least take a quick look at Section 1.7 to understand the Unified Modeling Language (UML) profile we are using.

If you are involved in capturing business patterns or creating high-level or even enterprise-level object models, you may find the "thought tools" presented in this chapter to be very valuable.

We begin with a general discussion of archetypes (Section 1.2), define what we mean by business archetypes and archetype patterns (Section 1.3), and discuss how we can model archetypes and archetype patterns using UML (Section 1.7). We look at the issue of pattern variation (Section 1.9) and introduce the powerful notion of pleomorphism as a way to understand how archetypes and archetype patterns adapt to specific business environments (Section 1.12).

1.2 What are archetypes?

The word *archetype* comes from the Greek *archetypo* (αρχετυπο), which means "original pattern." Here is a definition.

An archetype is a primordial thing or circumstance that recurs consistently and is thought to be a universal concept or situation.

According to the psychologist Carl Gustav Jung [Jung 1981], archetypes arise from a common fund of human experiences (the collective unconscious) that uses archetypes as one of its ordering and structuring principles. In fact, wherever there is a commonality of human experiences over extended periods of time, archetypes arise to help structure these experiences.

One of the most intriguing aspects of the Jungian archetypes is that they naturally exhibit *variability*—they change their form to adapt themselves to specific cultural contexts while their core semantics remain fixed. For example, the Hero archetype looks very different in the Native American paradigm than in the Australian Aboriginal paradigm, and yet the Hero is still somehow always recognizable as the Hero. We'll see shortly that this natural variability is an important feature of archetypes.

Because archetypes are a basic human mechanism for organizing, summarizing, and generalizing information about the world, you can reasonably expect them to have some application in the field of software development.

Human beings have been involved in business activities for millennia, and we think it is quite reasonable to suppose that many archetypes have arisen in the business domain. For example, if you think about the basic business activity of selling, the earliest recorded instances of this activity occurred some 5,000 years ago. There's no doubt that this activity was also occurring much earlier than this. All selling over this enormous span of time has in some way involved the basic concepts of product, price (in terms of a notion of the value of the product), seller, and buyer.

You can see that there are some very fundamental (we would say archetypal) concepts here and that there is also an archetypal pattern of relationships between these concepts. For example, the price is always associated in some way with the product.

The agenda of this book is to try to capture some of these archetypes and archetype patterns in UML object models. To do so, we introduce the following new concepts.

1. Business archetypes
2. Business archetype patterns
3. Archetype and archetype pattern variability
4. Pleomorphism
5. Pattern configuration

We'll discuss the first four of these ideas in the next few sections, and we devote much of Chapter 2 to pattern configuration.

1.3 What are business archetypes?

Object-oriented (OO) software systems reflect the business domains in which they operate. You can therefore expect to find archetypes in the business domain, in software systems, and in models of those systems. We call this type of archetype a *business archetype*.

A business archetype is a primordial thing that occurs consistently and universally in business domains and business software systems.

A good example of a business archetype is `Party`. A `Party` represents an identifiable, addressable unit that may have a legal status. Usually this represents a person or an organization of some sort. All business systems have some concept of `Party`. You can look at the actual definition and semantics of the `Party` archetype in Chapter 4.

The notion of archetypes is very general, and there are certainly archetypes in other domains (such as health care and engineering) as well as in the business domain. You may use the term `<domain name> archetype` to refer specifically to these other archetypes. However, in this book, we limit ourselves to the business domain.

As well as there being archetypal things in business systems, these things can interact in patterns that are themselves archetypal. For example, the collaboration between the archetypes `Party`, `Product`, and `Order` is the basis of virtually every business that sells goods or services. We refer to these archetypal collaborations as *business archetype patterns*.

A business archetype pattern is a collaboration between business archetypes that occurs consistently and universally in business environments and software systems.

In this book, for convenience we usually refer to business archetypes and business archetype patterns simply as archetypes and archetype patterns.

The essential characteristics of archetypes and archetype patterns are listed below.

- Universal: for something to be archetypal, it must occur consistently in business domains and systems.
- Pervasive: they occur in both the business domain *and* the software domain. When building OO systems, you should expect to find things and patterns that are archetypal in the business domain occurring in much the same form in the software domain. This is the principle of convergent engineering described in [Taylor 1995] and more recently in [Hubert 2001].
- Deep history: for example, the product archetype has been around ever since people first began to barter and sell.
- Self-evident to domain experts: this is not always the case, but if an archetype isn't obvious to a domain expert, you should certainly question whether it is really an archetype.

A quick word about terminology: the term *archetype* has been used in the context of computing by other authors. Peter Coad and his colleagues define *archetype* as "a form from which all things of the same kind more or less follow" [Coad 1999, p. 2]. Coad uses his archetypes in a way that is in some respects similar to how we use ours, but the Coad archetypes occur at a *much* higher level of abstraction and lack any formal UML profile.

Mellor and Balcer define *archetype* as "a fragment of data access and text manipulation logic that states formally how to embed an executable UML model into text" [Mellor 2002, p. 294]. In other words, the term is used to de-

scribe a specific aspect of an executable UML model. This is very different from any dictionary definition of the term.

Both of these uses of the word *archetype* are different from our usage in the term *business archetype*. Generally, whenever we use the term *archetype* in this book, we are using it as a shorthand for *business archetype* unless we explicitly state otherwise.

1.4 Archetypes and analysis classes

In object modeling, there are two fundamentally different types of classes, as shown in Table 1.1.

Table 1.1

Type of class	Semantics
Analysis class	Represents a crisp abstraction in the problem domain Maps onto real-world business concepts
Design class	A class whose specification is complete to such a degree that it may be implemented Incorporates features from both the problem domain and the solution domain (implementation technology)

We discuss both types of classes in much more detail in an earlier book [Arlow 2001], so we won't repeat that detailed discussion here. But to summarize, an analysis class arises directly from the problem domain (e.g., selling furniture) and has no implementation-specific features. On the other hand, a design class may contain features from both the problem domain and the solution domain (e.g., J2EE, .NET, or Web services). Analysis classes are for understanding the business, while design classes are for understanding the technical solution.

It's important to realize that archetypes are *always* at a higher level of abstraction than normal analysis classes. From a conceptual point of view, this is because archetypes are about consciously recognizing and capturing *universal* concepts, whereas analysis classes are not necessarily concerned with universality at all. From a technical point of view (as you will soon see), archetypes *generate* one or more analysis classes.

1.5 What are patterns?

We'll give a very brief introduction to patterns in this section, but for more details we advise you to refer to the key text on patterns, *Design Patterns—Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides [Gamma 1995].

According to [Gamma 1995], a pattern is a solution to a problem in a context. To be more precise, a pattern consists of a description of a problem, the context of the problem, and a possible solution to that problem in that context. You can think of a pattern as a "recipe" that describes how you may solve a particular problem under particular circumstances.

The idea of patterns originated in the work of Christopher Alexander on the architecture of towns and buildings [Alexander 1977]. Alexander said that a pattern describes a recurring problem and the key elements of the solution to that problem in a way that allows you to apply the solution again and again, each time in a novel but consistent fashion.

Gamma and his colleagues applied this idea to software systems in their book. Each pattern they defined has four elements.

- Pattern name: the name of the pattern. This allows you to talk about the pattern without having to always describe its details. Pattern names define a language that allows designers to communicate about designs at a high level of abstraction.
- Problem: the description of the problem that the pattern solves, for example, how to design an object that may have only a single instance (the Singleton pattern).
- Solution: the design of the pattern itself as a UML model. This design doesn't describe actual classes but rather a collaboration that classes in your model may implement.
- Consequences: the effects of applying the pattern.

Patterns can exist at many different levels of abstraction. [Gamma 1995] describes design patterns that are possible solutions to common problems encountered in OO design. It should be an essential component of your OO designer toolset!

Fowler has extended the idea of patterns into the analysis domain in his book, *Analysis Patterns—Reusable Object Models* [Fowler 1996]. This book contains some interesting patterns, but they are generally quite abstract and need a lot of refinement before you can apply them in a real development situation.

We compare and contrast archetype patterns with analysis patterns in the next section.

1.6 Archetype patterns and analysis patterns

This section is for those of you who want to know how archetype patterns and analysis patterns differ. If you are not interested in this topic, you can safely skip this section.

When we've presented material on archetype patterns at conferences prior to publication, we've sometimes been asked, "Aren't archetype patterns just analysis patterns?" The answer to this is no—archetype patterns have many unique features that make them much *more* than analysis patterns.

As we mentioned above, analysis patterns were first described by Fowler and defined as follows: "Analysis patterns are groups of concepts that represent a common construction in business modeling. It may be relevant to one domain, or it may span many domains" [Fowler 1996, p. 8].

Notice that there is no notion of *archetypal* concepts in this definition. This is the primary conceptual difference between analysis patterns and archetype patterns.

In fact, archetype patterns are much richer than analysis patterns, both conceptually and in terms of technology. These differences are summarized in Table 1.2.

Table 1.2

Feature	Reference	Archetype patterns	Analysis patterns
Is concerned with archetypal concepts	Chapter 1	Always	Sometimes
Incorporates the principle of convergent engineering	[Taylor 1995]	Always	Often
Is supported by a UML profile	Section 1.7	Yes	No
Is sufficiently detailed to feed into the Model Driven Architecture (MDA) development workflow as a platform-independent model (PIM)	Chapter 2	Yes	No
Supports variability of model elements	Section 1.9	Yes	No

Table continued on next page

Table 1.2 (Continued)

Feature	Reference	Archetype patterns	Analysis patterns
Supports pleomorphism	Section 1.12	Yes	No
Introduces pattern configuration rules to support pattern configuration	Section 2.5	Yes	No
Defines a set of platform-independent models	Section 2.3	Yes	No
Applicable across different business domains	Chapter 1	Often	Often
Supplied as literate models	Chapter 3	Yes	No
May be automated using MDA modeling tools	Chapter 2	Yes	No

As you will see in Section 1.15 and in Chapter 2, business archetype patterns are also applied in a very different way than analysis patterns.

1.7 UML profile for archetypes and archetype patterns

You can introduce new modeling extensions into UML by defining a UML profile. This consists of a set of stereotypes, tagged values, and constraints that define the semantics for the new modeling extensions you want to introduce. A profile extends the UML metamodel with a set of new modeling elements.

Our UML profile for archetype patterns is defined in Table 1.3. We've tried to keep it as simple as possible.

Table 1.3

Business archetype UML profile		
Stereotype	**Applies to**	**Semantics**
«archetype»	Class	A primordial thing that occurs consistently and universally in business environments and business software systems All archetypes in a business archetype pattern are optional

Table 1.3 (Continued)

Business archetype UML profile		
Stereotype	**Applies to**	**Semantics**
«archetype pattern»	Collaboration Package	A collaboration between business archetypes that occurs consistently and universally in business environments and software systems
«archetype pattern library»	Package	A subtype of the standard UML stereotype «model» The package is a model that contains one or more archetype patterns You should ensure that each archetype pattern library has a globally unique name to avoid namespace clashes—we recommend that you use your domain name as the name of the package, e.g., "clearviewtraining.com"
«o»	Composition Aggregation Attribute Operation	A feature that is optional and may be omitted When «o» is applied to a composition or aggregation relationship, it indicates that the relationship is optional
«pleomorph»	Refinement relationship between archetype patterns	The archetype pattern at the source of the arrow is a variation (pleomorph) of the archetype pattern pointed to by the arrow We discuss pleomorphism in detail in Section 1.12

These stereotypes may be used as illustrated in the following sections.

1.7.1 «archetype»

You can model business archetypes by using the class icon and adding the stereotype «archetype» to indicate that the classifier represents an archetype. Figure 1.1 shows a simple example.

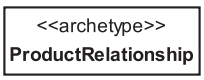


Figure 1.1

By definition, archetypes are optional and can, if wished, be omitted from any model based on the archetype pattern.

1.7.2 «archetype pattern»

You may use package notation for an archetype pattern as shown in Figure 1.2. You may also use collaboration notation (see Figure 1.17 later in this chapter).

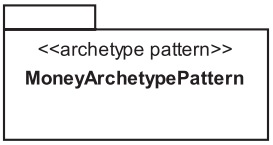


Figure 1.2

1.7.3 «archetype pattern library»

You can model an archetype pattern library as shown in Figure 1.3.

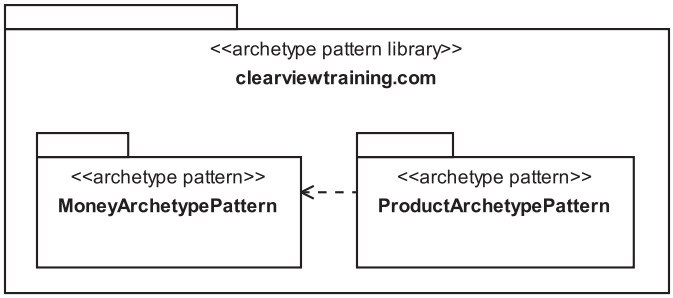


Figure 1.3

1.7.4 «o»

You can use the stereotype «o» (optional) to specify the parts of an archetype or archetype pattern that are optional. We'll discuss the reasons why we need this new stereotype in detail in Sections 1.9 and 1.10.

Figure 1.4 shows examples of the «o» stereotype being used on attributes, operations, and relationships.

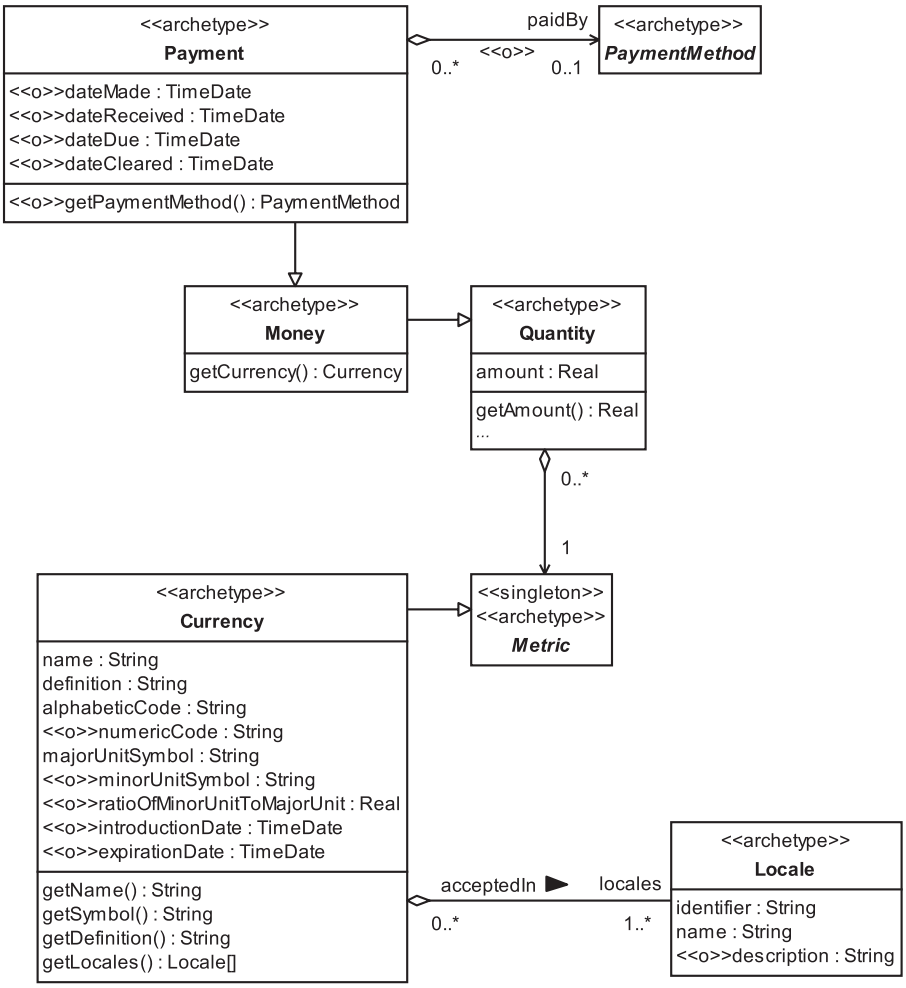


Figure 1.4

Reading Figure 1.4, you can see the following information.

- The `Currency` attributes `numericCode`, `minorUnitSymbol`, `ratioOfMinorUnitToMajorUnit`, `introductionDate`, and `expirationDate` are optional.
- All the `Payment` attributes are optional. This gives you a lot of flexibility in how the archetype can be used in different business contexts. For example, a system that makes `Payments` would usually need the optional

attribute `dateMade` but not the others. However, a system that accepts `Payments` would not need `dateMade` but would include one or more of `dateReceived`, `dateDue`, and `dateCleared`. We discuss this use of `Payment` more fully in Section 11.9.

- The relationship `paidBy` between `Payment` and `PaymentMethod` is optional, and thus the `Payment` operation `getPaymentMethod()` is optional.

Note that the «`singleton`» stereotype on `Metric` simply indicates that there needs to be only a single instance of the `Metric` archetype at runtime. The Singleton pattern is described in [Gamma 1995].

1.7.5 «pleomorph»

This stereotype may be applied to refinement relationships between archetype patterns as shown in Figure 1.5. The archetype pattern at the source of the arrow is a variation of the archetype pattern pointed to by the arrow. In the figure the `IdenticalProduct` archetype pattern is a variation of the `Product` archetype pattern for a specific business domain. We discuss pleomorphism in detail in Section 1.12.

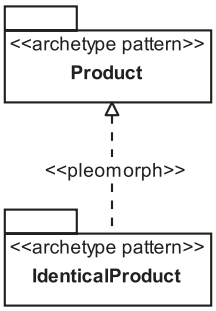


Figure 1.5

1.8 Modeling style

Whenever you create a UML model, it's a good idea to define a modeling style that you then use consistently throughout the model.

The modeling style used to create the models you see in this book arises from the specific requirements listed below.

- Make the models as readable as possible.
- Make the models as useful as possible.
- Make the models as precise as possible.
- Make the diagrams fit harmoniously within the bounds of this book.

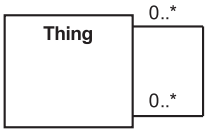
We have adopted the following modeling style to satisfy these requirements.

- We usually don't show `set` and `get` methods for attributes. You may assume they are there unless an attribute is explicitly marked as private. This modeling style is described in *Convergent Architecture* [Hubert 2001] as the Compact Attribute style. It saves a lot of space on UML diagrams!
- As we described in our UML profile for archetypes in Section 1.7, we indicate that an attribute, operation, composition, or aggregation relationship is truly optional by using the stereotype «o».
- Archetypes are always optional.
- Everything that is not explicitly optional is mandatory.
- We show navigability wherever we can—this reduces the coupling between modeling elements, so we always try to put the maximum amount of navigability on our diagrams.
- We always show multiplicity explicitly. Some modelers assume that when multiplicity is not shown, it automatically defaults to 1, but this is a false assumption—when multiplicity is not shown, it means that it is *undecided*.
- We try to refine each association relationship as much as we can. This means that we use aggregation and composition wherever possible. Aggregation and composition have very specific semantics (see [Arlow 2001] and www.businessarchetypes.com) that are very useful in the models we create.

The goal of our modeling style is to try to create models that are as precise and constrained as it is possible to make them while still maintaining their generality and readability.

Figure 1.6 shows a completely general purpose UML model. This model is so *un*constrained as to be totally meaningless. (Oddly enough, in our consulting work, we do occasionally come across UML models somewhat like this

one!) In fact, UML models become *more* meaningful the larger the number of constraints you can apply. This is because constraints capture information.



A general purpose model

Figure 1.6

Finally, we are always guided by what we refer to as "the principle of maximum utility"—we consciously strive to make the diagrams and text as useful to you in every respect as possible. Our ultimate aim is to try to make the diagrams talk to you about the business domain.

1.8.1 OCL data types

Ultimately, all models are built up from combinations of a relatively small number of basic data types such as `int`, `float`, `double`, `String`, and so on. In order to make our archetype patterns as universal as possible, we take our set of basic types from the Object Constraint Language (OCL), which is a part of UML that provides a formal language for expressing constraints. OCL defines a set of predefined types that allow us to express business archetype patterns as completely language- and platform-independent UML models.

The advantage of using the OCL types instead of (for example) Java or .NET data types is that the OCL types can be very easily mapped onto those of other languages. This can be done manually when you instantiate an archetype pattern or completely automatically if you are lucky enough to be using an MDA-enabled modeling tool.

The types are summarized in Table 1.4.

Table 1.4

OCL type	Semantics
`Real`	Represents the mathematical concept of a real number
`Integer`	Represents the mathematical concept of a whole number

Table 1.4 (Continued)

OCL type	Semantics
`String`	Represents an ASCII string of characters Although OCL specifies an ASCII string, you should assume that `String` in our models represents a Unicode string so that archetype patterns may be used internationally—this is our only departure from OCL
`Boolean`	Represents a value that is `true` or `false`

These types have exactly the sort of operations (+, –, /, and so on) that you might expect. You can find the full details in the UML specification (www.omg.org/uml).

We add `TimeDate` to this set of OCL types. This represents a point in time as defined in ISO 8601. You can assume that `TimeDate` provides a set of operations for performing calculations on time as well as comparison operations. Most programming languages provide a type or library component that maps onto `TimeDate`, so we don't provide any more details here.

1.9 Variation

One of the unique aspects of the archetype pattern approach is that it explicitly addresses the problem of pattern variation.

Sometimes a specific model of something, such as a model of products, may be suitable for use in one business area but not in another. This is what we refer to as *the principle of variation:* different business domains often seem to require different models of the same thing.

This principle just seems to be a fact of life. Often there is no way around it even if you choose to make some modeling compromises.

Because of variation, the construction of generic, highly reusable object models, such as enterprise object models, has proven to be rather difficult. You may even have heard some pundits say that such activities have failed and are, in principle, impossible.

However, in our experience *you can succeed* at such activities, and we'll tell you how.

Although you can't usually make variation go away, there is always another option. By carefully analyzing and understanding the variation, you can work with it constructively to create archetype patterns that are adaptable and that can change their form to adapt themselves to different business contexts.

The first step in understanding variation is to look at the types of variation possible in archetype patterns. You will find that there are three different kinds of variation.

1. Archetype variation: archetypes may need *different features* (attributes, operations, constraints) to be effective in different business contexts.
2. Archetype pattern variation: optional features in the patterns may be omitted if they are not needed.
3. Pleomorphism: in this special type of archetype pattern variation, the pattern may take on a *different structure* to adapt itself to the specific requirements of a business context. This may mean different archetypes, archetype features, and relationships in each of the variants.

We will look at these types of variation in detail in the next three sections.

1.10 Archetype variation and optionality

This type of variation occurs when an archetype needs to be adapted to a particular business use. There are really only two ways in which an archetype may be varied.

1. Some new features may be added.
2. Optional features may be omitted.

The key to dealing with this kind of variation successfully is to ensure that the core semantics of the archetype *remain fixed* for every variant you create. You therefore need a way to show which parts of the archetype are optional and can be omitted from variants derived from the archetype.

In our models, we indicate that a feature is optional by using the stereotype «o». When we mark an attribute (or operation) as optional, this means that the feature may be *omitted entirely* from the class.

You can see how we indicate variability in the `PartySignature` archetype shown in Figure 1.7. The attribute `reason` is optional and so is the operation `getAuthentication()`. The attribute `when` is not explicitly marked as optional, so it is mandatory. The `PartySignature` archetype itself is optional within the context of the `PartyArchetypePattern` (in which it appears) because all archetypes are optional by default.

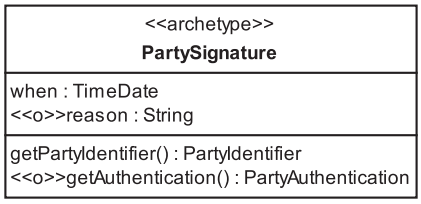


Figure 1.7

To understand optionality fully, consider the example shown in Figure 1.8, which shows a simple archetype pattern that contains only a single archetype, `A`. This archetype has one mandatory feature (attribute `a1`) and two optional features (attribute `a2` and operation `o1()`). When this archetype pattern is instantiated in one of your models, there are four possible ways to make this instantiation, as we show in the figure.

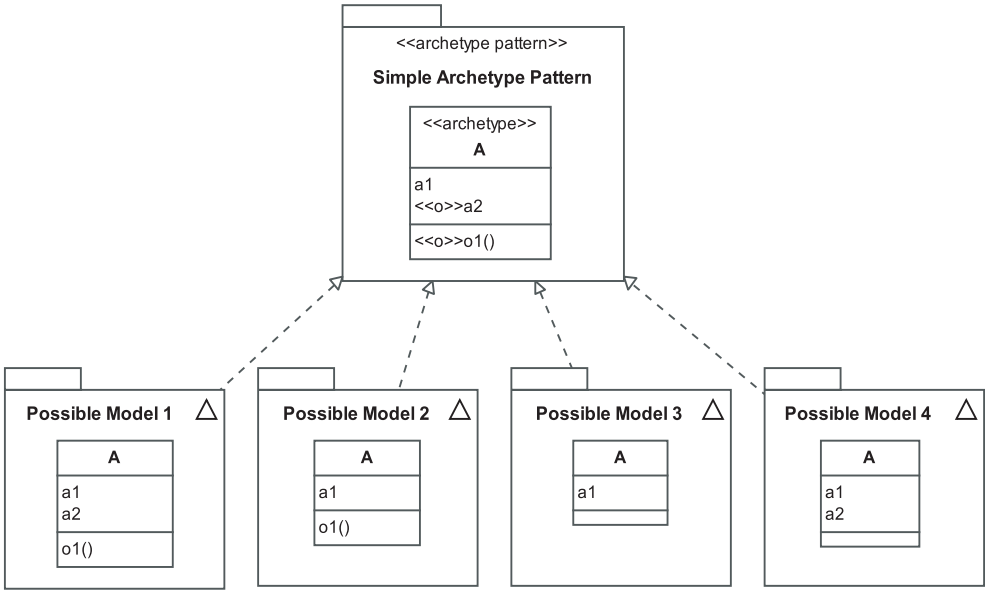


Figure 1.8

We assume that you make the simplest possible instantiation of the pattern—you just turn the archetype into a class in your model. You can instantiate the pattern manually by copying the pattern into your model yourself, or

semi-automatically by using a suitably equipped MDA modeling tool (described in detail in Chapter 2). Notice that the optional elements may be *absent entirely from an instantiated pattern.*

This idea of optionality is very important because it allows archetypes (and archetype patterns) to be configurable. Pattern configuration is one of the major topics in Chapter 2.

It's worth noting that UML also has a notion of optionality, but this is at the instance level, rather than at the class level. UML provides a syntax to indicate that a class attribute can take the value `null` in instances of that class. You can do this by appending the multiplicity `[0..1]` to the attribute as shown in Figure 1.9.

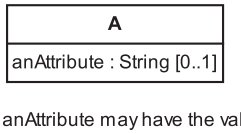


Figure 1.9

However, note that the slot for the attribute *still exists* in an instance even when it holds the value `null`. This is very different from the attribute being truly optional.

1.11 Archetype pattern variation

As described in Section 1.7, any feature of an archetype pattern that is marked with the «o» stereotype, and anything stereotyped «`archetype`», is optional.

For example, Figure 1.10 shows the complete `Money` archetype pattern, which is fully described in Chapter 11.

Suppose that you need to use only part of this pattern. You are concerned with `Money` but *not* with payments or currency exchange. Furthermore, suppose that you are interested only in ISO currencies. The archetype pattern has optional features, so you can "prune" the pattern down to just the bits you need. In this case, you would have the result shown in Figure 1.11.

You must use your judgment as a modeler to adapt the pattern in such a way that what remains is still semantically well formed. For example, if you