An Approach to Support the Modeling and Usage of Analysis Patterns

By He Fei

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Takuya Katayama

June, 2002

An Approach to Support the Modeling and Usage of Analysis Patterns

By He Fei (010030)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Takuya Katayama

and approved by
Professor Takuya Katayama
Professor Kokichi Futatsugi
Associate Professor Katsuhiko Gondow

March, 2002 (Submitted)

Chapter 1

Introduction

Typically, there are two respects in works around analysis patterns:

One is on pattern itself, which describes the elements of patterns, such as the actual structure of solution, the intent or problem behind.

The other is on the surrounding of patterns, that means things supporting the modeling, usage of patterns, the relationship with other patterns, and so on.

The first one needs great understanding and expertise of diverse domains that beyond my reach, so I put my main efforts to find the common principles behind the second respect through the understanding of other peoples' analysis patterns.

Also around the two respects, Martin Fowler offers a set of analysis patterns and support patterns respectively. But on the second respect, the support patterns in Martin Fowler's *Analysis Patterns* mainly address problems in building an actual software system with analysis patterns, just like how to fit those analysis patterns into system architecture and how to transform the model in analysis pattern into an explicit specification model in design or an implementation. And the only pattern used to examine modeling techniques and to advance modeling constructs on analysis patterns is the *association pattern*. My works can be seen as a complement or extension to the latter part of Martin Fowler's.

As to the structure of this paper, first in chapter 2, I give the explanation of these basic concepts related with my works, then discuss the detailed patterns on the principles of the second respect in Chapter 3, at last, chapter 4 offering a conclusion for my works.

Chapter 2

Background

Analysis emphasizes an investigation of the problem and requirements with the application logic in mind, rather than a solution. For example, if a library system is desired, how will it be used? "Analysis" is a broad term, best qualified, as in requirement analysis (an investigation of the requirements) or object analysis (an investigation of objects in the problem domains). The latter is also the feature of object-oriented analysis.

Analysis patterns are group of concepts that represent a common construction in business modeling. It may be relevant to only one domain, or it may span many domains. Analysis patterns belong to the object analysis.

The most important artifact created in analysis patterns is the *domain model*. Combining the conceptual classes, associations and attributes discovered in the investigation yields the domain models, same with conceptual models used in Martin Fowler's *Analysis Patterns*. Keep in your mind that a domain model is a visual representation of conceptual classes or objects in a domain of interest for easily comprehending. There are only relatively useful models, no such things as a single correct model. All models are approximations of the domain we are attempting to understand. A good domain model captures the essential abstractions and information required to understand the domain.

Qualities of a Pattern

In addition to containing the aforementioned element, a well-written pattern should exhibit several desirable qualities.

Encapsulation and Abstraction. Each pattern encapsulates a well-defined problem and its solution in a particular domain. Patterns should provide crisp, clear boundaries that help crystallize the problem space and the solution space.

Openness and Variability. Each pattern should be open for extension or parameterization by other patterns so that they may work together to solve a specialized or a larger problem. A pattern solution should be also capable of being realized by an infinite variety of implementations (in isolation, as well as in conjunction with other patterns).

Generativity and Composability. Each pattern, once applied, generates a resulting context, which matches the initial context of one or more other patterns in a pattern language. These subsequent patterns may then be applied to progress further toward

the final goal of generating a "whole" or complete overall solution by the means of piecemeal growth. But patterns are not simply linear in nature, more like fractals in that patterns at a particular level of abstraction and granularity may each lead to or be composed with other patterns at varying levels of scale.

Equilibrium. Each pattern must realize some kind of balance among its forces and constraints. This may be due to one or more invariants or heuristics that are used to minimize conflict within the solution space. The invariants often typify an underlying problem solving principle or philosophy for the particular domain, and provide a rationale for each step or rule in the pattern.

UML

UML is an industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. As with any language, the UML has its own notation and syntax. Its notation comprises a set of specialized shapes for constructing different kinds of software diagrams. Each shape has a particular meaning, and the UML syntax dictates how the shapes can be combined. Although many major object-oriented analysis and design methods influenced the development of the UML, it is derived primarily from three notations: Booch OOD (Object-Oriented Design), Rumbaugh OMT (Object Modeling Technique), and Jacobson OOSE (Object-Oriented Software Engineering). In 1997, the Object Management Group (OMG) made the UML a standard modeling language for object-oriented applications.

Types of UML diagrams. Each predefined UML diagram is designed to let developers and customers view a software system from a different perspective and in varying degrees of abstraction:

Class Diagram. Models class structure and contents using elements such as classes, packages and objects. It also displays relationships such as containment, inheritance, associations and others.

State-chart Diagram. Expresses possible object combinations of a specific class diagram.

Sequence Diagram. Shows one or several sequences of messages sent among a set of objects.

Use-Case Diagram. Illustrates the relationship among actors and use cases.

Collaboration Diagram. Describes a complete collaboration among a set of objects.

Activity Diagram. Describes activities and actions taking place in a system.

Component Diagram. A special case of class diagram used to describe components within a software system.

Deployment Diagram. A special case of class diagram used to describe hardware within a software system.

Because the class diagram can be used to not only on the specification perspective (describing software abstraction) and implementation perspective (interpreting the software implementation) but also on the conceptual perspective, I use the same UML diagramming notation to illustrate the domain models in my works. This also lowers the representation gap with software designs.

Chapter 3

Support Patterns

There are many performances around analysis patterns: such as the construction, composition, decomposition, derivation, and adoption. If we can summarize the principles behind, we can make those performances into a structured and organized basis, and make sure that the pattern and its particular application can keep the required qualities.

I abstract the principles behind those performances into patterns with the term *support pattern*, for a set of common infrastructures that describes how to construct and apply analysis patterns independent of a specific domain rather than modeling actual analysis patterns themselves. With the support patterns, patterns can be exposed to outside more clearly and cooperate together more easily.

Those patterns are not an ad hoc collection of loosely related concepts but instead aims to originated from an insight on a small number of necessary and sufficient basic building blocks that are ubiquitous in the works around analysis patterns.

The support patterns in Martin Fowler's *Analysis Patterns* mainly address problems in building an actual software system with analysis patterns, just like how to fit those analysis patterns into system architecture and how to transform the model in analysis pattern into an explicit specification model in design or an implementation. And the only pattern used to examine modeling techniques and to advance modeling constructs on analysis patterns is the association pattern. My works can be seen as a complement or extension to that of Martin Fowler's.

But people should also notice the difference between support patterns and *meta patterns*, the latter is a set of design patterns that describe how to construct frameworks independent of a specific domain. Also meta patterns are based on the basic principles of template and hook methods, and the set of meta patterns capture different configurations of classes containing these methods.

Patterns are often organized and classified into a set of categories independent of the problem domain they come from. There are different possibilities for categorizing patterns, but a common delineation is as follows:

Functional. Describing the functionality of system.

Structural. Dealing with structure issues.

Behavioral. Capturing the behavioral aspects in dynamic descriptions.

The support patterns presented in this paper have another categorization:

Structure patterns.

Collaboration patterns.

These categories will be discussed in more details shortly.

There is no rigorous form for the support patterns described in this paper; I just simply state the intents or motivations behind, show and a visual representation of the pattern structure in UML diagram, and illustrate by some examples. Generally, those patterns only have static type models.

It is important to note three facts, one is that a pattern belongs to a certain category does not mean that the pattern can only possess characteristics for that category. Placing a pattern in a particular category means that the pattern is based more on the category under which it falls; the second is that the most successful patterns are those that can be rewired in many configurations in effective response to changes; and the third is simplicity as the highest priority while modeling and applying those patterns in your own works.

3.1 Structure Patterns

Actor-Role Pattern
Title-Thing-Information Pattern
Generalization-Specialization Pattern
Association Pattern
Definition Pattern
Document Pattern

Actor-Role pattern and Title-Thing-Information pattern help people to identify and represent the meaningful (to modeler or required system) concept entities. Identifying a suitable set of objects or conceptual classes is at the heart of the analysis works, and well worth the effort in terms of pay-off during the later design and implementation works. Generalization-Specialization pattern and Association pattern are used to make a clear insight of the relationships among those objects or classes. Definition pattern and Document pattern offer the auxiliary services to other patterns. All those patterns help people not get lost in the maze of concepts, or hard to define and locate the corresponding classes.

These patterns establish rules or constraints as the inherent part of those objects or classes. Constraints structure and affect those classes and the attributes thereof, all this means that constraints cannot be separated from the structure where they exist.

Using these patterns, modelers can have a clear idea about what the system structure looks like right now and can abstract this information into a more generic model not only applicable today but flexible for future changes and extensions.

About the sources of those patterns: you can find almost all the needs of the patterns from the process of domain model generation in Craig Larman's *Applying UML and Patterns*. The Generalization-Specialization pattern is also apparent in user-interface frameworks, such as the well-known Model-View-Control architecture. Association pattern directly comes from that of Martin Fowler's *Analysis Patterns*. The Document pattern was also inspired by document patterns found in David Hay's *Data Model Patterns*. Title-Thing-Information pattern is derived from the Item-Item Description pattern in Peter Coad's *Object Models: Strategies, Patterns and Applications*, and Larman call them the need for specification or description classes.

3.1.1 Actor-Role Pattern

The Actor-Role pattern provides guidelines for using actor and role concepts, including how they should be separated and how can be combined.

An actor is someone or something that functions on its own, such as a person. You can employ actors within a company. A role may have a certain series of actions taken by an actor. You cannot employ the role; it is defined in certain structure that uses it. For example, company employs a person as president; thus, the person is an actor playing the role of president.

An actor can have more than one role at a time, and the same role can be played by more than one actor. However, the actor, who can and is allowed to play more roles, cannot play them at the same time in some cases. The actor-role pattern makes it easier to model and describe the constraints for such occasion, which can be very hard to model if the actor and role are not properly separated. An example is a system that handles sensitive data where the roles of system operator and system administrator need to be separated, because a system administrator adds or removes user accounts to the system, whereas a system operator has access to the data within the system. In this case, an actor that has the role of system administrator can never take on the role of system operator, because one would like to avoid the risk of a system administrator who creates accounts for himself and thus gains access to sensitive data. In this example, separating the roles eliminates a security risk.

Other use for the actor-role pattern is when an actor needs to be matched to different roles. Roles and actors have different attributes. Actors have attributes that describe their abilities. Roles have attributes that describe operational directions (such as responsibility attached to that role) and, often, requirements for the actors who play those roles. These requirements can be based on the actor's defined attributes. This pattern help to identify which actors are most qualified or even permitted to have certain roles.

If constraints are assigned to different roles and there is no separation of actor and role, the constraints are difficult or even impossible to express because a single entity involves both the actor and the role. If the actor has several roles, there is a big risk that the roles will be simply lost in the model, or that the actor will be defined as having an aggregated role that may become very specialized for that actor and be intermingled with the actor's attributes. The distinction of different roles becomes lost because the roles are not separated from the actors that play them.

As to the applicability of the actor-role pattern, it can be used in all problem situations in which there is a need to separate actors from roles. For example, an everyday business rule for any bank could be that all huge withdraws must be approved by the bank's office manager. But if the roles of office manager and bank clerk are not separated, and only defining a bank employee to an actor, it would be hard to express precisely which actors are allowed to approve those withdraw, and finally, this would cause problems in the organization management of the bank. The actor-role pattern models the actor, the different roles and the rules to ensure that they are mutually exclusive.

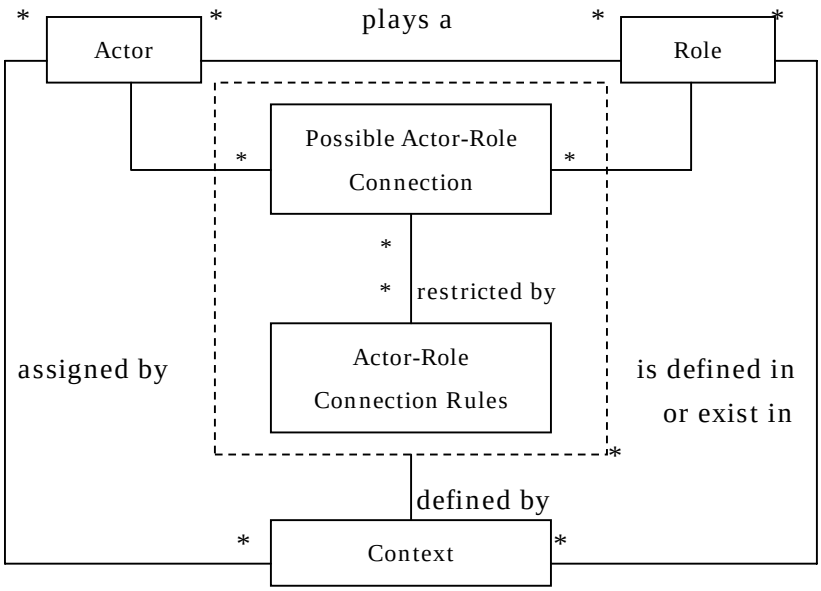

Figure 3.1 The Actor-Role pattern's structure

Context is the situation in which the actors exist and for which the roles are defined.

Actor class describes the actors.

Role is a description that tells the actor how to function in a particular context. Actor and Role can be specialized into subclasses.

Possible Actor-Role connection expresses possible or allowed connections between actors and roles.

Actor-Role Connection Rule is the basis for the Possible Actor-Role Connection class, for example, only one of the possibilities is allowed in certain point of time.

Actor-role pattern enable the easy separation of actors and their attributes from the roles that they play. And enable you to have a more clear insight of the organization structure the actor plays within, since the roles are one of the key reflections of structure. This makes it can be combined with the Organization and Party pattern in *Analysis Patterns*, typically by creating an association from the role class in the

actor-role pattern to the organization unit class in the organization and party pattern, then by connecting and defining the role so it is a specific part in the organization.

One advantage of using the actor-role pattern is that you can identify roles that cannot be played at the same time by the same actor, alone with certain actor requirements that are dependent upon the role played. Using this pattern also makes it possible to locate and define certain connections, such as that a certain actor can play a set of roles in one context but not in another,

Note: If the Actor-Role pattern is always used in situations where a one-to-one relationship exists between the actor and the role, then this pattern will lean to more complex in the model.

Within the context of management of component trading, those member companies and persons can have possible connections with the roles of user, supporter, vendor, and criterion keeper. As the snapshot of a real situation showed in Figure 3.2, the actor-role connection rule can state AND that means all associated roles must be played at the same time, in other case, it may be XOR which means that two roles cannot be played simultaneously, only one of the roles referenced can be valid at a time.

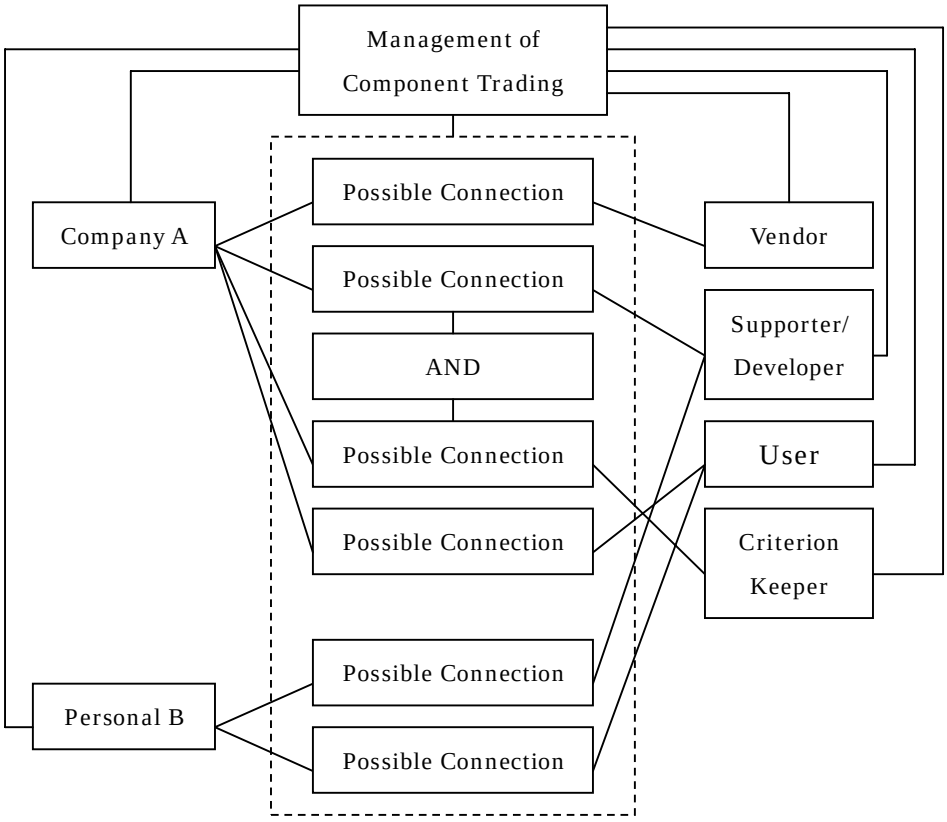

Figure 3.2 An example of the Actor-Role pattern

3.1.2 Title-Thing-Information Pattern

The Title-Thing-Information Pattern is a special pattern. In the thing-information pair, it eliminates the focus–shifting by referring to two frequently used modeling views (thing focus and information focus) and how they are related to each other, and in the title-thing pair, it also help modelers to simplify the modeling process for systems that involve thing and titles of the thing that may share the same name but having different attribute.

Because information is named according to what it represents, it is not always easy to distinguish the thing and the information thereof, especially when both appear in same model. So, during construction of domain models, it is necessary to analyze and structure both the thing and information about it. For instance, logistics in a company comprise both the actual transportation of goods and the information about the good and details on the transportation. The difference is that the goods have attributes such as size, color and package form while the information about the goods has attributes such as delivery address and date. If you model the information and thing object in the same class, these concepts are intermingled, making it difficult to determine which attribute describes the physical thing and which attribute provides information about the thing. This causes problems when maintaining and updating the information. But people should keep the thing and its information class in the same model, because they both are parts of the logistics.

Another example is about the customer-information pair, many business systems, just like the client database or e-business, handle customers; however, they do not handle or store the actual customers, only the information about those customers. In this case, the customer class in the systems contains information of a customer.

A title is a concept that typically refers to a thing or item as its description on properties. One concrete example is the problem domain of library. In the library, both the book and physical copy (thing), which sharing the same title, have to be handled. Searching a book often concerns with the title, key words, authors, ISBN, but the result may show you multiple physical copies of the book which keep the quite different information on whether the copies are on shelf, who own those copies currently or where they locate. Supposed you model the physical copy and the title together. If your clear old books from the library and delete those classes, then you will never know whether you have it or not in history. Also let the instances have redundant or duplicated data and get be space-inefficient.

It is a very generic need to defining those three elements separately and clearly, so

misunderstanding and confusion can be avoided, and the future maintenance of the models and the building of information system based on the models will be much easier.

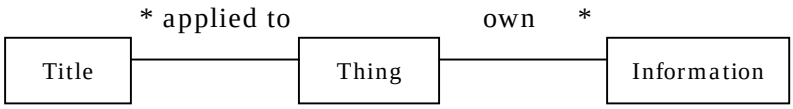

Figure 3.3 The structure of Title-Thing-Information Pattern

Thing is an object that can be concrete and physical, such as customer, or abstract, such as the party. Things form the building block of an enterprise, and can be specialized to other types such as products, persons.

Title represents the title concept corresponding to a thing, it may have categorized attributes, and different category has different limitations, just like biology encyclopedia dictionary belongs to a category of natural science, but according to physical copy category, it is a dictionary not a book, so you cannot borrow it out.

Figure 3.4 An example of the Title-Thing-Information Pattern

In the above example, the product (thing) can be an engine, it may only show the properties that customers have interest in, such as the rate of burning efficiency or fuel consumption; the product description (title) keeps all the detailed records of the specification of the engine and production thereof, usually only the manufacturers, not the customers, are interested in this description, the product information (information) may have the quantity, the delivery date, or the current market price of the engine or the factors concerned within a contract which both sides of the deal care for.

3.1.3 Generalization-Specialization Pattern

The Generalization-Specialization pattern structures the essentials in a problem domain with the purpose of building well-structured and easily changeable models. The superclass or core objects (generalized) of a business are things that rarely change fundamentally; conversely, subclasses or the representations of core objects (specialized) often change or are extended in its own context, and have many forms. A modeler should take this into consideration and separate the superclass from its subclasses by

generalization and specialization.

Generalization is to identify commonality among concepts and defining superclass and subclass relationships, and allow us to understand concepts in more general, refined and abstract terms. It leads to economy of expression, improving comprehension and a reduction in repeated information. The specialization is operated under the motivation that additional attributes or associations of interest need to be handled in the subclasses.

Generalization and specialization are fundamental concepts in domain modeling, which form the conceptual class hierarchies that are often the basis of inspiration for software class hierarchies.

Models that use the Generalization and specialization pattern can handle changes in the representation without redefining the core object. It is also possible to add new representations at a later date without affecting the definition of the core. So you can create adaptive models for your system, and less expensive to maintain.

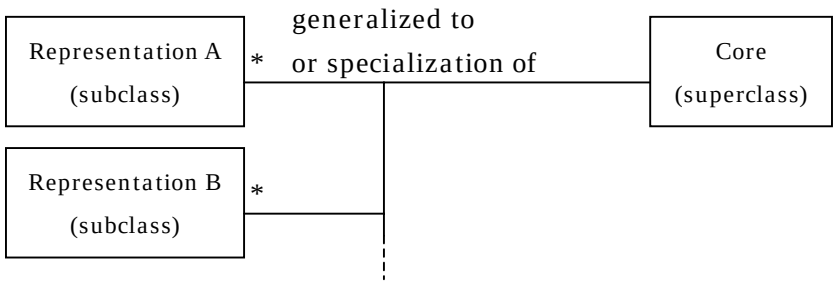

Figure 3.5 The structure of the Generalization-Specialization pattern

In many businesses, the concepts *CashPayment*, *CreditPayment*, and *CheckPayment* (*CP*) are frequently used and all very similar. In this situation, it is possible and necessary to organize them by this pattern, where the *Payment* class represents the more general concept, the superclass, and the three *CP*s as the more specialized ones, the subclasses.

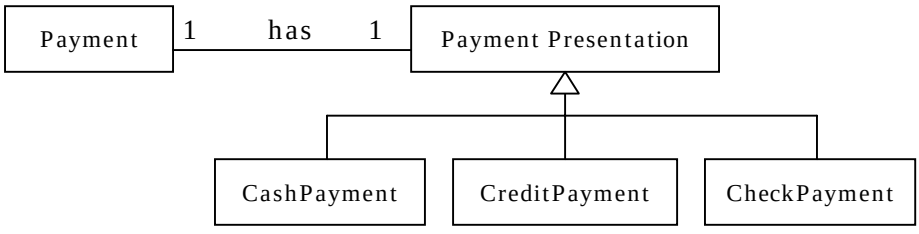

Figure 3.6 An example of the Generalization-Specialization pattern

3.1.4 Association Pattern

Usually, when an association has it own attributes, the instances of the association have a lifetime dependency, or there is a many-many association between two concepts, it is necessary to build the association in a separated class. More detailed explanation can be found in *Analysis Patterns*.

Note: a special case in a domain model, if a class C can simultaneously have many values for the same kind of attribute A, do not place attribute A in C. Place attribute A in a new class that is associated with C. For example, a *Person* may have many contact information, just like e-mail address, fax and phone numbers, place those in another class, such as *ContactInformation*, and associate to the *Person*.

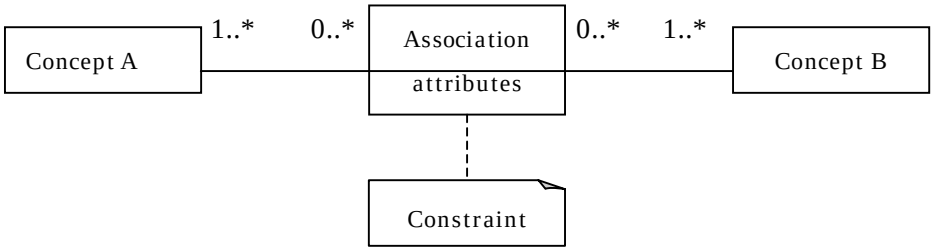

Figure 3.7 The Structure of the Association pattern

Aggregation is a kind of association used to model whole-part relationships between things; the whole is called as the composite. But identifying and illustrating aggregation is not profoundly important; it is quite feasible to exclude it from a domain model, because most of the following benefits on it relate to the design rather than the analysis: the first, it clarifies the domain constraints regarding the eligible existence of the part independent of the whole. In composite aggregation, the part may not exist outside of the lifetime of the whole; the second, operation applied to the whole often propagate to the parts.

Here, we should also notice the difference between roles as concepts and roles in associations. Roles in associations are appealing because they are a relatively accurate way to express the notion that the same instance of a person takes on multiple (and dynamically changing) roles in various associations. I, a person, simultaneously or in sequence, may take on the role of designer, parent, and so on. On the other hand, roles as concepts provide ease and flexibility in adding unique attributes, associations, and additional semantics.

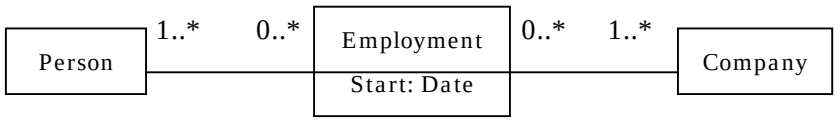

Figure 3.8 An example of the Association pattern

Martin Fowler also mentioned two other patterns, one is *Keyed Mapping*, the other is *Historic Mapping*, see more details in *Analysis Patterns*.

3.1.5 Definition Pattern

The Definition Pattern captures and organizes term definitions in business or problem domains, in order to manage them and improves the vocabulary of patterns.

All businesses and problem domains have many critical concepts that must be communicated clearly, accurately, and easily via terms, that is, words. The important terms for these concepts, call *domain definitions*, must be defined unambiguously. The word associations we make to the same concept are very individual is well known in most fields. Therein lies the need to rigorously define critical terms used to describe concepts within a domain. It is not sufficient to just define the concept that the term represents. As people will undoubtedly use the same term differently, so it is also necessary to demonstrate the various uses for the term, in short, it must be carefully defined for each potential situation or group users.

Domain definitions are composed of a term (actual word or words), a description of how it's used, and a concept (semantics of the term, the actual meaning of the word).

In the field of object-oriented modeling the term *multiplicity* is widely used to describe the number of instances allowed at the end of an association. In the world of data modeling, another term, *cardinality*, stands for the same concept. This is an example of two terms used to describe the same concept, but in different contexts (object-oriented versus the data-modeling community). This classifies the need to differentiate the use of the term and its concept. In some cases, which show that the same concept can have more than two different terms. This is referred to as the *term usage*. Clearly, to define a pattern in certain contexts, it is vital to define the various terms, the correct use of those terms, and the concepts that those terms represent. This is where the Definition Pattern comes in.

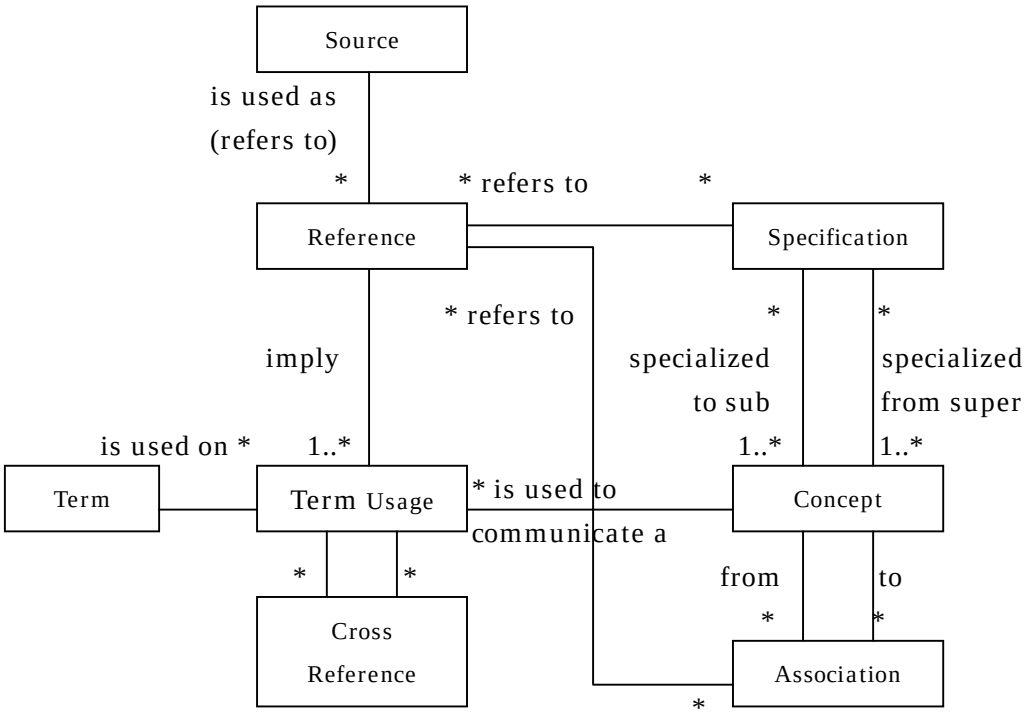

Figure 3.9 The Definition Pattern's structure

Term is represented with words and has a name, which can be a text string or an equivalent. Term is used to communicate one or many concepts. The terms and concepts are connected to each other through the Term Usage.

Term usage class is the connection between the terms and the concepts, or how the term is used by a specific group of users. Typical attributes can be a description of the term's users. The usage itself is not expressed as an attribute; it is expressed through the associated classes Reference, Term, and Concept. In some cases, it is useful to connect the Term Usage to an explicit user or group of users. This may be achieved by adding a user class and associating it with the Term Usage class.

Concept is an understanding or interpretation of something in the real world. Terms are used to communicate concepts among people. The concept name or label is the term used. A concept is usually defined through its relationships to other concepts. The relationship can be of two kinds: Association or Specialization.

Association is used to combine or relate concepts to each other. An association exists between the instances of the concepts, meaning that associations have multiplicity.

Specialization is another potential relationship between concepts. As opposed to Association, Specialization is used only between concepts, it cannot be with instances and do not have multiplicity.

Source is the point of origin form that the different kinds of term usage are generated and described.

Reference class is between the Source and the Term Usage. The Reference uses a

Source with reference to both specializations and associations between the Concepts referred by the Term Usage and other concepts to present a picture of how to use a Term that reference a concept.

Cross-reference is used to support the term usage in multi-domains, which concerned with different terminology applications.

3.1.6 Document Pattern

The intent of the Document Pattern is to provide a practical way to approach the issues inherent in the modeling and management of documents of a pattern, all these works result in a structured understanding of the pattern and the effective usage thereof, such as how to apply, adjust or improve it in your own system.

The development and maturation of a pattern are achieved incrementally, so documenting the ongoing process and studying the documents are good ways to understand the pattern, and also make the pattern management easily. The consistent structure of documents lends uniformity to patterns, letting people compare them easily. Structured documents also help people search for information. Less structure means more prose, which might be fine for casual reading but unacceptable for comparison and reference purposes. Once you have settled on structured documents, make sure you follow it consistently. You need not be afraid to change the structure, but you will have to change it in every pattern, and that gets increasingly expensive as your patterns mature.

Also, patterns are not to guarantee reusable software, higher quality or productivity, etc., people should not overemphasizes solution at the expense of problem, context, teaching and so on. As the experience and expertise, the generativity of pattern is in the parts of a pattern dedicated to teaching and communication through deep understanding of its documents, not only the solution but the forces behind, the possible consequence, the example, the instance, etc.. Put it simple, patterns themselves do not guarantee anything, people who use them do, and they do it only if both they and the patterns they are up to snuff.

The essential documents of patterns usually comprise:

The pattern itself, describes all the elements of the original pattern, such as the author, intent, structure of model, participants, and so on.

The versions of the pattern, record the evolution of the pattern, such as the changes in the structure of the pattern; the feedback that measures and evaluates the results of

the pattern; the trade-off, balance, special requirements or configurations set in the pattern while applied to certain contexts or domains, which can be different from the original domain where the pattern discovered.

The pattern instance, which can be considered as the application or execution of the pattern in actual system. Separating the pattern from its instances clarifies the distinction between pattern and a pattern instance. The distinctions mean the generic properties of the pattern and the individual properties of the instance.

A pattern instance is always generated under a certain version of the pattern, a classic instance can be a powerful example or explanation of the pattern. Each version and instance object should have kept a record of the properties of the context in which the pattern had been involved, such as when, where, how.

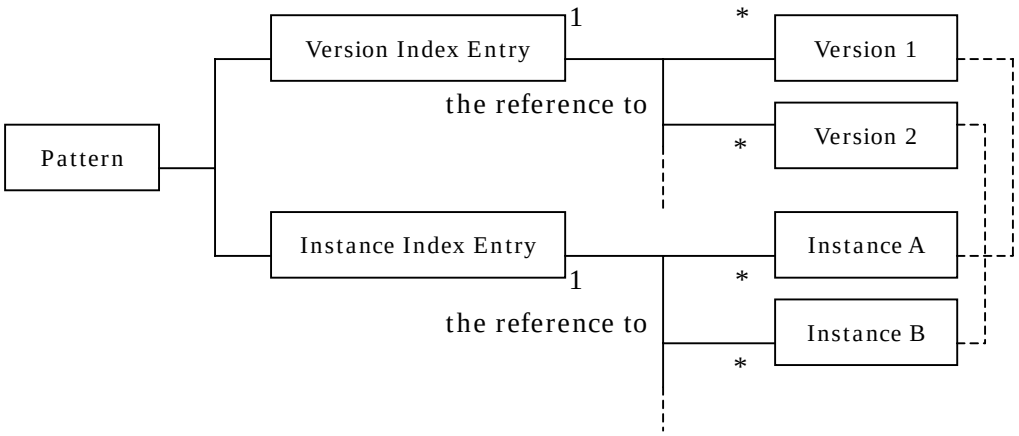

Figure 3.10 The structure for the Document pattern

Index Entry is a class used to index documents. A document can be indexed on version or pattern instances, I separate the Index Entry into Instance Index Entry and Version Index Entry to simplify the model, so I do not need to care the specialized relationships between Index Entry Instance Index Entry and Index Entry Version Index Entry. Each index entry is a reference to one or more objects. The index is a strategy for identifying documents through a set of information associated with those documents.

3.1.7 Summary

In this part, I wish to find out those elementary building blocks that provide a relative small set of patterns, based on those principles or from which complex patterns can be constructed more efficiently, and also hope them can be compact and expressive, as to the saying, “less is more, less is powerful”. But I still think we need further discussion

on the potential usages of many other patterns, just like the observation pattern, organization pattern, party pattern or category pattern which help us restrict our discussion to topologies of interest.

3.2 Collaboration Patterns

Layer Supply Pattern
Layer Control Pattern
Composition Pattern

The collaboration patterns describe the relationships between patterns and help to organize them into certain hierarchy structures to support the pattern application.

There are two ways in UML to represent a pattern. To review the notations, one is by using the symbol, called *collaboration*, to represent both the structure (typically class diagrams) and behaviors (typically a sequence or a collaboration diagram), in order to see them, you need to zoom into the pattern, see Figure3.11 a. The other is *package* shown as tabbed folder. Subordinate packages may be shown within it. In the case of pattern, you can show the domain models, constraints within it. The package name is within the tab if the package needs to depict its elements; otherwise, it is centered within the folder itself, see Figure3.11 b.

a. collaboration b. package

Figure3.11 The UML notations for patterns

Packages are linked by *visibility relationships*. And all possible visibility relationships must be explicitly declared in a package. This is required for any service between different packages: such as holding in an attribute, or passing as a parameter. You should keep in your mind there are *multiple access levels* to a package, this always results in different needs on interfaces from different packages. But when developing a large system, people should try to minimize the visibilities between packages so that the system has less dependency and is thus easier to manage. Martin Fowler also mentions other problems and solutions, just like the mutual visibility, but it is beyond the current scope of this paper.

Any element is *owned* by the package within which it is defined, but may be *referenced* in other packages. A class shown in a foreign package may be modified with new associations, but most otherwise remain unchanged. If a model element is in some way dependent on another, the dependency may be shown with a dependency relationship that can be an arrowed line. A *package dependency* indicates that elements of the dependent package in some way know about or are coupled to elements in the

target package.

In my works, I choose package to represent a pattern, for I thought that package notation is a more controlled mechanism and more intuitive to organize the elements of analysis patterns.

A domain model of a analysis pattern can easily grow large enough that it is desirable to factor it into smaller packages of strongly related concepts, as an aid to comprehension and parallel analysis works in which different people do domain analysis with different sub-domains. To partition the domain model into packages, place elements together that:

are in the same subject area, closely related by concepts or goals;

are in a class hierarchy together;

are strongly associated.

3.2.1 Layered Patterns

The following two patterns state the structure of patterns layered in a hierarchy with the concerns of goals or problems behind.

Goals are not only what the business models and the resulting business process strive for, but also the intents or motivations behind each analysis pattern. They establish the reason of the existence of the domain model in analysis patterns and the definition of the resource objects and constraints that related with the achievement of goals in the models. Any pattern without corresponding goals should be eliminated. The more clearly a goal is stated, the easier it is to identify the corresponding patterns so that the goal can be achieved.

Also certain problems may hinder the achievement of these goals. The goal and the problems thereof are just the flip sides of the same coin counted on a pattern.

No matter what kind of business or problem domains, it will always have its main goals or senior problems for its existence and some auxiliary goals working as supplying factors. This hierarchy reflected in corresponding analysis patterns results in the *Layer Supply pattern*.

Goal decomposition is used to streamline the goal modeling process by breaking down the goal into hierarchies. In this way, high-level goals can be divided into more concrete

sub-goals that are then allocated to specific patterns. With this as the internal force to the hierarchy of corresponding analysis patterns, we have the *Layer Control pattern*.

3.2.1.1 Layer Supply Pattern

Because patterns are motivated by the goals or problems thereof, we also should model the hierarchy of patterns according to the properties of these goals or problems.

The Layer Supply pattern organizes the structure of complex organization of patterns into primary and supporting patterns by the priority of goals or problems behind them. Breaking the organization down into primary and supporting patterns allows for a better understanding of the entire organization and provides a stable and solid foundation for future work efforts. The division can be made in several layers where one pattern can supply and be supplied at the same time.

Patterns that achieve the primary goals and solve the main problems thereof in a system, called *primary patterns*, Patterns that perform the supporting works on junior goals or problems of a system, called *supply patterns*.

To state the obvious: in a car manufacture system, managing the process of producing car components is much more important than ordering materials from a third parties, the later only offer supply services; but in a network meeting system, the pattern that handles information exchange or negotiation with other members are the key features of the system, more essential than the pattern describing local storage for information exchanged.

There is a special case, in which patterns are grouped under certain sequence, and you cannot tell the priority of its goal or problem among them. The achievement of previous goal just forms the required context or pre-condition for the next goal, and the whole process is completed step by step. However, you can still regard the works of previous goals as the supply to the following goals.

The Layer Supply pattern is related to the Layer Control pattern, up next, which is also organized in a hierarchy of layers. In the latter pattern, each layer controls the layer below, whereas each layer in this pattern supplies and creates the conditions required for the layer above.

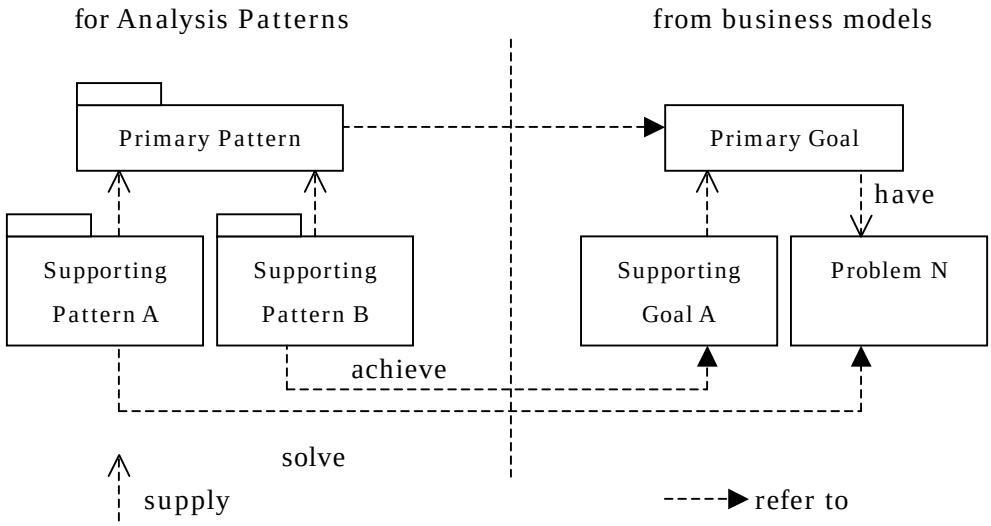

Figure 3.12 The structure of the Layer Supply pattern (left)

Many business systems, which concerned with production, sales and delivery, describe product-to-market and product-to-customer, where the product-to-market supplies the product-to-customer with a set of products. Here, the product-to-customer is the primary part, for it is the part that can really bring the business benefits; and the corresponding patterns are primary patterns. Also illustrated in Figure 3.13, there is another pattern that maintains the infrastructure, called the maintain pattern, and supplies the product-to-market and product-to-customer pattern both. Related to maintain pattern the product-to-market pattern can also be considered as a primary pattern.

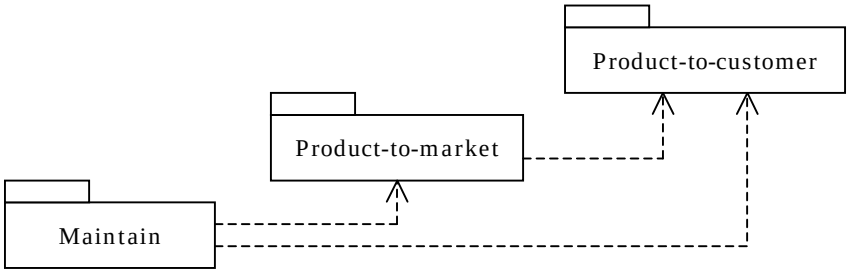

Figure 3.13 An example of the Layer Supply pattern

The product-to-market can include product plan and development patterns to help implement the products needed. The product-to-customer can comprise contract and sale patterns to achieve the services for customers. The maintain pattern can have protocol and resource allocation patterns as its elements to collaborate the performance of those primary patterns, so those patterns can keep their productivity.

3.2.1.2 Layer Control Pattern

Layer Control is a pattern that helps to structure complex relationships of patterns for the purpose of reengineering or understanding them. The fundamental principle is that all patterns are layered to control the layer underneath.

The hierarchy of patterns can be studied and modeled from several perspectives, two of which are very useful:

Target-oriented perspective. Each pattern enables the pattern above it; the pattern at the top is motivated by the overall goals of the system. This perspective is used in the process layer supply pattern previously described.

Control-oriented perspective. The difference is that the pattern on top, which is directly motivated by the overall goal, controls the pattern underneath, which in turn controls the next pattern below, and so on. This perspective is used in the Layer Control Pattern. The layer control pattern usually affects our strategies in pattern management and application. That means if your patterns generated in a complex domain are not well structured, you may finally lose control of your development.

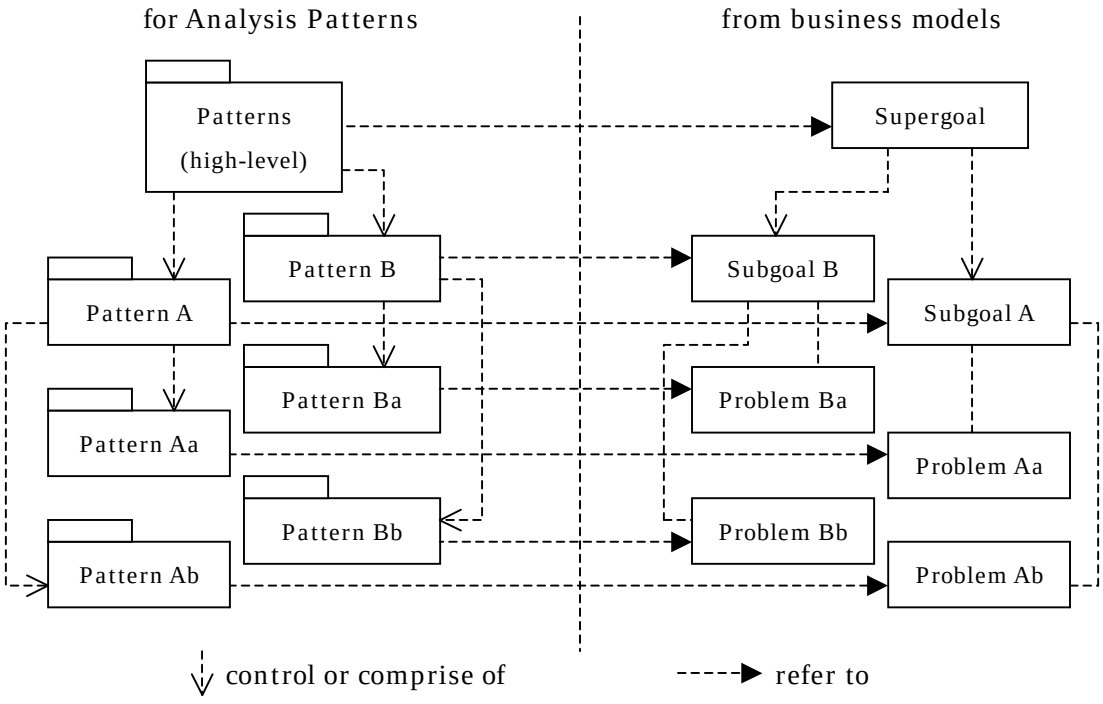

Figure 3.14 The structure of Layer Control pattern (left)

The special case of a set of sequence goals or problems can also be explained in this way by consider the each step as a sub-goal in the whole process.

Here I use the example of *trade* to illustrate the layer control pattern, illustrated in

Figure 3.15. Trade is about buying and selling of goods, and the value of these goods with respect to changing market conditions; this makes people have to understand the value of the net effect of these trades in different circumstances.

Each trade is described by a *contract*. The contract can either buy or sell goods and is useful for businesses which need to track both directions of deals. People can look at the net effect of a number of contracts by using a *portfolio*, so people can manage the joint risk and assemble them easily to select contracts in different ways or under certain criteria.

To understand the value of a contract, people need to understand the price of the goods being traded. Goods are often priced differently depending on whether they are bought or sold. This two-way pricing behavior can be captured by a *quote*.

In volatile markets, prices can change rapidly. Traders need to value goods against a range of possible changes. The *scenario* puts together a combination of the conditions that can act as a single state of the market for valuation. Scenarios can be complex, but we still need to use the same scenario construction at different times in a consistent manner. Scenarios are useful for any domain with complex price changes.

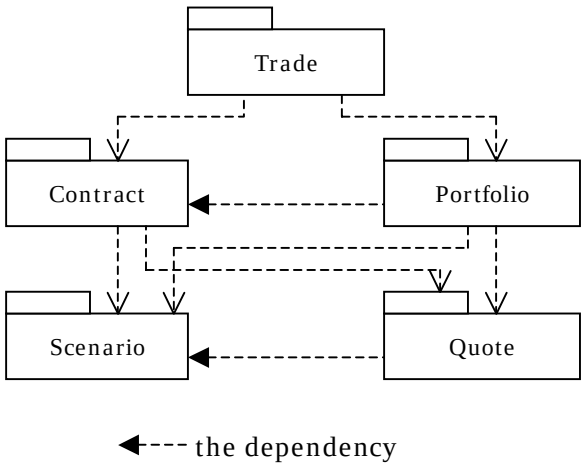

Figure 3.15 An example of the Layer Control pattern

No hierarchy exists simply supplying the above or controlling the bellowing, the Layer Supply pattern and the Layer Control pattern are then used together often. I combine the examples of the above two patterns to illustrate this, seeing the example in Figure 3.16.

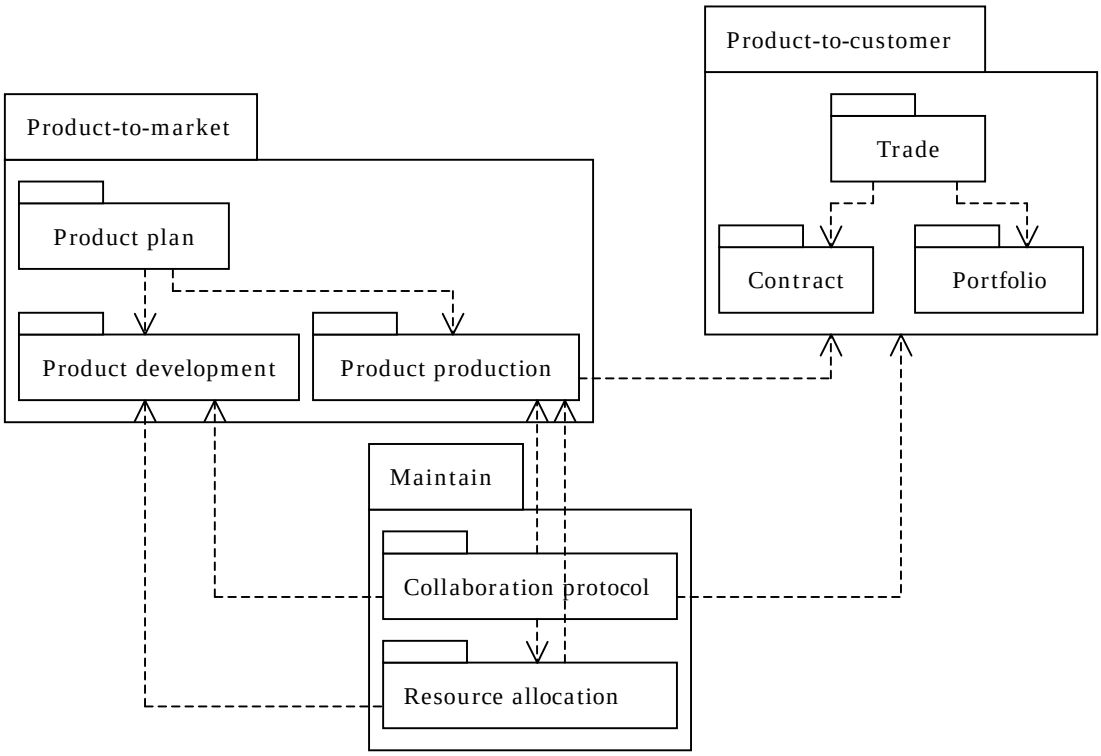

Figure 3.16 An example of the combination of layer supply and control patterns

3.2.2 Composition Pattern

The composition pattern is to build more complex patterns in a incremental way of combining patterns to progress further toward the final goal of generating a "whole" or complete overall pattern that state the whole domain or certain aspects of the domain. The composition pattern is based on the sharability and reusability of patterns. To reach the goal, patterns must support the following features:

1. The pattern must be consisted of predefined normative modeling constructs, not only with modeling manners and notations.
2. Predefined modeling constructs should include the common atomic objects or aggregated objects (by using the atomic objects) that are agreed by all members and are needless to be discussed when they are used.
3. The typical relationship among common elementary aggregated or simple objects should also be predefined as normative modeling constructs.
4. Because pattern also encapsulates constraints within it, it is also necessary to include mechanisms for constraint inheritance and composition among patterns.

Figure 3.17 illustrates the principle of composition, which organizes patterns into

recursive structures to represent part-whole hierarchy, and give users a uniform way of dealing with these patterns whether they are internal nodes or leaves.

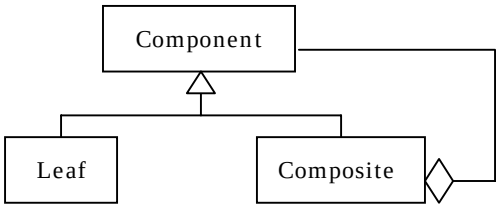

Figure 3.17 The recursive principle of composition

As usual, there should have three basic forms of patterns. First, simple or elementary pattern which is a pattern consisting of minimal elements needed to form a pattern and does not involve another pattern. Second, inherited pattern which is a pattern defined by inheriting from another pattern. And the third is composite pattern that is a pattern defined as a result of combining more than two patterns in a recursive way. The instantiation of a composite pattern in this hierarchical structure becomes possible by resolving pattern inheritance and collaboration with the “unfold” performance.

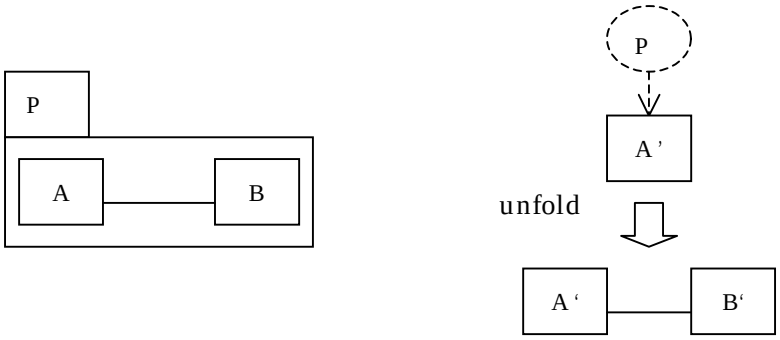

Figure 3.18 The elementary pattern (left) and the application thereof

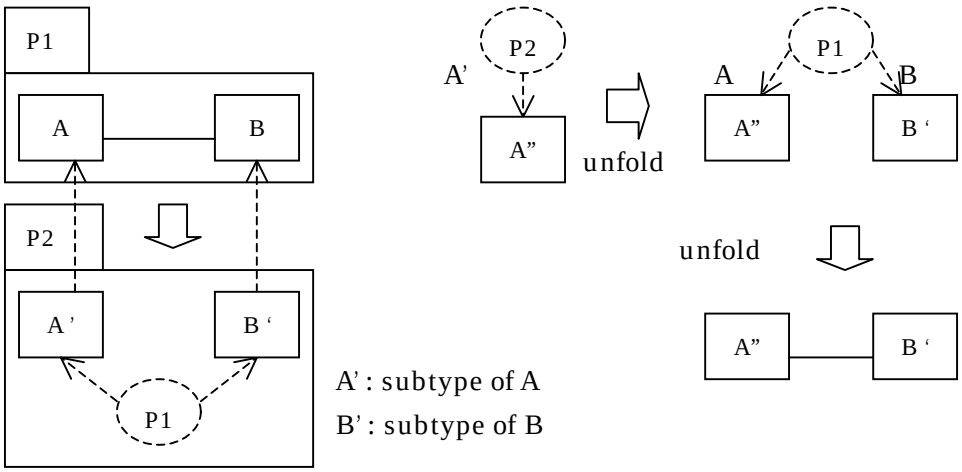

Figure 3.19 The format of pattern inheritance (left) and the application thereof

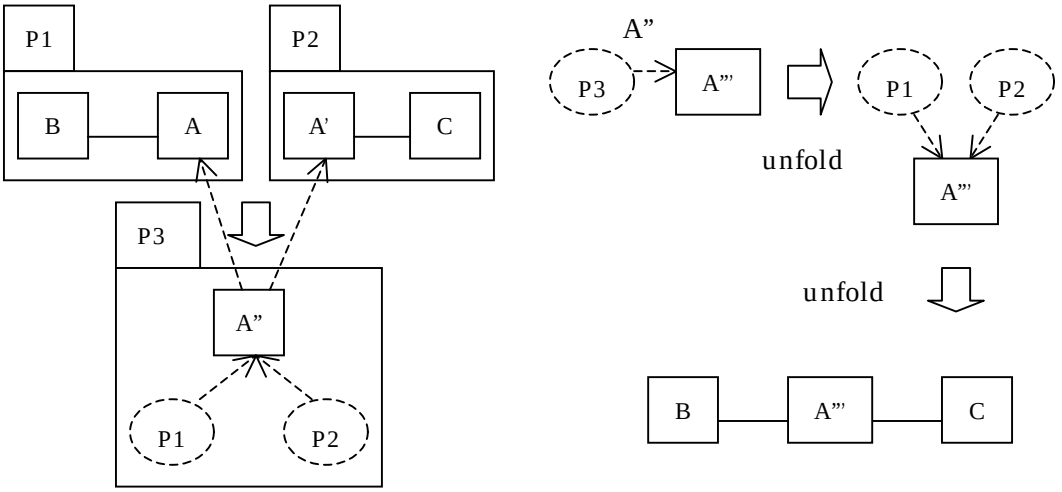

Figure 3.20 The structure of pattern composition (left) and the application thereof

The P1 and P2 can be seen as the elementary patterns or composite patterns; the P3 is a new composite pattern, the result of composition of P1 and P2. In the package of P3, Pattern P1 and P2 should both reference to the common characteristics (a set of common objects, the structure of those objects, and the constraints thereof, or even patterns) shared by them. The dependencies among patterns within a pattern package can be derived from the hierarchy structure described by layer supply pattern and layer control pattern. UML parameterized collaboration mechanism is used to materialize the pattern integration.

Based on the concepts from *accountability pattern* in *Analysis Patterns*, resulting a simple example to illustrate how to apply the composition pattern, seeing Figure 3.21.

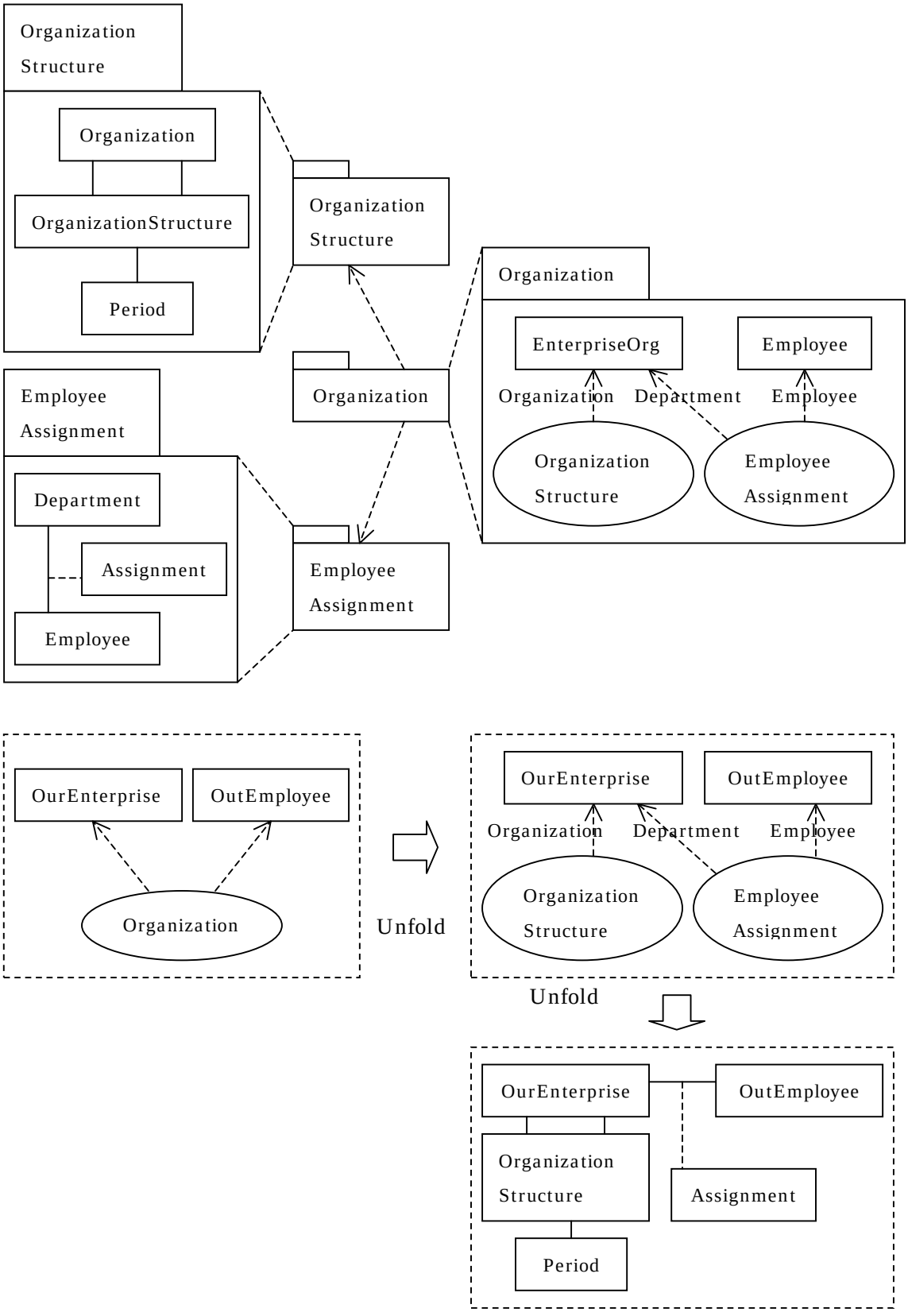

Figure 3.21 An example of Composition pattern and unfolding thereof

3.2.3 Summary

Many applications of patterns are an available-enable process, an adoption process. The process demonstrates from demand to demand satisfaction by choosing the suitable patterns from all the available patterns to enable the target system. It is also used to demonstrate the combination of the previous patterns.

The detailed configurations of the goals, problems or constraints defined by the domain generate the demands that patterns must satisfy. Under the hierarchy described by layer control and layer supply patterns, an adopter chooses one or more patterns by the prosperities recorded in the document pattern to form the certain pattern or patterns that may be derivations or extensions of the original patterns or a composition of a group of patterns. The available-enable process encourages precluding spending time and efforts on unnecessary searching and leads to a more effective pattern or a group of patterns to your needs.

And usually, there is more than one pattern available for choosing, especially when people develop a series of systems, they will find out that many patterns possess the quite similar properties, or you can say they origin from the same ancestry. This leads to the concepts of *pattern family*. All the member patterns in a family should be organized as sub-hierarchies of the global class hierarchy under the root of their ancestor. Recursively, a family can comprise sub-families. The family can add virtual or late instance creation to these sub-hierarchies and suggest certain adoptions on appropriate family members. This way, the family could ensure that users of the family are always provided with instances of the patterns that best suit their needs, without needing to know the detail of these member patterns in a family.

The ancestor is naturally abstract, usually not necessary to have its own instances, and specifies the properties shared by all its descendants. Every member pattern of the family also has unique properties that distinguish it from other members or relatives. In fact, the differences are the only reasons to specify member patterns or sub-patterns. Yet there are many situations in which a user need not see nor care about (unless it desires to) the distinction between the different members, it simply requires that the instance it needs haves what expected, that is, conforms to the interface specified by the abstract ancestor, the root of the family.

Instance creation is provided by virtual or late constructors within the root, but defined by the properties of its descendants, and the concrete pattern of created instance is determined by the constructor and the parameters of the construction.

Chapter 4

Conclusion

In my works, I hope to possess a set of support patterns as a complement or extension to that of Martin Fowler's with some desirable properties: generality, abstraction, elementary, compact and expressive. Although, adopting those patterns would not guarantee your success in yours own works, but it should at least help you focus your efforts profitably, and more impact they will have.

Also this paper reflects the incomplete state of satisfaction on the ultimate goals. I think you even feel unsatisfied, that there are more to discuss and more to extend, such as those patterns listed in the appendix: table of patterns in *Analysis Patterns* need to be re-organized in order to generate new patterns. I also need to find out more examples that can be applied to, check the existing problems, gaps, limitations, or variation. Simplified, trying all those patterns out in more business or problem domains.

Now, if you agree, then consider what Alfred North Whitehead said in 1943, admitted in different context, that might nonetheless make a more appealing conclusion:

> *Art is the imposing of a pattern on experience, and our aesthetic enjoyment in recognition of the pattern.*

References

[1] Martin Fowler, Analysis Patterns, Reusable Object Models, Addison-Wesley, 1997.

[2] Craig Larman, *Applying UML and Patterns*, Prentice Hall PTR, 2002.

[3] Hans-Erik Eriksson, Magnus Penker, *Business Modeling with UML*, John Wiley, 2000.

[4] Gamma, E., R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Object-Oriented Software*, Addison-Wesley, 1995.

[5] Desmond F. D'Souza, Alan Cameron Wills, *Objects, Components, and frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1999.

[6] The Unified Modeling Language Resource Center,
http://www.rational.com/uml/index.html