Building complex object-oriented systems with patterns and XP

Eduardo B. Fernandez

Abstract

Many important systems are complex. These systems have a large number of interacting entities, complex constraints, and need to satisfy nonfunctional requirements. We use Semantic Analysis Patterns (SAPs), a type of analysis patterns where each pattern corresponds to a basic set of use cases, to build a global conceptual model in an incremental way. This global model provides XP with a structure where global aspects such as distribution, security, testability, and others can be considered. SAPs can also be used at each incremental stage in XP to guarantee the application of good software development principles.

1 Introduction

XP has been used mostly to build systems of medium or small complexity. Because of its fine-grain increments, one or a few classes each time [Bec00,Mar01], it is difficult to consider global aspects such as distribution, authorization, and concurrency. However, many important systems, e.g., manufacturing, vehicle navigation, business planning, are quite complex. Their complexity comes from a large number of interacting entities, many relationships between their units, complex constraints on the values of their variables, and the need to satisfy nonfunctional requirements, which normally implies dealing with concurrency and distribution aspects.

The lack of global view in XP comes from its emphasis in immediate implementation, without explicit analysis and design stages. While not explicitly excluded, there is little incentive for modeling; typically, partial models are used and then discarded [Fow01]. This precludes finding commonalities and optimizations in the global conceptual model. Refactoring cannot correct some conceptual problems or consider nonfunctional aspects.

Patterns let us start a conceptual model in the right direction and use the knowledge of others. If the patterns used are carefully selected, they embody good design principles, and a designer can apply these principles transparently [Fer00], which results in high quality models. After this, there is a variety of architectural patterns that can be used to deal with design aspects.

We propose building first the conceptual model and the tests for the complete system (or close to it) and postpone the start of implementation until this model is relatively complete. This allows the designers to consider global aspects and build more complex applications. Classes can still be built and tested incrementally, but now we have the guidance of a global model. We have presented earlier the concept of Semantic Analysis Patterns, SAPs [Fer00a], mini-applications corresponding to a basic set of use cases or user stories; they can be used to build the global model incrementally. They can also be used at each XP stage to guarantee the application of good software development principles.

2 Semantic Analysis Patterns

An analysis pattern is a set of classes and associations that have some meaning in the context of

an application; that is, it is a conceptual model of a part of the application. However, the same structure may be valid for other applications, and this aspect makes them very valuable for reuse and composition. Analysis patterns can be atomic or composite [Rie97]. Analysis patterns have been studied in [Coo94] and [Fow97], inspired by the work of Hay [Hay96]. In particular, the analysis patterns discussed in these references are atomic patterns, composed of a few classes; our interest is in larger patterns. Larger patterns can be more effective with respect to reusability and can be used as building blocks for conceptual models. We have proposed a type of composite patterns that we call Semantic Analysis Patterns, SAPs[Fer00a]. A SAP is a minimal application that corresponds to a few basic use cases and defines a semantic unit that can be combined with other SAPs to build complex systems. The use cases are selected in such a way that the pattern can be applied to a variety of applications. We have developed several SAPs, including patterns for inventories [Fer00b], for reservation and use of reusable entities [Fer99], for order processing [Fer00c], and others.

3 Development of SAPs

SAPs are based on well-known principles such as abstraction, composition, minimal coupling, and regularity. They also incorporate other principles, not usually included in most methods, such as authorization, precision, and testability [Fer00]. We illustrate here the use of some of these principles in the development of a SAP for inventories.

Abstraction

Abstraction is the most fundamental principle of the object-oriented approach. It implies including only the essential aspects of a model, leaving out details which are not relevant or

needed. We illustrate this concept with a basic model for an inventory system for discrete items. Its basic functions can be summarized as:

1) Keep track of different varieties of stocks. Keep track of the quantities of each item in stock. Several kinds of quantities may be needed, e.g. onHand and Available.

2) Keep track of the locations of items. The inventory should record the distribution of items in specific locations. Locations could be subdivided for easier location of items. Each flow of stock material between locations should be reflected in the inventory distribution.

The most basic inventory model just keeps track of the quantities of some type of item in stock (Figure 1). The items in this inventory can be finished products, components used in manufacturing, machinery, etc.; in other words, anything of whose existence and quantity we want to be aware. Each item belongs to a unique type, ItemType. This is a fundamental abstraction that corresponds to requirement 1). Any inventory system model must have this model as a component. Similarly, we can make abstractions of any other requirements. To realize Requirement 2 we need to break down the inventory quantity into location quantities (usually there exist several locations where an inventory item can be stored). This model is shown in Figure 2.

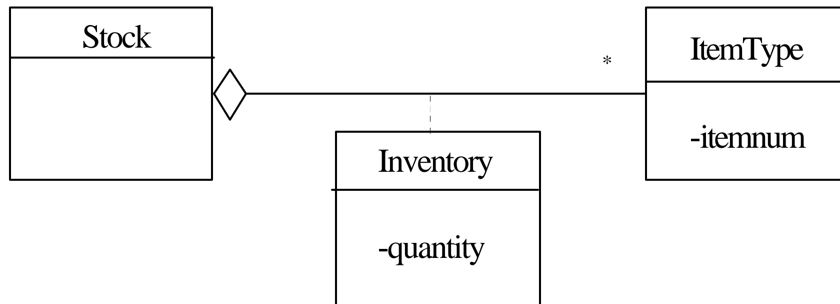

Figure 1. A basic inventory system

We can consider the diagrams of Figures 1 and 2 to be atomic analysis patterns. They describe two abstractions that correspond to two fundamental use cases:

- Keeping track of quantities of discrete items (Figure 1).
- Keeping track of the locations of those items (Figure 2).

Each stage of XP typically implements one or two of these use cases or stories.

Decomposition

To be able to model a complex system we need to apply decomposition, the divide and conquer principle. For example, a manufacturing system is far too complex to handle as a single unit. A decomposition for this system would consist of a set of UML packages such as Inventory Control, Shop Orders, Customer Orders, Shipping, and Money. The interactions between packages are important to define the needed operations in the classes of each package. Decomposition is very important for XP, each stage implies an explicit or implicit decomposition of the requirements.

Composition

Patterns can be composed to build larger models [Fer00]. For example, we can compose the patterns of Figures 1 and 2 to form a larger pattern that keeps track of stock quantities and their distribution. Figure 3 shows the composite pattern (this includes authorization, discussed below). Clearly, this principle is also basic for XP, to put together the results of each stage.

Projection

Projection is the combination of different diagrams or views of the system to provide a more complete picture. It can be resembled to the need to describe a mechanical piece using different views or projections; a single view would give a distorted and incomplete picture of its 3-dimensional properties. This is a principle not emphasized XP. In particular, complex applications need several diagrams to be fully understood.

Minimal Coupling

This principle is frequently used in design patterns, where a main objective is flexibility [Gam95]. It is also heavily used in Fowler's analysis patterns [Fow97]. The idea is to separate some aspects to allow them to evolve independently.

In the example, if we had a detailed description of the structure of items we could use the Composite pattern [Gam95] to describe this structure recursively. This would decouple the structural aspects of a component or product from more basic aspects. We could also decouple other aspects of the item description.

Precision

A model such as the one of Figure 1 may not be precise enough to represent some of the constraints in the requirements. For example, to indicate that the sum of inventory quantities must be constant in inventory transfers between locations we need to add a UML constraint. This can be added to class Inventory as

{sum of quantities must be constant}

However, this is not precise enough for many cases. If we had more complex constraints, statements of this form may be ambiguous. This ambiguity cannot be tolerated in safety-critical applications, for example. Both Z [Coo94] and OCL [Wan98], have been used to add precision to UML constraints. A comparison of Z and OCL as constraint languages is given in [Jia99].

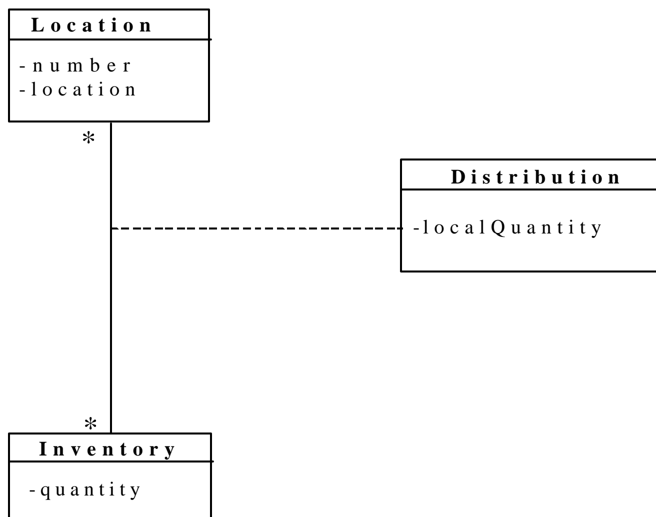

Figure 2. Distribution of items.

Authorization

When we build systems using atomic patterns or SAPs we know the actor roles in the use cases that correspond to these units. These actors need specific rights to perform the functions defined in the use case. We can then define authorization rules according to the principles of Role-Based Access Control (RBAC) [Fer97]. For example, in the inventory we could include the authorizations shown in Figure 3, where classes StockManager, etc., indicate roles (denoted by the UML stereotype <<role>>). For example, here the InventoryManager is authorized to transfer stock between locations.

Testability

Use cases (or user stories) correspond to the necessary interactions with the system and define sequences of actions that can be used to test the system [Jac98]. A system defined following use cases is then implicitly highly testable. For example, for the inventory we could build test cases to follow the actions of the sequence diagrams of this application [Fer00] and check if a Shop Order changes state and the inventory quantities get updated when specific events occur.

4 SAPs and XP

The explicit use of good software development principles requires considerable experience and is prone to error. For most users, a better way to apply the principles is implicitly, by building a conceptual model through the application of SAPs or another appropriate methodology. As indicated in [Fer00a] a procedure to build a conceptual model would be as shown below. We assume we have a catalog of atomic patterns and SAPs. We examine the use cases and/or other requirements and:

- Look for SAPs. We look first for patterns that match exactly or closely the requirements. Then we try to find analogous patterns that may apply.
- Look for atomic patterns.

This procedure results in a skeleton, where some parts of the model are fairly complete while other portions are partially covered or not covered at all. We still need to cover the rest of the model in an ad hoc way but we already have a starting model. Naturally, we can still add design patterns in the design stage.

The point here is that a user applying SAPs is implicitly applying the principles if the patterns have been built by careful application of these principles. This should result in a good quality conceptual model. This model can be used to guide an XP design by relating each new class to be built to its place in the global order. Individual SAPs can be used at each stage in XP. A well-built conceptual model is easy to change and is well suited to changing requirements. It is also highly reusable. Some nonfunctional aspects should be defined or specified in the global conceptual model, e.g., security as shown above. Patterns for distribution, concurrency, real-time, and fault tolerance can also be used at this level. Of course, the lower architectural levels must implement and enforce the nonfunctional constraints defined in the conceptual model. This can be done through the use of more specialized patterns [Fer01].

5 Conclusions

We have developed several atomic patterns and SAPs that incorporate principles of good design and we are producing a catalog of analysis patterns that can be used to produce good quality conceptual models even by inexperienced designers. All this can be used as a basis for XP

development, where the SAPs can guide each stage by relating the partial implementations to the complete system model. This global model can be used as a reference to decide about aspects such as distribution, security, and other nonfunctional aspects. The incremental nature of XP is likely to produce redundancies and a global model can avoid many of them. SAPs are being tested with students at two universities but industrial tests are necessary. Initial results have shown that students learning object-oriented concepts can develop rather complex models using SAPs, what we need to verify next is that having these conceptual models helps XP build more complex systems.

References

[Bec00] K. Beck, *Extreme Programming explained: Embrace change*, Addison-Wesley 2000.

[Coa97] P. Coad, "*Object models: Strategies, patterns, and applications*" (2nd Edition), Yourdon Press, 1997.

[Coo94] S. Cook and J. Daniels, "Let's get formal", *JOOP*, July-August 1994, 24 and 64-66.

[Fer97] E.B.Fernandez and J.C.Hawkins, "Determining role rights from use cases", *Procs. of the 2nd ACM Workshop on Role-Based Access Control*, November 1997, 121-125.

[Fer99] E.B.Fernandez and X.Yuan, "An analysis pattern for reservation and use of entities, *Procs. of PLoP99* , http://jerry.cs.uiuc.edu/~plop/plop99

[Fer00] E.B. Fernandez, "*Principles for Building Complex Object-Oriented Conceptual Mod*els", Tech. Report TR-CSE-00-24, Dept. of CSE, Florida Atlantic University, August 2000.

[Fer00a] E.B.Fernandez, and X.Yuan, "Semantic analysis patterns", *Procs. of 19th Int. Conf. on Conceptual Modeling*, *ER2000*, 183-195.

[Fer00b] E.B. Fernandez, "Stock Manager: An analysis pattern for inventories", *Procs. of PLoP 2000,* http://jerry.cs.uiuc.edu/~plop/plop2k

[Fer00c]] E.B.Fernandez, X.Yuan, and S.Brey, "An analysis pattern for the order and shipment of a product", *Procs.of Pattern Languages of Programs Conf. (PLoP'2000),* http://jerry.cs.uiuc.edu/~plop/plop2k

[Fer01] E B. Fernandez and R.Y. Pan, "A pattern language for security models", *Procs. of PloP 2001*, http://jerry.cs.uiuc.edu/~plop/plop2001/accepted_submissions/accepted-papers.html

[Fow97] M. Fowler, *Analysis patterns -- Reusable object models*, Addison- Wesley, 1997.

[Fow01] M. Fowler, "Is design dead?", *Software Development*, April 2001, 42-46.

[Gam95] E.Gamma, R.Helm, R. Johnson, and J.Vlissides, *"Design patterns –Elements of reusable object-oriented software",* Addison-Wesley, 1995.

[Hay96] D.C.Hay, *Data model patterns -- Conventions of thought*, Dorset House Publishing, New York, 1996.

[Jac98] I. Jacobson, G. Booch, and J.Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1998.

[Jia99] Z. Jiang, E.B. Fernandez, and J. Wu, *"Comparing OCL and Z as constraint language for UML",* Tech. Report. TR-CSE-99-28, Dept. of CSE, Florida Atlantic University, May 1999.

[Mar01] R.C.Martin and R.C.Koss, "The Bowling Game. An example of test-first pair programming, ", http://www.objectmentor.com/publications/articlesByDate.html

[Rie97] D.Riehle, "Composite design patterns", *Procs. of OOPSLA'97*, 218-228.

[Wan98] J. Wanner and A.Kloppe, *The Object Constraint Language: Precise modeling with UML*, Addison-Wesley 1998.

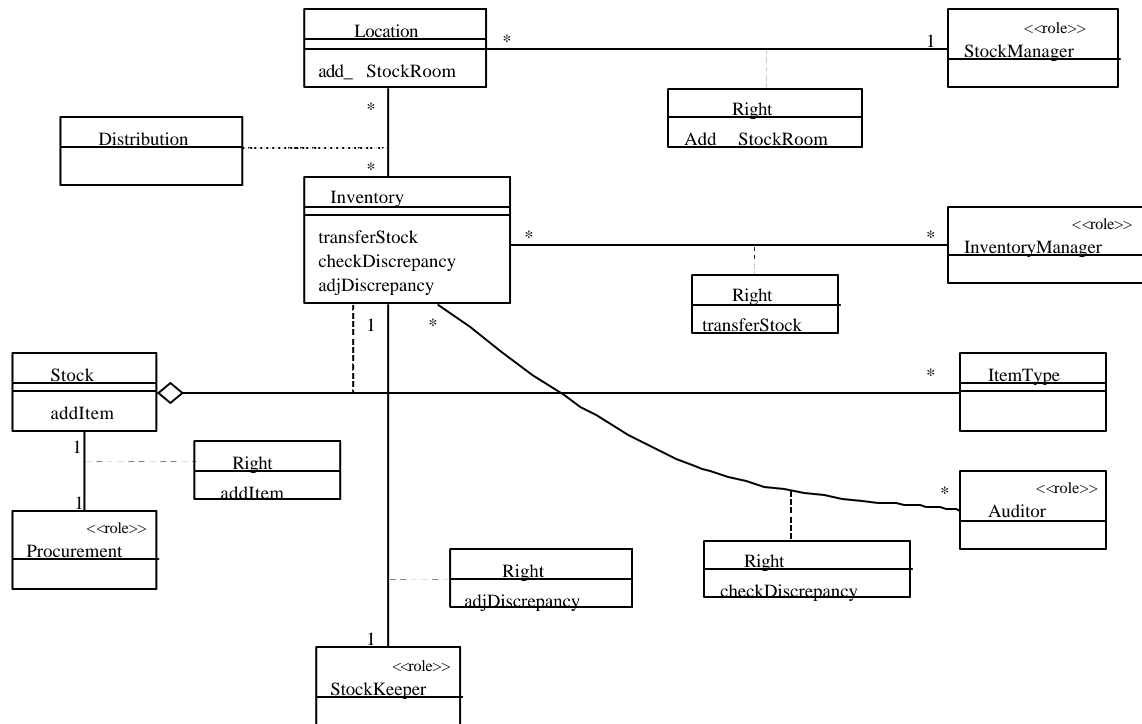

Figure 3. Authorized inventory.

Keywords

Analysis patterns, complex systems, conceptual models, Extreme Programming, lightweight process, object-oriented analysis, patterns, use cases.