Semantic Analysis Patterns

Eduardo B. Fernandez and Xiaohong Yuan

Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL 33431
{ed, xhyuan}@cse.fau.edu

Abstract. The development of object-oriented software starts from requirements expressed commonly as Use Cases. The requirements are then converted into a conceptual or analysis model. Analysis is a fundamental stage because the conceptual model can be shown to satisfy the requirements and becomes the skeleton on which the complete system is built. Most of the use of software patterns until now has been at the design stage and they are applied to provide extensibility and flexibility. However, design patterns don't help avoid analysis errors or make analysis easier. Analysis patterns can contribute more to reusability and software quality than the other varieties. Also, their use contributes to simplifying the development of the analysis model. In particular, a new type of analysis pattern is proposed, called a Semantic Analysis Pattern (SAP), which is in essence a miniapplication, realizing a few Use Cases or a small set of requirements. Using SAPs, a methodology is developed to build the conceptual model in a systematic way.

1 Introduction

The development of object-oriented software starts from requirements expressed normally as Use Cases [17]. The requirements are then converted into a conceptual or analysis model. Analysis is a fundamental stage because the conceptual model can be shown to satisfy the requirements and becomes the skeleton on which the complete system is built. No good design or correct implementation is possible without good analysis, the best C++ or Java programmers cannot make up for conceptual errors. The correction of analysis errors becomes very expensive when these errors are caught in the code. It is therefore surprising how poorly understood is this stage and how current industrial practice and publications show a large number of analysis errors [6]. We have found that industrial software developers usually have trouble with analysis. What is worse, even serious journals and conferences publish papers or tutorials that contain clear analysis errors.

A possible improvement to this situation may come from the use of patterns. A pattern is a recurring combination of meaningful units that occurs in some context. Patterns have been used in building construction, enterprise management, and in several other fields. Their use in software is becoming very important because of their value for reusability and quality; they distill the knowledge and experience of many designers.

Most of the use of patterns until now has been at the design stage. However, design patterns don't help to avoid analysis errors or to make analysis easier. We believe that we need analysis patterns to improve the quality of analysis and they can contribute more to reusability and software quality than the other varieties. We also intend to show that their use contributes to simplifying the development of the application analysis model. In particular, we propose a new type of analysis pattern, called a Semantic Analysis Pattern (SAP), which is in essence a miniapplication, realizing a few Use Cases or a small set of requirements [8]. Using SAPs we develop a methodology to build the conceptual model in a systematic way. We use UML (Unified Modeling Language) [2], as a language to describe our examples.

Section 2 introduces SAPs and how they are obtained. We show analogy and generalization as ways to develop SAPs. Section 3 describes how SAPs are used in producing conceptual models from Use Cases. Section 4 compares SAPs to other varieties of analysis patterns and evaluates their use. A last section presents conclusions and suggestions for future work.

2 Analysis Patterns and Their Use

2.1 Semantic Analysis Patterns

The value of analysis is played down in practice. The majority of the papers published about object-oriented design as well as the majority of textbooks concentrate on implementation. Books on Java, C++, and other languages outnumber by far the books on object-oriented analysis/design (OOA /OOD). On top of that, most books on OOA/OOD present very simple examples. To make things worse, professional programmers need to implement as soon as possible, there is pressure to show running code and they may skip the analysis stage altogether. What is deceiving is that software may appear to work properly but may have errors, not be reusable or extensible, be unnecessarily complex. In fact, most of the software built without some model exhibits some or all of these defects. Most schools emphasize algorithms, not the development of software systems. There is a large literature on methods of system development that although oriented to other disciplines [24], is very applicable to software, but rarely used (In fact, design patterns originated from ideas about buildings). Some people believe that with components we don't need to understand what is inside each component. The result of all this is that analysis is skipped or done poorly.

We need to look for ways to make analysis more precise and easier for developers. The use of patterns is a promising avenue.

A Semantic Analysis Pattern is a pattern that describes a small set of coherent Use Cases that together describe a basic generic application. The Use Cases are selected in such a way that the application can fit a variety of situations.

Semantic Analysis Patterns differ from design patterns in the following ways:

- Design patterns are closer to implementation, they focus on typical design aspects, e.g., user interfaces, creation of objects, basic structural properties.
- Design patterns apply to any application; for example, all applications have user interfaces, they all need to create objects.
- Design patterns intend to increase the flexibility of a model by decoupling some aspects of a class.

An instance of a SAP is produced in the usual way: Use Cases, class and dynamic diagrams, etc. We select the Use Cases in such a way that they leave out aspects which may not be transportable to other applications. We can then generalize the original pattern by abstracting its components and later we derive new patterns from the abstract pattern by specializing it (Figure 1). We can also use analogy to directly apply the original pattern to a different situation.

We illustrate these two approaches in the next sections. We develop first a pattern from some basic use cases. We then use analogy to apply it to a different situation, then we generalize it and finally we produce another pattern for another application specializing the abstract pattern. We then show how to use these patterns in building conceptual models.

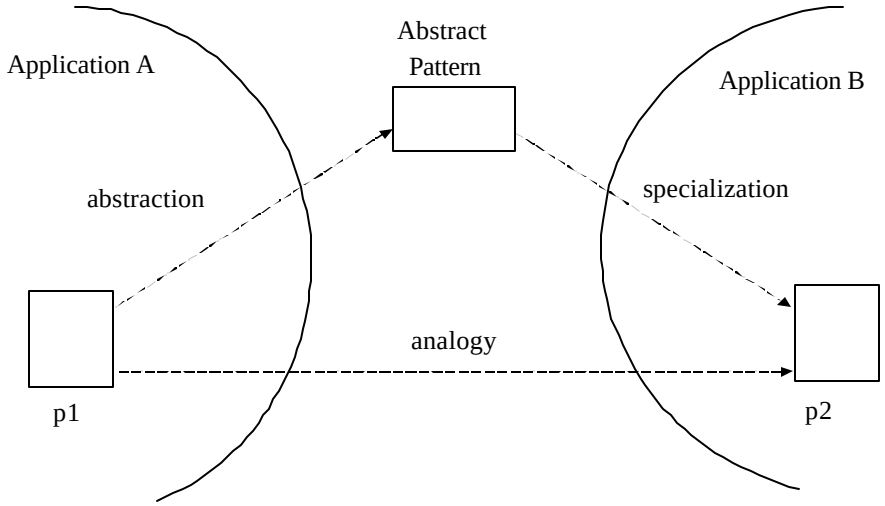

Figure 1. Pattern generation

2.2 An Example

In a project we developed a design for a computer repair shop. The specifications for this application are: A computer repair shop fixes broken computers. The shop is part of a chain of similar shops. Customers bring computers to the shop for repair and a reception technician makes an estimate. If the customer agrees, the computer is assigned for repair to some repair technician, who keeps a Repair Event document. All the Repair Event documents for a computer are collected in its repair log. A repair

event may be suspended because of a lack of parts or other reasons.

These requirements correspond to two basic Use Cases:

- Get an estimate for a repair
- Repair a computer

A class diagram for this system is shown in Figure 2, while Figure 3 shows a state diagram for Repair Event. Figure 4 shows a sequence diagram for assigning the repair of some computer to a technician. The class diagram reflects the facts that a computer can be estimated at different shops in the chain and that one of these estimates may become an actual repair. A computer that has been repaired at least once has a repair log that collects all its repair events. The collection of repair shops is described by the repair shops chain.

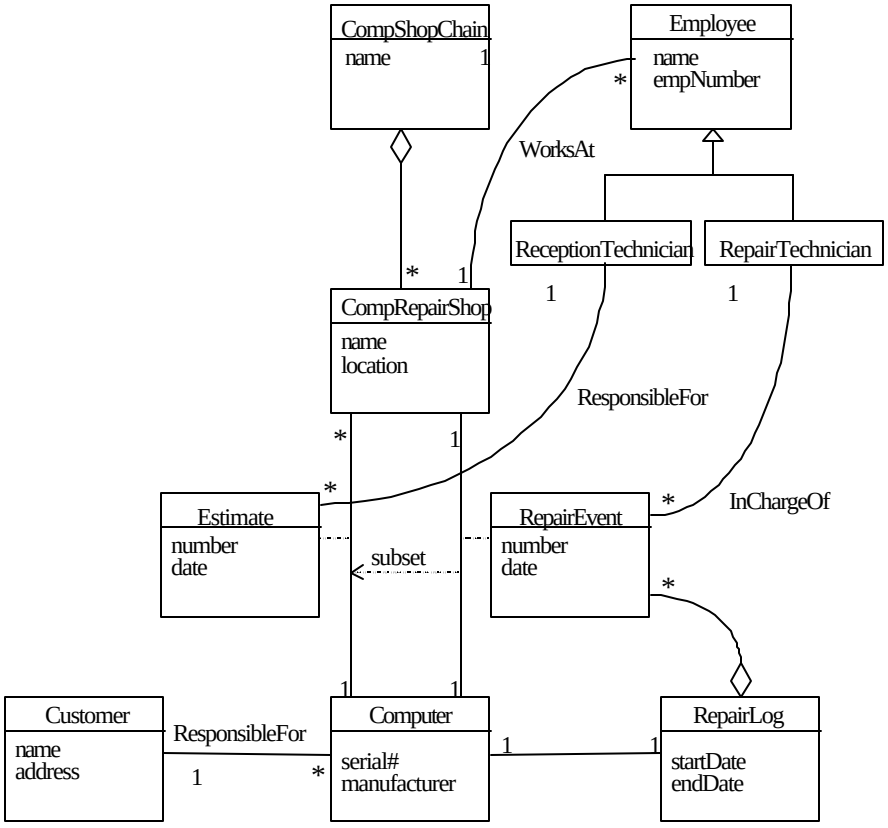

Figure 2. Class diagram for the computer repair shop.

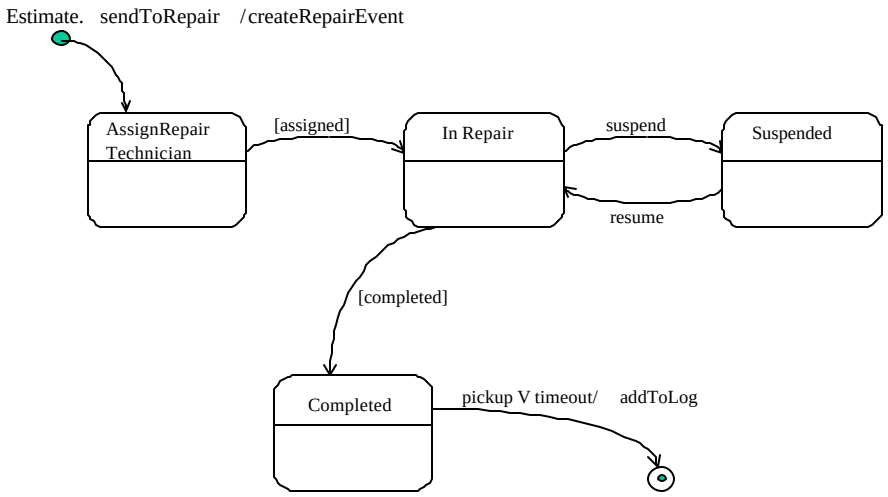

Figure 3. State diagram for Repair Event

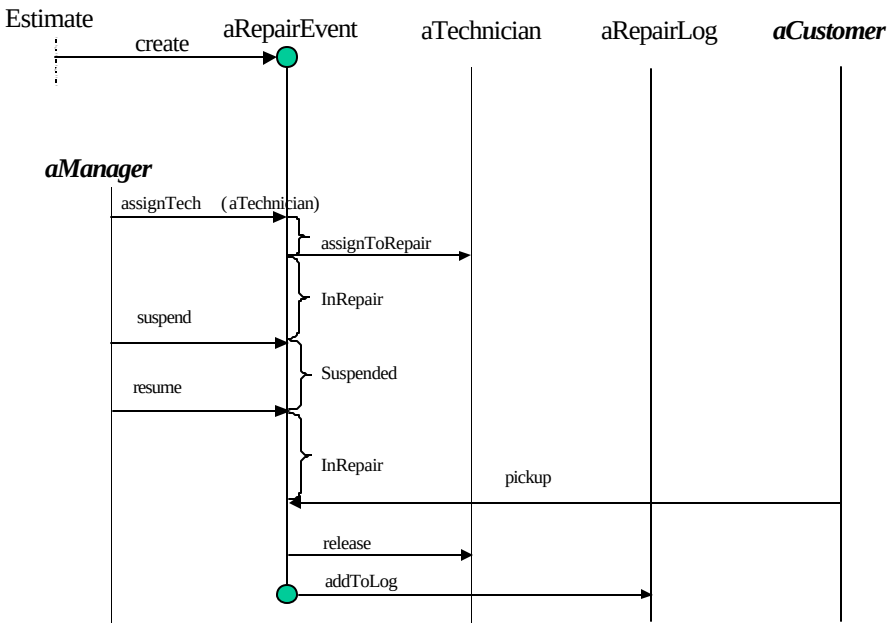

Figure 4. Sequence diagram for assigning repair jobs to technicians

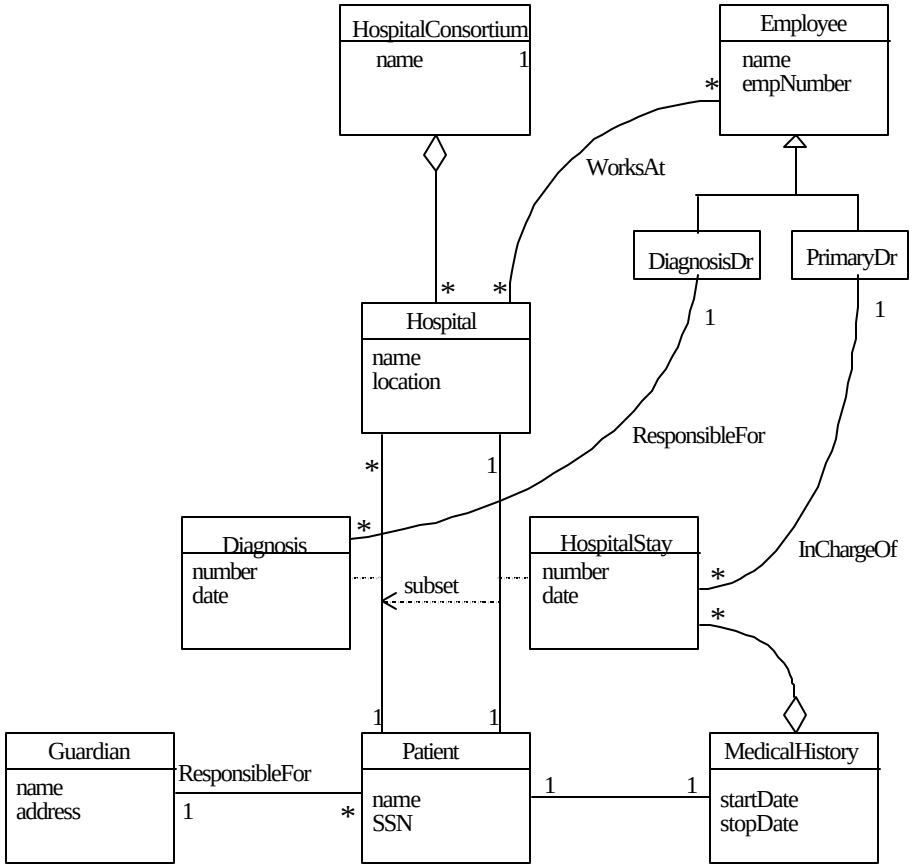

Figure 5. Class diagram for hospital registration.

2.3 Looking for Analogies

In a later project we needed to design a system to handle patient registration in a hospital. Noticing that a hospital, instead of broken computers, fixes sick people, we arrived at the class diagram of Figure 5. We just needed to reinterpret each class in the repair shop as a corresponding class for a hospital. For example, the computer becomes a patient, the estimate becomes the diagnosis, the repair event a hospital stay, etc. Similarly, sequence diagrams and state diagrams are developed in analogy with those used in computer repair [8].

2.4 Pattern Generalization

We can generalize the patterns of Figures 2 and 5 by noticing that their essential actions are:

- Application to a collection of places
- Selection of one place to stay
- Keep a Stay record for each stay.
- Keep History of stays
- Personnel is assigned to the evaluation of applications and to do something during the stays.

With these concepts we can define the abstract pattern of Figure 6, that includes the two previous patterns. Here the specific institutions have become generic institutions, patients have become applicants, etc. This pattern fits a wide range of applications and we can use it in a new application, student admissions in a university (Figure 7). Here, we particularize institution into university, applicant into student, etc.

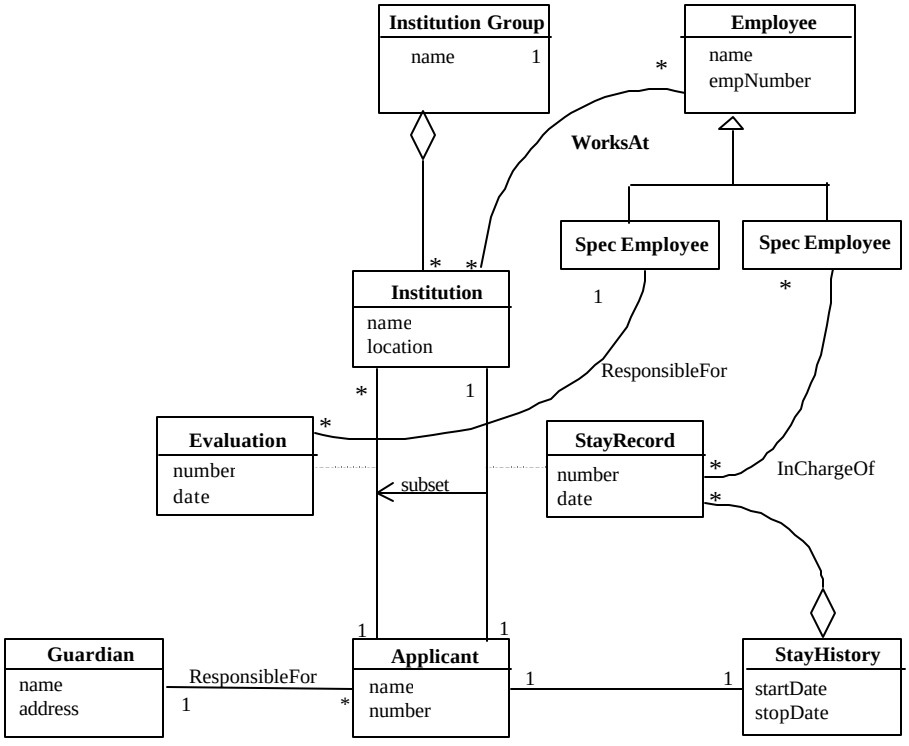

Figure 6 Admissions pattern

One can take portions of these patterns and find simpler patterns. For example, the set of personnel-related classes in Figures 2 and 5 can define a Personnel pattern, the

collection pattern appears twice, as a collection of shops and as a collection of repair events. Also, some design patterns, e.g., the composite pattern [13], are useful in analysis.

3 Analysis Method Using SAPs

To use the methodology it is necessary to have first a good collection of patterns. We have developed four analysis patterns [8], [9], [10], [11]. We are also collecting the most interesting analysis patterns that have appeared in the literature [1], [3], [4], [18], [21], [27].

A possible analysis method using SAPs is described now. We assume we have a catalog of concrete and abstract patterns as well as catalogs of subpatterns, Fowler-style patterns[1], and design patterns. We examine the Use Cases and/or other requirements and:

- Look for SAPs. We look first for concrete patterns that match exactly or closely the requirements. Then we try to specialize analogous or abstract patterns that may apply. This stage is shown in Figure 8, where patterns p3, p5,…,have been identified and cover some of the requirements.
- Look for smaller patterns, such as the subpatterns of Figures 2 and 5.
- See if there are appropriate design or architectural patterns. As indicated earlier, some design or architectural patterns may be useful in analysis.
- Add Fowler-style patterns for flexibility and extensibility. This involves examining classes for possible breakup.

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.

As an example, consider the following requirements: We need a system to handle the Soccer World Cup. Teams represent countries and are made up of 22 players. Countries qualify from zones, where each zone is either a country or a group of countries. Each team plays a given number of games in a specific city. Referees from different countries are assigned to games. Hotel reservations are made in the city where the teams play.

Figure 9 shows that this model was almost completely covered with the following patterns:

1) An instance of the composite pattern [13]
2) An instance of the collection pattern (a subpattern of Figure 2)
3) An instance of the reservation pattern [9]
4) Another instance of the collection pattern
5) Another instance of the reservation pattern

In addition to these patterns one needs several associations to connect them. These

[1] Fowler tries to increase flexibility by decoupling some aspects of a class into a separate class, e.g., the physical characteristics of a person would be separated from class Person [12]. His patterns in general are small, two or three classes.

associations correspond to specific requirements , e.g., a referee represents a country, a game is played in a given city. Of course, this is a simple example, larger examples are not so easily covered. However, the example exposes the flavor of the methodology.

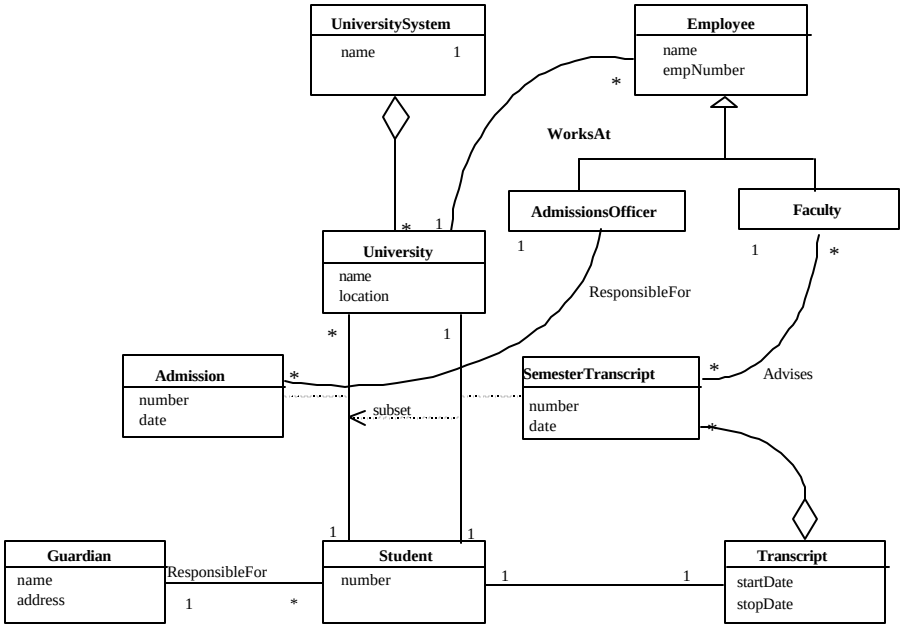

Figure 7. Student admissions

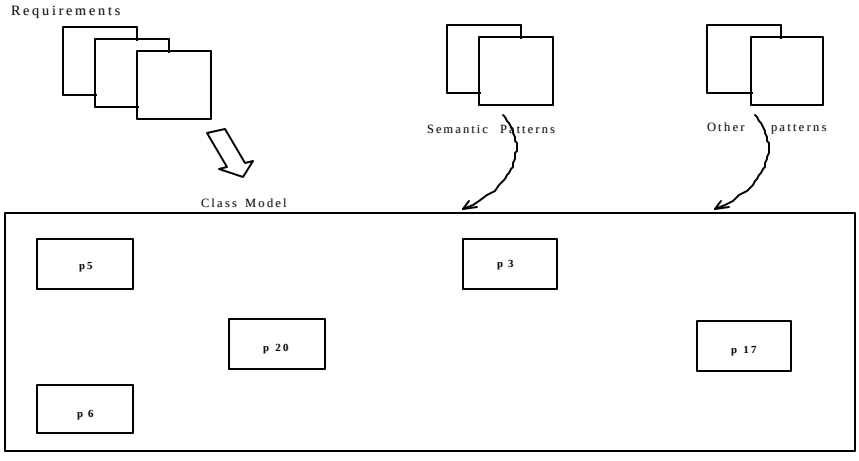

Figure 8. Use of semantic APs

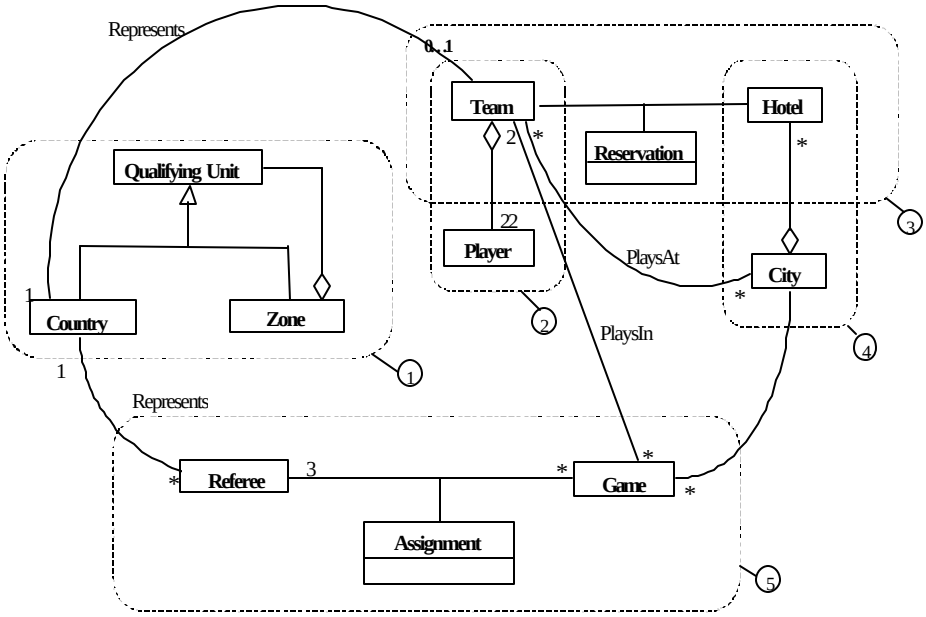

Figure 9. Analysis model for World Cup example

4 Discussion

The first work on object-oriented analysis patterns is due to P. Coad [5]. More influential has been the work of M. Fowler, who produced the first book on this topic [12]. His patterns (and Coad's) emphasize similar objectives as design patterns, i.e., to decouple parts of a class to increase the model flexibility and extensibility. These authors consider some dynamic aspects in the form of sequence diagrams but do not use statecharts. R. Johnson and his group at the University of Illinois have developed several analysis patterns [19], including banking and security; in general, they follow Fowler's style. Nature's project in the UK intends to classify application requirements into problem domains [22]. Their emphasis is not on modeling although they describe requirements using class models. Note that our approach results in much fewer patterns since we can abstract patterns. Another related work is the book of D.Hay [15], where he describes data patterns; however, he doesn't consider dynamic aspects.

All these projects have different objectives; in particular they do not emphasize synthesis of complex models. They are certainly a source of possible patterns and they are being mined to build the SAP catalog. The design patterns group shows in its book [13] and in a paper [14], another approach to synthesis: start from a class or two

and keep adding patterns until the requirements are satisfied[2]. This approach doesn't appear very feasible when designing systems with long and complex specifications. However, it could be useful in the latter stages of our approach. Synthesis of complex systems is also the objective of a project at Georgia State University [23], and some of their ideas could be useful in this work.

Methods for synthesis of models using SAPs could result in considerable improvement to the quality of the software produced in industry. Design patterns have had a strong effect in design, we need now to do the same for analysis. Other than improving the analysis modeling there are more benefits:

- Test cases are developed from Use Cases but a good conceptual model helps define the needed preconditions and postconditions. Traceability can also be improved.
- Testing of object-oriented systems involves inspection of the models developed at each stage. In particular, domain models require careful validation [20]. The use of SAPs can help make these inspections faster and more accurate.

The actors of the use cases in the minimal application must be given the required rights to perform their functions. We can consider actors as roles and if we assign rights accordingly we have a Role-Based Access Control model of security [7]. We can define in this way "authorized SAPs', that include the needed authorizations.

The Object Constraint Language can be used to define precise constraints in the models [26]. Using OCL the minimal application is defined more precisely using business rules. Authorizations can also be expressed more precisely.

5 Conclusions

A good analysis model for a portion of a complex system can be abstracted and become an analysis pattern that can be used in other applications[1]. Analogy and abstraction play an important role in reusing an analysis pattern. Subsets of a pattern may also have their own application in other situations. All this can save time and improve the quality of a system. One of the most difficult steps in practice is to get an initial model; this approach makes it easier. In subsequent steps the initial model can be modified to suit better the needs of the application. An analysis model using patterns is easier to understand and to extend. It should also result in a higher quality design. A software architecture constructed this way is more reusable and extensible than an architecture defined directly from the requirements or where patterns are applied later. Note that SAP-based development is different from domain analysis, SAPs cut through several domains.

There are several aspects that we are developing or intend to develop:

- The design methodology has been applied to relatively small examples and it appears useful, but we need larger examples. We are collecting large-system specifications that we intend to model.

[2] A similar approach is used by Johnson's group [28].

[1] We can think also that a SAP is a pattern recurrent in several frameworks.

- Related SAPs can be combined into frameworks. For example, Order Entry, Inventory, and Reservations could make up a manufacturing framework. We have used this approach with security patterns [16], but we have not applied it to SAPs.
- Pattern languages. The SAPs by their nature leave out many aspects. A collection of related patterns, a pattern language, is needed to cover a domain or a significant part of it. We are developing a pattern language for reservation and use of entities, to complement the basic reservation SAP [9].
- SAPs as composite design patterns. A composite design pattern [25] is a pattern composed of other patterns where the composite is more than the sum of its parts. SAPs can be studied as special cases of composite design patterns.

References

1. Arsanjani, A.: Service provider: A domain pattern and its business framework implementation, *Procs. of PloP'99*. http://st-www.cs.uiuc.edu/~plop/plop99
2. Booch, G., Rumbaugh, J., Jacobson, I.: *The Unified Modeling Language User Guide*, Addison-Wesley 1998.
3. Braga, R.T.V., Germano, F.S.R., Masiero, P.C.: A confederation of patterns for resource management, *Procs. of PLoP'98*, http://jerry.cs.uiuc.edu/~plop/plop98
4. Braga, R.T.V., Germano, F.S.R., Masiero, P.C.: A pattern language for business resource management, *Procs. of PloP'99*, http://st-www.cs.uiuc.edu/~plop/plop99
5. Coad, P.: *Object models – Strategies, patterns, and applications* (2nd. Edition), Prentice-Hall 1997
6. Fernandez, E. B.: Good analysis as the basis for good design and implementation, Report TR-CSE-97-45, *Dept. of Computer Science and Eng., Florida Atlantic University*, September 1997. Presented at OOPSLA'97
7. Fernandez, E.B., Hawkins, J.: Determining role rights from use cases, *Procs. 2nd ACM Workshop on Role-Based Access Control*, 1997, 121-125
8. Fernandez, E.B.:Building systems using analysis patterns, *Procs. 3rd Int. Soft. Architecture Workshop (ISAW3)*, Orlando, FL , November 1998 , 37-40
9. Fernandez, E.B, Yuan, X.: An analysis pattern for reservation and use of entities, *Procs.of PLoP99* , http://st-www.cs.uiuc.edu/~plop/plop99
10. Fernandez, E. B.: Stock manager: An analysis pattern for inventories, *Procs. of PLoP 2000.*
11. Fernandez, E. B., Yuan, X., Brey, S.: Analysis Patterns for the Order and Shipment of a Product, *Procs. of PLoP 2000.*
12. Fowler, M.: *Analysis patterns -- Reusable object models*, Addison- Wesley, 1997
13. Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design patterns –Elements of reusable object-oriented software*, Addison-Wesley 1995
14. Gamma, E., Beck, K.: JUnit: A cook's tour, *Java Report,* May 1999, 27-38
15. Hay, D.: *Data model patterns-- Conventions of thought*, Dorset House Publ., 1996
16. Hays, V., Loutrel, M., Fernandez, E.B.: The Object Filter and Access Control Framework, to appear in *Procs. of PLoP 2000*
17. Jacobson, I., Booch, G., Rumbaugh, J.: *The Unified Software Development Process*, Addison-Wesley 1999
18. Johnson, R., Woolf, B.: Type Object, Chapter 4 in *Pattern Languages of Program Design 3*, Addison-Wesley, 1998

19. Johnson, R.: http://st-www.cs.uiuc.edu/users/Johnson
20. McGregor, J. D.: Validating domain models, *JOOP*, July-August 1999, 12-17
21. Mellor, S.J.: Graphical analysis patterns, *Procs. Software Development West98*, February 1998. http://www.projtech.com
22. Nature Project. http://www.city.ac.uk/~az533/main.html
23. Purao, S. and Storey, V.: A methodology for building a repository of object-oriented design fragments. *Procs. of 18th International Conference on Conceptual Modeling (ER'99)*, 203-217.
24. Rechtin, E.: The synthesis of complex systems, *IEEE Spectrum*, July 1997, 51-55
25. Riehle, D.: Composite design patterns, *Procs. of OOPSLA'97*, 218-228.
26. Wanner, J., Kloppe, A.: *The OCL: Precise modeling with UML*, Addison-Wesley 1998
27. Yoder, J. and Johnson, R.: *Inventory and Accounting patterns*, http://www.joeyyoder.com/marsura/banking
28. Yoder, J., Balaguer, F.: Using metadata and active object-models to implement Fowler's analysis Patterns, Tutorial Notes, OOPSLA'99