IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Suite 200, Hershey PA 17033, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP4572

Analysis Pattern Definition in the UML

Ernest Teniente
Universitat Politècnica de Catalunya
Dept. Llenguatges i Sistemes Informàtics
Jordi Girona 1-3, 08034 Barcelona (Catalonia)
teniente@lsi.upc.es

ABSTRACT

We identify the UML diagrams and elements that must be used to define an analysis pattern and we explain how analysis patterns defined in this way can be used in the context of the Unified Process. Our proposal is illustrated by means of an example aimed at modelling a generic sports competition.

1. INTRODUCTION

A pattern identifies a problem and provides the specification of a generic solution to that problem. The use of patterns in software development increases reusability of software components and reduces errors of the software delivered. Patterns can be used at each stage of the software development process. Thus, we distinguish among analysis, architectural, design and language patterns.

A *design pattern* [GHVJ95] describes the structure of the solution to a problem that appears repeatedly during software design and the interaction between the different components involved in the solution. Therefore, a design pattern is domain-independent since it is applicable to any software system, provided that the problem addressed by the pattern is encountered during the design of that system.

The analysis model constitutes a permanent model of the reality in itself and, as such, it is independent of a particular implementation technology [Pre00, Mac01]. Therefore, an *analysis pattern* must describe both the structural and dynamical properties of a basic, generic, application domain as perceived by the system user. In this sense, an analysis pattern is application dependent since its semantics describes specific aspects of some domain or software system [Fer98].

The general structure provided by an analysis pattern can be used to define several software systems sharing the features described by the pattern. Developing a particular system applicable to a specific application domain corresponds to adapt the pattern to take the specific aspects of the domain into account.

Analysis patterns provide several advantages to software development. First, they reduce the costs of information systems development because of the reuse of existing solutions. Second, they speed up the development of concrete analysis models that capture the main requirements of a generic application domain. Third, they improve the quality of analysis models by favouring reusability and reducing software errors.

Although some authors like [Fer98, Fow99, FY00] have used the UML to show examples of analysis patterns, it does not exist yet, as far as we know, a precise statement of the UML diagrams that must conform an UML analysis pattern. This is the main goal of this paper: to determine the UML diagrams that must be specified to define an UML analysis pattern.

Moreover, we identify some UML elements that may not be present in these diagrams to ensure we develop an analysis model and we explain how our analysis patterns can be used according to the Unified Process [JBR99, Lar02]. Our proposal is illustrated by means of an analysis pattern that models a generic sports competition.

2. ANALYSIS PATTERNS AND THE UML

Unfortunately, we do not find a clear agreement regarding the kind of patterns that can be defined at the analysis level of information systems development. For instance, [Fow97] proposes analysis patterns that define appropriate solutions to model specific constructs that may be found during the specification of different information systems. On the other hand, [FY00] proposes patterns that define a conceptual model for a single information system domain.

In fact, we may distinguish two different approaches regarding the definition of patterns at the analysis stage of the software development process: conceptual modelling patterns and analysis patterns. A *conceptual modelling pattern* is aimed to represent a specific structure of knowledge (for instance a Part-Of relationship) that we encounter in different domains. An *analysis pattern* specifies a generic, domain-dependent, knowledge required to develop an application for specific users.

Our notion of analysis patterns coincides with that of [FY00]. [Fow97] patterns correspond more to conceptual modelling patterns, according to our terminology.

Unfortunately, previous work does not provide, to our knowledge, a sufficient proposal to define analysis patterns in the UML. For instance, [Fer98, p. 37] states that "an analysis pattern is a set of classes and associations that have some meaning in the context of the application" but their examples are illustrated not only by means of class diagrams, as we could expect from the previous definition, but also with state and sequence diagrams.

Later on, [Fow99] develops UML versions of analysis patterns that appear in some chapters of [Fow97]. However, only class diagrams are translated into the UML and, unfortunately, no discussion is given about which of the several UML diagrams should be used to define analysis patterns in this language.

More recently, [FY00, p. 184] states that "a semantic analysis pattern is a pattern that describes a small set of coherent use cases that together describe a basic generic application". It provides some examples of analysis patterns described with analysis class diagrams, state diagrams, sequence diagrams, etc. However, their sequence diagrams specify object interaction and this can only be done if responsibilities are assigned to objects during the analysis stage. Taking this decision involves design and technological issues and, in this way, it is not possible to define an analysis model which is technologically independent.

3. DEFINITION OF ANALYSIS PATTERNS IN THE UML

According to our previous definition, we view an analysis pattern just as a conceptual schema (an analysis model) of a generic application. The UML includes nine types of diagrams to represent different parts of the system [BRJ99]. However, some of them are not useful to define an UML analysis model since they address technological issues. For instance, a deployment diagram shows the configuration of run-time processing nodes and, thus, it is not independent of a particular implementation environment. For similar reasons, object, activity and component diagrams can be discarded to define analysis patterns in the UML.

Therefore, an analysis pattern should be defined in terms of use case diagrams, class diagrams, interaction diagrams and statechart diagrams. However, each of these diagrams may be defined at different stages of the development process and the way they are defined (and also the UML constructions used to define them) depends on the particular stage we are involved in. For this reason, the definition of analysis patterns in the UML requires a clear statement of the boundary between analysis and design.

Craig Larman [Lar98] provides a good criteria that can be used to define this boundary. His main idea, also sketched in [Boo96, FS97], is to define the system behaviour as a 'black box' at the analysis level, before proceeding to a logical design of how a software application will work. According to this criteria, operations responding to external events are not assigned to classes during analysis and they are recorded in an artificial type named *system*.

We also assume that the UML diagrams that define an analysis pattern are non-redundant [CST02]. A UML specification is redundant when a certain aspect of the system is defined in more than one diagram. As shown in [CST02], non-redundant conceptual schemas contribute to desirable properties of the specifications and facilitate software design.

3.1 The Use Case Diagram

Use cases define possible ways users may use a system to meet their goals. They are documented by means of the *Use Case Diagram*, which identifies the main ways a user may interact with the system, and the *Use Case Definition*, which describes the typical course of events that occurs as a result of actor actions required to perform a particular execution of a use case and the system responses to them.

Use cases are mainly used to model the context and the requirements of the software system. However, we believe that it is useful to define, at least, the use case diagram in an UML analysis pattern since it gives a clear idea of all functionalities provided by the software system and the relationships among them. On the contrary, we do not see a clear contribution of use case definitions to UML analysis patterns since the information they provide is also stated by means of system interaction diagrams and operation contracts as we will see in Section 3.3. In this sense, we regard use case definitions as a mean to define those diagrams more than as a permanent model in itself.

The use case diagram of Figure 3.1 defines the most important functionalities of a generic sports competition.

We hope the name of the use cases is clear enough to describe intuitively its intended functionality. Note that some of the use cases require the execution of other use cases to perform satisfactorily. Thus, for instance, to add a new player and to add a new referee requires to add them also as a new person. Moreover, removing a team requires to remove all its players. We provide a more precise definition of the behaviour of some of these use cases in Section 3.3.

3.2 The Analysis Class Diagram

The Analysis Class Diagram specifies the structural properties of the classes that model concepts of the problem domain. It is described by means of *class diagram* in which no operations are defined and it is complemented with *textual constraints*, that define conditions that the information must satisfy but that can not be graphically specified in the UML, and with *derived attributes*, that specify information that can be computed from other elements of the class diagram.

Figure 3.2 shows an analysis class diagram of a generic sports competition. For the sake of simplicity we assume that we model a single league and

Figure 3.1 – Main use cases for a generic sports competition

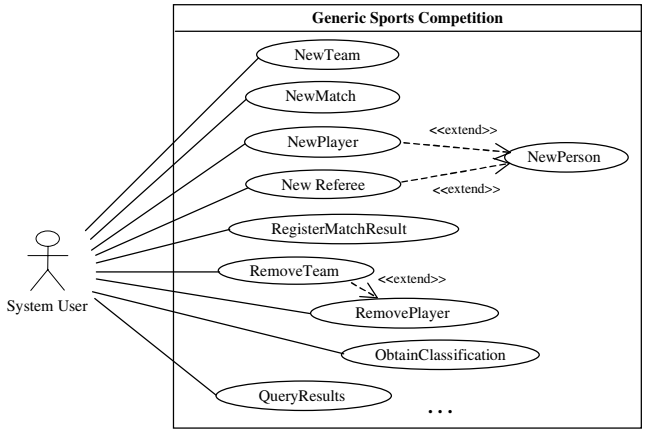
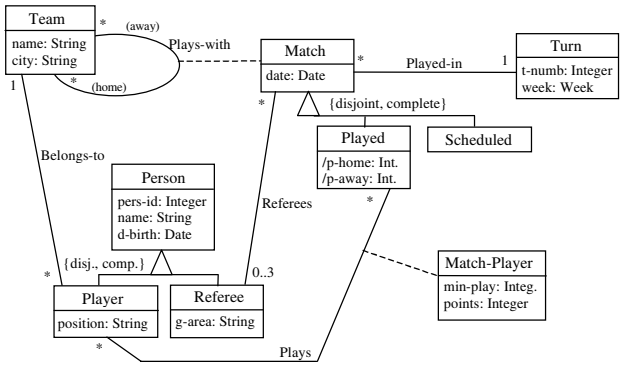


Figure 3.2 – Analysis class diagram for a generic sports competition



Textual Constraints:
- Class identifiers: (Team, name); (Turn, t-numb); (Person, pers-id)
- A player may not play a match if his team is not one of the teams involved in the match
- The two teams that play a match are different teams
- A team plays exactly 1 match in a certain turn

Derived Attributes:
- p-home of Played = sum of points scored by the players of the home team of the match
- p-away of Played = sum of points scored by the players of the away team of the match

that a team plays exactly two times (home and away) with any other team. A match is defined by two teams and it is played in a certain turn. A match can be scheduled or played. If it is played, we also know the points of both teams that played the match.

Several people are involved in the sports competition. They can be classified into either players or referees. A player belongs to a team and may play several matches. Clearly, a player may not play a match if his team is not involved in the match. Moreover, a player may not be also a referee of the competition.

The formalization in the OCL of some textual constraints and derived attributes of the previous example would be:

context Team **inv**: — two different teams may not have the same name
Team.allInstances -> forAll(t1, t2 | t1 <> t2 **implies** t1.name <> t2.name)

context Match-Player **inv**:
— A player may not play a match if his team is not involved in the match
(self.Player.Team = self.Match.home) **or** (self.Player.Team = self.Match.away)

context Match **inv**: — the two teams that play a match are different teams
self.home <> self.away

context Played **inv**: — derivation rule for p-home
p-home = self.match-Player -> select (mp | mp.Player.Team = self.home) -> sum( )

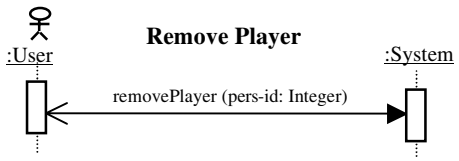
As we said, operations are not specified in the analysis class diagram since we regard the system as a black box during the definition of analysis patterns. Moreover, there are several other elements that can be used in the definition of UML class diagrams in general but that do not make sense for analysis class diagrams. For each of such elements [OMG01], we briefly justify why this is the case:

- **Attribute visibility**: it specifies whether an attribute can be used by other classifiers. This concept does not model the problem domain but the solution domain and, thus, it does not make sense at the analysis level.
- **Navigability**: it states whether an association may be traversed towards other instances in that connection. However, at the analysis level, associations represent existing relationships among real-world concepts. Therefore, all associations can be traversed in all possible directions and so we do not have to specify its navigability.
- **Association**-end visibility: it specifies the visibility of the association end from the viewpoint of the classifier on the other end. Visibility is required to be able to navigate from one end of an association to another. As we have just seen, navigability is a design issue which is not relevant at the analysis level.

3.3 System Interaction Diagrams and Operation Contracts

When the system behaviour is specified as a "black-box", it does not make sense to specify the interaction among objects to fulfil a given functionality but just to specify the interaction among the external actors and the system regarded as such "black-box". This is why we talk about *system interaction diagrams*.

A system interaction diagram shows the external actors that interact with the system, the system as a 'black box', system events that actors generate, their order and the system response. Interaction diagrams may be illustrated either by means of collaboration or sequence diagrams. The following sequence diagram defines the interaction required by the use case NewTeam:



This interaction required to execute newTeam is very simple. In fact, it is enough to specify the name and the city of the team to register a new team.

System interaction diagrams are complemented with operation contracts to precisely specify the system response to the external events. There is a one to one correspondence between events and operations and, therefore, we have to specify an operation contract for each event occurring in a system interaction diagram.

In the UML, an operation contract includes the *signature of the operation*; its *precondition*, i.e., a set of conditions that are guaranteed to be true when the operation is executed; and its *postcondition*, i.e., a set of conditions that hold after the operation execution. The following contract specifies the semantics of the operation *newTeam*:

context System :: newTeam (name: String, city: String)
post: t.oclIsNew ( ) and t.oclIsTypeOf (Team) and t.name = name and t.city = city

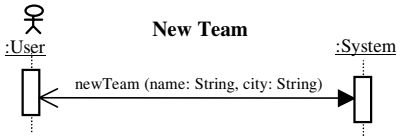
As a consequence of the execution of this operation it happens that an object t of the class Team is created, with attribute values corresponding to the operation parameters name and city. Note that, since we consider that our UML analysis pattern is non-redundant, the previous operation contract must not check that any other team identified by name exists because this is guaranteed already by the textual constraints of the analysis class diagram.

There are several elements that can be used to define operations on the UML but that do not make sense at the analysis level. For each of such elements [OMG01], we briefly justify why this is the case:

- **Assigning operations to classes**: it can only be done if responsibilities are assigned to objects during analysis. However, this decision involves design issues and, therefore, it makes difficult to define an analysis model which is technologically independent. Moreover, it does not make much sense to specify the internal behaviour of an information system when it is regarded as a "black-box".
- **Operation visibility**: it specifies whether an operation may be invoked by other operations. It involves modelling the internal behaviour of the system and, thus, it does not make sense for analysis patterns. In fact, at this level we assume that all operations are public since all of them can be executed by external actors.
- **Abstract operation**: an operation is abstract if it is not implemented in the class where the operation is defined, i.e. no method for it is provided on that class. Clearly, this issue involves technological considerations and, thus, it does not make sense at the analysis level.
- **Completeness of postconditions**: [Lar02, p. 181] suggests that it is not necessary to specify completely the postcondition of the operation contracts. However, we disagree with this opinion since we think that the behaviour of the system can only be precisely specified if we define a complete set of non-redundant postconditions. Therefore, we believe that an analysis pattern should contain a complete and non-redundant operation contract for each event appearing in a system interaction diagram.

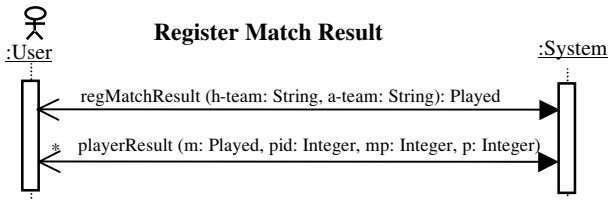
Another important aspect that must be considered during the definition of system interaction diagrams of analysis patterns is the way presentation details are taken into account. In fact, a system interaction diagram describes the basic interaction that an actor must perform to execute a given use case, without going into particular details on how it will be actually performed for a given interface.

As an example, in the following system sequence diagram we do not care about whether the user selects the player to be removed from a list of players or whether he just writes the pers-id of the player on a certain form.



Clearly, presentation details have nothing to do with the concrete semantics of the application. Moreover, different implementations of a particular application semantics may require different presentation details according to the specific preferences of the users of each application. Nevertheless, considering different presentations does not imply any change on the application semantics.

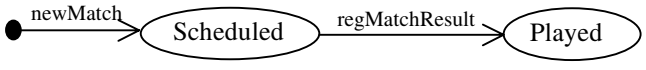
Another example of a more complex system sequence diagram to specify the interaction required by the use case RegisterMatchResult, with the corresponding operation contracts, is:



context System :: regMatchResult (h-team: String, a-team: String): Played
post: let m = Match.allInstances -> select (m | m.home = a-team and m.away = a-team) **in**
m.oclIsTypeOf (Played) and result = m
context System :: playerResult (m: Played, p-id: Integer, min: Integer, p: Integer)
post: mp.oclIsNew ( ) and mp.oclIsTypeOf (Match-Player) and
mp.min-play = min and mp.points = p and
mp.Played = m and mp.Player = (Player.allInstances-> select (p | p.pers-id = p-id))

3.4 Statechart Diagrams

Statechart Diagrams illustrate the states of objects and the behaviour of these objects in reaction to an event. An analysis pattern must include a statechart diagram for each object class with an important dynamic behaviour. As an example, the dynamic behaviour of *Match* may be specified by means of the following statechart diagram:



At the analysis level, statechart diagrams are defined by means of *protocol state machines* [OMG01, 2-170]. Each event appearing in a protocol state machine requires a corresponding operation of the class for which the statechart diagram is defined. Its behaviour is defined by an operation contract instead of the specification of action expressions on transitions.

4. ANALYSIS PATTERNS AND THE UNIFIED PROCESS

The use of our UML analysis patterns does not enter in contradiction with the incremental and iterative nature of the Unified Process [JBR99, Lar02].

The inception phase of the Unified Process involves the identification of relevant use cases, which are specified by means of the use case diagram and the use case definition. Our analysis patterns cover the inception phase since they provide the corresponding use case diagram, while the behaviour stated by the use case definition is provided by means of system sequence diagrams and operation contracts.

During the first iteration of the elaboration phase, system sequence diagrams, the analysis class diagram and operation contracts are developed in the Unified Process. Clearly, this phase is also covered by our patterns since they include the corresponding diagrams.

Statechart diagrams and some concepts of use case diagrams (like relating use cases) or the analysis class diagram (like modelling generalization) of our analysis patterns are usually delayed until the third iteration of the elaboration phase in the unified process (at least in Larman's interpretation of this process [Lar02]).

We can conclude, therefore, that our UML analysis patterns cover the diagrams developed during the inception phase and the first part of the elaboration phase of the unified process. For this reason, given an analysis pattern, we can apply the unified process as usual by assuming that those steps have been performed already. In fact, it would be enough to adapt the pattern to take the specific aspects of the domain where the system is to be developed into account and, then, proceed with the other phases of the unified process.

5. CONCLUSIONS

We have shown that analysis patterns must be defined in the UML by means of a use case diagram, an analysis class diagram, system interaction diagrams with their corresponding operation contracts and statechart diagrams. We have also identified some UML elements that these diagrams may not contain to ensure that an analysis pattern corresponds to an analysis model and we have shown that analysis patterns defined in this way can be used in the context of the Unified Process.

Our proposal has been illustrated by means of a (partial) example aimed at modelling a generic sports competition. Since the general structure provided by an analysis pattern is valid to define several software systems sharing the features described by the pattern, it is enough to adapt this pattern to develop a software system applicable to any specific sports competition.

ACKNOWLEDGMENTS

This work has been partially supported by the Ministerio de Ciencia y Tecnología and the FEDER funds, under the project TIC2002-00744.

REFERENCES

[Boo96] G.Booch. "Object Solutions: Managing the Object-Oriented Project", Addison-Wesley, 1996.

[BRJ99] G.Booch; J.Rumbaugh; I.Jacobson. "The Unified Modeling Language User Guide", Addison-Wesley, 1999.

[CST02] D.Costal; M.R.Sancho; E.Teniente. "Understanding Redundancy in UML Analysis Models", 14th Int. CAiSE Conference, LNCS 2348, Springer, 2002, pp. 659-674.

[Fer98] E.B.Fernandez. "Building Systems Using Analysis Patterns", 3rd Int. Software Architecture Workshop (ISAW3), ACM, 1998, pp. 37-40.

[Fow97] M.Fowler. "Analysis Patterns – Reusable Object Models", Addison-Wesley, 1997.

[Fow99] M.Fowler. "Analysis Patterns", http://www.martinfowler.com/apsupp/, 1999.

[FS97] M.Fowler and K.Scott. "UML Distilled", Addison-Wesley, 1997.

[FY00] E.B.Fernandez; X.Yuan. "Semantic Analysis Patterns", 19th Int. Conf on Conceptual Modeling (ER'00), LNCS 1920, Springer, 2000, pp. 183-195.

[GHJV95] E.Gamma; R.Helm; R.Johnson; J.Vlissides. "Design Patterns – Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[JBR99] I.Jacobson; G.Booch; J.Rumbaugh. "The Unified Software Development Process", Addison-Wesley, 1999.

[Lar98] C.Larman. "Applying UML and Patterns", Prentice Hall, 1998.

[Lar02] C.Larman. "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process", 2nd Ed., Prentice Hall, 2002.

[Mac01] L.A.Maciaszek. "Requirements Analysis and System Design – Developing Information Systems with UML", Addison-Wesley, 2001.

[OMG01] OMG. "Unified Modeling Language Specification", Version 1.4, September 2001.

[Pre00] R.Pressman. "Software Engineering: A Practitioner's Approach", Fifth Edition. McGraw-Hill, 2000.