A Pattern Language for Requirements Analysis

Daniel A. Rawsthorne, PhD
BDM Air Safety Management Company
University of Denver
drawstho@bdm.com

Abstract

It is well known that the most difficult and important process within Software Engineering is Requirements Analysis - the production of analytic models that accurately and adequately represent the requirements of the system to be built. This paper presents a series of patterns that can be followed in order to produce such models in an object-oriented context.

Introduction

Design patterns "provide a solution to a problem in a given context" [Booch, pg 48]. These patterns can be at many levels of detail and fidelity, from language idioms to behavioral mechanisms to methodology frameworks. In other words, a pattern consists of three things: a problem, a context, and a solution. Additional information, such as forces or comments, are supplied as needed to assist the potential user of the pattern.

In this paper we present a series of patterns that can be used to support object-oriented requirements analysis. This presentation is brief and incomplete. In part, this is because we believe that patterns should provide a minimally sufficient set of directions, and leave as much leeway as possible to the analyst. However, the main reason is that it is impossible to completely describe the processes and their uses in such a short paper.

There is no specific notation required to use these patterns; however, some notations may be better suited than others. We have our preferences, which we present at the end of the paper. It is our hope that the presentation in this paper is adequate, and will help anyone who wishes to provide useful analytic models. The patterns presented in this paper can be used by themselves, or in conjunction with other patterns, such as those found in [Whitenack].

The patterns presented in this paper include:

- Requirements Analysis
- Domain Analysis
- Requirements Specification
- Define Structural Relationships
- Define Inheritance Relationships
- Define Interaction Relationships
- Define Constrants
- Pattern Scavenging
- Model Validation
- Identify/Evaluate Reuse Candidates
- Prototyping
- Develop/Refine Test Cases

Pattern: Requirements Analysis

Problem: An analyst wishes to produce an understandable object-oriented analytic model that represents a system that needs to be built. This analytic model should not only adequately describe the system to be built, but should also capture and present common features in the problem domain that could possibly be exploited during design.

Context: There is a system that needs to be built. This system may or may not have a formal specification - different development projects start at different points. For example, the analyst may start with a 600-page written specification (some government contracts) or the plaintive call "I need a way to ..." from the user down the hall. In any case, the analyst is not completely sure of what the system should be, and needs to conduct requirements analysis to determine this.

There is an identifiable customer (or surrogate) that can be used to provide the customer's needs and wants and aid in evaluation and validation of software products produced. Providing a customer to play this role is a responsibility of the "purchaser" of the system, and is strongly recommended in ISO 9000-3 [ISO9000, paragraph 4.1.2]. The existence of such a customer is crucial to successful development of a system, and it is unlikely that successful requirements analysis can be conducted without one.

There is an identifiable domain expert that can be used to provide knowledge of the problem domain and aid in evaluation and validation of software products produced. The responsibility of the domain expert is to understand the domain within which the system will live, and communicate this information to the analyst. The domain expert knows what will and won't work within the domain, and is quite often a member of the development team, rather than a customer.

Solution: There are two distinct, though intertwined, processes involved in Requirements Analysis: Requirements Capture and Pattern Identification. The directions for their use are quite loose and informal, and mostly left up to the discretion of the analyst.

The analyst plays a number of roles within this process:

- the analyst extracts needs and wants from the customer, and validates them for feasibility with the domain expert;
- the analyst extracts domain knowledge from the domain expert, and focuses it on the information relevant to the needs and wants of the customer; and
- the analyst captures all this information into analytic models.

Requirements Capture is the specification of what the system must do. This specification is captured in an analytic model that must be both adequate and correct. The focus is on important behavior and constraints (from the customer's point of view), and the process consists of iterative use of the patterns **Domain Analysis**, **Requirements Specification**, and **Model Validation**.

Pattern Identification is the extraction of similarities and patterns inherent in the requirements and expression of them in refinements of the analytic models. The result is the identification of analytic patterns (both behavioral and structural) that are inherent in the system, and that may be exploited by the designer. This process consists of iterative use of the patterns **Pattern Scavenging** and **Model Validation**.

Pattern: Domain Analysis

Problem: An analyst wishes to produce or modify analytic models based on the relevant aspects of the problem domain.

Context: Analytic models may or may not already exist. There is an identifiable domain expert that can be used to provide knowledge of the problem domain.

Solution: There are three basic types of requirements: structural, behavioral, and constraint. Structural and behavioral requirements are documented directly in the analytic model, and constraint requirements augment this model.

For behavioral and structural requirements, the analyst views the domain as a set of interrelated objects, and the domain expert will help the analyst determine what these objects are, and what the relationships are between them.

The analyst and domain expert work together to determine those relationships that are "natural" within the domain, and that are relevant to the system being developed. There are three types of relationships to be captured:

- structural relationships, using the pattern **Define Structural Relationships**,
- inheritance relationships, using the pattern **Define Inheritance Relationships**, and
- interaction relationships, using the pattern **Define Interaction Relationships**.

It is the job of the analyst to focus the domain expert on the problem as it has been communicated by the customer (see the pattern **Requirements Specification**). It is the job of the domain analyst to understand the domain within which the system lives, and provide this understanding to the analyst. It is our experience that the domain analyst provides primarily structural and inheritance relationships that occur within the system, and interaction relationships only secondarily. The domain analyst also helps the analyst determine constraints on the system, using the pattern **Define Constraints**.

Another aspect of domain analysis is the identification and evaluation of potential software reuse candidates. What the analyst is looking for is existing software product that can be reused as a part of development of the system. Other systems that operate over the same problem domain are considered part of the domain, and are identified and evaluated as part of domain analysis. The evaluation for potential reuse is done using the pattern **Identify/Evaluate Reuse Candidates**.

We note that this pattern is tightly coupled with pattern **Requirements Specification**, as it is difficult to determine which portions of the analytic models are based on knowledge of the problem domain and which are based on the customer's needs and wants.

Pattern: Requirements Specification

Problem: An analyst wishes to produce or modify analytic model based on the Customer's needs and wants.

Context: Analytic models may or may not already exist. There is an identifiable customer (or surrogate) that can be used to provide the customer's needs and wants.

Solution: There are three types of requirements: structural, behavioral, and constraint. Typically, structural and behavioral requirements are documented as part of the analytic model, and constraint requirements augment this analytic model. The analyst views the system as a set of interrelated objects: those that reside within the domain, and those that are specific to the system. The customer will help the analyst determine what these objects are (particularly those specific to the system), and what the relationships are between them.

These relationships are those that allow the system to satisfy the customer's wants and needs, and the analyst and customer must work together to prioritize and manage them in developing the analytic model. There are three types of relationships that are captured:

- structural relationships, using the pattern **Define Structural Relationships**,
- inheritance relationships, using the pattern **Define Inheritance Relationships**, and
- interaction relationships, using the pattern **Define Interaction Relationships**.

It is common that the customer's needs and wants are elicited, expanded, or clarified through prototyping, using the pattern **Prototyping**. Additionally, when the analyst is eliciting requirements from the customer, the analyst must constantly be looking for necessary tests. Structural and behavioral requirements usually, but not always, require tests for validation, and these tests will be developed and validated using the pattern **Develop/Refine Test Cases**.

It is the job of the analyst to extract specific needs and wants from the customer, and communicate to the customer what the domain expert has said is common within the domain, what is possible/impossible, and so on. The customer must work with the analyst to develop a set of specific requirements for the system that are consistent with the view of the domain provided by the domain expert (see the pattern **Domain Analysis**). It is our experience that the customer primarily provides interaction relationships between the objects that have been identified by the customer and the domain expert. The customer will also help the analyst define constraints on the system, using the pattern **Define Constraints**.

We note that this pattern is tightly coupled with pattern **Domain Analysis**, as it is difficult to determine which portions of the analytic models are based on the problem domain and which are based on the customer's needs and wants. It is commonly thought that the objects derived during requirements specification are "layered" on top of those determined during domain analysis, but it is tough to draw the line between them.

Pattern: Define Structural Relationships

Problem: The analyst wishes to refine the structural relationships portion of the analytic models.

Context: The analyst has access to an expert (either a customer or a domain expert) to aid in developing these relationships.

Solution: There are a number of different types of structural relationships that can exist between objects, and different methodologies use different names for them. The following paragraphs attempt to describe common understandings of these relationships, and uses no specific naming conventions.

Aggregation: this refers to a relationship between classes, and indicates when an instance of one class is made up of instances of other classes, and this relationship is fixed. For example, a Car is always made up of an Engine, a Body, four Tires, and so on.

Subsystem: this refers to a relationship between objects, and indicates when one object is made up of other objects. The internal objects are distinct (though may be of the same class), and are "known by name" to the system. Typically, we think of a system as a one-of-a-kind decomposition of an object into other objects. For example, the Menu subsystem consists of FileMenu, EditMenu, ViewMenu, ...

Container: this refers to a relationship between an object or a class (the container, or a class of containers) and a class (the class of the contained objects), and indicates that one object holds a number of other objects, each of the same class and virtually indistinguishible to the container. For example, TodaysTransactions is a container that is a list of all the Transactions executed today, the ClassRoom class is a container of Students, and so on.

In both aggregations and containers there is the concept of multiplicities, when one object can own a number of objects of the same class. These multiplicities are normally captured within the analytic model.

One of the more complicated things about structural relationships is based on the fact that a given object can "belong" to a number of others within the system. For example, in a model that involves Members and Clubs, it is clear that a Club is a container of Members, and that any particular Member can belong to more than one Club. This information must be captured in order for the system to be adequately understood.

During this process the analyst must keep in mind that it is an analytic model, not a design model, being constructed. As such, it is more important to understand the structural relationships between individual objects than it is to generalize them to relationships between classes.

Pattern: Define Inheritance Relationships

Problem: The analyst wishes to refine the inheritance relationships portion of the analytic models.

Context: The analyst has access to an expert (either a customer or a domain expert) to aid in developing these relationships

Solution: Inheritance (in an analytic model) is a simple, and often misunderstood, concept. Basically, it involves an object's need to be similar to other objects in some way. These similarities are analytic, and have little to do with the inheritance of classes one finds in object-oriented languages.

For example, we could inherit based on objects needing to know similar information, needing to perform similar services (having similar responsibilities), playing the same or similar roles within the system, or based on looking alike in some other way that is relevant to the system. It is quite common for the expert to refer to objects as abstractions; it is the analyst who must determine when these abstractions are useful.

Typically, abstractions represent "placeholders" - entities that can be replaced by any of a set of objects within a particular context. For example, within an ATM system, the expert may refer to a Transaction, which could represent a DepositTransaction, WithdrawalTransaction, or some other type. In this case we would say that Transaction is an abstraction that the others inherit from. Abstractions used as placeholders typically occur quite naturally in the domain.

Inheritance is also used to indicate when one object is an "extension" of another. For example, a CancerPatient is an extension of a Patient, so we could say that CancerPatient inherits from Patient.

Often, objects found in the system are extensions of objects in the domain - extended to provide specific system functionality.

During this process it must be kept in mind that one captures inheritance based on how an object behaves in the system, not inheritance based on implementation constructs.

Pattern: Define Interaction Relationships

Problem: The analyst wishes to refine the interaction relationships portion of the analytic models.

Context: The analyst has access to an expert (either a customer or a domain expert) to aid in developing these relationships.

Solution: Most object-oriented systems accomplish their functionality by having objects interact. It is crucial that the analytic models capture these interactions, in order that the designers and testers have an adequate understanding of the system.

Most methodologists agree that one should capture these interactions, and use constructs called "use cases", "collaborations", "scenarios", "transactions", and so on, to do so. We don't believe that it matters what we call these things, as long as we have them.

Many interactions are part of the domain, and involve typical behavior within the domain. For a particular system, however, it is more important that the interactions that provide the system-specific behavior be determined.

This is possibly the most important process the analyst and expert can perform. Without adequate definitions of the interactions needed for the system to accomplish its required functionality, it is doubtful that the system can be developed efficiently.

Pattern: Define Constraints

Problem: The analyst wishes to augment the analytic models with constraints concerning the system.

Context: Analytic models should already exist. There is an identifiable expert (either customer or domain expert) that can be used to provide system constraints.

Solution: The analyst and expert view the system as a set of interacting objects, and they must define constraints in terms of this view. Therefore, the forms these constraints take is, in large measure, determined by the forms of the analytic models used.

In general, we view constraints as augmentations, or comments, on portions of the analytic models. Some constraints, such as overall CPU footprint, must be defined at the system level itself, while others are assigned to specific objects, subsystems, interactions, and so on, within the system.

One of the most common kind of constraint concerns speed of calculation; in particular, speed of a certain algorithm, or process, the system must carry out. This kind of constraint can be documented as a comment to the interaction(s) used to carry out the requirement. Another common kind of constraint concerns the sizes of Containers; for example, that a list must be able to contain up to 2000 members. This kind of constraint can be documented as a comment on the structure diagram showing the relationship.

In all cases, the analyst and expert should try to assign the constraint to a particular piece of the analytic model, at as detailed a level as possible. We have found it useful to have a matrix, showing the constraints and the pieces of the model they pertain to.

Pattern: Pattern Scavenging

Problem: An analyst wishes to identify and capture patterns of structure and behavior within an existing analytic model, to "scavenge for patterns among scenarios and the domain model, ..." [Booch, pg 104]. In this quote, the word "pattern" means "a combination of qualities, acts, tendencies, etc., forming a consistent or characteristic arrangement" [Webster], rather than "a solution to a problem in a given context." We trust that this dual use of the word "pattern" will cause no confusion.

Context: There is an existing analytic model.

Solution: The analyst examines the existing analytic models in an effort to determine similarities in structure and behavior. Basically, we are looking for similar "shapes" in the existing graphical depiction of the models.

When similarities are identified, abstractions (both objects and responsibilities) are developed to act as "placeholders" for the objects and responsibilities that play similar roles within the models. These abstractions are assembled into abstract models that represent the ones being analyzed.

One of the major strengths of object-oriented technology is inheritance. During requirements analysis both structural and behavioral responsibilities of the system are discovered and captured in analytic models. In order to provide consistency and reuse within the application; similarities between these responsibilities should be identified and captured within the models. Typically, these similarities are encapsulated within abstract objects and responsibilities, and disseminated throughout the system via inheritance.

This process is an extension of what we do already - extract class abstractions by looking at objects. In this process we extend this to look for patterns of behavior between objects, or patterns of structure among objects. Many of the design patterns found so far ([GOF] and others) are precisely such abstractions.

Pattern: Model Validation

Problem: The analyst wishes to have the analytic model validated.

Context: There is an existing analytic model. There is an identifiable customer (or surrogate), an identifiable domain expert, or both, that can be used to evaluate and validate the model.

Solution: The first, and most important, thing to do is assure that the person(s) doing the validation understand the notation used to represent the models. Have the analyst(s) that produced the models present during the validation process in order to explain any subtleties and resolve any misunderstandings about the meaning of the models.

Review all portions of the analytic models in an orderly fashion. This includes all constraints that have been documented as comments, or augmentations, to the models. Make notes about all issues or questions that arise. For each feature of the models being evaluated, determine whether or not a test

case will be needed to validate this feature or constraint in the final system. If it is determined that a test case is not necessary, document this fact. If it is determined that a test case will be necessary, use the pattern **Develop/Refine Test Cases** to develop one. In a very real sense, the test is as much a part of the specification as the analytic model itself.

Appropriate parts of the analytic models can be validated by either a domain expert or a customer. Any analytic model should be validated by both as being complete before it is used as a starting point for design. In many cases the analytic model has been developed with the assistance of the customer or domain expert. The analyst should not assume that validation will be automatic in these cases.

The best way to validate something is to develop and agree on a test that will validate the finished product. In many ways the test is the requirement, as it provides a method to determine whether the product meets the requirement. Validation of the model often means validation of the tests.

Once the validation process is complete, prepare an action plan to resolve the issues and questions that arose. This action plan should involve extending or modifying the analytic models using the patterns **Domain Analysis** and/or **Requirements Specification**.

Pattern: Identify/Evaluate Reuse Candidates

Problem: An analyst wishes to identify and evaluate reuse candidates in order to determine if there are any software products that can be reused within the current system.

Context: The analyst has access to a domain expert to aid in evaluation.

Solution: For each candidate system, there are a number of software products that should be evaluated for reuse: code, plans, models, tests, and so on. Each should be evaluated for applicability to the system being developed. At the very least, existing systems should be studied to better understand the problem domain.

Each product's applicability must be determined on a Case-by-Case basis. Remember that if any significant part of a product needs to be reworked (the definition of significant ranges from 5% to 25%, according to the literature), it is probably more cost effective to rework the whole product than attempt reuse.

It must be noted that much of the information being evaluated may not be expressed in an object-oriented fashion. In order to convert these products to object-oriented ones, we recommend looking at the patterns in [Kerth].

Pattern: Prototyping

Problem: The analyst wishes to determine or refine requirements through prototyping.

Context: Analytic models may or may not already exist. There is an identifiable customer (or surrogate) that can be used to help develop and evaluate prototypes for applicability.

Recipe: With the aid of the customer, an analyst develops prototypes to explore limited sets of requirements. This exploration is not limited to user interface issues; these requirements could include algorithms, structural relationships within the system, and so on. Remember that a prototype

is a simulation of a portion of the system, and may be developed in many ways. In fact, prototyping on paper or in some other non-software medium can be quite useful.

If possible, use a simulation tool that guarantees that your customer will not expect reuse of the prototype in the finished product - expect to throw away the prototype. If the analyst believes that the prototype has some reuse possibility, it should be evaluated using the pattern **Identify/Evaluate Reuse Candidates**.

Requirements for a system are often not fully expressed, nor are they fully understood. Prototyping often aids the customer in formalizing or solidifying the system's requirements.

Pattern: Develop/Refine Test Cases

Problem: The analyst has identified the need to have a set of test cases that will be used to evaluate and validate portions of the system.

Context: Particular requirements have been identified that may require test cases. There is an expert identified who can aid in evaluation and validation of the test cases. There is a test team (which could be the analytic team) whose responsibility it is to develop the system's tests.

Solution: When the analyst is working with the expert, he must constantly ask the questions: "how do we know that we have done it right?" and "how do we test for that?" These questions could be integrated as part of the patterns **Domain Analysis**, **Requirements Specification**, and **Define Constraints**, or they could be asked in special sessions with the expert. In either case, the refinement and documentation of an identified test case is done by the test team as part of this (**Develop/Refine Test Cases**) pattern.

There are three basic types of requirements: behavioral, structural, and constraint. When it is identified that a requirement requires a test case, this test case should be developed by the test team in conjunction with the expert and documented as part of the analytic model. In some real sense, the requirement is not complete until the test case exists, is documented, and validated by the expert.

The test team must make sure that the requirement is testable, and that a test is specified along with the requirement if it is. It is possible that the statement of a requirement fully defines the test, but the analyst must make sure that the test is sufficient and documented as such.

The basic idea is that the analyst and the expert have determined the need for a test, and they work with the test team to outline and prepare the test. Test case definition is possibly one of the most important steps in requirements analysis. Without knowledge of the tests that will be used to validate the system, it is impossible for the analyst to be sure that the analytic model of the requirements is correct.

Summary

The patterns discussed in this paper are currently being used at BDM to conduct requirements analysis of an Air Traffic Control (ATC) system. We are conducting daily analytic sessions, using CRC cards augmented with Transaction Based Analysis (TBA), which is a home-grown method for capturing complex object interactions (see [Rawsthorne95] and [Rawsthorne96]). We are finding the Pattern Scavenging pattern particularly helpful, as there are many abstract behavioral patterns in this domain.

We are not currently using a robust object model (e.g., Booch, Rumbaugh, UML) because the structural requirements in this domain are not particularly complex. We have replaced this model with a simple subsystem diagram for this domain.

The processes embodied in these patterns have been evolving through use on a number of projects. Another recent one is a data management system used to hold airspace information for a variety of ATC systems. The expression of behavioral patterns has allowed us to capture and explain to our users how the new system will interact with their systems.

In addition, the fundamentals of these methods have been taught to almost 200 students at the University of Denver. Based on their comments, and our experiences, I feel that the relative simplicity of the processes make them ideal for eliciting and documenting user's requirements.

References

[Booch] Booch, G. 1995. *Object Solutions: Managing the Object-Oriented Project*. Menlo Park, California: Addison-Wesley Publishing Company.

[ISO9000] International Organization for Standardization, 1991. *ISO 9000-3: guideline for application of ISO 9001 to the development, supply, and maintenance of software*.

[GOF] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley Publishing Company.

[Kerth] Kerth, N. 1995. "Caterpillar's Fate: A Pattern Language for Transformation from Analysis to Design", chapter in *Pattern Languages for Program Design*, James C. Coplien and Douglas C. Schmidt, editors. Menlo Park, California: Addison Wesley Publishing Company.

[Rawsthorne95] Rawsthorne, D. 1995. "The Transaction Hierarchy: an Extensino of Use Cases", presented at the OOPSLA 95 Workshop "Requirements Engineering: Use Cases and More"

[Rawsthorne96] Rawsthorne, D. 1996. "Capturing Functional Requirements Through Object Interactions", *Proceedings of the Second IEEE International Conference on Requirements Engineering (ICRE 96)*

[Whitenack] Whitenack, B. 1995. "RAPPeL: A Requirements Analysis Process Pattern Language for Object Oriented Development", chapter in *Pattern Languages for Program Design*, James C. Coplien and Douglas C. Schmidt, editors. Menlo Park, California: Addison Wesley Publishing Company.