Analysis Patterns Specifications: Filling the Gaps

Marta Pantoquilho[1], Ricardo Raminhos[2], João Araujo[3]

Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Quinta da Torre, 2829-516 Caparica
Portugal
E-mail: [1] mbp@netvisao.pt, [2] rfr@netvisao.pt, [3] ja@di.fct.unl.pt

Abstract. Patterns present solutions for recurrent problems in software engineering. They are applicable at different stages of the software development process. This paper focuses on patterns at requirements and analysis level. Although the term "requirements patterns" has appeared in the requirements engineering community, the name "analysis patterns" is more established in the patterns community. Here we briefly discuss these terms and the existing approaches. The main goal of this paper is to propose a new template to fill some gaps concerning the specification of analysis patterns.

1 Introduction

The traditional software development lifecycle includes the following phases: requirements elicitation, analysis, design, implementation and test. Each phase creates a more detailed image of the system than the previous one. Nevertheless, to be effective, software development must consider reuse since early stages. Patterns are considered a successful technique to help reusing previous specifications and solutions.

Software patterns are classified in different categories depending on various factors including their application to the software development phases (see Figure 1). The most common patterns are analysis and design patterns. Anti-patterns are a kind of pattern that embraces all the development phases (including the test phase), as well as the project management area.

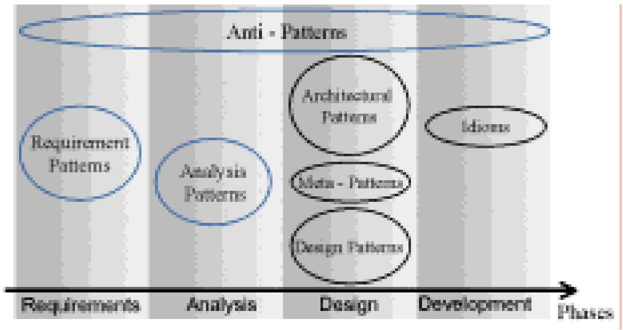


Figure 1 - Project lifeline with the corresponding patterns to each phase.

The term requirements patterns appeared in the requirements engineering community [Robertson, 1996] [Konrad and Cheng, 2002], but it is not widely used. Requirements patterns document user needs and specify the generic system behaviour at a high level of abstraction. Requirements patterns are also suitable to describe generic actions that developers can take to improve non-functional requirements, such as performance, security, reliability, maintainability and accuracy. These actions are related to client-system interaction or operator-system relation.

The purpose of analysis patterns is to build an analysis model, which will focus on business conceptual structures processes instead of software implementations. The main concerns of these patterns are the conceptual models and the flexibility and reuse of the resulting systems. The conceptual structures are represented by a static structure, entity relations (e.g. objects or functions) and data transformations.

The main goal of this paper is to propose a new template to specify analysis patterns. But before that we want to discuss and compare some of the existing approaches and clarify the difference between analysis and requirements patterns.

This paper is organised as follows. Section 2 describes and compares some previous work on requirements and analysis patterns. Section 3 proposes a template to describe analysis patterns. Section 4 illustrates the pattern with an example. Finally, Section 5 draws some conclusions and discusses some future work.

2 Requirements and analysis patterns approaches

In this section we described the state-of-the-art of requirements and analysis patterns. Afterwards, we establish a comparison between these patterns.

2.1 Requirements patterns

In [Robertson, 1996], S. Robertson uses an event/use case approach and employs a very simple template for the pattern description with only 4 fields: name, context, solution and related patterns. Robertson suggests that events and use cases should be used to divide the system in small chunks. These chunks can then be structured into a pattern. Patterns are, therefore, catalogued, based on the name of the use case to which they refer. In her paper, Robertson shows how a particular problem can be abstracted at different levels in order to become a pattern used in different problems.

S. Konrad and B. Cheng [Konrad and Cheng, 2002] focused on requirements patterns for embedded systems. They use a UML approach (class, use case and sequence diagrams) for the pattern definition. Also, they explain the pattern context using problem frames [Jackson, 2000]. A very extensive and detailed template is used to describe the pattern (13 fields), based upon the one suggested by GoF [Gamma et al, 1995] for design patterns.

We notice that the term requirements patterns does not differentiate from analysis patterns described as follows.

2.2 Analysis Patterns

M. Fowler [Fowler, 1997], initially proposed the concept of analysis patterns for the representation of conceptual models for commercial processes (accountability, commercial trades and organizational relations). Refinement patterns (design, architectural, etc) are never suggested, and the solution is mostly conceptual. The author presents each pattern through an informal / technical discussion without any kind of structured template.

E. B. Fernandez and X. Yuan present the Semantic Analysis Pattern (SAP) approach [Fernandez and Yuan, 2000]. SAP is “a pattern that describes a small set of coherent Use Cases that together describe a basic generic application”. The selection of use cases is realised carefully to maximise reusability.

The work by A. Geyer-Schulz and M. Hahsler [Geyer-Schulz and Hahsler, 2001] introduces some structure to analysis patterns. They focused on the cooperative work domain and collaboration between applications.

Analysis patterns proposals include patterns for oil refineries [Zhen and Shao, 2002], the order and shipment of a product [Fernandez et al., 2000], the repair of an entity [Fernandez and Yuan, 2001], negotiation [Hamza and Fayad, 2003], course management [Yuan and Fernandez, 2003]. Also, in [Hamza and Fayad, 2002] a pattern language is proposed to achieve stability while constructing analysis patterns.

2.3 Comparison between Requirements and Analysis patterns

Here we present a short comparison between requirements and analysis patterns, depicted in table 1. In this comparison, we point out the main characteristics of the approaches of both kinds of patterns and also what we consider to be their limitations.

Table 1 - Comparison between Requirements and Analysis patterns.

	Characteristics	Limitations
Requirements Patterns	• They capture in detail functional and non-functional requirements. • They can be extended by design or architectural patterns. • They allow a smooth transition to the implementation phase, due to the pattern detailed description. • They are a more directed form to the programmer understanding.	• Little research in this field. • High commitment to the solution domain, due to decisions expressed in the pattern. • The existence of a variety of templates for the different approaches. • The existing approaches do not seem to justify the term requirements patterns as they use similar principles as analysis patterns.
Analysis Patterns	• They are suitable for the description of conceptual problems. • They present low commitment to the solution domain allowing a high level of freedom for implementation due to their sparse specification details. • Due to the high abstraction level, there is a huge gap between the patterns specification and implementation. • They provide a more directed form to the architect understanding. • In order to migrate to the implementation level, an extra iteration is needed. This extra step could be the transformation of an analysis pattern into a requirement pattern. We would be passing from a low level to a high-level implementation detail. • Lots of work in this field.	• The presentation form (degenerate template) used by M. Fowler [Fowler, 1997], is low in specification detail and information about the pattern description. • The existence of a variety of templates for the different approaches.

This comparison is important to highlight the fact that what defines requirements patterns is not significantly different from analysis patterns, from the approaches studied. This is a result of their proximity, i.e., they are closely related and share a similar level of abstraction. To avoid confusion with these terms, we suggest that patterns at this level of abstraction be called only analysis patterns, by requirements engineering and patterns community. Furthermore, not having a consensus on how these patterns should be specified prevents them from being accepted widely.

We propose a template with elements that are common to these approaches and new elements to fill some gaps that we consider are missing. In the next section we will present such template. Note that this template is only for analysis patterns, since it still is not clear what requirements really are as their objectives. However, the template also comprises requirements patterns aspects.

3 A Template for Analysis Patterns

In this section we present a template to specify analysis patterns. Table 2 shows the attributes of the template and their respective descriptions. The attributes which were not a part of any previous template are ticked in the final column. The proposed template is based upon the one described in the POSA approach [Buschmann et al., 96] [Schmidt et al., 2000].

Table 2 - Template for requirements and analysis patterns.

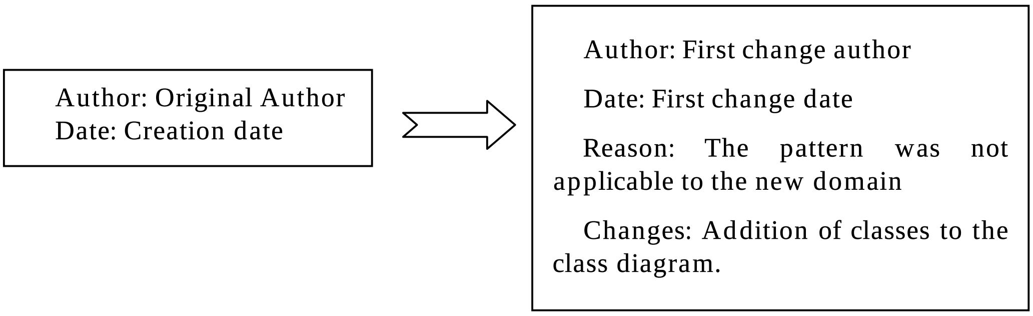
Attributes			Short Description	New
Name*			Pattern identifier.	
Also Known as			Additional names that can also identify this pattern.	
Evolution*			Chronological register of all previous versions of this pattern. The following notation should be used: {Date, Author, Reason, Changes}. To be used by developers who have already used the pattern to check its changes.	✓
Structural Adjustments*			Presentation of field extensions and omissions to the pattern template.	✓
Problem*			A short description of the problem that this pattern solves.	
Motivation*			Description of a problematic situation intended to motivate the use of the pattern.	
Context*			Wide description of the environment in which the problem and solution recur and for which the solution is desirable.	
Applicability*			Description of the conditions wherein the pattern can be applied.	
Requirements*	Functional*		List of all functional requirements organised through use cases.	✓
	Non-Functional*		List of all non-functional requirements.	✓
	Dependencies*		Identification of dependencies for requirements. This could be represented through a graph.	✓
	Priorities*		Definition of priorities among the requirements. This could be represented by a hierarchical structure.	✓
	Conflict Resolution*		Explanation for requirements interaction and conflict resolution.	✓
	Participants*		Identification and description of the actors that interact with the system.	✓
Modelling*	Structure*	Class Diagram*	Structure of the elements of the pattern.	✓
		Object description*	Objects description and their responsibilities.	✓
	Behaviour*	Collaboration or Sequence Diagrams	Suitable for scenarios description.	✓
		Activity Diagrams	Suitable for scenarios and overall description.	✓
		State Diagrams	Suitable for scenarios and overall description.	✓
	Variants		Description of alternative solutions.	
Resultant Context*			System configuration after the pattern application. Includes the description of all requirements not addressed	

Consequences*	Advantages and disadvantages of the pattern application	
Anti-Patterns Traps	Most common pitfalls that can be originated from the pattern application	✓
Examples*	One or more application examples that illustrate: initial context, how the pattern was applied and all transformations necessary to the initial context so that it could be applied	
Related Patterns	List of similar patterns (describing similar problems and solutions)	
Refinement Patterns	Design or architectural patterns that can be used for further refinement	✓
Implementation	Advices on how the pattern should be implemented (without specific details e.g. code)	
Known Uses*	Describes known pattern occurrences and applications in existing systems. This should include at least three different systems	

* - Required field

Below we discuss the newly introduced attributes.

- **Evolution**: This attribute explains all the transformations the pattern suffered. With the addition of the evolution field we can track the pattern progress: from current state to the original version. This helps developers that have already used the pattern identify what changes have taken place. This makes it easier for the developers to adapt to the new version of the pattern. Also, if they want to propose modifications to the pattern, they should know what has been done before. This can help validate the new modifications. In the construction of the original pattern this field should only contain information about the Date and Author. In Figure 2 we illustrate the chronological evolution for an abstract pattern.



Author: Original Author
Date: Creation date

Author: First change author

Date: First change date

Reason: The pattern was not applicable to the new domain

Changes: Addition of classes to the class diagram.

Figure 2 - Evolution tracking system.

- **Structural Adjustments:** This field explains the structure adopted to describe the pattern. It should include all additional extensions, all omitted fields and the reasons for those decisions. With this information the reader can easily understand the used structure. We suggest the use of a layout similar to the one presented in Table 3.

Table 3: Suggested structural adjustments layout.

Attribute	Extension	Omission	Reason
Implementation		✓	Reason for the omission
New attribute	✓		Reason for the extension

- **Requirements:** This field (divided in six sub-fields) contains a description of all requirements that must be addressed to solve the problem (how they interact and are balanced). A highly detailed problem specification is gained trough the addition of this attribute. With the division, it becomes simpler the understanding of the requirements involved, their type (functional, non functional), their dependencies and priorities and how they are solved. Concerning non-functional requirements, we do not show how they are addressed in the design model, but the approach by Araujo and Weiss [Araujo and Weiss, 2002] can be used for this. They use the NFR framework [Chung et al, 2000] for describing a design context and also a set of related patterns. The outcome is that several design issues related to system architecture may be addressed by the integration of various patterns. We can also extract a use case diagram from the requirements, which shows the services used by the actors (participants).

For the *Priorities* field we suggest the use of a hierarchical structure as depicted in Figure 3.

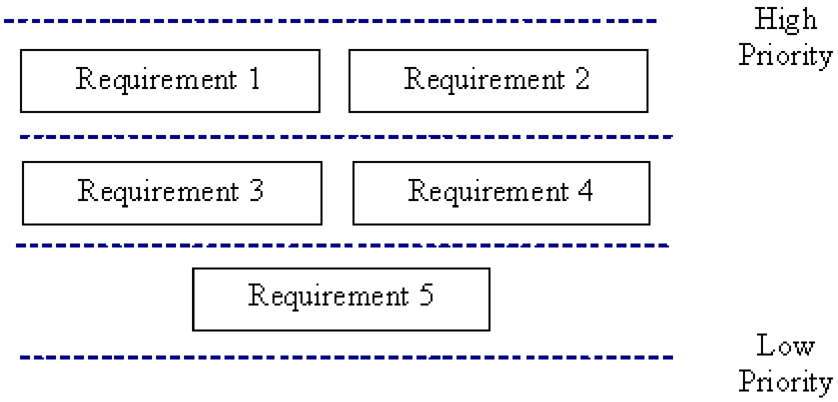


Figure 3: Suggested hierarchical diagram.

For the *Dependencies* field we suggest the use of a dependencies graph. One example is presented in Figure 4. We read Requirement 1 depends on Requirement 2 and Requirement 3, and so on.

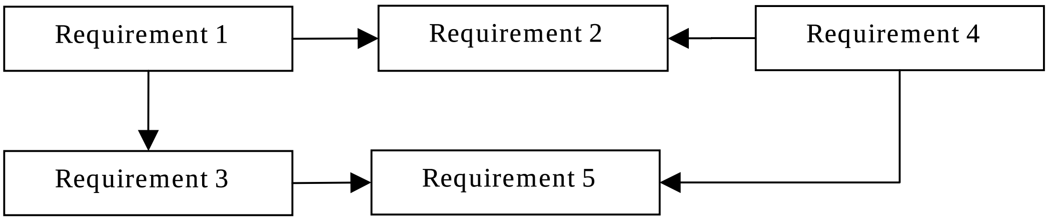


Figure 4: Suggested dependency graph.

In prior template versions, the requirements were addressed in the field *forces*. This was an unstructured field that contained a mixture of functional and non-functional requirements. With our approach, we add structure to the requirements identification and documentation. We also add information about their relationship - dependencies and priorities. To ease the pattern understanding we propose the use of illustrative diagrams: a hierarchical diagram for the requirements priorities resolution and a dependency graph for the dependencies establishment. With the inclusion of the participants attribute, we can describe the actors that will manipulate the system. These actors are the ones that will interact with the use cases identified in the *Functional* field.

- **Modelling:** In this section are presented several models that illustrate the problem solution. This solution is divided in two main groups: behaviour and structure.

- **Structure:** This group represents the solution's static structural aspects using a UML class diagram. More detailed specification can be obtained in the attribute *Object Description*. This field describes all objects that are present in the class diagram. Note that the class diagram can be represented in different levels of abstraction.

- **Behaviour:** Offers an illustrative set of scenarios, and also describes the overall pattern behaviour. The pattern should contain at least one scenario example, in an abstract level, and one overall description. Two distinct levels are focused: scenarios examples that show only part of the system, and the overall system behaviour, which illustrates the system's functioning as a whole. There is a great freedom on the diagrams choice that illustrates this field. At least an activity diagram showing the overall system behaviour should be included. In other templates this section only contained examples of part of the system behaviour. Although important, this is not sufficient, because the user does not have a global vision of the system functionality.

The modelling description is presented in previous pattern templates under the name *Solution*. Although some fields are commonly used in both *Modelling* and *Solution*, the modelling approach offers a more detailed, structured and visual (with the addition of several diagrams) understanding.

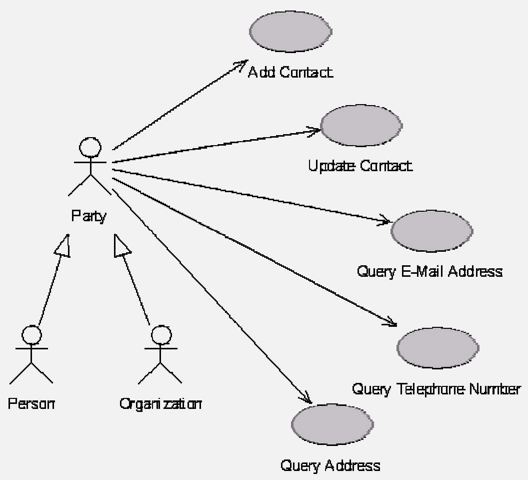
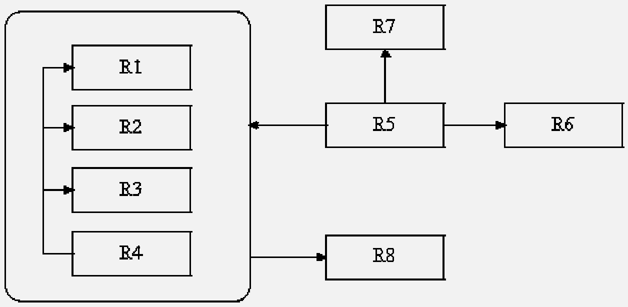
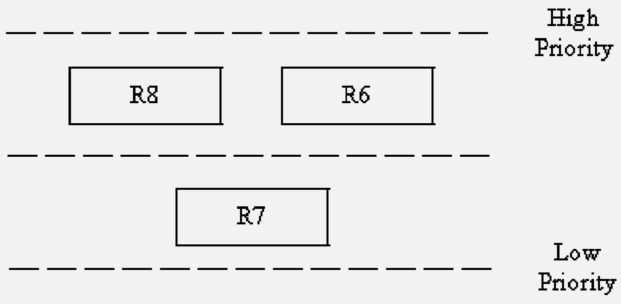
- **Anti-patterns traps:** With this field we try to avoid common errors in this pattern application by presenting the most common negative results. This field should contain a list of anti-patterns names and a short description of each. To recover from a negative solution the user, using the anti-pattern name as lookup key, should refer to William Brown in [Brown, 1998].

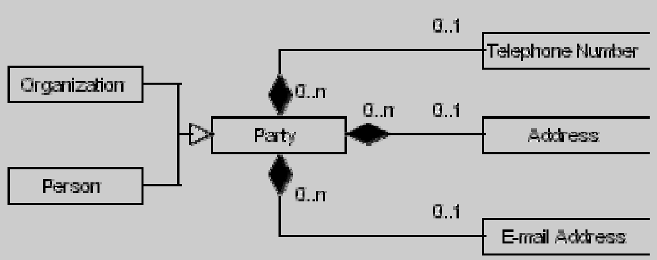
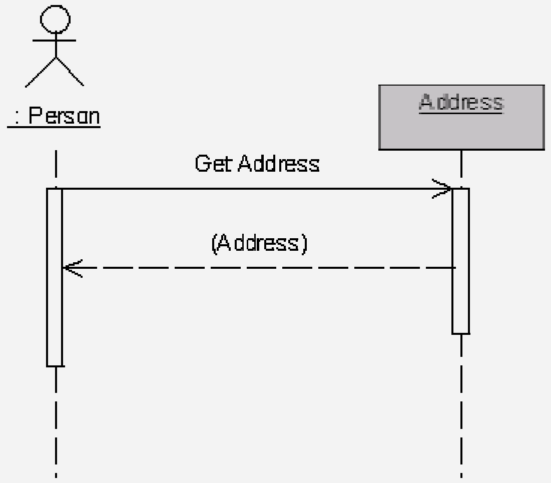
- **Refinement patterns:** This attribute is used to suggest or identify suitable patterns (e.g. design or architectural) that can be applied to the implementation of this pattern.

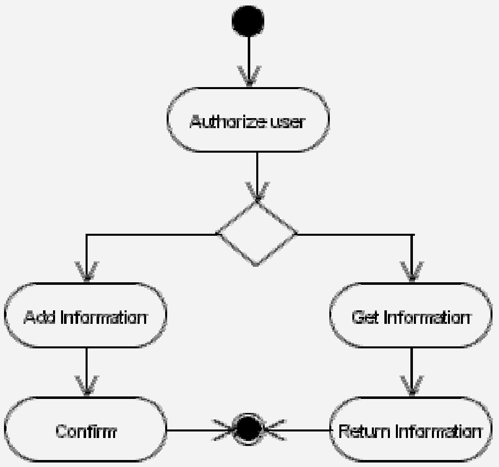
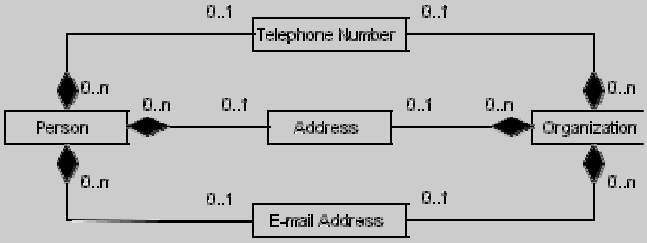
4 Example

To illustrate, we present an example that uses the template to specify the analysis pattern *Party* from [Fowler, 1997].

Attributes	Short Description
Name	Party
Also Known as	To be determined.
Evolution	{ date: 1997, author: Martin Fowler } {date: 2003, authors: M.Pantoquilho, R. Raminhos and J. Araújo, reasons: to add more information Changes: Adaptation of the degenerative form to a structured template.}
Structural Adjustments	None.
Problem	To model an address book that contains people and companies.
Motivation	"Take a look through your address boo, and what do you see? You will see a lot of addresses, telephone numbers, the odd email address … all linked to something. Often that something is a person, however the odd company shows up." [Fowler, 1997]
Context	Persons and Organizations are present in almost every system that deals with people. The address book is just an example. Persons and Organizations share a common behaviour: they address a common set of objects (telephone numbers, email addresses …) and operations over them.
Applicability	When you have people and organizations in your model and you see common behaviour.

Requirements	Functional	R1: Each Person or Organization has none or more telephone numbers. R2: Each Person or Organization has none or more addresses. R3: Each Person or Organization has none or more e-mail addresses. R4: Person and Organization share some descriptive data. R5: Each Person or Organization may obtain, add or modify information about other persons or organizations. Use case model:  Add Contact Update Contact Party Query E-Mail Address Query Telephone Number Person Organization Query Address
	Non-Functional	R6: Person / Organization must be authorized (security). R7: Information must be obtained in a short period of time (performance). R8: Information must be correct (accuracy).
	Dependencies	
	Priorities	

	Conflict Resolution		The non functional requirements of performance (R7) and security (R6) may enter in conflict. However this conflict is solved by the assignment of priorities to the different requirements.
	Participants		Person and Organization
Modelling	Structure	Class Diagram	 Organization Person Party 0..n 0..n 0..n 0..1 0..1 0..1 Telephone Number Address E-mail Address
		Object Description	**Person:** Defines a person. **Organization**: Defines an organization. **Party**: Super type defining a Person or Organization. Contains all common functionalities to both of them. **Telephone Number**: Describes a telephone number. **Address**: Defines an address. **E-mail Address**: Defines an e-mail address. The *Person* and *Organization* classes contain the specific data to each entity that they describe. The common attributes are held at the *Party* class. The *Party* is defined through the relations with the classes *Telephone number*, *Address*, and *E-mail Address*.
	Behaviour	Collaboration or Sequence Diagrams	Sequence diagram for the use case Query Address 

	Activity Diagrams	 Authorize user Add Information Get Information Confirm Return Information
	Variants	Another possible solution is presented in [Fowler, 1997] (below). Although this variant also solves the problem, it does so adding a lot of redundancy (inheritance is eliminated and associations are doubled). Therefore, the solution presented in "Class Diagram" is a better one, for the reasons pointed in the "Consequences" field.  0..1 Telephone Number 0..1 0..n 0..n Person 0..n 0..1 Address 0..1 0..n Organization 0..n 0..n 0..1 E-mail Address 0..1
Resultant Context		“Party is defined as the super type of person and organization. This allows me to have addresses and phone numbers for departments within companies, or even informal teams.” [Fowler, 1997].
Consequences		Advantages: 1. Elimination of data and code duplication.
Anti-Patterns Traps		**Golden Hammer** – Persons and Organizations can be modelled with other patterns than *Party*. Refer to *Related Patterns*, for other pattern information. **The Blob** - *Party* must only contain the common attributes to Person and Organization. You should resist the temptation to incorporate all data and operations in *Party*.
Examples		“In the UK National Health Service, the following would be parties: Dr. Tom Cairns, the renal unit at St. Mary’s Hospital, St. Mary’s Hospital, Parkside District Health Authority, and the Royal College of Physicians.” [Fowler, 1997].
Related Patterns		*Role Object*
Refinement Patterns		Unknown.
Implementation		Use an object-oriented language to the implement this pattern. Although this is a very simple system, you should resist the temptation to implement it in a single class. (avoid the Blob anti-pattern)
Known Uses		Not specified.

To illustrate the behaviour specification of this pattern, we show sequence and activity diagrams. The sequence diagram for QueryAddress is described in an abstract form, without including interface and control objects, as these are normally added in the design stage. The activity diagram is used here to specify the global pattern behaviour.

The original *Party* template used the "Degenerative Form". This is a very unstructured template form. Other templates for requirements and analysis patterns have a better structure and a high level of detail. In [Fernandez et al., 2000] and [Fernandez and Yuan, 2000] patterns are depicted in a systematic way, through context, problem, forces, solution, consequences, known uses and related patterns. The solution part is described by class, state transition, sequence and activity diagrams. Patterns described in [Hamza and Fayad, 2002] adopt a similar strategy.

Our approach also uses those diagrams, but organised in a different way. Moreover, we include, more explicitly, some aspects that we found important, e.g., evolution, dependency, priorities and conflict resolution of requirements, anti-patterns.

To demonstrate the template utility and adequacy we filled up the gaps in the original Party pattern description (but note that we did not include all the sequence diagrams, for space reasons). The additional information makes the pattern more complete and easier to understand.

We believe that this template will contribute to provide more useful and organised descriptions of analysis patterns.

5 Conclusions

This paper discussed and compared the main characteristics and limitations of some approaches for requirements and analysis patterns, and appropriateness of the term "requirements patterns". Afterwards, we presented an approach to represent analysis patterns, i.e., a pattern for analysis patterns.

This was accomplished by defining a template that gathered elements from existing approaches and incorporated new ones that we considered that were missing. The approach was illustrated by applying it to the *Party* pattern described in [Fowler, 1997]. We believe that the approach presented will improve the specification of analysis patterns with more detailed information.

For future work we intend to apply the template to different patterns as a rigorous way of validation and define a process model for requirements that incorporates the approach described here.

References

[Araujo and Weiss, 2000] I. Araujo and M. Weiss (2002). "Linking Patterns and Non-Functional Requirements", PLoP 2002, Allerton Park, Monticello, Illinois, USA.

[Buschmann et al., 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal (1996). *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons.

[Brown et al, 1998] W. Brown, R. Malveau, H. McCormick, T. Mowbray (1998). *Anti-Patterns: Refactoring Software, Architectures and Projects in Crisis*, John Wiley & Sons.

[Chung et al., 2000] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos (2000), *Non-Functional Requirements in Software Engineering*, Kluwer.

[Fernandez and Yuan, 2000] E.B. Fernandez, X. Yuan (2000), "Semantic Analysis Patterns", 19th International Conference on Conceptual Modeling, ER2000, Salt Lake City, UT, USA, pp. 183-195.

[Fernandez et al., 2000] E. B. Fernandez, X. Yuan, S. Brey (2000). "Analysis Patterns for the Order and Shipment of a Product", PLoP 2000, Allerton Park, Monticello, Illinois, USA.

[Fernandez and Yuan, 2000] E. B. Fernandez, X. Yuan, "An Analysis Pattern for the Repair of an Entity", PLoP 2001, Allerton Park, Monticello, Illinois, USA.

[Fowler, 1997] M. Fowler (1997). *Analysis Patterns - Reusable Object Models*, Addison Wesley.

[Gamma et al., 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, New York, Addison-Wesley.

[Geyer-Schulz and Hahsler, 2001] A. Geyer-Schulz, M. Hahsler (2001). "Software Engineering with Analysis Patterns", Technical Report 01/2001, Institut für Informationsverarbeitung und -wirtschaft, Wien, Austria.

[Hamza and Fayad, 2002] H. Hamza, M. Fayad (2002). "A Pattern Language for Building Stable Analysis Patterns", PLoP 2002.

[Hamza and Fayad, 2003] H. Hamza, M. Fayad (2003). "The Negotiation Analysis Pattern", EuroPLoP 2003, Irsee, Germany.

[Jackson, 2000] M. Jackson, *Problem Frames: Analyzing and Structuring Software Development Problems*, Addison Wesley, 2000.

[Konrad and Cheng, 2002] S. Konrad, B. Cheng (2002). Requirements Patterns for Embebed Systems. IEEE Joint International Conference on Requirements Engineering, Essen, Germany, 2002.

[Robertson, 1996] S. Robertson (1996). "Requirements Patterns Via Events / Use Cases", The Atlantic Systems Guild. http://www.systemsguild.com/GuildSite/SQR/Requirements_Patterns.html

[Schmidt et al., 2000] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann (2000). *Pattern-Oriented Software Architecture. Volume 2: Patterns for concurrent and network objects*, John Wiley & Sons.

[Yuan and Fernandez, 2003] X.Yuan, E.B.Fernandez (2003). "An Analysis Pattern for Course Management", EuroPLoP 2003, Irsee, Germany.

[Zhen and Shao, 2002] L. Zhen, G. Shao (2002). "Analysis patterns for oil refineries", 9th. Pattern Languages of Programs Conference (PLoP 2002), Allerton Park, Monticello, Illinois, USA.