ELEPUS: EXTENDED LANGUAGE FOR PATTERN UNIFORM SPECIFICATION

A Thesis

Submitted to the Faculty

of

Purdue University

by

Sivakumar Chinnasamy

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 2000

– To my *Amma* and *Appa*.

ACKNOWLEDGMENTS

I thank you for your voices, thank you, / Your most sweet voices.

-William Shakespeare (1564-1616)

Writing a thesis is almost always a long, lonely journey. It is also perhaps the most memorable and cherishable achievement of a graduate student. There are quite a few people, without whose guidance, I could not have completed this thesis.

First and foremost, I profoundly thank *Prof. Rajeev R. Raje*, my advisor, for his valuable comments and guidance throughout the lifetime of this work. Without his creative insights I could not have mined the logical ones from the myriad stream of chaotic ideas that sprang up to my mind. He was always available and motivated me whenever I needed it. I benefited greatly from his guidance and knowledge. He was more than an 'advisor'- he was the proverbial friend, philosopher and guide for me.

I deeply thank my thesis committee members, *Prof. Zhiqing Liu* and *Prof. Andrew Olson* for their constructive comments and guidance during the discussions and for sparing the time for my thesis progress presentations in spite of their busy schedules. I have always benefited from Prof. Liu's knowledge and insight not only through his courses, but also through the discussions I had with him. Prof. Olson's patience is surpassed only by his sincerity, and I am grateful to benefit from his guidance in software engineering and Object-oriented methodology. His comments and suggestions were crucial in producing the final draft of this thesis.

I thank my parents, sister, friends and relatives for always standing by me. It was their unlimited love and care that carried me through in all the tough times. Finally, I thank the Department of Computer and Information Science and Prof. Mathew Palakal for providing me with the RIF Fellowship that supported me during the period of this research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Chinnasamy, Sivakumar, M.S., Purdue University, August, 2000. ELePUS: Extended LanguagE for Pattern Uniform Specification. Major Professor: Rajeev R. Raje.

Software design patterns capture proven design expertise and knowledge. Design patterns are specified in a format that facilitates understanding and usage. The existing specification of patterns are mostly informal and have many deficiencies. There are a few formal approaches proposed to address these issues. However, all existing approaches have many limitations. This thesis proposes a language, ELePUS (Extended LanguagE for Pattern Uniform Specification), for the formal specification of design patterns. ELePUS is an extended and enhanced version of LePUS, which is an existing formal specification language for software design patterns. LePUS allows the specification of only the structural component of design patterns. Such a specification is not comprehensive and does not completely depict the power of design patterns. ELePUS overcomes this drawback by providing the necessary constructs to specify other aspects of patterns, such as, Intent, Applicability, and Collaborations. ELePUS also addresses the other limitations of LePUS, by providing a better support for specifying the implementation-level details, behavioral relations, and object interactions. ELePUS, like LePUS, is a declarative language and utilizes the power of predicate calculus. This thesis presents the features of ELePUS, along with the syntax and semantics and its validation by specifying a substantial portion of the GoF catalog patterns.

CHAPTER 1. INTRODUCTION

Design patterns indicate efficient solutions to common re-occurring problems in software engineering that have been cataloged for the benefit of object-oriented (OO) architecture and design. Generally, patterns are extracted by exploring the past and present designs for the successful and efficient solutions to re-occurring software problems. Thus, they capture years of design experience and knowledge. The way in which a pattern is written, for wider dissemination and usage, is by itself a topic of major discussions and disagreements. There exist many formal and informal ways of specifying software design patterns. Although most of these pattern-writeups tend to be generic, there are a lot of domain or language specific patterns catalogs as well [1, 2]. The degree of the pattern specifications varies from superficial descriptions of the structure of patterns to an in-depth analysis of the consequences and trade-offs of using the patterns [3], to the specification of temporal behavior of patterns [4]. The current means of pattern extraction, cataloging and usage have many advantages and limitations. This research addresses many of these constraints and proposes solutions for them.

1.1. Problem Definition and Motivation

1.1.1. Pattern Cataloging - The Current Way

Existing pattern specification approaches include specification of the four components of a pattern (namely, *name, problem description, solution*, and *consequences* and *side-effects*) along with other information, such as the implementation tips or a detailed account of participating objects and their collaborations or inter-relationships with other patterns. Pattern catalogs, such as [3, 5, 6], use text-based descriptions and diagrams based on popular object notations like UML [7] and OMT [8]. Moreover, there exists no proper and a formal approach for naming the patterns. It remains an intellectual and intuitive exercise for the pattern writer, and is left to his/her own judgment. One of the popular specification method, used in [3], is:

- **Pattern Name and Classification:** Gives a meaningful name to the pattern along with any classification to which it belongs. The classification could be based on the structure or application of the pattern.
- **Intent:** Gives the objective of the pattern and states the problem solved by the pattern.
- **Also Known As:** Documents any other known or potential aliases for the pattern.
- **Motivation:** Provides verbal descriptions of the scenario that illustrate the design problem which warrant that particular design solution.
- **Applicability:** Gives the practical situations where the design pattern can be applied.
- **Structure:** Indicated as a graphical representation of the classes in the pattern using Object Modeling Technique (OMT). It can also be described using any valid object notation.
- **Participants:** Indicates the classes or objects involved in the solution.
- **Collaborations:** Lists the collaborations (relationships and dependencies) among the participants.
- **Consequences:** Mentions the tradeoffs and results of using the pattern.
- **Implementation:** States any implementation/language specific issues.

- **Sample Code:** Indicates a sample implementation, in a popular OO language, for the purpose of understanding.

- **Known uses:** Lists the proven implementations, from real systems, that use this pattern.

- **Related Patterns:** Mentions any closely related patterns and their differences and/or interdependence.

The advantages of the informal method of specification is that it is easy to understand and the applicability of patterns can be explained with examples. However, since the entire specification is done in a text-based explanation, using natural languages, it is prone to be ambiguous and vague. The success of any such catalog depends, to a great extent, on the ability and expertise of the pattern writer to precisely and clearly state the pattern. An ambiguous specification may lead to incorrect understanding and usage.

1.1.2. Limitations of Informal Methods & Object Notations

The limitations of informal specification methods have been widely reported [9, 10, 11]. Informal specifications are plagued by the following handicaps:

- **Precision:** The informal methods of specification are not always precise, and can lead to ambiguity in understanding the specification. They rely on verbal descriptions to convey special properties or constraints. Thus, the ability of the pattern writer to convey the tricky and trivial issues, in an unambiguous manner using verbal descriptions, becomes critical.

- **Limitations of Object Notations for pattern specifications:** Using object notations towards specifying design patterns is infeasible and inappropriate. There are inherent factors that contribute to the inexpressiveness of object notations while specifying design patterns [10]. These are:

– Object notations are to describe programs, which are constituted from the building blocks provided by the programming languages in which they are written. In contrast, Patterns are at a higher level of abstraction. They abstract design concepts and structures in a language-independent manner. So, applying the same technique to describe both is inherently flawed.

– A pattern inherently has many constraints like *'there exists an isomorphic relationship between class A and class B'*. There is no proper construct or vocabulary in the object notation to clearly specify such constraints and these are left to the mercy of verbal explanations.

– Object notations provide mechanisms to represent syntactically identifiable structures. As there is no one-to-one mapping between the syntax structures of the object notations and the design patterns, the object notations will fail to precisely represent patterns.

– Expressing generalizations is difficult by using object notations. By their very nature, patterns capture generalizations like *'each factory method produce a particular product object'*, which cannot be represented using the object notations in a straightforward manner.

– In most of the object notations, procedures are not considered first class citizens (first order entities), a privilege reserved only for the classes. This implies that the key properties of patterns, involving procedures or methods, can only be specified informally. This contrasts with the patterns, since patterns capture design concepts and any construct, whether classes or methods, that is needed for the realization of the design is given its due recognition.

1.1.3. Issues in Pattern Specification and Cataloging

The issues related to the proper use of design patterns and their specifications have been well studied and enumerated [12, 11, 13, 14, 15, 16]. A few ones related to cataloging patterns are given below:

- **Validation:** This is equivalent to a mapping of implementations to the patterns. There is no proper agreed-upon way to verify whether implementations strictly adhere to the patterns used in the design process. Failures of which, may lead to serious problems like traceability (tracing the pattern from the implementation). This negates the advantages like forming a documentation vocabulary, offered by patterns.

- **Mapping of the design to the implementation:** Even though the implementation does not gravely deviate from the pattern specification, they are at two different levels of abstraction. Patterns are at a higher level of abstractions that capture re-occurring design structures, while implementations are constructed from lower-level building blocks offered by the implementation language. So, a mechanism has to be devised to easily correlate patterns with their implementations.

- **Issues in Cataloging:** Cataloging of patterns involves categorization or classification of patterns, providing a meaningful nomenclature for patterns and formulating implementable guidelines on issues like Inclusion, Refinement and Duplication. The intuitive definitions of these issues are given below:

Refinement If two pattern write-ups overlap, to a great extent, they are said to refine one another. Since the degree of overlap is subjective, clear guidelines are needed to establish refinement.

Duplication Duplication refers to intentional or accidental specification of the same pattern under two different names. This negates the notion of unique design vocabulary offered by patterns.

Inclusion Inclusion refers to the addition of a candidate pattern into the existing pattern repositories (pattern catalogs). The issues of duplication and refinement should also be considered apart from other issues like usability or correctness of the candidate pattern.

These issues are not addressed by any of the existing informal pattern specification methods.

1.1.4. Formalization of pattern specifications

1.1.4.1. Formalization and Standardization

Formalization of patterns [4, 17, 18, 10] refers to the formalization of the pattern specifications that help to address the issues regarding pattern specification (discussed before, in section 1.1.3). The major aspect of formalization is to provide a precise and unambiguous specification of the intent, structure and behavior of the patterns. Apart from addressing those concerns, the formalization will allow the exploration of the automated mining of patterns. This will be possible if the design of the most, if not all, of the object-oriented systems are represented using the formal pattern specification process. Then the patterns can be automatically identified, since extracting information from the formal specifications can be automated easily when compared with their non-formal counterparts.

We define *Standardization* [12] as the process of applying a set of a commonly agreed upon and well-defined rules that complement the process of mining and cataloging the patterns. Already some de-facto rules are followed - such as 'the rule of three' (This rule has been suggested and quoted in the online patterns discussion groups [19]), which states, "for any design structure to qualify as a pattern, it should be implemented in at least three designs of different OO systems". [11] proposes a set of criterion to streamline the process of acceptance of a new pattern and the classification of a pattern as a 'refinement' of existing pattern. In order to provide more support for automated tools, to resolve ambiguity in the specifications, and to avoid duplications and resolve refinements, we need a more rigorous pattern specification methodology.

1.1.5. Feasibility and Practicality of Formalization

As per Christopher Alexander, the pioneer of the concept of design patterns in the field of Architecture, *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"* [20, 3]. This makes abstractness and transparency (of environment specific details) the key properties of design patterns. These properties are hard to preserve if the text-based informal definition alone is adopted for specifying design patterns. A precise specification of design patterns is one way of preserving these properties and also addressing the concerns stated in previous sections. Moreover, the feasibility of formalizing design patterns has been theoretically proved [21].

1.1.5.1. Resistance to formalism

The common apprehensions/myths voiced against pattern formalizations in various fora including online discussion groups and informal academic discussion [10], and clarifications for them are presented below:

- *As patterns are already abstractions, formalization will lead to another level of cumbersome abstraction.* Only the representation of patterns is formalized. This does not necessitate another level of unnecessary abstraction.

- *Formalization of design patterns makes it difficult to learn and have the overhead of learning the formal languages as well.* Although this argument is true to a certain extent, the benefits of formalization far outweigh the additional work involved. Also, the person writing the specification need not be the one who writes the pattern. Reading formal specifications is easier than writing a specification, and hence the overhead may not be too excessive.

- *A formalized solution narrows the applicability of patterns considerably. It will become very difficult to create new valid variants as formalizations restrict the scope of the design structure of patterns.* This argument assumes that the formalized specifications cannot represent generalizations. Since formalization provides a better tool support for patterns, it actually widens the applicability of patterns. Also [10] proves that formal representations can capture generalizations in elegant ways.

- *Patterns are fuzzy by nature and not concrete and are not served well by a precise notation. They can be learnt and appreciated only through experience.* Although there is no denying the fact that patterns have emerged out of real-life experiences, this argument can prohibit a wider spread of patterns in the software development as the new object-oriented designers do not have much experience to appreciate patterns easily.

From the above discussions, it can be inferred that there are no proven limitations to formalization. Most of the resistance is because of misunderstandings and prejudice regarding formal specifications. There exist many formal specification languages and methodologies. However, no elaborate studies are available for their suitability with respect to formal specification.

From the discussions presented so far, it can be concluded that

1. Informal Methods of pattern specification have many limitations.

2. Formal specification of design patterns is feasible and does not have any serious limitations.

3. Formal pattern specifications help harness the full potential of design patterns.

4. There is no comprehensive survey or evaluation of formal specification languages for design patterns.

1.2. Objectives: Statement of Goals

1. Create a comprehensive survey and evaluation of various formal specification methodologies for specifying design patterns, and analyze their merits and demerits.

2. Develop a comprehensive specification language making use of the proven techniques and contributions of the existing formal methods, as established by the evaluation.

3. Define and describe the semantics and syntax of the proposed language and indicate its applicability with respect to various aspects of design patterns specification and usages. Also, validate the language by applying it to specify the design patterns from [3].

4. Identify and propose potential research or tool support that will make use of the developed language to exploit the full potential of design patterns.

1.3. Contributions of this thesis

The contributions of this thesis are:

1. Surveys and evaluates the existing formal methods and provides a cause-effect relationship of the techniques or characteristics of a language/methodology and its applicability. This survey establishes that the formal basis of LePUS (LanguagE for Pattern Uniform Specification) [10, 22] as sufficient and powerful for achieving the stated goals.

2. Developes a language ELePUS (Extended LePUS), which is an enhanced and extended version of LePUS that can specify other aspects of design patterns like Intent, Applicability, Structure and Contributions. The thesis also presents the language constructs, their syntax and semantics and BNF grammar for the language.

3. Design patterns in the Gang-of-Four catalog [3] are specified using ELePUS, thus validating the expressiveness and comprehensive nature of ELePUS.

1.3.0.2. Organization of this thesis

This thesis is organized into 6 chapters. The introduction of the problem domain, problem statement, definition of goals and the organization of the thesis were presented in this chapter. Chapter 2 provides a survey of the existing formal languages and methodologies for pattern specification. It lists a set of well defined criteria for the evaluation of the existing formal methods and provides an assessment by applying each of the formal languages to specify two design patterns from [3]. The mathematical basis of LePUS or ELePUS, limitations of LePUS and additions of ELePUS are provided in Chapter 3. Chapter 4 gives a comprehensive listing of all ELePUS language constructs, their definition of syntax and semantics. It also presents the BNF grammar of ELePUS. Each of the ELePUS constructs is accompanied by an illustration of its application in specifying a known design pattern from [3]. Chapter 5 validates ELePUS by assessing it with respect to the evaluation criteria proposed in Chapter 2 and by specifying a substantial number of GoF patterns using ELePUS. Finally, this thesis concludes with a discussion on ELePUS and descriptions of proposed tool support that uses ELePUS to exploit the full power of design patterns in Chapter 6.

CHAPTER 2. EVALUATION OF EXISTING FORMAL PATTERN SPECIFICATION LANGUAGES

True genius resides in the capacity for evaluation of uncertain, hazardous, and conflicting information.
- Winston Churchill (1874-1965)

The previous chapter discussed the need for a comprehensive survey and evaluation of specification methodologies or languages for the formal specification of software design patterns. Design patterns are abstractions and do not require a specification catering to low-level details. Moreover, as patterns cut across module boundaries, a specification that cannot support such features cannot effectively specify design patterns. These observations, along with the capability to provide an effective formal specification of patterns, were applied in identifying the languages or methodologies appropriate to specify design patterns.

The scientific community has proposed a number of languages or methodologies for formally specifying software designs. Most of these languages have a more generic focus than what is required for specifying design patterns. For example, formal specification languages, like Z [23, 24], Larch [25], and VDM [26], are too complex and also provide limited support for object-orientation, and have too much of detail [27]. Architectural Description Languages, like Intercol [28] or Rapide [29], are meant for describing the properties and interactions of modules of the software architecture, and so are at a disadvantage with respect to design patterns, which cut across module boundaries [10]. Based on these arguments with respect to their suitability for pattern specifications, *Contracts, DisCo, Constraint Diagram notation, LePUS* and *CDL* were selected for a comprehensive evaluation and assessment.

2.1. The Formal Approaches for Pattern Specification

This section gives a brief overview of each of the above mentioned formal methods. The features of these methods will become more clear as some examples are discussed in the next section. For comparison purposes, an abridged informal specification [3] of the design patterns, used in the evaluation, is also provided in this section.

2.1.1. Contracts

Contracts [30] is a specification language for specifying behavioral compositions and the obligations on participating objects. Contracts, explicitly and abstractly, capture the behavioral dependencies amongst cooperating objects. Contracts define type obligations, which capture the participants and contractual obligations, which specify constraints on behavior and capture the behavioral dependencies between objects. A contract also defines preconditions on participants required to establish the contract, and the invariant to be maintained by these participants. The contract stays clear of implementation. So to bridge the implementation and contract, a conformance declaration is used, which specifies how a particular class implementation meets a particular object's contractual obligations. This is a invariant-based specification and can be considered as an extension to first order logic.

2.1.2. DisCo

DisCo (*Dis*tributed *Co*-operation) [31] is a formal specification framework for reactive systems. It incorporates a specification language, a methodology for developing specifications using the language, and a tool support for the methodology. DisCo has been applied for formal specification of design patterns [4]. DisCo can specify and model object interactions at a high level of abstraction. It also has provisions for representing the temporal behavior of patterns. It claims to formalize the temporal behavior of patterns in terms of high-level abstractions of communication. This is a textual language and is an extension of first order predicate calculus.

2.1.3. Constraint Diagram

The Constraint Diagram notation [32] aims at providing a way of achieving purely visual, yet formal, specification of patterns using constraint diagrams [33]. It has a three-tier model for the representation of patterns and uses UML for specifying the dynamics of the pattern. The first model, "role-model" is the most abstract and depicts only the essential spirit of the pattern, excluding inessential application/domain specific details. The second model, "type-model" provides a concrete refinement of the "role-model" by adding abstract state and operation interfaces. The final model, "class-model" realizes the "type-model", thus, deploying the underlying pattern in terms of concrete classes. It also supports the diagrammatic pattern specifications with textual supplements. These textual supplements serve two purposes: a) of reinforcing diagrams with supporting information, and b) to disambiguate pattern diagrams.

2.1.4. LePUS

The LanguagE for Pattern Uniform Specification (LePUS) [17, 18, 10] is a rigorous formal language for pattern specification and uses Higher Order Monadic Logic [34] and predicate calculus. This language has a well established formulae-based representation and a complementing visual representation. Although the visual representation is not fully complete, proper mappings exist between the formulae-based specification and visual representation. LePUS is based on a compact but a sufficient set of building blocks, which are used to represent patterns. LePUS formulae manifest design patterns in terms of logic statements. LePUS transcribes patterns to formulae and every program is represented as a model. It also provides suggestions on repository management issues based on the underlying mathematical model. ELePUS, the language proposed in this research, is based on LePUS and hence, LePUS will be discussed in more detail in the next chapter.

2.1.5. CDL

Category Description Language (CDL)[35] is intended to describe and name the environmental or system constraints of software systems in a formal fashion. It is not intended to specify design patterns, but its methodology may find some application in specifying design constraints of design patterns. It is a first order logic language of parse trees that can be used to express constraints over the behavior and relations of collaborators in any design or design pattern in terms of the implementation language. It is not a specification language in the general sense, but acts as a specification language to facilitate validation of patterns.

Although, the concept of constraint specification of CDL may be useful in a pattern specification language (as how to model platform specific constraints when discussing pattern implementations), as such it is not capable of being a pattern specification language (due to its narrow focus on just specifying the environmental or system constraints), and so is not further discussed. It was studied for its ability to specify design constraints. However, no practical evaluation of CDL has been done for the above stated reasons.

2.2. Evaluation of Formal Pattern Specification Languages

This section describes the evaluation criteria and evaluates the above mentioned approaches by applying them to two well known design patterns. The objectives of the evaluation process are to:

- Identify the merits and demerits of the existing formal pattern specification languages.

- Identify an ideal specification scheme or language among the candidates if possible.

- If no ideal specification language is available, identify the most suitable language(s) for the purpose of pattern specification and elaborate the additions or modifications to be done to make any promising existing language to address its limitations.

2.2.1. The Evaluation Criteria

The evaluation criteria [36] fall broadly under the three categories:

Formalism: Mathematical foundation, precise visual notation, and conciseness constitute Formalism.

Comprehensiveness: Ability to specify structural aspects, behavioral aspects, and design constraints determine the comprehensiveness of a pattern specification language.

Versatility: The factors defining versatility are: scalability, support for object-orientation, complementing object notation, ease-of-use, multi-level representation, representation of low-level details and potential for pattern repository management.

A brief description of each of the above mentioned criteria is given below. Unless otherwise stated, each language or formal methodology will be ranked either *Poor, Fair, Good* or *Very Good* and a brief ranking scheme for each criteria is also provided.

- **Mathematical Foundation:**
  - Measures the rigorousness of the mathematical model underlying the approach.
  - This criterion ranks the evaluated languages based on the rigorousness of the mathematical model, formal definitions of language constructs and their relationships, and the suitability of the mathematical model to specify design patterns.
- **Precise Visual Notation:**

  - Measures the availability of an unambiguous and self explaining visual notation for the structural part of the pattern specification.
  - This criterion ranks the evaluated languages based on preciseness, completeness and independence of the visual notation.

- **Conciseness:**
  - Measures the average size of the specification.
  - This criterion ranks the evaluated languages based on the approximate number of language constructs required to capture all the aspects of patterns and the number and complexity of the language constructs.

- **Specification of Structural Aspects:**
  - Measures the capability of a language to specify the structural aspects of a pattern.
  - This criterion ranks the evaluated languages based on a complete and unambiguous specification of the structural aspects of patterns, including the participants of the pattern and all their structural relationships.

- **Specification of Behavioral Aspects:**
  - Measures the capability of a language to specify the behavioral aspects of a pattern.
  - This criterion ranks the evaluated languages based on their abilities to capture all the behavioral relationships and dependencies between participants.

- **Specification of Constraints:**
  - Measures the ability of the language to specify system design constraints and environmental limitations.

- This criterion ranks the evaluated languages based on their provisions for specifying system constraints, environmental limitations and other Object-oriented design constraints, such as the restrictions on number of object instantiations, accessibility restrictions, etc.

- **Scalability:**
  - This criterion ranks the evaluated languages based on their abilities to specify compositions of patterns and reuse of the pattern specifications in complex design specifications.

- **Complementing Object Notations:**
  - Denotes the availability of a well-defined mapping between the language and a popular object notation.
  - Ranking is either *Nil* (not available/applicable) or the name of the object notation that has a well defined mapping with this language.

- **Support for Object-orientation:**
  - Measures the provisions for specifying Object-oriented properties of the pattern.
  - This criterion ranks the evaluated languages based on the degree of direct representations available for specifying OO properties like inheritance, abstraction, encapsulation, dynamic binding and the notion of classes and objects.

- **Ease of Use:**
  - This criterion ranks the evaluated languages based on a comparative ease of use in learning the language and applying it to specify design patterns. Simplicity and size of the language vocabulary are the major determinants.

- **Multi-level representation:**

  - Measures the ability and provisions of the language to specify patterns at more than one level of abstraction.
  - This criterion ranks the evaluated languages for the availability of the provision or the lack of it (*Nil* or *Exists*).

- **Representation of the low-level details:**
  - Measures the ability to represent of the low-level details, like the number of object instantiation, etc., needed to specify properties of patterns that are implementation specific.
  - This criterion ranks the evaluated languages based on provisions for implementation specific constraints like number and definitions of functions, parameter types and their equivalence, etc.

- **Potential for Pattern repository Management:**
  - Pattern repository management involves resolving the issues of refinement, duplication, inclusion and validation.
  - This criterion ranks the evaluated languages based on the ability of the language to support or facilitate resolution of the above mentioned issues.

2.2.2. The Evaluation Process

The evaluation process involved using the previously discussed formal specification approaches to represent two design patterns. Based on the experiences and analyses of the representation scheme, each method was graded against the evaluation criteria. For the analysis to be effective, the selection of the patterns chosen for the study was crucial. The two patterns chosen were Abstract Factory and Factory Method from [3]. The reason for choosing these two patterns was that they have many similarities in structure and function and yet have distinct differences. In [3], both Factory Method and Abstract Factory are categorized under the purpose of Creation, but in different scopes (static scope or compile time binding and dynamic scope or run-time binding). Factory Method falls under the category of Class-patterns implying that it deals with relationships between classes and their subclasses that are staticly fixed at compile-time. On the other hand, Abstract factory is an Object-pattern and is more dynamic (changeable at run-time). Factory method *"defines an interface for creating an object, but let subclasses decide which class to instantiate. In other words, it defers instantiation to subclasses."* [3]. Abstract factory *"provides an interface for creating families of related or dependent objects without specifying their concrete classes."* [3]. Abstract factory patterns are often implemented with Factory Method patterns.

One objective of the evaluation was to verify and document how well the nuances of these two patterns are captured by various specifications. The results of the evaluations and inferences drawn are shown in the tables 2.1 and 2.2. In the remainder of this section, the specification of Abstract Factory pattern and Factory Method pattern using the various formal methodologies are provided. For the purpose of comparison, an abridged version of specification using the informal GoF methodology [3] is also provided.

2.2.3. GoF Representation

This section provides the representation of patterns as given in the GoF catalog [3]. For the sake of brevity, only the sections most pertinent to the specification of the participants, structure and behavior of the patterns are presented.

2.2.3.1. Factory Method

Intent: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiations to subclasses.

Applicability: Use Factory Method pattern when:

- a class cannot anticipate the class of objects it must create.
- a class wants its subclasses to specify the objects it creates.
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Structure: See Fig.2.1

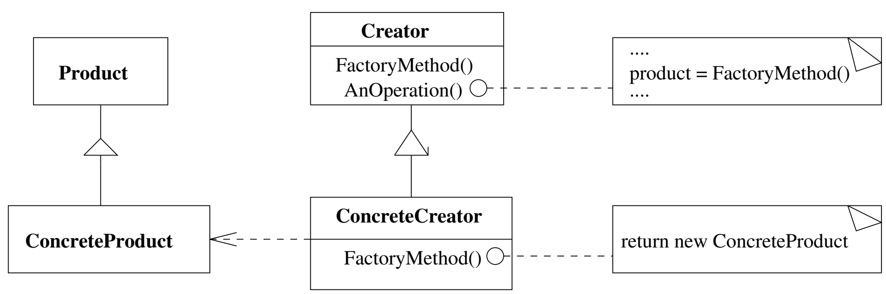

Figure 2.1. Factory Method Pattern

Participants:

Product defines the interface of objects the factory method creates.

ConcreteProduct implements the product interface.

Creator declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object. Creator may call the factory method to create a Product object.

ConcreteCreator overrides the factory method to return an instance of a ConcreteProduct.

Collaborations: Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct.

2.2.3.2. Abstract Factory

Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Applicability: Use Abstract Factory pattern when:

- a system should be independent of how its products are created, composed and represented.
- a system should be configured with one of multiple families of products.
- a family of related product objects is designed to be used together, and you need to enforce this constraint.
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Structure: See Fig.2.2.

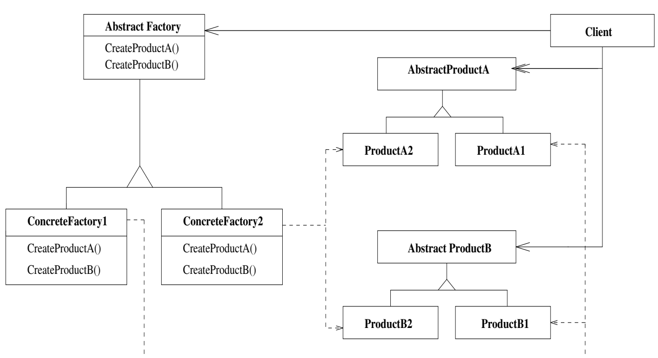

Figure 2.2. Abstract Factory Pattern

Participants:

AbstractFactory declares an interface for operations that create abstract product objects.

ConcreteFactory implements the operations to create concrete product objects.

AbstractProduct declares an interface for a type of product objects.

ConcreteProduct defines a product object to be created by the corresponding concrete factory. Also, implements the AbstractProduct interface.

Client uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Collaborations:

- Normally a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different concrete factory.
- AbstractFactory defers the creation of product objects to its ConcreteFactory subclass.

2.2.4. Specification Using Contracts

2.2.4.1. Factory Method Pattern:

This section expresses the behavioral relationships among participants in Factory Method Pattern as Factory Method Contract. As noted earlier in the section 2.1.1, the contract specification involves defining the participants, their contractual obligations (type and causal), the invariant(s) to be preserved during the course of the contract, and the preconditions on participants to establish the contract and methods which instantiate the contract. Type obligations capture interface and type details, like Creators needed to hold a variable of type Product and ConcreteCreators needed to hold a ConcreteProduct variable. Causal obligations capture the behavior of the participants. For example, the factoryMethod() function of ConcreteCreator returns an object of type ConcreteProduct and that there exists a one-to-one and isomorphic relationship between factoryMethod and ConcreteProduct, which is the invariant to be maintained. The invariant elegantly captures the fact that at any instant, the factoryMethod() function of ConcreteCreator returns an object of type ConcreteProduct and that there exists a one-to-one and isomorphic relationship between factoryMethod and ConcreteProduct. The Instantiation statement of a contract generally specifies the methods that need to be executed for the contract to be established. It does not apply in the examples under discussion, so the Instantiation sections in the following examples are left empty.

Since, Contracts are defined independently of classes, they also provide separate mappings between the class implementations and participant specifications, called "Conformance Declarations". Abstractly, a conformance declaration contains a set of bindings of the form $a : \alpha \leftarrow b : \beta$, which maps an identifier b of type β, defined in a class, to an identifier a of type α in a participant [30].

contract FactoryMethod

    Product supports [ ]

    ConcreteProducts: Set (ConcreteProduct)

        where each ConcreteProduct supports [ ]

    Creator supports [

        product: Product

        factoryMethod(): Product → return product

    ]

    ConcreteCreators: Set (ConcreteCreator) where

        each ConcreteCreator supports [

        cp: ConcreteProduct

        factoryMethod(): ConcreteProduct →

        $\{ \; factoryMethod() \; returns \overset{i}{\leftrightarrow} \; cp \; \}$

    ]

    Invariant:

ConcreteCreator.factoryMethod():

ConcreteProduct $\rightarrow$

$\langle\forall$ cc: cc $\in$ ConcreteCreators,

$\forall$ cp: cp $\in$ ConcreteProducts,

cc.factoryMethod $returns^{\overset{i}{\leftrightarrow}}$ cp $\rangle$

Instantiation:

$\langle \quad \rangle$

end contract

Note:

The predicate $returns^{\overset{i}{\leftrightarrow}}$ implies that the relationship is one-to-one and isomorphic.

Conformance Declaration:

class AbstProduct conforms to product in

FactoryMethod

$\langle$ requires subclass $\rangle$

end conformance

class ConcProduct conforms to ConcreteProduct in

FactoryMethod

$\langle$ inherits from AbstProduct $\rangle$

end conformance

class AbstCreator conforms to Creator in

FactoryMethod

AbstCreator supports

product: AbstProduct;

factoryMethod(): AbstProduct $\rightarrow$

return product;

$\langle$ requires subclass $\rangle$

end conformance

class ConcCreator conforms to ConcreteCreator in
FactoryMethod

⟨ inherits from AbstCreator ⟩

ConcCreator supports

cp: ConcProduct;

factoryMethod(): ConcProduct $\rightarrow$ return cp

end conformance

2.2.4.2. Abstract Factory Pattern:

The definition of participants, type obligations and causal obligations are the similar to that of Factory Method pattern shown in previous subsection. The invariant shows the fact that a family of products can be created by the respective create operations in the concrete factory objects. Similar to the previous representation, the conformance declaration specifies the bindings of the class level implementation to the contractual specification.

contract AbstractFactory

AbstractProductA supports [ ]

AbstractProductB supports [ ]

AbstractFactory supports [

apA: AbstractProductA

apB: AbstractProductB

CreateProductA(): AbstractProductA $\rightarrow$ return apA

CreateProductB(): AbstractProductB $\rightarrow$ return apB

ProductA: Set ($ProductA_i$) where

each $ProductA_i$ suppports [ ]

ProductB: Set ($ProductB_i$) where

each $ProductB_i$ suppports [ ]

ConcreteFactory: Set ($ConcreteFactory_i$) where each $ConcreteFactory_i$ supports [

$cpA_i : ProductA_i$

$cpB_i : ProductB_i$

$createProductA_i() : ProductA_i \rightarrow$ return cpA_i

$createProductB_i() : ProductB_i \rightarrow$ return cpB_i

]

Invariant:

$\forall ConcreteFactory_i \in ConcreteFactory : \{$

$ConcreteFactory_i.createProductA() :$

$ProductA_i \rightarrow$

$\langle \forall cpA_i : cpA_i \in$ ProductA,

$\forall$ cf: cf $\in$ ConcreteFactory,

cf.createProductA $returns^{\overset{i}{\leftrightarrow}}$ cpA $\rangle$

$\bigwedge ConcreteFactory_i.createProductB() :$

$ProductB_i \rightarrow$

$\langle \forall cpB_i : cpB_i \in$ ProductB,

$\forall$ cf: cf $\in$ ConcreteFactory,

cf.createProductB $returns^{\overset{i}{\leftrightarrow}}$ cpB $\rangle$

$\}$

Instantiation:

$\langle \quad \rangle$

end contract

Note:

The predicate $returns^{\overset{i}{\leftrightarrow}}$ implies that the relationship is one-to-one, onto and isomorphic.

Conformance Declaration:

class AbstProductA conforms to AbstractProductA in AbstractFactory

⟨ requires subclass ⟩

end conformance

class AbstProductB conforms to AbstractProductB in

AbstractFactory

⟨ requires subclass ⟩

end conformance

class $ConcProductA_i$ conforms to $ProductA_i$ in

AbstractFactory

⟨ inherits from AbstProductA; ⟩

end conformance

class $ConcProductB_i$ conforms to $ProductB_i$ in

AbstractFactory

⟨ inherits from AbstProductB; ⟩

end conformance

class AbstFactory conforms to AbstractFactory in

AbstractFactory

AbstFactory supports

apA: AbstProductA;

apB: AbstProductB;

createProductA(): AbstProductA $\rightarrow$ return apA

createProductB(): AbstProductB $\rightarrow$ return apB

⟨ requires subclass ⟩

end conformance

class $ConcFactory_i$ conforms to $ConcreteFactory_i$ in

AbstractFactory

⟨ inherits from AbstCreator ⟩

$ConcFactory_i$ supports

$cpA_i : ConcProductA_i$;

$cpB_i : ConcProductB_i$;

createProductA(): $ConcProductA_i \rightarrow$ return cpA_i

createProductB(): $ConcProductB_i \rightarrow$ return cpB_i

end conformance

2.2.5. Specification Using LePUS

LePUS [22] is a declarative language, which is a subset of HOML [34]. LePUS specifies each pattern as a set of variables and the relationship among them. LePUS variables include *ground variables* (functions and classes), *hierarchy variables* (inheritance class hierarchies) and higher dimension variables (ranging over higher dimension (i.e., sets of) entities). The relationships among participants include ground relations (which are 'simple' and have direct implementation in OO languages), generalized relations, which build systematically from ground relations, and 'commuting' relations. An overview of the LePUS constructs and their semantics is presented in Chapter 3. A few of the LePUS constructs are provided here for understanding the specification of the two patterns represented using LePUS.

Production (m,p) indicates that m creates and returns an object of class p.

Return-Type (m,p) indicates that the return type of function m is class p.

$Commute_{R_1,R_2}$ indicates a commutative relation between R_1 and R_2.

Commute Relationship is defined as follows:

Given variables $V,\ W$, and binary regular relations $1^{\leftrightarrow}$, $2^{\leftrightarrow}$ (each may be a composition of regular relations), then,

$Commute_{1,2}$ $(V,\ W)$ indicates that 1 and 2 are *equal* in $V,\ W$. This is explained by an analogy from graph theory: In the graph defined by the vertices $V_1, \ldots, V_n$ and the edges $\beta_1 \ldots \beta_n$, $Commute_{\beta_1 \ldots \beta_2}(V_1, \ldots, V_n)$ indicates that for every pair of vertices V_i and V_j, all the paths in the graph from V_i to V_j are equal, in the sense that each path in the graph determines a composition of relations and these relations are equal in V_i and V_j.

A *Clan* is a special case of function sets. All the functions involved in a particular *clan* share the same signature, and each is defined in a different class (of a given set). Members of a *clan* bear relationships which typically allow dynamic selection of functions.

*Tribe*s are simply sets of clans with additional commonality.

The three types of *generalized relations* are:

(unary) α(V) $\rightarrow \forall$ v $\in$ V: α(v)

(total) $\beta^{\rightarrow}(V, W) \rightarrow \forall$ v $\in$ V $\exists$ w $\in$ W: β(v,w)

(regular) $\beta^{\leftrightarrow}$(V,W) indicates that β is an *invertible* function (1:1 and *onto*) from V to W.

Class, function and hierarchy variables are represented as c, C, C^d, f, F, F^d, H, H^d, depending upon their dimensions. The specification of both the patterns using LePUS is reproduced verbatim from [22]. The mapping between the formula based notation and visual notation of LePUS, is outside the scope of this paper. However, for the sake of understanding, some definitions for the visual constructs used in the examples, are given below:

Rectangles denote classes, while shaded rectangles denote set of classes. Ellipse denote a function, while shaded ellipse denote a function-set. Similarly, triangles and shaded triangles represent hierarchy variables and Set-of-hierarchy variables respectively.

2.2.5.1. Factory Method Pattern

The arrow between the factory-method function and Product class denotes 'Production' relationship. The arc (with embedded triangles) circumscribes the relations and domains exhibiting 'Commute' relationships. $\exists factory\text{-}method \in 2^F$, $Creators$, $Products \in H$:

$clan(factory\text{-}method, Creators) \wedge$

$Production^{\leftrightarrow}(factory\text{-}method, Products) \wedge$

$Return\text{-}Type^{\leftrightarrow}(factory\text{-}method, Products) \wedge$

$Commute_{Return\text{-}Type, Creation}(factory\text{-}method, Products)$

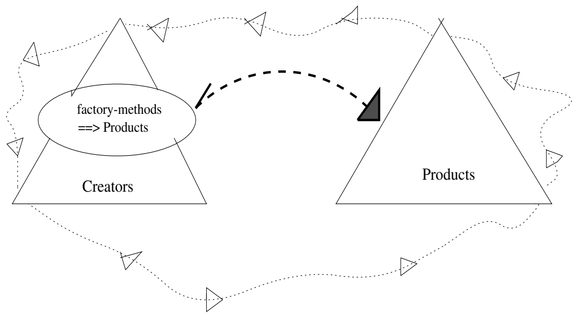

Figure 2.3. Factory Method

2.2.5.2. Abstract Factory Pattern

As in the previous examples, the arrow between the factory-method tribe and Products class-set denotes 'Production' relationship. The arc (with embedded triangles) circumscribes the relations and domains exhibiting 'Commute' relationships. $\exists Factory\text{-}Methods \in 2^{2^F}$, $Creators \in H$, $Products \in 2^H$:

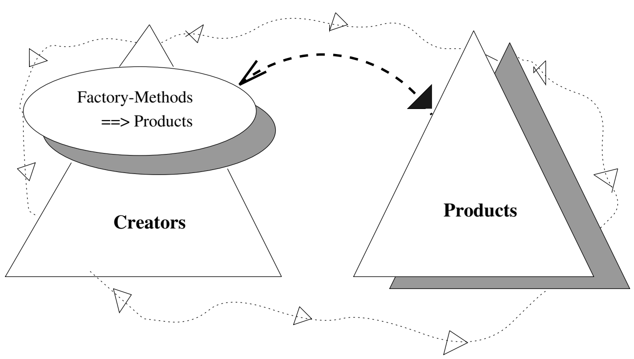

Figure 2.4. Abstract Factory

$$tribe(Factory\text{-}Methods, Creators) \wedge$$
$$Production^{\leftrightarrow}(Factory\text{-}Methods, Products) \wedge$$
$$Return\text{-}Type^{\leftrightarrow}(Factory\text{-}Methods, Products) \wedge$$
$$Commute_{Return\text{-}Type, Creation}(Factory\text{-}Methods, Products)$$

2.2.6. Specification Using DisCo:

DisCo [4] provides a way for the formal specification of participants of a pattern and its temporal behavior. It allows for specifying classes and inheritance relationships in a simple and intuitive way. It provides for the introduction of *classes, relations* and *actions*. Classes are formulae defining the form of possible *objects*. Relations are used for associating objects with each other. A relation is generally defined in the form $relation\ (n).R.(m)\ :\ C \times D$, where relation R associates n instances of class C with m instances of D. Actions are atomic units of execution, and consist of a list of required participants and parameters, an enabling condition, and the definition of state changes caused by an execution of the action. Classes and actions can be extended by *refinement*. A class D derived from a base class C can be specified as $class\ D\ =\ B\ +\ \{\cdots\}$. In the following specifications, the text following the # symbol is a comment, explaining the reasoning behind the statement, and is typeset with a different font from rest of the specifications for visibility.

2.2.6.1. Factory Method Pattern

class Product

Defines a class Product

class ConcProduct = Product + { ... }

class ConcProduct inherits from class Product

class Creator = { product : Product}

class Creator has a datum, product of type Class Product

class ConcCreator = Creator + { concProduct : ConcProduct }

relation (0..1).$created.(*)$: $Creator \times Product$

A relation, created, is defined between 0 to 1 instance of class Creator and any number of instances (denoted by ∗)of class Product

relation (0..1).$ConcCreated.(*)$: $ConcCreator \times ConcProduct$

Actions

FactoryMethod $(c : Creator\,; p^* : \, Product) :$

$\rightarrow c.Created'.p$

$\wedge p \;=\; product$

This action specification implies that after p is created, it will be assigned to a product object. This indicates the nature of the action which is supposed to dynamically produce different products. Marking participant p with an asterisk denotes a fairness *requirement, stating that if an object could repeatedly take this role in this action, the action will be executed for the object. Such requirements are essential for liveness properties [4]. Unprimed and primed variables refer to the values of variables before and after the execution of the action, respectively, thus defining the state change caused by the execution of an action.*

ConcFactoryMethod $(cc : \, ConcCreator,$

$cp^* : \, ConcProduct) :$

$refines\ FactoryMethod(c : Creator = cc;\ p : \ Product = cp)$

$for\ c \in ConcCreator,\ p \in ConcProduct$

$\wedge cc.ConcCreated'.cp$

$\wedge cp = ConcProduct$

This specification indicates that concFactoryMethod refines/overrides FactoryMethod, by indicating the changes in the required participants and the actions on them.

2.2.6.2. Abstract Factory Pattern

class AbstractFactory = {apA: AbstractProductA, apB: AbstractProductB }

class AbstractFactory is defined with two data entities, apA of type class AbstractProductA and apB of type AbstractProductB.

class AbstractProductA

class AbstractProductB

class ConcreteFactory1 = AbstractFactory +

{a1: ProductA1, b1: ProductB1}

class ConcreteFactory2 = AbstractFactory +

{a2: ProductA2, b2: ProductB2}

class ProductA1 = AbstractProductA + { ...}

class ProductA2 = AbstractProductA + { ...}

class ProductB1 = AbstractProductB + { ...}

class ProductB2 = AbstractProductB + { ...}

relation (0..1).$Created.(*)$: $AbstractFactory \times$

AbstractProductA

relation (0..1).$Created.(*)$: $AbstractFactory \times$

AbstractProductB

relation (0..1).$ConcCreated.(*)$: $ConcreteFactory \times$

ProductA1

relation (0..1).$ConcCreated.(*)$: $ConcreteFactory \times$

ProductB1

relation(0..1).$ConcCreated.(*)$: $ConcreteFactory \times$

ProductA2

relation(0..1).$ConcCreated.(*)$: $ConcreteFactory \times$

ProductB2

<u>Actions</u>

AbstractCreateProductA (af: AbstractFactory,

apa: AbstractProductA):

$\rightarrow$ $af.Created'.apa$

$\wedge$ apa = apA

ConcreteProductA1(cf: ConcreteFactory1, pa1: ProductA1):

refines AbstractCreateProductA (af: AbstractFactory = cf,

apa: AbstractProductA = pa1)

for af $\in$ ConcreteFactory1, apa $\in$ ProductA1

$\wedge$ $cf.ConcCreated'.pa1$

$\wedge$ pa1 = a1

ConcreteProductA2(cf: ConcreteFactory2, pa2: ProductA2):

refines AbstractCreateProductA (af: AbstractFactory = cf,

apa: AbstractProductA = pa2)

for af $\in$ ConcreteFactory2, apa $\in$ ProductA2

$\wedge\ cf.ConcCreated'.pa2$

$\wedge$ pa2 = a2

AbstractCreateProductB (af: AbstractFactory,

apb: AbstractProductB):

$\rightarrow af.Created'.apb$

$\wedge apb = apB$

ConcreteProductB1 (cf: ConcreteFactory1, pb1: ProductB1):

$refines\ AbstractCreateProductB$

$(af : AbstractFactory = cf,$

$apb : AbstractProductB = pb1)\ for$

$af \in ConcreteFactory1, apb \in ProductB1$

$\wedge cf.ConcCreated'.pb1$

$\wedge pb1 = b1$

ConcreteProductB2(cf: ConcreteFactory2, pb2: ProductB2):

$refines\ AbstractCreateProductA(af :$

$AbstractFactory = cf,$

$apb : AbstractProductB = pb2)$

$for\ af \in ConcreteFactory2, apb \in ProductB2$

$\wedge cf.ConcCreated'.pb2 \wedge pb2 = b2$

2.2.7. Precise Visual Specification Using Constraint Diagrams

As discussed in section 2.1.3, the constraint diagram based notation [32, 33] provides for pattern specifications in three different levels of abstraction. The specification of Abstract Factory pattern is reproduced from [32] in figures 2.5, 2.6, and 2.7, while the specification of Factory Method pattern was created as a part of the evaluation process. To specify the dynamics of the patterns, UML sequence diagrams may be used. However, since the limitations of object notations has been already discussed (in section 1.1.2), the UML sequence diagrams to specify the pattern dynamics are not provided here. Like any other object notation, UML has limitations in specifying aspects of patterns other than structure, such as Intent, Applicability, etc.

2.2.7.1. Factory Method Pattern

From the figures 2.5, 2.6, and 2.7, it is apparent that the role-model provides an

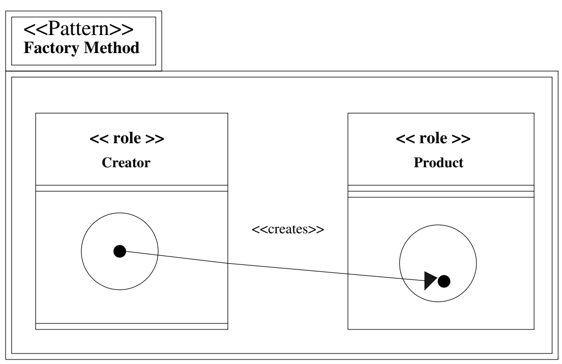

Figure 2.5. Factory Method - Role Model

abstract description of the structure of co-operating objects along with their static and dynamic properties. The type model is obtained by adding application-domain specific details to the role-model. The constraint diagrams are used to specify the role-model. Constraint diagrams depict sets as Venn diagrams. An arbitrary number of a set is depicted via a dot within or on the edge of the set. This role-model is

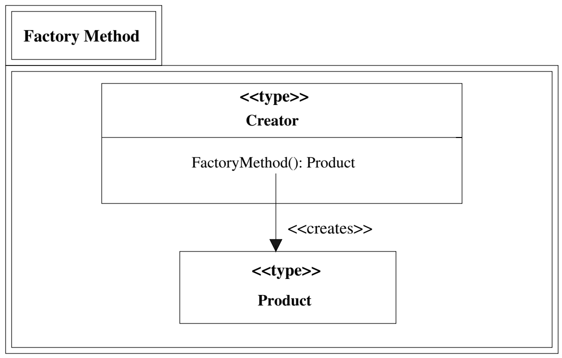

Figure 2.6. Factory Method - Type Model

refined into a type-model by adding the information of the method that produces the products. In other words, type model makes a commitment to concrete-operation syntax from the role-model. This type-model is further refined/realized into the class model, where the behavioral semantics, instances and state structure (inherits from, abstract class, etc.) are concretized. Hence, this three-tier model of specification varies from capturing the design concepts in the purest form (role-model) to specifying a realization (implementation) of the pattern (class-model).

2.2.7.2. Abstract Factory Pattern

See Figures 2.8, 2.9, and 2.10.

2.2.8. Observations

The method of Contracts provides an elegant mechanism for capturing the behavioral compositions of objects that cut across module boundaries. Its specification of the behavioral compositions is independent of classes that implement them. Instead, it provides for a separate conformance declaration, that binds the possible implementation with the specified behavior contract. This property provides a means of decou-

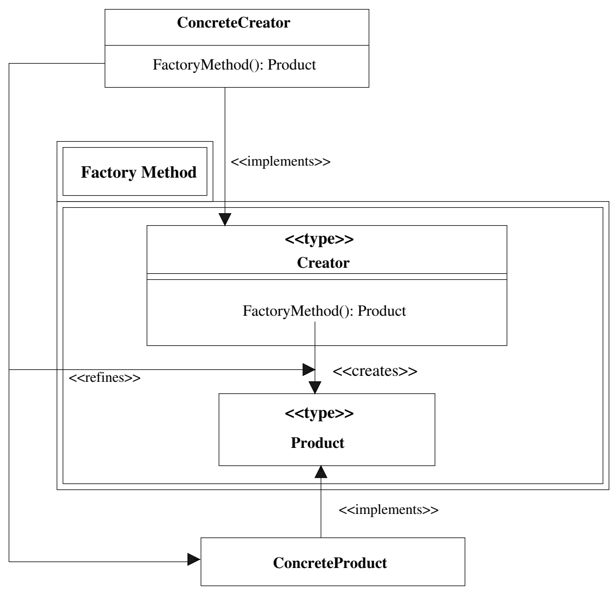

Figure 2.7. Factory Method - Class Model

pling the pattern specification and pattern implementations. Contracts also provides features, such as Contract refinement and Contract Inclusion, by which pre-existing contracts can be refined with modifications or extended without modifications. Although the specification of the two patterns taken for the study did not warrant the use of these features, they are crucial from the perspective of scalability, as they make it easier to specify large and complex patterns. The concept of invariants is a useful feature that help specify the characteristic features of the pattern that need to be satisfied all throughout the lifetime of pattern execution.

LePUS provides a strong mathematical foundation, and has the power to express relationships among sets of entities. The language is concise and harnesses the expressive power of predicate calculus. Its mathematical representation and precise specification of relationships between entities makes it a suitable language to com-

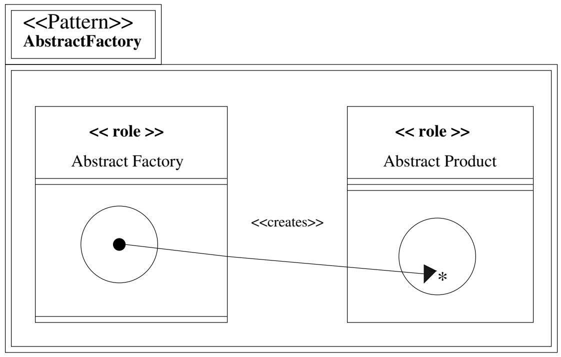

Figure 2.8. Abstract Factory - Role Model

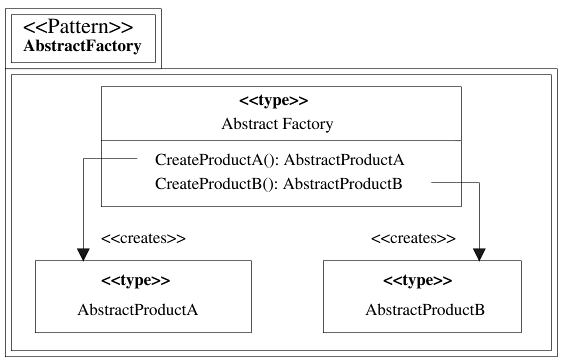

Figure 2.9. Abstract Factory - Type Model

pare two different patterns for duplication, refinement, etc. Its relational constructs inherently specify mathematical properties like exclusivity, one-to-one relationships, reflexivity and commutativity. Since each pattern is expressed as a simple mathematical formula, they can be directly provided tool support using prolog. It also has a complementing visual notation. The relationship between Factory Method and Abstract Factory patterns is more illuminated in their LePUS specifications (both visual and formula based). LePUS formulae encapsulate low-level details and yet capture all the needed subtle constraints and features of the pattern specified.

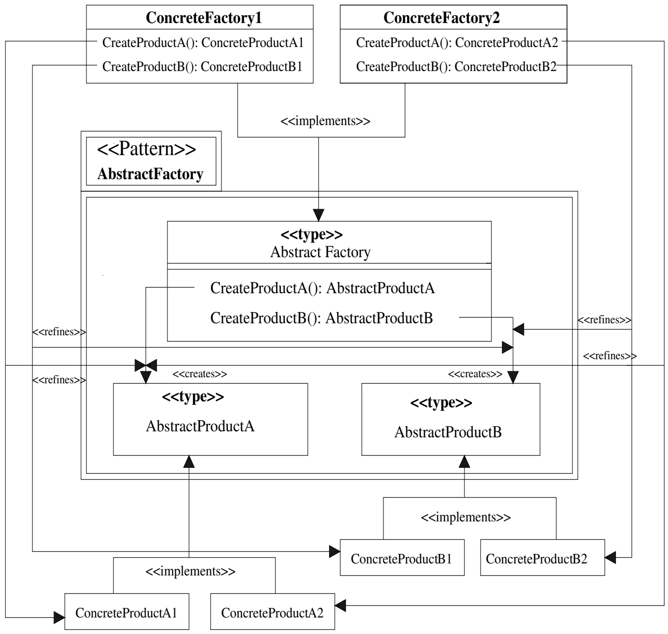

Figure 2.10. Abstract Factory - Class Model

DisCo offers means of specifying temporal behavior of patterns which may be useful in specifying behavioral aspects of patterns. DisCo allows formalizing instantiations and combining specifications of two different patterns in complex systems. It elegantly specifies the aspects of functions that are refined in inherited subclasses. However, it lacks constructs that explicitly specify mathematical relationships like reflexivity, commutativity, etc., among participants. It is also not suited for pattern repository management as it provides no means of resolving ambiguities and conflicts regarding duplication and refinement. It lacks a complementing precise visual representation.

Pure visual specification of design patterns is based on the belief that constraint diagrams, when used together with UML, in an intelligent 3-tier representation scheme can be used for unambiguous specification of design patterns. While a multi-level representation of patterns helps in representing and analyzing the pattern at different levels of abstractions, this scheme lacks the means of expressing complex and subtle mathematical relationships between entities, other than a few set-theoretic relationships. Moreover, it is harder to provide a tool support for patterns with only diagrammatic representation, unless the diagrams can be represented by means of formal syntax. For the sake of brevity, this scheme of pure visual representation using UML and constraint diagrams will hereafter be referred to as *Visual*.

All the merits and demerits of the various approaches, observed as a result of the experimentation and evaluation, are given in a tabular form in Table 2.1.

Table 2.1
Evaluation of Formal Pattern Specification Approaches

	Contracts	LePUS	DisCo	Visual
Mathematical Foundation	Fair (Invariant based and Extends First Order Logic)	Very Good Strong Mathematical Model (HOML + predicate calculus)	Fair Temporal Logic of Actions[37]	Fair Set theoretic notation & constraint diagrams
Precise Visual Notation	Nil	Exists	Nil	Exists
Conciseness	Fair	Very good	Fair	Fair
Specification of Structural Aspects	Fair	Very good	Fair	Good
Specification of Behavioral Aspects	Very Good	Fair	Very good	Very good
Specification of Constraints	Good	Good	Fair	Poor
Scalability	Very good	Good	Very good	Fair
Complementing Object Notations	Nil	Nil	Nil	UML
Support for OOP	Good	Good	Good	Very good
Ease of Use	Fair	Good	Fair	Good
Multi-level representation	Nil	Nil	Nil	Exists
Representation of low-level details.	Good	Fair	Good	Very good
Support for pattern-repository Management	Poor	Good	Poor	Poor

The evaluation Table 2.1 provides information on what attributes of the specification impart which character to the specifications; a few prominent ones are captured

Table 2.2
Inferences of Evaluation

Core Features	Resultant Behavior
Provision for reuse of specifications	Scalability
Visual Representation	Ease of Use
Complementing Object Notations	Ease of Use, learning and tool support
Rigorous Mathematical Model	Potential for pattern-repository Management
Multi-layer Approach	Ability to specify designs at various levels of abstractions

as cause-effect relationships in Table 2.2. Moreover, when analyzed with respect to the concept of pattern repository management, all these methodologies were found inadequate. What is needed is a more comprehensive specification capable of specifying all the information pertaining to patterns including the design problems and constraints they are required in. Hence, a specification language capable of covering the entire gamut of design patterns, and not just the structure aspect of it is needed.

2.3. Conclusions

It is evident from the evaluation, that out of the existing approaches, none is comprehensive enough for formal specification of software design patterns. However, it can be observed that LePUS has many merits that makes it an ideal starting place, which can be enhanced to be a comprehensive language for specification of software design patterns. However, any extension or enhancement of LePUS should address the following issues.

1. Specification of behavioral aspects needs to be strengthened by adding more relations for specifying dynamic properties.

2. Specification of constraints and low-level system details needs to be provided by means of modifications and additions of system or environmental variables and relations capturing implementation level details.

3. Support for object-oriented programming needs to be enhanced by providing representation for specifying objects with the first class citizen status and providing constructs for specifying object relationships.

4. There is a need to formally specify other aspects of patterns, such as, Intent, Applicability or Collaboration, etc. This will also make automated mining of design patterns for a given design problem possible.

5. Provide for a BNF grammar for the extended and enhanced language, so as to facilitate tool supports.

The following chapters present an enhanced and extended version of LePUS, called ELePUS, which addresses these limitations of LePUS. A more detailed overview of LePUS is also presented in the next chapter.

CHAPTER 3. ELEPUS - SETTING THE BACKGROUND

This chapter lays the ground work for ELePUS and (a) illustrates the need to specify aspects of patterns such as Intent, Applicability and Collaborations apart from structure, (b) provides a formal description of the language features of LePUS that are inherited by ELePUS, and (c) provides the overall structure of ELePUS and indicates the areas of LePUS that are modified and new features that are added.

3.1. Why Extend LePUS?

From the evaluation of various formal approaches presented in section 2.3, it can be inferred that an ideal language for tapping the full benefits of pattern formalism can be developed in two ways, either by extending and enhancing LePUS to address its limitations, or by developing a new language from scratch. The former alternative was chosen because LePUS is a proven language and has a strong mathematical basis. Moreover, the limitations of LePUS can be addressed by adding new language constructs and relations to it. The following section discusses the necessity for formalizing aspects of patterns other than the structure - such as Intent, Applicability and Collaborations.

3.1.1. Need to Formalize Pattern Aspects Other Than Structure

Formalizing only the structural part of patterns is incomplete, as it captures only one aspect of the pattern. Hence, if such a specification is used for a tool-based support, it will be limited to only the structural aspects of patterns; applications, like automated matching of pattern intents with applicable design scenarios, will not be possible. Moreover, to compare any two patterns for a better understanding and to resolve issues such as refinement, duplication, etc., the structural specification alone is not sufficient. An analysis of all the patterns in [3], reveals that there are patterns that share similar structures but their real differences are found in their intent (or applicability or collaboration). The following text is reproduced verbatim from [3], pp 219 - 220. "... The Adapter (139) and Bridge (151) patterns have some common attributes. Both promote flexibility by providing a level of indirection to another object. Both involve forwarding requests to this object from an interface other than its own.... The key difference between these patterns lies in their intents". Also, Composite and Decorator patterns differ mainly in their intents. "....Composite (163) and Decorator (175) have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects. This commonality might tempt you to think of a decorator object as a degenerate composite, but that misses the point of the Decorator pattern. The similarity ends at recursive composition, again because of differing intents" [3].

Moreover, the main rationale behind the concept of patterns is to provide encapsulated design ideas to object-oriented system architects. This purpose will be defeated if specifications of patterns do not facilitate an easy and correct understanding of patterns. Hence, the formalization of intent, applicability and collaboration are also required. ELePUS specification of a pattern involves specification of Intent, Applicability, Structure and Collaboration aspects of the pattern using ELePUS, thereby capturing all the aspects of a software design pattern.

3.2. Overview of ELePUS and Structure of LePUS

3.2.1. LePUS - The Language Structure and Features

An abridged version of the mathematical model of LePUS is reproduced in this section from [17] and [22]. This will serve as the background for the definition of constructs for ELePUS.

In LePUS, patterns are transcribed to formulae while a program is assumed to be represented as a *model*. LePUS formulae manifest design patterns in the form of logic statements. Given the transcription of a pattern π to a LePUS formula Ψ, we can say that a program p **conforms** to π iff the *model* of p satisfies Ψ. In this section, the primary abstractions of LePUS are presented.

3.2.1.1. Model

A model M is a pair **<P,R>** where P is a collection of **ground entities**, each of which is either a function or a class, and R $=R_1, \ldots R_n$ is the set of relations amongst them.

3.2.1.2. Dimension

The *dimension* of a variable is defined inductively:

- Ground variables have a dimension '0'.
- A set whose entities are of the same type and dimension d is an entity of dimension d+1.

The domain of classes (functions) of dimension 1 is denoted as 2^C (2^F), of dimension 2 as 2^{2^C} (2^{2^F}), and so forth.

3.2.1.3. Formulae

A *formula*, in LePUS, consists of the following building blocks:

1. **Variables:**
   - *ground* variables, ranging over ground entities.

- *higher dimension* variables, ranging over higher dimension (i.e., sets of) entities. E.g., C^d and F^d vary over sets of classes and functions of dimension d, respectively. Lowercase letters, c or f denote C^0 or F^0, and C or F are shorthand for C^1 or F^1.
- *hierarchy* variables, ranging over *inheritance class hierarchies*

2. **Relation** Symbols:

- *ground* relations, corresponding to those in **R**, including *transitive* relations.
- *generalized* relations, which derive systematically from the *ground* relations.
- *commuting* relations.

A *formula* in LePUS has the form:

$$\exists\ (x_1, \ldots x_n) : \bigwedge_i P_i\ ,$$

where P_i are predicates of the form $\Re_i(y_{i_1}, \ldots y_{in_i})$, $\Re_i$ are relation symbols, $y_{i_1} \ldots y_{in_i}$ are arguments or parameters of the relation symbols $\Re_i$, and $x_1, \ldots x_n$ are all the variables in P_i.

3.2.1.4. Ground Relations

LePUS provides a small, and potentially extensible, set of ground relations, that are defined between ground entities. Ground relations capture the structure and behavior embodied in most design patterns. Relations like "*c is the first argument of f*" or "*f is defined in c*" are examples of ground relations. The only restriction for adding a new ground relation is that it must have a canonical(simple), straightforward implementation in more than one Object-oriented programming languages. This is to simplify the mapping of any given setting of relations to a set of "conforming" programs. A transitive ground relation β is represented as β^{+}. The following are examples of ground relations.

- *Abstract*(c) is a ground relation, and is satisfied by a ground class of **P**, if it is abstract.
- The possible indirect inheritance relation between *concrete-class* and *abstract-class* transcribes to $Inheritance^{+}(Concrete - class, abstract - class)$.

3.2.1.5. Generalized Relations

All relations in LePUS derive systematically from the ground relations in a finite number of steps. Below the indication of every *generalized relation*, when applied to a variable of dimension 1, is presented. However, each definition extends naturally to higher dimensions. Let α represent a ground unary relation, and β a ground binary relation. Let w, v, represent *ground* variables, and V, W represent 1-dimensional variables. The following *generalized relations* are admitted:

unary $\quad \alpha(V) \overset{\text{def}}{=} \forall\, v \in V : \alpha(v)$

total $\quad \beta^{\rightarrow}(V, W) \overset{\text{def}}{=} \forall\, v \in V\ \exists\, w! \in W : \beta(v, w)$

regular $\quad \beta^{\leftrightarrow}(V, W) \overset{\text{def}}{=} \forall\ \mathrm{v} \in V\ \exists\, w! \in W : \beta(v, w) \wedge \forall\ \mathrm{w} \in W\ \exists\, v! \in V : \beta(v, w)$

(β is an invertible function (1:1 and onto) from V to W)

A total relation may be applied to variables of different dimensions with the following interpretation.

$$\beta^{\rightarrow}(V, w) \overset{def}{=} \forall\, v \in V : \beta(v, w)$$

$$\beta^{\rightarrow}(v, W) \overset{def}{=} \exists\, w \in W : \beta(v, w)$$

For instance, the *Factory Method* pattern requires that every function of the set *Factory-Methods* creates exactly one class of *Products*, and that every *Products'* class is created by exactly one *Factory-Methods'* function. This is specified as a regular *Creation* relation between the sets.

$\exists FactoryMethods \in 2^F,\ Products \in 2^C$:

$Creation^{\leftrightarrow}(Factory - Methods, Products)$

Restrictions on relations' domains are specified through *exclusive* relations:

For a generalized relation $\Re$, if:

$$\Re(V!, W) \overset{def}{=} \Re(V, W) \wedge [\ \forall\ v\ : \Re(v, w) \wedge (w \in W) \rightarrow (v \in V)\ ]$$

we say that $\Re$ is **exclusive** to V with respect to W. The dual case is defined for $\Re(V, W!)$, and $\Re(V!, W!)$ is equivalent to $\Re(V!, W) \wedge \Re(V, W!)$.

3.2.1.6. Commutativity

Commutativity relations are used to specify that a chosen list of *regular* relations *commute*. Given the variables V, W, and the binary regular relations $\beta_1^{\leftrightarrow}$, $\beta_2^{\leftrightarrow}$ (each may be a composition of regular relations), the *Commute* relation is defined as follows:

$Commute_{\beta_1,\beta_2}(V, W)$ indicates that β_1 and β_2 are *equal* in V,W.

Commute may apply to more than two relation symbols and more than two variables. Representing *regular relations* as edges and variables as vertices, the above definition for Commute is generalized by the condition that for every pair of vertices V and W, all the paths in the graph from V to W are equal, in the sense that each path in the graph determines a (composition of) relation and these relations are equal in V and W.

3.2.1.7. Well Formed Formula

A formula

$$\exists\ (x_1, \ldots x_n) :\ \bigwedge_i\ P_i\ ,$$

is **well formed** iff all predicates P_i are well formed. A *well formed predicate* combines a relation symbol ρ (either *ground, generalized,* or the *commute* relation) declared on variables (arguments) of number, type and dimension as allowed by ρ.

3.2.1.8. Auxiliary Abstractions

LePUS provides mechanisms for introducing additional abstractions, as long as they are defined by means of the constructs defined above. The following are some of such additional abstractions that are crucial in describing patterns.

clan F^d is a **clan** in C^d iff F^d consists of functions of identical signature, each of which is defined in a different class of C^d. Formally:

$$\begin{aligned} Clan(F^d, C^d) \ &\overset{def}{=}\ Defined-In^{\leftrightarrow}(F^d, C^d) \wedge \\ &\quad [\forall\, F_1^{d-1}, F_2^{d-1} \in F^d :\ Same-Signature^{\rightarrow}(F_1^{d-1}, F_2^{d-1})] \end{aligned}$$

tribe F^{d+1} is a **tribe** in C^d iff every function $F^d \in F^{d+1}$ is a *clan* in C^d.

The elements of a *tribe* do not necessarily have the same signature as elements of a *clan* do. In the case where F and C are of dimension 0, *clan(f,c)* simply means that f is defined in c. In the general case, F^d is a clan in C^d iff the union of classes in F^d is a clan in C^d.

For instance, *factory-method* of the *Factory Method* pattern is manifested as a clan in Creators set, in the LePUS formula of *Factory Method* pattern as can be seen in Figure 2.3. Each *Creators'* function of the *Abstract-Factory* pattern (Figure 2.4) produces objects of a different class hierarchy of *Products*, thereby forming a *clan* per

Products hierarchy. The set of all *Creators* clans give rise to a *tribe* in *Factories*, and is stated as *Tribe(Creators,Factories)*.

3.2.1.9. Hierarchies

A hierarchy is a set of ground classes that constitutes a ground class h_r^0 and a set of remaining elements h_N^1 that conforms to the following conditions:

- $Inheritance^{+\rightarrow}(h_N^1, h_r^0)$
- $Abstract(h_r^0)$

The set of hierarchy entities is denoted H and H $\subset 2^C$. The model $M =< P, R >$ is extended to $< P, R, H >$ to accommodate hierarchy variables along with other ground variables. Ground hierarchy variables which range over the domain **H** are represented like $h, h_1, \ldots,$ and their dimension is defined as 1. Hierarchy variables H^d, of dimension d range over hierarchies of dimension d.

3.2.1.10. H-total Generalization

Hierarchy variables are not allowed in *total* relations but only in H-total relations, which are variations of *total* relations. Intuitively, in a *H-total* relation, a ground hierarchy variable stands for the hierarchy's *root* if positioned as the argument for the $\exists$ quantifier in the *total* counterpart, but is interpreted as the hierarchy's set of *nodes* whenever it acts as the argument for the $\forall$ quantifier.

The *H-total generalization* of a binary ground relation, β, is a relation $\beta^{\rightarrow H}$ with the following induction, depending on its arguments:

$$\begin{array}{lll} \beta^{\rightarrow H}(V, h) & \text{means} & \beta^{\rightarrow}(V, root(h)) \\ \beta^{\rightarrow H}(h, V) & \text{means} & \beta^{\rightarrow}(Nodes(h), V) \\ \beta^{\rightarrow H}(h_1, h_2) & \text{means} & \beta^{\rightarrow}(Nodes(h_1, root(h_2)) \end{array}$$

A *total* relation applied to variables of different dimensions is defined as follows:

$$\beta^{\rightarrow}(V, w) \stackrel{\text{def}}{=} \forall\, v \in V : \beta(v, w)$$

$$\beta^{\rightarrow}(v, W) \stackrel{\text{def}}{=} \exists\, w \in W : \beta(v, w)$$

A H-total relation combining hierarchy variables with single classes is defined as follows:

$$\beta^{\rightarrow H}(h, w) \stackrel{\text{def}}{=} \beta^{\rightarrow}(Nodes(h), w)$$

$$\beta^{\rightarrow H}(v, h) \stackrel{\text{def}}{=} \beta^{\rightarrow}(v, root(h))$$

3.2.2. Other LePUS Constructs

Provided below are intuitive interpretations to common relation symbols used in LePUS.

Assignment(f,c_1, c_2) designates that the reference from c_1 to c_2 is assigned a value within the flat structure of the body of function f.

Creation(f,c) designates a creation of an object of class c within the flat structure of f ("f creates c").

Invocation(f_1, f_2) designates a function call to f_2 within the flat structure of f_1 ("f_1 calls (or invokes) f_2).

Inheritance(c_1, c_2) designates that c_1 "inherits" from c_2. The definition of inheritance is as in [3], i.e., delivering *substitutability* and allowing dynamic dispatch of overridden procedures ("virtual functions").

Forwarding(f_1, f_2) designates a function call to f_2 within the flat structure of f_1, using the formal arguments of f_1 as actual arguments in the same order, and that the signatures of the two functions are identical. It can be safely assumed that *Forwarding(f_1, f_2)* implies *Invocation(f_1, f_2)*.

Production(f, c) indicates that an object of class c is created within the flat structure of f and is returned as the result of f. Generally, *Production(f,c)* implies *Creation(f,c)*.

Reference-To-One/Many designate both compositional and associative relations between classes. Thus, class attributes (also "data members" or "instance variables") are not distinguished from other associations.

3.2.3. Enhancements in ELePUS

The following are the limitations of LePUS, that are addressed in ELePUS, in order to extend LePUS to specify other parts of pattern specifications as well.

- Objects are not considered as ground variables in LePUS. Since design pattern by definition, involve "descriptions of communicating objects and classes" [3], an explicit mechanism needs to be provided to accommodate objects as first class citizens.

- Specification of dynamic properties and properties involving explicit specification of dynamic binding and delegation of responsibilities are not provided in LePUS. However, they are required for specifying aspects of patterns such as intent or applicability.

- Specification of other Object-oriented properties, such as access regulations, encapsulations, and other such factors, need to be added to LePUS constructs.

- To specify collaborations, additional functionality for specifying temporal behavior of patterns and object dynamics, including the notion of cardinality of objects, is required.

3.3. Overview of ELePUS

ELePUS is an enhanced and extended version of LePUS. New ELePUS constructs are defined without altering the mathematical model of LePUS. The major extensions and modifications of LePUS that are incorporated in ELePUS are broadly classified into three categories.

- **Amendments to basic abstractions:** These involve addition of new ground variables and modifications of syntax of existing constructs.

- **Addition of new constructs:** A number of new relations - both ground and general relations, are added to ELePUS. These additions make ELePUS better equipped to handle specifications involving access regulations, types of bindings and temporal properties of relations.

- **Modifications to representation of patterns:** In LePUS, the specification of a pattern involves only representing the structure part of the specification using both formula based mechanism and visual representation. However, for a complete representation of patterns, ELePUS extends this specification to specify intent, applicability, and collaborations as well.

3.3.1. ELePUS Format for Specifying Patterns

Each aspect of patterns, intent, applicability, structure, and collaborations, has its own characteristic features and participants. Hence, ELePUS provides for specifying each aspect and its participants independent of others. This also facilitates an aspect-wise comparisons of two patterns. For example, in specifications of the applicability aspect, the participants may be fewer in number and their definitions may not be as rigid as in the structure aspect. This difference is because the structure specification details the well-formed structure of the pattern, while the applicability aspect specifies possible design scenarios, where all details of participants may not be specified.

Each aspect of the pattern write-up is represented as an ELePUS *formula*, which is similar to the LePUS *formula* except it contains an enriched set of ground variables and vocabulary terms that are available in ELePUS. Moreover, to facilitate a tool support for patterns, using ELePUS specifications, a *BNF* grammar for ELePUS is also provided. The following section gives the generic structure of ELePUS specifications.

3.3.1.1. Structure of ELePUS Specifications

In the descriptions of ELePUS structure given below, statements following the # symbol are comments and are typeset in a different font for visibility.

Intent

$$\exists\ (x_1, \ldots x_n) : \bigwedge_i P_i \ ,$$

#P_i *are predicates of the form* $\Re(y_{i_1}, \ldots y_{i_{n_i}})$, $\Re_i$ *are ELePUS relations, and* $x_1, \ldots x_n$ *are the free variables in* P_i.

Applicability

$$\exists\ (x_1, \ldots x_n) : \bigwedge_i P_i \ ,$$

#P_i *are predicates of the form* $\Re(y_{i_1}, \ldots y_{i_{n_i}})$, $\Re_i$ *are ELePUS relations, and* $x_1, \ldots x_n$ *are the free variables in* P_i. *It can be seen that intent and applicability are specified in a similar manner. However, since their foci are different, the variables and their relationships may be different in both the aspects.*

Structure and Collaborations

$$\exists\ (x_1, \ldots x_n) :$$

$$structure \bigwedge collaboration$$

where *structure* is defined as

$$\bigwedge_i P_i \ ,$$

and *collaboration* is defined as

$$\bigwedge_j P_j$$

P_i are predicates of the form $\Re(y_{i_1}, \ldots y_{i_{n_i}})$, $\Re_i$ are ELePUS relations, and $x_1, \ldots x_n$ are all the free variables in P_i. However, it should be understood that Cardinality (P_j) $\leq$ Cardinality (P_i), where the Cardinality operation refers to the number of free variables of either P_j or P_i, whichever is its argument. This property ensures that all participants in collaborations are defined in the structural specification of the pattern.

Other aspects of pattern write-ups like *Consequences, Implementation, Sample Code, Known Uses and Related Patterns* are not specified using ELePUS because they are too descriptive in nature and cannot be specified using a concise set of formal relations. In fact, this research did not attempt to formalize the other aspects, Consequences, Implementations, Sample Code, and Related Patterns, because, either they require verbal descriptions (known uses and related patterns) or are programming language specific (sample code, implementation and consequences).

The following chapter provides a comprehensive account of new additions and modifications in ELePUS, their syntax, semantics and illustrations depicting their usage. Chapter 5 validates ELePUS with respect to the GoF patterns [3] and the evaluation criteria discussed in Chapter 2.

CHAPTER 4. LANGUAGE FEATURES OF ELEPUS

The finest language is mostly made up of simple unimposing words.
- George Eliot (1819-1880)

New features and relations introduced in ELePUS (that are not available in LePUS) are listed in this chapter. Each such addition is accompanied by its syntax and associated semantics. Each definition is accompanied by an illustration of one or more examples, where that construct/feature of ELePUS is needed in specifying the pattern. The new features are formulated and applied for patterns presented in the design patterns catalog [3]. The modifications and new additions in ELePUS were identified by analyzing and observing repeating behaviors or properties of design patterns. ELePUS was developed in a recursive manner with applying existing and proposed constructs to a large set of patterns from [3] and improvising based on the feedback and experiences gained.

A BNF grammar detailing in a formal fashion the syntax of the ELePUS constructs is presented at the end of this chapter. Some of the examples presented may use relations that are defined later in this chapter. For reference purposes, informal definitions for all ELePUS relations/constructs are provided in Appendix A.

4.1. ELePUS Variables

4.1.1. Object Variables

Design patterns are *"Descriptions of communicating* **objects and classes** *that are customized to solve a general design problem in a particular context"* [3]. Object compositions and relationships play a major role in most descriptions of patterns. Hence, objects were incorporated as ground-variables or first class citizens along with classes and functions (in LePUS). The dimension rules apply to objects in the same way they apply to classes. They are represented in ELePUS formulae by the symbol O^d, where d is the dimension of the set of objects. Objects are related to Classes by the total relation $Instantiation^{\rightarrow}(Object, Class)$, where Object ∈ O; Class ∈ C. *Instantiation (class) and Class (object)* are unary relations that refer to the role a particular variable takes in a particular relation, and they are explained in later sections. A provision for object variables facilitates the representation of the state information and object dynamics.

<u>Example - Objects in Singleton Pattern</u>

In the Singleton pattern, constraints are placed on the number of Singleton objects. Since cardinality is an issue that can be only described on objects, a Singleton Object is declared as a participant and the constraint is specified in the Intent section of this pattern.

Intent specification of Singleton:-

“Ensure a class only has one instance, and provide a global point of access to it” [3].

$\exists\ aSingleton\ \in O;\ Singleton\ \in C\ :$

$Instantiation^{\rightarrow}(aSingleton, Singleton) \wedge$

$Public\text{-}Access(aSingleton) \wedge$

$Cardinality(aSingleton, 1)$

4.1.2. Environment Variables

Environment variables represent the domain of influence of pattern participants. Intuitively, this might represent the pattern as a whole, the program or a subsystem component that contains the pattern. Environment variables are represented as Env. One constraint on the Environment variables is that they cannot have the dimension relation defined on them, since there is no notion of sets of environments for design patterns. Environmental variables are required to specify the relationship between a participant and the system or the constraints the system places on the participants.

Example - Environment variables in Prototype

The Prototype pattern places the constraint on the system in which the pattern is defined that it should be independent of the creation process of products. This constraint is captured using the Decouple relation (to be discussed later) and an environmental variable depicting the system. For ease of readability, environmental variables are specified in capital letters.

Specification of Applicability section of Prototype

"Use the prototype when a system should be independent of how its products are created, composed and represented and...to avoid building a class hierarchy of factories that parallels the class hierarchy of products ..." [3].

$\exists\ Products \in 2^{H},\ SYSTEM \in Env\ :$

$Decouple(SYSTEM, Creation(Products)) \wedge$

$NOT(Equivalence(Cardinality(Factory), Cardinality(Products)))$

The definition of syntax and semantics of all relations and constructs in ELePUS are specified with respect to all of the variable types possible, namely, Classes, Functions, Objects and Environmental variables.

4.2. Attribute Qualifiers

Attribute Qualifiers are generally unary relations that are defined on a ELePUS variable. These relations are *qualifiers* because their definitions on a variable map/qualify the variable to a well defined set of elements. For example, *Abstract (Class), Class* $\in C$, is an attribute qualifier, and it qualifies the Class variable into a set whose members are abstract classes. In other words, attribute qualifiers define certain behavior or impart a characteristic to a variable. The attribute qualifiers added to ELePUS are given below along with a brief note on the need for them and an example usage.

4.2.1. Predicates for Object-Orientation

4.2.1.1. Abstraction

An abstract relation is defined on a class variable of any dimension d. However, an abstract relation cannot be declared on a Hierarchy of variables as a whole, since all the nodes of the hierarchy cannot be abstract. The Abstract relation refers to the concept of abstraction in object-oriented (OO) parlance. Practical examples of an abstract relation in popular OO programming languages are 'abstract classes' in C++ and 'interface classes' in Java. Hence, this is a ground-relation as per LePUS guidelines.

Abstract Relation in Singleton

A Singleton class in the Singleton pattern should be subclassable. This is ensured in the specification of the pattern by defining the Abstract relation on the Singleton class.

Applicability section of Singleton:-

$\exists SingletonObject \in O,\ Singleton \in C$:

$Cardinality(Instantiation(Singleton), 1) \wedge$

$Abstract(Singleton)$

4.2.1.2. Encapsulation

Encapsulation is similar to the abstract relation and is defined on class variables. It refers to access regulations and hiding of internal details of the class from the outside environment. Generally, if there is a constraint that some or all of internals of a class or sub-system (set of classes) should not be visible from outside, it is specified using the Encapsulation relation. Although an encapsulation relation can be constructed from access regulation predicates like *Public-Access* and *Restricted-Access* (to be defined later), due to the number of instances where this predicate is used, it has been provided as a generalized relation and not as a syntactic-sugar derivable from other ground/generalized relations.

Encapsulation Relation in Strategy

The Strategy pattern has a requirement that the algorithm or internals of strategy should be separated from the outside. This feature is captured by the Encapsulation predicate on the Strategy variable. Only the Encapsulation relation from the Intent specification is provided here, since the other relations used are yet to be defined.

Intent section of Strategy:

$\exists Strategy \in H$:

$Encapsulation(Strategy) \land \cdots$

4.2.1.3. The Abstraction of Algorithm

Algorithm is a generic concept; intuitively it stands for a sequence definition of statements and function invocations. An Algorithm can be defined for a function or a class variable. The ELePUS relation *Algorithm (v)*, $v \in C^d$ or $v \in F^d$ is a syntactic-sugar for all the relations that collectively represent the function invocations made in a function and their sequence. The Algorithm relation defined on a class variable is interpreted to mean the collective Algorithm relation defined on its functions.

Example: Algorithm Relation in Intent of Builder

The Builder pattern needs to decouple the process of construction of a complex object from its components so that the same process can create different representations. This is captured by the Algorithm relation in the Intent specification of Builder.

$\exists\ Products \in 2^C,\ Builder \in H$:

$Creation(Builder, Products) \land$

$Decouple(Algorithm(Builder), Products)$

4.2.2. Access Specifiers

The specification of accessibility settings for variables is a crucial aspect of Object-oriented software specification. Hence, for that purpose, ELePUS incorporates two access specifiers, *Public-Access* and *Restricted-Access*.

4.2.2.1. Public-Access

Definition of *Public-Access* is trivial for functions with dimension 0. Since unary relations in ELePUS are defined recursively for elements with higher dimension, this relation can be applied to function variables of any dimension. However, for class variables, it should be applied recursively to all the functions f that have a *Defined-In* relation with the class. Public-Access relation defined on a object variable is interpreted to be defined for class (object). The syntax for this relation is:

Public-Access (V), where V $\in F^d$ OR

Public-Access (W), where W $\in C^0$:-

Public-Access (V), $V \in F^1$ $\wedge \forall$ v $\in$ V, *Defined-In (v, W)*

Since this predicate has parallels in C++ and Java (public declaration), this is a ground relation.

Example of Public-Access in Singleton

In the Singleton pattern, there is a requirement to provide global access to the singleton object. This is specified in the ELePUS specification of Intent section of Singleton as:

$\exists\ aSingleton \in O$:

$Public\text{-}Access(aSingleton) \wedge \cdots$

The $\cdots$ symbol denotes that some other relations are omitted here, since they are not yet defined.

4.2.2.2. Restricted-Access

A *Restricted-Access* of a function (or class) means that only a select few other classes or functions can access this function (or class). This is a ground relation and is similar to the protected or private access regulators in C++ or Java programming languages. Unlike the *Public-Access* relation, *Restricted-Access* is a unary relation defined on a

class or a function. Intuitively, Restricted-Access of a function implies private access and Restricted-Access of a class implies a protected access regulator as per C++ or Java terminology. The syntax of this relation is:
Restricted-Access (V), where V is a class or function variable of dimension 0.

4.2.3. Additional Relations for the Structural Aspects of Patterns

Two commonly occurring structural constructs in design patterns observed were lists and trees. The following two unary relations qualify a set of objects of a class as either a list or a tree.

4.2.3.1. List

The List relation is applicable to a class or an object variable (of dimension one). Once a class (set of objects of the class) has been declared as a List, the ELePUS relation, *Sequential-Access* can be applied to it. The List relation can be equated with Vector constructs in Java and ANSI C++ libraries, and thus qualifies as a ground relation. The List relation, however does not provide any other detail about list implementation such as array, linked list, etc.

Example of List relation in Iterator

The Iterator pattern provides a way to access elements of an aggregate class sequentially. The concept of an aggregate class, which is a conceptual and physical list is represented using the List predicate as:

$\exists\, AggregateObject \in C$:

$\quad List(AggregateObject) \wedge \cdots$

The relations indicated by $\cdots$ are not yet formally defined. Hence they are omitted from this specification for ease-of-reading.

4.2.3.2. Tree

Trees are another data structure that are very commonly used in most software and form an inherent part of the design. Just like the List relation, the predicate Tree, is a unary ground relation defined on a class variable (or the set of object instantiations of the class). Just as Hierarchy variables have members as Nodes of hierarchies and one member as root of the hierarchy, Tree has *Root, Node, Child* and *Leaf* qualifiers to distinguish its members. Tree captures a set of classes of dimension 1, where one class act as the root, and all other classes hold a transitive *Child* relation with the root. *Leaf* qualifiers are applicable only to those elements to which no other element processes a *Child* relationship. *Node* is a generic relation used to represent either a root or any element e, $Child^{+}(e, root)$. Child relation is similar to Inheritance relation in Hierarchy variables. The Root of a Tree cannot hold Child relation to any other element in the tree.

Definition of Tree

A set of ground classes or objects t is a **Tree** iff there exist a (unique) ground class(or object) ${t_r}^0 \in$ t, such that:

- $Child^{+\rightarrow}({t_N}^1, {t_r}^0)$

- $Root({t_r}^0)$

where ${t_N}^1$ is the set of remaining elements in t, namely $t - \{{t_r}^0\}$. Any $t_i \in t$ is a Node(t).

Example of Tree Relation in Composite

In a composite pattern, there is a need to compose objects into tree-structures, which is captured by the Tree relation as shown below:

$\exists$ $Objects \in 2^{O}$:

$Tree(Objects)$

4.3. Predicates for Temporal Relationships

Temporal relationships refer to behavior of the system with respect to a time-line. This property is crucial in specifying collaborations among pattern constituents. Many ELePUS relations can be extended to include time-stamping. For example, the relation $Invocation(f_1, f_2)$ can be extended to a time-stamped version by adding the time instance of its occurrence; like $Invocation(f_1, f_2)_{t_i}$, where t_i refers to the time instance at which the invocation relation is realized. However, it can be empirically verified that such large-scale modifications are not needed for a complete specification of patterns and that they add to unnecessary confusion and difficulty in understanding. Hence, ELePUS provides only two predicates for which temporal behavior is indispensable.

4.3.1. State-Change Relation

State-Change of an object variable refers to any change of state of the object variable during a specified time interval. This predicate becomes true if there is a change of state during the specified interval. Such relations are required to trigger actions based on any state change. This will be especially useful in guaranteeing the Invariant properties like data-consistency between the object variable and its other copies or clones. The State-Change relation is defined only on a object variable as *State-Change (Obj)*, where Obj $\in O^d$.

4.3.1.1. Example of State-Change in Observer

Observer represents a scenario, where there is a need to maintain data consistency between many observers and a subject (an object variable). Whenever the state of Subject changes, a update operation has to be triggered to update other observers so that they maintain data consistency with subject. A State-Change relation is very effectively used in this scenario, as given below: $\exists Subject \in C$, $Observers \in 2^C$, $Update \in F$:

$Trigger(State\text{-}Change(Subject), Update)$

4.3.2. Specifying State of Objects and Roll-back

Objects have an inherent attribute called *state* that differentiates one object from another. State refers to the values of the internal variables defined in the object. Since object dynamics involve specifying object states, ELePUS provides an unary ground relation called *State (object), object* $\in O^d$.

Roll-back of a transaction or an event refers to reverting back to a system and environment state that existed before the execution of that transaction or the event. Alternatively, this can also be specified as nullifying the effect of the transactions or events over a period of time. *Roll-back* is a binary relation defined for state of an object variable of any dimension, and the instant of time it should revert back to. That is:
Roll-back (State(α),t_i), $t_i \leq t_{current}$, where $t_{current}$ refers to the instant of time at the realization of the Roll-back relation.

4.3.2.1. Relationship between Roll-back and State-Change Relations

The State-Change relation indicates whether a change has occurred in the state of the class variables during a time interval. The Roll-back (α, t_m) relation will result in a State-Change (α, t_i, t_j), where t_i and t_j are time instances before the activation of the Roll-back relation and after its activation ended, if there has been an occurrence of a State-Change relation during the time interval t_m and t_i.

4.3.2.2. Example of Roll-back in Memento

The Memento pattern is applicable to situations which demand a roll-back of the state of a class variable back to a time instant in the past. ELePUS specification of Applicability section of design patterns captures this property as shown below:

$\exists Originator \in 2^C, :$
$Rollback(State(Originator), Time_i)$

4.4. Other Predicates and Properties

This section introduces new constructs and relations that are introduced in ELePUS, that are neither attribute qualifiers nor have temporal characteristics. Section 4.4.2 discusses specification of recursiveness of certain relations. Since the notion of object variables have been introduced in ELePUS, section 4.4.3 defines the behavior of LePUS constructs with respect to object variables.

4.4.1. New Relations with Illustrations

In the following descriptions, f, c, and o represent ground function, class and object variables and F, C, and O represent function, class and object variables of dimension one respectively.

4.4.1.1. Coupling Relations

In object-oriented design, the level of coupling between participants is a crucial design factor. Coupling refers to the degree of interconnectivity and interdependence between two entities. This property also manifests itself in design patterns. Constraints like "two participants need to have more coupling" or "two participants should be independent of each other", are found in many patterns such as Abstract Factory, Builder, Bridge, Chain-of-Responsibility, etc. ELePUS provides two generalized total relations for specifying the coupling between participants.

- The Couple Relation is defined between two class variables or elements of a set of class variables of dimension at-least one.

$$Couple^{\rightarrow}({C_1}^{d_1}, {C_2}^{d_2}) \overset{def}{=} \forall\, {C_1}^{d_1-1} \in {C_1}^{d_1},\ {C_2}^{d_2-1} \in {C_2}^{d_2} : \\ Couple^{\leftrightarrow}({C_1}^{d_1-1}, {C_2}^{d_2-1})$$

$$Couple^{\rightarrow}({C_1}^{0}, {C_2}^{0}) \overset{def}{=} Hold\text{-}Reference^{\leftrightarrow}(C_1, C_2)$$

In the above definition, $Hold\text{-}Reference(C_1, C_2)$ is a LePUS ground binary relation that indicates that the class C_1 holds the reference of an instantiation of class C_2. When the Couple relation is defined for only one class variable, it is taken to imply that the class variable takes part in the relation in the place of both the participants. When the Couple relation is defined for hierarchy variables, it applies to the set of classes that are Nodes of the Hierarchy.

Couple Relation in Abstract Factory

Abstract Factory pattern has the requirement that the nodes of a particular group of hierarchy variables should be used together and are closely coupled. This is specified in the ELePUS specification of applicability section of Abstract Factory as:

$\exists Product\text{-}Family \in 2^H,\ SYSTEM \in Env$:

$Couple(Nodes(Product\text{-}Family))$

- The Decouple Relation is specified as Decouple (V,M), where V and M are either class or environment variables or other ELePUS relations. Intuitively, this implies that the two participants of the relation are independent of each other and change in one participant does not affect the another. It should be noted that Decouple relation is not the same as NOT (Couple). This is because two participants can be independent of each other even if one or both of them vary. A sample application of Decouple relation in Abstract Factory is given below:

  $\exists Product\text{-}Family \in 2^H,\ SYSTEM \in Env$:

  $Decouple(SYSTEM, Creation(Product\text{-}Family))$

In this example, the SYSTEM environment is not affected by the mode of creation of Product-Family or which function or object created Product-Family.

4.4.1.2. Extracting and Storing State Information

Extract-State (obj), obj $\in$ o, indicates that the state information of the object *obj* can be computed from the state of other system variables, or it can be restored into the environment (for example, a disk storage) to be retrieved later.

Example 1: Extract-State in Flyweight

$\exists Objects \in O,\ APPLICATION \in Env$:

$Extract\text{-}State(Objects)$

Example 2: Extract-State in Memento

$\exists Originator \in 2^C$:

$Extract\text{-}State(Originator)$

4.4.1.3. Specification of Events Triggering

The *Trigger*($\Re$,*f*) relation is similar to the trigger construct in database query languages. Trigger relation is declared between another ELePUS relation and a function f. The function f will be invoked at the occurrence of the relation $\Re$. This construct is crucial for specifying the cause-effect relationships that are often found in design patterns. Also, this construct proves to be useful in specifying collaborations between the pattern participants.

Example: Trigger in Observer pattern to maintain data-consistency.

$\exists\, Subject \in C,\ Observers \in 2^C, Update \in F$:

$Clan(Update, Observers) \wedge$

$Trigger(State\text{-}Change(Subject), Update)$

4.4.1.4. Equivalence Relationships

The *Equivalence* relation acts as a comparator between numerically enumerable relationships or syntactically equitable entities like cardinality, interface or function signatures. An *Equivalence* relation takes, as its parameters, two participants of the same type - function variable, Cardinality Relations defined on objects or Interface Relations defined on classes. Equivalence of Interface implies an *Interface-Compatible* relation (to be defined later).

Example: Equivalence Relation in Iterator:

The Iterator pattern is applicable to all list-like aggregate objects where the interface of all elements in the list object are the same. The Equivalence relation is used here to satisfy interface equality.

$$\exists\, ListObject \in 2^{O},\ Object \in O\ :$$
$$List(ListObject, Object) \wedge$$
$$Equivalence(Interface(ListObject_i), Interface(ListObject_j)),$$
$$\forall ListObject_i, ListObject_j \in ListObject$$

4.4.1.5. Interface Compatibilities Among Classes

The *Interface-Compatible* relation is defined on two classes (interfaces of classes), c_1 and c_2 to indicate compatibility between their interfaces. However, it may not imply Equivalence of Interfaces of c_1 and c_2. The reason for the differences between these two structures is that, for two Interfaces to be equivalent, the number of functions in each interface, their names and their signatures should be identical, while in Interface-Compatible relation between two classes, it is not entirely necessary for the number of functions in each interface to be equal. The only requirement is that the function signatures of some of the functions in one class matches those of the other class, so that forwarding of requests can be achieved.

Example: Interface-Match in Adapter:

The Adapter pattern is applicable in situations where client classes are not compatible with legacy class interfaces, but we can provide for an adapter which maintains interface compatibility between both client classes and legacy classes.

Applicability of Adapter

$\exists Client,\ Adapter,\ Adaptee \in C\ :$

$NOT(Interface\text{-}Compatible(Client, Adaptee) \wedge$

$Interface\text{-}Compatible(Client, Adapter) \wedge$

$Interface\text{-}Compatible(Adapter, Adaptee)$

4.4.1.6. Delegation of Responsibilities

A *Delegate* is a ternary relationship defined between two participating classes or objects and a relation or function, the responsibility for whose invocation is delegated. It generally occurs in situations where a clan or tribe is available among the participating classes which realizes the relation delegated or invokes the function delegated. The syntax for this relation is Delegate (X, Delegator, Receiver), where Delegator, Receiver $\in C^d$ and X is either $\Re$, an ELePUS relation or f, f $\in F^d$.

Example: Delegation of the Creation Relation in the Factory Method:

$\exists Client\ \in C,\ ProductObject\ \in O,\ Creators \in H\ :$

$Delegate(Creation(ProductObject), Root(Creators), Nodes(Creators))$

4.4.1.7. Dynamic-Binding

One of the major limitations of LePUS is in specifying dynamic properties with static observations. Some constructs indicating dynamic behavior and temporal characteristics and provision for specifying objects and object interactions are included in ELePUS to address this limitation of LePUS. Dynamic-Binding is one such binary ground relation defined between two class or object variables. A Dynamic-Binding relation defined on class variables is inherently viewed as a Dynamic-Binding relation between Instantiations of the class variables. Hence, this is a pure object relation.

Example: Dynamic-Binding in the Factory-Method

The Factory Method pattern dynamically configures a client object with a different Product object. This property is captured by the Dynamic-Binding relation as given below:

Intent of Factory Method

$\exists Client \in C, ProductObject \in O, Creators \in H$:

$DynamicBinding(Client, ProductObject)$

4.4.1.8. Specifying Cardinality

Cardinality is a polymorphic relation in ELePUS that acts both as an attribute specifier and attribute representer for object, function, class or hierarchy variables. The Cardinality relation on ELePUS variables capture the number of elements in the set. The Cardinality relation as an attribute specifier is defined as

Cardinality (V,n), $V \in O^d \mid V \in C^d \mid V \in F^d \mid V \in H^d \wedge n \in [0,\ldots,\infty)$.

This qualifies as an attribute specifier, because this relation places a constraint on the value of the cardinality attribute for the concerned participant. The Cardinality relation as a attribute representer is a unary relation defined on the participant, like Cardinality (V), $V \in O^d$ or $V \in C^d$ or $V \in F^d$ or $V \in H^d$. This relation qualifies as an attribute representer, as it can substitute for a numeric value in the context of cardinality of the object/other participating variable concerned. This kind of Cardinality relation participates in other relations like Equivalence, etc.

Example 1: Cardinality Relation as Attribute Specifier in Singleton

$\exists\, SingletonObject \in O, Singleton \in C$:

$Cardinality(Instantiation(Singleton), 1)$

Example 2: Cardinality Relation as Attribute Representer in Prototype

$\exists\, Products \in 2^H;\ Factories \in H$:

$NOT(Equivalence(Cardinality(Factory), Cardinality(Products)))$

4.4.2. Specifying Recursive Behavior

The property of recursion is made use of to specify the delegation of relations in lists and trees. A recursive relation is applied repeatedly on the constituents of the participants until an end condition is reached. The termination condition for a delegate relationship is provided by another relation, *Invocation (c,f)*, where $f \in F^d$ and $c \in C^d$. In the recursive delegate relation, the class variables and object variables are interchangeable. It can be safely assumed that these relations inherently convert class variables to object variables using the Instantiation relation.

The recursive property is defined for the Delegate relation and is specified by means of an $*$ (asterisk) symbol. The recursive delegate relation defined for a function variable f, a class/object variable defined as a list or tree, whose elements are called Handlers, with the termination condition, a Handle relation can be specified as:

$Delegate^*$ (f, $Element_i$(Handler), $Element_j$(Handler),
  Invocation ($Element_j$(Handler),f))

The *Element* unary relation is a syntactic-sugar for a set of ELePUS/predicate calculus relations that represent a member element of either a List or Tree. $Element_i(Handler)$, $Element_j(Handler)$ represent distinct member elements of Handler. It is not provided as an ELePUS construct as it is seldom used anywhere else.

Recursive Delegate in Chain of Responsibility

$\exists Client \in C;\ Handle\text{-}Request \in 2^F;\ Handler \in 2^C$:
  $Clan(Handle\text{-}Request, Handler) \wedge$
  $List(Handler) \wedge$
  $Invocation(Client, Handle\text{-}Request) \wedge$
  $Delegate^*(Handle\text{-}Request, Element_i(Handler), Element_j(Handler),$
    $Invocation(Node_j(Handler), Handle\text{-}Request))$

4.4.3. Redefining LePUS Relations for ELePUS Environment

Due to the introduction of explicit object variables and the broader scope of ELePUS, some of the LePUS relations need to be redefined with respect to the new additions in ELePUS. For the purpose of readability and ease-of-use, some relations have been duplicated with a different name and modified definitions. This section gives a brief description of such changes along with examples.

- **Define as semantic double of Defined-In relation**

  The *Defined-In(f,c)* relation specifies that a function f is defined in class c and can be invoked via an instantiation of c. However, since ELePUS specifies the Intent and Applicability sections as well, where the need for a a function f to be defined in a class c may have to be specified. Hence for such specifications, for the purpose of readability, ELePUS offers *Define (f,c)*, which indicates that function f needs to be defined in class c. After this relation is specified, f can participate in other relations as if it had been defined through a *Defined-In* relation.

  <u>Example : Define in Observer</u>

  In specifying the Intent section of Observer pattern, a need for data consistency through one-many dependency needs to be realized. This can be achieved using the definition of an Update function on Observer classes. This need is specified using the Define relation on Update and Observer as given below:

  $\exists\, Subject \in C,\ Observers \in 2^{C}, Update \in F :$

  $\quad Reference\text{-}To\text{-}Many^{\rightarrow}(Subject, Observers) \wedge$

  $\quad Define(Update, Observers) \wedge$

  $\quad Trigger(State\text{-}Change(Subject), Update)$

- **Invocation (o,f)**

  In specifying collaborations, it is not always possible to define Invocation (f_1, f_2), since a function f_1 may not have been defined at all, but it will be necessary to declare the invocation of function f by an instantiation of a class variable. *Invocation (c,f)* is interpreted to mean *Invocation (Instantiation(c),f)* .

Example : Invocation (o,f) in Prototype

$\exists\, Prototype \in H$, $Client \in C$, $Clone \in 2^F$:

$clan(Clone, Prototype) \wedge$

$Invocation(Client, clone)$

- **Polymorphism of Creation Relation**

  In LePUS, *Creation (f,c)* was defined. It is interpreted in ELePUS to mean *Creation (f,Instantiation(c))* . A more accurate definition is *Creation (f,o)*. However, in many instances, when the knowledge of the function that does the creation is not available, *Creation (c,o)* or *Creation (o,o)* are used. *Creation (c,o)* $\Rightarrow$ *Creation (f,o)* $\wedge$ *Defined-In (f,c)*.

  Example of Creation (c,o) in Builder

  In the specification of Collaboration of the Builder pattern, an object of the Director class is created by client object, as given below:

  $\exists\, Client$, $Director \in C$:

  $Creation(Client, Director)$

  Creation (Client, Director) is interpreted as *Creation (Instantiation (Client), Instantiation(Director))*.

4.5. Observations and ELePUS Grammar

4.5.1. Observations

1. Formal specification does not eliminate existing text-based specifications altogether; it only complements them.

2. It may not be possible or not even fully necessary to formally specify all aspects of all patterns. It is sufficient to have a small and powerful set of vocabulary that can specify most patterns successfully. Even for patterns whose components

cannot be formally specified fully, formal specification of major characteristics can be accomplished and are almost as useful as specification of all aspects of pattern components.

3. The Structural aspect of patterns specifies well defined class, object and function variables and their structural relationships. Hence, for ease-of-understanding, structural specifications are accompanied with visual representations. However, other aspects of patterns such as intent, applicability, and collaborations specify the design constraints and behavioral relationships at a higher level of abstraction and they do not need visual representations for understanding purposes.

4.5.2. BNF Grammar of ELePUS

This section provides a BNF grammar for ELePUS to facilitate tool support. The following conventions are used in typesetting the grammar. The Start symbol is given in bold face capital letters. The non-terminals are in capital letters and terminals in italised font in small case letters whenever appropriate. Each production is of the form LHS $\longrightarrow RHS_1 \mid RHS_2 \mid \ldots \mid RHS_n$, where is RHS is either a sequence of terminals or non-terminals or both. Non-terminals are expanded through production relations until they reach a set of sequence of terminals. Terminals and Non-terminals are separated from each other and from one another by whitespace. A null production is identified by an ϵ (epsilon) symbol.

4.5.2.1. Terminals and Keywords of the Grammar

A brief description of the various terminals, literal symbols and keywords of ELePUS is given below.

Literals The literal symbols that occur in ELePUS grammar are, '(', ')', ',' , ':', '$\rightarrow$', '$\leftrightarrow$', '+', and '!'.

Character Symbols The character symbols *c, C, o, O, f, F, H, Env* are used to denote variable types in ELePUS apart from the higher order logic symbols like $\in$, $\wedge$, $\exists$, and $\forall$.

Keywords The following are the keywords of ELePUS. *Abstract, Argument-1, Assignment, Cardinality, clan, Class, Commute, Couple, Decouple, Define, Defined-In, Delegate, Dynamic-Binding, Encapsulation, Equivalence, Extract-State, Forwarding, Hold-Reference, Inheritance, Instantiation, Interface, Interface-Compatible, Invocation, Leaf, List, Node, NOT, Production, Public-Access, Reference-To-Many, Reference-To-One, Restricted-Access, Return-Type, Roll-back, State, State-Change, Tree, Tribe, Trigger*

Others Some of the other terminals used in ELePUS are *identifiers, time-stamp, dimension and dim*. The syntax for these expressions are given using regular expressions in the next section.

4.5.2.2. Notes on the Terminals of ELePUS Grammar

This section provides brief descriptions of the terminal symbols used in ELePUS grammar. The descriptions are supported with regular expressions or grammars specifying the syntax of the terminals in unambiguous manner to facilitate identification of the terminal tokens in a given ELePUS specification. Standard notation for regular expressions are used in the following specifications.

Identifier Identifiers are alphanumeric strings of arbitrary length (length ≥ 1) used to name the variables. Underscore characters (_) are permitted along with alphabets and digits.

$\textit{identifiers} \rightarrow ([a-zA-Z]|[0-9]|\_)^{+}$

digit $\rightarrow$ [0-9]

alphabets $\rightarrow$ [a-zA-Z]

Dimension This token specifies any numeric value greater than 1. *dimension* $\rightarrow$ 0*[2-9] | 0*[1-9]$([0-9])^+$

Time-stamp This specifies the time instant of the occurrence of an event. Though, an *identifier* can act as a *time-stamp*, to make comparisons between them possible, only natural numbers are admitted as time-stamps.
time-stamp $\rightarrow ([0-9])^+$

4.5.2.3. ELePUS Grammar

ELePUS-FORMULA $\longrightarrow$
PARTICIPANT-LIST : RELATION-LIST

PARTICIPANT-LIST $\longrightarrow$
PARTICIPANTS , PARTICIPANT-LIST | PARTICIPANTS

PARTICIPANTS $\longrightarrow$
IDENTIFIER-LIST $\in$ VARIABLE-TYPE

IDENTIFIER-LIST $\longrightarrow$
identifier IDENTIFIER-LIST | *identifier*

VARIABLE-TYPE $\longrightarrow$
CLASS-VAR | OBJECT-VAR | FUNC-VAR | HIER-VAR | ENV-VAR

CLASS-VAR $\longrightarrow$
ZDIM-CVAR | ODIM-CVAR | HDIM-CVAR

ZDIM-CVAR $\longrightarrow$ c

ODIM-CVAR $\longrightarrow C$

HDIM-CVAR $\longrightarrow C^{dimension}$

OBJECT-VAR $\longrightarrow$
ZDIM-OVAR | ODIM-OVAR | HDIM-OVAR

ZDIM-OVAR $\longrightarrow o$

ODIM-OVAR $\longrightarrow O$

HDIM-OVAR $\longrightarrow O^{dimension}$

FUNC-VAR $\longrightarrow$
ZDIM-FVAR | ODIM-FVAR | HDIM-FVAR

ZDIM-FVAR $\longrightarrow f$

ODIM-FVAR $\longrightarrow F$

HDIM-FVAR $\longrightarrow F^{dimension}$

HIER-VAR $\longrightarrow$
ODIM-HVAR | HDIM-HVAR

ODIM-HVAR $\longrightarrow H$

HDIM-HVAR $\longrightarrow H^{dimension}$

ENV-VAR $\longrightarrow$ *Env*

RELATION-LIST $\longrightarrow$

RELATION $\wedge$ RELATION-LIST | RELATION

RELATION $\longrightarrow$

CLANS | TRIBES | CARDINALITY-RELATION | STATE-RELATION | GROUND-RELATION | GENERAL-RELATION

CLANS $\longrightarrow$

clan (FUNC-VAR , CLASS-VAR)

TRIBES $\longrightarrow$

tribe (FUNC-VAR , CLASS-VAR)

CARDINALITY-RELATION $\longrightarrow$

BINARY-CARDINALITY |

UNARY-CARDINALITY

BINARY-CARDINALITY $\longrightarrow$

Cardinality (CLASS-VAR | OBJECT-VAR | FUNC-VAR | HIER-VAR | UNARY-INSTANTIATION, *digit*)

UNARY-CARDINALITY $\longrightarrow$

Cardinality (CLASS-VAR | OBJECT-VAR | FUNC-VAR | HIER-VAR | UNARY-INSTANTIATION)

STATE-RELATION $\longrightarrow$

State (OBJ-VAR)

GROUND-RELATION $\longrightarrow$

GROUND-UNARY | GROUND-NARY | GROUND-POLY

GROUND-UNARY $\longrightarrow$

Abstract (CLASS-VAR) |

Class (OBJ-VAR) |

Public-Access (FUNC-VAR) |

Public-Access (CLASS-VAR) |

Restricted-Access (FUNC-VAR) |

Restricted-Access (CLASS-VAR) |

List (CLASS-VAR) |

List (FUNC-VAR) |

Tree (CLASS-VAR) |

Tree (FUNC-VAR) |

GOUND-NARY $\longrightarrow$

$Assignment^{RDESCRIPTOR}$ (FUNC-VAR-EXCL , CLASS-VAR-EXCL , CLASS-VAR-EXCL) |

$Inheritance^{RDESCRIPTOR}$ (CLASS-VAR-EXCL , CLASS-VAR-EXCL) |

$Invocation^{RDESCRIPTOR}$ (FUNC-VAR-EXCL , FUNC-VAR-EXCL) |

$Invocation^{RDESCRIPTOR}$ (CLASS-VAR-EXCL , FUNC-VAR-EXCL) |

$Forwarding^{RDESCRIPTOR}$ (CLASS-VAR-EXCL , CLASS-VAR-EXCL) |

$Production^{RDESCRIPTOR}$ (FUNC-VAR-EXCL , FUNC-VAR-EXCL) |

$Production^{RDESCRIPTOR}$ (CLASS-VAR-EXCL , FUNC-VAR-EXCL) |

$Dynamic\text{-}Binding^{RDESCRIPTOR}$ (CLASS-VAR-EXCL , CLASS-VAR-EXCL) |

$Reference\text{-}To\text{-}One^{RDESCRIPTOR}$ (CLASS-VAR-EXCL , CLASS-VAR-EXCL) |

$Reference\text{-}To\text{-}Many^{RDESCRIPTOR}$ (CLASS-VAR-EXCL , CLASS-VAR-EXCL)

|

$Define^{RDESCRIPTOR}$ (FUNC-VAR-EXCL , CLASS-VAR-EXCL) |
$Defined\text{-}In^{RDESCRIPTOR}$ (FUNC-VAR-EXCL , CLASS-VAR-EXCL) |
$Argument\text{-}1^{RDESCRIPTOR}$ (FUNC-VAR-EXCL , CLASS-VAR-EXCL) |
$Return\text{-}Type^{RDESCRIPTOR}$ (FUNC-VAR-EXCL , CLASS-VAR-EXCL) |
$Hold\text{-}Reference^{RDESCRIPTOR}$ (CLASS-VAR-EXCL , CLASS-VAR-EXCL)

GROUND-POLY $\longrightarrow$
CREATION-REL | INSTANTIATION-REL

CREATION-REL $\longrightarrow$
UNARY-CREATION |
BINARY-CREATION

UNARY-CREATION $\longrightarrow$
Creation (CLASS-VAR)

BINARY-CREATION $\longrightarrow$
$Creation^{RDESCRIPTOR}$ (CLASS-VAR-EXCL | FUNC-VAR-EXCL , CLASS-VAR-EXCL | OBJ-VAR-EXCL | UNARY-INSTANTIATION)

INSTANTIATION-REL $\longrightarrow$
UNARY-INSTANTIATION
BINARY-INSTANTIATION

UNARY-INSTANTIATION $\longrightarrow$
Instantiation (CLASS-VAR)

BINARY-INSTANTIATION $\longrightarrow$

$Instantiation^{RDESCRIPTOR}$ (CLASS-VAR-EXCL | OBJ-VAR-EXCL , CLASS-VAR-EXCL)

RDESCRIPTOR $\longrightarrow$
TRANSITIVE TOTAL | REGULAR | H-TOTAL

TRANSITIVE $\longrightarrow$ + | ϵ

TOTAL $\longrightarrow$ $\rightarrow$ | ϵ

REGULAR $\longrightarrow$ $\leftrightarrow$ | ϵ

H-TOTAL $\longrightarrow$ $\rightarrow$H | ϵ

CLASS-VAR-EXCL $\longrightarrow$ CLASS-VAR EXCLUSIVE

OBJ-VAR-EXCL $\longrightarrow$ OBJ-VAR EXCLUSIVE

FUNC-VAR-EXCL $\longrightarrow$ FUNC-VAR EXCLUSIVE

HIER-VAR-EXCL $\longrightarrow$ HIER-VAR EXCLUSIVE

EXCLUSIVE $\longrightarrow$ **!** | ϵ

GENERAL-RELATION $\longrightarrow$
GENERAL-UNARY | GENERAL-NARY

GENERAL-UNARY $\longrightarrow$
Node (HIER-VAR) |

Leaf (CLASS-VAR) |

Extract-State (OBJ-VAR) |

Encapsulation (HIER-VAR | CLASS-VAR) |

INTERFACE-REL

INTERFACE-REL $\longrightarrow$

Interface (CLASS-VAR)

GENERAL-NARY $\longrightarrow$

$Decouple^{REGULAR}$ (RELATION | CLASS-VAR | HIER-VAR | ENV-VAR , RELATION | CLASS-VAR | HIER-VAR | ENV-VAR) |

$Couple^{RDESCRIPTOR}$ (CLASS-VAR , CLASS-VAR | HIER-VAR) |

$Equivalence^{RDESCRIPTOR}$ (OBJ-VAR , OBJ-VAR) |

$Equivalence^{RDESCRIPTOR}$ (CLASS-VAR , CLASS-VAR) |

$Equivalence^{RDESCRIPTOR}$ (FUNC-VAR , FUNC-VAR) |

$Equivalence^{RDESCRIPTOR}$ (UNARY-CARDINALITY , UNARY-CARDINALITY) |

$Equivalence^{RDESCRIPTOR}$ (INTERFACE-REL , INTERFACE-REL) |

$Interface\text{-}Compatible^{RDESCRIPTOR}$ (CLASS-VAR , CLASS-VAR) |

$Delegate^{RDESCRIPTOR}$ (FUNC-VAR , CLASS-VAR , CLASS-VAR) |

$Delegate^{RDESCRIPTOR}$ (FUNC-VAR , CLASS-VAR , OBJ-VAR) |

$Delegate^{RECURSIVE}$ (RELATION , CLASS-VAR , CLASS-VAR , TERMINAL-RELATION) |

$Delegate^{RECURSIVE}$ (RELATION , OBJ-VAR , OBJ-VAR , TERMINAL-RELATION) |

$State\text{-}Change$ (OBJ-VAR , time-stamp , time-stamp) |

$Roll\text{-}back$ (STATE-RELATION , time-stamp) |

$Trigger^{RDESCRIPTOR}$ (RELATION , FUNC-VAR)

RECURSIVE $\longrightarrow *$

The following chapter presents the validation of ELePUS by applying it to the design patterns in the GoF catalog [3]. It also evaluates ELePUS with respect to the evaluation criteria discussed in Chapter 2.

CHAPTER 5. ELEPUS: VALIDATION AND APPLICABILITY

Every word or concept, clear as it may seem to be, has only a limited range of applicability.
- Werner Karl Heisenberg (1901-1976)

The previous two chapters (3 & 4) provided a comprehensive account of ELePUS, along with the need for ELePUS, the language features, syntax and semantics of the constructs and the language grammar. This chapter validates ELePUS by specifying the patterns in GoF catalog and assessing ELePUS with respect to the evaluation criteria described in chapter 2. Potential areas of application of ELePUS are also outlined and illustrated with examples, wherever appropriate.

5.1. Validation

This section validates ELePUS by evaluating it against the evaluation criteria of chapter 2. The ELePUS specifications of the two patterns taken for evaluation, *Abstract Factory (AF)* and *Factory Method (FM)*, are presented below. This is followed by the assessments of ELePUS against the evaluation criteria, for which LePUS had limitations. Since the informal GoF specifications of these two patterns, AF and FM were presented previously (sections 2.2.3.2 & 2.2.3.1 respectively), only the *intents* of the two patterns are repeated here, to avoid redundancy.

Intent of Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Intent of Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

5.1.1. ELePUS Specification of Abstract Factory (AF)

Intent

$\exists\ Creators,\ Products \in H,\ factory\text{-}method \in 2^F :$

$Define(factory\text{-}method, Creators) \wedge$

$Couple(Nodes(Products)) \wedge$

$Production(factory\text{-}method, Products)$

Applicability

$\exists\ Product\text{-}Family \in 2^H,\ SYSTEM \in Env :$

$Couple(Nodes(Product\text{-}Family)) \wedge$

$Decouple(SYSTEM, Creation(Product\text{-}Family))$

Structure

Refer to Figure 2.4 in Chapter 2.

$\exists\ Factory\text{-}Methods \in 2^{2^F},\ Creators \in H,\ Products \in 2^H :$

$tribe(Factory\text{-}Methods, Creators) \wedge$

$Production^{\leftrightarrow}(Factory\text{-}Methods, Products) \wedge$

$Return\text{-}Type^{\leftrightarrow}(Factory\text{-}Methods, Products) \wedge$

$Commute_{Return-Type,Creation}(Factory\text{-}Methods, Products)$

Collaborations

$\exists\ Creators \in H,\ Products \in 2^H,\ CreateProduct \in 2^{2^F} :$

$Cardinality(Instantiation(Node(Creators), 1)) \wedge$

$tribe(CreateProduct, Products) \wedge$

$Delegate(Production^{\leftrightarrow}(CreateProduct, Products), Root(Creators), Nodes(Creators))$

5.1.2. ELePUS Specification of Factory Method (FM)

<u>Intent</u>

$\exists\ Factory\text{-}Methods\ \in 2^F,\ Products,\ Creators \in H\ :$

$Production(Factory\text{-}Methods, Products) \wedge$

$Delegate(Instantiation(Product), Root(Creators), Nodes(Creators))$

<u>Applicability</u>

$\exists\ Client\ \in C,\ ProductObject\ \in O,\ Creators \in H\ :$

$Dynamic\text{-}Binding(Client, ProductObject) \wedge$

$Delegate(Creation(ProductObject), Root(Creators), Nodes(Creators))$

<u>Structure</u>

Refer to Figure 2.3 in Chapter 2.

$\exists\ factory\text{-}method\ \in 2^F,\ Creators,\ Products\ \in H\ :$

$clan(factory\text{-}method, Creators) \wedge$

$Production^{\leftrightarrow}(factory\text{-}method, Products) \wedge$

$Return\text{-}Type^{\leftrightarrow}(factory\text{-}method, Products) \wedge$

$Commute_{Return-Type,Creation}(factory\text{-}method, Products)$

<u>Collaboration</u>

$\exists\ Creators,\ Products\ \in H,\ \ factory - method\ \in 2^F\ :$

$Delegate(Production^{\leftrightarrow}(factory\text{-}method, Products), Root(Creators), Node(Creators))$

The following section presents the evaluation results of ELePUS for each evaluation criterion along with a brief explanation. For each criterion, the corresponding feature of ELePUS is ranked as either one of *Poor,Fair, Good* or *Very Good* or *Exists* or *Nil*.

5.1.3. Evaluation Conclusions for ELePUS

To avoid redundancy, this section provides the evaluation of ELePUS for only the criteria for which LePUS had limitations.

- **Specification of Behavioral Aspects:** ELePUS is rated *Very Good*, while LePUS was rated *Fair*. This is due to the provision for specification of Collaborations and added relations for specification of dynamic properties. For example, the delegation of the *Production* relationship in both the AF and FM patterns is captured elegantly in ELePUS specifications.

- **Specification of Design Constraints:** ELePUS is rated *Very Good*, while LePUS was rated *Good*. This is because, ELePUS allows for specifying Applicability sections, and provides for new predicates that capture design constraints. For example, in the AF pattern presented above, the need to use the nodes of Product-Family together is specified by the *Couple* relation available in ELePUS. Similarly, the need to make the *SYSTEM* independent from the creation of *Products* is also captured by means of the *Decouple* relation introduced in ELePUS.

- **Complementing Object Notations:** ELePUS, like LePUS, does not provide a well-defined mapping with any popular object notations. This is because of the limitations object notations have with respect to formally specifying design patterns. However, since ELePUS specifications accompany the text based specifications that make use of object-notations, this limitation is adequately addressed.

- **Support for Object-orientation:** ELePUS is rated *Very Good*, while LePUS had a rating of *Good*. This is because of the provisions for representation of objects as ground entities in ELePUS and provision of relations to describe object dynamics. For example, the notion of dynamic binding between the *Client* class and the *Product* object in the FM pattern is captured by the *Dynamic-Binding* relation provided in ELePUS.

- **Ease of Use:** Since ELePUS follows the same structure, mathematical model and feel of LePUS, the assessment remains the same (*Good*). This is because, unlike informal notations, any formal notation, however simple and concise its constructs may be, requires some learning activity.

- **Multi-level Representation:** ELePUS provides for Multi-Part Representation of patterns by allowing for independent specifications of the Intent, Applicability, Structure and Collaboration aspects of a pattern. However, it cannot be considered as equivalent to Multi-level Representation, which involves providing specifications at different levels of abstractions.

- **Representation of low-level details:** ELePUS is rated *Very Good*, while LePUS was rated *Fair*. This is a result of the availability of number of constructs in ELePUS aimed at facilitating representation of implementation specific details that are found in design patterns. For example, the limitation on the number of instances of *Creator* object in the AF pattern is specified using the *Cardinality* relation in the Collaboration section of the ELePUS specification.

- **Support for Pattern-repository Management:** ELePUS is rated *Very Good* while LePUS had a *Good* rating. This is because, pattern repository management issues such as inclusion, refinement, duplication and validation are better solved by means of the more complete specifications offered by ELePUS.

This proves that ELePUS has successfully addressed the limitations of LePUS and has achieved the goals of this research.

5.1.4. Validation of ELePUS Using the GoF Patterns

ELePUS was used to specify design patterns in the GoF catalog [3]. The ELePUS specifications of Intent, Applicability and Collaboration sections of these GoF patterns are provided in the Appendix B. For the Intent component, more than 70% of the patterns were comprehensively specified using eLePUS constructs. For the remaining few, ELePUS was able to capture the major characteristics. Complete specifications of applicability component achieved 67% success rate and a substantial specification of characteristics was achieved for the remaining 33% of the patterns. This is partly due the plethora of scenarios in which patterns are applicable and many require domain specific vocabulary. For example in the Iterator pattern, the actual intent and applicability cannot be specified fully unless new relations specific for the purpose of capturing the process of interpretation or compilation are introduced. The ELePUS vocabulary was designed to be generic and simple. However, new constructs or relations can be added, as and when required, as long as they follow the mathematical model of ELePUS. Greater success (about 90%) was achieved in Collaborations because ELePUS captures almost all of the interrelations between the ground variables that are the participants of the pattern. This success in specification of a well established pattern catalog (GoF) using ELePUS, validates the expressiveness and applicability of ELePUS as a specification language.

5.2. Applications of ELePUS

This section proves the effectiveness of ELePUS with respect to providing precise, unambiguous specifications for the major components of patterns involving intent, applicability, structure (using LePUS) and collaborations.

5.2.1. Unambiguous Specification

This section gives a few sample cases of misunderstandings or ambiguity arising out of current informal specification of design patterns. Such situations are ideal testbeds for ELePUS and hence, it is also illustrated how ELePUS specifications resolve these ambiguities.

5.2.1.1. Providing Unambiguous Specification to Intent Section of the Composite Pattern

The design-patterns mailing list [19] had a discussion about the exact meaning of the *intent* of the Composite pattern (Compose objects into **tree** structures to represent part-whole hierarchies) [3]. One OO architect argued that "...I think this is not adequately worded. The name 'tree' is over determined in the programming society. In this case it clearly means inheritance tree, but also very clearly people ... think they understand the Composite as some other tree (we have binary tree, B-tree, etc.). So that alone is an argument against the choice of the word 'tree' here. ... composite has nothing to do with trees, apart from inheritance trees that is". The clarification from Ralph Johnson (one of the authors of [3]) is "....I am afraid you are wrong. Composite is exactly about binary trees and other kind of trees....Composite also tell you how to make inheritance trees but lots of the design patterns do that. Composite is the only one in 'design patterns' about how to make objects that are trees".

Such misinterpretations, although they may be the mistake of the reader, are unavoidable with textual specifications. This misunderstanding could have been avoided if ELePUS specifications had accompanied the informal text based specifications. ELePUS provides a way to specify structural properties of patterns using static relations like *List* (α) or *Tree* (α). It should also be noted that LePUS, by itself, does not provide a complete solution to this problem as it does not specify aspects other than structure and does not have the vocabulary to describe constructs such as Lists or Trees. The ELePUS specification of the Intent aspect of the Composite pattern is reproduced here.

<u>Intent of the Composite Pattern</u>

$\exists\ Objects \in 2^{O}$:

$Tree(Objects) \wedge$

$Equivalence(Interface(Objects), Interface(Tree(Objects)))$

It is obvious from the ELePUS specification that *Composite* pattern aims to compose objects as a tree structure and it does not refer to the inheritance trees, as misunderstood by the initiator of the above presented discussion.

5.2.1.2. Creational Patterns

In online discussion groups, like [19], an often repeated posting by pattern users is the exact difference between some of the Creational patterns. Also asked are clear definitions of the pertinent situations in which each of the Creational patterns are applicable. In most cases, the ambiguities revolve around abstract factory, factory method and prototype patterns. This confusion of which pattern to apply in which scenario, is resolved by ELePUS specifications of the Applicability and Intent sections of these patterns. Reproduced below is one such posting in [19]. Since the posting was verbose, only the pertinent sections are reproduced here: "...I have a problem with Factory Method (FM) : when it is used rather than a Prototype? ... It seems that FM is much less useful than Prototype and that it should only be considered when Prototype cannot be used. If I want to create objects of a class which is unknown (it is only known that it's a subclass of class X) to some other class then cloning seems simpler than creating an artificial parallel hierarchy...". The OO analyst also gives an in-depth analysis of many scenarios to buttress his claim. The response from Ralph Johnson (one of the authors of [3] which presented the patterns under discussion) is: "...*Useful* is the wrong word. Factory method is much less *powerful* than Prototype. If you don't need the power of Prototype, you should just use FM...Whenever a class produces an object that it is going to return, I use FM instead of producing objects in-line. If I want to avoid a parallel hierarchy of factories, I replace FM by another Creational pattern like Prototype...". Formal ELePUS specifications of the Applicability aspect of both the patterns are reproduced below. The conciseness, preciseness and effectiveness of ELePUS specifications can be observed in this example, with respect to resolving the above discussion.

Specification of Applicability Section of Factory Method

$$\begin{aligned}
&\exists Client \in C,\ ProductObject \in O,\ Creators \in H\ :\\
&\quad Dynamic\text{-}Binding(Client, ProductObject) \wedge\\
&\quad Delegate(Creation(ProductObject), Root(Creators), Nodes(Creators))
\end{aligned}$$

Specification of Applicability Section of Prototype

$\exists Products \in 2^{H};\ Factories \in H$:

$Decouple(SYSTEM, Creation(Products)) \wedge$

$NOT(Equivalence(Cardinality(Factory), Cardinality(Products)))$

It can be observed that Prototype may be applied in situations that warrant elimination of parallel hierarchy of Factories and Products as captured by the ELePUS relation,

NOT (Equivalence (Cardinality (Factory), Cardinality (Products))). It is also evident that the Factory Method is applicable in situations where a class does not know the class of objects it creates and delegates those responsibilities to its subclasses.

The next chapter discusses the future research activities identified with respect to providing tool support using ELePUS. Finally the next chapter concludes the thesis by summarizing the contributions of this thesis.

CHAPTER 6. FUTURE WORK AND CONCLUSIONS

This chapter presents a discussion of future possible threads of research for providing tool support for patterns using ELePUS. It also provides a summary of the research presented in the previous chapters and the conclusions of this research.

6.1. Discussions and Future work

To realize the full potential of ELePUS, organized tool support is a pre-requisite. Tool support can be provided using ELePUS in two ways. They are (i) Applying design patterns in CASE tools and providing other such tool support to facilitate use of design patterns in software design and (ii) To develop tools and guidelines for the science of patterns itself.

The concept of pattern languages are gaining more acceptance as a way to approach design from a higher level. A pattern language provides a set of patterns that solve problems in a specific domain. Pattern languages document the relationships between patterns. They imply the process to apply the language to completely solve a specific set of design problems [38].

6.1.1. Pattern-Oriented Analysis and Design

The concept of Pattern-Oriented Analysis and Design (POAD) [39, 38] involves utilizing patterns as building blocks or components at design level. The approach glues the design structures of patterns at various levels of abstractions for the purpose of developing pattern-oriented designs. However, the tool support provided assumes that the designer knows all the patterns in the repository and their applicability. By using ELePUS based specifications, POAD tools can automatically mine the required patterns given a ELePUS based specification of the design problem. The same use of ELePUS can also be made use of in CASE tools like Rational Rose, ClearCase, GDPro, etc.

6.1.2. Bruhaspati

Bruhaspati is a proposed pattern framework, which refers to the set of tools, guidelines and integrated pattern repository of ELePUS specification of patterns. This framework is for enhancing the field of design patterns itself. This will help achieve the goal of resolving the management of the pattern repository. The creation and maintenance of a pattern-repository involves addressing issues such as, *Inclusion*, *Duplication*, *Refinements*, and *Validation*. *Inclusion* refers to the process by which a design solution is accepted as a pattern. By definition [3], if a problem re-occurs, then the proven efficient solutions to that problem qualify as a pattern. However, it is possible that the same pattern is already identified and incorporated into the repository. This is called *duplicity*. When two patterns exhibit similarity and differ only in a minor fashion (we can have two similar solution for a problem), then one pattern is said to *refine* another. *Inclusion* of a design pattern, as a new entry, into the repository involves making sure that *duplicity* with pre-existing patterns in the repository is nil and the new entrant is not a *refinement* of an already existing pattern. In case of *refinement*, the contender is added to the repository and all the patterns that exhibited a *refinement* relationship with this pattern are grouped together as set of refinements for the root pattern, which is the one that was first added to the repository.

A pattern A is said to *duplicate* a pattern B if $\Phi \rightarrow \Theta \wedge \Theta \rightarrow \Phi$, , where Φ and Θ are the ELePUS formulae representation of patterns A and B (The relations apply to all corresponding formulae for various sections like Intent, Applicability and Structure between the two patterns). A pattern A is said to *refine* a pattern B, if

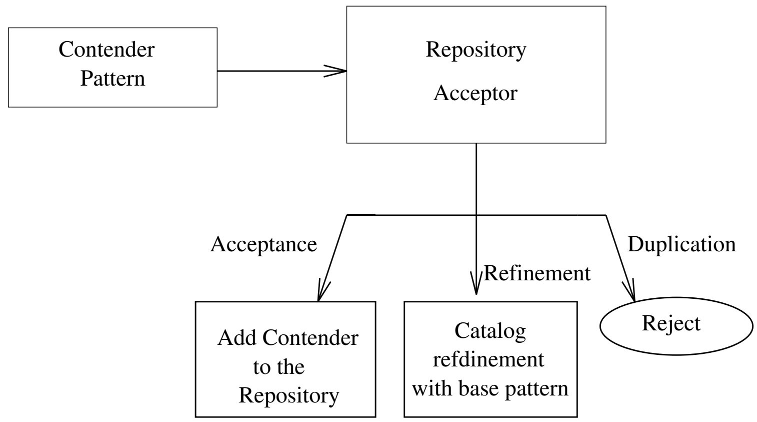

Figure 6.1. Pattern Repository: The procedure for Inclusion

$\Phi \rightarrow \Theta$, where Φ and Θ refer to the ELePUS formulae for the two patterns A and B. LePUS considered only the structural specification of patterns for resolving the issues of refinement and duplication, while ELePUS considers the Intent and Collaborations as well, thereby providing for a more comprehensive scheme to resolve the issues. The Figure 6.1 represent the various phases of pattern-repository management that are involved in Inclusion or addition of a candidate pattern into the repository. Hence it is evident that ELePUS offers a way of formally specifying design patterns and can help harness the full potential of design patterns as identified in Chapter 1.

6.2. Conclusions

Software design patterns are gaining increasing acceptance and usage. They capture design experience and have many advantages - as a learning aid, documentation aid, design vocabulary and greatly assist in the overall design process. The full benefits of patterns are not exploited by the existing informal specification schemes. Formal pattern specification has been theoretically proved feasible and allows for many advantages. The prominent ones of them are:

- Unambiguous, precise and concise specifications.

- Facilitates a provision for the tool support as formal specifications are easier to incorporate into tools, like CASE tool support for design patterns.

- Provides for mechanisms to logically and formally compare the relationships between two patterns.

- A formal specification of design problems and design patterns can assist automated mining of pertinent pattern(s) from a pattern repository.

We have seen that there are some semi-formal and formal approaches proposed for injecting formalism into pattern specification. However, there has been no comprehensive evaluation or study undertaken on these approaches to benchmark the formal approaches against a well defined criteria for useful specification that will realize the above mentioned possible advantages of patterns. This research work aimed at the following goals.

1. To provide a comprehensive analysis, survey and evaluation of various formal specification methodologies for specifying design patterns, their merits and demerits.

2. If all the languages available are deficient in one sense or another, develop a new, comprehensive specification language either from scratch or build using the most promising language identified.

3. To define and describe the semantics and syntax of the new language and indicate its applicability with respect to various aspects of design patterns specification and usage. Also, apply the approach to a large set of widely used design patterns to form an infant pattern repository.

4. To identify new research areas which will enhance the applicability of the specification language developed or identified.

Existing formal mechanisms for patterns were surveyed and were assessed with respect to comprehensive evaluation criteria that broadly fall into three categories -

mathematical model, comprehensiveness, and versatility. The evaluation process involved theoretical analysis and experimental evaluation of the formal methods by applying each of the formal methods, viz., LePUS (LanguagE for Pattern Uniform Specification), Contracts, Precise Visual Notation, and DisCo (DIStributed CO-operation), to specify two selected patterns, Abstract Factory and Factory Method. The evaluation results established that none of the existing methodologies realizes the full potential of pattern formalism. LePUS (LanguagE for Pattern Uniform Specification) was identified as a having many positive aspects and a very rigorous mathematical model and is extensible.

LePUS is a predicate calculus based language which expresses the structure part of each pattern as a formula of LePUS predicates. While the approach and mathematical foundation of LePUS were sufficient for the stated goals, LePUS had major problems that made LePUS not a complete language for specifying patterns and achieving the objectives of realizing all the benefits of formalism. Some of the crucial deficiencies of LePUS were that (i) It specified only the structure aspect of patterns and not other aspects like Intent, Applicability and Collaborations, (ii) LePUS lacked a complete vocabulary to specify in detail, object relationships, temporal behavior and other generic system descriptions and a well formed Grammar. This research extended LePUS by addressing the above issues; it arrived at an extended, enhanced variant of LePUS called ELePUS.

ELePUS adhered to the mathematical model of LePUS. The major enhancements made in ELePUS are:

- Objects were introduced as ground entities, and relations for specifying object dynamics, like the binding types and specifying state information of objects were added.

- New constructs and relationships were added to specify implementation level details like the cardinality of the ground variables, etc.

- Provisions were made to facilitate specification of object-oriented relations like encapsulation, abstractions, public and restricted access specifiers.

- ELePUS specifies sections of patterns other than Structure like, Intent, Applicability and Collaborations.

- To specify the additional aspects of patterns, suitable relations and variables were introduced, like the notion of environmental variables, recursive relations, etc.

- The LePUS specifications were redefined with respect to the new ground variables and a comprehensive and formal BNF grammar of the language was provided to facilitate tool support.

The development of the language was followed by illustrations of its applicability. ELePUS was validated against all the design patterns presented in GoF catalog [3]. ELePUS was also evaluated against the established evaluation criteria and the enhancements ELePUS offered over LePUS were illustrated. The evaluation established that ELePUS has successfully enhanced and extended LePUS and achieved the research objectives.

Finally, it can be concluded that the existing informal specifications of design patterns need to be augmented with formal specifications to harness the full potential of design patterns and to facilitate a tool support. All of the existing formal approaches for pattern specification have many limitations. Of the existing formal languages, LePUS (LanguagE for Pattern Uniform Specification) provides the required degree of formalism suitable for specifying design patterns. However LePUS has many limitations and specifies only the structure component of design patterns. ELePUS (Extended LePUS) is an enhanced and extended version of the language, that builds on the formal base provided by LePUS and addresses all the major limitations of LePUS adequately. ELePUS provides unambiguous and complete specifications for software design patterns and is proven for a substantial number of patterns from the

GoF catalog [3]. ELePUS proves to be the most suitable vehicle to exploit the full benefits offered by design patterns.

APPENDIX

APPENDIX A

A.1. Glossary of Terms

This section provides brief and intuitive definitions for most of the often used ELePUS language constructs. The definitions for the LePUS relations that are inherited by ELePUS are reproduced from [22]. This section is provided for the purpose of readability and for quick reference. The relations are presented in alphabetical order. Unless otherwise stated, the characters *c, f, o, h* refer to class, function, object and hierarchy variables respectively, of arbitrary dimensions.

Abstract(c) The class c, is an abstract class.

Argument-1(f,c) The first argument of f is c.

Assignment(f,c_1, c_2) This relation designates that the reference from c_1 to c_2 is assigned a value within the flat structure of the body of function f.

Cardinality (v) The unary cardinality relation is defined on a class, object, function, or hierarchy variable. This designates the count of the instantiations (for class and object variables) or function arguments (for function variables) or number of nodes in the hierarchy tree (for hierarchy variables).

Cardinality (v,n) Binary cardinality is similar to the unary cardinality (presented above). However, it designates a numeric value to Cardinality (v).

Clan (f,c | h) designates that the function f is a clan in the class c or the hierarchy h. Clans are a special case of function sets, and all the functions involved in a clan relationship share the same signature and each is defined in a different class.

class(o) Designates the class variable of the object instantiation o.

Couple(v_1, v_2) Designates that the set of variables denoted by v_1 and v_2 have high degree of interconnectivity and interdependence.

Creation(f | c | o, c | o) Designates the creation of the class or object (argument 2) within the flat structure of function f or some function defined in c or o.

Decouple(v_1, v_2) Designates that the two variables are independent of one another and have minimal interconnectivity.

Define (f,c) | Defined-In (f,c) Indicates that the function f, is defined in the class c.

Delegate (f | $\Re$,c_1,c_2) Indicates that the responsibility for invocation of the function f or realizing the ELePUS relation $\Re$ is delegated to c_2 by c_1.

Dynamic-Binding (c | h, c | h) Indicates that binding between the two class variables (or nodes(h)) is dynamic or changes at runtime.

Encapsulation(c) Indicates that the internal details of class c are hidden from outside.

Equivalence(v_1, v_2) v_1 and v_2 are both of the same type and can be either class or object or function variables or unary cardinality relations. This function when satisfied, indicates an equivalence of interfaces for classes and objects and numerical equivalence for cardinality relations and equivalence of name and signature for function variables.

Extract-State(o) Designates that the state information of the object variable o, can be computed from the environment or can be stored for future calibration of the object.

Forwarding(f_1, f_2) Designates a function call to f_2 within the flat structure of f_1, using the formal arguments of f_1 as the actual arguments of f_2 in the same order and that the signatures of the two functions are identical.

Hold-Reference($c_1 \mid o_1$, $c_2 \mid o_3$) Indicates that the class or object variable c_1 or o_1 holds a reference to the instantiation of c_2 or o_2, whatever the case be.

Inheritance(c_1, c_2) Indicates that the class c_1 (possibly indirectly) "inherits" from the class c_2.

Instantiation(c) Refers to the object variables that are instantiations of this class.

Interface-Compatible (c_1, c_2) Indicates that the the interfaces of the two classes are "compatible".

Invocation(f_1, f_2) Designates a function call to f_2 within the flat structure of f_1.

List(c | o) Indicates that the objects o, or the object instantiations of the class c form a list.

NOT($\Re$) Indicates the negation of the ELePUS relation $\Re$.

Production(f,c) An object of class c is created and returned by f. Production (f,c) implies Creation (f,c) and Return-Type (f,c).

Public-Access (f | c | o) If defined on a function variable, it denotes "public" access for the function. If it is defined on class variables or object variables, it is indicates Public-Access for all the functions that are defined in the class or the class of the object.

Reference-To-Many/Single (c_1, c_2) Designates both compositional and associative relation between the classes c_1 and c_2.

Restricted-Access(f | c | o) Similar to Public-Access relation defined above, but access is restricted to only a select few other classes or functions.

Return-Type(f,c) Designates that the function return an object of class type c.

Roll-back (State(o), t_i) This is a temporal relation that indicates that the state of the object o is to be reverted back to what it was at the time instant t_i.

State(o) Denotes the state information of an object.

State-Change(o) Denotes the time instant when a change of state information of the object o occurs.

Tree (c | o) Designates the set of objects o, or the set of object instantiations of the class c, in a tree structure.

Tribe (f,c | h) Designates the function set f as a tribe with respect to the set of classes c or Nodes(h). Tribes are set of clans with additional commonality.

Trigger($\Re$,f) Trigger relation designates that the function f be invokes when the ELePUS relation $\Re$ is realized.

APPENDIX B

B.1. ELePUS Specification of Intent Aspect of Patterns

Provided below are the informal specification of *Intent* section of the pattern write-ups from GoF catalog [3] along with their formal specification using ELePUS.

Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

$\exists\ Creators,\ Products \in H,\ \ factory\text{-}method \in 2^F :$

$Define(factory\text{-}method, Creators) \wedge$

$Production(factory\text{-}method, Products)$

Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

$\exists\ Products \in 2^C,\ Builder \in H\ :$

$Creation(Builder, Products) \wedge$

$Decouple(Algorithm(Builder), Products)$

Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

$\exists\ Factory\text{-}Methods \in 2^F,\ Products,\ Creators \in H\ :$

$Production(Factory\text{-}Methods, Products) \wedge$

$Delegate(Instantiation(Product), Root(Creators), Nodes(Creators))$

Singleton

Ensure a class only has one instance, and provide a global point of access to it.

$\exists\ Singleton\ \in C, aSingleton \in o\ :$

$Instantiation(aSingleton, Singleton) \wedge$

$Cardinality(aSingleton, 1) \wedge$

$PublicAccess(aSingleton)$

Adaptor

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

$\exists\ Target, Adaptee \in C\ :$

$Interface\text{–}Compatible(Target, Adaptee)$

Bridge

Decouple an abstraction from its implementation so that the two can vary independently.

$\exists\ Abstractions, Implementations\ \in H\ :$

$Dynamic\text{–}binding(Abstraction, Implementation) \wedge$

$Decouple(Abstraction, Implementation)$

Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and composition of objects uniformly.

$\exists\ Objects\ \in 2^O\ :$

$Tree(Objects) \wedge$

$Equivalence(Interface(Objects), Interface(Tree(Objects)))$

<u>Decorator</u>

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

$\exists\ Objects\ \in C,\ Responsibility\ \in F\ :$

$Dynamic\text{-}Binding(Object, Responsibility)$

<u>Facade</u>

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

$\exists\ Facade\ \in C, Subsystem\ \in 2^C, Requests\ \in 2^F\ :$

$Delegate(Requests, Facade, Subsystem) \wedge$

$Interface\text{-}Compatible(Facade, Subsystem)$

<u>Proxy</u>

Provide a surrogate or placeholder for another object to control access to it.

$\exists\ Realsubject,\ Proxy\ \in C\ :$

$Hold\text{-}Reference(Proxy!, Realsubject) \wedge$

$Interface\text{-}Compatible(Proxy, Subject)$

<u>Chain Of Responsibility</u>

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

$\exists Clients\ \in 2^C, Requests\ \in 2^F, Handler\ \in 2^H\ :$

$Decouple(Clients, Handler) \wedge$

$Delegate(Requests, Handler_i, Handler_j), Handler_i \& Handler_j\ \in Handler \wedge\ i \neq j$

<u>Iterator</u>

Provide a way to access the elements of an aggregate object sequentially without ex-

posing its underlying representation.

$\exists AggregateObject, Object \in C$:

$List(AggregateObject, Object) \wedge$

$Sequential\text{-}Access(AggregateObject) \wedge$

$Abstraction(AggregateObject))$

<u>Memento</u>

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

$\exists Originator \in C$:

$Encapsulation(Originator) \wedge$

$Rollback(Originator, t_i, t_j), t_i \geq t_j$

<u>Observer</u>

Define a one-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

$\exists Subject \in C,\ Observers \in 2^C, Update \in F$:

$Reference\text{-}To\text{-}Many^{\rightarrow}(Subject, Observers) \wedge$

$Define(Update, Observers) \wedge$

$Trigger(State\text{-}Change(Subject), Update)$

<u>Strategy</u>

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

$\exists Strategy \in H$:

$Encapsulation(Strategy) \wedge$

$Decouple(Algorithm(Strategy), Clients)$

B.2. ELePUS Specifications of Applicability Aspect of Patterns

Abstract Factory

$\exists Product\text{-}Family \in 2^H,\ SYSTEM \in Env\ :$

$Couple(Nodes(Product\text{-}Family)) \wedge$

$Decouple(SYSTEM, Creation(Product\text{-}Family))$

Builder

$\exists ProductObject \in O, :$

$Decouple(Algorithm(Creation(ProductObject)), Components(ProductObject)) \wedge$

$Class(ProductObject)$

Factory Method

$\exists Client \in C,\ ProductObject \in O,\ Creators \in H\ :$

$Dynamic\text{-}Binding(Client, ProductObject) \wedge$

$Delegate(Creation(ProductObject), Root(Creators), Nodes(Creators))$

Prototype

$\exists Products \in 2^H;\ Factories \in H\ :$

$Decouple(SYSTEM, Creation(Products)) \wedge$

$NOT(Equivalence(Cardinality(Factory), Cardinality(Products)))$

Singleton

$\exists SingletonObject \in O,\ Singleton \in C\ :$

$Cardinality(Instantiation(Singleton), 1) \wedge$

$Public\text{-}Access(Instantiation(Singleton)) \wedge$

$Abstract(Singleton)$

Adapter

$\exists Client,\ Adapter,\ Adaptee \in C\ :$

$NOT(Interface\text{-}Compatible(Client, Adaptee) \wedge$

$Interface\text{-}Compatible(Client, Adapter) \wedge$

$Interface\text{-}Compatible(Adapter, Adaptee)$

<u>Bridge</u>

$\exists Abstraction \in C,\ Implementation \in C,\ Clients \in 2^C :$

$NOT(Couple(Abstraction, Implementation)) \wedge$

$Abstract(Abstraction) \wedge$

$Inheritance^{+}(Implementation, Abstraction) \wedge$

$Reference\text{-}To\text{-}Single^{\rightarrow}(Clients, Implementation)$

<u>Facade</u>

$\exists Client,\ Facade \in C,\ Subsystems \in 2^{2^C} :$

$Decouple(Client, Subsystems) \wedge$

$Interface\text{-}Compatible(Subsystem, Facade)$

<u>Flyweight</u>

$\exists Objects \in O,\ APPLICATION \in Env :$

$Cardinality(Objects, \infty) \wedge$

$Extract\text{-}State(Objects))$

<u>Chain of Responsibility</u>

$\exists Request \in 2^F,\ Handlers \in 2^H :$

$Clan(Request, Handlers) \wedge$

$Dynamic\text{-}Binding(Request, Handlers)$

<u>Iterator</u>

$\exists ListObject \in 2^O,\ Object \in O :$

$List(ListObject, Object) \wedge$

$Sequential\text{-}Access(ListObject) \wedge$

$Cardinality(Algorithm(Sequential\text{-}Access(ListObject, n))), n > 1 \wedge$

$Equivalence(Interface(ListObject_i), Interface(ListObject_j)),$

$\quad ListObject_i, ListObject_j \in ListObject$

Memento

$\exists Originator \in 2^C, :$

$Extract\text{-}State(Originator) \wedge$

$Encapsulation(Originator) \wedge$

$Rollback(State(Originator), Time_i, Time_j), Time_i \geq Time_j$

Strategy

$\exists Client \in C,\ Algorithm \in 2^F,\ Strategy \in H\ :$

$Clan(Algorithm, Strategy) \wedge$

$Encapsulation(Strategy) \wedge$

$Reference\text{-}To\text{-}Many(Client, Strategy)$

B.3. ELePUS Specification of Collaboration Aspect of Patterns

Generally, specification of collaboration is done along with specification of the structure aspect of a pattern write-up. This section provides only the ELePUS specification of collaboration aspect of patterns. For the LePUS specification of the structure aspect of patterns, refer [10, 22]. However, to increase readability, some relations specific to the structure aspect of specification, which are necessary for specifying collaborations are also given here. Definitions of *clans, tribes* and *Defined-In* relationships are among such reproductions.

Abstract Factory

$\exists Creators \in H,\ Products \in 2^H,\ CreateProduct \in 2^{2^F}\ :$

$Cardinality(Instantiation(Node(Creators), 1)) \wedge$

$tribe(CreateProduct, Products) \wedge$

$Delegate(Production^{\leftrightarrow}(CreateProduct, Products), Root(Creators), Nodes(Creators))$

<u>Builder</u>

$\exists Client,\ Director\ \in C,\ Builder\ \in H,\ Build\text{-}Part\ \in 2^F,\ Construct,\ Get\text{-}Result\ \in F\ :$

$Defined\text{-}In(Build\text{-}Part, Builder) \wedge$

$Defined\text{-}In(Construct, Director) \wedge$

$Defined\text{-}In(Get\text{-}Result, Node(Builder)) \wedge$

$Creation(Client, Director) \wedge$

$Creation(Client, Builder) \wedge$

$Invocation(Build\text{-}Part, Construct) \wedge$

$Invocation(Client, Get\text{-}Result)$

<u>Factory Method</u>

$\exists Creators,\ Products\ \in H,\ factory\text{-}method\ \in 2^F\ :$

$Delegate(Production^{\leftrightarrow}(factory\text{-}method, Products), Root(Creators), Node(Creators))$

<u>Prototype</u>

$\exists Prototype\ \in H,\ Client\ \in C,\ Clone\ \in 2^F\ :$

$clan(Clone, Prototype) \wedge$

$Invocation(Client, clone) \wedge$

$Creation(Clone, Instantiation(Prototype))$

<u>Singleton</u>

$\exists SingletonObject\ \in O,\ Instance\ \in F\ :$

$Returns(Instance!, SingletonObject)$

Adapter

$\exists Client,\ Adapter,\ Adaptee\ \in C,\ request,\ specific\text{-}request\ \in F\ :$

$Defined\text{-}In(Adapter, request) \wedge$

$Defined\text{-}In(Adaptee, specific\text{-}request) \wedge$

$Invocation(Client, request) \wedge$

$Invocation(request, specific\text{-}request)$

Bridge

$\exists Abstraction\ \in C,\ Refined\text{-}Abstraction\ \in 2^C, Implementors\ \in H,\ Operation\, in 2^F,$
$Operation\text{-}Imp\ \in 2^{2^F}\ Clients \in 2^C\ :$

$tribe(Operation, Abstraction) \wedge$

$tribe(Operation\text{-}Imp, Implementors) \wedge$

$Invocation(Clients, Operations) \wedge$

$Invocation(Operation, Operation\text{-}Imp)$

Decorator

$\exists Component\ Decorator\ \in H, Decorator\text{-}Operation,\ Component\text{-}Operation\ \in 2^F\ :$

$clan(Decorator\text{-}Operation, Decorator) \wedge$

$clan(Component\text{-}Operation, Component) \wedge$

$Inheritance(Root(Decorator), Component) \wedge$

$Holds\text{-}Reference(Decorator, Component) \wedge$

$Forwarding(Decorator\text{-}Operation, Component\text{-}Operation)$

Facade

$\exists Implementations,\ Requests\ \in 2^F,\ Subsystem\text{-}Classes\ \in 2^C,\ Facade\, Clients\ \in C\ :$

$clan(Implementations, Subsystem\text{-}Classes) \wedge$

$tribe(Requests, facade) \wedge$

$Invocation(Clients, Requests) \wedge$

$Invocation^{\rightarrow}(Requests!, Implementations)$

Flyweight

$\exists Flyweight\text{-}Factory \in C,\ Flyweight \in H$:

$Creation(Flyweight\text{-}Factory!, Node(Flyweight))$

Proxy

$\exists Proxy,\ Real\text{-}Subject,\ Subject \in C,\ Request \in 2^F$:

$Inheritance(Real\text{-}Subject, Subject) \wedge$

$Inheritance(Proxy, Subject) \wedge$

$clan(Request, Subject) \wedge$

$Forwarding(Request_{Proxy}!, Request_{RealSubject})$

Chain of Responsibility

$\exists Client \in C;\ Handle\text{-}Request \in 2^F;\ Handler \in H$:

$Clan(Handle\text{-}Request, Handler) \wedge$

$List(Handler) \wedge$

$Invocation(Client, Handle\text{-}Request) \wedge$

$Delegate^{*}(Handle\text{-}Request, ElNode_i(Handler), Node_j(Handler),$

$Handle(Node_j(Handler)))$

Command

$\exists Command \in H;\ Client,\ Invoker,\ Receiver \in C; Execute \in 2^F; Action \in F$:

$Creation(aClient, Instantiation(Node(Command))) \wedge$

$Clan(Execute, Command) \wedge$

$Defined\text{-}In(Action, Receiver) \wedge$

$Hold\text{-}Reference(anInvoker, Node(Command)) \wedge$

$Invocation(anInvoker, Execute) \wedge$

$Invocation(Execute, Action)$

Memento

$\exists Originator,\ Memento,\ CareTaker\ \in C;$

$SetMemento, CreateMemento, GetState, SetState\ \in F\ :$

$Defined\text{-}In(SetMemento, Originator) \wedge$

$Defined\text{-}In(CreateMemento, Originator) \wedge$

$Defined\text{-}In(GetState, Memento) \wedge$

$Defined\text{-}In(SetState, Memento) \wedge$

$Argument\text{-}1(SetMemento, Memento) \wedge$

$Return\text{-}Type(CreateMemento, Memento) \wedge$

$Invocation^{\rightarrow}(CareTaker, CreateMemento) \wedge$

$Production(CreateMemento, Memento) \wedge$

$Invocation^{\rightarrow}(CreateMemento, SetState) \wedge$

$Invocation^{\rightarrow}(CareTaker, SetMemento) \wedge$

$Invocation^{\rightarrow}(SetMemento, GetState)$

Observer

$\exists subject\ concrete\text{-}Subject\ \in C;\ Observer\ \in H;\ Set\text{-}State, Update\ \in 2^F;$

$attach, detach, notify, get\text{-}state\ \in F\ :$

$clan(Update, Observer) \wedge$

$clan(attach, Subject) \wedge$

$clan(detach, Subject) \wedge$

$clan(notify, Subject) \wedge$

$clan(get\text{-}state, concrete\text{-}subject) \wedge$

$tribe(Set\text{-}State, concrete\text{-}subject) \wedge$

$Invocation^{\rightarrow}(Node(Observer), Set\text{-}State) \wedge$

$Invocation^{\rightarrow}(Set\text{-}State, notify) \wedge$

$Invocation^{\rightarrow}(notify, Update) \wedge$

$Invocation^{\rightarrow}(Update, get\text{-}state)$

<u>Visitor</u>

$\exists Visit \in 2^{2^F}$; $accept$, $Operation \in 2^F$; $Visitors$, $Elements \in H$; $ObjectStructure \in C$:

$tribe(Visit, Visitors) \wedge$

$clan(accept, Elements) \wedge$

$Argument\text{-}1^{\rightarrow H}(accept, Visitors) \wedge$

$Argument\text{-}1^{\leftrightarrow}(Visit, Elements) \wedge$

$Defined\text{-}In(Operation, Node(Elements)) \wedge$

$Creation(Client, Node(Visitors)) \wedge$

$Invocation(ObjectStructure, accept) \wedge$

$Invocation^{\leftrightarrow}(accept, Visit) \wedge$

$Invocation^{\leftrightarrow}(Visit, Operation)$

LIST OF REFERENCES

LIST OF REFERENCES

[1] Mark Grand, "Patterns in Java: Volume 1," *Addison-Wesley*, 1998.

[2] Doug Lea, "Concurrent Programming in Java - Design Principles and Patterns," *Addison Wesley*, 1997.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Patterns: Elements of Reusable Object Oriented Software," *Addison-Wesley*, 1995.

[4] Tommi Mikkonen, "Formalizing Design Patterns," *Proceedings of The International Conference on Software Engineering,ICSE'98, IEEE Computer Society Press. pp. 115-124*, 1998.

[5] Frank Buschmann, Regine Meunier, Hans Rohernert, Peter Sommerlad, and Michael Stal, "A System of Patterns," *John Wiley & Sons*, 1996.

[6] James O. Coplien and Doughlas C. Schmidt, "Pattern Languages in Program Design," *Addison-Wesley Publications*, 1995.

[7] G. Booch, J. Rumbaugh, and I. Jacobson, "UML: A Unified Modeling Language, version 1.0.," *http://www.rational.com/uml*, 1997.

[8] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson, "Object Oriented Modeling and Design," *Prentice Hall*, 1991.

[9] Amnon H. Eden, "Giving 'The Quality' a Name," *Journal of Object Oriented Programming, guest column, SIGS Publications*, June 1998.

[10] Amnon H. Eden, Joseph Gil, Yoram Hirshfeld, and Amiram Yehudai, "Motifs in Object Oriented Architecture," *Submitted: IEEE Transactions on Software Engineering. Available at: http://www.csd.uu.se/~eden/lepus/*, 1999.

[11] Ellen Agerbo and Aino Cornils, "How to Preserve the benefits of Design Patterns," *Proceedings of OOPSLA'98*, 1998.

[12] Sivakumar Chinnasamy, Rajeev R. Raje, and Zhiqing Liu, "Specification of Design Patterns: An Analysis," *Proceedings of The 7th International Conference on Advanced Computing And Communications (ADCOM'99)*, 1999.

[13] S. Ducasse, "Message Passing Abstractions as Elementary Bricks for Design Pattern Implementation," *Object-Oriented Technology (ECOOP'97 Workshop reader), Jan-Bosch and Stuart Mitchell (Ed.), LNCS 1357, Springer-Verlag, pp. 96-99* , June 1997.

[14] J. Bosch, "Design patterns as language constructs," *Journal of Object Oriented Programming*, 1997.

[15] J. Bosch, "Language support for design patterns," *In proceedings of TOOLS'96*, pp. 197-210,1996.

[16] J. Bosch, "Design patterns and frameworks: On the issue of language support," *Object-Oriented Technology (ECOOP'97 workshop reader), Jan Bosch and Stuart Mitchell (Ed.), LNCS 1357, Springer-Verlag*, 1997.

[17] Amnon H. Eden, Joseph Gil, Yoram Hirshfeld, and Amiram Yehudai, "Towards a Mathematical foundation for Design Patterns," *Technical Report 1999-004, Department of Information Technology, Uppsala University*, 1999.

[18] Amnon H. Eden, Yoram Hirshfeld, and Amiram Yehudai, "Precise Notation for Design Patterns," *http://www.math.tau.ac.il/~eden/bibliography.html*, 1998.

[19] On-line Discussion Group, "Mailing List for discussion of the theory of patterns in general," *http://hillside.net/patterns/List.html#patterns-discussion*, 1999.

[20] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fixdahl-King, and S. Angel, "A Pattern Language," *Oxford University Press, New York*, 1977.

[21] P. E. Boas, "Resistance is Futile; Formal Linguistic Observations on Design Patterns," *Technical Report CT-1997-03, The Institute For Logic, Language, and Computation (ILLC)*, University of Amsterdam, 1997.

[22] Amnon H. Eden, "Precise Specification of Design Patterns and Tool support in Their Application," *Ph.D Thesis, Department of Computer Science, Tel Aviv University, Tel Aviv Israel*, 1999.

[23] E. Bunke and E. Foxley, "Logic and its applications," *Prentice-Hall Inc.*, 1996.

[24] R. Duke, G. Rose, and G. Smith, "A Specification Language advocated for description of standards," *Technical Report 94-45, SVRC, School of Information Technology, The University of Queensland*, 1994.

[25] Guttag, J., et al., (Ed.), "Larch: Languages and tools for formal specification," *Texts and monographs in Computer Science*, 1993.

[26] E. H. Durr and J. V. Katwijk, "A formal specification language for object-oriented designs," *Proceedings of Technology of Object-oriented Languages and Systems, pp. 63-78, Prentice-Hall Inc.*, 1992.

[27] Joseph R. Kiniry, "Leading to a Kind Description Language: Thoughts on Component Specification," *COOTS'99 Advanced Topics Workshop on Validating the Composition/Execution of Component-Based Systems*, 1999.

[28] W. F. Tichy, "Software Development Control Based on System Structure Description," *Ph.D. Thesis, Carnegie-Mellon University, Computer Science Department*, 1980.

[29] D. C. Luckham and J. Vera, "An Event Based Architecture Definition Language," *IEEE Transactions on Software Engineering, Vol.21, No.9, pp. 717-734*, 1995.

[30] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay, "Contracts: Specifying Compositions in Object Oriented Systems," *OOPSLA '90, SIGPLAN Notices, Vol. 25 No. 10*, 1990.

[31] H. M.Järvinen and R. K. Suonio, "DisCo specification language: marriage of actions and objects," *Proceedings of the 11th International Conference on Distributed Computing Systems, pp. 142-151, IEEE Computer Society Press*, 1991.

[32] Anthony Lauder and Stuart Kent, "Precise Visual specification of Design Patterns," *Procs. ECOOP'98*, 1998.

[33] Stuart Kent, Ali Hamie, John Howse, Franco Civello, and Richard Mitchell, "Semantics through Pictures," *ECOOP'97 workshop Reader, LNCS, Springer Verlag*, 1997.

[34] J. Barwise, "Handbook of Mathematical Logic," *pp. 4-56*, 1976.

[35] Nils Klarlund, Jari Koistinen, and Michael I. Schwatzbach, "Formal Design Constraints," *Proceedings of OOPSLA'96*, 1996.

[36] Sivakumar Chinnasamy and Rajeev R. Raje, "Specification of Design Patterns: A Survey, Evaluation and Suggestions," *Submitted to IEEE Transactions on Software Engineering, Ed. Richard Kemmerer, IEEE Press*, 2000.

[37] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems 16, 3, pp. 872-923*, May, 1994.

[38] Sherif M. Yacoub and Hany. H. Ammar, "Pattern-Oriented Analysis and Design (POAD): A Structural Composition Approach to Glue Design Patterns," *Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems, TOOLS USA 2000*, July-August 2000.

[39] Sherif M. Yacoub, "Pattern-Oriented Analysis and Design," *Ph.D Dissertation, West Virginia University*, Dec 1999.