



Master 2 Recherche - *Informatique et Télécommunications*

Spécialité *Systèmes Informatiques et Génie Logiciel - SIGL*

2007 - 2008

Ontologies et Mauvaises Pratiques de Conception

Dania HARB

Sous la direction de :

M. Hervé LEBLANC, Maître de Conférences, UPS

M. Cédric BOUHOURS, doctorant, IRIT

Mots Clés: Ontologies, OWL, Patrons de conceptions, Anti-pattern

Résumé: En génie logiciel, le savoir-faire informel dénotant une pratique sociale de référence est généralement édicté sous la forme de catalogue de patrons, du codage à l'architecture en passant par la conception. Afin de les formaliser et de les utiliser, des ontologies ayant pour objet les patrons de conception ont été conçues. De manière à ce qu'elle soit partageable via le web, elles ont été implémentées en utilisant le langage OWL et elles peuvent être interrogées en utilisant différents langages de requêtes outils dépendants. Notre travail a consisté à étendre une ontologie dédiée à l'intention des patrons de conception en lui adjoignant des concepts de modèles alternatifs et de points forts faisant référence aux anti-patterns. Nous avons validé cette approche en outillant une étape d'une activité baptisée revue de design, en référence à la revue de code. Cette activité, nécessaire à l'amélioration de la qualité des architectures à objets et dirigée par les patrons de conception peut trouver naturellement sa place dans un processus de développement à base de modèles.

SOMMAIRE

1	Introduction.....	5
2	Ontologies, OWL and Software Engineering	8
2.1	Knowledge representation.....	9
2.2	Ontologies	10
2.3	OWL and Description Logic.....	12
2.4	Semantic Web Technologies in Software Engineering	15
3	Ontology Development Tool – Protégé.....	18
3.1	Ontology Classes.....	18
3.2	Ontology Properties	19
3.3	SPARQL panel.....	20
4	DPIO ontology: the Design Pattern Intent Ontology.....	22
4.1	Design Patterns.....	22
4.1.1	Definition	22
4.1.2	Classification	22
4.1.3	Description.....	23
4.2	The core structure of the DPIO	27
4.3	Extracting knowledge.....	30
5	Alternative models and Design Patterns’ strong points	34
6	Extending the DPIO ontology	36
6.1	Conception and Implementation	36
6.2	Queries using SPARQL	42
7	Conclusion	46

Remerciements

Je tiens à remercier sincèrement M. Hervé LEBLANC, Maître de Conférences, pour me donner l'opportunité de réaliser ce stage, pour sa confiance et la liberté qu'il m'a accordée et pour tous ses précieux conseils et sa grande disponibilité tout au long de ce stage.

Je remercie également M. Christian PERCEBOIS, Professeur et Responsable de l'équipe MACAO pour m'avoir accueilli au sein de son équipe à l'IRIT.

Je tiens à exprimer mes sincères remerciements à Madame Nathalie AUSSENAC-GILLES, Chargé de Recherche et Responsable de l'équipe IC3 à l'IRIT pour sa collaboration et ses précieux conseils.

Je remercie également M. Thierry MILLAN, Maître de Conférences, pour son support dans les problèmes techniques qui ont pu survenir pendant ce stage.

Je tiens à exprimer ma plus vive gratitude à M. Cédric BOUHOURS, Doctorant dans l'équipe MACAO pour sa collaboration et la confiance qu'il m'a témoignée en mettant à ma disposition son étude de thèse, et c'est grâce à cette étude que ce stage a vu le jour.

Je remercie également tous ceux qui, de près ou de loin, m'ont soutenu moralement et matériellement pour l'achèvement de ce stage. Je ne citerai de noms de peur d'en oublier.

A tous merci

Enfin, je tiens à rendre un hommage chaleureux à mes parents, mes sœurs et mon frère au Liban pour leur sacrifice, leur amour et leur confiance. Une fière chandelle à mon fiancé qui m'a sans cesse apporté le soutien moral durant toute cette année.

Illustrations

Figure 1: Design review activity	6
Figure 2: Protégé screenshot on creating OWL classes	19
Figure 3: Protégé screenshot on creating OWL properties	20
Figure 4: Protégé screenshot on interrogating the ontology using SPARQL	21
Figure 5: Implementation of a Graphic Application	24
Figure 6: Structure of the Composite Design Pattern	25
Figure 7: Graphical overview of the core structure of the DPIO	27
Figure 8: Instantiation of the DPIO UML design	28
Figure 9: Screenshot of the dialogues constraining problem predicates and problem concepts.....	31
Figure 10: Screenshot of the result window suggesting design patterns for the problem of varying an algorithm	32
Figure 11: Protégé screenshot of the result window suggesting design patterns for the problem of composing an object	33
Figure 12: Generalization of the Composite alternative model and its structural features	35
Figure 13: Graphical overview of the structure of the extended ontology.....	37
Figure 14: Structure of the Composite Design pattern.....	39
Figure 15: Structure of the Alternative Model class hierarchy	40
Figure 16: Structure of the Strong Point class hierarchy	40
Figure 17: Structure of the alternative model 5 of the Composite Design pattern.....	41
Figure 18: Instantiation of the UML model of the extended ontology	42
Figure 19: Screenshot of the result window suggesting suitable design pattern and its intent.....	44
Figure 20: Screenshot of the result window presenting the strong points perturbed	45

1 Introduction

La communauté scientifique IDM (Ingénierie Dirigée par les Modèles) proclamant le caractère productif des modèles a proposé un nouveau processus de développement appelé model-driven process. Cependant, afin d'obtenir une valeur ajoutée certaine aux modèles en cours d'élaboration, ces processus se doivent de promouvoir la réutilisation d'expertise généralement exprimée sous forme de patrons d'analyse (Martin Fowler, 1997), de conception (Gamma et al., 1995) ou d'architecture (Buschmann, 1996) approuvés par une communauté d'experts. L'activité dans laquelle nous devons intervenir se situe au temps de la conception d'un système, et à ce temps nous allons nous intéresser spécifiquement aux patrons de conception. En référence aux activités de revue de code, cette activité se nomme revue de conception et elle a pour but d'augmenter la qualité des architectures à objets par l'utilisation massive de patrons de conception. Celle-ci consiste à retrouver dans des modèles industriels des fragments substituables par un patron de manière automatique et d'opérer la substitution lorsque les intentions du patron et du fragment concordent, le tout avec l'accord préalable d'un expert ou du concepteur.

Cette activité, décrite dans la Figure 1 peut être décomposée en quatre étapes.

De façon à pouvoir rechercher des fragments dans des modèles industriels correctement conçus, la première étape s'assure d'un minimum de respect des règles de conception à objets. Le respect de ces règles est assuré par un interprète OCL et la plate-forme d'exécution Neptune. Ces règles sont inspirées des principes de conception de Bertrand Meyer et des GRASP patterns de Graig Larman.

Lorsque le modèle à analyser n'invalide aucune des ces règles, il est alors autorisé à passer la deuxième étape. Celle-ci consiste à rechercher des fragments de modèles ayant exactement les mêmes particularités structurelles et comportementales que celles des modèles alternatifs aux patrons qui sont de fait un catalogue de mauvaises pratiques de conception attachés à l'intention de chaque patron de conception. Ces fragments de modèles deviennent alors substituables par le patron de conception qu'ils remplacent dans la conception courante. Cette détection est réalisée de manière automatique par un algorithme générique de recherche de motifs exacts acceptant la richesse sémantique des graphes d'instances UML. De plus, des familles de fragments de modèles peuvent être retrouvées par un mécanisme de points d'extension implémenté dans l'algorithme. Cette technique peut alors s'apparenter à une technique de redocumentation à distinguer de la retro-conception et va permettre d'enclencher d'éventuelles opérations de restructuration (des refactorings).

Chaque fragment de modèle trouvé est un candidat potentiel à une substitution ou un nouvel appariement. Mais, les particularités structurelles n'infèrent en aucune manière de l'intention du concepteur de ce modèle. Une étape de médiation s'avère alors nécessaire avec le concepteur et éventuellement un expert en architecture à objets. Il faut alors s'assurer que l'intention du fragment de modèle correspond à

l'intention du patron que l'on veut injecter et quel jeu en vaille la chandelle, c'est à dire que l'effort de restructuration demandé améliore de façon significative une ou plusieurs qualités logicielles attendues de ce fragment de modèle.

Enfin, si la médiation a réussi, la dernière étape consiste à l'intégration des différentes propositions au sein du modèle considéré.

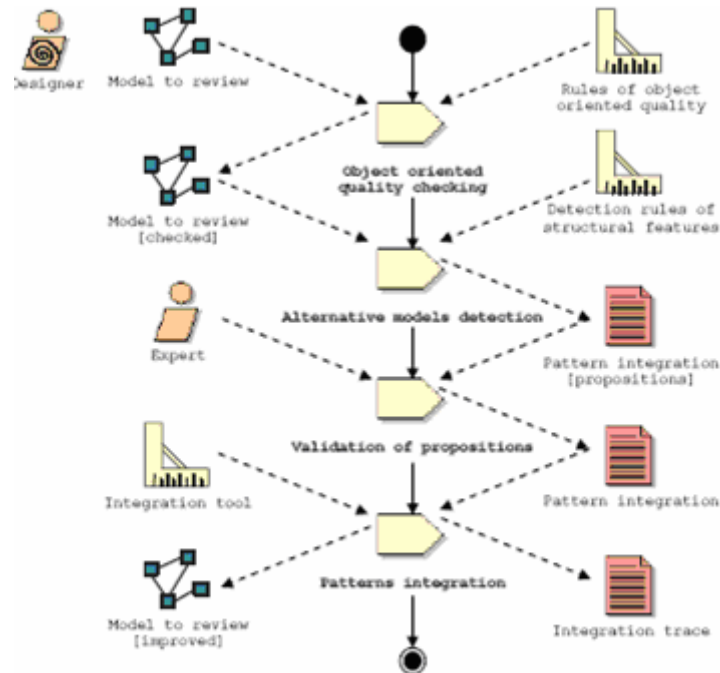


Figure 1: Design review activity

La contribution proposée dans le cadre de ce stage va être alors d'étudier et de valider une approche à base d'ontologies pour outiller la troisième étape de cette activité.

Plus précisément, l'idée développée en [1] est d'interroger une base de connaissances au format OWL pour aider les concepteurs à choisir le meilleur patron à instancier lorsqu'ils sont en présence de problèmes de conceptions. Cette base de connaissances est structurée en « problèmes de conception » eux-mêmes décomposés en « problèmes atomiques de conception ». Elle est suggérée par l'item « Intention » du catalogue des patrons de conception [2], et utilisé par un système interactif de question-réponse durant la phase de conception. L'idée de ce travail est de réappliquer cette stratégie plus tard dans le cycle de développement, en utilisant une base de mauvaises pratiques de conception [3]. Cette base est constituée de modèles alternatifs au patron de conception, et est destinée à être utilisée pour vérifier si des patrons n'ont pas été oubliés dans les conceptions. Un modèle alternatif résout un problème soluble par un patron avec une structure plus complexe ou différente que le patron, et est destiné à être remplacé par un patron. La recherche de ces modèles alternatifs est

uniquement guidée par des particularités structurelles, et est de fait insuffisante pour garantir la pertinence des substitutions proposées. Le but de ce travail est de proposer et de valider un schéma de base de connaissances compatible avec des logiques de description, afin notamment :

- D'aider au filtrage des modèles alternatifs issus de l'analyse des modèles.
- D'aider à la structuration de la base de mauvaises pratiques de conception en utilisant des mécanismes de classification des logiques terminologiques.

Une collaboration avec l'équipe IC3 a été entreprise, pour une aide sur la formalisation et les outils basés sur les logiques terminologiques.

Le plan de ce rapport se décompose alors comme suit : tout d'abord nous introduisons les concepts d'ontologies et de langages d'ontologies, nous nous intéresserons plus particulièrement à OWL, langage retenu pour ce travail. Ensuite nous enchaînerons sur la présentation de l'outil utilisé pour nos expérimentations, l'outil protégé développé par l'université de Stanford. Puis nous étudierons en détail la base de connaissances de départ mis à disposition par les auteurs de [2]. Nous présenterons les connaissances supplémentaires à intégrer dans cette base, ces connaissances émanent de notre équipe d'accueil lors de ce stage [3]. Et de fait, nous présenterons l'apport de notre étude et la validation de l'approche considérée pour l'outillage de l'activité de revue de conception. Enfin, nous concluons et donnerons des perspectives à notre travail.

2 Ontologies, OWL and Software Engineering

*Dans cette section, nous présentons les concepts afférents aux ontologies et plus particulièrement celles publiées sur le Web. Puis nous présentons le langage d'ontologie choisi pour notre étude : *Ontology Web Language*. Enfin, nous esquissons un état de l'art sur l'utilisation du web sémantique dans des problématiques « génie logiciel ».*

Tout d'abord, nous introduisons les concepts de web sémantique et la hiérarchie des langages utilisés basés sur XML. Elle se compose par couches successives de RDF : langage décrivant des ressources et des relations, RDFS : langage décrivant la structure d'un document RDF et OWL : prenant en compte les fonctionnalités nécessaires afin que l'on puisse parler d'un langage d'ontologies.

Ensuite, nous présentons les concepts d'ontologies comme un ensemble de concepts reliés par un ensemble de rôles et des possibilités de raisonnement suivant la logique de description choisie. Nous présentons leur utilité en répondant à la question quand doit-on construire une base de connaissances ? Nous présentons une démarche de développement que nous avons respecté dans notre étude.

Puis, nous nous intéressons au langage OWL et précisons ce qu'il apporte en plus des langages existants pour représenter et raisonner sur des ontologies. Suivant la précision et la consistance des données que nous voulons manipuler, il nous faut choisir entre les différents niveaux de consistance que ce langage propose. OWL lite ne permet que de représenter des données, OWL DL permet de raisonner sur les ontologies et de valider la cohérence des données, OWL full a une expressivité maximale sans aucune contrainte sur la description des données. Nous avons choisi OWL DL pour les possibilités de raisonnement et la cohérence automatique de la structuration des données. En effet, nous ne sommes pas spécialistes de ces langages et voulions nous assurer de la validité de l'extension proposée à une base de connaissance existante.

Enfin, nous proposons un début d'état de l'art sur l'utilisation des ontologies du web pour le génie logiciel. L'utilisation principale est la structuration d'un ensemble de connaissances « informelles ». Ces connaissances peuvent être de type terminologique : un vocabulaire commun, qu'il soit générique ou spécifique à un domaine est partagé par des équipes de développement distantes. Ces connaissances peuvent aider à retrouver des informations utiles à un savoir qualifié de « pratique sociales de références » émanant d'une communauté de développeur, de concepteurs ou de managers. C'est typiquement le cas des patrons de manière générale et des patrons de conception qui nous intéressons dans cette étude. Avant d'introduire l'ontologie que nous avons étudiée et étendue pour nos propres besoins, nous présentons les premières tentatives de formalisation des patrons de conception par l'utilisation de logiques de descriptions.

The World Wide Web is the main source to get information; it is aimed to be understandable by human only. It is syntax-based; the content of its documents and resources cannot be treated by machines.

The power of the Semantic Web [4], as the next generation of the Web, will be realized when software agents are able to understand the Web content, process the information and exchange the results with other software agents. It aims to represent semantically its content, link them in such a way that they can be easily processable by machines not just to display information or data, but also for automation, integration, reuse of data and share content across various applications and software agents. According to the co-inventor of the World Wide Web, Tim Berners-Lee, the Web Semantic can be seen as a vision of the Web of tomorrow, a large space of resources exchanges between human being and machines to exploit a huge volume of information. Most of the Web's content today is designed for humans to read, not for computer programs to manipulate meaningfully.

Informational Retrieving depends on syntaxes, retrieves the pertinent pieces though the last phase will be kept to humans to interpret, and decide the relevant fragments that could really help. So because computers have no reliable way to process the semantics, and in order to provide more help to people, the Semantic Web came to bring structure to the meaningful content of Web pages. It extends the current Web with formalized knowledge and data that can be processed by computers.

2.1 Knowledge representation

Adding logic to the Web is one of the key requirements to express this knowledge. It will become necessary to construct a powerful logical language on which rules will be adapted on to make inferences and describe the content of information sources. The challenge of the Semantic Web, therefore, is to provide a language that expresses both data and rules for reasoning about the data and that allows rules from any existing knowledge-representation system to be exported onto the Web [4].

To describe complex properties of web resources but not so complicated, researchers attempt to adopt the layered approach [5], where the upper layer is extended from the lower layer with enhanced expressive power. This allows that different applications can choose the logic language suiting their needs most.

W3C has proposed a series of technologies that can be applied to achieve this vision. The Semantic Web extends the current Web by giving the web content a well-defined meaning, better enabling computers and people to work in cooperation:

XML is aimed at delivering data to systems that can understand and interpret the information. XML is focused on the syntax (defined by the XML schema or DTD) of a document and it provides essentially a mechanism to declare and use simple data structures. However there is no way for a program to actually understand the knowledge contained in the XML documents.

Resource Description Framework (RDF) [6] is a foundation for processing metadata; it provides interoperability between applications that exchange machine-

understandable information on the Web. RDF uses XML to exchange descriptions of Web resources and emphasizes facilities to enable automated processing. The RDF descriptions provide a simple ontology system to support the exchange of knowledge and semantic information on the Web. It is built on the triple, a 3-tuple consisting of subject, predicate and object. The subject denotes the resource; the object represents a data or another resource and the predicate is a resource as well, applicable on the subject, represents a relationship, a way to link the subject to the object. Subjects and objects are represented either by a *Uniform Resource Identifier* (URI) or by a *blank node* while the predicates are always identified by URIs.

RDF Schema [7] provides the basic vocabulary to describe RDF documents. RDF Schema can be used to define properties and types of the web resources. The advent of RDF Schema represented an early attempt at a SW ontology language based on RDF.

OWL, The Web Ontology Language, [8] is a standard (W3C recommendation) for expressing ontologies in the Semantic Web. The OWL language facilitates greater machine understandability of Web resources than that supported by RDFS by providing additional constructors for building class and property descriptions (vocabulary) and new axioms (constraints), along with a formal semantics.

URIs are compact strings of characters used to define names or resources that can interact over the network, typically the World Wide Web, allowing anyone to define their own concepts and verbs by defining a URI for them somewhere in the web, where they can be re-used by everyone. As a result, different identifiers will be used to express the same concept. So when users need to combine the information of two databases having same concepts expressed in different ways, there should be a way to show this relation. A solution to this problem is provided by collections of information called ontologies.

2.2 Ontologies

An **ontology** is a formal explicit description of concepts in a domain of discourse (**classes** (sometimes called **concepts**)), properties of each concept describing various features and attributes of the concept (**slots** (sometimes called **roles** or **properties**)), and restrictions on slots (**facets** (sometimes called **role restrictions**)). An ontology together with a set of individual **instances** of classes constitutes a **knowledge base**. In reality, there is a fine line where the ontology ends and the knowledge base begins [9].

It's a formal representation of a set of concepts within a domain and the relationships between those concepts. It is used to reason about the properties of that domain, and may be used to define the domain. It defines a common vocabulary for researchers who need to share information in a domain including machine-interpretable definitions of basic concepts in the domain and relations among them.

Why would someone want to develop an ontology? [9] Some of the reasons are:

- To share common understanding of the structure of information among people or software agents
- To enable reuse of domain knowledge
- To make domain assumptions explicit
- To separate domain knowledge from the operational knowledge
- To analyze domain knowledge

Sharing common understanding of the structure of information among people or software agents is one of the more common goals in developing ontologies (Musen 1992; Gruber 1993). For example, suppose several different Web sites contain medical information or provide medical e-commerce services. If these Web sites share and publish the same underlying ontology of the terms they all use, then computer agents can extract and aggregate information from these different sites. The agents can use this aggregated information to answer user queries or as input data to other applications.

Enabling reuse of domain knowledge was one of the driving forces behind recent surge in ontology research. For example, models for many different domains need to represent the notion of time. This representation includes the notions of time intervals, points in time, relative measures of time, and so on. If one group of researchers develops such an ontology in detail, others can simply reuse it for their domains. Additionally, if we need to build a large ontology, we can integrate several existing ontologies describing portions of the large domain. We can also reuse a general ontology, such as the UNSPSC ontology, and extend it to describe our domain of interest.

Making explicit domain assumptions underlying an implementation makes it possible to change these assumptions easily if our knowledge about the domain changes. Hard-coding assumptions about the world in programming-language code make these assumptions not only hard to find and understand but also hard to change, in particular for someone without programming expertise. In addition, explicit specifications of domain knowledge are useful for new users who must learn what terms in the domain mean.

Separating the domain knowledge from the operational knowledge is another common use of ontologies. We can describe a task of configuring a product from its components according to a required specification and implement a program that does this configuration independent of the products and components themselves (McGuinness and Wright 1998). We can then develop an ontology of PC-components and characteristics and apply the algorithm to configure made-to-order PCs. We can also use the same algorithm to configure elevators if we “feed” elevator component ontology to it (Rothenfluh et al. 1996).

Analyzing domain knowledge is possible once a declarative specification of the terms is available. Formal analysis of terms is extremely valuable when both attempting to reuse existing ontologies and extending them (McGuinness et al. 2000).

Developing an ontology includes:

- Defining classes in the ontology: concepts, properties
- Arranging the classes in a subclass/superclass hierarchy
- Defining slots and describing allowed values for these slots
- Filling in the values for slots for instances.

We can then create a knowledge base by defining individual instances of these classes filling in specific slot value information and additional slot restrictions.

Many questions need to be answered when developing an ontology in order to define its domain:

1. What is the domain that the ontology will cover?
2. For what are we going to use the ontology?
3. For what types of questions the information in the ontology should provide answers?
4. Who will use and maintain the ontology?

We will need to answer these questions while developing our own ontology, which will be discussed later.

2.3 OWL and Description Logic

OWL, the Web Ontology Language recommended by the World Wide Consortium (W3C), is a semantic markup language for publishing and sharing ontologies on the World Wide Web. It's intended to provide a language that can be used to describe the classes and relations between them that are inherent in Web documents and applications.

The OWL Web Ontology Language is designed for use by applications that need to process the content of information instead of just presenting information to humans. OWL, developed as a vocabulary extension of RDF, facilitates greater machine interpretability of Web content than that supported by XML, RDF, and RDF Schema (RDF-S) by providing additional vocabulary along with a formal semantics.

OWL has been designed to meet this need for a Web Ontology Language. OWL is part of the growing stack of W3C recommendations related to the Semantic Web.

- XML provides a surface syntax for structured documents, but imposes no semantic constraints on the meaning of these documents.
- XML Schema is a language for restricting the structure of XML documents and also extends XML with datatypes.
- RDF is a datamodel for objects ("resources") and relations between them, provides a simple semantics for this datamodel, and these datamodels can be represented in an XML syntax.
- RDF Schema is a vocabulary for describing properties and classes of RDF resources, with a semantics for generalization-hierarchies of such properties and classes.
- OWL adds more vocabulary for describing properties and classes: among others, relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes.

Owl ontology consists of Individuals, Properties, and Classes.

- Individuals represent objects in the domain of interest. Individuals are also known as instances. It can be referred to as being instances of classes or concepts.
- Properties are relationships between two things i.e. a concept/individual links to a concept/individual known as object property or a concept/individual link to an XML schema datatype value or an rdf literal known as datatype property. . Properties can be limited to having a single value; to being functional or multiple values i.e. to being non-functional. Also, they can be either transitive or symmetric. Properties are also known as roles in description logics, slots in Protégé, and attributes in UML and other object-oriented notions.
- Classes are a concrete representation of concepts interpreted as sets that contain individual(s). Individuals may belong to more than one class. Classes may be constructed in a superclass-subclass hierarchy, which is also known as taxonomy. Subclasses are subsumed by their superclasses.

Thus, Classes are interpreted as sets of objects that represent the individuals in the domain of discourse. Properties are binary relations that link individuals, and are interpreted as sets of tuples, which are the subsets of the cross product of the objects in the domain of discourse.

OWL is heavily influenced by Description Logic (DL) research; DL is a family of knowledge representation languages which can be used to represent the terminological knowledge of an application domain in a structured and formally well-understood way. The name *description logic* refers, on the one hand, to concept descriptions used to describe a domain and, on the other hand, to the logic-based semantics which can be given by a translation into first-order predicate logic. It refers to a logic that focuses on descriptions as its principal means for expressing logical

expressions. A description logic system emphasizes the use of classification and subsumption reasoning as its primary mode of inference.

OWL ontologies may be categorised into three species or sub-languages: OWL-Lite, OWL-DL and OWL-Full. A defining feature of each sub-language is its expressiveness. OWL-Lite is the least expressive sub-language. OWL-Full is the most expressive sub-language. The expressiveness of OWL-DL falls between that of OWL-Lite and OWL-Full. OWL-DL may be considered as an extension of OWL-Lite and OWL-Full an extension of OWL-DL.

OWL Lite

OWL-Lite is the syntactically simplest sub-language. It is intended to be used in situations where only a simple class hierarchy and simple constraints are needed. For example, while it supports cardinality constraints, it only permits cardinality values of 0 or 1. It should be simpler to provide tool support for OWL Lite than its more expressive relatives, and OWL Lite provides a quick migration path for thesauri and other taxonomies. Owl Lite also has a lower formal complexity than OWL DL, see the section on OWL Lite in the OWL Reference for further details.

OWL DL

OWL-DL is much more expressive than OWL-Lite and is based on Description Logics (hence the suffix DL). Description Logics are a decidable fragment of First Order Logic² and are therefore amenable to automated reasoning. It is therefore possible to automatically compute the classification.

Our work focuses on OWL-DL: OWL DL includes all OWL language constructs, but they can be used only under certain restrictions (for example, while a class may be a subclass of many classes, a class cannot be an instance of another class). OWL DL is so named due to its correspondence with description logics, a field of research that has studied the logics that form the formal foundation of OWL.

OWL Full

OWL-Full is the most expressive OWL sub-language. It is intended to be used in situations where very high expressiveness is more important than being able to guarantee the decidability or computational completeness of the language. It is therefore not possible to perform automated reasoning on OWL-Full ontologies. It is meant for users who want maximum expressiveness and the syntactic freedom of RDF with no computational guarantees. For example, in OWL Full a class can be treated simultaneously as a collection of individuals and as an individual in its own right. OWL Full allows an ontology to augment the meaning of the pre-defined (RDF or OWL) vocabulary. It is unlikely that any reasoning software will be able to support complete reasoning for every feature of OWL Full.

There are some simple rules to choose the appropriate sub language to use:

- The choice between OWL-Lite and OWL-DL may be based upon whether the simple constructs of OWL-Lite are sufficient or not.
- The choice between OWL-DL and OWL-Full may be based upon whether it is important to be able to carry out automated reasoning on the ontology or whether it is important to be able to use highly expressive and powerful modeling facilities such as meta-classes (classes of classes).

The Protégé-OWL plugin does not make the distinction between editing OWL-Lite and OWL-DL ontologies. It does however offer the option to constrain the ontology being edited to OWL-DL, or allow the expressiveness of OWL-Full.

2.4 Semantic Web Technologies in Software Engineering

Over the years, the software engineering community has developed various tools to support the specification, development, and maintenance of software. However, the Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries.

Ontologies are used to capture knowledge in some domains of interest. Ontology describes the concepts in the domain and also the relationships that hold among those concepts and as such provide the formal vocabulary applications use to exchange data. Beside the Web, the technologies developed for the Semantic Web have proven to be useful also in other domains, especially when data is exchanged between applications from different parties.

Software engineering is one of these domains in which recent research shows that Semantic Web technologies are able to reduce the barriers of proprietary data formats and enable interoperability. Model Driven Engineering (MDE) represents a key technology with a far-reaching vision for the future of software engineering. The main promise of MDE is to raise the level of abstraction from technology-platform-specific concepts to the higher levels of platform-independent and computation-independent modeling.

The Semantic Web vision starts from another perspective: sharing data, resources and knowledge between parties that belong to different organizations, different cultures, and/or different communities. Ontologies and rules play the main role in the Semantic Web for publishing community vocabularies and policies, for annotating resources and for turning Web applications into inference-enabled collaboration platforms.

Although these two technologies have been developed by two different communities, they share a number of principles and goals; there have been attempts to bring together languages and tools developed for Software Engineering with Semantic Web languages. There are important synergies that can be achieved by combining them with each other.

One of the most recent of these attempts is the development of the Object Management Group's Ontology Definition Metamodel (ODM) [10]. Until recently, this work has been motivated largely by an interest to exploit the popularity and features of Unified Modeling Language (UML) tools for the creation of vocabularies and ontologies for the Semantic Web. ODM standard can be viewed as a first step towards bridging MDE and the Semantic Web.

Wongthongtham et al. [11] proposed two software engineering ontologies, Generic Ontology and Application specific ontology using OWL.

The Generic ontology provides the vocabulary for the terms in software engineering while the Application-specific ontology is an explicit specification of software engineering for a particular software development project which can be used for communication in project agreement providing consistent understanding among project members.

Siricharoen [12] believes that merging ontologies with existing methods, techniques and tools used during the analysis phase of complex object oriented software systems can contribute significantly to reaching better decisions. It can be done by transforming user requirements represented as text description into an object model of the system under development, based on the use of ontologies which will improve the effectiveness and efficiency of the existing methodology for high-level system analysis in object oriented software engineering.

Design patterns have been used successfully in recent years in the software engineering community in order to share knowledge about the structural and behavioural properties of software.

There is a growing body of research in the area of design pattern detection and design recovery, requiring a formal description of patterns which can be matched by tools against the software that is to be analyzed.

The work of Dietrich et al. [13] uses also OWL to formally describe design patterns. However, only the structure of design patterns is considered, not the *intent* or *applicability*. They use their ontology to detect patterns in software artifacts, not to help in detecting models that could be replaceable by design patterns.

Design patterns are knowledge that is shared across a community and that is by nature distributed and inconsistent. By using the web ontology language (OWL) Kampffmeyer and Zschaler [1] defined the 23 *design patterns* [2] and a couple of related concepts such as *design pattern problem* and *problem concepts* to classify the design patterns according to their intents.

Their work was based on the work of Tichy [14], who developed a catalogue, not a formal ontology, listing over 100 design patterns. In his classification he concentrates on the problems patterns solve. They formalized and refined Tichy's classification and "made it available to mechanical treatment and to computer-aided querying by software developers".

Since our work rely on detecting “bad practices design” and replace them by design patterns if their intentions meet, an existing knowledge base on design patterns classified by their intent could really help us extracting this intention in order to discuss it with the user.

The aim of ontologies is to be reusable and shared. Therefore, we chose the DPIO ontology to reuse it and extend it with new concepts to fit our need.

3 Ontology Development Tool – Protégé

Dans cette section, nous nous attachons à présenter l’outil de développement libre Protégé développé à l’université de Standford. Cet outil permet de gérer des bases de connaissances décrites en OWL. Nous décrivons comment créer un nouveau concept, comment lui associer des propriétés et des contraintes, comment visualiser une ontologie, comment lancer des requêtes dans le langage intégré SPARQL et comment visualiser le résultat de ces requêtes.

Protégé [15] is a, open source ontology editor and knowledge-base framework. It can represent ontologies consisting of classes, properties, property characteristics and restriction and instances.

Protégé has a number of different plug-ins including OWL Plug-in. The OWL Plug-in is a complex Protégé extension which can be used to edit and create OWL files and databases. The Protégé-OWL editor enables users to build ontologies for the Semantic Web, in particular in the W3C's Web Ontology Language (OWL). Protégé ontologies can be exported into a variety of formats including RDF(S), OWL, and XML Schema.

We referred to a user guide [16] on how to develop an ontology using *Protégé* and *OWL Plug-in*. In this section, we will describe how to create OWL classes, link them by OWL properties and use the SPARQL panel to interrogate the knowledge base and retrieve results using Protégé – OWL.

3.1 Ontology Classes

OWL classes are interpreted as sets that contain individuals. They are described using formal (math-ematical) descriptions that state precisely the requirements for membership of the class. Classes may be organized into a superclass-subclass hierarchy, which is also known as taxonomy.

As can be seen in the “SubClass Explorer” in Figure 2, the class owl:Thing is the super class of all owl classes. OWL classes are assumed to overlap. Therefore one cannot assume that an individual is only a member of a particular class. In order to separate a group of classes, they must be disjoint from each other. This assures that an individual who has been asserted to be a member of one of the classes in the group cannot be a member of any other class in that group. For example, all the subclasses of the Composite_AM should be disjointed from one another. In the “Disjoints panel”, we have the classes Composite_AM_1, Composite_AM_2, Composite_AM_3, Compsite_AM_4 or Compsite_AM_6 disjointed from the selected class in the hierarchy Composite_AM_5. This means that there is no chance for an individual to be more than one class.

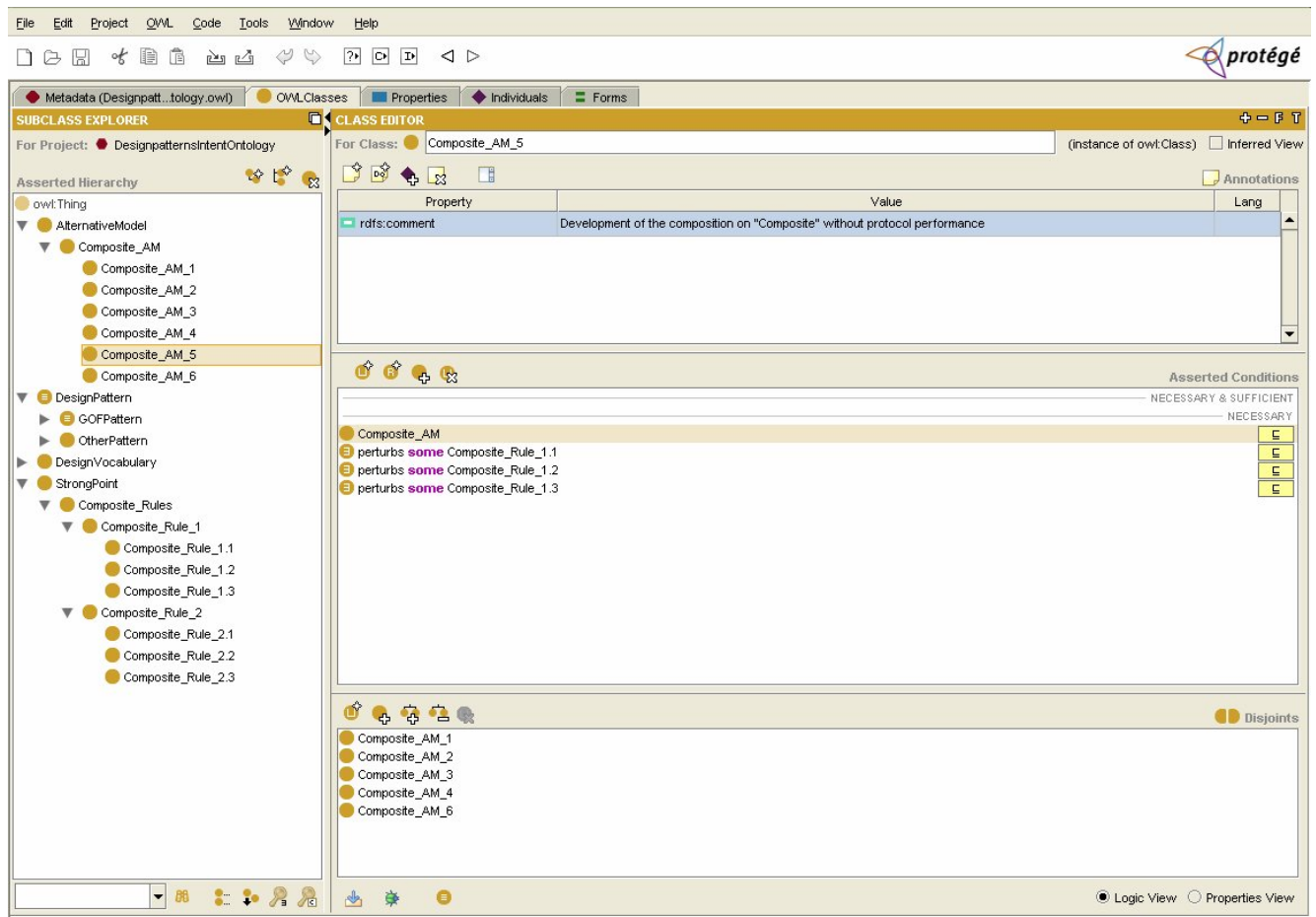


Figure 2: Protégé screenshot on creating OWL classes

3.2 Ontology Properties

OWL Properties represent relationships between two individuals. There are two main types of properties, Object properties and Datatype properties. Object properties link one class or individual to another. Datatype properties link an individual to an XML Schema Datatype value or an rdf literal. OWL also has a third type of property – Annotation properties. Annotation properties can be used to add information (metadata— data about data) to classes, individuals and object/datatype properties.

As we can see, Figure 3 is a screenshot from Protégé depicted that the object property named *perturbs* linking from class *AlternativeModel* to class *StrongPoint*. The “Asserted Conditions” panel centred in Figure 2, depicts the object property *perturbs* which links between *Composite_AM_5*, a sub class of *AlternativeModel* class, and three sub classes of the *StrongPoint* class (*Composite_Rule_1.1*, *Composite_Rule_1.2*, *Composite_Rule_1.3*)

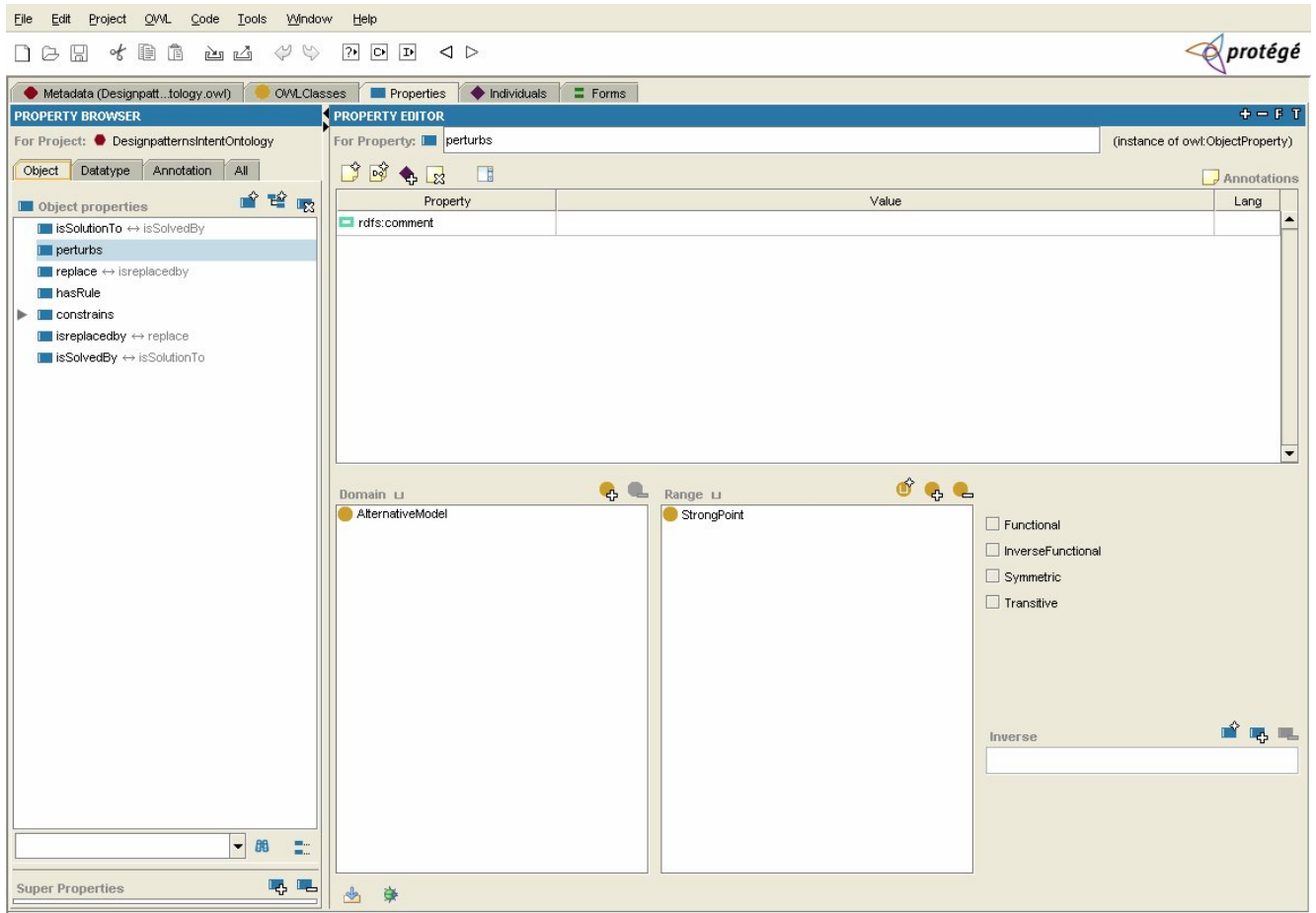


Figure 3: Protégé screenshot on creating OWL properties

The meaning of properties is enriched through the use of property characteristic. The various characteristics that properties have are functional, inverse functional, transitive, and symmetric. These characteristics can be set as shown in the bottom right of the Figure 3.

3.3 SPARQL panel

SPARQL [17] is a W3C Candidate Recommendation towards a standard query language for the Semantic Web. Its focus is on querying RDF graphs at the triple level. *SPARQL* can be used to query an RDF Schema or OWL model to filter out individuals with specific characteristics.

To use SPARQL in Protégé-OWL, version 3.2 or above is needed. In the OWL tab, select “Open SPARQL Query panel” menu item. This will open a panel at the bottom left of the screen shown in Figure 4.

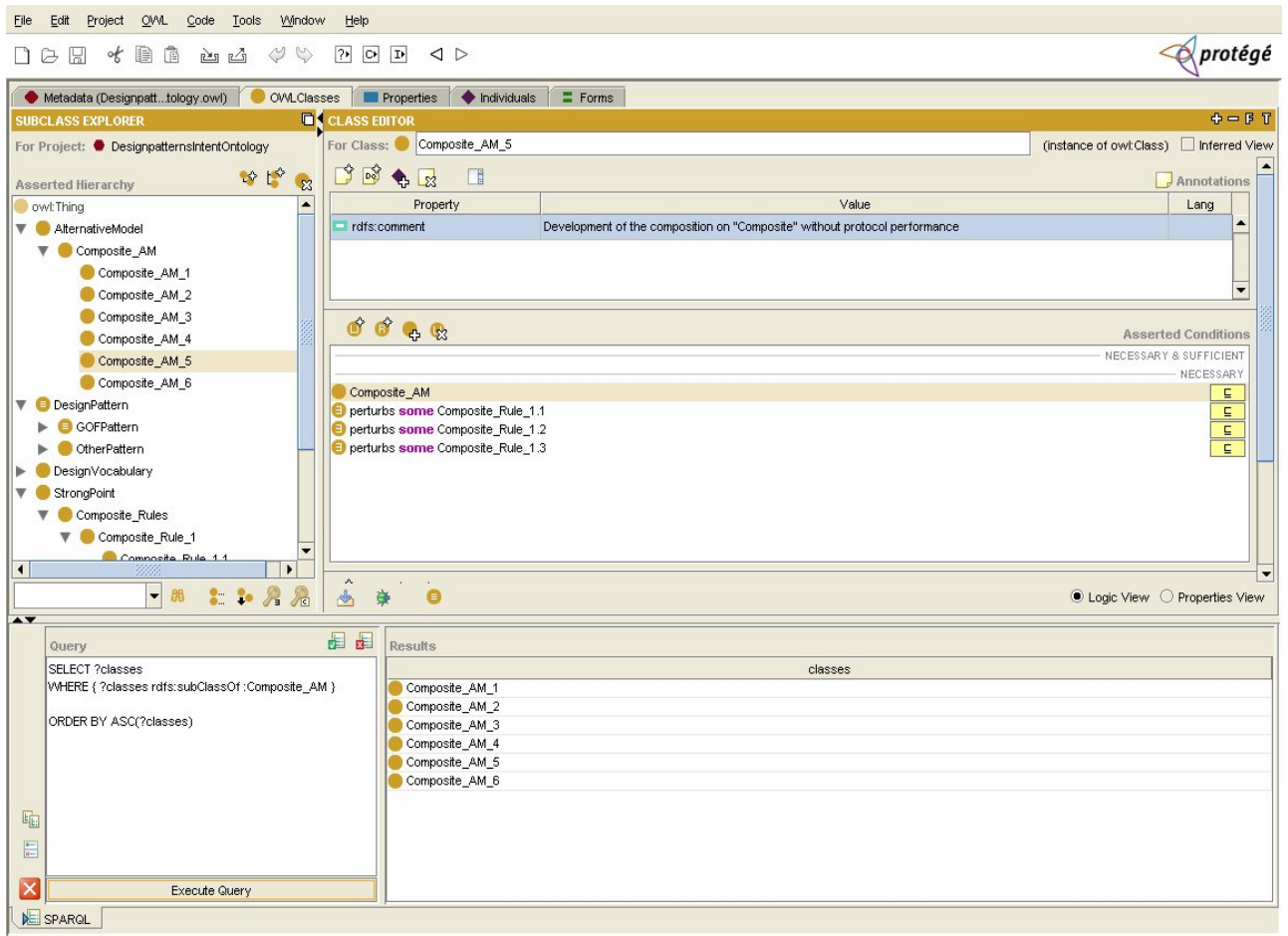


Figure 4: Protégé screenshot on interrogating the ontology using SPARQL

In the query panel, we generated a query that search for all the subclasses of *Composite_AM*. The “Results” panel on the bottom right shows that the results are identical to those existing in the ontology shown in the “Asserted Hierarchy”

We exemplify using a simple query just to give an overview on how to use SPARQL queries in *Protégé*. More complex queries will be depicted in *section 4* and *section 6*.

4 DPIO ontology: the Design Pattern Intent Ontology

Dans cette section, nous allons nous recentrer sur le point de départ de nos travaux qui a consisté à l'étude d'une base de connaissances dédiée à l'intention des patrons de conception : the DPIO ontology (Design Pattern Intent Ontology). Pour ce faire, nous rappelons des définitions essentielles sur les patrons de conception en donnant leur classification et leur pattern de présentation. Ce pattern sera illustré par un exemple qui est le patron composite. Puis nous décrivons en détail la structure de la base de connaissances étudiée et donnons des exemples de description en OWL et des exemples de requête en SPARQL. Pour ce faire, nous avons utilisé l'éditeur protégé et récupéré leur base de connaissances au format XML. Les requêtes de l'article ont été réécrites et validées dans notre environnement.

This *Design Pattern Intent Ontology* DPIO [1] is an extensible knowledge base of design patterns classified by their intent. It aims to help developers choose the right pattern that could solve their design problems. In designing its structure, they took into consideration the 23 design patterns from the Gang of Four (GoF) book [2].

4.1 Design Patterns

4.1.1 Definition

After what Alexander said about patterns in buildings and towns: "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" [18], another idea of reusing design by applying patterns to solve design problems has been used in object oriented software design by the GoF in their seminal book. They described a design pattern as "a solution to a problem in a context" [2].

4.1.2 Classification

Because Design patterns vary in their granularity and level of abstraction and due to the sheer amount of existing design patterns, they classified them by two criteria (cf. Table 1). The first criterion, called purpose, reflects what a pattern does. Patterns can have either creational, either structural, or behavioral purpose. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Table 1: Design pattern space

4.1.3 Description

In order to describe them in a consistent format, they divided each one of them into sections according to the following template. We will illustrate it by using an example: the Composite Design Pattern.

Pattern Name and Classification

The pattern's name conveys the essence of the pattern succinctly. The pattern's classification reflects the scheme introduced in Table 1. For example, the *Composite* design pattern is an example of a *Structural Object* pattern. It describes how to build a class hierarchy made up of classes for two kinds of objects: primitive and composite. The composite objects let you compose primitive and other composite objects into arbitrarily complex structures.

Intent

A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

The intent of the Composite design pattern is to “*Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly*”.

Also Known As

Other well-known names for the pattern, if any.

Motivation

A scenario that illustrates a design problem and how the class and object structures in the pattern, solve the problem. For example, in graphics applications like drawing editors and schematic capture, systems let users build complex diagrams out of simple components. The user can group components to form larger components, which in turn can be grouped to form still larger components. A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives.

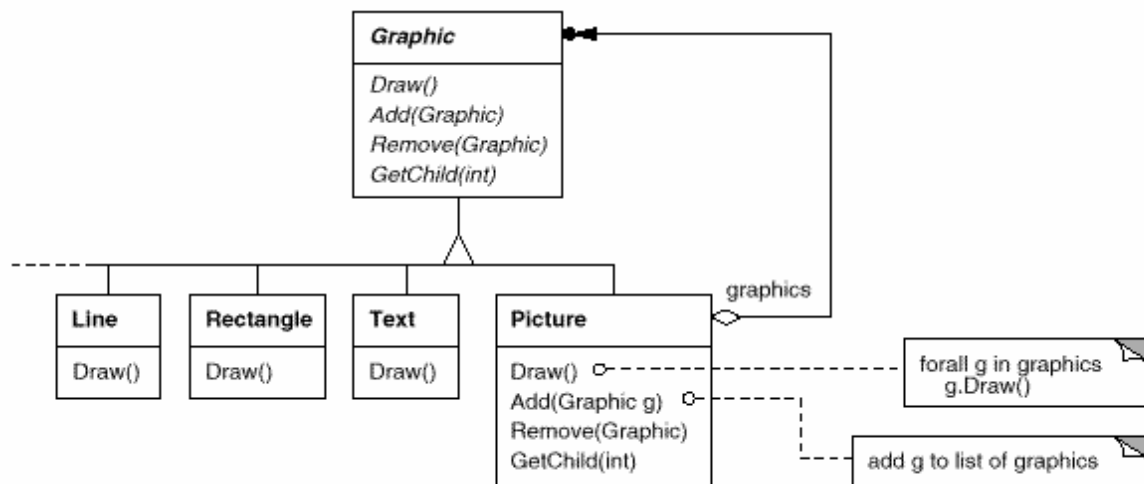


Figure 5: Implementation of a Graphic Application

But there's a problem with this approach because code that uses these classes must treat primitive and container objects differently; even if most of the time the user treats them identically. Having to distinguish these objects makes the application more complex. The Composite pattern describes how to use recursive composition so that clients don't have to make this distinction (cf. Figure 5).

Applicability

What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations? The Composite design pattern could be used when: we want to represent part-whole hierarchies of objects, we want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Structure

A graphical representation of the classes in the pattern (cf. Figure 6) using a notation based on the Object Modeling Technique. Interaction diagrams were used to illustrate sequences of requests and collaborations between objects.

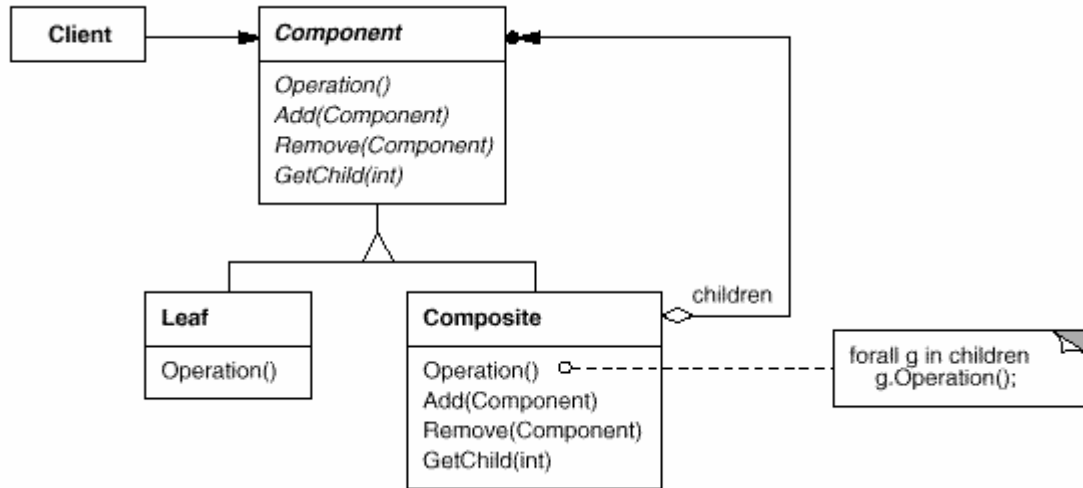


Figure 6: Structure of the Composite Design Pattern

Participants

The classes and/or objects who participates in the design pattern along with their responsibilities. Here, for example, the responsibilities of the classes/objects participating in the *Composite* design pattern.

- **Component (Graphic):**
 - declares the interface for objects in the composition.
 - implements default behavior for the interface common to all classes, as appropriate.
 - declares an interface for accessing and managing its child components.
 - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- **Leaf (Rectangle, Line, Text, etc.):**
 - represents leaf objects in the composition. A leaf has no children.
 - defines behavior for primitive objects in the composition.
- **Composite (Picture):**
 - defines behavior for components having children.
 - stores child components.

implements child-related operations in the Component interface.
- **Client:**
 - Manipulates objects in the composition through the Component interface.

Collaborations

How the participants collaborate to carry out their responsibilities. In the *Composite* design pattern, *Clients* use the *Component* class interface to interact with objects in the *composite* structure. If the recipient is a *Leaf*, then the request is handled

directly. If the recipient is a *Composite*, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

Consequences

How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

The *Composite* pattern:

- defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively. Wherever client code expects a primitive object, it can also take a composite object.
- makes the client simple. Clients can treat composite structures and individual objects uniformly. Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite component. This simplifies client code, because it avoids having to write tag-and-case-statement-style functions over the classes that define the composition.
- makes it easier to add new kinds of components. Newly defined Composite or Leaf subclasses work automatically with existing structures and client code. Clients don't have to be changed for new Component classes.
- can make your design overly general. The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.

Implementation

What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

Sample Code

Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

Known Uses

Examples of the pattern found in real systems. We include at least two examples from different domains.

Related Patterns

What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

For a developer to understand whether a design problem is a solution to his problem, he has to refer to the intent of a design pattern. But the sheer amount of patterns available makes it hard on developers to find the appropriate patterns for their design problems. And here it come the idea to formalize the design patterns, based on their intention [1]. They created the Design Pattern Intent Ontology (DPIO) using the Ontology Web Language (OWL).

4.2 The core structure of the DPIO

The core structure of the DPIO (*cf.* Figure 7), extracted from their paper, is represented by UML classes and associations. The relations between *DesignPattern*, *DesignProblem* and *ProblemConcept* classes are depicted using UML-notations. UML classes symbolize OWL classes and UML associations symbolize OWL object properties. Each *DesignPattern* is a solution to one or more design pattern problem *DPPProblem*. The association between them indicates an object property *isSolutionTo* which is an inverse property of *isSolvedBy*. *DPPProblem* is defined that is the root node for more specific problems. The association class *constrains* indicates an owl object property that can be specialized also by subproperties. *DPPProblem* are a set of classes that describe a problem by *constraining* a *ProblemConcept*.

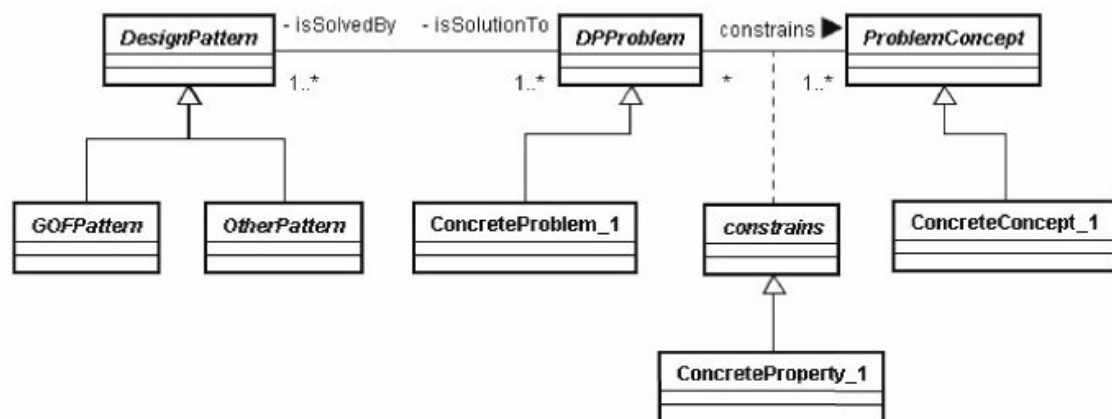


Figure 7: Graphical overview of the core structure of the DPIO

All the 23 GoF patterns inherit from the *DesignPattern* class. *DPPProblem* and *Problemconcept* are the root classes of sub-class hierarchies.

We will illustrate the structure of the DPIO ontology using the example of the *Composite* design pattern. Figure 8 is an instantiation of the UML model (*cf.* Figure 7) for the *Composite* pattern.

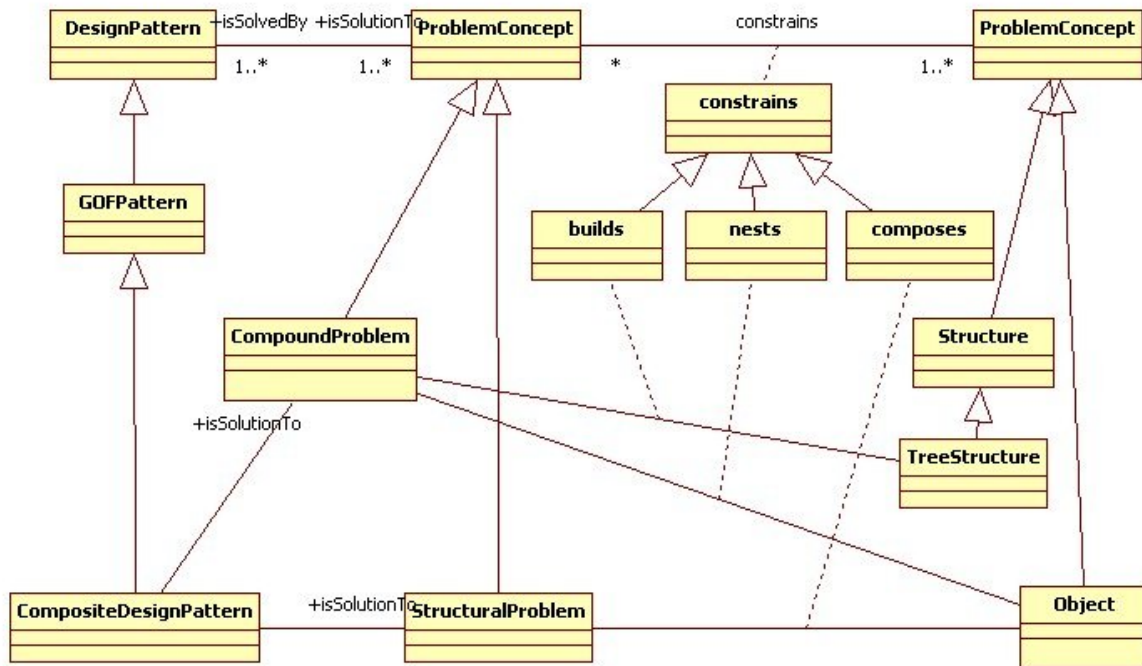


Figure 8: Instantiation of the DPIO UML design

Listings below show a description of the above in OWL Manchester syntax [19]. Listing 1 states that an individual is a *CompositeDesignPattern* if it is a subclass of *GOFPattern* and is a solution to certain problems as per the intent of the *Composite* design pattern: “*Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly*” [2].

```

Class: CompositeDesignPattern

SubClassOf: GOFPattern

and isSolutionTo some StructuralProblem

and isSolutionTo some CompoundProblem
  
```

Listing 1: Definition of the OWL design pattern class CompositeDesignPattern

The design problems formalizing these aspects are *StructuralProblem* and *CompoundProblem*. Together they define a set of design problem facets with the need to compose object and nest them to build tree structure. *StructuralProblem* and *CompoundProblem* constrain each his proper problem concepts. *StructuralProblem* composes *Object* (cf. Listing 2) while *CompoundProblem* nests *Object* and builds *TreeStructure* (cf. Listing 3).

```
Class: StructuralProblem
SubClassOf: DPPProblem
and composes some Object
```

Listing 2: Definition of the OWL design problem class StructuralProblem

```
Class: CompoundProblem
SubClassOf: DPPProblem
and nests some Object
and builds some TreeStructure
```

Listing 3: Definition of the OWL design problem class CompoundProblem

Object inherits from ProblemConcept class (cf. Listing 4) while TreeStructure (cf. Listing 5) inherits from Structure, a subclass of ProblemConcept.

```
Class: Object
SubClassOf: ProblemConcept
```

Listing 4: Definition of the OWL problem concept class Object

```
Class: TreeStructure
SubClassOf: Structure
Class: Structure
SubClassOf: ProblemConcept
```

Listing 5: Definition of the OWL design problem classes TreeStructure and Structure

OWL object property constrains, modeled as a UML association class is the root property of more specialized object properties like composes, nests and builds (cf. Listing 6). They are used to model the design problems.

```
ObjectProperty: composes
    SubPropertyOf: constrains

ObjectProperty: nests
    SubPropertyOf: constrains

ObjectProperty: builds
    SubPropertyOf: constrains
```

Listing 6: Definition of the OWL object properties: constrains, nests and builds

4.3 Extracting knowledge

The DPIO contains the vocabulary for describing the intent of design patterns. In order to extract knowledge from the ontology, the user has to execute queries on it. To this end, they developed a user friendly tool, the *Design Pattern Wizard* which serves as a front-end for generating well-defined queries [1]. It suggests a set of matching design patterns based on a visually description of a design problem.

In order to evaluate their ontology, they checked if it can answer the kind of questions that are likely to be asked of it. A catalogue of *competency questions* was developed and translated into a formal version. They used the nRQL : new Racer Query Language [20] in formalizing these questions.

Here is a kind of question the ontology should be able to answer: *Which design pattern is a solution to the problem of composing object?*

A formalization of this question could be done by generating a query using nRQL syntax (cf. Listing 7) to retrieve all design patterns that are a solution to composing an object.

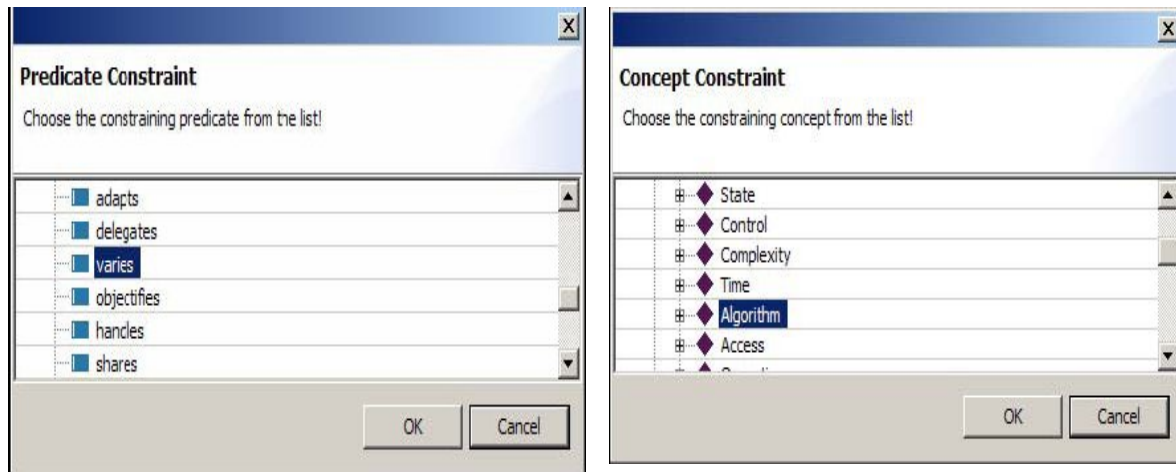
```
(retrieve (?x )
  (and (?x |GOFPattern|)
    (?x ?p |isSolutionTo|)
    (?p ?a |composes|)
    (?a |Object|)
  ))
```

Listing 7: nRQL query to retrieve all design patterns that are a solution to composing an object

In their tool, they generated nRQL queries based on the description of the problem. In this example above, users trying to figure out the design pattern that could solve this problem, should select “*composes*” from the *Predicate Constraint* panel (cf.

Figure 9 (a)) which contains the list of all the inherited properties of *constrains*, and “Object” from *Concept Constraint* panel (cf. Figure 9(b)) that contains the list of all the inherited classes from *ProblemConcept*.

A list of all matching design patterns will be displayed along with a description of each one of them. For example, referring to the detailed description of the *Composite* design pattern shown before, we can assume that this pattern will be part of the list of results since it “composes Object”.



(a) Constraining problem predicates

(b) Constraining problem concepts

Figure 9: Screenshot of the dialogues constraining problem predicates and problem concepts

These are screenshots of the “*Design Pattern Wizard*” tool, taken from their paper on how to find the solution to the problem of “*varying an Algorithm*”.

Their tool suggests the *Strategy* and the *Template* patterns to the proposed problem and gives the description of each pattern for more details (cf. Figure 10).

Based on this study, and since we have no access on their tool, and in order to evaluate it, we used the Protégé- OWL to load their ontology so we can check the description of all their classes and the relations between them, and interrogate their knowledge base using SPARQL queries. To answer the competency question about the design pattern that could solve the problem of “composing objects”, we have generated a SPARQL query (cf. Listing 8) that retrieves all the matching design patterns:

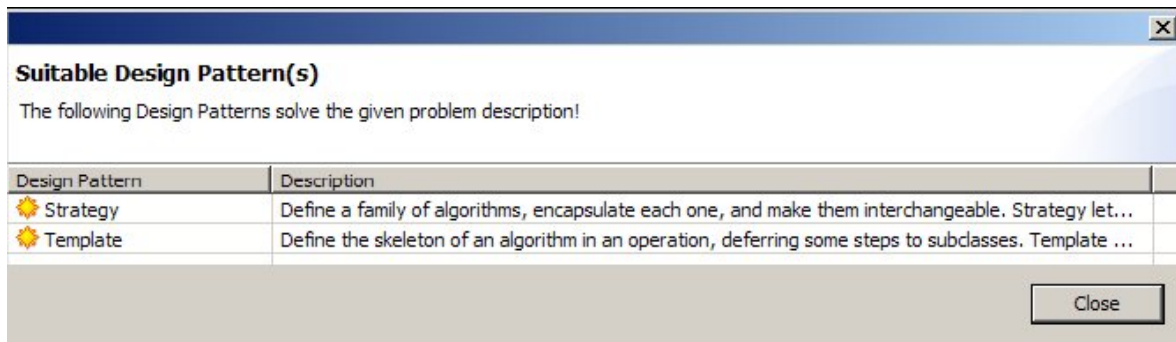


Figure 10: Screenshot of the result window suggesting design patterns for the problem of varying an algorithm

```

SELECT ?DesignPattern
WHERE {
  ? DesignPattern rdfs:subClassOf :GoFPattern.
  ? DesignPattern rdfs:subClassOf ?x.
  ?x rdf:type owl:Restriction.
  ?x owl:onProperty :isSolutionTo.
  ?x owl:someValuesFrom ?designproblem.
  ?designproblem rdfs:subClassOf ?y.
  ?y rdf:type owl:Restriction.
  ?y owl:onProperty :composes.
  ?y owl:someValuesFrom :Object.}

```

Listing 8: SPARQL query to retrieve all design patterns that are a solution to composing an object

We used this query to search for the design pattern which is a subclass of GoFPattern and can solve the problem of composing object. A list of design patterns could solve this problem including the CompositeDesignPattern (cf. Figure 11). We have generated all the queries in the paper using SPARQL syntax, and along with this, the queries that will answer to the competency questions of our new ontology.

Since in our work, we have to replace “bad design practices” with *Design Patterns* if they have the same intention, that means that we need to retrieve the intent of a design pattern instead of getting the design pattern based on the user’s intent. It’s much like reversing the query to get the pertinent data from this ontology. This is why we need to import the DPIO ontology and add our new concepts so we can benefit from the study they have already done about design patterns and their intents. We will talk about this later after we introduce our new concepts for the new ontology.

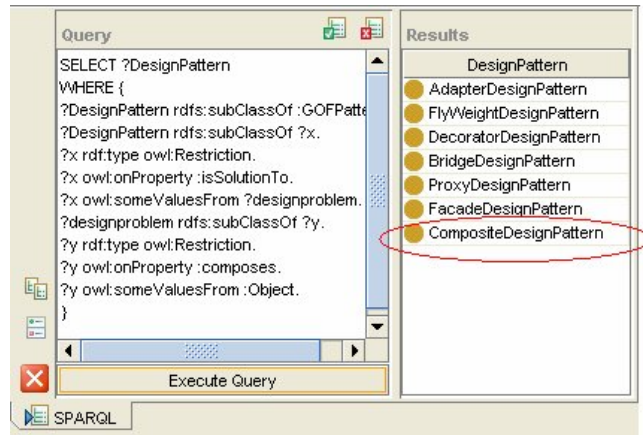


Figure 11: Protégé screenshot of the result window suggesting design patterns for the problem of composing an object

5 Alternative models and Design Patterns' strong points

Après avoir présenté l'existant, nous présentons les concepts à greffer à la base de connaissances étudiée. Un patron de conception n'est plus représenté uniquement par son intention, mais aussi par un ensemble de points forts, eux-mêmes composés de sous-caractéristiques expliquant les qualités logicielles apportées par l'intégration de ce patron dans un modèle de conception. Ces points forts sont déduits d'une partie non exploitée jusqu'à présent de la section intention d'un patron et des conditions d'applicabilité. Ils sont validés par la valuation des différents modèles alternatifs de ce patron. Un modèle alternatif est un modèle qui résout le même genre de problème qu'un patron de conception avec une structure en général plus complexe et donc moins optimisé.

As we can see, the DPIO ontology gather all the information about the 23 design patterns and aim to help the analyst find the solution to his design problem. This work is really helpful for the designer in the conception phase because it forces him to use *Design Patterns* in his design which is an advantage in the maintenance and the development phases.

Based on the *Intent* and the *Applicability* of the Design patterns, Bouhours et al. [3] have defined the strong points that characterize a design pattern. A strong point is a key design feature which permits the pattern to resolve a problem more efficiently. For example, the Composite pattern resolves the problem: "How compose and use object hierarchies as simply as possible for a client in keeping extensibility possibilities on components". So the two strong points they found in this pattern are Uniform Protocol and Decoupling and Extensibility. Their study focuses on a detailed "design review activity" which precedes the coding phase. The problematic starts when the expert tries to review the design, to check if there is a misuse of Design Patterns. He needs to optimize each design by replacing "bad design practices" called Alternative Models [3] by Design Patterns whenever it's necessary.

Alternative Model is a "model which solves the same problem as the pattern, but with a more complex or different structure than the pattern". But the huge number of classes in a model makes the detection of those models a difficult job. So they introduced a "design review" activity that could replace part of the expert's job in parsing designs, detecting the alternative models and replacing them by design patterns if the intent of the designer and the expected architectural qualities are compatible.

They searched first for remarkable features that allow their detection in each model, automated the search using OCL rules supported by the NEPTUNE platform (Neptune2003) and tested those rules on industrial models (NEPTUNE 2 platform) and standard meta-models (SPEM 1.0 and core UML 1.4).

For their study, they chose students in 3rd and 5th year of software engineering, who are able to make UML models and had courses on object oriented software, to find solutions to the problems solved by design patterns. Although those students had no idea on design patterns, they did solve these problems without the need of design patterns.

From three hundred models obtained, they have selected fifteen that could be considered as a good design, the others were either duplicated or incorrect.

After their experiment, they collected between two and six alternative models considered as “bad design practices” for each design pattern. In order to distinct the different alternative models of a design pattern; they classified them by their structure and the strong points they lack in.

Alternative models perturbs the strong points of each pattern, they may damage some or all of them. In order to specify the degree of the damage, they expanded each strong point by adding sub features.

And finally to detect each alternative model in a design, they applied OCL rules on UML models to find the occurrences of their generalization. Figure 12 shows a generalization of an alternative model of the CompositeDesignPattern

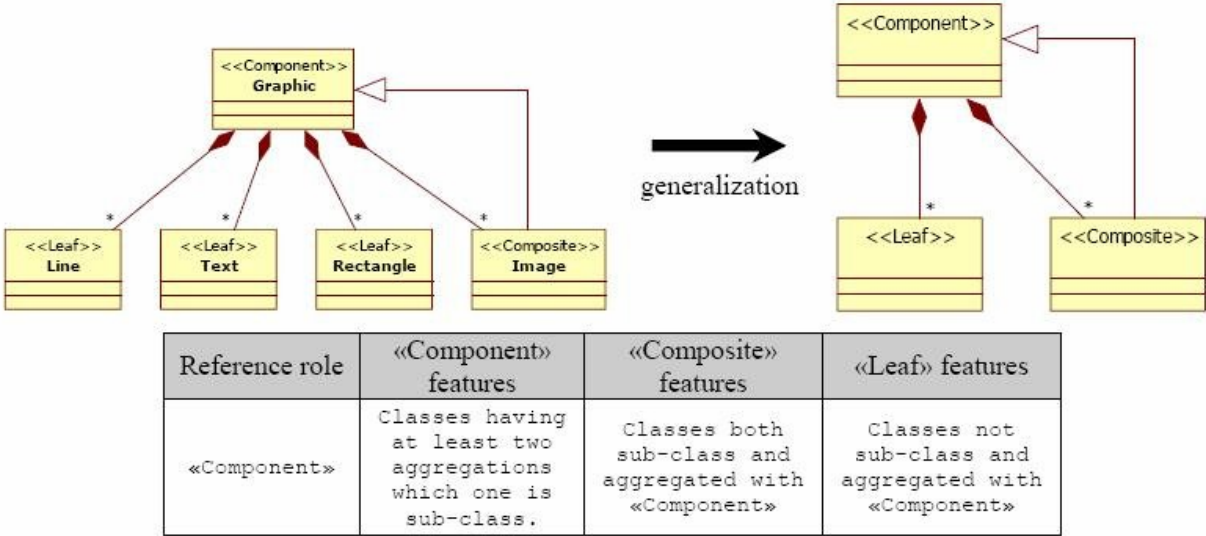


Figure 12: Generalization of the Composite alternative model and its structural features

6 Extending the DPIO ontology

Notre travail a consisté à l'extension de la base de connaissances DPIO avec les concepts introduits dans la section précédente. Pour ce faire, nous avons procédé de la manière suivante : nous avons déterminé un ensemble de questions clefs pour le domaine considéré.

Ces questions étaient : à quel design pattern correspond le modèle alternatif donné ? Quelle est l'intention du design pattern considéré ? Quels sont les points forts que le modèle alternatif n'atteint pas ? En effet, dans l'activité revue de design que nous devons outiller, ces questions sont primordiales pour dialoguer avec l'expert ou le concepteur du modèle. Nous avons à notre disposition une base de modèles alternatifs. Chacun de ces modèles alternatifs est recherché de manière automatique dans un modèle industriel. Imaginons que nous trouvions un fragment de modèles qui correspond exactement aux particularités structurelles d'un modèle alternatif que nous nommons MA. Les particularités structurelles remarquables ne nous donnant aucune information sur la sémantique d'un fragment de modèle, nous devons interroger le concepteur afin qu'il valide ou invalide l'égalité d'intention existante entre le fragment de modèle et le modèle MA. Pour ce faire, nous devons être capable de générer une requête en SPARQL renvoyant l'intention du patron de conception que le modèle MA a remplacé dans le modèle industriel considéré. Si les intentions « concordent », l'injection du patron dans le modèle industriel n'est pas automatique. En effet, une telle opération de restructuration peut s'avérer coûteuse, surtout s'il existe déjà une implémentation industrielle. Il s'avère donc nécessaire de présenter les avantages que l'utilisation du patron de conception amènerait en interrogeant la base sur les points forts non atteints par le fragment de modèle industriel.

Nous avons proposé un modèle ajoutant des concepts et des rôles au modèle de la base DPIO, puis nous avons implémenté ce modèle dans l'outil protégé et nous l'avons validé en exhibant des requêtes répondant aux questions clés posées. Notre étude est illustrée par un scénario d'utilisation sur un modèle alternatif au modèle composite.

6.1 Conception and Implementation

To determine the scope of an ontology, there are kind of questions called “competency questions” the ontology should be able to answer [9]. So after we introduced the new concepts that we wish to add to the DPIO ontology, we will show how we benefit from the classes already created in the DPIO, and how we took into account all the possible relations between them and the new classes.

Our work would be defined in 3 steps:

First, when an alternative model is detected, we need to interrogate our base to know which design pattern could replace it.

Second, after knowing the design pattern that could replace it, we need to check if the intention of the designer in structuring this alternative model is the same intention of the design pattern found.

Last but not least, if that is, we need to show the designer where his model lack by showing him the strong points that could offer him a design pattern and which is missed in his model.

If the designer finds that he needs to have those strong points in his model, this fragment should be substitutable with the appropriate design pattern.

So we have three competency questions that we need the new ontology to be able to answer:

1. Which design pattern could replace a given alternative model?
2. What is the intent of a design pattern?
3. Which are the strong points that lack in an alternative model?

In designing the structure of the new ontology, we took into consideration the possible relations between the classes in the DPIO model and the classes we want to add:

1. Each *Alternative Model* could be replaced by one and only one *Design Pattern*. But a *Design Pattern* will replace one to many alternative models.
2. Each *Design pattern* has at least one *Strong Point* that characterizes it from being an alternative model. One *Strong Point* could belong to more than one *Design Pattern*.
3. An *Alternative model* perturbs the *Strong Points* of the *Design Pattern* that can replace him.

From this analyze, we extend the UML model found in [1] by adding our new concepts.

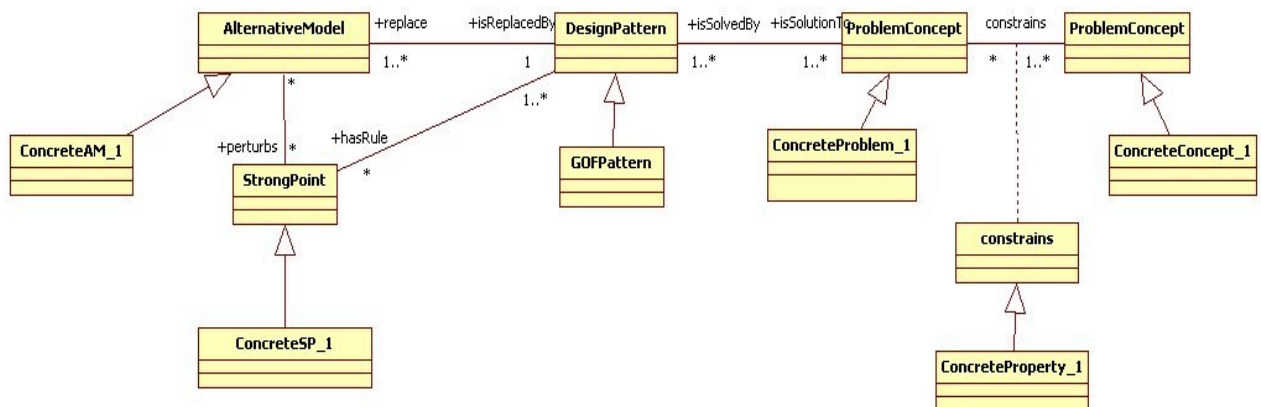


Figure 13: Graphical overview of the structure of the extended ontology

Figure 13 represents the new structure of the extended UML model. Based on this structure and the relations between classes, we created in the new ontology base owl classes and properties as follow:

1. Two new OWL classes:
 - a. *AlternativeModel*: the root class of all sub alternative models.
 - b. *StrongPoint*: the root class of all the strong points and their sub features. Each sub feature inherits from a strong point.
2. Four new owl properties:
 - a. *isReplacedBy*: owl property that links the *AlternativeModel* and *DesignPattern* classes.
 - b. *Replace*: an owl inverse property of *isReplacedBy*.
 - c. *Perturbes*: an owl property that links the *AlternativeModel* and *StrongPoint* classes
 - d. *hasRule*: owl property that links *DesignPattern* class to the *StrongPoint* class.

As a result, the 1st question on which design pattern could replace a given alternative model is answered due to the association between *AlterantiveModel* and *DesignPattern* class.

Moreover, the DPIO ontology which contains all the information about the intent of a design pattern helped us answering the 2nd question on what is the intent of a design pattern. Instead of using their queries that searches for design patterns for a given problem, we will reverse it to retrieve the problem that could be solved by a given design pattern. This is the intent of the design pattern.

For the last question on which are the strong points that lack in an alternative model, the relations existing between *StrongPoint* and *AlternativeModel* in one hand, and the *AlternativeModel* and the *DesignPattern* in the other hand lead us to retrieve the pertinent results.

To discuss the structure of the new ontology in more detail, we will present a design problem, the pattern that could solve it along with its strong points and its alternative models.

Design problem: Design a system enabling to draw a graphic image: a graphic image is composed of lines, rectangles, texts and images. An image may be composed of other images, lines, rectangles and texts.

This problem could be solved ideally using the *CompositeDesignPattern*, we will present its structure in Figure 14 and his strong points:

Structure:

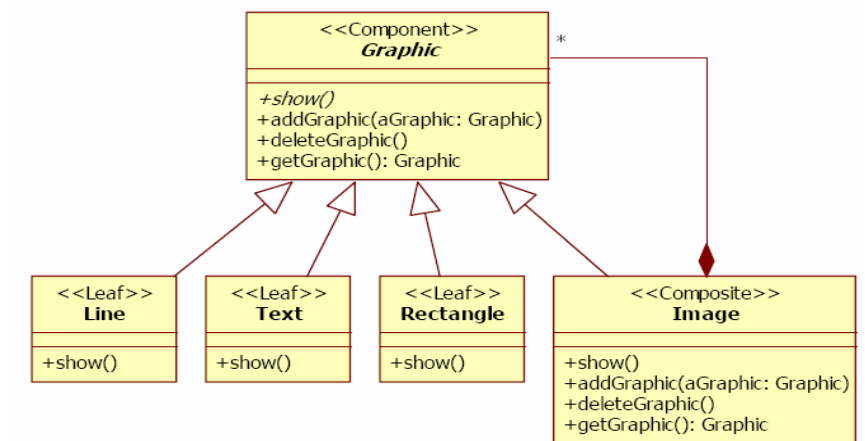


Figure 14: Structure of the Composite Design pattern

Strong Points:

1. Decoupling and extensibility.
 - 1.1. Maximal factorization of the composition.
 - 1.2. Addition or removal of a leaf does not need code modification.
 - 1.3. Addition or removal of a composite does not need code modification.
2. Uniform protocol.
 - 2.1. Uniform protocol on operations of composed object.
 - 2.2. Uniform protocol on composition managing.
 - 2.3. Unique access point for the client.

Another solution to this problem could be using an Alternative Model:

The *CompositeDesignPattern* have 6 alternative models. They satisfy between zero to three 0:6 strong points. To integrate them into the DPIO base, we have created a super class *Composite_AM* for all the alternative models of the *Composite* design pattern that inherits from the *AlternativeModel* class (shown in Figure 15) while the strong points and their sub features was created with the same hierarchy defined in their definition (cf. Figure 16).

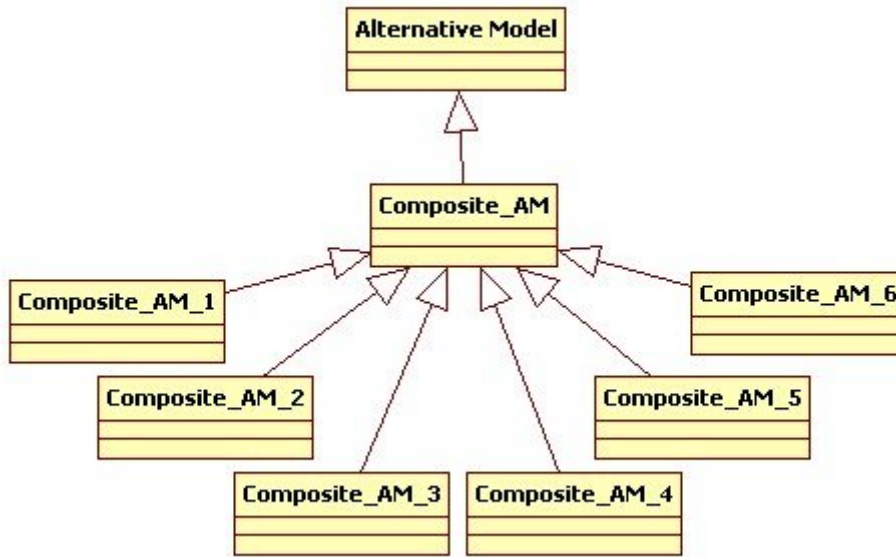


Figure 15: Structure of the Alternative Model class hierarchy

All the alternative models inherits from a super class having the name of the design pattern that could replace them.

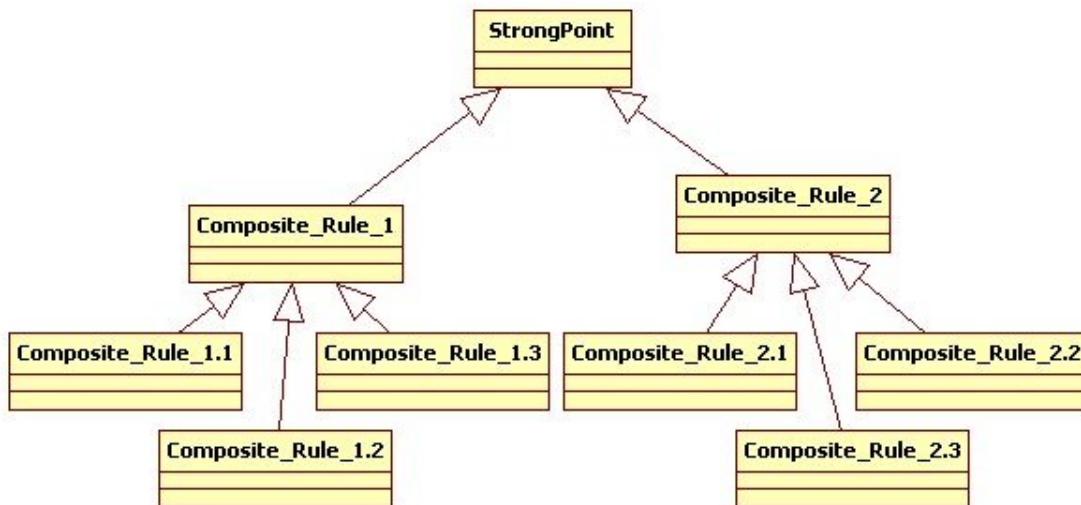


Figure 16: Structure of the Strong Point class hierarchy

The Strong point class is the super class of all the design patterns’ strong points where each strong point has sub features. The *Composite* design pattern has two strong points having each three sub features (*cf.* Figure 16)

We will introduce an example of one of the alternative models of the *Composite* pattern: the “development of the composition on “Composite” without protocol conformance, along with its structure (Figure 17) and the strong points that it perturbs.

This is the alternative model 5 in the list, named *Composite_AM_5* in the new ontology. You can refer to [3] for more details on the alternative models studied and the strong points that they satisfy.

Alternative model 5: Development of the composition on “Composite” without protocol conformance:

Structure

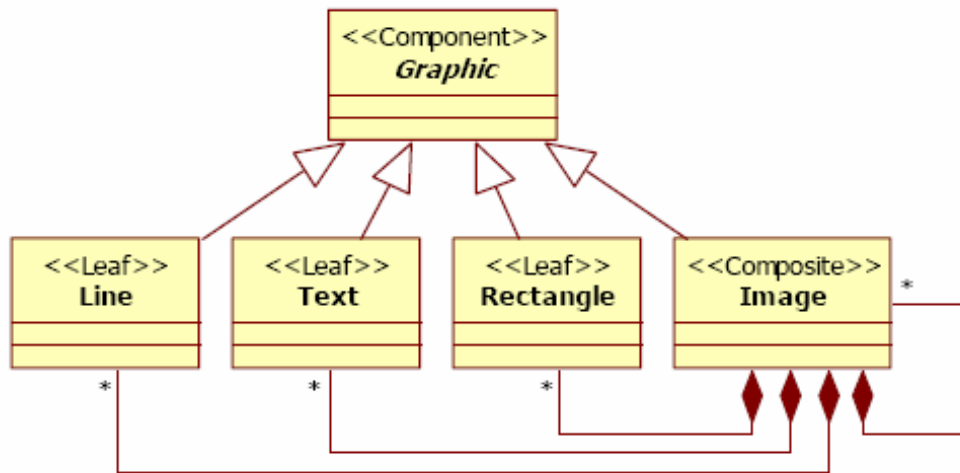


Figure 17: Structure of the alternative model 5 of the Composite Design pattern

Strong Point Perturbations

Error! Reference source not found. shows the strong point of the design pattern that was not satisfied by the alternative model 5:

The alternative model 5 has lack in the following highlighted strong points of the *CompositeDesignPattern*:

1. Decoupling and extensibility.
 - 1.1. Maximal factorization of the composition.
 - 1.2. Addition or removal of a leaf does not need code modification.
 - 1.3. Addition or removal of a composite does not need code modification.
2. Uniform protocol.
 - 2.1. Uniform protocol on operations of composed object.
 - 2.2. Uniform protocol on composition managing.
 - 2.3. Unique access point for the client.

Strong points perturbation			
1.1	✘	2.1	✔
1.2	✘	2.2	✔
1.3	✘	2.3	✔

Table 2: Strong Points perturbation

Figure 18 will show an instantiation of the UML model (cf. Figure 13) to show the relation between the alternative model *Composite_AM_5* and the strong points of the *CompositeDesignPattern*:

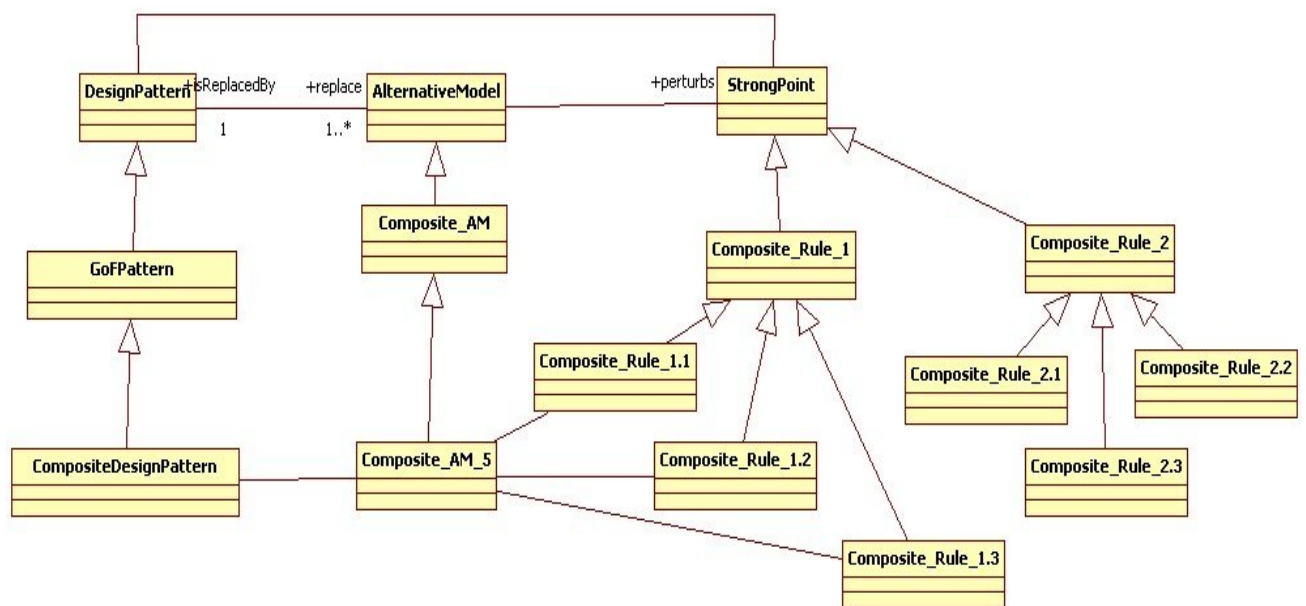


Figure 18: Instantiation of the UML model of the extended ontology

If we find that a design model lack in good object design practice, we will discuss with the designer the intent of his fragment and suggest the design pattern that could replace it, if their intentions meet, by showing him a list of the unsatisfied strong points. In the next section, we will present how we interrogate our base to retrieve the results we need to question the designer.

6.2 Queries using SPARQL

After adding our new concepts to the DPIO, the knowledge base could now be interrogated according to the competency questions we mentioned earlier. Standard ontology reasoning can be used to retrieve the results responding to queries. We used SPARQL (supported by Protégé-OWL editor) syntax when generating our queries to filter out the pertinent results.

We suppose in our example that we detected a fragment similar to the generalization of the alternative model 5 shown above which is named in our new ontology as “Composite_AM_5”, our job is to display the intention of the design pattern that could replace this fragment. So the designer could check if his fragment is substitutable with the chosen design pattern.

Listing 9 shows a formal representation using SPARQL syntax. This query retrieves the intention of the design pattern that could replace the alternative model “Composite_AM_5”, along with its intent:

```
SELECT ?DesignPattern ?constrains ?ProblemConcept
WHERE {
    ?DesignPattern rdfs:subClassOf ?x.
    ?x rdf:type owl:Restriction.
    ?x owl:onProperty :replace.
    ?x owl:someValuesFrom :Composite_AM_5.
    ?DesignPattern rdfs:subClassOf ?y.
    ?y rdf:type owl:Restriction.
    ?y owl:onProperty :isSolutionTo.
    ?y owl:someValuesFrom ?pbconcept.
    ?pbconcept rdfs:subClassOf ?z.
    ?z rdf:type owl:Restriction.
    ?z owl:onProperty ?constrains.
    ?z owl:someValuesFrom ?ProblemConcept.
}
```

Listing 9: SPARQL query to retrieve the design pattern that could replace an alternative model along with its intent

Since each design problem that can be solved by a design pattern, is described using constrains property and a concept class , we will use them to display the intent of a design pattern. In this query we’ll retrieve the design pattern that could replace the “Composite_AM_5” and its intent.

Our first question to the designer will be based on the results shown in Figure 19. For example, we have detected that your fragment is an alternative model of the *CompositeDesignPattern* where its intent is to “compose recursively and hierarchically the objects”, is it your intention?

And based on his answer, if positive, we will continue to the next step and display the strong points unsatisfied by his fragment.

Results		
DesignPattern	constrains	ProblemConcept
CompositeDesignPattern	composes	Object
CompositeDesignPattern	builds	TreeStructure
CompositeDesignPattern	nests	Object

Figure 19: Screenshot of the result window suggesting suitable design pattern and its intent

Listing 10 shows how to retrieve those strong points:

```
SELECT ?Points_Forts ?Sous_Points_Forts
WHERE {
    :Composite_AM_5 rdfs:subClassOf ?x.
    ?x rdf:type owl:Restriction.
    ?x owl:onProperty :perturbs.
    ?x owl:someValuesFrom ?SSP.
    ?SSP rdfs:subClassOf ?SP.
    ?SP rdfs:comment ?Points_Forts.
    ?SSP rdfs:comment ?Sous_Points_Forts.
}
ORDER BY ASC(?Points_Forts)
```

Listing 10: SPARQL query to retrieve the strong points of a design pattern which is perturbed by an alternative model

This query retrieves the strong points and their sub features perturbed by the designer’s alternative model detected (*cf.* Figure 20):

After retrieving those results, we will move to ask a new question to the designer, as for example: Our analyze shows that you have problems of “Decoupling and Extensibility”; your model is unable to satisfy those points:

1. Maximal factorization of the composition.
2. Addition or removal of a leaf does not need code modification.
3. Addition or removal of a composite does not need code modification.

It’s for the designer to check whether he needs to have those strong points in his model or not and if yes, it should be automatically replaced by a design pattern.

Results	
Points_Forts	Sous_Points_Forts
Decoupling and Extensibility	Addition or removal of a Composite does not need code modification
Decoupling and Extensibility	Addition or removal of a Leaf does not need code modification
Decoupling and Extensibility	Maximal factorization of the composition

Figure 20: Screenshot of the result window presenting the strong points perturbed

7 Conclusion

Nous voudrions rappeler dans un premier temps la transversalité de notre étude, nous avons à étudier d'une part la problématique de notre équipe d'accueil qui est une problématique spécifique des architectures à objets et des patrons de conception (comprendre les besoins en terme de base de connaissances et d'outillage) et d'autre part le domaine du Web Sémantique, des ontologies et des langages dédiés (OWL, nRQL, SPARQL) et la mise en œuvre dans un outil spécifique *Protégé* développé par l'université de Standford.

Dans ce contexte, nous avons apporté des réponses et des solutions aux problèmes posés. Nous avons proposé et implémenté un nouveau schéma de base de connaissances prenant en compte les travaux existants. Nous avons validé ce schéma par un ensemble de requêtes répondant aux questions clés déduites de notre analyse des besoins. Notre travail va servir à notre équipe d'accueil comme base d'un outillage de la 3ème étape de l'activité de « revue de conception » proposé par notre équipe d'accueil.

Les perspectives à notre travail pourraient être les suivantes :

- Prise en compte dans la base de connaissances des relations inter-patrons.
- Prise en compte dans la base de connaissances des conditions d'applicabilité de chaque patron.
- Hiérarchisation des concepts associés aux modèles alternatifs et aux points forts.
- Utilisation des possibilités d'inférence pour une classification automatique des nouveaux modèles alternatifs et points forts trouvés par expérimentation.

De manière plus générale, proposer des bases de connaissances pour explorer des domaines de connaissances non formalisés du Génie Logiciel. Par exemple, les patrons d'analyse, les patrons architecturaux, les anti-patterns, les patrons de modélisation, les patrons de méta modélisation,...

Bibliography

- [1] Kampffmeyer, H. & Zschaler, S. Engels, G.; Opdyke, B.; Schmidt, D. C. & Weil, F. (ed.): *Finding the Pattern You Need: The Design Pattern Intent Ontology*, MoDELS, Springer, 2007.
- [2] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Professional, 1995.
- [3] Cédric Bouhours, Hervé Leblanc, Christian Percebois. : *Alternative Models for a Design Review Activity*. In: Workshop on Quality in Modeling - ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, NASHVILLE, TN (USA), 30/09/07-05/10/07, Ludwig KUZNIARZ, Jean-Louis SOURROUILLE, Miroslaw STARON (Eds.), Springer, p. 65-79, octobre 2007.
- [4] Berners-Lee Tim, Hendler James & Lassila Ora., *The Semantic Web*, Scientific American, May 2001.
- [5] H. Wang, J. S. Dong, J. Sun and J. Sun.: *Reasoning Support for Semantic Web Ontology Family Languages Using Alloy*. International Journal of Multiagent and Grid Systems, IOS press, Vol-2(4):455-471, 2006.
- [6] O. Lassila and R.R. Swick, (eds): *Resource description framework (rdf) model and syntax specification*, Feb 1999.
<http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/>
- [7] D. Brickley and R.V. Guha, (eds): *Resource description framework (rdf) schema specification 1.0*, March 2000.
<http://www.w3.org/TR/2000/CR-rdfschema-20000327/>
- [8] D.L. McGuinness and F. van Harmelen: *OWL Web Ontology Language Overview*, 2004.
<http://www.w3c.org/TR/owl-features/>
- [9] Noy, N.F., McGuinness, D.L.: *Ontology development 101: A guide to creating your first ontology*. Technical Report KSL-01-05, Knowledge Systems Laboratory, Stanford University, Stanford, CA, 94305, USA, March 2001.
- [10] *Ontology Definition Metamodel ODM*, <http://www.omg.org/ontology/>
- [11] P.Wongthongtham, E. Chang, T.S. Dillon, I. Sommerville: *Software Engineering Ontologies and their implementation*, In: IASTED International Conference on Software Engineering, pp. 208-213, Innsbruck, Austria, Feb 2005.
- [12] Waralak V. Siricharoen: *Ontologies and Software Engineering*, International Conference on Computational Science (2), 2007.

- [13] Dietrich, J., Elgar, C.: *A formal description of design patterns using OWL*, in: Australian Software Engineering Conference (ASWEC'05), pp. 243–250. IEEE Computer Society, Los Alamitos, 2005.
<http://doi.ieeecomputersociety.org/10.1109/ASWEC.2005.6>
- [14] Tichy, W.F.: *A catalogue of general-purpose software design patterns*. In: TOOLS'97. Proceedings of the Tools-23: Technology of Object-Oriented Languages and Systems, IEEE Computer Society, Washington, DC, USA, 1997.
- [15] *Protégé ontology editor* and knowledge acquisition system (2006).
<http://protege.stanford.edu/>
- [16] M. Horridge, H. Knublauch, A. Rector, R. Stevens, and C.Wroe, *A practical guide to building owl ontologies using the Protégé-OWL plugin and CO-ODE tools*. Online tutorial, The University Of Manchester and Stanford University, August 2004.
- [17] Prud'hommeaux E., Seaborne: *SPARQL Query Language for RDF*, January 2008.
<http://www.w3.org/TR/rdf-sparql-query/>
- [18] Alexander, C., Ishikawa, S., Silverstein, M.: *A Pattern Language: towns, buildings, construction*. Center for Environmental Structure Series, vol. 2. Oxford University press, New York, 1977
- [19] Horridge, M., Drummond, N., Goodwin, J., Rector, A., Stevens, R., Wang, H.H: *The Manchester OWL syntax*, 2005.
http://owl-workshop.man.ac.uk/acceptedLong/submission_9.pdf
- [20] Racer Systems GmbH & Co. KG. *Racerpro users guide version 1.9*, 2005.
<http://www.racer-systems.com/products/racerpro/manual.phtml>