# Patterns for Programming in the Semantic Web

Luís Cruz-Filipe,Isabel Nunes,Graça Gaspar

DI–FCUL–TR–2012–06

DOI:10455/6889

(http://hdl.handle.net/10455/6889)

November 2012


Published at Docs.DI (http://docs.di.fc.ul.pt/), the repository of the Department of Informatics of the University of Lisbon, Faculty of Sciences.

# Patterns for Programming in the Semantic Web

Luís Cruz-Filipe* Isabel Nunes† Graça Gaspar†

October 11, 2012

**Abstract**

Originally proposed in the mid-90s, design patterns for software development played a key role in object-oriented programming not only in increasing software quality, but also by giving a better understanding of the power and limitations of this paradigm. Since then, several authors have endorsed a similar task for other programming paradigms, in the hope of achieving similar benefits.

In this paper we discuss design patterns for the Semantic Web, giving new insights on how existing programming frameworks can be used in a systematic way to design large-scale systems. The common denominator between these frameworks is the combination between different reasoning systems, namely description logics and logic programming. Therefore, we chose to work in a generalization of dl-programs that supports several (possibly different) description logics, expecting that our results will be easily adapted to other existing frameworks such as multi-context systems. This study also suggests new constructs to enforce legibility and internal structure of logic-based Semantic Web programs.

## 1 Introduction

In the mid-nineties, the Gang of Four's work on software design patterns [12] paved the way for important advances in software quality; presently, many valuable experienced designers' "best practices" are not only published but effectively used by the software development community. From very basic, abstract, patterns that can be used as building blocks of several more complex ones, to business-specific patterns and frameworks, dozens of design patterns have been proposed (e.g. [1, 10, 11, 19, 20, 22, 28]), establishing a kind of common language between development teams, which substantially enriches their communication, and hence the whole design process.

Most of the work around design patterns has been focused in the object-oriented paradigm, although some of the patterns are fundamental enough to be independent of the used modeling and programming paradigms. Some effort has also been made in adapting some of these best practices to other paradigms and in finding new paradigm-specific patterns [2, 13, 26, 29]. But all that glitters is not gold, and design patterns have also been criticized [13, 23, 25] e.g. by the unnecessary increase in complexity when innapropriately used, by somehow being a sign of the lack of essential features in a programming language, and by not producing reusable components. We consider that the benefits of design patterns greatly transcend their drawbacks, and carried the task of identifying several basic and other, more complex, patterns in the paradigm of dl-programs [8] – which join description logics with rules –, a powerful and expressive approach to reasoning over general knowledge bases or ontologies.

To the best of our knowledge, design patterns have not been studied in the framework of dl-programs, in spite of their importance. That is the goal of this paper. In order to enhance both the modular nature of dl-programs and the relevance of some important design principles, we introduce multi description logic programs (Mdl-programs), a straightforward generalization of dl-programs to accomodate for several description logics.

The remainder of the paper is structured as follows. Section 2 explains the context and goals of this work in more detail. Section 3 introduces the concepts that will be relevant throughout the presentation. Sections 4–7 present nine different design patterns, divided in three groups of growing complexity: elementary patterns (Section 4), derived patterns (Section 5) and elaborate patterns (Section 7), together

*Escola Superior Náutica Infante D. Henrique, Portugal; CMAF, Lisboa, Portugal
†Faculdade de Ciências, Universidade de Lisboa; LabMag, Lisboa, Portugal

with a realistic large-scale example illustrating their application (Sections 6 and 7). Finally, Section 8 summarizes the contributions presented earlier.

## 2 Motivation

The usefulness of combining description logics with rule-based reasoning systems led to the introduction of dl-programs [8, 7]. A dl-program is a pair $\mathcal{KB} = \langle \mathcal{L}, \mathcal{P} \rangle$, where $\mathcal{L}$ is a description logic knowledge base, which we will refer to simply as "knowledge base" from this point onwards, and $\mathcal{P}$ is a generalized logic program – a logic program[1] extended with dl-atoms in rules. The role of dl-atoms is to allow $\mathcal{P}$ to query $\mathcal{L}$, thereby establishing communication between both components. A dl-atom is of the form

$$DL[S_1\ op_1\ p_1, \ldots, S_m\ op_m\ p_m; Q](t)\,,$$

where each $S_i$ is either a concept, a role, or a special symbol in $\{=, \neq\}$; $op_i \in \{\uplus, \cup\}$[2]; each $p_i$ is a unary or binary predicate symbol depending on the corresponding $S_i$ being a concept or a role; and $Q(t)$ is a *dl-query*, that is, it is either a concept inclusion axiom $F$ or its negation $\neg F$, or of the form $C(t)$, $\neg C(t)$, $R(t_1, t_2)$, $\neg R(t_1, t_2)$, $=(t_1, t_2)$, $\neq (t_1, t_2)$, where $C$ is a concept, $R$ is a role, $t$, $t_1$ and $t_2$ are terms.

The operators $\uplus$ and $\cup$ are used to extend the knowledge base $\mathcal{L}$ locally. Intuitively, $S_k \uplus p_k$ (resp., $S_k \cup p_k$) increases $S_k$ (resp., $\neg S_k$) by the extension of $p_k$.

The program $\mathcal{P}$ is a set of dl-rules, which are rules of the form $a \leftarrow b_1, \ldots, b_k, \mathsf{not}\ b_{k+1}, \ldots, \mathsf{not}\ b_m$ where any $b_1, \ldots, b_m$ may be a dl-atom.

We illustrate the use of dl-programs by means of the following example from [17].

**Example 1.** Consider the dl-program $\mathcal{KB} = \langle \Sigma, \mathcal{P} \rangle$, where:

$$\Sigma: \quad (\geq 2\mathsf{PapToRev}.\top) \sqsubseteq \mathsf{Over} \qquad (i_1)$$

$$\mathsf{Over} \sqsubseteq \forall \mathsf{Super}^+.\mathsf{Over} \qquad (i_2)$$

$$\{(\mathsf{a}, \mathsf{b})\} \sqcup \{(\mathsf{b}, \mathsf{c})\} \sqsubseteq \mathsf{Super} \qquad (i_3)$$

$$\mathcal{P}: \quad \mathsf{good}(X) \leftarrow DL[;\mathsf{Super}](X, Y), \mathsf{not}\ DL[\mathsf{PapToRev} \uplus \mathsf{paper}; \mathsf{Over}](Y) \qquad (r_1)$$

$$\mathsf{over}(X) \leftarrow \mathsf{not}\ \mathsf{good}(X) \qquad (r_2)$$

$$\mathsf{paper}(\mathsf{b}, \mathsf{p}_1) \qquad (r_3)$$

$$\mathsf{paper}(\mathsf{b}, \mathsf{p}_2) \qquad (r_4)$$

We briefly recall this program's intended meaning as explained in [17]. Axioms $(i_1)$ and $(i_2)$ indicate that someone who has more than two papers to review is overloaded, and that an overloaded person causes all their supervised persons to be overloaded as well. Axiom $(i_3)$ defines the supervision hierarchy. Rule $(r_1)$ indicates that, if $X$ is supervising $Y$ and $Y$ is not overloaded, then $X$ is a good manager. Rule $(r_2)$ indicates that, if $X$ is not a good manager, then $X$ is overloaded.

The two components of a dl-program are kept independent, communicating only through dl-atoms. So, although the two components function separately, giving dl-programs nice modularity properties, there is a bidirectional flow of information via dl-atoms. However, the power of dl-atoms is limited. In some settings, one wants to share a predicate in both components, in the sense that it should be used for reasoning both in $\mathcal{L}$ and in $\mathcal{P}$. A typical example is applying closed-world reasoning to a concept or role in the open-world knowledge base $\mathcal{L}$. With this in mind, in [6] we proposed a global mechanism, which we called *lifting*, by means of which a predicate is effectively shared between the two components of the system and changes made in one of the components are automatically reflected in the other. In practical terms, lifting consists of applying two operations: *adding* some rules to $\mathcal{P}$, which allow the lifted concepts and roles of $\mathcal{L}$ to be visible in $\mathcal{P}$; and *changing* every dl-atom, porting the changes $\mathcal{P}$ makes to these concepts and roles back into $\mathcal{L}$ in a systematic, global way.

[1] Throughout this paper, we will assume that the logic programming language is Datalog$^\neg$, described in [17], which in particular contains negation-as-failure.

[2] The original definition of dl-programs included a third operator, but in the context of this paper that operator can be defined as an abbreviation of the other two.

When considered independently, the two operations used in the lifting construction have a clear semantics of their own. The rules that are added to $\mathcal{P}$ define new predicates whose only function is to observe their counterparts in $\mathcal{L}$, automatically reflecting any changes made on $\mathcal{L}$'s side; the systematic changes made to dl-atoms have a dual effect, guaranteeing that any changes made to those new predicates by $\mathcal{P}$ are automatically reflected in $\mathcal{P}$'s view of the knowledge base. Each of these bears striking similarities to what is known in the literature as the *observer* design pattern: a component of a program registers as an observer of another component when it wants to be informed of any changes made to the latter.

With this in mind, we decided to study design patterns in dl-programs, following what was already done for other programming paradigms – object-oriented [12], service-oriented [10], functional [2, 13, 25], logic [29] and others. As several of these authors observed, studying design patterns in different programming paradigms is far from being a trivial task: each paradigm has its specific features, meaning that patterns that are very straightforward in one paradigm can be very complex in another, and vice-versa. For example, a comparative study [25] of the implementation of 23 design patterns presented in an object-oriented setting [12] and in a functional environment concluded that, in 16 cases, the Lisp implementation was qualitatively simpler than its C$^{++}$ equivalent.

Looking at dl-programs, it is clear that they represent a completely different programming paradigm – not only are they closely related to the logic programming paradigm, but they involve description logic knowledge bases, in the presence of which the study of design patterns attains a different quality: on the one hand, some patterns become trivial (such as ***Façade***) or meaningless (such as ***Dynamic Binding*** or ***Singleton***), on the other hand some patterns pose totally new problems that have not been addressed in other paradigms where they do not arise (such as ***Proxy***, which we will discuss in Section 7).

We claim that dl-programs, being a dual-component system, with a description logic and a logic-based rule language, provide the adequate setting for the study of design patterns for the Semantic Web. We will, however, work not with the original dl-programs, but with a straightforward generalization that we will define precisely in Section 3: Mdl-programs (for Multi Description Logic programs), which incorporate not one, but a finite set of description logic knowledge bases. Mdl-programs are, therefore, multi-component systems connected by a logic program. This makes perfect sense when we consider some of the more elaborate design patterns, whose real power can only be appreciated in this more general setting. Furthermore, as we show, most theoretical results known for dl-programs immediately translate to similar results about Mdl-programs, thereby justifying the use of the latter.

There are two questions that immediately come to mind, though. In the first place, if one wishes to consider several ontologies, what are the advantages of using Mdl-programs instead of merging all the desired ontologies and use the result in a standard dl-program? Also, why define Mdl-programs and not work within other existing frameworks, such as the more general HEX-programs [9] or multi-context systems [3]? We feel it is important to provide satisfactory answers to these two pertinent issues before proceeding.

## 2.1 Keeping ontologies separate

The approach we will propose in the next section allows one to work with different ontologies at the same time, while keeping them separate from each other. We will show that the use of Mdl-programs allows us to exercise a very fine control on how the different knowledge bases interact (in particular, via design patterns).

One can naturally wonder whether it would not be simpler to merge all the relevant knowledge bases into a single, unified, ontology. There are a number of advantages to keeping them separate, however, which essentially coincide with the reasons universally invoked to defend modularity of large-scale systems. Indeed, it is much more convenient to have independent knowledge bases (which might even be physically separated, or independently managed) than a single, gigantic one.

Merging ontologies is in itself a mighty task with its own specific problems. Combining ontologies immediately presents a number of issues relating to consistency (not only logical, but also structural) that have to be addressed [5, 16].

Furthermore, ontologies are typically very wide-spectrum, and in practice one does not need to work with each of them in its entirety. Therefore, keeping them separate also reduces the number of compatibility issues one has to deal with to the essential minimum. Clearly, if there are two concepts in two different ontologies that should be identified (from a particular application's perspective) but are

logically inconsistent, this will always be a problem. However, if that particular inconsistency is irrelevant for the goal at hand (because the concepts involved are not used), then *not* merging the ontologies will simply not raise that question – and our approach allows us to bypass it altogether.

Also, the separation of the knowledge bases allows us to make the most of the positive aspects of each knowledge base – which is relevant if, say, one of them is very efficient at performing specific reasoning tasks, while another features richer concept and role constructions.

For these reasons, we feel that a setting that keeps all relevant knowledge bases separate (where their interaction is externally controlled) is preferable not only for the purposes of this paper, but also as a general principle.

### 2.2 Restricting to description logics

The need identified in [21] for the representation and manipulation of contextual information as a means to solve the problem of generality led to several proposals, namely the one of contexts as a tool for formalizing the locality of reasoning as first proposed in [14], where context is taken as "the set of facts used locally to prove a given goal plus the inference routines used to reason about them". This work and others following it (e.g. [15, 27, 4, 3]), use so-called *bridge-rules* to be able to represent the flow of information between contexts; the idea is that formulas are labelled with the context they belong to and inferences can be made that add new facts to a given context depending on facts coming from the others. In [27, 4], contexts are homogeneous (in what concerns the languages supported), and non-monotonic reasoning is possible. Heterogeneous contexts are supported in [15] and [3], the difference between these being that the former does not support non-monotonic reasoning. Hex-programs [9] were also proposed as a heterogeneous programming language for the Semantic Web, being designed for interoperating with heterogeneous sources via external atoms and for meta-reasoning via higher-order literals; those authors also studied, in that framework, conflict resolution in semantic integration of ontologies (using a "forgetting" construction).

Mdl-programs, presented in this paper, limit heterogeneity to two different frameworks: description logics for the knowledge bases part and logic programming for the rule part; the latter somehow represents the "conductor" that "coordinates" the other parts. However, they fully support non-monotonicity (even at the level of the description logic knowledge bases as will be seen later by application of a specific basic pattern). Mdl-programs are therefore a simpler framework than the alternatives discussed above – which is an advantage for our purpose; still, description logics *are* at the core of the Semantic Web, with a huge effort being currently invested in the interchange between OWL – an extension of the description logic SROIQ and a W3C recomendation [18] – and a diversity of rule languages, namely logic programming languages, via the the definition of RIF (Rule Interchange Format), which is also a W3C recommendation [24].

Furthermore, as we will show in the next section, Mdl-programs inherit all the good properties of dl-programs, which made them interesting in the first place, while simultaneously keeping enough key ingredients of multi-context systems and HEX-programs to make the study of design patterns general.

## 3 Interfacing between multiple DLs and logic programs

In this section we establish the grounds for our work, giving a formal definition of Mdl-programs and a summary of their properties.

### 3.1 Mdl-programs

We generalize the definition of dl-programs in [8], presented at the top of Section 2, to accommodate for several description logic knowledge bases. This is in line with [30], although we will stick to the original operators $\uplus$ and $\cup$ in dl-atoms – see the discussion at the end of this section for a more detailed explanation of this.

**Definition 1.** A *dl-atom* relative to a set of description logic knowledge bases $\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$ is

$$DL_i\left[S_1\ op_1\ p_1, \ldots, S_m\ op_m\ p_m; Q\right](t)\,,$$

which we often abbreviate to $DL_i[\chi; Q](t)$, where:

- $1 \leq i \leq n$;
- each $S_k$ is either a concept or a role from $\mathcal{L}_i$ or a special symbol in $\{=, \neq\}$;
- $op_k \in \{\uplus, \cup\}$;
- $p_k$ are the *input predicate symbols*, which are unary or binary predicate symbols depending on the corresponding $S_k$ being a concept or a role;
- $Q(t)$ is a dl-query in the language of $\mathcal{L}_i$.

The description logics underlying the $\mathcal{L}_i$s need not be the same. Note that we use the two operators $\uplus$ and $\cup$. The third operator in [8] can be defined in terms of $\cup$, and furthermore not including it simplifies the semantics of Mdl-programs, as was discussed in [30] in a slightly different context.

**Definition 2.** An *Mdl-program* (for Multi Description Logic program) is a pair $\langle \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}, \mathcal{P} \rangle$ where:

- each $\mathcal{L}_i$ is a description logic knowledge base;
- $\mathcal{P}$ is a set of (normal) *dl-rules*, i.e. rules of the form

$$a \leftarrow b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_p$$

  where any $b_j$ may be a dl-atom relative to $\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$.

The semantics of Mdl-programs is a straightforward generalization of the semantics for dl-programs. As usual, the Herbrand base of $\mathcal{P}$, denoted by $HB_{\mathcal{P}}$, contains all ground atoms built from predicate symbols and constants in $\mathcal{P}$.

**Definition 3.** An interpretation $I \subseteq HB_{\mathcal{P}}$ *satisfies* a ground atom $a$, $I \models a$, if:

- $a \in HB_{\mathcal{P}}$ and $a \in I$;
- $a$ is $DL_i[\chi; Q](c)$, with $\chi = S_1\, op_1\, p_1, \ldots, S_m\, op_m\, p_m$, and $L_i(I; \chi) \models Q(c)$, where $L_i(I; \chi) = L_i \cup \bigcup_{k=1}^{m} A_k(I)$ with

$$A_k(I) = \begin{cases} \{S_k(e) \mid p_k(e) \in I\} & \text{if } op_k = \uplus \\ \{\neg S_k(e) \mid p_k(e) \in I\} & \text{if } op_k = \cup \end{cases}.$$

From this concept, we can define answer set semantics and well-founded semantics for Mdl-programs exactly as for dl-programs. All the results in [8] and [7] hold. Also, Mdl-programs are a generalization of dl-programs in the following sense.

**Theorem 1.** Let $\langle \mathcal{L}, \mathcal{P} \rangle$ be a dl-program. Then:

1. a set $I \subseteq HB_{\mathcal{P}}$ is a strong answer set for $\langle \mathcal{L}, \mathcal{P} \rangle$ iff $I$ is a strong answer set for the Mdl-program $\langle \{\mathcal{L}\}, \mathcal{P} \rangle$;
2. $WFS(\langle \mathcal{L}, \mathcal{P} \rangle) = WFS(\langle \{\mathcal{L}\}, \mathcal{P} \rangle)$.

## 3.2 Mdl-programs with observers

We now define a syntactic construction on Mdl-programs, which generalizes the lifting construction in [6] as explained earlier: we identify the predicates from $\mathcal{L}$ being observed in $\mathcal{P}$, and the predicates from $\mathcal{P}$ being observed in $\mathcal{L}$.

**Definition 4.** An *Mdl-program with observers* is a tuple

$$\langle \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}, \mathcal{P}, \{\Lambda_1, \ldots, \Lambda_n\}, \{\Psi_1, \ldots, \Psi_n\} \rangle$$

where:

- $\langle \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}, \mathcal{P} \rangle$ is an Mdl-program;

- for $1 \leq i \leq n$, $\Lambda_i$ is a finite set of pairs $\langle S, p\rangle$ where $S$ is a concept, a role, or a negation of either, from $\mathcal{L}_i$ and $p$ is a predicate from $\mathcal{P}$;
- for $1 \leq i \leq n$, $\Psi_i$ is a finite set of pairs $\langle p, S\rangle$ where $p$ is a predicate from $\mathcal{P}$ and $S$ is a concept, a role, or a negation of either, from $\mathcal{L}_i$.

For each pair in $\Psi_i$ or $\Lambda_i$, if $S$ is a concept, then $p$ is a unary predicate; if $S$ is a role, then $p$ is a binary predicate. The sets $\Lambda_1, \ldots, \Lambda_n, \Psi_1, \ldots, \Psi_n$ will occasionally be referred to as the *observers* of $\langle\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}, \mathcal{P}\rangle$.

Intuitively, $\Lambda_i$ contains the concepts and roles in $\mathcal{L}_i$ that $\mathcal{P}$ needs to observe, whereas $\Psi_i$ contains the predicates in $\mathcal{P}$ that $\mathcal{L}_i$ wants to observe. Note that a specific symbol (be it a predicate, concept or role) may occur in different $\Psi_i$s or $\Lambda_i$s; this will in fact be crucial for some of the examples in the next sections.

For simplicity, when we consider Mdl-programs with observers that only have one knowledge base, we will omit the braces and refer to them as dl-programs with observers.

From an Mdl-program with observers we can obtain a standard Mdl-program in a straightforward way as follows.

**Definition 5.** Given the above Mdl-program with observers, it implicitly defines the Mdl-program $\left\langle\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}, \mathcal{P}^{\Psi_1,\ldots,\Psi_n}_{\Lambda_1,\ldots,\Lambda_n}\right\rangle$ where $\mathcal{P}^{\Psi_1,\ldots,\Psi_n}_{\Lambda_1,\ldots,\Lambda_n}$ is obtained from $\mathcal{P}$ by:

- adding the rule
$$p(X) \leftarrow DL_i[; S](X)$$
for each $\langle S, p\rangle \in \Lambda_i$;
- in each dl-atom $DL_i[\chi; Q](t)$ (including those added in the previous step), adding $S \uplus p$ to $\chi$ for each $\langle p, S\rangle \in \Psi_i$ and $S \uplus p$ to $\chi$ for each $\langle p, \neg S\rangle \in \Psi_i$.

## 3.3 Discussion

At this stage, we would like to point out some features of Mdl-programs with observers that are relevant for the remainder of this paper.

In the first place, we observe that we stick to the two original operators $\uplus$ and $\cup$ of dl-programs in e.g. [8], instead of adopting the more recent proposal of $\oplus$ and $\ominus$ as suggested in [30]. The motivation for this choice has to do with the clearer semantics thus obtained, namely in the case (not discussed in [8]) when a dl-atom extends a particular concept or role in more than one way – e.g. in $DL[P \uplus p, P \cup q; Q](t)$) –, which will often occur in our examples. From the semantics of $\uplus$ and $\cup$, this means that all instances of $p(X)$ will be added to the extent of $P$, and all instances of $q(X)$ will be added to the extent of $\neg P$.

There is a clear problem with this: if $p$ and $q$ are incompatible (for example, if $p(a)$ and $q(a)$ both hold), then the corresponding dl-query $Q$ will be posed to an inconsistent knowledge base (one that contains both $P(a)$ and $\neg P(a)$), which of course is not desirable. However, although this is an important point to take into consideration, it is not likely that a well-designed program would lead to this kind of situation. In fact, as we will see, a well thought-out application of the design patterns herein presented will help *preventing* such problems.

The alternative proposed by Wang [30] was to replace $\uplus$ and $\cup$ by two operators $\oplus$ and $\ominus$, where the first two of each pair are equivalent, but $P \ominus q$ removes instances of $q$ from the extent of $P$ instead of adding them to $\neg P$. In this way, inconsistency is avoided in many cases (although not always, since $\oplus$ still adds instances to the knowledge base). However, two other problems arise. First, the semantics of simultaneous extensions are not clear – in the example above, writing $DL[P \oplus p, P \ominus q; Q](t)$ would add and remove $P(a)$, it not being clear what the final result should be: if we keep the original knowledge base, then the context of the query is misleading, but if we apply the operators in order then $DL[P \oplus p, P \ominus q; Q](t)$ and $DL[P \ominus q, P \oplus p; Q](t)$ are not equivalent, which is very counter-intuitive. Second, the effect of removing instances from the knowledge base is not always obvious: consider the case where the knowledge base contains the axioms $P \sqsubseteq Q$ and $P(a)$, while the logic program contains the fact $x(a)$. What should the dl-atom $DL[Q \ominus x; Q](X)$ return?

We feel that these two problems, which are not discussed in [30], are severe enough to justify the use of the original operators of [8]. With the original operators, the possibility of reaching an inconsistent

environment (which is a feature of dl-programs) is the price to pay for having a clear semantics for dl-atoms.

A second comment that we feel is important to make regards the assymetry in the generalization from dl-programs to Mdl-programs. The reader might be wondering why we do not take this construction a step further, and define a system with several description logic knowledge bases and several logic programs. The answer lies in the assymetry already present in dl-programs themselves: the logic program is not only a component of the system, it is also the component responsible for interaction with the outside world. In particular, the dl-program's view of $\mathcal{L}$ is the view from each dl-atom – all changes made by $\mathcal{P}$ to the knowledge base only affect the logic program's view of it. Therefore, it is natural to consider several knowledge bases with a single logic program as the orchestrator (and where the outside world only "sees" that program's view of each knowledge base) rather than a more democratic system with several logic programs and several knowledge bases freely interacting with each other. This does not exclude that, in the future, it might be worth to study connecting patterns between two or more Mdl-programs (for instance, using a Mdl-program as a module of another Mdl-program, as a way to protect and restrict the access to certain knowledge bases).

A third aspect concerns additional similarities and differences between Mdl-programs and multi-context systems (apart from the ones discussed earlier). In both cases, we are faced with several independent components connected by a set of rules – a logic program, in the case of Mdl-programs, and the sets of bridge rules, in multi-context systems. However, all components of multi-context systems are observable from the outside world, since models of multi-context systems contain models of each individual component. It is also interesting to observe that the *flavour* of bridge rules is pretty much captured by the generalization to Mdl-programs with observers: the pairs in the observer sets achieve the effect of systematically extending a predicate, concept or role in one component with information from another component, in the same way bridge rules control the information flow within a multi-context system.

Finally, we would like to point out that the definition of lifting in [6] is a particular case of the more general construction we presented above: the dl-program with lifting $\langle \mathcal{L}, \mathcal{P}, \Gamma \rangle$ is equivalent to the Mdl-program with observers $\langle \{\mathcal{L}\}, \mathcal{P}, \{\Lambda\}, \{\Psi\} \rangle$ where $\Lambda = \{\langle S, p^+ \rangle, \langle \neg S, p^- \rangle \mid S \in \Gamma\}$ and $\Psi = \{\langle p^+, S \rangle, \langle p^-, \neg S \rangle \mid S \in \Gamma\}$. In fact, in Section 5 we will present lifting as a design pattern in itself – so this work generalizes that earlier paper. We refer the reader to [6] for a detailed motivation of the lifting construction and examples of its usage.

The presentation of the design patterns is structured as follows. The next two sections introduce the simplest design patterns. For each of them, we begin by presenting a motivational example and a possible solution; from this, we abstract the underlying pattern and how it should be dealt with in general. Section 6 then discusses a large-scale example where these design patterns can be found, in order to show how this methodology can be used in practice. Finally, Section 7 discusses more complex design patterns building on the example from Section 6.

# 4 Elementary design patterns

In this section, we introduce the elementary design patterns, which are the building blocks from which more complex patterns will be defined.

The definition of each design pattern is presented within the context of an Mdl-program with observers

$$\langle \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}, \mathcal{P}, \{\Lambda_1, \ldots, \Lambda_n\}, \{\Psi_1, \ldots, \Psi_n\} \rangle ,$$

even though for simplicity the examples in this section all assume $n = 1$.

## 4.1 Observing a knowledge base

The first design pattern we consider occurs when the logic program component wants to systematically import information from a knowledge base in order to define a predicate, keeping track of changes made to the relevant concept or role in its defining component.

**Scenario.** The National Zoo keeps an ontology containing information about all the animals currently living in its premises, as well as several rules pertaining to the characteristics of the different species. This ontology $\mathcal{L}$ is used also by the company that provides food for all the zoo's feline inhabitants, coupled with a rule-based program $\mathcal{P}$ in a dl-program $\langle \mathcal{L}, \mathcal{P} \rangle$. In this program, it is necessary that $\mathcal{P}$ be constantly updated with the information about the current feline population of the zoo, hence the following rule was included.

$$\mathsf{feline}(X) \leftarrow DL[;\mathsf{feline}](X)$$

Note that, although their name is the same, the feline in the head of the rule is a predicate from $\mathcal{P}$ whereas the feline in the body is a concept from $\mathcal{L}$. No confusion can arise from this, since the only place where predicates from $\mathcal{L}$ may occur in $\mathcal{P}$ is within dl-atoms.

The company could have obtained the same result by removing the above rule from $\mathcal{P}$ (thus leaving feline undefined) and working with the dl-program with observers $\langle \mathcal{L}, \mathcal{P}, \{\langle \mathsf{feline}, \mathsf{feline} \rangle\}, \emptyset \rangle$. In both cases, any changes made to feline in the zoo's ontology will be automatically reproduced in $\mathcal{P}$. Although the effect is the same, the latter option makes it very clear that feline in $\mathcal{P}$ is an observer of its homonym from $\mathcal{L}$, making the global dl-program with observers easier to understand and maintain.

---

**Pattern *Observer Down.***

*Problem.* A predicate $p$ from $\mathcal{P}$ needs to be updated every time the extent of a concept or role $S$ (of the same arity as $p$) in $\mathcal{L}_i$ is changed.

*Solution.* Add the pair $\langle S, p \rangle$ to $\Lambda_i$.

---

## 4.2 Dynamically modifying the view of a knowledge base

A second scenario occurs when one of the description logics' functionality relies on the observation of a predicate from $\mathcal{P}$. Consider the following example.

**Scenario.** A sweets distributor provides an ontology to all the shops selling its products containing general information and reasoning about the different types of sweets on sale. Being a generic knowledge base, this ontology has no concrete facts: in particular, the extent of the predicate sweet is empty. Each shop interacts with the central ontology by means of a particular dl-program $\mathcal{KB} = \langle \mathcal{L}, \mathcal{P} \rangle$, where $\mathcal{L}$ is the central ontology and $\mathcal{P}$ contains facts detailing the specific brands of sweets being sold in that particular location. In order for the program to function appropriately, sweet (in $\mathcal{L}$) needs to be automatically updated whenever a new fact is added to the relevant predicate in $\mathcal{P}$, so that this fact can be taken into account.

Sweets In Heaven uses a predicate forSale in $\mathcal{P}$ to store the information about the brands available at their shop. In order to use $\mathcal{L}$ to reason about these products, they need to replace all dl-atoms $DL[\chi; Q](t)$ in $\mathcal{P}$ with

$$DL[\mathsf{sweet} \uplus \mathsf{forSale}, \chi; Q](t)$$

meaning that $\mathcal{L}$ behaves as though its predicate sweet was actually extended with all sweets for sale at Sweets In Heaven. This is exactly the same as replacing $\mathcal{KB}$ by the dl-program with observers $\langle \mathcal{L}, \mathcal{P}, \emptyset, \{\langle \mathsf{forSale}, \mathsf{sweet} \rangle\} \rangle$.

---

**Pattern *Observer Up.***

*Problem.* A concept or role $S$ from $\mathcal{L}_i$ needs to be updated every time the extent of a predicate $p$ (of the same arity as $S$) in $\mathcal{P}$ is changed.

*Solution.* Add the pair $\langle p, S \rangle$ to $\Psi_i$.

---

### 4.3 Closing the world

The third building block addresses a very typical situation in ontology design and usage: in order for a concept or role to work as expected, it should be given closed-world semantics.

**Scenario.** A digital library has an ontology with information about all the titles it contains. One of the concepts it defines is forSale, applicable to electronic books that are available for download for a price. However, description logics are not able to perform closed-world reasoning, so a query to the ontology about ¬forSale is not sufficient to inform a user that a given title is *not* for sale.

With this in mind, the software developers working for the digital library integrated the ontology $\mathcal{L}$ in the framework of a dl-program $\langle \mathcal{L}, \mathcal{P} \rangle$, where $P$ is a very simple program containing only the following rules.

$$\mathsf{forSale}(X) \leftarrow DL[;\mathsf{forSale}](X)$$
$$\mathsf{notForSale}(X) \leftarrow \mathit{not}\ \mathsf{forSale}(X)$$

The information about titles that are not for sale is now directly available in the predicate notForSale of $\mathcal{P}$.

Using observers, the resulting program is $\langle \mathcal{L}, \mathcal{P}, \{\langle \mathsf{forSale}, \mathsf{forSale}\rangle\}, \{\langle \mathsf{notForSale}, \neg\mathsf{forSale}\rangle\}\rangle$, where $\mathcal{P}$ contains only the second of the above rules.

Furthermore, if the library later wants to extend $\mathcal{P}$ further, then all dl-atoms must be of the form $DL[\mathsf{forSale} \uplus \mathsf{notForSale}, \chi; Q](t)$ – thus querying $\mathcal{L}$ extended with the desired closed-world interpretation of forSale. This is the reason for the pair in the second observer set of the dl-program with observers defined above.

---

**Pattern *Closed-world.***

*Problem.* The concept (or role) $S$ from $\mathcal{L}_i$ should follow closed-world semantics.

*Solution.*

Choose predicate symbols $s^+, s^-$ not used in $\mathcal{P}$.

Add $\langle S, s^+\rangle$ to $\Lambda_i$.

Add $\langle s^-, \neg S\rangle$ to $\Psi_i$.

Add the rule $s^-(X) \leftarrow \mathit{not}\ s^+(X)$ to $\mathcal{P}$.

---

# 5 Derived design patterns

We now present a second set of general-purpose design patterns that can be seen as organized combinations of the previous ones, but are also useful as components of more complex patterns.

Throughout this section all patterns are again defined within the context of an Mdl-program with observers

$$\langle \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}, \mathcal{P}, \{\Lambda_1, \ldots, \Lambda_n\}, \{\Psi_1, \ldots, \Psi_n\}\rangle\,.$$

### 5.1 Flexible definitions

We now introduce a design pattern that allows one to define a predicate in $\mathcal{P}$ abstracting from how it is represented in the knowledge bases.

**Scenario.** Vineyards are a major tourist atraction of Wineland. Knowing this, the country's Tourist Information Office has decided to combine information from the three major regional wine producers' associations in an Mdl-program $\mathcal{KB} = \langle\{\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3\}, \mathcal{P}\rangle$, where the $\mathcal{L}_i$s are the knowledge bases containing information about the wines produced in each region, by means of which tourists can access the information present in all three.

Since the three knowledge bases were developed independently, the need arose for a predicate unifying the concept of wine, which is distributed among them. Therefore, $\mathcal{P}$ needs to import all the information from the corresponding predicates wine from $\mathcal{L}_1$, wineBrand from $\mathcal{L}_2$ and (very unimaginatively) product from $\mathcal{L}_3$.

To unify these three concepts in a single predicate in $\mathcal{P}$, the Tourist Information Office included the three following rules in its program.

$$\begin{aligned}
\mathsf{wine}(X) &\leftarrow DL_1[;\mathsf{wine}](X)\\
\mathsf{wine}(X) &\leftarrow DL_2[;\mathsf{wineBrand}](X)\\
\mathsf{wine}(X) &\leftarrow DL_3[;\mathsf{product}](X)
\end{aligned}$$

This amounts to replacing $\mathcal{KB}$ by the Mdl-program with observers $\langle\{\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3\}, \mathcal{P}, \{\Lambda_1, \Lambda_2, \Lambda_3\}, \{\emptyset, \emptyset, \emptyset\}\rangle$ where

$$\begin{aligned}
\Lambda_1 &= \{\langle\mathsf{wine}, \mathsf{wine}\rangle\}\\
\Lambda_2 &= \{\langle\mathsf{wineBrand}, \mathsf{wine}\rangle\}\\
\Lambda_3 &= \{\langle\mathsf{product}, \mathsf{wine}\rangle\}
\end{aligned}$$

---

**Pattern *Polymorphic Entities.***

*Problem.* In $\mathcal{P}$ there is a predicate $p$ whose instances are inherited from concepts or roles $S_1, \ldots, S_k$ where $S_j$ comes from the knowledge base $\mathcal{L}_{\varphi(j)}$.

*Solution.* For each $j$, add the pair $\langle S_j, p\rangle$ to $\Lambda_{\varphi(j)}$.

---

Note that ***Polymorphic Entities*** consists of a combined application of several ***Observer Down***, all with the same observer predicate in $\mathcal{P}$.

This pattern captures the essences of the Polymorphism and Dynamic Binding patterns from object-oriented programming [19]: these patterns deal (at different levels) with definitions that are kept separate from their usage. In typed languages, this poses a number of technical issues that require careful planning. In the world of logic programming, however, several of these difficulties do not exist: predicates are not typed, so when using them in bodies of rules one does not have to worry about how they are defined; and their definitions are very simple to change.

In ***Polymorphic Entities***, we deal with a concept that is kept as independent as possible from its definition. Instead of defining clauses, the instances are plugged in through the use of Mdl-programs with observers, thus externalizing the definition of the predicate – in the spirit of Dynamic Binding. The possibility of using different concepts (eventually from different knowledge bases) captures the essence of Polymorphism.

## 5.2 Interconnecting description logics

Another variant of the Observer design pattern occurs when a description logic's functionality relies on the observation of a predicate in a *different* description logic; this can be achieved by combining both the ***Observer Up*** and ***Observer Down*** patterns, thus making the logic program $\mathcal{P}$ a mediator. A particular case arises when an ontology designed primarily for reasoning interacts with a knowledge base that is mostly about particular instances. In practice, this design pattern appears mostly in combination with ***Definitions with Holes***, so we will postpone the presentation of a concrete scenario to the next subsection.

---

**Pattern *Transversal Observer*.**

*Problem.* A concept (or role) $S$ from $\mathcal{L}_i$ needs to be updated every time the extent of a concept (resp. role) $R$ from $\mathcal{L}_j$ is changed (with $i \neq j$).

*Solution.*

Choose a predicate symbol $p$ not used in $\mathcal{P}$.

Add $\langle R, p\rangle$ to $\Lambda_j$.

Add $\langle p, S\rangle$ to $\Psi_i$.

---

Note that ***Transversal Observer*** consists of a combined application of ***Observer Up*** together with ***Observer Down***, relative to two different description logics.

## 5.3 Underspecification

In a modular system, it makes perfect sense to use concepts whose definition is left to other components. This is the motivation for the next design pattern.

**Scenario.** A company is designing an ontology to help with school management. One of the rules included in the ontology is

$$\mathsf{canAskDiplomma} \sqsubseteq \mathsf{graduated} \sqcap \neg \mathsf{onHold}\,,$$

stating that a student can ask for his diplomma after graduating if he has no pending issues with the school. Although graduated is a predicate defined in this ontology, the company has no way of knowing *a priori* which criteria to use to decide whether a student still has pending issues with the school, as this varies from school to school.

Therefore, the ontology will not have any information about instances of onHold – the definition of this concept will be left to other components that will be integrated with the ontology through an Mdl-program via ***Observer*** or ***Polymorphic Entities***.

---

**Pattern *Definitions with Holes*.**

*Problem.* In $\mathcal{L}_i$ there is a concept or role $S$ needed for reasoning but its definition will be in $\mathcal{L}_j$ (with $i \neq j$) or $\mathcal{P}$.

*Solution.*

Use $S$ in $\mathcal{L}_i$ without defining it (so the extent of $S$ is empty).

Later, connect $S$ to its definition using ***Observer Up***, ***Observer Down*** or ***Transversal Observer***, possibly coupled with ***Polymorphic Entities***.

---

In our example, suppose the company's ontology $\mathcal{L}_1$ is integrated in an Mdl-program $\langle\{\mathcal{L}_1, \mathcal{L}_2\}, \mathcal{P}\rangle$, where $\mathcal{L}_2$ is the school's own knowledge base (essentially a database containing information about the school's students) and $\mathcal{P}$ contains rules connecting both and performing other reasoning tasks. In that school, a student is *onHold* if he is owing money to the school (stored in $\mathcal{L}_2$ by means of the concept inDebt); hence, one can apply ***Transversal Observer*** to define onHold as an observer of inDebt. In a different, elite, school, the same concept is implemented differently, since the school forces its students to get extra credit by attending at least five seminars; so here onHold is an ***Observer*** of $(\leq_4 \mathsf{attendedSeminar}) \sqcup \mathsf{inDebt}$. Note that, in this situation, the school's knowledge base would need to provide an auxiliary concept equivalent to this condition due to the restrictions on the syntax of dl-queries.

This pattern corresponds to the Template Method pattern of object-oriented programs [12], and to the Programming with Holes technique of [22].

In the next section we will show an example where the holes are filled in by resorting to ***Polymorphic Entities***.

## 5.4 Distributed definitions

In [6], we presented lifting as a first-class mechanism encapsulating a set of changes to a dl-program working together for a specific purpose. We now present it in a more abstract and general way as a design pattern. Lifting occurs when a predicate's definition is split between different components of the Mdl-program.

**Scenario.** A company sells cars in several dealers across town. Each dealer has access to the company's car knowledge base $\mathcal{L}$, which specifies several properties of the available models and the dealers which sell them. One of the dealers with the company has his own information stored as a rule-based program $\mathcal{P}$. To integrate both systems, he uses a dl-program; however, he still needs to share some concepts between both components. For example, the concept $\mathsf{car}$, whose instances include all cars centrally available for sale, is defined in $\mathcal{L}$, but $\mathcal{P}$ has information about second-hand cars that occasionally go through the store, stored in a predicate $\mathsf{secondHandCar}$ and which the dealer wants to reason about in his view of $\mathcal{L}$; $\mathcal{P}$ also has information about motor bikes, that also occasionally go through the store, stored in a predicate $\mathsf{motorBike}$, and which should be considered distinct from cars, when reasoning in $\mathcal{P}$'s view of $\mathcal{L}$. In order to achieve full integration between the two components, he added two new predicates $\mathsf{car}^+$ and $\mathsf{car}^-$ to $\mathcal{P}$, connecting them to his predicates by means of the two rules

$$\mathsf{car}^+(X) \leftarrow \mathsf{secondHandCar}(X)$$
$$\mathsf{car}^-(X) \leftarrow \mathsf{motorBike}(X)$$

and with $\mathsf{car}$ from $\mathcal{L}$ via the two rules

$$\mathsf{car}^+(X) \leftarrow DL[;\mathsf{car}](X)$$
$$\mathsf{car}^-(X) \leftarrow DL[;\neg\mathsf{car}](X)\,.$$

Furthermore, every dl-atom $DL[\chi;Q](t)$ needs to be replaced by $DL[\mathsf{car} \uplus \mathsf{car}^+, \mathsf{car} \uplus \mathsf{car}^-, \chi; Q](t)$. In this way, one can use positive and negative facts about $\mathsf{car}$s both in $\mathcal{L}$ and in $\mathcal{P}$. The definition of $\mathsf{car}$ is truly split between $\mathcal{L}$ (which contains all the relationships between this and the other concepts in the knowledge base) and $\mathcal{P}$ (which is where the information about second-hand cars and motor bikes, which are not cars, is fed into it).

Using Mdl-programs with observers, the same construction can be designed in a simpler way: the only rules one needs to include in $\mathcal{P}$ are the two first ones (relating $\mathsf{car}^+$ with $\mathsf{secondHandCar}$ and $\mathsf{car}^-$ with $\mathsf{motorBike}$), while the rest is achieved by taking $\Lambda = \{\langle \mathsf{car}, \mathsf{car}^+\rangle, \langle \neg\mathsf{car}, \mathsf{car}^-\rangle\}$ and $\Psi = \{\langle \mathsf{car}^+, \mathsf{car}\rangle, \langle \mathsf{car}^-, \neg\mathsf{car}\rangle\}$.

---

**Pattern *(Named) Lifting.***

*Problem.* The definition of a predicate should be distributed among some of the $\mathcal{L}_i$s (in the form of concepts or roles $S_i$) and $\mathcal{P}$ (in the form of two predicates $p^+$ and $p^-$, corresponding to the predicate and its negation).

*Solution.*

For each $i$, add $\langle S_i, p^+\rangle$ and $\langle \neg S_i, p^-\rangle$ to $\Lambda_i$.

For each $i$, add $\langle p^+, S_i\rangle$ and $\langle p^-, \neg S_i\rangle$ to $\Psi_i$.

---

***Named Lifting*** generalizes the original lifting construction of [6] because it allows the programmer to choose the names for the predicates in $\mathcal{P}$. We will refer to it simply as ***Lifting*** when no confusion may arise between the earlier construction and the design pattern.

It should come as no surprise that ***Lifting*** consists simply of a set of applications of ***Observer Up*** and ***Observer Down*** with the same predicate in $\mathcal{P}$ observing and being observed simultaneously by all the $S_i$s involved in the pattern. However, ***Lifting*** is essentially different from ***Observer***: in ***Observer***, a predicate is defined in *one* component and used in others; in ***Lifting***, not only the usage, but also the *definition* of the predicate is split among several components, so that one must look at the whole

Mdl-program to understand it. This is also part of the reason to include the negations of the predicates involved in the observers: the different predicates must all end up with the same semantics.

It is possible to apply ***Lifting*** when $\mathcal{P}$ does not participate in the predicate's definition. In this case, $\mathcal{P}$ is simply a mediator, and $p^+$ and $p^-$ can be any fresh predicate names.

# 6 A comprehensive example

In this section, we show how the different design patterns introduced so far can be used in a realistic example, making several programming tasks much more systematic. This example will also be used in the next section as a context in which to introduce more sophisticated design patterns.

**Scenario.** The software developers at WishYouWereThere travel agency decided to develop an Mdl-program to manage several of the agency's day-to-day tasks. Currently, WishYouWereThere has two active partnerships: one with an aviation company, another with a hotel chain. Therefore, the Mdl-program to be developed uses three different ontologies:

- $\mathcal{L}_A$ is a generic accounting ontology for travel agencies, which is commercially available, and which contains all sorts of rules relating concepts relevant for the business. This ontology is strictly terminological, containing no specific instances of its concepts and roles.
- $\mathcal{L}_F$ is the aviation partner's knowledge base, containing information not only about available flights between different destinations, but also about clients who have already booked flights with that company.
- $\mathcal{L}_H$ is a similar knowledge base pertaining to the hotels owned by the partner hotel chain.

One of the points to take into consideration is that the resulting Mdl-program with observers

$$\langle \{\mathcal{L}_A, \mathcal{L}_F, \mathcal{L}_H\}, \mathcal{P}, \{\Lambda_A, \Lambda_F, \Lambda_H\}, \{\Psi_A, \Psi_F, \Psi_H\} \rangle$$

should be easily extended so that the travel agency can establish partnerships with more aviation companies and hotel chains, as long as those provide their own knowledge bases. At the end of this section, we will show how the systematic use of design patterns and observers helps towards achieving this goal.

By establishing partnerships, WishYouWereThere's client basis is extended with all the clients who have booked services of its partners. In this way, promotions made available by either partner are automatically offered to every partner's clients, as long as the bookings are made through the travel agency. In return, the partners get publicity and more clients, since a person may be tempted to fly with their company or book their hotel due to these promotions, thereby becoming also their clients.

**Updating the client database.** Ensuring that each partner's clients automatically become WishYouWereThere's clients can be achieved by noting that this is exactly the problem underlying ***Observer Down***. Assuming $\mathcal{L}_F$ and $\mathcal{L}_H$ have concepts Flyer and Guest, respectively, identifying their clients, and that the agency's clients will be stored as a predicate client in $\mathcal{P}$, all that needs to be done is to register client as an observer of Flyer and Guest, which, according to the pattern, is achieved by ensuring

$$\langle \mathsf{Flyer}, \mathsf{client} \rangle \in \Lambda_F \qquad \langle \mathsf{Guest}, \mathsf{client} \rangle \in \Lambda_H \, .$$

**Identifying pending payments.** The designers of $\mathcal{L}_A$ resorted intensively to ***Definitions with Holes***, since many of the concepts they use can only be defined in the presence of a concrete client database. In particular, $\mathcal{L}_A$ contains a role hasDebt, which computes the total amount owed by a client from information about specific purchases he has made and not paid for so far. The ontology collects this information from the role toPay, about which it contains no membership axioms.

For example, if $\mathcal{L}_A$ is enriched with $\mathsf{toPay}(\mathsf{John}, 500e)$ and $\mathsf{toPay}(\mathsf{John}, 300e)$, then querying it for hasDebt would return (possibly among others) the information $\mathsf{hasDebt}(\mathsf{John}, 800e)$.

There are two ways of completing this definition. The more direct one stems from noting that toPay should be an observer of adequate roles in $\mathcal{L}_F$ and $\mathcal{L}_H$. We will assume that these roles are mustPayFlight

and stayToPay. Applying twice ***Transversal Observer*** (which is the adequate pattern), one needs to ensure that

$$\begin{aligned}\langle \mathsf{mustPayFlight}, \mathsf{toPayF}\rangle &\in \Lambda_F\\ \langle \mathsf{toPayF}, \mathsf{toPay}\rangle &\in \Psi_A\\ \langle \mathsf{stayToPay}, \mathsf{toPayH}\rangle &\in \Lambda_H\\ \langle \mathsf{toPayH}, \mathsf{toPay}\rangle &\in \Psi_A\,.\end{aligned}$$

The major drawback of this solution is that it requires adding two dummy predicates to $\mathcal{P}$ whose only purpose is to serve as go-between from both knowledge bases to $\mathcal{L}_A$.

An alternative solution is to create a single auxiliary predicate toPay in $\mathcal{P}$ and make toPay from $\mathcal{L}_A$ an observer of this predicate applying ObserverUp. In turn, we use the ***Polymorphic Entity*** pattern to connect toPay to mustPayFlight and stayToPay. The resulting Mdl-program with observers is such that:

$$\begin{aligned}\langle \mathsf{mustPayFlight}, \mathsf{toPay}\rangle &\in \Lambda_F\\ \langle \mathsf{stayToPay}, \mathsf{toPay}\rangle &\in \Lambda_H\\ \langle \mathsf{toPay}, \mathsf{toPay}\rangle &\in \Psi_A\,.\end{aligned}$$

As we will discuss later, this solution will also simplify the process of adding new partners to the agency.

**Offering promotions.** WishYouWereThere offers a number of promotions to its special clients. For example, in February the agency offers them a 20% discount on all purchases. Because of the partnership, the concept of special client is distributed among all partners: a client is a special client if it fulfills one of the partners' requirements – e.g. having traveled some number of miles with the airline partner, or booked a family holiday in one of the partner's hotels, or bought one of the agency's pricey packages. The partnership protocol requires that each knowledge base provide a concept identifying which clients are eligible for promotions, so that the partners can change these criteria without requiring WishYouWereThere to change its program.

This is a situation where the ***Lifting*** design pattern applies. Assuming that $\mathcal{L}_F$ uses topClient for its special clients, $\mathcal{L}_H$ uses gold and $\mathcal{P}$ defines special, these three predicates are given the same semantics through ***Lifting***. Intuitively, this means that all three concepts equally denote *all* special clients, regardless of where they originate. The application of the pattern translates to

$$\begin{aligned}\langle \mathsf{topClient}, \mathsf{special}\rangle &\in \Lambda_F & \langle \neg\mathsf{topClient}, \mathsf{notSpecial}\rangle &\in \Lambda_F\\ \langle \mathsf{special}, \mathsf{topClient}\rangle &\in \Psi_F & \langle \mathsf{notSpecial}, \neg\mathsf{topClient}\rangle &\in \Psi_F\\ \langle \mathsf{gold}, \mathsf{special}\rangle &\in \Lambda_H & \langle \neg\mathsf{gold}, \mathsf{notSpecial}\rangle &\in \Lambda_H\\ \langle \mathsf{special}, \mathsf{gold}\rangle &\in \Psi_H & \langle \mathsf{notSpecial}, \neg\mathsf{gold}\rangle &\in \Psi_H\end{aligned}$$

Furthermore, in order to determine whether a particular client is entitled to promotions, it is useful to give closed-world semantics to these predicates. Since they are all equivalent, we can do this very simply in $\mathcal{P}$ by adding the rule

$$\mathsf{notSpecial}(X) \leftarrow \mathit{not}\ \mathsf{special}(X)\,.$$

Note that we did not need to apply the ***Closed-world*** pattern because special is a predicate from $\mathcal{P}$, where the semantics is closed-world: the application of ***Lifting*** ensures that gold and topClient, being equivalent predicates, also have closed-world semantics.

In order for one of the partner companies to make its clients eligible for special promotions, its ontology just needs to contain inclusion axioms partially characterizing special clients. For example, one could have

$$\begin{aligned}\exists \mathsf{flies.10000OrMore} \sqsubseteq \mathsf{topClient} &\quad \in \mathcal{L}_F\\ \mathsf{familyBooking} \sqsubseteq \mathsf{gold} &\quad \in \mathcal{L}_H\\ \mathsf{special}(X) \leftarrow \mathsf{booked}(X,Y), \mathsf{expensive}(Y) &\quad \in \mathcal{P}\end{aligned}$$

A subtle issue now appears regarding the consistency problems that may arise from the use of the ***Lifting*** pattern. Since this pattern identifies concepts from different knowledge bases, it does not *a priori*

guarantee that the resulting knowledge bases are consistent, as we already discussed in [6]. However, there are particular cases where consistency is guaranteed; one of them is when none of the knowledge bases is allowed to contain instances of the negated concepts being lifted. In this setting, this means that no partner is allowed to deny privileges to WishYouWereThere's clients explicitly via its dedicated concept (e.g. by including axioms such as $\neg\mathsf{topClient}(\mathsf{John})$ or $\mathsf{young} \sqsubseteq \neg\mathsf{gold}$). This is perfectly in line with the partnership protocol as stated above.

An example of a promotion offered by WishYouWereThere to special clients would be

$$20\%\mathsf{Discount}(X) \leftarrow \mathsf{special}(X)\,.$$

All special clients will benefit from this discount, regardless of who (the travel agency, the hotel partner or the aviation company) decided that they should be special clients.

However, in some cases partners may want to deny their promotions to particular clients. For example, the aviation company is offering 100 bonus miles to special costumers booking a flight on a Tuesday, but this promotion does not apply to its workers.

In order to allow this kind of situation, partners may define a dedicated concept identifying the non-eligible clients. Since all clients external to that partner are automatically eligible, this concept needs to have closed-world semantics so that (in our example) $\mathcal{L}_F$ can include the rules

$$\begin{aligned} \mathsf{100BonusMiles} &\sqsubseteq \mathsf{topClient} \sqcap \neg\mathsf{blocked} \\ \mathsf{worker} &\sqsubseteq \mathsf{blocked} \end{aligned}$$

still giving the promotion to all clients from the other partners. Although each knowledge base can enforce this semantics in its domain, in order to extend it to other clients the ***Closed-World*** pattern must be applied, so we will have

$$\begin{aligned} \langle \mathsf{blocked}, \mathsf{blockedF} \rangle &\in \Lambda_F \\ \langle \mathsf{nonBlockedF}, \neg\mathsf{blocked} \rangle &\in \Psi_F \\ \mathsf{nonBlockedF}(X) \leftarrow \mathit{not}\ \mathsf{blockedF}(X) &\in \mathcal{P} \end{aligned}$$

Suppose that airline employee Ann qualifies for WishYouWereThere promotions because she spent three weeks in Jamaica with her husband and their five children, hence $\mathsf{gold}(\mathsf{Ann})$ holds in $\mathcal{L}_H$ and therefore Ann is a special client. She is therefore eligible for WishYouWereThere's promotions, but she will still not earn the bonus miles because it is $\mathcal{L}_F$ who decides whether someone gets that particular promotion, and even though $\mathsf{topClient}(\mathsf{Ann})$ holds that knowledge base will not return $\mathsf{100BonusMiles}(\mathsf{Ann})$. However, she will earn the $20\%\mathsf{Discount}$, since it is offered directly by WishYouWereThere.

**Adding new partnerships.** We now discuss briefly how new partners can be easily added to the system later on, as this illustrates quite well the advantages of working both with design patterns and in the context of Mdl-programs with observers.

Summing up what we have so far relating to the partnerships, the sets $\Lambda_F, \Lambda_H, \Psi_F$ and $\Psi_H$ are as follows.

$\Lambda_F$: $\langle \mathsf{Flyer}, \mathsf{client} \rangle$
$\langle \mathsf{mustPayFlight}, \mathsf{toPay} \rangle$
$\langle \mathsf{topClient}, \mathsf{special} \rangle$
$\langle \neg\mathsf{topClient}, \mathsf{notSpecial} \rangle$
$\langle \mathsf{blocked}, \mathsf{blockedF} \rangle$

$\Lambda_H$: $\langle \mathsf{Guest}, \mathsf{client} \rangle$
$\langle \mathsf{stayToPay}, \mathsf{toPay} \rangle$
$\langle \mathsf{gold}, \mathsf{special} \rangle$
$\langle \neg\mathsf{gold}, \mathsf{notSpecial} \rangle$
$\langle \mathsf{blocked}, \mathsf{blockedH} \rangle$

$\Psi_F$: $\langle \mathsf{special}, \mathsf{topClient} \rangle$
$\langle \mathsf{notSpecial}, \neg\mathsf{topClient} \rangle$
$\langle \mathsf{nonBlockedF}, \neg\mathsf{blocked} \rangle$

$\Psi_H$: $\langle \mathsf{special}, \mathsf{gold} \rangle$
$\langle \mathsf{notSpecial}, \neg\mathsf{gold} \rangle$
$\langle \mathsf{nonBlockedH}, \neg\mathsf{blocked} \rangle$

Also, the application of the design patterns added the following rules to $\mathcal{P}$.

$$\mathsf{nonBlockedF}(X) \leftarrow \mathit{not}\ \mathsf{blockedF}(X)$$
$$\mathsf{nonBlockedH}(X) \leftarrow \mathit{not}\ \mathsf{blockedH}(X)$$

The similarity between $\Lambda_F$ and $\Lambda_H$, and between $\Psi_F$ and $\Psi_H$, is a clear illustration of the changes required when future partners of WishYouWereThere are added to the system. Furthermore, the names they use for each concept or role are not relevant – they just need to indicate how they identify their clients, their clients' debts, their special clients, and the clients they wish to exclude from their promotions.

# 7 Elaborate design patterns

In this section, we give examples of more complex design patterns that show how the simpler ones discussed previously can work together towards a common goal. This presentation has no claims to being exhaustive; the reader is invited to study how other known design patterns could be adapted to the setting of Mdl-programs in a similar way. As before, all patterns are defined within the context of an Mdl-program with observers

$$\langle \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}, \mathcal{P}, \{\Lambda_1, \ldots, \Lambda_n\}, \{\Psi_1, \ldots, \Psi_n\} \rangle .$$

## 7.1 Expediting component replacement

In some contexts, a component of a system may not be known or available at the time of implementation of others, yet it is necessary to query it. A way to get around this is to use a prototype knowledge base that will later on be connected to the concrete component in a straightforward way.

The same problem may also arise if one wishes to be able to replace a knowledge base with another with a similar purpose, but whose concept and role names may be different.

**Scenario.** With the expansion of WishYouWereThere's business, requests started coming in for organized group tours. Therefore, the agency's directors are working on establishing a collaboration with a (single) bus rental company that will provide transportation for groups of fifty people.

The relationship between the travel agency and the bus company is not that of partnership, but of outsourcing. This means that, on the one side, there will be only one bus company involved in the process, and, on the other side, the specific company may change in the future.

To comply with these restrictions, the software developers added an empty knowledge base $\mathcal{L}_I$ which only fixes the names of the concepts and roles being used. In extending $\mathcal{P}$, they implemented all queries to the bus partner as dl-queries to this interface knowledge base, such as:

$$\begin{aligned} \mathsf{busAvailable}(B, C, D) \leftarrow & DL_I[;\mathsf{operatesOnCity}](C), \\ & DL_I[;\mathsf{busOnCity}](B, C), \\ & DL_I[;\mathsf{freeOnDay}](B, D) . \end{aligned}$$

The concepts and roles of $\mathcal{L}_I$'s syntax provide the interface that will be used to communicate with the bus company's knowledge base. In order to couple a concrete knowledge base $\mathcal{L}_B$, the bus company needs to provide the concepts and roles from $\mathcal{L}_B$ that correspond to the names of the interface's concepts and roles, e.g. a predicate freeBus corresponding to $\mathcal{L}_I$'s freeOnDay. These will be connected with the interface by means of multiple applications of ***Transversal Observer***, yielding

$$\begin{aligned} \langle \mathsf{freeBus}, \mathsf{freeOnDay}^+ \rangle &\in \Lambda_B \\ \langle \mathsf{freeOnDay}^+, \mathsf{freeOnDay} \rangle &\in \Psi_I \\ \langle \neg\mathsf{freeBus}, \mathsf{freeOnDay}^- \rangle &\in \Lambda_B \\ \langle \mathsf{freeOnDay}^-, \neg\mathsf{freeOnDay} \rangle &\in \Psi_I \end{aligned}$$

(where $\mathsf{freeOnDay}^+$ and $\mathsf{freeOnDay}^-$ are auxiliary fresh predicate symbols introduced by the application of the pattern), and similar for all other concepts and roles in the interface (namely $\mathsf{operatesOnCity}$ and $\mathsf{busOnCity}$).

---

**Pattern *(Straight) Adapter*.**

*Problem.* One wants to work with $\mathcal{L}_k$ independently of its particular syntax.

*Solution.*

Add an empty interface knowledge base $\mathcal{L}_I$ to $\mathcal{KB}$ using the desired concept and role names.

Connect each concept and role in $\mathcal{L}_I$ with its counterpart in $\mathcal{L}_k$ by means of an application of the ***Transversal Observer*** pattern.

---

There is one important characteristic of this implementation of the usual Adapter design pattern: the usual dl-program syntax for local extensions to dl-queries only works in the particular case where the query is over a concept or role being directly extended. Because all queries go through the interface knowledge base, where no axioms exist, any other extensions are lost.

Consider a simple example where the interface $\mathcal{L}_I$ specifies two concepts $P$ and $Q$, which are made concrete in $\mathcal{L}_C$ as $A$ and $B$. Furthermore, $\mathcal{L}_C$ also contains the inclusion axiom $A \sqsubseteq B$. Finally, $\mathcal{P}$ contains the single fact $\mathsf{thisIsTrue}(\mathsf{ofMe})$. In $\mathcal{P}$, the direct query $DL_C[A \uplus \mathsf{thisIsTrue}; B](X)$ would return the answer $X = \mathsf{ofMe}$, since $\mathcal{L}_C$ is extended with $A(\mathsf{ofMe})$ in the context of this query. However, the corresponding indirect query (i.e. the same query, but passing through the adapter)

$$DL_I[P \uplus p^+, P \uplus p^-, Q \uplus q^+, Q \uplus q^-, P \uplus \mathsf{thisIsTrue}; Q](X)$$

after extending $\mathcal{P}$ with the rules

$$p^+(X) \leftarrow DL_C[; A](X) \qquad q^+(X) \leftarrow DL_C[; B](X)$$
$$p^-(X) \leftarrow DL_C[; \neg A](X) \qquad q^-(X) \leftarrow DL_C[; \neg B](X)$$

as introduced by the concretization of the observer sets would still return no answer, since $\mathcal{L}_I$ only knows the facts about $B$ that are directly given by $\mathcal{L}_C$ through $q^+$.

Note, however, that the dl-atom $DL_C[A \uplus \mathsf{thisIsTrue}; A](X)$ *is* equivalent to

$$DL_I[P \uplus p^+, P \uplus p^-, Q \uplus q^+, Q \uplus q^-, P \uplus \mathsf{thisIsTrue}; P](X)\,,$$

since the query is directly on the concept whose extent was altered. In practice, this is a common enough situation that this issue is less restrictive than it may seem.

This is a restriction with respect to the full power of dl-programs; but the authors see this as a *feature* of the ***Straight Adapter*** design pattern. Should a context arise where such flexibility is essential, then this is not the right design pattern to apply.

In order to implement the typical, full-fledged ***Adapter*** pattern, one would have to move to a different context than Mdl-programs with observers. A possibility would be to implement it as a syntactic mechanism that would have much more invasive effects on $\mathcal{P}$. However, the authors feel that the more restricted version presented above is still in spirit with the object-oriented philosophy behind ***Adapter***.

## 7.2 Safeguarding potential changes

In a practical context, it is not uncommon to have a function whose definition is variable, for example rotating cyclically among several possibilities. The pattern herein presented provides a clean way to implement such a function in a way that minimizes changes to the program.

**Scenario.** Apart from the fixed discounts offered by WishYouWereThere, the travel agency features several on-and-off promotions that are not always available. A typical example are seasonal promotions, for example a discount on beach resorts during the winter; or offers destined to even out irregularities on sales, such as discounts for trips booked on a Monday.

In order to avoid changing the program on an almost daily basis to reflect the variations in the daily offer, the software designers decided to enrich their Mdl-program with a dedicated ontology $\mathcal{L}_V$ that determines which promotions are available at any given time. This information is obtained by querying upon concept available, whose instances are the currently active promotions.

The predicate promotion is then defined by the following single rule, in $\mathcal{P}$.

$$\mathsf{promotion}(X, Y) \leftarrow DL_V[; \mathsf{available}](X), \mathsf{isEligible}(X, Y)$$

This definition will be enriched by elegibility rules in $\mathcal{P}$, e.g.:

$$\begin{aligned}
\mathsf{isEligible}(\mathsf{mondayDiscount}, X) \leftarrow & \mathsf{topClient}(X) \\
\mathsf{isEligible}(\mathsf{winterVacation}, X) \leftarrow & \mathsf{topClient}(X), \mathsf{wantsToBook}(X, Y, Z), \\
& \mathsf{winterMonth}(Y), DL_H[; \mathsf{beachResort}](Z) \\
\mathsf{isEligible}(\mathsf{bigSale20\%Off}, X) \leftarrow & \mathsf{topClient}(X), DL_A[; \mathsf{estimatedCost}](X, Z), \mathsf{bigSale}(Z)
\end{aligned}$$

We leave the concrete implementation of the auxiliary predicates to the reader's imagination. The reasoner in $\mathcal{L}_V$ would be a more or less complicated engine that would e.g. look at the current date in order to decide which instances of available hold. A simplistic (although not the most practical) implementation would simply require an employee of WishYouWereThere to select the available promotions from a list at the beginning of each workday. When the company decides to create a new promotion, the changes required are simply to add new axioms to $\mathcal{L}_V$ (in order to indicate when it is available) and a new elegibility rule to $\mathcal{P}$ (defining it).

---

**Pattern *Protected Variations*.**

*Problem.* There is a predicate $p$ (in $\mathcal{P}$) whose definition varies, but $\mathcal{P}$ should be protected from these variations, in the sense that it should suffer minimal and easily identifiable modifications.

*Solution.*

If no ***Protected Variations*** knowledge base exists, add it to $\mathcal{KB}$. Let $\mathcal{L}_k$ be this knowledge base.

Define $p$ with a set of rules, each one protected by a query to $\mathcal{L}_k$ on a concept $S$.

Define $S$ in $\mathcal{L}_k$ such that the satisfiable clauses of $p$ are the ones corresponding to its current definition.

---

In the example above, the roles of $S$ and $p$ in the above definition are played, respectively, by the predicates available and promotion. This design pattern requires a dedicated knowledge base; however, several applications of ***Protected Variations*** can use the same knowledge base, so there is no point in having multiple dedicated knowledge bases. Also, this knowledge base does not have to be separate from the rest of the Mdl-program: one could include the necessary concepts and roles (such as available in our example) in an already existing general-purpose knowledge base.

The ***Protected Variations*** design pattern corresponds to the pattern with the same name in object-oriented programming [19].

## 7.3 Beyond these design patterns

The presentation in these last sections does not by any means claim to be exhaustive. Design patterns for several programming paradigms have been around for more than two decades, and dozens of different patterns have been proposed and applied, often in very specific contexts.

The goal of this paper was to show how some of the most elementary design patterns – in the sense that they are the basic building blocks for more complex ones – can be implemented within the framework

of dl-programs. Among the more sophisticated design patterns, our selection took into account the ones that can be more naturally formalized using Mdl-programs with observers.

When we presented the ***Adapter*** design pattern, we discussed some of the problems that may arise in these implementations. To get the feeling for other issues which occur, we now briefly discuss the ***Proxy*** design pattern. This pattern is used when one wants to control or restrict access to a resource, for example a database containing sensitive information. In practice, this is not very different from the ***Adapter*** design pattern – but ***Adapter*** is an algorithm-free pattern that just defines interfaces, whereas an entity implementing ***Proxy*** is expected to do some processing before passing on the information it receives.

If we attempt an implementation along the lines we have followed so far, it would be natural to explore the possibility of a proxy knowledge base to serve as a mediator between two components. In the setting of dl-programs, this is actually not possible to achieve directly, since all queries must go through the logical program. The only other option is to encode the proxy in the logic program itself, forcing every dl-query to the protected resource to be immediately preceded by some atoms implementing the proxy – which from the ***Proxy*** design pattern perspective is not completely satisfactory.

There would be ways to go around this problem, namely by defining appropriate syntactic constructions. Our motivation for defining Mdl-programs with observers was, primarily, to guarantee that *all* dl-queries were appropriately extended, *even the ones that were written after deciding that a concept or role should be observing a predicate.* As it turned out, the construction we proposed is powerful enough to allow for elegant implementations of all the design patterns we discussed earlier. By defining other adequate syntactic constructions, other design patterns may become easily accessible. Note that in the solutions we proposed for all problems the necessary changes are localized: they consist of changing dl-atoms (by means of adding pairs to $\Psi_i$) or adding rules to $\mathcal{P}$ (either directly, as in the case of ***Closed-world***, or by adding pairs to $\Lambda_i$). In all cases, these changes are only reflected in $\mathcal{P}$, and they can be divided into two or three distinct types. This is in line with the whole philosophy of dl-programs: there is an asymmetry between their components where the logic program is the orchestrator between all components as well as its façade: it is the only entity interacting with the outside world.

There is an aspect that cannot be overstressed: the sets $\Lambda_i$ and $\Psi_i$ are syntactic sugar. As such, they do not add to the expressive power of Mdl-programs, but they substantially increase their legibility and internal structure. By working with an Mdl-program with observers, one can more easily understand the core of the program (which is the logic program $\mathcal{P}$) without being disturbed by the presence of myriads of rules that connect $\mathcal{P}$ with the several knowledge bases. In particular, most of the design patterns we presented can be expressed simply as adding specific pairs to carefully chosen observer sets $\Lambda_i$ and $\Psi_i$ – yielding a clean program that is also very easy to maintain and extend. At the end of the day, though, the Mdl-program with observers simply translates into an Mdl-program.

# 8 Conclusions

We proposed a generalization of dl-programs to systems that connect several description logics by means of a logic program. The resulting Mdl-programs are less expressive than other similar systems, e.g. multi-context systems, but they still have a wide range of applicability. Being less general, Mdl-programs are also simpler to use and reason about, inheriting many of the desirable properties which are already known of dl-programs.

On top of Mdl-programs, we defined a syntactic construction (Mdl-programs with observers) that supports communication between the logic program and the different knowledge bases in a structured way, allowing one to externalize several communication aspects and focus on the development of the algorithmic parts of the program. The advantages of using Mdl-programs with observers were made clear by showing how several well-known design patterns from software engineering can be implemented in a very elegant and straightforward way intensively using this syntactic mechanism. Note that it has been argued that the study of design patterns can be a key ingredient in deciding what syntactic extensions should be added to specific programming languages [13]. We feel that we have more than justified the inclusion of observers as a syntactic tool in the context of Mdl-programs.

This paper is by necessity limited in its scope, and it is the authors' intent to explore how more sophisticated design patterns could be applied to Mdl-programs. In this process, it will probably be necessary to introduce different syntactic constructions whose properties will have to be studied.

Since Mdl-programs with observers share several features with multi-context systems, there being some similarities between the former's observer sets and the latter's bridge rules, it would also be interesting to explore how the mechanisms herein discussed can be applied to multi-context systems. However, multi-context systems are a more complex environment with its own specific problems, which will also have to be addressed in such a study.

Another aspect that will have to be discussed relates to the practical aspects of the usage of design patterns. *Ad hoc* solutions to specific problems may be more efficient than the application of systematic methods, but they tend to yield less generalizable and less extensible software applications. On the other hand, the use of observers introduces higher complexity, especially when non-stratified negation (which occurs in the examples discussed earlier) is involved. It is important to have a precise understanding of the compromise between efficiency and quality obtained by a systematic use of design patterns.

The purpose of the present study, at this stage, is to show that design patterns, which have proved essential in building quality software and establishing good programming practices, also have a place in the world of the Semantic Web. One can foresee a future where there is a widespread usage of systems combining description logics with rules, and the availability of systematic design methodologies is a key ingredient to making this future a reality.

# References


[1] M. Adams, J. Coplien, R. Gamoke, R. Hanmer, F. Keeve, and K. Nicodemus. Fault-tolerant telecommunication system patterns. In *Pattern Languages of Program Design 2*, pages 549–562. Addison–Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[2] S. Antoy and M. Hanus. Functional logic design patterns. In Z. Hu and M. Rodríguez-Artalejo, editors, *Functional and Logic Programming, 6th International Symposium, FLOPS 2002, Aizu, Japan, September 15–17, 2002, Proceedings*, volume 2441 of *LNCS*, pages 67–87. Springer, 2002.

[3] G. Brewka and T. Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 385–390. AAAI Press, 2007.

[4] G. Brewka, F. Roelofsen, and L. Serafini. Contextual default reasoning. In M.M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6–12, 2007*, pages 268–273, 2007.

[5] J. de Bruijn, M. Ehrig, C. Feier, F. Martíns-Recuerda, F. Scharffe, and M. Weiten. Ontology mediation, merging, and aligning. In J. Davies, R. Studer, and P. Warren, editors, *Semantic Web Technologies: Trends and Research in Ontology-based Systems*. John Wiley & Sons, Ltd, Chichester, UK, 2006.

[6] L. Cruz-Filipe, P. Engrácia, G. Gaspar, and I. Nunes. Achieving tightness in dl-programs. Technical Report 2012;03, Faculty of Sciences of the University of Lisbon, July 2012. Available at `http://hdl.handle.net/10455/6872`.

[7] T. Eiter, G. Ianni, T. Lukasiewicz, and R. Schindlauer. Well-founded semantics for description logic programs in the semantic Web. *ACM Transactions on Computational Logic*, 12(2), 2011. Article 11.

[8] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12–13):1495–1539, 2008.

[9] T. Eiter, G. Ianni, R. Schindlauer, H. Tompits, and K. Wang. Forgetting in managing rules and ontologies. In *2006 IEEE / WIC / ACM International Conference on Web Intelligence (WI 2006), 18–22 December 2006, Hong Kong, China*, pages 411–419. IEEE Computer Society, 2006.

[10] T. Erl. *SOA Design Patterns*. Prentice Hall, New York, 2009.

[11] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison–Wesley, 2002.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1995.

[13] J. Gibbons. Design patterns as higher-order datatype-generic programs. In R. Hinze, editor, *Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP 2006, Portland, Oregon, USA, September 16, 2006*, pages 1–12. ACM, 2006.

[14] F. Giunchiglia. Contextual reasoning. *Epistemologia*, XVI:345–364, 1993.

[15] F. Giunchiglia and L. Serafini. Multilanguage hierarchical logics, or: how we can do without modal logics. *Artificial Intelligence*, 65(1):29–70, 1994.

[16] B. Grau, B. Parsia, and E. Sirin. Combining OWL ontologies using e-connections. *Journal of Web Semantics*, 4(1):40–59, 2005.

[17] S. Heymans, T. Eiter, and G. Xiao. Tractable reasoning with DL-programs over Datalog-rewritable description logics. In H. Coelho, R. Studer, and M. Wooldridge, editors, *Proceedings of 19th European Conference on Artificial Intelligence (ECAI)*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 35–40. IOS Press, 2010.

[18] M. Kifer and H. Boley (eds.). RIF overview, June 2010. W3C Working Group Note, available at `http://www.w3.org/TR/2010/NOTE-rif-overview-20100622/`.

[19] C. Larman. *Applying UML and Patterns*. Prentice–Hall, 2004. 3rd Edition.

[20] T.G. Mattson, B.A. Sanders, and B.L. Massingill. *Patterns for Parallel Programming*. Addison–Wesley, 2005.

[21] J. McCarthy. Generality in artificial intelligence. *Communications of ACM*, 30(12):1030–1035, 1987.

[22] B. Meyer. *Object-Oriented Software Construction*. Prentice–Hall, 1997. 2nd Edition.

[23] B. Meyer and K. Arnout. Componentization: The visitor example. *IEEE Computer (IEEE)*, 39(7):23–30, July 2006.

[24] B. Motik, P.F. Patel-Schneider, and B.C. Grau (eds.). OWL 2 Web Ontology Language direct semantics, October 2009. W3C Recommendation, available at `http://www.w3.org/TR/owl2-direct-semantics/`.

[25] P. Norvig. Design patterns in dynamic programming. Tutorial slides presented at Object World, Boston, MA, May 1996, available at `http://norvig.com/design-patterns/`.

[26] B.C. Oliveira and J. Gibbons. TypeCase: a design pattern for type-indexed functions. In D. Leijen, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2005, Tallinn, Estonia, September 30, 2005*, pages 98–109. ACM, 2005.

[27] F. Roelofsen and L. Serafini. Minimal and absent information in contexts. In L.P. Kaelbling and A. Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30–August 5, 2005*, pages 558–563. Professional Book Center, 2005.

[28] D.C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.

[29] L. Sterling. Patterns for Prolog programming. In A.C. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, volume 2407 of *LNCS*, pages 374–401. Springer, 2002.

[30] K. Wang, G. Antoniou, R.W. Topor, and A. Sattar. Merging and aligning ontologies in dl-programs. In A. Adi, S. Stoutenburg, and S. Tabet, editors, *Rules and Rule Markup Languages for the Semantic Web, First International Conference, RuleML 2005, Galway, Ireland, November 10–12, 2005, Proceedings*, volume 3791 of *LNCS*, pages 160–171. Springer, 2005.