A Minimalist Visual Notation for Design Patterns and Antipatterns

D. Ballis[1], A. Baruzzo[1] and M. Comini[1]

Dipartimento di Matematica e Informatica (DIMI), University of Udine,
Via delle Scienze 206, 33100 Udine, Italy.

Abstract Achieving a quality software system requires UML designers a good understanding of both design patterns and antipatterns. Unfortunately, UML models for real systems tend to be huge and so hardly manageable, especially for models automatically generated from source code. Thus it would be advisable to have tools to automatically identify particular instances of patterns. For this a formal language to express them is needed. However, a textual formalization of such a language is barely usable by UML practitioners.

In this paper we propose a visual notation obtained by adding to UML as little graphical elements as possible in order to express both patterns and antipatterns (with the needed formality). As such additions are really few and intuitive, we believe that this approach has low cognitive load, thus being both usable by practitioners and still enough rigorous for implementation.

This notation will be used to add a GUI front-end for a prototypical tool, that we have recently developed, which is able to discover (anti)patterns in models.

1 Introduction

Achieving a quality software system requires UML designers a good understanding of both design patterns [9] and antipatterns [5]. Design patterns are, for their nature, at the boundary of programming languages and design models, suffering from a lack of formalism. For this reason, their application often remains empirical and manually performed (which is certainly tedious and error prone).

The knowledge on patterns is potentially very important in the whole software life cycle. For example maintaining software involves recognizing places that can be improved by using better design decisions, like those advocated by design patterns. Nevertheless, there is a lack of tools automatizing the use of design patterns to achieve well-designed pieces of software, to identify recurrent architectural forms, and to maintain software.

In [3] we proposed a rule-based matching algorithm to identify all instances of a pattern in the graph which underlies the designers' diagrams. A pattern is given in a general textual language that can express *at the same time* patterns and antipatterns. However, a textual specification is barely usable by UML practitioners. This work indeed was just our first step towards a more ambitious goal: providing a pattern-matching tool that works with a graphical language which extends UML adding as little graphical primitives as possible in order to be usable, yet at the same time retaining the needed precision to be effectively implementable.

In this paper we propose such a visual notation. As the additional primitives are really few and intuitive, we believe that this approach involves a lean cognitive load for the UML designer. Moreover, we provide a description of an algorithm to convert the visual representation of a pattern to an equivalent textual description in terms of the language proposed in [3]. This description can eventually be fed to the prototypical tool that we have so far developed.

The paper is structured as follows. In Section 2 we express our motivations related to the existing literature. In Section 3 we introduce our visual notation for patterns and antipatterns, providing some examples of use. In Section 4 we describe an algorithm to translate patterns in our graphical notation to terms of the textual language for pattern specification. In Section 5 we discuss some future work directions. Section A contains the graphical representations (with our notation) of *all* GOF patterns not presented in the former part of the paper.

2 Motivations and related work

In the area of software pattern recognition there have been lots of proposals. Without pretending to be exhaustive, we could categorize them as follows.

Precise detection of predefined design patterns (like [16,4]) which provides tools that aim at reconstructing the presence of some of the most common design patterns from the code. The patterns which are identified are hard-coded in the tools. This unfortunately has a quite limited pragmatic applicability, as plenty of small variations (still provenly solid) of recognizable patterns are present in real systems, and these tools would not find them. Moreover, the designer might desire to expressly find his customized "variations over the theme".

Precise detection of predefined antipatterns (like [13]) which aims instead to detect antipatterns. Also in this case only predefined antipatterns can be detected.

Detection of approximate design patterns tries to encompass the rigidity of exact pattern detection, with approximate matches. For example [14] defines a sort of approximated graph matching[1] with suitable similarity measures, while [10] represents patterns and models as strings and then performs string matching. Instead [2,1] defines patterns at the meta-model level and find matches by solving Constraint Satisfaction Problems.
However, since algorithms cannot distinguish the meaning of user customizations, the number of presumed instances of patterns can be quite huge (as tables in [10] show clearly), sensibly reducing the usefulness of the results [2].

[1] after having reduced the model to limit the complexity of the algorithm

[2] This is probably due to the fact that design pattern are based on a very limited set of simple structures (typically one or two hierarchy and a bunch of classes connected

Design pattern description languages approaches like [11,6,7,8] instead introduce a (graphical) formal language which provides a much higher expressive power. Actually for [11,8] there is no tool support that, starting from the (description of the) pattern and a target model, finds all instances, as their focus is just to *precisely* define the intended semantics of a pattern (which is indeed typically expressed in a quite informal way in design patterns books like [9]).

With this approach the designer can easily redefine the descriptions of canonical patterns to specify his customizations, or define new ones from scratch, or either simply look for arbitrary compositions of other patterns at the same time.

Nevertheless, we do not believe these formalisms are good candidates as a graphical language for a pattern-matching tool. Indeed [11] is really difficult to draw, while [8], even if it is quite easy for whom who already know (and understand) the UML meta-model, probably is not that usable by most designers.

The graphical language LePUS of [6,7] is designed on the graphical version of some basic building blocks that are ubiquitous in object-oriented design. Coupled with the language there is a tool which identifies pattern instances in Java/C++ code. Pragmatically, however, what the LePUS notation misses is the integration with UML which is nowadays a standard *de-facto* for specifying object-oriented systems. Moreover, LePUS is not powerful enough to express descriptions of several relevant properties which are needed for antipatterns [3].

In spite of the defects, we appreciate the spirit of LePUS approach. Anyway, instead of redefining UML notations, we prefer just to add what is needed to express a design pattern precisely. Moreover the resulting notation should work for antipatterns too, as both pattern categories are important aspects of a quality design.

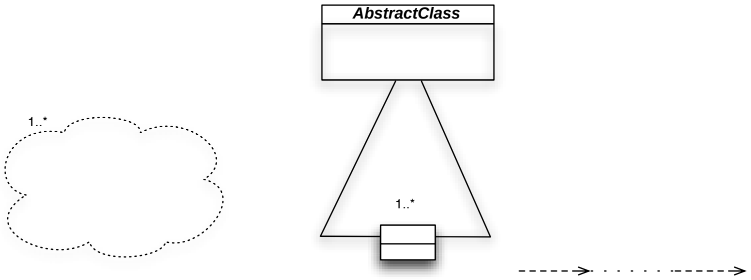
Actually there are also some other kinds of patterns which do not fall in neither of these two categories, but which we believe are also relevant for quality design. One emblematic example is circular dependencies: typically a circular dependency represents an accidental complexity of a software architecture that is introduced by inexperienced designers or hasty developments/maintenance. Sometimes, however, it is due to essential complexity of the application domain and thus it is not considerable as an antipattern. Following the "code smells" terminology coined by Kent Beck we will call, in the following, this kind of patterns *smelly patterns*.

What we aim to get then is a graphical language which

- is able to express all these three kinds of patterns;

together with some relations such as dependencies or associations). Hence, it is very likely that these simple structures can be very similar to substructures of real-world systems which are not instances of patterns.

[3] Not to mention that it works on code and not on UML models.



(a) Replication (b) Hierarchy (c) Concatenation

Figure 1. New graphical primitives

- extends UML with as little graphical elements as possible in order to be usable with a very low cognitive load,
- yet retains the needed precision to model patterns faithfully and,
- last but not least, has a coupled tool that, after getting as input the graphics of a pattern and a target UML model, provides as output the graphics of (all) actual instances of the pattern within the target model.

3 A minimalist visual notation for patterns

In this section we present the graphical primitives we add to UML for (visual) patterns specification. Then we discuss some issues about applicability and pragmatics in the use of our notation.

3.1 Visual primitives

The underlying idea of patterns is that classes are to be considered as variables over classes within a target model. So they are meant more to describe a role of an actual instance. However many times happens that to describe more precisely a pattern something like a variable representing a whole bunch of classes would be needed. The same consideration applies for relations.

Our proposal is to extend UML (which most object-oriented designers already know) with just three new graphical primitives.

Replication (Figure 1(a)) This "cloud" is meant to be used to surround a part of a pattern which, in an actual instance, could be replicated an arbitrary number of times according to the specified multiplicity.

This construct cannot be expressed using UML [4], because the language lacks a primitive to talk about a *variable set* of classes, related all together with one or more model elements.

[4] at least without resorting to the meta-model, which we do not want to do, as explained in Section 2.

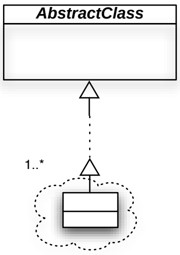


Figure 2. Hierarchy is "syntactic sugar"

Hierarchy (Figure 1(b)) This is meant to refer to a whole hierarchy. The number of possible concrete classes in the hierarchy has to respect the specified multiplicity. The class at the base of the triangle is a placeholder for any *concrete* class in the hierarchy.

We exploit heavily this two primitives for design patterns, as shown by the following examples.

Concatenation (Figure 1(c)) This is meant to specify that an arbitrary concatenation of a certain relation (also according to the specified multiplicity) is admissible. We exploited this primitive for smelly patterns and antipatterns (see the following examples).

Note that actually we could have used just the concatenation and replication primitives, as the hierarchy can be expressed just by means of the other two, as illustrated in Figure 2. However, considering that hierarchy is so pervasively used in (design) patterns and also it is so much intuitive than its "de-sugared" version, we prefer to keep it in our visual notation.

With this three constructs we believe it is possible to define in a rather intuitive, yet precise, way all patterns, as shown by the following examples.

Example 1 (Abstract Factory). In Figure 3 we represent the Abstract Factory pattern using our notation. This description captures the structural constraints concerning the dependencies between the AbstractFactory and the AbstractProduct hierarchies. The replication primitive (represented with the cloud surrounding the AbstractProduct hierarchy) describes the binding between one concrete factory with one or more concrete products, each one belonging to its specific AbstractProduct hierarchy.

Example 2 (Interpreter). In Figure 4 we represent the Interpreter pattern using our notation. This pattern represents another interesting use of the replication primitive, this time to represent a variable set of relationships between two model elements. The variable set of composition relationships between the NonTerminalExpression and the AbstractExpression *roles* is an example of such situation.

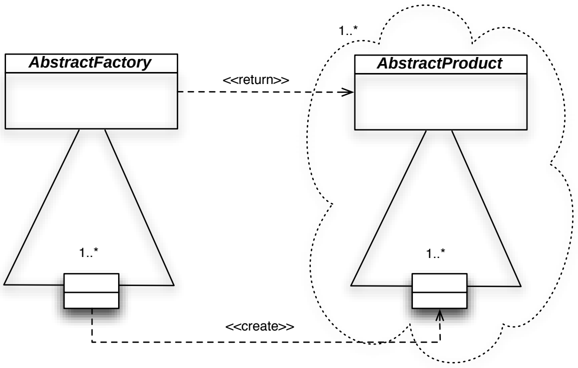


Figure 3. Abstract Factory Pattern

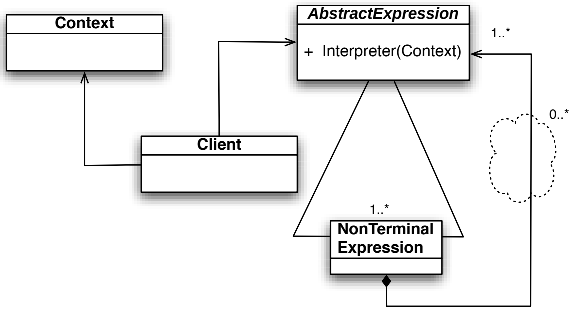


Figure 4. Interpreter Pattern

Example 3 (Bridge). In Figure 5 we represent the Bridge pattern. In this case, our notation captures all the pattern structural constraints concerning the Abstraction and the Implementation hierarchies. If we compare this representation with the canonical representation provided in the GOF book [9], we lack the representation for the note associated to the method Operation declared in the Abstraction class. We discuss in the following how we expect to convey such non-structural constraints in a precise way (see Section 3.2).

Example 4 (Facade). In Figure 6 we represent the Facade pattern using our notation. A potential drawback of our representation is the need to express directly in the pattern's graphical definition the cardinality of the variable sets for both the Client and the Subsystem roles. We use the multiplicity to bound the range of instances to be included in the set. Whereas selecting a value for the upper bound usually is not an issue, picking a proper value for the lower bound is critical, as we discuss in the Section 3.2.

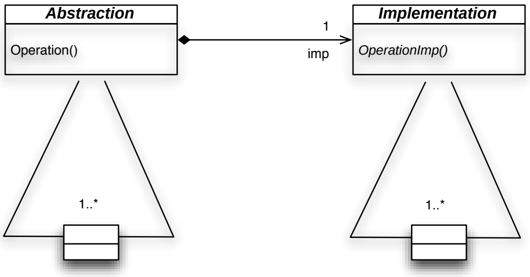


Figure 5. Bridge Pattern

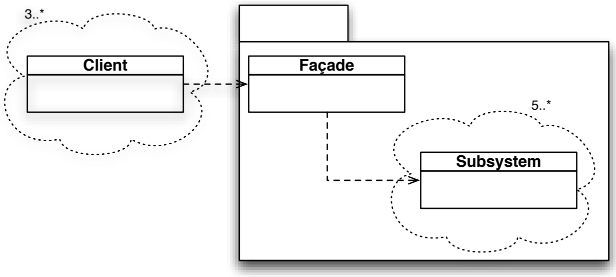


Figure 6. Facade Pattern

Example 5 (Composite Variant). It is straightforward to change/customize a pattern. For example in Figure 7 we illustrates a variation in the structure of the Composite pattern which is used in the Incremental Testing Framework pattern.

Example 6 (Proxy Variant). In Figure 8 we show a well-known variant of the Proxy pattern [15] in which the designer moved the original aggregation between Proxy and Real Subject roles to an aggregation between Proxy and Subject roles.

Example 7 (Circular Dependency). In Figure 9 we use our notation to describe the circular dependency pattern. It illustrates the use of the concatenation primitive, which requires a path of dependencies coming back the same class whose length is at least 2.

Example 8 (Blob). In Figure 10 we describe the Blob antipattern [5]. As in the case of the Facade pattern, we should adjust the multiplicities lower bounds. We have chosen the values in figure as suggested in [5].

3.2 Applicability and expressiveness

In this section we discuss some aspects of pattern specifications that we consider very important for both applicability and pragmatics considerations.

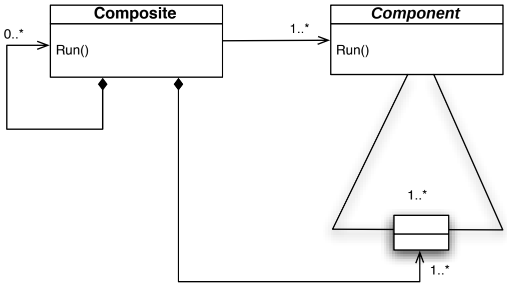


Figure 7. A Variant of Composite Pattern

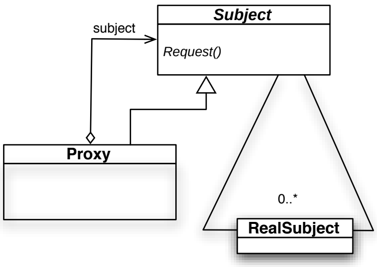


Figure 8. A Variant of Proxy Pattern

At first, as we just extend UML we believe we have a rather rich and scalable methodology, as we have at least the expressive power of UML.

Our notation exploits the UML concept of multiplicity in both hierarchy and replication primitives. This sometimes poses the problem of selecting proper values for lower bounds. In fact, to avoid many false positive it is better to choose high values, thus reducing the number of matches. At the same time, anyway, we would need low values in order to avoid any missed match, increasing the recall. Preserving good levels of both precision and recall is not possible. In practice we plan to provide a GUI with some kind of sliders so users can customize values quickly when looking for matches of patterns like Facade and Blob. A much clever way to cope with this issue could be to introduce variables in multiplicity plus an OCL constraint relating these variables. For example in the Blob antipattern we could use n and m instead of the two fixed lower bounds (representing both the number of attributes and the number of methods in the complex controller class) and then add a constraint like $n + m > 50$, closely resembling what stated in [5].

The canonical forms of many design patterns are often enriched with (textual) annotations concerning the methods' implementation, in order to provide informal semantics about specific constraints that the graphical representation alone cannot convey. A good example of these constraints are Bridge and Composite

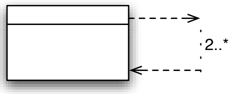


Figure 9. Circular Dependency Smelly Pattern

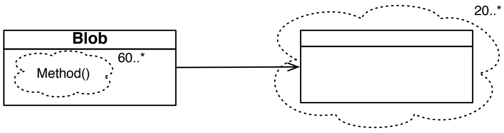


Figure 10. Blob Antipattern

patterns (see Examples 3 and 5). In the case of Bridge, our visual notation cannot specify that the Operation method (polymorphically) calls the OperationImp method declared in the Implementation class. However, all these informal notes clearly cannot be used in a tool. Instead of inventing a new primitive to provide this information, we believe that a better choice should be to add either UML sequence diagrams or OCL declarations. Sequence diagrams, in particular, can be precise enough to show the Bridge dynamics and, at the same time, easier to understand for a UML practitioner (which usually is less comfortable with OCL expressions).

Another interesting issue to cope with is the specification of run-time type constraints. For example, in Example 6 the Proxy aggregates a Subject object. Because Subject is an abstract base class, this is clearly an example of polymorphic binding with a derived class. Anyway, also Proxy is derived from Subject, so we should express the constraint that the Subject role can be covered by any class in the hierarchy, except Proxy itself. This could be easily handled by adding an OCL constraint specifying that the type of `Proxy->subject` has to be different from Proxy.

4 Automatic translation from visual notation to pattern language

Hoping to have convinced the reader that the previous visual notation is really intuitive (making it suited for a low cognitive tool), it remains to face the problem of employing it in practice to do the matching on the target model. As the visual notation is meant for persons, it is very high level and thus difficult to implement directly. Actually we have decided to solve this issue by splitting it into two phases.

1. Design a set of constructs providing basic building blocks for which it is quite easy to provide an abstract machine which, taken a description of a pattern and a target model, realizes all possible matches.
2. Then provide an algorithm to convert the visual representation to an equivalent textual description in terms of the language of Point 1

We started to approach Point 1 in [3] and then, while the development of the corresponding prototypical tool went by, we made some enhancements. In the following we briefly summarize this work in order to give a formal description of what follows (for details consult [3]). Then in Section 4.2 we describe the conversion from the graphics representation to this building blocks.

4.1 A low level language to express patterns

The spirit we had/have in developing this language is to provide a (possibly wide) set of small building blocks which are

1. small enough to be easily implementable,
2. convey a precise semantics,
3. are altogether expressive enough to cover all possibile uses in a UML design.

In the following, we assume that x, y are (pattern) class variables, a, b class names variables, and v, w method names variables. With $c(P, M)$ we denote a class pattern where c is the class name, P is a set of class properties (like `abstract`, `concrete`, `static`, *etc.*) and M is a set of methods (of the form $m : \mathtt{s}_1, \ldots, \mathtt{s}_n \rightarrow \mathtt{s}$, where $\mathtt{s}$'s are either basic type names or class names). Actually in the tool we also consider all typical method properties like visibility, staticity, stereotypes, parameter passing modalities (in, out, ...), *etc.*. Besides, we have also the set of attributes. For the sake of simplicity we will not clutter this presentation with all these details.

Formally, a *class diagram pattern* (or simply pattern) p and *sub-pattern* sp are defined by means of the following BNF-like grammar:

$$\begin{aligned} p ::= \; & \mathtt{inh}(x, y) \mid \mathtt{dep}(x, y) \mid \mathtt{assoc}(x, y) \mid \mathtt{aggr}(x, y) \mid \mathtt{comp}(x, y) \\ & \mid \mathtt{any}(x, y) \mid \mathtt{star}(a, b, sp) \mid \mathtt{path}(a, b, sp) \mid \mathtt{span}(a, sp) \mid \mathtt{hierarchy}(x) \\ & \mid \mathtt{all2one}(a, b, sp) \mid \mathtt{one2all}(a, a, sp) \mid \mathtt{onto}(a, b, sp) \mid \mathtt{into}(a, b, sp) \\ & \mid \mathtt{iso}(a, b, sp) \mid p \oplus p \\ sp ::= \; & \lambda v\, w.p \end{aligned}$$

Observe that class name variables in a pattern might be bound via an abstraction binder λ. This induces the usual notion of free variables (i.e., those not bound by λ's).

Roughly speaking, our pattern language is equipped with constructs to recognize simple class diagrams relations (such as inheritance, aggregation, *etc.*), as well as to perform more complex matches against the target class diagram. The sense of matching is not just the usual one, i.e., a variable is instantiated with some value, as high order constructs like `hierarchy`(x) do instantiate x but also (possibly) return a (sub)diagram with all subclasses of the actual x.

Now we briefly explain the meaning of all constructs (for details consult [3]). Note that by $sp(a,b)$ we denote the pattern which is obtained from $sp = \lambda v\, w.p$ by replacing v with a and w with b in p.

$p \oplus p$ $\oplus$ is a binary, associative and commutative operator, which can be employed to build compound patterns starting from simpler ones. The pattern $p_1 \oplus p_2$ fails to find a match, whenever either p_1 or p_2 fail on the target class diagram. Otherwise the union of the matches computed by the compound pattern is delivered.

$arrow(x, y),\ arrow \in \{$`inh`, `dep`, `assoc`, `aggr`, `comp`$\}$ These constructs allow to define patterns modeling the class diagram relations (i.e., inheritance, dependency, association, aggregation, composition relations) from a class pattern x to a class pattern y.

`any`(x, y) This construct allows to match any class diagram relation from class pattern x to class pattern y independently of the relation kind.

`path`(a, b, sp) This construct allows to find a concatenated sequence of matches of subpattern sp starting from a class identified by a and ending in a class identified by b.

`hierarchy`(x) The construct `hierarchy` selects all the inheritance relations of a given hierarchy whose root class matches the class pattern x.

`span`(a, sp) The `span` operator extracts all the class diagram relations matching the union of the patterns $sp(a, z)$, for any class name z.

`star`(a, b, sp) The `star` operator returns the set of all the class diagram relations matching the subpattern $sp(a, b)$.

`all2one`(a, b, sp) Let H be a hierarchy whose root class has a name matching a, and L be the set of all the class names of the target classes of H. Let cn be a class name matching b. If $sp(l, cn)$ does not fail, for $l \in L$, then the `all2one` operator returns the union of all $sp(l, cn)$, $l \in L$, otherwise it fails.

$morph(a, b, sp),\ morph \in \{$`onto`, `into`, `iso`$\}$ *morph* constructs allow to recognize surjective (`onto`), injective (`into`), and bijective (`iso`) "morphisms" between the concrete classes of two hierarchies whose root class names match a and b respectively. Morphisms are represented by the subpattern sp.

Actually, in the current prototype we introduced also the construct

`hierarchy`(x, sp) that selects all the inheritance relations of a given hierarchy whose root class matches the class pattern x.

Moreover all high order constructs (like `span`, `star`, *etc.*) have multiplicities to give fine control over the number of admissible matches. For the sake of simplicity we won't clutter the following presentation with all this details, as we believe it is always clear from the context which are the multiplicity values.

4.2 Description of the translation algorithm

The level of granularity of the textual language is certainly suited for implementation (we indeed have a prototype for it). It can be used to express precisely which is the structure of admitted instancies. However, it is very fine graned and it is even too expressive, in the sense that it allows to express constructions that in practice have little sense.

In order to provide a simpler algorithm to convert the graphical notation to low level language, we adopted some reasonable assumptions on the structure of patterns that we consider valid inputs for our tool. These assumptions are satisfied by any pattern presented here, and for all those described in [9], [5], and [15] (which we have not included in this paper due to obvious space limitations). The most relevant is that, whenever in a pattern specified with our notation there is some relationship between two placeholders belonging in two different hierarchies, we assume that there should also be a relationship between the hierarchies roots. This assumption reflects the object-oriented practice of using public inheritance only in conjunction with polymorphism, preserving the Liskov Substitution Principle [12]. The same argument in general is applicable for any hierarchy and its clients.

We assume to have an underlying data structure which:

- is able to enumerate in some way all relationships (in order to define the algorithm by induction);
- can identify all relationships which incides on a given class;
- can determine which subpattern is under a certain cloud and if a certain relation is under a cloud.

The algorithm works in two phases. Note that the classes on which relations incide are translated along the way trivially (for example AbstractExpression of Figure 4 is converted to $\mathit{AbstractExpression}(\{\texttt{concrete}\}, \{\mathit{Interpreter} : \mathit{Context} \rightarrow \mathit{void}\})$.

Phase 1 We first start translating the pattern ignoring clouds around subpatterns. The translation is done componentwise on the set of all relationships, including concatenation and hierarchy (which just have a special treatment), by processing first relations and concatenations between classes, then hierarchies and (within that process) relations between any placeholder and everything else (i.e., other classes or placeholders). All single translations are "composed" with $\oplus$. As $\oplus$ is commutative and associative, the order in processing does not matter.

UML relationships between classes Here the translation is almost trivial. For each relation we have the corresponding textual primitive[5]. The only special case is for the dependency relation. Whenever in a pattern we have a dependency with a stereotype like ≪create≫ and ≪return≫ it

[5] Remember that in the actual implementation of the language this primitives have fields for relation role and multiplicities.

is clearly the case that in the target model we need to match only dependency with that stereotype. In this case we translate to `dep`. Otherwise we translate to `any`.

Concatenation between classes If we have the concatenation of a relation and $p(a, b)$ is the translation of that relation, we translate to $\texttt{path}(a, b, \lambda c\, d.p(c, d))$[6].

Clouds on relations If we have a cloud on a relation and $p(a, b)$ is the translation of that relation, we translate to $\texttt{star}(a, b, \lambda c\, d.p(c, d))$.

Hierarchy When we have a hierarchy between a class x and a placeholder y then

- for all relations to a class z, where the translation of the relations to z is $p(y, z)$, we translate to $\texttt{all2one}(x, z, \lambda y'\, z'.p(y', z'))$.
- for all relations to a placeholder z, where the translation of the relations to z is $p(y, z)$, in case the relation between the root classes of the two placeholders has
  multiplicity 1 we translate to $\texttt{iso}(x, z, \lambda y'\, z'.p(y', z'))$,
  otherwise we translate to $\texttt{onto}(x, z, \lambda y'\, z'.p(y', z'))$.
- in case we have no relations on y (excluding the hierarchy itself) we translate to $\texttt{hierarchy}(x)$.

Note that actually the renaming of variable names is irrelevant as the λ binder introduces local variables. We showed renamed variables for the sake of clarity.

Phase 2 We consider the parts of the pattern which are under a cloud. For the sake of clarity let consider a subpattern under the cloud that has a class y in relation r to another class outside the cloud x. Let call p the translation of all this part (both the subpattern and r). We than substitute p with $\texttt{span}(x, \lambda x'\, y'.p)$. The case where r is a concatenation or a "clouded" relation is analogous. It is just a matter of considering the free variables involved in the cloud and the related surroundings.

Example 9. We show now the translation of Examples 1 and 2.

Abstract Factory: phase 1 delivers

$$\begin{aligned}&\texttt{dep}(\mathit{AbstractFactory}(\{\texttt{abstract}\}, \emptyset), \mathit{AbstractProduct}(\{\texttt{abstract}\}, \emptyset)) \oplus\\ &\quad \texttt{iso}(\mathit{AbstractFactory}, \mathit{AbstractProduct}, \lambda c\, d.\texttt{dep}(c(\emptyset, \emptyset), d(\emptyset, \emptyset)))\end{aligned}$$

and, as `iso` is under a cloud and there is a `dep` between *AbstractFactory* and *AbstractProduct*, phase 2 delivers

$$\begin{aligned}&\texttt{span}(\mathit{AbstractFactory}(\{\texttt{abstract}\}, \emptyset), \lambda a\, b.\\ &\quad \texttt{dep}(a(\{\texttt{abstract}\}, \emptyset), b(\{\texttt{abstract}\}, \emptyset)) \oplus\\ &\quad \texttt{iso}(a, b, \lambda c\, d.\texttt{dep}(c(\emptyset, \emptyset), d(\emptyset, \emptyset))))\end{aligned}$$

[6] We also store the multiplicity values in the corresponding primitive's field.

Interpreter: the algorithm delivers

$$\begin{aligned}
&\texttt{assoc}(\mathit{Client}(\{\texttt{concrete}\},\emptyset),\mathit{Context}(\{\texttt{concrete}\},\emptyset))\oplus\\
&\texttt{assoc}(\mathit{Client}(\{\texttt{concrete}\},\emptyset),\mathit{AbstractExpr}(\{\texttt{abstract}\},\emptyset))\oplus\\
&\texttt{dep}(\mathit{AbstractExpr}(\{\texttt{abstract}\},\emptyset),\mathit{Context}(\{\texttt{concrete}\},\emptyset))\oplus\\
&\texttt{all2one}(\mathit{AbstractExpr},\mathit{AbstractExpr},\lambda a\,b.\texttt{star}(a,b,\lambda c\,d.\texttt{comp}(c,d)))
\end{aligned}$$

5 Conclusions and future works

In this paper we proposed a visual notation to express design, smelly and anti patterns. As it relies on UML plus just three intuitive graphic primitives we believe that this approach should be fruitfully usable by UML practitioners. We provided a description of an algorithm to convert the visual representation of a pattern to an equivalent textual description in terms of the language proposed in [3]. This promises that eventually we'll be able to combine this graphical notation in a intuitive GUI for the prototypical tool that we have developed so far.

Apart from the implementation of such a combined tool, in the future we think it would also be interesting to cope with specification and matching of sequence diagrams, as they are sometimes needed to convey the precise meaning of certain implementation constraints (that in GOF patterns for example are presented informally). Sequence diagrams can be precise enough to specify method dynamics and are easy to understand for a UML practitioner. This could be useful to better tune the matching.

Another possibility is to employ OCL constraints in pattern specification. This would certainly enhance the expressive power of both class and sequence pattern diagrams, introducing a flexible way for defining patterns more precisely and thus reducing the number of false positives. It would however be important to see if this extra feature would be used by UML practitioners, which usually are less comfortable with OCL expressions.

References

1. H. Albin-Amiot, P. Cointe, Y.-G. Gueheneuc, and N. Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. *ase*, 00:166, 2001.
2. H. Albin-Amiot and Y.-G. Guhneuc. Meta-modeling design patterns: Application to pattern detection and code synthesis. In Bedir Tekinerdogan, editor, *Proceedings of ECOOP Workshop on Automating Object-Oriented Software Development Methods*, 2001.
3. D. Ballis, A. Baruzzo, and M. Comini. A rule-based method to match Software Patterns against UML Models. In *The 8th International Workshop on Rule-Based Programming*, 2007.
4. F. Bergenti and A. Poggi. Improving uml design using automatic design pattern detection. In *Proc. 12th. International Conference on Software Engineering and Knowledge Engineering (SEKE 2000)*, pages 336–343, 2000.

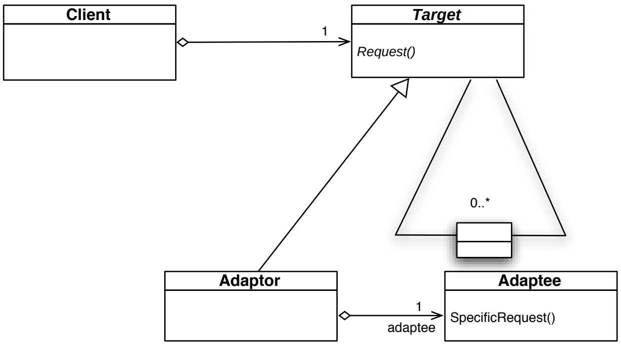
5. W. J. Brown, R. C. Malveau, H. W. McCormick, III, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, New York, 1998.
6. A. H. Eden. Formal Specification of Object-Oriented Design. In *Proc. Int'l Conf. Multidisciplinary Design in Engineering CSME-MDE 2001, Montreal, Canada*, 2001.
7. A. H. Eden. LePUS: A Visual Formalism for Object-Oriented Architectures. In *Proc. 6th World Conf. Integrated Design and Process Technology—IDPT 2002, Pasadena, CA, USA*, 2002.
8. R. B. France and S. Ghosh. A UML-Based Pattern Specification Techniques. *IEEE Transactions On Software Engineering*, 30(3):193–206, 2004.
9. E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.
10. O. Kaczor, Y.-G. Gueheneuc, and S. Hamel. Efficient identification of design patterns with bit-vector algorithm. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 175–184, Washington, DC, USA, 2006. IEEE Computer Society.
11. A. Lauder and S. Kent. Precise visual specification of design patterns. In Eric Jul, editor, *ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 114–134. Springer, 1998.
12. B. Liskov and J. M. Wing. Family values: A behavioral notion of subtyping. Technical report, Pittsburgh, PA, USA, 1993.
13. N. Moha, Y.-G. Gueheneuc, and P. Leduc. Automatic generation of detection algorithms for design defects. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 297–300, Washington, DC, USA, 2006. IEEE Computer Society.
14. N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S.T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, 2006.
15. J. M. Vlissides. *Pattern Hatching: Design Patterns Applied.* Addison-Wesley, Reading, MA, 1995.
16. M. Vokác. An efficient tool for recovering Design Patterns from C++ Code. *Journal of Object Technology*, 5(1):139–157, 2006.

A (Precise) Graphical representation of GOF patterns

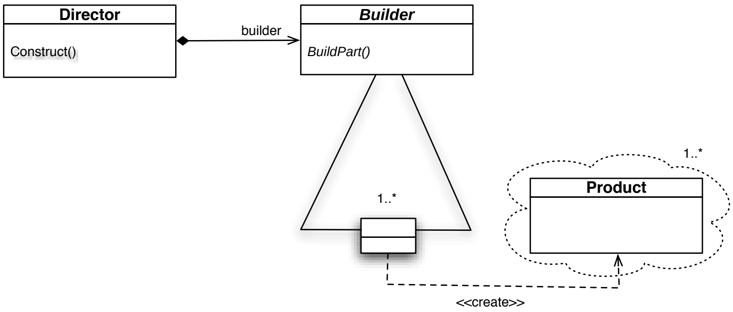
This appendix contains the graphical representations (with our notation) of *all* GOF patterns not included in the former part of the paper as well as some well known (architectural) anti pattern.

A.1 Missing GOF patterns

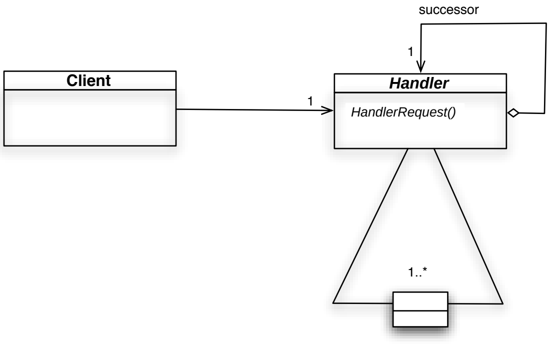
Adapter



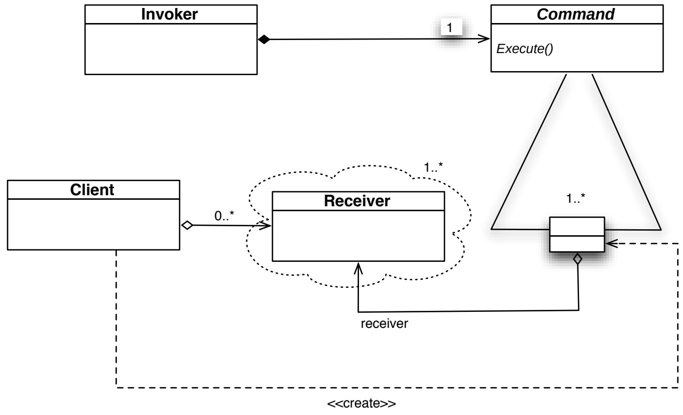
Builder



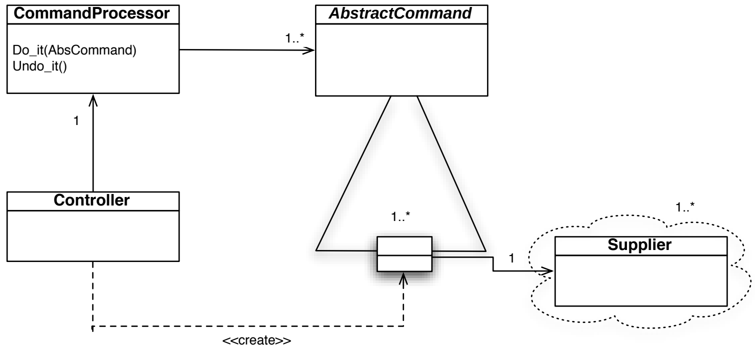
Chain of Responsibility



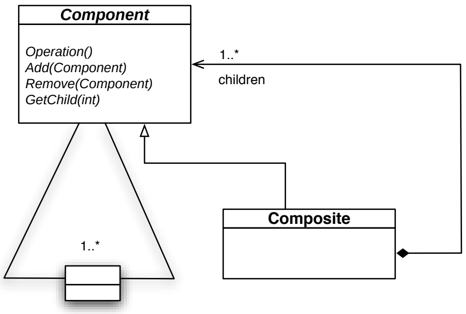
Command



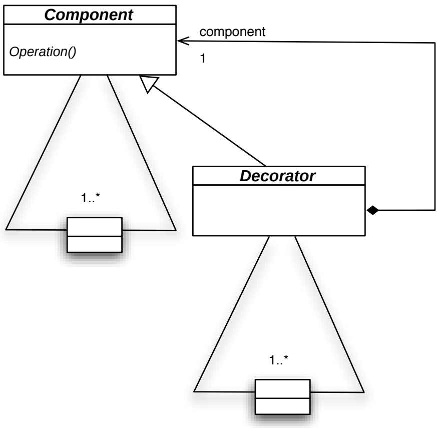
Command Processor



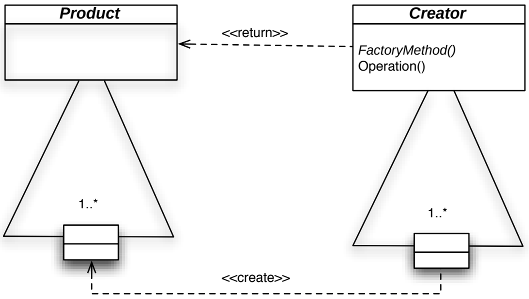
Composite



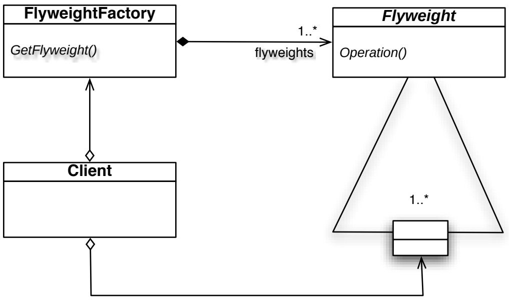
Decorator



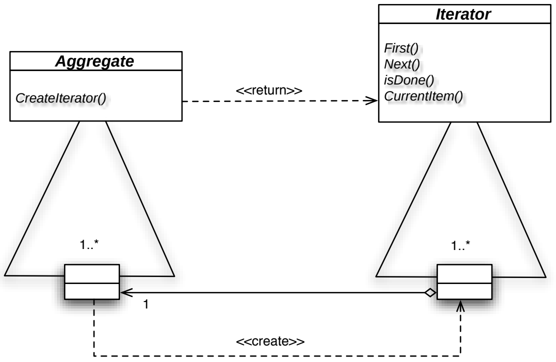
Factory Method



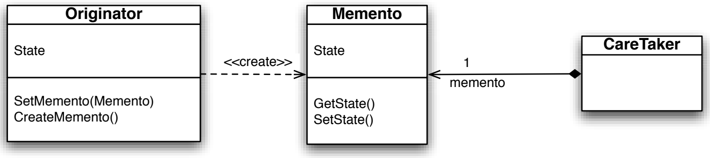
Flyweight



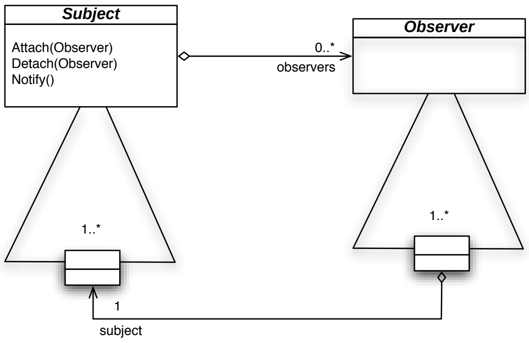
Iterator



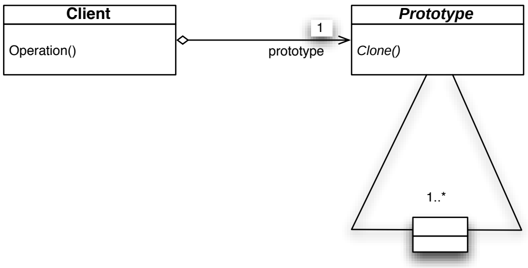
Memento



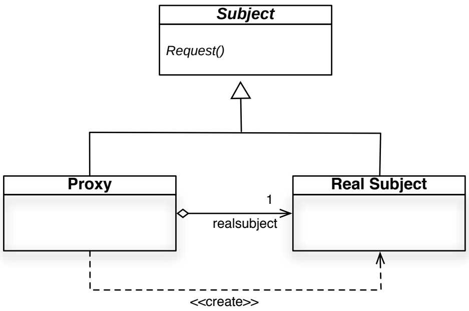
Observer



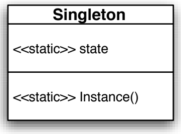
Prototype



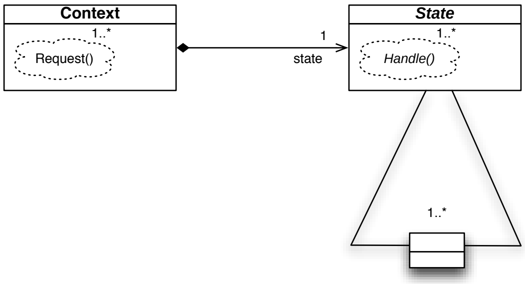
Proxy



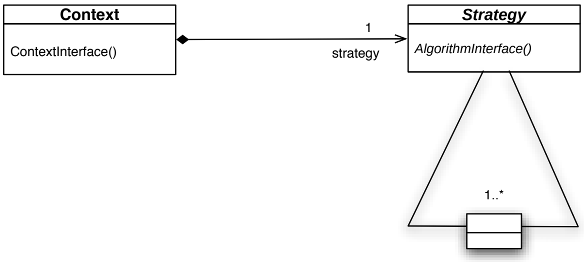
Singleton



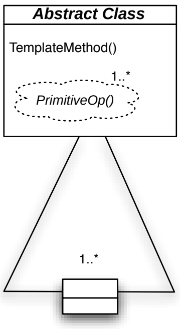
State



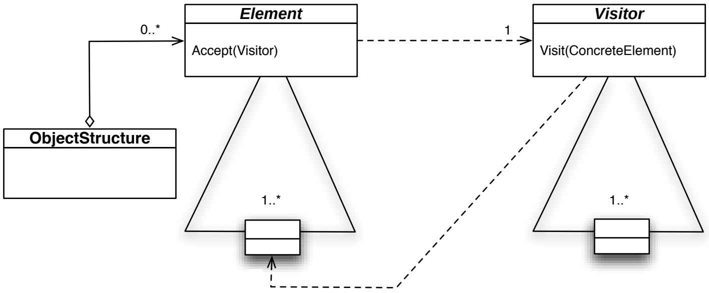
Strategy



Template Method

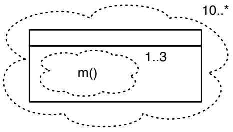


Visitor



A.2 More anti patterns

Functional Decomposition



Poltergeist