This paper has been accepted at TOOLS USA 2001.

Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-class Design Defects

Yann-Gaël Guéhéneuc*, Hervé Albin-Amiot†
École des Mines de Nantes
4, rue Alfred Kastler – BP 20722
44307 Nantes Cedex 3
France
{guehene|albin}@emn.fr

Abstract

*Developing code free of defects is a major concern for the object-oriented software community. In this paper, we classify design defects as those within classes (*intra-class*), those among classes (*inter-classes*), and those of semantic nature (*behavioral*). Then, we introduce guidelines to automate the detection and correction of inter-class design defects: We assume that design patterns embody good architectural solutions and that a group of entities with organization similar, but not equal, to a design pattern represents an inter-class design defect. Thus, the transformation of such a group of entities, such that its organization complies exactly with a design pattern, corresponds to the correction of an inter-class design defect. We use a meta-model to describe design patterns and we exploit the descriptions to infer sets of detection and transformation rules. A constraints solver with explanations uses the descriptions and rules to recognize groups of entities with organizations similar to the described design patterns. A transformation engine modifies the source code to comply with the recognized distorted design patterns. We apply these guidelines on the Composite pattern using* PTIDEJ*, our prototype tool that integrates the complete guidelines.*

Keywords: *OO design, design defects, design patterns, constraints, source code transformation.*

1: The problem of detecting and correcting design defects

Developing quality code is a major concern for the software community. Producing bug-free, extensible, and adaptable code is a hard task. It requires skills, experience, and a deep understanding of the structure and behavior of the software under development. Methods (such as eXtreme Programming [3]) and tools (suchlike IDEs[1]) exist to alleviate the coding task for developers. However, helping in the comprehension and modification of the application is still a major problem. This problem is even more acute for maintenance.

The maintenance tasks consist in (1) analyzing (unknown) source code to understand[2] its structure and behavior; in (2) spotting the places that need modifications to implement new requirements or to correct defects; finally, in (3) modifying the code without breaking up the rest of the software. Maintenance represents 75 percent of the life cycle of an application and the comprehension of the software takes up to 80 percent of the maintainers' time [42]. Therefore, the correction, extension, and adaptation of legacy object-oriented software are left to experts with both forward engineering and reverse engineering skills that can efficiently picture large and complicated software.

To perform maintenance tasks, developers must first accurately identify defects[3] in the software design. Systems [5, 25, 26, 33, 43] exist to visualize software using high-level representations to help the identification. Then, developers may use refactoring [13, 22, 35] to eliminate defects and to improve the overall software quality. Refactoring decreases the cost of maintenance and facilitates modification in response to changing requirements. However, there is a general lack of tools for automating the task of identifying design defects.

*This work is partly funded by Object Technology International Inc. – 2670 Queensview Drive – Ottawa ON K2B 8K1 – Canada.
†This work is partly funded by Soft-Maint – 4, rue du Château de l'Éraudière – 44 324 Nantes – France.

[1] IDE stands for Integrated Development Environment, such as IBM's VISUALAGE FOR JAVA or Symantec's VISUALCAFÉ.

[2] The associated documentation should solve this problem. However, documentation often is either incomplete or missing (as stated by [33] and others).

[3] In the rest of this paper, the term defects includes design defects and any architectural choices that impede the implementation of new requirements.

We believe that methodologies and tools can help developers in automatically detecting and correcting design defects. Unfortunately, design defects do not belong to a unique category. We classify design defects according to their scope within the application. Using this criterion, we distinguish three main types of design defects: *Intra-class*, *inter-class*, and *behavioral*. In section 2, we propose a classification of design defects found in the literature following these three categories. We show that intra-class design defects have been studied and automated extensively in literature. We also show that little has been done to automate the detection and correction of behavioral and inter-class design defects. The main reason is the lack of formalization and the need to rely on experts' intuition.

Consequently, we propose guidelines to automate the detection of inter-class design defects based on design patterns and using constraints. In Section 3, we postulate that detecting group of entities (classes, interfaces) with participants (type, number, etc.) and relationships (inheritance, association, etc.) similar to well-known design patterns is equivalent to the detection of a subset of inter-class design defects. The guidelines, presented in Section 4, consist in (1) formalizing design patterns using a meta-model; in (2) decorating the resulting model with detection (constraints) and transformation rules; in (3) using a constraints solver with explanations to detect into source code groups of entities similar to a modelled design pattern; and finally, in (4) transforming the source code to comply with the specification given by the design pattern model. We illustrate these guidelines with an example based on the Composite pattern and using PTIDEJ, a prototype tool that integrates the four steps of the guidelines.

Finally, in section 5, we discuss the guidelines proposed and the results obtained, and we give future direction of studies.

2: A classification of design defects

By introducing a classification of design defects, we aim to discover recurring detection and correction methods. This classification will ease the assessment of new reengineering techniques and tools. Sorting and classifying design defects is complex because of the multiple points of view available. In the presented classification, we do not intend to give an exhaustive view on design defects but to offer a common framework to be discussed, enriched, and evolved.

2.1: The basis

We establish the current classification from the literature [1, 5, 9, 10, 12, 13, 16, 19, 20, 21, 22, 27, 29, 35, 38, 39, 40, 41, 42], based on the scope of the design defects inside an object-oriented applications. It distinguishes among design defects involving the internal structure of a class, design defects involving interactions among classes, and design defects relating to the application semantics. We retain these three categories because they represent three distinct levels of abstraction and thus must rely on different detection and correction techniques.

- $DD_1 \triangleq$ **intra-class**: This category includes any design defect related to the internal structure of a class. It embodies *stylish* and *syntactic* defects: Defects in the structure of the class or its members. For examples, lines longer than 70 characters [19] are difficult to read and print. Unused fields [38] disturb or confuse readers and make the classes bigger. Methods with too many invocations or statements [10] are error-prone and difficult to maintain and to extend.
- $DD_2 \triangleq$ **behavioral**: All the design defects related to the application *semantics* belong to this category. The "Year 2000 problem" [42] (due to the storage of years on only two digits) is a typical behavioral design defect. Another example of behavioral design defects concerns changes in the environment of a system. An algorithm computing the trajectory of fireworks may be perfectly valid on Earth and completely wrong on the Moon, because of the change of gravity, the lack of atmosphere, ...
- $DD_3 \triangleq$ **inter-class**: This category encloses any design defect related to the external structure of the classes (their public interface) and their relationships. All design defects in the application *architecture* belong to this category. For example, mixing different algorithms within a single data structure is an architectural defect: The algorithms overweigh the data structure, the data structure extension is slowed down because it must be modified every time a new algorithm is added and it is likely to grow rapidly out of control [15].

2.2: An evolution

These three categories are not orthogonal and several design defects do not fit simply in a single category. We define four additional categories as the intersections of the three previous ones. These categories allow a finer-grain classification of the design defects. It is now possible to distinguish among

design defects relative to any combination of syntactic, structural, semantic, or architectural defects. These supplementary categories also allow differentiating among design defects that root in one category and imply changes in another one. For example, duplicated code across classes [10, 13, 40] is detected into the internal structures (DD_1) of the classes, but the resulting defects appear in both their internal structures (DD_1) and their architecture (DD_3).

This combination of the location and the effects on the architecture of the design defects might seem confusing, but we believe it is relevant to enhance the distribution of design defects over the different categories and to allow finer-grain comparisons of the associated detection and correction techniques.

Thus, evolving from the three main categories, we introduce ten categories to classify the design defects, as in Figure 1. These categories are intra-class design defects (DD_1), behavioral design defects (DD_2), inter-class design defects (DD_3), and:

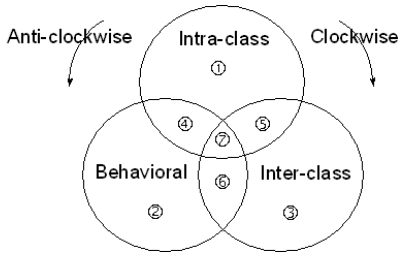

Figure 1: The ten categories.

- $\boldsymbol{DD_4 = DD_1 \cap DD_2}$: Design defects involving both the semantics of the class and its internal structure. This set is divided in two parts: $DD_4^{clockwise}$, defects in the behavior of the class which corrections imply changing its structure. For example, equality tests on `String` instances [40]. And $DD_4^{anti-clockwise}$, defects in the internal structure of the class which correction implies changing its behavior.
- $\boldsymbol{DD_5 = DD_1 \cap DD_3}$: Design defects related to both the internal and external structures of the classes. It is partitioned between $DD_5^{clockwise}$, the group of design defects internal to the class which corrections imply changes to the application architecture. For example, duplicated code among classes [27] reveals a need to change the architecture to factor out the duplicated code. And $DD_5^{anti-clockwise}$, design defects in the architecture involving changes to the internal structure of classes. For example, the use of the Composite pattern [15] when a single field could be used [39].
- $\boldsymbol{DD_6 = DD_2 \cap DD_3}$: Design defects related to both the architecture and the behavior of the classes. This set is composed of $DD_6^{clockwise}$, set of design defects related to the application architecture and which corrections imply changing the semantics of its classes. For example, a "God" class [34] is a sign of a bad architecture, which improvement implies changing the semantics of (at least) the "God" class. And $DD_6^{anti-clockwise}$, design defects in the behavior of classes, which corrections imply changes in their architecture. For example, a derived class not preserving its base class invariant [41] needs to have its inheritance link reconsidered.
- $\boldsymbol{DD_7 = DD_1 \cap DD_2 \cap DD_3}$: The set of all the design defects implying the structure, the semantics, and the architecture of the application.

2.3: The results

Table 1 on the right presents the number of design defects found in the literature following the ten previous categories. We found 61 different design defects [17]. Among those 61 design defects, 13 were related to the actual architecture of the classes[4] (9 in the DD_3 category and 4 in the $DD_5^{anti-clockwise}$ category), whereas 39 correspond to defects in the internal structure of the classes (27 in the DD_1 category and 12 in the $DD_5^{clockwise}$ category). That is, 25 percent of the intra-class design defects have automatic detection or correction methods (not including the design defects for which it exists manual procedures). This figure drops to 8 percent for the inter-class category.

Design defects	Number of design defects	Number of automatic correction or detection methods
DD_1 intra-class	27	5
DD_2 behavioral	1	1
DD_3 inter-class	9	1
DD_4	0	0
$DD_5^{clockwise}$	16	5
$DD_5^{anti-clockwise}$	4	0
DD_6	0	0
DD_7	8	0

Table 1: Number of design defects by category.

These results corroborate what we had intuitively expected: Intra-class design defects have been more extensively studied and automated. A reason is the similarity of intra-class design defects with defects in procedural programming languages. Procedural programming languages exist for more than 50 years and have been subject to several studies [24, 37], whereas the first catalogue of good object-oriented architectural practices only appeared in 1994 with "Design Pattern: Elements of Reusable Object-Oriented Software" [15].

[4] In the rest of this paper, intra-class design defects refer to the design defects from category DD_3 and $DD_5^{anti-clockwise}$.

3: The problem of automating the detection and correction of inter-class design defects

3.1: The problem and its hypotheses

Many studies [10, 13, 20, 38, 40] address the problems of automating the detection and the correction of intra-class design defects. Few studies aim at the automation of both the detection and correction of inter-class design defects. Inter-class design defects [10, 32, 39] appear in the architecture of the application, their detection requires understanding the software architecture. Inter-class design defects are difficult to define independently of the application and its context (foreseen evolution, life span, cost, ...): A same architecture may be valid for an application and incorrect for another. Thus, we need a repository of good designs independent of the context for reference. We must restrict the problem of automatically detecting and correcting inter-class design defects to defects that are context-independent and related to well-known architecture.

We make *four hypotheses*: (1) Design patterns published in [15] embody quality architectural solutions, independent of the context or of the language; (2) Inter-class design defects and design patterns relate to one another; (3) The detection of groups of entities similar to a design pattern corresponds to the detection of a subset of design defects; and, (4) The modification of the groups of participants and their relationships, such that they comply precisely with the design pattern, improves the quality of the architecture.

The first hypothesis is legitimate because the patterns in [15] assume a general Smalltalk- or C++-level object-oriented programming language. These patterns address common everyday-life object-oriented programming problems. Therefore, we assume that these design patterns are language and domain independent. The second hypothesis reminds that software architecture is the common denominator between design patterns and inter-class defects. The third and fourth hypotheses are working hypotheses [31] and thus arguable. Future studies on the automatic detection and correction of inter-class design defects will prove, disprove, or modify these hypotheses.

3.2: The design patterns as references of good designs

[15] contains twenty-three design patterns. A design pattern describes a standard solution to solve a common design problem. This solution captures design concepts of experienced software engineers [4]. It is presented in a semiformal record including four essential sections: (1) A name identifying the pattern in a unique and meaningful way (related to the problem addressed); (2) A description of the design problem, with examples or situations where this problem may arise; (3) A solution to the problem, including graphical representations (based on OMT), entities participating to the solution and their responsibilities; and, (4) A list of the consequences and results to be expected from the solution.

In our attempt at automating design defects detection and correction, we choose to ignore the section describing the problem addressed by a design pattern. The problem is presented more often as the rationale to apply the pattern rather than as the design defects to correct. It is difficult to deduce formal detection rules from the rationale, although they are highly useful to understand the purpose of the pattern. On the contrary, design patterns solutions define good design architecture (or micro-architecture [25]) in a semiformal way. We can use these solutions as our repository of references. Detecting these solutions in source code help understanding the application [25, 28, 43] and we believe that the detection of architectures similar, but not equivalent, to these solutions highlights poor design solutions [20] needing improvements. We use the formalization of design patterns solutions[5] using our meta-model for both code understanding and design defects detection.

3.3: A meta-model for design patterns

There exist several meta-models for representing design patterns but none are specifically designed towards detection and code transformation. [30] introduces a meta-model for design patterns instantiation and validation but without support for code generation. In the tool PatternGen [36], the meta-model does not support code generation (another module handles it) and it offers no patterns detection. In [11], the fragments-based system allows only representation and composition of design patterns.

We define a meta-model that handles uniformly design patterns instantiation[6] and detection. The meta-model embodies a set of entities and the interaction rules among them. All the entities needed

[5] In the rest of this paper, we refer to design patterns solutions as "design patterns" because the solutions capture the very intent (architecture) of the design patterns.

[6] Instantiation refers to the transformation of existing code to comply with the specification of a design pattern or to the production of the code implementing a design pattern.

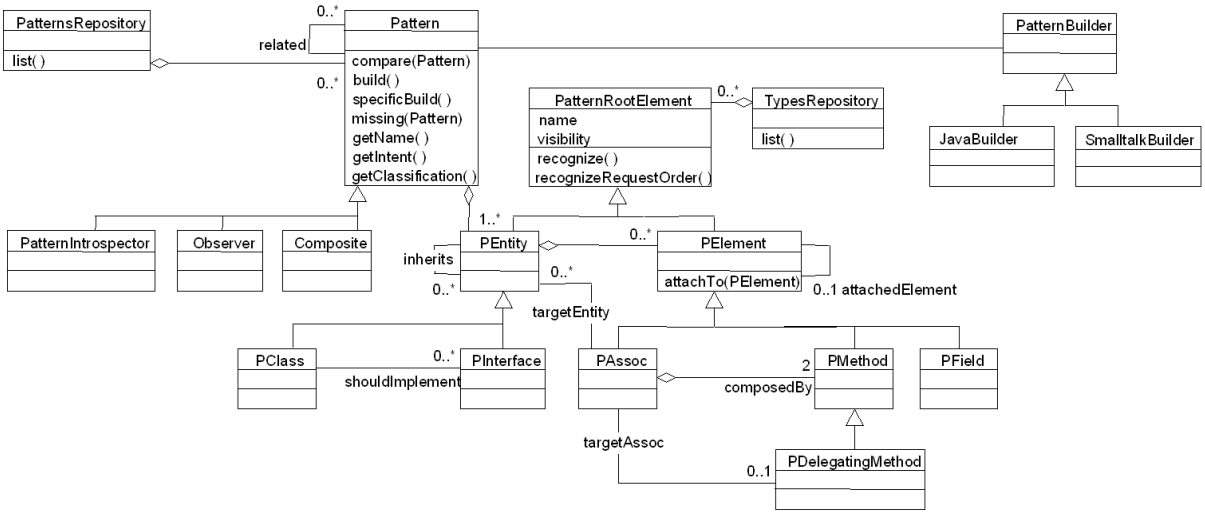

Figure 2: A UML-like partial representation of our meta-model.

to describe structure and behavior of design patterns introduced in [15] are present. Figure 2 shows a fragment of the meta-model.

A model of pattern is reified as an instance of a subclass of class `Pattern`. It consists in a collection of entities (instances of `PEntity`), representing the notion of participant as defined in [15]. Each entity contains a collection of elements (instances of `PElement`), representing the different relationships among entities. If needed, new entities or elements can be added by specialization.

The meta-model defines the semantics of the design patterns. A pattern is composed of one or more classes or interfaces, instances of `PClass` and `PInterface` and subclasses of `PEntity`. An instance of `PEntity` contains methods and fields, instances of `PMethod` and `PField`. The association and delegation relationships are expressed as elements of `Pentity`, using the `PAssoc` and `PDelegatingMethod` classes. The detailed rules of operation and use of the meta-model are out of the scope of this paper and will not be discussed further.

We formalize design patterns using the meta-model. The reified design patterns become first-class entities that we can manipulate and about which we can reason. We use the reified design patterns to generate source code and to detect similar set of entities (classes and interfaces) in source code: Design patterns are detected and instantiated depending on their declarations, not depending on external specifications. Section 4 presents an example of how the meta-model is used to describe design patterns.

3.4: The meta-model, the constraints, and the transformation rules

We use the meta-model to define abstract models of design patterns. An abstract model describes the entities of a design pattern and their relationships. It is qualified as *abstract* because of its independence from any particular context. We use this abstract model to detect sets of entities matching the description of this design pattern in a given source code. Once the identification of sets of entities done, we transform the corresponding source code such that it complies with the design pattern abstract model.

To detect sets of entities similar to a design pattern, we use a constraints solver with automatic constraint relaxation [18]. We express the constraints using the Claire programming language [8] v2.5.4, used to implement the Propagate and Learn with Move (PaLM) constraints solver. The PaLM solver [23] is different from other constraint solvers (such as the *Ilog* constraint solver) because it provides contradiction explanation and automatic constraint relaxation. A contradiction explanation is a set `<X, E>`, where `X` is a set of variables, and `E` $\subset$ `C` a subset of the constraints, such as `<X, E>` is a problem with no solution. Each constraint is weighted and the system automatically relaxes the constraints from `E` within a given weight range when no more solutions are found. The weights do not create a hierarchy of constraints. They distinguish constraints from one another and solutions from one another.

This approach presents two main advantages: (1) The system automatically discovers distorted forms of a design pattern in source code from a precise definition of it. Thus, this approach differs from the use of logic programming [25, 33, 43] where developers need to foresee and explicitly express all possible distortions. (2) Each solution of a distorted form is associated with the set of constraints relaxed to obtain it. Thus, it is possible to explain a solution according to the constraints relaxed. For example, if a constraint states `class must be public`, the distorted solutions – found without this constraint – reference this constraint as the reason why they are solutions.

A constraint is a set <<V_1, D_1>, <V_2, D_2>,..., <V_n, D_n>, R> where V_i is a variable, D_i the domain of the variable and R the relation to be maintained among the variables V_i over their domains D_i. The detection, in a given source code, of the entities which relationships satisfy the description given by an abstract model of a design pattern implies that: (1) The variables corresponds to the constituents of the abstract model. Therefore, a variable may be any of the constituents defined by the meta-model (i.e., a class, an interface, an association, etc.). (2) The domain of a variable is the set of all the constituents of the source code which types are equal to the variable type. (3) The relationships among variables enforce the relationships declared in the abstract model. For example, the `class must be public` constraint is defined as:

`ClassMustBePublic:` <<aClass, D_class> `public` $\in$ `aClass.modifiers>` with: D_class = {All the classes in given the source code}, Class.modifiers = {Modifiers of a class}

Because we use the constituents extracted from the source code as domains for the variables corresponding to the constituents of the abstract models, we exploit the same meta-model to describe both the abstract models of design patterns and the source code.

A distorted solution knows which constraints the solver relaxed to obtain it (i.e., which constraints specified by the design pattern abstract model are not verified). We use this information to automatically transform the source code. Transformation rules are associated with each constraint. These rules express the transformations our transformation engine, JavaXL [2], performs on the source code to verify the associated constraint. JavaXL is a source-to-source transformation engine dedicated to Java. Its purpose is to provide a set of operations to modify any Java source code. For example, the transformation rule: `Class c | c.setVisibility(PUBLIC)` contains the operation: `setVisibility(PUBLIC)` and is associated with the constraint `class must be public`. A solution with this constraint relaxed contains classes that are not public[7] . The transformation is applied automatically on all the classes of the solution to change their visibility to `public`.

4: The guidelines and the Composite pattern

This section illustrates our guidelines using the Composite pattern example. The Composite pattern composes "*objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of object uniformly*" ([15] page 163). We use the implementation version where child management is defined in the Composite class, which is the most common use of this pattern (for instance, see classes `Component` and `Container` of the Java AWT[8] package). The Composite pattern is a relatively simple structural design pattern and is often used as an example either for code production [6] or for detection [5, 43]. Despite its simplicity, it includes all the common constituents of structural patterns: Classes and interfaces; methods and fields; and, inheritance, association, and delegation relationships.

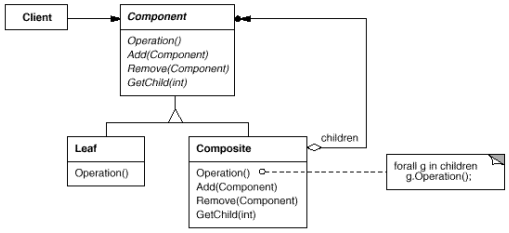

Figure 3: The Composite pattern architecture.

Using the Composite pattern, the guidelines consist: In (1) defining the abstract model of the design pattern to detect. The pattern Composite is described using the constituents provided by the meta-model. This description is straightforward following the preceding class diagram. In (2) declaring the detection and transformation rules according to the design pattern specification. In (3) modelling the given source code using the meta-model. In (4) applying the detection rules on this model. And, in (5) applying the transformation rules associated with the constraints relaxed for each solution (if a solution corresponds to the complete specification of the design pattern, it is not associated with any relaxed constraint, therefore no transformation rules are applicable).

A tool, Ptidej[9] , integrates these guidelines. It is used (1) to map the meta-model over a given source code; (2) to offer a graphical representation of the mapping; (3) to call the solver; (4) to present the solutions to the constraints system; (5) to apply the transformations; and, (6) to present graphically the modified source code.

4.1: The Composite pattern abstract model, detection, and transformation rules

The abstract model is expressed as the Java class `Composite`, subclass of the class `Pattern`, in a declarative manner using constituents of the meta-model. The table on the left, in Figure 4, shows a fragment

[7] Since the constraint `class must be public` has been removed, `public` classes also belong to the distorted solutions.

[8] Abstract Window Toolkit

[9] Patterns Trace, Identification, Detection and Enhancement for Java.
A demo is available at `www.yann-gael.gueheneuc.net/Work/PtidejDemo.html`

of this declaration. The table on the right, in Figure 4, shows the constraints used to detect groups of entities similar to the Composite pattern into a given source code. The constraints also contains the transformation rules.

Composite pattern abstract model definition
`class Composite extends Pattern {`
The declaration takes place in the `Composite` *class constructor*
`Composite(...) {` `...`
Declaration of the `Component` *actor*

`component = new PInterface("Component")`
`operation = new Pmethod("operation")`
`component.addPElement(operation)`
`this.addPEntity(component)`

Declaration of the association children *targeting the* Component *actor with cardinality* n
`children = new PAssoc("children", component, n)`
Declaration of the Composite *actor*

`composite = new PClass("Composite")`
`composite.addShouldImplement(component)`
`composite.addPElement(children)`

The operation *method defined into the* Composite *actor implements the method* operation *of the* Component *actor and is linked to it through the association* children

`aMethod = new PdelegatingMethod( "operation", children)`
`aMethod.attachTo(operation)`
`composite.addPElement(aMethod)`
`this.addPEntity(composite)`

Declaration of the Leaf *actor*

`leaf = new PClass("Leaf")`
`leaf.addShouldImplement(component)`
`leaf.assumeAllInterfaces()`
`this.addPEntity(leaf)`
`}`

Declaration of services specific to the Composite pattern
`...`
For example, the service addLeaf *to dynamically adds new* Leaf *actor to the current instance of the Composite pattern*

`void addLeaf(String leafName) {`
`PClass newPClass = new PClass(leafName)`
`newPClass.addShouldImplement(`
`(PInterface)getActor("Component"))`
`newPClass.assumeAllInterfaces()`
`newPClass.setName(leafName)`
`this.addPEntity(newPCclass)`
`}`

Definition of the detection and transformation rules
*Four variables are declared for the constraint problem: One for each entities of the pattern (*Composite,Component*, and* Leaf*) and one for the type of the association between* Composite *and* Component*. The type could just be* Component*, however, we need an extra variable to allow a finer-grain detection of similar groups of entities.*

`[problemForCompositePattern() : PalmEnumProblem ->`
`...`
`let pb := makePalmEnumProblem("Composite Pattern",`
`length(listC), length(listC)),`
`leavesTypes := makePtidejIntVar(pb, "LeavesType", ...`
`leaves := makePtidejIntVar(pb, "Leave", ...`
`composites := makePtidejIntVar(pb, "Composite", ...`
`components := makePtidejIntVar(pb, "Component", ...`

The entities playing the role of component must be sub-classes of the entities playing the role of composite.

`post(pb, makeStrictInheritanceConstraint(`
`"Composite, Component |`
`javaXL.XClass c1, javaXL.XClass c2 |`
`c1.setSuperclass(c2.getName());",`
`composites, components), 50),`

The entities playing the role of leaves must have a common super-class and this super-class must be the entity playing the role of LeavesTypes *in the given source code.*

`Post(pb, makeStrictInheritanceConstraint(`
`"Leave, LeavesType |`
`javaXL.XClass l1, javaXL.XClass l2 |`
`l1.setSuperclass(l2.getName());",`
`leaves, leavesTypes), 75),`

An inheritance constraint exists among the entities playing the role of LeavesTypes *and the entities playing the role of* Components.

`Post(pb, makeInheritanceConstraint(`
`"throw new RuntimeException(\"...");",`
`leavesTypes, components), 16),`

The entities playing the role of composite have a composition link with the entities playing the role of leaves.

`Post(pb, makeCompositionConstraint(`
`"throw new RuntimeException(\"...");",`
`composites, leaves), 90),`

The type of the components must be the same as the type of the common super-class of all the leaves, LeavesTypes*. This constraint is not deduced from the abstract model. It is added by experience.*

`Post(pb, makePropertyTypeConstraint(`
`"throw new RuntimeException(\"...\");",`
`composites, leavesTypes, componentsType), 90),`

Composite and leaves are different.
`Post(pb, composites <> leaves, 100), pb)]`

Figure 4: The Composite pattern abstract model, detection, and transformation rules.

4.2: The Composite pattern in a text document description application

The input source code in which we want to detect the Composite design pattern is a simple application of text document description. A `Document` contains elements (class `Element`), which can be `Title`, `Paragraph` or indented paragraph, `ParaIndent`. The `Main` class creates an instance of `Document` and fills it up with titles, paragraphs, and indented paragraphs. The relationships among the classes `Element`, `Document`, `Title`, `Paragraph` and `ParaIndent` are typical of a Composite pattern. However, class `Document` should be a subclass of `Element` because we want a uniform interface between the composition object and individual objects.

We apply the constraints defined for the Composite pattern on the source code corresponding to the application modelled using the meta-model. Figure 5 (on the left) shows a graphical UML-like[10] representation of the application.

4.3: The application of the detection

The constraints solver generates the set of all the groups of entities similar to the abstract model of the Composite pattern found. These groups are visible on the Figure 5 (in the middle) as the gray boxes

[10]A class is depicted as a box containing the class name. An association is represented as a plain arrow from the aggregate class to its component with a diamond at its base. A dotted arrow pictures an instance creation. A square line with an empty triangle corresponds to inheritance.

outlining the classes. The group of entities `Element`, `Paragraph` and `Document` is one example. The greater is the number of constraints relaxed, the less similar to the Composite pattern is the group solution and the lighter are the outlining boxes.

4.4: The application of the transformations

The gray-shaded boxes represent entities belonging to (at least) one group of entities similar to the Composite pattern. When a box is selected, the tool highlights all the entities belonging to this particular group and presents the related information: The degree of similarity of group with the original abstract model, the constituents of this group, their values and the associated transformation rules.

On the following example the group composed of classes `Element`, `Document` and `Paragraph` is similar at 50% to the Composite pattern. The transformation to apply is given by the `XCommand` field:

```
Composite, Component |
javaXL.XClass c1, javaXL.XClass c2 |
c1.setSuperclass(c2.getName());
```

That is, the class playing the role of `Composite` must be subclass of the class playing the role of `Component`: In the example, the class `Document` must be subclass of class `Element`.

The transformation engine performs automatically the modifications on the application by executing the `XCommand` on the source code. Then, the result is loaded back into the tool. Figure 5 (on the right) illustrates the resulting architecture of the application:

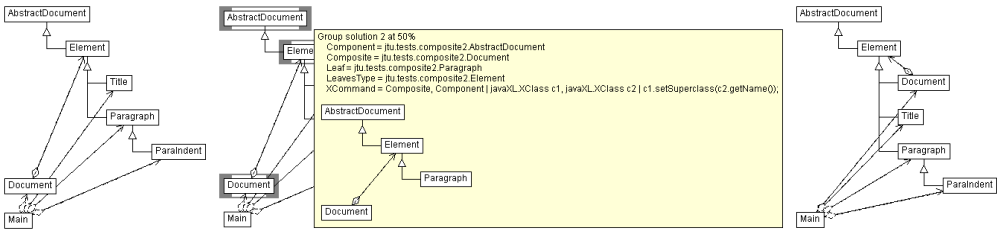

Figure 5: The original application, a selection, and the refactored application.

5: The Guidelines: fitness and pitfalls

We now discuss our guidelines: It is desirable to automate detection and correction of design defects related to design patterns and guidelines using constraints are original. However, this problem is difficult, and full automation is unachievable. We present here observations that lead us to lessen our claim for a fully automatic reengineering process.

- **Description of the design patterns**. The meta-model we introduced is specifically oriented towards design pattern detection and instantiation. It allows us to describe and to detect most of the design patterns in [15]. However, our meta-model suffers the limitations of other meta-models: (1) It lacks of expressiveness with respect to the dynamic aspect of the application, it expresses relationships among entities, not among instances; (2) It needs to be further specialized to allow a finer-grain control over the constraints and transformation rules. We demonstrate this problem when declaring the constraints associated with the Composite pattern. We need a constraint on the type of the entities playing the role of leaves to detect a wider range of similar groups. This problem is comparable to the use of logic programming, although it is less severe because it does not compromise the automation of the detection of a subset of design defects. It increases only the number of false negative, and thus decreases the number of groups of entities found to need improvements.
- **Definition of the detection rules**. We express the detection rules as constraints over the set of the meta-model constituents. The set of rules to detect a given design pattern is so far written by hand independently of the pattern abstract model. It is one of our objectives to automatically generate the set of detection rules needed to detect a design pattern from the corresponding abstract model. Furthermore, we want to use the weights associated with each constraints as a mean to differentiate solutions. However, setting weights automatically may not be possible and may impede further the automation.
- **Application of the constraints**. The domain of the constraints is the given source code. We express the source code with the same meta-model used to describe design patterns,

because the variables in the constraints correspond to the constituents of the design pattern abstract models. We need to ensure that a model of the given source code can be obtained automatically. Because not all the information is always decidable, the lack of information limits the expressiveness of the meta-model and the automation of the detection process. For example, if a generic collection[11] is used, the type of an association may be known only at runtime. Scalability is also an issue: Mapping our meta-model over a large application is a slow and fragile process and requires a large amount of memory. This fragility reduces the range of applications we can deal with. However, once the meta-model is obtained, the detection does not present the same weaknesses: Constraints are already used to solve industrial problems.

- **Application of the transformations rules**. The system can automatically apply all the needed transformations to convert distorted forms of design patterns into the described forms. Sets of transformation rules are associated with each constraint. They contain the transformations necessary to modify the source code to satisfy the constraints. But, it is unlikely that transformation rules can be associated with all constraints. We deduce constraints from design pattern abstract models, using meta-model constituents. Transformation rules work at the source code level. We cannot cross easily the semantic gap between constraints and transformation rules. For example, the composition constraint of the Composite pattern involves an abstraction that has many implementations at the source code level.

6: Conclusion

In this paper, we first introduced a classification of design defects based on their scope within the application and related to the automation of their detection and correction. The three main categories are *intra-class* design defects, *behavioral* design defects, and *inter-class* design defects. This classification shows that intra-class design defects, related to the structure of classes (classes, methods, fields, etc.), have been extensively studied, whereas inter-class design defects, related to the architecture of the application, need to be further considered.

This classification into three categories (plus seven additional composite categories) is an interesting starting point. However, the classification does not provide a sufficiently precise division. We believe we ought to divide design defects into additional categories. For example, we will divide intra-class design defects into: (1) syntactic design defects; (2) structural design defects; and, (3) intra-methods design defects. Such an extra partitioning will distribute design defects more accurately and will make possible the comparison of related techniques and of detection and correction tools. The classification also needs to include more works from the literature.

Then, we propose guidelines for the automatic detection and correction of inter-class design defects, based on design patterns and using constraints with explanations. We hypothesize that design patterns represent good architectures and that pieces of code similar to design patterns represent potential places for improvements. We define a meta-model used to model design patterns and source code. We deduce detection rules and transformation rules manually from design pattern abstract models. The detection rules are defined as constraints on the source code, associated with transformation rules that modify the source code such that it complies with the constraints.

However, we must develop our hypotheses: We must consider domain specific-knowledge in the meta-model because what seems to be an improvement in most cases may introduce a defect in few cases. Moreover, the guidelines do not address the different levels of abstraction between detection constraints and transformation rules. The preceding limitations reduce our aspiration to fully automatic detection and correction of design defects.

We plan to further investigate the possibilities of including domain-specific knowledge and mechanisms to measure the degree of improvement — using design metrics [7]. Another direction of research consists in integrating both constraints and transformation rules in the meta-model. It will imply a finer-grain definition of the meta-model that may impede its robustness and conciseness. Two distinct meta-models may be a solution: One (high-level) for the design pattern specification and another (low-level) for the source code description, with the constraints and the transformation rules linking the two levels.

We will also test our guidelines on larger real-life applications to evaluate precisely its efficiency and scalability. For example, JHotDraw [14] contains more than 125 classes and identifies several design patterns.

Finally, we will develop our guidelines into a complete methodology including other categories of design defects.

[11] The class `Vector` in Java is an example of generic collection: All of its elements are typed as `Object` instances at compile-time.

Acknowledgements

We thank Pierre Cointe, Kevin McGuire, Annya Romanczuk, and Isabelle Borne for their comments and insights on this paper.

References

[1] H. Albin-Amiot. Java code conventions - complément de la version officielle de sun microsystems, inc. Technical report, Soft-Maint, 1998.
[2] H. Albin-Amiot. JavaXL, a Java source code transformation engine. Technical Report 2001-INFO, Ecole des Mines de Nantes, 2001.
[3] K. Beck. *Extreme Programming Explained: Embraced Change*. Addison-Wesley, 1999.
[4] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998.
[5] K. Brown. Design reverse-engineering and automated design pattern detection in Smalltalk. Technical Report TR-96-07, University of Illinois at Urbana-Champaign, 1996.
[6] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2), 1996.
[7] D. N. Card and R. L. Glass. *Measuring Software Design Quality*. Prentice Hall, Englewood Cliffs, New Jersey 07632, USA, 1990.
[8] Y. Caseau and F. Laburthe. CLAIRE: Combining objects and rules for problem solving. *Proceedings of JICSLP, workshop on multi-paradigm logic programming*, 1996.
[9] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. *Proceeding of TOOLS*, 30:18–32, 1999.
[10] S. Demeyer, S. Ducasse, and O. Nierstrasz. Object-oriented reengineering OOPSLA'00 tutorial. *OOPSLA Tutorial Notes*, 2000.
[11] G. Florijn, M. Meijers, and P. V. Winsen. Tool support for object-oriented patterns. *Proceedings of ECOOP*, 1997.
[12] B. Foote and W. F. Opdyke. Life cycle and refactoring patterns that support evolution and reuse. *Proceeding of PLoP*, 1, 1994.
[13] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.
[14] E. Gamma. JHotDraw, 1998. Available at http://members.pingnet.ch/gamma/JHD-5.1.zip.
[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
[16] T. Géraud and A. Duret-Lutz. Generic programming redesign of patterns. *Proceedings of EuroPLoP*, 2000.
[17] Y.-G. Guéhéneuc. Design defects: A taxonomy. Technical Report INFO-2001, Ecole des Mines de Nantes, 2001.
[18] Y.-G. Guéhéneuc and N. Jussien. Quelques explications pour les patrons – une utilisation de la PPC avec explications pour l'identification de patrons de conception. *Proceedings of the 7th JNPC*, pages 953–964, 2001.
[19] S. Hommel. Java code conventions. Technical report, Sun Microsystems Inc., 2000.
[20] J. Jahnke and A. Zündorf. Rewriting poor design patterns by good design patterns. *Proceedings the Workshop on Object-Oriented Reengineering at ESEC/FSE*, September 1997.
[21] R. Johnson. Classic Smalltalk bugs, 1997.
[22] R. E. Johnson and W. F. Opdyke. Refactoring and aggregation. Technical report, Department of Computer Science, University of Illinois, 1993.
[23] N. Jussien and V. Barichard. The PaLM system: Explanation-based constraint programming. *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems*, pages 118–133, 2000.
[24] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.
[25] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. *Proceedings of the Working Conference on Reverse Engineering*, pages 208–215, 1996.
[26] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. *Proceedings of ICSM*, 1998.
[27] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. *Proceeding of OOPSLA*, 1996.
[28] M. O'Cinnéide and P. Nixon. A methodology for the automated introduction of design patterns. *Proceedings of ICSM*, 1998.
[29] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.
[30] B.-U. Pagel and M. Winter. Towards pattern-based tools. *Proceedings of EuropLop*, 1996.
[31] L. B. Railsback. T. c. chamberlin's "method of multiple working hypotheses": An encapsulation for modern students, 2001.
[32] C. Rich and R. C. Waters. The programmer's apprentice: A research overview. *IEEE Computer*, 21(11):10–25, November 1988.
[33] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. *Proceedings of ICSM*, 1999.
[34] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.
[35] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for Smalltalk, 1999.
[36] G. Sunyé. *Mise En Oeuvre de Patterns de Conception : Un Outil*. PhD thesis, Université de Paris 6 - LIP6, July 1999.
[37] T. Teitelbaum and T. Reps. The cornell program synthesizer: A syntax-directed programming environment. *Proceedings of CACM*, 24(9):563–573, September 1981.
[38] F. Tip, C. Laffra, and P. F. Sweeney. Practical experience with an application extractor for Java. *Proceedings of OOPSLA*, pages 292–305, November 1999.
[39] W. C. Wake. XPlorations - from 0 to composite (and back again). Technical report, ACM, 2000.
[40] W. C. Wake. XPlorations - refactoring: An example ; an example, extended. Technical report, ACM, 2000.
[41] B. F. Webster. *Pitfalls of Object Oriented Development*. M & T Books, 115 West 18th Street, New York, New York 10011, USA, 1995.
[42] S. G. Woods, A. E. Quilici, and Q. Yang. *Constraint-Based Design Recovery for Software Reengineering - Theory and Experiments*. Kluwer Academic Publishers, Kluwer Academic Publishers Group, Distribution Center, Post Office Box 322, 3300 AH Dordrecht, The Netherlands, 1998.
[43] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. *Proceedings of TOOLS USA*, pages 112–124, 1998.