JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 7, No. 4, December 2013, pp. 251-262

A Catalog of Bad Smells in Design-by-Contract Methodologies with Java Modeling Language

Thiago Viana*
Coordenaçao de Sistemas de Informaçao, Instituto Federal de Pernambuco, Recife, PE, Brasil
tamnv@recife.ifpe.edu.br

Abstract

Bad smells are usually related to program source code, arising from bad design and programming practices. Refactoring activities are often motivated by the detection of bad smells. With the increasing adoption of Design-by-Contract (DBC) methodologies in formal software development, evidence of bad design practices can similarly be found in programs that combine actual production code with interface contracts. These contracts can be written in languages, such as the Java Modeling Language (JML), an extension to the Java syntax. This paper presents a catalog of bad smells that appear during DBC practice, considering JML as the language for specifying contracts. These smells are described over JML constructs, although several can appear in other DBC languages. The catalog contains 6 DBC smells. We evaluate the recurrence of DBC smells in two ways: first by describing a small study with graduate student projects, and second by counting occurrences of smells in contracts from the JML models application programming interface (API). This API contains classes with more than 1,600 lines in contracts. Along with the documented smells, suggestions are provided for minimizing the impact or even removing a bad smell. It is believed that initiatives towards the cataloging of bad smells are useful for establishing good design practices in DBC.

Category: Human computing

Keywords: Java Modeling Language; Bad smells; Design-By-Contract; Refactoring

I. INTRODUCTION

Bad smells naturally arise in source code, usually as a consequence of *ad hoc* evolution. These smells consist of symptoms that convey likely problems, even though the program is working correctly. Examples of bad smells in object-oriented programs that motivate refactorings [1] stem from classes or methods that are too long to classes using more members from other classes than its own members (Feature Envy [1]). Refactoring activities are often motivated by the detection of bad smells.

Design-by-Contract (DBC) [2] establishes a method for building software by explicitly specifying what each function in a module requires in order to correctly operate, and also specifying what it provides to the caller (*contracts*). They constitute a collection of assertions—mainly invariants and pre- and post-conditions for methods—that precisely describe what methods require and ensure with respect to client classes. Although DBC is a built-in development method for the Eiffel programming language [3], contracts can also be written with extensions to general-purpose languages, such as the Java Modeling Language (JML) [4].

With the increasing adoption of DBC methodologies in

Open Access http://dx.doi.org/10.5626/JCSE.2013.7.4.251 http://jcse.kiise.org

Received 24 July 2013, **Accepted** 12 August 2013
*Corresponding Author

 pISSN: 1976-4677 eISSN: 2093-8020

formal software development, evidence of bad design practices can similarly be found in programs that combine actual production code with contracts. If these problems are not addressed properly, they may hinder the quality benefits of DBC development, such as encapsulation, maintainability, and readability.

In this paper, we present a catalog of bad smells that may appear during DBC practice. Six smells were catalogued for this paper; and are described in detail. Smells include symptoms like long specifications with several alternative behavior cases (*Long Specs*), private fields exposed in public contracts (*Open Doors*), excessive accesses to fields that represent internal data (*Field Obsession*), and complex predicates that easily become very difficult to read and understand (*Illogical Contracts*). This work considers JML as a language for specifying contracts [5]. Smells are described over JML constructs, although several can appear in other DBC languages. Along with names and symptoms, actions are suggested to eliminate or minimize the effect of these smells. It is believed that initiatives towards cataloging bad smells are useful for establishing good design practices in DBC.

The recurrence of the catalogued bad smells are evaluated in two ways: first by describing a small study with graduate student projects, and second by counting occurrences of smells in library classes from the JML models application programming interface (API). This API offers classes that support specifications in JML, many of which present rather complete specifications. The API contains classes with approximately 1,600 lines of contracts. One class was chosen for each category by sampling: out of 113 classes and interfaces, a representative subset of six files was picked. The analyses showed at least one type of bad smell in every exemplar, with a total of seven distinct smells. Despite the focus of this API being verification rather than DBC, it is assumed that they will be read and manipulated by developers, so detecting the presence of bad smells is desirable.

Discussions about related work and conclusions are included into Sections VI and VII, respectively. The contributions of this paper are summarized as follows.

- A catalog of DBC code smells with JML as the target contract language (Section IV).
- Evaluation of bad smells in student projects, and classes from the JML models API (Section V).

II. JAVA MODELING LANGUAGE

The JML is a behavioral interface specification language [4] tailored to Java. JML serves to describe contracts with static information that appears in Java declarations and how they act. JML specifications are written in the form of *special annotation* comments that are inserted directly into the source code of programs. These comments must begin with an at-sign (`@`) and can be written in two ways: by using `//@ ...` or `/*@ ... @*/`. The following fragment shows contracts for a class `Person` [5].

```
public class Person {
  //@ public model int weight;
  //@ public model String name;

  private String _name;
  private int _weight;

  //@ private represents name <- _name;
  //@ private represents weight <- _weight;

  //@ public invariant !name.equals("") &&
  //@                  weight >= 0;

  //@ ensures \result == weight;
  public /*@ pure @*/ int getWeight() { .. }

  /*@ requires kgs > 0;
    @ assignable weight;
    @ ensures weight == \old(weight + kgs);
    @*/
   public void addKgs(int kgs) { .. }
}
```

The `model` modifier introduces specification-only fields, which are also called *model fields*. A model field should be thought of as an abstraction of a set of concrete fields used in the implementation of this type and its subtypes [6]. In the class `Person`, there are two model fields, `name` and `weight`, representing the concrete attributes `_name` and `_weight` via the `represents` clause, respectively.

The `invariant` clause defines predicates that are true in all visible states of the objects of a class. The invariant in the example has public visibility and establishes that the value of the attribute `_name` is different from an empty string, and that the value of `_weight` is greater than or equal to zero.

JML uses the `requires` clause to specify the obligations of the caller of a method regarding what must be true to call a method. For instance, the precondition of the method addKgs insists on the added value to be greater than zero. A postcondition specifies the implementor's obligation regarding what must be true at the end of a method, just before it returns to the caller. In JML, the `ensures` clause introduces a postcondition. In the example, it asserts that the value of the attribute `_weight` at the end of the method `addKgs` is equal to the value of the expression `\old(weight + kgs)`. The value of an expression in the pre-state of a method can be referred to by using the `\old` operator.

The `assignable` clause gives a frame axiom for a specification. Only the locations named and their associations can be assigned during method execution. In the method `addKgs`, it is stated that only `weight` is change-

able. The JML modifier purely indicates that the method does not have any side effects and can hence appear in specifications.

III. MOTIVATING EXAMPLE

In this section, the importance of finding bad smells is expressed, as are the kinds of problems bad smells may bring to DBC/JML developers. As an example, consider a document editing system for LaTeX-based [7] papers. The system allows papers to be written in collaboration using a Web-based interface. The internal structure includes classes, such as `Document`, `Section`, and `Author`, among others. A simplified UML class diagram is shown in Fig. 1.

This diagram shows that a Document can be written by 1 main Author and can have contributions from 0 or many other Authors. The diagram also shows that a Document can have many versions. Also, a Document is composed of Sections, and each Section can have 0 or many Commands (such as links or buttons), Figures, or Tables.

Focusing the investigation on the `Document` class, version control is a system requirement. Versions can be defined as a list of objects in the first document object. This can be implemented as a private field in Java.

```
public class Document {
  private ArrayList versions = new ArrayList();

  void changeVersion() {
    if (this.versions != null)
      versions.add(new Document(..));
  }
}
```

In order to specify a contract for users of this class, developers may introduce invariants over `Document` objects. A plausible invariant is to avoid states in which a document appears more than once in the list of versions. Following the DBC approach, the invariant is visible to other objects, so it can be specified with a JML public invariant. As long as `versions` is a private field, JML offers a modifier that allows the field to appear in public invariants called `spec_public`, as shown in the next fragment.

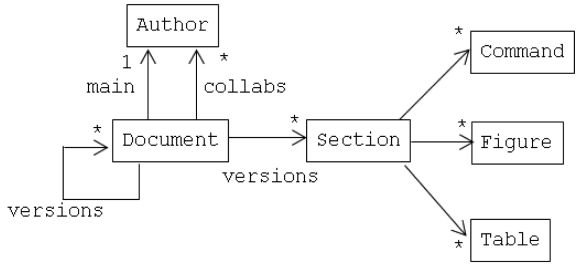

Fig. 1. Online document editing system.

```
public class Document {
  private /*@ spec_public @*/ ArrayList
      versions = new ArrayList();

  //@ public invariant !versions.contains(this)
      ;
..}
```

The contract for `Document` is syntactically correct and provides the desired constraint. However, careful analysis raises a number of issues with encapsulation.

- As contracts are part of the public interface, clients will use them as a basis for development. In this case, clients rely on implementation details for the class, as the `ArrayList` field is visible.
- The contract relies on a private field when using the `Document` class. This scenario may cause classes to be harder to change, without affecting other classes.

Alternative designs can be applied to avoid encapsulation issues, with the same practical issues. The JML language allows developers to hide the internal details of a class from a contract and its clients. *Model fields* can be used for this purpose, representing a more abstract view of class data, as they can be freely viewed by clients. The implementation for the model field must be provided by concrete, possibly private fields. The `represents` clause allows for the expression of functional abstractions between a model and concrete fields. Model field types can be defined as immutable objects provided by the JML models API [8], which provides classes that emulate mathematical objects (including sets, bags, and sequences). These types are appropriate for abstract fields. In this example, the `JMLEqualsSet` class is used, which implements a set of values.

```
public class Document {
  private ArrayList _versions = new ArrayList()
      ;

  //@ public model JMLEqualsSet versions;
  //@ public invariant !versions.has(this);
..}
```

A concrete definition for this model field must be provided by the class developer. Here, a `JMLEqualsSet` object is built from the elements in the concrete `_versions` list. The following `represents` clause illustrates this approach, with a *model method*. Model methods provide a useful abstraction for procedures and functions that will only be used within JML contracts. In this case, the method copies every element from the concrete list to the mathematical set, so the model field can be evaluated.

```
public class Document {..
  //@ private represents versions <-
  //@                       abstractVersions();

  /*@ private model pure JMLEqualsSet
    @   abstractVersions() {
    @     JMLEqualsSet ret = new JMLEqualsSet()
        ;
    @     Iterator it = _versions.iterator();
    @     while (it.hasNext()) {
    @       ret.insert(it.next());
    @     }
    @     return ret;}
    @*/
..}
```

In this particular scenario, the method may be overkill, with a very operational way of filling the mathematical set. It would be hard to write and maintain such code for numerous methods. This clause can be refactored to a better solution by using methods from the JML models API itself (the `getVersions` method).

```
public class Document{ ..
  //@ private represents versionsSet <-
  //@   JMLEqualsSet.convertFrom(getVersions())
      ;
..}
```

Regarding method contracts, in the class `Section`, the `beginEdition` method locks the section for editing by request by a given author. The preconditions state that the section is unlocked, and the requesting author is included as an author for the given document (which cannot be null). As a postcondition, the contract guarantees that the section is locked after the call.

```
public class Section {..
  /*@ requires !locked;
    @ requires document.authors.has(author);
    @ requires document != null;
    @ ensures locked;
    @*/
  public void beginEdition(Author author){..}
..}
```

The repetitive use of fields in the specification can be observed, even though the correct behavior is provided. As creating a model field for each concrete field clutters the class, an alternative can be found for better encapsulating internal details. A good option is using a public getter method (and if one does not exist, it could be created).

```
public class Section {..
  /*@ requires !isLocked();
    @ requires isAuthorIncludedInDoc(author);
    @ requires getDocument() != null;
    @ ensures isLocked();
    @*/
  public void beginEdition(Author author){..}
..}
```

In this example, several correct contract parts can be considerably improved for DBC contexts. These "bad smells" are catalogued in this work, so that efforts towards effective DBC development can take advantage of tool support for detecting these smells, and refactorings can then be more effectively employed.

IV. A CATALOG OF BAD SMELLS IN DBC DEVELOPMENT

In this section, a catalog of bad smells in contracts that result from DBC development is provided. As shown in the example, smells are not necessarily hazardous, but they show evidence of possible errors and hard-to-change programs. For a more concise and uniform explanation of bad smells and for ease of identification, a specific format for smell descriptions was adopted. The format used for the catalog was inspired by code smells from Wake's book on refactoring [9]. Smells are described with the following properties.

- **Brief Description**: Description emphasizing the problems behind the smell.
- **Symptoms**: Clear signs of the described bad smells in contracts and code.
- **Example**: Example of the bad smell, using the system described in Section III.
- **Causes**: Likely ways of having this smell present in the program.
- **What to do**: Ideas on how to refactor the program for eliminating or minimizing the impact of the smell (although this is not the focus in this paper).
- **Example Solution**: A possible refactored program.
- **Payoff**: Advantages in avoiding these smells in terms of general quality of DBC development.
- **Contraindications**: Situation in which removing the smell may not be desirable.

Focused was centered on JML as a contract language, so smells are in principle specific to particular JML constructs. Still, they are likely to appear in similar constructs in other contract languages (JML, Eiffel, Spec# [10], among others). Six smells are described in detail: *Open Doors, Field Obsession, Illogical Contracts, Complexity Magnetism, Long Specs,* and *Specification Overkill.* The set of DBC smells catalogued so far are presented in Table 1.

A. Bad Smell: Open Doors

1) Brief Description

Contracts that expose private data, threatening encapsulation.

2) Symptoms

Indication is given with direct access to private or protected

Table 1. Catalog of Design-by-Contract smells

Name	Brief description
Open Doors	Contracts that expose private data
Field Obsession	Many direct accesses to variables in contracts
Illogical Contracts	Contracts with logics that gets too hard to understand
Complexity Magnetism	Complex definition of model fields
Long Specs	Unnecessary heavyweight style
Specification Overkill	Excess of redundant specifications

fields (or methods) in public contracts. In JML, it is made explicit with the `spec_public` or `spec_protected` modifiers. The forall operator represents a universal quantifier, with three components separated by semicolons: variable declaration, variable delimitation, and Boolean expression with the predicate.

3) Example

```
public class Document {
  private /*@ spec_public @*/ Document[]
      versions;
  /*@ invariant(
    @   \forall int i; 0<=i && i<versions.length
        ;
    @     versions[i].getAuthor().equals(
    @                              getAuthor())
    @ );
    @*/
..}
```

4) Causes

Developers need a way to specify which components of internal data must be modified in method contracts or invariants that have visibility that is less restricted than data. Also, this can be caused by a lack of abstraction when specifying methods.

5) What To Do

In cases where fields must be used to indicate state changes and contracts, a model field can be created, representing a hidden concrete field. The contracts must then be changed to use the model field, replacing accesses to the concrete field. However, if many fields are used in contracts, developers should consider using query methods instead (see Section IV-B). Regarding methods, model methods can be used, in which code delegates the call to concrete methods.

6) Solution to the Example

```
public class Document {
  /*@ public model JMLEqualsSet versionsSet;
    @ invariant(
    @   \forall int i; 0<=i && i<versionsSet.
        size;
    @     ((Document)versionsSet.itemAt(i)).
    @         getAuthor().equals(getAuthor())
    @ );
    @*/
..}
```

7) Payoff

Encapsulation is promoted, even in the presence of contracts. Removing this smell tends to bring abstraction to contracts, which is highly desirable in DBC.

8) Contraindications

Excess model fields tend to increase the complexity of contracts. In this case, query methods should be a better option. In some fields, however, query methods might not be usable due to some encapsulation requirement in the application. In this case, the contracts should be revised, as they could possibly not expose these particular fields.

B. Bad Smell: Field Obsession

1) Brief Description

There are an excessive number of direct field accesses in contracts, making these contracts more sensitive to changes in internal data.

2) Symptoms

Excess direct access to several fields from a class in contracts.

3) Example

For the `setSections` method, the contract includes precondition and postcondition and includes the JML `assignable` clause, which indicate frame conditions (variables that may possibly be assigned values). The invariant refers to fields and `sections`.

```
public class Document { ..
  //@ public invariant 0<=n && n<sections.
      length;

  /*@ assignable sections,n;
    @ ensures n == secs.length
    @ ensures(
    @   \forall int i; 0<=i && i<n;
    @     sections[i] == secs[i]
    @ );
    @*/
  public void setSections(Section[] secs){..}
..}
```

4) Causes

In general, this smell happens when developers avoid using methods, perhaps fearing long specifications. This can also happen regardless of concrete or model fields.

5) What To Do

Mainly, solutions must include the use of query (accessor) methods. In Eiffel, for instance, field accesses automatically behave as query methods. JML does not offer this feature, since it would demand changes in the semantics of the programming language (Java).

6) *Solution to the Example*

```
public class Document { ..
  //@ public invariant 0<=getSectionsSize() &&
  //@   getSectionsSize()<sections.length;

  /*@ assignable sections, n;
    @ ensures getSectionsSize() == secs.length;
    @ ensures (\forall int i;
    @         0<=i && i<getSectionsSize();
    @         getItemFromSections(i)==secs[i]);
    @*/
  public void setSections(Section[] secs){..}
..}
```

7) Payoff

Besides adding abstraction to contracts, this solution usually makes the contract easier to understand.

8) Contraindications

The contract can get considerably long if too many fields are accessed. In this scenario, contracts can be rewritten with improvements by using auxiliary model methods (see Section IV-C).

C. Bad Smell: Illogical Contracts

1) Brief Description

Contracts defining predicates with logics that become too hard to understand.

2) Symptoms

The bad smell *Illogical Contracts* is identified by long specifications where a chain of Boolean predicates is presented. Long and chained Boolean predicates are hard to read and understand, especially when universal quantifications are used.

3) Example

In the `Document` class, there may be an invariant stating that the sections for older versions of a document must be present in the later versions. In JML, this must be written as two universal quantifications.

```
public class Document { ..
/*@ invariant(
  @  \forall int i; 0<=i && i<versionsSet.size;
  @   (\forall int j; 0<=j && j<sectionsSet.
      size;
  @           ((Document)versionsSet.
  @           itemAt(i)).sectionsSet.
  @           has(sectionsSet.itemAt(j)))
  @ );
  @*/
..}
```

4) Causes

Some complex specifications may be required. This might be more common in JML, as it follows Java syntax with extensions, which makes Boolean predicates verbose. Abstraction in this scenario is a challenge.

5) What To Do

Declaration of JML-pure Boolean methods that represent predicates. In JML, *pure methods* are required to have no side effects. The methods are used in contracts as auxiliary predicates. In this context, naming is important for improving readability.

6) Solution to the Example

In JML, auxiliary predicates can be declared as model methods.

```
public class Document{ ..
  /*@ invariant(
    @  \forall int i; 0<=i && i<versionsSet.
       size;
    @    hasAllOldVersionSections(versionsSet
    @             itemAt(i)
    @ );
    @ public model pure boolean
    @   hasAllOldVersionSections(Document d){
    @     return d.sectionsSet.has(
    @       getBaseDocumentSections());
    @ }
    @
    @ public model pure JMLEqualsSet
    @   getBaseDocumentSections(){
    @     return this.sectionsSet;
    @ }
    @*/
..}
```

7) Payoff

The solution helps raising abstraction in contracts, making them easier to read and edit.

8) Contraindications

In some situations, it may be necessary to create too many methods, which can make the specification longer than the original. Therefore, it is sensible to revise the contracts in order to find better ways to rewrite complex statements (which is not always possible). In addition, if predicates become substantially complex due to the application domain, separation and comments in natural language can be used, especially if the contracts will not be subject to tool-assisted verification. For instance, JML allows for predicates in natural language, although they are not considered for reasoning.

D. Bad Smell: Complexity Magnetism

1) Brief Description

Model fields with complex definitions in `represents` clauses.

2) Symptoms

When using abstract model fields, the `represents` clause may become very complex. This scenario hinders readability and maintainability, especially for class implementers (clients should not have access to `represents`).

3) Example

In this case, the `versionsSet` model field has a complex definition, which is encapsulated in a model method. This method copies each element of the concrete `ArrayList` to the abstract set.

```
public class Document { ..
  //@ public model JMLEqualsSet versionsSet;

  //@ private represents versionsSet <-
  //@    abstractVersions();

  /*@ private model pure JMLEqualsSet
    @   abstractVersions() {
    @     JMLEqualsSet ret = new JMLEqualsSet()
        ;
    @     Iterator it = versions.iterator();
    @     while (it.hasNext()) {
    @       ret.insert(it.next());
    @     }
    @     return ret; }
    @*/
..}
```

4) Causes

This situation arises depending on how hard it is to abstract from concrete data. Also, the complexity of specific data structures is critical (for example, collection manipulation from Java).

5) What To Do

For simple collections, the JML model API offers methods like `convertFrom`. The mathematical toolkit of the language should support this solution. In cases that result in complex model methods that abstract away details from internal data, other auxiliary methods can be extracted (analogous to the Extract Method refactoring [1]).

6) Solution to the Example

```
public class Document{ ..
  //@ public model JMLEqualsSet versionsSet;

  //@ private represents versionsSet <-
  //@    JMLEqualsSet.convertFrom(getVersions()
      );
..}
```

7) Payoff

Contracts become easier to write and understand by implementers. Abstraction in general is higher. Despite this example, a variation of this smell can happen in the opposite situation: a one-line definition of `represents` can become too complex. In this case, the model method should be included.

E. Bad Smell: Long Specs

1) Brief Description

Unnecessary heavyweight style in DBC applications, covering every possible behavior for methods.

2) Symptoms

In JML, developers may use two styles of contracts (specification cases): heavyweight or lightweight. For the first style, in contrast to the latter, JML expects that developers only omit parts of the specification when the default is appropriate, specifying behavior completely. This is indicated by the clauses `normal_behavior` and `exceptional_behavior`. An excess of heavyweight contracts obstructs good DBC development, decreasing abstraction and readability.

3) Example

In the contract, the `also` keyword indicates the complementarity of specification cases. The `signals_only` and `signals` constructs define the Java exceptions possibly thrown by the method in exceptional behavior.

```
public class Document { ..
/*@ public normal_behavior
  @   requires s.getAuthor().equals(a) &&
  @            !s.isLocked();
  @   assignable s;
  @   ensures s.isLocked();
  @ also
  @ public exceptional_behavior
  @   requires s.isLocked();
  @   assignable nothing;
  @   signals_only Exception;
  @   signals (SectionLockedException e);
  @*/
  public void editSection(Section s, Author a)
      throws SectionLockedException {..}
..}
```

4) Causes

Sometimes, it is required that specifications be replicated, with the intention of specifying complex contracts. Also, developers may wish to document every situation in which exceptions are thrown.

5) What To Do

In DBC, lightweight contracts are much more desirable, as both clients and implementers should often refer to contracts as documentation. In this scenario, excep-

tional behaviors should be removed, omitting exceptional situations that are not relevant for implementing the program. Also, the `normal_behavior` clause can be removed.

6) Solution to the Example

```
public class Document { ..

  /*@ requires s.getAuthor().equals(a) &&
    @          !s.isLocked();
    @ assignable s;
    @ ensures s.isLocked();
    @ signals (SectionLockedException e);
    @*/
  public void editSection(Section s, Author a)
    throws SectionLockedException {..}
..}
```

7) Payoff

More importantly, better readability and maintainability of contracts. In avoiding heavyweight specifications in DBC, another benefit arises in that developers eliminate the risk of introducing specification mistakes by accidently defining overlapping specification cases (superposition of preconditions that do not make clear which one is valid for a given state).

8) Contraindications

When JML is applied in complete tool-assisted verification or test-case generation, covering all specification cases is critical. In this scenario, this is certainly not considered a bad smell.

F. Bad Smell: Specification Overkill

1) Brief Description

Excess of redundant specifications.

2) Symptoms

Repeated use of redundant clauses, such as `ensures_redundantly` or `requires_redundantly`, which means that the contract was already valid in the given context, usually for documentation purposes. Even different predicates with the same semantic are allowed.

3) Example

```
public class Document {
  /*@ assignable versionsSet;
    @ requires v != null;
    @ ensures versionsSet.has(v);
    @ ensures_redundantly
    @   (\exists
    @     int i; 0<=i && i<versionsSet.size;
    @       versionsSet.itemAt[i] == v);
    @*/
  public void addVersion(Version v){..}
..}
```

4) Causes

This bad smell may appear to be caused by needs for repeating specifications with the intention of explaining something complex. But, in other contexts, this can expose a lack of attention to the existing predicates.

5) What To Do

Mostly, redundant clauses can be removed, because the redundancy might show the existence of an unnecessary specification. In some cases, a good revision of predicates is enough. Also, it is better to use the simplest forms of the specifications, because if it is possible to re-explain something more simply, it is always best to use this simpler explanation in a single turn.

6) Solution to the Example

```
public class Document {
  /*@ assignable versionsSet;
    @ requires v != null;
    @ ensures versionsSet.has(v);
    @*/
  public void addVersion(Version v) {..}
..}
```

7)Payoff

The specification becomes clearer and more easily understood. Inconsistencies are avoided as redundancy is removed.

8) Contraindications

In a number of situations, the use of `ensures_redundantly` is justified as a good way to explain complex contracts. In these cases, it is important to look for ways to leave contracts in the simplest form possible. There is still an alternative of removing Smell Section IV-C.

V. EVALUATION

With the goal of finding evidence of bad smells appearing in contracts, an attempt was made to detect their characterization in systems with considerable use of contracts. First, graduate students in medium-sized projects were observed, and an account of the experience is provided in Section V-A. The students had little experience with formal methods. In another experiment, contracts written by more experienced developers were analyzed for similar smells. Some classes annotated with contracts from the JML models API were used, and Section V-B shows the results.

A. JML Student Projects

The first idea of gathering bad smells initially developed from observing five graduate student projects for JML-

based subjects at the University of Pernambuco (Brazil) in the Systems Specification and Verification course. Targeting diverse application domains, students had to write contracts and code from previously elicited requirements, during three small iterations. Requirements were modeled using Alloy [11] as a formal specification language.

In the first iteration, it was observed that students complained mostly about the JML syntax, since they had never experienced the language or DBC, but the source code did not present noticeable issues regarding encapsulation and readability. Some projects did not properly apply model fields for implementing abstract concepts, so the *Open Doors* smell was often found in initial versions of these projects.

In later iterations, students had to evolve previously written contracts in order to accommodate modifications from changing requirements. This scenario is commonly seen in real project settings, so it is likely to occur if DBC is used in such a context. Contract maintenance became more difficult as the end of the projects drew near, and some smells were more recurrent, such as *Open Doors*, *Long Specs*, *Field Obsession,* and *Illogical Contracts.* Results from these projects feature most smells from the catalog.

B. JML Models API Analysis

Greater evidence of how often the smells can appear in applications was found when the JML models API source code was analyzed [8]. Classes of this API offer an ample spectrum of specifications. For instance, a single class, `JMLEqualsSequence`, contains more than 1,600 lines of specification code. The classes were developed by experienced JML developers (some of whom are JML pioneers), and they certainly constitute one of the richest available sources of contracts, considering open source systems. Currently, most well-designed DBC applications with JML employ classes from this API in contracts, showing its substantial importance.

The API offers classes for abstract specifications, as they emulate mathematical objects (such as sets, bags, and sequences). It includes more than one hundred classes and interfaces, all of which contain JML contracts. From this API, Collection objects are equally divided in Object, Value, and Equals collections. Object collections treat components as object references, while Value collections have values, for which any inserted object is cloned from the original reference. Equals collections are hybrid, using the Java equals methods. The API also offers classes that represent Relations and Maps [6]. In the analysis, with the intent of analyzing a representative subset of classes, one API class was chosen with no previous inspection for each of the following categories.

- Basic type: `JMLChar`,
- Composite type: `JMLString`,
- Object collection: `JMLObjectSet`,
- Value collection: `JMLValueBag`,
- Equals collection: `JMLEqualsSequence`,
- Interface: `JMLCollection`,
- Relation: `JMLEqualsToEqualsRelation`.

These six classes are representative in the sense that other classes present very similar specifications for different types, so the same smells are likely to be detected. The results of manual detection of bad smells in the selected classes are shown in Table 2. These numbers are *an interpretation* of the analyzed specifications, under the DBC perspective. Developers of the API aim at complete mathematical specifications (which they call "equational reasoning"), not DBC development. Still, developers must inspect these specifications in order to gain deep understanding of the API services, so avoiding bad smells can be relevant.

The number of detected bad smells is proportional to the class size, which is not surprising, since smells are more frequent in bigger contracts. In JML, due to its use of Java expressions, more complex predicates tend to become expressions that are several lines long. Further on in the results, some of the catalogued smells were not found. On the other hand, other smells are recurrent: *Long Specs*, *Illogical Contracts,* and *Open Door*s. It is believed that these smells and their variations are the most commonly found in practice (the same result was observed in students' projects).

Among the selected classes, `JMLValueBag` happens to exhibit more smells than any other. For instance, the protected field `size` is made available to contracts by `spec_public`. Fields in this class are constants since the JML mathematical types are immutable [8].

```
public /*@ pure @*/ class JMLValueBag
  extends JMLValueBagSpecs implements
      JMLCollection{..

  protected final JMLValueBagEntryNode the_list
      ; ..
  protected /*@ spec_public @*/ final int size;

  //@ public invariant size >= 0; ..
```

Exemplars of *Field Obsession* are seen in method postconditions, such as with the `getMatchingEntry`. This method, which declares non-pure methods that have no assignable clause, also present the *Purely Nothing* smell.

```
/*@  assignable \nothing;
  @  ensures the_list==null ==>
  @       \result == null;
  @  ensures \result != null ==>
  @       0<=\result.count &&
  @       \result.count<=size;
  @*/
protected JMLValueBagEntry getMatchingEntry
    (JMLType item){..}
```

Table 2. Case study results

Classes from the API	Approx. contract size (LOC)	Detected smells
JMLChar	155	*Long Specs*
JMLString	105	*Long Specs*
JMLObjectSet	508	*Open Doors, Specification Overkill, Long Specs*
JMLValueBag	821	*Open Doors, Field Obsession, Specification Overkill, Long Specs, Illogical Contracts, Purely Nothing*
JMLEqualsSequence	1,600	*Long Specs, Illogical Contracts*
JMLEqualstoEqualsRelation	645	*Long Specs*
JMLCollection	44	-

API: application programming interface, LOC: line of contract, JML: Java Modeling Language.

Instances of *Long Specs* and *Specification Overkill* are found in some method contracts, as in convertFrom.

```
/*@ public normal_behavior
  @    requires a != null;
  @    ensures \result != null &&
  @        \result.int_size()==a.length;
  @    ensures_redundantly \result!=null &&
  @       \result.int_size()==a.length &&
  @       (\forall int i;
  @         0<=i && i<a.length;
  @         \result.has(a[i])
  @       );
  @*/
  public static JMLValueBag convertFrom(
      JMLType[] a){..}
```

The intent with this evaluation was only to provide evidence that smells are present in applications with contracts in JML. Still, experiments must be carried out in order to point out the most frequent smells and reason about causes that lead to smells in DBC code.

VI. RELATED WORK

Fowler et al. [1] presented 22 bad code smells as indications that there are issues in code that can be solved by refactoring. Among the bad smells presented, one of the most commonly found in practice is **Duplicate Code**. All the code smells are textually described along with variations of them. For each variation, there are suggestions of refactorings to solve the smell. Wake [9] presented bad smells by means of a standard format which introduces the smell name, symptoms, and causes, and also indicated what to do, the payoff, and contraindications. In this paper, a similar structure has been adopted. Moreover, Wake classified code smells into categories according to their relation to classes, regarding whether they can occur within classes or between classes. In addition, Wake added six bad code smells to the work by Beck and Fowler.

Work that closely resembles bad smells for DBC can be found in the book by Mitchell and McKim [12]. They list *principles* that should be followed by DBC developers for achieving quality contracts. These principles include spatial organization of contracts (by separating queries from commands, or basic from derived queries), as well as better ways to write precondition or postcondition (based on query methods), and guidelines for writing effective invariants. It can be assumed that failure in fulfilling these principles probably results in bad smells. In the present catalog, it is believed that bad smells like *Illogical Contracts* and *Field Obsession* result from a lack of similar principles. Regarding the JML models API, previous analysis has found errors in specifications (as the case with JMLObjectSet [13]). It is believed that bad smells can lead to errors if not addressed during evolution activities.

Mantyla et al. [14] developed a taxonomy of bad code smells, classifying the 22 code smells proposed by Fowler into 7 higher-level categories, which describe code that has grown so that it cannot be handled, unnecessary code that can be removed, and code that hinders the software modification, among other characteristics. Furthermore, they developed an empirical study that provides an initial correlation between code smells. For instance, they observed in the software studied that the *Message Chains* smells correlate with the opposite smell *Middle Man*. A similar study with bad smells in DBC would surely have correlations, like *Long Specs* and *Illogical Contracts.*

Garcia et al. [15] defined *architectural smells* as a commonly used architectural decision that negatively impacts system quality. One of their causes can be the applications of design abstraction at the wrong level of modularity. As an example, the *Scattered Functionality* smell is related to a system in which multiple components realize the same high-level concern, but some of them also are responsible for orthogonal concerns. This smell indicates a violation of the separation of concern principle.

Tsantalis et al. [16] presented the JDeodorant Eclipse

plug-in which identifies *Type-Checking* bad smells in Java source code. They presented two cases of smells. In the first, an attribute of a class acts as a type field; in the second case, there is a conditional statement that casts a reference from a superclass type to the actual subclass type in order to invoke methods of a specific subclass. The code smells are removed by application of the refactorings "Replace Type Code with State/Strategy" and "Replace Conditional with Polymorphism" [1]. Following a similar approach, it is believed that DBC smells can be automatically detected as well.

Regarding smells that can appear when using languages with modeling purposes, Correa et al. [17] defined OCL [18] *smells* as constructions present in OCL expressions that affect the understandability or maintainability of OCL specifications. The *Implies Chain* OCL *smell*, for example, corresponds to OCL expression in the form *b1 implies (b2 implies b3),* which can be solved by the application of a refactoring *Replace Implies Chain by a Single Implication.* Other OCL *smells* are associated, for example, with expressions that are bigger than necessary *(Verbose Expression*), or with the use of OCL operations to give the type of objects on which the result of an expression depends *(Type Related Conditionals*). These smells are mostly related to the DBC smell *Illogical Contracts.* An experimental study was also presented to evaluate the impact of *smells* and refactorings on the understandability of OCL specifications. The study indicated the negative impact of OCL *smells*. Cabot and Teniente [19] proposed a set of equivalence transformation rules for OCL expressions. They did not present OCL *smells* explicitly, but the reason for defining the rules was that a designer may not define constraints the best way according to intents like understandability or efficiency. Reynoso et al. [20] investigated the hypothesis that the structural properties of OCL expressions affect the understandability and reduce maintainability. They defined a suite of measures for the structural properties of OCL expressions. In particular, they investigated the relationship between object coupling in OCL expressions and the understandability and modifiability of these expressions. For instance, they found that the number of classes used in navigations, and the number of collection operations, influence understandability efficiency. The number and length of navigations influence modification tasks. They considered that the results are empirical evidence that the coupling defined in OCL expressions by means of navigations and collection operations is correlated with the maintainability of OCL expressions.

VII. CONCLUSION

This paper has presented a catalog of bad smells that recur in DBC development with JML. Bad smells are not bugs, but might bring problems in the readability and extensibility of contracts, offering opportunities for refactoring. The paper's focus is not on refactoring, although actions were suggested for removing bad smells as part of their description.

The main source of bad smells in DBC was a number of graduate student projects that involved writing contracts at several development stages. In order to evaluate the recurrence of these smells in more robust projects, several classes from the JML models API were manually analyzed. Currently, most well-designed DBC applications with JML employ classes from this API in contracts, showing its substantial importance. The most recurrent smells were detected in the analyzed classes, showing that they can happen in real development contexts.

From the catalogued smells, it can be concluded that good DBC contracts are succinct and abstract. For effective DBC development, contracts must be a source of readable information for implementers and clients. Most importantly, they should not threaten encapsulation in the specified program. Otherwise, good programming practices can be made useless if the associated contracts expose internal data. Also, contract size is often an important alarm for the need of simplification, as shown in the results of the evaluation.

This catalog of DBC smells is the first step towards the definition of techniques for refactoring programs with contracts. Based on previous results [21, 22] in synchronizing model and program refactorings, future work will include the investigation of how refactored contracts affect programs (and vice-versa). The work with bad smells will point out targets for refactoring in the context of DBC.

Regarding extensions of the bad smells catalog, further steps will consider more advanced JML constructs (such as refinement), in addition to issues related to behavioral subtyping. Moreover, plans have been established to build a tool support to automatically detect the smells and to analyze other classes of the JML API.

REFERENCES

1. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Boston, MA: Addison-Wesley, 1999.
2. B. Meyer, "Design by contract," *Advances in Object-Oriented Software Engineering*, D. Mandrioli and B. Meyer, editors, New York, NY: Prentice-Hall, 1992, pp. 1-50.
3. B. Meyer, *Eiffel: The Language*, New York, NY: Prentice-Hall, 1992.
4. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G.T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 3, pp. 212-232, 2005.
5. G. T. Leavens and Y. Cheon, "Design by contract with JML," ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf.

6. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Muller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl, "JML reference manual," Technical Report, 2008.
7. L. Lamport, *LATEX: A Documentation Preparation System: User's Guide and Reference Manual*, Reading, MA: Addison-Wesley, 1994.
8. G. T. Leavens, C. Ruby, and A. L. Baker, "Package org.jmlspecs.models," http://www.eecs.ucf.edu/~leavens/JML-release/javadocs/org/jmlspecs/models/package-summary.html.
9. W. C. Wake, *Refactoring Workbook*, Boston, MA: Addison-Wesley, 2003.
10. M. Barnett, K. R. M. Leino, and W. Schulte, "The spec# programming system: an overview," in *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, Marseille, France, 2004, pp. 49-69.
11. D. Jackson, *Software Abstractions: Logic, Language and Analysis*, Cambridge, MA: MIT Press, 2006.
12. R. Mitchell and J. McKim, *Design by Contract, by Example*, Boston, MA: Addison-Wesley, 2002.
13. A. Darvas and P. Muller, "Faithful mapping of model classes to mathematical structures," *IET Software*, vol. 2, no. 6, pp. 477-499, 2008.
14. M. Mantyla, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *Proceedings of the International Conference on Software Maintenance*, Amsterdam, The Netherlands, 2003, pp. 381-384.
15. J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, Kaiserslautern, Germany, 2009, pp. 255-258.
16. N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: identification and removal of type-checking bad smells," in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, Athens, Greece, 2008, pp. 329-331.
17. A. Correa, C. Werner, and M. Barros, "An empirical study of the impact of OCL smells and refactorings on the understandability of OCL specifications," in *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, Nashville, TN, 2007, pp.76-90.
18. Object Management Group (OMG), *UML 2.0 OCL Specification*, OMG Adopted Specification ptc/03-10-14, 2003.
19. J. Cabot and E. Teniente, "Transformation techniques for OCL constraints," *Science of Computer Programming*, vol. 68, no. 3, pp. 179-195, 2007.
20. L. Reynoso, M. Genero, M. Piattini, and E. Manso, "Does object coupling really affect the understanding and modifying of OCL expressions?" in *Proceedings of the ACM Symposium on Applied Computing*, Dijon, France, 2006, pp. 1721-1727.
21. T. L. Massoni, R. Gheyi, and P. Borba, "An approach to invariant-based program refactoring," in *Proceedings of the 3rd Workshop on Software Evolution through Transformations: Embracing the Change*, Natal, Brazil, 2006.
22. T. Massoni, R. Gheyi, and P. Borba, "Formal model-driven program refactoring," in *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering*, Budapest, Hungary, 2008, pp. 362-376.

Thiago Affonso de Melo Novaes Viana

Thiago Affonso de Melo Novaes Viana has earned his Sc.D. in Education from UFPB in 2003, M.S. in Soft Engineering from UPE in 2010, and B.A. in Computer Science from IFPE in 2008. He is a Professor in Computer Science at Federal University of Pernambuco (Universidade Federal de Pernambuco, UFPE) from 2011.