Tool Support for Contracts in FeatureIDE

Florian Proksch
University of Magdeburg
florian.proksch@st.ovgu.de

Stefan Krüger
University of Magdeburg
stefan3.krueger@st.ovgu.de

ABSTRACT

In matters of research in feature-oriented programming, the need for efficient analysis and evaluation of existing projects arises prominently. The open-source project FeatureIDE seeks to satisfy the demand for tools of such kind. The implementations concerned within this work therefore aim to offer significant improvements in statistical analyses on the usage of contract-based mechanisms as well as support the work flow in feature projects making use of contracts.

Keywords

Design by contract, contract composition, JML, software product lines, feature-oriented programming, FeatureIDE, FeatureHouse

1. INTRODUCTION

Product lines specify the creation of specialized final goods of any kind, by making use of a singular basis from which all variations are derived. This process offers numerous opportunities from both economical and verification stand points in matters of software engineering [1].
FeatureIDE is an open-source extension for the integrated development environment Eclipse [11][6][9]. The main goal is to provide an interface for numerous feature-oriented programing schemes as well as to enable *WYSIWYG* editing of feature models and software product lines. As such, FeatureIDE is an essential software in various research concerning software product lines and testing capabilities of feature-oriented programming in general. An integral part in the concept of *design by contract* is the ability to specify certain conditions on input and the resulting outcome of operations. The particular working mechanisms of the operation are therefore not of interest as its only purpose is to ensure the previously specified parameters of the generated output. These assurances are called *contracts*. The hereafter described libraries and tools revolve around the realization of the concept of contracting in software engineering. [5] The Java Modeling Language *(JML)* is a behavioral interface specification language, used to specify specific behavior of Java classes [7][4].
FeatureHouse is an open-source tool employed by FeatureIDE for the purpose of implementing feature-oriented project composition, including the generation of feature structure trees (*FST* - i.e., a tree structure composed of software artifacts) of source code [2]. *JML* is used by *FeatureHouse* for implementing contracts, providing the ability to specify logical pre- and post conditions for methods as well as – *class invariants* – guaranteeing integrity constrains throughout every method call within a class. [8][12] *Contracts* can also be *refined* – meaning the composition of multiple instances of contracts [10][3]. FeatureIDE encompassed the functionality of various libraries to generate general statistics about the structure of feature-oriented projects and breaks down the usage of specific keywords and elements.

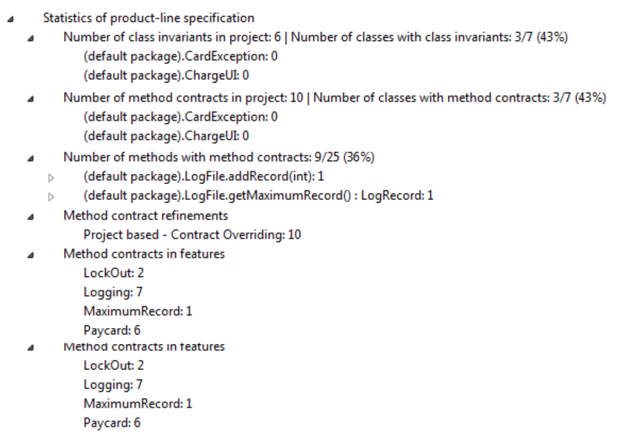
2. STATISTICS VIEW

The previously existing *Statistics View* in *FeatureIDE* has the ability to summarize a feature-oriented project's implementation in the following fashion:

1. Statistics of the feature model
2. Statistics of product-line implementation

In order to be able to visualize a project's structure in relation to the usage of contract-based elements these statistics are to be extended. Proceeding the recent additions made by Benduhn [3] and Weigelt [13], contracts can now also be refined according to method-specific composition mechanisms, instead of exclusively project-wide settings. These changes lead to an increase in the complexity of feature abstraction and composed predicate logical expressions, introducing both new possibilities in project design and challenges in project maintenance.
We therefore propose the addition of several new statistics elements for keeping track of progressing degeneration as well as maintaining manageable class and feature structures. The implemented structure consists of the following values describing contract complexity:

1. Class invariants
2. Method contracts by class
3. Method contracts by method
4. Method contracts by refinement keyword
5. Contracts in features



```
Statistics of product-line specification
    Number of class invariants in project: 6 | Number of classes with class invariants: 3/7 (43%)
        (default package).CardException: 0
        (default package).ChargeUI: 0
    Number of method contracts in project: 10 | Number of classes with method contracts: 3/7 (43%)
        (default package).CardException: 0
        (default package).ChargeUI: 0
    Number of methods with method contracts: 9/25 (36%)
        (default package).LogFile.addRecord(int): 1
        (default package).LogFile.getMaximumRecord() : LogRecord: 1
    Method contract refinements
        Project based - Contract Overriding: 10
    Method contracts in features
        LockOut: 2
        Logging: 7
        MaximumRecord: 1
        Paycard: 6
    Method contracts in features
        LockOut: 2
        Logging: 7
        MaximumRecord: 1
        Paycard: 6
```

Figure 1: Contracts summarization in the statistics view

For each element of the above enumeration, occurrences are shown in relative and absolute numbers respectively. They contain the aggregated values of its substructures in increasing granularity. Data is presented in three separate perspectives, being grouped by either methods, features or classes, as can be seen in Figure 1.

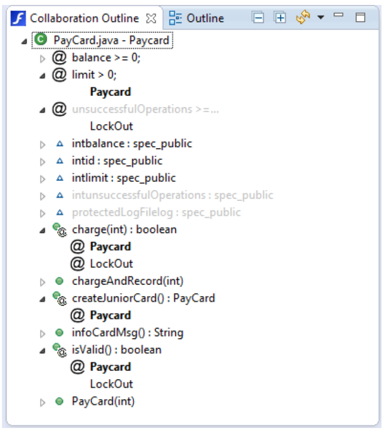
The newly offered data in the statistics view is processed by evaluating the FST created by FeatureHouse. Specifically method contracts and class invariants that are recognized as JML specification are extracted and edited to fit FeatureIDE's internal data structures. Additionally gained data - for instance the contract specification body - is therefore stored and available for usage in future additions to FeatureIDE in a conveniently accessible fashion.

3. COLLABORATION OUTLINE & COLLABORATION DIAGRAM

The collaboration outline is a specialized version of the Eclipse outline providing specific information concerning a role's properties in the feature project [11].

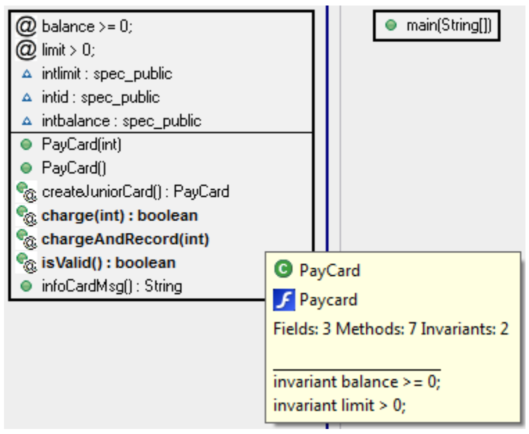
The displayed elements are therefore to be extended by the class invariants included in the role. Additionally, the icons displaying each methods access specifiers are extended to visualize method's having a contract. As *JML* establishes no naming convention for class invariants, the first 20 characters trimmed by any exceeding white spaces and mandatory keywords of the invariant's body are shown. Figure 2 shows the adjusted layout of the collaboration outline. The displayed tree view structure is composed of three different scopes: the currently opened class, the elements of the class – such as class invariants, fields, and methods –, and the features implementing each of the elements. The features implementing a specific element are now also in an alphabetical order and each feature containing a contracted is marked with a new icon.

The collaboration diagram delivers a graphical overview of the implemented features and classes and how they interact by means of roles. Similarly to the collaboration outline, the layout has been extended to display class invariants and method contracts. Figure 3 shows the new layout of the collaboration diagram. Class invariants are now displayed on top of fields and methods and class invariants and methods



```
PayCard.java - Paycard
    @ balance >= 0;
    @ limit > 0;
        Paycard
    @ unsuccessfulOperations >=...
        LockOut
    intbalance : spec_public
    intid : spec_public
    intlimit : spec_public
    intunsuccessfulOperations : spec_public
    protectedLogFilelog : spec_public
    charge(int) : boolean
        @ Paycard
        @ LockOut
    chargeAndRecord(int)
    createJuniorCard() : PayCard
        @ Paycard
    infoCardMsg() : String
    isValid() : boolean
        @ Paycard
        LockOut
    PayCard(int)
```

Figure 2: Contracts in collaboration outline



```
@ balance >= 0;
@ limit > 0;
intlimit : spec_public
intid : spec_public
intbalance : spec_public
PayCard(int)
PayCard()
createJuniorCard() : PayCard
charge(int) : boolean
chargeAndRecord(int)
isValid() : boolean
infoCardMsg() : String

main(String[])

PayCard
Paycard
Fields: 3 Methods: 7 Invariants: 2
invariant balance >= 0;
invariant limit > 0;
```

Figure 3: Contracts in collaboration diagram

with contracts both now have a novel icon. Additionally the information of the tool tip has been adjusted to show class invariants.

4. CONTRACT REFINEMENT WARNINGS

4.1 General Concept

The debugging environment of FeatureIDE has the ability to analyze feature projects, detect errors in the structure of feature projects and issue warnings beyond problems with the programming languages themselves. In the matters of method-based contract refinement numerous errors can arise. For instance depending on the specific selection in a product-line, or even the assigned feature order.

In order to avoid – possibly distracting – interruptions in work flow, the following problems in contract refinement procedure are to be marked as *Warnings* within the Eclipse environment instead of *Errors*.

1. Method-based contract refinement
   FeatureIDE offers the ability to select a project-wide setting according to which the refinement of contract in classes and methods is executed. Any setting but "Method-Based Contract Refinement" will therefore prompt warning markers on each method that specifies method-based keywords, that are in this case simply ignored.

2. Final methods
   The keyword $final\_method$ requires that there is no further refining of the method within the refinement process of the containing class. This property is highly depended on the aforementioned *feature order* and can prompt errors solely by a change in the selected product. If such a problem occurs, all refining functions get a warning marker.

3. Final contracts
   Similarly to the above case, there can be no further contract specification in any method refining another method possessing the keyword $final\_contract$. On occurrence, warning markers are placed consistently with aforementioned case.

4. Original contracts
   Method contracts implementing the *original* keyword have to have a contract that was introduced beforehand. If the feature model allows for the possibility of no introduction beforehand, warnings are issued for the method falsely implementing the *original* keyword.

5. Validity of overriding sequence
   Furthermore the sequence in which contract refinement strategies can be overridden is not universal and valid only in the order seen in Figure 4. In case of a violation, again all involved contracts are marked. [13]

4.2 Implementation Details

Any occurrence of above mentioned errors is computed by inquiries to FeatureIDE's module for checking the satisfiability of predicate logical expressions *(SatSolver)*. Equation 1 shows the expression tested for finding invalid uses

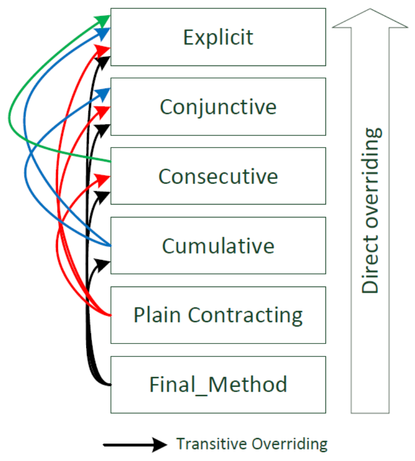


Figure 4: Permitted order of refinement keyword overriding. [13]

of the *original* keyword, whereas FM describes the feature model and f single features indexed by their occurrence in the feature order list. The equation is tested for all $i \in \{1, 2, \ldots, n\}$ with n being the number of features implementing the method in question. The testing ensures that if feature f_i is selected in a configuration, at least one of the features $f_1, \ldots, f_{i-1}$ has to be selected as well, thereby verifying that the method has to have been introduced beforehand.

$$\neg(FM \implies (f_i \implies (f_1 \vee f_2 \vee \ldots f_{i-1}))) \qquad (1)$$

With similar nomenclature as above, Equation 2 is tested for each method containing either the $final\_method$ or $final\_contract$ keyword, verifying that there is no feature f_i refining a method, that contains the respective keyword in a feature f_1 with $i \in \{1, 2, \ldots, n\}$ for n being all features after f_1 in the feature order list. The singular testing of two respective features in Equation 2 instead of the logical union of all features as in Equation 1 is done to be able to directly mark the exact features violating the validity.

$$\neg((FM \wedge f_i) \wedge f_1) \qquad (2)$$

Equation 3 is tested for all pairs of features $(f_i, f_j) \in F_{k_i} \times F_{k_j}$ with F_{k_n} being the set of features containing a refining keyword in ascending feature order and $k_i < k_j$. In the cases that f_j implements a keyword that is not permitted to override the keyword used in f_i, a warning is issued for the concerning method in both features. The order in which keywords are allowed to override each other can be found in fig.4.

$$\neg((FM \wedge f_i) \wedge f_j) \qquad (3)$$

5. CONCLUSIONS

The added functionality offers improved handling of method contracts and class invariants, both during navigation in the project as well as during debugging. Thanks to the contract statistics further research concerning the analysis of feature project structure can now utilize the aggregated data in the FeatureIDE statistics view and it is no longer necessary to determine contract-specific properties of the project by inspecting the source code manually. The newly introduced changes to collaboration outline and diagram allow faster navigation and immediate overview over the contract structure of classes, methods, and including features. For problems with contract-refinement that are hard to detect manually, FeatureIDE now prompts interactive warnings that greatly increase efficiency in debugging of such.

6. ACKNOWLEDGMENTS

This paper was written as part of a team project in the master's course of computer science. The work was supervised by Prof. Gunter Saake and Thomas Thüm of the computer science department of the university of Magdeburg. Additionally we would like to thank Thomas Thüm for his most valuable input, suggestions, and the review of both this paper and the accompanying implementations.

7. REFERENCES

[1] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer, Berlin, Heidelberg, 2013.

[2] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 221–231, Washington, DC, USA, May 2009. IEEE.

[3] F. Benduhn. Contract-Aware Feature Composition. Bachelor's thesis, University of Magdeburg, Germany, 2012.

[4] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An Overview of JML Tools and Applications. *Int'l J. Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[5] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral Interface Specification Languages. *ACM Computing Surveys*, 44(3):16:1–16:58, June 2012.

[6] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: A Tool Framework for Feature-Oriented Software Development. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 611–614, Washington, DC, USA, May 2009. IEEE. Formal demonstration paper.

[7] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[8] G. T. Leavens and Y. Cheon. Design by Contract with JML, Sept. 2006.

[9] T. Leich, S. Apel, L. Marnitz, and G. Saake. Tool Support for Feature-Oriented Software Development - FeatureIDE: An Eclipse-Based Approach. In *Proc. Workshop Eclipse Technology Exchange*, pages 55–59, New York, NY, USA, 2005. ACM.

[10] T. Thüm, F. Benduhn, S. Apel, and G. Saake. Feature-Oriented Contract Composition, 2014. Unpublished Manuscript.

[11] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming (SCP)*, 79(0):70–85, Jan. 2014.

[12] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, and G. Saake. Applying Design by Contract to Feature-Oriented Programming. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, volume 7212 of *LNCS*, pages 255–269, Berlin, Heidelberg, New York, London, Mar. 2012. Springer.

[13] A. Weigelt. Methoden-basierte Komposition von Kontrakten in Feature-orientierter Programmierung. Bachelor's thesis, University of Magdeburg, Germany, Aug. 2013. In German.