An Initial Test Suite for Automated Extract Class Refactorings

Keith Cassell, Peter Andreae, Lindsay Groves, James Noble
School of Engineering and Computer Science
Victoria University of Wellington
Wellington, New Zealand
{kcassell, pondy, lindsay, kjx}@ecs.vuw.ac.nz

September 17, 2010

Abstract

When developing object-oriented classes, it is difficult to determine how to best refactor large, complex classes to create smaller, more cohesive ones. Automated algorithms can recommend solutions, but how can a programmer feel confident that an algorithm's recommendations are good ones? The test suite described here provides test classes for use as inputs to these automated algorithms, together with the preferred results - the class members that should be distributed to the extracted class. By comparing the actual results to the expected ones, a programmer can have some confidence that his algorithms are providing useful suggestions for refactoring.

1 Introduction

Large object-oriented classes are often cited as being maintenance problems. Fowler [8] says that "a class with too much code is prime breeding ground for duplicated code, chaos, and death" and recommends the *Extract Class* refactoring as a remedy for the problem. Demeyer, et al, [6] have similar negative things to say about "god classes", and they also recommend extracting a new class as a solution.

There are several researchers [1, 3, 11, 12] actively developing algorithms that specify how to split large classes. With the output from these algorithms, a programmer can use the Extract Class refactoring to create classes with improved values for several software metrics associated with improved maintainability [5]. Thus, these algorithms are expected to identify splits that will do the most to decrease a class's size, increase its cohesion, and limit its coupling.

As a baseline for evaluating the efficacy of class-splitting algorithms, we have created a test suite of Java classes to serve as input data. These test classes should be noncontroversial; in other words, most programmers should agree that the splits proposed in this paper are desirable and correct. A preliminary version of this report was posted to the Yahoo Refactoring group for comment [2].

This paper does not address the specific criteria for how to select the classes in need of refactoring, in particular, it does not specify thresholds for class size or cohesion. To make the test classes easier to understand, we have made them relatively small. Some people might argue that classes of this size do not warrant refactoring. Lorenz and Kidd [10], for example, state that classes should have fewer than 20 methods and fewer than 5 attributes. Most classes in this suite are smaller than that. (On the other hand, others say that programmers should "refactor unrelentingly" [7], which presumably often creates much smaller classes.)

This remainder of this paper describes the test suite. The body of the paper discusses each test class and its purpose. The code for each test class is stored as part of the *cohesion-tests* open source project.[1]

2 Test Classes

The test suite contains nine classes that illustrate various issues that can complicate refactoring. Following this brief high-level summary of the suite, each test class is discussed in its own subsection.

Inspired by an example of Henderson-Sellers [9], the first set of six test classes have characteristics of both a person and a car. These classes follow the naming pattern `PersonCar*`, e.g. `PersonCarDirect`, `PersonCarAsym`, etc. They are much the same "semantically" - they all have three attributes pertaining to a person (`firstName, lastName, id`), three attributes relating to a car (`make, model, vin`), and get and set accessors for those attributes. They vary in several ways:

- Mechanism of attribute access - Methods may access the attributes directly, indirectly, or both.
- Degree of cohesion - Different classes have differing number of methods that access both the car attributes and person attributes.
- Presence or absence of "special methods" [4] - Some methods, such as constructors, toString, etc., are not really part of the class's business logic, but may be part of the "infrastructure" of the class (toString, equals, hashCode).
- Presence or absence of "cross-cutting" concerns - Some functionality, e.g. logging, is used by many classes but serves an auxiliary purpose.

Each of the `PersonCar*` classes should be split into separate `Person` and `Car` classes.

In addition to the `PersonCar*` collection, there are additional classes for testing how the proposed refactorings deal with:

- Imposed groupings - A proposed refactoring should not break constraints imposed by interfaces or inheritance.
- Method-only classes - Some classes may not have any attributes but still need to be split.

The following subsections provide more detail on each of these test classes. Each subsection discusses an input class, the reason for its existence, and the content of the classes that are the expected output of the Extract Class refactoring. A graph is included in each subsection to help summarize the structure of the class. In these graphs, there are two types of vertices - circles representing methods and stars representing attributes. Arrows between vertices represent either methods calling other methods or methods accessing attributes.

[1] http://code.google.com/p/cohesion-tests/

2.1 PersonCarDisjoint

2.1.1 Input

PersonCarDisjoint.java[2] has two distinct subgroups of interacting members. There are person attributes (`firstName, lastName, id`) with a set of methods that access them and car attributes (`make, model, vin`) with a distinct set of methods that access them. All methods access attributes directly; there are no indirect accesses to attributes, i.e. through other methods.

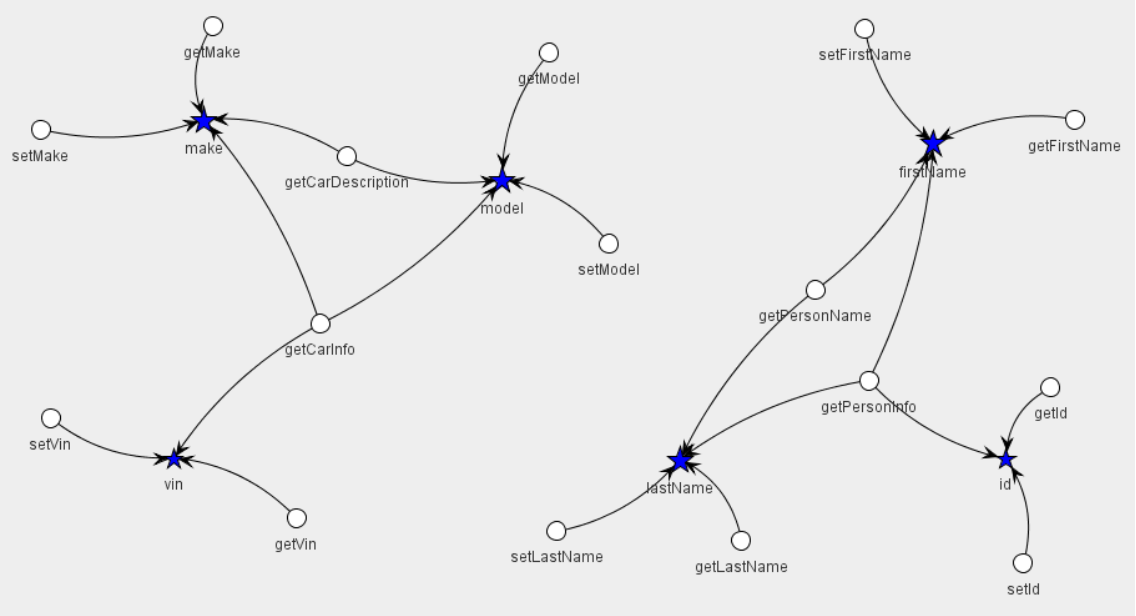

Figure 1: PersonCarDisjoint

2.1.2 Purpose

This test case checks that an algorithm can split classes that have a disconnected graph. This class should be the most straightforward to split. There are two groupings of methods and attributes that do not interact.

2.1.3 Output

PersonCarDisjoint 1 (Person)

firstName, lastName, id, getFirstName, setFirstName, getLastName, setLastName, getId, setId, getPersonInfo, getPersonName.

PersonCarDisjoint 2 (Car)

make, model, vin, getMake, setMake, getModel, setModel, getVin, setVin, getCarInfo, getCarDescription.

[2] http://code.google.com/p/cohesion-tests/source/browse/trunk/src/nz/ac/vuw/ecs/kcassell/personcars/PersonCarDisjoint.java

2.2 PersonCarDirect

2.2.1 Input

PersonCarDirect.java[3] is the same as PersonCarDisjoint.java except that a `toString` method connects the otherwise disconnected person and car members.

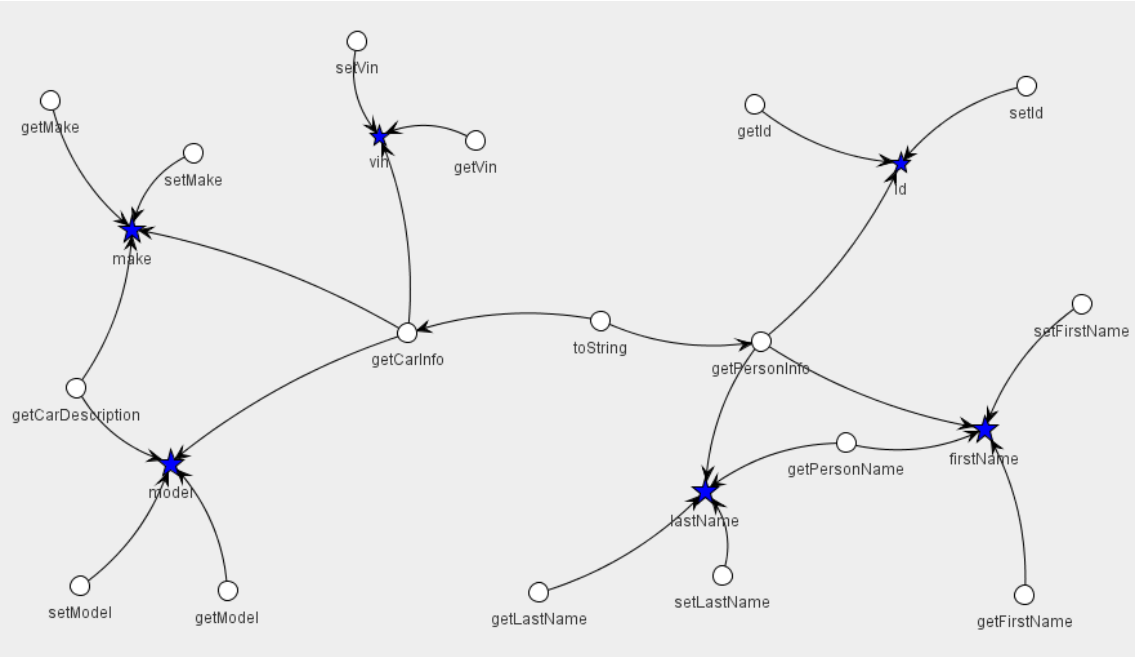

Figure 2: PersonCarDirect

2.2.2 Purpose

This test case checks that the algorithm can split classes that have a connected graph. Because the only connection between the two groups is the `toString` method, this should be split into two classes, each with its own `toString`.

2.2.3 Output

PersonCarDirect 1 (Person)

firstName, lastName, id, getFirstName, setFirstName, getLastName, setLastName, getId, setId, getPersonInfo, getPersonName, toString.

PersonCarDirect 2 (Car)

make, model, vin, getMake, setMake, getModel, setModel, getVin, setVin, getCarInfo, getCarDescription, toString.

[3] http://code.google.com/p/cohesion-tests/source/browse/trunk/src/nz/ac/vuw/ecs/kcassell/personcars/PersonCarDirect.java

2.3 PersonCarIndirect

2.3.1 Input

PersonCarIndirect.java[4] is the same as PersonCarDirect.java except that only *get* and *set* accessors directly access the attributes. All other methods indirectly access the attributes through these.

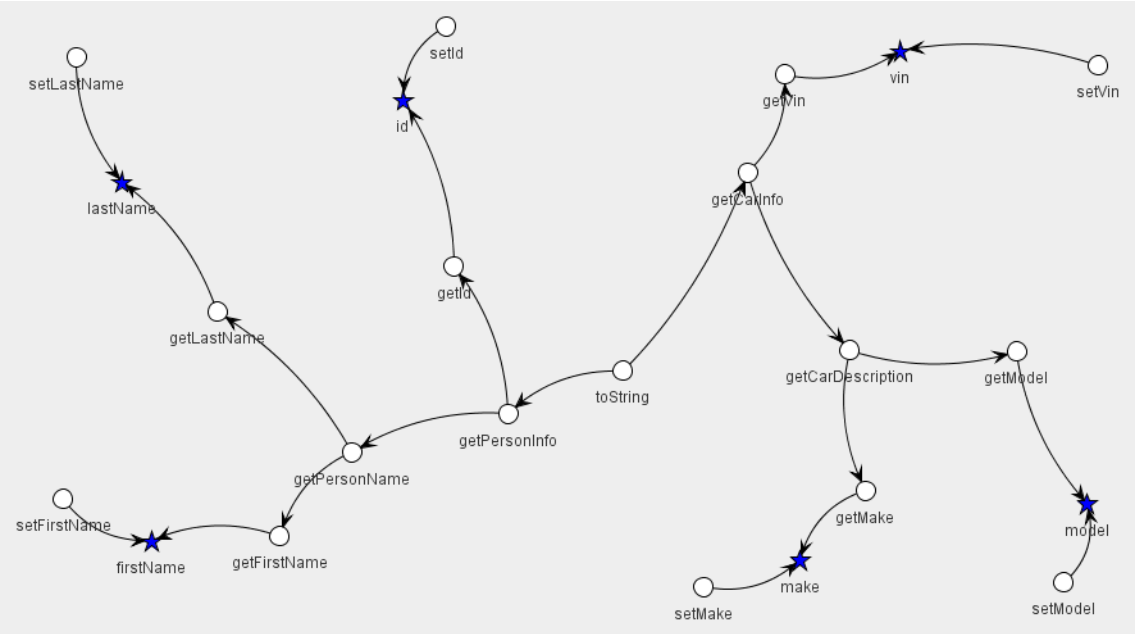

Figure 3: PersonCarIndirect

2.3.2 Purpose

This tests the algorithm's ability to handle classes with connected graphs, where parts of the graph are extended with longer "stringy" call chains.

2.3.3 Output

PersonCarIndirect 1 (Person)

firstName, lastName, id, getFirstName, setFirstName, getLastName, setLastName, getId, setId, getPersonInfo, getPersonName, toString.

PersonCarIndirect 2 (Car)

make, model, vin, getMake, setMake, getModel, setModel, getVin, setVin, getCarInfo, getCarDescription, toString.

[4] http://code.google.com/p/cohesion-tests/source/browse/trunk/src/nz/ac/vuw/ecs/kcassell/personcars/PersonCarIndirect.java

2.4 PersonCarAsym

2.4.1 Input

PersonCarAsym.java[5] is the same as PersonCarDirect.java except that the car methods only use *get* and *set* accessors to indirectly access the attributes. The person methods directly access the person attributes without using *get* and *set* accessors.

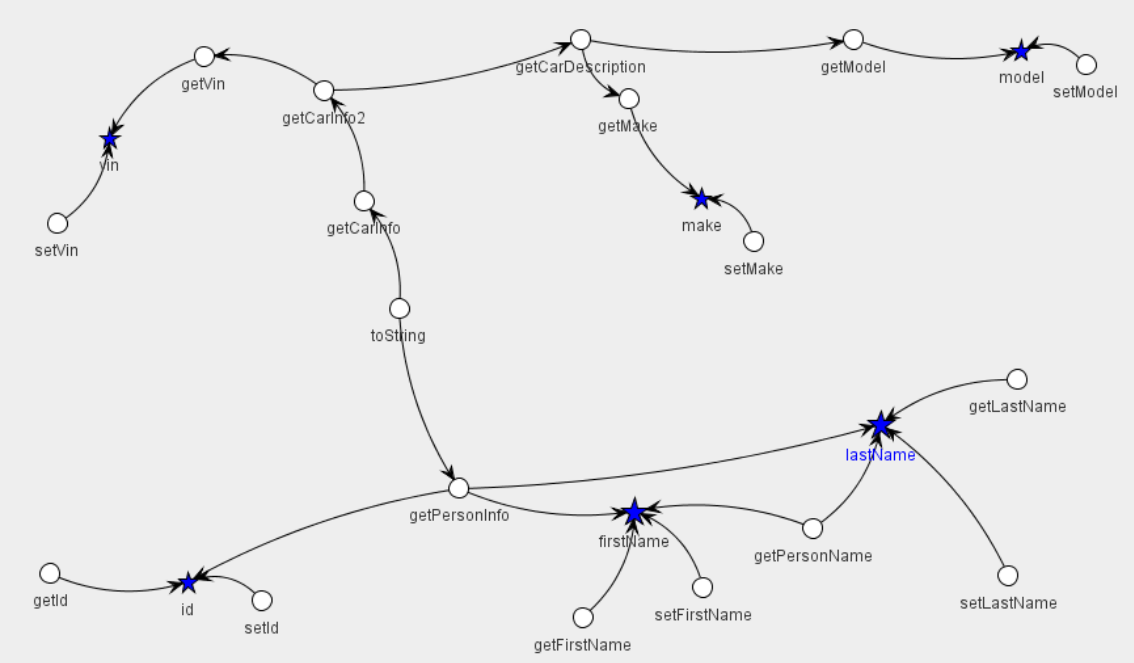

Figure 4: PersonCarAsym

2.4.2 Purpose

This tests the algorithm's ability to handle classes with a mixture of longer "stringy" call chains and more compact ones.

2.4.3 Output

PersonCarAsym 1 (Person)

firstName, lastName, id, getFirstName, setFirstName, getLastName, setLastName, getId, setId, getPersonInfo, getPersonName, toString.

PersonCarAsym 2 (Car)

make, model, vin, getMake, setMake, getModel, setModel, getVin, setVin, getCarInfo, getCarInfo2, getCarDescription, toString.

[5] http://code.google.com/p/cohesion-tests/source/browse/trunk/src/nz/ac/vuw/ecs/kcassell/personcars/PersonCarAsym.java

2.5 PersonCarSpecial

2.5.1 Input

PersonCarSpecial.java[6] is similar to PersonCarDirect.java, however, several methods inherited from `Object` are implemented - `toString, hashCode, equals`. These each access every attribute, as does the constructor. In addition, PersonCarSpecial has a `logger` attribute, which is heavily used by the methods.

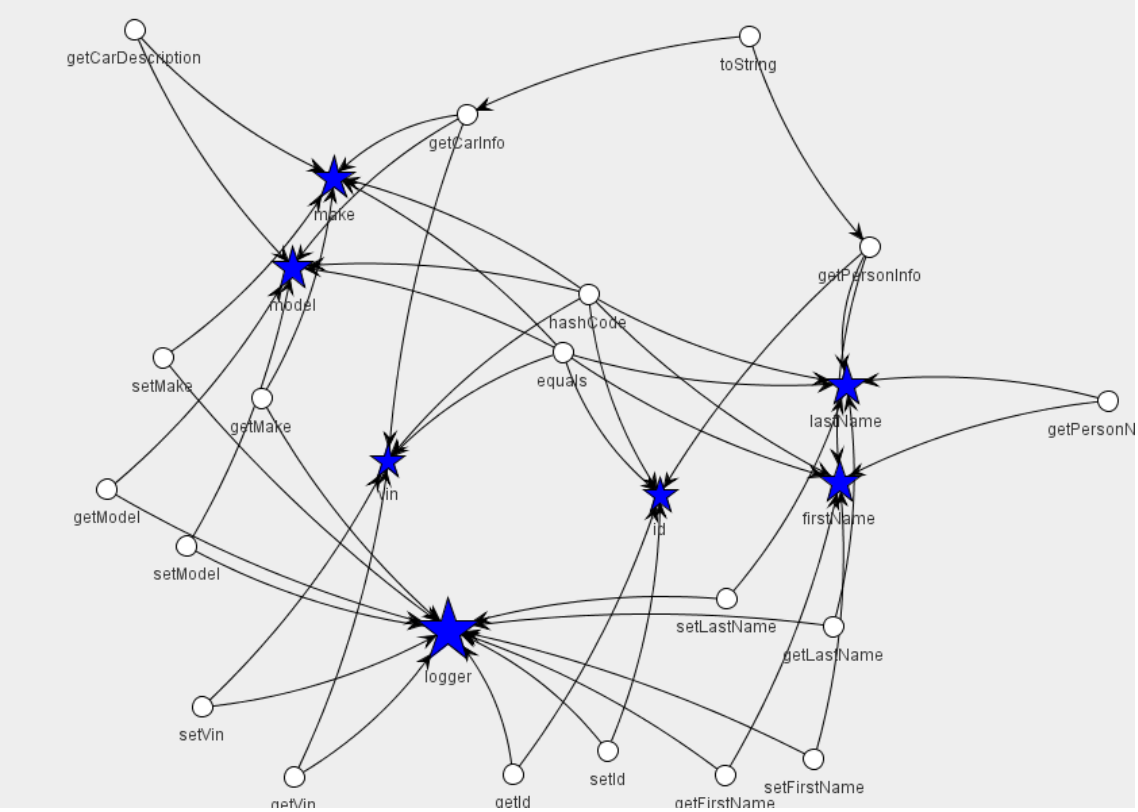

Figure 5: PersonCarSpecial

2.5.2 Purpose

This class checks the algorithm's ability to split classes in the presence of "non-business logic" methods (like `toString, equals`), and in the presence of cross-cutting concerns (the logger). Each new class should have their own versions of the "non-business logic" methods and logger.

2.5.3 Output

PersonCarSpecial 1 (Person)

firstName, lastName, id, getFirstName, setFirstName, getLastName, setLastName, getId, setId, getPersonInfo, getPersonName, hashCode, equals, toString, logger.

[6] http://code.google.com/p/cohesion-tests/source/browse/trunk/src/nz/ac/vuw/ecs/kcassell/personcars/PersonCarSpecial.java

PersonCarSpecial 2 (Car)

make, model, vin, getMake, setMake, getModel, setModel, getVin, setVin, getCarInfo, getCarDescription, hashCode, equals, toString, logger.

2.6 RectangleD

2.6.1 Input

Every method of RectangleD.java[7] directly accesses every attribute. There are no *get* or *set* accessors.

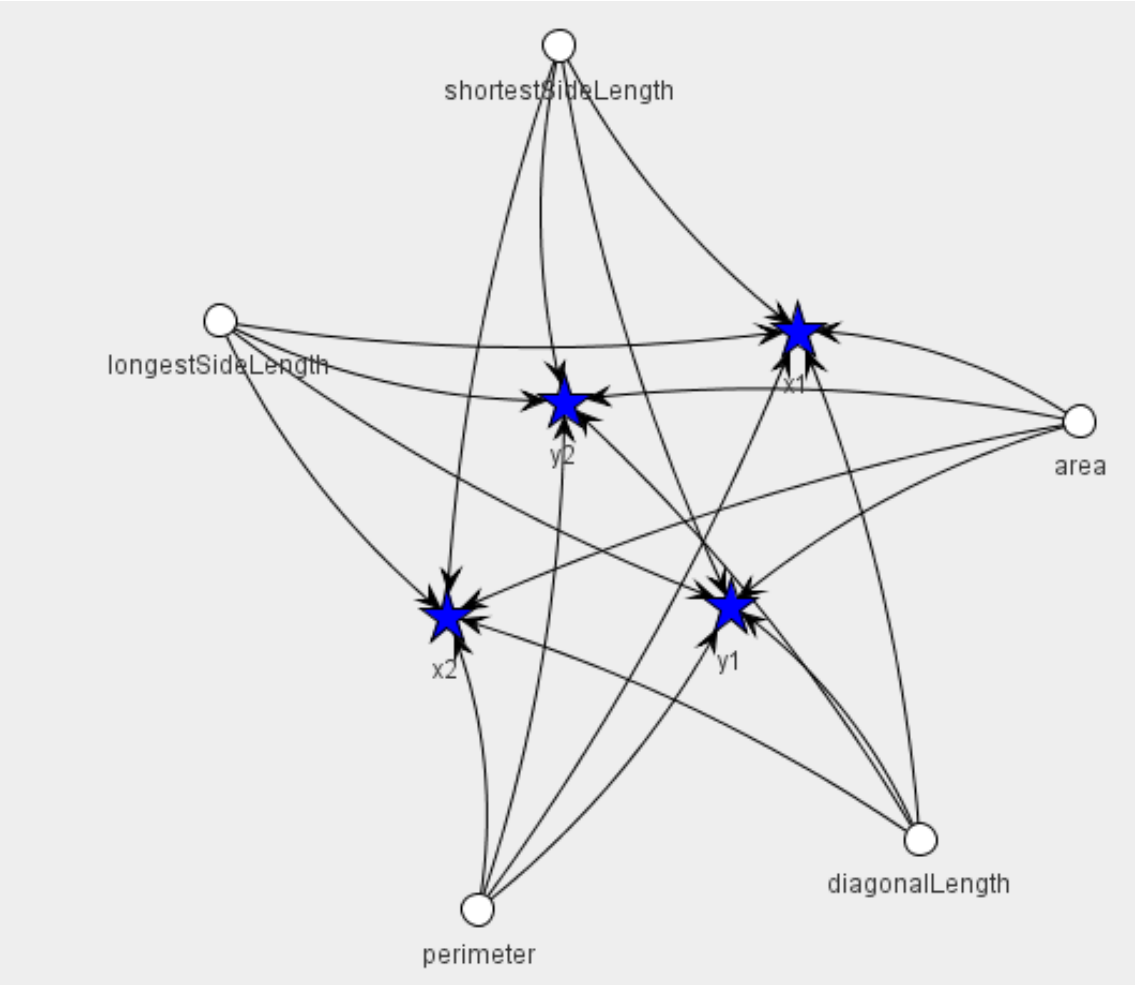

Figure 6: RectangleD

2.6.2 Purpose

This class is would be considered perfectly cohesive by several cohesion metrics in that every method accesses every attribute. Consequently, it should not be split (unless the algorithm has a large amount of semantic information available to it, sufficient to incorporate X and Y values into a `Point` class).

2.6.3 Output

No new class should be proposed.

[7] http://code.google.com/p/cohesion-tests/source/browse/trunk/src/nz/ac/vuw/ecs/kcassell/geometry/RectangleD.java

2.7 RectangleI

2.7.1 Input

`RectangleI`[8] is like `RectangleD` only using accessors. Every non-accessor method of RectangleI indirectly accesses every attribute via *set* accessors.

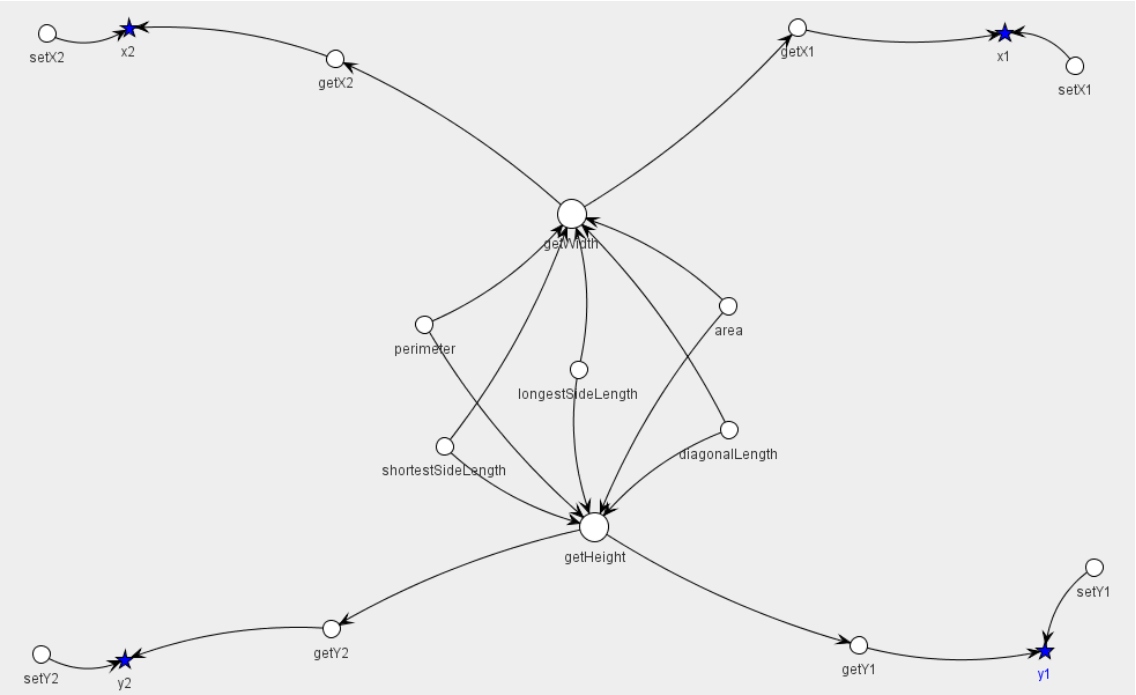

Figure 7: RectangleI

2.7.2 Purpose

This class tests the algorithm's ability to deal with stringy chains introduced by accessors.

2.7.3 Output

No new class should be proposed.

[8] http://code.google.com/p/cohesion-tests/source/browse/trunk/src/nz/ac/vuw/ecs/kcassell/geometry/RectangleI.java

2.8 PointShape

2.8.1 Input

PointShape.java[9] is a class whose methods are all required by an interface. Several of these methods have no logic and do not access any attributes.

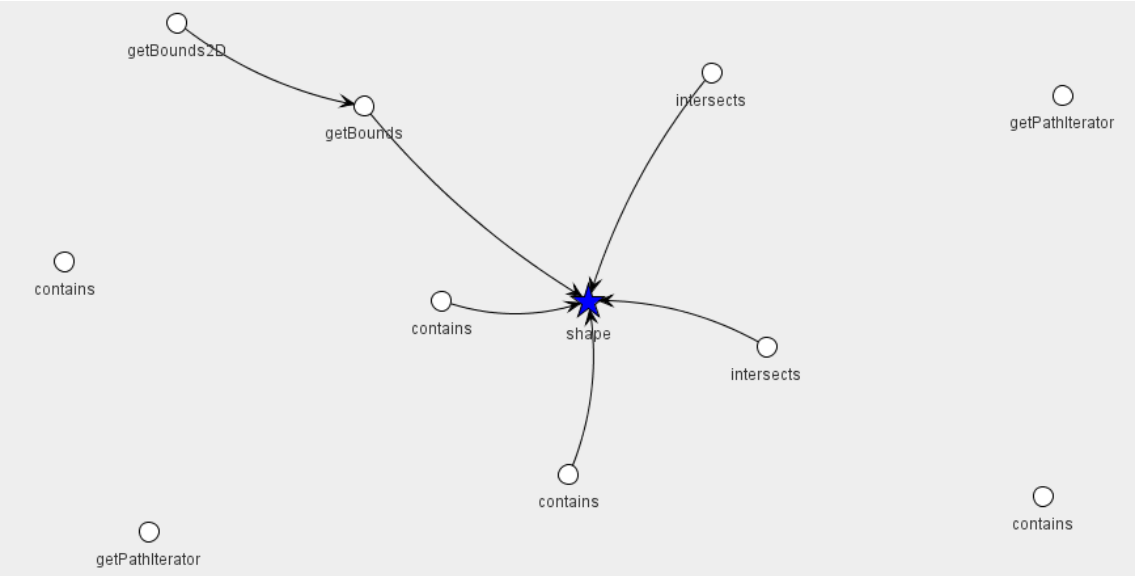

Figure 8: PointShape

2.8.2 Purpose

This tests the algorithm's awareness of interfaces that should not be split.

2.8.3 Output

No new class should be proposed.

[9] http://code.google.com/p/cohesion-tests/source/browse/trunk/src/nz/ac/vuw/ecs/kcassell/geometry/PointShape.java

2.9 AnonymousPersistence

2.9.1 Input

AnonymousPersistence.java[10] has no attributes. It has two disjoint sets of methods. One set saves and restores serializable objects to a file. The other set saves and retrieves infromation from a database.

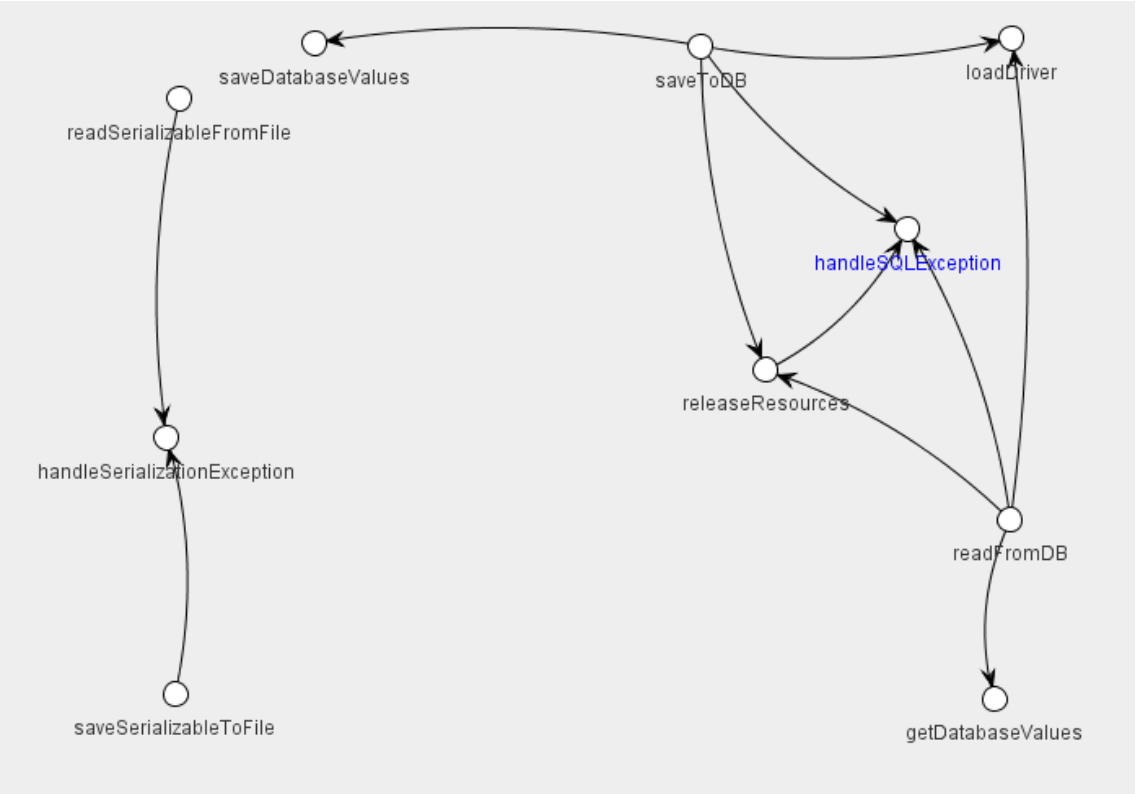

Figure 9: AnonymousPersistence

2.9.2 Purpose

This tests the algorithm's ability to split classes in the complete absence of attributes.

2.9.3 Output

AnonymousPersistence 1 (File)

saveSerializableToFile, readSerializableFromFile, handleSerializationException

AnonymousPersistence 2 (Database)

saveToDB, loadDriver, getDatabaseValues, saveDatabaseValues, handleSQLException, releaseResources

[10] http://code.google.com/p/cohesion-tests/source/browse/trunk/src/nz/ac/vuw/ecs/kcassell/anonymous/AnonymousPersistence.java

References

[1] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto. A Two-Step technique for extract class refactoring. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, Antwerp, Belgium, Sept. 2010.

[2] K. Cassell. refactoring : Message: Suites for testing automated refactoring tools. http://tech.groups.yahoo.com/group/refactoring/message/10396, May 2010.

[3] K. Cassell, P. Andreae, L. Groves, and J. Noble. Towards automating Class-Splitting using betweenness clustering. In *24th IEEE/ACM International Conference on Automated Software Engineering*, pages 595–599, Auckland, New Zealand, Nov. 2009.

[4] H. S. Chae, Y. R. Kwon, and D.-H. Bae. A cohesion measure for object-oriented classes. *Software Practice and Experience*, 30(12):1405–1431, 2000.

[5] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476493, 1994.

[6] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object Oriented Reengineering Patterns*. Morgan Kaufmann, 1st edition, July 2002.

[7] B. Foote and J. Yoder. Big ball of mud. Monticello, Illinois, 1997.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley, Boston, 1999.

[9] B. Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1996.

[10] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice Hall object-oriented series. PTR Prentice Hall, Englewood Cliffs, NJ, 1994.

[11] A. D. Lucia, R. Oliveto, and L. Vorraro. Using structural and semantic metrics to improve class cohesion. In *IEEE International Conference on Software Maintenance, 2008.*, pages 27 – 36, Beijing, Sept. 2008.

[12] G. Serban and I. Czibula. Object-Oriented software systems restructuring through clustering. In *Artificial Intelligence and Soft Computing - ICAISC 2008*, pages 693–704. Springer-Verlag, Berlin / Heidelberg, 2008.