Side-effect disparity as a metric for identifying behavioral subtyping violations

[COM S 610 — Fall 2013 — Final project report]

Loránd Szakács
Iowa State University
lorand@iastate.edu

ABSTRACT

When dealing with class hierarchies in object oriented languages programmers are faced with the problem of verifying that subtypes respect the contracts of their supertypes. Formal specification solves this, but it suffers from the caveat that specification has to be written in addition to code. This paper proposes an alternative, light-weight heuristic for detecting behavioral subtyping violations by looking solely at the side-effects of the code. We have evaluated this heuristic against a very large corpus of code and we have concluded that it has the potential of being applicable in practice.

1. INTRODUCTION

Behavioral subtyping (*a.k.a. Liskov Substitution Principle*) is an object oriented heuristic that aims to make a distinction between extension and subtyping, whereas extension means that the subclass simply inherits implementation and instantiations of it are assignable to references of its superclass' with no guarantee that at runtime it will behave as expected. While subtyping implies guarantees that when instantiations of the subclass are used, they behave as expected allowing for programmers to reason in terms of what is called supertype abstraction [12]. An essentially equivalent definition of behavioral subtyping is given by [2, 13, 14, 18]: subtypes are allowed to weaken the pre-conditions and strengthen the post-conditions imposed by their supertypes.

The *Liskov Substitution Principle* is regarded as a corner stone guideline in object oriented design, being part of the so called SOLID principles [15, 16]. This idea has been part of the zeitgeist of industry programmers since at least 1988 [18] and has been brought forward as a good guideline by famous speakers like Robert C. Martin and Bertrand Meyer. Therefore the importance of this practice cannot be ignored and the need to create automated support to help programmers follow this practice is also a good idea.

The research community has described behavioral subtyping in terms of formal specification [2, 13, 14]. Which has its disadvantages, chiefly as the fact that formal specification has to be written separately from, and in a different language than the source code. Therefore testing for behavioral subtyping violations in terms of specification [3, 10, 8, 6] might not be an option at all.

We offer an alternative way of testing whether or not behavioral subtyping is violated by defining a heuristic in terms of side-effects, more precisely, the disparity between the side-effects of a method in a supertype and the side-effects of its overridden version in any of its subtypes. We test this heuristic using the Boa language [4, 5] by collecting data from 5113 open source Java projects. We use these initial empirical results to suggest improvements on the heuristic presented below. Furthermore, the collected data serves as a baseline for comparison of future results obtained with an improved heuristic.

1.1 Heuristic

Looking at the side effects of a method might not be enough to infer its behavior, but given the fact that we want to look at the difference in behavior we think that looking at the difference in the side-effects of methods is a good starting point. This heuristic has the advantage of being simple, both in terms of the complexity of its definition and in terms of relative ease with which it can be automatically applied to code. Essentially, this heuristic allows the inference of and ad-hoc specification for a method purely in terms of its side-effects. Disadvantages are discussed in detail in section 5.

1.2 Related work

There exist numerous tools and approaches that rely on formal specification to test for behavioral subtyping violations, or as it is referred to in literature behavioral contract violations [3, 7, 8, 9, 20]. These approaches have the obvious limitation that a code specification *has* to be written. Our approach, on the other hand, relies solely on the source code, therefore making it universally applicable. However, unlike the above mentioned work, our approach cannot be applied to verifying behavioral contract violations by implementors of interfaces and abstract methods. Neither does it concern itself with the actual values of object state, which formal specification tools do.

To the author's knowledge no work exists that uses side-effect disparity between supertype-subtype method effects to test for behavioral subtype violations.

2. APPROACH

This study seeks to answer two research questions: *(RQ1) is side-effect disparity a valid heuristic for identifying behavioral subtyping violations?* And, a secondary, less substantiated research question whose answer depends on the answer of the first research question: *(RQ2) to what extent does behavioral subtyping happen in practice?*

To address these questions we took the following high level approach:

- define a heuristic for identifying behavioral subtyping violations
- use Boa to apply this heuristic to a large number of real world projects (this answers *RQ2*)
- manually inspect several violations to determine whether or not the identification was accurate (this answers *RQ1*)
- based on results, suggest improvements

2.1 Definition of side-effects

For the purposes of this study we looked at a restricted definition of side-effects. We included only:

- assignments to the fields of a class, includes inherited fields
- method calls on fields of the class
- calls to methods from the class in question

For instance, consider the following contrived example:

```
void foo(P param) {
  this.myField = 42;
  super.superField = 24;
  int localVar = this.f1;
  param.field = 42;
  localBar();
  myField.otherBar();
}
```

The effects of this method would be: *myField, superField, localBar() and otherBar()*. Methods *localBar()* and *otherBar()* are not substituted for their effects.

2.2 Side-effect disparity

Side-effect disparity as a heuristic for determining behavioral subtyping violations is motivated by two arguments:

1. from Leavens et al [10, 11] we can see that in the JML specification language the post conditions of a method are the side-effects of a method. Therefore, if we compute the side effects of a method in a super-class, we can use its side-effects as an inferred post-condition of the superclass method. Therefore, by comparing the side-effects of two methods we are implicitly comparing their post-conditions. And disparities in the post-conditions can imply violations in behavioral subtyping.
2. side-effects are also a reflection of the behavior of a program, although in of themselves they cannot be used to infer any behavior. However, the difference in the side-effects of two methods implies a difference in behavior. The extent to which this difference represents a behavioral subtyping violation is the approximation in this heuristic.

Given that we use this heuristic to determine behavioral subtyping violations, only the effects of overridden and overriding methods need to be computed.

2.3 Comparison

In this sub-section we will present a high level overview of the employed analysis. The Boa implementation and its limitations are further explored in section 3.

The following algorithm operates on an entire project. The results stored in the *violations* collection are aggregated for all projects.

```
Input: Project
PBS ← gatherAllSubtype-SuperTypePairs
for all classPair in PBS do
    OM ← gatherOverridenMethods(classPair)
    for all (subMethod, superMethod) in OM do
        subEff ← computeSetOfEffects(subMethod)
        superEff ← computeSetOfEffects(superMethod)
        if subEff - superEff != 0 then
            violations ← (Project, classPair,
            subMethod, superMethod)
        end if
    end for
end for
```

Very important to note is that when gathering all subtype-supertype pairs we do not account for transitive inheritance (class A extends B which extends C; so pair A, C is never returned). As [2, 25] point out, interface inheritance and implementation inheritance are theoretically two different mechanisms, but in Java they are treated the same therefore the programmer might not intend to subtype the entire family of supertypes, only the explicitly named one.

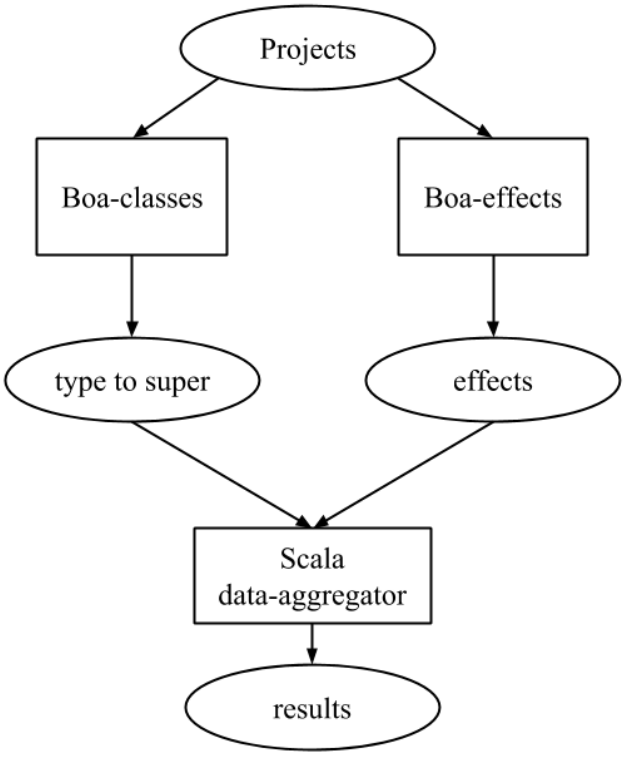
Another important remark is the fact that we cannot gather subtype-supertype pairs where the supertype is imported from a library or from a project. Thus we do not test *all* subtyping relationships that exists within a project.

The effects of a method are computed as a set of single effects that fit the description given in sub-section 2.1.

3. IMPLEMENTATION

The source code of our analysis is available at [22]. It contains all the programs necessary to re-run this analysis as well as a readme file containing all the relevant information to do so. The repository will not be updated after the date of December 13th 2013 so that it reflects the analysis as it was at the time of writing of this report.

The following image describes at a high level the components involved in the implementation of the above pseudo code algorithm:



Due to an apparent bug in the runtime of the Boa framework the program had to be divided as above. Otherwise, every computation would have been done in Boa. The program *Boa-classes* outputs a list of all available classes and a list of pairs of classes in a subtype-supertype relationship grouped per project. The program *Boa-effects* program uses the *Boa-classes* as a subroutine to determine all pairs of subtype-supertype and then outputs a list of all the effects of the methods of these classes grouped per method, per class, per project. This data is then fed to the Scala [19] program *Scala data-aggregator* which determines the set of pairs of overriden methods, compares their effects and outputs the results. The output is in the form of high level statistics (number of subtype-supertype pairs, etc.) and a file containing a list of *all* behavioral subtyping violations together with the information needed to locate them in the codebase (project id, fully qualified class name, method name).

3.1 Advantages of Boa

Boa had three major advantages that greatly work in our favor:

- easy and convenient access to a very large code base. Cummulatively it took approximately 20 minutes to run the Boa analyses on the 5133 projects, giving us more time to focus on refining the theoretical aspects of this paper.
- convenient and compact syntax for defining program analysis at an AST node level. This allowed for a relatively simple implementation of side-effect computation by visiting each AST node within a method and retaining only the relevant ones mentioned in section [**?**].
- reproducibility. The source code of the analyses used to gather the results discussed here can be accessed via [23] and [24], the webpages make the output of the analyses downloadable. Then, the *Scala data-aggregator* program has to be run on the downloaded output to produce the results.

3.2 Limitations of Boa

Boa doesn't offer semantic static analysis data (e.g. fully resolved type information), consequently our analyses had to implement a few features using approximations, and we had to push back others to the future work section (substituting methods with their computed effects). An example sub-routine that had to be written because of the lack of type information is identifying the fully qualified name of the supertype after an *extends* clause. The only information we had about the extended class was its name in string form. So we had to build a fully qualified name based on the following cases and check to see if it is available in a previously computed set containing the fully qualified names of all classes in the project: the actual extended class can be referred to by its fully qualified name, it can be explicitely imported, it can be imported with a wildcard, or it can exist in the same package as its subtype. Determining the fully qualified names of denoted supertypes is sound and complete with respect to the set of classes within the project.

4. EVALUATION

The Boa framework runs on cached snapshots of the major repository hosting services. When running a Boa program the user is presented with an option to choose on which snapshot to run the analysis. We ran the analysis on the *September 2013 (medium)* snapshot.

4.1 Results

Running our analysis on the above snapshot yielded the following results:

RES1. *1286* of *5133* projects contained at least one class that used implementation inheritance (override a method that already had an implementation). Showing that approximately 25% of projects use implementation inheritance. Conversely, this shows that 75% of projects do not use implementation inheritance at all and are, therefore, implicitly using behavioral subtyping. Unfortunately, testing whether or not behavioral subtyping is violated in the case of these 75% can be done only with the formal specification approach described in the related work section.

RES2. *739* of these *1286* eligible projects contained at least one subtyping violation.

RES3. *11600* of *101433* pairs of subtype-supertype contained at least one violation.

RES4. *283216* of *2701596* overridden methods were violating behavioral subtyping.

To answer *RQ1* we have chosen, at random, 4 projects that contained at least one violation, randomly selected 5 violations from each, making for a total of 20 violations; the we tried to determine whether or not they truly constitute behavioral subtyping violations.

A complete list of the 20 violations, together with each individual verdict is given in section 4.3. In summary, it was determined that *9* out of the *20* violations were false positives. Giving our analysis a precision of 55%. Given a *correct* specification, the tools mentioned in the related work section have a precision of 100%. But, also considering (i) the low prevalence of specification, (ii) the room for error in writing the specification, (iii) the fact that our heuristic can be easily applied to any implementation inheritance hierarchy, (iv) the suggestions for improving this heuristic discussed in section 6; we think that this heuristic can have the potential to be a real alternative to approaches that use formal specification.

Given this precision we can now give an answer to *RQ2*: considering results *RES3 and RES4* we can deduce that 6.2% of subtype-supertype pairs violate behavioral subtyping, and 5.7% of overridden methods do so as well. Keeping in mind the threats to the validity of this study discussed in section 5, and the fact that there is no previous work that tries to answer *RQ2*, we advise the reader to consider these numbers with a healthy dose of skepticism.

4.2 A closer look at the evaluation

The following example is lifted from violation 3a:

```
//supertype: net.sf.saxon.style.StyleElement
public void validate() throws XPathException {}
//subtype: net.sf.saxon.style.XSLIf
public void validate() throws XPathException {
        checkWithinTemplate();
        test = typeCheck("test", test);
}
```

The above code represents a classic Template design pattern [26]. It appeared in similar forms in the following violations: *1d, 3b, 4b, 4c, 4e*. All of these were counted as false positives because we assumed that given the fact that this is a well known pattern, the programmers were aware of the context in which instantiations of either class would behave as expected.

We noticed another peculiarity in the false positive *3c*, the sets of effects were completely disjoint. Essentially, they were partitioned between using fields of the superclass and fields of the subclass.

4.3 Data

This section presents all the method pairs that were manually inspected. The format of the data is as follows:

1. **Project Name** Project URL

   (a) fully qualified name of the subclass
   fully qualified name of the superclass
   methodName
   confirmed violation or false positive.

1. **Akrogen Plugin** http://sourceforge.net/projects/akrogen
   all package names are normally prefixed with org.akrogen

   (a) xuil.gui.swing.widgets.groups.SwingGuiRadioImpl
   xuil.gui.swing.widgets.SwingGuiWidgetImpl
   buildWidget
   false positive

   (b) core.codegen.template.xslt.XSLTTemplateEngineImpl
   core.codegen.template.AbstractTemplateEngine
   merge
   confirmed

   (c) core.codegen.velocity.VelocityTemplateEngineImpl
   core.codegen.AbstractTemplateEngine
   setConfiguration
   false positive due to bug in implementation.

   (d) xuil.core.internal.dom.XuilWidgetElementImpl
   xuil.core.internal.dom.XuilElementImpl
   registerAttrDescriptor
   false positive

   (e) xuil.gui.swt.widgets.trees.viewers.SwtTreeViewer
   xuil.gui.swt.ex.viewers.TreeCellsViewer
   doUpdateItem
   confirmed

2. **CVSGrab** http://sourceforge.net/projects/cvsgrab
   all package names are normally prefixed with org.akrogen

   (a) cvsgrab.web.Chora2_0Interface
   cvsgrab.web.ViewCvsInterface
   detect
   confirmed

   (b) cvsgrab.web.Chora2_0Interface
   cvsgrab.web.ViewCvsInterface
   guessWebProperties
   confirmed

   (c) cvsgrab.web.FishEye1_0Interface
   cvsgrab.web.ViewCvsInterface
   detect
   confirmed

   (d) cvsgrab.web.CvsWeb2_0Interface
   cvsgrab.web.ViewCvsInterface
   detect
   confirmed

   (e) cvsgrab.web.ViewCvs0_9Interface
   cvsgrab.web.ViewCvsInterface
   getProjectRoot
   confirmed

3. **Saxon XSLT and XQuery Processor**
   http://sourceforge.net/projects/saxon
   all package names are normally prefixed with net.sf.saxon

   (a) style.XSLIf
   style.StyleElement
   validate
   false positive

   (b) style.XSLSequence
   style.StyleElement
   validate
   false positive

(c) functions.Collection
functions.SystemFunction
checkArguments
false positive

(d) expr.BooleanExpression
expr.BinaryExpression
optimize
confirmed

(e) expr.BooleanExpression
expr.BinaryExpression
typeCheck
confirmed

4. **Xendra** http://sourceforge.net/projects/xendra

(a) org.compiere.process.InventoryCountCreate
org.compiere.process.SvrProcess
doIt
confirmed

(b) org.rvpf.simple.SimpleThread
org.rvpf.service.ServiceThread
setUp
false positive

(c) org.rvpf.mbean.ServiceBase
org.rvpf.mbean.StatsHolder
setObjectName
false positive

(d) org.xendra.process.update.workflowtoxml
org.compiere.process.SvrProcess
doIt
confirmed

(e) org.rvpf.store.server.sink.ScriptSink
org.rvpf.store.server.sink.AbstractSinkModule
open
false positive

5. THREATS TO VALIDITY

The most severe threat to validity is the lack of an oracle that can determine whether or not our analysis yielded false positives. The lack of an oracle is a consequence of both to the lack of community benchmarks for behavioral subtyping and of the lack of a formal specification of the methods used in the sample. Therefore it was left to the author's judgment to determine whether or not the analysis presented here was accurate.

Although it was not evident in the random sample, but the narrow definition of side-effects prevents the computation of the effects from being a complete analysis (it does not find all the effects of a method), thus potentially creating false positives when an effect was detected in a method, but was missed in its counterpart. A concrete example would be:

```
//supertype
void foo() {
    this.f = 42;
}

//subtype
@Override
void foo() {
    this.bar();
}

void bar() {
    super.f = 42;
}
```

Ultimately both methods have the same effect, but because we do not substitute methods with their computed effects this creates false positives. Additionally, there are the inherent approximations we make when computing the effects that are due to Boa's static analysis limitations, as described in section 3.

In formal specification languages [10, 11] the values associated with effects are also considered, this is done so for good reason. For example, consider a random number generator supertype that generates only odd numbers, any subtype that also generates even numbers will behave differently even though they might have the same side-effects. We cannot account for these kind of differences with out heuristic.

We impose no ordering on the effects when computing them. This can pose a problem when it is essential that an effect f has to happen before another effect g for the methods to have the same behavior. So in this case we also miss actual behavioral subtyping violations.

And last, given all the above observation it follows that our analysis is neither sound nor complete. Taking into account the fact that we cannot give a bound on the extent of the incompleteness and unsoundness (more than 20 samples and a better oracle are required to test the precision of the analysis) it might render this heuristic unusable in practice.

6. FUTURE WORK

First order of business is to improve the evaluation of the heuristic's precision. One way of doing this is finding software that has formal specification from which behavioral subtyping violations can be derived using tools described in the related work section, and then using these results as an oracle to test the soundness of our heuristic. There is one foreseeable problem with this improvement, mainly, the fact that software that uses formal specification might not have *any* behavioral subtyping violations.

Second, we can make great improvements to the soundness of the analysis by:
Broadening our list of measured side-effects to include the side-effects on the parameters of a method. This will ensure that our inferred post-conditions (side-effects) will contain information about the state of the parameters of a method as it is usually done with formal specification [11].

Substituting methods for their computed effects. This will still leave room for unsoundness because we cannot determine the exact target of a call site due to polymorphism. But we can determine with great confidence how often polymorphic calls tend to happen with a rather simple Boa analysis. By doing this latter analysis we can also put a bound on the imprecision of our analysis.

Improving the heuristic so that it accounts for the cases de-

scribed in section 4.2. Adjusting the heuristic for the case of the disjoint (in terms of ownership) sets of effects is a sensible, easy to implement modification that applies generally. However accounting for the template method pattern might pose a few difficulties. For one, do we hard-code the particular cases we encountered? That seems foolish. What about other design patterns? Do we try and find these as well, and how to we account for all of these with an elegant generally applicable mechanism? These are the apparent issues facing this line of research, and no clear solution is yet visible.

To address the issue of ordering of effects, commutativity analysis [21, 1] will have to be employed. This will allows us to determine if two effects produce the same observable result regardless of their ordering. The main advantage of commutativity analysis over equivalence analysis [17] is that it is more scalable, which, given the large amount of our data is a must.

With every iterative improvement of the heuristic we can reevaluate *RQ1* and *RQ2* and see if there are any measurable differences. We hope that the foundation of this heuristic will become sound enough so that we can confidently ascertain whether or not behavioral subtyping happens in practice.

7. CONCLUSION

We have defined and evaluated a novel heuristic for identifying behavioral subtyping violations for inheritance in object oriented languages. We have used the Boa programming language and framework to evaluated this heuristic against a large corpus of projects. This has the advantage that the results are easily verifiable and replication friendly by third parties.

During our evaluation we have determined that our heuristic offers 55% precision on a random sample of 20 violations. Furthermore, based on the data gathered we have determined several ways of improving this heuristic. Keeping this precision rate in mind, we have also determined that 6.2% of subtype-supertype pairs violate behavioral subtyping and 5.7% of overridden methods do so as well. But due to several caveats in our approach we were unable to confidently ascertain the effectiveness of our heuristic. But, described in the previous section, are the reasons why we remain hopeful that this heuristic can be improved to the point where it can be reliably used in practice.

8. REFERENCES

[1] F. Aleen and N. Clark. Commutativity analysis for software parallelization: letting program transformations see the big picture. In *ACM Sigplan Notices*, volume 44, pages 241–252. ACM, 2009.

[2] P. America. Designing an object-oriented programming language with behavioural subtyping. In *Foundations of Object-Oriented Languages*, pages 60–90. Springer, 1991.

[3] A. Duncan and U. Hölzle. Adding contracts to java with handshake. *University of California at Santa Barbara, Santa Barbara, CA*, 1998.

[4] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 422–431. IEEE Press, 2013.

[5] R. Dyer, H. Rajan, and T. N. Nguyen. Declarative visitors to ease fine-grained source code mining with full history on billions of ast nodes. In *Proceedings of the 2013 International Conference on Generative Programming: Concepts and Experiences*, 2013.

[6] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. *ACM SIGSOFT Software Engineering Notes*, 26(5):229–236, 2001.

[7] B. Gomes, D. Stoutamire, B. Vaysman, H. Klawitter, and J. Feldman. A language manual for sather 1.1, 1996.

[8] M. Karaorman, U. Hölzle, and J. Bruno. jcontractor: A reflective java library to support design by contract. In *Meta-Level Architectures and Reflection*, pages 175–196. Springer, 1999.

[9] R. Kramer. icontract-the java design by contract tool. In *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, pages 295–307. IEEE, 1998.

[10] G. T. Leavens. JmlâĂŹs rich, inherited specifications for behavioral subtypes. In *Formal Methods and Software Engineering*, pages 2–34. Springer, 2006.

[11] G. T. Leavens and Y. Cheon. Design by contract with jml. *Draft, available from jmlspecs. org*, 2006.

[12] G. T. Leavens and W. E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, 1995.

[13] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[14] B. H. Liskov and J. M. Wing. Behavioral subtyping using invariants and constraints. Technical report, DTIC Document, 1999.

[15] R. C. Martin. Design principles and design patterns. *Object Mentor*, pages 1–34, 2000.

[16] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[17] V. Menon, K. Pingali, and N. Mateev. Fractal symbolic analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(6):776–813, 2003.

[18] B. Meyer. *Object-oriented software construction*, volume 2. Prentice hall New York, 1988.

[19] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. Technical report, Citeseer, 2004.

[20] R. Plosch and J. Pichler. Contracts: From analysis to c++ implementation. In *Technology of Object-Oriented Languages and Systems, 1999. TOOLS 30. Proceedings*, pages 248–257. IEEE, 1999.

[21] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):942–991,

1997.

[22] L. Szakacs. *610 Project Blackfyre*, 2013 (accessed Dec 13, 2013). `https://github.com/lorandszakacs/fall2013-610-project-blackfyre`.

[23] L. Szakacs. *Boa-classes analysis*, 2013 (accessed Dec 13, 2013). `http://boa.cs.iastate.edu/boa/?q=boa/job/public/2750`.

[24] L. Szakacs. *Boa-effects analysis*, 2013 (accessed Dec 13, 2013). `http://boa.cs.iastate.edu/boa/?q=boa/job/public/2751`.

[25] C. Szyperski. *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[26] J. Vlissides, R. Helm, R. Johnson, and E. Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49:120, 1995.