Towards the Detection of Hidden Familial Type Correlations in Java Code

Alin-Petru Roşu
LOOSE Research Group
Politehnica University of Timişoara
Timişoara, Romania
alin.rosu@student.upt.ro

Petru-Florin Mihancea
LOOSE Research Group
Politehnica University of Timişoara
Timişoara, Romania
petru.mihancea@upt.ro

***Abstract*—Family polymorphism is an object-oriented programming feature which facilitates the definition of groups of classes (families) that are allowed to be used together while statically forbidding them to be mixed with classes outside their families. Unfortunately, this feature has not been yet adopted by mainstream industrial-strength programming languages. Consequently, in Java, the idea of non-mixable families is prone to be implemented in a statically unsafe fashion, affecting the programs' intelligibility. In order to support program comprehension, we present an approach to detect code fragments where types of references are correlated within a family; nonetheless, these correlations are hidden behind the references' declarations. We obtained promising results during the initial design and evaluation iterations, based on the analysis of a software system where the presence of families was previously reported.**

***Index Terms*—program comprehension, family polymorphism, metrics, static analysis**

I. Introduction

Polymorphism is a motif of expressiveness, reusability and flexibility in the object-oriented programming languages. Generally, the mainstream ones (such as Java) support three kinds of polymorphism i) *ad-hoc polymorphism* i.e., method overloading ii) *parametric polymorphism* i.e., generic programming and iii) *subtype polymorphism* i.e., type inheritance. However, there is a less known kind of polymorphism, whose associated language mechanisms have been proved to be necessary in practice - family polymorphism.

Family polymorphism has been initially detailed in [1]. Its importance can be more easily understood using an example. Consider Figure 1, which presents two class hierarchies, *Wine* and *Glass*, and a common client, *WaiterTray*. By analysing the *WaiterTray*'s fields, one might be tempted to believe that their referred objects are instances of any concrete types of the *Glass* and *Wine* hierarchies, grouped in any arbitrary combination. However, from the *WaiterTray*'s clients it can be implied that the programmer's intention is different. In fact, the true intention is to always *correlate* the instances of the two hierarchies based on their type. Concretely, a *RedWine* instance can only be used together with a *RedWineGlass* instance, while a *WhiteWine* instance must only be used together with a *WhiteWhineGlass* instance. Conclusively, this example consists of two *families*: {*RedWine, RedWineGlass*} and {*WhiteWine, WhiteWineGlass*}, whose members should not be intermixed.

Family polymorphism has been proposed so as to allow programmers to express constraints such as the ones presented above i.e., to statically guarantee that families cannot be accidentally mixed. Unfortunately, this feature has not been yet adopted by mainstream industrial-strength languages, forcing developers to create families based on the available mechanisms.

Consequently, two such possible patterns have been presented and compared in [1]. The first one uses parametric polymorphism, and ensures safety i.e., statically forbids families mixing. The second one disregards safety, being similar with the presented example: statically, nothing stops a developer to create a *WaiterTray* using a combination of *WhiteWine* and *RedWhineGlass* objects. This kind of unsafe handling appears to be used in practice, as it has been recently shown in [2]; downcast operations are usually involved as also discussed in [1]. Consequently, not only intermixing families may lead to unexpected system behaviour (e.g., ClassCastExceptions), but also mislead developers from understanding the code's meaning. The brief observation of some reference variables declared with a particular supertype (e.g., the *_wine* and *_glass* declared with *Wine* and *Glass* respectively) does not actually show the real design intent. In effect, the familial correlations between the concrete types intended to be referred by those variables (e.g., when *_wine* references a *RedWine* object, *_glass* must also reference a *RedWineGlass* object) will remain hidden beneath the subtype polymorphism mechanism.

Aiming to support program comprehension, we propose an approach to detect Java code fragments where familial type correlations exist between some references' types, yet are hidden by the way the variables are declared. This approach would make a developer aware of the references' true design intent, possibly triggering a refactoring activity to statically forbid families intermixing using parametric polymorphism.

The rest of the paper is structured as follows: Section II details the detecting approach; Section III discusses promising current results obtained while analysing an open-source software system where the presence of families was previously reported; Section IV presents the related work; lastly, Section V concludes the paper and draws future work directions.

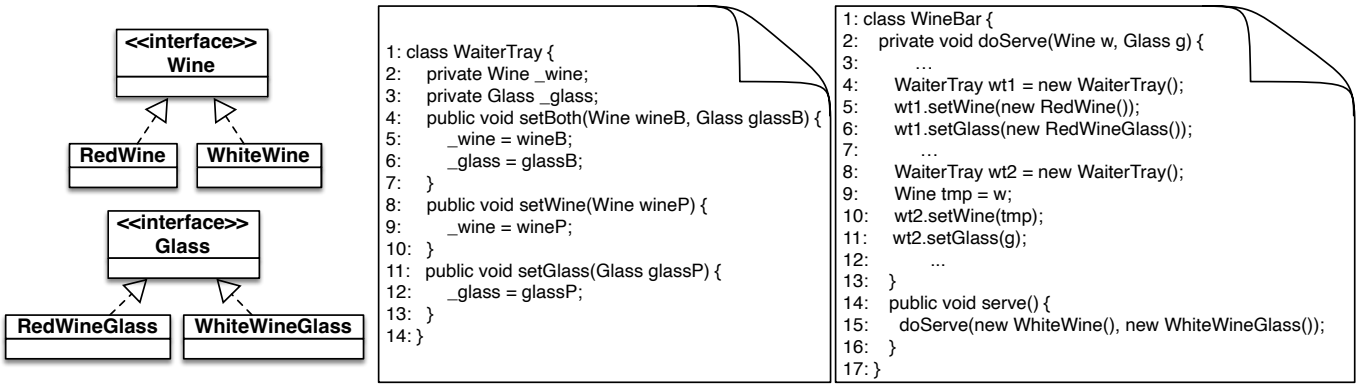

Fig. 1. Example of Class Families

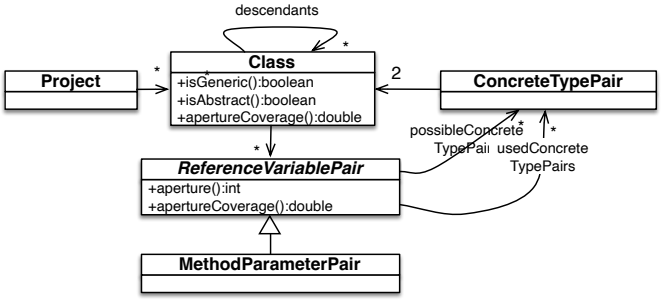

Fig. 2. The Conceptual Metamodel

II. Detecting Hidden Familial Type Correlations

We propose a metric-based approach, capable of identifying code fragments containing hidden familial type correlations. We start by describing the metamodel, which organises the information required to quantify the property of the problematic fragments. Afterwards, we present the proposed software metrics together with their interpretation models and implementation alternatives. The entire approach has been implemented as a publicly available Eclipse plugin[1].

A. The Metamodel

In our approach, we are initially interested in gathering information about the system under investigation and its classes. These are represented by the *Project* and *Class* elements of the metamodel in Figure 2.

Next, we take a closer look at reference variables (e.g., method parameters), trying to estimate the concrete types of the objects they may be referring at runtime. At first sight, it might appear sufficient to capture this information for each reference individually. For instance, in the example from Figure 1, observing the *WaiterTray*'s setters invocations indicate that the *wineP* parameter might refer both *WhiteWine* and *RedWine* objects. Similar is the case of the *glassP* parameter and *WhiteWineGlass*/*RedWineGlass* objects. However, this way disregards an important aspect: when *setWine* is invoked with a *RedWine* object, the *setGlass* is also invoked, on the same tray, with a *RedWineGlass* object. This means that we omit the correlation between the *wineP* and *glassP*'s types when setting the content of a tray. Consequently, we must focus on *pairs* of variables e.g., (*wineP*, *glassP*) and *pairs* of concrete types e.g., (*RedWine*, *RedWineGlass*). We ultimately decided to represent these elements as first-class entities i.e., *ReferenceVariablePair* and *ConcreteTypePair* in the metamodel.

For the moment, we decided to analyse only pairs of method parameters i.e., *MethodParameterPair* metamodel's entity. The reason is that fields are usually set through method calls. Therefore, a possible correlation between the concrete types of some fields would very likely induce a correlation between the concrete types of some parameters that the fields are being assigned with.

In terms of (current) limitations, we emphasise that: i) we do not analyse generic code, nor parameters of static methods ii) we do not analyse arrays, nor containers e.g., *Collection* objects iii) we pair only the parameters of methods declared within a class, disregarding the inherited ones iv) constructor parameters are paired only with other parameters of the same constructor.

Lastly, the remaining elements of our metamodel (i.e, the *possibleConcreteTypePairs* and *usedConcreteTypePairs* relations) play an essential role, and are discussed in detail in the following sections.

B. The Aperture Metric

For a pair of reference variables, we are interested in the number of distinct combinations of concrete types that can be referred solely based on the variables' declarations.

For this purpose, for each reference, we compute the set of all concrete types of the type hierarchy rooted by the declared type. For instance, for the *wineP* parameter, this set is {*RedWine*, *WhiteWine*}, computed based on the type hierarchy of *Wine*. Subsequently, at the variables pair level, we compute the cartesian product between the previously described sets, associated to each member. Thus, for the *(wineP, glassP)* pair, the result is {*(RedWine, RedWineGlass)*,

[1]https://github.com/SourceCodeCodex/jFamilyCounselor

(RedWine, WhiteWineGlass), *(WhiteWine, RedWineGlass)* and *(WhiteWine, WhiteWineGlass)*}. This computation represents the *possibleConcreteTypePairs* relation from the metamodel.

Finally, for a references pair, we name the cardinality of the described cartesian product as the *aperture* of that pair. This metric quantifies how large is the space of all concrete types combinations that could be referred by two particular variables (type combinations permitted by subtype polymorphism).

C. The Aperture Coverage Metric

Apart from all possible types pairs, represented by the *aperture*, it is more important to know how many of these pairs are *actually referred* by some references in discussion. For instance, for the *(wineP, glassP)* pair we can easily observe that only the {*(RedWine, RedWineGlass)*, *(WhiteWine, WhiteWineGlass)*} pairs are actually used, out of the entire possible combinations. This subset of the possible combinations corresponds to the *usedConcreteTypePairs* relation from the metamodel.

Based on the *possibleConcreteTypePairs* and the *usedConcreteTypePairs* we define the *aperture coverage* metric with Formula 1.

$$AC((x,y)) = \frac{|usedConcreteTypePairs((x,y))|}{|possibleConcreteTypePairs((x,y))|} \quad (1)$$

In our example, it is easy to see that the *aperture coverage* of the *(wineP, glassP)* pair is $2/4 = 0.5$. In general, a value in the $(0, 1)$ interval signifies that not all possible combinations of concrete types are actually used. Consequently, it might be the case that some type correlations are hidden behind the apparently general variable declarations. In fact, the lower the value the higher the likelihood that correlations do exist.

This metric is also extended at the class level. The aperture coverage of a class is the minimum of all aperture coverages of any references pair of the class in discussion. Similarly, the lower the value the higher the chances that the class might be a client of type families.

Determining the *usedConcreteTypePairs* set is, unfortunately, not easy. In order to define heuristics for its adequate approximation, we needed to observe some real cases of hidden familial type correlations. Thus, we started by implementing a rough approximation of *usedConcreteTypePairs* based on classes' names. We stress the fact that, although this approximation can detect code fragments that contain hidden correlations, the indicated correlated types used in those fragments will probably be imprecise. However, using our intuition, and based on the observations drawn from the results of the *name-based* estimation, we started prototyping a more in-depth algorithm i.e., the *assignment-based* estimation.

a) Name-based Estimation of Used Types Combinations: This approximation started from our conjecture that there could be a connection between the need to accommodate type families in a program's design and *parallel class hierarchies* - code smell defined in [3]. A possible symptom of this smell is that the correlated types' names of both hierarchies might have a common prefix [3]. Therefore, we decided to roughly approximate the *usedConcreteTypePairs* based on the names' similarity.

Let us consider a references pair (*x*, *y)*. For each reference, we compute the set of concrete types from the hierarchy rooted by the references' declared types; let these sets be X and Y, respectively. Next, we say that two concrete types $a_i \in X$ and $b_j \in Y$ can be used only together when Formula 2 holds. For the remaining concrete types for which no such relation is found, we consider that they can be used in any combination. In Formula 2, we mention that $Tokens(t)$ is the set of tokens obtained by splitting the name of t based on a naming convention (e.g., CamelCase).

$$\frac{|Tokens(a_i) \cap Tokens(b_j)|}{avg(|Tokens(a_i)|, |Tokens(b_j)|)} \geq SIMRATIO \quad (2)$$

For instance, let us consider the $SIMRATIO = 0.5$. For the (*wineP*, *glassP*) pair, the sets of concrete types are {*RedWine*, *WhiteWine*} and {*RedWineGlass*, *WhiteWineGlass*}, respectively. *RedWine* is a compound of the {Red, Wine} tokens, and *RedWineGlass*, of {Red, Wine, Glass}. These sets having 2 tokens in common and an average size of 2.5, the names' similarity is $2/2.5 = 0.8 \geq 0.5$. Consequently, the pair (*RedWine*, *RedWineGlass*) is considered part of the *usedConcreteTypePairs* set. In contrast, *RedWine* cannot be paired with *WhiteWineGlass* as their name similarity is $1/2.5 = 0.4 \leq 0.5$. Ultimately, the resulted *usedConcreteTypePairs* set is {(*RedWine*, *RedWineGlass*), (*WhiteWine*, *WhiteWineGlass*)}.

b) Assignment-based Estimation of Used Type Combinations: The prototypical algorithm we propose to approximate the *usedConcreteTypePairs* is represented by an heuristic inter-procedural flow-insensitive and (most of the time) object-insensitive static analysis, being divided in two phases, described below. We emphasise that, as mentioned in Section II-A, we currently consider only method parameter references. Consequently, the next phases are currently specific to the analysis of parameters and not of all kinds of references.

Assignments Combination. Let us consider a parameter pair *(x,y)* for which we want to estimate the *usedConcreteTypePairs* set. For each parameter *x/y*, we identify its corresponding actual arguments based on the method's invocation sites[2]. We then build a set of *assignments* entities i.e., tuples of the form *(parameter, callTargetReference, actualExpression)*. In our example, for the *wineP* parameter, this set is {*(wineP, wt1, new RedWine()), (wineP, wt2, tmp)*} due to the *setWine* invocations in lines 5 and 10 from the *WineBar* class. Similarly, for the *glassP* parameter, this set is {*(glassP, wt1, new RedWineGlass()), (glassP, wt2, g)*}.

Further, for a parameters pair (x, y), we construct the cartesian product between the *assignments* sets of x and y

[2]Currently, we consider call-sites only based on static method resolution

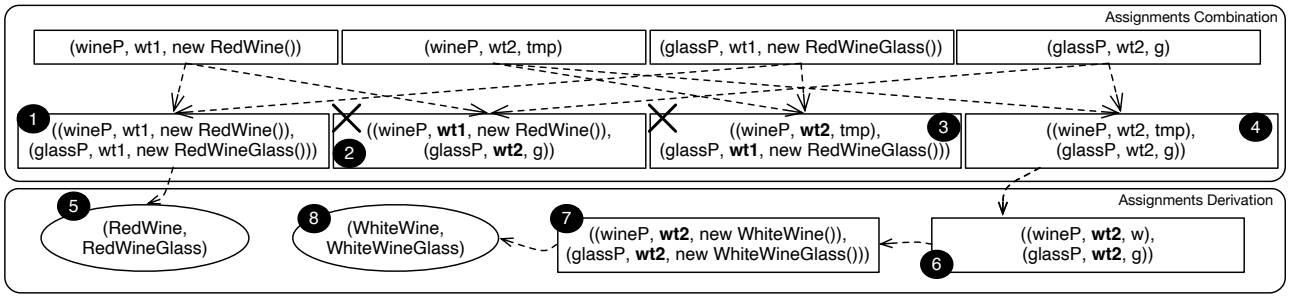

Fig. 3. Assignment-based Estimation Example

and we filter out the pairs whose assignments are performed using different target references. This is because we must capture correlated invocations on the *same object*, as it will use/store the members of the type family. For the parameters pair *(wineP, glassP)* the set of assignments pairs is {*((wineP, wt1, new RedWine()), (glassP, wt1, new RedWineGlass())), ((wineP, wt2, tmp), (glassP, wt2, g))*} i.e., elements marked with 1 and 4 in Figure 3. The other combinations are discarded as their assignments have a different target references i.e., the elements marked with 2 and 3.

Assignments Derivation. The assignments pairs resulted from the previous phase are used as starting point for an iterative worklist algorithm - stops when this list is empty. In each iteration, a single assignment pair is retrieved from the list and then analysed. Let it have the form *((x,callTargetReference,expAssignedToX), (y,callTargetReference,expAssignedToY))*.

When both types of *expAssignedToX* and *expAssignedToY* can be statically resolved to a single concrete one, the pair of resolved types are added to the *usedConcreteTypePairs* set e.g., the element marked with 1 in Figure 3 generates the concrete pair type marked with 5. Otherwise, from the current assignments pair, we derive new ones, and add them to the worklist for further analysis[3]. The derivation process depends on the kind of the *expAssignedToX* and *expAssignedToY* expressions.

- If one is a local variable, we generate new assignments pairs, replacing that variable with each of the expressions assigned to it. A similar derivation is done for other simple expressions e.g., conditional expressions, casts. An example of such a derivation appears in Figure 3, when the assignment pair marked with 4 produces the element marked with 6.
- If both are method parameters, we generate new assignments pairs, replacing them with their actual arguments, retrieved from all method call-sites.[4] An example of such a derivation appears in Figure 3 when the assignment pair marked with 6 produces the element marked with 7.
- In any other cases, the *expAssignedToX/Y* are not derived and we consider the case inconclusive.

[3]We ensure that equivalent assignment pairs are analysed only once and thus, the algorithm do not enter into an infinite loop

[4]We additionally limit the number of this kind of derivation to a user-defined threshold; when exceeded, the assignments pair is treated as inconclusive

TABLE I
DETECTED CLASSES WITH HIDDEN FAMILIAL TYPE CORRELATIONS

Classes with relevant pairs	Name-based AC	Assignments-based AC Max Depth of 3	Assignment-based AC detection precision
488	412	60	86%

When the list is empty, if the number of cases when a concrete types pair was identified is greater than the number of inconclusive cases, we disregard the latter. Otherwise, we consider as members of *usedConcreteTypePairs* all possible concrete types pairs resulted from the static types of *expAssignedToX/Y*, including the inconclusive cases.

III. CURRENT RESULTS

In order to design and verify our detecting approach, we analysed the *Kettle-Engine* project (containing about 2000 classes), of the *Pentaho-Kettle*[5] system, within which the presence of families was previously reported in [2].

Initially, we identified the potential clients of families that might contain type correlations hidden behind parameters. Specifically, these clients are classes and interfaces containing relevant parameters pairs i.e., each parameter can refer different object types at runtime. We then filtered only those for which the *aperture coverage* computed using the *name-based* estimation was a non-zero yet below 0.5 value. After manually analysing some of these classes, we managed to refine the very initial version of the *assignments-based* algorithm. Table I contains the current detection results using both approximations. We mention that: i) to the best of our understanding, the precision of detecting code fragments containing hidden familial type correlations using the *assignments-based* aperture coverage is of 86% and ii) apart from one, all cases detected using the *assignment-based* algorithm were also detected by the *name-based* one.

Listing 1 presents two cases detected by both approaches. Apparently, invoking the *processRow operation* is valid using any combination of *StepMetaInterface* and *StepDataInterface* objects. However, its parameters hide type correlations, which are harder to observe in the *BaseStep*'s source code, yet easier

[5]https://github.com/pentaho/pentaho-kettle

in all of its descendants. This is due to the use of downcast operations as seen in the *Calculator* class.

Listing 1. Detected Code Fragments

```
public class BaseStep ... {
  public boolean processRow(StepMetaInterface smi,
    StepDataInterface sdi) throws ... { ... }
}

public class Calculator extends BaseStep ... {
  public boolean processRow(StepMetaInterface smi,
    StepDataInterface sdi) throws ... {
      meta = (CalculatorMeta) smi;
      data = (CalculatorData) sdi; ...
    }
}
```

Furthermore, these examples can be used to contrast the two estimations of the *aperture coverage*. In the case of *Calculator*, the *assignment-based* algorithm captures a single type correlation i.e., between *CalculatorMeta* and *CalculatorData*, as it actually happens in reality. The *name-based* approach, however, identifies multiple correlations, which in fact correspond to different *BaseStep*'s descendants. That is because the *name-based* technique can only identify correlations that generally exist between some types, without indicating those that are actually *used* in a particular problematic code fragment. As we previously mentioned, the *name-based* estimation of *usedConcreteTypePairs* is too rough. Additionally, although we did not observe such cases, we can already imagine situations where the presence of type families does not imply that their members have correlated names, and thus hindering the effectiveness of the *name-based* estimation. Due to these reasons, we need the *assignment-based* way to compute the *aperture coverage* metric.

Nevertheless, the *assignment-based* algorithm presents a natural drawback: it cannot work properly with insufficient data e.g., there are none/too few object types flowing through the parameters. For instance, some of the *BaseStep*'s descendants are not detected using *assignment-based aperture coverage*, although they contain hidden type correlations. This is due to the fact that many type correlations flow from tests; these false-negative cases are in fact not tested. Interestingly, these cases are although detected when *aperture coverage* is *name-based* estimated. Thus, even if the used type correlations are erroneous, the *name-based* estimation at least reveals the code containing hidden type correlations.

Based on this discussion and other observations drawn from the analysed code, we conclude that i) further improvements of the *assignment-based* estimation are required/possible and ii) we might define very specific use-cases, when one *aperture coverage* approximation should be preferred in the detriment of the other.

IV. Related Work

Initially detailed in [1], *family polymorphism* is a language mechanism, disregarded by most of the mainstream object-oriented languages. In its absence, the idea of non-mixable families is prone to be implemented in an statically unsafe manner.

This fact is supported by the work of Mastrangelo et. al. in [2], where the authors analysed the occurrence of cast operations in Java programs. They have shown that 6.86% of casts are caused by the need to accommodate type families. This implies that references whose runtime types are correlated despite their declared type do exist in practice (i.e., hidden familial type correlations). Therefore, so as to support program comprehension, we aim to detect code fragments containing such references, together with their underlying correlations.

In [4], the authors present a pattern that safely and statically accommodates type families, which makes use of the *F-bounded* parametric polymorphism, available in Java. Consequently, the results of our detecting approach could also be used to design/trigger refactoring actions, using the mentioned pattern, in order to increase the program type safety.

Still, using parametric polymorphism might involve other problems, as explained in [1]. For instance, in C++, it makes code less reusable since it tends to depend on concrete type families, not on abstract ones. In other words, the principle "program to an interface, not an implementation" [5] might be disobeyed. To favour both, code reuse and safety, a dedicated language for type families is therefore needed.

V. Conclusions and Future Work

In this paper we proposed the detection of Java code containing hidden familial type correlations, which very likely cause comprehensibility problems. We also presented a detection approach that was refined by analysing a system known to contain families of types.

As the current results are promising, we thus plan further refinements of the detection technique (e.g., analyse pairs of fields, handle collection references whose contained concrete types might be correlated, enhance the derivation phase of the *assignments-based* estimation by detailing more expressions kinds, such as fields and method calls). We are also considering dynamic analysis as an alternative implementation of our approach. What is more, an important aspect is to also enrich the number of case-studies, especially to observe other real cases of hidden correlations and their particularities.

Additionally, the definition of specific use-cases for the usage of one *aperture coverage*'s estimation instead of the other is also considered.

References

[1] E. Ernst, "Family polymorphism," in *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, ser. Lecture Notes in Computer Science, J. L. Knudsen, Ed., vol. 2072. Springer, 2001, pp. 303–326.

[2] L. Mastrangelo, M. Hauswirth, and N. Nystrom, "Casting about in the Dark: An Empirical Study of Cast Operations in Java Programs," *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–31, 10 2019.

[3] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[4] B. Greenman, F. Muehlboeck, and R. Tate, "Getting F-Bounded Polymorphism into Shape," in *PLDI*, 2014, pp. 89–99. [Online]. Available: http://www.cs.cornell.edu/ ross/publications/shapes/

[5] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994.