Empiric Evaluation of the Usability of Virtual Function Calls within Constructors

Tali Shragai

Technion - Computer Science Department - M.Sc. Thesis MSC-2009-07 - 2009

Empiric Evaluation of the Usability of Virtual Function Calls within Constructors

Research Thesis

Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Tali Shragai

Submitted to the Senate of the
Technion - Israel Institute of Technology

Kislev 5769 Haifa December 2008

The research thesis was done under the supervision of Assoc. Prof. Yossi Gil in the Department of Computer Science.

I would like to express my deepest gratitude to Assoc. Prof. Yossi Gil for his devoted guidance throughout the work.

This work is dedicated to my dear family and friends. Their love and support give me the power to fulfill my dreams.

In loving memory of my father, David Shragai.

Contents

List of Figures

List of Tables

Abstract

Most OO languages abide by the semantics that the constructor of a derived class is a refining extension of one of the base class's constructors. As this base constructor computes, it may invoke dynamically bound methods which are overridden in the derived class. When such methods are invoked, they receive a "half baked object", i.e., an object whose derived class portion is uninitialized. Such a situation may lead to confusing semantics, and complicated coupling between the base and the derived classes. Further, this possibility makes it difficult to introduce mechanisms for expressing design intent within the programming language, and to develop automatic tools for the checking of these. Prime examples include: *non-null annotation* (denoting reference values which can never be null), *read-only* annotation for fields and variables (expressing the intention that these cannot be modified after they are completely created) and *class invariants* (part of the "design by contract" methodology). A read-only field, for example, becomes immutable only after the creation of the enclosing object is complete.

This work is an empirical investigation of the current programming practice in JAVA [1] of calling dynamically bound methods inside constructors. In a data set comprising a dozen software collections with over sixty thousand classes, we found that although the potential for such a situation is non negligible (prevalence $> 8\%$), i.e., there are many constructors that make calls to methods which *may* be overridden in derived classes, this potential is rarely realized. We found this behavior in less than 1.5% of all constructors, inheriting from less than 0.5% of all constructors. Further, we found that over 80% of these incidents fall into eight "patterns", which can be relatively easily transformed into equivalent code which does not make pre-mature calls to methods.

Another similarly undesirable situation occurs when a constructor exposes the self identity to external code, and this external code chooses to call methods overridden in the derived class. Our estimates on the prevalence of this exposition are less accurate due to the complexity of interprocedural dataflow analysis. The

resulting estimates are high, but there are indications that this estimation arises from a relatively small number of base constructors.

Chapter 1

Introduction

Women who have given birth can testify that the process is not infinitesimally short. Objects are no different than babies in this respect: it takes time to mature a raw memory block into a live object, and during that time computation may occur. Consider a class D that inherits from a class B.[1] Then, (in most object oriented (OO) programming languages) the process of construction of a D-object includes an invocation of a suitable constructor of B, followed by an invocation of a constructor of D. What is the status of the object in the course of the evaluation of the constructor of B? On one hand, this object cannot be thought of as a mature, ordinary object of class D, since D's constructor was not invoked yet. On the other hand, thinking of the object as an instance of class B, may lead to surprising results, e.g., in the case that B is an abstract class. Concretely, suppose that B's constructor invokes a dynamically bound member function implemented in both B and D. The dominating *thesis*, taken by languages such as JAVA and C$^{\#}$ [15], is that of *dynamic binding* within constructors, i.e., D's implementation is executed. The *anti-thesis* of *static binding*, taken in languages such as C++ [27], dictates that B's implementation is executed.

This research sets its objective in understanding how such "half-baked" objects are used in actual programs. Our research method is primarily *empirical*: Following the tradition of works such as those done by Cabral and Marques [5], Chalin and James [6], Baxter et al. [2] and Eckel and Gil [10], we apply static analysis techniques combined with manual inspection to a large software data set. The interest in the study is raised by the inherent limitations of both the dynamic- and the static- binding approaches. Any new, competing proposals, must be evaluated

[1] Throughout this thesis, we will refer to the inherited (or extended) class B as *base* class or *superclass*, and to the inheriting class D as *derived* class or *subclass*.

against the common programming practice which this research tries to discover.

1.1 The Static vs. the Dynamic Binding Semantics within Constructors

1.1.1 Static Binding within Constructors

Somewhat paradoxically, the static binding approach may compromise static type safety, as demonstrated in the C++ code excerpt in Figure 1.1.

```
1 class Shape { public: Shape() { draw(); }
2               public: virtual void draw() = 0;
3 };

5 class Circle: public Shape {
6       public: Circle() { cout << "Circle::Circle()\n"; }
7       public: void draw() { cout << "Circle::draw()\n"; }
8 };
```

Fig. 1.1: Pure virtual function call in C++.

In the figure, we see an abstract class `Shape` containing an abstract ("pure virtual" in the C++ jargon) function `draw` (Line 2) which is then implemented (Line 7) in the inheriting concrete class `Circle`. In this example, any attempt to instantiate `Circle` will result in a runtime error: `Circle`'s constructor implicitly invokes the default constructor of `Shape`, which in turn, as a consequence of the static binding semantics of C++, invokes the bodiless function `Shape::draw`. More precisely, the C++ semantics attributes that error to the attempt to call a pure virtual function, rather than to the fact that this function has no body; the error would have occurred even if `draw` had body.

Clever compilers (GCC [26] is a case in point) may detect and warn the programmer against this particular case in which the call to a pure virtual function from within the constructor is so obvious. The general case, which may involve a chain of aliases and virtual function calls is however untraceable [13]. Consider the example in Figure 1.2, which revisits the previous example, with a slight change in the constructor of `Shape`.

Figure 1.2 is similar to Figure 1.1 in that in both code excerpts, the pure virtual function `Shape::draw()` is called as a result of declaring the variable `c` of class `Circle` and invoking its constructor. However, since the virtual call here

```
1 class Shape{
2    public:
3       virtual void draw()=0;

5       void value(){
6          ...
7          draw();
8          ...
9       }

11      Shape(){
12         value();
13      }
14 };
```

Fig. 1.2: Non-trivial example for pure virtual function call in C++.

is done indirectly, through the non virtual function `Shape::value()`, as written in lines 5–12, it is more difficult for the compiler to alert the programmer against this run-time error.

Static binding semantics is also found in C++'s destructors, where it causes a similar problem with pure virtual function calls invoked from the base destructor.

Figure 1.3 displays the error message appearing when this type of run-time error occurs in the code of the popular web browser Internet Explorer. This is an unfortunately common example for the user-visible outcome of static type safety being compromised due to static binding semantics in C++'s constructors.

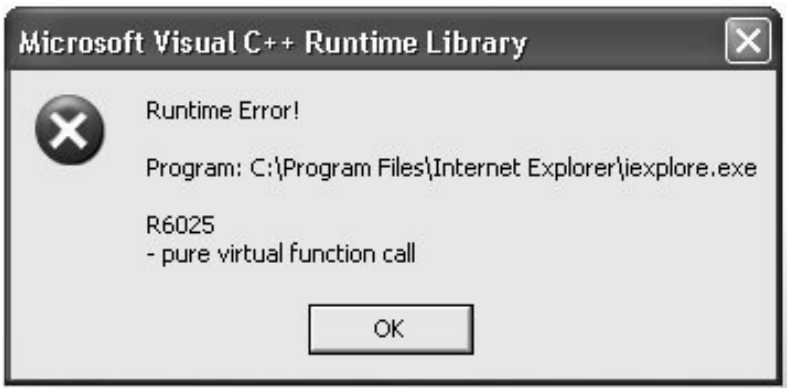

Fig. 1.3: A common problem with pure virtual functions.

1.1.2 Dynamic Binding within Constructors

C++ designers chose the static binding semantics within constructors probably because the language defines no default initial value of data members. In languages with such a default value, the dynamic binding approach makes sense: an object is in some defined state even prior to actual invocation of the construction. The JAVA equivalent of Figure 1.1 behaves as follows when an instance of `Circle` is created: first the constructor of `Shape` is invoked, which then invokes the `Circle`'s version of `draw`; then the constructor of `Circle` is completed.

The difficulty with this approach is that with modern software architectures, the predefined state, i.e., **`null`** in all reference fields, 0 in numerical fields, etc., is too degenerate to be useful. In our little example, it is not clear that a circle can be drawn before the constructor of this class has set crucial data such as location and radius. More generally, this pre-defined state contradicts non-null promises, **`final`** guarantees etc.

Dynamic binding in constructors means that methods may be called prematurely. Such methods are restricted since they cannot rely on any of the fields of the derived class for being properly initialized, and in general should be ready to deal with an object whose construction is incomplete. The working of the constructor is complicated by its coupling with dynamically bound methods. The fact that the constructor is a method called precisely once for each object, whereas the other methods may be invoked any number of times, may even add to the complexity.

Figure 1.4 demonstrates the confusing situation of a prematurely called method in actual industrial code. In the figure we see (parts of) class `Compiler`, drawn from package `org.eclipse.jdt.internal.compiler` of the Eclipse JDT. The constructor of this class, calls method `initializeParser` (Line 12), which as its name indicates, is in charge of initializing instance variable `parser`.

The implementation of derived class `CodeSnippetCompiler` is depicted starting Line 15 of Figure 1.4 . The overridden version of `initilializeParser` in lines 18–21 specializes the `parser` field with a parser suitable for parsing code snippets. The newly created `CodeSnippetParser` object is constructed in the overridden version of the function from three data members of the overriding class: `evaluationContext`, `codeSnippetStart` and `codeSnippetEnd` (defined in lines 24–25).

The constructor of this class starts by calling the refined base constructor in Line 32. The overridden version of `initializeParser()` is called from the refined constructor, but this function cannot complete its mission correctly, since

```
1 public class Compiler {

3    public Parser parser;

5    public void initializeParser() {
6       this.parser = ...;
7    }
8    public Compiler( ...constructor's arguments omitted for brevity... ) {
9       // create a problem handler given a handling policy
10      this.options = new CompilerOptions(settings);
11      //...
12      initializeParser(); // call to a non−final function
13   }
14 }
15 public class CodeSnippetCompiler extends Compiler {

17  public void initializeParser() {
18   this.parser = new CodeSnippetParser(
19      this.problemReporter, this.evaluationContext,
20      this.options.parseLiteralExpressionsAsConstants,
21      this.codeSnippetStart, this.codeSnippetEnd);
22   }

24  EvaluationContext evaluationContext;
25  int codeSnippetStart, codeSnippetEnd;

27  public CodeSnippetCompiler( ...arguments omitted for brevity...
28     EvaluationContext evaluationContext,
29     int codeSnippetStart, int codeSnippetEnd){

31     super(environment, policy, settings, requestor,
32          problemFactory);
33     this.parser = new CodeSnippetParser(
34          this.problemReporter, evaluationContext,
35          this.options.parseLiteralExpressionsAsConstants,
36          codeSnippetStart, codeSnippetEnd);
37     this.parseThreshold = 1;
38   }
39 }
```

Fig. 1.4: Polymorphic behavior in base constructor.

the three data members it relies on belong to the derived class and could not have been initialized yet. These data members will only be initialized in the context of the derived class, and thus using their values in the context of the base class can cause unexpected results.

In fact, we see that the constructor of `CodeSnippetCompiler` repeats (in lines 33–36) the body of function `initializeParser` (that is lines 18–21), immediately after the call to the refined constructor. The fact that the constructor of `CodeSnippetCompiler` forgets to initialize the three said data members, even though it receives the values for these from its arguments is probably an indication that the code was corrected after it was discovered that the language does not support the design behind `Compiler`.

The code in Figure 1.4 and other programming mistakes (e.g., call to an abstract function to retrieve a member value) we have found in our study show that the dynamic binding is confusing. Furthermore, JAVA forbids making a call to a member function or passing an instance data member as an argument when refining a base constructor or in delegating to another constructor of the same class, as illustrated in Figure 1.5. This figure shows class `Derived` defining a **`static`** data member and a **`static`** function in lines 3–6, and also an instance data member and method in lines 8–11. Class `Derived` defines 3 constructors. The first constructor, calls the base constructor and passes the instance data member and method in Line 14. This is forbidden according to JAVA's restrictions, and therefore will be enforced as a compilation error. The second constructor, however, passes the static members to the **`super`**`()` call, as depicted in Line 18, and since the static members are associated with the class rather than with the created object, this call is allowed and will successfully compile. The third constructor, in Line 22, delegates the object creation work to the second constructor. The first parameter to the **`this`**`()` call is the instance data member, which is not allowed to be passed as a parameter to **`this`**`()`. The second parameter is the **`static`** method invocation, which is a legal parameter, so the compiler will only issue an error for the first argument of **`this`**`()`. The restriction on passing instance members to **`this`**`()` and **`super`**`()` construction calls, is also an indication that a call to an overridden function was not intended to be allowed in JAVA.

But, beyond the confusing semantics, and arguably more importantly, the dynamic binding approach makes it difficult to introduce notions such as non-nullity [12, 6, 20], immutability (e.g.,JAVARI [3, 29] and JAC [18]) and class invariant [23, 19] guarantees into the language. Such guarantees are typically achieved by the constructor. But, the possibility of methods being executed before the constructor even begun, makes it impossible to rely on these guarantees. This is the

```
1 public class Derived extends Base {

3    static int staticDataMember;
4    static int getStaticMember(){
5       return staticDataMember;
6    }

8    int dataMember;
9    int getDataMember(){
10      return dataMember;
11   }

13   public Derived() {
14     super(dataMember, getDataMember()); // compilation error!
15   }

17   public Derived(int i, int j){
18      super(staticDataMember, getStaticMember()); // O.K.
19   }

21   public Derived(int i){
22      this(dataMember, staticDataMember); // compilation error!
23   }
24 }
```

Fig. 1.5: JAVA restrictions on **super**() and **this**() calls.

reason that many of these works introduce non-standard types and annotations to deal with half-baked objects, as detailed below:

1. *Non-null data members.* The ability to distinguish non-null references from possibly-null references at the type level can help detect null dereferencing errors at compile time, rather than as run-time exceptions. Recent researchers, such as Fähndrich and Leino [12], Chalin and James [6] or Male and Pearce [20] have suggested various solutions to the nullity problem in JAVA.

   A main complexity identified in these papers regards object construction. As explained above, the transition from an initial raw state into a mature object requires computation, which is done precisely at the constructor. In the case that a field is declared non-null, the transition between its initial

null value (JAVA and C++'s default) to non-null value, takes place in the constructor, whose duty is to ensure non-nullity of such fields. If the constructor is allowed to make dynamically bound calls, the targets of these calls may be surprised to receive objects which are inconsistent with the non-nullity declaration. Figure 1.6 demonstrates how this can be a problem for non-null fields.

```
1 class A {

3    A(){
4       f();
5    )

7    public int f(){
8       return 0;
9    }
10 };

12 class B extends A{

14   private @NonNullString s;

16   B(){
17      s = "hello";
18   }

20   public int f(){
21      return s.length;
22   }
23 };
```

Fig. 1.6: Non-nullity inconsistency in dynamically bound constructors.

When an object of type `B` is created, the constructor of `A` is called first. The call to `f()`, in Line 4, binds dynamically to `B::f()` , which accesses `s` in Line 21, prior to its initialization in `B`'s constructor, thus causing a null dereferencing error. Fähndrich and Leino suggested a fine-grained solution to the described problem, introducing a secondary type notation for temporarily uninitialized references and annotating the methods that handle those as `[Raw]` [12]. Another variation was presented by Chalin and

James [6]. Their system employed two mechanisms for overcoming the above problem:

(a) A simple data flow analysis for ensuring that all data members are initialized by the class constructor.

(b) Using `@Raw` annotation for reference variables that were not yet initialized, including **`this`**, which is implicitly typed `@Raw` inside the constructor. The `@Raw` type is a super-type of normal reference, as reads from variables of this type may return **`null`**. This results in some limitations on the usage of **`this`**: **`this`** cannot be assigned to a non-`@Raw` field or passed in a non-`@Raw` argument. Likewise, we cannot call methods on **`this`** whose receiver type is not declared as `@Raw`.

2. *Immutable data members.* A language's support for reference immutability is another significant research focus these days. Many researchers have noticed the need for a mechanism for specifying and checking immutability. Recent examples, such as JAVARI [3, 29], JAC [18] and immutability using JAVA Generics [31] have introduced reference immutability extensions to JAVA's type system. The basic idea in these systems is that a type may be given the "read only" annotation. A "read only" reference cannot be used to mutate the internal state of the object it refers to. This notation further strengthens the reference assignability protection provided by the JAVA standard **`final`** keyword.

   A "read only" field in a class is only immutable once the object creation has been completed. This implies that the constructor code is allowed to mutate a read only field, optionally through a non-**`public`** class method. A relatively simple static analysis can ensure that such a method is invoked only from the constructor code, and hence is allowed to mutate the read only data member. However, in case the method is virtual, we can no longer maintain the guarantee on any usage an inheriting class would make of the method's overridden version. For example, the subclass may choose to use the virtual function outside of the object creation context, and thus may cause a violation to the reference immutability property. This case is demonstrated in Figure 1.7: class `Base` defines a `@Readonly` data field `id`, which is set in the class constructor by method `setId()`. Inspecting the implementation of `Base`, it is evident that `setId()` is used only from the constructor code. However, in the derived class `Derived`, the overridden version of

`setId()` is invoked from the externally accessible method `changeId()`, thus violating the `@Readonly` property of field `id`.

```
1 class Base {

3    @Readonly int id;
4    Base(int i){
5       setId(i);
6    )

8    protected void setId(i){
9       this.id = i;
10   }
11 };

13 class Derived extends Base{
14   ...

16   protected void setId(int i){
17      this.id = i * 10;
18   }

20   public void changeId(int j){
21      setId(j);
22   }
23 };
```

Fig. 1.7: Immutability inconsistency in dynamically bound constructors.

In their research on reference and object immutability using JAVA Generics [31], Zibin and colleagues introduced a unique annotation for constructors. The `@AssignsFields` annotation is used in order to prevent a `ReadOnly` **`this`** from being used as a `Mutable` reference and at the same time allow the constructor to assign value to fields and invoke `Mutable` methods.

3. *Design By Contract.* The main idea of Design By Contract [23, 19], or DBC, lies in the collaboration of software system elements with each other, on the basis of mutual obligations and benefits. The concept of DBC was initially developed as part of the EIFFEL [16] programming language, but

today DBC is a popular software design methodology in any programming language.

A method designed under DBC would not only provide a certain functionality, but may also:

- Impose an obligation that the method's caller must supply (called a pre-condition).
- Guarantee a certain property on exit (called a post-condition).
- Maintain a certain property, assumed on entry and guaranteed on exit (called the class invariant). The class invariant is assumed to be kept for every method call after the object's creation has been completed.

Focusing on methods called within the constructor, we note an interesting case: as the class invariant is only valid after the object is fully created, a method invoked during construction need not be expected to maintain the invariant.

A simple solution would be to force such a method to be called only within a constructor, thus allowing to safely avoid the invariant assertion checks for it. For a base class implementation this may be easily achieved by static flow analysis.

However, a simple analysis will not suffice for a virtual function that may be used at any context in an inheritor class potentially causing a class invariant violation outside the creational code. As a result, a more complex solution needs to be developed solely for the handling of virtual function calls inside constructors.

1.2 Safe Constructors

A natural and appealing resolution of the dilemma in choosing between these approaches is in a *synthesis* which *forbids* the process of object creation from making any computation in which there is a difference between the dynamic and static binding semantics. (An interesting alternative is offered by EIFFEL in which the creation of a derived class does not involve a creation of a subobject of the base class.)

We call constructors whose execution is the same in both semantics *safe constructors*. Note that the definition of safe constructors is not focused on preventing

access to uninitialized fields from internally invoked methods, but rather on maintaining predictable results for the process of construction during inheritance. The *benefits* of safe constructors should be clear:

1. *Type Safety.* Safe constructors avoid the type safety problem of the static binding approach.

2. *Reduced Coupling with Base Classes.* With safe constructors, a method defined in a class D can be certain that it receives a D object (more precisely, an object for which a constructor of D has at least begun its operation). This reduces and simplifies the dynamic binding's typical coupling between the class and its base.

3. *Crisp Boundary Between Initialization and Use.* Safe constructors are consistent with the OO thinking by which objects are created and only then used. The predicaments of a pre-maturely called method are avoided, and a method should not be aware of the fact that it may be called from a constructor of a base class. Thus, the premature call to `draw` in Figure 1.1 is simply signalled by the compiler.

4. *Simplified Language Extensions.* With safe constructors the introduction of non-nullity, immutability and invariant statement is simplified. (The problem in introducing these is not completely solved, since one still has to address the problem of a method being called from a constructor of the class itself).

This research is concerned mostly with the *cost* to be paid in introducing safe constructors into languages such as JAVA. Towards this end, we try to estimate the prevalence of non-safe constructors in existing code, and to characterize the use of dynamic binding within constructors.

Our search for non-safe constructors in actual code relies on the following definition of safety:

> *A constructor is* safe *if it is both* monomorphic *(that is, it does not call methods of* **`this`** *which raise the binding question) and* modest *(that is, it does not expose the* **`this`** *reference to an external object).*

The auxiliary notions of monomorphism and modesty are explained in greater detail below, but the intuition should be clear: The examples set in Figure 1.1 and in Figure 1.4 demonstrate cases of polymorphic behavior during construction. To

```
1 public class Thread {
2    public Thread() {
3       init(null, null, "Thread-" + nextThreadNum(), 0);
4    }
5    private void init(ThreadGroup g, Runnable t, String n,
6                      long s) {
7       //...
8       setPriority(priority);
9       //...
10   }
11   public final void setPriority(int newPriority) {
12      checkAccess();
13      //...
14   }
15   public final void checkAccess() {
16      SecurityManager security =
17                System.getSecurityManager();
18      if (security != null)
19         security.checkAccess(this);
20      //...
21   }
22   //...
23 }
```

Fig. 1.8: A constructor revealing a self reference.

see why we would like constructors to be "modest", consider for example the standard JAVA class `Thread`, depicted in part in Figure 1.8.

The no-arguments constructor invokes function `init`, which invokes method `setPriority` which then invokes function `checkAccess`. This chain of calls poses no polymorphic construction risk, since all functions in the chain are either **`final`** or **`private`**.

Tracing the possible chain of calls from this point is more difficult, since we do not know the runtime type of variable `security`. The problem is that in Line 19 function `checkAccess()` delegates part of its work to an external class through the method call to `security.checkAccess(this)`. The implementation of `checkAccess` in class `SecurityManager` may choose to invoke methods on the passed parameter. If the invoked methods are overridden in descendants of `Thread`, then they may be surprised to find an incomplete object.

To make constructors truly safe, we need to make a concrete language definition

forbidding both polymorphic calls and identity exposure from within construction. There is a variety of ways in which such concretization can be made: A naïve, and probably too restrictive, approach is to disallow any function calls from within constructors. A more permissive alternative is to allow constructors to invoke only **`final`** methods which are also *anonymous*, where anonymous methods are defined by the four constraints of Boris Bokowski and Jan Vitek [4]:

$\mathcal{A}1$ "*The reference* **`this`** *can only be used for accessing fields and calling anonymous methods of the current instance.*"

$\mathcal{A}2$ "*Anonymity declarations must be preserved when overriding methods.*"

$\mathcal{A}3$ "*The constructor called from an anonymous constructor must be anonymous as well.*"

$\mathcal{A}4$ "*Native methods must not be declared anonymous.*"

An amalgam of the two extremes is in e.g., introducing of a new method tag **`init`** (which could be realized as an annotation for example) which is to be used for a complete separation of the construction process from the invocation of methods on a constructed object. That is requiring that **`init`** methods are called only by constructors and other **`init`** methods, and forbidding constructors and **`init`** methods from calling non-**`init`** methods. (The demand that **`init`** methods are not overridden and do not expose object identity is mundane.) One may also consider allowing such **`init`** methods "semi-static methods", i.e., methods which are bound dynamically yet have no receiver, in the sense that they do not access non-**`static`** fields or methods.

There is also an alternative perspective in which constraints are placed only on constructors which are invoked by constructors of a derived class. The language design space is further enriched by the many other variants to guarantee that the self reference is not aliased: Bokowski and Vitek alone enumerate and compare six different methods of alias control, and the body of literature on aliasing and ownership (see e.g., a dedicated journal issue [24] or a survey in the PhD thesis of Wrigstad [30]) is still increasing at a staggering rate.

1.3 This Research

The evolution of programming language constructs tends to follow a three stage life cycle: (a) intuitive understanding, (b) language legalese and (c) formalization.

This research begins from the premise that such concrete language definitions and placement of restrictions on software designers require better understanding of how "half-baked" objects are actually used in practice; this is our primary focus in this study. Issues of the actual language definition, and careful weighing of the relative merits of alternatives sketched above and their formalization are left to future work.

This choice of ours is guided by our belief that greater care should be exercised before introducing language constructs preventing self-aliasing in *all* constructors for example, rather than in adding e.g., confined types which do not pertain to all code.

Accordingly, two hypotheses were initially set out for examination:

1. *unsafe constructors in actual code are rarities, and*
2. *most of these can be easily made safe.*

Verification of the first conjecture should make the notion of safe constructors a candidate worthy of inclusion in new languages. Verification of the second conjecture should help encourage changes in the semantics of current programming languages. Alternatively, the understanding of actual use of unsafe constructors in code should help to evaluate the price of placing safe construction requirement in a new language on the clients of that language.

Experiments were run in a software corpus comprising circa 75,000 JAVA user defined types featuring some 85,000 class constructors assembled from a dozen different collections drawn from a variety of application domains. Two principal kinds of measures are reported: First, our estimates on the *number of cases* of use of polymorphism and immodesty in constructors should help in appreciation of the penalty designers have to pay if safe constructors become in effect. A second kind of measure, should be indicative of the *amount of work* required to correct and eliminate such unsafe behavior from the code.

It is difficult in general to define the relative size of a code fragment in which a certain phenomena occurs. Cabral and Marques [5] relied on line counts for measuring the relative code size dedicated to exception handling. Unfortunately, such a number may be dependent on formatting style—the relative increase in line count due to a decision to locate curly brackets on a separate line is not the same in small and large counts. A better measure could be the number of tokens, but this number is still influenced by style. More stable is the number of classes, functions and constructors; fortunately, unlike the problem that Cabral and Marques [5] faced, this measure is suitable for our case. This is the reason that our estimates

of "unsafe" behavior are both class- and constructor- based. We believe that both may be useful, and may be used together in appreciating the tendency of unsafe constructors to accumulate in the same class.

Our investigation here concentrates on the occurrence of polymorphic behavior in constructors. Nevertheless, we report quantitative data of immodest behavior in constructors and classes. As it turns out, our conservative estimates of the prevalence of these are high, which made the task of manual analysis of these more difficult.

Outline. The remainder of this thesis is organized as follows: Chapter 2 describes the software corpus used in our study. Chapter 3 presents our results on the prevalence of polymorphic behavior in constructors, while Chapter 4 describes the results of our manual analysis of a large portion of these cases. Our finding on immodest constructors is presented in Chapter 5. Chapter 6 concludes.

Chapter 2

The Software Corpus

The software corpus used in our empirical study was assembled from the union of collections used in the empirical study of Chalin and James [6] and that of Gil and Maman [14]. We decided however to eliminate the *SoenEA* project from the ensemble of Chalin and James in the interest of reproducibility—an official web page describing the project could not be found. The impacts of this omission should be negligible since this collection is relatively small (52 classes).

Overall, the corpus comprises twelve collections of JAVA code, all of which are freely available on the web:

1. *JRE.* The JAVA *R*untime *E*nvironment is the standard library common to almost all JAVA applications, offering a wide variety of services, such as data structure management, communication, file system and database access, graphical user interface, reflection, decompiling `.class` files, etc. In our experiments we used the most recent (and largest) version of the JRE, i.e., 1.6.0_01 [1].

   Note that the JRE is used in almost all empirical studies of JAVA code (e.g., [14, 8, 22, 2]), although naturally, each such experiment uses a different version of the library.

2. *JBoss.* The JBoss [2] application server is a large, open source implementation of the J2EE standard of a middleware framework. J2EE (and therefore JBoss as well) offers services typical to such frameworks, such as transaction management, logging, etc. The JBoss version used in our experiments

[1] http://download.java.net/jdk6/
[2] http://www.jboss.org

was 3.2.6. Unfortunately, not all sources of this large collection (circa 1,000 packages) were available.

3. *Eclipse.* Arguably one of the most popular interactive development environments for JAVA, Eclipse [3] is an open-source software framework offering services such as GUI, incremental compilation, file buffers, text handling, text editors, etc. In our experiments, we used version 3.0.1 of the project.

   Note that Eclipse was used in the empirical study of Chalin and James [6], although, in contrast with their work which examined just the JDT core (circa 1,130 classes), we used the entire Eclipse implementation.

4. *Poseidon.* This is a popular UML modeling tool delivered by Gentleware[4]. We used version 2.5.1 of the community edition of the product.

   Again, sources of this collection were not available, and worse, the code was apparently obfuscated by an automatic tool. This type of sources is usable for automatic analysis, however it cannot be used for further manual inspection.

5. *Tomcat.* This collection is servlet container used by HTTP servers to allow JAVA code to create dynamic web pages; it is part of the *Apache Jakarta Project*[5]. We used version 5.0.28 of this collection.

6. *Scala.* This is the implementation of the compiler of SCALA [25] programming language; We used version 1.3.0.4 of the project.

   Just like Poseidon, sources of the SCALA distribution were largely unavailable, but this is because the compiler itself is written in SCALA.

7. *JML.* The common JML tools is a set of software tools used for the implementation of the JAVA Modeling Language [19]; version 5.5 of the tools was used in our experiments.

8. *ANT.* This is another component of the Apache [6] project—a build tool which offers functionality that is similar, in principle, to the popular make utility (version 1.6.2).

[3] http://www.eclipse.org
[4] http://www.gentleware.com
[5] http://jakarta.apache.org
[6] http://ant.apache.org

9. *MJC.* MultiJAVA, is a JAVA language extension [7] which adds open classes and symmetric multiple dispatch to the language; MJC is the multiJAVA compiler; version 1.3 of the project was used in the experiments.

10. *JEdit.* This is version 4.2 of the programmer's text editor written in JAVA with a Swing GUI.

11. *ESC.* The *E*xtended *S*tatic *C*hecker programming tool that tries to check some of JML assertions through static analysis. Version 2.0b2 of the product (sometimes called ESC/Java2) was used in the experiments.

12. *Koa.* The Koa Tallying subsystem is a Dutch Internet voting application[7].

Naturally, our software corpus cannot include all freely available specimen of JAVA code, but it should be evident that it represents a sample of non-trivial JAVA libraries, drawn from a wide variety of application domains and implemented by independent groups or vendors. We are therefore inclined to believe that our conclusions will reasonably reflect most common JAVA programs.

Table 2.1 summarizes the size properties of the software collections comprising our corpus. Overall, we have more than 75,000 user defined types organized in some 3,500 packages. We also see that the total number of constructors is greater than 85,000 and that there are a total of more than 66,000 classes.

Examining the table we see that the software collections vary in size: the largest collection is JBoss with close to 16,000 classes, while the smallest has less than forty (the median size is 2,440 classes). We can also see that the majority of the code in our corpus is drawn from three large collections: JRE, JBoss and Eclipse, which are of relatively the same size. The other collections are smaller.

The constructor count was produced by a binary analysis of the bytecode representation of the software. (In general, all automatic analysis reported in this work was done on this representation. We turned to the source for manual inspection as necessary and as described below.) In this representation, with the exception of interfaces, all classes have at least one constructor, since a default, no-arguments constructor is created by the compiler for every class that has no programmer defined constructors.

Note that the number of constructors is close to the number of classes, but the numbers are not the same: a class has on average 1.3 constructors. This does not necessarily mean that the relative number of constructors in which half-baked

[7] http://sort.ucd.ie

Collection	Packages	Types	Classes	Interfaces	Constructors	Avg. No. of Constructors
JBOSS	997	18,697	15,786	2,911	22,089	1.40
JRE	740	16,816	14,603	2,034	20,388	1.39
ECLIPSE	587	16,049	14,232	1,817	15,840	1.11
POSEIDON	593	10,045	8,686	1,359	11,078	1.28
TOMCAT	280	4,335	3,756	579	5,198	1.38
SCALA	96	3,379	2,754	625	3,144	1.14
JML	67	2,316	2,127	189	2,938	1.38
ANT	120	1,968	1,611	357	2,015	1.25
MJC	41	1,140	1,025	115	1,436	1.40
JEDIT	23	805	776	29	895	1.15
ESC	35	643	632	11	713	1.13
KOA	2	37	36	1	38	1.06
Total	3,581	76,230	66,024	10,027	85,772	1.30
Median	108	2,847	2,440	468	3,041	1.26

Tab. 2.1: Size statistics of the twelve collections in the corpus.

objects are used is the same as the relative number of classes in which such objects are used.

As reported previously [14], there are inevitably duplications in the corpus: certain classes occur more than once in the different collections. These repetitions are often due to different versions of the same software base. There were even a few cases in which the same class occurred more than once in the same collections. Nevertheless, repetitions were not too frequent (less than 10%) and since we are trying to determine the prevalence of a rather rare phenomena, the error in not eliminating these is small.

Table 2.2 shows how many base classes were found in the collections of the software corpus, and Table 2.3 displays the same statistics for "base constructors". That is to say, counts of the actual number of classes that have subclasses in each of the collections, and the number of constructors in those classes.

The three columns in each of these tables demonstrate an interesting experimental difficulty, raised in its full gravity by this study. As might be expected, other than the JRE, software collections are not self contained: inevitably, there are classes in each such collection which inherit from classes found in other libraries (most often the JRE). The interaction between constructors of base classes found in one library with constructors of derived classes found in another library

Collection	Internal Base Classes		External Base Classes		Total Base Classes	
JBOSS	1,809	(11.46%)	180	(1.14%)	1,989	(12.60%)
JRE	2,212	(15.15%)	0	(0.00%)	2,212	(15.15%)
ECLIPSE	1,537	(10.80%)	61	(0.43%)	1,598	(11.23%)
POSEIDON	1,140	(13.12%)	308	(3.55%)	1,448	(16.67%)
TOMCAT	543	(14.46%)	71	(1.89%)	614	(16.35%)
SCALA	350	(12.71%)	81	(2.94%)	431	(15.65%)
JML	391	(18.38%)	70	(3.29%)	461	(21.67%)
ANT	230	(14.28%)	39	(2.42%)	269	(16.70%)
MJC	149	(14.54%)	63	(6.15%)	212	(20.68%)
JEDIT	41	(5.28%)	71	(9.15%)	112	(14.43%)
ESC	106	(16.77%)	31	(4.91%)	137	(21.68%)
KOA	2	(5.56%)	13	(36.11%)	15	(41.67%)
Total	8,510	(12.89%)	988	(1.50%)	9,498	(14.39%)
Median	370.5	(13.70%)	66.5	(3.12%)	446	(16.51%)

Tab. 2.2: Base classes in the corpus.

may make the reasoning about unsafe behavior more difficult.

As suggested by the tables, our analysis considers also "external base classes". In most collections, the majority of base classes are internal. In JEDIT and in KOA however, most base classes are external: JEDIT is a typical GUI application, with many of its classes inheriting from the GUI classes of the JRE. KOA is not different in this respect, and it relies on GUI and XML processing services of the JRE, inheriting from the appropriate classes.

The basis for the relative numbers reports is the number of classes and constructors in the collection without external bases. The 102.63% value reported in the external base constructors column for KOA reflects the fact that the number of external base constructors is greater than the total number of constructors found in KOA itself. This does not happen in other collections, and the relative number of external constructors and external bases is typically small, with median value in the 1%–3% range.

It is a fundamental property of JAVA that every non-**`final`** class (with at least one non-**`private`** constructor) may be subclassed. It is also fundamental that every such constructor may be refined. But, how many classes are subclassed in practice? How many constructors get refined? Theoretically, the minimal number

Collection	Internal Base Constructors		External Base Constructors		Total Base Constructors	
JBOSS	2,857	(12.93%)	469	(2.12%)	3,326	(15.06%)
JRE	3,583	(17.57%)	0	(0.00%)	3,583	(17.57%)
ECLIPSE	1,952	(12.32%)	143	(0.90%)	2,095	(13.23%)
POSEIDON	1,714	(15.47%)	689	(6.22%)	2,403	(21.69%)
TOMCAT	819	(15.76%)	185	(3.56%)	1,004	(19.32%)
SCALA	428	(13.61%)	251	(7.98%)	679	(21.60%)
JML	578	(19.67%)	211	(7.18%)	789	(26.86%)
ANT	328	(16.28%)	102	(5.06%)	430	(21.34%)
MJC	233	(16.23%)	195	(13.58%)	428	(29.81%)
JEDIT	66	(7.37%)	223	(24.92%)	289	(32.29%)
ESC	126	(17.67%)	91	(12.76%)	217	(30.43%)
KOA	2	(5.26%)	39	(102.63%)	41	(107.89%)
Total	12,686	(14.79%)	2,598	(3.03%)	15,284	(17.82%)
Median	503	(15.61%)	190	(6.70%)	734	(21.64%)

Tab. 2.3: Constructors of base classes in the corpus.

of classes with no descendants and unrefined constructors is one. In practice, the data in Table 2.3 suggests that only 14.8% of internal constructors are constructors of base classes. The fraction of base constructors increases to about one in five if "external constructors" are included. Also, even if a collection is augmented with all bases, only about one in seven classes serves as a base for other classes.

The observation that even in large software collections most classes do not have descendants, and the majority of constructors are not refined guided our analysis and we have separate measurements of constructors with potentially unsafe behavior and constructors in which this potential is realized.

Comparing the total number of external base classes (988) with the total number of constructors found in these classes (2,598), we find that the average number of constructors in these classes is 2.63, i.e., much greater that the 1.30 average over all classes reported in Table 2.1 above. If only internal base classes and base constructors are considered, the average is still high: 1.49. If all bases, internal and external, are considered together, then the average is 1.61. We conjecture that this phenomenon is explained by two properties of JAVA software: (a) most classes are not intended to serve as bases (as argued above), and (b) classes with more constructors are more likely to serve as bases.

Applying the standard χ^2 test to compare the distribution of the number of constructors in classes with no children, and classes with children, supports claim (b). The test reveals a significant difference between the two distributions and that the fraction of classes with two constructors or more is significantly (at a 99.99% confidence level) higher in classes which serve as bases.

Chapter 3

Polymorphic Constructors

Having described the data set, we turn now to the description of the research method and results. This chapter is devoted to the study of the occurrence of polymorphism within constructors. In the following chapter we explain how such polymorphic behavior may be eliminated. Chapter 5 reports on our finding regarding the exposing of the **`this`** identity within constructors.

3.1 Definitions

As explained above, the polymorphic behavior during the construction process occurs when a derived constructor refines a base constructor. To capture the subtleties of this interaction we distinguish between three kinds of "polymorphic" behavior in constructors:

Polymorphic Constructors. We say that a constructor is *polymorphic* if it calls, directly or indirectly, a method overridden in any one of its derived classes.

A constructor is *monomorphic* if it is not polymorphic. In the example of Figure 1.1, the constructor `Shape::Shape()` is polymorphic while the constructor of the derived class, `Circle::Circle()` is monomorphic. The constructor `Shape::Shape()` is polymorphic since it calls method `draw()`, which is overridden in the subclass `Circle`. The constructor `Circle::Circle()` is monomorphic, since it is empty, and thus poses no danger of polymorphic behavior. Likewise, the constructor of class `Compiler` in Figure 1.4 is polymorphic since it invokes a method which is overridden in a derived class, specifically, method `initializeParser()`, overridden in class `CodeSnippetCompiler`.

Polymorphic Falls Constructors. The definition of polymorphic constructors puts the "blame" for the polymorphic behavior on the refined constructor. Indeed, the fault lies at the refined constructor; however the problem is manifested only when the refining constructor is invoked. We therefore say that a constructor of a derived class is a *polymorphic fall* if it refines a constructor of a base class which sends, directly or indirectly, a message to **`this`**, such that this message is bound to different methods in the base and in the derived classes.

The constructor `Shape::Shape()` in Figure 1.1 is not a polymorphic fall since it refines no other constructors. In contrast, the no-arguments constructor of the derived class, `Circle::Circle()` is a polymorphic fall since it refines the polymorphic constructor `Shape::Shape()`, and moreover, this polymorphic constructor calls the function `draw` whose implementations in the base `Shape` and the derived `Circle` are different. The constructor of class `CodeSnippetCompiler` is likewise a polymorphic fall.

The search for polymorphic falls is guided by our empirical and practical standpoint — every polymorphic fall constructor must be inspected if unsafe behavior is to be understood, and every such constructor should be processed if such unsafe behavior is to be eliminated.

Polymorphic Pitfall Constructors. There are constructors that bear the potential for polymorphic behavior. However, the polymorphic behavior may not be manifested, in case none of the derived classes override any of the potentially polymorphic methods invoked by the constructor. (Recall that only one in seven classes have descendants, and that the majority of constructors are not refined at all, which implies that we might find a substantial difference between the polymorphic pitfalls and the actual polymorphic falls .)

We say that a constructor of a certain class is a *polymorphic pitfall* if it calls, directly or indirectly, a method of its class which *might* be overridden in a derived class, i.e., a method which is non-**`final`**, non-**`static`** and non-**`private`**. Thus, the constructor is a polymorphic pitfall even if this method is never overridden in any of the class's descendants.

3.2 Method

Our analysis was carried out first on the binary representation of the code, using the Java Tools Language (JTL) [9]—a declarative language for code analysis. JTL itself is implemented on top of the Byte Code Engineering Library (BCEL)[1], formerly known as *JavaClass*—a toolkit for static analysis and dynamic creation or

[1] http://jakarta.apache.org/bcel/index.html

transformation of JAVA class files. The analysis was then completed by manual inspection of the source.

The JTL code in Figure 3.1 demonstrates how the search for polymorphic fall constructors was conducted:

The unary predicate `polymorphic_fall_constructor_class` matches all classes which have a polymorphic fall constructor. A constructor of a base class which makes a call to a non-final non-static function, will thus be included in our report each time a derived class overrides this function.

```
polymorphic_fall_constructor_class := !abstract class {
  exists constructor refines* C and infringes C;
};

refines* C := refines C | refines C' and C' refines* C;

refines C := invokespecial C, C constructor and
    declared_in T, C declared_in T', T extends T';

infringes C :=
    declared_in T, C internal_call* M, M overriden_in T;

internal_call* M := internal_call M
    | internal_call* M', M' internal_call M;

internal_call M :=
    declared_in T, invoke M, M declared_in T;

overriden_in T := T declares M, M overrides #;
```

Fig. 3.1: JTL query for classes with polymorphic *fall* constructors.

It is important to note that the search is conservative: predicate `refines` is supposed to match cases in which a constructor relies on a constructor of a base class to create its **`this`** parameter. The predicate, however, also captures cases in which a constructor of a base class is invoked for other purposes. Similarly, in tracing the chain of internal calls by predicates `internal_call*` and `internal_call`, no attempt is made to ensure that these are invoked on the implicit **`this`** parameter.

The analysis represented by Figure 3.1 may therefore flag false positives, but it will not allow any polymorphic fall constructors to go undetected. In our manual

inspection of 226 cases of polymorphic falls in constructors found in the JRE only 24 false positives were found, i.e., the accuracy of the analysis in this collection is about 90%. In 259 such cases inspected manually in the Eclipse collection, only one such false positive was found. We therefore estimate the accuracy of the algorithm as being at least 85%. Classes with polymorphic constructors were found by analyzing the report of constructor falls.

The JTL equivalent for finding polymorphic pitfalls is much simpler, as presented in Figure 3.2. Predicate `polymorphic_pitfall_constructor_class` searches for class constructors invoking a polymorphic function, i.e., a method that may change its functionality through inheritance.

```
polymorphic_pitfall_constructor_class := !final class
    {
        exists constructor calls_polymorphic M;
    };

calls_polymorphic[M:MEMBER]:=
    declared_in T , T declares* M and
    dynamic_invoke M , M polymorphic;

dynamic_invoke[M:MEMBER] :=
                    invokevirtual M | invokeinterface M;

polymorphic := !static , !private and
               (!final | calls_polymorphic M);
```

Fig. 3.2: JTL query for classes with polymorphic *pitfall* constructors.

3.3 Findings

Table 3.1 shows the prevalence of polymorphic behavior in constructors in each of the collections in the software corpus.

The third column of the table tells us that in total, a polymorphic fall occurred in only 1,200 constructors, which constitute slightly less than 1.4% of the total of 85,772 constructors in the corpus. The variety among the different collections is not too large: in some collections no polymorphic construction behavior was found at all, and the maximum ratio of such constructors is 2.91%, achieved at

Collection	Polymorphic Constructors	Polymorphic Fall Constructors	Polymorphic Pitfall Constructors
JBOSS	70 (2.10%)	140 (0.63%)	1,570 (7.11%)
JRE	120 (3.35%)	396 (1.94%)	1,314 (6.44%)
ECLIPSE	86 (4.11%)	302 (1.91%)	1,671 (10.55%)
POSEIDON	55 (2.29%)	209 (1.89%)	1,281 (11.56%)
TOMCAT	12 (1.20%)	32 (0.62%)	335 (6.44%)
SCALA	9 (1.33%)	37 (1.18%)	259 (8.24%)
JML	25 (3.17%)	35 (1.19%)	260 (8.85%)
ANT	5 (1.16%)	10 (0.50%)	148 (7.34%)
MJC	7 (1.64%)	13 (0.91%)	141 (9.82%)
JEDIT	1 (0.35%)	26 (2.91%)	107 (11.96%)
ESC	0 (0.00%)	0 (0.00%)	27 (3.79%)
KOA	0 (0.00%)	0 (0.00%)	3 (7.89%)
Total	390 (2.55%)	1,200 (1.40%)	7,116 (8.30%)
Median	9 (1.48%)	32 (1.04%)	259 (8.07%)

Tab. 3.1: Prevalence of polymorphic behavior in constructors (conservative analysis).

JEDIT. The relatively high rate at this collection is explained by its heavy reliance and inheritance from GUI classes, where polymorphic pitfalls constructors exist in the top of the inheritance hierarchy.

In the second column of the table we see the number of constructors which caused these falls. In total, there were 390 such bad constructors, which make 0.45% of *all* constructors. The second column in the table also shows the fraction of polymorphic constructors from base constructors only. With 1.48% median value, even this fraction is small.

Comparing the second and the third columns we see that on average, every polymorphic constructor creates a polymorphic fall in four other refining constructors.

The fourth column of the table gives the numbers of constructors (internals and externals combined) which are polymorphic pitfalls, that is may cause a polymorphic fall by descendants. We see that the numbers in this column are much higher, with median prevalence exceeding 8%.

Every polymorphic constructor is necessarily a polymorphic pitfall, so it is no wonder that the numbers in the fourth column are greater than those reported in the second. But, a striking conclusion can be drawn from comparing the relative values: the density of polymorphic constructors within base constructors is invari-

ably smaller than the density of polymorphic pitfalls among all constructors. For example, in Eclipse, only about 4% of base constructors created a polymorphic fall, whereas more than 10% of all constructors in this collection have a polymorphic pitfall.

The fact that actual polymorphic behavior is smaller than what might be expected by the potential for it can be attributed to one of two, non-mutually exclusive, reasons:

1. *Few Descendants Conjecture.* Classes with polymorphic pitfall constructors are less likely to be extended

2. *Unrealized Potential Conjecture.* Potentially polymorphic constructors do not realize this potential in full during inheritance, because the potentially polymorphic methods invoked from a constructor of the base are not always overridden.

An experiment or measurement to verify the second conjecture is not simple. Our research continued to test the first explanation against the null hypothesis by which the occurrence of potentially polymorphic behavior within constructors does not change the probability of a class serving as a base.

Consider now Table 3.2 which is similar to Table 3.1 except that it revolves around classes instead of constructors. That is, in Table 3.2 we report on the number of classes whose constructors can be categorized according to the three varieties of polymorphic behavior.

Note again that the number of classes with polymorphic constructors is presented in the table as a fraction of the total number of base classes. The 292 such classes are however only 0.44% of the total of 66,024 classes of our corpus and only 0.38% of the 76,230 types in the corpus.

Also take note that each base class with a polymorphic constructor is, on average, "responsible" for three classes in which an actual polymorphic call occurs.

A comparison of the total lines in tables 3.1 and 3.2 shows that the relative prevalence of constructors and classes is quite similar. The prevalence of polymorphic, polymorphic falls and polymorphic pitfalls constructors is (respectively) 2.55%, 1.40%, and 8.30% whereas the corresponding numbers for classes in which this behavior is found are 3.07%, 1.41% and 8.83%. The similarity also occurs in the median line, and (to a lesser extent) in each of the prevalence values.

The similarity is a bit suspicious, since, as observed above (Chapter 2), classes which serve as bases tend to have more constructors. We should therefore have

Collection	Classes with Polymorphic Constructors	Classes with Polymorphic Fall Constructors	Classes with Polymorphic Pitfall Constructors
JBOSS	44 (2.21%)	122 (0.77%)	1,192 (7.55%)
JRE	89 (4.02%)	262 (1.79%)	968 (6.63%)
ECLIPSE	67 (4.19%)	265 (1.86%)	1,498 (10.53%)
POSEIDON	43 (2.97%)	155 (1.78%)	1,119 (12.88%)
TOMCAT	9 (1.47%)	27 (0.72%)	256 (6.82%)
SCALA	9 (2.09%)	24 (0.87%)	236 (8.57%)
JML	18 (3.90%)	32 (1.50%)	199 (9.36%)
ANT	5 (1.86%)	10 (0.62%)	105 (6.52%)
MJC	7 (3.30%)	13 (1.27%)	124 (12.10%)
JEDIT	1 (0.89%)	18 (2.32%)	103 (13.27%)
ESC	0 (0.00%)	0 (0.00%)	24 (3.80%)
KOA	0 (0.00%)	0 (0.00%)	3 (8.33%)
Total	292 (3.07%)	928 (1.41%)	5,827 (8.83%)
Median	9 (2.15%)	24 (1.07%)	199 (8.45%)

Tab. 3.2: Prevalence of classes with constructors with polymorphic (conservative analysis).

expected that base classes would be more prone to have at least one polymorphic constructor.

To better understand the situation, we applied a statistical test to check whether classes with polymorphic behavior in one of their constructors have the same number of descendants as other classes.

The statistical test resulted in the following:

1. Classes with polymorphic pitfalls constructors tend to have *more* descendants than classes without such constructors.

2. Classes with polymorphic constructors have a *greater* number of descendants than other base classes.

Both results were found to be statistically significant (with confidence level of at least 99%) by a variant of the of the Mann-Whitney [21] test for comparing ordinal non-normally distributed unpaired data sets. These findings indicate that the second conjecture is more likely to be true: polymorphic pitfalls are not realized as often as they can be during inheritance.

3.3.1 Topology Analysis

Another interesting aspect of polymorphic behavior is observed when researching the class hierarchy "topology" of classes with polymorphic constructors and their descendants. For that purpose, we examined the polymorphic constructors identified in our analysis of the most commonly used JAVA code; that of the JRE and Eclipse projects.

Table 3.3 summarizes our findings of inheritance structure in clusters of classes with polymorphic behavior. Each of the classes with polymorphic constructors may have more than one such constructor, and each such constructor may have more than one polymorphic method invocation. However, we found that out of 439 classes examined, there were 485 cases of polymorphic behavior in constructors, and hence in most cases, there is only one fall for each such class, i.e., in most classes, only one of the constructors will cause a polymorphic fall.

	Topology													
	Pairs	**Chains**	**Stars**											**Other**
Collection		**3**	**3**	**4**	**5**	**6**	**7**	**13**	**14**	**18**	**23**	**30**	**75**	
Java JRE	43	0	12	8	0	0	1	1	1	0	0	1	0	1
Eclipse	26	1	16	4	3	2	2	0	0	1	1	0	1	0
Total	69	1	28	12	3	2	3	0	1	1	1	1	1	1

Tab. 3.3: Topology of classes with polymorphic falls constructors.

As the table indicates, the vast majority of such cases fall in fairly simple topologies, which do not complicate reasoning about (and resolution of) polymorphic failures. The most common topology is also the simplest one of all: pairs, in which the polymorphic interaction occurs between a class and an ancestor *and* neither the class nor its ancestor are involved in any other such case. Chains, i.e., cases involving a polymorphic tie between a class and its parent, and a tie between the parent and its parent were rare. Only one chain of length three was found in our data. Stars, that is a class influencing two or more of its descendants were rather frequent. The data in the table also indicates that fairly large stars do occur, e.g., class `org.eclipse.jdt.core.dom.ASTNode` of Eclipse has 75 descendants which redefine method `getNodeType0()`, that is called from the base constructor, and thus cause polymorphic behavior in the many derived classes.

3.4 Summary

Our experimental findings in this corpus show that polymorphic constructors are rather rare—the prevalence of this phenomena is between 1% and 2%, depending on how the measurements are made. More precisely, we found that:

- About 98.6% of all constructors in the corpus do not have a polymorphic fall; also, about 98.6% of all classes do not have a polymorphic fall.

  The complement of this ratio is indicative of the total amount of work required to eliminate such falls.

- These polymorphic falls are caused by the 390 polymorphic constructors; the remaining 99.55% of all constructors are monomorphic; the ratio of classes with such behavior is similar.

  The complement of this ratio is indicative of the number of distinct cases to be considered if such falls are to be eliminated. On average each such case involves three descendant classes and four refining constructors.

- About 8% of all constructors are a polymorphic pitfall, that is, pose a risk to have descendants with polymorphic falls. Still, even though classes with polymorphic pitfall constructors tend to have more descendants, the fall is not realized in all of these descendants.

- Most of the classes with polymorphic behavior are found in relatively simple topologies. This information may be found useful when applying techniques to eliminate polymorphic behavior in constructors, as detailed in Chapter 4.

Recall that these conclusions are drawn based on a conservative code analyzer, whose errors are only false reports on polymorphic behavior. The true results are probably (slightly) better, in the sense that polymorphic behavior is scarcer than the above numbers indicate.

Chapter 4

Patterns of Polymorphic Behavior in Constructors

In order to better understand the nature of polymorphic calls in the code base, we conducted a detailed manual inspection of 485 cases of polymorphic failures. A *case of polymorphic failure* is defined as a triple of (i) a constructor of a base class, (ii) a refining constructor of a derived class, and (iii) a method called by the base constructor with different implementation in the base and the derived class. 226 of these cases were drawn from the JRE; the remaining 259 cases were taken from Eclipse.

4.1 Polymorphic Solution Patterns

Our manual inspection of the said cases revealed that the polymorphic behavior during construction appears in a relatively small number of patterns. We have identified those patterns and created a group of solutions targeted at each pattern: CONSTANT INITIALIZER, SEMI-CONSTANT INITIALIZER, INITIALIZER OBJECT, FUNCTION OBJECT, MULTI-FUNCTION OBJECT, FACTORY and INLINE DELTA.

Figure 4.1 depicts the relationship between these patterns. An arrow from one such pattern to another indicates that the former generalizes the latter.

The most general pattern is MULTI-FUNCTION OBJECT, while the most specific one is CONSTANT INITIALIZER. Patterns FUNCTION OBJECT and INITIALIZER OBJECT present distinct generalizations of NON-CONSTANT INITIALIZER, while MULTI-FUNCTION OBJECT further generalizes by unifying the behavior of both FUNCTION OBJECT and INITIALIZER OBJECT. FACTORY and INLINE DELTA

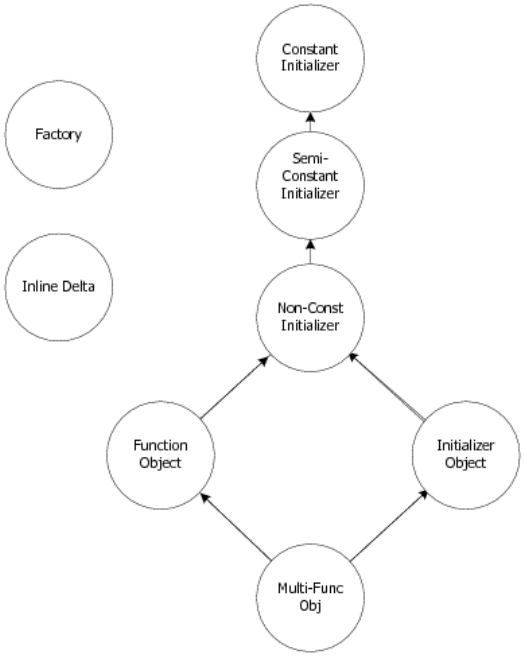

Fig. 4.1: Design patterns for de-virtualization constructors.

are isolated in the sense that they do not generalize, nor are being generalized by, any of the other patterns.

More concretely, these patterns are:

1. CONSTANT INITIALIZER: the most common type of virtual methods called inside a constructor is the type of methods that return a constant value, or a static field, that is known in the subclass only, and needed by the superclass. Examples for this type of behavior may be found in some of the large inheritance star shaped topologies such as those rooted by JRE's `Enumerated` (from Sun's jmx.snmp package), `ElementProxy` (from Sun's apache package) and Eclipse's `ASTNode` (from JDT package). For example, the constructor of class `ASTNode` invokes the abstract method `getNodeType0()`. The subclasses of `ASTNode` all implement this method by returning a class constant value.

   These virtual calls may be avoided by adding a parameter to the super constructor and passing the constant value or static data member in the call to **`super`**`(...)`.

2. SEMI-CONSTANT INITIALIZER: similarly to the previous case, a method

with no arguments invoked from a superclass constructor may return different newly created objects, depending on the subclass implementation. The overriding methods in each subtype contain a single **new** statement to create and return a new object of a specific and distinct type per each subclass. Furthermore, the constructor invocation uses no receiver fields.

For example, the Eclipse class `NewPluginTemplateWizard` calls method `createTemplateSections()` in order to initialize one of its instance variables. Figure 4.2 displays subclass `MultiPageEditorNewWizard`. Its constructor performs a simple, no arguments call to **super**`()` (Line 5). The overriding method `createTemplateSections()` returns a newly created object in Line 9.

```
1 public class MultiPageEditorNewWizard
2       extends NewPluginTemplateWizard {

4    public MultiPageEditorNewWizard() {
5             super();
6       }

8    public ITemplateSection[] createTemplateSections() {
9             return new ITemplateSection [] {
10                          new MultiPageEditorTemplate(),
11                          new NewWizardTemplate() };
12      }
13 }
```

Fig. 4.2: SEMI-CONSTANT INITIALIZER pattern in a subclass.

An alternative for this pattern of behavior may be implemented similarly to the CONSTANT INITIALIZER case, by passing the new object creation expression as a parameter to the **super**`()` call. Eliminating the non-safe code in our previous example requires a change in both the superclass and the subclass: The constructor of `NewPluginTemplateWizard` will be changed to receive one parameter of type `ITemplateSection[]`, that will be used instead of `createTemplateSections()`. The call for **super**`()` from subclass `MultiPageEditorNewWizard` will be now changed to the code appearing in Line 5 of Figure 4.3, where we can see that the creation expression is now a parameter to the **super**`()` call.

```
1 public class MultiPageEditorNewWizard
2       extends NewPluginTemplateWizard {

4    public MultiPageEditorNewWizard() {
5             super(new ITemplateSection [] {
6                         new MultiPageEditorTemplate(),
7                         new NewWizardTemplate() });
8       }
9 }
```

Fig. 4.3: Eliminate polymorphic calls with SEMI-CONSTANT INITIALIZER.

3. NON-CONSTANT INITIALIZER: a more general case requires the subclass to perform computation on its constructor arguments or static data members. As in the previous case, this is done inside the overridden method, resulting in a value used for the superclass constructor.

   Such polymorphism can be resolved as in the previous pattern: the computation itself can be written as an argument in the call to `super(...)`, thus passing the computed value from the subclass to the superclass as a constructor parameter.

   The examples in Figure 4.4 and Figure 4.5 demonstrate the NON-CONSTANT INITIALIZER pattern in Eclipse class `CyclePartAction` and its subclass `CycleEditorAction`. The code presented includes only the constructors of both classes and both versions of the polymorphic method called from the superclass constructor; the rest of the implementation is irrelevant for our discussion and therefore omitted. In Figure 4.4, the constructor of `CyclePartAction` calls method `updateState()` in Line 8. The original version of this method appears in Line 11, and the overridden version, which will be invoked for the construction of a `CycleEditorAction` object, appears in Line 8 of Figure 4.5. Interesting to note is Line 6 in Figure 4.5, where the constructor of the subclass calls the overriding version of `updateState()` explicitly, although it has already been executed from the base constructor, due to JAVA's dynamic binding property. This is possibly an indication for how confusing the semantics of the dynamic binding is for JAVA programmers, as the writer of `CycleEditorAction` obviously did not expect the correct version of `updateState()` to be called.

   A first look on this code might seem difficult to modify, but additional analysis added the following useful information: methods `setEnabed()`

and `getActivePage()`, used inside both versions of `updateState()`, are implemented as **`final`** methods in superclasses of `CyclePartAction`. The implementation of method `getActivePage()`, as a part of base class `PageEventAction`, returns the `page` instance variable, set in the class constructor to the value of `window.getActivePage()`. Note that `window` is a constructor parameter for all the discussed classes in this hierarchy, therefore we may use it directly from the subclasses.

```
1 public class CyclePartAction extends PageEventAction{

3    public CyclePartAction(IWorkbenchWindow window,
4                           boolean next) {
5       super("", window);
6       forward = next;
7       setText();
8       updateState();
9    }

11   protected void updateState() {
12      IWorkbenchPage page = getActivePage();
13      if (page == null) {
14         setEnabled(false);
15         return;
16      }
17       // enable iff there is at least one other part to switch to
18      int count = page.getViewReferences().length;
19      if (page.getEditorReferences().length > 0) {
20         ++count;
21      }
22      setEnabled(count >= 1);
23   }
24 }
```

Fig. 4.4: NON-CONSTANT INITIALIZER pattern in base class.

Figure 4.6 demonstrates how the NON-CONSTANT INITIALIZER pattern may be used in this case for eliminating the polymorphic behavior. In Figure 4.6, the original 2–arguments constructor of `CyclePartAction` is replaced with the 3–arguments constructor in Line 5, where the third parameter is used as a parameter for `setEnabled()` (Line 9), instead of the calcu-

```
1 public class CycleEditorAction extends CyclePartAction{

3    public CycleEditorAction(IWorkbenchWindow window,
4                             boolean forward){
5       super(window, forward);
6       updateState();
7    }
8    public void updateState(){
9        WorkbenchPage page = getActivePage();
10       if (page == null) {
11          setEnabled(false);
12          return;
13       }
14       // enable iff there is at least one other editor to switch to
15       setEnabled(page.getSortedEditors().length >= 1);
16    }
17 }
```

Fig. 4.5: NON-CONSTANT INITIALIZER pattern in subclass.

lation done previously in `updateState()`. The original 2–arguments constructor still exists, and is implemented using a call to the 3–arguments constructor. The third parameter to the **`this`**`(...)` call in lines 16–19 replaces the call to `updateState()` with an equivalent boolean expression. A similar trick is used in Figure 4.6 for the constructor of `CycleEditorAction`, where the **`super`**`(...)` call now invokes the new 3–arguments base constructor, with the relevant boolean expression, as depicted in lines 30–32.

Note that both constructors no longer use method `updateState()`, therefore it is omitted from the code segment in Figure 4.6.

4. INITIALIZER OBJECT: the ability to write a computation as a function argument to the **`super`**`(...)` call is limited to relatively short and simple expressions. Additionally, passing a large number of parameters to the **`super(...)`** call may be inconvenient for the programmer.

   An alternative is passing an INITIALIZER OBJECT as the **`super`**`(...)` parameter. This object will be used to pass the setting values of multiple superclass fields. Additionally, when creating the INITIALIZER OBJECT, its constructor may perform any type of calculation on the values to be set in the superclass fields. In this manner, any subclass may define an INITIAL-

```
1 public class CyclePartAction extends PageEventAction{

3    public CyclePartAction(IWorkbenchWindow window,
4                           boolean next,
5                           boolean enabled) {
6       super("", window);
7       forward = next;
8       setText();
9       setEnabled(enabled);
10   }

12   public CyclePartAction(IWorkbenchWindow window,
13                          boolean next) {
14      this("",
15          window,
16          window.getActiveWindow() == null
17          ? false
18 :(window.getActivePage().getEditorReferences().length+
19  window.getActivePage().getViewReferences().length)>=1
20          );
21   }
22 }

24 public class CycleEditorAction extends CyclePartAction{

26   public CycleEditorAction(IWorkbenchWindow window,
27                          boolean forward){
28      super(window,
29           forward,
30           (window.getActiveWindow() == null
31           ? false
32  :window.getActivePage().getSortedEditors().length>=1)
33           );
34   }
35 }
```

Fig. 4.6: Eliminate polymorphic calls with NON-CONSTANT INITIALIZER.

IZER OBJECT to meet its own needs, and use it to set any number of fields in the superclass.

The previous example in Figure 4.6 may also use a simple INITIALIZER OBJECT to eliminate the polymorphic behavior, by replacing the boolean condition passed to the base constructor with an object whose constructor performs the same computation.

5. FUNCTION OBJECT: further generalization of the INITIALIZER OBJECT is targeted at the setting of superclass data members that are composite components, and are dependant on other data members. As a result, the computation of a dependant data member needs to be delayed till the other data members are set.

   This may be done using the FUNCTION OBJECT micro pattern [14]: the subclass would call for **`super`**`(...)` with a new FUNCTION OBJECT. The creation of the Function Object may set some values from the subclass, that would be used for the superclass data member computation. The superclass constructor will start by setting independent data members. Next, it will invoke the main method of the FUNCTION OBJECT, pass all the needed data members, and receive the value of the composite component as the return value of the FUNCTION OBJECT method.

   The example in Figure 4.7 and Figure 4.8 demonstrates the FUNCTION OBJECT pattern in Eclipse classes `AbstractTableInformationControl` and `BasicStackList` accordingly. The base constructor invokes method `createFilterText` in Line 15 of Figure 4.7, passing `fComposite` as an argument. Note that `fComposite` was initialized inside this constructor in lines 8–10. The base version of this method is **`abstract`**, but the subclass implements it, as depicted in Figure 4.8, starting at Line 10. From the above we get a case in which a base class member is initialized by a subclass method that relies on another base class member, and this is exactly where the FUNCTION OBJECT may help us eliminate the polymorphic behavior.

   Figure 4.9 demonstrates the transformed code of the base and derived classes according to the FUNCTION OBJECT pattern. The base constructor now has a new argument of type `ITableViewInitializer`, that will be used in Line 9 to replace the original call to `createTableViewer`. Similarly, the **`super`**`(...)` call in the subclass constructor in Figure 4.9 now adds a third argument, which is a new object of an inner class implementing interface `ITableViewInitializer`. Inner class `ListTableViewInitializer`

```
1 public abstract class AbstractTableInformationControl{
2    public AbstractTableInformationControl(Shell parent,
3                                           int shellStyle,
4                                           int controlStyle){
5        fShell = new Shell(parent, shellStyle);
6        Display display = fShell.getDisplay();
7        ... // some more initialization of fShell
8        fComposite = new Composite(fShell, SWT.RESIZE);
9        GridLayout layout = new GridLayout(1, false);
10       fComposite.setLayout(layout);

12       createFilterText(fComposite);

14       fTableViewer =
15         createTableViewer(fComposite, controlStyle);
16       ...
17   }

19   protected abstract TableViewer createTableViewer(
20                                 Composite parent,
21                                 int style);
22 }
```

Fig. 4.7: FUNCTION OBJECT pattern in base class.

```
1 public class BasicStackList
2    extends AbstractTableInformationControl {

4    public BasicStackList(Shell parent,
5                          int shellStyle,
6                          int treeStyle){
7        super(parent, shellStyle, treeStyle);
8    }

10   protected TableViewer createTableViewer(
11                            Composite parent,
12                            int style) {
13       Table table =
14         new Table(
15            parent,SWT.SINGLE|(style & ~SWT.MULTI));
16       table.setLayoutData(
17         new GridData(
18            GridData.VERTICAL_ALIGN_BEGINNING));
19       TableViewer tableViewer = new TableViewer(table){
20         ... // definition of class methods
21       };
22       tableViewer.addFilter(new NamePatternFilter());
23       tableViewer.setContentProvider(
24         new BasicStackListContentProvider());
25       tableViewer.setSorter(
26          new BasicStackListViewerSorter());
27       tableViewer.setLabelProvider(
28          new BasicStackListLabelProvider());
29       return tableViewer;
30    }
31 }
```

Fig. 4.8: FUNCTION OBJECT pattern in subclass.

defines the method `execute()` (Line 24) according to the needs of its outer class `BasicStackList`, and performs the actions previously done by the polymorphic method without carrying the polymorphic burden.

6. MULTI-FUNCTION OBJECT: finally, a combination of the INITIALIZER OBJECT and the FUNCTION OBJECT can be implemented using a MULTI-FUNCTION OBJECT. The MULTI-FUNCTION OBJECT differs from the FUNC-

```
1 public abstract class AbstractTableInformationControl{
2    public AbstractTableInformationControl(
3                        Shell parent,
4                        int shellStyle,
5                        int controlStyle
6                        ITableViewInitializer tableInit){
7          ... // same construction code as above
8          fTableViewer =
9            tableInit.execute(fComposite, controlStyle);
10         ...
11   }
12 }

14 public class BasicStackList
15   extends AbstractTableInformationControl {
16   public BasicStackList(Shell parent,
17                         int shellStyle,
18                         int treeStyle){
19       super(parent, shellStyle,
20         treeStyle, new ListTableViewInitializer());
21   }
22   public class ListTableViewInitializer
23            implements ITableViewInitializer{
24      protected TableViewer execute(Composite parent,
25                                    int style) {
26         Table table = new Table(
27               parent, SWT.SINGLE|(style & ~SWT.MULTI));
28         table.setLayoutData(new GridData(
29               GridData.VERTICAL_ALIGN_BEGINNING));
30         TableViewer tableViewer =
31            new TableViewer(table){
32            ... // definition of class methods
33         };
34         ... // further initialization of tableViewer
35         return tableViewer;
36       }
37    }
38 }
```

Fig. 4.9: Eliminate polymorphic calls with FUNCTION OBJECT.

TION OBJECT by externalizing more than just a single component initialization method, and thus may be used for the setting of all the superclass's data members, as done by the INITIALIZER OBJECT for the simpler data members (which are independent of other data members).

Any extension for the example of FUNCTION OBJECT applies here as well, where instead of a single `execute()` method, we will use several methods in order to separately initialize a number of data members that may be differently set in each of the derived classes.

7. FACTORY: this solution is required when the construction process of the object may conceptually be divided into two phases. The FACTORY is an auxiliary wrapper class, which is responsible for creating and initializing objects of the superclass or the subclasses types, without the need for their constructors to be public. Having the construction wrapped by the FACTORY allows for the removal of polymorphic initialization methods from the base constructor, as the FACTORY itself will handle the second phase of the initialization.

   Figure 4.10 demonstrates superclass `PDEFormEditor` and its (indirect) subclasses `SchemaEditor` and `SiteEditor`, all taken from Eclipse project, for which usage of the FACTORY pattern will eliminate polymorphic behavior. Note that this case is very similar to the SEMI-CONSTANT INITIALIZER, since each subclass creates a specific type of `InputContextManager` according to its special needs, and the value is set in the base constructor in Line 9. However, the important difference stems from the fact that **`this`** is passed as an argument to functions invoked from the various version of `createInputContextManager` (Line 19 and lines 28–30). In this case, the functions receiving **`this`** as an argument, are in fact receiving a half-baked object, since none of the constructors have finished executing. A FACTORY wrapper class will separate the construction of the object from the call to `createInputContextManager` into two distinct phases. In this manner, the **`this`** object will only be exposed to the functions called inside `createInputContextManager` after it has been fully constructed. The code in Figure 4.11 depicts the relevant changes: addition of the FACTORY classes `SchemaEditorFactory` and `SiteEditorFactory`, and the phased creational methods in Line 16 and Line 24. Note that the call to method `createInputContextManager` is now deleted from the base constructor (Line 8).

```
1 public abstract class PDEFormEditor extends FormEditor
2             implements
3                   IInputContextListener,
4                   IGotoMarker, ISearchEditorAccess {
5    public PDEFormEditor() {
6             PDEPlugin.getDefault().getLabelProvider().
7                               connect(this);
8             inputContextManager =
9          createInputContextManager();
10       }
11   protected abstract
12   InputContextManager createInputContextManager();
13 }

15 public class SchemaEditor extends MultiSourceEditor{
16   protected
17   InputContextManager createInputContextManager(){
18      SchemaInputContextManager contextManager =
19         new SchemaInputContextManager(this);
20      return contextManager;
21   }
22 }

24 public class SiteEditor extends MultiSourceEditor{
25   protected
26   InputContextManager createInputContextManager(){
27       SiteInputContextManager contextManager =
28         new SiteInputContextManager(this);
29       contextManager.setUndoManager(
30         new SiteUndoManager(this));
31       return contextManager;
32    }
33 }
```

Fig. 4.10: FACTORY pattern in base class and (indirect) derived classes.

The topic of **this**-exposing from within the constructor is another significant aspect of this research, and will be discussed in further detail in Chapter 5. For the FACTORY solution, **this**-exposing is a relatively simple example, although the generic concept is not limited to treating **this**-exposing, and may be found useful for many other complex initialization

```
1 public abstract class PDEFormEditor extends FormEditor
2             implements
3                   IInputContextListener,
4                   IGotoMarker, ISearchEditorAccess {
5    public PDEFormEditor() {
6             PDEPlugin.getDefault().getLabelProvider().
7                               connect(this);
8    // removed the call to createInputContextManager() !
9        }
10   protected abstract
11   InputContextManager createInputContextManager();
12 }

14 public class SchemaEditorFactory{

16   public SchemaEditor getSchemaEditor(){
17      SchemaEditor se = new SchemaEditor();
18      se.createInputContextManager();
19      return se;
20   }
21 }

23 public class SiteEditorFactory{
24   public SiteEditor getSiteEditor(){
25         SiteEditor se = new SiteEditor();
26         se.createInputContextManager();
27         return se;
28      }
29 }
```

Fig. 4.11: FACTORY pattern applied.

processes that can be split into phases.

8. INLINE DELTA: a derived class may refine a method invoked from its base class for the purpose of adding on to the superclass functionality with initialization of the derived class's own data members. This implies that the fields of the subclass are set during superclass construction, rather than during the construction of the derived class itself. This case follows a pattern of an overriding method starting by invoking the superclass's version, and then adding a delta of subclass-specific initialization. The solution for

```
1 public abstract class StoredObject
2          extends Observable
3          implements Referable, Insertable{

5    protected StoredObject(Field f,
6                           ObjectStore store,
7                           ObjectAddress address)
8                        throws ObjectStoreException{
9       if (f.length() < getMinimumSize()) {
10         throw
11            new ObjectStoreException(
12               ObjectStoreException.ObjectSizeFailure);
13      }
14      if (f.length() > getMaximumSize()) {
15         throw
16            new ObjectStoreException(
17               ObjectStoreException.ObjectSizeFailure);
18      }
19      extractValues(f);
20      setStore(store);
21      setAddress(address);
22   }

24   protected void extractValues(Field f)
25                throws ObjectStoreException{
26       type =
27         f.subfield(TYPE_OFFSET, TYPE_LENGTH).getInt();
28       if (type != getRequiredType())
29          throw
30            new ObjectStoreException(
31               ObjectStoreException.ObjectTypeFailure);
32    }
33 }
```

Fig. 4.12: INLINE DELTA pattern in base class.

```
1 class IndexAnchor extends IndexedStoreObject{

3     // data members initialized inside extractValues()
4     protected Field numberOfEntriesField;
5     protected int numberOfEntries;

7     protected Field rootNodeAddressField;
8     protected ObjectAddress rootNodeAddress;

10    public IndexAnchor(Field f,
11                       ObjectStore store,
12                       ObjectAddress address)
13       throws ObjectStoreException{
14        super(f, store, address);
15     }

17    protected void extractValues(Field f)
18          throws ObjectStoreException{
19       super.extractValues(f);
20       setFields(f);
21       numberOfEntries =
22          numberOfEntriesField.getInt();
23       rootNodeAddress =
24          new ObjectAddress(rootNodeAddressField.get());
25    }
26 }
```

Fig. 4.13: INLINE DELTA pattern in a subclass.

```
1 class IndexAnchor extends IndexedStoreObject{

3    ...
4    public IndexAnchor(Field f,
5                       ObjectStore store,
6                       ObjectAddress address)
7          throws ObjectStoreException{
8       super(f, store, address);
9       setFields(f);
10      numberOfEntries =
11         numberOfEntriesField.getInt();
12      rootNodeAddress =
13         new ObjectAddress(rootNodeAddressField.get());
14   }

16   // extractValues() is removed from the class interface
17 }
```

Fig. 4.14: Eliminate polymorphic calls with INLINE DELTA.

this type of polymorphic call is simply to inline the section regarding the subclass into the subclass constructor, and thus avoid overriding the base class version of it. This pattern is quite common among the classes in our manually inspected collections, and it may be easily eliminated using automatic tools. One example is Eclipse class `StoredObject`, and the method `extractValues` invoked from its constructor, as depicted in Line 19 of Figure 4.12.

Class `IndexAnchor` is one of several subclasses of `StoredObject` in Eclipse. Figure 4.13 displays the overridden version of `extractValues` in Line 17. The method starts, in Line 19, by invoking the original version of `extractValues`, and continues with initializing the fields of the subclass (lines 20–24). Note that this method is not part of the class's external interface, as it is limited to **`protected`** access level, hence it is safe to eliminate the polymorphic behavior by removing method `extractValues` from subclass `IndexAnchor`, and inlining the rest of the initialization work in its constructor. This solution is depicted in Figure 4.14, where the delta of subclass fields is initialized in lines 9–13 of the constructor.

Table 4.1 depicts the prevalence of the various patterns found in our manual inspection of JRE and Eclipse projects.

The most common pattern is also the simplest— CONSTANT INITIALIZER, appearing in over 40% of the cases in both JRE and Eclipse. The next most common pattern is the FUNCTION OBJECT, which allows for a delayed execution of computation inside the base constructor through an object that was passed by the derived class. This pattern was found in 21.78% of the JRE cases, but only 10.85% of the Eclipse cases. The rest of the patterns are less prevalent, and used to resolve a smaller number of specific cases.

Collection	**JRE**	**Eclipse**	**Total**
Constant-initializer	82 (40.59%)	110 (42.64%)	192 (41.74%)
Function-object	44 (21.78%)	28 (10.85%)	72 (15.65%)
Inline-delta	19 (9.41%)	59 (22.87%)	78 (16.96%)
Native	14 (6.93%)	0 (0.00%)	14 (3.04%)
Unresolved	14 (6.93%)	23 (8.91%)	37 (8.04%)
Non-constant-initializer	9 (4.46%)	5 (1.94%)	14 (3.04%)
Code-rewrite	9 (4.46%)	1 (0.39%)	10 (2.17%)
Semi-constant-initializer	5 (2.48%)	13 (5.04%)	18 (3.91%)
Redundant	5 (2.48%)	8 (3.10%)	13 (2.83%)
Multi-function-object	1 (0.50%)	1 (0.39%)	2 (0.43%)
Initializer-object	0 (0.00%)	5 (1.94%)	5 (1.09%)
Factory	0 (0.00%)	5 (1.94%)	5 (1.09%)

Tab. 4.1: Summary of de-virtualization design patterns in JRE and Eclipse

Note that about 85% of the polymorphic behavior examples fit one of the 8 patterns above. The remaining 15% are also included in Table 4.1, and are found in one of the 4 following cases:

1. A "code rewrite" solution applies for cases where the super constructor invokes a public method that is part of the class interface. In such cases, the derived class overrides the original implementation, but in fact, when invoked through the constructor of the base class, only the original implementation is executed. For example, take class `JDialog` from the JRE's javax.swing package. Its method `setLayout()` is invoked (indirectly) through the constructor of class `Window` (from java.awt package). The implementation of the overridden version of `setLayout()` is depicted in Figure 4.15.

This implementation queries a boolean data member in Line 9. This boolean is initialized to **false**, and is set only through the constructor of `JDialog`. As a result, when `JDialog::setLayout()` is invoked through the super constructor, the value of the boolean data member will always be **false**, and so only the super version of `setLayout()` is executed (Line 12).

The suggested code rewrite solution is done on the base class `Window`. An alternative to the invocation of `setLayout()` from the constructor of `Window` would be to use a private method which contains the complete implementation of the original `setLayout()`. Then, this private method may be invoked from both the public `setLayout()` and from the constructor of `Window`.

```
1 class JDialog extends Dialog{
2 protected boolean rootPaneCheckingEnabled = false;

4 protected boolean isRootPaneCheckingEnabled() {
5       return rootPaneCheckingEnabled;
6 }

8 public void setLayout(LayoutManager manager) {
9       if(isRootPaneCheckingEnabled())
10         getContentPane().setLayout(manager);
11      else
12         super.setLayout(manager);
13    }
14 }
```

Fig. 4.15: Eliminate polymorphic behavior by rewriting the code.

2. A "native" case describes a case where the base class invokes an abstract method whose implementation in a derived class is declared **native**. Since in these cases we have no access to the native code, we could not analyze it. This case was encountered only in JRE and appeared in nine concrete subclasses of `WComponentPeer` in sun.awt package (where the method create() is **native**), and in the superclass of `WCustomCursor` from the same package (by invoking `createNativeCursor()`).

3. The "unresolved" cases are those where the overriding method contains a complex series of actions that are also very different from the original base

implementation. We marked 6.9% of the falls identified for JRE as "unresolved", and less than 9% in Eclipse.

4. The "redundant" cases are those in which the call to the polymorphic method from the base constructor code is in fact redundant, as all the overridden versions of this method have a trivial implementation of either invoking the **`super`** version only, or initializing the derived class's data members with zero values, as done by the compiler itself. For example, the constructor of class `AbstractConfigurationBlockPreferencePage` from Eclipse's JDT package invokes method `setDescription()` which is **`abstract`** in the base class. However, the class's 2 derived class in Eclipse both implement `setDescription()` as an empty function. As a result, the method invocation from within the base constructor is redundant.

Chapter 5

Immodest Constructors

Coding and maintenance is complicated when a constructor refines a polymorphic constructor, since in such a class, methods may be executed before any of its own constructors started executing. Our search for polymorphic behavior during construction in Chapter 3 was restricted to chains of direct messages sent to the created object. But, such half-baked objects can also be encountered through aliasing—an exposed reference can be used to invoke dynamically bound methods on a half-baked object.

This chapter describes the results of our search for constructors which expose the `this`-identity, denoted *immodest* constructors. Note that exposing `this` from a constructor is common with two-way associations, however, our search for immodest constructors is focused on cases where the exposed `this` identity is altered during inheritance. In this case, the external class unexpectedly receives a reference to the derived class object, rather than receiving a reference to the base class.

5.1 Definitions

Section 3.1 defined three varieties of polymorphic behavior during construction. The three kinds of exposition defined here are similar in nature.

Immodest Constructors. A constructor is *immodest* if (i) it is refined by a constructor of a derived class, *and* (ii) it exposes `this` to external code, e.g., by invoking an external method and passing `this` to it as an argument. A constructor is modest if it is not immodest.

Immodest Fall Constructors. We say that a constructor is an *immodest fall* if it refines a constructor of a base class which exposes (to code outside the class) a reference to the constructed object.

Immodest Pitfall Constructors. We say that a constructor is an *immodest pitfall* if it exposes the **this** identity. Consider for example the JAVA class `Frame` depicted in Figure 5.1 (drawn from the `java.awt` package). Then, both constructors of

```
1 public class Frame {
2    public void init(String title,
3                     GraphicsConfiguration gc){
4       this.title = title;
5       SunToolkit.checkAndSetPolicy(this, false);
6    }
7    public Frame(String title) throws HeadlessException{
8       init(title, null);
9    }
10   public Frame() throws HeadlessException{
11      this("");
12   }
13 }
```

Fig. 5.1: Immodest constructor in JAVA.

this class are immodest pitfalls: The first since it invokes method `init` which exposes the **this** pointer to an external class. The second constructor is also such a pitfall since it delegates its construction task to the first constructor. Observe however that both constructors are also polymorphic pitfalls, because they invoke the **public** method `init()` that me be overridden in derived classes.

A constructor that refines an immodest pitfall constructor of a base class is necessarily unsafe. To understand why, consider again Figure 5.1 and a subclass of `Frame`. If this subclass does not override function `init`, then all of its constructors are immodest falls since they necessarily refine one of `Frame`'s constructors which exposes the **this** identity. If however, the said subclass overrides `init`, then all of its constructors are by definition polymorphic falls (which would also make `Frame`'s constructors polymorphic).

Note that the above reasoning also shows that there are constructors which are *both* polymorphic and immodest. Since the overlap was small, we chose to categorize all such cases as being polymorphic.

5.2 Method

What is known in the JTL jargon as *pedestrian predicates* [9] were used to identify cases in which constructors invoke, directly or indirectly, polymorphic member functions. A more sophisticated analysis involving dataflow analysis (using scratches as they are called in JTL), was used to identify cases in which constructors allow external code, i.e., code which is not part of the ancestors chain of a class, to access a half-baked object.

Our conservative search for incidents of immodesty used inexact yet conservative interprocedural analysis starting at the base constructor and exact intraprocedural dataflow analysis. Searching for immodest pitfalls, our query is defined to find all cases where `this` is passed as an argument to any method invoked (directly or indirectly) from the constructor. These cases include method invocations where `this` is the hidden parameter, or the target of the method invocation, which mean that the internal object is not exposed to an external class. In order to eliminate this type of false alarms, we chose to cross the results with all cases where a constructor passes any parameter to a method that belongs to a different class.

The analysis was complemented by a laborious manual inspection of the violating code.

5.3 Findings

Table 5.1 shows the prevalence of immodest behavior in constructors in each of the collections in the software corpus.

Examining the second column of the table we see that there is a great variance in the prevalence of immodest constructors, ranging from 0% to 9%; even the median (2.42%) is very different from the average prevalence (5.94%). Comparing this average with the average prevalence of polymorphic constructors (2.55% see Table 3.1) we see that there are more than twice as many immodest constructors as there are polymorphic constructors.

These two phenomena occur also in the third column of the table: the prevalence of immodest fall constructors is large (from less than 2% to almost 15%), and their total number is greater than the number of polymorphic fall constructors by a factor greater than 4.

Interestingly, the prevalence of constructors with immodest behavior when compared to the entire constructors population is still small and is equal to about 1.06%. The prevalence of immodest pitfalls constructors is not quite as small: 5.64%.

Collection	Immodest Constructors		Immodest Fall Constructors		Immodest Pitfall Constructors	
JBOSS	129	(3.88%)	718	(3.25%)	957	(4.33%)
JRE	283	(7.90%)	1,111	(5.45%)	1,178	(5.78%)
ECLIPSE	186	(8.88%)	906	(5.72%)	1,704	(10.76%)
POSEIDON	215	(8.95%)	1,384	(12.49%)	1,351	(12.20%)
TOMCAT	10	(1.00%)	109	(2.10%)	81	(1.56%)
SCALA	53	(7.81%)	298	(9.48%)	402	(12.79%)
JML	15	(1.90%)	90	(3.06%)	183	(6.23%)
ANT	1	(0.23%)	18	(0.89%)	30	(1.49%)
MJC	6	(1.40%)	56	(3.90%)	130	(9.05%)
JEDIT	7	(2.42%)	131	(14.64%)	189	(21.12%)
ESC	3	(1.38%)	13	(1.82%)	21	(2.95%)
KOA	0	(0.00%)	4	(10.53%)	4	(10.53%)
Total	908	(5.94%)	4,838	(5.64%)	6,230	(7.26%)
Median	10	(2.16%)	109	(4.67%)	183	(7.64%)

Tab. 5.1: Prevalence of immodest behavior in constructors (conservative analysis).

Perhaps surprisingly, in examining the fourth column we find the number of *immodest pitfall constructors* is smaller (!) than the number of *polymorphic pitfall constructors*. But, the variety in this column is even greater than in the other columns (from less than 1.5% to more than 21%).

Table 5.2 shows the prevalence of classes with constructors with immodest behavior.

The data in this table can be summarized as follows: The phenomena we found for immodest constructors in Table 5.1, including great variety in the prevalence of the various types of immodest behavior, and a higher incidence rate in these than in their polymorphic counterpart, are similarly noticeable for immodest classes in Table 5.2.

The comparison of the findings regarding constructors and the findings regarding classes indicates that an immodest constructor is "responsible" on average for almost six actual immodest falls, and that every class with an immodest constructor has on average almost 9 classes with immodest pitfall constructors. This indicates that immodest constructors tend to be grouped together in a smaller number of classes.

Collection	Classes with Immodest Constructors		Classes with Immodest Fall Constructors		Classes with Immodest Pitfall Constructors	
JBOSS	61	(3.07%)	582	(3.69%)	650	(4.12%)
JRE	100	(4.52%)	894	(6.12%)	649	(4.44%)
ECLIPSE	117	(7.32%)	821	(5.77%)	1,374	(9.65%)
POSEIDON	115	(7.94%)	1,146	(13.19%)	1,018	(11.72%)
TOMCAT	7	(1.14%)	88	(2.34%)	61	(1.62%)
SCALA	43	(9.98%)	274	(9.95%)	312	(11.33%)
JML	9	(1.95%)	82	(3.86%)	98	(4.61%)
ANT	1	(0.37%)	15	(0.93%)	23	(1.43%)
MJC	4	(1.89%)	51	(4.98%)	74	(7.22%)
JEDIT	3	(2.68%)	123	(15.85%)	152	(19.59%)
ESC	1	(0.73%)	12	(1.90%)	15	(2.37%)
KOA	0	(0.00%)	4	(11.11%)	4	(11.11%)
Total	461	(4.85%)	4,092	(6.20%)	4,430	(6.71%)
Median	8	(2.68%)	105	(5.77%)	125	(7.22%)

Tab. 5.2: Prevalence of classes with constructors with immodest behavior (conservative analysis)

5.3.1 Manual Analysis

The problem with immodest constructors is realized when the reference to the half-baked object is used to invoke a virtual method that is overridden in the refining subclass. Detecting these cases required a tedious inspection of the code, including covering all the methods to which `this` is exposed, and checking the usage of this object through a dataflow analysis that considers possible method overriding in the external class.

Since this examination is extremely laborious, we decided to conduct it only on a sample of 35 classes from the JRE collection that were detected by our query as immodest classes. We randomly selected 35 numbers from the range of 1–894, to cover the range of all JRE classes with immodest fall constructors.

The items below summarize our findings:

1. The selected 35 classes with immodest fall constructors are subclasses of only 25 base classes.

2. Only 29 of the 35 cases are in fact exposing the `this` identity to an external class. The remaining cases are counted among the false-positives that are a result of our conservative automatic search.

An example for such a false-positive is a case where the implicit creation of an inner class is wrongly identified as exposing **this** to an external class. This is demonstrated in JAVA's class `Component` (from the `awt` graphics package), that is caught by our query because: (a) it contains an inner class (whose creation used the **this** parameter) and (b) its constructor invokes an external method.

3. In 11 out of the 29 cases of **this**-exposing to an external class, no polymorphic methods are invoked on the half-baked object. Among these cases we count: (a) cases where no method was invoked on the externalized reference (for example, in class `BranchInstruction` from `apache.bcel` package the reference is only added to or removed from a `HashSet`); and (b) cases where the method invoked on the externalized reference is non-virtual, i.e., it cannot be overridden in a subclass. For example, class `Thread` externalizes **this** through a call to `SecurityManager.checkAccess()` from its constructor. However, the exposed reference is only used with a call to `Thread`'s **final** methods.

4. Out of the remaining 18 cases, where polymorphic methods are invoked on the exposed **this** reference, we found that in 3 cases the only polymorphic method called was JAVA's `getClass()`, which is automatically defined per class, returning its type. We refer to this case as a special case, since this method is indeed dynamically bound, but it does not access any of the half-baked object's fields, hence it behaves like a static method. This type of methods, which we denote *semi static* methods, is a possible solution for the binding problem inside constructors: it preserves type safety (through dynamic binding), as well as any properties set on the object's data members (since it cannot access them).

   The examples we found for this type of behavior in our sample were subclasses of class `UnicastRemoteObject` from JRE's remoting library.

5. Another 12 of the cases where a polymorphic method was invoked in the external code context on the exposed and partially constructed **this** were cases where either the polymorphic method was not at all overridden in subclasses, or not overridden in the subclass that was selected in our manual sampling.

   This type of behavior was found in our sample in classes from JAVA's graphics libraries `awt` and `swing`. The constructor of class `Frame`, for exam-

ple, exposes **this** to `SunToolkit.checkAndSetPolicy()`, where the virtual method `setFocusTraversalPolicy()` is called on the uninitialized reference. However, none of `Frame`'s subclasses in JRE overrides this method, and hence the polymorphic behavior is not realized. Another example is class `JComponent`. This class is the base class of over 100 classes in the JRE, and 4 of which were randomly selected in our sample. The constructor of `JComponent` exposes the **this** reference to `LookAndFeel.installProperty()`, which in turn invokes the method `firePropertyChange()` on the passed reference. Only 3 subclasses of `JComponent` override this method, and none of them was among the classes selected in our sample.

6. In 3 cases detected in our manual inspection, we found a large number of polymorphic methods invoked on the exposed **this** reference from external classes, and additionally we found refining versions of both the methods invoked on **this**, and of the external methods. These cases were found in subclasses of `JButton` and `JLabel` from the `swing` package and in subclass of `PropertyInfoImpl` from package `com.sun.xml`.

Chapter 6

Conclusions and Further Research

Our main conclusion is that polymorphic behavior in constructors is scarce, occurring in about 1.4% of all classes and in about 1.4% of all constructors. The base constructors and base classes responsible for this behavior are even scarcer; their prevalence is less than 0.5%. This prevalence is in interesting contrast with the fact that the potential for such a polymorphic behavior occurs with at least 8% prevalence, *and* the fact that classes with potentially polymorphic behavior in their constructors tend to have (with statistical significance greater than 99%) more descendants.

In light of these results, we believe language designers should consider forbidding calls to methods which may be overridden in subclasses to be done from within the constructor. The variety of ways of doing that are discussed briefly in the opening chapter of this thesis.

An inspection of the patterns of polymorphic behavior in constructors and of their coverage rate suggests that such semantics may be even feasible in existing languages with extensive code base modifications (probably as a compiler option or extension).

We also believe that the time may be ripe for introducing what we call *Semi-Static Methods*, i.e., functions which are *dynamically* bound, but are allowed to invoke only **`static`** and semi-static methods of the class. Semi static methods are prohibited from accessing instance methods and variables, which means that they cannot change the internal state of the object. Recall that the issues with dynamic binding, as presented in Section 1.1 were a result of field access in methods invoked from the constructor. A familiar example is JAVA's `getClass()` method, which returns the **`class`** of the object, without accessing its internal state. Such a feature is useful in constructors, as demonstrated by the construction patterns that

can be more readily implemented using this feature. For example, in Chapter 4 we discussed the CONSTANT INITIALIZER pattern, where the refining versions of the method invoked from the base constructor returned a **`static`** constant value. We suggested an easy conversion of the code for eliminating of polymorphic behavior, but in fact these methods are semi-static by our definition, and therefore may be used safely.

We also discussed alternatives for enforcing modest behavior on constructors (so to speak), a prime alternative being a model similar to Bokowski and Vitek's confined types [4]. Unfortunately, we were unable to collect sufficient data to support the introduction of these into current languages. Automatic analysis proved to be time consuming while manual inspection is probably unreasonable, due to the difficulties in doing a full dataflow OO analysis. Still, enforcing such a behavior may be a good idea for new languages.

We should indicate that our partial manual inspection, as well as the summarized data, suggests that the number of classes responsible for such exposures is rather small.

Also, based on a manual exploratory findings of immodest behavior we conjecture that in the majority of such cases, the external code does not send any messages to the revealed reference, and if such messages are sent, they are rarely overridden in any of the derived classes. If this conjecture is verified, and the incidents are found to be of sufficient importance then perhaps the time is ripe for introducing a second pass initialization phase. The first initialization phase can remain a constructor with semi-static binding semantics, whereas the second phase will allow additional actions, probably non trivial and more complex in nature, that are required to be performed before the object is to be used by its clients.

Appendix A
Automatic Analysis Project Sources

The automatic analysis project source code is available for download and usage from the homepage of the Systems and Software Development Laboratory of the Computer Science department in the Technion.

Main page location is http://ssdl-wiki.cs.technion.ac.il/wiki/index.php. Section *Devirtualizing Constructors* contains information regarding this project download and usage options, as well as instructions for possible future extensions.

Bibliography

[1] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, Massachusetts, 1996.

[2] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of Java software. In Tarr and Cook [28].

[3] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In J. M. Vlissides and D. C. Schmidt, editors, *Proc. of the 19th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'04)*, pages 35–49, Vancouver, BC, Canada, Oct. 2004. ACM SIGPLAN Notices 39 (10).

[4] B. Bokowski and J. Vitek. Confined types. In *Proc. of the 14th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'99)*, pages 82–96, Denver, Colorado, Nov.1–5 1999. ACM Press, ACM SIGPLAN Notices 34 (10).

[5] B. Cabral and P. Marques. Exception handling: A field study in Java and .NET. In Ernst [11], pages 151–175.

[6] P. Chalin and P. R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In Ernst [11], pages 227–247.

[7] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Prog. Lang. Syst.*, 28(3), May 2006.

[8] T. Cohen and J. Gil. Self-calibration of metrics of Java methods. In *Proc. of the 37th Int. Conf. on Technology of OO Lang. and Sys. (TOOLS'00 Pacific)*, pages 94–106, Sydney, Australia, Nov. 20-23 2000. Prentice-Hall.

[9] T. Cohen, J. Y. Gil, and I. Maman. JTL—the Java tools language. In Tarr and Cook [28].

[10] N. Eckel and J. Gil. Empirical study of object-layout strategies and optimization techniques. In E. Bertino, editor, *Proc. of the 14th Euro. Conf. on OO Prog. (ECOOP'00)*, volume 1850 of *LNCS*, pages 394–421, Sophia Antipolis and Cannes, France, June 12–16 2000. Springer.

[11] E. Ernst, editor. *Proc. of the 21st Euro. Conf. on OO Prog. (ECOOP'07)*, volume 4609 of *LNCS*, Berlin, Germany, July/Aug. 2007. Springer.

[12] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In R. Crocker and G. L. S. Jr., editors, *Proc. of the 18th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'03)*, pages 302–312, Anaheim, California, USA, Oct. 2003. ACM SIGPLAN Notices 38 (11).

[13] J. Gil and A. Itai. The complexity of type analysis of object oriented programs. In E. Jul, editor, *Proc. of the 12th Euro. Conf. on OO Prog. (ECOOP'98)*, volume 1445 of *LNCS*, pages 601–634, Brussels, Belgium, July 20–24 1998. Springer.

[14] J. Gil and I. Maman. Micro patterns in Java code. In Johnson and Gabriel [17], pages 97–116.

[15] A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C# Programming Language*. Addison-Wesley, Reading, Massachusetts, 2nd edition, Oct. 2003.

[16] ISE. *ISE EIFFEL The Language Reference*. ISE, Santa Barbara, CA, 1997.

[17] R. Johnson and R. P. Gabriel, editors. *Proc. of the 20th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'05)*, San Diego, California, Oct. 2005. ACM SIGPLAN Notices.

[18] G. Kniesel and D. Theisen. JAC access right based encapsulation for Java. *Softw. Pract. Exper.*, 31(6):555–576, 2001.

[19] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, Mar. 2006.

[20] C. Male and D. J. Pearce. Nonnull type inference with type aliasing for Java. Technical report, Computer Science, Victoria University of Wellington, NZ, August 2007.

[21] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. In *Annals of Mathematical Statistics*, volume 18, pages 50–60, 1947.

[22] H. Melton and E. Tempero. Static members and cycles in Java software. *International Symposium on Empirical Software Engineering and Measurement*, 0:136–145, 2007.

[23] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, New Jersy, 2^{nd} edition, 1997.

[24] J. Noble and D. Lea. Editorial: Aliasing in object-oriented systems. *Soft. Practice & Experience*, 31(6):505, 2001.

[25] 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 IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[26] R. M. Stallman. *Using the* GNU *Compiler Collection (*GCC*):* GCC *Version 4.1.0*. Free Software Foundation, 2005.

[27] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 3^{rd} edition, 1997.

[28] P. L. Tarr and W. R. Cook, editors. *Proc. of the 21^{st} Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'06)*, Portland, Oregon, Oct.22-26 2006. ACM SIGPLAN Notices.

[29] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In Johnson and Gabriel [17].

[30] T. Wrigstad. *Ownership-Based Alias Managemant*[1]. PhD thesis, KTH, Computer and Systems Sciences, May 2006.

[31] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieżun, and M. D. Ernst. Object and reference immutability using Java generics. In *ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, September 5–7, 2007.

[1] http://www.diva-portal.org/kth/theses/abstract.xsql?dbid=3956

האובייקט עליו היא מופעלת הינו "חצי-אפוי". תוצאות הניתוח מצביעות על מספר רב יחסית של פונקציות בנייה חושפניות, אולם ניתן לייחס אותן למספר קטן יחסית של פונקציות בנייה במחלקות בסיס. בנוסף, אנו מעריכים שנתוני השכיחות של מצבים אלו פחות מדויקים מפאת הסיבוכיות הגבוהה שבניתוח יחסי גומלין בין פרוצדורות שונות, וכן בשל השימוש בשאילתות מחמירות, אשר גורמות לזיהוי שגוי של מקרים בהם למעשה לא הייתה חשיפה של האובייקט העצמי. הקפדנו להשתמש בשאילתות מסוג זה על מנת שלא נחמיץ מקרים כלליים וחמקמקים יותר של ההתנהגות ה"חושפנית" אותה חיפשנו. על החיפוש האוטומטי הוספנו בדיקה ידנית מדגמית, אשר הראתה כי שגיאת השאילתה המחמירה הייתה נמוכה יחסית. בדיקה זו גילתה בנוסף, כי בחלק מהמקרים שאותרו, האובייקט החיצוני למחלקה לא הפעיל מתודות פולימורפיות על האובייקט הפנימי שהועבר אליו, אלא ביצע פעולות פשוטות יחסית כמו הוספה לרשימה (שבה כל האובייקטים הם מטיפוס הבסיס של Object – JAVA). עם זאת, נתגלו גם מקרים שבהם היו הפעלות של מתודות פולימורפיות, ולעיתים אף כאלו אשר הוגדרו מחדש בחלק מהמחלקות יורשות.

תוצאות אלו הביאו אותנו למסקנה כי ניתן להימנע משימוש בקישור הדינאמי של מתודות מתוך פונקציית בנייה של מחלקת בסיס, ובכך להתגבר על החסרונות של שתי גישות הקישור הקיימות היום בשפות התכנות השונות. על פי ניתוח הקוד באוסף התוכנה הגדול שבחנו, ניתן לראות כי תופעת השימוש בקריאות פולימורפיות ו"חושפניות" מתוך פונקציות בנייה איננה נפוצה, וכן היא ניתנת להמרה לקוד שקול אשר יעלים לחלוטין את הפעלת המתודות הפולימורפיות בעת יצירת האובייקט. כמו כן, אנו סבורים שתכן עבור שפת תכנות חדשה ירוויח רבות מהתייחסות הולמת לבעיה, למשל באמצעות הגדרה מיוחדת של מתודות חצי-סטטיות, אשר יקושרו באופן דינאמי לגרסה שהוגדרה עבורן במחלקה היורשת, אולם יהיו מוגבלות בפעילותן בדומה לפונקציות מחלקה סטטיות, אשר אינן מותרות בגישה למשתני מופע במחלקה או למתודות שאינן סטטיות.

הוא שייך נוצר במלואו, ולא ברור כיצד יוכלו לגשת לשדה זה מתוך מתודות שנקראו מתוך פונקציית הבנייה.

תיאור העבודה

מחקר זה בוחן אמפירית את השימוש הנפוץ כיום בקישור דינאמי של מתודות מתוך פונקציות בנייה של מחלקות, כפי שמופיע כיום במערכות JAVA מרכזיות בתחומי יישום מגוונים. השתמשנו בקובץ של 12 אוספי תוכנה, המונה יותר מ-65,000 מחלקות, אשר הופיעו במחקרי תיכון תוכנה בשנה האחרונה, וביצענו ניתוח של התנהגות פונקציות הבנייה באמצעות שאילתות אוטומטיות בשפת JTL. שאילתות אלו יועדו לאיתור שני סוגי התנהגות של מתודות הנקראות מתוך פונקציות בנייה ואשר עלולות לגרום לתופעות הבעייתיות שתוארו לעיל:

פונקציית בנייה פולימורפית – זוהי פונקציית בנייה אשר מתוכה מתבצעת קריאה למתודה פולימורפית, כלומר פונקציה וירטואלית אשר יש לה גרסה חדשה במחלקה יורשת. בחרנו לבחון בנוסף גם את הפוטנציאל להתנהגות פולימורפית כזו באמצעות מציאת פונקציות בנייה אשר קוראות למתודה אשר קיימת אפשרות לדריסתה במחלקה יורשת, אך אפשרות זו לא בהכרח מומשה. זווית התבוננות נוספת שבחרנו ללמוד היא מכוונה של המחלקה היורשת עצמה, אשר גורמת למימוש הפוטנציאל להתנהגות פולימורפית של פונקציית הבנייה במחלקת הבסיס באמצעות הגדרה מחדש של מתודה אשר נקראת מתוך פונקציית בנייה זו. תוצאות הניתוח על אוסף התוכנה מעידות כי למרות שהפוטנציאל למצב הבעייתי הנ"ל אינו זניח (שכיחות של יותר מ-8%), כלומר קיימות פונקציות בנייה רבות אשר מפעילות מתודות שעלולות להיות מוגדרות מחדש במחלקות היורשות, פוטנציאל זה כמעט ואינו ממומש. מצאנו שהתנהגות זו מופיעה בפחות מ-1.5% מפונקציות הבנייה, המרחיבות פחות מ-0.5% מכלל פונקציות הבנייה. המשכנו לחקור את פונקציות הבנייה הפולימורפיות בניסיון למפות תבניות אופייניות לשימוש בהן, ואכן גילינו ש-80% מהמקרים הללו מתחלקים ל-8 תבניות שניתן להמירן בקלות יחסית לקוד שקול הנמנע מגישה מוקדמת למתודות.

פונקציית בנייה חושפנית - מצב נוסף שרצוי להימנע ממנו הוא כאשר פונקציית הבנייה חושפת את האובייקט שטרם נוצר לקוד חיצוני אשר ניגש למתודות במחלקה היורשת. באופן דומה להגדרה הקודמת, בחנו את הבעיה ממספר זוויות: ראשית, הגדרנו את פונקציות הבנייה החושפניות כאלו אשר מעבירות מצביע לאובייקט הפנימי בקריאה לפונקציה של אובייקט חיצוני למחלקה, ובנוסף מדובר במחלקת בסיס אשר פונקציית הבנייה שלה מורחבת ע"י פונקציית בנייה של מחלקה יורשת בתוך היררכית המחלקות של אוסף התוכנה. שנית, הגדרנו פונקציות בנייה עם פוטנציאל להתנהגות חושפנית ככאלו אשר חושפות את האובייקט הפנימי ע"י העברתו כפרמטר לפונקציה של אובייקט חיצוני למחלקה. ובנוסף, בחנו גם את המחלקות היורשות ופונקציות הבנייה שלהן, אשר מרחיבות פונקציית בנייה חושפנית במחלקת הבסיס. למעשה, הבעייתיות האמיתית נעוצה בפעולה שמתבצעת על האובייקט שנחשף מתוך פונקציית הבנייה שלו. פעולה זו עשויה להיות קריאה למתודה פולימורפית, כלומר עבור מחלקת בסיס, תיקרא מתודה אשר הוגדרה עבורה גרסה חדשה במחלקה יורשת. הפעלת מתודה פולימורפית במחלקה החיצונית עשויה לגרור תוצאות מפתיעות, שכן

תקציר

הסמנטיקה הרווחת כיום ברוב שפות התכנות מונחות העצמים היא שפונקציית הבנייה של מחלקה היורשת ממחלקה אחרת הינה הרחבה של פונקציית בנייה במחלקת הבסיס. יצירת אובייקט חדש מטיפוס המחלקה היורשת מתחיל ביצירת אובייקט הבסיס, באמצעות קריאה לפונקציית הבנייה של מחלקת הבסיס. פונקציה זו עשויה לגרום להפעלתן של מתודות בעלות קישור דינאמי אשר הוגדרו מחדש במחלקה היורשת. כתוצאה מכך עלול להיווצר מצב של סמנטיקה מבלבלת וקשר הדוק ומסובך בין מחלקת הבסיס למחלקה היורשת. נשאלת השאלה האם קישור המתודות הנקראות במהלך ביצוע פונקציית הבנייה של אובייקט הבסיס עדיף שיהיה סטטי או דינאמי.

קישור סטטי מול קישור דינאמי

המצב כיום הינו שחלק משפות התכנות , למשל C++, בחרו בקישור הסטטי, כלומר קריאה למתודות מתוך פונקציית הבנייה של מחלקת הבסיס תפעיל את גרסת הבסיס שלהן. הקישור הסטטי לכאורה דואג לעקביות בין זהות האובייקט הנבנה לבין המתודות המופעלות עליו, שכן הם כולם שייכים לאותה מחלקת בסיס. אולם גישה סטטית זו עשויה להיות בעייתית. כך למשל במקרה שבו פונקציית הבנייה במחלקת הבסיס קוראת למתודה אחרת, אשר הגרסה שלה במחלקת הבסיס הינה אבסטרקטית, כלומר לא קיים עבורה מימוש. הפעלת מתודה זו תגרום לשגיאה בזמן ריצה. קומפיילרים מתוחכמים מסוגלים לסייע בהימנעות ממצבי מלכודת כאלו, אולם יכולת זו הינה מוגבלת ביותר, ואינה מסוגלת לזהות קריאה עקיפה למתודה אבסטרקטית מתוך פונקציית בנייה של מחלקת בסיס.

מן הצד השני, שפות תכנות אחרות, לדוגמת JAVA, בחרו בגישה של קישור דינאמי מתוך פונקציית הבנייה של מחלקת הבסיס. על פי גישת הקישור הדינאמי, גרסת המתודה שתופעל מתוך פונקציית בנייה של אובייקט הבסיס תהא זו שהוגדרה מחדש במחלקה היורשת. גם גישה זו חושפת פוטנציאל בעייתי משום שבזמן ביצוע מתודות אלו אנו נמצאים בתחילת תהליך הבנייה של האובייקט מהמחלקה היורשת שאליו הן שייכות. אולם בשלב זה, האובייקט הקיים הינו עדיין "חצי אפוי", שכן ייתכן ושדותיו טרם אותחלו. כתוצאה מכך, מתודות מן המחלקה היורשת למעשה מטפלות באובייקט ממחלקת הבסיס במקום באובייקט מהטיפוס לו ציפו. בנוסף לכך, אפשרות למצב שכזה מקשה על פיתוח מנגנונים לביטוי תכן באמצעות שפת תכנות וכן על פיתוח כלים לבדיקתו. דוגמאות בולטות למנגנונים כאלו מופיעות במחקרים רבים בנושא תיכון תוכנה, למשל: סימון משתנים כ- NON-NULL (המתייחס למצביעים שערכם לעולם לא יהיה שווה לאפס), ציון שדות ומשתנים שניתנים לקריאה בלבד (המבטא את הכוונה שאלו לא יהיו ניתנים לשינוי לאחר שנוצרו באופן מלא) וכן הגדרת שמורות מחלקה (חלק ממתודולוגית התכן עפ"י חוזה - design by contract). הבעייתיות נובעת מכך ששדה שהוגדר לקריאה בלבד, לדוגמא, הופך בפועל לבלתי ניתן לשינוי רק לאחר שהאובייקט אליו

המחקר נעשה בהנחיית פרופ/ח יוסי גיל בפקולטה למדעי המחשב.

אני רוצה להביע את הוקרתי העמוקה לפרופ׳/ח יוסי גיל על הנחייתו המסורה לאורך הדרך.

אני רוצה להודות בנוסף לבני משפחתי וחבריי היקרים. עזרתם ותמיכתם נותנות לי כוחות בכל פעם מחדש.

עבודה זו מוקדשת לזכר אבי, דוד שרגאי ז״ל.

הערכה אמפירית של מידת השימושיות של פונקציות וירטואליות הנקראות מתוך פונקציות בנייה

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

טלי שרגאי

הוגש לסנט הטכניון - מכון טכנולוגי לישראל

חשוון תשס״ט חיפה נובמבר 2008

הערכה אמפירית של מידת השימושיות של פונקציות וירטואליות הנקראות מתוך פונקציות בנייה

טלי שרגאי