Are We Ready for a Safer Construction Environment?

Joseph (Yossi) Gil[1,2] and Tali Shragai[2]

[1] Google, Inc.
[2] The Technion

"*Unfortunately, the mainstream languages* C# *and* Java *give access to the object being constructed (through this) while construction is ongoing.*"
Fähndrich and Leino [12]

Abstract. The semantics of many OO languages dictates that the constructor of a derived class is a refining extension of one of the base classs constructors. As this base constructor runs, it may invoke dynamically bound methods which are overridden in the derived class. These invocations receive an "half baked object", i.e., an object whose derived class portion is uninitialized. Such a situation may lead to confusing semantics and to hidden coupling between the base and the derived. Dynamic binding within constructors also makes it difficult to enhance the programming language with advanced mechanisms for expressing design intent, such as *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 famous design by contract methodology). A read-only field for example becomes immutable only after the creation of the enclosing object is complete.

We investigate the current programming practice in JAVA of calling dynamically bound methods. 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, actual such dynamic binding is scarce, found in less than 1.5% of all constructors, inheriting from less than 0.5% of all constructors. Further, we find that over 80% of these incidents fall into eight "patterns", which can be relatively easily transformed into equivalent code which refrains from premature method invocation.

A similar predicament occurs when a constructor exposes the self identity to external code, which then invokes methods overridden in the derived class. Our estimate on the prevalence of this exposition is less accurate due to the complexity of interprocedural dataflow analysis. Although the estimate is high, there are indications that it arises from a relatively small number of base constructors.

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 which inherits from a class B. Then, in most OO languages the construction of a D-object is what we call a *refinement* of the construction of a B object, in that the body of any constructor of D is executed only after an explicit or implicit

S. Drossopoulou (Ed.): ECOOP 2009, LNCS 5653, pp. 495–519, 2009.

invocation of one of the constructors of B. [1] What is the status of the D object in the course of this invocation? 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 [1] 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++ [26], 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 [5, 6, 2, 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. We briefly describe here a *synthesis* of the approaches which addresses these limitations. But before this or any other new, competing proposal, can be considered, it must be evaluated against the common programming practice which this research tries to discover.

1.1 The Static vs. the Dynamic Binding Semantics within Constructors

Object creation can be divided into three conceptual stages: *(i)* memory allocation, *(ii)* preliminary field initialization, and *(iii)* establishing invariants. Allocation is often automatic, especially in languages with memory management. Preliminary initialization also depends on the language model (vacuous in C++, as opposed to default zero initialization in Java), and is not very interesting. What we are interested in here is the final stage, that of establishing invariants, which often involves some computation. This final stage is realized by the user-defined constructor. This section serves as a brief reminder of the distinction, in the context of constructors, between static- and dynamic-binding semantics and its consequences.

Somewhat paradoxically, the static binding approach of C++ may compromise static type safety, as demonstrated in Fig. 1. 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 realized (Line 7) in the inheriting concrete class `Circle`. Instantiating `Circle` here results 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 pure virtual function `Shape::draw`.[2] Clever compilers (GCC [25] 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 intractable [13].

[1] It could be the case that this constructor of B invokes yet another constructor of B, which may invoke yet another constructor B, or of a parent of B, in which case we say that the constructor of D refines all of these constructors.

[2] More precisely, the C++ semantics attributes this error to the attempt to dynamically invoke a pure virtual function, rather than to the fact that this function has no body; for various reasons, C++ allows defining pure-virtual functions with body, but the runtime error would have occurred *even* if `Shape::draw` had body!

The C++ design choice of static binding semantics within constructors is probably due to 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 Fig. 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 predefined state contradicts non-null promises, **`final`** guarantees, etc.

```
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. Pure virtual function call in C++

Dynamic binding in constructors means that methods may be called prematurely. When this happens, 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 invariant was not fully established. 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 other methods may be invoked any number of times may add to the complexity.

Fig. 2 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. Note that the last statement of the constructor of this class, calls function `initializeParser`, which as its name indicates, is in charge of initializing instance variable `parser`.

Consider now the implementation of the derived class `CodeSnippetCompiler`, as depicted in Fig. 3. We see in the figure (lines 3–6) that this class overrides function `initilializeParser`, specializing the `parser` field with a parser suitable for parsing code

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

Fig. 2. A base class invoking a polymorphic function

```
1 public class CodeSnippetCompiler extends Compiler {
2  public void initializeParser() {
3    this.parser = new CodeSnippetParser(
4       this.problemReporter, this.evaluationContext,
5       this.options.parseLiteralExpressionsAsConstants,
6       this.codeSnippetStart, this.codeSnippetEnd);
7    }
8  EvaluationContext evaluationContext;
9  int codeSnippetStart, codeSnippetEnd;

11 public CodeSnippetCompiler( ...initial arguments omitted for brevity ...
12    EvaluationContext evaluationContext,
13    int codeSnippetStart, int codeSnippetEnd
14 ) {
15    super(environment, policy, settings, requestor, problemFactory);
16    this.parser = new CodeSnippetParser(
17          this.problemReporter, evaluationContext,
18          this.options.parseLiteralExpressionsAsConstants,
19          codeSnippetStart, codeSnippetEnd);
20    this.parseThreshold = 1;
21   }
22 }
```

Fig. 3. A derived class overriding a function called from the base constructor

snippets. Three data members are passed to the constructor of of `CodeSnippetParser` in the overridden version `initilializeParser`. These are: `evaluationContext`, `codeSnippetStart` and `codeSnippetEnd` (defined in lines 8–9).

The constructor of this class starts by calling the base constructor in Line 15. This refined base constructor calls the overridden version of `initializeParser()`, but this function cannot complete its mission correctly, since the three data members it relies on belong to the derived class and could not have been initialized yet.

In fact, we see that the constructor of `CodeSnippetCompiler` repeats (lines 16–19) the body of function `initializeParser` (that is lines 3–6), 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 "bad smell" code in figures 2 and other bugs (e.g., a call to an abstract function to retrieve a member value—omitted from this excerpt) we found in our study show that the dynamic binding is confusing. The fact that JAVA forbids making a call to a member function when refining a base constructor or in delegating to another constructor of the same class[3] is also an indication that a call to an overridden function was not intended to be allowed.

But, beyond the confusing semantics, and arguably more importantly, the dynamic binding approach makes it difficult to introduce notions such as non-null [12, 6, 20], immutability (e.g.,JAVARI [3, 28] and JAC [18]) and class invariant [22, 19] guarantees into the language. Such guarantees are typically achieved by the constructor. But,

[3] that is, the code `class D extends B { D() {super(f()); }}` is illegal if `f` is a function member of either `B` or `D`

the possibility of methods being executed before the constructor even begun, makes it impossible to rely on these guarantees. This is the reason that much of this work introduces non-standard types and annotations to deal with half-baked objects, e.g., Fähndrich and Leino introduce `[Raw]` methods types [12] and Zibin and colleagues `@AssignsFields` annotations [30].

1.2 Hardhat Constructors and Destructors

Problems of this sort may occur not only when the constructor calls, directly or indirectly, methods overridden in the derived class. It could also be the case that the constructor reveals `this` to code external to the class, either by passing it as a parameter to an external function, or by storing it in an externally accessible field, making it possible to invoke overridden methods before construction is complete. Detecting cases of this sort could be difficult, especially in a multithreaded execution environment.

A similar problem occurs in C++ which imposes a refining semantics on destructors: a class destructor implicitly invokes the destructor of the parent class after its body completed execution. Runtime errors due to a call to a pure virtual function may thus occur in the course of a destructor's execution. The situation is exacerbated by the fact that destructors are typically called implicitly, e.g., as part of stack unrolling due to exception handling.

A natural and appealing resolution of the dilemma in choosing between the static and dynamic approaches is in a *synthesis* which *forbids* the processes of object creation and destruction from making any computation in which there is a difference between the two binding semantics. (An interesting alternative is offered by EIFFEL [16] in which the creation of a derived class does not involve a creation of a subobject of the base class.) We propose a language model enforcing constructors and destructors in which no polymorphic calls could be made, what we call *hardhat* execution. Thus, in this model, the premature call to `draw` in Fig. 1 is simply signalled by the compiler. The advantages should be clear:

1. *Type Safety.* The hardhat semantics avoids the type safety problem of the static binding approach.
2. *Reduced Coupling with Base Classes.* 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, or that the destructor of D has not finished its execution.). This reduces and simplifies the dynamic binding's typical coupling between the class and its base, and makes the analysis of multithreaded programs a bit easier.
3. *Crisp Boundary Between Initialization and Use.* Hardhat constructors are consistent with the OO thinking by which objects are created and only then used. The predicaments of a prematurely called method are avoided: a method should not be aware of the fact that it may be called from a constructor of a base class, and the analysis of multithreaded programs become
4. *Simplified Language Extensions.* With hardhat 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 paper is concerned mostly with the *cost* to be paid in introducing hardhat constructors into languages such as JAVA. Towards this end, we try to estimate the prevalence of constructors which deviate from the hardhat model in existing code, and to characterize the use of dynamic binding within constructors.

Our search for transgressing constructors in actual code relies on the following definition of hardhat execution:

> *A constructor is* hardhat *if it is both* monomorphic *(that is, it does not make any chain of of* `this` *method calls which raises the binding question) and* modest *(that is, it does not expose the* `this` *reference by storing it in a variable or passing it as an explicit parameter).*

The auxiliary notions of monomorphism and modesty are explained in greater detail below, but the intuition should be clear: The examples set in Fig. 1 and in Fig. 2 and Fig. 3 demonstrate cases of polymorphic behavior during construction. To see why we would like constructors to be "modest", consider for example the standard JAVA class `Thread`, depicted in part in Fig. 4.

```
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, long s) {
6       //...
7       setPriority(priority);
8       //...
9    }
10   public final void setPriority(int newPriority) {
11      checkAccess();
12      //...
13   }
14   public final void checkAccess() {
15      SecurityManager security = System.getSecurityManager();
16      if (security != null)
17         security.checkAccess(this);
18      //...
19   }
20   //...
21 }
```

Fig. 4. A constructor revealing a self reference

The no-arguments constructor invokes function `init`, which invokes `setPriority` which then invokes function `checkAccess`. This calls' chain poses no polymorphic construction risk, since all functions in the chain are either `final` or `private`. But, further inspection may be more difficult, since the runtime type of variable `security` is unknown: function `checkAccess()` delegates (Line 17) part of its mission to an external class through the `security.checkAccess(this)` call. 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 these may be surprised to find that their receiver is an incomplete object.

To make constructors hardhat, we need to make a concrete language definition forbidding both polymorphic calls and identity exposition 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 Bokowski and Vitek's constraints [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. The requirements are then that *(i)* `init` methods are called only by constructors and other `init` methods; *(ii)* constructors and `init` cannot call non-`init` methods; *(iii)* `init` methods cannot be overridden; and *(iv)* `init` are anonymous in the Bokowski-Vitek sense. Or, one may also consider replacing requirement *(iii)* by the demand that `init` methods "semi-static methods" (sometimes called *raw* in the literature), i.e., methods which are bound dynamically yet are not allowed to access neither `this` nor any non-`static` fields or methods. (Obviously, in languages with destructors, there should also be methods tagged as `destruct`, with similar requirements. But, for simplicity, we shall henceforth concentrate in constructors.)

There is also an alternative perspective in which constraints are placed only on constructors which are invoked by constructors of a derived class; this requires a mechanism for denoting a constructor as "final", meaning that it cannot be refined in derived classes. The language design space is further enriched by the many other variants for providing the means 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 [23] or a survey in [29]) 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; our primary focus is on 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, than in adding e.g., confined types which do not pertain to all code.

Accordingly, two hypotheses were initially set out for examination: *(i)* constructors which are not hardhat in actual code are rarities, and *(ii)* most of these can be easily made safe. Verification of the first conjecture should make the notion of hardhat constructors a candidate worthy of inclusion in new languages. Verification of the second should help encourage changes in the semantics of current programming languages. Alternatively, the understanding of actual use of non-hardhat constructors in code should help to evaluate the price of placing the hardhat requirement in a new language on the customers..

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

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 at least in binary form: *JRE 1.6.0_01* [4] (used in almost all empirical studies of JAVA, e.g., [14, 8, 21, 2]); although naturally, each such experiment uses a different version of the library); *JBoss 3.2.6* [5] (circa 1,000 packages of sources were not available); *Eclipse 3.0.1* [6] (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 1130 classes, we used the entire Eclipse implementation); *Poseidon 2.5.1 community edition* [7] (sources of were not available, binaries were apparently obfuscated by an automatic tool); *Tomcat 5.0.28* [8]; *Scala 1.3.0.4* Just like Poseidon, sources of the SCALA [24] distribution were largely unavailable, but this is because the compiler itself is written in SCALA; *JML 5.5* (a set of software tools used for the implementation of the JAVA Modeling Language [19]); *ANT 1.6.2*[9]; *MJC 1.3* (MultiJAVA is a JAVA language extension [7] which adds open classes and symmetric multiple dispatch to the language; MJC is multiJAVA the compiler); *JEdit 4.2*; *ESC 2.0b2* (the *E*xtended *S*tatic *C*hecker programming tool that tries to check some of JML assertions through static analysis); and *Koa* [10] (the Koa Tallying subsystem is a Dutch Internet voting application).

Tab. 2 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.

Table 1. Size statistics of the twelve collections in the corpus

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

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 3,000 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

[4] http://download.java.net/jdk6
[5] http://www.jboss.org
[6] http://www.eclipse.org
[7] http://www.gentleware.com
[8] http://jakarta.apache.org
[9] http://ant.apache.org
[10] http://sort.ucd.ie

have at least one constructor, since a default, no-arguments constructor is generated 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 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.

Tab. 2 shows how many base classes and how many "base constructors" were found in the collections in the software corpus. 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.

Table 2. Base classes and constructors in the corpus

	Internal		External		Total	
Collection	**Classes**	**Ctor's**	**Classes**	**Ctor's**	**Classes**	**Ctor's**
JBOSS	1,809	2,857	180	469	1,989	3,326
JRE	2,212	3,583	0	0	2,212	3,583
ECLIPSE	1,537	1,952	61	143	1,598	2,095
POSEIDON	1,140	1,714	308	689	1,448	2,403
TOMCAT	543	819	71	185	614	1,004
SCALA	350	428	81	251	431	679
JML	391	578	70	211	461	789
ANT	230	328	39	102	269	430
MJC	149	233	63	195	212	428
JEDIT	41	66	71	223	112	289
ESC	106	126	31	91	137	217
KOA	2	2	13	39	15	41
Total	8,510	12,686	988	2,598	9,498	15,284
Median	370.5	503	66.5	190	446	734

The three column groups in the table 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 may makes reasoning a bit more difficult.

As suggested by the table, 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 also relies on GUI and XML processing services of the JRE, inheriting from the appropriate classes. We see that in JEDIT the number of external bases is disproportionally large; in KOA, the number of external base constructors is much greater than internal base constructors. This however does not happen in other collections, and the relative number of external constructors and external bases is typically small, with median and median value of the relative number of external bases, both constructors and classes, is 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 are actually refined? Theoretically, the minimal number of classes with no descendants and unrefined constructors is one. In practice, it can be inferred from Tab. 2

hat about 15% 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 for non-hardhat 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 (as can be computed in Tab. 2). 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 (99.99% confidence level) higher in classes which serve as bases.

3 Polymorphic Constructors

Having described the data set, we turn now to the description of the research method and results. This section is devoted to the study of the prevalence of *polymorphic constructors*. We say that a constructor is polymorphic if it may execute differently due to overriding, that is if there is a chain of method calls with `this` as the receiver, starting at the constructor which leads to a call to an overridden method.

In the following section we explain how such polymorphic behavior may be eliminated. We exclude from our attention here and in the next section cases in which the call to an overridden method occurs as a result of assigning `this` to a variable or passing it as a parameter, and then using this variable or parameter as a receiver. This kind of non-hardhat behavior is the subject of Sec. 5.

3.1 Definitions

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

Polymorphic Pitfall Constructors. Recall that only one in seven classes have descendants, and that the majority of constructors are not refined at all. There are therefore many constructors that bear the potential for polymorphic behavior, but the polymorphic behavior may, or may not be manifested, depending on whether the enclosing class

has any derived classes, and whether any of these derived classes overrides any of the potentially polymorphic methods invoked by the constructor.

We say that a constructor of a certain class is a *polymorphic pitfall* if it calls, directly or indirectly, a method of its class and of an ancestor class which *might* be overridden in a derived class, i.e., a method which is non-`final`, non-`static` and non-`private`. Determining whether a constructor is a polymorphic pitfall, does not require whole-world analysis; only the class itself and its ancestors must be inspected. In the example of Fig. 1, the constructor `Shape::Shape()` is a polymorphic pitfall.

Polymorphic Falls Constructors. The definition of polymorphic constructors puts the "blame" on the polymorphic behavior on the refined constructor, which by definition must be a polymorphic pitfall. Still, even though the fault occurs at the refined constructor; 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 polymorphic pitfall constructor, in such a way that the refined constructor makes a method call chain in which a message, which is bound to different methods in the base and in the derived classes, is sent to `this`. Again, determining whether a constructor is a *polymorphic fall* can be decided by inspecting its enclosing class and all its bases.

The constructor `Shape::Shape()` in Fig. 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()` which calls method `draw` whose implementations in the base `Shape` and the derived `Circle` are different. The constructor of class `CodeSnippetCompiler` is likewise a polymorphic fall.

The case of `abstract` classes is somewhat special in that even if a constructor of such a class may demonstrate polymorphic behavior, we classify it as a pitfall, since this polymorphic behavior can only be realized if this constructor is refined.

With the above two definitions, we can give an alternative characterization of polymoprhic constructors: A *polymorphic constructor* is a polymorphic pitfall constructor for which we found one or more refining *polymorphic fall constructors*. Thus, the decision of whether a polymorphic fall constructor is indeed polymoprhic is relative to the code base.

3.2 Method

Our analysis was carried out first on the binary representation of the code, using the *J*ava *T*ools *L*anguage (JTL) [9]—a declarative language for code analysis. JTL itself is implemented on top of the *B*yte *C*ode *E*ngineering *L*ibrary (BCEL)[11], formerly known as *JavaClass*—a toolkit for static analysis and dynamic creation or transformation of JAVA class files. The analysis was then completed by manual inspection of the source.

The JTL code in Fig. 5 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.

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

```
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 overridden_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;

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

Fig. 5. A JTL query for finding 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 Fig. 5 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%.

The JTL equivalent for finding polymorphic pitfalls is much simpler and is not provided here. Polymorphic classes were found by analyzing the report of constructor falls.

3.3 Findings

Tab. 3 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 JEDIT. The relatively high rate at this collection is explained by its heavy reliance and inheritance from GUI classes with polymorphic behavior.

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 1.64% median value, even this fraction is small.

Table 3. Absolute and relative prevalence of polymorphic behavior in constructors (conservative analysis)

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%)

Comparing the second and the third column we see that on average, every polymorphic pitfall is responsible to about three polymorphic falls..

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 invariably 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 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 Tab. 4 which is similar to Tab. 3 except that it revolves around classes instead of constructors. That is, in Tab. 4 we report on the number of classes whose constructors can be categorized according to the three varieties of polymorphic behavior.

Table 4. Prevalence of classes with polymorphic behavior in their constructors (conservative analysis)

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%)

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 totals line in tables 3 and 4 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 (Sec. 2), classes which serve as bases tend to have more constructors. We should therefore have 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.

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 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.

3.4 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; 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.

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.

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 Solutions 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, MULTIFUNCTION OBJECT, FACTORY and INLINE DATA.

Fig. 6 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 MULTIFUNCTION OBJECT, while the most specific one is CONSTANT INITIALIZER. Patterns FUNCTION OBJECT and INITIALIZER OBJECT both generalize NON-CONSTANT INITIALIZER, while MULTIFUNCTION OBJECT generalizes and unifies the behavior both. FACTORY and INLINE DATA are isolates in the sense that they do not generalize, nor are being generalized by, any of the other patterns.

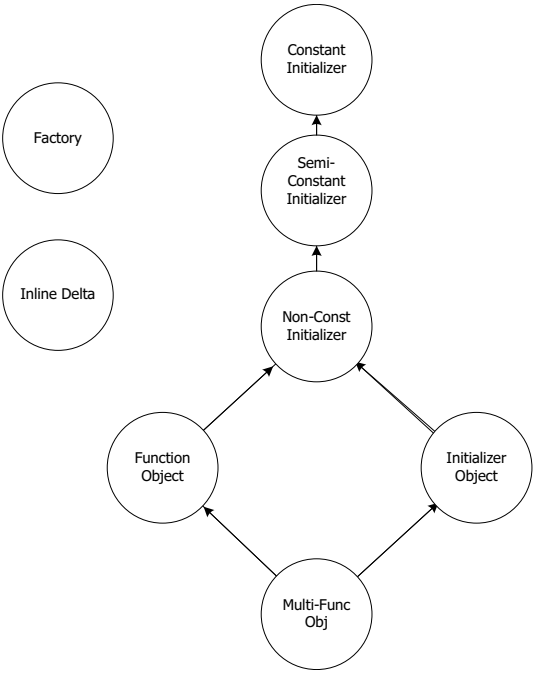

Fig. 6. Design patterns for devirtualization constructors

1. CONSTANT INITIALIZER: the most common type of virtual methods called inside a constructor are 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 `com.sun.jmx.snmp.Enumerated` and `com.sun.org.apache.xml.internal.security.utils.ElementProxy` and Eclipse's `org.eclipse.jdt.core.dom.ASTNode`.
   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 no-argument method 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 type specific for each subclass. Furthermore, the constructor invocation uses no receiver fields.
   An alternative for this is implemented as done for the CONSTANT INITIALIZER, by making the **`new`** `SomeField(...)` expression a parameter of the **`super`**`()` call.
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 performed 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.

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 INITIALIZER OBJECT to meet its own needs, and use it to set any number of fields in the superclass.
5. FUNCTION OBJECT: further generalization of the INITIALIZER OBJECT is targeted at the setting of superclass data members that are composite components, and are dependent on other data members. As a result, the computation of a dependent 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 received the value of the composite component as the return value of the FUNCTION OBJECT method.
6. MULTIFUNCTION OBJECT: finally, a combination of the INITIALIZER OBJECT and the FUNCTION OBJECT can be implemented using a MULTIFUNCTION OBJECT. The MUTLIFUNCTION-FUNCTION OBJECT is different than the FUNCTION OBJECT because it externalizes 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 date members).
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 of 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.
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 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.

Table 5. Applying the devirtualization design patterns on JRE and Eclipse

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. 5 depicts the prevalence of the various patterns found in our manual inspection.

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.

Tab. 5 also includes some cases where the techniques described above were not applied:

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 Fig. 7. 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

```
1 class JDialog {
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. 7. Overriding `setLayout()` in `JDialog`

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`.

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 methods contains a complex series of actions that are also very different than the original base implementation. We marked 6.9% of the falls identified for JRE as "unresolved", and less than 9% in Eclipse.

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 Sec. 3 was restricted to chains of direct message sends 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 section describes the results of our search for constructors which expose the `this`-identity, what we call *immodest* constructors.

5.1 Definitions

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

Immodesty Pitfall Constructors. We say that a constructor is an *immodesty pitfall* if it exposes the `this` identity, by assigning it into a variable, which may be accessed by external code or serve as a target of an internal method, or by passing it as a parameter to external code.

Immodest Fall Constructors. We say that a constructor is an *immodest fall* if it refines a an immodesty pitfall constructor, and overrides a method defined by the class of the pitfall constructor.

Immodest Constructors. A constructor is *immodest* if (i) it is an immodesty pitfall constructor *and* (ii) it is a refined by an immodest fall constructor.

Consider for example the JAVA class `Frame` depicted in Fig. 8 (drawn from the `java.awt` package). Then, both constructors of this class are immodesty pitfalls: The first since

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

Fig. 8. Constructors revealing a self reference in JAVA

it invokes method `init` which exposes the **`this`** pointer to an external class. The second constructor is such pitfall since it delegates its construction task to the first constructor. Observe however that both constructors are monomorphic.

To understand why immodesty is undesirable, consider again Fig. 8 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 patterns* 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. The analysis was complemented by a laborious manual inspection of the violating code.

5.3 Findings

Tab. 6 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 Tab. 3) we see that there are more than twice as many immodest constructors than there are polymorphic constructors.

Table 6. Prevalence of immodest behavior in constructors (conservative analysis)

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%)

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 immodest 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%.

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%).

Tab. 7 shows the prevalence of classes with constructors with immodest behavior.

Table 7. Prevalence of classes with constructors with immodest behavior (conservative analysis)

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%)

The data in this table can be summarized as follows: The same phenomena we found for immodest constructors in Tab. 6, including great variety in the prevalence of immodest and immodest falls types of behavior, and a higher incidence rate in these than in their polymorphic counterpart.

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

6 Conclusions and Further Research

Our main conclusion is that polymorphic construction is scarce, occurring in about 1.4% of all classes and 1.4% 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 tend to have (with statistical significance greater than 99%) more descendants.

It might be useful to repeat our study in a framework set of mind, That is, examine polymorphic pitfall constructors manually, concentrating on those for which no corresponding falls were found, in attempt to determine whether the pitfalls were intentional, serving a "hot-spot" purpose.

Unfortunately, the results of the analysis of exposure were not as striking. We found that there is a *potential* of leaking the `this` reference to external code in about 6% of the constructors. It is not clear however whether this potential leakage is significant, since it could be the case that the external code does not actually make use of this reference. For example, `this` could be assigned to one of the class's fields, as is often done in initializing a circular linked list, but even though this field has default visibility, no other class in the package uses it as a receiver, or even reads this field. Also, even if the external code sends a message to the leaked `this`, this sending could be done only after the class was fully constructed. Clearly, more work is required in this direction.

Based on an initial manual exploratory findings of immodest behavior we conjecture that in the majority of immodesty 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 contradicted, and the incidents are found to be of sufficient importance then perhaps the time is ripe for introducing two initialization phases: one in which the object is constructed internally, and another in which the object initial interconnection network is established. This second phase could be useful e.g., for a model-view-controller architecture, in which the construction of each view component includes storing its address in the update list of the model component. We suspect however that such cases are rare, and could be addressed by construction patterns tailored for this purpose.

We 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. The

data we collected shows that the phenomena is sufficiently prevalent that designers of new languages should consider appropriate way of addressing it.

In light of our finding regarding polymoprhic constructors, language designers should consider introducing `init` (and `destruct`) methods is in setting a clear cut boundary between between the construction (and destruction) process and the normal object's life time. The variety of ways of doing that are discussed briefly in the opening section of this article. An inspection of the patterns of polymorphic behavior in constructors and their coverage rate suggest that such semantics may be even feasible in existing language with extensive code base (probably as compiler option or extension).

We also believe that the time may is ripe for introducing what we have called *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. A familiar example is JAVA's `getClass()` method. Such a feature is useful in constructors, as demonstrated by the construction patterns that can be more readily implemented using this feature.

References

1. Arnold, K., Gosling, J.: The Java Programming Language. The Java Series. Addison-Wesley, Reading (1996)
2. Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., Tempero, E.: Understanding the shape of Java software. In: Tarr and Cook [27]
3. Birka, A., Ernst, M.D.: A practical type system and language for reference immutability. In: Vlissides, J.M., Schmidt, D.C. (eds.) Proc. of the 19th Ann. Conf. on OO Prog. Sys., Lang., & Appl (OOPSLA 2004), Vancouver, BC, Canada, October 2004. ACM SIGPLAN Notices, vol. 39 (10) (2004)
4. Bokowski, B., Vitek, J.: Confined types. In: Proc. of the 14th Ann. Conf. on OO Prog. Sys., Lang., & Appl (OOPSLA 1999), Denver, Colorado, November 1-5, 1999. ACM SIGPLAN Notices, vol. 34 (10), pp. 82–96. ACM Press, New York (1999)
5. Cabral, B., Marques, P.: Exception handling: A field study in Java and.NET. In: Ernst [11], pp. 151–175
6. Chalin, P., James, P.R.: Non-null references by default in Java: Alleviating the nullity annotation burden. In: Ernst [11], pp. 227–247
7. Clifton, C., Millstein, T., Leavens, G.T., Chambers, C.: MultiJava: Design rationale, compiler implementation, and applications. ACM Trans. Prog. Lang. Syst. 28(3) (May 2006)
8. Cohen, T., Gil, J.: Self-calibration of metrics of Java methods. In: Proc. of the 37th Int. Conf. on Technology of OO Lang. and Sys (TOOLS 2000 Pacific), Sydney, Australia, November 20-23, 2000, pp. 94–106. Prentice-Hall, Englewood Cliffs (2000)
9. Cohen, T., Gil, J.Y., Maman, I.: JTL—the Java tools language. In: Tarr and Cook [27]
10. Eckel, N., Gil, J.: Empirical study of object-layout strategies and optimization techniques. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 394–421. Springer, Heidelberg (2000)
11. Ernst, E. (ed.): ECOOP 2007. LNCS, vol. 4609. Springer, Heidelberg (2007)

12. Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. In: Crocker, R., Steele Jr., G.L. (eds.) Proc. of the 18th Ann. Conf. on OO Prog. Sys., Lang., & Appl (OOPSLA 2003), October 2003. ACM SIGPLAN Notices, vol. 38 (11) (2003)
13. Gil, J., Itai, A.: The complexity of type analysis of object oriented programs. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 601–634. Springer, Heidelberg (1998)
14. Gil, J., Maman, I.: Micro patterns in Java code. In: Johnson and Gabriel [17], pp. 97–116
15. Hejlsberg, A., Wiltamuth, S., Golde, P.: The C# Programming Language, 2nd edn. Addison-Wesley, Reading (2003)
16. ISE. ISE EIFFEL The Language Reference. ISE, Santa Barbara, CA (1997)
17. Johnson, R., Gabriel, R.P.: Proc. of the 20th Ann. Conf. on OO Prog. Sys., Lang., & Appl (OOPSLA 2005), San Diego, California. ACM SIGPLAN Notices (2005)
18. Kniesel, G., Theisen, D.: JAC access right based encapsulation for Java. Softw. Pract. Exper. 31(6), 555–576 (2001)
19. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes 31(3), 1–38 (2006)
20. Male, C., Pearce, D.J.: Non–null type inference with type aliasing for java. Technical report, Computer Science, Victoria University of Wellington, NZ (August 2007)
21. Melton, H., Tempero, E.: Static members and cycles in Java software. In: International Symposium on Empirical Software Engineering and Measurement, pp. 136–145 (2007)
22. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
23. Noble, J., Lea, D.: Editorial: Aliasing in object-oriented systems. Soft. Practice & Experience 31(6), 505 (2001)
24. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland (2004)
25. Stallman, R.M.: Using the GNU Compiler Collection (GCC): GCC Version 4.1.0. Free Software Foundation (2005)
26. Stroustrup, B.: The C++ Programming Language, 3rd edn. Addison-Wesley, Reading (1997)
27. Tarr, P.L., Cook, W.R. (eds.): Proc. of the 21st Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA 2006), Portland, Oregon, October 22-26. ACM SIGPLAN Notices (2006)
28. Tschantz, M.S., Ernst, M.D.: Javari: Adding reference immutability to Java. In: Johnson and Gabriel [17]
29. Wrigstad, T.: Ownership-Based Alias Managemant.[12] PhD thesis, KTH, Computer and Systems Sciences (May 2006)
30. Zibin, Y., Potanin, A., Ali, M., Artzi, S., Kieżun, A., Ernst, M.D.: 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)

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