Grenoble INP – Ensimag
Technical Report

Contracts for Java: A Practical Framework for Contract Programming

Nhat Minh Lê

May 25th–August 27th 2010 Internship[1] *for*
Google Switzerland GmbH[2]

[1]In French, *Stage de deuxième année.*
[2]Brandschenkestrasse 110. 8002 Zürich-Enge.

Contracts for Java: A Practical Framework for Contract Programming

Nhat Minh Lê

January 12, 2011

Abstract

This report introduces Contracts for Java (Cofoja), a new framework for contract programming in Java, and the successor to Johannes Rieken's Modern Jass. Based on the same standard Java technologies such as annotation processing and bytecode instrumentation, it improves upon the stub-based separate compilation strategy of its predecessor.

Contracts for Java promotes a minimalistic yet expressive feature set, reminiscent of the original Design by Contract model pioneered by the Eiffel language, while concentrating on robustness, performance, and enforcing non-intrusiveness in the compilation process.

1 Introduction

Contract programming is an object-oriented software design paradigm originally described by Bertrand Meyer[12] and popularized by his Eiffel programming language and its built-in Design by Contract features. In essence, it extends the subtyping relationships of a class hierarchy with programmatically-enforceable constraints called *contracts*. In Meyer's original proposal, these are expressed as method preconditions and postconditions, similar to those of Hoare logic[8], and class invariants, which act as both preconditions and postconditions to all interfaces of the class. Contracts can be inherited in accordance with the Liskov substitution principle[11].

While contract programming has seen most success in its original Eiffel implementation, where it is revered as a fundamental idiom, it has since been exported and adapted with varying degrees of success to other languages, including Java.

Contracts in Java must rely on third-party frameworks, as the language itself only includes rudimentary assertion support[1] as of Java 6. Numerous such efforts have produced a dozen alternatives over the years, a few of which are: JML[10], C4J[2], Contract4J[5], JASS[9], and the related Modern Jass[14], upon which Contracts for Java is based.

Contracts for Java brings together and improves upon interesting aspects that have been introduced separately in previous works, with the goal of producing a robust standalone framework dedicated to contract programming in Java:

- concise contract specification through Java annotations as introduced in version 5 (Contract4J, Modern Jass);

- full Java expression support in contracts through auxiliary compilation using the standard compiler API (Modern Jass);
- compile-time syntax and type checking (no interpretation at run time of the contract expressions, as in Contract4J);
- separate non-intrusive compilation of contracts (integrates but does not replace the Java compiler, as JML does, or precompile Java source files, as JASS does);
- online and offline weaving of contracts into their targets through bytecode rewriting (C4J, Contract4J, Modern Jass);
- and a lightweight code base with few dependencies (in particular, no reliance on AspectJ, as is the case with Contract4J).

The main contribution of this implementation is an enhanced version of the compilation model introduced in Modern Jass, inspired by aspect-oriented programming techniques, and based on use of the standard Java compiler to produce stubs and mocks.

Sections 2 and 3 give an overview of the Contracts for Java framework and its use, while sections 4 and 5 go into the details of how contracts are represented, translated and evaluated internally.

2 Contract Annotations

All features of Contracts for Java are expressed through a small set of Java annotations: `Requires`, `Ensures`, `Invariant`, directly borrowed from Eiffel[6], and the less common `ThrowEnsures` and `Contracted`. The first four annotation types specify contracts as lists of *clauses* (see fig. 1 for an example), containing arbitrary Java expressions which must all hold for that piece of contract to be enforced.

As in Eiffel, contracts can be inherited, composing fragments along the inheritance chain using the *and* and *or* boolean operators.[1]

These annotations cover the complete set of basic contract specifications, on top of which it is easy to build more complex behaviors. Indeed, since clauses can hold arbitrary Java expressions, any check that can be written in Java can be written in a Cofoja contract.

As stated in its goals, Contracts for Java does not aim to provide other advanced and specialized constructs that have been introduced in packages such as JML or JASS. Instead, it focuses on usability and availability of its core features.

Adding even the smallest feature to the entire array of constructs present in an existing language such as Java has proven to be a difficult task, and previous attempts at contract programming frameworks for Java have often ended with partial support: for instance, JML only recognizes limited Java 5 in spite of recent efforts to bring it up to date[3]; Modern Jass fails to compile code with generics; and libraries that forgo code or bytecode manipulation can only contract interfaces[7].

[1]Contravariant preconditions are composed using *or*; covariant postconditions and invariants use *and*.

```
@Invariant("size() >= 0")
interface Stack<T> {
  public int size();

  @Requires("size() >= 1")
  public T peek();

  @Requires("size() >= 1")
  @Ensures({
    "size() == old(size()) - 1",
    "result == old(peek())"
  })
  public T pop();

  @Ensures({
    "size() == old(size()) + 1",
    "peek() == old(obj)"
  })
  public void push(T obj);
}
```

Figure 1: A contracted stack example.

One purpose of Contracts for Java is to bring contracts to the largest subset possible of the Java language[2], with an emphasis on simplicity and maintainability. The following sections describe the fundamental techniques that underlie the implementation of contracts in Contracts for Java. Auxiliary features such as support for debugging are only mentioned where appropriate and would deserve a study of their own, which is outside the scope of this report.

3 Components and Architecture

Figure 2 describes the process of compiling and running Cofoja-enabled contracts. Contracts for Java integrates at various points with the standard Java toolchain, both during compilation and at run time:

1. Before the Java compiler, as a read-only *source preprocessor*. It is important to note that this step does *not* alter the original source files; its sole purpose is to extract information that is not otherwise available to annotation processors (see below). The preprocessor is not needed if the Java compiler exposes the `com.sun.source` non-standard API.

2. During the compilation of Java files, as an *annotation processor*. The annotation processor[3] is responsible for the actual compilation of the con-

[2]At the time of writing, there are a few known cases where support is lacking, and probably many more that have not been tested. Fixes and workarounds are being developed, though some limitations such as contracts being ignored on anonymous classes are inherent to the programming interfaces Contracts for Java is built upon and hence cannot be lifted without a change in the underlying framework, especially the annotation processing API.

[3]The acronym APT, which stands for Annotation Processing Tool, is often encountered.

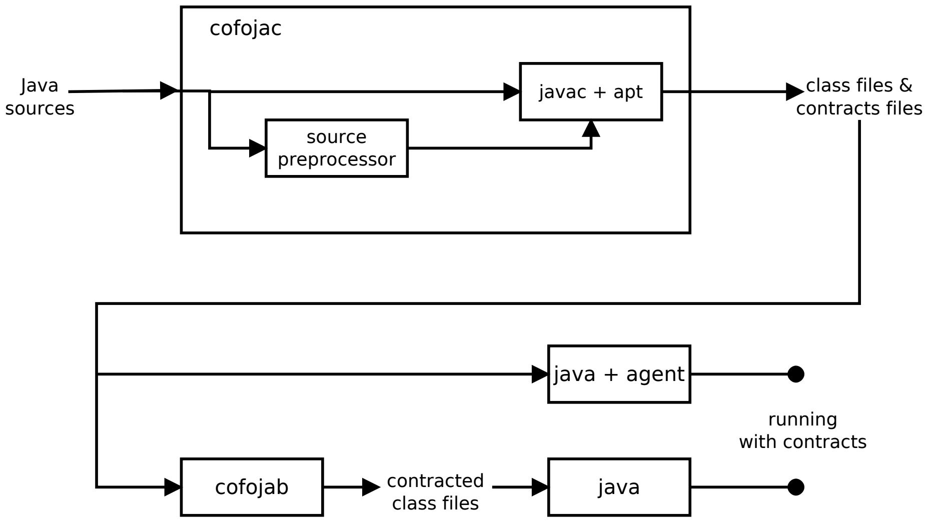

Figure 2: The compilation and deployment process.

tract code into separate bytecode files.

3. At class loading time, as a *Java agent*, a plugin to the Java launcher. The Java agent rewrites the bytecode[4] of contracted classes to weave contracts into their target methods.

4. Optionally, the instrumentation step can take place outside of the Java launcher, as a standalone process that merges normal contract-free bytecode (in the form of standard class files) with the corresponding contract bytecode (in the form of contracts files) to produce contract-enabled class files. These classes always check their contracts and do not require the Java agent to run.

Also bundled with Contracts for Java are two utility scripts: `cofojac` and `cofojab` that wrap around the first two steps, and the last step, respectively, for easy invocation from the command-line.

One key idea inherited from Modern Jass is that contracts are compiled *separately*. The original source files are never altered in any way. This has some drawbacks, mostly syntactical (e.g. contract code needs to be enclosed in double quotes), but guarantees the following two properties, which contribute to making the system more robust:

- The compilation of contracts does not interfere with the normal compilation process; this limits the possibilities of bugs in the contract compiler affecting the intended behavior of the original code.

- The contract compiler is not needed for code with contracts to compile; linking against the annotation definitions is enough. If, for some reason (e.g. a bug in the framework, or an unsupported evolution of Java),

[4]Bytecode rewriting is generally referred within the official Java documentation as *instrumentation*, owing to the fact that its first and primary use is to allow external components to take measurements during execution.

Contracts for Java cannot be used on a specific piece of code, it is still possible to continue development by giving up on contracts selectively or temporarily.

Since they are processed concurrently with the classes they are bound to, and because of their initial representation as inert data (i.e. string literals), contracts must go through multiple forms during their life. The following sections discuss how contracts are translated from their specification to actual runnable bytecode. Section 4 describes the general compilation technique, borrowing ideas and concepts from aspect-oriented programming, while section 5 is an in-depth case study of its implementation in Contracts for Java.

4 Contract Aspect Compilation

From the perspective of the aspect-oriented doctrine, checking contracts constitutes a crosscutting concern. Though the contracts themselves specify properties of the business logic of the program, *checking* them can be seen as a different aspect of the application.

The verification and failure handling code can be naturally expressed as advice to the contracted methods[15]. Modern Jass introduced a new compilation technique based on stubs and the use of the standard Java compiler to produce bytecode for the advice, that can later be easily weaved in through a Java instrumenter agent.[14] That approach had the advantage of not requiring a totally different compiler with its own understanding of the Java language. It was, however, tightly coupled with the logic of the contract paradigm.

Contracts for Java improves on that design by generalizing the compilation process to a systematic construction that maps more transparently to aspect programming concepts.[5]

4.1 Advice Compilation

The main challenge in compiling advice code lies in that such code requires access to the same environment as the advised method[6]. In other words, a piece of advice expects to be compiled as if it were in the *same scope* as the code it modifies. A precondition, for example, is likely to examine the arguments to the original call; more generally, any contract may want to call a private method of the class. This is easily achieved if the compiler knows about such needs; the standard Java compiler is, however, oblivious to them.

In order to use the Java compiler, advice code is injected back into a copy of the original classes, turned into bytecode, and the resulting compiled methods extracted. These can then readily be weaved into the final classes (as additional methods) before they are loaded. In this way, bytecode belonging to the auxiliary aspect is both compiled and stored separately, and directly reusable.

The technique is further refined by noticing that original class methods can be replaced by stubs. What matters is that the compiler properly generates

[5]Such a construction could in theory be applied to other kinds of advice, but the practicality of compiling mixed verbatim user-supplied Java and generated code, and retrieving the potential resulting compilation errors makes it unlikely to find widespread usage outside of very specific cases.

[6]This report only discusses method advice.

calls to these methods from within the advice code; their actual contents are irrelevant. Incidentally, the standard annotation processing API provides just enough information to easily generate these stubs by mirroring the reflected elements of the class hierarchy.[7] An added benefit is the extended applicability to classes available only as bytecode: method signatures can be read from class files, ignoring the associated bodies.

4.2 Join Points and Weaving

Weaving is accomplished through bytecode rewriting: the additional methods are added to the class, and the advised methods are augmented with calls to them on entry and exit.

Transient data such as arguments and the return value, if any, need to be explicitly copied, as they are not implicitly carried over from method to method.

In addition, pieces of advice that wrap around the method, of which post-conditions with old-value expressions are a trivial special case, need to maintain a context. In Contracts for Java, the decision has been made to keep the context as a set of additional local variables, which are added to the signature of the post-execution advice method.

Contracts lend themselves especially well to this approach as the context information naturally splits into a set of disjoint variables. Two alternatives are: inlining of the advice bytecode, and splitting of the advised method into an auxiliary instance.

Inlining is an appealing solution if the advice code is not inherited or is guaranteed not to access any private members of ancestor classes. Otherwise, inlining will break access privileges.

A more general strategy consists of splitting the advised method into a helper, which contains the actual code, and a wrapper, which borrows the original name and signature but actually defers its work to advice methods, which, in turn, are responsible for calling the auxiliary procedure.

4.3 Advice Inheritance

Inheritance is central to the contract paradigm. It is, however, rather orthogonal to aspect-oriented programming. With this construction, it should be clear that advice methods are naturally inherited through the normal hierarchy and are thus accessible at the time of weaving. With some more effort, they can also be made available to advice source code.[8]

Depending on the application, inherited advice recombination may vary in nature and purpose and the different possible strategies are beyond the scope of this report.

However, this brings the question of interfaces. It should be noted that writing advice for an interface is only relevant if there is a way to incorporate these pieces into the concrete classes deriving from that interface. Obviously,

[7] This is not actually entirely accurate. There is a corner case that occurs whenever an enumeration type implements an interface by providing a distinct implementation of its methods in all of its constant members, but no base implementation in the enumeration class itself. In that scenario, the constant members each trigger the creation of an anonymous class, which is not reflected by the annotation processing API.

[8] They need their own stubs!

injecting the aspect code into the original interface cannot be done; the resulting Java file would not even compile, which defeats the purpose of this technique.

Instead, the additional methods can either live in a concrete class implementing that sole interface, or in a utility class within the same scope as the interface they are attached to (same package or same outer class).

The first solution leverages the existing weaving infrastructure: classes implementing the interface can simply absorb the advice methods and proceed as usual. Care must be taken, though, to turn all references to the implementing class into references to the actual class into which the methods are injected. There are also some issues with code duplication.

The second solution requires that pieces of advice for interfaces be altered to work with an extra parameter standing for the effective object to be operated on. However, they avoid any duplication. It should be noted that relying on external procedures does not limit the expressivity of these constructs, as interfaces only have public members.

Section 5.2.1 addresses these issues from the practical perspective of contracts: it compares the implementations from Modern Jass, which uses a variation of the first strategy, and Contracts for Java, which employs the second.

4.4 Limitations of the Compilation Technique

This approach to aspect compilation relies on the guarantee that the Java compiler does not optimize across method boundaries. In particular, it requires that source and bytecode methods match one-to-one, e.g. no method splitting.[9] In the presence of stubs, inlining is another major unwanted optimization.[10] In practice, this should not pose a major problem, however, considering current trends towards optimization at the level of the virtual machine rather than the compiler. Moreover, contracts, specifically, are intended mainly for development and debugging purposes, and hence will likely not be compiled with optimizations turned on.

Another perhaps more serious issue is with compilation errors. Error reporting requires much care and is prone to subtle inconsistencies. The simple idea of echoing the results from the underlying compiler process to the user has been shown to generate messages unrelated to the actual *user* code, due to its being mixed with generated elements. Contracts for Java features *ad hoc* enhancements to help clarify the most frequent diagnostics; a more general solution would most likely make use of enhanced compiler integration programming interfaces.

5 Life Cycle of a Contract

The above compilation strategy is employed in Contracts for Java to transform annotations into runnable checks. Contracts start their lives as simple code enclosed in string literals. These are extracted and compiled into a set of methods, which is stored in contracts files (cf. fig. 2). Finally, these methods are

[9]Auxiliary generated methods are manageable though, and there are cases of current Java compilers synthesizing such procedures, e.g. access methods for inner-to-outer member references.

[10]Compile-time inlining in Java is fairly limited, though, as all methods are virtual by default.[13]

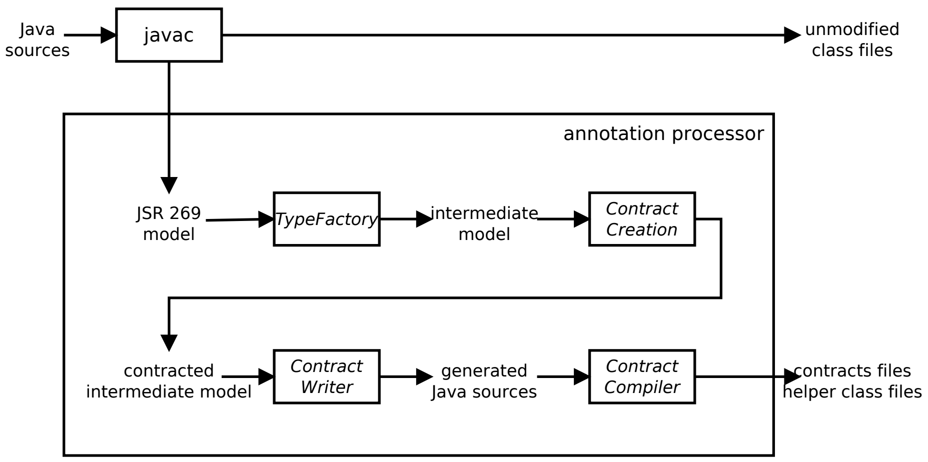

Figure 3: The execution of the annotation processor.

injected into the original classes and the call code is inserted at places where the run-time checks should occur.

Section 5.1 gives an overview of the compilation process, while sections 5.2 and 5.3 focus on contract representation and invocation.

5.1 A Zoom on Contract Compilation

As discussed previously, contract compilation takes place in two stages: a preprocessing step, followed by annotation processing.

The source preprocessor is a simple parser that provides two pieces of information to the annotation processor: the import statements in effect in a source file, and a line number map between contracts and the lines that hold their definitions.

This information is not available to the annotation processor through the standard JSR-269 API. However, compilers based on the Sun/OpenJDK implementation export an abstract syntax tree extension[4]. Contracts for Java can take advantage of that fact and does not require any preprocessing if run with a compatible Java compiler. Otherwise, the data is shared through temporary files managed by `cofojac`.

Preprocessing is necessary in absence of specific extensions. Because the standard API can reflect classes in any readable form available[11], not only does it skip source-specific information (e.g. line numbering), it does not even offer access to the underlying file object. Hence, these source files need to be examined in a separate step from annotation processing proper. This pass is an addition to the architecture of Modern Jass; Contracts for Java uses these data to mint fine-grained debug information that were previously unavailable in its predecessor[12].

As illustrated in figure 3, the actual contract compiler operates in several steps and transitions through various formats of the program. Contracts primarily exist under three forms:

[11]In practice: these classes can be in source or bytecode form.
[12]See “Missing Debug Information,” in [14] p. 98.

1. Originally, they are code enclosed in string literals and bound syntactically to elements of the program through annotations.
2. They retain their string nature in the intermediate representation of the program, but are now associated with their metadata (i.e. what type of contract they are, and what targets they are bound to) instead of generic annotations.
3. Finally, an *alternate version* of the original classes is generated as plain Java text, which includes the contracts, this time as first-class citizens. The contracts files are the result of compiling these new source files.

As described previously, the generated code *only* implements the contract methods. It does not feature any of the real application logic and substitutes all these methods with stubs. In that sense, contracts are compiled as true Java code, albeit in a mock environment that mimics the original. The fake components are replaced by real class code during the bytecode instrumentation step, either during class loading or offline.

By compiling against a full mock environment, Contracts for Java leverages the underlying Java compiler infrastructure and thus allows for all the constructs accepted by the source version of the language in contract expressions.

5.2 Contract Method Layout

Once compiled, contracts reside as methods in contracts files. Contracts for Java places great emphasis on compile-time transformations, especially as compared to its ancestor, Modern Jass. In particular, in Modern Jass, inheritance is handled during injection, at the call site.[13] Contracts for Java originally followed that same idiom until problems arose regarding inheritance with use of bridge methods. In the presence of bridge methods, the inheritance graph between methods is costly to compute accurately. Fortunately, JSR-269 abstracts these compilation artifacts, and thus it made perfect sense to move the process to compile time.

Consequently, contracts files implement the full contract logic, including inheritance, and only need to be weaved into the contract-free class files, with proper call sites inserted, to be enabled.

Figure 4 shows the call graph for a hypothetical method `m` of a class `B`, inheriting from `A` and implementing interface `J`. It illustrates the different situations that can occur when contracts are propagated throughout the class hierarchy.

Compilation of class contracts differs from that of interface contracts, but they both adhere to the following pattern.

The *code* for a declared contract C is compiled as a method called a *helper contract method*, or simply helper method. The role of the helper method is to evaluate precisely the declared contract C. It has no knowledge of the context in which it is evaluated and does not deal with inheritance issues, although it takes the *variance* of the contract type into account (see section 5.3 for a more thorough discussion on the subject).

For each *method* M to which contracts apply, a *contract method* is created. The role of the contract method is to execute *all contracts* that apply to M. It

[13]See "Checking Contracts at Runtime," in [14] p. 55.

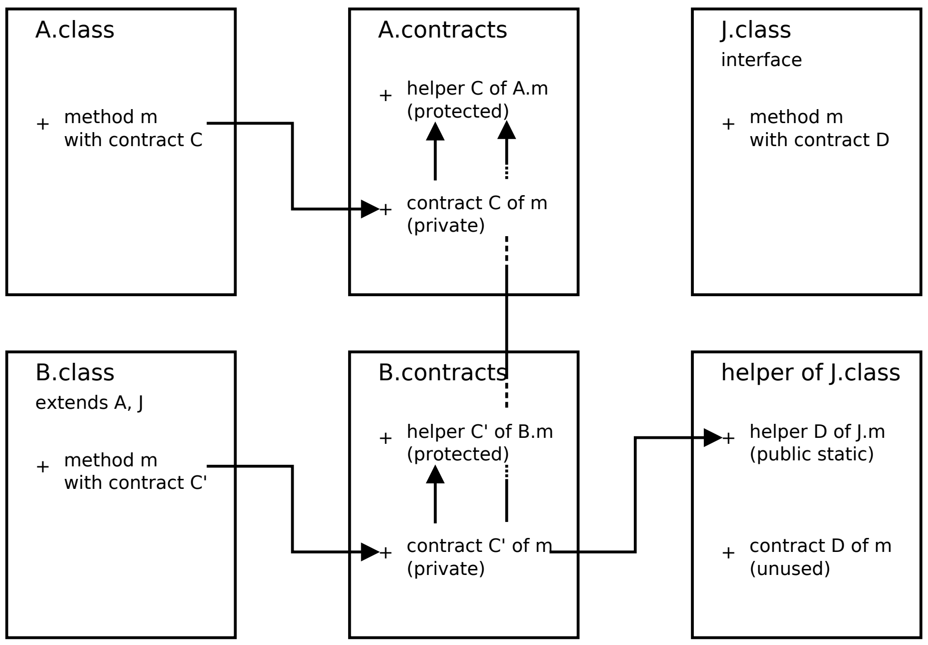
A.class
+ method m with contract C
A.contracts
+ helper C of A.m (protected)
+ contract C of m (private)
J.class
interface
+ method m with contract D
B.class
extends A, J
+ method m with contract C'
B.contracts
+ helper C' of B.m (protected)
+ contract C' of m (private)
helper of J.class
+ helper D of J.m (public static)
+ contract D of m (unused)

Figure 4: The layout of a contract call.

does not evaluate any contract itself, and instead relies on calling the individual helper methods of these contracts.

The decision to use a helper method for each contract may appear unreasonably costly at first. Alternatives include inlining and injection of private methods in each class that needs them. Helper methods are however necessary in order to honor all access restrictions (a helper method executes in the context of its declared class, and hence has access to even its private members). The separation also allows for debug attributes[14], and, more generally, easier source tracking, run-time debug support as well as compile-time error reporting.

5.2.1 Classes and Interfaces

As explained in section 4, since interfaces cannot contain code, interface contracts are put in a *helper class*. There is one helper class for each interface, which lives in the same scope as the interface it implements contracts for.

This split between classes and interfaces is another departure from Modern Jass, which employs an uniform model that translates all types to abstract classes. While such a scheme has the advantage of simplicity, there are a few issues inherent to its design:

- It cannot support full mock builds: since the nature of the contracted interface changes to that of a class during contract compilation, types that depend on it being an interface (e.g. child interfaces or implementing

[14]Class file attributes that give information on the correspondence between the source file and the bytecode. The Java class file format only allows a class file to represent *one source file*. Hence, any approach that injects inherited contract methods into a class will inevitably have source file name mismatch for these.

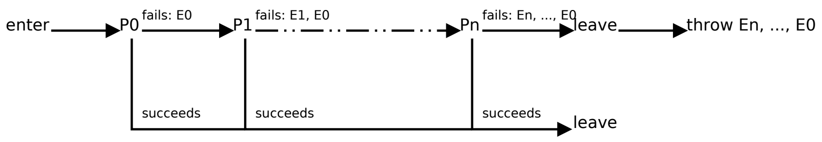

Figure 5: A contravariant call sequence.

classes) cannot be compiled in the same batch. This affects compilation performance negatively. By translating interfaces to interfaces, Contracts for Java does not have this problem.

- It requires sanitizing of the interface calls within the contract expressions: calls to interface and class methods use different Java bytecode opcodes. Determining which sites need to be sanitized is a costly per-call operation[15].

- And most importantly, it requires that contract methods be injected into each class implementing the interface: contrary to classes, interfaces cannot hold their own code, and hence their descendants do not inherit the contract methods and must have them reinserted[16]. Beside the bytecode duplication, this implies a caching mechanism of some sort on the part of the agent, if reloading of the interface contract bytecode is to be avoided for each implementing descendant. A Cofoja helper class, conversely, only exists in one copy and is actually loaded in the same way as any other utility class.

In effect, this has led to a sensibly more complex code generation process in Contracts for Java, which aims at exposing a perfect mock version of the original code whenever possible. The bytecode rewriting component, on the other hand, is simpler and leaner and can notably instrument each class file independently of any others, since the knowledge necessary is already contained in the associated compiled contracts file.

5.3 Run-time Contract Behavior

Once weaved and loaded, contracts are ready to be called to carry out their checks. As explained above, the actual contract code is executed by helper methods.

A helper method evaluates its clauses in a short-circuit way, and in a well-defined order. A contract method, on the other hand, does *not* make any guarantee on the evaluation order of the different helpers it calls, though it still uses a short-circuit approach.

For contravariant contracts (preconditions), which are combined using the *or* operator, this means contract evaluation will stop at the first established contract. (See fig. 5.)

[15]As an exercise, consider a method `I.f` that takes an argument `x` of the same interface type `I` it belongs to, and a contract on `f` which calls some other method `g` on `x`. How do we determine that this site must be changed to an interface call?

[16]Unless one of their ancestors also implement said interface.

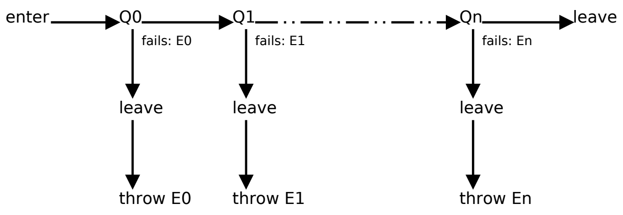

Figure 6: A covariant call sequence.

For covariant contracts (postconditions and invariants), which are combined using the *and* operator, this means contract evaluation will stop at the first failing contract. (See fig. 6.)

When a contract method fails, it throws an exception. An important difference between covariant and contravariant implementations is the responsibility of creating and throwing the exception. The following table summarizes these roles:

	Contravariant	Covariant
Helper method	create	create and throw
Contract method	throw	—

Since the method that creates the exception also fills it with debug information, it must always be the helper method that handles this task, so that contract violations are reported at the source, corresponding to the contract expression that has failed, instead of at the call site (inside the contract method), where the contract is applied. These may differ in case a descendant is enforcing the contracts of its ancestor.

5.3.1 Contract Recursion

Beside calling its helper methods, a contract method also calls two static bookkeeping methods: `enter` and `leave` of the `ContractRuntime` class, in the `com.google.java.contract.core.runtime` package. Their role is to guard against contract recursion.

In Contracts for Java, only the first level of contracts is enforced: contracts that would be triggered by calls in other contracts are not checked. This is to avoid infinite recursion in very common cases.

This rule is implemented through a thread-local flag[17] that is set by a contract through the `enter` method, and cleared by calling `leave`. If the flag is already set when `enter` is called, then the contract is skipped, as if it had been asserted successfully.

5.4 The Special Case of Old-value Expressions

Evaluation of old-value expression is not shown on the layout and sequence diagrams but are an integral part of contract compilation and invocation.

For contracts containing old-value expressions, in addition to the helper and contract methods, *old-value contract methods* are created. These methods are called by the contracted method after any precondition, and their values

[17]The code uses the word *lock*, which can be confusing since it does not refer to the concurrent concept of mutual exclusion; here, each thread has its own lock and no race condition can ever occur.

are stored in local variables. These variables are later passed as additional arguments to the postconditions for further evaluation.

At compile time, old-value expressions incur some additional work in order to separate the expressions from the place where their values are used. Each old-value subexpression is replaced with a reference to the corresponding extra argument. In order for the resulting code to be statically correct type-wise, Contracts for Java coerces the variable into its proper static type by abusing Java genericity.[18]

With regard to method layout, though old-value expressions are not a proper form of contracts, old-value contract methods are implemented using the same helper–contract pattern as other kinds of contracts. In this case, the helper computes the real value, while the contract method merely calls the correct helper. Here, the difference between the two may be harder to spot, but it is analogous to the difference between helper and contract methods for other kinds of contracts: a helper method is generated for each old-value expression and computes that precise expression; contract methods, on the other hand, are created for each contracted method that must have access to that old value, either because of its direct contracts, or the ones it inherits. Therefore, there are as many contract methods as the number of old-value expressions in *all contracts* verified by the method.

6 Discussion

Owing to its origins, Contracts for Java is strongly related to Modern Jass in its design and implementation.[19] This is especially visible in its reliance on the combination of the three annotation processing, compiler tools and instrumentation standard interfaces available with Java 6 and above. As discussed above, Contracts for Java improves upon the core techniques of Modern Jass in order to provide comprehensive language support; whereas Modern Jass was an experiment in implementation techniques, Contracts for Java aims for usability.

Initially developed as an internship project at Google, Contracts for Java is designed so as to easily fit into an existing build and test environment, while encouraging a progressive transition from existing practices, such as defensive programming and *ad hoc* precondition and postcondition checking. The non-intrusive compilation model and concise Java syntax allow partial contracts to be incorporated into projects, both existing and new.

Contracts for Java has been applied to its own source code and successfully produces contract-enabled class files of the compiler and run-time instrumenter through offline weaving[20]. These modified classes are suitable for normal operations of the framework[21] and have been used, in particular, as the default

[18]It is impossible for Contracts for Java to infer the type of the expression from its string representation without first parsing it, as Modern Jass does, hence the local variables used to store old values must have type `Object`; however, it is possible to trick the compiler used to compile the contract code into believing the variable is used with the correct type (cf. `ContractRuntime.magicCast`).

[19]The amount of actual code shared between the two projects, though, is today minimal.

[20]No Java agent is allowed to instrument its own bytecode.

[21]Preliminary measurements suggest a run-time overhead between 30% and 60% for an invocation of `javac` with annotation processing enabled over the entire Contracts for Java source tree.

version to run the test suite routinely for several development cycles.

Being able to circularly compile and enforce contracts on itself has certainly been a milestone for the project, and hopefully will make for a solid basis upon which support for missing language features and evolutions can be added in the future. Beside this primary axis of development, further efforts should now be directed towards improving error reporting[22], debugger support[23], and more generally aspects related to integration with the Java development tools ecosystem.

7 Conclusion

This report has presented how Contracts for Java makes use of compilation techniques inspired by aspect-oriented programming to provide Java with an Eiffel-like Design by Contract environment. These methods are well-suited to the implementation of contract checking and have been successfully employed to support most of the Java programming language in a robust way, remaining compatible with standard Java compilers and tools and open to future evolutions of the language.

The simple feature set and non-intrusive nature of Contracts for Java make it an ideal candidate for progressive or partial transitions to a contract-aware development methodology as well as further work targeting Java with contracts as a support.

Thanks to the efforts of the Contracts for Java 20% time team at Google, the code source will soon be available as open source under the terms of the LGPL. We hope that through contributions from the community and involvement within the team, Contracts for Java will emerge as a stable and versatile solution for contract programming in Java.

8 Acknowledgments

Modern Jass, without which there would be no Contracts for Java today, is the work of Johannes Rieken; thanks to him for his support of our initiative and encouragement. The idea of improving on Modern Jass to bring about a robust framework that could be used on all kinds of projects at Google and beyond originated from Andreas Leitner and David Morgan, now part of the Contracts for Java 20% time team. Many aspects of the design of Contracts for Java are the result of a collaborative effort. And I would like to thank the team for offering me the opportunity to work on this project, their continued support and guidance during the internship.

References

[1] J. Bloch. *JSR 41: A Simple Assertion Facility.* November 1999.

[22]Most errors are currently being relayed directly from the underlying compiler invocation on the generated code, which may make for puzzling messages.

[23]Contracts for Java supports weave-time generation of full standard debug information, including line numbers, in the resulting class files; this works reliably with offline contract injection, but can be problematic if done by the instrumenter and the debugger bypasses the agent in some way, e.g. by doing its own compilation.

[2] C4J. http://c4j.sourceforge.net/.

[3] D. R. Cok. *Adapting JML to generic types and Java 1.6.* Seventh International Workshop on Specification and Verification of Component-Based Systems, pp. 27–34, November 2008.

[4] Compiler Tree API. http://download.oracle.com/javase/6/docs/jdk/api/javac/tree/overview-summary.html.

[5] Contrac4J. http://www.contract4j.org/.

[6] Eiffel Software. *The Power of Design by Contract.* http://www.eiffel.com/developers/design_by_contract.html.

[7] A. Eliasson. *Implement Design by Contract for Java using dynamic proxies.* February 2002. http://www.javaworld.com/javaworld/jw-02-2002/jw-0215-dbcproxy.html.

[8] C. A. R. Hoare. *An axiomatic basis for computer programming.* Communications of the ACM, Volume 12, Issue 10, pp. 576–580. October 1969.

[9] Java with Assertions. http://csd.informatik.uni-oldenburg.de/~jass/.

[10] Java Modeling Language. http://www.jmlspecs.org/.

[11] B. Liskov, J. Wing. *A behavioral notion of subtyping.* ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 16, Issue 6, pp. 1811–1841. November 1994.

[12] B. Meyer. *Applying "Design by Contract."* Computer (IEEE), Volume 25, Issue 10, pp. 40–51. October 1992.

[13] Oracle Corporation. *The Java HotSpot Virtual Machine, v1.4.1.* http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_4.html

[14] J. Rieken. *Design By Contract for Java - Revised.* April 2007.

[15] D. Wampler. *Contract4J for Design by Contract in Java: Design Pattern-Like Protocols and Aspect Interfaces.* 2006.