A Verification-centric Software Development Process for Java

Daniel M. Zimmerman
Institute of Technology
University of Washington Tacoma
Tacoma, Washington 98402, USA
Email: dmz@acm.org

Joseph R. Kiniry
School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 4, Ireland
Email: kiniry@acm.org

Abstract

Design by Contract (DBC) is an oft-cited, but rarely followed, programming practice that focuses on writing formal specifications first, and writing program code that fulfills those specifications second. The development of static analysis tools over the past several years has made it possible to fully embrace DBC in Java systems by writing, type checking, and consistency checking rich behavioral specifications for Java before *writing any program code. This paper discusses a DBC-based, verification-centric software development process for Java that integrates the Business Object Notation (BON), the Java Modeling Language, and several associated tools including the BON compiler* BONc*, the ESC/Java2 static checker, a runtime assertion checker, and a specification-based unit test generator. This verification-centric process, reinforced by its rich open source tool support, is one of the most advanced, concrete, open, practical, and usable processes available today for rigorously designing and developing software systems.*

1. Introduction

Design by Contract (DBC) [1] is a design technique for (usually object-oriented) software that uses assertions both to document and enforce restrictions on data and to specify class and method behavior. The Business Object Notation (BON) [2] is an analysis and design notation for object-oriented systems based on DBC. Both were originally developed for use with the Eiffel programming language; DBC has since been added to many other programming languages through various language extensions and preprocessors.

Ideally, developers using the DBC technique write formal specifications (the *contracts*) for their software first, and write executable code only after completing, and performing at least some basic checking of, the contracts. However, tool support for DBC has until recently been limited primarily to *runtime* assertion checkers and unit test generators. There has been little support, beyond basic type checking, for verifying the logical consistency of contracts that have no implementations. In essence, the "design" component of DBC has been absent; DBC has meant writing contracts and code *in tandem* and using runtime assertion checking (RAC) and generated unit tests to check the contracts and code simultaneously. Unfortunately, when code is written in this way, it is often difficult to determine whether problems detected by RAC and unit testing are caused by errors in the code or by errors in the contracts.

In recent years, however, the tool landscape has changed significantly. The Java Modeling Language [3] (JML) has enabled both DBC and additional model-based specifications for Java programs, and the ESC/Java2 static checker has been extended to support the entirety of JML [4]. Additionally, a variety of static checkers have been added to ESC/Java2 to support reasoning about specifications. These tools allow developers to write, type check, and statically check the consistency of rich behavioral specifications *before* writing any program code.

The combination of BON and DBC (for design), ESC/Java2 and other tools (for static checking), and RAC and the automatic generation of unit tests from specifications (for runtime checking) results in a *verification-centric* development process, in which verification of both behavioral specifications and program implementation occurs continuously throughout development. This process supports the entire range of development artifacts via refinement, including concepts, requirements, feature models, types, mathematical models, informally and formally annotated source code, and formally annotated object code. We have previously discussed our use of ninja techniques to teach this process to our software engineering students [5]; here, we present the steps of this process from the practitioner's perspective, along with more detail about the critical tools and techniques involved. We also present process guidelines, including our code standard and our testing requirements.

2. Background

Our verification-centric software development process incorporates several tools and techniques. In this section, we introduce and provide background on the most important of these. Much more information is available in the cited works and in the documentation accompanying the tools.

2.1. Design by Contract/Contract the Design

Design by Contract (DBC) is a term coined in the 1980s by Meyer [1]. Our historical perception is that early proponents of DBC were Eiffel users and pragmatic correctness-by-construction formal methodologists who actually wrote software systems.

Eiffel encourages programmers to use contracts both by incorporating specification constructs like preconditions, postconditions, and invariants directly into the language and by providing a rich set of libraries with specifications and extensive design and development documentation [6]. The use of Javadoc [7] in the early development of Java is another example of such implicit proselytizing; the high-quality documentation available for the core Java libraries, and the integration of that documentation into the source code, has encouraged many Java developers to follow suit.

It has now been over fifteen years since Eiffel's introduction, and a new standardization effort was recently completed [8]. However, Eiffel still has limited support for rich contracts with quantifiers, and little implemented support for mathematical models, data and program refinement, and other specification constructs that have been found to be invaluable in adjacent research communities such as the JML community [9], [10]. Also, while Eiffel's tool support has evolved, it is not as comprehensive as that of larger and less fragmented communities.

As a result, Eiffel programmers (as well as developers who use DBC frameworks in other languages) usually end up performing "*Contract the Design*" (CTD) instead of DBC. Process-wise, CTD is the logical dual of DBC; in CTD, the contracts are written *after* the executable code in an attempt to formally specify its desired behavior and subsequently test the existing code against that behavior. CTD allows DBC advocates to, in the words of Parnas and Clements [11], "fake" a full, "rational" DBC-based process. Clearly, delivering software with accurate documentation and specifications—even those written after the program code is complete—is significantly better than delivering a system with no formal descriptions whatsoever.

Our experience has been that, though CTD is sometimes necessary, starting with DBC and writing formal specifications before writing any code results in both better specifications and better code. In order to effectively practice DBC in tandem with CTD, quality tool support for reasoning about program development artifacts other than executable code is necessary. As previously noted, such tool support does not currently exist for Eiffel. However, it does exist for Java in the form of various tools that work with JML.

2.2. The Java Modeling Language

The Java Modeling Language (JML) [12] is a specification language for Java programs. In addition to supporting class and method contracts in the DBC style, it allows developers to specify more sophisticated properties up to and including full mathematical models of program behavior. Several tools work with JML, including compilers, static checkers, test generators, and specification generators [13].

The *Common JML* tool suite is the "canonical" set of JML tools. It includes a type checker (`jml`), a JML compiler (`jmlc`) that compiles JML annotations into runtime checks, a runtime assertion checker (`jmlrac`), a version of Javadoc (`jmldoc`) that generates documentation including JML specifications, and a unit test generation framework (JML-JUnit, discussed further in Section 2.4). These tools have interfaces similar to those of their Java counterparts; for instance, `jmlc` behaves very much like `javac`. They can be used from the command line, in project build configurations with build systems like GNU Make and Ant, and in integrated development environments such as Eclipse.

Support for modern Java (1.5 and later) syntax in JML—including generic types—is still being developed. The JML4 project [14] is an effort to integrate JML into the Eclipse IDE, and OpenJML[1] is an effort to build a JML compiler based on the current OpenJDK[2] code base. While the process described here has been developed with the Common JML tool suite, the newer JML toolsets will function as drop-in replacements when they become generally available.

2.3. ESC/Java2

ESC/Java2 [4], an evolution of the original Digital SRC ESC/Java [15], is an extended static checker for Java. ESC/Java2 statically analyzes JML-annotated Java classes and interfaces at compile time to perform two main functions. First, it identifies common programming errors such as null pointer dereferences, invalid class casts, and out-of-bounds array indexing. Second, it performs several kinds of verification to attempt to ensure that the program code is correct with respect to its associated JML specifications.

Several new checkers have been developed atop ESC/Java2 to address the challenges we have faced during verification-centric systems development. For example, it is easy, especially for developers who are just learning DBC, to write inconsistent specifications such as preconditions that collapse to a logical false. Originally, ESC/Java2 would happily claim that methods fulfilled such specifications, because any verification condition can be proven valid from a false antecedent; with the addition of a specification consistency checker, such specifications now generate warnings.

The new checkers that we use in our process include a soundness and completeness warning system [16], a specification-aware dead code detection system [17], an improved loop invariant generation subsystem [18], the aforementioned specification consistency checker, and others.

1. http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/OpenJML/
2. http://openjdk.java.net/

As with JML, support for modern Java syntax and constructs in ESC/Java is still being developed. It is expected that the next major version of ESC/Java will support modern Java; another extended static checking tool, ESC4 [19], is also being developed as part of the JML4 toolset.

2.4. Unit Testing and Static Checking: Partners in Quality

Static checking is not sufficient to find all potential code problems, as there are many program properties that cannot be statically checked automatically [20]. Unit testing, a technique that tests individual units (packages, classes, methods) of a system with known inputs to verify that they generate expected outputs, helps address this insufficiency.

Considerable research has been devoted to efficient ways to run and analyze the results of unit tests, as well as to the generation of good sets of unit tests for particular systems [21]. In practice, however, the vast majority of unit tests are written using a manual process whereby the developer considers each method under test, determines good inputs to test it with, determines what the generated outputs for those inputs should be, and writes a *test oracle* to call the method with the test inputs and validate its outputs.

The JML-JUnit tool takes a different approach to unit testing: it generates unit tests that use the JML runtime assertion checker as a test oracle, eliminating the need for the developer to write one [22]. All the developer must do is tell JML-JUnit what data values to use as method parameters for testing. In this way, a test passes if there is no runtime assertion checking error generated when the method is called, and fails otherwise. This technique saves considerable effort, since the developer no longer has to write his own test oracles; however, its effectiveness depends on the correctness and completeness of the JML specifications, which is why it must be used in conjunction with the previously-discussed static checking techniques.

2.5. Other Specification Tools

We have used several other tools that support JML in various projects. The main tools that have demonstrated a great deal of utility in our process are Daikon [23], Houdini [24], and RCC/Java [25].

Daikon performs invariant detection by dynamically observing a program as it executes and reporting properties that held throughout the execution. Since Daikon generates JML, it is sometimes used to "bootstrap" the identification of invariants in CTD processes. It can also be helpful in finding invariants that a designer simply missed during DBC.

Houdini also generates JML but, instead of observing a system as it runs, uses heuristics to statically guess likely assertions. These guesses are then analyzed with other static checkers, such as ESC/Java, to determine their validity.

RCC/Java, unlike the previous two tools, does not generate JML. Instead, it statically analyzes concurrent Java source code to find potential race conditions. Annotations are used to indicate the relationships among objects (via references) and locks to express ownership, and among locks to express lock orderings.

These tools, along with the rest of the JML-related tools already discussed, are being integrated into the Mobius Program Verification Environment (PVE)[3] as part of the ongoing theoretical and development work of the EU FP6 Mobius project[4]. Our plans for formal process integration (concrete tool support for guiding a developer through our process) include environment documentation and support for process checklists, development waypoints, and context-aware hints.

2.6. Other Code Analysis Tools

In addition to JML and its related tools, the Mobius PVE includes custom-tuned versions of FindBugs[5], PMD[6], JavaNCSS[7], and CheckStyle[8], all of which are publicly-available open source static analysis tools for Java. These enforce the guidelines of our development process, including our coding and documentation standards (described in Section 4). By carefully configuring these tools for use within our verification-centric process, we enable a "flow like water"[9] style of quality software creation.

3. A Verification-Centric Process

Our verification-centric Java software development process is derived from the BON method [2], which we have refined with the specific goal of generating verifiable software. We use BON instead of the widely-used Unified Modeling Language (UML) for two main reasons. First, BON was originally designed to be used with Eiffel's DBC process, so it is a good fit for our verification-centric process. Second, BON has an easily-readable, English-based textual notation, in addition to a graphical notation that is much simpler than UML. An analogue to our process could certainly be implemented with UML and the Object Constraint Language, but we will not discuss such an implementation here.

Our development process has six steps: concept analysis; identification of queries, commands and constraints; refinement of concepts; refinement of queries and commands; refinement of constraints; and finally, implementation. In this section, we describe each step in detail.

3. http://mobius.ucd.ie/
4. http://mobius.inria.fr/
5. http://findbugs.sourceforge.net/
6. http://pmd.sourceforge.net/
7. http://www.kclee.com/clemens/java/javancss/
8. http://checkstyle.sourceforge.net/
9. . . .as expressed by Chinese philosopher Laozi in the Tao Te Ching.

While we describe the process in a "waterfall" style, in practice it is rarely used that way. The process has been used in a "spiral" fashion as well as with extreme programming techniques. The reason the process works within these different development styles is that our underlying formal foundations, as realized in our specification languages and tools, are refinement-centric. In particular, the refinements used are all *reversible* (discussed further in Section 3.7).

The general notions and facets of this process are *not novel*. What *is* novel is our particular, concrete instantiation of these facets via underlying theory and tools and, more importantly, the fact that we and others have *actually used* this process over the last several years to develop a number of complex industrial and academic systems.

3.1. Concept Analysis

The first step in the process, *concept analysis* (also known as *domain analysis*), involves identifying and naming the important concepts (also called *classifiers*) in the software system and collecting them into sets of related classifiers called *clusters*. We also specify the *is-a* and *has-a* relationships among concepts. At this stage of development, we are *not* talking about "classes" in the implementation sense. A classifier might eventually refine to a class, and its relationships to other classifiers might refine to inheritance and containment relationships among classes; it might also refine to more than one class, or even refine away entirely.

The goal of concept analysis is to generate a vocabulary for working with the desired system, as well as a very coarse "picture" of the relationships within it. For example, if we are trying to model an automobile, the classifiers we list might include a wheel, an engine, a radio, a tuner, a volume control, a dial, etc. The relationships might include "a radio *has-a* tuner" and "a volume control *is-a* dial".

Concepts and clusters are specified using BON class and cluster charts. The BONc compiler checks that every concept mentioned in a system specification has a definition and a cluster, and that all clusters are organized into a hierarchy. It also checks the high-level type structure of concepts and clusters for type and containment cycles.

Figure 1 shows a BON class chart for an extremely simplified North American FM radio tuner, including the queries, commands and constraints described in the next section. In an actual implementation we would likely use inheritance to constrain a generic FM tuner (a North American FM tuner *is-a* FM tuner); for space reasons, we do not do so here.

class_chart TUNER
query
"What station are you tuned to?",
"What is the station assigned to preset *the_preset*?",
"What is the frequency of station *the_station*?"
command
"Tune to station *the_station*!",
"Tune to preset *the_preset*!",
"Assign station *the_station* to preset *the_preset*!"
constraint
"A station preset is an integer from 1 to 8 inclusive.",
"A station is an integer from 200 to 300 inclusive.",
"The frequency of station *the_station* is
$((\mathit{the\_number} \ / \ 5) + 47.9)$ MHz."

Figure 1. BON class chart for an FM tuner.

3.2. Queries, Commands and Constraints

Once we have our vocabulary and our set of relationships, we identify the queries, commands, and constraints associated with each concept. A *query* is a question that a concept must answer, such as "What station are you tuned to?"; a *command* is a directive that a concept must obey, such as "Assign station *the_station* to preset *the_preset*!"; and a *constraint* is a restriction on query responses or command contexts, such as "A station preset is an integer from 1 to 8 inclusive." Composite query/commands (or query/queries) such as "Tune to preset *the_preset* and tell me the station you were previously tuned to!" are not allowed.

Each query, command, and constraint must be a simple English sentence written using a restricted vocabulary: the concepts identified in the concept analysis step; numbers and basic arithmetic operators; comparison terms ("at least", "at most", etc.); quantification terms ("every", "any", "some", etc.); articles; and some common nouns and verbs. Each query ends with a question mark, each command with an exclamation point, and each constraint with a period. These requirements help to eliminate ambiguity in the process of defining queries, commands and constraints; because each must be a simple English sentence, each must describe a small, manageable piece of functionality, and it is clear exactly what each does. The strict separation between queries and commands—reinforced by the fact that one cannot write a composite query/command as a simple English question or exclamation—makes verification easier, because it explicitly differentiates operations that can cause changes to system state from operations that cannot.

3.3. Refinement of Concepts

After identifying all the queries, commands, and constraints, we refine our concepts into appropriate module-like constructs (such as Java packages) and type-like constructs (such as Java classes and JML model classes). The refinement targets are nearly always constrained by one's environment (team, company, process, etc.); Java and JML constructs are not the only possible choices for use with our method—we could use C# classes instead, or refine to B specifications or Z schemas—but they are the ones we have developed and used the method with, and thus the ones we describe here.

```
/** @return What is the frequency of station
  the_station? */
//@ requires 200 <= the_station & the_station <= 300;
//@ ensures \result == (the_station / 5) + 47.9;
public /*@ pure @*/ float frequency
  (final int the_station) {
  //@ assert false;
  assert false;
  return 0.0f;
}

/** Tune to station the_station! */
//@ requires 200 <= the_station & the_station <= 300;
//@ ensures station() == the_station;
public void tuneToStation(final int the_station) {
  //@ assert false;
  assert false;
}
```

Figure 2. Method signatures with JML specifications.

Clusters refine to module-like constructs. With JML and Java, they typically refine to Java packages. If aggregating the higher-level concepts makes sense from a maintenance perspective, a set of clusters can refine to a single package. For example, having a package that contains only a class or two is often more trouble than it is worth.

Concepts, on the other hand, refine to type-like constructs. With JML and Java, they refine to primitive types, interfaces, abstract classes, concrete classes, and (possibly pure or immutable) model classes. Both clusters and concepts may be eliminated or aggregated at this stage, primarily because they refine to implementations that already exist within the system under design, the Java language, or the standard Java class libraries.

Once the Java and JML constructs are identified, the queries, commands, and constraints are transferred into their corresponding constructs as specially-formatted comments. In addition, each construct is given a Javadoc comment, which is transferred from its concept definition. These transfers are currently done using manual cut-and-paste; however, the next release of BONC will automate this functionality. For space reasons, we do not show this step in the transformation here; however, a different example can be found in our previous paper [26].

3.4. Refinement of Queries and Commands

After the Java modules have been identified and appropriate comments inserted for their queries, commands, and constraints, the queries and commands refine to method signatures. Each query or command refines to exactly one associated method signature, named using the standard Java method naming convention (e.g., *assignPreset*); queries refine to method signatures with non-*void* return types, while commands refine to method signatures with *void* return types. The parameter and return types of these methods are chosen exclusively from the set of Java constructs generated by concept refinement.

We give every method a Javadoc comment, which is transferred from the concept definition. The *@return* tag of a query is exactly the original English query ("What station are you tuned to?"), and the method description of a command is exactly the original English command ("Tune to station *the_station*!"). We declare all method parameters *final* and give them names, using only lowercase letters, numbers, and underscores, that start with articles (*the_station*, *a_button*) or are indexed with numbers (*button_1*, *button_2*). This systematic naming helps with verification, as it indicates when verification conditions refer to method parameters vs. methods vs. instance fields.

We fill in all method bodies with exactly the JML assertion *//@ assert false;*, the Java assertion *assert false;*, and, for methods that return values, the Java statement *return null;* or a return of an appropriate default value (such as *false* for *boolean*). This initial method body explicitly signifies that the method has not been implemented (instead of having intentionally been given an empty implementation) and is the "bottom" implementation with respect to refinement. Classes with such methods can be compiled, and the consistency of their specifications can be checked by ESC/Java2; there is a consistency problem with a class specification if ESC/Java2 can prove that a method terminates successfully when its body contains an *assert false* statement.

This refinement step is currently done manually; however, the next release of BONC will automate significant portions of this step.

3.5. Refinement of Constraints

Once the queries and commands have been translated into Java method signatures, we add basic method preconditions and postconditions (for example, to ensure that setter methods correctly modify state based on their parameters) and translate the constraints into JML specifications such as class invariants, class temporal constraints, and method preconditions and postconditions. In addition, we label every query method with the JML annotation "*/*@ pure @*/*", which indicates that the method changes no state. Figure 2 shows the refinement of one query and one command from Figure 1 to Java method signatures and JML specifications. The text in boldface type is added in this refinement step; the remainder is the result of the previous refinement step.

Typically, the refinement of a constraint from an English sentence to one or more JML assertions involves several steps. These may seem somewhat contrived at first blush; however, this refinement technique is the one we actually use in practice, and it quickly becomes second nature to developers who use our process. Like the refinement of queries and commands to method signatures, the refinement of constraints to JML assertions is currently a manual

process; however, the next version of BONc will perform most of this refinement automatically [27].

The first step is to assign software entities (methods, classes, etc.) to the nouns in the sentence. Since each noun is either a concept from the original concept analysis or a basic noun (like a numeric constant), this requires knowledge of what software entity each noun has been refined to. Consider the constraint on our tuner that "A station is an integer from 200 to 300 inclusive." The method "*station()*", an accessor that gives the currently tuned station, is assigned to the noun "station". Similarly, the integer constants *200* and *300* are assigned to the nouns "200" and "300", respectively.

Once the nouns have been replaced by software entities, the next step is to translate the rest of the English sentence into a mathematical expression. In this case, after noun substitution, we have the sentence "*station()* is an integer from *200* to *300* inclusive." We translate this into the expression "$200 \leq station() \leq 300$". We then further translate this expression into valid JML syntax, to arrive at the class invariant "*200 <= station() & station() <= 300*".

The final step is to add method preconditions and postconditions, if required, to methods that are related to the software entities in the invariant. For example, consider the method *tuneToStation(int the_station)*. Its postcondition, *station() == the_station*, ensures that the station has been set properly. Its precondition must therefore constructed in a way such that the invariant will never be violated by establishing the postcondition. In this case, the method parameter *the_station* must be restricted to the valid range of stations, namely "*200 <= the_station & the_station <= 300*". Such preconditions can be identified during analysis, found via a manual mind's-eye weakest precondition analysis, or generated using tools like Houdini and Daikon.

Once all the constraints have been refined from English to JML, we use ESC/Java2 to type check and statically check the resulting behavioral specifications. Thus, we can determine whether there are deficiencies in the specifications before writing any of the executable program code.

3.6. Implementation

The final step in the process, implementation of executable code, takes place only after all method signatures and JML specifications are completed. At this point, the programming is primarily a "fill-in-the-blanks" exercise to fulfill the specifications that are already in place. This involves replacing "unimplemented" method bodies with implementations that fulfill the method specifications, creating appropriate fields and constructors, and (usually) writing a *main()* method. We typically start by implementing simple queries, then move on to more complex methods. This strategy lends itself to incremental correctness; since more complex methods often depend upon less complex ones, having correct implementations of the latter makes for easier correct implementations of the former.

Our naming requirements for methods and method parameters were discussed above; for static fields we use the standard Java convention of naming with *CAPS_AND_UNDERSCORES*, and for instance fields we require all names to start with *my_* and contain only lowercase letters, numbers and underscores. These naming requirements enforce visible distinctions among the names of methods, static fields, instance fields, method parameters, and local variables. This has numerous desirable effects, including completely preventing the hiding of instance fields by method parameters or local variables.

As fields are created and method bodies are filled in, static analysis tools can perform correctness checks on the system even before it is completely implemented. Moreover, automated tests can be generated and repeatedly run as new code is added.

3.7. A Note on Reversibility

An important aspect of our development process is that it is not strictly linear, despite the fact that we have presented it as a series of refinement steps. Every one of these steps is *reversible*, and it is possible (and often necessary) to revise the outputs of any step after they are initially generated. Reversibility means that changes to the artifacts at either end of a refinement can be propagated to the opposite side of the refinement. Depending on the situation, such propagation can happen entirely manually, entirely automatically, or manually with tool assistance.

For example, one may add a new constraint at the constraint refinement step due to new information or new realizations about the system; this causes corresponding changes to the original output of the query/command/constraint generation step and subsequent steps. Similarly, one may realize at implementation that a concept is missing from the concept analysis, that two concepts that were thought be distinct are actually aspects of the same concept, that inheritance or containment relationships that should exist do not (or vice versa), etc. These changes, once made, cause corresponding changes in the concept analysis.

In order for the outputs of all the refinement steps (that is, the design documentation and the implementation) to remain consistent, the tools used in the refinement must support reversibility. Ideally, this should be transparent; for example, when using the EiffelStudio tools for the original BON development method with Eiffel, a change in the Eiffel source code of a system is instantly reflected in its BON specification. The next release of the BONc compiler will have significant support for this kind of analysis and automatic translation. Therefore, while reversibility in our process currently must be done manually, this will change soon as new versions of the tools are released.

4. Process Guidelines

We have developed a number of guidelines for our verification-centric process over the years, focusing on all the artifacts of software development; we use them in our own software projects, including those discussed in Section 5. Many of these guidelines, which originally required significant manual enforcement effort, are now automatically checked by static analysis tools. While these guidelines are prescriptive and have worked well for us, we realize that they will not work in all environments and organizations or for all development efforts.

4.1. Coding and Documentation Standards

Our development process relies heavily on formal specifications. However, capturing essential aspects of a system's design in a clear, concise, precise, declarative fashion is more pervasive than merely rigorous use of JML. We also document design decisions, action items, implementation tradeoffs, and more in code annotations. Our full coding standard is available in a separate document[10] from the KindSoftware Research Group website[11]. Over the past decade this standard has been adopted, in part or in full, by many other research groups and companies. Furthermore, our code annotations have formal semantics, expressed in a categorical logic called kind theory [28].

4.1.1. Documentation and Source Code Standards. Our coding standard requires *all* Java entities—including those that are not part of the public API—to have Javadoc comments. We enforce this requirement with Checkstyle. One reason for this is consistency; having the same requirements for both "internal" and "external" documentation means less ambiguity for developers. Another reason is that IDEs can parse Javadoc comments and display them to developers in appropriate and useful contexts, even if the comments are never used to generate external documentation.

Our coding standard also specifies import guidelines, spacing and bracket placement rules, and method, field, parameter and variable name requirements, all of which are enforced by Checkstyle. This is primarily to impose discipline; we don't presume to assert that one way of placing brackets is inherently superior to another, or that our way of naming instance fields is superior to any other distinctive way of naming instance fields. Rather, we believe that it is necessary both to enforce visual *consistency* in a code base and to mandate visible *distinctions* among the various constructs (static fields, instance fields, etc.).

4.1.2. Code Size and Complexity. Most of the tools we use perform modular checking, but some—due more to technical than theoretical reasons—act on entire systems. For example, `jmlc` takes several minutes on a modern machine to compile ESC/Java2 (a codebase with around 600 annotated classes and around 150K "raw" LOC). Compiling, statically checking overly complex methods, and running full unit tests takes significant time.

Thus, our rules of thumb are that each method should (1) be no longer than (literally) a hands-width on the screen and (2) have a cyclomatic complexity of at most 6. These rules, which are enforced automatically by Checkstyle and PMD, directly impact both program design and refinement, and indirectly impact verification condition generation (and therefore prover performance and behavior).

4.2. Continuous Verification and Testing

We strongly encourage developers to continuously perform builds, run static checks, and execute unit tests. When using command-line tools or manually triggered builds, this means clicking a button or twitching a finger *after every few lines of documentation, specification, or code are written or modified*. Appropriately-configured modern IDEs trigger the tools automatically. For example, in the configuration shipped with the Mobius PVE, if all options are enabled, two compilers and a half dozen static checkers are automatically run *every time a class is saved*. The feedback provided by this continuous verification process allows a developer to quickly find and fix potential issues before they snowball into all-night bug hunting sessions or refactoring flurries.

4.2.1. Verification vs. Testing. Deciding where to focus verification effort is critical to successful adoption of formal methods. Some subsystems simply cannot be verified with a reasonable amount of effort due to a variety of constraints (technical, social, and legal). These may include model complexity, rapidly changing requirements that necessitate major design or code changes (and thus, new specifications and re-verification), or even a lack of source availability. Other subsystems cannot be fully tested due to challenges like state space explosion, which we believe is usually the result of poor API design or limited use of invariants[12]. Static analysis is exactly what is called for in these subsystems.

In general, our rules are: (1) enable all runtime assertion checking, so long as it does not increase execution time or resource utilization beyond reasonable limits; (2) perform full unit testing on all modules that do not have complex external state dependencies; (3) perform static checking of code style, complexity, and common design and programming errors on the entire system; (4) perform lightweight extended static checking (for null pointer dereferences, invalid class casts, array index bounds violations, etc.) on all source; and (5) perform full extended static checking on critical subsystems.

10. http://www.kindsoftware.com/documents/whitepapers/code_standards/
11. http://www.kindsoftware.com/

12. We have no hard evidence for this thesis, which represents an excellent opportunity for future empirical research.

4.2.2. Unit Test Thoroughness. When unit testing in a verification-centric process, it is not immediately clear how much testing is "enough". Attempting to obtain 100% value, branch, or statement coverage is a rabbit-hole we dare not crawl down [29].

However, with specification-based testing tools like JML-JUnit, obtaining 90% coverage of even a large system typically requires only an afternoon or two of work. Since the JML-JUnit tests are automatically generated, they can adapt to changes in requirements and architecture without developer intervention.

We believe that recent work by ourselves and others on identifying interesting data values [30] holds a great deal of promise. A new version of JML-JUnit under development takes such analyses a significant step forward in the form of specification-aware, semantically rich reflective unit test generation.

5. Case Studies

This development process and these tools have been used in numerous case studies over the past several years, a number of which have been published in various fora [4], [31]–[35]. In this section we briefly summarize some of these. Details about others are available in the cited papers, and much more information is available by downloading the actual software systems.

5.1. Student Coursework

5.1.1. The Eindhoven OOTI Course. A postgraduate course in applied formal methods has been taught at TU/Eindhoven for several years. Small teams of students design, implement, test, and verify medium-sized smart-card-based systems. Students verify the applet that runs on the smart-card (in JavaCard [36]) and unit and system test their terminal software. Example systems include digital wallets, "value club" point cards, petrol rationing chits, etc.

5.1.2. Undergraduate Software Engineering at UCD. The undergraduate software engineering project courses at UCD use our process starting from the students' first year. Students work in small teams on a variety of software projects. Team sizes, problem complexity, and program sizes grow as the students mature. Nearly a hundred different systems have been built over the past three years, including everything from a Guinness screen saver (in the first-year course) to new implementations of classic 8-bit Commodore 64 games like "Space Taxi" (in the third-year course). All student projects are open source and available for download from collaborative development environments hosted at UCD or from the authors.

5.2. Production Systems

5.2.1. ESC/Java2. We use our process in the development of the ESC/Java2 tool itself, in the belief that we must actually apply our methods if we intend to recommend them to others. As a result, ESC/Java2 is a robust and well tested piece of software. In addition, because ESC/Java2 is a complex system—comprising around 600 classes for a total of approximately 150K "raw" LOC—our use of the process has provided valuable insights about where performance improvements must be made in the various verification tools.

5.2.2. The KOA System. In 2003, the Dutch Parliament commissioned an Internet-based remote voting system for use by Dutch expatriates. This system, called KOA, was constructed by the SoS group at Radboud University Nijmegen. The system was partitioned into three components: file I/O, graphical I/O, and the "core" data structures and algorithms. Because of time constraints imposed by the Dutch government (the system was to be developed and delivered in only *four weeks*, with only three developers), the full verification-centric process was used only on the core subsystem, though the other subsystems were also given partial JML specifications. The process yielded a working system that has been used in an EU Dutch election, along with a set of nearly 8,000 automatically-generated unit tests with 100% code coverage and a 100% success rate. There are no known bugs in this system, and several followup analyses, both by new research group members and by external teams (e.g., at MIT), have found no errors in the specification or implementation of this system.

6. Related Work

While our language of choice is Java, and our tools of choice are JML and ESC/Java2, there are a number of other languages and tools available that can be used to implement a development process similar to ours. We are unaware, however, of any work that ties together rigorous object-oriented software engineering techniques with continuous verification in as comprehensive a way as we have described.

The Spec# programming system from Microsoft Research [37] includes both an extension to the C# programming language that supports contracts (class invariants, method preconditions and postconditions) and a static program verifier. Spec# can be used to apply our development process to C# applications. However, Spec# does not support some of the advanced modeling capabilities of JML, and lacks support for abstracting refinement to requirements.

SPARK-Ada [38] is a safe subset of the Ada programming language, and a corresponding toolset, designed for developing high-reliability software. Like JML, formal annotations in SPARK are written as specially-formatted comments. The SPARK toolset performs static analysis similar to that

of ESC/Java2, though in a sound fashion. Some of our development process can be applied to SPARK-Ada systems; however, because SPARK has minimal support for the object-oriented constructs of Ada, one must perform refinement in a data and procedural fashion, targeting Ada *modules* rather than *classes*.

The B method and its associated system, both historical/commercial [39] and modern/open source [40], also focuses heavily on a refinement-centric approach to software construction. Our method and process differ from the B approach in two main ways: first, B does not support refinement above the level of an abstract machine, and thus does not abstract refinement to requirements; and second, B focuses on *generating* program code. In contrast, our approach focuses on refinement to *and from* program code, including code that was written from scratch without regard to our refinement semantics, methodology, or process.

SpecWare [41] is an automated software development system that uses rigorous stepwise refinement to transform abstract system models into concrete applications. Like B, SpecWare has little support for bidirectional refinement and no support for refinement to requirements.

7. Conclusion

We have described a complete, verification-centric software development process for sequential Java systems based on the Java Modeling Language and a host of supporting tools. While we have provided several examples of its successful application, we have also discussed two undeniable factors—tool performance, and the fact that many of the tools currently do not support all the features of modern Java—that act to limit its widespread adoption. The tools are rapidly evolving to address these issues, and we continually refine the process as we (and others) develop new verification tools and improve the performance of existing ones. We are optimistic that our process will be applied to modern Java systems in the near future.

While many teams that define new concepts and artifacts for software engineering rarely, if ever, use their own concepts and artifacts in *real systems construction*, we use the verification-centric process described here to develop our own software systems and we enthusiastically teach it in our software engineering classes. We believe that our particular combination of methodology, process, specification languages, and implementation with broad and deep tool support represents one of the most widely used, widely taught, and broadly applied concrete examples of dependable software development available today.

References

[1] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice–Hall, Inc., 1988.

[2] K. Waldén and J.-M. Nerson, *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems*. Prentice–Hall, Inc., 1995.

[3] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, "How the design of JML accommodates both runtime assertion checking and formal verification," in *Proceedings of the International Symposium on Formal Methods for Components and Objects (FMCO) 2002*, ser. Lecture Notes in Computer Science, vol. 2852. Springer-Verlag, 2003, pp. 262–284.

[4] J. R. Kiniry and D. R. Cok, "ESC/Java2: Uniting ESC/Java and JML," in *Proceedings of the International Workshop on the Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS) 2004*, ser. Lecture Notes in Computer Science, no. 3362. Springer-Verlag, 2005.

[5] J. R. Kiniry and D. M. Zimmerman, "Secret ninja formal methods," in *15th International Symposium on Formal Methods (FM)*, Turku, Finland, May 2008.

[6] B. Meyer, *Reusable Software: The Base Object-Oriented Component Libraries*, ser. The Object-Oriented Series. Prentice–Hall, Inc., 1994.

[7] Sun Microsystems, "JavaDoc tool and API," Sun Microsystems, 2002, available via http://java.sun.com/j2se/javadoc/.

[8] E. Bezault, M. Howard, A. Kogtenkov, B. Meyer, and E. Stapf, "Eiffel analysis, design and programming language," ECMA International, Tech. Rep. ECMA-367, Jun. 2005.

[9] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards, "Model variables: Cleanly supporting abstraction in design by contract," *Software—Practice and Experience*, vol. 35, no. 6, pp. 583–599, May 2005.

[10] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, "Beyond assertions: Advanced specification and verification with JML and ESC/Java2," in *Proceedings of the International Symposium on Formal Methods for Components and Objects (FMCO)*, ser. Lecture Notes in Computer Science, vol. 4111. Springer-Verlag, 2006, pp. 342–363.

[11] D. L. Parnas and P. C. Clements, "A rational design process: How and why to fake it," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, pp. 251–257, Feb. 1986.

[12] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, "How the design of JML accommodates both runtime assertion checking and formal verification," in *Proceedings of the International Symposium on Formal Methods for Components and Objects (FMCO) 2002*, ser. Lecture Notes in Computer Science, vol. 2852. Springer-Verlag, 2003, pp. 262–284.

[13] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, Feb. 2005.

[14] P. Chalin, P. R. James, and G. Karabotsos, "JML4: Towards an industrial grade IVE for Java and next generation research platform for JML," in *International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, Toronto, Canada, Oct. 2008.

[15] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," *ACM SIGPLAN Notices*, vol. 37, no. 5, pp. 234–245, 2002.

[16] J. Kiniry and A. Morkan, "Soundness and completeness warnings in ESC/Java2," in *5th International Workshop on the Specification and Verification of Component-Based Systems (SAVCBS)*, Portland, Oregon, 2006.

[17] M. Janota, R. Grigore, and M. Moskal, "Reachability analysis for annotated code," in *Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, Dubrovnik, Croatia, Jun. 2007.

[18] M. Janota, "Assertion-based loop invariant generation," in *1st International Workshop on Invariant Generation (WING)*, Hagenberg, Austria, 2007.

[19] P. R. James and P. Chalin, "Extended static checking in JML4: Benefits of multiple-prover support," in *ACM Symposium on Applied Computing, Software Verification and Testing Track (SAC-SVT)*, Hawaii, Mar. 2009.

[20] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," *ACM SIGPLAN Notices*, vol. 37, no. 5, pp. 234–245, 2002, proceedings of the International Conference on Programming Language Design and Implementation (PLDI) 2002.

[21] M. Clermont and D. Parnas, "Using information about functions in selecting test cases," in *Proceedings of the ICSE 2005 Workshop on Advances in Model-Based Software Testing (A-MOST)*, 2005.

[22] Y. Cheon and G. T. Leavens, "A simple and practical approach to unit testing: The JML and JUnit way," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) 2002*, ser. Lecture Notes in Computer Science, vol. 2374. Springer-Verlag, 2002, pp. 231–255.

[23] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 35–45, Dec. 2007.

[24] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," Digital's Systems Research Center, Tech. Rep. 2000-003, Dec. 2000.

[25] M. Abadi, C. Flanagan, and S. N. Freund, "Types for safe locking: Static race detection for Java," *ACM Transactions on Programming Languages and Systems*, vol. 28, no. 2, Mar. 2006.

[26] J. R. Kiniry and D. M. Zimmerman, "Secret ninja formal methods," in *Proceedings of the Fifteenth International Symposium on Formal Methods (FM)*, 2008, in press.

[27] J. R. Kiniry and F. Fairmichael, "Ensuring consistency between designs, documentation, formal specifications, and implementations," in *12th International Symposium on Component-based Software Engineering (CBSE)*, East Stroudsburg, Pennsylvania, June 2009, to appear.

[28] J. R. Kiniry, "Kind theory," Ph.D. dissertation, Department of Computer Science, California Institute of Technology, 2002.

[29] K. Beck and E. Gamma, "Test infected: Programmers love writing tests," *Java Report*, vol. 3, no. 7, pp. 37–50, 1998.

[30] D. K. Peters and D. L. Parnas, "Using test oracles generated from program documentation," *IEEE Transactions on Software Engineering*, vol. 24, no. 3, pp. 161–173, 1998.

[31] N. Cataño and M. Huisman, "Formal specification of Gemplus' electronic purse case study using ESC/Java," in *Proceedings of Formal Methods Europe (FME) 2002*, ser. Lecture Notes in Computer Science, no. 2391. Springer-Verlag, 2002, pp. 272–289.

[32] H. Meijer and E. Poll, "Towards a full formal specification of the Java Card API," in *Smart Card Programming and Security: International Conference on Research in Smart Cards (E-smart 2001)*, ser. Lecture Notes in Computer Science, I. Attali and T. Jensen, Eds., no. 2140. Springer-Verlag, Sep. 2001, pp. 165–178.

[33] B. Jacobs, M. Oostdijk, and M. Warnier, "Source code verification of a secure payment applet," *Journal of Logic and Algebraic Programming*, vol. 58, no. 1-2, pp. 107–120, 2004.

[34] J. R. Kiniry, D. Cochran, and P. Tierney, "A verification-centric realization of e-voting," in *International Workshop on Electronic Voting Technologies (EVT) 2007*, Boston, Massachusetts, 2007.

[35] C.-B. Breunesse, N. Cataño, M. Huisman, and B. Jacobs, "Formal methods for smart cards: An experience report," *Science of Computer Programming*, vol. 55, no. 1-3, pp. 53–80, Mar. 2005.

[36] B. Jacobs, "JavaCard program verification," in *Proceedings of Theorem Proving in Higher Order Logics (TPHOL) 2001*, ser. Lecture Notes in Computer Science, no. 2151. Springer-Verlag, 2001, pp. 1–3.

[37] M. Barnett, K. Leino, and W. Schulte, "The Spec# programming system: An overview," in *Proceeding of the International Workshop on the Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS) 2004*, ser. Lecture Notes in Computer Science, no. 3362. Springer-Verlag, 2004.

[38] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*. Addison–Wesley Publishing Company, 2003.

[39] J. Abrial, *The B Method: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[40] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin, "An open extensible tool environment for Event-B," in *Proceedings of ICFEM 2006*, ser. Lecture Notes in Computer Science, no. 4260. Springer-Verlag, 2006, pp. 588–605.

[41] Y. Srinivas and R. Jüllig, "Specware: Formal support for composing software," in *Proceedings of Mathematics of Program Construction (MPC) 1995*, ser. Lecture Notes in Computer Science, no. 947. Springer-Verlag, 1995, pp. 399–422.