Secret Ninja Formal Methods

Joseph R. Kiniry[1] and Daniel M. Zimmerman[2]

[1] School of Computer Science and Informatics, University College Dublin, Belfield, Dublin 4, Ireland
kiniry@acm.org
[2] Institute of Technology, University of Washington, Tacoma, Tacoma, Washington 98402, USA
dmz@acm.org

Abstract. The use of formal methods can significantly improve software quality. However, many instructors and students consider formal methods to be too difficult, impractical, and esoteric for use in undergraduate classes. This paper describes a method, used successfully at several universities, that combines ninja stealth with the latest advances in formal methods tools and technologies to integrate applied formal methods into software engineering courses.

1 Enter the Ninja

Software development tools and techniques based on formal methods hold great promise for improving software quality. Unfortunately, many undergraduate computer science and software engineering curricula include no formal methods instruction beyond the introduction of basic concepts such as the assertion and the loop invariant. Moreover, even when formal methods concepts are introduced, they tend not to be used outside of toy examples. Many students and instructors, it seems, believe that the very words "formal methods" imply writing equations on paper for hours on end with no computers in sight. Those who have never used modern formal tools and techniques generally consider formal methods to be irrelevant to "real" computer programming.

Our goal is not only for our students to *use* formal methods in their software design and implementation process, but also for them to *enjoy* doing so. To accomplish this lofty goal, we employ *shinobi-iri*[1] (stealth and entering methods)—we sneak up on our blissfully unaware students, slip a dose of formal methods into their coursework and development environments, then with a thunderclap disappear in a puff of smoke.

We teach our students to design software systems using a (formal) notation that appears to be merely structured English, and to implement them using sophisticated tool support that is almost entirely hidden behind simple instructor-provided scripts and IDE plugins. Details about the automated theorem proving,

[1] The terminology of the ninja may be inscrutable to the uninitiated; the curious reader may more intensively exercise his *chōhō* (espionage skill) at http://en.wikipedia.org/wiki/Ninjutsu.

J. Cuellar and T. Maibaum (Eds.): FM 2008, LNCS 5014, pp. 214–228, 2008.

static code analysis, runtime assertion generation, and other processes underlying the system are not revealed to the students until after they have implemented software projects. By the time our initiates realize they are using "formal methods" in developing their software, they have experienced firsthand what formal methods can do for them, and are likely to continue to follow in their masters' silent footsteps.

Over the past 10 years we have used this approach to varying degrees, with considerable success, in classes taught at the California Institute of Technology, Radboud University Nijmegen, University College Dublin, and the University of Washington, Tacoma. And, while we are aware of the formal methods teaching literature [1], we and the colleagues with whom we have corresponded about teaching know of no other academics that combine the tools and techniques described herein to practice formal methods ninjutsu in their classrooms.

2 The Ninja Arts

A formal methods ninja has many subtle and effective techniques at his command. We use only a few of these in our classrooms. In this section, we briefly introduce the formal methods concepts we use and then discuss the tools we employ in the software development process. For more details on any of these formal methods or tools, please see the cited sources.

2.1 Formal Methods

Assertions. The assertion [2] is a core concept we use and emphasize in software design and development. Our notion and use of assertions are much broader than just formal assertions in program code; we also classify informal documentation of conceptual constraints and compiler pragmas [3], both of which are encoded as semantic properties in BON and JML (see below), and logging messages as forms of assertions.

Design by Contract. Design by Contract (DBC) [4] is a design technique for object-oriented software that uses assertions to document and enforce restrictions on data and specify class and method behavior. Contracts are used throughout the *entire process* of creating a software system, from analysis and design to implementation and maintenance.

BON. The Business Object Notation (BON) [5] is an analysis and design method for object-oriented software originally developed for use with the Eiffel programming language. We use an extended version of BON (EBON) that folds user-defined domain-specific languages into BON. We use BON instead of UML because BON is simple and has a clear semantics.

JML. The Java Modeling Language (JML) [6] is a specification language for Java programs. It is used both to write class and method contracts in a DBC

style and to specify properties beyond simple partial correctness of method specifications and class invariants. We use JML both because its Java-like syntax is easy for students to learn and because it has excellent tool support.

Underlying Semantics. Underlying the concepts in our realization of DBC, BON, and JML is a rich set of semantics embodied in several logics and tools. A detailed discussion of these semantics is beyond the scope of this paper; however, we will highlight how some of them are naturally expressed to the students in Section 3.

2.2 Tools and Technologies

The main "hook" that we use to get students interested in and excited about trying new development techniques is *tool support*. In fact, we consider the existence of rich, high quality, automated tools *mandatory* for any kind of real adoption of applied formal methods. Moreover, such tools must be integrated into development environments with which students are already familiar.

The tools that we use include some that we have helped develop and some from other teams with which we have little interaction. The former is motivated not by selfishness, but by the principle that we should "eat our own *mochi*."[2] We believe that one cannot propose and provide a tool to the software developers of the world unless one *at least* uses the tool himself, preferably in the tool's own development.

Common JML Tools. The *Common JML* tool suite [6] contains several tools, nearly all of which we use in teaching. The tool suite includes a JML typechecker (`jml`), a JML compiler (`jmlc`) that compiles JML specifications into runtime checks, a runtime assertion checking environment (`jmlrac`), an augmented version of Javadoc (`jmldoc`) that generates browsable documentation containing specifications, and a unit test generating framework (`jmlunit`, discussed below).

Although these tools are quite easy to use, as `jmlc` behaves very much like `javac`, `jmldoc` very much like `javadoc`, etc., our young apprentices need not learn their details as we provide pre-defined build configurations (using GNU Make, Ant, and Eclipse) for them to use.

For example, a freshman programmer need only type `make build` in a shell to generate a full runtime assertion checking build of his project. The same applies to the GNU Make targets `test`, `docs`, etc. Similar targets exist in the predefined Ant build scripts and Eclipse build configurations. Using these predefined build targets, students catch errors in their programs early and often, in reliable and repeatable ways.

In addition, students get to see their hard work on writing documentation and specifications published in an attractive format for all the world, or at least

[2] Others "eat their own dog food". We prefer *mochi*, a delicious Japanese treat made of glutinous rice pounded into paste and molded into shape.

their fellow students, to see. In fact, publishing documentation in this fashion sometimes initiates intra-class rivalries where different teams try to "out-doc" each other, delving into the use of more sophisticated code and documentation presentation mechanisms such as MathML in Javadoc, fancy hyperlinked source processors like Doxygen, etc.

JUnit with JML. Using the Common JML tool `jmlunit`, one can generate arbitrary numbers of different unit tests for an annotated API. Contracts are used as test oracles and data values are identified manually by the developer.

In some of our courses, when appropriate, we generate tests for the students to use and simply provide a build system with a `test` target. The students do not know how these (thousands of) tests are generated, nor do they really care... at first. All they care about is that, by running the automated tests occasionally, they know what piece of code is responsible for a given test failure (as they are taught that precondition failures are the fault of the caller and postcondition failures are the fault of the implementer), and can more easily find and fix bugs.

This style of automated project evaluation is the first example of how we align assessment and project process, development methodology, and code quality in our teaching. Through the use of such stealthy alignment, students are inclined to take "suggestions" like full documentation coverage seriously, as not doing so impacts their grades.

ESC/Java2. The problem with relying solely upon runtime checking and unit testing, even in the presence of tens of thousands of tests, is that one can neither test for the absence of errors nor test a subsystem that is not yet completely implemented. We believe that students need feedback on the quality and correctness of their system's architecture and specified behavior *before* the implementation is complete. To achieve this, the true formal methods ninja reaches into his *shinobi shokozu*[3] for static checkers.

ESC/Java2 is an extended static checker for Java [7]. It *statically* analyzes JML-annotated Java modules (classes and interfaces)—it does not run the code, but instead checks the code and its annotations at compile time. Its capabilities are twofold: (1) it identifies common programming errors like null pointer dereferences, class cast exceptions, out-of-bounds array indexing, etc.; (2) it performs lightweight full-functional verification, ensuring that program code conforms to (sometimes quite rich) specifications written in JML. That is, ESC/Java2 modularly and statically checks that each method body fulfills its contract.

We note a few points about our use of ESC/Java2. First, students run the tool unknowingly, via a build system, exactly as they run the JML compiler and other tools. Second, students are encouraged to run this tool early and often, as static checking is modular and does not depend upon having a running system. Finally, students do not know what kind of (very complex) analysis is being rapidly and efficiently performed in the background; they know only that error

[3] The traditional garb of the ninja, which we wear when giving all our ninja-related conference talks.

messages that look exactly like those produced by `javac` or `gcc` are displayed on their screens or, if they are using Eclipse, that problem markers and red squiggles dynamically appear in their editors.

The fact that ESC/Java2 is carrying out weakest precondition and strongest postcondition reasoning on a Hoare logic using several different automated theorem provers sails over the students' heads like a errant *shuriken* (throwing star). This *hensōjutsu* (disguise and impersonation technique) is highly effective, and only when a student wonders aloud or asks in class how this build rule performs its magic do we begin to reveal the true nature of the connections between this tool and the seemingly highly abstract "nonsense" the student may have witnessed in theory courses.

It is essential that we carefully approach this dialogue. Subtlety is critical, as pushing this formal material, or its connection to the tools being used, too hard can cause the students to crack. It is better to let the advanced students ask the questions, investigate the material on their own, and espouse the ideas, methods, and tools to their less enthusiastic fellow students. Finding the fulcra in the classroom is critical to developing the students' *juhakkei* (ninja skills).

Moodle. We use a *Moodle*-based Virtual Learning Environment (VLE) in our teaching[4]. We use our Moodle servers for typical course-related purposes: posting lecture slides; hosting web and email forums for discussing course organization and concepts; providing a course calendar for scheduling lectures, special tutorials, and instructor/teaching assistant/student group collaborative hack sessions; posting, collecting, and grading homework assignments; and referencing supplementary materials like book lists and tutorials. However, we also integrate our VLE with our collaborative development environment (see below) and our development methodology.

In particular, we use two Moodle components in an integrated fashion. First, we use the Moodle's wiki module to document and evolve the class project's co-analysis and co-design (see the discussion in Section 3 for more information). Second, we use the Moodle's support for automatic implicit dictionary entry hyperlinking to document and cross-reference all concepts identified during the analysis phase of our software development method. The result of these two approaches is that, at any point in time, a student or instructor can: (a) browse the current project architecture, or any previous version thereof, in the wiki; (b) make updates and proposals directly in the wiki; and (c) jump to a single consistent set of concept definitions, as written during concept analysis.

GForge. All project development is managed via a web-based Collaborative Development Environment (CDE). We have used a GForge server for the past several years[5]. CDEs like the GForge provide a variety of services including web forums, email lists, version control repository management and browsing, user

[4] Our Moodles are all available via the KindSoftware research group website.

[5] The UCD GForge server that contains hundreds of student projects is http://sort.ucd.ie/.

polls, a ticket tracker (for features, bugs, patches, etc.), release and download services, etc.

Students are taught not only how to wield a CDE and its critical dependent services (especially version control and ticket tracking), but also how to integrate these practices with their groups' work. In particular, we have an extensive code standard [8] with domain-specific code annotations that students use to communicate about, and through, their system artifacts.

For example, in addition to Javadoc annotations (which are used extensively), we provide special pragmas written in a familiar Javadoc-like syntax. These special pragmas include everything from informal markup (such as copyright, version information, and bug and feature tracking cross-references) to formal annotations about concurrency semantics and time and space complexity.

The students feel like they are just writing normal Javadoc-like documentation. This is our ninja 気 (*qi*, a kind of "life force" or "spiritual energy") flowing through the students, mesmerizing them into believing they are doing something quite simple. In fact, these annotations have formal semantics that are statically checked by the tools we supply.

In summary, by integrating our VLE, our CDE, and project analysis, design, development, deployment, and maintenance, we more deeply engage the students and accurately measure (and potentially reward) their participation, as our VLE and CDE both stealthily track user actions in great detail.

Eclipse and its Plugins. Students are encouraged to use rich editors and development environments. In fact, they receive lectures on, and homework about, the classic yin and yang of `emacs` and `vi`. But many students, in the end, use Eclipse. Thus, we provide a rich set of pre-configured Emacs features and Eclipse plugins and align assessment with the regular use of these tools. In particular, we use plugins for evaluting code standard conformance (CheckStyle), code complexity analysis (NCSS and, in future classes, Metrics), and source and bytecode-level design and implementation analysis to find common programming errors (PMD and FindBugs).

Each of these plugins provides interactive feedback to students, who are easily dazzled by such *kayakujutsu* (pyrotechnics and explosives), while subtly and stealthily training them to use better design and programming practices. Little do they know that our plugins are "tuned" with an eye toward developing *formally verifiable* software. That is, the rigorous practices that our students follow, and the results against which they are assessed, are those necessary to develop robust, reliable, dependable software of very high quality.

Other Common Software Engineering Tools. A number of other concepts, tools, and practices are introduced, with complementary homework assignments, in our courses. As mentioned earlier, build systems like GNU Make and Apache Ant are used. Version control systems like RCS, CVS, and Subversion are critical, and thus introduced early in the semester so that all homeworks can be stored, and sometimes submitted, via commits. Also, unit testing frameworks like JUnit are used.

The aforementioned assessments encourage students to learn about each of these tools. Additionally, as previously stated, student inertia is overcome by precise *bōryaku* (military strategy) in the form of pre-written build system specifications, initial extensive (but not complete) unit tests, and pre-configured version control repositories. This encourages our students to follow the *Way of the Formal Methods Ninja*.

Reflecting on Our Technology Choices. While some of these choices in concepts, tools, technologies, and languages are predictable, many are also surprising. Why not use UML instead of EBON? Why not use Eiffel instead of Java?

Most applied formal methods ninjas have extensive experience with these alternative choices, and these weapons are indeed found in our *dōjō*. However, while we would love to use, for example, Eiffel in instruction, all ninjas have limitations imposed by the local *daimyō* (i.e., the head of the department). Moreover, some choices, at least in the domain of rigorous software development, are simply poor ones, and we avoid them.

3 Ninjutsu in the Classroom

Every ninja knows that his choice of weapon must be appropriate for the situation at hand; a bad choice can mean the difference between victory and defeat. The software development process we teach our students illuminates the right situations and wrong situations in which to use each tool and technique previously described. In this process, no executable code is written until *after* the important engineering work has taken place.

Our process is derived from the BON process [5], but has been modified over the years with an aim toward developing verifiable software, and is conducted as *co-analysis* and *co-design*—rather than lecturing at our students, we run interactive analysis and design sessions with active student involvement. Students propose and argue over terminology (Section 3.1), debate the best informal interface (Section 3.2) and type specification (Section 3.4) for each concept, and argue over appropriate formal specifications (Section 3.5). The remainder of this section describes our six-fold path of software development and gives an example of a single software concept as it travels the path. This concept is necessarily limited in complexity so that we can, within this manuscript, depict multiple steps of its journey; for further, more complex application examples, our course websites and example projects are available online.

3.1 Concept Analysis

The first step in the process, *concept analysis*, involves identifying and naming the important concepts (also sometimes called *entities*, *properties*, or, most often,

classifiers) in the desired software system and collecting them into *clusters*, sets of related classifiers. We explicitly do *not* use the word "class" at this stage, because we want the students to think about basic concepts rather than about software artifacts such as classes, interfaces, and objects. In fact, students are forbidden from using words like "class", "variable", "array", and "loop".

We ask the students to analyze things from the real world, such as desks and automobiles. In a recent class (the video of which is available on the course website), the students analyzed a desk and identified several important associated concepts: a leg, a top, a drawer, a knob (for the drawer), screws, etc.

A more complicated system than a desk (such as a cellular automaton simulator or a Tetris-like game, both examples that we have used in our classes) requires more concepts. At this stage of the process, the goal is to devise a set of classifiers that is *as small as possible* while capturing all the important concepts of the desired system.

3.2 Queries, Commands and Constraints

Once the students have devised a set of concepts for their system, the next step is to identify the queries, commands, and constraints associated with each concept. A *query* is a question that an concept must answer, such as "How tall are you in feet?"; a *command* is a directive that a concept must obey, such as "Open your drawer!"; and a *constraint* is a restriction on query responses or command contexts, such as "A desk must be made of at least one material." or "An open drawer cannot be opened." Composite query/commands (and query/queries) such as "Lock your drawer and tell me whether it was already locked!" are not allowed.

Students identify the queries, commands, and constraints for each concept as simple sentences using a restricted English vocabulary. This vocabulary includes the following: the concept names identified during concept analysis; numbers; comparison terms ("at least", "at most", etc.); articles; and some common nouns and verbs. Each query must end with a question mark, each command with an exclamation point, and each constraint with a period. When reading queries and commands aloud in class, this punctuation is overemphasized to drive the point home. This reinforces the fact that composite queries and commands are forbidden, because such mixed constructs cannot be written as simple English questions or exclamations.

By the end of this co-analysis step, the students and the instructor (recall that much of this process is performed initially *with* the instructor, thus *co*-analysis) have identified queries, commands, and constraints for every concept. Similar to the concept analysis, the goal is to have as few of these as possible while capturing the important characteristics of the concepts. Figure 1 shows a set of queries, commands, and constraints for a simple desk with a single drawer. This is by no means the only possible such set of queries, commands, and constraints; for instance, the length, width, and height are (roughly) what we consider "useful" sizes for a desk.

Queries

How tall are you in feet? / How wide are you in feet? / How deep are you in feet? / What materials are you made of? / Is your drawer open? / Is your drawer locked?

Commands

Open your drawer! / Close your drawer! / Lock your drawer! / Unlock your drawer!

Constraints

A desk must be between 2 and 8 feet tall. / A desk must be between 2 and 20 feet wide. / A desk must be between 2 and 8 feet deep. / A desk must be made of at least one material. / An open drawer cannot be opened. / A closed drawer cannot be closed. / A locked drawer cannot be opened.

Fig. 1. Queries, commands, and constraints associated with a simple desk

3.3 Java Module Skeletons

After identifying the queries, commands, and constraints, it is finally time for the students to start using a programming language, which at our current universities, for good or ill, is Java. However, students do not write any executable code at this stage. Instead, the concepts identified during analysis are refined into Java modules (classes, abstract classes, and interfaces, as appropriate) and primitive types, and clusters are refined into Java packages. Only a subset of the concepts identified in the first stage are refined into Java modules—there is rarely a one-to-one correspondence between concepts and module skeletons by the end of this step.

The queries, commands, and constraints associated with each concept are (literally) cut-and-pasted into the appropriate module as specially-formatted comments. Every module created in this step also has a Javadoc comment, which is likewise cut-and-pasted from its concept's definition.

Our simple desk concept is refined into a Java class *SimpleDesk*, and we assume the existence of a Java module for *Material*. We do not need new Java modules for the dimensions, which are represented by existing primitive types (e.g., `float`, `double`). Figure 2 shows the Java class skeleton for *SimpleDesk*.

3.4 Method Signatures

Having created the Java modules, the students move on to writing method signatures for each concept. Each query or command has exactly one method signature associated with it. Method signatures associated with queries must have non-`void` return types, and method signatures associated with commands must have `void` return types. The parameter types and return types of the methods

```
package formalmethods.ninja.furniture;

/**
 * A representation of a desk with a single drawer.
 *
 * @author Daniel M. Zimmerman
 * @author Joseph R. Kiniry
 * @version 9 November 2007
 */

public class SimplifiedDesk
{
  // @bon query

  // @query How tall are you in feet?
  // @query How wide are you in feet?
  // @query How deep are you in feet?
  // @query What materials are you made of?
  // @query Is your drawer open?
  // @query Is your drawer locked?

  // @bon command

  // @command Open your drawer!
  // @command Close your drawer!
  // @command Lock your drawer!
  // @command Unlock your drawer!

  // @bon constraint

  // @constraint A desk must be between 2 and 8 feet tall.
  // @constraint A desk must be between 2 and 20 feet wide.
  // @constraint A desk must be between 2 and 8 feet deep.
  // @constraint A desk must be made of at least one material.
  // @constraint An open drawer cannot be opened.
  // @constraint A closed drawer cannot be closed.
  // @constraint A locked drawer cannot be opened.
}
```

Fig. 2. Java class skeleton for a simple desk

are chosen from among the previously-created Java module skeletons and the core Java libraries and primitive types.

Every method has a Javadoc comment, which is written entirely using cut-and-paste. The `@return` tag of a query is exactly the original English query ("What materials are you made of?"), and the method description of a command is exactly the original English command. Method parameters, if any, are named starting with articles (`the_width`, `a_material`) or indexed with numbers (`material_1`, `material_2`). All these guidelines are automatically checked against our code standard using the aforementioned static style checker.

At this stage, each method body consists of exactly the following: (1) the JML assertion `//@ assert false`; (2) the Java assertion `assert false`; and (3) for methods with return types, the Java statement `return null` (or a return of an appropriate default value of a primitive type, such as `0` for integral types). This default body enables the classes to compile before the methods are implemented, and also allows our tools to properly analyze the methods, as this initial implementation both signals that the method has not been implemented

(differentiating it from a legal empty implementation) and is the "bottom" implementation with respect to refinement. Of course, our initiates do not know or understand these theoretical subtleties; they merely know that this is a very practical way to generate method stubs.

Figure 3 shows the *SimpleDesk* class with method signatures; for space reasons, we omit some of the methods.

3.5 JML Specifications

The method signatures from the previous step are the translation of the queries and commands into Java. The next step is the addition of JML specifications in the form of basic preconditions and postconditions on methods, and the translation of the BON constraints into JML invariants. Also, every query is labeled with the JML annotation "`pure`", which indicates that the method does not change any system state.

For example, the constraint "A desk must have at least one leg." (on a more complex desk class than the one in our example) might be translated into both a class invariant (`0 < numberOfLegs()`) and a precondition on the method `removeLeg` (`1 < numberOfLegs()`). These specifications are written collaboratively (remember, this is *co*-design); young students are only expected to be able to read them, while older, more advanced students are expected to be able to write them as well.

Figure 4 shows the *SimpleDesk* class from Figure 3 after JML specifications have been added.

3.6 Method Bodies and Fields

The final step, which takes place only after all method signatures and JML specifications are completed, is when the students, working individually or in teams, finally get to write executable code. They take this step without our direct involvement.

At this point, programming is something of a fill-in-the-blanks exercise. All the students need to do is write code in each method to fulfill the specification, concretize fields to represent essential data, and (optionally) write a `main()` method somewhere to actually run the system. They are encouraged to implement methods in a bottom-up fashion, focusing on "leaf" methods and simple queries first and complex methods later.

Recall that students have thousands of pre-generated test cases as well as tools like ESC/Java2 at their disposal. They are encouraged to regularly run these tests and tools as they write their code. In the vast majority of cases, the code that students write at this stage "just works" on the first try; this is a very different result from the code written in most introductory software engineering classes, and gives students a concrete sense of accomplishment.

```
package formalmethods.ninja.furniture;

import java.util.Collection;

/**
 * A representation of a desk with a single drawer.
 *
 * @author Daniel M. Zimmerman
 * @author Joseph R. Kiniry
 * @version 9 November 2007
 */

public class SimpleDesk {
  // @bon query

  /** @return How tall are you in feet? */
  public float height() {
    //@ assert false;
    assert false;
    return 0.0f;
  }

  // width() and depth() are symmetric with height()

  /** @return What materials are you made of? */
  public Collection<Material> materials() {
    //@ assert false;
    assert false;
    return null;
  }

  /** @return Is your drawer open? */
  public boolean isDrawerOpen() {
    //@ assert false;
    assert false;
    return false;
  }

  // isDrawerLocked() is symmetric with isDrawerOpen()

  // @bon command

  /** Open your drawer! */
  public void openDrawer() {
    //@ assert false;
    assert false;
  }

  // closeDrawer(), lockDrawer(), and unlockDrawer() are symmetric with openDrawer()

  // @bon constraint

  // @constraint A desk must be between 2 and 8 feet tall.
  // @constraint A desk must be between 2 and 20 feet wide.
  // @constraint A desk must be between 2 and 8 feet deep.
  // @constraint A desk must be made of at least one material.
  // @constraint An open drawer cannot be opened.
  // @constraint A closed drawer cannot be closed.
  // @constraint A locked drawer cannot be opened.
}
```

Fig. 3. Java class skeleton with method signatures for a simple desk

```
package formalmethods.ninja.furniture;

import java.util.Collection;

/**
 * A representation of a desk with a single drawer.
 *
 * @author Daniel M. Zimmerman
 * @author Joseph R. Kiniry
 * @version 9 November 2007
 */

public class SimpleDesk {
  // @bon query

  /** @return How tall are you in feet? */
  public /*@ pure */ float height() {
    //@ assert false;
    assert false;
    return 0.0f;
  }

  // width() and depth() are symmetric with height()

  /** @return What materials are you made of? */
  //@ ensures \result.size() >= 1;
  public /*@ pure */ Collection<Material> materials() {
    //@ assert false;
    assert false;
    return null;
  }

  /** @return Is your drawer open? */
  public /*@ pure */ boolean isDrawerOpen() {
    //@ assert false;
    assert false;
    return false;
  }

  // isDrawerLocked() is symmetric with isDrawerOpen()

  // @bon command

  /** Open your drawer! */
  //@ requires !isDrawerOpen();
  //@ requires !isDrawerLocked();
  //@ ensures isDrawerOpen();
  public void openDrawer() {
    //@ assert false;
    assert false;
  }

  // closeDrawer(), lockDrawer(), and unlockDrawer() are symmetric with openDrawer()

  // @bon constraint
  // @constraint A desk must be between 2 feet and 8 feet tall.
  //@ public invariant 2.0 <= height() && height() <= 8.0;
  // (width and depth constraints and invariants are symmetric with height)
  // (other constraints have no corresponding invariants)
}
```

Fig. 4. Java class skeleton with JML specifications for a simple desk

4 Notes from the Dōjō

We now reflect upon some of our choices, successes and failures, and student reactions in our classrooms. To date, we have received primarily informal student feedback through in-class and online anonymous questionnaires, the results of which are public and available via the aforementioned website. The qualitative evidence from this feedback suggests that the training we provide in our formal methods *dōjō* is both well-received and successful. However, we recognize that more quantitative evidence is necessary to refine our training techniques, and are currently undertaking a study to gather data about student adoption of formal methods.

Student reactions to several of our choices have been excellent. As previously mentioned, Java is used in other courses at all the universities where we have used this approach, and the students are comfortable with it. JML feels just like Java with a handful of extra keywords, the tool support for JML with Java 1.4 is very good, and students generally enjoy using our enriched Eclipse and the Moodle online course management system. Also, students seem to enjoy our process and adopt it well, and many use it in subsequent software engineering and design courses.

On the other hand, our tool arsenal is currently lacking in two main respects. First, EBON tool support is poor. We provide a minimal shell script for extracting BON specifications from annotated Java code and use both EiffelStudio and BlueJ for carrying out the initial design stages of our approach. However, these tools are not a perfect fit, as EiffelStudio does not support Java and BlueJ does not support BON. Work is underway on new tools to directly support EBON, and we will quickly adopt these tools once they become available. In fact, the first version of our new BON specification checker, BONc, was released recently and is now being used in our software engineering courses.

Second, because JML does not currently support Java 1.5 language features such as generics, enhanced `for` loops, and autoboxing, the contexts in which we can use the JML tools are more limited than we would like. Students that have already been exposed to these language features are (understandably) reluctant to do without them in order to use the tools. Projects such as JML4 [9] that aim to update JML for use with current Java virtual machines will alleviate this problem in the near future.

In addition to these tool-related shortcomings, we have received significant negative feedback about the user interface of the GForge. We have therefore decided to replace the GForge with a Trac server for the current academic year. Trac's excellent interface and integration of a wiki, a tracker, and version control allow us to eliminate the haphazard use of various suboptimally-realized subsystems in the Moodle (e.g., its wiki) and the GForge.

We have customized our Trac server significantly, using over a dozen plugins to enrich its capabilities for our teaching practices. One of these plugins supports Mylyn, an Eclipse feature for task management that we will use in some classes this year. With Mylyn, students are able to interact with the Trac server directly from within Eclipse. Mylyn also provides support for context-aware, task-focused

software development—a style we have taught previously, but have been unable to enforce.

5 Conclusion

We hope that you, the reader, have not been offended by our ninja metaphors and are, perhaps, intrigued by our unique integration of applied formal methods into undergraduate instruction. We welcome your inquiries, and have made large amounts of quality pedagogical materials available including slides, projects, videos, tutorials, papers, etc. Perhaps you, too, might enter our *dōjō* and adopt the *Way of the Formal Methods Ninja*.

References

1. Dean, C.N., Hinchey, M.G. (eds.): Teaching and Learning Formal Methods. Academic Press, London (1996)
2. Hoare, S.T.: Towards the Verifying Compiler. In: Aichernig, B.K., Maibaum, T.S.E. (eds.) Formal Methods at the Crossroads. From Panacea to Foundational Support. LNCS, vol. 2757, pp. 151–160. Springer, Heidelberg (2003)
3. Grogono, P.: Comments, assertions, and pragmas. ACM SIGPLAN Notices 24(3) (1989)
4. Jézéquel, J., Meyer, B.: Design by contract: The lessons of Ariane, January 1997, pp. 129–130. IEEE Computer Society Press, Los Alamitos (1997)
5. Waldén, K., Nerson, J.M.: Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems. Prentice-Hall, Inc., Englewood Cliffs (1995)
6. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K., Poll, E.: An overview of JML tools and applications. In: International Journal on Software Tools for Technology Transfer (February 2005)
7. Kiniry, J.R., Cok, D.R.: ESC/Java2: Uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
8. Kiniry, J.R.: The KindSoftware coding standard. Technical report, KindSoftware Research Group, UCD (2005), `http://secure.ucd.ie/`
9. Chalin, P., James, P.R., Karabotsos, G.: An integrated verification environment for JML: Architecture and early results. In: Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS), Cavtat, Croatia, September 2007, pp. 47–53 (2007)