Test Wizard: Automatic test case generation based on Design by Contract™

Karine Arnout[1]
Karine.Arnout@inf.ethz.ch

Xavier Rousselot[3]
xavierrousselot@ifrance.com

Bertrand Meyer[1, 2]
Bertrand.Meyer@inf.ethz.ch

[1]Chair of Software Engineering, Swiss Federal Institute of Technology (ETH)
CH-8092 Zurich, Switzerland

[2]Eiffel Software
356 Storke Road, Goleta, CA 93117, USA

[3]Amadeus
Sophia-Antipolis, France

This article is a draft (version 2) reporting on work in progress. Reference: Karine Arnout, Xavier Rousselot, Bertrand Meyer. *Test Wizard: Automatic test case generation based on Design by Contract™*, draft report, ETH, June 2003.

Abstract

Using Design by Contract™ provides a low-cost solution to unit-testing, which is usually unaffordable — and not carried out — in the industry. If contracts — preconditions, postconditions, class invariants — are systematically associated with classes, they provide an invaluable source of information for producing systematic tests, directly based on the software's expressly intended semantics. We present a complete specification of a Test Wizard for contract-equipped classes, after giving a glimpse of our testing approach based on probabilistic game theory.

1. INTRODUCTION

The purpose of software engineering is to build high quality software. "*Quality through components*" ([1], page 2) is a well accepted idea in both the academic and the industrial worlds. To be beneficial though, software reuse [17] must be applied well. Indeed, reusing software elements means increasing both the good and the bad aspects of the software; hence the even greater importance of testing. Robert Binder nicely summarizes the correlation between reuse and software testing in [6] (page 68); he asserts that: "*Components offered for reuse should be highly reliable; extensive testing is warranted when reuse is intended.*"

However, software testing is still not carried out extensively in the industry nowadays. It is reduced to the bare minimum and often regarded as an expensive and not rewarding activity. Companies are content with software that is "good enough" [26] to ship, even if still contains bugs. The NIST (National Institute of Standards and Technology) report on Testing of May 2002 [18] estimates that the costs resulting from insufficient software testing range from 22.2 to 59.5 billion dollars. These figures do not even reflect the "costs" associated with mission critical software where failure can be synonymous of loss of life or catastrophic failure. The report shows that testing techniques and tools are "*still fairly primitive: The lack of quality metrics leads most companies to simply count the number of defects that emerge when testing occurs*".

James A. Whittaker [25] quotes the definition of a software tester given by Karl Wiegers and Dave Card, editors of Nuts & Bolts, which emphasizes — with somewhat folksy words — the underdeveloped state of software testing:

> *"The effective tester has a rich toolkit of fundamental testing techniques, understands how the product will be used in its operating environment, has a nose for where subtle bugs might lurk in the product, and employs a bag of tricks to flush them out."*

Such an acknowledgment of failure shows that there is still a lot to be done in the area of testing. We think that automating even parts of the testing process would facilitate spreading out the practice of testing, especially in the industry. It would lower the costs and avoid overlooking some test cases. We also believe that contracts — as defined in the Design by Contract™ method (see [15], [18], [20], and [21]) — contain a solid information basis to generate black-box test cases automatically. This article explains how we want to proceed.

- Section 2 explains our view on software testing, based on probabilistic game theory.
- Section 3 describes a possible implementation of an automatic tool, which we call the Test Wizard, to generate test cases automatically from the contracts present in Eiffel libraries.
- Section 4 presents some research works exploring the idea of automatic test-case generation based on the software specification.
- Section 5 sketches some further research directions.

2. TESTING: A PROBABILISTIC GAME OF GOOD AND EVIL

2.1 Purpose of testing

Before describing what a "Test Wizard" could be, it is useful to give a more precise definition of "software testing".

A fashionable way of looking at software testing is to regard it as a way to increase the reliability of the software under test. We think that testing is almost not suited for this purpose, because we can only test a minute part of the software. There is no other way than proofs and formal methods to assess the reliability of a software component.

Dijkstra already pointed us to that intrinsic limitation of software testing in 1972 while advocating proof of correctness, asserting that: "*Program testing can be used to show the presence of defects, but never their absence!*" [9].

Falling into step with Dijkstra and Myers [22], we think that:

The only purpose of testing is to find errors.

Meaning that:

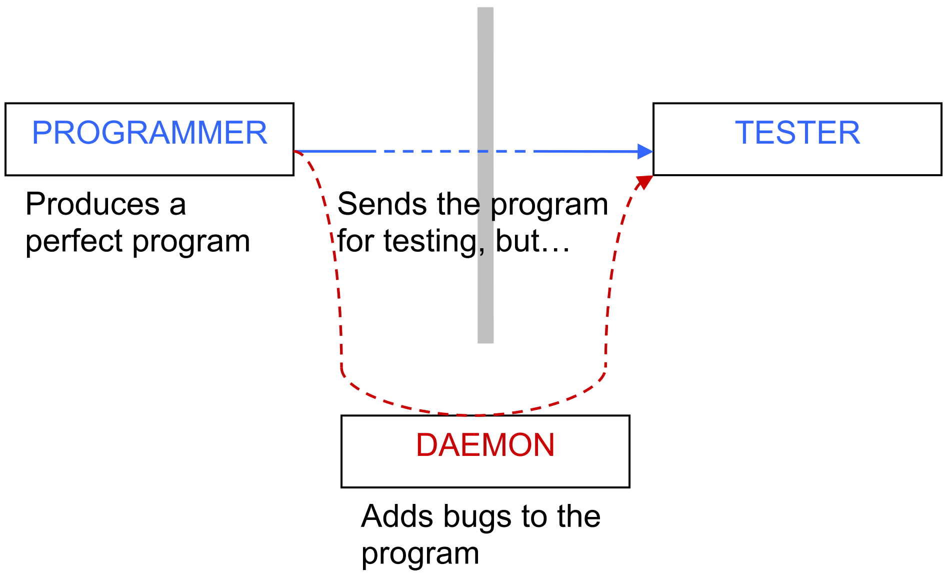
A successful test is a test that fails.

2.2 A Mephistophelean game

We view software testing as a game between the Tester and the Programmer. The goal of the game is to find bugs.

Note that we deviate from Dijkstra's view here: he criticizes the word "bug", whereas we consider it should be the core of any testing theory.

Let's assume that the Programmer writes perfect software, and that bugs are added by a Daemon (a "bug seeder"). Testing becomes a game between the Tester and the Daemon: the Tester (the "Good") defeats the Daemon (the "Evil").


PROGRAMMER
Produces a perfect program
Sends the program for testing, but…
TESTER
DAEMON
Adds bugs to the program

Testing: A game of Good and Evil

If we stop here though, the Daemon wins 100 percents of the time. Let's take an example to make this point clear. Consider a square root function. An Eiffel implementation could be:

```
sqrt (x: REAL): REAL is
        -- Mathematical square root of x, within epsilon
    require
        x_positive: x >= 0
    do
        -- Square root algorithm here
    ensure
        good_approximation: abs (Result ^2 – x) <= 2 * x * epsilon
    end
```

The Daemon can, say every 178567 calls to the *sqrt* function, return a wrong result (randomly generated); the Tester would have no way to find it.

2.3 Probabilistic distribution of bugs

The only way to have a "real" game between the Tester and the Daemon is to introduce probabilities. For example, boundary values tend to be more risky: they would have a higher probability. We talk about the "Mephisto factor":

Definition: Mephisto factor

Likelihood of a test to be successful (meaning that makes the system fail).

Our approach to testing is based on the following conjecture (which has some intuitive appeal, but needs to be proven):

Bug Probability Conjecture

A system that has some bugs is likely to have other bugs.

How to choose the probability distribution? We don't have a concrete answer to this question yet, rather an approach we would like to follow:

The idea is to create an historical database of real bugs encountered during testing some software systems. This "repository of bugs" should highlight, for example, input values that are more likely to catch bugs and, in this way, help us define a heuristic for testing. In fact, we could use such a repository in two ways:

- A *deductive* way: use the list of bugs as an automatic test-case generator.
- An *inductive* way: analyze the list of bugs and find out their probability distribution.

We favor the second way of establishing the probability distribution of bugs; the ultimate goal would be to develop a "theory of bugs" to be able to apply it to different software components. The theory should highlight some resemblance between bugs because all daemons — human developers in practice — tend to think and react the same (human) way, due to laziness, automatisms, negligence, and analogue — not logical — thoughts.

3. TEST WIZARD

3.1 Objectives

The purpose of developing a Test Wizard is to have a workbench to try out different testing strategies. The tool should be highly parameterizable and integrate the notion of testing strategy — which is likely to become an important abstraction during design.

The target of the Test Wizard is contract-equipped libraries — typically Eiffel libraries — because the test-cases are generated automatically from the contracts expressed in the library.

The testbed will be the Eiffel library for fundamental structures and algorithms: EiffelBase (see [10] and [17]). We will use software fault injection, infecting EiffelBase with errors on purpose, to test the Test Wizard and assess its efficiency at detecting bugs with different input parameters (number of requested tests, selected testing strategy, etc.).

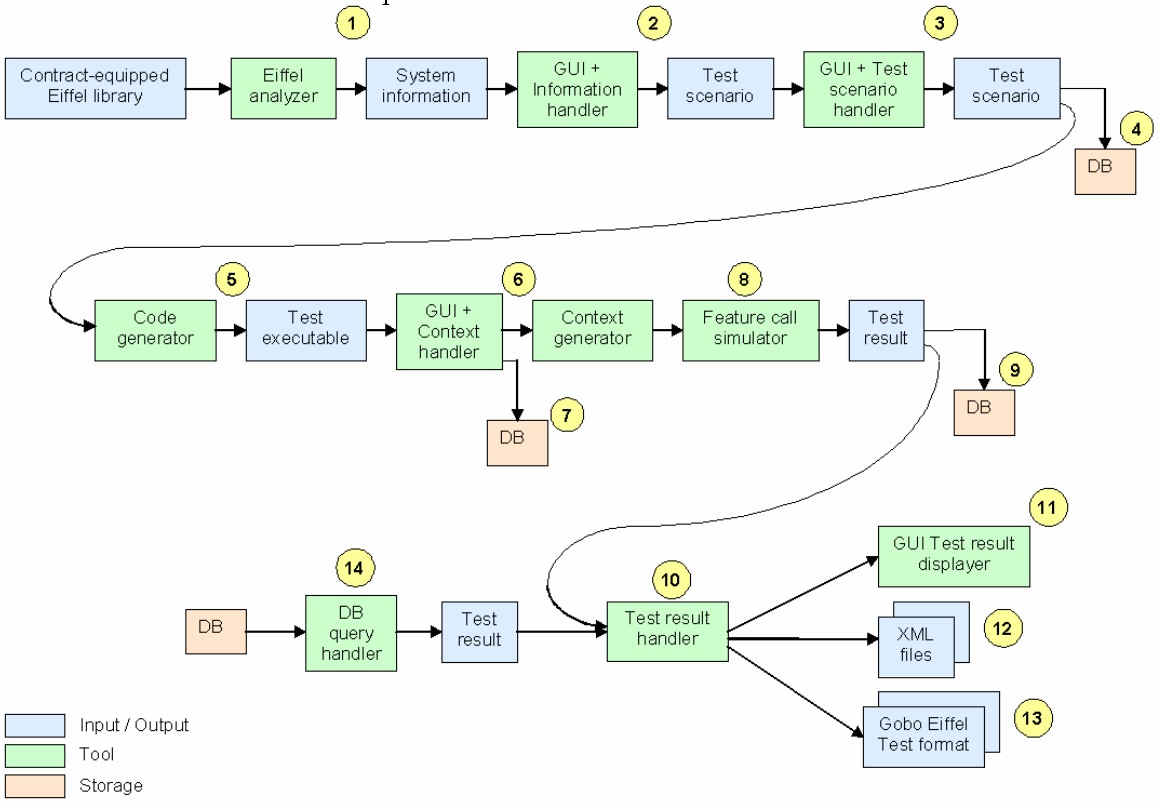
3.2 Architecture of the tool

The Test Wizard takes an Eiffel library as input and automatically generates black-box test cases from the library specification, which in Eiffel is expressed with assertions (preconditions, postconditions, and class invariants). The test results are provided to the users under various forms: graphical representation with diagrams, XML files, and Gobo Eiffel Test [6] files.

The figure on next page shows the internal architecture of the tool. From the Eiffel library to the test results, the Test Wizard follows a five-step process:

- Parsing the contract-equipped Eiffel library provided in input to get the system information (list of classes, features, etc.) to be displayed to the user.
- Gathering user information (list of features to be tested, with what arguments, etc.) to define the test scenario.
- Generating the corresponding test executable.
- Running the test executable and storing the test results into a database.
- Displaying the results to the user.

The next sections cover each step in detail.



Architecture of the Test Wizard

(1) Gather system information.

(2) Display system information to the user and gather user information.

(3) Build the test scenario from the criteria selected by the user.

(4) Store the test scenario into a database (for regression testing).

(5) Generate a test executable corresponding to the test scenario.

(6) Run the executable: it creates a pool of objects (the “Context”) possibly helped by the user.

(7) Store the order of class instantiations (for regression testing).

(8) The executable performs feature calls on the pool of instantiated objects.

(9) Store the test results into a database.

(10) Output the results to the users.

(11) Display the results graphically with diagrams.

(12) Generate XML files corresponding to the test results.

(13) Generate files using the Gobo Eiffel Test [6] format.

(14) Query the database and retrieve test results to be passed to the test result handler.

3.3 Gathering system information

The first step is to gather information about the Eiffel library given as input, namely collecting the list of classes of the system under test, the list of features, and so on. We could write our own Eiffel parser to do this. But we prefer to use a standard parsing tool, which will be maintained and follow the evolutions of the Eiffel language.

Gobo Eiffel Lint (gelint) [5] is a good candidate. It is part of the Gobo Eiffel 3.1 delivery [4], although it is still under development. Gelint is able to analyze Eiffel source code and report validity errors and warnings. The Eiffel code analysis is the part we are interested in. Indeed, it provides all the functionalities of the ISE Eiffel compiler from degree 6 to degree 4 [12]:

- Gelint reads an Ace file and looks through the system clusters to map the Eiffel class names with the corresponding file names (equivalent to ISE degree 6).
- It parses the Eiffel classes (equivalent to ISE degree 5).
- For each class, it generates feature tables including both immediate and inherited features (equivalent to ISE degree 4).

The feature tables built by gelint provide us with the information we need to generate test cases, namely: the list of clusters, the list of classes, the inheritance hierarchy, the export clauses, the list of features, and the feature signatures.

3.4 Defining the test scenario

The second step is to define the test scenario. It requires interaction with the user, and involves two parts numbered (2) and (3) on the previous figure:

(2) The *Information handler* receives the system information built from the analysis of the Eiffel library (see 3.3); it interacts with a GUI where the user can feed test parameters and generates a first shot of the test scenario.

(3) The resulting test scenario is given to the *Test scenario handler*, which interacts with a GUI, where the user can adapt the automatically generated scenario (for example, reorder tasks, add or delete one task, change the randomly generated arguments, etc.).

At the end of the phases (2) and (3), the Test Wizard has a final test scenario, which will be used to create a test executable (see 3.5).

Before looking at the test executable, it is worth giving a few more details about the *Information handler* and the *Test scenario handler*.

Information handler

The Information handler enables choosing the following test criteria:

- *Scope of the test*: which clusters, classes and features should we test? The GUI part associated with the Information handler lists all clusters and proposes to test sub-clusters recursively. Testing a cluster means testing all classes of the cluster. The user can also make a more fine-grained selection and choose some classes only, or even just a few features. The scope selection yields a list of features to be tested.
- *Exhaustiveness*:
  - How many times should we call each feature under test? The tool enables changing the number of calls to be performed at several levels: globally (for all features to be tested), per class (for all features of a class), and per feature.
  - Should we test a feature in descendants? The tool gives control to the user.
- *Context*: The test executable, generated from the test scenario, will create a pool of objects on which to perform the selected feature calls. We call this pool: "the Context". The Test Wizard lets the user decide how classes will be instantiated:
  - *Bounds used for arguments*: The wizard has predefined default bounds for common types, such as basic types (integers, characters, etc.) and strings, and lets the user define bounds for other types. For example, it is possible to specify that a feature returns a given type in a given state.

    - *Objects on which features are called*: The wizard enables the user to choose the objects on which to perform the feature calls.
    - *Creation procedure used for instantiation*: The user can also select which creation procedure to use to instantiate classes, overriding the default procedure the wizard would choose.
    - *Level of randomness of targets and arguments*: From the scope defined by the user, the Test Wizard knows which classes must be instantiated. Depending on the criteria just cited (argument bounds, target objects, creation procedures), the tool will call some modifiers on the created objects to get several instances of each class. The resulting pool of objects makes up the test "Context". The level of randomness fixes the number of variants to be created.
- *Tolerance*: when should we consider that a test has passed?
    - *Rescued exception*: If an exception occurred during a test execution and was rescued, do we say the test has passed or not? This is a user-defined criterion.
    - *Assertion checking*: By default, all assertions are checked on the tested features. Preconditions are checked on the supplier features. Nevertheless, the users may want to disable checking of some assertions to adjust their need. For example, it may be convenient to turn postcondition and class invariant checking off to focus on the bugs that make the library crash, before working on bugs that lead to erroneous results.
- *Testing order*: The user can choose between:
    - *Performing tests on one feature at a time* (i.e. performing all requested calls on a feature before testing the next one).
    - *Performing tests one class at a time* (i.e. calling all features of a class once before performing subsequent calls — if necessary).
    - *Testing as many features as possible* (i.e. calling all requested features once before performing further calls — if necessary), which is the default policy.

The *Information handler* generates a first test scenario from these user-defined parameters and passes it to the *Test handler*.

The GUI part of the *Information handler* is likely to be a "wizard" with a succession of screens where the user can select the input parameters. A sketch of the wizard screens is presented in Appendix B.

Test handler

The *Test handler* outputs the automatically generated scenario and gives the user an opportunity to modify this scenario before the test executable gets created. Flexibility concerns:

- *Call ordering*: The user can modify the order in which the calls are performed.
- *List of calls*: The user can add or remove calls from the list of calls to be performed.
- *Actual calls*: The user can modify the arguments that will be used to perform the calls.

The most common use of the Test Wizard though is to leave the automatically generated scenario unchanged and rely on the defaults. The level of control and parameterization just sketched addresses more advanced users.

3.5 Generating a test executable

The next step is to generate the test executable. It corresponds to number (5) on the figure describing the Test Wizard's architecture (see **3.2**). The *Code generator* generates Eiffel code corresponding to the requested feature calls (defined in the test scenario), and calls the Eiffel compiler, which builds the actual test executable.

The test executable, which is launched automatically, contains the necessary code to create the test "Context" (the pool of randomly generated objects) and to perform the requested feature calls on these objects. It corresponds to steps (6) and (8) shown on the architecture figure (see **3.2**).

Instantiating classes

The *Context generator* takes care of instantiating the needed classes, once the *Context handler* has solved possible problems:

- A class *A* may have a creation procedure that takes an instance of type *B* as argument, and the class *B* may also require an instance of type *A* to be created. To solve most of the problems, the *Context handler* sorts the classes to be instantiated in topological order, defining equivalence classes when there is a circle. For the remaining circles (inside the equivalence classes), the wizard tries to instantiate classes with **Void** instances (for example, try to instantiate *A* with a **Void** instance of type *B*); if it fails, it will prompt the user for a means to create this object.
- Another subtlety deals with the exportation status of creation procedures. If the creation features are exported, there is no problem: the wizard uses any creation feature available, passing various arguments to them. If the creation features are not exported, the wizard asks the user for a means to instantiate the class; if the user does not help, the tool tries to force a call to the creation procedure anyway; in case of failure, it gives up instantiating that class.

Once the *Context handler* has established the order and means to instantiate the needed classes, the *Context generator* actually generates the objects, and calls modifiers randomly on those objects to build the pool of test objects (the "Context").

Calling features

The *Call simulator* performs the required calls. It chooses feature arguments and target objects among the pool of generated objects whose type matches the one needed. All feature calls are wrapped in a rescue block to handle possible exceptions occurring during the test execution. The exception analysis carried out in the rescue block distinguishes between five cases:

- No exception caught
- Exception raised and caught internally
- Precondition violation (in the tested feature or at a higher level)
- Other assertion violation
- Other exception

We explain the policy of the Test Wizard regarding exceptions in the next section (**3.6**).

3.6 Outputting test results

The Test Wizard reports to the user after each feature execution. This "real-time" feedback becomes very important when the user wants to run tests for hours. Two questions arise:

- When do we consider that a test has passed?
- What information should be reported to the user?

Test result: passed or not?

Results are obtained at feature level. They are classified in four categories:

- *Passed*: At least one call was allowed (not breaking the feature precondition) and all the effective calls raised no exception or fit the user definition of "pass". (The Test Wizard enables the user to say whether to take rescued exceptions into account.) Typically, a feature has passed if all effective calls ended in a state satisfying the feature postcondition and the class invariants.
- *Could not be tested*: It may happen that no call to this feature can be issued, because the target object or the object passed as argument cannot be instantiated.
- *No call was valid*: All calls resulted in a precondition violation. Thus, the function never failed, but never passed either. (Because we issue calls randomly on the objects of the pool, it is perfectly normal to get precondition violations; such calls are simply ignored.)
- *Failed*: At least one call to the feature failed. A call fails if it raises an uncaught exception or if it fits the user definition of "fail". Indeed, the Test Wizard lets the user choose whether to accept rescued exceptions as correct behavior (see the tolerance parameter of the *Information handler* in section **3.4**).

Result display

A test result includes the following information (depending on the result category shown before):

- *Passed*: The test result gives the total number of successful calls to this feature.
- *Could not be tested*: The test result explains why this feature could not be tested:
    - It was impossible to instantiate the target class (because there was no exported creation procedure, or the creation feature used keeps failing, or the class is deferred).
    - It was impossible to instantiate a class used in the feature arguments (same possible reasons as above).
- *No call was valid*: The test result gives the tag of the violated preconditions.
- *Failed*: The test result includes some debugging information, in particular the call stack (object state) of the failed test.

The *Test result handler* — numbered (10) in the figure describing the Test Wizard's architecture — provides the user with result information under different formats — numbers (11), (12), (13) on the same picture:

(11) *A graphical representation* displayed in the *GUI Test result displayer*, with result diagrams, etc.

(12) *XML files*: to have a standard exchange format.

(13) *Files using the Gobo Eiffel Test format* **[6]**: to enable the user to reuse the test results in a standard unit-test tool. Gobo Eiffel Test (getest) is the "JUnit of Eiffel". (For more information about *JUnit*, see **[2]**, **[3]**, **[8]**, **[14]**, and **[24]**.)

3.7 Storing results into a database

To handle regression testing, the Test Wizard stores some test information into a database. These storage parts of the wizard are numbered (4), (7), (9), and (14) in the picture of section **3.2**:

(4) The Test Wizard makes the test scenario persistent to avoid asking the user to reenter the test parameters when running the tool for the second time (or more). Note that if new classes have been added to the system, they will not be taken into account for the regression tests. (Information about the system — classes, features, etc. — is stored into the database together with the scenario.)

(7) The Test Wizard also keeps track of the order according to which classes should be instantiated (to avoid the business of topological sort and possible interaction with the users to disambiguate remaining cases).

(9) The test results are also made persistent, and the database updated after each feature call. (The user may change the update frequency, but the default policy is one, meaning after each test.)

(10) The *Test result handler* takes a "test result" as input and generates different representations of it. It is an independent wizard, meaning that the test result may be coming directly from the *Call simulator* or retrieved from the database through the *Database query handler* (14). Other possible database queries include: finding out all successful tests (remember it means making the system fail), finding out the feature calls that kept causing precondition violations, etc.

3.8 Limitations

The Test Wizard also has limitations. Here are the three major weaknesses:

- *Assertion limitations*: Not all conditions can be expressed through assertions, although agents increase the expressiveness of Eiffel contracts. (About agents, see [10] and the chapter 25 of ETL3 [19], describing the next version of Eiffel.) Missing preconditions will yield "unnecessary" failures. Missing postconditions will yield "hidden" behavioral bugs.
- *Class instantiation*: It may be impossible to instantiate some classes automatically. Besides, generic classes can never be fully tested. Therefore the Test-Wizard will need some help (Eiffel code) from the users to instantiate some classes and set up a correct environment.
- *Execution errors*:
  - o Features under test may enter *infinite loops*: Multithreading [13] can help to a certain extent (using a timer).
  - o Features may provoke *unhandled exceptions*, such as stack overflows

  In both cases, the Test Wizard reports to the user which feature caused the execution to fail, but it cannot do much more.
- *Harmful features*: Even if the Eiffel method advocates the *Command-Query separation principle* (see [18] page 751), Eiffel programmers can use features with side-effects in assertions. Effects may include: hard disk accesses, excessive use of memory, lock ups of system resources, screen resolution switching, etc. Besides, such "harmful effects" may be the primary goal of the feature under test (for example a command setting a new screen width, or a feature to open a certain file).
- *Execution time*: Under certain conditions (for example, if the number of features to be tested is large), executing the generated test program may be extremely long. Nevertheless, the level of parameterization offered by the Test Wizard and the real-time feedback (after each test) should not hamper the use too much.

4. RELATED WORKS

[TO BE COMPLETED]

- Case study by Lehman and Beladi [REFERENCE?].

5. CONCLUSION

The Test Wizard is not more than a specification for the moment. However, we think the resulting product will bring Eiffel developers with a powerful and easy-to-use tool to exercise their software, in particular libraries. Indeed, typical Eiffel libraries already express the contracts. Therefore, using the Test Wizard implies just a small overhead on the user's part (simply provide the minimal test setup parameters). Besides, the Test Wizard is designed in a way that many features can be tested with no user intervention.

The ultimate goal is to make the Test Wizard part of EiffelStudio. It would allow rather advanced unit testing at a low cost for any project developed with EiffelStudio. It would be especially useful for libraries where assertions are usually heavily used.

The Test Wizard would be entirely based on Design by Contract™ and reinforce the image of quality linked to Eiffel.

Further research directions include elevating routine preconditions to the rank of first-class citizens and give them a more important role for test-case generation.

BIBLIOGRAPHY

[1] Karine Arnout. "Contracts and Tests". *Ph.D. research plan, ETH Zurich*, 2002. Available from http://se.inf.ethz.ch/people/arnout/phd_research_plan.pdf. Accessed June 2003.

[2] Kent Beck, and Erich Gamma: *JUnit Cookbook*. Retrieved June 2003 from http://junit.sourceforge.net/doc/cookbook/cookbook.htm.

[3] Kent Beck, and Erich Gamma: *Test Infected: Programmers Love Writing Tests*. Retrieved June 2003 from http://members.pingnet.ch/gamma/junit.htm.

[4] Éric Bezault: *Gobo Eiffel Project*. Retrieved June 2003 from http://www.gobosoft.com/eiffel/gobo/index.html.

[5] Éric Bezault: *Gobo Eiffel Lint*. Retrieved June 2003 from http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/gobo-eiffel/gobo/src/gelint/.

[6] Éric Bezault: *Gobo Eiffel Test*. Retrieved June 2003 from http://www.gobosoft.com/eiffel/gobo/getest/index.html.

[7] Robert V. Binder: *Testing Object-Oriented Systems, Models, Patterns, and Tools*. Addison-Wesley, 1999.

[8] Mike Clark: *JUnit Primer*, October 2000. Retrieved June 2003 from http://www.clarkware.com/articles/JUnitPrimer.html.

[9] O.J. Dahl, E. Dijkstra, and C.A.R. Hoare: *Structured programming*. New York: Academic Press, 1972.

[10] Paul Dubois, Mark Howard, Bertrand Meyer, Michael Schweitzer, and Emmanuel Stapf. From calls to agents". In *Journal of Object-Oriented Programming* (JOOP), vol. 12, no. 6,

June 1999. Available from http://www.inf.ethz.ch/~meyer/publications/joop/agent.pdf. Accessed June 2003.

[11] Eiffel Software Inc. *EiffelBase*. Retrieved June 2003 from http://docs.eiffel.com/libraries/base/index.html.

[12] Eiffel Software Inc. *EiffelStudio: A Guided Tour. 16. How EiffelStudio compiles*. Available from http://archive.eiffel.com/doc/online/eiffel50/intro/studio/index-17.html. Accessed June 2003.

[13] Eiffel Software Inc. *EiffelThread*. Retrieved June 2003 from http://docs.eiffel.com/libraries/thread/index.html.

[14] JUnit.org: *JUnit, Testing Resources for Extreme Programming*. Retrieved June 2003 from http://www.junit.org/index.htm.

[15] Bertrand. Meyer. "Applying 'Design by Contract'". *Technical Report TR-EI-12/CO, Interactive Software Engineering Inc.*, 1986. Published in *IEEE Computer*, vol. 25, no. 10, October 1992, p 40-51.

[16] Bertrand Meyer: *Eiffel: The Language*. Prentice Hall, 1992.

[17] Bertrand Meyer: *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice Hall, 1994.

[18] Bertrand Meyer: *Object-Oriented Software Construction, second edition*. Prentice Hall, 1997.

[19] Bertrand Meyer: Eiffel: The Language, Third edition (in preparation). Available from http://www.inf.ethz.ch/~meyer/ongoing/etl/. (User name: Talkitover; password: etl3).

[20] Bertrand Meyer: *Design by Contract*. Prentice Hall (in preparation).

[21] Robert Mitchell and Jim McKim: *Design by Contract, by example*. Addison-Wesley, 2002.

[22] Glenford J. Myers: *The Art of Software Testing*. John Wiley & Sons, 1979.

[23] National Institute of Standards and Technology (NIST). "The Economic Impacts of Inadequate Infrastructure for Software Testing". *Planning Report 02-3, RTI Project Number 7007.011*, May 2002.

[24] SourceForge.Net: *JUnit*. Retrieved June 2003 from http://junit.sourceforge.net/.

[25] James A. Whittaker. "What Is Software Testing? And Why Is It So Hard?". *IEEE Computer*, vol. 17, no.1, January/February 2000, p 70-79. Available from http://www.computer.org/software/so2000/pdf/s1070.pdf. Accessed June 2003.

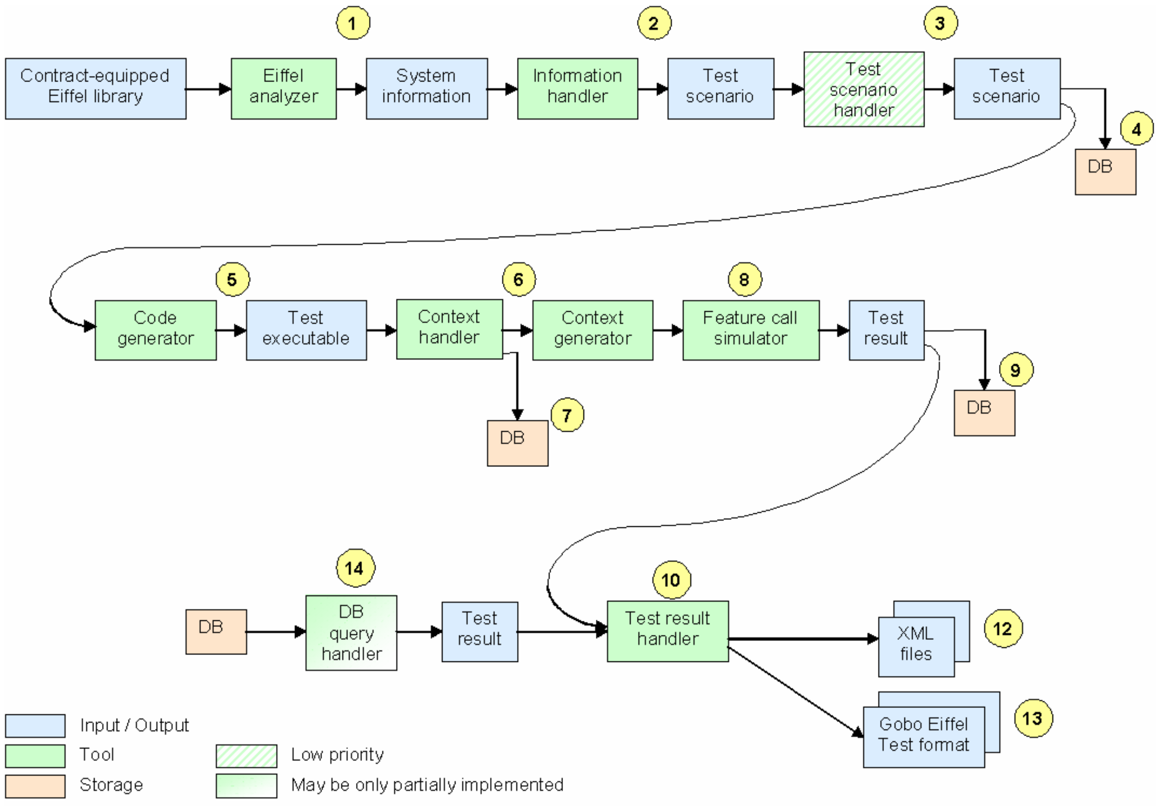
[26] Ed Yourdon. "A Reengineering Concept for IT Organizations: 'Good-Enough' Software". *Computer World Italia*, August 1995. Available from http://www.yourdon.com/articles/GoodEnuf.html. Accessed June 2003.

ACKNOWLEDGEMENTS

[TO BE COMPLETED]

APPENDIX A: SCOPE OF THE PROJECT

The following figure shows what parts of the Test Wizard should be implemented during this project:



Minimal architecture of the Test Wizard

(1) Gather system information.

(2) Display system information to the user and gather user information.

(3) Build the test scenario from the criteria selected by the user.

(4) Store the test scenario into a database (for regression testing).

(5) Generate a test executable corresponding to the test scenario.

(6) Run the executable: it creates a pool of objects (the “Context”) possibly helped by the user.

(7) Store the order of class instantiations (for regression testing).

(8) The executable performs feature calls on the pool of instantiated objects.

(9) Store the test results into a database.

(10) Output the results to the users.

(12) Generate XML files corresponding to the test results.

(13) Generate files using the Gobo Eiffel Test [6] format.

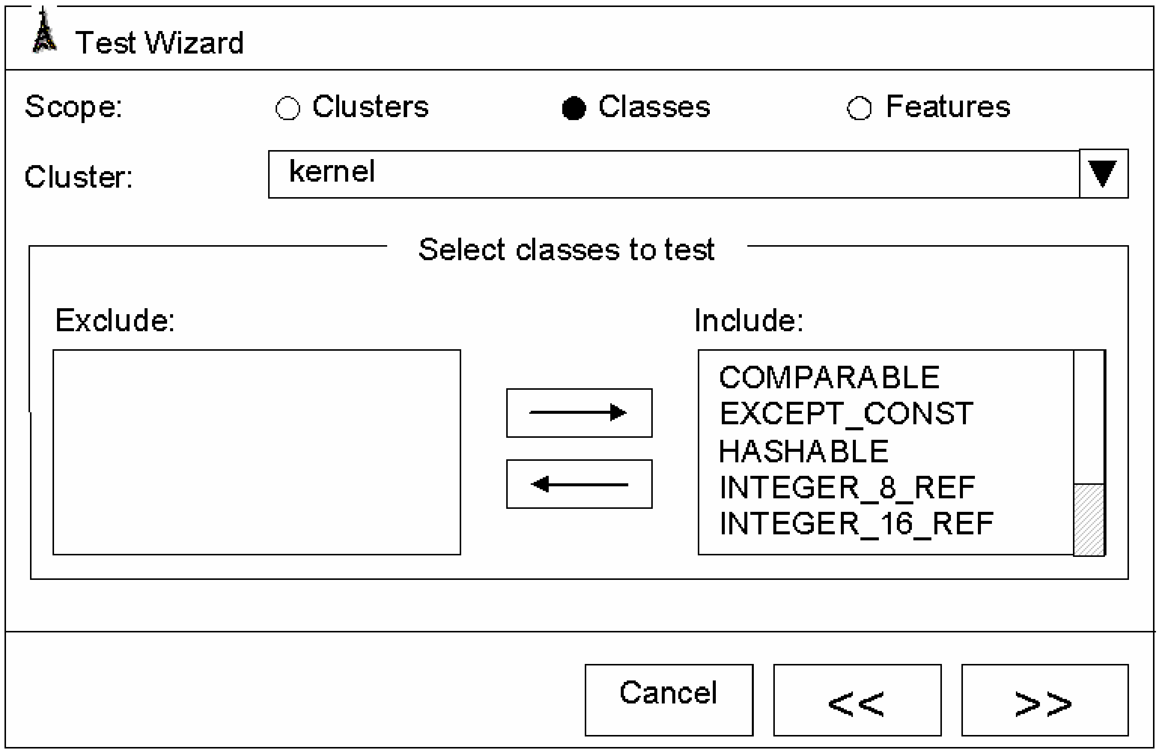
(14) Query the database and retrieve test results to be passed to the test result handler.

- The project will not include any GUI development. Therefore, the *GUI test result displayer* — number (11) on the architecture picture given in section 3.2 — will not be implemented at all.
- The *Test scenario handler* — enabling the user to change the automatically generated scenario — has a low priority; it should be implemented only at the end of the project if time permits. (It is a quite independent functionality, which can be added later on.)

- The *Database query handler* — giving the user the ability to query the database storing the test results — may be only partly implemented. It should be possible to retrieve the test results; the functionalities to support more fine-grained queries can be added in a second version of the tool.

APPENDIX B: INFORMATION HANDLER (GUI)

Although the GUI aspect is not part of the current project, we still include a few pictures of what the Information handler wizard may look like to have a complete description of the tool:


Test Wizard

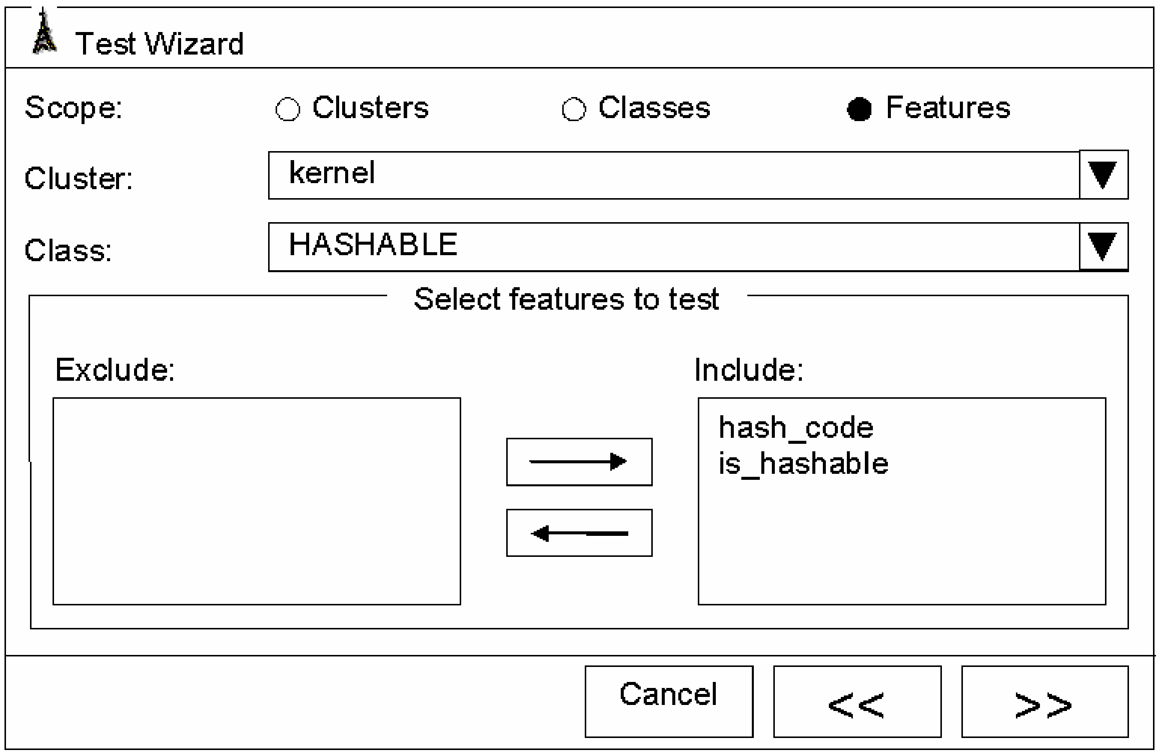
Scope: ○ Clusters [✓ ● Classes] ○ Features

Cluster: kernel

Select classes to test

Exclude:

Include:
COMPARABLE
EXCEPT_CONST
HASHABLE
INTEGER_8_REF
INTEGER_16_REF


Test Wizard

Scope: ○ Clusters ○ Classes [✓ Features]

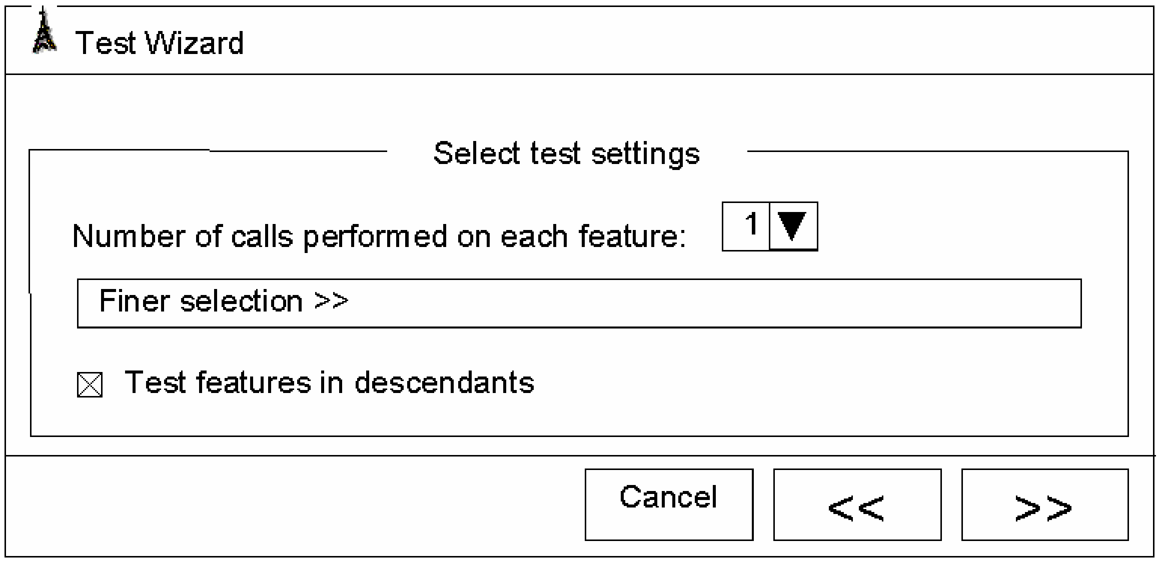
Cluster: kernel

Class: HASHABLE

Select features to test

Exclude:

Include:
hash_code
is_hashable

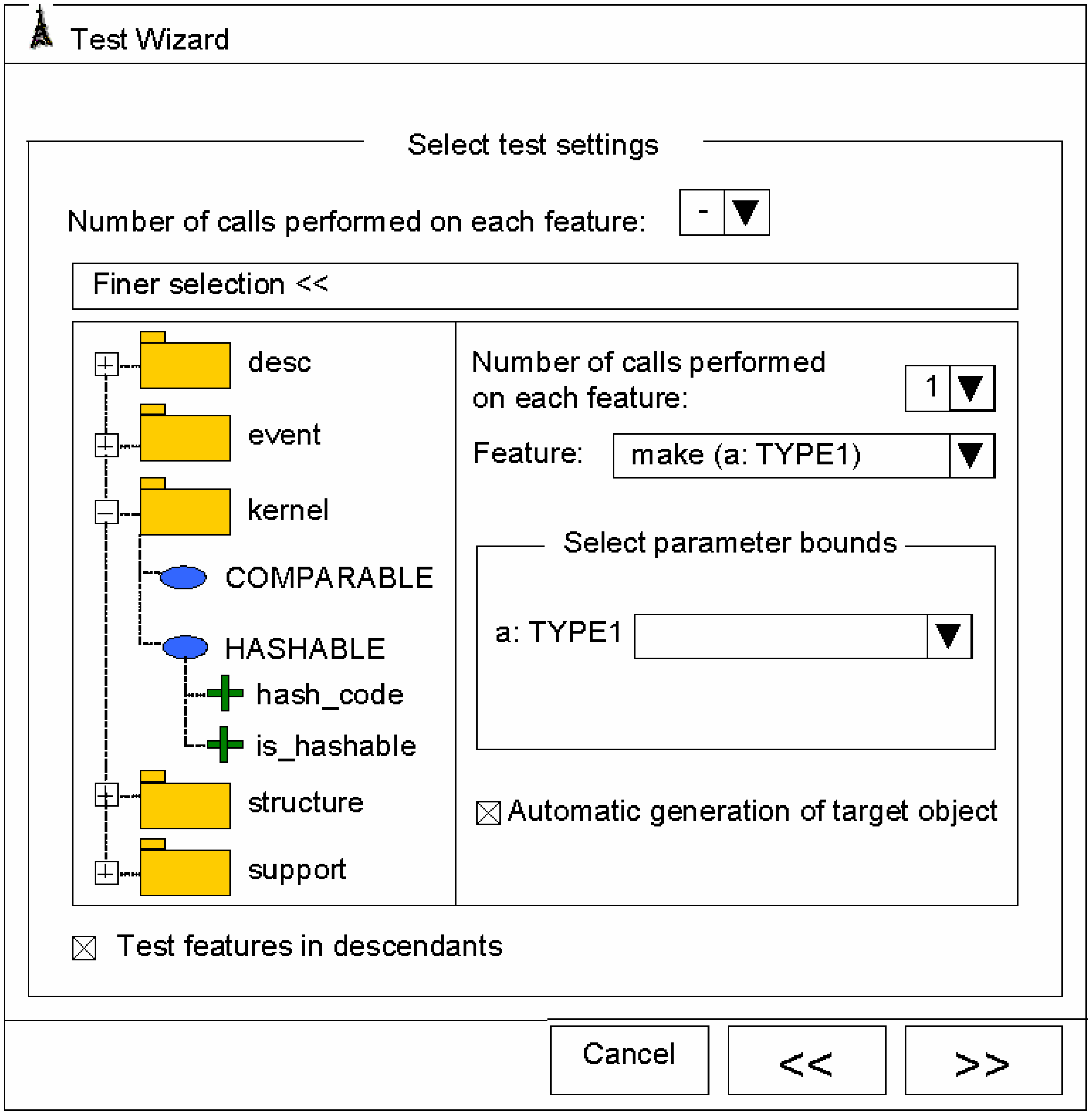

Test Wizard

Select test settings

Number of calls performed on each feature: 1

Finer selection >>

[✓ Test features in descendants]



Number of calls performed on each feature: -

Finer selection <<

Number of calls performed on each feature: 1

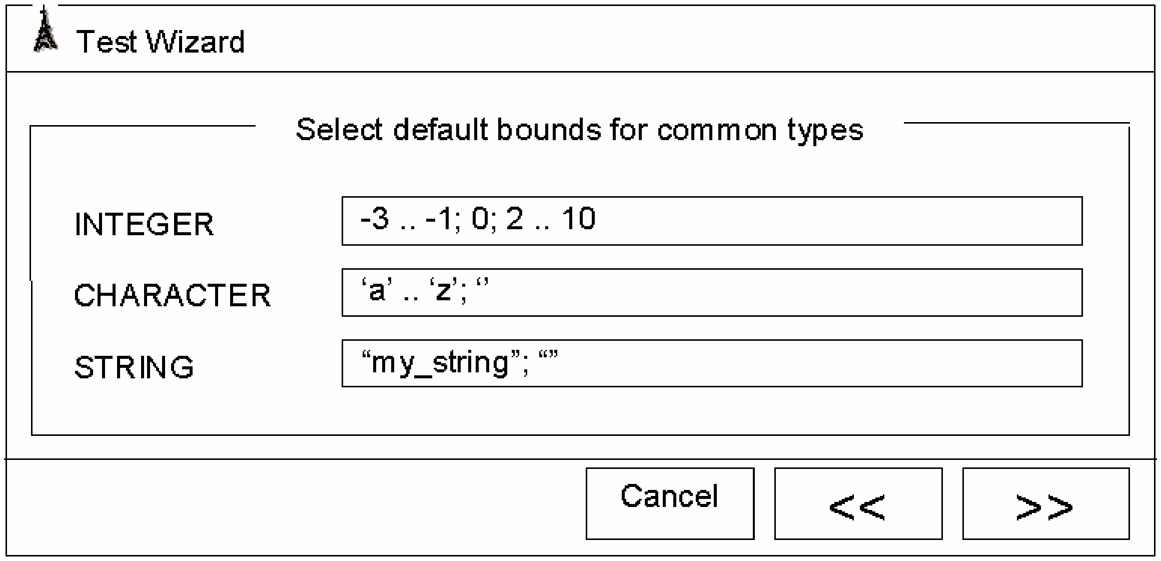
Feature: make (a: TYPE1)

Select parameter bounds

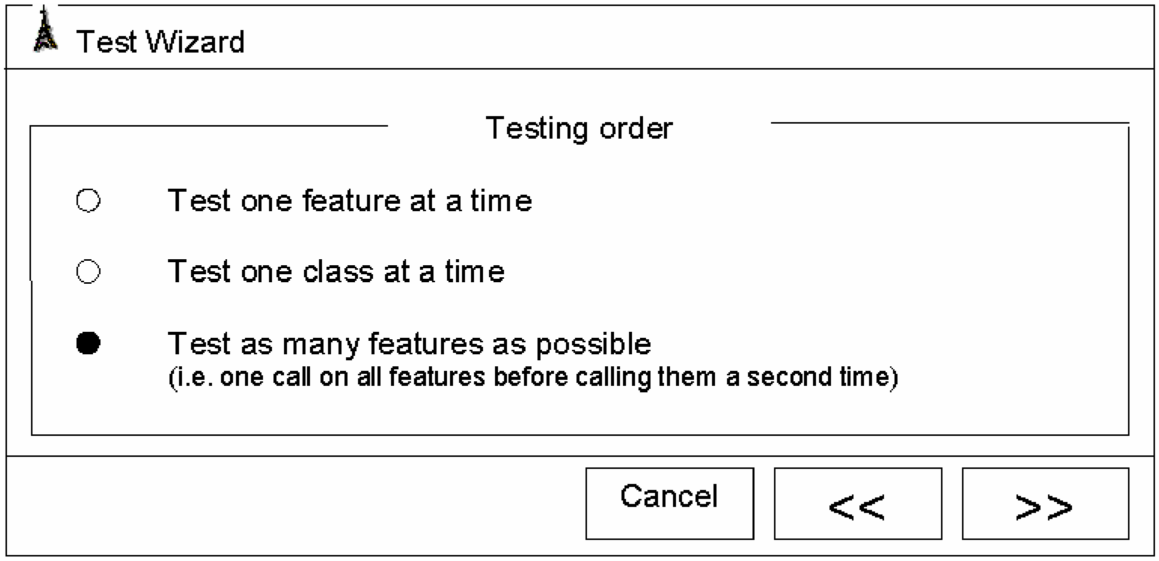
a: TYPE1

☒ Automatic generation of target object

☒ Test features in descendants



INTEGER	-3 .. -1; 0; 2 .. 10
CHARACTER	‘a’ .. ‘z’; ‘’
STRING	“my_string”; “”

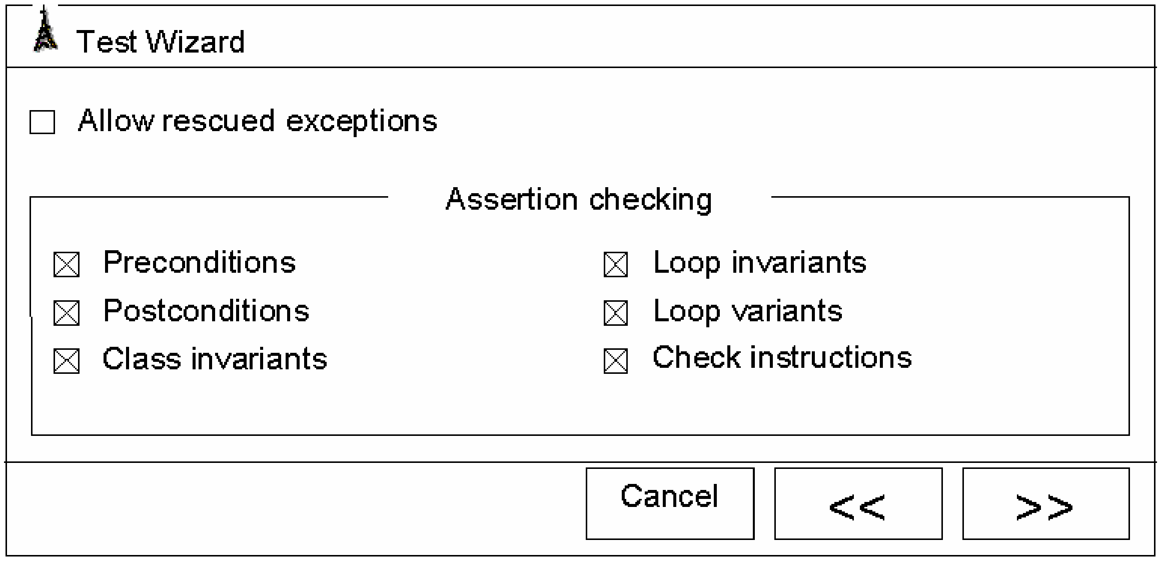

Test Wizard

Testing order

○ Test one feature at a time

○ Test one class at a time

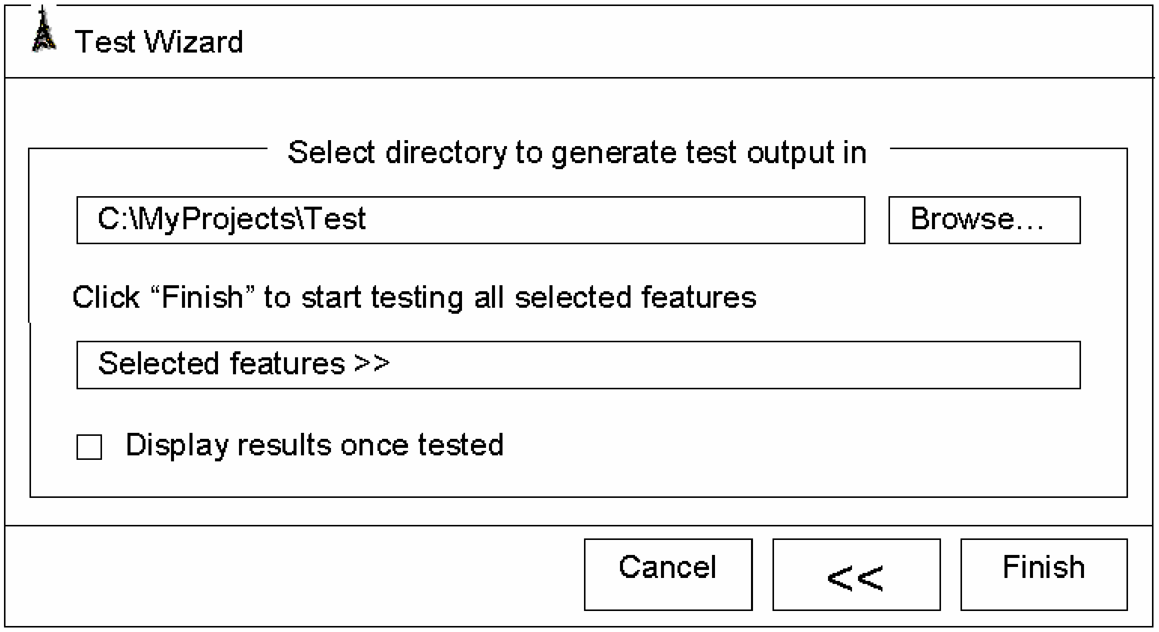
● Test as many features as possible
(i.e. one call on all features before calling them a second time)


Test Wizard

☐ Allow rescued exceptions

Assertion checking

☒ Preconditions	☒ Loop invariants
☒ Postconditions	☒ Loop variants
☒ Class invariants	☒ Check instructions

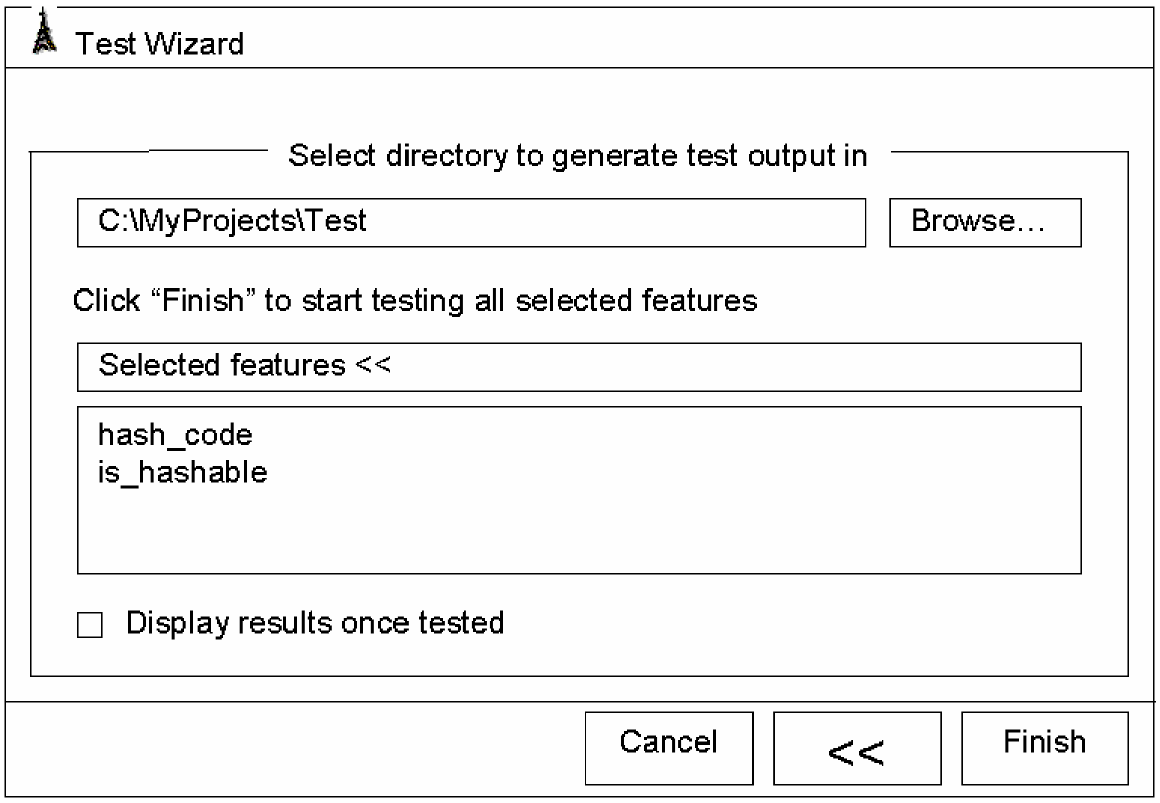

Test Wizard

Select directory to generate test output in

C:\MyProjects\Test

Click “Finish” to start testing all selected features

☐ Display results once tested


Test Wizard
Select directory to generate test output in
C:\MyProjects\Test
Browse…
Click “Finish” to start testing all selected features
Selected features <<
hash_code
is_hashable
☐ Display results once tested
Cancel
<<
Finish