Example Centric Programming

Jonathan Edwards
MIT CSAIL
32 Vassar St
Cambridge, MA 02139
edwards@csail.mit.edu

ABSTRACT

Programmers tend to understand programs by thinking of concrete examples. *Example Centric Programming* seeks to add IDE support for examples throughout the process of programming. Instead of programmers interpreting examples in their head, the examples are written down and the IDE interprets them automatically. Advanced UI techniques are used to present the results closely integrated with the code. Traditionally distinct programming tools (the editor, Read-Eval-Print-Loop, debugger, and test runner) are unified into a single tool that might be called an *example-enlightened* editor. This is expected to benefit a wide spectrum of programming activities, for both novice and experienced programmers. Some novel methods for testing and development are made possible. In the longer term, example centrism has implications for the design of future programming languages. A prototype has been implemented for Java in Eclipse.

Categories and Subject Descriptors

D.2.6 [Software Engineering]: Programming Environments --- Integrated environments, Graphical environments; D.2.3 [Software Engineering]: Coding tools and techniques --- Program editors; D.2.5 [Software Engineering]: Testing and Debugging --- Debugging aids, Testing tools, Tracing.

General Terms

Human Factors, Languages.

Keywords

Debugging, Unit testing, Examples, Integrated Development Environment, Eclipse.

1. INTRODUCTION

Software is abstract: this is the source of both its power and its problems. We start with information abstracted from a problem domain. Software that processes this information is a second-level abstraction. Concepts used to organize and structure software are thus third-level abstractions. When the problem domain is socially constructed, another level of abstraction is added.

Many of the pathologies of software are due to this extreme level of abstractness, as compared to other fields of design and engineering. Abstract thinking is hard, requiring talent, training, and attention - all of which are scarce resources. It is widely observed in all fields that the best way to learn and understand abstractions is with examples. Accordingly in practice we tend to understand code by working through examples in our head. We are mentally running an interpreter when there is a computer sitting right in front of us!

The goal of this research is to provide automated IDE support for the use of examples in programming. The idea is this: the programmer explicitly writes down examples, and the environment uses code instrumentation techniques to trace their execution automatically in the background. Advanced user interface techniques are used to tightly integrate trace data with the display of the program in the editor. The effect is to *enlighten* the program source with examples: code is seen side-by-side with the results of its execution on an example, and changing the code or the example immediately updates this view. Giving examples a pervasive role in the process of programming in this way will be called *Example Centric Programming*.

This simple idea has surprisingly far-reaching ramifications. The plan of this paper is, naturally, to start with an example: a prototype implementation of Example Centric Programming. The ways in which this goes beyond existing technologies and tools will be highlighted. The connection with prior research in programming language design is then explored. Finally there is a discussion of the strategic implications for future innovation in programming languages

2. EG

The EG tool is an Eclipse [6] plug-in for Example Centric Programming in Java. A screen shot of an early prototype is shown in Figure 1[1]. On the right-hand side is the standard Eclipse Java editor on a factorial program. Two examples are shown appended at the bottom of the file[2]. In general, examples are standalone snippets of code that call the code under observation. Unit tests [2][13] are a good source of examples, and should be automatically recognized as such.

OOPSLA'04, Oct. 24–28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.

[1] The screen shots in all the figures are real – only the colors have been altered for better grayscale contrast. The color version can be seen at http://sdg.lcs.mit.edu/pubs/2004/examplecentric.pdf

[2] This is a stopgap: the proper place for examples is discussed in section 4.4.

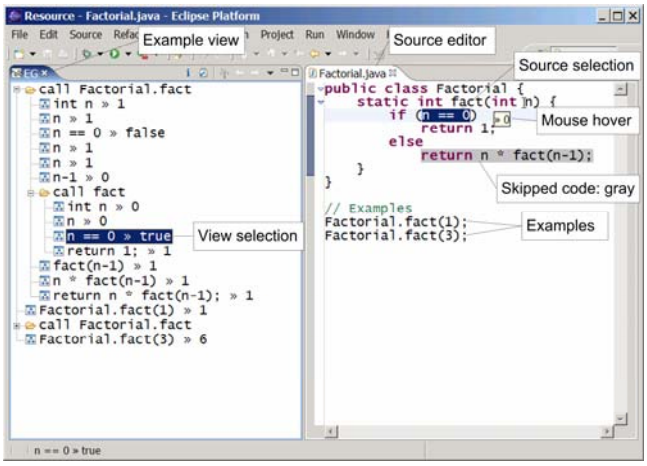

Figure 1. Example Viewer

On the left-hand side of the screen shot is the example view provided by the EG plug-in. This view uses a familiar outline GUI to present a tree-structured trace of the execution of the examples. There are two kinds of lines in the outline: one shows the result of evaluating an expression or statement in the source, and has the form `<source> » <value>`. The other kind of line records a call to a constructor or method, and has the form `call <name>`. A source expression that causes a traced call will be recorded in two lines: the call followed by the (non-void) returned value.

The outline can be expanded to show the details of a call (by clicking on the "+" icon in the standard way), as has been done in Figure 1 for the first example. Indented under a call are all the traces from the execution of that method or constructor, in the precise order in which they actually executed. Additional lines are prefixed to display the value of each formal argument bound in the call. The recursive call has also been expanded in Figure 1.

Clicking in the outline selects and highlights the line under the mouse. Simultaneously, the corresponding fragment of the source code is highlighted in the editor, either an expression or a method name. The up and down arrow keys move the outline selection forward and backward in time, with synchronized highlighting in the source code. This is somewhat like single-stepping in a debugger, with the added ability to step backwards (not to mention the ability to see the actual values of expressions, lacking in virtually all debuggers).

Control flow is shown in the source with a gray background on code that is skipped, as in the else clause in Figure 1. Control flow can vary between different calls to the same method – the selected line in the example view picks out the context of the specific call being visualized[3]. Hovering the mouse over an expression in the source will bring up a transient window showing the value of that expression. Mouse hovers are contextual to the example view in the same way as the control flow annotation.

There is another level of outline structure not shown in the screenshot (because it has not yet been implemented), which is that lines reporting object values can be expanded to see the fields of the object. Likewise the values of these fields can be expanded, arbitrarily traversing the heap. Note that the values displayed by drilling down into the heap in this manner are all synchronized with the global state at that point in the execution trace. Exploring the same or a different object can be done at another point in the trace to see the global state at a different time. It is a strict law of the visual metaphor that time flows downward in the example view. The example view allows you to see multiple times at once – rather than the "peephole" of debugging-style interfaces, it presents the whole panorama of history.

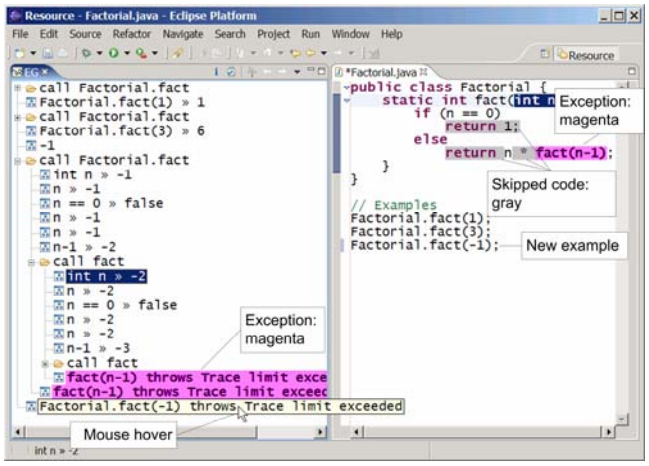

Figure 2. Exceptions

Exceptions are visualized as a special kind of value. Figure 2 is a snapshot taken after adding another example: `factorial(-1)`. Lines reporting exceptions in the example view have the form `<source> throws <exception>`. Exceptions and their corresponding source in the editor are highlighted in magenta. Figure 2 shows the result of expanding several of the calls tracing back the origin of the exception, and hovering the mouse to see the full exception value. This particular exception is automatically thrown when an example executes too long. Note how the source view annotation shows exactly what part of the expression threw the exception, and which parts were consequently skipped.

The GUI shown so far is an early prototype needing much further development. Extraneous information needs to be filtered out. More fundamentally, there is a need to explore alternatives to the split-screen approach that could provide a single integrated view: either merging code into the execution traces, or merging execution data into the code view. This is a major challenge for continuing research.

3. BEHIND THE CURTAINS

What is really going on here is that the set of examples is being executed in the background in a specially instrumented interpreter (BeanShell [1]) running in a tethered Java VM. The architecture is shown in Figure 3. Any change to the code will cause an automatic re-execution after a set idle time. Syntax errors are treated similarly to exceptions.

[3] Control flow can also be displayed aggregating all calls to a method, highlighting code not executed in any example. This visualizes example coverage.

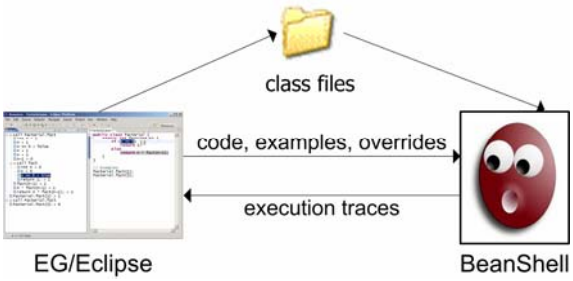

Figure 3. Architecture

The instrumented interpreter emits a stream of trace information about every expression evaluated in the subject code. Initially only the class associated with the example is traced, not external code such as in the Java libraries. Such calls to external code will trace just the fact of their execution and their return value. However the user can still expand such external calls in the outline, just as if a detailed trace was available. When that occurs the example is re-executed with the external class traced and the additional results displayed as if they had been present all along. This depends on examples executing largely deterministically[4], which seems a reasonable expectation for an example.

The same strategy is used for exploring state. Only the `toString()` values of objects are recorded by default. Expanding the displayed value in the outline causes extra instrumentation to be added to the execution in order to dump out the contents of the object at that point. This projects the illusion that the complete details of execution and state are present when in fact only the minimum necessary is being recorded. The idea of incremental on-demand instrumentation was first used by Tolmach and Appel for ML [1]. The technique is here extended to side-effects and mutable state by focusing solely on deterministic executions.

An instrumented interpreter suits the needs of this prototype research, and perhaps even the needs of student programmers, but it is likely that a byte-code based instrumentation technique will be required to scale up to industrial-strength programming.

4. IMPLICATIONS

Example centrism has implications for the entire spectrum of programming activities, for both novice and experienced programmers. These are explored in the following sub-sections.

4.1 Teaching with Examples

It seems reasonable to expect that an example centric environment would be a useful teaching tool. Examples are central to many approaches to teaching [12], and are featured in programming textbooks. Amazon lists dozens of "<*Language*> Programming by Example" titles. Programming environments tailor-made for teaching attempt to make the execution of examples as immediate and visual as possible [7][8][15]. Example Centric Programming plays directly into all of these trends.

[4] An example of innocuous non-determinism is the internal object identifiers exposed by Java in the `Object.toString()` method. They may change on each execution, and affect hash table keys, but do not affect the observable execution of examples.

4.2 Example Centric Debugging

In Example Centric Programming, debugging becomes a straightforward matter of inspection, at least for examples and tests and any automatically reproducible problem. You simply "browse to the bug". There is actually nothing new in the back-end technology being used to trace example execution: it has all been done before in experimental debuggers [3][15][23][20]. Zstep [25] also annotated source with a single moving value window. What is new here is the application of this technology "outside the box" of a debugger.

A debugger is used in a different mode than the editor – first you edit your code, then you switch to the debugger and manually run the code with some inputs. The debugger presents an entirely different UI and mode of interaction than the editor. The goal here is to eliminate this mode-switching by unifying the debugger and editor into a single tool with a consistent UI. This can be described as an *example-enlightened* editor.

In addition to sidelining the debugger, this approach supplants the need for a Read-Eval-Print-Loop: the canonical exploratory UI to an interpreter. Expressions typed into a REPL are instead now just example snippets in a source file, with their results appearing in the example view rather than inserted into the transcript. Results are automatically refreshed whenever the code changes, which avoids the hidden pitfalls of anachronistic definitions [7].

As will be discussed below, the need for a separate test-runner tool is also eliminated. Tests, seen as examples, are continuously run while programming, and test failures are resolved by inspecting their execution rather than firing up a debugger.

The unification of these tools is not an accident: debuggers, REPL's, and test-runners are the tools conventionally used to help make programs concrete. Debuggers let us observe the internals of an execution, while REPL's let us easily try out executions to see their results, and tests are a way of tying the program to concrete specifications. Unifying these tools with the editor enables synergies between them and reduces mode-switching overhead, improving the focus and flow of programming.

4.3 Example Centric Testing

There is a close connection between Example Centric Programming and Unit Testing [13][2]. Unit tests can serve as examples. More deeply, unit testing and example centrism share the goal of anchoring abstractions to concrete reality. Unit testing does this by repeatedly testing code for conformance at a fine-grained level. Example centrism additionally uses visualization techniques to make code semantics more concrete. The benefit to practitioners of unit testing is to expose the details of how tests execute, and to help debug them. Even better, writing tests is made easier, in the following way.

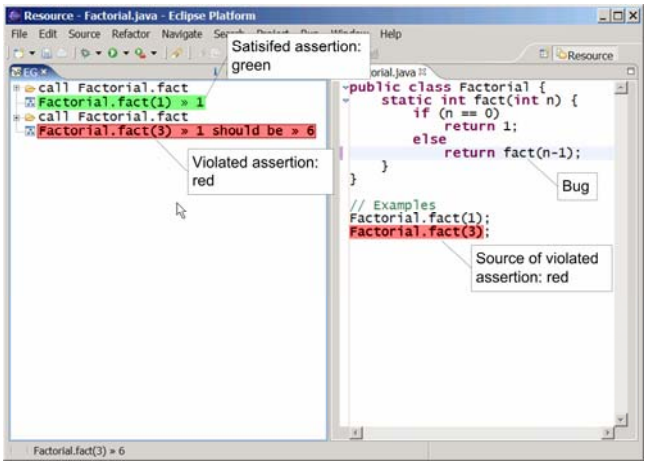

Figure 4. Assertions

From a right-click menu any value shown in the example view can be declared as an *assertion*. This means that the current value is frozen, and is expected to remain constant subsequently. Assertions that are satisfied display in green, whereas violated assertions are in red, and include the details of the mismatch. The source of a violated assertion is also highlighted in red in the editor. Figure 4 shows the result of converting the examples into assertions and introducing a bug into the code that violates one of them. *Assertions are unit tests grown from examples.*

Assertions provide the same immediate feedback as Auto-testing [10] and Continuous Testing [20]. More importantly, they eliminate the need to hand-code checks in tests. A common complaint about unit testing is that it feels like writing the code twice: it is necessary to construct the expected result in each test in order to compare it with the actual result. Tests can be very redundant code. Assertions allow each example to be gotten to work once and then frozen into a unit test. The end result is to make unit testing lighter-weight, and user-friendly. Similar goals for testing spreadsheets are the subject of WYSIWYT (What You See Is What You Test) [20].

4.4 Deep Testing

Assertions are contextual – they apply to a specific point in the traced execution tree of a specific example. For this purpose the execution trace can be thought of as a tree of code regions, each being an expression or statement, with each parent in the tree being a call site. The location of an assertion in the tree is a path of nested calls leading from one example to the expression being asserted. Code edits conserve these code regions, allowing assertions to persist through edits that do not affect their execution[5].

A consequence of the persistent contextuality of assertions is that they can be made deep inside the code. Conventional unit testing is "black-box" – it can only work through the API. Deep assertions allow intrusive "glass-box" testing, where it is possible to probe the internal workings of the code.

The dual of an assertion is an *override*, which sets the value of some code expression to be the value of a different expression. As with assertions, this is bound to a specific location in the execution tree, so that it applies to one specific call frame. Overrides can be used to simulate external systems for a test. The conventional solution for this is to build a mock-up of the external system that behaves in a certain canned fashion just as the external system would [18]. Mock-ups can become complicated, particularly when their behavior must vary depending on the test or the call site.

Overrides permit a simpler ad-hoc solution: hard-wire the values seen at the interfaces to the external system, even if they are deep inside the tested code. In other words, "Mock Interfaces" instead of Mock Objects. Deep assertions and overrides are novel capabilities for testing, and will be explored in subsequent research.

A question that has been ignored so far is exactly where assertions and overrides are stored. To serve in testing they must persist with the source itself. But as described above, they are contextual to specific locations in the execution trace of an example. The solution is to reify example execution traces into persistent metadata associated with source files. Assertions and overrides are embedded into this metadata. In fact the example source code itself should also be stored in this metadata, rather than the current stopgap implementation of letting them be vagrant code snippets in the source.

A common solution for associating tests with code is to embed them in nearby comments – perhaps first done in Poplar [19]. This approach suffers from the disadvantage that the tests become opaque to normal code support features in the IDE, such as syntax coloring, incremental compilation, and debugging. An alternative approach, taken in JUnit [13], is to put tests into a parallel class hierarchy using naming conventions. This approach has the disdvantages of weighing down tests with boilerplate wrapper code, and also making the connection between testing and tested code distant and implicit. The approach taken by Example Centric Programming is to turn example executions into a new kind of program artifact associated with source code, within which examples, tests, assertions, and overrides live like annotations to the source (more strictly speaking, as annotations to example executions of the source).

4.5 Example-Driven Development

Up to this point the discussion has been about how examples can assist reading and testing code. It is even more important to examine how they can assist writing and modifying code, which is the heart of programming. *Example-Driven Development* starts with a set of well-chosen examples (which not coincidentally is also a good way to specify a program, sometimes called use cases). The code can then be written using these examples as a guide. A set of tools is provided to assist in this process, dividing it into a series of small, simple steps.

[5] See the discussion of "orphans" in section 4.5 for the other case.

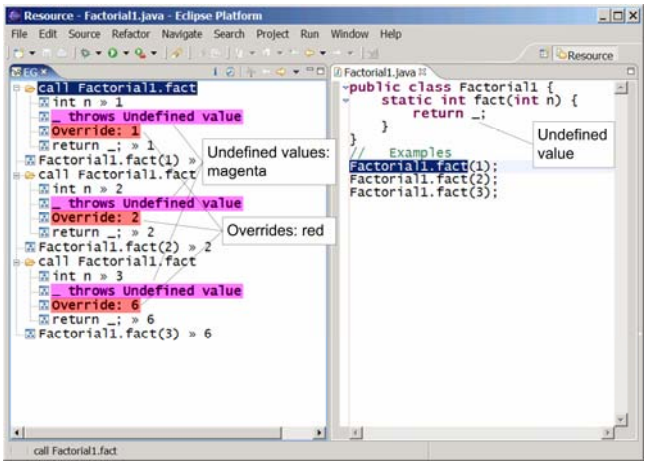

Figure 5. Overrides

To start with, a stub method is generated that just returns the special literal "_", which throws an exception when evaluated. The programmer writes some well-chosen examples that call the method. The trace of these examples will show the undefined value exceptions thrown by the "_". These are then overridden by the programmer to the correct values for each example. The result at this point is shown in Figure 5. Note that the overrides are in red, signaling that there is work to be done (overrides are green when the overridden expression has the same value as the override).

The examples now all return the correct values, though only because the vacuous stub code has been manually overridden to do so. The technique of Example-Driven Development is to grow the code "outside-in" from these working examples. Overrides are converted into real code in a series of transformation steps that preserve the behavior of all the examples. This is similar to the way that refactorings [9] restructure code while preserving its semantics. These transformations are called *assimilations*, two of which are discussed here: *conditionalization* and *generalization*.

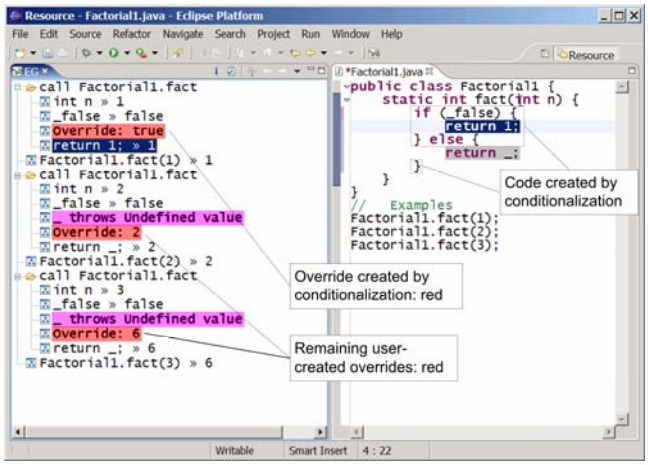

Figure 6. Conditionalization

Conditionalization is used to differentiate examples from each other, declaring that they take different code paths at some point. This is done by invoking the conditionalization command from a right-click on an override. Figure 6 shows the result of conditionalization of the first override, which is the base case of the recursive function. The statement being overridden, "`return _`", has been cloned, with the "_" replaced by the expression from the override, "`1`". That override has been deleted. The original statement and its clone are gated by a new `if` statement testing the special literal "`_false`", which always returns `false` when evaluated. The "`_false`" is overridden to `true` in the originally selected example to force the correct branch to be taken in that case.

As can be seen, all of the examples execute with unchanged results. The example containing the selected override now takes a different code path, and its return value is now real code rather than an override. A new override forcing the correct branch has been introduced in exchange. The net gain is that conditional structure has been introduced into the code, where only undifferentiated examples existed before. The conditionalization assimilation took care of all the clerical work involved – inserting boilerplate code and adjusting overrides, while guaranteeing unchanged example behavior. A new obligation has been made to write the code of the conditional test, but this task has been deferred till latter. Breaking up the mental effort involved in programming into simpler distinct steps is a big win for the programmer.

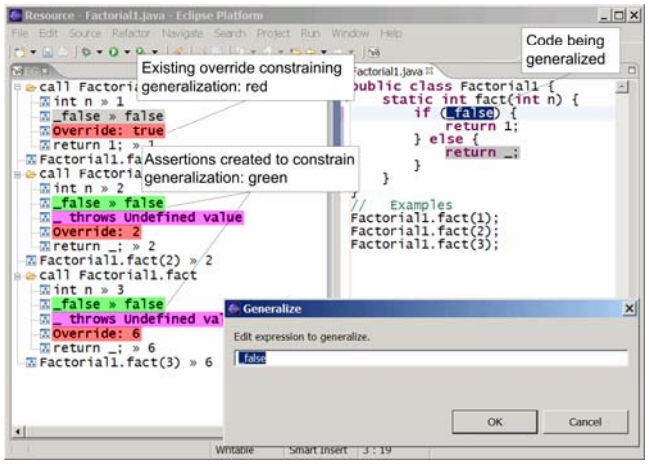

Figure 7. Generalization

The generalization assimilation is used next to replace dummy code with real code. It is called generalization because the new code is constrained to execute equivalently to the way the old code (as overridden) does in each example. This process will be detailed for the case of "`_false`". Generalization is invoked from a right-click on any trace of that expression. This action causes all the traces of that expression in all examples to have their current values asserted. The exception to this is when the expression is overridden, as in the base case example. These overrides are left in place. A dialog is opened to prompt the programmer to replace the expression being generalized[6]. The state at this point is shown in Figure 7.

[6] A better UI design would be non-modal, with a list of multiple pending generalizations.

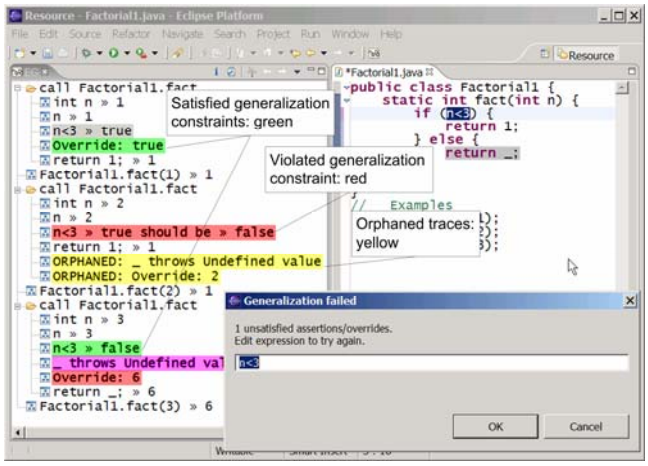

Figure 8. Failed generalization

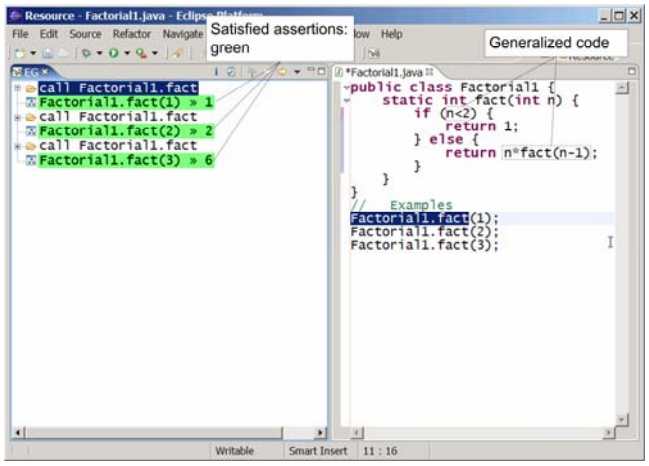

Figure 9. Completed generalizations

The new assertions, together with the existing overrides, serve as constraints on any new generalized code, testing that it really executes the same in all cases as the old code. If the programmer mistakenly violates one of these constraints they will turn red and an error dialog will be opened to prompt a resolution. An example of this is shown in Figure 8. When a code generalization satisfies its constraints the (now green) assertion/override constraints are optionally deleted. The crucial benefit of the generalization assimilation is the hand-holding it provides the programmer to ensure that the new code really does generalize the behavior of the old code/overrides.

Note in Figure 8 the two trace lines in yellow prefixed by "`ORPHANED`". The erroneous conditional expression caused the wrong branch of the if statement to execute, so the override of "`_`" to "`2`" did not execute as expected. Overrides and assertions that fail to execute are called *orphans*, and are kept in the execution view until their destiny is resolved. In this case a subsequent correction to the conditional expression will cause the orphans to be automatically repatriated. In the case of permanent orphans created by intentional changes of control flow, manual intervention is required to resolve the situation, which can be as simple as dragging and dropping them to a new location in the execution trace.

Two generalizations are required to complete the example driven development of this program: generalizing "`_false`" to be "`n<2`", and generalizing "`_`" to be "`n*fact(n-1)`". Once the code is fully generalized, assertions can be added to test it henceforth. The final state is shown in Figure 9. Note that the programmer still had to write the key snippets of code: generalization holds their hand, but doesn't try to think for them. This is in contrast to research in Programming by Example [5][17], which uses various machine learning techniques to automatically write a program from examples. The two approaches may naturally complement each other: generalization assimilations could offer the programmer a set of induced guesses to pick from.

In summary, Example-Driven Development tackles the difficulty of programming with a strategy of divide-and-conquer. This is in the same spirit as Test-Driven Development [2], in which programming is guided by incrementally satisfying tests. Example-Driven Development provides IDE support for this process, assisting and amplifying it. Examples provide a kind of *scaffolding* for the program under construction. Overrides are a way to temporarily stage behavior within examples without having to first make design decisions about the general code expression and exact control-flow position. Assimilations help evolve the program by dividing large mental leaps of abstract thought into a sequence of simple assisted steps.

Note that a program in the midst of Example-Driven Development is in a curious state. It is not runable, nor even compilable outside the EG tool. It consists not only of fragmentary source code but also the overrides and assertions annotated into its example executions. These annotations serve to create example-specific variants of the program. A program in this state is a novel kind of artifact: it is essentially poised midway between the concrete reality of examples and the pure abstraction of a finished program. It is a *partial abstraction*. Like an informal sketch, it is inconsistent and incomplete, yet it is also a precisely specified formal artifact with automated tool support. Partial abstraction seems to offer a bridge between the informal and the formal. This intriguing idea warrants further investigation.

5. EXAMPLE CENTRIC LANGUAGES

Example Centric Programming attempts to improve programming by making it more concrete. This is not a new idea: it has also inspired a variety of programming language research. Self [23][26] eloquently and elegantly espoused concreteness and immediacy in the union of programming language and environment. Some of the research in Programming By Example [5][17] has a similar motivation. Visual programming languages inspired by spreadsheets [4][14] have attempted to make program execution concrete with continuous data-flow semantics. VisiProg [11] was an early attempt with the same goal for limited uses of Basic. The common thread in all this research is the goal of *concrete code*: connecting abstract code to concrete reality.

Example Centric Programming partially realizes this idea within a conventional language via an IDE. The run-time semantics of the language are not affected, but in the restricted programming-time

world of examples, Java is made concrete. It turns out that this partial realization still provides significant practical benefits. This is not an accident: Example Centric Programming is intended to be a "Trojan Horse" for a truly concrete programming language.

Ultimately, a new programming language is required. Adding layers of tools can only increase complexity – eventually there must be simplification. An IDE can not go beyond programming-time, and will inevitably reveal imperfect seams[7]. What is needed is a language designed to be example centric from the ground up.

An example centric IDE can foster language progress in two ways. First, by proselytizing example centrism as a practical programming tool for a mainstream language, the gate is opened for a new language that would further expand upon the same benefits. Secondly, and more crucially, an outline can be sketched of how a truly concrete programming language should look and feel, based on real-world experience. This is the topic of subsequent research, but several initial observations can be made:

1. Prototype semantics, as in Self [23][26], are a natural way to tie code and examples together.
2. An example centric user interface, fully integrating code and execution, should be the "native" representation of the language, not a contrived illusion.
3. To achieve this integration, the programming language and its IDE must be seen as an inseparable whole, transcending the traditional constraint that a language be usable with only a text editor.

6. CONCLUSION

Example Centric Programming provides an IDE that illuminates code with examples. Abstract code is made concrete and easier to understand. The conventional development tools that partition programming into separate modes of editing, debugging, exploring, and testing code are unified into a single focused interface. The ideal is that all code is illuminated by some example, and that the programmer rarely needs to leave this example-enlightened environment. Novel methods of testing and development are made possible.

Building support for examples into programming tools is, in retrospect, an obvious idea. Yet it has surprisingly far-reaching consequences. This paper has demonstrated some of the possibilities: much further work is needed to fully realize them. In the longer term, example centrism may pave the way towards a new kind of programming language.

7. ACKNOWLEDGEMENTS

Discussions with Daniel Jackson and Derek Rayside contributed to this paper.

8. REFERENCES

[1] BeanShell http://www.beanshell.org/home.html

[2] Beck, K. *Test-Driven Development: By Example*. Addison-Wesley 2002.

[3] Booth, S.P., and Jones, S.B. Walk Backwards to Happiness – Debugging by Time Travel. *3rd International Workshop on Automated Debugging (AADEBUG 97)* Linkoping, Sweden, May 1997.

[4] Burnett, M., Atwood, J., Djang, R., Gottfried, H., Reichwein, J., and Yang, S. Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. *Journal of Functional Programming* 11, 2 (March 2001).

[5] Cypher, A. *Watch What I Do: Programming by Demonstration*. MIT Press 1993.

[6] Eclipse Platform Technical Overview http://www.eclipse.org/whitepapers/eclipse-overview.pdf

[7] Eisenstadt, M., and Brayshaw, M. A fine-grained account of Prolog execution for teaching and debugging. *Instructional Science*, 19(4/5), pp.407-436, 1990. http://kmi.open.ac.uk/marc/papers/InstrSci-90.ps.gz

[8] Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., and Felleisen, M. DrScheme: A programming environment for Scheme. *Journal of Functional Programming* 12, 2 (March 2002).

[9] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[10] Gamma, E., and Beck, K. Contributing to Eclipse. Addison-Wesley. October 2003.

[11] Henderson, P., and Weiser, M. Continuous Execution: The VisiProg Environment. *ICSE '85* London, England, 1985.

[12] Hoffman, D., Walsh, P. Teaching Programming with Minimal Examples. *Western Canadian Conference on Computing Education*. Nanaimo, BC, Canada 1997 http://www.cs.ubc.ca/wccce/program97/walsh/walsh.html

[13] Hunt, A., and Thomas, D. *Pragmatic Unit Testing*. http://www.pragmaticprogrammer.com/starter_kit/ut/index.html

[14] Jones, S.P., Blackwell, A., Burnett, M. A User-Centered Approach to Functions in Excel. *ICFP '03*. Uppsala Sweden, 2003.

[15] Kölling, M., Quig, B., Patterson, A. and Rosenberg, J., The BlueJ system and its pedagogy. *Journal of Computer Science Education*, Vol 13, No 4, Dec 2003 http://www.bluej.org/papers/2003-12-CSEd-bluej.pdf

[16] Lewis, B. Debugging Backwards in Time. *5th International Workshop on Automated Debugging (AADEBUG 2003)* Ghent, Belgium, September 2003

[17] Lieberman, H. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann, 2001.

[18] Mackinnon, T., Freeman, S., and Craig, P. Endo-Testing: Unit Testing with Mock Objects. In *Extreme Programming Explained*. Addison-Wesley 2001.

[19] Morris, J.H., Schmidt, E., Wadler, P. Experience with an applicative string processing language. *7th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '80)* Las Vegas, Nevada, 1980.

[7] One such seam may be the as-yet unresolved issue of class initialization in examples.

[20] Reiss, S.P. Graphical program development with PECAN program development systems. In *Proceedings of first ACM Software Engineering Symposium on Practical Software Development Environments.* 1984

[21] Rothermel, G., Li, L., DuPuis, C., and Burnett, M. What You See Is What You Test: A Methodology for Testing Form-Based Visual Programs. *20th International Conference on Software Engineering* (ICSE 98). April, 1998.

[22] Saff, D., and Ernst, M.D. Reducing wasted time via continuous testing. In *Fourteenth International Symposium on Software Reliability Engineering*, (Denver, CO), November, 2003.

[23] Smith, R.B., and Ungar, D. Programming as an Experience: The Inspiration for Self. *ECOOP '95 Conference Proceedings*, Aarhus, Denmark, August, 1995.

[24] Tolmach, A., and Appel, A.W. A Debugger for Standard ML. *Journal Functional Programming* 1, 1 (January 1993)

[25] Ungar, D., Lieberman, H., and Fry, C. Debugging and the Experience of Immediacy. *CACM*. 40, 4 (April 1997).

[26] Ungar, D., and Smith, R.B. SELF: The Power of Simplicity. *OOPSLA '87*. Orlando, FL, October, 1987