In Hasso Plattner, Christoph Meinel, and Larry Leifer (eds.)
Design Thinking Research: Studying Co-creation in Practice (pages 259-277)
Springer 2011 (doi:10.1007/978-3-642-21643-5_15)

Applying Design Knowledge to Programming

Bastian Steinert and Robert Hirschfeld

Abstract Arguably programming involves design: computational logic - the program - is constantly reorganized to keep complexity manageable and provide for current and future coding activities to be feasible. However, design practices have gained less attention in the field of programming, even though decades of research on design have led to a large body of knowledge about theories, methods, and best practices. This chapter reports on first results of our research efforts to transfer and apply design knowledge to programming activities. We improved tool support for software developers in two respects, both of which are based on key concepts in design practices: continuous feedback and ease of exploration.

1 Introduction

Agile software development and Design Thinking build on similar values and principles. Agile processes such as Extreme Programming or Scrum are based on short iterations. This approach has many advantages. It results in regular delivery of value to the customer and it enforces developers to constantly face feasibility questions, resulting in feedback on different aspects. Agile processes assume co-evolution of problem understanding and the implementation of a proper solution.

Techniques and values of Design Thinking can be a useful supplement to Agile principles [23]. Both Design Thinking and Agile processes value feedback and encourage team members to interact closely with each other and prospective users. They also emphasize the importance of directness and doing – being continuously involved and in dialog with the product to be created.

Efforts to bring Design Thinking to the development of software systems should not be limited to the domain of user interfaces and end-user interaction, but needs

Software Architecture Group
Hasso Plattner Institute, University of Potsdam, Germany
e-mail: firstname.lastname@hpi.uni-potsdam.de

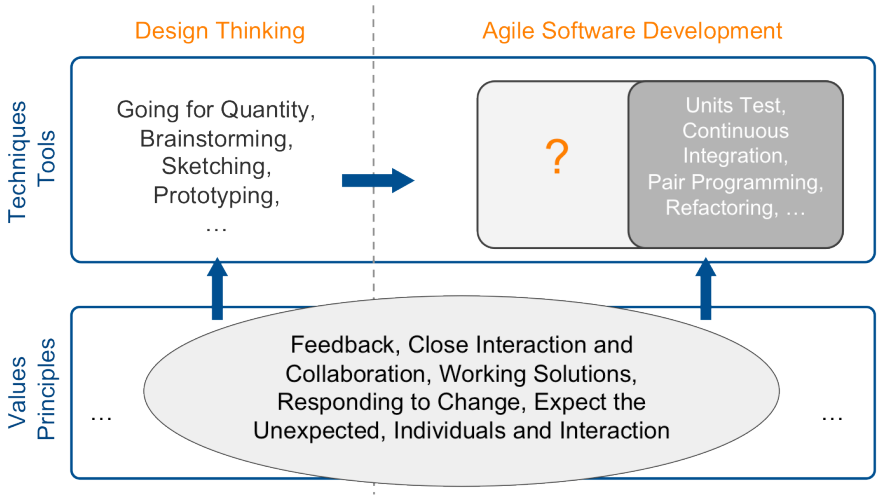

Fig. 1 Learning from Design Thinking.

to be carried far beyond that (Fig. 1). Developers are constantly involved in design activities while working on a software system. This includes, for example, the selection and representation of domain concepts and the organization of programs in logical units and code entities. Main goals of these design activities are conceptual integrity and ease of understanding. These characteristics are important as software systems are improved and enhanced over time. Requirements change if new functionality needs to be supported and existing functionality must be modified and updated. Every such change builds on the system's current design. New features and modified requirements can be realized more easily, if the system's design features simplicity and ease of understanding. Thus, keeping the software system as simple as possible is an important design goal. Following this line of thought, programming can be regarded as a design discipline that has programmers as affected users of the design outcome.

While programming arguably involves design, knowledge about design has gained less attention in the field of programming. Driven by pure curiosity and also economical interest, the nature of design has been studied since decades [10]. Design-related aspects has been investigated from various perspectives ranging from social sciences over artificial intelligence to brain research, considering design as a collaborative endeavor, as a problem-solving activity, as a conversation with materials, or as hard work towards creative leaps, amongst other. Efforts are put to scientise design [10] to allow for better reflection on design activities and to develop theories and methods that may provide guidance if needed. All this investigations led to a huge body of knowledge about design, the application of which should not be limited to interface design and end-user interaction.

We investigate the transfer and application of design knowledge to programming activities and the design of software systems. To take advantage of experience from the design domain, software developers need to be provided with both methods and tools that allow them to work and interact with their materials and artifacts as designers do with theirs. We expect that the transfer of such methods and the provision of accompanying tools allows developers to work more efficiently on design tasks.

In this chapter, we present our first results of this research effort. We applied two key concepts of design practice to improve on development support for programming activities, which are described in the next section and the section after next respectively. First, we present continuous selective testing, our approach to provide for continuous feedback on current coding activities and thus allows for instantly assessing their effect. Second, our interactive approach to run-time analysis provides for immediate access to visualizations of run-time information, which arguably support understanding abstractions.

2 Continuous Feedback on Programming Activities

A manual and explicit activity, the frequent selection and execution of tests require considerable discipline. Our approach automatically derives a subset of tests based on actual modifications to the code base at hand, then continuously executes them transparently in the background, and so supports developers in instantly assessing the effect of their coding activities with respect to the overall set of unit tests to be passed. We apply techniques of selective regression testing, mainly relying on dynamic analysis. By taking advantage of the internal program representation available in IDEs, we do not need to rely on expensive comparisons of different program versions to detect modified code entities.

2.1 Motivation

Test-driven development [5] (TDD) is a cornerstone of agile software development methodologies such as Extreme Programming [19] (XP). This technique suggests writing test cases before the code they are intended to cover. Written first, tests serve multiple purposes. First, they represent a specification for the system to be developed. Next, they document the system and help other developers in comprehending the system. Finally, they ensure that every single change violating one of the required features described in the executable form of a test is reported.

While testing is an important part of regular development activities, Integrated Development Environments (IDEs) have little support for selecting and (re-) executing tests relevant with respect to modifications applied to the system under development [16].

There are a few approaches that support (re-) running the test suite automatically every time a file is saved in the IDE [27, 16]. However, *test selection* as such is traditionally not performed: it is always the complete test suite that is run, including irrelevant tests, leading to an execution overhead that is larger than it actually needs to be.

For that reason, developers often manually select a few tests that seem appropriate, run them explicitly, and wait for feedback. The manual, regular, and explicit selection and execution of tests requires considerable discipline. Moreover, success is guaranteed only if no relevant test cases are omitted in the selection. A solution that automatically selects test cases to be executed in the background based on the applied changes to source code is preferable.

Approaches to test case selection are established: *Selective regression testing* [24] has long been a subject of research. Selective regression testing is concerned with reducing the set of tests that need to be executed to detect failures caused by recent modifications to the code base. However, researchers have not yet investigated the potential of integrating this technique into an IDE and having selected tests execute continuously in the background.

We suggest selecting and executing tests automatically whenever the code status demands this. More precisely, it would be desirable to have support for TDD that, whenever source code is changed, *automatically executes exactly those tests that are affected by the actual modification*, giving developers instant feed-back on whether the applied change breaks something or not.

In this section, we describe *continuous selective testing* (CST) and present an implementation thereof in Squeak Small-talk[1] [18]. Using an implementation of the suggested approach, developers will be supported as follows:

- Sets of relevant tests are selected based on dynamic analysis during the regular execution of tests,
- Relevant tests are executed continuously in the background after every modification to the code base,
- Developers are instantly informed about places in code that, resulting from an applied change, are no longer covered by tests,
- The introduction of new defects is made apparent immediately, which in turn lets developers focus on problems right away.

With that, our approach significantly improves on the way IDE tools provide immediate feedback in a development process adopting TDD.

The main contributions of this work are as follows:

- We present continuous selective testing as an approach relieving developers from the burden to select and run tests explicitly,
- We describe how test case selection in general can benefit from the internal program representation already available in IDEs and how differencing of two versions of a program can be avoided,

[1] `www.squeak.org`

- We describe our approach to test case selection based on dynamic analysis, being not limited to statically-typed languages.

The remainder of this section is organized as follows. First we summarize TDD and state of the art in tool support. Afterwards we motivate the need for improvement and describe our proposed CST approach.[2]

2.2 Background

We briefly introduce the terms and concepts of TDD. We then dis-cuss current practices of developing tests and application code in accordance with TDD and point out the need for better tool support. Afterwards, we introduce the concepts of regression test selection and discuss current approaches.

The Three Phases of Test-driven Development

Test-driven development distinguishes three phases of development [5]:

- **Red** Tests are written that specify new requirements on the system in an executable manner. When these new tests are run for the first time, failures or errors occur, as the system does not yet support the new requirements. An important guideline is to avoid writing application code if there is no test case that fails.
- **Green** Developers enhance the code base to make the failed test "green", i. e., run successfully. It is recommended to add only functionality that is essential to the test in question. A successful test signals that the developer is done implementing the new requirement. Note that it might happen that the system already fulfills a newly defined requirement, without adding new code.
- **Refactor** The developer refactors towards the simplest design they can imagine. By definition of refactoring [11], new functionality must not be added during this phase. The tests can ensure that all required and specified features work after a refactoring. Running tests after each and every little change helps to avoid breaking features and provides instant feedback.

We can observe that tests and the regular execution of tests play an important role when developers employ the principles of TDD.

Tool Support for Test-driven Development

Best practices in working with tests suggest to make only small changes and run tests immediately afterwards to get feedback. This suggestion is based, amongst others, on the following observations:

[2] The evaluation of CST is described in the original paper [29].

- Implementing new application functionality is a very complex activity. As every single step is inherently fault-prone, regular feedback is essential for detecting faults.
- Modifying source code without breaking existing functionality is also difficult. Adapting source code to new requirements or refactoring source code to a simpler design requires very detailed understanding, which to acquire is hard since source code abstracts from concrete execution paths. Having tests covering all parts of the respective code entities and running these tests regularly helps to detect faults early.
- The more steps are passed without getting feedback, the more difficult locating the source of a fault becomes. When a couple of source code entities are changed without running tests, and one ore more tests fail later on, isolating the modification that has caused the failure is not straightforward. Typically, developers are unaware of the complete set of modifications done before running the tests. Moreover, multiple failures might have different causes and combinations of modifications might lead to completely unexpected behavior. To locate the defects, developers can revert modifications step by step or debug the current version. Both ways are tedious and time-consuming.

Running tests often and regularly helps developers to detect faults early, reduces the time required to localize defects, and gives confidence for the next adaptions and refactorings. However, running tests as often and regularly as suggested requires much discipline.

The necessary discipline is sometimes hard to bring up, for apprentices as well as experts. It is all too easy to ignore TDD theory, though well-understood and accepted, and continue modifying code without running tests. It is not necessarily only external factors, such as project schedules, that influence such decisions, but also internal ones like the strong will to finish a task. These aspects contradict with the required discipline.

Another issue with the theory of testing and test-first development is the implicitness of the relationship between test cases and application code they cover. When code is refactored or new features are implemented, existing code has to be modified. However, while developers are aware of recently implemented tests, they cannot know the set of all tests relying on a particular method. Hence, developers do not know the set of tests to be executed after a modification of a particular method. Consequently, all tests should be run after each modification, which is, however, increasingly time-consuming as projects grow. As a result of this, developers run only some tests regularly and the suite of tests is rarely executed, e. g., during integration builds.

Both aspects discussed above, the implicitness of the relationship between test cases and application code as well as the discipline required to run tests after each modification, question the usefulness of tests and test-first development. Our work provides tool support for TDD that alleviates these limitations and strengthens the benefits of testing.

2.3 Continuous Test Queuing, Selecting, and (Re-)Executing

In this subsection, we describe our approach called continuous selective testing (CST). It enables the continuous execution of selected tests directly after code modifications. Such automation relieves developers from the burden of executing tests manually. Selecting a subset of all tests and omitting those that cannot reveal faults reduces execution time and helps to provide feedback instantly. We have implemented the suggested approach in Squeak Smalltalk.

In the following, we will first introduce the concepts of regression test selection and then present the use of the IDE's program representation to detect and handle modifications to the code base. After that, we describe the queuing of tests and the selection and (re-)execution of tests according to the modification at hand. Finally, we present our extensions to the IDE providing instant feedback on test results.

Regression Test Selection

Regression testing refers to the practice of validating modified software; in particular, asserting that applied changes do not affect the software adversely [15]. The simplest approach to regression testing is to reuse the test suite used to exercise the previous version of the software. Fully running a large test suite can be unnecessarily costly, e. g., if only a few parts of the system were changed.

A technique to reduce the number of tests is *regression test selection*. It selects tests that have to be re-run to reveal a fault resulting from a particular change. Selecting an optimal set of tests is, however, generally inefficient [24]. Still, the set of tests traversing modifications can be computed efficiently. This set of *modification-traversing tests* can be considered a superset of the *fault-revealing tests* when the *Proper Regression Testing Assumption* [24] holds (P refers to a program and P' refers to the modified version of this program):

> When P' is tested with t, we hold all factors that might influence the output of P', except for the code in P', constant with respect to their states when we tested P with t.

A regression test selection technique is furthermore considered *safe* if it ensures to not omit tests revealing faults [15]. Several safe techniques have been proposed for purely procedural (e. g., [2, 8, 25]) as well as for object-oriented programming languages (e. g., [26, 15]). Object-oriented programming is special as inheritance, polymorphism and thus late-binding have to be considered.

The most efficient and safe test selection technique is based on detecting modified code entities, such as functions or storage locations [24]. This technique was first implemented in TestTube [8] for software written in C. The technique is based on dynamic analysis [3]; test coverage information are recorded during each test run. For a new version of a software, the set of modified code entities can be detected. Based on coverage information, the technique selects and re-executes all tests that exercised the modified code entities in the previous version of the software. For object-oriented languages, the modified entity selection technique requires

additional considerations due to language features such as inheritance and polymorphism enabling late binding.

Our approach, CST, is based on this technique of detecting modified code entities. CST records coverage information and selects tests on a method level. This procedure may select tests that do not traverse the modifications, because a test might only traverse unmodified parts of a method, for example. However, tracing on a more fine-grained level is much more expensive and does not pay off unless methods contain many control blocks [6].

Propagating Modifications to the Code Base

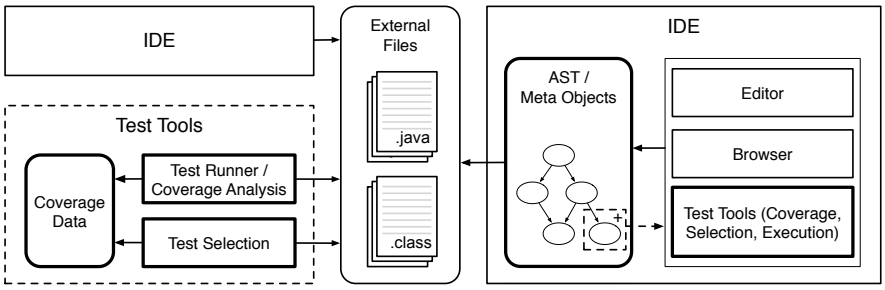

Fig. 2 The left-hand side shows a traditional setup where test selection tools and IDE work independently of each other. The right-hand side depicts CST integrating test selection into the IDE and taking advantage of the internal program representation.

Most approaches to test selection are based on comparing the new with an earlier program version to detect change entities. Our approach takes advantage of an IDE's internal program representation. Fig. 2, on the left, depicts the setup of traditional approaches. IDE and test tools are not integrated and do not work together, each of them works rather separately on external program representations. In this setup, however, a test selection technique requires a comparison of program versions to detect modifications between two versions of a software. There exist differencing concepts and tool for both source code [1, 15] and byte code [17].

We suggest to better integrate the tools for testing and test selection into the IDE as depicted on the right of Fig. 2. Every modification applied to the code base can produce an event notifying the IDE about the respective change. Using this notification mechanism, the test tools can process each modification to the code base. The tools are now able, for example, to automatically select and re-execute a set of test cases as necessary for the modification applied.

The set of events used to propagate code modifications to IDE tools has to be designed for the particular programming language and IDE, respecting the features of the language and the architecture of the IDE. In Squeak Smalltalk, for exam-

ple, there are basically two operations to create or modify code objects. Sending a subclass-message to a class *c* creates a new or modifies an already existing subclass of class *c*. Sending the *compile:* message to a class object allows to compile a source code text of a method and puts it in the method dictionary of the corresponding class. Based on the effects of this two operations, the following change events can be defined for the Smalltalk [13] programming language, which is a rather simple language and does, for example, not provide any visibility modifiers; *class added*, *class removed*, *superclass changed*, *instance variable added*, *instance variable removed*, *method added*, *method modified*, and *method removed*. Note that class-specific ("static") state or behavior do not require special treatment as classes are also normal objects whose state and behavior are defined by meta-classes.

Queuing and Executing Tests for TDD

CST builds upon a well-defined set of different kinds of modification to the system. The event mechanism described above, with the possible modification events it includes, allows for the continuous selection and execution of tests according to the current state of development.

Our approach distinguishes code entity modifications by their referal to test case code or non-test code. By convention, those methods of a class extending *TestCase* that are prefixed with *test* are treated as *test case methods*. Source code entities of test classes that are non-test methods, that is, attributes, *setUp*, *tearDown*, and other utility methods, are treated equally to application code.

When the creation of a test case method or modifications to one are reported, the developer is assumed to be in the red phase of the TDD cycle. The test runner will immediately execute the corresponding test case and provide instant feedback on the result. If the test fails, it will be queued. Failed test cases will be re-executed whenever a modification not related to a test method is reported. Now the developer is expected to be in the green or refactor phase, so the change has the potential to fix a test. All tests that still fail stay in the queue. A change of an entity can fix one or more tests cases, but the change can also introduce a fault that breaks other test cases. All test cases that might be affected by the reported change need to be re-executed. A technique to select the corresponding test cases is presented in the next subsection. The tests in the queue, failed before, are run first, providing earlier feedback on whether the current modification makes the failed test(s) pass.

To provide feedback on the test runs, we extended the tools for browsing and editing code. Whenever a modification is reported and the test runner executes tests, a newly introduced GUI widget will inform the developer about the test runner's activities and the current status of the test result (Fig. 3). The widget turns red as soon as one test has failed. Tests are executed in a background process allowing the developer to navigate to the next code entity of interest and start editing it.

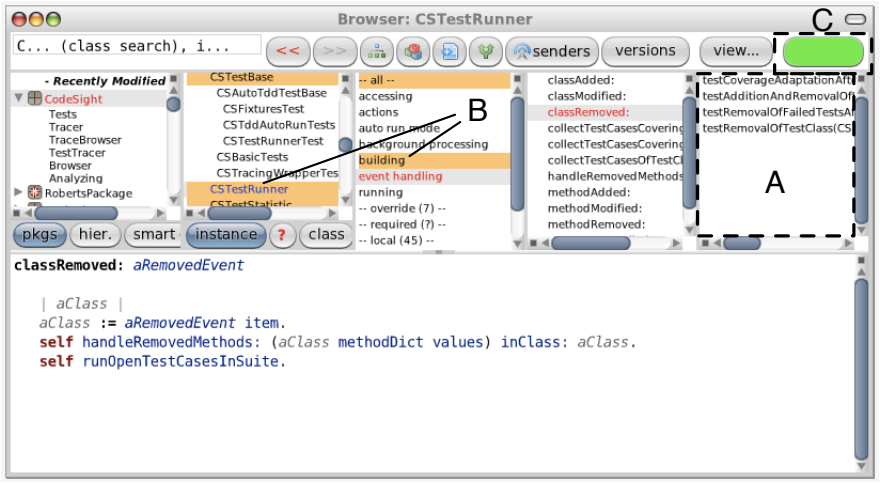

Fig. 3 An extended code browser in Squeak; having an additional panel on the right (A) that shows test cases covering the selected method named *classRemoved:*. Uncovered classes and methods are highlighted (B). A new widget (C) informs the developer on the current status of the test runner; whether it is currently running tests, and about the number of tests that have failed.

Re-executing Selected Tests for OO software

The set of tests to be re-executed for an applied change should be minimized. CST relies on collecting test coverage information, and using this information to select tests that might be affected by a modification.

Using this coverage information of previous test runs, the CST tools can determine the set of tests that is to be re-executed for any reported change. Selecting af-fected test cases is a two-step procedure:

1. If a non-test method is modified, the test runner collects and re-executes all test cases that covered this method previously. Therefore, the test runner can simply navigate the coverage relationship between the corresponding method objects.
2. CST also deals with modifications such as adding a method or changing the superclass that might affect late-bound method invocations. When, for example, an application method m' is added to a class c', and m' overrides a method m in a superclass c, CST will execute tests that have covered m'. More precisely, it will select those tests that previously exercised m for instances of c.

As mentioned above, the set of meaningful events, which reports modified code entities, may vary between languages providing different sets of features. The algorithms to be applied to determine a safe set of tests may vary as well. If the language supports multiple inheritance, for example, the algorithms have to consider the possibility of multiple superclasses and the respective linearization order applied to method dispatch.

As pointed out in [15], a safe test selection technique for object-oriented software must also consider exception handling. CST allows to consider exceptions similarly to other code entities. A basic method constructing an exception object needs to be instrumented; for instance, default constructors in Java, or *basicNew* in Smalltalk. Using the receiver's dynamic type recorded for each method call, we can determine whether an exception was created and thrown during the execution of a test case. If the exception class hierarchy is changed, all test cases that might be affected can be identified easily.

Establishing a Coverage Relationship

Test coverage information used for test selection is collected during regular test execution. We decided to collect this information only for packages and classes of interest. This typically excludes basic development classes such as the collection or system libraries. The selection of relevant packages and exclusion of others avoids unnecessary overhead [14]. To record method coverage information, we use method wrappers [7]. Actual method code is wrapped in tracing code that records the call of the wrapped method in the context of the currently running test case, and forwards the sent message to the wrapped method afterwards.

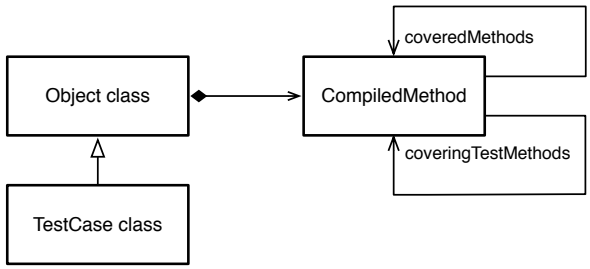

Fig. 4 The coverage relationship between test methods, included in *TestCase* classes, and application methods covered by them.

Test coverage information is integrated into the IDE's program representation. In CST, we establish and maintain a coverage relationship between test case methods and methods covered during test execution, as depicted in Fig. 4. Here, we generally refer to objects representing methods in the IDE; Squeak Smalltalk provides so-called *CompiledMethod* objects to reflect upon and work with methods in the system.

Employing the test-first principle and using CST, tests run frequently and the coverage relationship has to be maintained for test runs. To avoid unnecessary start-up costs, tracing logic is installed incrementally after each compilation step. When the developer selects packages and classes of interest, wrapper logic is initially in-

stalled. If source code entities matching the selection criteria are added, they are wrapped directly after creation. This incremental approach avoids the need to instrument source code for each test run.

Using CST, developers can also be provided with instant feedback regarding test coverage. Classes and methods that are not covered any more are highlighted in the code browser (Fig. 3). The feedback supports developers in ensuring high method coverage. We further extended the code browser with an additional fifth panel (Fig. 3) that shows all test cases covering the currently selected method. This extension makes the coverage relationship visible and the applied test selection technique transparent for developers.

3 Providing Examples to Support Learning the Abstract

Visualizations of actual run-time data support program comprehension, like examples support the explanation of abstract concepts and principles. Unfortunately, the required run-time analysis is often associated with an inconvenient overhead that renders current tools impractical for frequent use.

We describe our interactive approach to collect and present run-time data. An initial shallow analysis provides for immediate access to visualizations of run-time information. As users explore this information, it is incrementally refined on-demand. We present an implementation that realizes our proposed approach and enables developers to instantly explore run-time behavior of selected code entities. Our empirical evaluation shows show that run-time data for an initial overview can be collected in less than 300 milliseconds for 95 % of cases.

3.1 Motivation

Developers of object-oriented software systems spend a significant amount of time on program comprehension [9, 4, 20]. They require an in-depth understanding of the code base that they work on; ranging from the intended use of an interface to the collaboration of objects, and the effect of a method activation during this collaboration. Gaining an understanding of a program by reading source code alone is difficult as it is inherently abstract.

The visualization of run-time information supports program comprehension as it reports on the effects of source code and thus helps understanding it. At run-time, the abstract gets concrete: variables refer to concrete objects and messages get bound to concrete methods. For example, profilers and debuggers support run-time exploration to answer questions such as: “What is the value of a particular method argument?” or “How does the value of a variable change?”

Unfortunately, the overhead imposed by current tools renders them impractical for frequent use. We argue that this is mainly due to two issues: a) Setting up an

analysis tool usually requires a significant configuration effort, as well as a context switch, b) performing the required in-depth analysis is time-consuming. Both issues inhibit immediacy and thus discourage developers from using these tools frequently.

We argue that the overhead imposed by current approaches to dynamic analysis is uncalled-for and that immediate accessibility of run-time information is beneficial to program developers. Continuous and effortless access to run-time views on source code supports developers in acquiring and evaluating their understanding. Run-time views are based on actual data. Thus, they arguably encourage the evaluation of assumptions and eliminate space for speculation.

We employ a new approach to dynamic analysis enabling a feeling of immediacy missing from current tools. The central contributions of this work are:

- A novel approach to dynamic analysis based on a shallow analysis and detached in-depth on-demand refinements,
- A realization of this approach by providing an integrated tool for accessing run-time information during program development,
- Empirical results to evaluate our claims with respect to feasibility.

We first highlight the benefits of dynamic views for program comprehension and discuss desired tool characteristics. Afterwards, we present our interactive approach to dynamic analysis that collects data exactly when needed.[3]

3.2 Background and Motivation

Due to its abstract nature, source code provides a limited perspective on software systems. Conversely, dynamic views support program comprehension as they aid developers in understanding how a system works. In this section, we illustrate this by means of a running example. We continue by discussing requirements that visualization tools should meet to encourage their frequent adoption in practice.

Exploring a Program's Run-time

Visualized run-time information helps developers to better understand program behavior. In our running example, a developer faces the task of understanding a simple clock application, which provides an analog and digital view. Figure 5 shows the structure of the application that is based on the Observer design pattern [12]. The `ClockTimer` subject represents a ticking clock, whose instances either of the two concrete observers can display. Each `tick` invocation notifies the observers about the change of state.

The developer in our example is unaware of these internals, but can use visualized run-time information to learn about them, and to eventually discover the Observer usage. This process could look as follows.

[3] The empirical evaluation of this approach is described in the original paper [22].

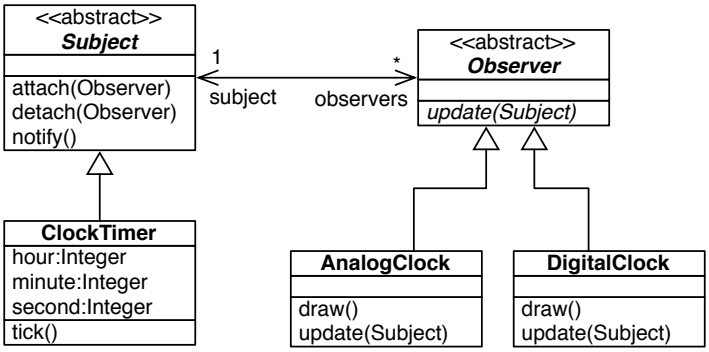

Fig. 5 Observer pattern running example.

The visualized information in Figure 6 primarily consists of a call tree that reflects a particular run of the application. A call tree provides comprehensive information of the entire program execution rather than a single execution path. Some of the tree nodes have been expanded to reveal details: for instance, it is evident that `tick` invokes `notify` (at index 1).

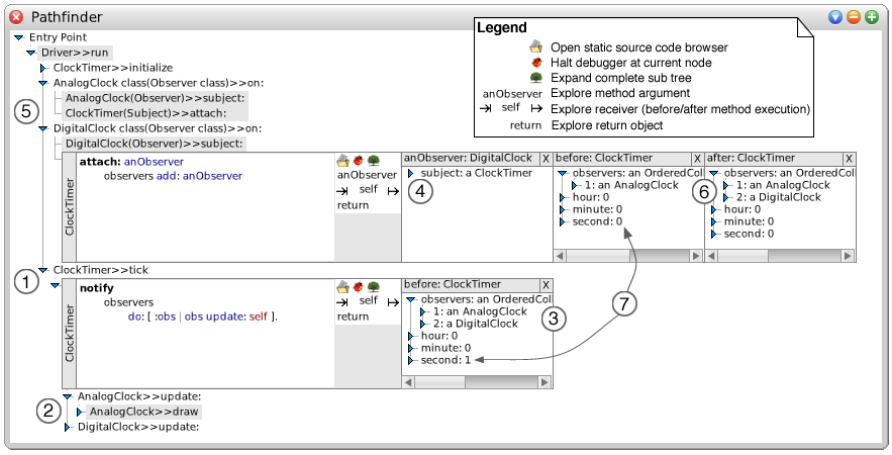

Fig. 6 Pathfinder is our interactive dynamic analysis tool for the Squeak IDE.

The figure shows (at index 2) that `notify` sends the `update:` message to two different clocks. From this information, the developer can conclude that there exist two observers, and ascertain this by inspecting the run-time state information attached to the execution of `notify`. The object explorer view at index 3 confirms that the `observers` list contains two clock objects. Moreover, index 4 highlights that a `ClockTimer` participates as the subject in the Observer pattern.

The provided run-time view helps to answer follow-up questions. For instance at index 5, the developer speculates that `attach:` is responsible for registering observers. In an expanded `attach:` invocation, at index 6, the combined *before* and *after* views of a method node execution show how a `ClockTimer` registers a `DigitalClock` observer. As another example, index 7 marks two views that show how the state of the subject changes after a `tick` execution. If interested, the developer could now further examine the implementation of that method to continue exploring.

In a nutshell, the developer is able to identify the conceptual structure of the Observer pattern as part of the application. In addition to comprehending structural aspects, the developer also gains deep insight about the interactions of structural entities at run-time.

Visualized run-time information sensibly augments the information available from static views on applications, e. g., their source code. For instance, the authors of the *Gang of Four* book on design patterns [12] aid comprehension of their examples in readers by presenting sequence diagrams alongside class diagrams to visualize collaborations among objects.

Visualizations of run-time data make the mental model readily available and obviate its manual elaboration. There exist valuable approaches to building mental models of software systems from static representations. IDEs support developers in navigating a code base, for example by tracing message sends, in order to understand how a system works. However, visualizations such as call trees put application source code and structure into meaningful behavioral contexts, and object explorers provide actual examples of objects rather than their abstract names.

The Need for Immediacy

Tools providing such visualizations of run-time data should allow for a feeling of immediacy to encourage frequent use. To that effect, two characteristics should be met. Firstly, visualization tools have to be integral parts of the programming environment. Developers would welcome a tool carrying them from method source code to the visualization of an actual run of the same method by means of one click. Secondly, response times have to be low. Visualized run-time information has to be available within some hundreds of milliseconds rather than minutes [28]. However, immediacy must not hamper the level of visual detail.

We intend to support program comprehension by reducing the effort of accessing run-time information. We aim to encourage developers to use our tools frequently. Developers shall be able to avoid guesswork and validate assumptions by inspecting actual run-time information instead. The main question that our work addresses is how to make dynamic analysis results available to developers immediately.

Immediacy through Interactivity

Our interactive approach to dynamic analysis enables immediacy. Traditional approaches are time-consuming as they capture comprehensive information about the entire execution up-front. Low costs can be achieved by structuring program analysis according to user interaction. More specifically, user interaction allows for dividing the analysis into multiple steps: A high-level analysis followed by on-demand refinements. This distinction reduces the overhead to provide visualizations of run-time information while preserving instantaneous access to detailed information.

Step-wise Run-time Analysis

Splitting the analysis of a program's run-time over multiple runs is meaningful because developers typically follow a systematic approach to understand program behavior. For example, in our scenario (Section 3.2), the developer first uses the presented call tree to gain an initial understanding (1). Later on, the developer identifies execution paths that lead to the population of the list of observers by inspecting relevant state (2). More generally, program comprehension is often tackled by exploring an overview of all run-time information, and continuing to inspect details.

This systematic approach to program comprehension guides our approach to dynamic analysis: Run-time data is captured when needed. (1) A first *shallow analysis* focuses on the information that is required for presenting an overview of a program run. For example, method and receiver names are sufficient to render a call graph as presented in Section 3.2. Further information about method arguments or instance variables are not recorded. (2) As the user identifies relevant details, they are recorded on-demand in additional *refinement analysis* runs. In our example, the developer clicks on the `observers` variable to see registered clocks. Information about instances contained in the list are recorded in a separate run triggered by user interaction.

This interactive approach to dynamic analysis requires the ability to reproduce arbitrary points in a program execution. In order to refine run-time information in additional runs, we assume the existence of entry points that specify deterministic program executions. For our implementation, we leverage test cases as such entry points, as they commonly satisfy this requirement [21]. However, our approach is applicable to all entry points that describe reproducible behavior.

Less Effort through Step-wise Analysis

Splitting run-time analysis and refining the results on-demand reduces the effort for providing an initial overview, as well as comprehensive details. The amount of required data for generating a run-time visualization to support an inital overview is limited compared to the information that is generated in an entire program run. The data on method activations is sufficient to render the call tree in our example. More

specifically, the overhead for collecting method name and receiver information is significantly less than performing a full analysis. A full analysis includes recording exhaustive information before each state change in the execution of a program. In contrast to performing a complete analysis up-front, minimizing the collected data imposes a reduced overhead with respect to the execution of the instrumented program.

User interaction with the initial overview can be leveraged to minimize the overhead of refinement analysis. As the user expresses interest in individual objects at explicit points of the execution, required information is loaded on-demand in additional analysis steps. Such a refinement step involves recording of object state at the specified point in execution. While recording object state may be time-consuming in general, we limit the extent of data collection: a refinement step imposes a minimal overhead by focusing on a single object at a particular execution step. This means that refinement analysis is hardly more expensive than execution without instrumentation.

Our approach divides the effort for dynamic analysis across multiple runs. The information required for program comprehension is arguably a subset of what a full analysis of a program execution can provide. While our approach entails multiple runs, the additional effort is kept to a minimum, especially when compared to a full analysis that has no knowledge of which data is relevant to the user. We reduce the costs by loading information only when the user identifies interest. This provides for quick access to relevant run-time information without collecting needless data.

Our tool Pathfinder (Figure 6) realizes the described interactive approach to dynamic analysis. It is integrated into the the Squeak Smalltalk IDE following our objective of achieving a feeling of immediacy. Pathfinder demonstrates the feasibility of our approach.[4]

4 Summary

In this chapter, we have reported on two improvements that are based on key concepts in design practices. We argued that programming involves design in several respects. Developers constantly prepare the program to reduce complexity whenever possible so that future coding activities remain feasible. This gives reason for investigating the transfer of design knowledge and its application to the methods and tools for programming tasks.

First, our idea of *continuous selective testing* (CSP) and its implementation in the Squeak/Smalltalk programming environment relieves developers from manually selecting and executing tests. Based on actual modifications, a selected set of tests is executed transparently in the background, reporting instantly on the effect of the applied changes with respect to the overall set of tests to be run. Our test selection technique is based on dynamic analysis and thus does not require a statically typed

[4] A screencast is available online at
`http://www.hpi.uni-potsdam.de/swa/projects/pathfinder/`

language for offline processing. It is the first approach to test selection that benefits from of run-time type information to reduce test sets.

Second, our interactive approach to collect and present run-time data helps developers to understand program behavior. We argued that user interaction can be leveraged to distribute dynamic analysis across multiple runs. Our combination of dynamic analysis and user interaction reduces the effort for providing an initial overview of a program's execution. Refinement steps provide relevant details on-demand and are associated with much lower costs. With Pathfinder we have shown that our approach can enable immediate access to run-time views for code entities at the push of a button.

References

1. T. Apiwattanapong, A. Orso, and M.J. Harrold. JDiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering*, 14(1):3–36, 2007.
2. T. Ball. On the limit of control flow analysis for regression test selection. *ACM SIGSOFT Software Engineering Notes*, 23(2):134–142, 1998.
3. Thomas Ball. The Concept of Dynamic Analysis. In *ESEC/FSE*, pages 216–234, 1999.
4. Victor R. Basili. Evolving and packaging reading technologies. *J. Syst. Software*, 38(1):3–12, 1997.
5. K. Beck. *Test-driven Development: By Example*. Addison-Wesley Professional, 2003.
6. J. Bible, G. Rothermel, and D.S. Rosenblum. A comparative study of coarse-and fine-grained safe regression test-selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):149–183, 2001.
7. J. Brant, B. Foote, R.E. Johnson, and D. Roberts. Wrappers to the Rescue. In *ECOOP*, pages 396–417, 1998.
8. Y.F. Chen, DS Rosenblum, and K.P. Vo. TestTube: A system for selective regression testing. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, pages 211–220, 1994.
9. Thomas A. Corbi. Program Understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.
10. Nigel Cross. Designerly Ways of Knowing: Design Discipline Versus Design Science. *Design Issues*, 17(3):49–55, 2001.
11. M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
13. A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
14. T. Gschwind and J. Oberleitner. Improving Dynamic Data Analysis with Aspect-Oriented Programming. In *CSMR*, pages 259–268, 2003.
15. M.J. Harrold, J.A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S.A. Spoon, and A. Gujarathi. Regression Test Selection for Java software. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 312–326. ACM New York, NY, USA, 2001.
16. M.J. Harrold and A. Orso. Retesting Software During Development and Maintenance. *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 99–108, 2008.
17. S. Huang, Y. Chen, J. Zhu, Z.J. Li, and H.F. Tan. An optimized change-driven regression testing selection strategy for binary Java applications. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 558–565. ACM New York, NY, USA, 2009.

18. Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *OOPSLA*, pages 318–326, 1997.
19. K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2004.
20. Andrew J. Ko, Robert DeLine, and Gina Venolia. Information Needs in Collocated Software Development Teams. In *ICSE*, pages 344–353, 2007.
21. Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall, 2006.
22. Michael Perscheid, Bastian Steinert, Robert Hirschfeld, Felix Geller, and Michael Haupt. Immediacy through Interactivity: Online Analysis of Run-time Behavior. In *17th Working Conference on Reverse Engineering*, pages 77 – 86, Beverly, USA, 2010. IEEE.
23. Jens Lincke Robert Hirschfeld, Bastian Steinert. Agile software development in virtual collaboration environments. In Larry Leifer Hasso Plattner, Christoph Meinel, editor, *Design Thinking – Understand, Improve, Apply*, pages 197 – 218. Springer, 2011.
24. G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.
25. G. Rothermel and M.J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, 1997.
26. G. Rothermel, M.J. Harrold, and J. Dedhia. Regression Test Selection for C++ Software. *Software Testing, Verification & Reliability*, 10(2):77–109, 2000.
27. D. Saff and M.D. Ernst. Reducing wasted development time via continuous testing. In *ISSRE*, page 281, 2003.
28. Ben Schneiderman. *Designing the User Interface*. Addison-Wesley, 1992.
29. B. Steinert, M. Haupt, R. Krahn, and R. Hirschfeld. Continuous Selective Testing. In *XP*, pages 132–146, 2010.