Explicit Use-case Representation in Object-oriented Programming Languages

Robert Hirschfeld Michael Perscheid Michael Haupt
Software Architecture Group, Hasso Plattner Institute, University of Potsdam, Germany
{firstname.lastname}@hpi.uni-potsdam.de

Abstract

Use-cases are considered an integral part of most contemporary development processes since they describe a software system's expected behavior from the perspective of its prospective users. However, the presence of and traceability to use-cases is increasingly lost in later more code-centric development activities. Use-cases, being well-encapsulated at the level of requirements descriptions, eventually lead to crosscutting concerns in system design and source code. Tracing which parts of the system contribute to which use-cases is therefore hard and so limits understandability.

In this paper, we propose an approach to making use-cases first-class entities in both the programming language and the runtime environment. Having use-cases present in the code and the running system will allow developers, maintainers, and operators to easily associate their units of work with what matters to the users. We suggest the combination of use-cases, acceptance tests, and dynamic analysis to automatically associate source code with use-cases. We present UseCasePy, an implementation of our approach to use-case-centered development in Python, and its application to the Django Web framework.

Categories and Subject Descriptors D.1.5 [*Software*]: Programming Techniques—Object-Oriented Programming; D.3.3 [*Software*]: Programming Languages—Language Constructs and Features

General Terms design, languages

Keywords use-cases, separation of concerns, traceability

1. Introduction

Users describe desired system behavior from a domain-centric point of view, deliberately avoiding technical details that are not part of their domain. Based on the recording of user-system-interactions, *use-cases* [20] represent one of the more recent approaches to capturing such descriptions to cover all observable behavior at a system's boundary from its users' or clients' points of view. Each such description is comprised of simple steps written down informally, covering technical details only if necessary. With use-cases, customers can conveniently express what they expect from a system, and developers can treat the system boundaries and all interactions across them as a foundation for modeling, implementation, testing, and even deployment. Therefore use-cases are regarded as important linking elements between the different perspectives of developers and users.

Developers frequently have to associate requirements expressed by a customer or user with the responsible locations in a system's implementation [32]. This, in turn, is often rather complex and only partly understood. In many cases, there is no simple one-to-one correspondence so that developers have to discover these mappings between users' descriptions and implementation details by themselves. Developers have to consider groups of instances, classes, and modules since a system's behavior emerges from the interaction of its parts. Unfortunately they often have great difficulty in recognizing what part of the system implements what use-case and vice versa. Contemporary object-oriented programming languages are very limited when it comes to reflecting the users' perspectives in the actual source code of the system [34].

In this paper, we propose the representation of use-cases as first-class entities in object-oriented programming languages and runtime environments. By making use-cases explicit in source code and at run-time, they can benefit the entire software life-cycle. *Use-case-centered development* extends the presence of use-cases from requirements engineering to implementation, testing, and run-time. By being able to easily identify use-cases always and everywhere, change requests or failure reports can be mapped onto corresponding user descriptions as well as source code entities.

The contributions of our paper are as follows:

- We introduce a new programming construct called `usecase` supporting use-case layers orthogonal to a system's class hierarchy. With this construct, developers can annotate methods to associate them and the classes in which they are implemented with the use-cases they contribute to. Moreover, we preserve this information until run-time so that both developers and tools are able to inspect and debug use-cases and their participating objects from within the system. We have implemented *UseCasePy* in the Python programming language to support our notion of *use-case-centered development*.
- We further present a use-case recovery technique that propagates use-case-associations by using dynamic analysis in combination with acceptance tests and use-case meta-objects. Developers need only annotate acceptance tests with corresponding use-cases and everything else, from test execution to source code annotation, will be done automatically and can so be repeated as often as necessary without additional burden to developers.
- We apply our approach to the Django Web framework [11]. With the help of our use-case recovery technique, we are able to identify more than 35 use-cases and automatically annotate large parts of Django with less effort. Based on our added lan-

DLS'11, October 24, 2011, Portland, Oregon, USA.

guage construct, we demonstrate how typical maintenance activities such as program comprehension, debugging, and reengineering can benefit from use-case-centered development.

The remainder of this paper is organized as follows: Section 2 discusses challenges related to the traceability of use-cases. Section 3 presents our approach to use-case-centered development. Section 4 explains our implementation in more detail. Section 5 describes the Django case study demonstrating the capabilities of UseCasePy. Section 6 discusses related work. Section 7 concludes our paper and presents ideas for future work.

2. Use-case Traceability

Although use-cases are the linking elements between the several representations of software systems [9], the traceability to them is increasingly lost in later more code-centric development activities. Due to the crosscutting characteristics of use-cases in design and implementation activities [21, 22], their realizing objects end up being scattered over and tangled with the rest of the system. This causes several difficulties that developers face when connecting requirements from the users' or customers' domain with their own implementation-centric perspectives.

2.1 Requirements, Analysis, and Design

The traceability of use-cases between requirements specifications, domain models, and design artifacts is well-established and supported by many tools such as Objectory [29]. A process based on use-cases typically looks as follows: First, requirements formulated in natural language are analyzed and transformed into several use-cases. During analysis, developers review and refine the requirements to acquire a more precise understanding of the application domain and the needs of their customers. Developers specify all the logical or domain objects relevant to the system and their relationships and interactions [20]. Use-cases can now be realized by subsets of the analysis objects based on their contribution through collaboration. This representation is often called *view of participating objects*. Figure 1 shows a group of objects involved in the execution of the dotted use-case (using Jacobson's notation [20]). Triggered by a user-induced stimulus to an interface object, a use-case's behavior is provided by several collaborating objects. This view of participating objects is the use-case description's counterpart *inside* a software system. Both analysis and design activities focus on such object descriptions whereas the traceability relationship is still preserved.

2.2 Use-cases Are Crosscutting Concerns

An object can be part of many use-case implementations, and a use-case implementation can be (and usually is) realized by several collaborating objects. Figure 1 illustrates these relationships. An ellipse denotes a use-case implementation. All objects inside an ellipse contribute to a use-case implementation.

If such a use-case implementation is distributed over several objects, this property is called *scattering*. Sometimes some of the objects and their communication paths are involved in more than one use-case implementation and play different roles in their execution. This property is called *tangling* and represented as the intersection of the two use-cases in Figure 1.

While users typically do not perceive scattering and tangling from their point of view, developers are constantly faced with them: they need to know which relationships exist between particular objects and use-case implementations. Traceability between use-cases and their realizations is needed especially in later development activities to ensure that developers can determine how changes to one use-case implementation affect others.

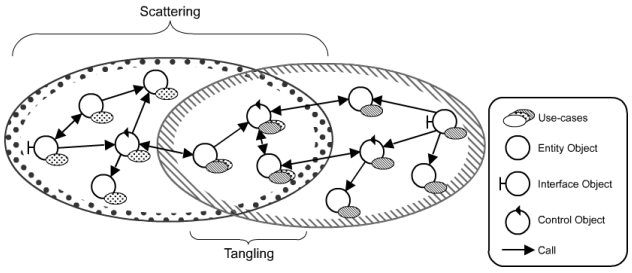

Figure 1. Scattering and Tangling of Use-cases

2.3 Implementation, Testing, and Deployment

One might argue that systems can be implemented straightforwardly based on a proper design model and so guarantee traceability as well. However, studies report that during implementation, design objects are split into approximately one to five classes and there are exceptional cases with up to 17 classes [20]. With that, a simple one-to-one correspondence between design objects and implementation classes is rather an exception than the rule and traceability between them is increasingly lost.

For these reasons, developers cannot determine from the source code alone which parts of the system are part of which use-case implementation and vice versa. We argue that this knowledge is important for implementation and maintenance activities because change requests are described by users and need to be mapped to locations in the source code by developers. Since traceability to use-cases is already lost in the source code, it is also not available during deployment or at run-time. With such traceability, however, we can imagine several scenarios supported by it: debugging of use-cases, reengineering support with use-cases, or deployment on a per use-case basis. While possible in principle, such scenarios involve intensive manual work with a high potential for failure.

3. Use-case-centered Development

Use-case-centered development is based on explicit representation of use-cases in object-oriented languages, addressing the problems related to their absence in later development activities. We introduce a new language construct and meta-objects to combine source code entities and run-time artifacts with the use-cases they contribute to. Based on these concepts and in combination with dynamic analysis, we introduce a process to semi-automatically recover use-case-implementations in legacy systems. A brief discussion of some aspects of the approach concludes this section.

3.1 @usecase Language Construct

To connect scattered parts of use-case implementations semantically belonging together, we introduce the `usecase` construct. It is realized as a source code-level annotation parameterized with unique semantic identifiers of all use-cases to whose implementations the annotated source code entity contributes. Using this construct, developers can easily augment computational units, for example methods, with use-case tracing information, resulting in implementation artifacts carrying information on their associated use-cases as described in requirements engineering. According to its annotation, the function in the code below is involved in the implementations of the two use-cases "draw a line" (`DrawALine`) and "draw a rectangle" (`DrawARectangle`); changes in this function will influence both use-case implementations.

```
@usecase(DrawALine, DrawARectangle)
def drawLine(x1, y1, x2, y2):
    ...
```

We propose to apply the `usecase` annotation at the granularity level of single methods. We believe this to be suitable for developers that want to obtain (and maintain) a sufficient overview of use-case implementations. In contrast, we argue that classes are too coarse-grained targets for annotation since they would likely yield to many use-case annotations per unit (class).

View of Involved Use-cases As mentioned in Section 2.2, one method can be involved in several use-case implementations. The `usecase` construct implicitly supports the expression of use-case implementation relationships and dependencies: Methods without an annotation are *not covered* by any use-case implementation. Methods that are executed in one particular use-case are *specific* to that use-case and annotated with exactly one unique use-case identifier. *Group* annotations describe computational units that are involved in more than one use-case implementation. The `usecase` annotation is parameterized with a list of all involved use-cases. Methods executed in a high number of or all use-case implementations are classified as being *infrastructural*. An empty parameter list describes these relationships to prevent crowded lists in methods with a high number of use-cases. Listing 1 shows four methods with all kinds of annotations and corresponding names.

```
@usecase(usecase1)
def aSpecificFunction():
    ...

@usecase(usecase1, usecase2)
def aGroupFunction():
    ...

@usecase()
def anInfrastructuralFunction():
    ...

def aNotCoveredFunction():
    ...
```

Listing 1. Different Use-case Relationships Expressed by Method Annotations.

The mapping of use-case relationships to the parameter list is easy to understand and extensible. With it how many and which use-case implementations an annotated method contributes to becomes evident. Thus, it can easily be inferred how changes to that method can influence the behavior of other use-case implementations. The language construct is practical for new use-case annotations as well as for changed use-case behavior since the parameter list makes the method's contribution explicit.

Mapping of Scattering and Tangling With our approach, developers can adopt two different perspectives on such annotated source code (see Listing 1). In the *tangling view*, they first look at a method, then its annotation, and finally the relationships linking to several use-case implementations. For instance, the function `aGroupFunction` is involved in the first and second use-case implementation, which is made explicit by its annotation.

Conversely, developers focus first on use-cases when taking the *scattering view*. It considers a given use-case and contains all involved methods, allowing developers to follow a use-case implementation through the system. The scattering view may be problematic as the different methods contributing to one use-case may indeed be defined in different source code units (such as files). Development tools can ease the task of maintaining top-sight by scanning the entire source code of the system, extracting use-case annotation information from it, and preparing a comprehensive view. We have implemented a prototype in the PyDev Python IDE [27]. Figure 2 shows the UseCasePy Layer plug-in in the PyDev workbench. A tree view displays projects with their use-case implementations, involved methods, and dependencies on other use-cases.

3.2 Use-cases as First-class Entities

Source code-level information can be retained as valuable meta-data at run-time. Based on the `usecase` annotation, we introduce use-cases as meta-objects, instances of "use-case template classes" (`UseCase`) that represent desired information about use-cases. Typically, each instance attribute of a use-case template class describes an interesting property of a use-case such as goal, actor, or preconditions. Developers and tools can obtain use-case information directly from the `UseCase` objects. Listing 2 illustrates a possible use-case class template and one instantiated meta-object describing the "draw a rectangle" use-case. The template consists of typical use-case attributes and an external link to a third-party requirements document for more information. The object is bound to the variable name `DrawARectangle`, which corresponds to the use-case identifier.

```
class UseCase(object):
    def __init__(self, name, desc, actor, doc):
        super(UseCase, self).__init__()
        self.name = name
        self.description = desc
        self.actor = actor
        self.externalDocument = doc

DrawARectangle = UseCase(
    "Draw a rectangle",
    "Add and draw a rectangle figure to the canvas.",
    "A typical user",
    File("/paint/usecases/drawRectangle.uc"))
```

Listing 2. Definition of Use-case Meta-objects.

All use-case objects must be unique within the system because their corresponding use-cases from requirements engineering are also unique. Therefore, we suggest a use-case meta-objects repository global to the system that can be accessed from modules with an import statement, thus avoiding accidental multiple creation of use-case meta-objects. The repository can be used to look up identifiers and use-case attributes, and `usecase` method annotations can refer to use-case objects in a consistent manner. Assigning meaningful names to use-case meta-objects by assigning them to global variables named after for example the goal of the use-cases ensures readable unique identifiers.

Use-case meta-objects are connected to methods annotated with the corresponding use-cases. This ensures that the mapping of methods to use-cases is preserved at run-time. Tools can introspect method objects during execution and observe which use-case implementations a specific method contributes to. For instance, it is conceivable to integrate use-case objects into symbolic debuggers and to *step* from one specific use-case method to the next one, ignoring unimportant group and infrastructural methods at run-time. Another example would be to link application objects to the use-cases they participate in, improving developers' understanding of running code. Thereby, use-case roles of objects can be computed as the set of all methods and corresponding meta-objects. However, it can only be decided what roles an object can play, but not which one it currently plays, as it is usually not known which use-case is currently being executed. This information has to be provided by other sources such as acceptance tests that check a particular use-case.

3.3 Use-case Recovery

Adopting a new programming concept requires a certain shift in how development is carried out, even if the concept, such as the

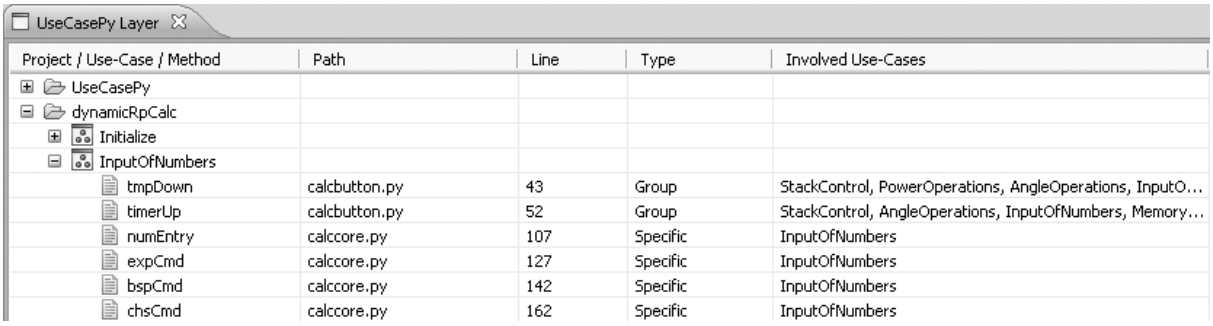

Figure 2. UseCasePy Scattering View Plug-in

`usecase` annotation, has a comparatively small impact on the shape of code produced. While new projects can adopt our approach without restriction, developers of legacy system have to introduce all use-case information to source code manually. This initial task consists of studying and analyzing an often only partially understood system to store the mappings of system functionalities to source code [25], which is likely to be nontrivial and time-consuming. It is hardly possible to determine solely from source code how objects contribute to the run-time behavior of the system, which is to a large degree due to language features such as late binding or polymorphism [8]. Moreover, dependencies between use-case implementations are not obvious, leading to missing or erroneous annotations [18]. Hence, concepts and tools are needed to assist developers in program comprehension and mapping of use-cases to source code.

We present a semi-automatic approach to identify use-case implementations in legacy systems by executing their corresponding acceptance tests. Figure 3 presents an overview of the entire process. Developers create use-case objects based on requirements descriptions and populate the use-case repository with them. They then link use-case meta-objects to corresponding acceptance tests by using the `usecase` annotation. Both steps are manual tasks but they need only be done once. An extended test runner automatically executes annotated tests observed by a tracer that collects all method calls and creates execution traces related to use-cases under test. It is determined at run-time with which use-case meta objects a given test has been annotated. After test execution, the collected traces are used to map source code entities to use-case objects in a so-called use-case method-map (adapted from feature analysis [17, 32])—use-case implementations are said to refer to a method when that method has been executed at least once in a specific use-case trace. In the last step, the mapping is used to introduce use-case annotations to the entire source code. Except for the first two steps, the entire process is done automatically and can be repeated as often as necessary.

The purpose of the semi-automatic annotation process is to provide developers with nearly complete use-case data in source code. Thus, developers can detect potential problems in the realization of their systems. For instance, each method without an annotation could be unused source code or an indication of missing tests. Also, a use-case method marked as specific residing within a module of many infrastructural methods could be considered a design disharmony. This comprehension of the application offers new possibilities in refactoring and structuring source code. In the remainder of this section, we will describe the three dynamic analysis steps in more detail.

Tracing of Use-cases Having annotated acceptance tests with related use-cases, developers execute these tests under observation of a dedicated tracing test runner. The tracer gathers all execution events of running tests and connects them to the related use-cases. Our approach is independent from a particular tracing technique since it only depends on execution data, regardless of how they are obtained.

Typically, a use-case is accompanied by several acceptance tests, each of which describes one possible path of execution, for example the flow of events to open a file, and all alternative paths such as those followed for error handling such as“access denied”. Instead of considering one trace as representing the entire use-case, we consider one trace as one particular *scenario*. Consequently, we have a *n-to-one* mapping between traces and use-cases.

In Figure 4, the view of participating objects shows the use-case denoted by the hatched ellipse. Executing the acceptance tests associated with that use-case leads to two traces corresponding to two scenarios being recorded, and the involved objects and method calls being identified. The two objects in the bottom-right corner are not found in one of the scenarios; hence, they and their methods will not be part of further analysis steps. Such mistakes can only be prevented by having exhaustive use-case specifications, as tracing will only recover use-case functionality in executed parts of the system. We would like to point out that a forgotten or omitted scenario or use-case is not a severe problem because dynamic analysis can be executed again later, with new tests with better coverage involved.

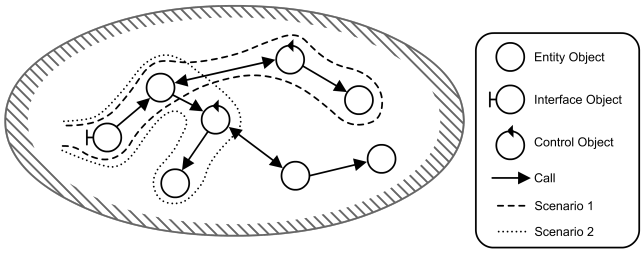

Figure 4. Use-cases, Scenarios, and Traces

Mapping Methods and Use-cases The next step is to connect use-cases, their scenarios, and methods. To that end, a so-called *use-case method-map* is created, consisting of columns denoting use-cases, and rows denoting methods. A cell is marked if a method was executed in at least one scenario or, more precisely, one trace. As explained above, there is a one-to-one mapping from traces to scenarios. Moreover, dependencies between use-cases and their scenarios are easy to obtain, as each trace corresponds to one scenario and use-case respectively.

Table 1 shows an example of a use-case method-map. The map presents data about three use-cases and four methods. Cells are marked with “X”. For instance, in the “draw a line” use-case, two methods were executed, namely `paintLine` and `paint`, albeit

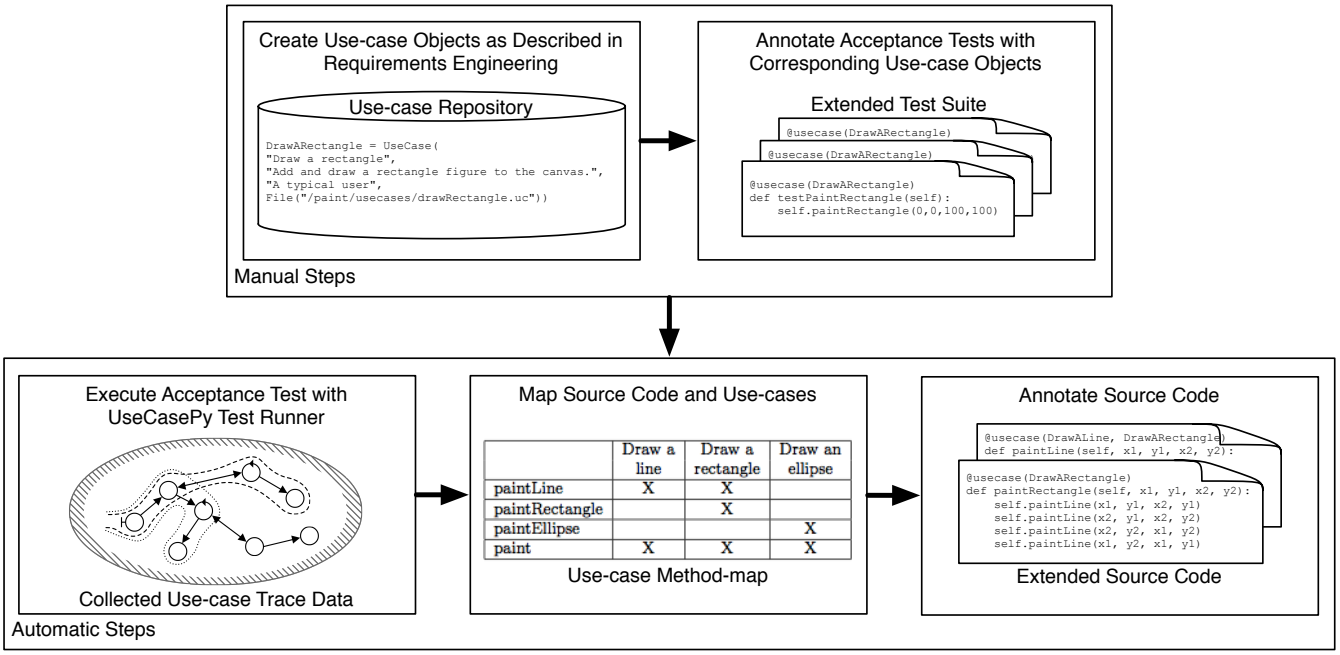

Figure 3. Use-case Recovery Process

possibly in different traces. Reading the table column-wise yields the scattering view, and reading it row-wise yields the tangling view with all use-case implementations the method in question contributes to.

	Draw a line	Draw a rectangle	Draw an ellipse
paintLine	X	X	
paintRectangle		X	
paintEllipse			X
paint	X	X	X

Table 1. Use-case Method-map Example

Annotate Source Code With the knowledge obtained from the mapping, the last step to annotate the source code is straightforward. The use-case method-map is visited row-by-row, and each encountered method is annotated with the corresponding use-case meta-objects as parameters. Optionally, developers can configure the algorithm to create infrastructural annotations with no parameterization if specific methods contribute to a majority of (or all) use-cases.

3.4 Discussion

The approach described above might face some problems in certain circumstances. In the remainder of this section, we describe these problems and how they can be addressed.

Lack of Tests and Use-case Descriptions Legacy software without accompanying test suites and use-case descriptions is problematic if the entities from which use-case annotations can be derived are missing. In case of missing tests, the use-case centric development approach can still be applied manually. If automatic link recovery is desired, tests have to be provided. In fact, it might turn out to be easier to write acceptance tests for an existing system, than to provide all links manually. Should use-case descriptions be missing, one can revert to using other requirements descriptions such as user stories. In this case, the approach can still be applied as before, apart from a change in terminology—entities originally representing use-cases would now represent user stories.

Conversion of Use-case Descriptions to Objects Frequently, use-cases are documented in a textual form that is not represented in source code. Minimally, a use-case object consists of a meaningful name and a link to the external document describing the use-case. Moreover, various properties can be attached to the use-case object, providing additional information.

It is not possible to provide a generic transformation strategy from textual use-case descriptions to use-case objects, as the diversity of possible input is too wide. However, manual creation of minimal use-case objects is straightforward and can be achieved quickly. Provided use-cases are described in a form governed by a structured template, it is easy to implement an automatic transformation. This transformation, however, is valid only for the given use-case template and document structure.

Manual vs. Automated Annotations Links can be defined either manually or automatically. Developers do not necessarily keep track of how particular annotations were created. It thus seems problematic to update links that were created automatically, as the automatic approach might override manually created annotations.

As we will describe in Section 4.2, the UseCasePy test runner maintains a database that keeps track of annotations created automatically. Annotations created manually are not stored but derived by intersecting the sets of all and automatically created annotations.

4. UseCasePy

We have implemented a prototype supporting use-case-centered development in Python. *UseCasePy* is a simple implementation based on several existing language features such as decorators or Python's unit test framework.

4.1 The @usecase Annotation

The implementation of the `usecase` annotation makes use of the concept of *method decorators* available in Python. A method deco-

rator is a function that is evaluated by the Python interpreter whenever it encounters a method definition augmented with the annotation corresponding to the decorator. The `usecase` decorator links use-case meta-objects to method meta-objects.

```
def usecase(*usecases):

  def decorator(method):
    if (type(method) in (classmethod, )):
      # Bound the method to a dummy class to retrieve
      # the original function object
      currentFunc = method.__get__(None, object).im_func
    elif (type(method) in (staticmethod, )):
      currentFunc = method.__get__(None, object)
    else:
      currentFunc = method
    # Add use-case knowledge to function objects
    currentFunc.__usecases__ = usecases
    return method

  return decorator
```

Listing 3. Implementation of the usecase Decorator

Listing 3 shows the implementation of the `usecase` function. The function accepts as parameter a list of use-case meta-objects, and returns an internal `decorator` function that is executed with the annotated method definition passed as parameter. Inside `decorator`, the type of the decorated method is identified and, if necessary, other decorators are unwrapped to get hold of the original function object. This object is extended with a new attribute called `__usecases__` that holds all use-case meta-objects with which the corresponding method is annotated. The method behavior is not altered.

The run-time overhead of this approach is very small as decorators are only executed when the interpreter encounters a method definition for the first time. Memory consumption increases only slightly because each use-case meta-object is unique within the system. Last but not least, we have implemented a small meta-level API for inspecting method annotations and objects to obtain related use-cases and roles.

4.2 Use-case Recovery Test Runner

Existing software systems providing an acceptance test suite for use-cases can be comfortably adapted to use-case-centered development by analyzing their run-time behavior. Figure 5 illustrates an overview of our use-case recovery system. At the bottom of the figure, the software system under consideration can be seen—besides the original source code of the system, there are use-case-related acceptance tests and the global use-case repository.

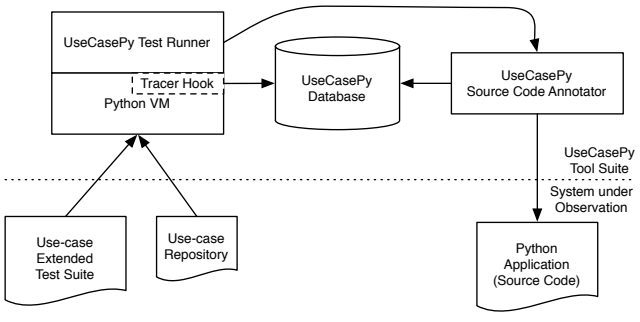

Figure 5. UseCasePy Test Runner Architecture

The upper left corner depicts the test runner and tracing mechanism. The UseCasePy test runner works like a normal test runner, executing test cases and presenting results to the user. Moreover, each time a test case exhibits a use-case annotation, our tracer hook is started and all use-case information, related scenarios, and run-time events are automatically queued, stored, and linked in the database. In case of failed tests, the trace is discarded. Having collected run-time data, the source code annotator creates the use-case method-map and subsequently annotates the entire source code with use-cases.

Tracing in Python The Python interpreter offers a hook function with which a trace function can be registered. The trace function is called each time a line of code is executed with arguments such as the current execution frame. We have implemented an own trace function that records executed methods per use-case. It is registered in the interpreter for multi-threaded applications—tracing over process boundaries is currently not supported.

Inside our trace function, we check whether the trace mechanism was activated by the test runner and if the upcoming event is a call. If so, the method currently being executed is extracted from the frame object, and we make sure that the observed source code file is included in the analyzed project. This is to include only application code in the use-case analysis; code from third-party libraries is not to be considered. After this check, the following four values are written to the database: the execution event as forwarded from the interpreter, the method name, file path, and line number of the currently executed source code.

Database Model Each trace is stored in a dedicated table with all collected events. Each use-case is represented by an entry in the use-case table; the template attributes are stored as columns. Subsequently, both tables are linked together via a third table describing scenarios with additional attributes such as their time stamps and test results.

A relational database is capable of handling large amounts of data very well, and of offering a query language for easy data access. For instance, the SQL `DISTINCT` keyword can be used to easily create a unique set of involved methods, and `UNION` can be used to connect several traces to one entire use-case. Based on these query results, creating the use-case method-map as well as annotating the source code is straightforward.

5. Case Study

We apply our use-case-centered development to *Django* (Version 1.0.2), a Python Web framework [11], and demonstrate its benefits in typical maintenance activities.With the help of our UseCasePy implementation, we present how use-cases can be recovered with less effort and how our programming constructs support developers in program comprehension, debugging, and reengineering.

5.1 Experimental Setup

Django is a typical model-view-based Web framework and consists of features such as an object-relational mapper, an administrative interface, a template engine, and a cache system. This mid-sized project has more than 75,000 lines of production and 35,000 lines of test code. The framework without tests is organized in about 4,500 methods and 1,200 classes structured in 120 packages. 615 unit tests ensure the functionality of methods, modules, and use-cases. Django offers an adequate number of test cases and is big enough to be used to validate our approach in a satisfying manner.

The Django framework documentation does not contain any use-case descriptions. Therefore, we first identified 37 use-cases from typical usage scenarios, including login to a Web page, upload a file, and reset a password. We found these scenarios in additional documentation such as Web sites and tutorials.

5.2 Use-case Recovery in Django

Having identified the use-cases, we applied our use-case recovery process. Since no internal knowledge about the system was avail-

able, about three hours were required to investigate use-cases and create their descriptions in the use-case repository. Afterwards, we located acceptance tests and annotated them with corresponding use-cases.

We annotated 414 out of 615 tests with use-cases. Besides a small number of group annotations, almost all acceptance tests were assigned to one specific use-case meta-object. All in all, a fully annotated test suite was obtained within five hours. We expect that an experienced Django developer would have taken much less time to complete this task.

The next process step is the automatic test execution with the use-case recovery test runner. The ensuing automatic test execution was performed on a MacBook with a 2.4 GHz Intel Core 2 Duo and 4 GB RAM running Mac OS X 10.5.6. The used Python version was 2.6; the test suite was launched from the PyDev Eclipse IDE [27].

Python's standard test runner executes the entire test suite in 220 seconds. In contrast our tracing test runner is about 6.5 times slower and requires 1,446 seconds. For this reason, we suggest to use the tracing test runner in scenarios where test performance is not of importance, e. g., during nightly builds. We would also like to note that running the use-case recovery process on an already annotated system will further slightly decrease performance because former automatic-created annotations need to be removed. Tracing tends to generate a large amount of data. Restricting the traced events to only method calls resulted in a relational database 139 MB in size (without this restriction, about 1 GB was collected). On average, our tracing test runner generates about 7 MB of data per minute.

Executing all use-case-annotated tests cover almost 33 % of the system's source code, resulting in the insight that 1,491 out of 4,573 methods (without tests) are part of one or more use-cases. This value seems lower than expected, but it has to be noted that the acceptance test suite is probably not complete.

5.3 Software Maintenance with Use-cases

In software maintenance, developers spend most of their time for program comprehension [31] which is in part due to missing traceability between requirements and source code. Use-case-centered development recovers this traceability and introduces use-cases as first class entities so that developers can benefit from this information to support their system understanding. In this case study, we first present the classification of the Django system from the user's point of view and then describe how our use-case entities support maintenance activities such as debugging and reengineering.

Program Comprehension with Use-cases To assess the possibilities of our use-case annotations for program comprehension in general, we are first interested in understanding the distribution of use-case implementations over the entire system. Figure 6 shows the relationship between all annotated methods and their involved use-case implementations. There are more than 500 methods that contribute to exactly one use-case. In other words, a third of all annotated methods implement specific use-case behavior. It is very likely for developers to concentrate on these particular methods to understand the implementation of a specific use-case. Should they not suffice, methods annotated with 2–5 use-cases can be examined. Together, these and the specific methods represent two thirds of the annotated code. This is very satisfying as it attributes large system parts to specific use-cases. From the observation that two thirds of methods in the Django source code are closely related to few use-cases we conclude that a very large part of the system exhibits a high cohesion. This in turn means that Django is well modularized in terms of cohesion.

The last third of all methods is infrastructural meaning library code that will be used by most of the use-cases. We argue that such code should be used during program comprehension to get a better understanding of the whole system behavior. We consider it is less important for particular use-cases.

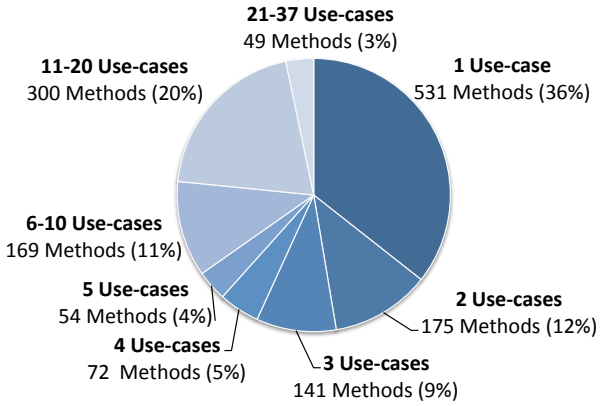

Figure 6. Distribution of Use-case Annotations within Django

Use-case-driven Fault Localization Debugging activities, particularly searching for failure causes, requires deep knowledge of the system and its behavior. Developers have to follow the infection chain backwards from the observable failure to the past defect, which is largely a search problem in time and space of programs [35]. Usually, failure reports as described by users, which are arguably closer to use-cases than to implementation artifacts, must be laboriously mapped to system entities if proper traceability is missing. With our recovered use-case implementations, we support developers in narrowing down failure causes to use-case-related locations that are likely to include defects.

Using data from use-case recovery, developers can reduce the overhead for searching relevant methods to a minimum. As an example, a user reports a problem while uploading image files. Having this failure report, developers can easily map this description to the "upload a file" use-case and reduce the search space to packages that include use-case-related annotations. In doing so, developers can narrow their further search to 21 out of 121 Django packages and 241 out of 4,573 methods. In other words, roughly 17.4 % of packages and 5.3 % of methods are interesting for detecting the failure cause.

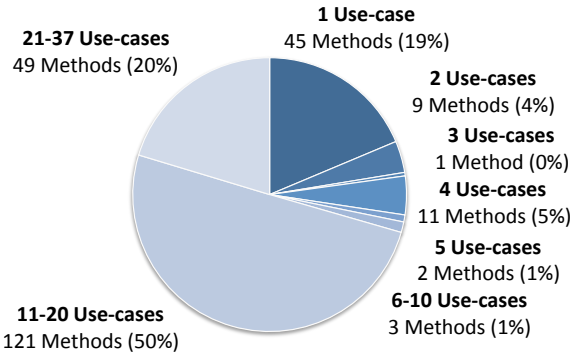

Figure 7. "Upload a File"-related Annotations

Figure 7 illustrates all annotations "upload a file" is part of. In contrast to Figure 6, it is apparent that only one third of methods

contributing to this use-case exhibit high cohesion[1]. Given the expectation that the core implementation of a use-case should be contained in just a few methods, it is satisfying to see that this seems to be the case for the "upload a file" use-case. In fact, the highly cohesive third in Figure 7 corresponds to about 1.75 % of the entire system.

If the failure is restricted to this specific use-case, developers can further reduce the search space to the most relevant packages and methods. Figure 8 lists the importance of all relevant use-case annotations within the 21 Django packages[2]. A package with only a few specific use-case method annotations indicates that its importance will be higher as compared to a package with many infrastructural methods. The `./core/file` package seems to be most relevant with a value of 30 %, followed by `./http` with 17.5 %. Developers can focus on a few packages and the methods within to understand the "upload a file" use-case. By starting with the "file" package, developers limit the search to about 30 mostly specific method annotations. Thus, only 30 out of 4,573 methods—0.65 % of the entire system—are of high relevance to the "upload a file" use-case and its including failure. Additionally, if the failure-inducing method is also part of other use-cases, developers can prioritize the necessary acceptance tests to ensure that the system still works as expected.

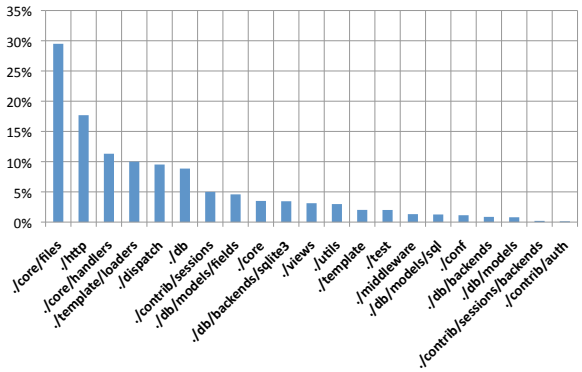

Figure 8. Importance of Packages for File Uploading

Use-cases for Reengineering For reengineering and refactoring specific system parts, developers require not only an in-depth understanding of source code; the dependencies to affected requirements are also of great value. First, involved use-cases reveal what functionality is primarily influenced by changes allowing developers to prioritize corresponding acceptance tests. Second, the traceability of use-cases supports the analysis of interfaces and most commonly used entry points. Third, requirements also allow developers to characterize a system part with respect to coupling and cohesion. As an example for reengineering, we choose Django's "authentication" package because it is a very important part of the Web framework and it will often be adapted if new authentication techniques arise.

In summary, the package consists of more than 100 methods where every second method is annotated and about 33 % of all

[1] We apply the term "cohesion" to single methods instead of larger-scale modules. A method exhibits high cohesion if its code contributes to few use-cases.

[2] We compute the importance of a package by first summing up weighted methods, where specific methods (one use-case) are assigned a weight of 8 and infrastructural methods (more than 21 use-cases) a weight of 1 with decreasing weights for other groups as shown in Figure 7 and then normalizing this value with the number of all methods multiplied by 8.

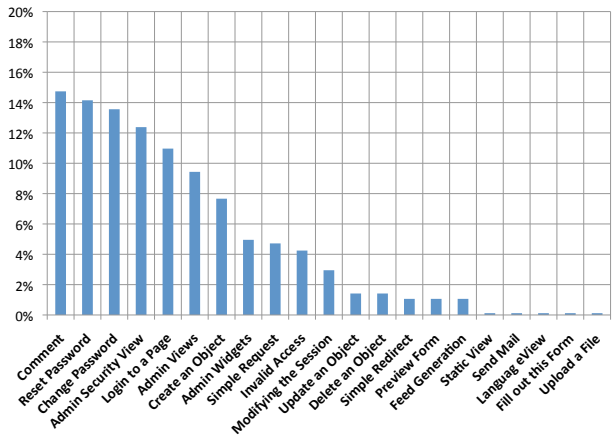

Figure 9. Importance of Use-cases within the Authentication Package

methods are related to just a few use-cases. Thus, most of the traced behavior in this package will only be used in a specific context. There are five methods referring to a high number of use-cases. These methods implement the entry points to this package; for instance, in `middleware.py`, we found the infrastructural method `process_request`. Developers are able to consider such methods first and afterwards follow a certain use-case through the package. We argue that this is a simple way to learn more about the internals of such a package and the details of a specific use-case.

With the help of use-case recovery, we conclude that Django exhibits, at least for this package and its clients, low coupling. The five methods constituting entry points into the "authentication" package are each responsible for a high number of use-cases. Had there been more methods with less use-case contributions, the overall interface of the package to the rest of the system would have been broader, suggesting higher coupling.

Although 21 out of 37 use-cases are involved in the behavior of the authentication package, not every use-case is equally influenced by this package. Figure 9 presents the importance of all observed use-cases, which was computed similar to package importance except that we sum up methods per use-case and normalize with all package methods. Each value is below 15 %; thus, all use-cases consume just a few methods of the entire functionality. The package is used by many use-cases and can be identified as a helper package. The ten least important use-cases do not use more than one or two methods. Furthermore, it can also be seen that the five most important use-cases require quite a few authentication mechanisms—for instance, the "comment" use-case requires authenticated users for adding new comments to a Web page.

6. Related Work

We divide related work into three categories: requirements traceability, feature localization, and use-cases as aspects.

Requirements Traceability Requirements traceability refers to the ability to follow the life of a requirement, both forwards and backwards [15].

Based on the event-notifier pattern, event-based traceability [7] establishes loosely coupled traceability links between requirements and other artifacts. To support software evolution, an event server notifies all subscribed entities if a requirement changes. This approach can also be used with use-cases and source code entities. After creating manual links, the traceability information can be obtained by the event server at development and run-time.

The TraceAnalyzer [12] is similar to our approach since it uses run-time information of scenarios to recover traceability links. The suggested process consists of hypothesizing (collecting scenarios), atomizing (building the footprint graph), generalizing and refining (propagating trace data within the graph). TraceAnalyzer considers only classes and does not reflect the gathered information in source code directly.

Based on a few links created manually, LeanArt [16] recovers traceability links between use-case diagrams and types and variables. It uses run-time monitoring, program analysis, and machine learning to propagate the small set of initial links within the source code. LeanArt mimics the human-driven procedure of searching for similarities between program entities and use-case elements. The approach does not consider the behavior of scenarios and the knowledge is also not used for explicitly representing use-cases in source code.

There are also some approaches based on information retrieval [2, 26]. These techniques have the benefit to process fully automatic but they have even the drawback to create many false-positive mappings. We prefer a high precision because developers can trust the traceability links even if we have not found each relationship.

Feature Localization Feature engineering [30] introduces features as the natural organization of a system's functionality from the users' point of view. It promotes features as first-class objects throughout the software life cycle to reduce the gap between developer and user perspectives in development. However, the term "feature" is not defined precisely enough and not as important as use-cases to guide developers in system implementation and maintenance. Nevertheless, the similarity between both concepts might suggest interesting ideas for future work.

Feature localization, also known as feature identification, is a technique to single out subsets of a program when exercising a feature. The Software Reconnaissance technique [32, 33] was the first to use dynamic analysis to locate features. This technique splits test cases into two sets, one of which is part of the searched-for feature. The other contains the remaining test cases. From the gathered traces, a mapping from system behavior to involved source code entities is derived. In the last step, the gathered results are combined with their relationships to the involved features: calls contained in the first set belong to the corresponding feature; calls contained in the first set only are *unique* to that feature; and calls only contained in the second set are not covered by the feature in questions. Further research has refined this concept by defining metrics [23, 34] and heuristics [14] or interactively analyzing features at the sub-method level [10] to quantify features and source code more precisely.

Another approach extends the idea with static analysis of dependency graphs and formal concept analysis [13]. Several relationships between features and source code artifacts are computed with the help of formal concept analysis guiding the static analysis technique to identify feature-specific source code units. Similar concepts [6] focus on test coverage [3] and recovering architectural entities [5]. A further approach uses static and dynamic concepts with several tools such as parser and processor emulation [1] solving difficulties of collecting dynamic data and combining features and micro architecture together.

Besides dynamic analysis, there are alternative approaches based on information retrieval techniques [24, 36]. For instance, latent semantic indexing will be used to locate similar source code identifiers in different documents [24]. We think that the combination with static analysis can prevent mistakes from the as-needed strategy of dynamic analysis and should attract attention in future work.

The usefulness of considering features as the missing link between the users' point of view and the implementation level was evaluated in several publications explaining the correlation between features and code, characterizing the functional role of classes and methods, and revealing how developers develop features [17].

Summarizing, all approaches primarily differ in measurements to quantify the relevance of code to a feature. However, feature analysis often focus on one feature at a time ignoring relations to other features or specified requirements.

Feature-oriented programming (FOP) [4] and context-oriented programming (COP) [19] are paradigms that capture increments in program functionality. FOP and COP lead to a layer-based software design where a layer corresponds to a feature. Each layer adds functionality to previously composed layers and different combinations of layers produce different programs. The focus of FOP is on compile-time selection and that of COP on dynamic composition. Both do not support layer/use-case objects at run-time.

Aspects Use-case modularity [21] argues that aspect-oriented programming can provide the missing link between use-cases and implementation. It proposes a separate layer across existing components supporting a decomposition based on use-cases. The approach comprises the separation and composition of use-case extensions which might be implemented with the help of aspect-oriented programming (AOP) [22]. However, it does not solve the problem of tangling use-cases and their separation in source code.

AspectU [28] introduces an aspect entity at the use-case requirement level similar to AspectJ at implementation level. A use-case is described as a tree structure of steps and extensions (join points) and the execution involves several nodes and paths in this tree. An AspectU pointcut matches and identifies a set of certain join points (subtrees) that are extended by additional behavior from advice constructs. AspectSD is introduced as intermediate language at the sequence diagram level transforming AspectU to AspectJ source code entities. Developers have to define links between use-cases, sequence diagrams, and source code. Their approach depends on extended use-case specifications so that it is hardly applicable for legacy systems.

7. Conclusion

With *use-case-centered development* we bridge the gap between how users describe their domains of interest and how developers approach the realization of software systems in their support. By representing use-cases as *first-class elements* throughout all software development activities, we are able to recover use-case traceability lost in more code-centric development activities.

Use-case-centered development extends use-case traceability from analysis and design all the way to implementation, debugging, and testing. Making use-cases first-class elements in the entire development tool-chain allows for direct and convenient access to code or even live objects *from requirements and vice versa*. This reuse of requirements documents can lead to a significant decrease in time, effort, and costs of software development [15].

With the new `usecase` language construct, as demonstrated in the application of our UseCasePy Python extension to the Django Web framework, developers can mark objects and methods as contributing to one or more use-cases. With that, dependencies between use-cases themselves and use-cases and their implementation are made explicit.

By representing use-cases as meta-objects at run-time, a number of development activities can be enhanced and tools improved. For example, debuggers can introspect objects and link them to the use-cases they participate in or quality assurance can check code coverage on a per-use-case basis.

Recovering use-cases from usage scenarios helps to restore the original use-cases the system was built to support and with that improve comprehension of an existing code base.

8. Acknowledgments

We would like to thank David A. Thomas, Martin von Löwis, Malte Appeltauer, Bastian Steinert, Jens Lincke, and Richard P. Gabriel for fruitful discussions and valuable contributions.

References

[1] G. Antoniol and Y.-G. Gueheneuc. Feature Identification: A Novel Approach and a Case Study. *ICSM*, pages 357–366, 2005.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering Traceability Links between Code and Documentation. *TSE*, 28(10):970–983, 2002.

[3] T. Ball. The Concept of Dynamic Analysis. In *ESEC/FSE*, pages 216–234, 1999.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. In *ICSE*, pages 187–197, 2003.

[5] D. Bojic and D. Velasevic. A Use-Case Driven Method of Architecture Recovery for Program Understanding and Reuse Reengineering. In *CSMR*, pages 23–31, 2000.

[6] D. Bojic, T. Eisenbarth, R. Koschke, D. Simon, and D. Velasevic. Addendum to Locating Features in Source Code. *TSE*, 30(2):140, 2004.

[7] J. Cleland-Huang. *Robust Requirements Traceability for Handling Evolutionary and Speculative Change*. PhD thesis, University of Illinois, 2002.

[8] T. A. Corbi. Program Understanding: Challenge for the 1990s. *IBM Syst. J.*, 28(2):294–306, 1989.

[9] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[10] M. Denker, J. Ressia, O. Greevy, and O. Nierstrasz. Modeling Features at Runtime. In *MODELS*, pages 138–152, 2010.

[11] Django. The Web Framework for Perfectionists with Deadlines (06/09/2011). http://www.djangoproject.com, 2011. URL `http://www.djangoproject.com`.

[12] A. Egyed. A Scenario-Driven Approach to Traceability. In *ICSE*, pages 123–132, 2001.

[13] T. Eisenbarth, R. Koschke, and D. Simon. Locating Features in Source Code. *TSE*, 29(3):210–224, 2003.

[14] A. D. Eisenberg and K. D. Volder. Dynamic Feature Traces: Finding Features in Unfamiliar Code. *ICSM*, pages 337–346, 2004.

[15] O. C. Z. Gotel and A. C. W. Finkelstein. An Analysis of the Requirements Traceability Problem. In *RE*, pages 94–101, 1994.

[16] M. Grechanik, K. S. McKinley, and D. E. Perry. Recovering and Using Use-Case-Diagram-To-Source-Code Traceability Links. In *ESEC-FSE*, pages 95–104, 2007.

[17] O. Greevy. *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, University of Berne, 2007.

[18] O. Greevy and S. Ducasse. Correlating Features and Code Using a Compact Two-Sided Trace Analysis Approach. In *CSMR*, pages 314–323, 2005.

[19] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *JOT*, 7(3):125–151, 2008.

[20] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[21] I. Jacobson. Use Cases and Aspects – Working Seamlessly Together. *JOT*, 2(4):7–28, 2003.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented Programming. *ECOOP*, pages 220–242, 1997.

[23] J. Kothari, T. Denton, S. Mancoridis, and A. Shokoufandeh. On Computing the Canonical Features of Software Systems. In *WCRE*, pages 93–102, 2006.

[24] A. Kuhn, S. Ducasse, and T. Gırba. Enriching Reverse Engineering with Semantic Clustering. In *WCRE*, pages 133–142, 2005.

[25] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *ICSE*, pages 492–501, 2006.

[26] A. Marcus and J. I. Maletic. Recovering Documentation-To-Source-Code Traceability Links Using Latent Semantic Indexing. In *ICSE*, pages 125–135, 2003.

[27] PyDev. Python Development Environment (06/08/2011). http://www.pydev.org, 2011. URL `http://www.pydev.org/`.

[28] J. Sillito, C. Dutchyn, A. D. Eisenberg, and K. D. Volder. Use Case Level Pointcuts. *ECOOP*, pages 244–266, 2004.

[29] C. O. Systems. *Objectory*. Wiley-QED Publishing, 1994.

[30] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. A Conceptual Basis for Feature Engineering. *JSS*, 49(1):3–15, 1999.

[31] A. von Mayrhauser and A. M. Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28:44–55, 1995.

[32] N. Wilde and M. C. Scully. Software Reconnaissance: Mapping Program Features to Code. *J. Softw. Maint.-Res. Pr.*, 7(1):49–62, 1995.

[33] N. Wilde, J. Gomez, T. Gust, and D. Strasburg. Locating User Functionality in Old Code. In *ICSM*, pages 200–205, 1992.

[34] E. W. Wong, S. S. Gokhale, and J. R. Horgan. Quantifying the Closeness between Program Components and Features. *JSS*, 54(2): 87–98, 2000.

[35] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2006.

[36] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a Static Noninteractive Approach to Feature Location. *ACM Trans. Softw. Eng. Methodol.*, 15(2):195–226, 2006.