Context-Oriented Programming: Beyond Layers*

Martin von Löwis[1] Marcus Denker[2] Oscar Nierstrasz[2]

[1]Operating Systems and Middleware Group
Hasso-Plattner-Institute at
University of Potsdam, Germany
http://www.dcl.hpi.uni-potsdam.de

[2]Software Composition Group
University of Bern, Switzerland
http://scg.iam.unibe.ch

Abstract. While many software systems today have to be aware of the context in which they are executing, there is still little support for structuring a program with respect to context. A first step towards better context-orientation was the introduction of method layers. This paper proposes two additional language concepts, namely the implicit activation of method layers, and the introduction of dynamic variables.

1 Introduction

Context-oriented programming provides mechanisms in the programming language to make the dependency of the program on its context explicit. Costanza and Hirschfeld [1] have previously proposed *method layers* to explicitly represent context dependency in a program. A new module concept, the layer, helps programmers to refactor a program to separate context-dependent behavior from the main program logic. There are several implementations of method layers, including the original ContextL implementation [1].

While method layers enable the modularization of the program with respect to context-awareness, the mechanism by which method layers are activated still poses a challenge. In ContextL, method layers need to be explicitly activated, for example, after evaluating some condition on sensor readings from the system's environment. While such explicit context activation can work in many cases, there are also cases where no single point in the progam exists that could detect the change in context, and consequently activate a method layer. In particular, if the context may change at any time during program execution, implicit activation of method layers becomes necessary.

In addition, method layers focus on context-dependent algorithms. In many cases, algorithms might not change as the context changes; what will change is the data on which the algorithms operate. For example, in client-server systems, each new request will might establish a new context for the algorithm, including information such as the user performing the request, correlation to earlier actions of the user (*i.e.*, sessions), transactional context of the operation, and so on. A method that needs to consider such context then must find out what the

* Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007), ACM Digital Library, 2007, pp. 143-156. DOI 10.1145/1352678.1352688

current context is. As methods call each other in a recursive manner, the context information must be passed from the point where the context is observed to the point where the context is consumed; many contemporary programming languages still don't address this need explicitly.

Two examples of contextual behavior are the dependency on the current user of a system, and the dependency on the current device on which information is displayed.

For context-dependent output, it is common that different output algorithms are used, *e.g.*, output to a PostScript file typically requires algorithms different from output to an HTML page. In an object-oriented program, output logic is commonly spread over multiple methods. To denote the various output modes, different method layers can be defined. When rendering the output, the proper layer can be activated, and all output methods will adapt themselves according to the output context.

Dependency on the current user has various facets: it may be that users have different roles, and the program behavior should again change significantly depending on the role in which a user acts. This dependency can be modeled using method layers again, defining different layers for the different roles. However, dependency on the current user also often involves operating on different data: in an email system, different users operate on different mailboxes. The program logic for accessing a mailbox will be the same independent of the user; the only difference is in which mailbox is presented to the user. A common approach to implement this dependency is to use the user's identification as a key into some associative data structure. In order to implement that lookup, the current user needs to be known in all places of the program.

In some cases, a combination of both contextual state and contextual behavior is necessary. For example, a web client application may need to provide a User-Agent header when requesting certain pages from a web server. The User-Agent header is a text string indicating the web browser which sent the request. Web servers sometimes respond with different pages, depending on what browser made the request. If the access is made in an automated manner, the web client library may either send no User-Agent header at all, or send one that indicates an automated agent. With context-oriented programming, the web client application can set the context for all HTTP requests to include a specific User-Agent header. This involves both a behavior change, and relies on contextual data: the header should be sent only if it was configured, and if so, the string that is sent is determined by contextual data.

In this paper, we discuss extensions to context-oriented programming that we developed as part of PyContext, a framework for context-oriented programming in Python. We studied a number of existing Python application to analyze what kind of context-dependency is present in these systems, and then defined a set of Python constructs that can help to better structure programs with respect to context-dependency.

Our main contributions are twofold: we present an extension to the concept of method layers to support implicit activation of layers, and we propose dynamic variables as a means to access context-dependent state.

In section 2, we present our analysis of existing software systems. Then we describe the notion of method layers (section 3), our extensions to that (section 4), and the PyContext implementation (section 5). Finally, we present some related work (section 6) before concluding (section 7).

2 Context-dependent behavior

We have analyzed a number of non-trivial Python applications with respect to dependency on context. This study both validates our assertion that context-dependency is common in current software systems, and helps us to identify concepts that can be added to programming languages to better support context-orientation.

We selected three applications that we considered to be naturally context-aware. The applications we studied were Django [2], roundup [3], and SCons [4]. Whereas the first two are both used in the domain of web applications, the third is from the area of software engineering and maintenance.

Django is a framework for web applications. It includes

- an object-relational mapper, to support persistent storage of application objects
- an HTML templating engine, to allow separation of the web page design from the application design, and
- a framework for web applications, including an HTTP server and a component system for web applications

Roundup is a bug-tracking application, allowing users to report issues found with a software system over the web or via email, and developers to collect information about these issues in order to eventually resolve them.

SCons is a software construction tool that can be used in the build process of software systems.

We identified context-awareness in these applications first through systematic inspection. As a starting point, we looked for known cases of context-awareness, such as dependency on system configuration and dependency on the current user of the system. Using source code review, we identified a number of places where context-dependency occurs. In the next step, we tried to generalize these findings and derive more systematic ways of identifying context-awareness. We present these findings starting with the specific examples, then going on to the generalizations.

2.1 Case studies

Django. Django supports a component system for web applications, where an individual web application can be deployed into an existing Django installation,

and transparently respond to HTTP requests targeted at that application. For this to work, Django uses two pieces of contextual information:

1. The URL of the HTTP request consists of contextual information provided by the client-side web browser; a part of the URL is meant to identify the application that should serve the request.
2. As neither the Django code nor the application code should incorporate a static mapping between URLs and applications, Django supports a URL configuration file (urls.py) that defines a mapping from URLs to application. During run-time of the web server, the server should respond to changes of the configuration file, adding new web applications as they are deployed.

To deal with the dependency on configuration data, Django internally maintains a global variable (conf.settings) that contains all configuration information. To react to dynamic changes of the configuration files (such as the url mappings), a separate thread reads the modification times of the configuration files every second, and restarts the web server from scratch if a change in the configuration files has been found.

To gain access to the URL path of the HTTP request, both an HTTP request object and a path object are passed through several layers of software until a URLResolver object maps the URL path to a callback function that is the entry point to the web application.

The HTTP request object is further passed as an explicit parameter to various routines, and provides additional contextual information. In particular, the request is passed to several layers of software called middleware. One such middleware module provides session context, based on information in the request.

Roundup. In roundup, several instances of the bug tracker may be running on a single machine. Each tracker instance acts as a context of execution for the tracking software, providing its own relational schema for trackers, its own database of recorded issues, its own set of access control lists, and so on.

Instead of recording the "current" instance in a global variable, roundup creates, per HTTP request to the tracker, a Client object which encapsulates:

- the current tracker,
- the current HTTP request,
- the id of the end-user invoking the current request (which may be obtained from the request, or through other authentication mechanisms), and
- the issue or issue detail on which an operation is to be performed.

Actions are then processed using the command pattern [5], where the command object is created with a reference to the Client object, and then the command is run.

SCons. SCons maintains an "environment" object, which contains information about the context of the current build activity. This environment is filled with information from various sources, such as:

- the host system and system type on which the build is run,
- locations of tools needed in various build steps,
- configuration information from the build scripts about parameters to be passed to the build steps,
- files that act as input and output to the build steps, and
- actions that still need to be run, or have already been completed.

This environment is passed as an explicit last parameter to all methods.

2.2 Detecting context-dependency

In studying context-dependency in these three applications, we noticed two kinds of code structure which hint towards context-dependent behavior:

1. In many cases, things are explicitly called "context" — authors of the software were clearly aware of the contextual nature of these aspects of the system.
2. In some cases, parameters occurred primarily as "pass-through" parameters, *i.e.*, they propagate recursively through a chain of method calls, until they are eventually consumed by some leaf function. As these parameters sometimes make it into interface definitions, the caller needs to pass the parameter independent of whether the callee actually needs it. This, in turn, may cause the caller to require the parameter as an input argument as well, even though it has no need for the parameter.

These two structures now allow for a more systematic search for contextual code. Unfortunately, detecting pass-through parameters in an automated manner requires tools that analyze the source code statically, inspecting each function's parameters, and usage of the parameters within the function. As no such tools are available today, we were only able to detect a few more cases of contextual information with manual inspection, by looking for functions that don't use parameters they receive.

Searching for things called "context" is much easier in a systematic manner than searching for pass-through parameters; we found these additional cases in the systems studied:

1. In Django, there is a module called `context_processors`. These are functions that return dictionaries of context information, such as the current user, the permissions of the current user, whether or not debugging information should be displayed, what (natural) language a web page should be displayed in, and so on.
2. The Django templating engine maintains a set of variables which can be accessed in rendering the page template, for place holders, conditional HTML inclusion, and repeated blocks of HTML. The collection of these variables is called "context".
3. The same holds for the roundup templating engine, which also calls all variables used for templating collectively "context".

3 Context-Oriented Programming

In [1], the authors propose the following language constructs to support context-oriented programming, for an implementation called ContextL:

- layers, which identify groups of classes and methods that will be used together in the dynamic scope of a program,
- layered classes, which are classes that have different partial class definitions for different layers,
- layered methods, which are defined through partial definitions for different layers,
- layered slots, which are instance attributes whose values depend on the active layer, and
- explicit layer activation, which selects a certain set of partial definitions in the dynamic scope of control flow.

With layers, it becomes possible to factor out those parts of method and class definitions that have been written for a specific context. For the example of different output methods given in the example, a method layer can be defined for each different mode of data output. A method that performs the actual output might then get multiple partial definitions, one for each layer. Some parts of the algorithm might be independent from the output algorithm, *e.g.*, the algorithm that iterates over all pieces of information and call the output method on each piece. If a certain kind of output is requested, the program needs to activate the corresponding layer, and call the output algorithm in that context.

As layers can be defined for independent contexts, multiple independent layers can be activated simultaneously, causing the partial definitions for all of these layers to become active. In the output example, a separate layer might be developed for a context in which special support for handicapped people is necessary (*e.g.*, by enlarging all text to improve readability). In such a case, a single layered method might need to take multiple partial definitions into account. ContextL defines a mechanism for combining partial methods, where each layer activation can modify the previous definition of a method with another fragment. In the definition of the partial method, the developer needs to specify whether the fragment is executed before, after, or instead of the original method. In the latter case, the developer can also choose to call the original method inside the fragment, making the fragment run around the original method.

4 Context beyond layers

We propose two further mechanisms to support context-oriented programming as defined above: implicit layer activation, and dynamic variables.

A shortcoming of the existing implementations of method layers is that layers must be activated explicitly rather than implicitly as the context of the application changes. While it is possible and desirable in many cases to explicitly control activation of layers (typically after evaluating some condition on the program

context), having to explicitly activate layers may sometimes violate modularity of contextual behavior definitions: If the condition that should trigger the activation can become true at any time in the program (and if it is necessary that the program react on the context change quickly), the check for the change of the condition needs to be added in many places of the code, potentially to the degree that these replicated changes become larger than the actual contextual behavior.

In our case studies, we observed that much context-dependency in applications is not in contextual behavior, but in contextual state. It was often the case that the program computed some variables from contextual conditions, and then indirectly called methods that needed to work with these variables.

In the case of the web applications (see section 2), the applications typically combined all contextual data associated with the current request object, which then is passed as an explicit parameter to all methods. If some interface does not provide for passing the request, the methods implementing it have no way of determining the context.

For configuration data, global variables were commonly used. While global variables work fine in single-threaded programs or programs where threads only read the values, they become a maintenance challenge in multi-threaded programs. For example, debugging and tracing support might be active only during a part of the program (the part which the developer currently studies), in these cases, the applications typically added additional parameters to the methods in order to make the context available.

5 PyContext

PyContext is a framework for context-oriented programming similar to ContextL [1], extending it with concepts that we determined to be desirable. PyContext also builds on the support for context-orientation that is already present in Python 2.5 [6].

We present PyContext in three stages. First, we review a recent addition to the Python programming language (the **with** statement) that allows one to scope contextual behavior to the dynamic extent of a block of code. We found that mechanism to be a useful basis for supporting cases of context-orientation where the program detects the context at some point, and then requires all subsequent actions to take that context into account.

Then, we discuss how we make the concepts found in ContextL available to Python programs. Specifically, we added support for the definition of method layers, for the definition of layered classes, and for the (nested) activation of layers.

Finally, we introduce the extensions we made in PyContext which aren't available in ContextL, namely the implicit activation of layers and the definition of dynamic variables.

5.1 Context managers in Python 2.5

In the Python Enhancement Proposal 343 [7], a new keyword **with** is introduced into the language; this is related to a kind of object protocol called "context managers". While this statement was introduced independently of this work, we found it to be extremely useful to represent context-dependency in a Python program, avoiding the need to come up with our own extension to the Python syntax.

This statement can be either used in the form:

```
with expression:
    statements
```

or as:

```
with expression as variable:
    statements
```

In either form, the expression is evaluated, and should result in a object that implements the "context manager" interface [7]. According to that interface, a method __enter__ is called on the context manager, and the statements are executed. Finally, __exit__ is called on the context manager. The language guarantees that __exit__ is called regardless of how the execution of the statements terminates (*i.e.*, whether they run to the end, are terminated through a return, continue, or break statement, or whether an exception is raised). The method __exit__ receives as a parameter information indicating whether the block was left normally, or by means of an exception.

In the second form, the result of the __enter__ method is assigned to the variable.

The PEP mentions the following use cases for this kind of statement:

- Synchronization and locking of blocks of code: the context manager should be an object supporting mutual exclusion; __enter__ should acquire a lock; __exit__ should release it. Python 2.5 provides such a context manager in its threading library, allowing users to write

  ```
  some_lock = threading.RLock() # create recursive lock
  # ...
  with locking(some_lock):
      some_code
  ```

- Symmetric acquisition and release of resources, such as operating file system handles. In Python 2.5, the file object was extended to become a context manager, making it possible to write statements like

  ```
  with open(pathname, "r") as f:
      for line in f:
          process(line)
  ```

  In this fragment, the file f will be closed automatically at the end of the for loop, even if an exception is raised during processing.

– Similarly, in a transactional system, the context manager might control the transaction context, automatically committing the transactions if the statements complete successfully, and rolling back the transaction if an exception is raised.
– In Python's module for decimal floating point arithmetic, the semantics of operations depends on a decimal context; this controls rounding and error mode of all operations. Applications typically want to apply the same rounding and error mode for the duration of an entire computation, and can use the **with** statement to scope the dynamic extent of the decimal context.

5.2 Layers

Similarly to ContextL, PyContext supports the notion of method layers. A layer is a collection of partial method definitions, spanning possibly multiple classes. The partial method definitions can augment or replace an existing method; the same method may have partial definitions in different layers. Dynamically, layers can get activated in the control flow of program; the meaning of a method of a class is then obtained by combining the partial definitions of the methods for all active layers.

As an example of a definition of layered methods, consider the following use case, which originates from the domain of internet client applications. Python offers several libraries to access HTTP URLs (universal resource locators): httplib, which offers direct access to the all wire details of the HTTP protocol, and urllib, which allows a more abstract API. As an example of using urllib, a download of a remote document can be written as follows:

```
import urllib
f = urllib.urlopen("http://www.esug.org")
print f.read()
```

In this case, the urllib library handles all details of the HTTP protocol. In some applications, it is necessary to tailor the protocol interaction to include a User-Agent header in each HTTP request, so that the web server is tricked into believing that the page was requested through a specific web browser (such as Microsoft's Internet Explorer), instead of detecting that it was programmatically accessed through Python's urllib module.

To achieve this functionality, the existing httplib library needs to be augmented to include the User-Agent header. Modifying the library to always set User-Agent to indicate a specific web browser is not acceptable; passing that configuration as a parameter is not possible, since urllib will use httplib in a hard-coded way. Instead, the information of whether a User-Agent header should be sent and, if so, which one, should be defined by the application context.

Using PyContext, it becomes possible to formulate the context using a method layer, and activate that method layer using the **with** statement:

```
import urllib
from useragent import HTTPUserAgent, MSIE6AgentString
```

```
with HTTPUserAgent(MSIE6AgentString):
    f = urllib.urlopen("http://www.esug.org")
    print f.read()
```

In this fragment, a method layer HTTPUserAgent is activated for a block of code, causing all HTTP requests in this block of code to include a User-Agent header; invocations of urllib outside this block (*e.g.*, in a different thread) are not affected. The API of urllib had not to be changed to introduce this context-awareness.

To define the method layer itself, a subclass of context.layers.Layer must be defined

```
from context import layers

class HTTPUserAgent(layers.layer):
    def __init__(self, layer):
        self.layer = layer
```

To define the methods of the layer, the syntax must indicate both what class a partial method belongs to, as well as what layer it belongs to. In PyContext, this is denoted through multiple inheritance (indicating an extension to both the class and the layer). To implement the HTTPUserAgent layer, the method end headers of httplib.HTTPConnection must be extended to explicitly add the User-Agent header. In addition, any attempt to send an additional User-Agent header must be blocked, *e.g.*, preventing urllib from adding its own User-Agent setting. The partial HTTPConnection class then reads as

```
class HTTPConnection(HTTPUserAgent, httplib.HTTPConnection):

    # Always add a User−Agent header
    @layers.before
    def endheaders(self, context):
        with layers.Disabled(HTTPUserAgent):
            self.putheader("User−Agent", context.layer.agent)

    # suppress other User−Agent headers added
    @layers.instead
    def putheader(self, context, header, value):
        if header.lower() == 'user−agent':
            return
        return context.proceed(header, value)
```

In order to produce the complete semantics of a method from its partial definition, the function decorators before, after, and instead can be used. An activation of a layer combines all methods with their newly-activated fragments, making the partial definitions run either before, after, or instead of the original method definition.

Within the context of an activated layer, it is sometimes necessary to disable the layer for further additional recursive calls. In PyContext, this can achieved with the Disabled context manager, which is also demonstrated in the example above.

Implicit layer activation. To support applications that need to factor out context activation from the main program logic, PyContext offers a mechanism to implicitly activate layers. Each layer may define a method active, which determines whether the layer is active. Then, when a layered method is called, the framework first determines which layers are active, and then produces a composition of all method definitions for all active layers. For that to work, the framework needs to know which layer object needs to be checked. Therefore, a function layers.register_implicit needs to be called to subscribe layers for the activation check.

We can imagine a number of design alternatives for defining the semantics of implicit activation. As the layer should be activated depending on context, one question is how often the context should be checked (*i.e.*, how often the active method should be called). One option would be to do this regularly, or whenever an external stimulus arrives (such as an interrupt). While there are cases where either of these approaches work well, there are also cases where they fail, *e.g.*, because a context change does not lead to a hardware or software interrupt. To support the most general case, PyContext evaluates the activation condition on each method invocation of a method potentially affected by layer activation. That may produce a lot of overhead; to reduce that overhead, the layer definition may apply caching techniques in case recomputation of the condition is not necessary every time.

5.3 Context variables

To ease the programming of applications that rely on contextual state, PyContext offers contextual variables, which maintain their value during the dynamic extent of a **with** statement. In that sense, they are similar to dynamic scoping of values in languages like Lisp or SNOBOL4.

A dynamic variable is represented in a globally-accessible Python object. A set method returns a context manager which sets the variable to its new value during execution of __enter__, and restores the previous value when __exit__ is called. Reading the variable is done through a get method, which returns the value of the variable in the current context.

As an example, consider a web application where the web programming framework provides the notion of a session context. With PyContext, the web framework may expose the session object to the application using a context variable. In this example, the access to the variable is still wrapped with a convenience function:

```
from context import Variable

_session = Variable()
def current_session():
    return _session.get()

def process_request(request):
    session = lookup_session(request)
```

```
with _session.set(request):
    dispatch_request(request)
```

In this code, the scoping of the variable session is still static, and still follows the semantics of the Python language. What is dynamically determined is the value associated with the variable. Rather than having a fixed binding of the variable to a value at any point in time, it depends on the execution context, and the dynamic extent of the **with** statement to determine the value the variable possesses.

More precisely, executing the set method on a variable under control of a **with** statement will bind a new value to the variable, hiding the previous value. Invoking the get method on the dynamic variable object fetches the value bound most recently to the variable in the current thread's context. Leaving the **with** statement restores the binding to the prior value of the variable.

In the example, we only show that the variable is read in the same module where it is written. However, the read access might also happen in any other module, as long as that module imports the module where the variable is defined.

5.4 Implementation strategy

The current implementation of PyContext does not perform any modifications to the Python Virtual Machine. Instead, the implementation was completely achieved as a library of regular classes using mechanisms already provided by the language.

Layers are implemented using the meta-object protocol in Python. Layer definitions are based on a custom meta-class provided by PyContext, allowing one to collect all partial method definitions at the point of definition of the layered classes. The original method is replaced by a proxy method which then dispatches to the various partial methods that need to be called, in the (reverse) order of layer activation.

Dynamic variables are implemented using thread-local storage provided by the Python threading library.

Layer activation and binding of dynamic variables uses the concept of context handlers introduced in Python 2.5, as discussed above.

6 Related work

PIE [8–11] extends the Smalltalk object model with of *views*. PIE provides code with multiple views, *i.e.*, representing design decisions from the perspectives of different developers. PIE views need to be explicitly activated similar to ContextL layers, PIE does not support any abstractions for implicit activation or contextual state.

Us [12] supports subjective programming where message lookup depends not only on the receiver of a message, but also on a second object, called the *perspective*. The perspective allows for layer activation similar to ContextL. Us does not support implicit activation of layers.

ContextL [1,13,14] is a language to support Context-Oriented Programming (COP) in LISP; we have discussed the relationship to PyContext above. Context-oriented Programming is discussed in more detail in [15].

Hanson and Proebsting discuss in [16] the notion of dynamic variables. They identify two constructs needed for dynamic variables; a **set** operation that binds the variable to a value, and a **use** operation, that makes it available in the local static scope of a block of code. Our approach deviates slightly from this pattern, as each read operation is denoted as a method call, rather than bringing the variable into local scope for a block of text. Our approach also differs in that no type declarations are necessary for the variables, consistent with the rest of the Python language, which does not require type declarations either. Hanson and Proebsting list a number of other languages that also provide dynamic variables, and discuss the impact of the introduction of dynamic variables to a language; their conclusions apply to this work also.

It's interesting to observe that a number of implementation strategies have been devised for dynamic variables in the past. Hanson and Proebsting mention that various implementations maintain a linked list of all variables that is traversed to determine the location where the value was last bound. They then propose an alternative algorithm that uses a stack walk, similar to the one implemented for exception handling. PyContext uses yet another approach for determining the value of a dynamic variable, namely by using thread-local storage [17].

7 Conclusion and future work

Existing software systems, in particular web applications and other server applications exhibit a great degree of context dependency which currently cannot be expressed adequately and explicitly in the program. With context-oriented programming, these dependencies become explicit, allowing readers of the program to understand more easily how context affects the program's behavior. In addition, properly-designed constructs in the language can help developers to express the context-dependency more concisely.

While we have studied existing software and proposed constructs that assist in addressing context-awareness. Further research is needed to determine the applicability of these constructs to other application domains and other programming languages, *e.g.*, by adding these constructs to existing implementations of context-oriented programming such as ContextL [1].

In this paper, we have mostly neglected issues of performance of the implementation. We believe that efficient implementations of the new programming concepts are possible, but have not made efforts yet to study the performance impact in a realistic scenario, or to improve the performance where that might be necessary.

Acknowledgements

We thank Robert Hirschfeld for the productive discussions, and for bringing together the authors of the paper in the first place. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008).

References

1. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: An overview of ContextL. In: Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05, New York, NY, USA, ACM Press (2005)
2. Holovaty, A., Kaplan-Moss, J.: The Django Book. Apress (2007)
3. Jones, R.: Roundup: an Issue-Tracking System for Knowledge Workers. (2007)
4. Knight, S.: SCons User Guide 0.97. (2007)
5. Gamma, E., Helm, R., Vlissides, J., Johnson, R.E.: Design patterns: Abstraction and reuse of object-oriented design. In Nierstrasz, O., ed.: Proceedings ECOOP '93. Volume 707 of LNCS., Kaiserslautern, Germany, Springer-Verlag (1993) 406–431
6. Kuchling, A.M.: What's new in Python 2.5. Technical report, Python Software Foundation (2007)
7. van Rossum, G., Coghlan, N.: The "with" statement (Python enhancement proposal 343). Technical report, Python Software Foundation (2006)
8. Bobrow, D.G., Goldstein, I.P.: Representing design alternatives. In: Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior. (1980)
9. Goldstein, I.P., Bobrow, D.G.: Descriptions for a programming environment. In: Proceedings of the First Annual Conference of the National Association for Artificial Intelligence. (1980)
10. Goldstein, I.P., Bobrow, D.G.: Extending object-oriented programming in Smalltalk. In: Proceedings of the Lisp Conference. (1980) 75–81
11. Goldstein, I.P., Bobrow, D.G.: A layered approach to software design. Technical Report CSL-80-5, Xerox PARC (1980)
12. Smith, R.B., Ungar, D.: A simple and unifying approach to subjective objects. TAPOS special issue on Subjectivity in Object-Oriented Systems **2** (1996) 161–178
13. Costanza, P., Hirschfeld, R., Meuter, W.D.: Efficient layer activation for switching context-dependent behavior. In: Joint Modular Languages Conference 2006 (JMLC2006). LNCS, Oxford, England, Springer (2006)
14. Costanza, P., Hirschfeld, R.: Reflective layer activation in contextl. In: SAC '07: Proceedings of the 2007 ACM symposium on Applied computing, New York, NY, USA, ACM Press (2007) 1280–1285
15. Robert Hirschfeld, Pascal Costanza, O.N.: Context-orientied programming. Journal of Object Technology (2008) (to appear).
16. Hanson, D.R., Proebsting, T.A.: Dynamic variables. Technical Report MSR-TR-2000-109, Microsoft Research (2000)
17. IEEE: POSIX P1003.4a — Threads Extension for Portable Operating Systems. (1992)