Living in Your Programming Environment

Towards an Environment for Exploratory Adaptations of Productivity Tools

Patrick Rein
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
patrick.rein@hpi.uni-potsdam.de

Jens Lincke
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
jens.lincke@hpi.uni-potsdam.de

Stefan Ramson
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
stefan.ramson@hpi.uni-potsdam.de

Toni Mattis
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
toni.mattis@hpi.uni-potsdam.de

Robert Hirschfeld
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

Abstract

Knowledge workers can benefit from adaptable software tools as they often have individual work flows adapted to their circumstances. To react directly to new use cases, users should be able to adapt the tools while using them and get immediate feedback on their adaptation. Exploratory programming environments already support such an exploratory style for developing software, however they are not used for everyday productivity tasks. In this paper, we describe our first steps towards an exploratory programming environment suitable for everyday productivity tasks. From our experiences of using the environment for eight months, we distilled features which improve adaptability and productivity of such environments.

CCS Concepts • **Software and its engineering → Integrated and visual development environments**;

Keywords exploratory programming, live programming, productivity tools, programming environment, desktop environment

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PX/17.2, October 22, 2017, Vancouver, BC, Canada
© 2017 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5522-3/17/10…$15.00
https://doi.org/10.1145/3167108

ACM Reference Format:
Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, and Robert Hirschfeld. 2017. Living in Your Programming Environment: Towards an Environment for Exploratory Adaptations of Productivity Tools. In *Proceedings of 3rd ACM SIGPLAN International Workshop on Programming Experience (PX/17.2).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3167108

1 Introduction

Knowledge workers such as scientists, financial officers, or journalists often use software tools, such as email clients, task lists, or data analysis environments, to solve domain-specific problems and organize their work. Their work flow is often optimized, tailored to circumstances, and personalized. Work processes throughout one field of work may vary between different people and scenarios and can and do change regularly [7, 14]. For example, journalists might at one time work with official statistics on data and at another time conduct a survey on a focus group.

Consequently, standard software tools, which are tailored for recurrent work patterns across domains, are not always suitable for the task at hand. With work patterns so diverse, it is also difficult for developers to anticipate desired configuration options in advance. For example, the "merge mail" tool in Microsoft Office 2016 can create personalized mass e-mails by drawing information from a Microsoft Excel spreadsheet. However, it is does not allow to add individual attachments to those emails. This can become a nuisance for anyone wanting to send out personalized questionnaires. To fit such specific tasks, the knowledge workers themselves should ideally be able to quickly adapt the software tools. The range of possible adaptations spans from changing the display string of a person to include the age to automating very particular workflows, for example when processing applications to a program whose rules regularly change. As knowledge work

can involve complicated tasks, which require multiple software tools in combination, users might also want to integrate their tools when desired.

Due to the unpredictable nature of their tasks, new use cases and requirements might only be revealed during the use of a tool. Thus, the environment should offer an exploratory style of tool customization allowing users to see the effects of changes immediately within the tool in use. As a consequence, they see the effects of their changes immediately in the situation which initially brought up the new use case. Thereby, users can experiment and explore different designs directly in their work environment.

Self-sustaining exploratory programming environments such as Squeak/Smalltalk [5] and Lively/Webwerkstatt [6] already support the integration and exploratory adaptation of programming tools. They achieve this through a single in-memory and persistent data representation and facilities and features for an exploratory style of development. However, in the recent past, such systems have not been used as environments for everyday tasks, but rather as environments for programming applications or authoring media. In contrast, the present environments for our productivity tools, such as operating systems or the web, make it difficult even for professional programmers to adapt their tools. For example, these environments do not give users access to the source code of an application and do not allow users to change the behavior of applications while they are running by default. Given that even professional programmers struggle to adapt their tools in such environments, tool adaptations by non-professional programmers using these environments seem unlikely.

In this paper, we describe our first steps towards a self-sustaining programming environment to be suitable for everyday tasks such as writing e-mails, managing task lists, or writing this paper. Our observations are derived from an experienced programmer using a Squeak/Smalltalk image as the primary environment for carrying out everyday tasks over a period of eight months. In particular, we describe the modifications to the language, the tools added to the environment, and the applications developed and used. Further, from our experiences with the environment, we derive further requirements and features beneficial to both programmers and users.

Structure of this Paper. After a short introduction to our programming/runtime environment, we describe the tools adjusted, built, and integrated. We comment on related environments and approaches and discuss resulting requirements and desired features for such environments.

2 Features of Exploratory Programming Environments

Exploratory programming is a technique useful in scenarios in which the requirements are not completely understood yet or likely to change often during use of software [11, 16]. Due to the simultaneous use and development of software the problem space is explored, and through experimenting with implementation alternatives, the design space is explored. This is particular interesting for software which is used daily and in a variety of different contexts. Several environments and tools have been proposed which support this style of programming [16].

A survey of exploratory programming environments and methodologies resulted in the following four essential features for "exploratory software" [16, 18]:

- "continuously executable": The product of the exploration process should be a tool which helps its users in working on information. Thus, a mere static representation of the software during development is not desirable. The tool to be created should ideally be continuously running.
- "easily extensible": Programmers should be able to modify the software easily and quickly get feedback on their modification. This means that changes should be easy to apply and should immediately become part of the application even while it is running.
- "conveniently explorable": In order to allow the exploration of design alternative, the environment should support the management of alternatives. This can be achieved, for example through providing branches in a version control system, or dynamic dispatch between different implementations.
- "usefully explainable": The exploratory programming process aims to allow programmers to understand the problem and design space. As such, the environment should provide means to enable programmers to understand the application, for example through state inspection or visualizations of the dynamic system behavior.

Our described environment is based on Squeak/Smalltalk which supports the four described features [5, 11].

2.1 Exploratory Programming Environments and Squeak/Smalltalk

Squeak/Smalltalk is an exploratory programming environment [5]. As it is based on the Smalltalk language, everything in the environment is an object, including the meta-structures of classes and methods [4].

Squeak/Smalltalk supports the "continuously executable" features as it allows developers to run applications next to their development tools in the same environment. The environment does not distinguish between the application and the development tools, as both are mere objects.

Further, as the environment can hot-swap methods, it makes applications "easily extensible". Programmers can edit the source code while the application is running. Their modifications become "live" immediately whenever they save

a method. As soon as their modified method is executed in the application they can potentially see the effects of their modification.

The support for convenient exploration is available for source code as well as runtime state. Alternative versions of the source code can be managed on a small scale through local versioning of methods. On a larger scale, alternative versions of the source code can be managed using integrated version control systems, such as Monticello or git. Runtime state can be versioned by writing the runtime state into an image file, which is a persistent representation of the memory content. While this allows for basic checkpoints of runtime state, it currently lacks support for merging different states or selecting single changes.

Finally, the environment provides tools to support the "usefully explainable" feature. With the object explorer and inspector tools, programmers can inspect and manipulate any object in the environment. The Squeak/Smalltalk debugger also enables programmers to stop the execution of any Smalltalk process and inspect and manipulate the state on the stack. To explore the runtime behavior, programmers can also change the source code of a method in the debugger and continue the execution from the beginning of the modified method.

3 Establishing Productivity Tools in a Development Environment

Squeak/Smalltalk is primarily designed as a programming, media authoring, and learning environment. In order to support integrated productivity tools, a number of adaptations are necessary. Besides adding the actual tools for getting everyday tasks done, we also extended the environment with features such as persistent objects and textual search on collections of objects, and extended the Smalltalk language, for example with a transparent notification mechanism for changes to object state.

All adaptations are derived from everyday use of the environment. Over the course of eight months, one of the authors gradually moved their everyday productivity tools from a Microsoft Windows and Web-based environment to the described Squeak/Smalltalk environment. At the time of writing, our system is in active everyday use (actually this paper was written in the environment[1]) in order to ensure that the tools are actually useful.

In order to illustrate the nature of the environment, we will first give a short walk through through our environment based on a scenario in which the user administers participants of a seminar. This will also provide a brief overview of some of the productivity tools implemented so far. Based on that, we will explain the more technical and generic adaptations of the environment.

3.1 Walkthrough

The following scenario is inspired by the everyday tasks performed in the environment. The concrete names have been changed.

Given a common day, the user arrives at the office and wants to check their email inbox. After starting the environment from the host operating system, they see the screen in Figure 1.

Figure 1. The environment directly after opening it. On the right there are three desktop icons (whitespace intended).

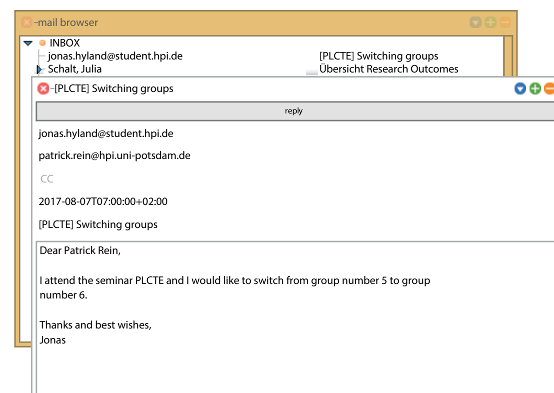


Figure 2. A screenshot showing the mail browser in the background and the mail reader with the student's email in the foreground.

[1]The paper was written as a `Text` object. We used Markdown to describe the structure in an editor supporting basic Markdown syntax highlighting and wrote the contents to a file. A shell script then converted the markdown file with PanDoc to Latex and started a Latex build process. The execution of the shell script could be triggered from the image but the corresponding module was not working at the time of writing. The editing of the final Latex file was also done in the image through a file text editor.

They double-click on the symbol labeled "HPI Inbox" to open a window that shows the emails in the "INBOX" folder of the "HPI" account. Skimming over the list, the user sees an email from a student attending one of his seminars and opens it by doing a double click on the list item. Another window opens showing the contents of the corresponding email object (see Figure 2).

The email says, that the student would like to switch to another group. As the user first has to check with another lecturer of the seminar, they create a todo item by opening a context menu on the list item in the mail browser (see Figure 3).

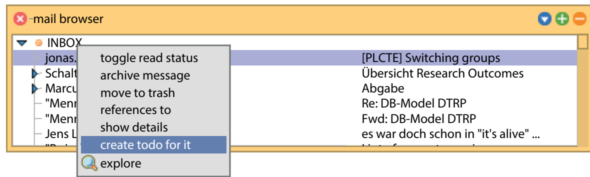


Figure 3. The context menu for a email object.

From a technical perspective this creates a *persistent object* of the class ToDo which is automatically persisted in the global object storage called *soup* protecting it from garbage collection.

After finishing other work and agreeing with the colleague that the student can switch groups, the user wants to come back to the task. Therefore, they open a tool showing the contents of the *object rack* (see Figure 4). The *object rack* is a hierarchical ordering system in which users can store objects in *folders* under self-assigned names. While it looks similar to a file system it merely serves as an organization tool and is not primarily a persistence mechanism.

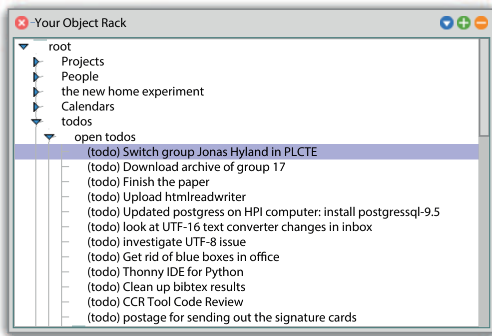


Figure 4. A tool showing the object rack with the open query folder that lists open todo items.

The user looks into the open todos folder. This folder is a query folder which selects persistent objects that comply with the following query:

```
[:object | object isToDo and: [object isDone not]]
```

In order to find the email the todo item refers to, the user performs a double click on the list item representing the todo item and thereby opens up an object explorer. The email the todo item refers to is information which is specific for this instance and is thus stored in an instance-specific field. The user opens the context menu on the email list item to show the details of the email which opens the email reader (see Figure 5).

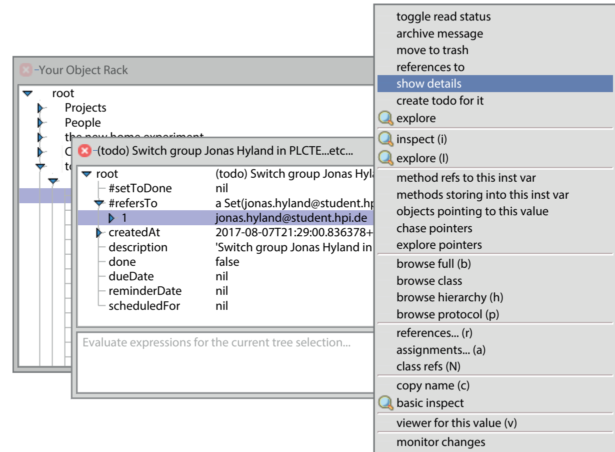
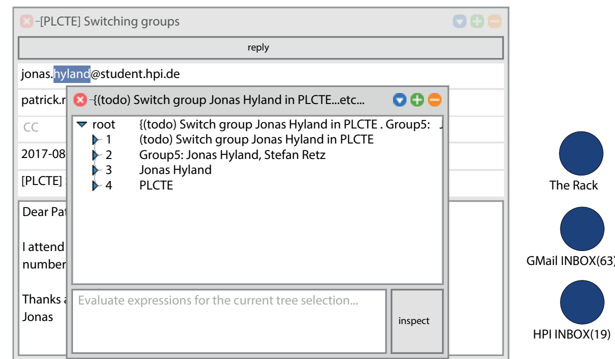


Figure 5. The context menu of a todo item in an ordinary Squeak/Smalltalk object explorer.

Now, in order to change the group the student is assigned to, the user retrieves information about the user by selecting the name of the student in the senders field and performing a full-text search with the name. The result of the full-text search is a set of objects displayed in an object explorer (see Figure 6).



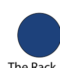


Figure 6. An object explorer showing the result set of the search for the selected string in the mail reader tool.

They also search for the other group by searching for the term `group5 TeachingGroup`. This search provides a set of instances of TeachingGroup matching the term `group5`.

Currently the environment does not include graphical tools to change arbitrary object references. Thus, the user wants to change the group membership of the student programatically. Therefore, the user drags both objects to a workspace in which they can switch the groups with the following script:

```
teachingGroup98123 removeMember: Person129321.
teachingGroup76876 addMember: Person129321.
```

The task is thereby completed. The user hence opens the context menu of the todo item in the explorer again and sets the item to done. This updates the list of open todo items because the tool is notified of all changes to its contained items.

As this was the third email for which the user had to look up the assigned group of the student, the user decides to extend the email reader to allow easier access to information related to a person. In order to implement that, they open up the Squeak/Smalltalk class browser and browse the class describing the email reader tool. As they want to change the way the sender is displayed, they change the `getSender` method to the following:

```
getSender
  | senders |
  sendersText := mailMessage from asText.
 (MailAddressParser addressesIn: mailMessage from)
    do: [:address | | start end block |
    start := mailMessage from findString: address.
      end := (start + address size) - 1.
      searchPerson := [(soup search:
      (Person all search: address)
        first asString) explore].
    sendersText addAttribute: (PluggableTextAttribute
     evalBlock: searchPerson) from: start to: end ].
  ^ sendersText
```

As this code is rather complex, such code might not be written by inexperienced programmers. However, as in our case, this adaptation of the tool was quickly done by an experienced programmer directly while using the tool.

In detail, the method executes the block `searchPerson` when users click on the email address in the senders field. The executed block uses the full-text object search to find information referring to the person the email address belongs to. In order to check whether this implementation already serves the purpose, the user re-opens the email reader on the same message and clicks on the student's email address to open an explorer on the set of information about the student (see Figure 7).

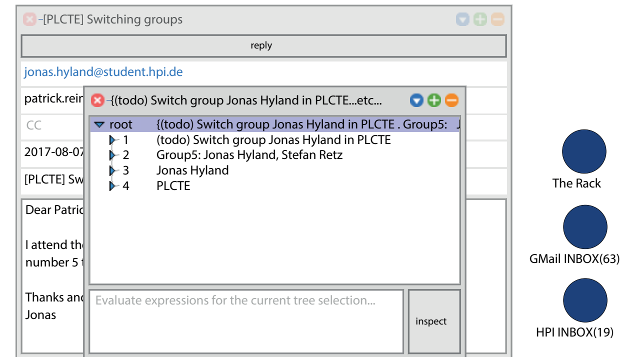


Figure 7. The adapted mail reader tool. The blue color of the text representing the sender indicates that the text can be clicked on. After clicking on it, an object explorer opens up that shows the search result.

3.2 Adjustments to the Language and Environment

Some of the illustrated tools and features are generic extensions of the Smalltalk language or the Squeak/Smalltalk environment, such as persistent objects, instance-specific state, generic observers, and full-text object search.

3.2.1 Persistent Objects

A central adjustment to the environment is the capability of persisting arbitrary objects. The Squeak/Smalltalk environment already provides persistent runtime state through the image mechanism which serializes the complete memory content in a file. However, within the runtime state objects are volatile again, as they are subject to garbage collection. In the Squeak/Smalltalk environment only meta-objects such as classes and some configuration objects are automatically protected from garbage collection. This is achieved by storing them in one of the few global root objects.

In an un-modified Squeak/Smalltalk environment, tools that want to persist their domain objects have to take care of the persistence themselves. A common strategy is to make use of the persistence of class objects by storing the domain objects in an instance variable of a class. This in turn makes integrating tools more difficult, as other tools can only access these domain objects through the idiosyncratic interface of the tool providing the objects. For example, a bibliography management tool might store the bibliography database as a set in one of its classes and only provide access to them through listing the reference keys and allowing to retrieve entries for a reference key.

In our environment, we implemented a central persistence mechanism called *soup* which is a special `Set` object stored in a global root. Any tool can access the soup and store and retrieve all stored objects through the same interface. Thereby, our exemplary bibliography tools could simply store its objects in the soup through the `add:` method. Further, all other tools working with bibliography data can now retrieve

these bibliography objects from there by filtering the *soup* set, for example by testing for class membership.

In order to make it more convenient for users to persist objects, we also provide the `PersistentObject` class. Any instance of that class or one of its subclasses is automatically added to the soup on creation. Beyond that, persistent objects are just common Smalltalk objects.

3.2.2 Instance-Specific State

As objects can be used in a variety of tools, it might be necessary to store unanticipated information in an object. For example, when we want to create a literature survey tool, we might want to store the information to which extent a publication was reviewed to the publication object itself.

However, the Smalltalk language itself only allows for class-defined set of instance variables [4]. In order to support the mentioned use cases, we added an interface for instance-specific state. We achieved this by maintaining a global dictionary mapping from an object and an instance variable name to a value. Access to this state is made transparent by overriding the `doesNotUnderstand:` handler which checks whether the message could be a setter (by checking whether the message takes one argument) or a getter and redirecting the call to an access of the global dictionary.

3.2.3 Generic Observer

To give users a sense of seeing the actual state of objects, graphical tools should always display the current state of objects. In order to update the graphical view on changes to the objects displayed in the tool, the observer pattern is often used, with the tool being the observer of the objects. For this to work, the developer of the domain object class has to be careful to notify the observers whenever relevant state changes.

Through the *soup*, new and unanticipated tools can access domain objects. These tools might need to be notified of state changes which do not yet trigger an observer notification. As the *soup* can contain objects from arbitrary classes, tool developers can not manually adapt these domain classes to trigger notifications.

To prevent this issue, we added a generic observer notification to all writes to instance variables. We achieved this by installing a custom compiler adding the call to the notification method after each assignment to an instance variable. For now, we have limited this modification to the `PersistentObject` subclasses, to prevent a major performance impact due to affecting the base system.

3.2.4 Full-Text Object Search

Users can store any object in the *soup* without storing any other reference to this object. This makes retrieving these objects difficult and users have to resort to either manually sift through the complete *soup* or to programatically filter it. Both ways are too time-consuming for everyday usage when users actually want to just navigate to an object of which they know the relevant terms, such as the name of a person or parts of the title of a publication.

Hence, we added a simple full-text search to the *soup* and all other collection objects in the environment through the message `search:`. The search goes through all objects in a collection, collects the values of their instance variables and instance-specific state, converts them into lowercase strings, and finally checks whether all search terms occur somewhere in these strings.

3.3 Tools

Besides the fundamental modifications of the environments, we have also extended the tool set: We added the *object rack* for organizing objects and we added a context menu mechanism which provides access to selected methods of objects.

3.3.1 Object Rack

The *object rack* is a hierarchical object organization tool. It consists of *folders* and *folder entries*. Users can drag objects into a folder and thereby create an entry which they can give a name. Further, users can create *query folders* which do not store objects but query the *soup* for matching objects and display them. Users can define the query as well as a post-processing script on the resulting set, for example to sort the results.

As the *rack* is persistent, any objects stored in the *rack* are also persisted. However, this is not the same as storing them in the *soup*. If the object is only stored in the *rack* and a user removes the object from there, the object will eventually be garbage collected. Objects that are also stored in the *soup* can be removed from the *rack* and remain persistent.

The environment also includes a graphical tool showing the rack as a hierarchical tree (see Figure 4).

3.3.2 Context Menu for Methods

Most tools in our environment are direct projections of the underlying domain objects onto view elements, for example list items (This perspective on tools is inspired by the model-view-controller (MVC) pattern and the Vivide environment [15]). As a result, the view elements are often direct representations of single objects. Consequently, we added a mechanism which allows users to invoke methods of these objects from the user interface via a context menu.

Therefore, we added the annotation `operationLabel:` which specifies the label under which the method should be available. Whenever a menu for an object is build, all methods with this pragma are collected and used to build the context menu.

```
setDone
  <operationLabel: 'set done'>
  self isDone ifFalse: [self done: true]
```

This works well for simple methods without parameters. For more complex operations, it is still necessary to create a new method which describes the user interactions, for example the method for editing the query of a *rack* query folder:

```
editQuery
  <operationLabel: 'edit query'>
  ^ UIManager default
    edit: self query decompile decompileString
    label: 'Edit folder query'
   accept: [:v | self query: (Compiler evaluate: v)]
```

4 Living in and Adapting Your Environment

Through the described features, its user was able to use the environment for everyday tasks. Further, in several situations, the advantages of an adaptable environment became clear. At the same time, we have also experienced several problems, with regard to the integrity and safety of data, as a consequence of the system providing little protection between tools.

To illustrate the extend to which the environment was used, we first give an overview of the domain models created and used.

4.1 Created Domain Models

The environment has mostly been used for work in the context of university research and teaching. Thereby, the following subclasses of `PersistentObject` were created:

- ToDo
- DiaryEntry
- DiaryHassle
- TeachingGroup
- TeachingTopic
- Course
- Agent
- Person
- CreativeWork
- Conference

Besides the ToDo class the following classes were also used for private projects:

- GardenPlanting
- GardenMap
- GardenPlant
- GardenPot

In total, this resulted in 847 persistent objects stored in the *soup* up until the day of writing. Most of them were created in the environment, some were imported, for example through a tool for importing BibTex data.

In addition to these domain models, an object-based interface to maildir directories and an object-based interface to CalDAV calendars was created. For the maildir objects, a mail browsing, reading, and writing tool was created (see Figure 2). For the calendars, a deadline calendar tool was created which shows all weeks of a year and whether they contain a deadline.

4.2 Adaptation and Integration

We want to further illustrate the potential for ad-hoc adaptations and integration in the environment with two examples. The first one illustrates how easily we can extend the environment with features affecting the complete environment. The second one demonstrates how powerful the tools can become through adaptations.

4.2.1 Creating Todo Items

As the complete behavior of the environment is accessible and adaptable, we can implement features changing the behavior of the complete environment. In this case, we implement that users can create a todo item for any object in the environment. Users should be able to execute this from the generic context menu of objects in all tools. In order to provide this generic operation, we implemented the method in the `Object` class, which is the superclass of most classes in the environment. The following method describes the operation:

```
interactiveCreateToDoForIt
  <operationLabel: 'create todo for it'>
  ^ (ToDo new: (UIManager default
        request: 'Describe the ToDo:'))
          refersTo: self;
          yourself
```

As this extension was developed spontaneously, we did not modify the list of instance variables of the `ToDo` class. Instead, we set the object referred to as instance-specific state through the call `refersTo:`.

This adaptation is also rather easy to integrate, as the existing tools, such as the Squeak/Smalltalk object explorer have already been adapted to respect the annotated methods, when building a context menu.

4.2.2 Z3 Solver for Seminar Topic Assignment

Having objects and programming facilities easily available also opens up new possibilities for leveraging computational resources. For example, when organizing lectures it is sometimes necessary to assign topics to groups of students. In this scenario, 17 student teams handed in three ordered wishes out of a list of 20 suggested project topics. Instead of determining the assignment manually, we generated a problem description for the Z3 solver using the following script:

```
weights := #(50 45 10).
result := ((teams collectWithIndex:
  [:t :i | '(declare-const G' , i asString , ' Int)'])
    joinSeparatedBy: String crlf).
```

```
result := result , String crlf ,
  '(assert (distinct ' ,
  ((teams collectWithIndex: [:t :i |
    'G' , i asString ]) joinSeparatedBy: ' ').
result := result , '))'.
result := result , String crlf ,
 ((teams collectWithIndex: [:t :i | | groupString |
    groupString := 'G' , i asString.
    t topicWishes collectWithIndex: [:w :index |
      '(assert-soft (= ',
      groupString,' ',(topics indexOf: w),
      ') :weight ',(weights at: index),')' ] ])
  flatten joinSeparatedBy: String crlf).
result := result , '
(check-sat)
(get-model)'.
```

We then used the resulting script as input for a Z3 solver and then parsed the result string back into objects from our teaching model:

```
(solved lines pairsCollect: [:g :t |
  (teams at:
   ((g findTokens:' ') second copyWithout: $G) asNumber)
   -> (topics at: ((t select: #isDigit) asNumber))]).
```

As these scripts were created ad-hoc for this particular use case, they exhibit several code smells and might be difficult to maintain. At the same time, these scripts were never intended to be reusable as they were easy to write and were only intended to solve the particular issue in that moment.

4.3 Data Loss and Corruption

Due to the open and integrated nature of the environment, a major issue we experienced is the loss or corruption of personal data. We have experienced both and want to explain the observed causes for these incidents.

4.3.1 Data Loss

Data loss occurred pre-dominantly because of technical reasons. We can distinguish between causes originating from the platform and causes originating in the developed tools.

Issues in the first category were mostly caused by the host operating system (Microsoft Windows) terminating the Squeak/Smalltalk process, for example due to an automatic restart after an applied update. The virtual machine (VM) process can be terminated, as it does not provide information on unsaved changes to the operating system. Another reason is a failing VM resulting in a corrupted image file. For stability reasons, we are using a 32bit VM which can lead to corrupted image files in case an object-space with too many objects is serialized. In the case of a corrupted image file, we were able to use a host system backup of the image file. In the case of a terminated VM process, there was not yet any backup available as the image file was not yet updated.

Issues in the second category were caused by faults in the new tools. For example, we had an undetected fault in the maildir synchronization library that writes changes in the objects back to the file system. As a result, six email were completely deleted from the mail account. As a consequence, we first improved the test coverage of the module before we continued using it.

4.3.2 Data Corruption

The data corruption that occurred was caused either by a fault in a tool or by unconsciously creating persistent objects.

As an example of the first cause, we experienced a failure during the importing of BibTex data. The failure occurred at a point of the import process at which new persistent `Person` objects were created. The failure left one such instance created but unpopulated. The instance lingered in the environment for a while until we discovered it accidentally while looking for information on another person.

Such empty instances were also often created when we worked on a script or new method. We were often not aware that the script created a new persistent instance. This was less problematic as long as the instance remained empty but become a nuisance when filled with sensible data, for example resulting in several objects which all represent the same person.

5 Discussion

The described modifications of the Squeak/Smalltalk system and the observations made are only first steps towards an environment for the productive use and exploratory adaptation of tools. Based on our experiences so far, we determined a list of beneficial features for such an environment. Further, we unveiled a number of conceptual challenges for such an environment.

5.1 Beneficial Features

The described environment is already useful for everyday productivity tasks. Based on our experiences, we argue that, amongst others, the following six features are beneficial for similar environments. This list does not represent a complete catalog of features of exploratory programming environments hosting productivity tools, but should serve as a starting point for developers creating similar environments.

- Central Object Persistence: The central persistence of objects allows all tools in the environment to access the data of other tools. As the persistence mechanism does not allow for separate storage for single tools, all objects can be accessed from the same location. Thereby, when users want to integrate objects normally accessed through another tool, the can make ad hoc use of them.
- Organization Tool for Objects: While object persistence helps to keep important objects, it does not help

with accessing the objects. Thus, some form of organization tool for objects, for example folders, tags, or a diary of recently used objects, is beneficial.
- Instance-specific Fields: As many tools can work together on the same object graph, there might be situations in which unanticipated information needs to be added to an object. The environment or programming language should allow for such information to be added to an object with the same effort as is required to add common information.
- Transactions: Tools and scripts have a direct effect on the persistent objects. As described above, this can lead to data corruption or even data loss. The fear of corrupting production data might in turn hamper the exploration of new features. To prevent data corruption, the environment should support transactions and a recovery system. Using transactions, programmers can experiment more easily and revert erroneous state changes.
- Backups: The loss of production data can destroy the trust of a user in the environment. Thus, a backup mechanisms should be in place to recover from any loss of data. Squeak/Smalltalk already provides a backup system for changes to methods and class definitions. Every change is written to a log file which can be replayed on a crash. However, this does not include changes to the runtime state. Based on the transaction mechanism, the environment should provide a backup mechanism for all persistent objects.
- Storage Scaling: With the current design of keeping all data in memory, there is a natural limit to the amount of information that can be managed by such an environment. To become a viable alternative to common operating systems, the environment has to allow users to manage amounts of information that are greater than the main memory. Therefore, the system has to provide a swapping mechanism through which the virtual machine only has to keep parts of the object graph in memory.

5.2 Conceptual Outlook

Interesting conceptual challenges arising from the development of the described environment span the question of programing expertise required to adapt tools, the integration of other languages and paradigms, and the integration of information whose life-cycle is managed by external services.

5.2.1 End-Users

So far, our environment is designed for a programmer with more than five years of programming experience in the Squeak/Smalltalk environment. As such, the experiment can only give us insights into how much such a system can be adapted by a person already capable of programming. Further, as the person using the environment is looking for opportunities to adapt the tools, the resulting list of adaptations will have be biased.

Consequently, the question arises to which extent the adaptation of tools can be made accessible to users without programming skills in a way that end users will make use of it. While changing the sorting criteria for a list might be well within reach, an adaptation of the mass email function to also allow for attachments might not be. First interesting steps would be to provide users with a block-based programming language for particular aspects of the application and study the impact on the number and extent of adaptations. Suitable first aspects to be made accessible to end-users would be anything that can be adapted through a simple functional projection from the domain data, such as sorting, changing the display string, or adding new information made up from other parts of domain data (e.g. constructing the full name of a person).

5.2.2 Integrating other Languages

Parts of the benefits of the presented environment have to be attributed to the fact that all applications are written in a single language using a data structure to represent information. Due to today's diverse application domains it would be beneficial to allow for a variety of languages and paradigms to be used in such an environment. However, to retain the advantages regarding adaptability and integration of tools, all these languages would need to integrate with the Smalltalk object model. This means that these languages need to be adapted to use objects as their central data structure. While several languages have been embedded into Smalltalk [10], the question remains whether these embedded languages could be adapted to use objects as a data structure without also adopting message dispatch as a control-flow mechanism.

5.2.3 Interfacing with External Systems

The described environment has an architecture which is an alternative to the architectures of existing computing environments used today. As such, special care needs to be taken to integrate this new environment with the existing environments. We have experienced the discrepancy between these existing systems and the described environment, when we tried to work with information managed by external systems. For example, when using the IMAP protocol to organize emails, we do not change local email objects but actually change the data structures hosted on a server [9]. At the same time, we would like to allow users to access their emails as persistent objects the way they can access, for example, local ToDo item objects.

In particular, this integration entails the issue of serialization and preservation of identity in the system. When importing bibliography data from a BibTex file, the list of

authors is only available as a list of names. The disambiguation has to happen at import time through user intervention. This intervention becomes necessary as the exact identity information for single people is not encoded in the BibTex file. The manual intervention is infeasible for larger amounts of data which users might browse, for example through a web-based application programming interface (API).

Further, to enable scripting, some form of local copy of objects has to be maintained by the environment. At the same time, the remote system might manage the life cycle of the object and might, for example, delete the object. This entails that first, the environment requires a way to get notified about these changes, and second, that the environment has to decide how to modify the local state. This should depend on what users would expect to happen to their objects, which is to be determined in future work.

6 Related Work

The idea of an adaptable environment for productivity tools can be found in numerous environments with varying support for exploratory adaptation.

A historically outstanding class of environments are the software systems running on Lisp Machines [12]. As the hardware was able to execute Lisp natively the complete system was written in Lisp. This enabled users to change everything in the system while using it and seeing the effects directly in their applications. Further, some of the Lisp systems implemented Common Lisp and thus supported object-oriented programming. As all software was running in Lisp, applications were able to exchange these objects directly. Finally, some of these systems [12] provided means to persist trees of Lisp objects. Unfortunately, there are no accounts of the experience of users working with these systems.

One of the currently most prominent environments which allows its users to adapt it to their needs are Unix environments. Most parts of these environments are accessible to the user through a single interface, which is files and file operations. Further, many tools can be adapted and extended using bash scripts. However, the support for exploratory development of these tools is limited by default. Runtime state can not be inspected and running programs can not be modified in general.

The EMACS environment improves on these limitations [13, 16]. Initially designed as a text editor, it has extensions which add productivity tools such as the org-mode for managing tasks [2] or Gnus [3] for reading emails. Most of the environment is adaptable directly from within the environment through the LISP language. Further, the environment provides tools for the exploratory development of the environment. Due to its original purpose, the environment is inherently text-based, which limits the tools which can developed for it.

Apple Script [1] allows users to automate common tasks by writing scripts which use Apple Script APIs of installed applications. While it allows for an easy extension of the environment, the capabilities of scripts are limited to the Apple Script API the applications provide.

The Microsoft Office tool set allows users to adapt tools through writing scripts for example in Visual Basic for Applications (VBA) [17]. The scripting engine is available in all major tools of the Office tool set. Users can call the VBA API of the office tools, the surrounding Microsoft Windows operating system, and of applications integrating the Microsoft script engines. Similar to Apple Script, the capabilities of the scripts are limited to exposed functionality and data.

The described approaches are designed around local data, most often stored in the form of files. New software tools are often implemented as mobile applications or web applications which work with services running on remote servers. Adapting single applications is hindered by the fact that important parts of the application are not running locally. However, services such as if-this-then-that (IFTTT) [8] allow users to adapt their tools by combining them as individual services. Through a block-based programming interface in the web users can define which actions should be executed whenever certain events occur. The events can stem from any service supported by IFTTT and correspondingly the actions can be executed by any supported service. In this approach, the user is limited to configuring explicitly exposed functionality. Further, there is no tool support to observe or explore the dynamic behavior of the combinations.

Users can also adapt web applications by writing browser plugins which then change the behavior of a web page after it loaded. Browser plugins require some technical knowledge with regard to the life-cycle of such a plugin. At the same time, users can write plugins using the development tools provided by the browser, for example the Chrome toolkit, which provide a live programming experience and allow users to work directly on the webpage they want to adapt. However, the access to data on the web remains restricted by the API the server provides. Furthermore, despite the fact that the source code is available for every webpage by design, the architecture of current web applications sometimes restricts the access to relevant objects in order to prevent adaptation by users[2].

7 Conclusion

Software tools should be adaptable and well integrated to support the unpredictable and unstructured work patterns of modern knowledge workers. In this paper, we illustrated how a modified self-sustaining exploratory programming environment can serve as a prototype for an architecture supporting this adaptation and integration of everyday software

[2] See for example https://github.com/openstreetmap/openstreetmap-website/issues/1166

tools. In particular, we described an initial set of adaptations to Squeak/Smalltalk that support the development and use of productivity tools within the environment. These modifications were derived from daily use of the environment and their benefits illustrated through example workflows of how at least an experienced programmer can adapt tools with ease. Through our prototypical environment, we hope to hint a way forward towards an architecture supporting the exploratory adaptation and integration of tools.

Acknowledgments

Sincere thanks also go to all PX workshop participants, who provided valuable feedback by discussing this topic thoroughly. We would like to thank Alexander Meissner who has influenced many design decisions through numerous discussions. We gratefully acknowledge the financial support of HPI's Research School (http://hpi.de/research_school) and the Hasso Plattner Design Thinking Research Program. (https://hpi.de/en/dtrp)

References

[1] William R. Cook. 2007. AppleScript. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. ACM, New York, NY, USA, 1–1–1–21. https://doi.org/10.1145/1238844.1238845

[2] Carsten Dominik. 2010. *The Org-Mode 7 Reference Manual: Organize Your Life with GNU Emacs*. Network Theory, UK. with contributions by David O'Toole, Bastien Guerry, Philip Rooke, Dan Davison, Eric Schulte, and Thomas Dye.

[3] Inc Free Software Foundation. 2015. *The Gnus Manual*. http://www.gnus.org/manual.html

[4] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley.

[5] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan C. Kay. 1997. Back to the Future: The Story of Squeak - A Usable Smalltalk Written in Itself. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA) 1997*. 318–326. https://doi.org/10.1145/263698.263754

[6] Jens Lincke. 2014. *Evolving Tools in a Collaborative Self-supporting Development Environment*. phdthesis. Universität Potsdam. https://lively-kernel.org/publications/media/Lincke_2014_EvolvingToolsInCollaborativeSelfSupportingDevelopmentEnvironment_PRINT.pdf

[7] Nicolas Mundbrod, Jens Kolb, and Manfred Reichert. 2012. Towards a System Support of Collaborative Knowledge Work. In *Proceedings of the Business Process Management Workshops (BPM) 2012*. 31–42. https://doi.org/10.1007/978-3-642-36285-9_5

[8] Steven Ovadia. 2014. Automate the Internet With "If This Then That" (IFTTT). *Behavioral & Social Sciences Librarian* 33, 4 (2014), 208–211. https://doi.org/10.1080/01639269.2014.964593 arXiv:http://dx.doi.org/10.1080/01639269.2014.964593

[9] Yakov Rekhter and Tony Li. 2003. *INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1*. RFC 3501. RFC Editor. https://tools.ietf.org/html/rfc3501.txt

[10] Lukas Renggli. 2010. *Dynamic Language Embedding*. phdthesis. Philosophisch-Naturwissenschaftliche Fakultät der Universität Bern.

[11] D. W. Sandberg. 1988. Smalltalk and Exploratory Programming. *SIGPLAN Not.* 23, 10 (Oct. 1988), 85–92. https://doi.org/10.1145/51607.51614

[12] Richard Stallman, Daniel Weinreb, and Moon David. 1984. *Lisp machine manual*. Massachusetts Institute of Technology. https://books.google.de/books?id=CX4ZAQAAIAAJ

[13] Richard M. Stallman. 1981. EMACS the Extensible, Customizable Self-documenting Display Editor. In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*. ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/800209.806466

[14] Witold Staniszkis. 2015. Empowering the Knowledge Worker: End-User Software Engineering in Knowledge Management. In *Proceedings of the Conference on Enterprise Information Systems (ICEIS) 2015*. Springer, 3–19.

[15] Marcel Taeumel, Michael Perscheid, Bastian Steinert, Jens Lincke, and Robert Hirschfeld. 2014. Interleaving of Modification and Use in Data-Driven Tool Development. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!) 2014*. ACM, 185–200.

[16] J. Trenouth. 1991. A Survey of Exploratory Software Development. *Comput. J.* 34, 2 (1991), 153–163. https://doi.org/10.1093/comjnl/34.2.153

[17] John Walkenbach. 2010. *Excel 2010 power programming with VBA*. Vol. 6. John Wiley & Sons.

[18] Simon Yun Pui Yung. 1992. *Definitive Programming: A Paradigm for Exploratory Programming*. Ph.D. Dissertation. University of Warwick.