A Soup of Objects

Convenience Interfaces for Accessing Domain Objects in a Global Object Graph

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

ABSTRACT

Conventional desktop systems, such as Microsoft Windows or MacOS, are structured around applications. From a technical perspective the domain objects, such as emails or tasks, are contained within these applications. This separation of object graphs restricts interactions and integrations between applications to cases for which the original developers added support. Through the Home system we want to explore an alternative architecture for desktop systems supporting such ad-hoc integrations. This architecture is based on a single shared runtime object graph spanning all applications. We evolved and evaluated our architecture and the described mechanisms by using the resulting environment for over 13 months for everyday productivity tasks.

CCS CONCEPTS

• **Software and its engineering** → **Software architectures**; *Software system structures*; • **General and reference** → *Empirical studies*;

KEYWORDS

personal information management systems, domain objects, exploratory programming environments, Squeak/Smalltalk

ACM Reference Format:
Patrick Rein. 2018. A Soup of Objects: Convenience Interfaces for Accessing Domain Objects in a Global Object Graph. In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming'18> Companion).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3191697.3213799

1 INTRODUCTION

Conventional desktop systems, such as Microsoft Windows or MacOS, are structured around *applications* [4, 8]. Each application is concerned with the organization and processing of certain *objects of a domain*. For example, an email application allows users to read, sort, and write emails and a teaching management application allows users to manage students and courses. While this architecture enables the independent development and execution of applications, it also prevents task-specific integration of applications.

<Programming'18> Companion, April 9–12, 2018, Nice, France

ACM ISBN 978-1-4503-5513-1/18/04…$15.00
https://doi.org/10.1145/3191697.3213799

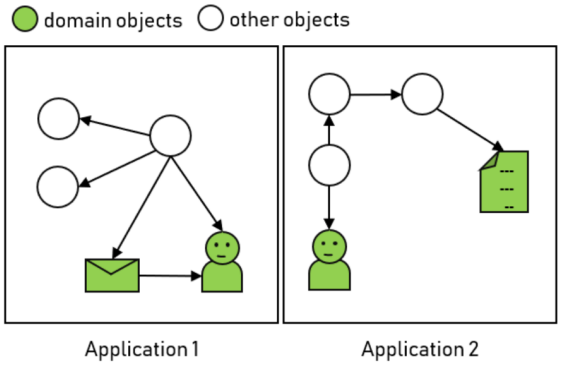

Figure 1: An illustration of the separation of object graphs in conventional desktop systems. The two applications have completely separated object graphs.

From a technical perspective the domain objects are contained within the application (see Figure 1). During run-time these objects of the domain are represented as runtime *domain objects*. These are stored in the application-specific run-time object graph which is separated by process boundaries from the object graphs of other applications. Further, when the application is not running, the domain objects are stored in an application-specific file system location which is also separated from the locations of other applications.

This separation of object graphs restricts interactions and integrations between applications to cases for which the original developers added support. However, most interactions between applications are activity-specific and depend on the particular applications used [7, 11]. For example, a lecturer communicating with students might want to reference email objects and course objects in To-do items. Also, from the To-do list application, it should be possible to open a To-do item referencing these objects and jump into the email or teaching application. This is currently only possible if both applications allow access to the domain objects and the To-do application explicitly makes use of this access.

Through the *Home* system we want to explore an alternative architecture for desktop systems supporting such ad-hoc integrations. This architecture is based on a single shared runtime object graph spanning all applications as is provided by Smalltalk-like systems or Lisp Machines (see Figure 2) [2, 5, 12]. Thereby, every application can access the objects of other applications and operate on them through calling methods.

The *Home* system is an extension of the existing Squeak/Smalltalk environment [6], which is a good starting ground as it already provides a persistent global runtime object graph. However,

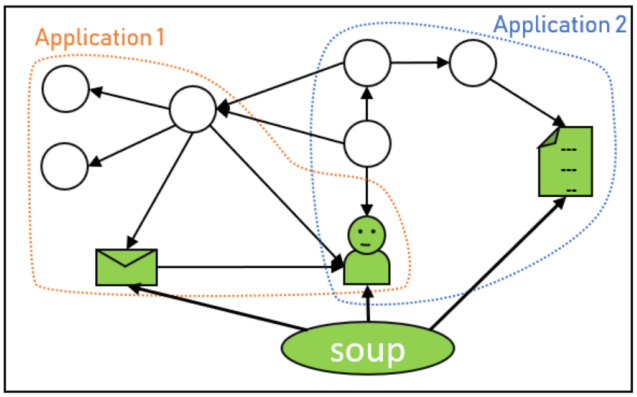

Figure 2: An illustration of the object graph in the Home system. The object graph of both applications is connected and the application boundaries are less distinct. Nevertheless, access to domain objects such as the document one on the right might be difficult without detailed knowledge about the implementation of the second application. The *soup* provides global access to all such domain objects.

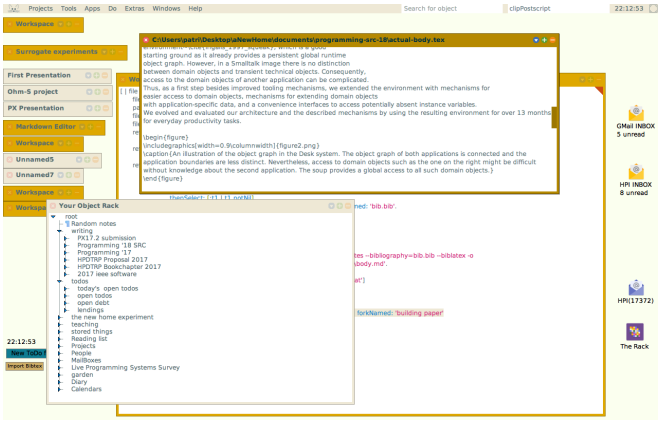

Figure 3: A screenshot of the current state of the Home system as it is being used by the author.

in a Smalltalk image there is no distinction between domain objects and transient technical objects. Consequently, access to the domain objects of another application can be complicated (see Figure 2). Thus, as a first step besides new tools, we extended the environment with mechanisms for easier access to domain objects, mechanisms for extending domain objects with application-specific data, and a convenience interfaces to access potentially absent instance variables. We evolved and evaluated our architecture and the described mechanisms by using the resulting environment for over 13 months for everyday productivity tasks.

2 MECHANISMS OF THE HOME SYSTEM

To allow easy access to domain objects, users of the system can implement their domain object class as a subclass of `PersistentObject`. This class automatically takes care of making these objects globally accessible through a special collection called *soup* (see Figure 2). The soup is a global set containing all domain objects in the system. In this architecture, applications get their data mostly from querying the soup. The following example shows a query for getting the list of open To-dos:

```
soup select: [:object | object isToDo
  and: [object ? #isDone = false]
  and: [(object
            answer: #scheduledFor
            or: Date tomorrow)
          <= Date today]]
```

Further, applications might want to store additional information on existing domain objects, so we added object-specific instance variables. In the example, the `scheduledFor` and `isDone` fields are not defined in the `ToDo` class but are added to instances. As these fields could be absent, code has to be more defensive in accessing fields. Thus, we added a convenience interface consisting of methods returning the value or an appropriate value representing absence (for example the `answer:or:` method in the example above).

3 EVALUATION

We evaluate whether such an open architecture can lead to a tighter integration by using the system for everyday tasks during the past 13 months [1]. So far, the author spends over 50% of his time at the computer within the environment and common productivity tasks (task, email, document management) happen almost exclusively in the environment (for an impression see Figure 3). We collect data by recording the time spent in the environment as well as a semi-structured diary to record interesting incidents [9]. Through this exploratory study we discover interesting synergies and future requirements. A recent insight is that the soup, even combined with a hierarchical ordering system for objects, leads to an impression of loosing track of data.

4 RELATED WORK

Operating and desktop systems build upon one global storage model can provide similar capabilities to users, for example Unix through the philosophy of "everything is a file" or Lisp Machines [2, 10]. Further, scripting languages on operating system level (for example AppleScript or Visual Basic Script [3, 13]) can enable ad-hoc integration to some extent. However, they restrict access to the interface provided by the original application developers.

REFERENCES

[1] Philip Agre. 1997. Towards a Critical Technical Practice: Lessons Learned in Trying to Reform AI. *Social Science, Technical Systems and Cooperative Work: Beyond the Great Divide. Erlbaum* (1997).
[2] Hank Bromley. 1986. LISP Lore: A Guide to Programming the LISP Machine. (1986).
[3] 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
[4] Donald Gentner and Jakob Nielsen. 1996. The Anti-Mac Interface. *Commun. ACM* 39, 8 (1996), (70 to: 82). https://doi.org/10.1145/232014.232032
[5] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, USA.

[6] Daniel Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Smalltalk and Exploratory Programming*, Vol. 32. ACM, (318 to: 326). https://doi.org/10.1145/263698.263754

[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] Pamela Ravasio, Sissel Guttormsen Schär, and Helmut Krueger. 2004. In Pursuit of Desktop Evolution: User Problems and Practices with Modern Desktop Systems. *ACM Trans. Comput.-Hum. Interact.* 11, 2 (June 2004), 156–180. https://doi.org/10.1145/1005361.1005363

[9] Colin Robson. 2002. *Real World Research.* Blackwell Publishing.

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

[11] 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.

[12] David Ungar and Randall Smith. 2007. Self. In *Proceedings of the Conference on History of Programming Languages (HOPL) 2007 (HOPL III)*. ACM, New York, NY, USA, 1 to: 9. https://doi.org/10.1145/1238844.1238853

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