In Hasso Plattner, Christoph Meinel, and Larry Leifer (eds.)
Design Thinking Research: Making Distinctions: Collaboration versus Cooperation (pages 171-194)
Springer 2017 (doi:10.1007/978-3-319-60967-6_9)

Making the Domain Tangible: Implicit Object Lookup for Source Code Readability

Patrick Rein, Marcel Taeumel, and Robert Hirschfeld

Software Architecture Group, Hasso Plattner Institute, University of Potsdam, 14482 Potsdam, Germany

Abstract Programmers collaborate continuously with domain experts to explore the problem space and to shape a solution that fits the users' needs. In doing so, all parties develop a shared vocabulary, which is above all a list of named concepts and their relationships to each other. Nowadays, many programmers favor object-oriented programming because it allows them to directly represent real-world concepts and interactions from the vocabulary as code. However, when existing domain data is not yet represented as objects, it becomes a challenge to initially bring existing domain data into object-oriented systems and to keep the source code readable. While source code might be comprehensible to programmers, domain experts can struggle, given their non-programming background. We present a new approach to provide a mapping of existing data sources into the object-oriented programming environment. We support keeping the code of the domain model compact and readable while adding implicit means to access external information as internal domain objects. This should encourage programmers to explore different ways to build the software system quickly. Eventually, our approach fosters communication with the domain experts, especially at the beginning of a project. When the details in the problem space are not yet clear, the source code provides a valuable, tangible communication artifact.

1 Introduction

Programmers acquire domain knowledge to better understand the problem space and create a solution that fits the users' needs [Evans2004]. For this, programmers and domain experts form a shared vocabulary to foster knowledge exchange. This vocabulary consists, to a broad extent, of terms describing real-world concepts. For example, the accountant may deal with transactions, the geologist with soil horizons, the biologist with DNA strands, and the cook with ingredients and recipes. To allow programmers to express this vocabulary in code, it is beneficial to make use

of an object-oriented programming language [Wegner1987]. In such a language, programs are made up of objects, which are virtual representations of relevant artifacts from the domain. In the cooking domain, objects can represent concrete things such as vegetables and also abstract concepts such as recipes. What we can observe and manipulate in the real world, we can express as object behavior and object relationships [Kay1996]. In a cooking simulation, we might want to slice an apple with a knife and add the resulting slices to the dough. Code that describes this process preferably looks like this: `(anApple cutWith: aKnife) do: [:eachSlice | dough add: eachSlice]`. It is usually possible to represent domain concepts as such interacting, message-exchanging objects [Meyer1997]. Consequently, understanding the problem space means understanding the sorts of objects that are required to construct the software system.

Software development is an iterative process [Beck2000] that benefits from exploratory programming strategies [Sandberg1988]. Even early versions of the software are examined by all parties to clarify requirements and future directions. Once the programmer presents first working prototypes, the domain expert gains a better understanding of how the domain data can be processed by the software.

At the same time, the shared understanding of the objects relevant for the domain improves iteratively during the development. Programmers and domain experts define what certain terms refer to in the context of the application during discussions of the application requirements. In an object-oriented programming language, these terms and definitions are ideally used for identifying and defining objects and messages. Through this domain-specific code, the source code evolves into a written documentation of the domain knowledge. The source code of an application is a suitable documentation as all terms defined in it are relevant to the application and their definitions are what makes up the application. Consequently, domain experts should be able to recognize relevant domain concepts and they should be able to actually *read* the source code. If the shared vocabulary is documented in this way and can be understood, it becomes tangible and a vital subject to discuss and refine next to the application itself.

However, some aspects of the software development process hinder programmers to directly express domain concepts in source code. Among the main challenges at the beginning of each software project, programmers have to learn about the existing information. Typically, there are several databases that store the domain information to be processed. Such databases are full of numerical data and text snippets, which represent domain-specific concepts such as cooking recipes. Programmers have to write code to access these databases and build bridges between variously shaped domain information and the object-oriented world [Papakonstantinou1995].

It is, however, challenging to create domain-specific objects from existing database data in new systems [Evans2004]. The basic means to access and process data originating from external systems are verbose and impede the readability of domain logic. For example, if programmers want to open files or make Web requests, they will have to invoke several helper objects and process the results to eventually get

domain objects. It takes usually several iterations of re-writing source code to improve readability [Fowler1999].

In the Squeak/Smalltalk programming environment[1], everything is an object. For example, the text snippet `'marcel.taeumel@hpi.de'` is, in fact, an object that represents some text and encodes an email address in this case. Programmers cannot ask this object for its `#authority`, which is `hpi.de`. First, a helper object has to convert the text object into a URL object, which then understands that message: `(Url absoluteFromText: 'marcel.taeumel@hpi.de') authority`. As text is prevalent in existing databases, the conversion of text fragments to domain objects is prevalent. Still, the process of writing such code in a readable fashion is prone to mistakes and takes time. Interestingly, programmers use many libraries that already know about many domain-agnostic structures such as URLs. Only their domain-specific activation is for the most part explicit. Imagine an address book stored in the file named `'friends.txt'`. In that book, the authority of a person's email address can be looked up like this: `('friends.txt' person: 'marcel') email authority`. Unfortunately, programmers have to struggle with many intermediate steps to access and process this data. Many of these steps cannot easily be hidden and remain visible in the code. Without additional efforts, knowledge exchange between programmers and domain experts in terms of code becomes unfeasible.

Based on these observations, we want to investigate various means to reduce programming effort when connecting object-oriented systems to external data sources. If a text looks like an email address, for example, programmers should be able to directly treat it like one in the code. We want to address the following research question:

> In object-oriented systems, how can we support programmers in writing domain-specific code to improve code readability through separating object access from object use?

It is beneficial to offer domain experts a readable form of object-oriented code as soon as possible. This fosters knowledge exchange and helps clarify requirements. Hence, a major goal is to be able to talk to domain experts about the very material that makes up the software system: the source code.

We want to elevate domain knowledge from various data sources to an object-oriented programming system. For this, we employ a set of extensible *predicates*, *resolvers*, and *mappers* to support transparent exploration of objects and their relationships based on primitives such as strings, numbers, or dictionaries originating from external sources. We focus on the self-supporting Squeak/Smalltalk programming environment to benefit from the object-oriented programming paradigm, tools with short feedback loops, and omnipresent run-time information. In our approach, any generic object can be treated as an *identifier* to be resolved, for example, by a

[1] See http://www.squeak.org for details

Web request. The resulting, usually generic, object will then be mapped to one or more specific objects, depending on predicate matches. If there is no appropriate class to create a domain-specific object, our mechanism establishes a user dialog to create one. Many domain-specific applications can be constructed on top of this mechanism. Since our approach works on generic objects that are already materialized in the system, we can treat any set of objects as a data source itself. This underlines the self-supporting nature of the Squeak/Smalltalk system.

In this chapter, we:

- Present the model of a framework that supports exploration of an object graph based on external data sources
- Elaborate on several scenarios to clarify the programmer's effort and added value
- Discuss opportunities and challenges regarding expert communication and system maintenance

The next section provides background information about how programmers work and think in terms of objects in the programming system Squeak/Smalltalk. We make a clear distinction between generic objects and domain objects. After that, section 3 explains the basic model and components of our approach. We elaborate on object roles, role transitions via resolve and map, as well as the impact on extensibility and code readability. Section 4 builds on several examples, which are provided in the previous sections, to illustrate how three tasks can be solved with our approach. Given many open questions about implementation and long-term maintenance, we discuss our approach in section 5. We conclude our thoughts in section 6.

2 From Generic Objects to Domain Objects

In this section, we elaborate on the improved communication between domain experts and programmers resulting from bringing domain concepts to the software system. We also illustrate how integrating external data can impede this communication.

First, we describe the programmers' situation when working in an object-oriented system. We choose Squeak/Smalltalk as an example system because it has a clear implementation of the object-oriented paradigm. Second, we describe the role of generic objects, how they result from integrating external data, and which issues they cause regarding programming effort and source code readability. Third, we introduce domain objects as a major design goal for programmers when developing software. We give a simple example on how to derive domain objects from generic ones by writing custom classes. Finally, we summarize the main challenges for our approach.

2.1 Objects and Messages in Squeak/Smalltalk

Our approach focuses on challenges of object-oriented software development that occur in the Squeak/Smalltalk programming system [Goldberg1983]. These challenges and our proposed solutions are, however, not specific to the Squeak/Smalltalk environment and can easily be transferred to other object-oriented programming languages and systems. Squeak implements the object-oriented programming paradigm with a clear meta-model. Its programming tools provide short feedback loops, and the programmer can inspect and manipulate any part of the application anywhere in the user interface.

Squeak implements a clear object-oriented meta-model, as everything in Squeak is an object. The behavior of the system is defined mostly in terms of objects collaborating through messages. Classes are blueprints to construct objects and to describe the messages objects can understand. Hence, Squeak is a class-based, object-oriented system [Wegner1987, Wegner1990]. Still, classes are also objects and so are messages. This leads to a powerful meta-object protocol, where programs can be written that modify other programs—and even themselves. Based on a general understanding of object-oriented applications, the Squeak system includes objects for running programs (such as method, context, process), user interfaces (such as cursor, event, window, button), data processing (such as string, number, collection), and many more. However, the roles of objects may blur, depending on the programmer's current perspective on the system.

Programmers that write Smalltalk code benefit from short feedback loops in programming tools [Sandberg1988]. There are *no text files* to be modified, but only the method objects of a class object. The system browser supports navigation in the system's classes. It can show a text editor for one method in a class. If one method gets modified, it will directly be compiled and integrated into the class. All instances of that class will immediately show the new behavior if that new method is involved. When an object sends a message to another object that does not understand the message, an error occurs. Debugging the system means that the programmer has to figure out why the participating objects made this communication error.

In a Smalltalk system, run-time information is omnipresent. Programmers can type any expression into any text field and evaluate that code. This is especially convenient for objects constructed from literals, such as numbers and strings. For example, evaluating `3 + 4` yields `7`. Classes and other global variables can be referenced by just typing their identifier. For example, evaluating `Rectangle origin: 0@0 extent: 20@20` creates a new rectangle object. `Rectangle` is the global variable, `#origin:extent:` the message. `0@0` creates a point object by sending the message `#@` to a number (literal). However, not all objects can be referred to by their global name. Some objects have to be accessed in a context object. For example, all graphical things on the screen are called "Morphs". Programmers can just point to a morph with the mouse, open an inspector tool, and start sending messages to that object. Given such a context object in any tool, such as the debugger, the code expressions are enriched with *bindings*. Bindings map

identifying objects, typically strings or symbols, to other objects. Then, even the expression `(foo bar) baz` can work if there is a binding for `foo` and the bound object responds to the message `#bar` and if the object resulting from that call understands `#baz`.

2.2 Generic Objects

We think about an object being a *generic object* if it belongs to the Squeak base system and not to any particular application domain. Generic objects are often strings and numbers. There are also generic objects that represent a collection of objects or object structures. Examples for generic objects for object collections are ordered collections, sets, associations, and dictionaries. Dictionaries are a collection of associations, that are themselves key-value pairs. Thus, dictionary objects are like real-world dictionaries in the sense of a book for translating foreign languages that contain a foreign word as a key and the translation as the value. We can describe the domain concept of people with their friends like this:

```
| marcel patrick |
marcel := Dictionary newFrom: {
   #firstname -> 'Marcel'.
   #lastname -> 'Taeumel'.
   #email -> 'marcel.taeumel@hpi.de' }.
patrick := Dictionary newFrom: {
   #firstname -> 'Patrick'.
   #lastname -> 'Rein'.
   #email -> 'patrick.rein@hpi.de' }.
marcel at: #friends put:  { patrick }.
patrick at: #friends put: { marcel }.
```

In this example, we create two concrete dictionaries for two people, Marcel and Patrick, and establish the mutual friendship relation by setting each person's key `#friends` to a collection with a single object: the friend. Since it is a collection, there is room for more friends. Note that `marcel` and `patrick` are variables and the `:=` operator is the variable assignment. These names of the variables are the names under which the two dictionaries are known in this short code section. The `{...}` notation represents a collection of objects. The message `#->` creates an association. Every dictionary responds to the messages `#at:put:` and `#at:`, which modify and read the contents.

The problem with such generic objects is that (1) the syntax is verbose, (2) the source code includes identifiers that do not belong to the domain, and (3) behavior cannot be easily added to the object to define new terms in the vocabulary. In theory, it is possible to put anonymous methods as objects into the dictionary and evaluate them. However, such an approach would circumvent the idea of classes, instances,

and messages. It is hence discouraged to do so. Instead, programmers work with domain-specific objects like this:

```
| marcel patrick |
marcel := Person firstname: 'Marcel'
  lastname: 'Taeumel'.
marcel email: 'marcel.taeumel@hpi.de'.
patrick := Person firstname: 'Patrick'
  lastname: 'Rein'.
patrick email: 'patrick.rein@hpi.de'.
marcel addFriendMutually: patrick.
```

This creates instances of the class `Person`, which implements the concepts of first name, last name, email address, and the friendship relationship. In this case, the friendship seems to be mutual. Implementation details in the class `Person` should deal with establishing the mutual friendship, specifically by also adding Marcel as a friend of Patrick. Hence, the syntax is clearer and behavior can be added to the concept of a person—now represented as a domain-specific object by having its own class. On the downside, programmers have to create and describe that class.

Generic objects are used by libraries to provide some object-oriented representation when accessing data from outside the Squeak environment. Examples for accessing information outside the environment include file access and Web requests. For example, the *Web client* in Squeak can make an HTTP (Hypertext Transfer Protocol) request. It returns an instance of `WebResponse`, which is a more specific object, but still not specific to the contents it is trying to fetch:

```
| url response contentType content |
url := 'https://www.gravatar.com/avatar
         /16d12ad253109aa61366e44ea8ab395e'.
response := WebClient httpGet: url.
contentType := response contentType. "image/jpeg"
content := response content.         "some bytes"
```

Assuming that the programmers know that the request should deliver a *picture*, they still have to interpret the fairly generic response to create an instance of `Form`. This instance represents the concept of pictures in Squeak. That is, they must write further code that converts the generic object into a specific one. Only then can they use the picture for display on the screen. Given that the message `#displayAt:` shows a picture at the given coordinates on screen, they have to write the following:

```
| picture |
response contentType = 'image/jpeg'
  ifTrue: [picture := ImageReadWriter formFromStream:
             response content readStream.
           picture displayAt: 0@0.]
  ifFalse: [...].
```

This presents several challenges. First, programmers have to write conversion code for any new application whenever they want to access this kind of resource. Second, this code might clutter the domain-specific implementation of the surrounding object. Third, programmers must recall these mapping rules and predicates, causing the cognitive load to increase. Actually, the programmer might just want to write the following:

```
| picture |
picture := 'https://www.gravatar.com/avatar
             /16d12ad253109aa61366e44ea8ab395e'.
picture displayAt: 0@0.
```

The variable `picture` should emphasize that the identifier of such a resource could come from any generic object. We could have directly sent `#displayAt:` to the string.

Text can also be used to transfer complex structures between databases, across the Internet. The text-based JSON[2] (JavaScript object notation) format is a preferable solution. JSON looks, to some extent, like the Smalltalk syntax for generic objects shown above. This similarity makes it easy for Squeak to generate generic objects from a JSON string. It uses dictionaries, arrays, strings, numbers, and Booleans. Here is an excerpt response for a Web request to the API[3] of StackOverflow[4] which is a question-answering platform for programmers:

```
jsonString := '{"items":[
    {"answer_count":4, "title":"Help"},
    {"answer_count":50,"title":"Help more"}],
  "has_more":true}'.

jsonDictionary := Dictionary newFrom: {
  #items -> {
    Dictionary newFrom: {#'answer_count' -> 4.
      #title -> 'Help'}.
    Dictionary newFrom: {#'answer_count' -> 50.
      #title -> 'Help more'} }.
  #'has_more' -> true }.
```

Concrete strings and numbers provide only slight cues about the underlying domain. For example, the list of questions is behind the `#items` key in the `jsonDictionary`. The name of the underlying domain concept "questions", however, does not occur at all. If programmers want to write code that looks like concepts in the domain, they have to write new classes that describe these concepts.

---

[2] See http://json.org/

[3] The Web request was sent to the URL `http://api.stackexchange.com/2.2/questions?tagged=Squeak&site=stackoverflow`

[4] See https://www.stackoverflow.com

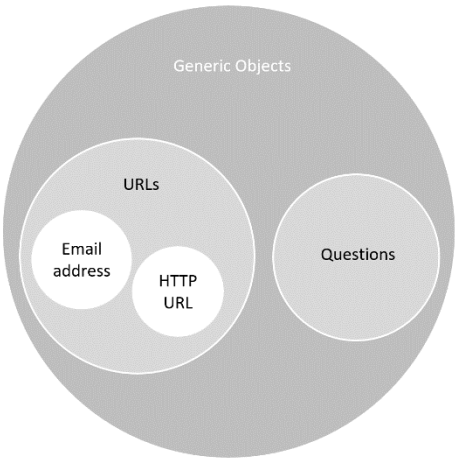

Fig. VennGenericToDomain. Objects can be ordered in a hierarchy from generic to domain-specific. Generic objects are not very specific and understand only generic messages. In contrast, an email address is a very specific object that can understand messages particular to email addresses. When integrating data from external databases into object-oriented systems it is necessary to provide a mapping from the generic objects to an object specific enough so it can understand the required messages.

2.3 Domain Objects

Domain objects have an interpretation specific to the domain for which the software system is created [Buschmann2007]. In Squeak, this means that programmers will create new classes that represent the domain concepts. Instances of these classes are then the domain objects. Existing classes in the Squeak system provide a high level of reuse and are agnostic to the domain in which they are used. For example, any chat tool, Web browser, or word processor benefits from text objects, picture objects, or button objects. Such generic objects become domain-specific only through their usage context and actual state they are holding. A text can represent the *manuscript* being written if it contains recognizable letters, words, and whole phrases. Still, programmers cannot always reuse generic classes but have to write custom classes to better reflect domain-specific concepts. There could be good reasons to write a class for `Manuscript`, which just *contains* (or *wraps*) a text object. It is not advisable to modify base classes because of interference with other applications. A custom class supports many degrees of freedom in describing any domain concept in source code. For example, manuscripts might not just be a large chunk of text but rather elaborate structures with sections and figures. A chat message, on the contrary, might not benefit from such extensions.

Programmers create domain objects from generic objects by either *wrapping* the generic objects or by *unpacking* them. If the generic object does not provide reusable behavior, one can just extract all of the state and map it to instance variables. After the object gets unpacked in this way, it is not useful anymore. If the generic object provides useful behavior, such as messages to derive new information, one should wrap the whole domain class around that object. The wrapping object can easily access state and behavior of the wrapped object as needed. Either way, the source code for constructing the domain object will usually be added to the respective class object as a *construction message* like `Url absoluteFromText: 'marcel.taeumel@hpi.de'`. Here, the `#absoluteFromText:` is a message used to create an instance of `Url` based on a generic string that looks like an email address. Here, the URL is considered a domain object and the string a generic object (see Figure VennGenericToDomain).

2.4 Domain Objects by Example

We want to write an application that manages *questions* and *answers* with a graphical user interface. As a starting point, we want to integrate StackOverflow, which is a Web-based system for the exchange of programming knowledge and experiences. StackOverflow has a website where programmers can ask questions about issues with specific languages, libraries, or systems. Fellow programmers provide answers and the community can rate all answers so that the whole database serves as a useful reference for any programmer that has similar problems. Hence, our object-oriented application should have objects for questions and objects for answers. In Squeak, this means that we will have a class `Question` and a class `Answer`. There is a set of questions and each question can have multiple answers.

We opt for unpacking the generic objects if feasible. The following steps are undertaken:

1. Perform a Web request to StackOverflow.
2. Create generic objects from the JSON part in the response.
3. Fill the domain objects with information. Discard the generic ones.
4. Repeat the steps until all objects and relationships are established.

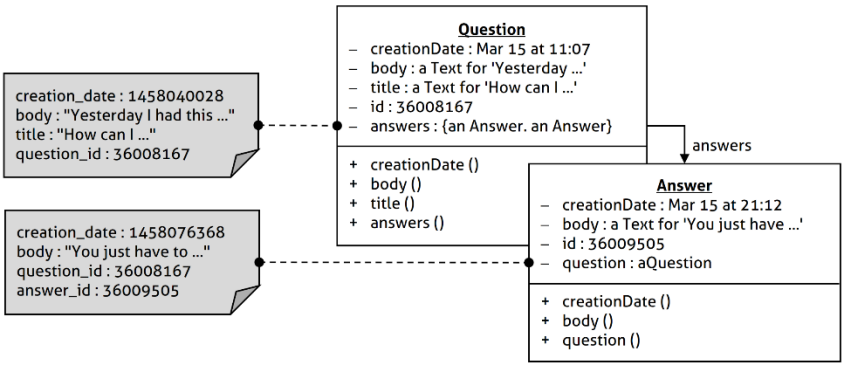

Fig. Conversion. An example for a mapping from generic information (left) to domain-specific objects (right). Note that the name of slots was converted from underscore style to came-case. The date was converted from an integer to a date object. The body was converted from a string to a text, which is a string with visual attributes such as color and weight. The relationship between question and answers was added from the outside with an additional Web request. Methods are merely accessors for the instance variables. (UML-flavored object diagram)

The conversion is illustrated in Figure Conversion. The following requests retrieve questions and answers:

- http://api.stackexchange.com/2.2/questions?tagged=squeak&site=stackoverflow
- http://api.stackexchange.com/2.2/questions/36008167?site=stackoverflow
- http://api.stackexchange.com/2.2/questions/36008167/answers?site=stackoverflow
- http://api.stackexchange.com/2.2/questions/36008167/answers/36009505?site=stackoverflow

The first request searches for multiple questions, the second fetches a single question, the third fetches all answers for a single question, the last fetches a single answer. Numbers are used to identify questions and answers. Numbers are also used to encode timestamps, such as the point of creation as the number of seconds elapsed since the beginning of 1970. Strings are used to hold a question's contents and an answer's contents. After the conversion, there are concrete objects for `DateAndTime`, `Text`, `Question`, and `Answer`. Note that underscore style was converted to camel-case to match Squeak's coding guidelines. Note that we did not have to create `DateAndTime` and `Text` because they are part of the base system.

Now that we have custom classes, we can add new behavior to the domain objects. For example, we can add the message `#isGood` if the answer got a high rating by the community. Any derived or computed information can be added this way. If programmers want to modify the domain objects, they have to write back to the external data source. We focus on exploring and understanding domain objects, which should be used in an interactive application.

All these steps are typically just the starting point. Programmers rewrite and improve source code regularly [Fowler1999]. If they recognize a way to better modularize pieces of code, they employ architectural patterns and design patterns. For example, they could put all source code related to querying StackOverflow into a specific class, which they could call `StackOverflowAccessor`. The mapping between generic objects and domain objects might happen in a `QuestionFactory`. Still, this example illustrates how much additional code programmers have to write to bring external data and domain objects to Squeak. All this additional code interferes with the goal of keeping the source code as an artifact that is understandable to both programmer and domain expert.

2.5 Challenges for Object Lookup in Object-Oriented Systems

The source code should look like a textual description of some domain model [Evans2004]. In the code, domain concepts should be discoverable by both programmers and domain experts. Only then is there a chance that code can be used as a tangible artifact to talk about the problem space and the solution space. Also, there is a reduced chance for misinterpretation when talking about possibilities and limitations in the software system. In an object-oriented system, such as Squeak/Smalltalk, programs consist of objects that communicate via messages. Reading source code means reading object names and message names, combined into sentences and whole paragraphs.

However, the traditional practices to get domain objects from databases impede both code reading and code writing. Programmers have to manually create domain classes, manage access of external sources, and transform generic objects into domain objects. Code reading is difficult because traces of the object conversions remain in the code such as in `(Url absoluteFromString: 'marcel.taeumel@hpi.de') authority`. The alternative `'marcel.taeumel@hpi.de' authority` would be more direct and more concise. Code writing is difficult because programmers have to be aware of existing techniques to convert generic objects. In Squeak, programmers have to know about, deliberately choose, and apply existing converter classes such as `ImageReadWriter` to read the binary data of a picture. Programmers must be careful not to simply reuse one conversion scenario in another situation. Building on the StackOverflow example above, programmers might want to interpret other numbers like `1458040028` as a data-and-time object, not only in the context of a question or answer object.

All these intermediate steps, also increase the semantic distance between the initial object and the object that will finally receive the message. The semantic distance is the number of operations we have to perform in order to achieve our goal. Each conversion step makes it harder to directly understand what a line of code

expresses. As a consequence, the impression of immediacy in programming can become worse, and this in turn hinders explorative programming [Ungar1997].

We are looking for a framework or mechanism that supports: (1) a concise description of the rules for resolving artifacts based on identifiers, (2) mapping any generic object into a domain specific one, and (3) triggers to promote clear source code that uses only vocabulary from the problem domain. Domain experts might be capable of reading and understanding that: `(someFramework shouldConvert: genericObjects) soThatEach: [:domainObject | domainObject worksIn: SoftwareSystem]`[5]. This would improve the communication between programmers and domain experts and eventually lead to an effective and efficient software system.

3 Our Approach: Implicit Object Lookup and Exchange

An object-oriented system consists of objects that communicate via messages. Sometimes the programmer intends to have another representation of the object to answer a given message. In some cases, the given object does not understand a certain message and it is vital to exchange the object with a more appropriate one. Otherwise, the execution would stop and the programmer would have to debug the system. So, if a string object *looks* like an email address (for example `'patrick.rein@hpi.de'`), it should be able to respond to the message `#authority`. However, in existing systems strings will not do that.

We describe our approach to add a novel means of object lookup and exchange to any object-oriented system. Our goal is to reduce programming effort and improve code readability, especially in the beginning of the exploration of a domain when the vocabulary between programmers and domain experts changes regularly. Our conceptual model is illustrated in the Figure Model. First, we explain the three different roles each object can take on, as well as the means to transition between these roles. Then, we elaborate on triggers and predicates which integrate these transitions into the ordinary system behavior. We elaborate on the object cache as means to manage object identity. Given the dynamic characteristics of the Squeak/Smalltalk system and explorative programming strategy when clarifying the system specifications, we also show the means to materialize the effective protocol for each object, which is a list of messages understood. Finally, we describe the impact of such an approach on source code readability.

[5] This is an example to show the possibilities for writing concise expressions in the Smalltalk language. Actually, this expression could be executed if there would be objects, such as `aFramework` and `SoftwareSystem` that understand the messages.

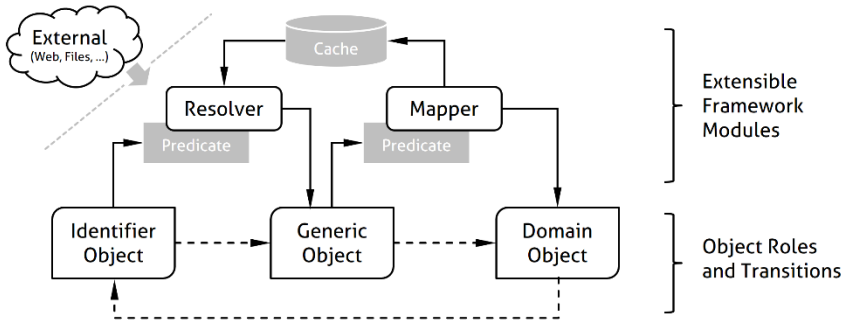

Fig. Model. We propose a mechanism that enriches existing classes with implicit instantiation. It works by implicitly resolving identifiers and mapping generic objects to domain-specific ones. Programmers can add new predicates, resolvers, and mappers to accommodate specific domains. Depending on the purpose, any object can take on the role of an identifier, generic object, or domain object.

3.1 Object Roles

Any object can take on one of three roles: *identifier object*, *generic object*, or *domain object*. Which role an object takes on depends on the context and the programmers' intent. For example, a URL object can be an identifier to be put into a resolver, which could make a Web request. That same URL object can also be the domain object after being mapped from a string that contains the same data. Consequently, it depends on the situation and the programmers' intent to determine whether one object is preferred over another.

In Squeak, *identifier objects* are usually strings, numbers, URL objects, or UUID objects. They can commonly be used to access additional information about the object they designate from an external database. For example, the text `'patrick.rein@hpi.de'` could be used to look up additional contact information on an address book server. Besides simple objects, one could also use any complex domain object, for example a person object in an address book application, and treat it as an identifier. Either the identity of the object itself, or just parts of its structure might be used to query the external database for additional information. For example, we might use a person object as an identifier for its corresponding social media profiles. For getting this profile from the Web, we only require the email address of the person. Thinking of the person as the identifier for the social media profile is likely to be closer to the underlying problem domain of the respective software system.

Generic objects can consist of more identifier objects or also containers for multiple objects. In Squeak, containers include sets, arrays, and dictionaries. Usually, these containers are agnostic to any particular domain. Still, these generic containers may sometimes be adjusted to domain-specific representations. For example, a set

of persons might be captured in a special `FriendSet` class to provide additional properties or behavior to specialize the friendship relationship between the friends. Dictionaries, on the other hand, are a means to structure multiple objects by some keys. This is similar to the way classes describe state in Squeak. However, dictionaries are primarily for storing and accessing structured data and we cannot easily add and invoke behavior on dictionaries. Further, dictionaries require more code for accessing their data in comparison to Smalltalk classes. Because of the cleaner syntax and the potential to add new state or behavior, domain-specific classes are preferable over containers.

Domain objects are the objects that represent the domain concepts. Programmers prefer working with such objects because the resulting source code is more readable. These objects understand domain-specific messages and store domain-specific state. For example, when a generic dictionary that represents an email understands only the `#at:` message, the domain object can respond to the specific `#authority` message. In Squeak, using domain objects can save many abstract messages. This improves code readability. As soon as new details are discovered in the problem domain, domain objects can be extended with new state and new behavior. In Squeak, all instances of a class get automatically updated if the respective class changes. For example, if you add a new message to the class, all instances will directly understand that message. If you add a new instance variable, all instances will have that, too.

With these three object roles, we try to encode the programmers' intents in different situations. We think that there are three situations where programmers usually access or convert objects:

1. "Oh, X is just the name. I have to somehow access the real data from a database."
2. "Hm, Y is only a plain dictionary. Useful but it does not understand the important messages. And the dictionary syntax in Smalltalk is kind of verbose."
3. "Ok, Z is useful concept, maybe I can use it in this other data source."

This is where our three object roles can extend the programmer’s conceptual model about objects.

3.2 Resolve and Map

We propose a system that can automatically look up an object in a database given another object that represents its name. We distinguish between *resolving* and *mapping*, where resolving refers to partially fetching data from outside the Squeak environment and mapping to converting the generic data to domain objects.

A resolver fetches information about a particular identifier from an external data source. This can be the file system, another application on the same computer, or an Internet resource. The resolution process can involve platform-specific details of how to connect to the database, as well as application-specific details of how to

correctly query for the desired information. A basic resolver might be one for HTTP requests that can handle various content types such as JSON, XML, or image data.

A mapper converts any generic object into a domain object. Considering the Squeak base system, there can be mappers that can instantiate picture objects, sound objects, or others based on raw data such as strings or byte arrays. For any new concept, programmers have to create or extend existing classes.

Both resolvers and mappers might be applicable in the same situation. Given a response from a Web request, for example, it is not obvious whether a resolver or a mapper should take care of making the first conversion after fetching the information. This flexibility allows the programmer to treat the outcome either as a generic object or as a domain object. The responsibilities of the two mechanisms also becomes clear when considering their operations on a shared *cache*. The cache should be used to skip object resolving and mapping so when an identifier should be resolved the system can directly return the resulting domain object. Only resolvers read the cache and only mappers write into it.

3.3 Triggers and Predicates

At best, our mechanism can be *triggered* implicitly whenever an object cannot understand a certain message *and* the programmers' intentions can be met. For example, resolving and mapping of things should enable the string `'marcel.taeumel@hpi.de'` to understand the message `#authority`. Later as the system evolves programmers might want to explicitly invoke our mechanism to make it more predictable. If the communication with the domain expert is not so frequent anymore, one might sacrifice code readability for the benefit of long term maintainability.

If there are many resolvers and many mappers, there will be no value in using every one of them. That is why we also propose *predicates* to support selecting appropriate and capable resolvers or mappers. For an email address stored in a string, a predicate might be a regular expression that verifies the structure: `'([a-zA-Z0-9.]+)@([a-zA-Z0-9\-.]+)'`. Only when a string matches this pattern, the resolver or mapper can be applied to it.

For complex scenarios, the process can be supported by a dialog between the system and the user. If, for example, an identifier does not match any resolvers yet, the user could be presented with a selection of existing resolvers to choose from. Such a dialog with the user makes sense if resolving or lookup is part of an interactive application. If an application runs without an interactive user interface however, the system should always continue without user interaction.

For programming tools, the user would be a programmer. For example, if the programmer navigates a data structure across several databases, the dialog can also be used to choose the database where a certain identifier should be resolved.

There might be several rounds of resolve and map until the desired domain object can be made available. If an object does not understand a message, the resolve-map-

cycle continues until at least one object is found that responds to the message. This cycle might also involve several interactive user dialogs. In this context, the resolve-map-cycle can be understood as a planning problem as described in the artificial intelligence domain [Russel2003]. The planning goal in this case would be that the object understands the required message. The planning operations are the resolvers and mappers.

3.4 Object Cache

It can be comparably slow to access external resources to bring them into the object-oriented world. A cache can be used to avoid making the same external requests over and over again. More importantly, such a cache allows for managing multiple object identities and names.

The object cache represents a table to map an identity object to a domain object. Given some identifier to resolve, the resolver first tries to look up that identifier in the cache. On a cache hit, the external request is skipped and maybe also the mapping between the generic and the domain object. There can be additional resolve-map cycles, depending on the current use case.

Programmers should not have to actively manage the contents of the cache. Our conceptual model primarily comprises the three object roles as well as means to resolve and map objects with the help of predicates. There can be applications where it is not useful to manage multiple object identities. In that case, the object cache could be disabled.

3.5 Tool Support and Virtual Object Protocol

Programmers should have a good understanding of which objects make up their application. If a piece of source code is too abstract, they can set a breakpoint, run the application up to that point, and inspect run-time state and concrete objects. In Squeak, programmers can evaluate any little piece of text and explore useful results.

With the introduction of our mechanism, programmers need new tool support to explore interactions between objects. Because our mechanism might exchange the original receiver of a message, programmers cannot rely on knowing the actual receiver of a message send. As an object could be replaced by a different representation for each message send, programmers can also not be sure anymore which messages an object understands. The set of messages an object understands is also called the *protocol* of the object. With our approach an object would also have a *virtual protocol* which is the set of messages the object would understand if it was processed by all applicable resolvers and mappers.

We think that it is feasible to ask resolvers and mappers of prospective actions or capabilities. Without actually resolving or mapping an object, the programmer

could be informed about the new messages that the object can understand. The string containing an email address, for example, could be enriched with `#authority` if some mapper acknowledges the capability of creating URLs from strings.

Tracing multiple object conversions for a single purpose is also beneficial for making sequences of conversions tangible. Further, for a new domain, programmers are likely to add new resolvers and mappers. Using the example with StackOverflow mentioned above, the connection between answers and questions cannot be derived from a generic resolver for HTTP URLs. There is additional knowledge required that has to be described in the form of a new resolver. If you send `#answers` to a question, a resolver's predicate should check for the prospective message receiver, recognize the domain "StackOverflow" and form an appropriate Web request. Consequently, programmers want to debug resolve-map cycles and check, whether their new resolvers or mappers behave as expected.

3.6 Extensibility and Readability

Our approach will work best if there already are some resolvers and mappers that help to acquire default resources by opening files or making Web requests. For a new domain-specific resource, there is now a place for programmers to describe access and mapping to the object-oriented world. The level of reuse compares with any other modularity mechanism in the Squeak environment. Programmers can specialize existing resolvers or mappers. They can also add additional predicates to existing resolvers or mappers.

By moving the source code away from the domain classes to classes for custom resolvers and mappers, programmers can write source code that directly reflects domain concepts. Even non-programming domain experts might be able to understand it and help express their actual requirements to be fulfilled in a software system.

For a new project, we think that there will an increased need for our mechanism. If the project's specifications mature, programmers are likely to move resolving and mapping code to a place where they have more direct control over it. We do not assume that there will be a very large number of resolvers or mappers for one project. But the available ones will have a large impact on productivity.

4 Scenarios

In this section, we describe several scenarios in which our approach can support programmers to quickly get access to domain objects. We first look at an example for a simple mapping from a local string to an object behaving like an email address. We then look at a scenario in which the string denotes a picture which can be fetched

from the Web. Finally, we discuss the mechanism for resolving objects with nested structures.

4.1 Simple: Create Email Address

Task. We have a set of strings that are email addresses. We want to convert them into instances of `Url`. An example string looks like this: `'patrick.rein@hpi.de'`. As specified in RFC 2822[6], the part after the `@` character is called *domain*. So, instead of `#authority` as in the examples above, we want to send the message `#domain`, which `Url` objects do not understand by default.

Resolver. We need a resolver that just looks up the provided string in the object cache. The predicate for this resolver is always true. It might, however, be restricted to not accept all kinds of identifier objects. Any more complex object such as pictures might not serve as an identifier for this resolver.

Mapper. We need a mapper that complements the construction methods in the `Url` class. The predicate is a regular expression: `'([a-zA-Z0-9.]+)@([a-zA-Z0-9\-.]+)'`. If that predicate matches, the mapper will create the URL object via `Url absoluteFromString: anObject`. The variable `anObject` is one sample in our list of email addresses. The resulting URL object will then be checked against the required message `#domain`. This means that the mappers must have access to the original message send, which is straightforward in the Squeak environment. If the message is not implemented, a template will be generated. The programmer will be asked to fill the template interactively, using the example object as a guidance. The mapper will store the new URL object in the object cache.

Summary. The first example works completely inside the Squeak environment. There is no access to the file system or the Internet. Since emails appear in many other domains, there is a high probability that it will be possible to reuse the resolver and the mapper in future tasks. The message `#domain` could be implemented automatically and could pass the call to `#authority`, which URLs already can understand. The option for an interactive dialog, however, renders the mapper usable also for other sorts of unknown messages.

4.2 External: Display a Picture from Gravatar

Task. We have a list of URLs pointing to Gravatar, which is a service to host recognizable profile pictures for people. The URLs look like this:

```
'https://www.gravatar.com/avatar
  /16d12ad253109aa61366e44ea8ab395e'.
```

[6] See https://www.ietf.org/rfc/rfc2822.txt

We want to display the pictures on the screen. We know that a `Form` in Squeak represents a displayable object. Such an object understands the message `#displayAt:`, given some screen coordinates. After performing one lookup cycle similar to the one described above with emails, we have a URL object to work with.

Resolver. We need a resolver that can fetch Web resources. It can be a very generic resolver accepting any kind of HTTP response and content type. The predicate for this resolver will have to check the HTTP schema in the URL object. The Web response contains binary data of some image format such as PNG or JPEG. It makes sense to return the Web response as the resolved object to be able to write a useful predicate for the mapper.

Mapper. We need a mapper that complements the construction methods in the `ImageReadWriter` class. It is basically a factory for `Form` objects with the capability to process various image formats such as PNG and JPEG. The predicate for this mapper checks the content type of the Web response for "image/*". Then, the body of the Web response will be fed into a stream to be processed like this: `ImageReadWriter formFromStream: webResponse contents readStream`. The mapper will store the new picture object in the object cache.

Summary. It will not always be obvious whether to put transformation rules into a resolver or a mapper. One of the influential factors is the context that the respective mapper and its predicates require. In Squeak, programmers can access many run-time information by employing introspection of the current message dispatch trace and other meta-programming facilities. Still, the resulting source code will be more readable if information exchange is made explicit.

4.3 Structure: Questions and Answers from StackOverflow

Task. We have a set of numbers that identifies questions on StackOverflow. We want to explore question data such as title and body as well was associated answers. Answers also have structured information such as body and rating. There are no classes for `Question` and `Answer` in the Squeak environment.

Resolver(s). We need a resolver that is able to complete the URL to the StackOverflow interface, given the identifying number. For example, `36008167` has to be converted to the following URL:

```
'http://api.stackexchange.com/2.2/questions/
  36008167/answers?site=stackoverflow'
```

The predicate might check the structure of the number or accept all integers. Then, the resolver makes the Web request or uses another existing resolver to do so. For answers, the resolver has to process the context of the Squeak message dispatch and look for signs of the respective domain concept. For example, if the message `#answers` was sent to an instance of `Question`, it will be obvious.

Mapper(s). We need a mapper that can process a Web response whose contents contain a JSON encoded string. After creating a generic dictionary from the JSON contents, the mapper has to build a class for the respective concept, `Question` or `Answer`. If the mapper cannot determine a good name for new classes, a dialog with the programmer should be established.

Summary. The interplay of multiple structured concepts, as here with questions and answers, poses a higher cognitive load to the programmers. They have to orchestrate a group of resolvers and mappers with predicates to bring the external information into the object-oriented system. New tools can help set up, maintain, and debug the resolver-mapper mechanism. Such tools should visualize the traces for predicate matching, resource resolving, and object mapping.

5 Discussion

In this section, we discuss the limitations and further implications of our approach on design decisions and maintenance.

Given a string with an email address, does the `URL` class or the `Email` class represent the domain concept?
This depends on the information required and the context in which the object is used. `Email` as a subclass of `URL` can add useful behavior and state. If the object is solely used to analyze where people have registered their email addresses, then `URL` is sufficient because it already responds to the message `#authority`. Conversely , if we want to send a letter to the email address, we need additional behavior such as `#send:`.

When does the system stop to do resolve-map cycles? Can there be an endless loop?
As each object can serve as an identifier, the object resulting from a mapper might itself match a resolver again. This can be controlled by keeping track of the current lookup with, for example, an identifier. Resolvers and mappers pass this identifier along with the objects to resolve or map. Then, they can count the cycle number and a maximum lookup depth can set a limit to avoid long or even endless cycles.

How is the object cache structured? Does it have a clean-up strategy?
The cache solely stores domain objects. Generic objects such as dictionaries are not used except if treated as domain objects in a mapper. Only the mapper writes into the cache, given an identifier object from the resolver in the respective lookup process. This way whenever an identifier needs be resolved and mapped, the resolver can directly return a domain object from the cache. Resolvers may have to store context along with identifier objects to make a domain object unique. For example, the string `'marcel.taeumel@hpi.de'` can be mapped to an instance of `URL` or `Email`. A least-recently-used (LRU) strategy can be used to manage the cache size. Otherwise, programmers have to account for manual cache clean-up.

Can the mechanism be used to write information back to the external database? Can we modify an object's state?
At the time of writing, our approach optimizes the retrieval and navigation of domain objects. If the lookup origins would be preserved, for example in the object cache, then there could also be a *writer* with custom predicates. The mapper might also have to deal with authentication protocols involved to write into the external databases.

Information might change outside of the current system and the objects representing them might not be up-to-date. How to get notified of updates?
Resolvers and mappers support accessing information stored outside of our current system, an approach that implies the challenge of synchronizing information stored in several places. Technically, this situation could be resolved by creating a central notification mechanism in the system which informs a resolver when a resource has changed. The resolver in turn can then fetch the new information. The mapper cannot simply create a new instance as there might already be a corresponding object. In this case, the mapper has to merge the incoming information with the information in the existing object. The notification mechanism inside the system could detect changes in external databases through polling or, if possible, via registering at an external event source.

If I want to build a Web browser, will a `URL` instance be an identifier, a generic object, or a domain object?
Whether an object is an identifier, a generic object, or a domain object depends on the context in which it is used. A Web browser retrieves and displays resources in the Internet. URLs are a standard for designating the location of such resources. Thus, they are an important concept for a Web browser and should be regarded as domain objects. For an application that displays StackOverflow answers, an URL, in contrast, is only a placeholder for other resources such as the profile picture of a user. The URL of the picture is not relevant for the user of the application. The fact that a URL represents the picture is only due to the technical implementation of the StackOverflow system and not inherent to the logic of a question-and-answer system.

The predicates of multiple resolvers or multiple mappers can match a single object. How can I manage such ambiguity?
If the ambiguity can be anticipated by users, they could pro-actively add filters to limit the applicable resolvers and mappers. Filters, however, would add another level of complexity to be managed by the programmer. If the ambiguity of the object cannot be anticipated, then the system itself could present users with all possible interpretations of the object and let them decide. The decision could be stored for future disambiguations.

What if there is no resolver for an identifier or no mapper for a generic object?
Traditionally, the system can indicate that an error occurred in the application. Alternatively, the user might be asked how to correctly interpret the identifier in this context. This might extend the predicates of the existing resolvers and mappers. The

user might then provide a small code snippet that resolves the issue for the current context. This snippet could be added to the existing set of resolvers and mappers. While most resolvers have to be written manually, mappers might also be generated automatically through techniques known from *ontology matching* [Euzenat2013].

If I am working with many objects that need to be resolved, is there a way to batch-process a set of identifiers?
When the resolvers are triggered explicitly, then the resolver might be able to resolve many identifiers at once. If, however, the resolving is part of mitigating a message which was not understood by the initial object, then it is not possible as the control flow depends on the resolution of this particular identifier.

This mechanism seems to impede maintenance and debugging to a great extent. Is there a way to reduce the level of automatic resolving once the specifications are clearer and corresponding classes exist?
It is possible to migrate to a semi-automatic approach, once the projects requirements become clearer and more stable. Programmers can extract knowledge from the resolvers and mappers and move them into an extra module. They have to rewrite the code which triggered the resolvers and mappers. The new code would execute the resolution as described in the resolvers and mappers most often used in this context. Thus, the interpretation of the identifier object becomes fixed and documented in the source code again. The resulting code's readability might be sacrificed to some extent. This might be sufficient if domain experts are not as involved as in the beginning of the project.

6 Conclusion

We presented an approach to improve the means for quickly and conveniently working with domain objects inside an object-oriented environment, while domain data resides in outside databases. We support programmers to implicitly or explicitly resolve and map identifiers to full domain objects. Additional source code for accessing and integrating external information is separated from the domain logic, which improves readability. Especially in the beginning of a software project, programmers and domain experts benefit from knowledge exchange on a regular basis. We think that our approach might make this exchange more likely to include source code as a tangible artifact in these discussions. With the code expressed in terms of the shared vocabulary, design decisions remain comprehensible, even for non-programmers. As a result, our approach can improve the collaboration between domain experts and programmers to indicate limitations and to reveal future possibilities of a system.

References

- **[Evans2004]** E. Evans, Domain-driven Design, Addison-Wesley Professional, 2004
- **[Euzenat2013]** J. Euzenat, P. Shvaiko, Ontology Matching (Volume 2), Springer-Verlag Berlin Heidelberg, 2013
- **[Russel2003]** S. Russel and P. Norvig, Artificial Intelligence: A Modern Approach, Pearson Education, Inc., 2003.
- **[Buschmann2007]** F. Buschmann, K. Henney, and D. Schmidt. Pattern-oriented software architecture, on patterns and pattern languages. Volume 4. John Wiley & Sons, 2007.
- **[Wegner1987]** P. Wegner, Dimensions of object-based language design, in Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, 1987, pp. 168–182.
- **[Wegner1990]** P. Wegner, Concepts and Paradigms of Object-oriented Programming, in SIGPLAN OOPS Mess. 1, 1 (August 1990), 7-87.
- **[Goldberg1983]** A. Goldberg and D. Robson, Smalltalk-80: The Language and its Implementation. Addison-Wesley Longman Publishing Co., Inc., 1983.
- **[Papakonstantinou1995]** Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources, in Proceedings of the 11th International Conference on Data Engineering, 1995, pp. 251-260.
- **[Meyer1997]** B. Meyer, Object-oriented Software Construction, Second Edition. Prentice Hall International, 1997.
- **[Kay1996]** A. C. Kay. The early history of Smalltalk. In History of programming languages---II, ACM, New York, NY, USA 511-598, 1996.
- **[Sandberg1988]** D. W. Sandberg, Smalltalk and exploratory programming, ACM Sigplan Notices, vol. 23, no. 10, pp. 85–92, 1988.
- **[Fowler1999]** M. Fowler, Refactoring - Improving the Design of Existing Code. Addison Wesley object technology series, Addison-Wesley 1999
- **[Beck2000]** K. Beck. Extreme programming explained: embrace change. Addison-Wesley Professional, 2000.
- **[Ungar1997]** D. Ungar and H. Lieberman and C. Fry. Debugging and the Experience of Immediacy. Communications of the ACM, ACM, 1997, 40, 38-43