Seuss: Better Class Responsibilities through Language-Based Dependency Injection

Niko Schwarz, Mircea Lungu, and Oscar Nierstrasz

University of Bern

Abstract. Unit testing is often made more difficult by the heavy use of classes as namespaces and the proliferation of static methods to encapsulate configuration code. We have analyzed the use of 120 static methods from 96 projects by categorizing them according to their responsibilities. We find that most static methods support a hodgepodge of mixed responsibilities, held together only by their common need to be globally visible. Tight coupling between instances and their classes breaks encapsulation, and, together with the global visibility of static methods, complicates testing. By making dependency injection a feature of the programming language, we can get rid of static methods altogether. We employ the following semantic changes: (1) Replace every occurrence of a global with an access to an instance variable; (2) Let that instance variable be automatically injected into the object when it is instantiated. We present Seuss, a prototype that implements this change of semantics in Smalltalk. We show how Seuss eliminates the need to use class methods for non-reflective purposes, reduces the need for creational design patterns such as Abstract Factory and simplifies configuration code, particularly for unit tests.

1   Introduction

Class methods, which are statically associated to classes rather than instances, are a popular mechanism in object-oriented design. Java and C#, for example, provide static methods, and Smalltalk provides "class-side" methods, methods understood by classes, rather than their instances. 9 of the 10 most popular programming languages listed by TIOBE provide some form of static methods.[1] In most of these languages, classes offer the key mechanism for defining namespaces. For this reason, static methods offer a convenient mechanism for defining globally visible services, such as instance creation methods. As a consequence, static methods end up being used in practice wherever globally visible services are needed.

Unfortunately this common practice leads callers of static methods to implicitly depend on the classes that provide these static methods. The implicit

---

[1] TIOBE Programming Community Index for January 2011, `http://www.tiobe.com`. Those 10 languages are Java, C, C++, PHP, Python, C#, (Visual) Basic, Objective-C, Perl, Ruby. The outlier is C, which does not have a class system.

dependency on static methods complicates testing. That is because many tests require that application behavior be simulated by a fixed script representing a predefined scenario. Such scripted behavior can hardly be plugged in from the outside when static methods are accessed by global names, and thus hard-wired into code. We therefore need to better understand the need for static methods in the first place.

Classes are known to have both meta-level and base-level responsibilities [2]. To see what those are, we examined 120 static methods, chosen at random from SqueakSource, a public repository of open source Smalltalk projects. We found that while nearly all static methods inherited from the system are reflective in nature, only few of the user-supplied methods are. Users never use static methods to define reflective functionality.

Dependency injection is a design pattern that shifts the responsibility of resolving dependencies to a dedicated dependency *injector* that knows which dependent objects to inject into application code [6,11]. Dependency injection offers a partial solution to our problem, by offering an elegant way to plug in either the new objects taking over the responsibilities of static methods, or others required for testing purposes. Dependency injection however introduces syntactic clutter that can make code harder to understand and maintain.

We propose to regain program modularity while maintaining code readability by introducing dependency injection as a language feature. *Seuss* is a prototype of our approach, implemented by adapting the semantics of the host language. Seuss eliminates the need to abuse static methods by offering dependency injection as an alternative to using classes as namespaces for static services. Seuss integrates dependency injection into an object-oriented language by introducing the following two semantic changes:

1. Replace every occurrence of a global with an access to an instance variable;
2. Let that instance variable be automatically injected into the object at instantiation time.

Seuss cleans up class responsibilities by reserving the use of static methods for reflective purposes. Furthermore, Seuss simplifies code responsible for configuration tasks. In particular, code that is hard to test (due to implicit dependencies) becomes testable. Design patterns related to configuration, such as the Abstract Factory pattern, which has been demonstrated to be detrimental to API usability [5], become unnecessary.

Structure of the article. In section 2 we analyze the responsibilities of static methods and establish the challenges for reassigning them to suitable objects. In section 3 we demonstrate how Seuss leads to cleaner allocation of responsibilities of static methods, while better supporting the development of tests. In section 4 we show how some creational design patterns in general and the Abstract Factory design in particular are better implemented using Seuss. In section 5 we go into more details regarding the implementation of Seuss. In section 6 we discuss the challenges for statically-typed languages, and we summarize issues of performance and human factors. In section 7 we summarize the related work and we conclude in section 8.

2   Understanding Class Responsibilities

Static methods, by being associated to globally visible class names, hard-wire services to application code in ways that interfere with the ability to write tests. To determine whether these responsibilities can be shifted to objects, thus enabling their substitution at run-time, in subsection 2.1 we first analyze the responsibilities static methods bear in practice. Then in subsection 2.2 we pose the challenges facing us for a better approach.

2.1   Identifying Responsibilities

We follow Wirfs-Brock and Wilkerson's [4] suggestion and ask what the current responsibilities of static methods are, for that will tell us what the new classes should be.

We determine the responsibilities following a study design by Ko *et al.* [8]. Their study identifies six learning impediments by categorizing insurmountable barriers encountered by test subjects. The authors of the paper independently categorize the impediments and attain 94% agreement.

We examined 120 static methods and classified their responsibilities from a user's point of view. For example, a static method that provides access to a tool bar icon would be categorized as providing access to a resource, regardless of how it produced or obtained that image. We chose 95 projects uniformly at random from SqueakSource[2], the largest open source repository for Smalltalk projects. We then selected uniformly at random one static method from the latest version of each of these projects. To avoid biasing our analysis against framework code, we then added 25 static methods selected uniformly at random from the standard library of Pharo Smalltalk[3], as shipped in the development environment for developers.

Of the 120 methods selected, two were empty. We randomly chose another two methods from SqueakSource to replace them. Two subjects then categorized the 120 methods independently into the categories, achieving 83% agreement. We then reviewed the methods that were not agreed upon. Most were due to lack of knowledge of the exact inner workings of the API they were taken from. After further review, we placed them into the most appropriate subcategory.

We identified the following three umbrella categories: *Instance creation*, *Service* and *Reflection*, each further subdivided into subcategories. Whenever a method did not fit into any of the subcategories, we marked it as "other".

Instance creation. (28 of 120) Instance creation methods create new instances of their own class. They are subdivided as follows.

Singleton. (4 of 28) These methods implement the singleton pattern [7] to ensure that the instance is created only once.

Other. (24 of 28) Some methods provided default parameters, some simply relayed the method parameters into setters of the newly created instance.

Only 3 methods did anything more than setting a default value or relaying parameters. These three methods each performed simple computations on the input parameters, such as converting from minutes to seconds, each no longer than a single line of code.

Services. (86 of 120) Service methods provide globally available functionality. They often serve as entry points to an API. We have identified the following sub-categories.

Install/uninstall a resource. (6 of 86) By resource, we mean a widely used object that other parts of the system need to function. Examples of installable resources that we encountered are: packages of code; fonts; entries to menus in the user interface.

Access a resource or setting. (41 of 86) These methods grant access to a resource or a specific setting in a configuration. Complex settings resemble resources, hence one cannot easily distinguish between the two. Examples include: a status object for an application; the packet size of headers in network traffic; default CSS classes for widgets; a sample XML file needed to test a parser; the default lifetime of a connection; the color of a GUI widget.

Display to/prompt user. (4 of 86) Examples: showing the recent changes in a versioning system; opening a graphical editor.

Access network. (2 of 86) These methods grant access to the network. Examples: sending an HTTP put request; sending a DAV delete request.

System initialization. (11 of 86) These methods set the system status to be ready for future interactions. Examples: setting operation codes; setting the positions for figures; asking other system parts to commence initialization.

Class indirection. (5 of 86) These return a class, or a group of classes, to provide some indirection for which class or classes to use.

Other. (17 of 86) Other responsibilities included: converting objects from one class to another; taking a screenshot; sorting an array; granting access to files; starting a process; mapping roles to privileges; signaling failure and mailing all packages in a database.

Reflection. (6 of 120) Unlike methods that offer services, reflective methods on a class are by their nature tightly coupled to instances of the class. We have found the following sub-categories.

Class Annotations. (5 of 6) Class annotations specify the semantics of fields of their class. All the examples we examined were annotations interpreted by Magritte [12], a framework for adapting an applications model and meta-model at run-time.

Other. (1 of 6) One method provided an example on how to use the API.

2.2   Challenges

Out of the 120 static methods we have analyzed, only 6 belonged naturally and directly to the instances of that class, namely the reflective ones. All other responsibilities can be implemented in instance methods of objects tailored to these responsibilities.

We conclude that static methods are defined in application code purely as a matter of convenience to exploit the fact that class names are globally known. Nothing prevents us from shifting the responsibilities of non-reflective static methods to regular application objects, aside from the loss of this syntactic convenience. In summary the challenges facing us are:

- to shift static methods to be instance responsibilities,
- while avoiding additional syntactic clutter, and
- enabling easy substitution of these new instances to support testing.

In the following we show how Seuss, our dependency injection framework allows us to address these challenges.

3   Seuss: Moving Services to the Instance Side

We would like to turn misplaced static methods into regular instance methods, while avoiding the syntactic clutter of creating, initializing and passing around these instances. Dependency injection turns out to be a useful design pattern to solve this problem, but introduces some syntactic clutter of its own. We therefore propose to support dependency injection *as a language feature*, thus maintaining the superficial simplicity of global variables but without the disadvantages. Dependency injection furthermore shifts the responsibility of injecting dependent variables to a dedicated *injector*, thus enabling the injection of objects needed for testing purposes. Let us illustrate dependency injection in an example.

In the active record design pattern [6, p. 160 ff], objects know how to store themselves into the database. In the SandstoneDB implementation of active record for Smalltalk [9] a Person object can save itself into the database as in Figure 1.

The code of the save method is illustrated in Figure 2. (The actual method is slightly more complicated due to the need to handle further special cases.)

The save method returns the result of evaluating a block of code in a critical section (self critical: [...]). It first evaluates some "before" code, then either stores or updates the state of the object in the database, depending on whether it has previously been saved or not. Finally it evaluates the "after" code.

In the save method, the database must somehow be referenced. If the database were an ordinary instance variable that has to be passed during instance creation, the code for creating Person objects would become cluttered. The conventional workaround is to introduce static methods storeObject: and updateObject: to encapsulate the responsibility of connecting to the database, thus exploiting the global nature of the Store class name, while abusing the mechanism of static methods for non-reflective purposes.

```
user := Person firstName: 'Ramon' lastName: 'Leon'.
user save.
```

Fig. 1. Using the active record pattern in SandstoneDB

```
save
    ↑ self critical: [
        self onBeforeSave.
        isFirstSave
            ifTrue: [Store storeObject: self]
            ifFalse: [Store updateObject: self].
        self onAfterSave.
    ]
```

Fig. 2. The save method in SandstoneDB, without dependency injection

Unfortunately, testing the save method now becomes problematic because the database to be used is hard-wired in static methods of the Store class. There is no easy way to plug in a mock object [10] that simulates the behavior of the database for testing purposes.

The dependency injection design pattern offers a way out by turning globals into instance variables that are automatically assigned at the point of instantiation. We add a method to Person that declares that Person is interested to receive a Store as an instance variable during instance creation by the runtime environment, rather than by the caller, as seen in Figure 3. Afterwards, instead of accessing the global Store (in upper case), save is re-written to access instance variable store (in lower case; see Figure 4).

```
store: anObject
    <inject: #Store>
    store := anObject
```

Fig. 3. Person declares that a Store should be injected upon creation

In the example in Figure 4, we also see that Person does not ask specifically for an instance of a class Store. It only declares that it wants something injected that is labelled #Store. This indirection is beneficial for testing. Method storeObject: may pollute the database if called on a real database object. Provided that there is a mock class TestStore, we can now inject instances of that class rather than real database objects in the context of unit tests.

Avoiding cluttered code by language alteration. The dependency injection pattern introduces a certain amount of clutter itself, since it requires classes to be written in an idiomatic way to support injection. This clutter manifests itself in terms of special constructors to accept injected objects, and factories responsible for creating the injected objects. Seuss avoids this clutter by incorporating dependency injection as a language feature. As a consequence, the application

```
save
   ↑ self critical: [
      self onBeforeSave.
      isFirstSave
         ifTrue: [store storeObject: self]
         ifFalse: [store updateObject: self].
      self onAfterSave.
   ]
```

Fig. 4. The save method from SandstoneDB rewritten to use dependency injection does not access the globally visible class name Store

developer may actually write the code as it is shown in Figure 2. The semantics of the host language are altered so that the code is interpreted as shown in Figure 4.

In Seuss, what is injected is defined in configuration objects, which are created in code, rather than in external configuration files. Therefore, we can cheaply provide configurations tailored for specific unit tests. Figure 5 illustrates how a unit test can now test the save method without causing side effects. The code implies that the storeObject: and updateObject: methods are defined on the instance side of the TestStore class.

```
testing := Configuration bind: [ :conf | conf bind: #Store to: TestStore new].
user := (Injector forConfiguration: testing get: #User).

user firstName: 'Ramon' lastName: 'Leon'.
user save.
```

Fig. 5. Unit test using dependency injection. The injector interprets the configuration, and fills all dependencies into user, including the TestStore

Typically, a developer using dependency injection has to explicitly call only one injector per unit test, and only one for the rest of the application, even though the injector is active during every object instantiation. Section 5 details how the injector is implicitly made available.

4   Cleaning Up Instance Creation

The design patterns by Gamma *et al.* are often ways of addressing language limitations. It is not surprising that by introducing a language change as powerful as dependency injection some of the design patterns will become obsolete. A special class of design patterns that we care about in this section are the creational ones, since we have seen in subsection 2.1 that a considerable percentage of static methods are responsible for instance creation.

The abstract factory pattern has been shown to frequently dumbfound users of APIs that make use of it [5]. Gamma defines the intent of the abstract factory pattern as to "provide an interface for creating families of related or dependent

objects without specifying their concrete classes" [7]. Gamma gives the example of a user interface toolkit that supports multiple look and feel standards. The abstract factory pattern then enables code to be written that creates a user interface agnostic to the precise toolkit in use.

Let us suppose the existence of two frameworks A and B, each with implementations of an abstract class Window, named AWindow and BWindow, and the same for buttons. Following the abstract factory pattern, this is how we could create a window with a button that prints "World!" when pressed:

```
createWindow: aFactory
    window := (aFactory make: #Window) size: 100 @ 50.
    button := (aFactory make: #Button) title: 'Hello'.
    button onClick: [Transcript show: 'World']. window add: button.
```

Fig. 6. Object creation with Abstract Factory

Ellis *et al.* [5] show that using this pattern dumbfounds users. When presented with the challenge of instantiating an instance that is provided by a factory, they do not find the required factory. In Seuss, the following code snippet may generate a window either using framework A or B, depending on the configuration, with no need to find (or even write) a factory:

```
createWindow
    window := Window size: 100 @ 50.
    button := Button title: 'Hello'.
    button onClick: [Transcript show: 'World'].window add: button.
```

Fig. 7. Replacing object creation with Dependency Injection

Seuss allows writing natural code that still bears all the flexibility needed to exchange the underlying framework. It can be used even on code that was not written with the intention of allowing the change of the user interface framework.

5   Dependency Injection as a Language Feature

Normally, using dependency injection frameworks requires intrusively modifying the way code is written. The developer needs to make the following modifications to the code:

- Add the definition of an instance variable.
- Specify through an annotation which instance variable gets injected (the inject annotation from Figure 3).
- Provide a method through which the dependency injection framework can set the instance variable to the value of the injected object. This is a setter method in Smalltalk (Figure 3) or a dedicated constructor in Java.

To improve usability, in Seuss we completely remove the requirement of modifying the code in any of the previously mentioned ways. As a result, the code in in Figure 2 is interpreted just as if the code in Figure 4 and Figure 3 had been written.

The feature that allows us to use dependency injection without the invasive modification of source code is a slight change to the Smalltalk language: for every global being accessed, the access is redirected to an instance variable. This instance variable is annotated for injection, made accessible through setters, and then is set by the framework *when the object is created*.

It is not enough to store an object representing the original class in an instance variable. That is because the class usually is not aware of Seuss and thus does not inject dependencies into objects it newly creates.
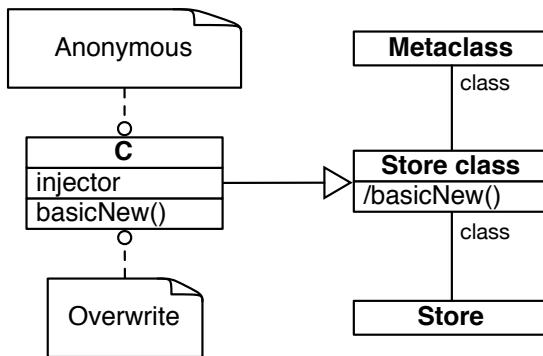

Fig. 8. Instances of C mimic Store, but use the injector when creating instances

Instead, we inject an object that knows the injector and calls it during instance creation. We achieve this by injecting an *instantiator object*. The class of the instantiator is an anonymous subclass of the metaclass of the original method's class. For example, in Figure 3 the object that is injected into instance variable store in is an instance of an anonymous metaclass C. As illustrated in Figure 8, C overwrites method basicNew which is inherited from Store class[4]. It changes basicNew so that it first invokes the injector, asking it to inject all dependencies into the newly created object, and then resets the class of the newly created object to be Store.

In order to change the semantics of a standard Pharo as described above, we use Helvetia [13], a language workbench for Smalltalk. Helvetia lets us intercept the compilation of every individual method. Helvetia requires us to specify our language change as a *Rule*, which is really a transformation from one method AST to another. When changing methods, we also modify the containing class when needed. During the transformation, we also create and update a default

---

[4] basicNew is a primitive that allocates memory for the new object. It is normally not overridden.

configuration, which lets the code run as before, if used. It can also be overridden by the user in unit tests. Algorithm 1 details the transformation.

Algorithm 1. Transforming ordinary code into dependency injected code.

---

1. Replace every occurrence of a global with an access to an instance variable. Add that instance variable if necessary.
2. Generate a setter method for that variable and annotate it so that the dependency injection framework can inject into that variable.
3. If the injected global is a class, act as follows. Generate an anonymous metaclass C as described above, and make its instance known to the default configuration. As described above, the instance should behave just like the original class, but should additionally inject all dependencies into newly created instances of class C.
4. Make the default configuration aware of the referred to global.

Introducing dependency injection as a language feature brings two advantages:

1. *Backwards compatibility.* Dependency injection can be used for code that was not written with dependency injection in mind. We were able to use the unit test from Figure 5 without having to modify the SandstoneDB project, which does not use dependency injection.
2. *Less Effort.* Other frameworks require that all dependencies be explicitly declared through some boilerplate code for each dependency. In our case, by automatically injecting needed dependencies where possible, the amount of code to write was reduced.

6   Discussion

We briefly explore the challenges for implementing Seuss in statically-typed languages like Java, and we summarize issues of performance and human factors.

6.1   Challenges for Statically Typed Languages

In a language where classes are reified as first-class objects, such as Smalltalk, classes can simply be injected as objects. In other languages, such as Java, a proxy must be used.

Seuss works by replacing access to globals by access to instance variables. In a statically typed language, the question arises what type injected instance variables ought to be. To see if our small language change would be feasible in a typed language, we ported part of Seuss to Java. In the following transformation by JSeuss, our Java version of Seuss, the access to the global Store is replaced by an instance variable store (note the lower case initial letter) of type ICStore.

```
class Before {
    void save() {
        Store.storeObject(this);
    }
}
```

is transformed into

```
class After {
    @Inject
    ICStore store;
    void save() {
        store.storeObject(this);
    }
}
```

The interface ICStore is a generated interface. Our Java transformation generates two interfaces for every class, one for all static methods, and one for all instance methods. The interfaces carry the same name as the class, except for the prefixed upper-case letters IC, or I, respectively. During class load time, all occurrences of type Store are then replaced by type ICStore, and so with all classes. All new calls on Store return instances of type IStore. On the other hand, existing interfaces are not touched.

The object of type ICStore serves as a proxy for the class ICStore. This is necessary since classes are not first class in Java, and thus cannot be injected directly. To avoid expensive recompilation, we use Javassist to modify all code at the bytecode level, during class load time.

The current implementation of JSeuss enables unit testing of the save method above, but is otherwise incomplete, thus currently prohibits meaningful benchmarking. We nevertheless learned from the experience that while Seuss for Java is complicated by the static type system of Java, it is still feasible.

6.2 Performance and Human Factors

Seuss impedes the performance of applications exclusively during object instantiation when there is some performance penalty for injecting all dependencies. In all other cases, a pointer to a global is replaced by a pointer to an instance variable, which is not slower than accessing a global in many languages, although it can prohibit inlining. Since every access to a global requires a new instance variable to be added, the memory footprint can grow considerably. However, space penalties can be ameliorated by introducing nested classes to a language, as demonstrated in Newspeak [3]. This should also improve performance during instantiation time, as dependencies can be moved to outer classes and thus need to be injected fewer times.

One might also argue that the new level of indirection may lead to confusion as to which object is being referred to, when an injected variable is referenced.

However, we believe that proper tool support can bring sufficient clarity. An IDE should be able to gather all configurations and use them to display which literals are bound to what.

6.3   Using Seuss to Sandbox Code

If Object's reflective methods are removed, then all objects can only find other classes through their dependencies or method parameters. Thus, any piece of code from within a configuration that does not include access to the File class prevents that code from reading or writing files. This concept of security by unreachability was described by Bracha [3].

7   Related Work

Dependency injection [6,11] is a design pattern that decouples highly dependent objects. Using it involves avoiding built-in methods for object construction, handing it off to framework code instead. It enables testing of components that would ordinarily be hard to test due to side-effects that would be intolerable in unit tests. There are other frameworks that support dependency injection like Google Guice [14] and Spring, after which Seuss's dependency injection capabilities are modeled. In contrast to Google Guice and Spring, Seuss turns dependency injection into a language feature that works even on code that was not written with dependency injection in mind. By superficially allowing the use of standard language constructs for object creation while using dependency injection under the hood, Seuss programs look in large parts like conventional source code.

Achermann and Nierstrasz [1] note that inflexible namespaces can lead to name clashes and inflexibilities. They propose making namespaces an explicit feature of the language and present a language named Piccola. Piccola does not get rid of using global namespace, but makes it a first-class entity. First-class namespaces in Piccola enable a fine degree of control over the binding of names to services, and in particular make it easy to run code within a sandbox. While Seuss sets the namespace of an object at that object's instantiation time, Piccola allows it to be manipulated in the scope of an execution (dynamically) as well as statically. Similarly, some mocking frameworks, such as PowerMock[5], allow re-writing of all accesses to global namespace to access a mock object. Piccola and PowerMock do not attempt to clean up static method responsibilities, but rather add flexibility to their lookup.

Bracha presents the Newspeak programming language [3], which sets the namespace of an object at that object's instantiation time, just like Seuss. However, while Seuss provides a framework that automatically injects individual dependencies into the dependent object during instantiation time, Newspeak leaves this to the developer. Bracha shows that by restricting a module to accessing the set of objects that were passed in during instantiation time, untrusted software

---

[5] http://code.google.com/p/powermock/

can be sandboxed reliably by not passing in the dependencies that it would need to be harmful, such as file system access modules. The same argument holds for Seuss so long as reflection is disabled. While the rewiring of dependencies is a strong suit of dependency injection, and while Newspeak makes it technically possible, the system's design makes it costly in lines of code to run a unit test in a new configuration. By manually searching for a module instantiation that happens in a unit test, we could not find a single unit test in Newspeak that makes use of Newspeak's capabilities to change namespaces.

8   Conclusion

Static methods pose obstacles to the development of tests by hardwiring instance creation. A study of 120 static methods in open-source Smalltalk code shows that out of the 120 static methods, only 6 could not equally well be implemented as instance methods, but were not, thus burdening their caller with the implicit dependency on these static methods.

Dependency injection offers a partial solution to separating the responsibility of instantiating application objects or test objects, but still entails tedious rewriting of application code and the use of boilerplate code to fulfill the dependency injection design pattern. We have shown how introducing dependency injection as a language feature can drastically simplify the task of migrating class responsibilities to instance methods, while maintaining code readability and enabling the development of tests. Moreover, a language with dependency injection as a feature becomes more powerful and renders certain design patterns obsolete.

We have demonstrated the feasibility of the approach by presenting Seuss, an implementation of dependency injection as a language feature in Smalltalk. We have furthermore demonstrated the feasibility of our approach for statically-typed languages by presenting JSeuss, a partial port of Seuss to Java.

Acknowledgments

References

1. Achermann, F., Nierstrasz, O.: Explicit Namespaces (chapter 8). In: Weck, W., Gutknecht, J. (eds.) JMLC 2000. LNCS, vol. 1897, pp. 77–89. Springer, Heidelberg (2000)

2. Bracha, G., Ungar, D.: Mirrors: design principles for meta-level facilities of object-oriented programming languages. SIGPLAN Not. 39(10), 331–344 (2004)

3. Bracha, G., von der Ahé, P., Bykov, V., Kashai, Y., Maddox, W., Miranda, E.: Modules as objects in newspeak. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 405–428. Springer, Heidelberg (2010)

4. Brock, R.W., Wilkerson, B.: Object-oriented design: a responsibility-driven approach. SIGPLAN Not. 24, 71–75 (1989)

5. Ellis, B., Stylos, J., Myers, B.: The Factory Pattern in API Design: A Usability Evaluation. In: 29th International Conference on Software Engineering (ICSE 2007), pp. 302–312. IEEE Computer Society Press, Washington, DC, USA (2007)

6. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley, Reading (2002)

7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software, 1st edn. Addison-Wesley, Reading (1994)

8. Ko, A.J., Myers, B.A., Aung, H.H.: Six Learning Barriers in End-User Programming Systems. In: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing, VLHCC 2004, pp. 199–206. IEEE Computer Society Press, Los Alamitos (2004)

9. Leon, R.: SandstoneDb, simple ActiveRecord style persistence in Squeak, `http://www.squeaksource.com/SandstoneDb.html`

10. Mackinnon, T., Freeman, S., Craig, P.: Endo-testing: Unit testing with mock objects, ch.17, pp. 287–301. Addison-Wesley Longman Publishing Co., Inc, Boston (2001)

11. Dhanji Prasanna. Dependency Injection. Manning Publications, pap/pas edition (August 2009)

12. Renggli, L., Ducasse, S., Kuhn, A.: Magritte – A meta-driven approach to empower developers and end users. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 106–120. Springer, Heidelberg (2007)

13. Renggli, L., Gîrba, T., Nierstrasz, O.: Embedding languages without breaking tools. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 380–404. Springer, Heidelberg (2010)

14. Vanbrabant, R.: Google Guice: Agile Lightweight Dependency Injection Framework. Apress (2008)