# Extensible Modeling with Managed Data in Java

Theologos Zacharopoulos
CWI, The Netherlands
theol.zacharopoulos@gmail.com

Pablo Inostroza
CWI, The Netherlands
pvaldera@cwi.nl

Tijs van der Storm
CWI & University of Groningen,
The Netherlands
storm@cwi.nl

## Abstract

Many model-driven development (MDD) tools employ specialized frameworks and modeling languages, and assume that the semantics of models is provided by some form of code generation. As a result, programming against models is cumbersome and does not integrate well with ordinary programming languages and IDEs. In this paper we present MD4J, a modeling approach for embedding metamodels directly in Java, using plain interfaces and annotations. The semantics is provided by data managers that create and manipulate models. This architecture enables two kinds of extensibility. First, the data managers can be changed or extended to obtain different base semantics of a model. This allows a kind of aspect-oriented programming. Second, the metamodels themselves can be extended with additional fields and methods to modularly enrich a modeling language. We illustrate our approach using the example of state machines, discuss the implementation, and evaluate it with two case-studies: the execution of UML activity diagrams and an aspect-oriented refactoring of JHotDraw.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures

***Keywords*** Managed Data, Model-driven Development, Proxies, Interpretation

## 1. Introduction

MDD aims to raise the level of abstraction in software development through the use of high-level, domain-specific models that capture the main properties and relationships between the elements of a software system [14]. Instead of directly encoding the high-level properties and relationships from models in a general-purpose programming language, the models act as high-level specifications from which the implementation is derived, often through code generation.

Since models are defined in a high-level modeling language, programmers need to use dedicated, separate tools to support the development of such models. This produces a gap between the programming language and the development environment, on the one hand, and between the language and the models developers create and maintain, on the other hand. This situation leads to problems such as tool lock-in and co-evolution between models and code.

Managed Data [7] is an alternative approach, which provides modular strategies for defining the structure and manipulation mechanisms of data, directly within a programming language. As a result, data management becomes highly programmable, and decoupled from the mechanisms that are hard-wired in the host programming language. Managed Data relies on three main components: 1) a data description language that describes the structure of the data, 2) data managers that create and manipulate data according to the description, and 3) integration with the host programming language, so that managed data can be used in the same way as regular unmanaged data.

Managed Data can address crosscutting concerns such as logging or security, by specifying the behavior of data in custom data managers. Another interesting application of Managed Data is to enforce invariants on the data, for instance, certain high-level properties such as inverse-of or containment. Thus, Managed Data allows the embedding of high-level metamodeling properties in a general-purpose programming language.

In this paper we introduce Managed Data for Java (MD4J), a framework for embedding models in Java implemented with Managed Data. Since Java is a statically-typed language, the main challenge is to reconcile static types with the intensive use of runtime reflection to implement the managed objects. It turns out that Dynamic Proxies are the right mechanism to achieve this. As an added benefit, Java developers can still enjoy features such as code completion or type checking when programming against MD4J models.

MD4J enables two kinds of extensibility. First, the data managers can be changed or extended to create models with



*GPCE'16*, October 31 – November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4446-3/16/10...$15.00
http://dx.doi.org/10.1145/2993236.2993240

different kinds of data handling semantics. For instance, this supports a kind of aspect-oriented programming, where access, modification and creation of modeling elements can be instrumented with additional behavior. Second, the metamodels themselves can be extended with additional fields and methods to modularly enrich a modeling language with additional features. The combination of these two dimensions of extensibility provides for a very flexible framework for modeling without having to leave the Java ecosystem.

We introduce MD4J using a simple modeling language for state machines. We discuss how the meta model is specified, how state machines are constructed using various data managers, and how the state machine meta model can be extended with additional fields and behavior. Section 3, presents MD4J from the implementation perspective. In particular we detail the two-tiered dynamic proxy architecture that forms the back bone of MD4J. To exercise both dimensions of model extensibility, we provide two case-studies in Section 4. The first is based on execution of UML Activity Diagrams [11]. In particular, this involves extending the Activity meta model with fields and methods to allow activities to be executed. Executing preexisting benchmarks shows that MD4J Activity execution is between 19 and 27 times slower than the reference implementation due to the reflective overhead of dynamic proxies. The second case-study consists of an aspect-oriented refactoring of the drawing application JHotDraw [1], where custom data managers are used to encapsulate cross cutting concerns. Based on separation-of-concerns metrics we can conclude that tangling and scattering are indeed reduced.

The contributions of this paper can be summarized as follows:

- We present MD4J, an implementation of managed data in Java, for extensible modeling (Section 2).
- We detail how a two-tier dynamic proxy architecture is instrumental in implementing MD4J (Section 3).
- We evaluate MD4J in two case-studies, one consisting of a modular model extension to support UML Activity diagram execution, and one consisting of an aspect-oriented refactoring of JHotDraw (Section 4).

The code of MD4J and the case-studies can be found online here: `https://github.com/cwi-swat/JavaMD`.

## 2. MD4J: Managed Data for Java

### 2.1 Overview

Managed data [7] is based on two key ingredients: *schemas*, and *factories*. A schema defines the structure of data: it defines classes, relations, attributes and constraints that define the layout and well-formedness of data. In our setting, schemas correspond to class-based meta models, as used in MDD. To create data conforming to a schema, a factory acts as an interpreter of a schema in order to create objects supporting the features and constraints as specified in the schema.

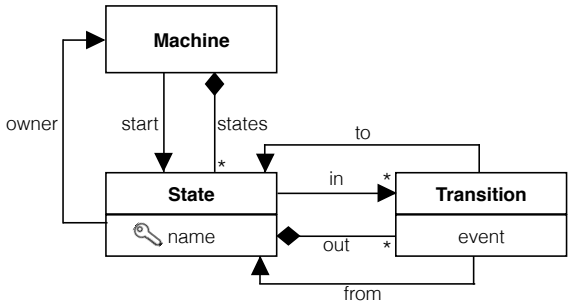


**Figure 1.** State machine metamodel

At a high level, MD4J relates schemas and factories through the following two functions:

$$load : Factory_{schema} \times \overline{Class}^{\Sigma} \rightarrow Schema_{\Sigma}$$
$$factory : Class\langle Factory_{\Sigma}\rangle \times Schema_{\Sigma} \rightarrow Factory_{\Sigma}$$

The function *load* takes a factory that is able to create *schemas*, and a sequence of reified Java interfaces that define a schema $\Sigma$, and then produces a schema structure that encodes $\Sigma$. Load can be seen as a parser of Java interfaces, turning them into a more abstract representation defined by the "schema of schemas" (the meta meta model). The *factory* function then takes a specification of the factory (a reified factory interface) and a schema describing $\Sigma$, and produces a factory object which can be used to create objects of the types defined in $\Sigma$.

The object $Factory_{\Sigma}$ acts like a standard Abstract Factory [3]. The key step that makes managed data (and hence MD4J) flexible in terms of data management, is that there can be different ways to obtain a factory object from a schema. In other words, the *factory* function can be defined in multiple, alternative ways, or base factories can be extended to provide additional behavior. The factory step thus decouples data definition from data management.

### 2.2 Example: State Machines

To make the abstract description of *load* and *factory* more concrete, let us consider the simple state machine meta model shown in Figure 1. A state machine consists of a number of named states; each state contains transitions to other states; and a transition has a field representing the event that triggers that transition. Figure 2 shows the corresponding MD4J schema.

The state machine schema contains a Java interface for every meta model class. Fields are defined methods with a varargs parameter. This is a convention to have a single declaration serving both as a getter (when invoked without parameters) and a setter (when invoked with one or more arguments). High-level metamodeling properties, such as containment, inverse relations and primary keys, are represented using Java annotations. All interfaces inherit from the M interface which is the base "managed data" interface of MD4J. We will come back to the M interface in Section 3.

```
interface Machine extends M {
  @Contain Set<State> states(State... states);
  State start(State... state);
}

interface State extends M {
  @Key String name(String... name);
  @Inverse(other = Machine.class, field = "states")
  Machine owner(Machine... machine);
  @Contain List<Transition> out(Transition... transition);
  List<Transition> in(Transition... transition);
}

interface Transition extends M {
  String event(String... event);
  @Inverse(other = State.class, field = "out")
  State from(State... from);
  @Inverse(other = State.class, field = "in")
  State to(State... to);
}
```

**Figure 2.** MD4J State machine schema

The interfaces that make up an MD4J schema represent the "source code" of a schema. To get an actual schema – that is, an object structure representing the information declared in the interface – it needs to be loaded. MD4J provides SchemaLoader, which, given a factory to create schema objects, can be used to create a schema for a set of interfaces like those of Figure 2.

Assuming we have schema factory schemaFactory, the following code creates the state machine schema:

```
Schema stmSchema = SchemaLoader.load(schemaFactory,
    Machine.class, State.class, Transition.class);
```

Schemas are represented as managed data themselves, to allow extensibility of data management behavior on schemas, and to decouple the schema interpretation as implemented by data managers, from the low-level reflection details of Java. We describe the "schema of schemas" in some detail in Section 3. The astute reader may have noticed an apparent circular dependency between *load* and *factory* if applied to the schema of schemas. This circularity needs to be broken using a bootstrap process, which we consider to be out of scope for this paper (cf. [18]).

### 2.2.1 Creation of Managed Objects

Once a schema is available, it can be given to a data manager which turns it into a factory to create objects of the types defined in the schema. In MD4J, the interface of such a factory needs to be explicitly defined, and must specify a factory method for every element of the schema. For example, this is the factory interface for the state machine schema:

```
interface StateMachineFactory{
    State State();
    Machine Machine();
    Transition Transition();
}
```

```
Machine doors(StateMachineFactory factory) {
  Machine m = factory.Machine();
  State s1 = factory.State();
  s1.name("Closed"); s1.owner(m);
  State s2 = factory.State();
  s2.name("Opened"); s2.owner(m);
  Transition t1 = factory.Transition();
  t1.event("open"); t1.from(s1); t1.to(s2);
  Transition t2 = factory.Transition();
  t2.event("close"); t2.from(s2); t2.to(s1);
  m.start(s1);
  m.states(s1, s2);
  return m;
}
```

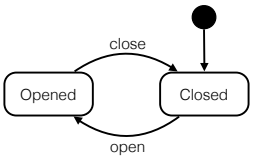


**Figure 3.** Creating a simple state machine controlling doors

Implementations of this interface will construct objects conforming to the interfaces defined in the state machine schema of Figure 2. The actual implementation of such a factory interface, however, is the responsibility of a data manager. Data managers specify the behavior of the objects created by the factories they produce.

MD4J includes a BasicDataManager, which implements default behavior for fields, associations, inverses, primary keys and dynamic type checking. Given the state machine schema defined earlier (stmSchema), and the factory interface StateMachineFactory, a concrete state machine factory is obtained as follows:

```
StateMachineFactory stmFactory =
  new BasicDataManager()
    .factory(StateMachineFactory.class, stmSchema);
```

The resulting stmFactory can be used just like any ordinary factory object in Java. In this case, however, the semantics of object creation, and the implementation of the declared fields in the schema interface is solely determined by the BasicDataManager.

Figure 3 shows a simple state machine for controlling doors and a method doors that creates the same state machine through a StateMachineFactory. The result of doors can be used as an ordinary Java object. For instance, the following method print prints out state machines to the console.

```
void print(Machine m) {
  for (State s: m.states()) {
    System.out.println("State " + s.name());
    for (Transition t: s.out())
      System.out.println(t.event() + " -> " + t.to().name());
  }
}
```

Calling the method print with the result of doors, produces the following output:

```
State Closed
open -> Opened
State Opened
close -> Closed
```

### 2.3 Extensible Modeling

The state machine schema defines a fixed structure for state machine models, and the BasicDataManager provides a default semantics for data management. Here we illustrate how state machine models can be extended in two dimensions:

- *Data manager extension* is used to instrument the constructed data for modifying its semantics. We illustrate this by adding logging and locking behavior to the basic data manager.
- *Schema extension*, is used to enrich the schema itself, without having to rewrite the model creation code (such as the doors method of Figure 3). We illustrate this feature by extending the state machine schema with execution semantics.

Data managers control the initialization and manipulation of the managed data. Thus, by switching the implementation of a manager, the behavior of the created objects can be altered. We may want, for example, to use a different data manager to log some operations when state machines are manipulated. Such extensions can be implemented by subclassing the BasicDataManager. The basic structure of BasicDataManager, the base class provided by the MD4J framework, is as follows:

```
class BasicDataManager implements IDataManager {
  ...
  @Override
  public MObject createManagedObject(Klass c,
    Object... xs) {
    return new MObject(c, xs);
  }
}
```

The createManagedObject (declared in IDataManager) method returns a new instance of the MObject class, which represents the basic behavior of a managed object. The MObject class implements the M interface as well as the java.lang.reflect.InvocationHandler interface. Thus, an instance of this class is able to respond to dynamic method invocations on the dynamic proxy, as we further elaborate in Section 3.

Data manager extensions are realized by overriding this method, and returning a custom subclass of MObject. For instance, a "logging data manager", will override the createManagedObject method in order to construct a specific type of MObject which has the logging behavior, as shown in Figure 4. The inner class Log overrides the _set method of MObject, which is the the extension point for field assignment. By having createManagedObject return a Log instead of a plain MObject, all created objects will have the logging behavior.

This data manager can now be used instead of the BasicDataManager:

```
class LoggingDataManager extends BasicDataManager {
  class Log extends MObject {
    public Log(Klass klass, Object[] inits) {
      super(klass, inits);
    }
    @Override
    public void _set(String name, Object value) {
      System.err.println("Setting " + name + " to " + value);
      super._set(name, value);
    }
  }
  @Override
  public Log createManagedObject(Klass klass, Object... inits) {
    return new Log(klass, inits);
  }
}
```

**Figure 4.** Implementing a data manager that logs field assignments

```
stmFactory = new LoggingDataManager()
  .factory(StateMachineFactory.class, stmSchema);
```

Passing stmFactory to the doors method of Figure 3 will print out Setting <x> to ... to the console for every field assignment.

In some cases, additional behavior may require additional interfaces for managed objects. A simple example of this scenario is "lockable objects": after constructing an object, it can be "locked" to prevent further modifications. To support lockable behavior, the data manager must somehow "inject" an interface into the managed objects to allow client code to call lock() on objects. MD4J anticipates such scenarios, by having the factory method accept *extra* interfaces. A subclass of MObject can then provide the required implementation.

For instance, the code for the LockableDataManager is shown in Figure 5. As in the case of the logging data manager, the createManagedObject method is overridden in order to create a more specific instance of MObject, in this case, one that disallows changes after the lockable object is locked. The factory method is overridden to add the Lockable interface to the constructed objects.

The following client code shows how the doors state machine can be locked after construction, using a factory created by the LockableDataManager:

```
stmFactory = new LockableDataManager()
  .factory(StateMachineFactory.class, stmSchema);
Machine doors = doors(stmFactory);
((Lockable)doors).lock();
```

### 2.4 Schema Extension

A common strategy in model execution is to represent runtime state as part of the model itself. MD4J supports schema extension through interface inheritance. This allows us, for instance, to extend the state machine schema with additional fields and methods to be able to execute state machines. The

```
class LockableDataManager extends BasicDataManager {
  interface Lockable { void lock(); }

  @Override
  public T factory(Class<T> f, Schema s, Class<?>... extra) {
    return super.factory(f, s, Lockable.class);
  }
  @Override
  public LockableObj createManagedObject(Klass klass,
                    Object... _inits) {
    return new LockableObj(klass, _inits);
  }

  class LockableObj extends MObject implements Lockable {
    private boolean isLocked = false;
    public LockableObj(Klass klass, Object... inits) {
        super(klass, inits);
    }
    public void lock() {  isLocked = true; }

    public void _set(String f, Object v) {
      if (isLocked) throw new IllegalStateException();
      super._set(f, v);
    }
  }
}
```

**Figure 5.** Data manager that constructs "lockable" objects

runtime state is represented by two new fields: current (in Machine), which represents the current state of a machine, and count (in State), which records the number of times a state has been visited during execution.

Behavior can be added to MD4J interface via Java 8 default methods. In this case it will be a single step method to execute state machines. Given an event, the machine checks if there is a corresponding transition from its current state. If so, it changes its current state to the target state of the transition, increasing the counter for the former current state and printing the executed transition.

Figure 6 shows the extended interfaces. RMachine, which extends Machine, represents "runtime" state machines. RState extends State with the additional count field.

To use these extended model interfaces, the data manager needs to be initialized with an extended schema and a refined factory interface. This last step is required, since the (reified) factory interface acts as the recipe for the data manager to decide which interface(s) a freshly created managed object needs to conform to. For executable state machines, the factory interface is refined as follows:

```
interface RStateMachineFactory extends StateMachineFactory {
  @Override RState State();
  @Override RMachine Machine();
}
```

Notice that this factory will only produce the executable versions of states and machines as it overrides the factory

```
interface RMachine extends Machine {
  RState current(RState... current);

  default void step(String event) {
    for (Transition t: current().out())
      if (t.event().equals(event)) {
        current().count(current().count() + 1);
        current((RState)t.to());
        return;
      }
  }
}

interface RState extends State {
  Integer count(Integer... count);
}
```

**Figure 6.** Extending the state machine schema with runtime state and behavior

methods State and Machine. Such factory overrides are valid due to Java's covariant return types.

The extended schema can now be used as follows:

```
stmSchema = SchemaLoader.load(schemaFactory,
  RMachine.class, RState.class, Transition.class);
stmFactory = new BasicDataManager()
  .factory(RStateMachineFactory.class, schema);
RMachine runtimeDoors = (RMachine)doors(factory);
runtimeDoors.step("open");
```

The schema now needs to incorporate the new interfaces. As the factory instance is now created using this new schema plus the extended factory definition from RuntimeStateMachineFactory, it produces objects conforming to RMachine. Calling the doors method with this factory, produces an RMachine. Since doors was defined to return a Machine, a downcast is needed if we want to execute it.

The code that was defined over the original schema remains valid. We could for instance pass the runtimeDoors object to the print method defined in Section 2, obtaining the same output.

### 2.5 Summary

In this section we have shown how MD4J support extensible modeling, using the example of state machines. The first two extensions involved new data managers to add logging and locking support. The third extension involved new schema classes extending the base schema for state machines with runtime state and behavior. In all three cases, code that was written to create state machines (e.g., doors), or code that traverses state machines (e.g., print), did not have to be rewritten.

## 3. Implementation: MD4J

The original implementations of Managed Data were hosted in dynamically-typed languages, using their reflective fea-

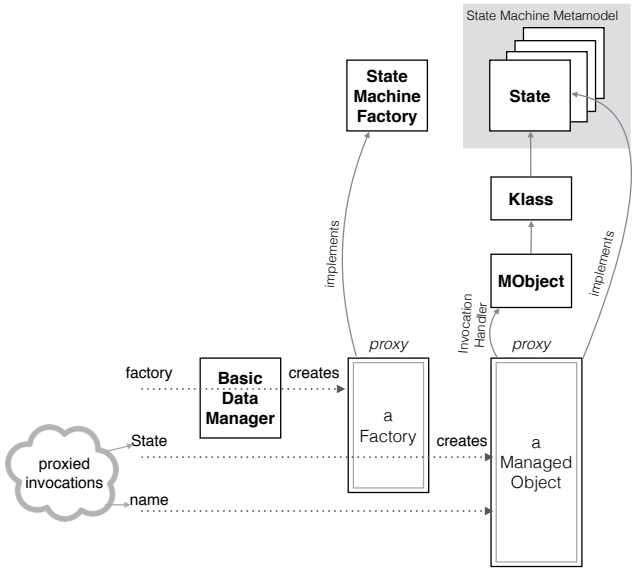


**Figure 7.** Two-tier proxies architecture of data managers

```
1  class BasicDataManager implements IDataManager {
2    ...
3    @Override
4    public T factory(Class<T> factoryClass, Schema schema,
5      Class<?>... additionalInterfaces) {
6      List<Class<?>> ifaces = interfacesForSchema(schema);
7      ifaces.addAll(Arrays.asList(additionalInterfaces));
8
9      return (T) Proxy.newProxyInstance(
10       factoryClass.getClassLoader(),
11       new Class<?>[]{factoryClass},
12         (proxy, method, args) ->
13           createManagedObjectProxy(factoryClass, schema,
14           ifaces, method, args));
15   }
16   private Object createManagedObjectProxy(
17     Class<?> factoryClass, Schema schema,
18     Method schemaFactoryCallingMethod, Object... inits) {
19     Klass schemaKlass =
20       klassForMethod(schemaFactoryCallingMethod, schema);
21
22     return Proxy.newProxyInstance(
23       factoryClass.getClassLoader(),
24       additionalIfaces.toArray(new Class[additionalIfaces.size()]),
25       createManagedObject(schemaKlass, inits));
26   }
27   @Override
28   public MObject createManagedObject(
29       Klass klass, Object... inits) {
30     return new MObject(klass, inits);
31   }
32 }
```

**Figure 8.** Code for BasicDataManager

tures [7]. In turn, MD4J is the embodiment of Managed Data in Java, a statically-typed language.

The key ingredient behind MD4J's implementation are Java Dynamic Proxies. Proxies provide a simple mechanism to support type-safe reflective behavior. Whilst in a dynamic language no assumptions about types are needed, a dynamic proxy needs to declare the interfaces that the proxied object conforms to, being up to the developer to provide the runtime behavior that meet these contracts.

In this section, we elaborate on this proxy-based architecture, exposing the criteria behind the main design decisions we made in order to bring Managed Data to a statically-typed language. We also discuss the implementation of our schema metamodel as Managed Data itself.

### 3.1 Data Managers

In MD4J, the design of data managers is based on a two-tier proxy architecture. The first proxy tier is in charge of reflectively creating factories, and the second tier corresponds to the proxies behind a call to the factory, that end up creating the custom semantics for the managed objects themselves. Figure 7 shows the two-tier proxy architecture of MD4J data managers, the core of our implementation. In order to make the architecture more concrete, the figure refers to the objects involved in the following client code:

```
StateMachineFactory =
  new BasicDataManager().factory(
    StateMachineFactory.class, schema);
State s1 = factory.State();
s1.name("OPEN");
```

The data manager creates a first-level proxy for the factory, in this case conforming to the StateMachineFactory definition. The implementation of this factory creates yet another proxy when a "creation" method is invoked. In the figure, the State method is invoked and therefore the second-level proxy's MObject is an invocation handler backed by a schema's metaclass (Klass) that refers to the original State interface. On this second-level proxy, the invocation of method name needs access to this data definition.

As discussed in Section 2, the BasicDataManager class is provided by the MD4J framework as a base implementation for data managers. Figure 8 shows the relevant code of its implementation. The factory method creates the first-level proxy, the one in charge of creating managed objects by analyzing the method that was called on the factory. This proxy exposes the interface corresponding to the factoryClass parameter. When a creation method is called on the factory proxy, the invocation handler (represented by the three-parameter closure) creates a new managed object by invoking the createManagedObjectProxy method. This method creates the second-level proxy, that represents a managed object. Notice that the invocation handler in this case is created by the method createManagedObject, which returns

```
interface Type extends M {
  @Key String name(String... name);
  @Inverse(other=Schema.class, field="types")
  Schema schema(Schema... schema);
  Class<?> javaClass(Class<?>... javaClass);
}

interface Klass extends Type {
  @Contain Set<Field> fields(Field... field);
  Set<Klass> supers(Klass... supers);
  @Inverse(other=Klass.class, field="supers")
  Set<Klass> subKlasses(Klass... subKlasses);
  @Optional Field key(Field... key);
}

interface Primitive extends Type {}
```

```
interface Field extends M {
  @Key String name(String... name);
  Boolean many(Boolean... many);
  Boolean optional(Boolean... optional);
  Boolean key(Boolean... key);
  Boolean contain(Boolean... contain);
  Type type(Type... type);
  @Inverse(other=Field.class, field="inverse")
  @Optional Field inverse(Field... field);
  @Inverse(other=Klass.class, field="fields")
  Klass owner(Klass... owner);
}

interface Schema extends M {
  @Contain Set<Type> types(Type... type);
}
```

**Figure 9.** Schema schema

an MObject. This object is responsible for interpreting the schema in order to implement the custom behavior. Note too that createManagedObject could be overridden by data manager extensions to extend the functionality of the basic MObject.

Consider again the client code that assigns the name "OPEN" to an instance of State:

```
1 State s1 = factory.State();
2 s1.name("OPEN");
```

Based on the listing of BasicDataManager, we can step into this snippet:

1. The method State is called on factory, so s1 is bound to a proxy whose invocation handler is an MObject. This managed object is built with a "klass" that encodes its schema, in this case, the one that corresponds to State (line 30, Figure 8). Notice that in order to obtain such "klass", the factory proxy has to match the invoked factory method StateMachineFactory.State() against the schema that the factory itself received (line 20, Figure 8).
2. The method name is called on s1 with the argument "OPEN", so the proxy's MObject responds to this invocation. First, it tries to find the field name in the klass that encodes State. Then, it checks the number of arguments. As in this case the number is one, it interprets the call as a setter. It first type-checks the argument, and then assigns the new value to the internal representation of the field.

### 3.2 Schema of Schemas

We have seen that in order to create factories, we need an instance of Schema that encapsulates the schema definition. This Schema itself is defined as managed data, defined by the Schema schema shown in Figure 9. Since we represent schemas as managed objects, we benefit from uniform access, being able to inspect reflective metadata in the same manner as ordinary managed objects. The Schema acts then as a mirror [2] of the Java interfaces that constitute the metamodel.

In the figure, we can see that the Schema consists of a set of Types, which can be either primitive (Primitive) or reference type (Klass). Primitives represent object that are not managed by MD4J. The structure of the annotated interface methods is encoded in the Field class. When this schema is loaded with a (bootstrapped version of) itself, the schema becomes self-describing.

The mechanism to represent Java interfaces in MD4J is used by the interface M, discussed earlier. This interface, that every metamodel element inherits from, ensures that every managed object points to the meta-information about the schema that defines itself, being this description encoded as an MD4J "klass":

```
interface M {
  Klass schemaKlass(Klass... schemaKlass);
}
```

***Summary*** We have presented how to implement the concept of Managed Data in Java using a two-tier proxy architecture: the first level exposes a factory interface, and implements the creation methods delegating to second-layer proxies. The latter represent the managed objects backed by the schema. These schemas are Managed Data themselves, providing a simple mirroring architecture.

## 4. Evaluation

We have evaluated MD4J in two case-studies. First, the extensibilitiy of models in MD4J is exercised in an implementation of UML Activity Diagrams [12]. The second case-study shows how the extensibility of data managers can be used to do aspect-oriented programming, if parts of an application are modeled using MD4J; this case-study is based on a refactored version of JHotDraw [1].

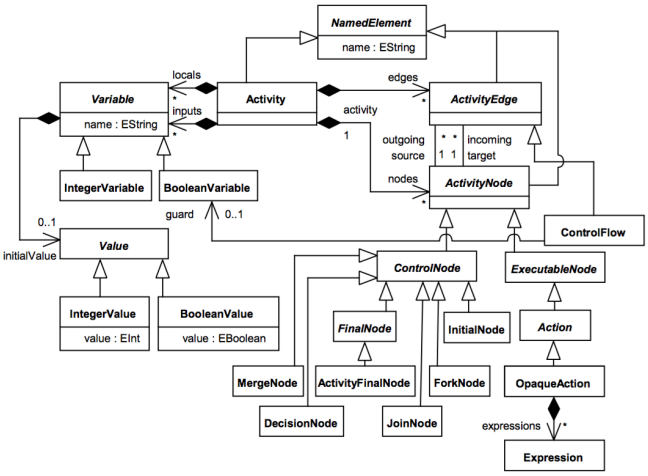


**Figure 10.** UML Activity Diagram metamodel

### 4.1 Executing Activity Diagrams

Execution of UML Activity Diagrams has been recently proposed as one of the challenges of the 8th Transformation Tool Contest (TTC'15) [11][1]. The goal is to extend a (simplified) metamodel of UML Activity Diagrams with additional classes, relations and attributes for representing the dynamic execution of Activities. Additionally, execution of an activity model can be seen as transforming this model at runtime, and the "output" of execution is captured by attributes in the model itself. The reference implementation realizes the behavior as ordinary methods in Java classes generated from an extended ECore metamodel [2].

The UML activity diagram metamodel is presented in Figure 10. This metamodel has been modeled using MD4J, in a similar style to how the state machine metamodel was defined in Section 2. Then in a separate package, the metamodel is extended in the following way:

- New classes for defining runtime concepts, such as tokens, activity trace, input values.
- New fields on existing entities. For instance, the type Activity is extended with a field containing the trace of execution.
- Methods defining the execution semantics. For instance, all activity node types get a method fire(List<Token>), triggering the execution of a node.

As a result, the execution semantics of UML Activity diagrams is defined in a completely modular way. In contrast, the reference implementation is realized by adding the execution methods to the generated code from the ECore metamodel.

The two separate activity diagram schemas are concise as well. The static schema takes up 363 source lines of code (SLOC). The runtime extension requires an additional 491 SLOC. The implementation classes (excluding generated interfaces) of the reference implementation consist of 3704 lines of code, but this includes some boilerplate code generated by EMF to deal with bidirectional associations. In MD4J this logic is encapsulated by the data manager code. However, the main benefit of MD4J is that the base activity schema remains open for extension: it is easy to add new interpretations, in addition to execution, without having to change the original code.

[1] http://www.transformation-tool-contest.eu/2015/

[2] https://github.com/moliz/moliz.ttc2015

| Test | Reference | MD4J |
|---|---|---|
| *TestPerformanceVariant_1* | 2.41 | 45.53 |
| *TestPerformanceVariant_2* | 2.70 | 34.39 |
| *TestPerformanceVariant_3_1* | 7.38 | 30.79 |
| *TestPerformanceVariant_3_2* | 0.17 | 4.57 |

**Table 1.** Comparing the performance of MD4J Activity Diagram execution to the reference implementation of the TTC'15 model execution challenge [11] (in seconds)

***Performance*** The TTC'15 Activity Execution challenge provided 3 test cases to evaluate performance. They consist of pathologically large activity diagrams. We have used these test cases to compare the performance of the MD4J solution to that of the reference implementation. The measured execution times for each test activity for both versions are presented in Table 1.

The results show that the MD4J implementation is between 19 and 27 times slower than the reference implementation. Since MD4J has been implemented using a large amount of Java Reflection and Dynamic Proxies, it is unfavorable for applications that need performance to use it. Data managers dynamically analyze the schemas through reflection, which makes it a lot harder for the compiler to optimize. More specifically, even though the HotSpot JVM has one of the best just-in-time compilers, Java's dynamic proxies introduce significant overhead [10]. In particular, the activity execution also creates model objects at runtime, which involves querying the schema. On the other hand, we expect that there are sufficiently many use cases where raw performance is not a requirement, and where modularity and extensibility is more important.

### 4.2 Aspect-Oriented Refactoring of JHotDraw

The Activity Diagram case study illustrated how schemas could be extended with additional data fields and behavior. In this section we present a case study exercising the other axis of extension: data managers. We show that by modeling parts of a regular software application using MD4J schemas, custom data managers can be used to encapsulate aspects [5].

The case study is based on JHotDraw, a Java drawing framework and application, which has been used to showcase the use of design patterns [1]. Earlier work has identified parts of the code that are candidates of aspect-oriented refactoring:

```
public void execute() {
  setUndoActivity(createUndoActivity());
  getUndoActivity().setAffectedFigures(view().selection());
  FigureEnumeration fe =
    getUndoActivity().getAffectedFigures();
  while (fe.hasNextFigure()) {
    fe.nextFigure().setAttribute(fAttribute, fValue); }
  view().checkDamage();
}
```

**Figure 11.** Refactoring the undo-related concerns from Command classes in JHotDraw: the boxed code is moved into a data manager in the refactored version of JHotDraw

concerns which resist traditional forms of modularization [9]. This analysis has been the basis of a refactoring of JHotDraw where such aspects have been specified separately using AspectJ [8]. In this case study we perform a similar refactoring, but using MD4J to encapsulate the aspects as part of custom data managers.

The aspects we have refactored are instances of the Observer pattern [3] and an instance of "Undo" behavior. The Observer pattern is realized by the StandardDrawingView class, which implements a FigureSelectionListener, to handle changes to the figure selection. The undo functionality is implemented in JHotDraw with the Command design pattern: when a command is executed, an undo activity is created. The logic of undo is thus tangled with the execution logic of the command itself. In our case study we eliminate this tangling from the ChangeAttributeCommand class, which deals with changes to visual attributes (e.g., color, font, line style, etc.). Figure 11 shows the execute method of ChangeAttributeCommand. The undo-related code is boxed. In the refactored version of JHotDraw this code is moved completely to a custom data manager.

***Extracting FigureSelectionListener Code*** The particular instance of the Observer pattern is realized through the FigureSelectionListener interface. It is used to notify a subject for changes in the current figure selection in the DrawingView of JHotDraw. Accordingly, the class StandardDrawingView realizes the Observable role, providing methods for adding and removing figure selection listeners. The concern specific code is scattered though a number of places of the StandardDrawingView class, and includes fields to store the listeners, and methods to add, remove, and notify listeners.

To extract this logic from the StandardDrawingView, we have modeled it using a schema, with default methods for the core functionality of the class. The code that is concerned with the listeners is extracted into a separate interface, which is passed as an extra interface to the custom data manager which implements the logic by means of subclassing the MD4J MObject class. The refactoring thus follows the pattern of making managed object "lockable" (Section 2.3).

***Extracting Undo Logic*** To extract the undo logic from the ChangeAttributeCommand, we again have modeled it as managed data. A custom data manager then enriches it with an interface that supports creating the Undo activity required for this particular command. It furthermore intercepts the execute method, to first register the undo activity, before actually executing the command itself.

After the aspect refactoring, all of the undo related functionality of the command has been removed from the command's code and are externally attached by the data manager. Following the original design, the creation of an UndoActivity instance is defined inside a nested class, but now resides inside the command-specific subclass of MObject. The original execute method of the command is unaware of the attached undo functionality.

***Effect on Scattering and Tangling*** To assess the effect of the aspect-oriented refactoring of JHotDraw, we computed a number of metrics designed to measure the amount of tangling and scattering in code [13]. We then compare the metrics from the original JHotDraw and our refactored version ("MDJHotDraw"). The metrics have been manually computed on the parts of the code that make up the two concerns. This is determined using the Vocabulary Size metric, which counts the participant components of a concern [13].

The metrics are the following: *Concern Diffusion over Components* (CDC) counts the number of primary components whose main purpose is to contribute to the implementation of a concern. This metric counts the degree of concern scattering at the level of classes. *Concern Diffusion over Operations* (CDO) counts the number of primary operations whose main purpose is to assist the implementation of a concern. This metric counts the degree of concern scattering at the level of methods. *Concern Diffusion over LOC* (CDLOC), counts the number of transition points for each concern through the lines of code. The higher the metric, the more intermingled is the concern code within the implementation; the lower the metric, the more localized is the concern code. This metric aims to compute the degree of concern tangling. In the case of MDJHotDraw, the abstract and reusable data manager classes have not been taken into consideration, since they represent generic components which can be reused in other concern refactorings (e.g., other instances of the Observer pattern).

The results are shown in Figure 12. As can be seen the MDJHotDraw implementations scores lower on all metrics, which indicates that the refactoring indeed reduced the scattering and tangling of the application, leading to more cohesive application classes.

### 4.3 Discussion

The case studies show that MD4J can be used to address realistic software development challenges, without heavyweight tools, and without losing the standard IDE support that developers are accustomed to. First of all, the extensibility of schemas supports a powerful way to modularly enrich

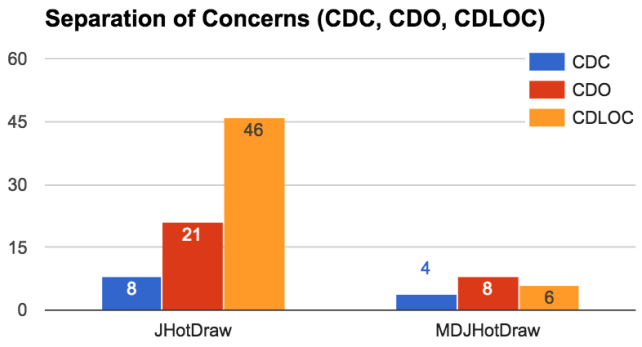


**Figure 12.** Metrics-based comparison of separation of concerns between vanilla JHotDraw and MDJHotDraw

models with additional data or behavior. Furthermore, the intermediate step of data managers – sitting in between the declaration of data and its manipulation – represents a flexible hook to inject generic behavior into managed objects. The JHotDraw refactoring shows that this can be used to do aspect-oriented programming.

A limitation in using MD4J is that methods are declared as part of interfaces. For instance, all methods are necessarily public, and there is no support for additional, non-managed fields. Finally, since all invocations on a managed object are handled by proxies, debugging MD4J models can be quite a challenge. These proxies also present a signficant performance overhead, as shown in the Activity execution case study. Note however, that recent work on reflective meta programming could elminate this overhead almost completely [10].

## 5. Related Work

***Managed Data*** Managed Data [7] is a general approach to data abstraction in which developers control the mechanisms for creating and manipulating data without leaving the host programming language. The original paper [7] implements the concept using the dynamic OO language Ruby. Data is described using a custom schema definition that is interpreted at runtime by a managed object. MD4J brings this idea to Java but in a fully hosted manner. The schema language is thus a subset of Java, corresponding to annotated interfaces.

Other traditional techniques than can be considered Managed Data include Lisp macros and Meta Object Protocols (MOPs).

In the context of the Lisp family, macros have been used to support custom data abstraction mechanisms, e.g., `defstruct` [15] and `define-type` [6]. Unlike MD4J, the data management is encoded in the transformation logic of a macro, and therefore is more rigid and less suitable for extensions.

MOPs [4] were introduced for CLOS (Common Lisp Object System) to inspect and modify the behavior of object and classes, conceiving a protocol with specific hooks to allow the customization of runtime semantics in a programmatic way. MOPs are a more general approach that Managed Data, as they allow reflection on the behavior of objects as well. In turn, MD4J focuses only on the data aspect. Even so, the design of the appropriate "hooks" methods in a MD4J managed object (class `MObject`) is analogous to the design of a data-centric MOP.

***Metamodeling*** Modeling frameworks provide tooling support to do model-driven development in a particular platform, enabling thus the integration of the model development lifecycle into the traditional software development workflow. For example, EMF ECore [16] is a widely-known modeling framework for the Java and Eclipse ecosystem. In EMF ECore models are defined in a custom high-level language (either in a textual or graphical concrete syntax), and from these models, Java code is generated for common scaffoldings, such as persistence and logging. While in MD4J, data management is configurable programmatically, in EMF this is possible only as part of the tool.

***Proxies*** Dynamic proxies have a long tradition in implementing dynamic behavior in dynamic languages. Examples of this are Smalltalk and, more recently, Javascript. In statically-typed languages like Java, a proxy object needs to anticipate the interfaces the object implement, limiting flexibility. Although focused on a dynamic language, in [17] the authors present a series of principles to design reflective APIs using proxies. As future work, MD4J could incorporate some of these guidelines.

## 6. Conclusion

We have presented MD4J, an approach to extensible modeling in Java based on managed data. Models are defined in Java, using schemas, which consist of ordinary Java interfaces. This provides a direct embedding of models in a mainstream programming language, without losing host language support (e.g., type checking, code completion, etc.).

We show how MD4J supports two kinds of extensibility: extending the schemas that describe the structure of models, and extending the internal data managers that interpret a schema. The two dimensions have been exercised in two case-studies, one based on executing UML Activity diagrams, and one consisting of an aspect-oriented refactoring of JHotDraw. The results show that MD4J presents a very flexible architecture to address realistic software development challenges.

The implementation of MD4J is based on an two-tiered architecture of dynamic proxies, which provides generic implementations for the types defined in MD4J schemas, as well as for factory interfaces to create objects. One aspect that needs future work is improving the performance of the data managers. Nevertheless, MD4J presents a simple framework for extensible modeling in Java, where both the structure of the data as well as its semantics can be modularly extended.

## References


[1] JHotDraw. Online, 2007. `http://www.jhotdraw.org/`.

[2] G. Bracha and D. Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 331–344, New York, NY, USA, 2004. ACM.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[4] G. Kiczales, J. Des Rivieres, and D. G. Bobrow. *The art of the metaobject protocol*. MIT press, 1991.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 - Object-oriented programming*, pages 220–242. Springer, 1997.

[6] S. Krishnamurthi. *Programming languages: Application and Interpretation*. 2007.

[7] A. Loh, T. van der Storm, and W. R. Cook. Managed data: modular strategies for data abstraction. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software (Onward!'12)*, pages 179–194. ACM, 2012.

[8] M. Marin. Refactoring JHotDraw's undo concern to aspectj. In *Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE 2004)*, 2004.

[9] M. Marin, A. Van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 132–141. IEEE, 2004.

[10] S. Marr, C. Seaton, and S. Ducasse. Zero-overhead metaprogramming: Reflection and metaobject protocols fast and without compromises. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 545–554. ACM, 2015.

[11] T. Mayerhofer and M. Wimmer. The TTC 2015 model execution case. In *TTC'15*, pages 2–18, 2015.

[12] Object Management Group. OMG unified modeling language. version 2.5. Online, March 2015. `http://www.omg.org/spec/UML/2.5/PDF`.

[13] C. Sant'Anna, A. Garcia, C. Chavez, C. Lucena, and A. Von Staa. On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proceedings of Brazilian symposium on software engineering*, pages 19–34, 2003.

[14] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[15] G. Steele. *Common LISP: the language*. Elsevier, 1990.

[16] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[17] T. Van Cutsem and M. S. Miller. Proxies: Design principles for robust object-oriented intercession APIs. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, pages 59–72, New York, NY, USA, 2010. ACM.

[18] T. Van Der Storm, W. R. Cook, and A. Loh. The design and implementation of object grammars. *Science of Computer Programming*, 96:460–487, 2014.