A Two Language Approach for Implementing Dynamic Systems

Eli Acherkan
Pontis LTD.
Gil Yam 46905, Israel
eliac8@gmail.com

Atzmon Hen-Tov
Pontis LTD.
Gil Yam 46905, Israel
atzmon.hentov@gmail.com

David H. Lorenz
Open University of Israel
Raanana 43107 Israel
lorenz@openu.ac.il

Lior Schachter
Open University of Israel
Raanana 43107 Israel
liorsav@gmail.com

ABSTRACT
An Adaptive-Object-Model (AOM) is a software architecture
style aimed at supporting system evolution without software
delivery. A known limitation of the AOM style is the type-duality
problem it introduces: both the host language type system and the
Type Object pattern represent domain classes. While previous
research tried to solve this problem by using a dynamic object-
oriented language, we present a solution that comprised Java as
the behavioral language and a new language named Ink as the
dynamic structure language.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and
Features – abstract data types, polymorphism, control structures.

General Terms
Language, Design

Keywords
AOM, Mirrors, Type Object, VM, Java

1. INTRODUCTION
Rapid changes in business practices create the need for dynamic
systems that support rapid changes in software. While dynamic
systems are gaining popularity, statically typed object-oriented
languages still account for the majority of the work force in the
software industry, as recent statistics indicate.1 The popularity of
these languages has created over the years a vast array of third-
party components and frameworks (Spring, Hibernate, etc.) that
reduce the application development effort significantly, which in
turn increases their popularity. Even dynamic systems are often
implemented in statically typed languages.

One way to implement a dynamic system in a statically typed
object-oriented language is to “decouple instances from their
classes” via a TypeObject [12] pattern. TypeObject “allows new
‘classes’ to be created dynamically at runtime, lets a system
provide its own type-checking rules” [12]. However, this comes at
a cost. These so-called “classes” are not actually classes. A Type
class instance plays the role of a class, although it is a terminal
instance in the hosting language. This leads to the following
problems:

- A mismatch occurs between the semantics of TypeObject and
  that of the underlying object-oriented language. This

mismatch places extra cognitive burden on the developers as
they have to repeatedly map between two type systems.

- Software tools and frameworks created for the object-oriented
  language will probably not work with the new type system.
  Developing the supporting infrastructure can be costly and
  complicated.

In prior research, Yoder et al. have shown that this mismatch is
significantly reduced when implementing the AOM system in a
language that natively supports metaclasses [10]. An
implementation of a dynamic AOM system was illustrated in
Smalltalk and MetaclassTalk. However, metaclasses are not
available in Java and other popular statically typed object-oriented
languages.

1.1 The Ink Approach
In this paper, we present a novel hybrid approach to implementing
AOM systems. In this approach, two languages are used
collaboratively for implementing an AOM system:

1. A dedicated language for describing OO structures:
The structure of the system is expressed using a new
dynamic OO-like language, called Ink. Ink can
dynamically define and change the class structure at
runtime. It supports explicit metaclasses [3].

2. A dedicated language for describing behavior: The
behavior of the system is expressed with a statically
typed OO language (namely, Java).

By using two languages rather than one, a dynamic language for
structure and a static language for behavior, we combine the
benefits of both worlds. This promotes a style of programming
with a strict separation of structure and behavior.

To avoid a type-duality problem between the two languages (the
problem we are trying to solve), our solution also includes an IDE
and a VM component:

- Integrated Development Environment: The IDE enforces
  mapping rules between the structure classes and the behavior
classes at design time. It validates that the imperative
  language code complies with the model defined in the
  structure language. The IDE also provides tooling support for
  the structure language with the same level of usability as that
  exists for the statically type object-oriented languages
  (incremental builder, auto-completion, etc.).

---

1 TIOBE Programming Community Index for June 2011,
http://www.tiobe.com/index.php/content/paperinfo/tpci/.
Virtual-Machine: The Ink VM (IVM) binds at runtime the structure and the behavior. The IVM provides an API for accessing the structure model. The IVM also provides a meta-object protocol (MOP) for manipulating the application model at runtime (e.g., defining a new class).

Finally, Ink as a generic solution for AOM systems can reduce the overhead of developing supporting infrastructures, by creating reusable libraries that can be used in different applications.

2. LANGUAGE FEATURES

Ink is class-based and supports explicit metaclasses. The Ink meta-metamodel is similar to the MetaclassTalk [2] meta-metamodel refined with concepts from the MOF [9]. It is implemented in the Ink Virtual Machine (IVM), and supports advanced object-oriented features, such as metaclasses, traits, object inheritance, and more.

The Ink meta-metamodel comprises a set of constructs to support a 4 meta-level system. Figure 1 depicts a class diagram showing the basic elements and relationships within the Ink metametamodel.

2.1 Objects

The InkObject class is the root of the Ink inheritance hierarchy. It is an instance of the metaclass InkClass. Every object in Ink is an indirect instance of InkObject.

InkObject defines the basic structure of all objects (except primitives) in the Ink language. An Ink object may have property values for the properties defined in its class. The type of each property value must conform to the type declared by the property definition in the class.

Listing 1A is a sketch of the definition of InkObject written in Ink. The definition declares the ID, class, and super-object of InkObject, and populates the personality property (defined in InkClass) with another Ink object (an instance of Personality).

```
Class id="InkObject" class="InkClass" super="nil" {
  personality class="Personality"{
  ...
  }
  ...
}
```

Listing 1A – InkObject basic structure and attributes.

Generally, the definition of an addressable object (one that is not contained within another object) must specify a unique ID, e.g., id="InkObject" in Listing 1. The definition of an object O2 in Ink must indicate the object’s class C2, and may also indicate its super-object O1 (which O2 extends). For example:

```
Class id="C1" class="InkClass" super="InkObject" {
  ...
}  
Class id="C2" class="InkClass" super="C1" {
  ...
}  
Object id="O1" class="C1" super="nil" {
  ...
}  
Object id="O2" class="C2" super="O1" {
  ...
}
```

Ink supports object inheritance. An object O2 of class C2 may extend another object O1 of class C1, if C2 is a descendant of C1. By extending O1, O2 inherits all the property values of O1, and may override them. A property value is either a primitive (string, integer, date, etc.), a complex (e.g., another Ink object), or a collection of primitives or Ink objects:

- **Primitive values**: if O1 contains a primitive value V1 for property P, then O2 may override this value by providing a different value V2 of the same primitive type. For example:

  ```
  Object id="O1" class="C1" super="nil" { P V1 ... }  
  Object id="O2" class="C2" super="O1" { P V2 ... }  
  ```

- **Complex values**: if O1 contains a complex value O3 for property P, and O3’s class is C3, then O2 may override the value O3 in a covariant manner, by providing a different
instance O4 of C4, where C4 is a descendant of C3. For example:

Object id="O1" class="C1" super="nil" { P O3 ...
Object id="O2" class="C2" super="O1" { P O4 ...

Object inheritance is applied recursively on the entire objects graph. The object O4 (contained within O2) inherits all the property values of O3 (which is contained within O1), and may override them. For example, to override the personality property of InkObject shown in Listing 1A, one would need to supply an instance of either Personality or one of its subclasses. In Figure 1C, InkClass overrides the personality property value inherited from its superclass InkObject.

2.2 Classes and Metaclasses

InkClass is the base metaclass in the meta-metamodel. InkClass is an instance of InkClass (self-referencing) and extends InkType (which extends InkObject). It is the root of the metaclass hierarchy. Every class (and metaclass) in Ink is an instance of InkClass or one of its subclasses, and extends InkObject or one of its subclasses.

An explicit metaclass in Ink is a class that extends InkClass. InkClass thus defines the structure of all classes and metaclasses (Listing 1B).

```java
Class id="InkClass" class="InkClass" super="InkType"{
  properties{
    property class="ListProperty"{
      name "properties" mandatory false
      item class="Reference"{
        type ref="Property"
        name "property"
      }
    }
    property class="Reference"{
      name "personality"
      mandatory true
    }
  }
}
```

Listing 1B. InkClass basic structure and attributes

Class inheritance in Ink is essentially object inheritance applied to classes. Like objects, classes may refine the type of an inherited property definition (covariance).

A class may define a property as either mandatory or optional. An instance is valid if all mandatory properties have values. For example, in Listing 1B the personality property is defined as mandatory (forcing all classes to have a value assigned to their personality property, either explicitly or inherited). In Listing 1A, the InkObject class supplies a value to the personality property.

2.3 Traits

Ink supports traits [4]. This is done using the Personality class, of which each property value is a Trait instance. The Trait class is an instance of the TraitClass metaclass, which extends InkClass. A trait is a kind of InkObject with a target property. The target of a trait is another InkObject (Figure 1).

A trait is typically used to handle crosscutting (technical or business domain) concerns that are not handled in the target. This promotes encapsulation of different concerns in reusable units. A trait may add properties to its target, which are injected to the target’s class and thus change the target class’s structure (injectedProperties in Figure 1). Consequently, instances of the target class will have to supply values to these properties, and the trait behavior can adapt its execution according to the supplied values.

The meta-metamodel defines two traits:

- **Reflection** – a mirror-based API, including introspection and self-modification [1].
- **Constraints** – an API for validations such as type checking, mandatory checking and other restriction enforcing.

A metaclass MC may add new traits by overriding the personality property definition in InkClass and providing a Personality subclass SP as the personality property type. A class C1 which is an instance of the metaclass MC will have to supply an instance of SP or one of its descendants as its personality value. A class C2 which extends C1 may inherit C2 traits [2] or override them in a covariant manner (as explained earlier). Listing 1C shows how InkObject defines Mirror as its reflection trait and Constraints as its constraints, while InkClass refines this definition to ClassMirror (which is a subclass of Mirror) and ClassConstraints, respectively.

```java
Class id="InkObject" class="InkClass"{
  personality class="Personality"{
    reflection class="mirror"
  }
  constraints class="Constraints"
}
```

```java
Class id="InkClass" class="InkClass" super="InkType"
  personality class="Personality"{
    reflection class="ClassMirror"
    constraints class="ClassConstraints"
  }
}
```

Listing 1C. InkClass refines the personality definition of InkObject

2.4 Properties

The Property class is an instance of InkClass and extends TypedObject. As shown in Listing 1B, InkClass defines the properties property, which is a list of Property’s. Properties determine the structure of objects, by specifying the name and type of the property as well as other metadata and constraints (see the definition of the personality property in Listing 1B).
A property in Ink is a first class object and can be extended by programmers. Figure 2 is a class diagram showing the subclasses of Property that are defined in the meta-metamodel. Those classes differ in the type of allowed values, as well as the metadata on the property definition:

- **Reference** – contains or points to an Ink object.
- **EnumAttribute** – contains an enum type value.
- **CollectionProperty** – contains maps (MapProperty) or lists (ListProperty).
- **Subclasses of PrimitiveAttribute** – contain the corresponding primitive values.

A Property subclass expresses additional characteristics or constraints that will be applied to the assigned value, e.g., in Figure 2 StringAttribute defines minLength, maxLength and regExp as constraints on its values. It can be further extended to express application-specific constraints. For example, a class called FirstNameAttribute can extend StringAttribute and validate that the first name provided as a value starts with a capital letter.

### 2.6 DSL Factory

The DSLFactory metaclass is used to define a DSL (Domain-Specific-Language) in Ink. Its responsibilities are to provide a lookup service for Ink instances, load them from the file system, and enable scoping by managing the relationships between DSLs. The ObjectFactory class is an instance of DSLFactory, which is responsible for object instantiation and dependency injection. Every Ink class contains a reference to the ObjectFactory, which identifies the DSL to which it belongs.

Ink follows ModelTalk [6] in its DSL-based approach. Ink applications are a set of interconnected textual DSLs. A DSL is a reusable component identified by a namespace, e.g., the namespace of the Ink meta-metamodel is "ink.core". A DSL describes a particular domain by providing the abstractions and restrictions that capture the domain. A DSL defines which of its instances can be exported to other DSLs.

A DSL can import other DSLs in order to use them in its context or to specialize them by refining the semantics. When in the context of one DSL there is a reference to an element from another DSL, the reference should include the fully qualified name of the element (i.e., namespace:ID).

### 2.7 Decoupling Structure and Behavior

A key characteristic of our two language approach is the strict separation between structure and behavior. System structure and static configuration are expressed using Ink, whereas the behavior is written in a statically typed OO language (e.g. Java, C#). The Ink IDE verifies that the behavior classes conform to the structure
An Ink application programming starts with describing the structure (classes and metaclasses) and static state (instances) of the domain. Classes and metaclasses always have a counterpart in the behavior language. This leads to a split representation, where an Ink class (or metaclass) consists of two parts: a structure definition and a behavior definition.

The behavior language counterpart can be inherited. In some cases (e.g., AOM), when the subclass introduces only structural or meta-data changes, the behavior of its super-class is sufficient and there is no need to have a particular behavior class in the behavior language. This allows for adaptation of system behavior solely by changing structure and configuration.

This simple set of mapping rules defines the association between the two parts, making it simple to navigate between them and easing the cognitive burden.

As stated above, a method of the behavior class can correspond to an Operation in the DSL. When such a method is executed at runtime, the interceptors as defined in the operation are invoked.

3. IMPLEMENTATION DETAILS

Our current implementation of Ink is JVM-based. Use of Java technology provides platform portability and compatibility with a large selection of third-party solutions. Extending Java with meta-programming features is achieved by an Ink Virtual Machine that runs on top of the JVM and handles object instantiation and dependency injection.

3.1 Ink Syntax

Ink programs are written in SDL (Simple-Declarative-Language [11]). SDL has a compact and readable syntax, well suited to describe structures. Ink adopted a restricted version of SDL. IDE support for the Ink syntax (coloring, auto-completion, problems view, etc.) is provided by an Eclipse plugin (editor and builder extensions).

3.2 Ink Virtual Machine (IVM)

The IVM runs on the JVM and is responsible for executing an Ink application. Listing 2 depicts the Java interfaces that provide the main APIs for interacting with the IVM: VM, DslFactory and Context.

The entry point to the IVM is the InkVM class, which is a singleton that is responsible for locating DSLs on the classpath, initializing them, and managing their life cycle. The InkVM class implements the interface VM, which provides the API for obtaining a DslFactory or Context.

A DSL is represented by an instance of the DslFactory metaclass. The IVM runtime contains a set of inter-connected DslFactory objects, which identify the scope of the running Ink application. The DslFactory provides methods for retrieving DSL elements, creating new ones, and optionally registering them in the IVM. It also contains functionality related to DSLs scoping and utility methods for creating instances and proxies using Java reflection API.

A Context is a trait of DslFactory that exposes limited functionality of the factory by providing such methods as getForObject(...), newInstance(...), and register(...).

An Ink application can be embedded in a regular Java application, co-existing with plain Java objects and third-party libraries. A standard Java class can access the IVM, retrieve a Context, and start working with Ink objects. This is done by invoking InkVM.instance().getContext().

2 http://code.google.com/a/eclipselabs.org/p/ink/
The separation between the interface and implementation aids encapsulation, and also allows the IVM to place a Java proxy in front of the implementation. (Proxies are used by the IVM to implement mechanisms such as interceptors, and standard Java proxies are restricted to interfaces, not classes.)

Inheritance between Ink classes is translated to their Java counterparts. If InkBase extends InkSuper, then in Java the behavior interface InkBase extends InkSuper, the state interface InkBaseState extends InkSuperState, the state inner class InkBaseState.Data extends InkSuperState.Data, and the behavior class InkBaseImpl extends InkSuperImpl.

When an Ink class doesn’t wish to change its superclass’s structure or behavior, parts of the described Java representation can be omitted, and the IVM instantiates the state inner class or behavior class of the superclass accordingly. In particular, this applies to AOM scenarios, where new classes are introduced to a running application. Instantiation of Ink objects is controlled by the IVM. When the IVM is requested to create an Ink object, it examines the class definition and determines which Java classes should be instantiated (in our example, InkBaseState.Data and InkBaseImpl). The decision is based on the following properties of InkClass (see Listing 1):

- **javaMapping** – an enumeration indicating the class’s corresponding Java elements. Each of the three constructs (state interface with inner class, behavior interface, behavior class) can be omitted, resulting in the following possible values:
  - **State_Behavior_Interface** – all Java constructs exist,
  - **State_Behavior** – there is no Java behavior interface,
  - **State_Interface** – no behavior class (used by abstract DSL classes),
  - **Behavior_Interface** – no state interface/inner class,
  - **Only_State** – no behavior interface/class,
  - **Only_Behavior** – no behavior interface, no state interface/inner class,
  - **Only_Interface** – no behavior class, no state interface/inner class (used by abstract classes),
  - **No_Java** – all the Java constructs are inherited from the superclass (used in AOM scenarios).
- **javaPath** – denotes the relative Java package of the Java classes and interfaces.

Java objects are created via Java reflection, and then combined using dependency injection: the state inner class instance holds a reference to the state interface in its behavior field, and the behavior instance has a reference to the state inner class instance in its state field. Other fields are similarly injected (e.g., the behavior’s context field is injected with the Context object that created it), and properties are initialized with values from an Ink instance, as well as default or final values when instantiating an “empty” instance. At the end of the instantiation process, the behavior class’s afterStateSet() method is called, to allow for application initialization logic.

Trait classes have the same basic structure. In addition, they contain a target field, which holds a reference to the state inner class instance of the target. This is done in order to have the same level of access to the property values as the behavior class has.

### 3.4 InkObject

The InkoObject interface provides the basic client API to work with Ink objects (Listing 3). A client holding an InkObject can obtain the object’s class, mirror and traits (getMeta(), reflect(), and asTrait(index) respectively). The cloneState() method returns a copy of the state object.
The state, value is -ies etc. once of sents the traits instances (.). The metadata

```
public interface InkObjectState extends Proxiable, Proxiable{ 
    public InkClass getMeta();
    public <M extends Mirror> M reflect();
    public <T extends Trait> T asTrait(byte index);
    public InkObjectState cloneState();
    ...
}
```

```
public class InkObjectImpl<S extendsinkObjectState> implements InkObject {
    private S state = null;
    public final <M extends Mirror> M reflect() {
        return state.reflect();
    }
    public Trait asTrait(byte aspect) {
        return state.asTrait(aspect);
    }
    ...
}
```

```
public interface InkObject extends Proxiable
```

```
public class InkObjectImpl {
    ...
}
```

```
Listing 3. InkObject API.
```

The InkObjectImpl class implements InkObject by delegating calls to its state. When implementing application specific code, a subclass of InkObjectImpl can add business logic code that depends on the state. The state could originate from source code, or be altered at runtime.

The InkObjectState interface provides the API for working with DSL elements structures. The getBehavior() method returns the corresponding behavior object (an instance of InkObjectImpl or its subclass). The getMeta() method returns a trait of the DslFactory that created the state object.

The InkObjectState.Data inner class implements the InkObjectState interface. It holds its data in an object array (myData) and its metadata in private members (id, mySuper, myClass, owner). Some of these variables are exposed via the InkObjectState interface and some by the Mirror interface (Listing 5). The metadata provides navigation capability in complex object graphs, the ability to obtain an object's super-object, its owner (the Ink object that contains this object), etc.,

The getValue(index), setValue(index, value) methods are used to retrieve or set a property value by its location in the myData array. When a property value is an Ink object, the getter method returns the behavior (Impl) of the value rather than its state, thus promoting information hiding and encapsulation.

The generated state interfaces and inner classes extend the InkObjectState/InkObjectState.Data hierarchy. The generated getters and setters invoke getValue(...) and setValue(...), providing a type safe and readable way to access the state. For example, for a string property called P1, the generated state class will contain a numeric constant p_p1 indicating this property’s index in the data array, as well as the following methods:

```
public String getP1() {
    return (String) getValue(p_p1);
}
```

```
public void setP1(String value) {
    setValue(p_p1, value);
}
```

### 3.5 InkClass

The Inkclass interface exposes the newInstance(...) method (Listing 4), which is implemented by InkClassImpl and delegates to its ObjectFactory.

InkClassState provides setter and getter methods to class structure, such as operations, properties etc. These methods are implemented by InkClassState.Data inner class, which delegates the calls to the getValue(...), setValue(...) methods in InkObjectState.Data.

### 3.6 Traits

The Personality interface is responsible for traits weaving and instantiation. On class creation the bind(...) method is invoked, which adds the trait’s injected properties to the class structure using the MOP API. When a client invokes the asTrait(...) method on InkObject O1, the adapt(...) method on Personality is invoked, instantiating a new trait instance with O1 as its target. To avoid repeated creation of traits instances, those instances are cached by the target Ink objects.

### 3.7 Reflection API

Ink’s reflection API is exposed through the Mirror and ObjectEditor hierarchies. The Mirror interface provides introspection capabilities, such as getting the object’s class (getClassMirror()), owner (getOwner()), and super
The Eclipse plugin uses the reflective capabilities of Ink to generically validate Ink class definitions, verifying that the Java constructs conform to their Ink counterparts. Auto-completion and navigation are also done in a generic fashion, so that there’s no need to re-generate the plugin (as is commonly required in other DSL workbenches). The plugin also automatically generates the state interfaces and inner classes.

We aim at providing the same level of usability in the Ink IDE as in Java. In the future, we intend to use OSGi capabilities (utilizing Eclipse’s built-in Equinox OSGi framework) to let DSL developers refine the generic nature of the Ink IDE without tool re-generation.

4. EVALUATION
To evaluate Ink and the development style it advocates, we re-implemented the AOM example introduced by Yoder et al. [10]. This example describes a video store, which rents out videos to customers. Each movie has a number of properties, such as a title and a MPAA rating, and there can be many physical tapes of each movie. A videotape can be rented to a customer.

Figure 4 is a class diagram of the video store example. A Movie is a meta-class, which defines the title and rating properties.
Figure 4. Example model, classes and instances

Metaclass id="Movie" class="ink.core:InkClass"
    super="ink.core:InkClass"
    {
        java_path ""
        java_mapping "State_Behavior_Interface"
        properties {
            property class="ink.core:StringAttribute"
            name "title"
            mandatory true
        }
        properties {
            property class="ink.core:StringAttribute"
            name "rating"
            mandatory true
        }
    }

Class id="Videotape" class="Movie"
    super="ink.core:InkObject" abstract=true
    {
        java_path ""
        java_mapping "State_Behavior_Interface"
        properties {
            property class="ink.core:BooleanAttribute"
            name "isRented"
            default_value false
        }
        properties {
            property class="ink.core:Reference"
            name "renter"
            mandatory false
        }
    }

Listing 6a. Movie and Videotape structure in Ink.

public class MovieImpl<S extends MovieState>
    extends InkClassImpl<S> implements Movie {
    public String getTitle() {
        return getState().getTitle();
    }
    ...
}

public interface Videotape extends InkObject {
    public boolean canRent(Customer customer);
    ...
}

public class VideotapeImpl<S extends VideotapeState>
    extends InkObjectImpl<S> implements Videotape {
    public boolean canRent(Customer customer) {
        return ! getState().getIsRented();
    }
    ...
}

Listing 6b. Movie and Videotape behavior in Java.

Class id="Terminator" class="Movie"
    super="Videotape"
    {
        java_mapping "No_Java"
        rating "R"
        title "The Terminator (1984)"
    }

Class id="Spiderman" class="Movie"
    super="Videotape"
    {
        java_mapping "No_Java"
        rating "PG-13"
        title "Spider-Man (2002)"
        properties {
            property class="ink.core:BooleanAttribute"
            name "hasSubtitles"
            mandatory true
        }
    }

Listing 6c. Specific Videotape classes in Ink.
Object id="TerminatorTape1" class="Terminator" {
  isRented true
  renter ref="Customer2"
}
Object id="SpidermanTape1" class="Spiderman" {
  isRented true
  renter ref="Customer1"
  hasSubtitles true
}
Object id="KillBillTape1" class="KillBill" {
  isRented false
}


@Test
public void testVideoStore() {
  Context context = InkVM.instance().getContext();
  Videotape terminatorTape1 = context.getObject("example.videostore:TerminatorTape1");
  Videotape terminatorTape2 = context.getObject("example.videostore:TerminatorTape2");
  Videotape spidermanTape1 = context.getObject("example.videostore:SpidermanTape1");
  Videotape killBillTape1 = context.getObject("example.videostore:KillBillTape1");

  CustomerState customer1 = context.getState("example.videostore:Customer1");
  CustomerState customer2 = context.getState("example.videostore:Customer2");
  assertTrue(terminatorTape1.canRent(customer1));
  assertTrue(!terminatorTape2.canRent(customer1));
  assertTrue(!terminatorTape1.canRent(customer2));
  assertTrue(killBillTape1.canRent(customer2));
  assertTrue(!terminatorTape1.canRent(customer1));
}

private Videotape createTape(String videotapeId) {
  Videotape newTapeState = InkVM.instance().newInstance(videotapeId);
  newTapeState.setIsRented(false);
  newTapeState.setRenter(null);
  return newTapeState.getBehavior();
}

@Test
public void createMovie(String id, String title, String rating) {
  Context context = InkVM.instance().getContext();
  InkObjectState videotape = context.getState("example.videostore:Videotape");
  ObjectEditor dynamicVideotapeEditor = videotape.reflect().edit().createDescendent();
  dynamicVideotapeEditor.
    setPropertyValue(InkClassState.p_java_path, "java_mapping:State_Behavior")
    setPropertyValue(InkClassState.p_java_mapping, "State_Behavior")
    dynamicVideotapeEditor.setPropertyValue(MovieState.p_rating, "PG");
  videotape.setRenter(null);
  videotape.setIsRented(false);
  videotape.setCanRent(CustomerState.renter.ref("Customer1").getAge() > 19);
}

Listing 10. Testing the videotape model.
 Videotape is an abstract class of the Movie metaclass. It defines properties to be supplied by its instances, such as isRented and renter. Specific movies (e.g. "Spiderman") are represented as classes that extend the Videotape class and supply values to the title and rating properties. Physical tapes are represented as instances of those classes.

Listing 6a shows the Ink representation of Movie and Videotape. The Movie metaclass is an instance of InkClass, and the Videotape class is an instance of Movie and a descendent of InkObject.

The java_mapping property defines which Java counterparts the given class has. Movie and Videotape have the full Java structure – state class (MovieState, VideotapeState), behavior interface (Movie, Videotape) and behavior implementation class (MovieImpl, VideotapeImpl). Listing 6b shows the corresponding Java code. Generated code (state classes) is omitted.

The specific movies, represented by classes that extend Videotape, don’t need specific behavior. The generic behavior in VideotapeImpl can be used. Thus their Java mapping is defined as NoJava (Listing 6c), at runtime their state and behavior will be represented by those of their superclass (VideotapeState and VideotapeImpl). As such, they are perfectly suited for AOM implementation. Defining a new movie title doesn’t require writing, editing or generating Java code or byte-code, and is well suited for non-programmers, especially when aided by an IDE’s auto-completion feature. Note that the Spiderman class modifies the inherited structure by adding the boolean property hasSubtitle.

In a more complicated scenario, classes can be created at runtime from AOM definitions, stored as metadata and edited via a GUI. A system can be developed and deployed with the build-time model (Movie and Videotape), and specific movies may be defined at runtime by means of a meta-object protocol. Listing 8 shows how a new class is created in the ObjectEditorImpl class. This method is invoked by the createMovie(...) method in Listing 10. After the values are set, the new class is registered in the Ink VM, and from that moment it becomes fully equivalent to classes that were written in the IDE.

As time goes by, a new requirement arises: restricted movies should be classified as such, and for those movies the renter’s age should be verified. The updated model and behavior appear in listings 7a and 7b. The RestrictedMovie metaclass sets the inherited rating property to a final value “R”, and introduces a new property: minimumAge. RestrictedVideotapeImpl overrides the canRent() method and compares the renter’s age with the minimum rental age as defined by its metaclass.

Listings 9 present several terminal instances while in listing 10 a simple JUnit test showing client code that works with the videostore model.

5. CONCLUSION

In this work we show how an AOM system can be developed using two languages: the Ink language for structure, and the Java language for behavior, avoiding the dual-type problem. The very idea of having two OOP languages as a development style may seem at first sight counter intuitive for a solution to the two type systems problem. However, the simple mapping between structural and behavioral classes, enforced by the IDE, causes developers to perceive the structural language as the hosting language and thus no semantic gap is created.

We also believe that introducing a generic language solution for AOM systems will help reducing the overhead of developing supporting infrastructures. Since libraries and tools developed for Ink can be shared and reused by many applications.

6. REFERENCES


http://www.omg.org/mof/


