

# Modeling Behavior with Personalities

**Luis Blando**

Operations Systems Lab.  
GTE Laboratories, Inc.  
40 Sylvan Road, MS-40  
Waltham, MA 02454 USA  
+1 781 466 3296  
lblando@gte.com

**Karl Lieberherr**

College of Computer Science  
Northeastern University  
360 Huntington Avenue  
Boston, MA 02115 USA  
+1 617 373 2077  
{lieber,mira}@ccs.neu.edu

**Mira Mezini**

## ABSTRACT

Decoupling behavior modeling from a specific inheritance hierarchy has become one of the challenges for object-oriented software engineering. The goal is to encapsulate behavior on its own, and yet be able to freely apply it to a given class structure. We claim that standard object-oriented languages do not directly address this problem and propose the concept of *Personalities* as a design and programming artifice to model stand alone behavior that embodies what we have termed *micro-framework* style of programming. Allowing behavior to stand alone enables its reuse in different places in an inheritance hierarchy. The micro-framework style ensures that the semantics are preserved during reuse. We present a potential Personalities implementation by extending the Java programming language.

## Keywords

functional vs. data decomposition, behavior modeling, behavior reuse, Java, frameworks

## 1 INTRODUCTION

If we take a bird's eye view of any given software system, we find that its sole purpose is to perform a *function* for its user. The "black-box" metaphor attests to exactly this fact. A given software application has a set of inputs, and produces a set of outputs. At a finer granularity, we find that we can decompose a large system into smaller functions that collaborate to produce the desired behavior. This strict *functional decomposition* fueled the *structured programming* approach to software development.

Doing structured programming means decomposing the functionality of the entire system into many functions, smaller in scope, and with clear interfaces. A "privileged"

function initiates the program's execution and maintains the flow of control, yielding to sub-functions as needed. It is easy to see "what the program is doing".

The main problem with strict functional decomposition, however, resides in the fact that the data over which the functions operate are spread throughout the program, with no explicitly guaranteed integrity. Object-oriented programming grew to address this problem. In addition to decomposing a system into functions, we find groups of data that share a set of characteristics and group them in new entities that we term *classes*. The functions are then mapped onto these classes. Explicit guarantees are set forth with respect to the integrity of the data an *object* (that is, an instance of a class) contains.

After initial analysis, we are left with both a functional and a data decomposition of the problem domain. The outcome of functional decomposition during the analysis and design phases of the software lifecycle is a list of roles and responsibilities. Some object-oriented methodologies, such as [1], are specifically oriented to the discovery and modeling of these roles while others are not. In the latter methodologies, roles can often be found when inspecting the dynamic views of your system (i.e., sequence diagrams, use-case diagrams, object-interaction diagrams, etc.)

Our position is that it is not only convenient but necessary to use role names (instead of class names) in the dynamic views of the system. Problem domain experts usually speak in these terms, thus allowing the software engineer to clearly validate her (role-based) model against these experts. From our experience in industry we have found that one of the first hurdles that a designer encounters when modeling a system is how to allocate these roles to different objects. Unfortunately, the mapping of the functional-space onto the data-space is not one-to-one or many-to-one, but rather many-to-many. In other words, a given function might be required of more than one data abstraction. In this paper, we'll call these functions *popular*.

Coming from an era where the duplication of the same data in different places of a program had created havoc for software engineers, the object-oriented camp naturally

leaned towards modeling the class hierarchy to closely resemble a data decomposition hierarchy. This gave practicing software engineers the “when in doubt, follow the data” rule of thumb. Practically, this means that early on, somewhat arbitrary decisions need to be made when assigning popular behavior to classes resulting in one of two possible consequences. Either spurious associations between classes are introduced or behavior code needs to be duplicated. The former results in the case of a class that needs to play the same role but has no domain-based relationship with the class that ends up implementing the role. If we aim to preserve a single implementation, the second class will need an association with the role-implementing one to make use of that implementation. The latter consequence arises when an association between the two classes playing the same role cannot be supported, and thus the code for the role needs to be duplicated.

In this paper, we show the drawbacks resulting from the fact that object-oriented languages are geared mostly towards supporting the data decomposition approach and lack appropriate support for expressing the functional decomposition of application domains. This motivates the need for linguistic constructs that would allow roles to stand alone throughout the analysis, design, and coding phases as a solution to this problem. We propose the concept of *Personalities* as a linguistic artifact to be added to the standard object-oriented concepts for explicitly encapsulating roles from the functional decomposition at the implementation level. A role (personality) is attached to a class via a *personifies* clause. In order to be a valid personification of a personality, a class must obey well-defined rules. The same holds for the clients of a personality.

The remainder of the paper is organized as follows. In the following section we illustrate the issues related to the lack of appropriate linguistic constructs for expressing the functional decomposition by means of an example. In section 3, the concept of *Personalities* is presented as the solution to these problems. Section 4 discusses work in progress. Related work is discussed in section 5. Section 6 concludes the paper.

## 2 ISSUES IN MODELING POPULAR FUNCTIONS

As a very simple example, assume the problem domain has to do with modeling the animal kingdom. Early on you detect that most animals perform the same basic functions (i.e. walk, fly, swim) in pretty much the same manner. These would be what we’ve termed “popular” functions. A (partial) functional decomposition of the problem domain is shown Figure 1.

From a design perspective, one would like to implement the popular functions (i.e. fly, walk, swim) using their sub-functions (i.e. takeoff, ascend, etc.) and then require subclasses to override these sub-functions. At this time in the design process, the software engineer can (and should,

in the authors’ opinion) fix the semantics of, for example, the `Fly()` function. One such semantics, using Java, follows:

```
void Fly(int miles, int altitude) {
    Takeoff();
    for (int a=0; a < altitude; a++) Ascend();
    while(miles--) Flap();
    for(int a=altitude; a > 0; a--) Descend();
    Land();
}
```

Meanwhile, analysis also yields a (simplified) partial data decomposition, shown in Figure 2. Abiding by the “follow-the-data” rule of thumb means that the class diagram will more than likely follow this data decomposition. Mapping the popular animal functions, however, presents the problems discussed previously, as more than one class needs to provide the same function(s). For example, both whales and penguins swim, mosquitoes and locusts fly, while ants and locust walk.

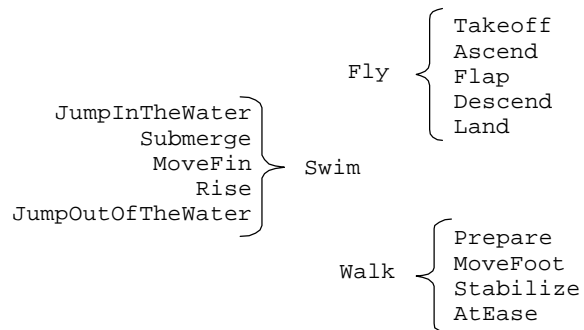


Figure 1: Animal kingdom functional decomposition

We can always push the mappings of these functions up the class hierarchy, even all the way up to the `Animal` class. Unfortunately, this is incorrect from a design perspective, since not all animals perform all functions. Another approach would be to duplicate the code for the popular behaviors (i.e. `Fly()`, `Walk()`, and `Swim()`) at the classes where they are needed. This is, however, undesirable from a maintenance perspective.

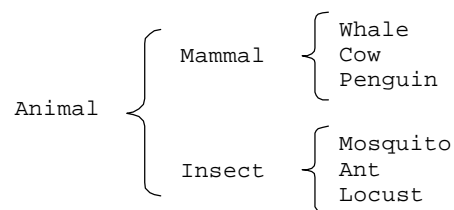


Figure 2: Animal kingdom data decomposition

Even if you could somehow manage to share one single implementation (and thus obtain a guaranteed unique semantics) of these popular behaviors, the problem of correctly allocating functionality to the classes still remain. At a programming level, you need to advertise that `Whale` supports `Swim()`, `Cow` supports `Walk()`, and `Penguin` supports both `Walk()` and `Swim()`. This allows for the

code of the clients of Whale, Penguin, and Cow to be type-checked at compile time.

The concept of *interfaces*, as understood in the Java programming language helps alleviate this latter problem but does not help with the former. Interfaces allow a class to advertise the implementation of selected methods by declaring its compliance with arbitrary sets of method signatures called interfaces. The popular as well as the smaller-granularity functions would be part of the interface. An interface contains only method signatures. Therefore, each class that implements the interface must provide the implementation of the popular function. This opens up the possibility that the semantics given to the `FLY()` function differ in the various implementations, greatly jeopardizing the client systems that depend on the specific (intended) semantics of a given interface.

Another approach to this problem would be to model the popular behaviors as abstract classes, using multiple-inheritance to extend from them as needed. As an example, code for modeling the flying and swimming behaviors as abstract classes is given below. Sample code for the class Penguin is also shown to illustrate how it could be modeled through multiple inheritance. The “complete” multiple-inheritance hierarchy for modeling our animal kingdom is presented in Figure 3.

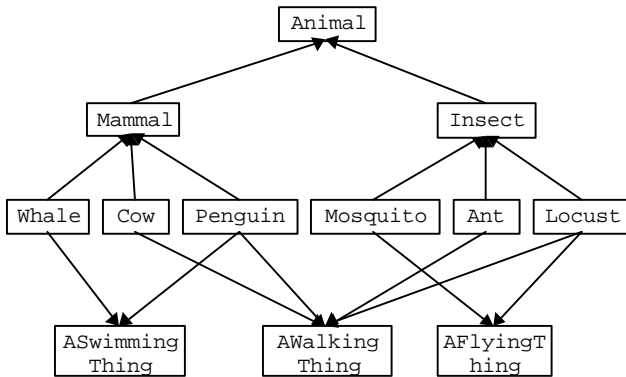


Figure 3: Animal kingdom using multiple inheritance

```
// multiple-inheritance modeling of Penguin
// sample code. Pseudo-Java used.
class AFlyingThing {
    final void Fly(int miles, int altitude) {...}
    abstract void Takeoff();
    abstract void Ascend();
    abstract void Flap();
    abstract void Descend();
    abstract void Land();
}
class ASwimmingThing {
    final void Swim(int miles, int depth) {...}
    abstract void JumpInTheWater();
    abstract void Submerge();
    abstract void MoveFin();
    abstract void Rise();
    abstract void JumpOutOfTheWater();
}
```

```
class Penguin extends AFlyingThing,
                    ASwimmingThing {
    void Takeoff()      {...}
    void Ascend()      {...}
    void Flap()        {...}
    void Descend()     {...}
    void Land()        {...}
    void JumpInTheWater() {...}
    void Submerge()    {...}
    void MoveFin()     {...}
    void Rise()        {...}
    void JumpOutOfTheWater() {...}
}
```

This alternative, while the closest in spirit to our goals, has a number of implications. First, the semantics of multiple-inheritance have traditionally been ambiguous and are not widely understood. Second, not all programming languages support multiple-inheritance. Specifically, Java does not. Third, and most important in our opinion, is the fact that the classes that embody the behaviors (i.e. `AFlyingThing`, `ASwimmingThing`) do not have any constraints in the implementation of their respective popular functions (i.e. `Fly()`, `Swim()`). While some languages (i.e. Java) do prevent the implementation of the popular behavior from being overridden, there are no constraints on the sub-functions that are meant to be defined by subclasses (i.e. `Takeoff()`, `Ascend()`, etc.) and are in turn used to define the popular behavior. If the underlying sub-functions are modified in an uncontrolled way by `Penguin` or one of its subclasses, the integrity of a system that relies in `AFlyingThing`'s `Fly()` semantics, for instance, would be severely compromised.

The multiple-inheritance solution is based on programming with artifacts that are not part of the problem domain. For instance, there is no concept of an `AFlyingThing` object in the application domain. Flying is merely a behavior that can be performed by several abstractions in the application domain. We are artificially creating new classes out of the need to turn behavior into first-class objects.

To summarize, we believe that the ideal situation would be one in which we are allowed to model the behavior (in a functional-decomposition-biased manner) and the class hierarchy (biased towards data decomposition) independently, and then freely apply behavior to classes. Such an approach would yield the most reuse and, more importantly, would allow us to design the program without unwanted artifacts that are not part of the problem domain.

### 3 MODELING WITH PERSONALITIES

For this purpose we introduce *personalities* as a language construct to encapsulate high-level, micro-framework style behavior. Our concept lies in the “middle-ground” between abstract classes and interfaces, as Java understands them. Personalities are close in spirit to abstract classes. However, unlike abstract classes, personalities provide guaranteed semantics to the client systems that use them.

## Syntax and Usage

Defining a personality is similar to defining a class, with a few added keywords. For example, the `Flier` personality can be defined as follows (new keywords are underlined>):

```
// Flier.pj
personality Flier {
// upstream interface. Must implement here.
public
void Fly(int miles, int altitude) {
    Takeoff();
    for (int a=0; a < altitude; a++) Ascend();
    while(miles--) Flap();
    for(a = altitude; a > 0; a--) Descend();
    Land();
}
// downstream interface. Don't impl here.
di void Takeoff();
di void Ascend();
di void Flap();
di void Descend();
di void Land();
private some_other_function() {...}
}
```

A personality definition consists of three basic elements:

- The **upstream interface**, made up of all the member functions that clients of this personality can access (one or more). These encapsulate what we have been calling “popular” behavior. It is here where the personality adds value to the design, since it provides a single and unique implementation of such behavior.
- The **downstream interface**, composed of only signatures for functions prepended by the `di` keyword. These are the functions that personifying classes must implement. Clients of the personalities cannot access these methods.
- Any **other functions** that the personality might need to implement the upstream interface. These functions are not visible to either clients or personifying classes.

When a class decides to personify a given personality, it needs to declare its intent (via the `personifies` clause), as well as provide the methods specified in the downstream interface. For example, a (trivial) definition of the `Mosquito` class could be:

```
public class Mosquito extends Insect
    personifies Flier
{
    int height = 0;
    public Mosquito() {}
    private boolean stingingSomebody() {...}
    private void jumpInTheAirAndStartFlapping() {}
    // downstream interface implementation
    public void Takeoff() {
        while ( stingingSomebody() ) {
            // wait for some time
        }
        jumpInTheAirAndStartFlapping();
    }
    public void Ascend() {...}
}
```

```
public void Flap() {...}
public void Descend() {...}
public void Land() {...}
}
```

Figure 4 shows the different classes, personalities, and their relationships for the running example. Personifies relationships, as well as personalities, are drawn in dashed lines.

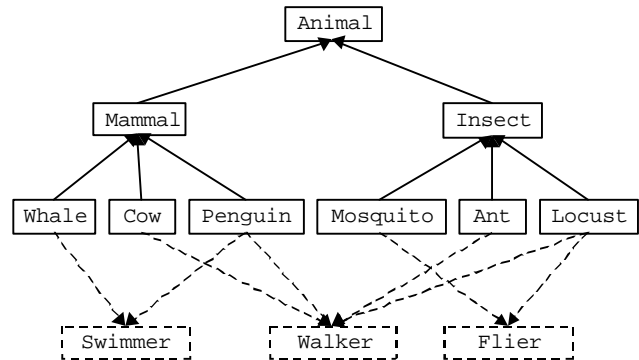


Figure 4: Animal kingdom using personalities

The link from the personalities to the classes that aim to personify them is through small-granularity functions. For instance, `Takeoff()`, `Ascend()`, etc., are examples of such functions. We call the classes that embody personalities the *personifying* classes/objects. A personality lies in between the client code that makes use of it and the class that embodies it. It is connected to the client code by the “upstream” interface, and to the personifying classes by means of the “downstream” interface. The personality expects from the personifying class the implementation of the lower-level functions. In turn it provides clients with the high-level functions in the upstream interface. This resembles the mental picture of the level of abstraction and the granularity diminishing as we move from client’s code, to personality’s code, to personifying classes’ code. Personalities restrict how they present themselves upstream, that is, to the user of the system. The idea is to only expose the popular behavior in the personality (i.e. `Fly()`) and disallow the use of any of the smaller granularity functions (i.e. `Takeoff()`) by the clients, as illustrated in Figure 5.

Thus, Personalities can be thought of as interfaces enhanced with the ability to implement behavior, plus the necessary programming language plumbing to make it happen. The underlying idea is to try to model *frameworks* [9] on a small scale. Frameworks usually encapsulate behavior (control of execution flow) for an entire system or application. Analogously, personalities do the same for individual classes and specific behavior. For this reason, we call programming with personalities a *micro-framework* style of programming. The entry points to a framework are usually the redefinition or creation of subclasses to implement certain methods. Similarly, the link from

personalities to the classes that aim to personify them is through small-granularity functions.

### Following the “Law of Personalities”

Personalities must follow a certain set of constraints in order to provide semantic value to the developer of a system. Abiding by certain rules guarantees the developer the reusability and, to some degree, the correctness of the design. It is partially in these rules where personalities improve over abstract classes. We consider the following set of requirements for fully exploiting the power of personality programming. The compiler needs to make sure that these are met.

1. The downstream interface must be a set of pure abstract functions, with clearly identified semantics.
2. Only basic object types (i.e. string, int, Vector<string>, etc.) should be passed in parameters and returned from functions in the downstream interface.
3. Clients of the personality (i.e. the “upstream” objects) must not use the links to the personifying class (i.e. the downstream interface). These clients should only access the high-level behavior functions the personality provides.
4. The links to the upstream objects should be semantically distinct from all the functions to be implemented by the personifying class.
5. The implementation of the popular functions must be protected against changes by personifying classes.
6. The implementation of the popular functions is allowed to use the smaller-granularity functions to communicate with the personifying class and nothing more.

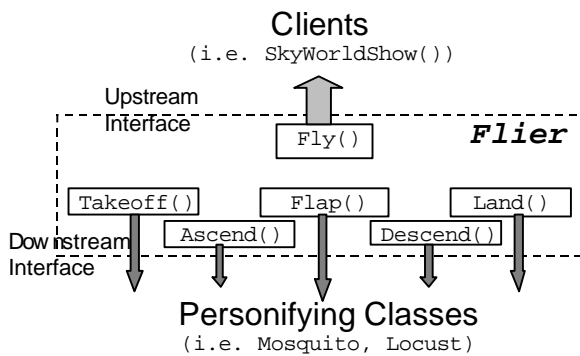


Figure 5: Anatomy of the Flier personality

Requirement #1 basically makes personalities uninstanciable on their own. The rationale is that a personality cannot possibly provide a “default” implementation. We are aiming at encapsulating behavior that might be reused by a number of distinct classes. Therefore, the “default” versions for the downstream

interface methods might vary greatly in different contexts and thus a common implementation does not make sense.

It is also important for the semantics of these functions to be clearly understood and defined. These functions are the weak link with regards to the semantic integrity of the entire system, since they are the ones implemented by the personifying objects. It is thus essential for them to be easily understood by the programmer.

For instance, a downstream interface that is ambiguously defined with respect to its return value format, such as:

```
// compute and return today's date
String Today();
```

would not be of much help to the developer personifying this personality, since it provides no clue about the format the answer must be in. Checking that the personality does not implement the downstream interface functions is simple. Automatically making sure the semantics of those functions are clear, on the other hand, is still an open problem.

A compiler can easily check requirement #2, which ensures an attainable minimum set of pre-required knowledge in order for any given object to personify a given personality. We restrict personalities from making the parameters and return values of the downstream interface methods user-defined types, since this will imply that the personality would forever need to be deployed with an implementation for the user-defined types it uses. We require restricting these signatures to the lowest common denominator for the given programming language. For instance, this rule hinders the programmer of a personality from the following declaration in the downstream interface:

```
MyDateClass Today(); // return today
```

This declaration couples the personality with the user-defined type MyDateClass and damages its reusability. Using Java’s “standard” Date class, the following would be preferable.

```
// return Java's Date for Today
Date Today();
```

Requirement #3 attempts to make personalities the clear boundary between the clients of the personality (i.e. the upstream objects) and the personifying classes (i.e. the downstream objects). This aims at providing a specific layer of design reuse at the personality level. In other words, by restricting the clients to only use the popular behaviors provided by the personality, we are guaranteeing the semantics of the client’s use of the personality. Once again, the compiler can easily enforce this requirement.

For example, a class that needs to interact with a Swimmer can only call Swim(int miles, int depth) and not any of the other functions (i.e. Submerge(), MoveFin(), etc). The di keyword in the personality’s definition is aimed at helping the compiler and the user of the personality to clearly discern what is allowed and what is

not. In the following sample code, both correct and incorrect use of a personality's interface are illustrated:

```
// SeaWorldShow() is a client of Swimmer pers.
void SeaWorldShow(Swimmer shamuorflipper) {
    shamuorflipper.Swim(10,10); // ok, ui used
    shamuorflipper.Submerge(); // error, di used
}
```

Simply creating a popular “wrapper” for each downstream function can circumvent requirement #3. Therefore, requirement #4 calls for substantial semantic difference between the upstream and downstream functions in order to avoid what we've termed the “delegation syndrome”. For instance, a roundabout way to get access to the Submerge() function would be to redefine the Swimmer personality and add to it the following:

```
Personality Swimmer {
    ...
    public void ProxySubmerge() { Submerge(); }
}
```

Checking for violations to this rule, however, is far from trivial and we cannot expect current compilers to enforce it.

Requirement #5 aims at making sure that the personifying classes do not change the originally intended semantics for the personality. In other words, if we would allow a LazyMosquito class to do something like:

```
Class LazyMosquito extends Insect
    personifies Flier
{
    ...implementation of downstream interface
    // we shouldn't redefine Fly(...)!
    public void Fly(int miles, int altitude) {
        Takeoff();
        for(int a = 0; a < altitude/2; a++) Ascend();
        while(miles--) Flap();
        for(int a = altitude; a > 0; a++) Descend();
        Land();
    }
}
```

the semantic integrity of the system would be compromised, since this special mosquito only flies at about half the altitude as what the personality promises it would. Furthermore, since this particular implementation of Fly(...) contains a logic error, its effects are undefined. Therefore, the compiler should make sure that personifying classes implement the downstream interface and any other private functions, but never the upstream interface.

Finally, requirement #6 makes personalities follow their own advice by requiring that all communication with the personifying class be restricted to the functions defined in the downstream interface. This aims at making sure that the set of functions is enough to support the semantics of the personality, and trigger the discovery of new ones if not. It also forces personalities and personifying classes to have only one meet point, namely the downstream functions. The same argument regarding the communication between clients and personalities set forth in requirement #3 is valid regarding the implementation of the personality's high-level behavior themselves. For example, allowing

```
personality Flier {
    ...
    void Fly(int miles, int altitude) {
        jumpInTheAirAndStartFlapping(); // not in
        // downstream interface !
    }
}
```

might restrict the applicability of this personality only to Mosquito and its subclasses (this is assuming, of course, that the proper method visibility allows this code to be accepted by the compiler in the first place!).

### Implementation Details

Implementing personalities in a language that allows multiple inheritance, such as C++, would be somewhat simple, with the personalities compiler concentrating mostly in enforcing the checkable requirements set forth in the previous section. Java, on the other hand, does not support multiple inheritance. It does, however, provide explicit support for interfaces, as has been explained before. We have, therefore, made use of the interface functionality as an implementation tool.

The architecture of our system assumes that the developers program solely in *Personalities/J*, for “Personalities in Java”. A Personalities/J compiler then translates their programs into standard Java programs at which point a Java compiler takes over. As a programming language, Personalities/J is loosely defined as a superset of Java in terms of syntax and semantics, with the additions of the new elements in the language that we have previously described.

Instead of presenting Personalities/J's formal semantics, we will use an example to describe the implementation details. We have Flier.pj (as shown previously in page 4) and Walker.pj for the Walker and Flier personalities, respectively; and Locust.pj as a personifying class for both

```
// Walker.pj
personality Walker
{
    public void Walk(int distance) {
        Prepare();
        while( distance-- ) {
            for(int f=0; f < NumberOfFeet(); f++)
                MoveFoot(f);
            Stabilize();
        }
        AtEase();
    }
    di void Prepare(); // downstream
    di int NumberOfFeet(); // interface
    di void MoveFoot(int);
    di void Stabilize();
    di void AtEase();
}
// Locust.pj
public class Locust extends Insect
    personifies Walker, Flier
{
    // ... details about Locust class ...
    // as required by Walker.pj
    void Prepare() { ... }
```

```

int NumberOfFeet() { ... }
void MoveFoot(int no) { ... }
void Stabilize() { ... }
void AtEase() { ... }
// as required by Flier.pj
void Takeoff() { ... }
void Ascend() { ... }
void Flap() { ... }
void Descend() { ... }
void Land() { ... }
}

```

The Personalities/J compiler will generate:

- (a) the interfaces Walker.java and Flier.java which lay down the signatures for both the upstream and downstream interfaces of the Walker and Flier personalities, respectively
- (b) the Walker\_Ego.java and Flier\_Ego.java classes which contain the implementations of the upstream behavior of Walker and Flier personalities, respectively, and
- (c) the class Locust.java for the concrete animal class that personifies both a Flier and a Walker.

Sample generated code is shown below:

```

// Walker.java
interface Walker {
void Walk(int distance);
void Prepare();
int NumberOfFeet();
void MoveFoot(int feetno);
void Stabilize();
void AtEase();
}
// Flier.java
interface Flier {
void Fly(int miles, int altitude);
void Takeoff();
void Ascend();
void Flap();
void Descend();
void Land();
}
// Locust.java
public class Locust extends Insect
personifies Walker, Flier
{
// ... details about Locust class ...
// as required by Walker.pj
void Prepare() { ... }
int NumberOfFeet() { ... }
void MoveFoot(int no) { ... }
void Stabilize() { ... }
void AtEase() { ... }
// as required by Flier.pj
void Takeoff() { ... }
void Ascend() { ... }
void Flap() { ... }
void Descend() { ... }
void Land() { ... }
// compiler-generated code
Walker_Ego _walker = new Walker_Ego();
public void Walk(int distance)
{ _walker.Walk(this, distance); }
Flier_Ego _flier = new Flier_Ego();
public void Fly(int miles, int altitude)
{ _flier.Fly(this, miles, altitude); }
}

```

```

// Walker_Ego.java
public class Walker_Ego
{
public void Walk(Walker host, int distance) {
host.Prepare();
while( (distance-- > 0 ) {
for (int f=0; f<host.NumberOfFeet(); f++)
host.MoveFoot(f);
host.Stabilize();
}
host.AtEase();
}
}
// Flier_Ego.java
class Flier_Ego
{
public void Fly(Flier host, int miles,
int altitude) {
host.Takeoff();
for (int a = 0; a < altitude; a++)
host.Ascend();
while((miles-- > 0) host.Flap();
for(int a = altitude; a > 0; a--)
host.Descend();
host.Land();
}
}
}

```

In order to better understand the dynamics between these different classes, the following sequence diagram depicts the interactions between them when a client invokes the Walk() high-level behavior on a Locust object.

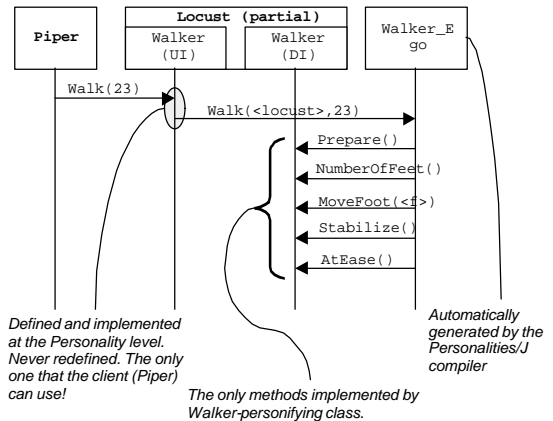


Figure 6: Sequence diagram for Walk()

#### 4 WORK IN PROGRESS

The current incarnation of Personalities/J provides only the basic framework with which to explore these concepts. We are working to extend and enhance these ideas.

First, we are investigating a way to allow an already-implemented class to personify a personality with little work. For instance, if we already have the class

```

class SpaceShuttle {
void EngageRockets() {...}
void ExitAtmosphere() {...}
void Orbit() {...}
void EnterAtmosphere() {...}
}

```

```

    void Land() {...}
}

```

we shouldn't need to define `Takeoff()`, `Ascend()`, `Flap()`, and `Descend()` so that they simply delegate to the respective `SpaceShuttle` methods. A simple name mapping mechanism would remove the need for these artifices to be created. For instance,

```

class SpaceShuttle personifies Flier
  with EngageRockets = Takeoff,
       ExitAtmosphere = Ascend, Orbit = Flap,
       EnterAtmosphere = Descend {
  // no changes to SpaceShuttle's code at all
  // Land() has same signature, so no mapping
}

```

Notice that such a mapping mechanism would also make it easier for a new class to personify two or more personalities with semantically equivalent downstream interface methods. For instance, if one personality calls for a `SaveToDB()` function and a different one calls for `PersistYourself()`, they both can be mapped to the same implementation code without having to create two implementations and having to explicitly delegate from one to the other.

A slight extension on this simple name mapping idea could further improve the usability of the personalities concept. If, for instance, you have purchased your ordering system and your data collection system from different vendors, you might get something like:

```

// from Vendor A (in Argentina, for instance)
personality FruitProducer {
  di boolean CheckStockHas(double kilograms);
}
// from Vendor B (in the US, for instance)
personality FreezerUser {
  di boolean ValidateProdQuant(double pounds);
}

```

in which case you cannot use the simple mapping technique explained before and are forced to do the conversion on your own, ending up with something like:

```

class TheRedAppleInc personifies FruitProducer,
                    FreezerUser {
  public boolean CheckStockHas(double kilograms)
  { /* implement behavior in metric system */ }
  public boolean ValidateProdQuant(double pounds)
  { return CheckStockHas( pounds * 0.454 ); }
}

```

Adding some new syntax and compiler support to provide “on-the-fly parameter conversion”, like

```

class TheRedAppleInc personifies FruitProducer,
                    FreezerUser with
                    CheckStockHas=ValidateProduce(<pounds>*2.2) {
  public
  boolean CheckStockHas(double kilograms) {...}
  // do not implement ValidateProduce() at all!
}

```

would allow slight semantic differences in “peer” personalities to be handled easily.

Second, we are studying making personalities a dynamic property of objects. As presented in this paper, classes’

attachment to personalities is determined at compile time. Making use of Java’s extensive introspection facilities, however, it would be possible to define a Personalities metamodel and use it to define all personalities, standardizing the personification process, as well as the introspection-based method dispatch, in a similar way as polymorphism works today.

Such a capability would allow a system’s objects to mutate throughout their lifespan, a goal that is elusive with current programming language technology. For example, it is quite reasonable for a `Person` object to transition from `Unemployable` to `Employed` to `LaidOff` at different times. Modeling these roles via inheritance is insufficient due to its compile-time nature. Dynamic Personalities would allow the same `Person` object to maintain its identity forever, while appearing as different personalities to other systems as its state changes.

Third, we are investigating and characterizing the relationship between Personalities and Adaptive Plug-n-Play components (APPCs) [17]. APPCs encapsulate behavior at a bigger scope, and can be thought of containing an entire slice of behavior at a system level (i.e. an entire collaboration between classes). We believe APPCs will be the “client” of Personalities and will make use of Personalities’ micro-framework behavior level semantic guarantees to implement their functionality.

We are also exploring the possibility of allowing personalities visibility to a larger set of classes with which to implement their micro-framework behavior. The interactions between these classes would be succinctly specified, following our previous work in the Demeter system [12,13]. Last, but certainly not least, we are thinking about different approaches to provide extensibility and composition of personalities.

## 5 RELATED WORK

One category of related works includes approaches that are based on using delegation to emulate modeling roles an object may play during its life, such as the work by LaLonde et al. on *Exemplar-Based Smalltalk* [11] and the work by Gottlob et al. on extending object-oriented systems with roles [6]. Both approaches support two kinds of hierarchies: class and role hierarchies (called exemplars in [11]). The main focus of these works is, however, on supporting dynamic modifications of an object’s behavior, as it undertakes/cancels certain roles and not on explicitly supporting functional decomposition. Artifacts that model roles, or exemplars, are strongly bound to a certain class in the inheritance hierarchy. As a result, it is not possible to apply the same behavior to different unrelated classes, as it is the case with, e.g., the `Walker` Personality being applicable to both `Ants` and `Cows`. Again, because of the focus on supporting evolving objects, there are no equivalent notions to the upstream and downstream interfaces of the Personalities.

On the other side, there are several works aimed at improving the expressiveness of the inheritance structure by relaxing the class-subclass relationships that could also support modeling stand-alone behavior that can be reused in several scenarios. This category includes the work on *mixin-based inheritance* [2,3], *contracts* [8], *mixin-methods* [14], *MixedJava* [4], *Rondo* [16], and *context relationship* [18]. These works share the fact that variations on a base behavior are modeled in stand alone artifacts called mixins in [2,3] and [4], contracts in [8], mixin-methods in [14], adjustments in [16], and context objects in [18]. These artifacts do not commit to any base behavior when defined. Rather, they refer to the base behavior by means of an (unbound) super parameter and the self reference. The individual approaches differ from each other on two main points: (1) the level at which the variation is specified – object vs. class level, and, (2) the time when variations can be applied – dynamically vs. statically.

From the perspective of this paper, the important point is that the variations are not coupled to a static inheritance hierarchy as with standard inheritance. One could use mixins to model high-level reusable functions, since classes and mixins can be freely arranged in inheritance chains. However, these approaches are lower-level with regard to modeling high-level popular functions as compared to Personalities. None of them provides for guaranteed semantics of the popular behaviors and for declaring the interface expected from the personifying classes. However, they provide more flexible behavior composition that could be used to implement Personalities instead of using delegation. In particular, Rondo and the context relationship approaches could be used in our future work on dynamic Personalities.

The work presented in [10] also considers the need for synthesizing object-oriented and functional decompositions. The visitor pattern [5] is considered as a technique for filling the gap. The visitor pattern could be used in our running example, as follows. First, each popular behavior will be modeled in a separate visitor class, with the individual visitor classes all being subclasses of an abstract Visitor class. The implementation of the popular behavior would be encoded in `visit()` messages. All animals must understand an `accept()` message taking a visitor object as a parameter. When the `accept()` message is invoked on an animal object with a visitor as a parameter, the animal object will invoke `visit()` to the visitor parameter, passing itself along the invocation. Thus a client wanting to invoke a popular function on a certain animal would create an instance of the visitor class for this popular function and call `accept()` on the animal with the visitor as a parameter.

There is a severe problem with this approach related to the fact that visitors are normal classes and thus do not have any notion of the downstream interface. Each visitor needs

to somehow declare to which types its popular behavior applies. It can not simply accept an object of the most general type `Animal` as the parameter of its visit method, since the compiler in a strongly typed language like Java would complain when “downstream” functions are applied to this object within the micro-framework of the popular function. In absence of a real downstream interface, each concrete visitor class would implement as many different `visit()` messages as there are concrete animal classes to which the popular behavior encoded by the visitor applies. For instance, there will be a visitor class for the Walker behavior, say `WalkerVisitor`. This will have a different `visit()` methods for `Cow`, `Penguin`, `Ant` and `Locust`, although the implementations of these messages are the same – each embodying the same micro-framework of the upstream message `walk()` in the `Walker` Personality. Not only is this solution awkward, but it also damages reusability, since popular functions are still strongly coupled to the data hierarchy. Adding new animal classes (data abstractions) and declaring them to personify an existing personality is impossible without changing the implementation of the popular functions.

The work on *subject-oriented programming* [7] aims at enabling the construction of object-oriented software as a sequence of collaborating applications, each providing its own *subjective view* of the domain to be modeled, and defined independently from the others. A *subject* is a collection of class fragments with each fragment providing only one subjective view of the “whole” data abstraction captured by the class. Personalities can serve for modeling these fragments, especially when enhanced with mechanisms for composing them that would enable to model the composition of fragments into subjects and of individual subjects into higher-level subjects.

In our own previous work, behavior is described by propagation patterns (in Demeter/C++ [12]) or adaptive methods (in Demeter/Java [13]), separate from specific classes. This separately specified behavior is later reused in many different class structures. Propagation patterns (or adaptive methods) are similar in spirit to personalities, they specify behavior for a family of classes and they both need to be mapped into specific classes. However, both propagation patterns and adaptive methods don't have the concept of a required and provided interface and they don't enforce the laws of personalities as described in this paper.

Our concept of upstream and downstream interfaces is very similar in spirit to that of *provided* and *required* interfaces in [15]. However, required interfaces refer to other program modules (ie. other interfaces), whereas a personality's downstream interface refers to a class that is part of the personified object itself. Furthermore, the different functions in the required set can be serviced from different modules in a system, whereas only one class must implement the entire downstream interface. We have

purposely kept a different nomenclature to emphasize the fact that [15] aims at defining an architecture whereas personalities work at a much smaller (class) granularity.

## 6 CONCLUSIONS

This paper proposes a small extension to the programming language Java that refines the interface concept to allow for partial implementations with some degree of semantic guarantees. The contributions of the paper are twofold. It provides a framework for independently modeling behavior and freely applying it to arbitrary inheritance lattices, and proposes a set of specific design and programming rules, some of them compiler enforceable, to enhance the semantic strength of the personalities concept and thus the final system.

## 7 ACKNOWLEDGEMENTS

We would like to thank Johan Ovlinger and the students of the Advanced OO Software class at Northeastern University for fruitful discussions. Also, we thank Tony Confrey of GTE Laboratories for his review of early drafts of this paper. This work has been partially supported by the Defense Advanced Projects Agency (DARPA), and Rome Laboratory, under agreement number F30602-96-2-0239. The views and conclusions herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, and the U.S. Government.

## REFERENCES

1. Andersen, Egil, and Reenskaug, Trygve. System Design by Composing Structures of Interacting Objects. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP) 1992*. Springer-Verlag, pp. 131-152.
2. Bracha, Gilad, and Cook, William. Mixin-based Inheritance. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) 1990*.
3. Bracha, Gilad, and Lindstrom, Gary. Modularity meets Inheritance. In *Proceedings of IEEE Computer Society International Conference on Computer Languages* (Washington, DC, April 1992), IEEE Computer Society, pp. 282-290.
4. Flatt, Matthew, Krishnamurthi, Shriram, and Felleisen, Matthias. Classes and Mixins. To appear in *Proceedings of the 1998 Principles Of Programming Languages (POPL) Conference*. January 1998 at San Diego.
5. Gamma Erich, Helm Richard, Johnson Ralph, and Vlissides John. Design Patterns. Elements of Reusable Software. Addison-Wesley, 1994.
6. Gottlob Georg, Schrefl Michael, and Roeck Brigitte. Extending Object-Oriented Systems with Roles. In *ACM Transactions of Information Systems*, Vol. 14, No. 3, July 1996.
7. Harrison William, and Ossher Harold. Subject-Oriented Programming (A Critique on Pure Objects). In *Proceedings of ACM Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) '93*. Sigplan Notices, Vol. 28, No. 10, pp. 411-428, 1993.
8. Holland, Ian. The Design and Representation of Object-Oriented Components. *PhD Thesis*. Northeastern University, 1993.
9. Johnson Ralph. Frameworks = (Components + Patterns). In *Communications of ACM*, Vol. 40, No. 10, 1997.
10. Krishnamurthi Shriram, Felleisen Mathias, and Friedman Daniel. Synthesizing Object-Oriented and Functional Design to Promote Reuse. In *Proceedings of ECOOP '98*, Lecture Notes on Computer Science, Springer Verlag, 1998.
11. LaLonde Wilf R., Thomas Dave, and Pugh John. An Exemplar-Based Smalltalk. In *Proceedings of OOPSLA '86*, ACM Sigplan Notices, Vol. 21, No. 11, pp. 322-330.
12. Lieberherr, Karl. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, Boston, 1996.
13. Lieberherr Karl and Orleans Doug. Preventive Program Maintenance in Demeter/Java (Research Demonstration). In *Proceedings of ICSE 1997*, pp. 604-605, ACM Press, 1997.
14. Lucas, Carine, and Steyaert, Patrick. Modular Inheritance of Objects Through Mixin-Methods. In *Proceedings of the 1994 Joint Modular Languages Conference (JMLC)*. Springer-Verlag, pp. 273-282.
15. Luckham, David, Vera, James, and Meldal, Sigurd. Three Concepts of System Achitecture. Stanford University Technical Report, CSL-TR-95-674, July 1995.
16. Mezini, Mira. Variation-Oriented Programming Beyond Classes and Inheritance. Ph.D. Thesis. University of Siegen, 1997.
17. Mezini, Mira, and Lieberherr, Karl. Adaptive Plug-and-Play Components for Evolutionary Software Development. To appear in *Proceedings of ACM Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) '98*.
18. Seiter, Linda, Palsberg, Jeng, and Lieberherr, Karl. Evolution of Object Behavior Using Context Relations. *IEEE Transactions on Software Engineering*. Vol. 24, No. 1, January 1998, pp. 79-92.