

# Modeling Behavior with Personalities

Luis Blando  
Operations Systems Lab  
GTE Laboratories, Inc  
40 Sylvan Road, MS-40  
Waltham, MA 02451 USA  
lblando@gte.com

Karl Lieberherr  
College of Computer Science  
Northeastern University  
360 Huntington Avenue  
Boston, MA 02115 USA  
lieber@ccs.neu.edu

Mira Mezini  
University of Siegen  
FB-12, Hoelderlinstr. 3. D-  
57068, Siegen, Germany  
mira@informatik.uni-  
siegen.de

## 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. Dynamic personalities, a variation to the basic ideas that helps, among other things, with the object migration problem, is also discussed. We present a potential Personalities implementation by extending the Java programming language.*

## 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

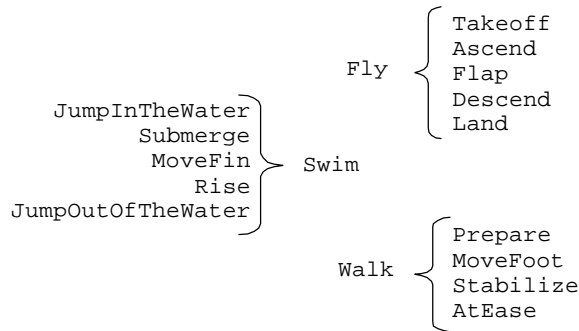
data that share a set of characteristics and group them in new entities that we term *classes*. The functions are then mapped onto these classes. Explicit guarantees are set forth with respect to the integrity of the data an *object* (that is, an instance of a class) contains.

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. Nevertheless, roles can often be found when inspecting the dynamic views of the 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 roles (instead of classes) 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. This has one of two possible consequences. Either spurious associations between classes are introduced or behavior code needs to be duplicated. The former arises 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.



**Figure 1: Zoo's functional decomposition**

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 introduces dynamic personalities while 5 discusses the relationship between personalities and frameworks. Section 6 briefly talks about the implementation. Section 7 suggests future work, 8 evaluates previous works, and 9 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 for a Zoo software system. 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.

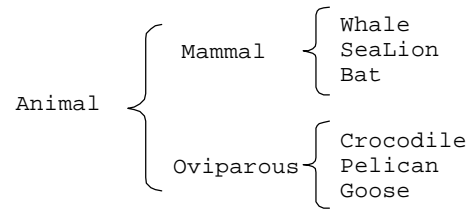
From a design perspective, one would like to implement the popular functions (i.e. fly, walk, swim) using their sub-functions (i.e. takeoff, ascend, etc.) and then require subclasses to override these sub-functions. At this time in the design process, the software engineer can (and should, in the authors' opinion) fix the semantics of, for example, the `Fly()` function. One such semantics, using Java [2], 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 partial data decomposition, shown (simplified) 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 crocodiles swim, pelicans and bats fly, while crocodiles and sea lions walk.

We can always push the placement of these popular 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: Zoo's data decomposition**

Even if you could somehow manage to share one single implementation of these popular behaviors, the problem of correctly allocating functionality to the classes still remain. At a programming level, you still need to advertise that `Whale` supports `Swim()`, `Pelican` supports `Walk()`, and `SeaLion` supports both `Walk()` and `Swim()`. This allows for the code of the clients of `Whale`, `Pelican`, and `SeaLion` 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. 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. The “complete” multiple-inheritance hierarchy for modeling our animal kingdom is presented in Figure 3.

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 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` “entity” 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. Inheritance is used for two distinct relationships; to extend data abstractions (i.e. Whale “IS-A” Mammal) and to connect behavioral abstractions (i.e. Whale “PLAYS-THE-ROLE-OF” `ASwimmingThing`). In other words, there’s no way to separate these two potentially different relationship types.

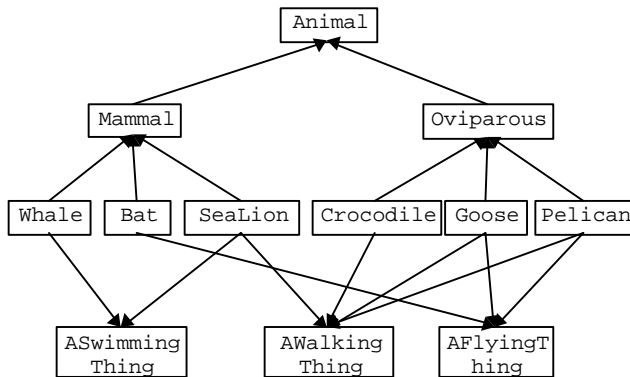


Figure 3: ZooSys using multiple inheritance

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 (using 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 system without unwanted artifacts that are not part of the problem domain.

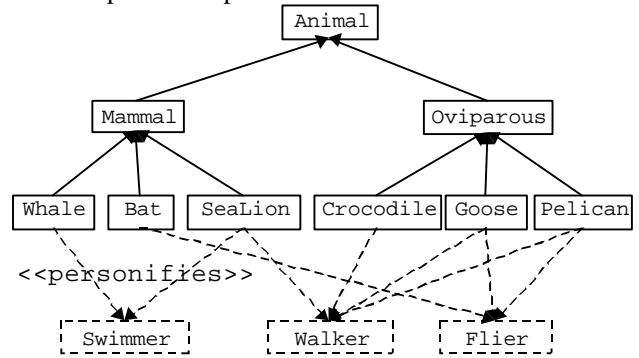


Figure 4: ZooSys using personalities

### 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 some semantic guarantees to the systems that use them.

#### 3.1 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 alt) {
    Takeoff();
    for (int a=0; a < alt; a++) Ascend();
    while(miles--) Flap();
    for(a = alt; 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.
- 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 *private functions* that the personality might need to implement its behavior. These functions are not visible either upstream or downstream.
- Any *role-specific attributes*, if needed.
- A *constructor* to initialize the state, if needed.

When a class decides to personify a given personality, it needs to declare its intent, as well as provide the methods specified in the downstream interface. For example, a definition of the Bat class follows:

```
public class Bat extends Mammal personifies Flier
{
    // intrinsic properties of the Bat class
    boolean in_Dracula_mode;
    void UpdateMode(Time time) {
        if (time>SUNLIGHTOUT)in_Dracula_mode=true;
        else in_Dracula_mode=false;}
    Bat() { in_Dracula_mode = false; }
    boolean BiteBeautifulLady(Lady lady) {
        if (in_Dracula_mode) lady.BittenBy(this);
        return in_Dracula_mode; }
    // DI implementation for Flier...
    Compass _compass = new Compass();
    void waitUntilInDracula() { // sleep until
        while(!in_Dracula_mode) { // we can go to
            UpdateMode(new Date()); // Dracula mode
            Thread.sleep(5000); } } // to Fly...
    void Takeoff() { waitUntilInDracula(); }
    void Ascend() { /* not shown */ }
    boolean ThereYet(int x, int y) {
        return _compass.where().x() == x &&
            _compass.where().y() == y; }
    void FlapTowards(int x, int y) {
        if (_compass.unset())_compass.set_tgt(x,y);
        // do whatever I need to move...
        _compass.update_position(); }
    void Descend() { /* not shown */ }
    void Land() { /* not shown */ } }
```

Figure 4 shows the different classes, personalities, and their relationships for the running example. 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. 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 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.

Even though Figures 3 and 4 look suspiciously similar, there is an essential difference in the type of links between the leaf classes (i.e. Whale, Bat, etc) and the behavior-encapsulating classes (i.e. AFlyingThing/Flier, etc). In Figure 3, inheritance is used to link both data and behavioral abstractions. While this duplicity might be convenient, it gets in the way of a proper design when you actually need to separate the concerns.

As an example, let’s take the class Pelican. Data-wise, it is a special case of the Mammal abstraction. At the same time Pelican can play the role of a Walker and a Flier. Using only MI, we are forced to use the same mechanism to express this relation. Hence Pelican would also inherit from Walker and Flier. However, I might want to express “IS-A” relationships also among behavioral abstractions. For instance, we could have a super popular behavior, called Movable, which has, for instance, an attribute called speed. This attribute is inherited from all derived behavioral abstractions. As a result, Pelican would inherit speed twice, which is incorrect! Furthermore, this attribute definition should be invisible for the Pelican, since it is part of the implementation of the popular behavior. However, it cannot be simply declared private since this would prevent behavior abstractions that are specializations of Movable to inherit the attribute.

Analogously, assume that two different developers implement Walker and Flier (without inheriting from the same behavioral abstraction). Suppose that the implementation of the respective popular functions uses operations that accidentally have the same name, say, calculateSpeed(). Again, we have a conflict in Pelican. Making the implementation private does not solve the problem, since it prevents us from specializing those methods for special behavioral abstractions, let say SpecialFlier.

In the authors’ opinion, different scoping rules for “IS-A” relationships between abstractions of the same kind and “PLAYS-THE-ROLE-OF” relationships between data and behavioral abstractions are needed. With Personalities this distinction is made very clear. A personality is the unit for behavioral abstractions. As such, it clearly separates: (a) what is expected in order to provide a popular behavior, (b) what is provided, (c) what is locally needed to implement what is provided, and (d) how what is provided is implemented. The interface to the data abstractions is different from the interface to possible special behavioral abstractions. None of the internal implementation details are visible to data abstractions. As far as special behavioral abstractions are concerned, the personality developer is free to decide which part to make private and which protected.

Thus, Personalities can be thought of as interfaces enhanced with the ability to implement behavior. The

underlying idea is to try and provide an artifact to model a relationship between data and behavioral abstractions that is missing in current OO languages.

### 3.2 Following the Law of Personalities (LoP)

Personalities must follow a certain set of constraints in order to provide semantic<sup>1</sup> 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.
2. Clients of the personality must not use the links to the personifying class (i.e. the downstream interface).
3. The implementation of the popular functions must be protected against changes by the personifying classes.
4. The implementation of popular functions is allowed to use the DI/UI/local functions and the methods of classes returned by DI/UI/local functions and nothing more. This includes methods of classes of local data and of return types. This is analogous to the Law of Demeter in [14].

Requirement #1 makes personalities uninstantiable on their own. 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.

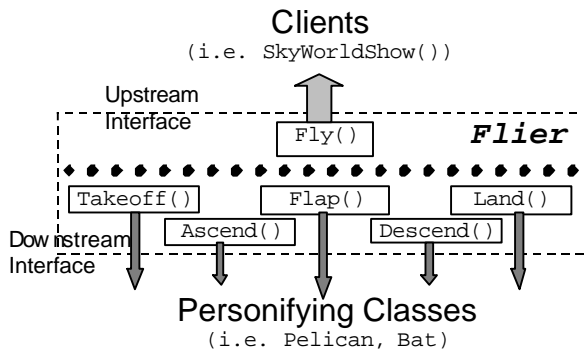


Figure 5: Anatomy of the Flier personality

Requirement #2 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

<sup>1</sup> We claim that Personalities help to provide some level of semantic guarantees to its users. Strong guarantees, on the other hand, are not possible since we know of no way of automatically validating the correctness of a piece of software.

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
}
```

Requirement #3 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 LazyPelican class to do something like:

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

the semantic integrity of the system would be compromised, since this special pelican 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 #4 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 #2 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 Pelican 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!)

**3.2.1 Coding guidelines.** We also have a couple of recommendations that help make the software cleaner but either are not compiler enforceable or would be too restrictive for large-scale, personality-based component deployment. The two recommendations follow:

1. The downstream interface functions should have clearly documented semantics.
2. Only basic object types (i.e. `string`, `int`, `Vector<string>`, etc.) plus the personality itself should be passed in parameters and returned from functions in the downstream interface (of the personality that acts as the boundary of the unit of deployment).

Guideline #1 recognizes that it is important for the semantics of the DI 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.

A compiler can easily check guideline #2, which ensures an attainable minimum set of pre-required knowledge in order for any given object to personify a given personality<sup>2</sup>. We suggest that the parameters and return values of the downstream interface methods not be 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.

```

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

```

## 4. Dynamic Personalities

Making personalities attachable and detachable at runtime allows us to solve the classic object migration problem [21]. For example, we might have an object of class `Person` that gets hired by a company, and thus needs to become an object of class `Employee`, and later gets promoted, and thus needs to be of type `Manager`. Common workarounds for this problem include reclassification (i.e. reinstantiating an object of the new class that "extends from" the old class) and delegation (i.e. creating objects contained by the original that perform the functions of the new class). Both of these approaches have drawbacks. Reclassification is not practical in distributed systems since the old object reference is lost to a new one. Therefore, all clients need to be updated (a non-trivial task). Delegation is problematic in strongly-typed languages (i.e. Java) specially since the old object reference (i.e. `Person`) needs to respond to a protocol that it does not know about (i.e. `Employee`).

We can extend the Personalities concept to make the personalities either active or inactive at any given time. A third-party takes care of activating or deactivating each of the object's personalities<sup>2</sup>. The semantics of `personifies` changes from "does" to "can". That is, the statement

```

class Person personifies Employee, Manager

```

states that the class `Person` might (in the future) personify an `Employee` or a `Manager`. A fresh instance of an object of class `Person` does not, by default, contain the behavior for `Employee` or `Manager`. A common protocol for all personifying classes is then created to maintain this set of personalities. The "dormant" personalities are activated by invoking `personify("Employee")` on the object. After that, the object will behave as an `Employee` as well. Similarly, the call `forget(«personality»)` will deactivate a role and return it to dormant state. Figure 6 shows the common protocol for all personifying classes.

<code>personify("&lt;p&gt;")</code>	Enable personification of <p>
<code>forget("&lt;p&gt;")</code>	Disable personification of <p>
<code>personifies("&lt;p&gt;")</code>	Returns true or false depending on whether <per> is enabled in the class.
<code>personalities()</code>	Returns a Vector of Strings with the names of all the personalities that are enabled in the class.

**Figure 6: Common protocol**

The type of a personifying class is the union of the Personalities it personifies plus its own type<sup>3</sup>. Thus, a

<sup>2</sup> In this paper we ignore the issue of safeguarding these state changes. However, any authentication/authorization mechanism would work.

<sup>3</sup> The problem of potential name clashes between UI or

Person object that personifies `Employee` and `Manager` can be treated as an `Employee` or a `Manager` object. Solving the object-migration problem now becomes a simple state change (i.e. `joe.personify("Employee")`). In this manner, Personalities allow us to preserve object identity while “increasing” the object’s type, even when using strongly-typed languages, such as Java.

However, the proposed solution has the drawback that the developer of the personifying class (i.e. `Person`) needs to “think ahead” about all the personalities that might be personified by the class during its lifetime. This is usually not a big constraint in controlled systems but might become a problem in evolving the system after some time. An ideal situation would be one in which it is not necessary for classes to declare their ‘expected’ personalities, but rather any personality can be attached to any class. Fully-Dynamic personalities, attempt to solve this problem by lifting all the personality-related code to generic “catch-all” functions that perform method dispatch dynamically, based on the personalities that are active/present at the time. A full discussion of this is outside of the scope of this paper. However, the reader can consult [3] for more information.

## 5. Personalities and Frameworks

Frameworks [11] encapsulate behavior at a bigger scope than Personalities. They provide one or several flows of control over a set of prototypical classes that abstract the problem the framework has been built to solve. Just as Personalities require personifying classes to implement their downstream interface, an application adapts to a framework by connecting to the framework’s “hotspots”. This connection takes the form of inheritance or delegation.

Inheritance, however, cannot be easily used when the class that is going to extend a hotspot is either already extending another class in the application domain or when the same application class should extend two (or more) of the framework hotspots. Both these situations lead to multiple-inheritance.

Delegation can be used in these cases, but it is inconvenient, to say the least. First, the application needs to instantiate both the framework objects and the delegates and then make sure that these are correctly registered to be called later. Second, delegation presents the object identity problem, as shown in Figure 7. An application-level client (i.e. `OviparousClient()`) is passed the hotspot reference. However, it cannot use the application-side object (i.e. `Oviparous/Pelican`) since the hotspot object and the application-side object are in fact two different types.

Personalities can replace the hotspots and become the interface between frameworks and applications.

---

DI methods of different personalities is acknowledged but not addressed. Renaming techniques could be used to alleviate this problem.

Application-level classes would then personify the necessary hotspot(s). The developers can still use the familiar multiple-inheritance like hierarchy without worrying about language constraints. Furthermore, the object identity problem is solved since a class that personifies a hotspot can actually be accessed as either the application-side class (i.e. `Pelican` in Figure 7) or the framework-side class (i.e. `Flier` in Figure 7).

Personalities do encapsulate behavior at the micro-level. Being an application of the template-method pattern [7], they impose a sequence of lower-granularity operations for a given high-level operation. We feel such a separation is healthy and forces the framework developer to clearly define the semantics that she will require from the application developer.

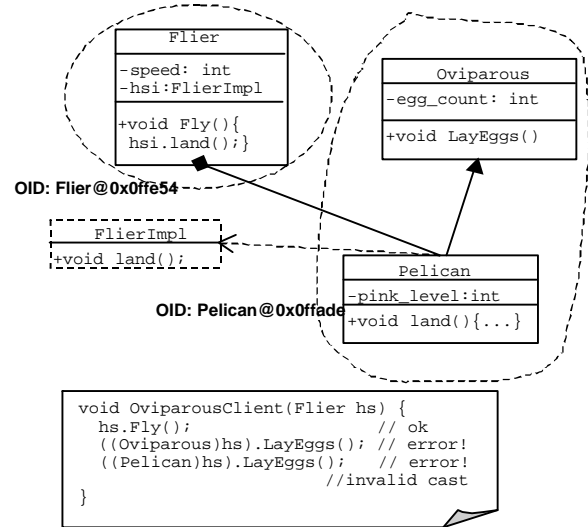


Figure 7: Object identity and delegation

As explained in [3], personalities can be used to cleanly compose or integrate frameworks as well. Personalities’ clear rules and non-shallow interfaces provide a nice demarcation point for the edges of the frameworks. Personalities can also work alongside other collaboration-based works, such as APPCs [19].

## 6. Implementation Details

The development environment assumes that the developers program solely in *Personalities/J*, for “Personalities in Java”. A compiler does the analysis in the *Personality/J* code and then translates it into standard Java 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.

The compiler generates Java files according to the diagram in Figure 8. Each personifying class gets one `$Ego` object per personified personality. They also get the

popular function proxy that immediately delegates to the `$Ego` object, passing itself as a parameter. The `$Ego` classes contain the popular function implementation and they delegate back to the personifying classes for the DI implementation. Space reasons do not permit us to include further details. However, the reader is encouraged to browse the examples in the Personalities home page [3].

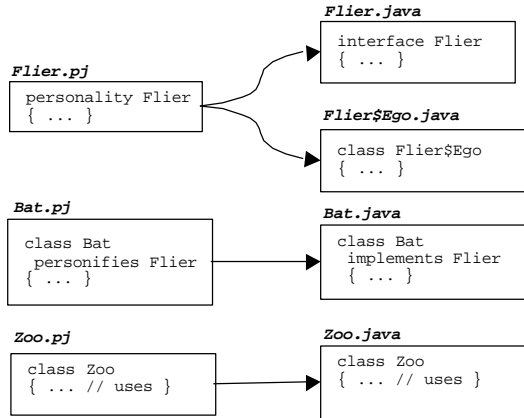


Figure 8: PJ to Java mappings

## 7. Work in Progress

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
  
```

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 personality

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 CheckProdQuant(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 kilos)
    /* implement behavior in metric system */
  public boolean CheckProdQuant(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=CheckProdQuant(<pounds>*2.2) {
  public
  boolean CheckStockHas(double kilos) {...}
  } // don't implement CheckProdQuant() at all
  
```

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

Second, we are thinking about different approaches to provide extensibility and composition of personalities. Inheritance of personalities seems to be straightforward, with the commonly-used semantics (i.e. those of Java) being already applicable with the current ideas. For example, a `Runner` extends a `Walker` and as such it can specialize `Walker`'s downstream interface and maybe require one of its own (probably at a lower granularity). Composition of personalities, on the other hand, seems to be a little bit more complicated. Composing two personalities, such as `Flier` and `Acrobat` to get a third, `AcrobaticFlier`, might require a third-party class to personify both, implement one's DI using the other's UI, and export the remaining DI for the end-user class to personify. Studying how to formalize this asymmetry remains a topic of current research.

## 8. Related Work

Our layered approach and usage of the Template Method pattern [6] makes personalities similar to *frameworks* [11]. Frameworks usually encapsulate behavior 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 the small-granularity functions in the DI.

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* [13] and the work by Gottlob et al. on extending object-oriented systems with roles [8]. Both approaches support two kinds of hierarchies: class and role hierarchies (called exemplars in [13]). 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 `Flier` Personality being applicable to both `Bats` and `Pelicans`. 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 *mix-in-based inheritance* [4,5], *contracts* [10], *mix-in-methods* [16], *MixedJava* [6], *Rondo* [18], and *context relationship* [20]. These works share the fact that variations on a base behavior are modeled in stand alone artifacts called mixins in [4,5] and [6], contracts in [10], mix-in-methods in [16], adjustments in [18], and context objects in [20]. 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.

The work presented in [12] also considers the need for synthesizing object-oriented and functional decompositions. The visitor pattern [7] 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. 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 `Pelican`, `Crocodile`, `SeaLion` and `Goose`, 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* [9] 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++* [14]) or adaptive methods (in *Demeter/Java* [15]), 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 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 [17]. 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 [17] aims at defining an architecture whereas personalities work at a much smaller (class) granularity.

## 9. Conclusions

This paper proposes a small extension to the Java programming language 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 trees, 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. As side effects, dynamic personalities provide a nice solution to the object-migration problem while the clear rules of LoP make integration and composition of frameworks simple.

## 10. 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.

## 11. References

[1] Andersen, E., Reenskaug, T. System Design by Composing Structures of Interacting Objects. In *Proceedings of ECOOP 1992*. pp. 131-152.  
[2] Arnold, K., Gosling, J. The Java Programming Language, 2<sup>nd</sup> Ed. Addison-Wesley, 1998.

[3] Blando, L. Designing and Programming with Personalities. *MS Thesis*. Northeastern University (TR# NU-CCS-98-12). December 1998. (<http://www.ccs.neu.edu/home/lblando/personalities>)  
[4] Bracha, G., Cook, W.. Mixin-based Inheritance. In *Proceedings of OOPSLA 1990*.  
[5] Bracha, G., Lindstrom, G.. Modularity meets Inheritance. In *Proceedings of IEEE Computer Society International Conference on Computer Languages* (Washington, DC, April 1992), IEEE Computer Society, pp. 282-290.  
[6] Flatt, M., Krishnamurthi, S., Felleisen, M. Classes and Mixins. To appear in *Proceedings of the 1998 POPL Conference*. January 1998 at San Diego.  
[7] Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns. Addison-Wesley, 1994.  
[8] Gottlob G., Schrefl M., Roeck Be. Extending Object-Oriented Systems with Roles. In *ACM Transactions of Information Systems*, Vol. 14, No. 3, July 1996.  
[9] Harrison W., Ossher H. Subject-Oriented Programming. In *Proceedings of OOPSLA 1993*. Sigplan Notices, Vol. 28, No. 10, pp. 411-428, 1993.  
[10] Holland, I. The Design and Representation of Object-Oriented Components. *PhD Thesis*. Northeastern University, 1993.  
[11] Johnson R. Frameworks = (Components + Patterns). In *Communications of ACM*, Vol. 40, No. 10, 1997.  
[12] Krishnamurthi S., Felleisen M., Friedman D. Synthesizing Object-Oriented and Functional Design to Promote Reuse. In *Proceedings of ECOOP '98*, Lecture Notes, 1998.  
[13] LaLonde W. R., Thomas D., Pugh J. An Exemplar-Based Smalltalk. In *Proceedings of OOPSLA '86*, ACM Sigplan Notices, Vol. 21, No. 11, pp. 322-330.  
[14] Lieberherr, K. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, Boston, 1996.  
[15] Lieberherr K., Orleans D. Preventive Program Maintenance in Demeter/Java (Research Demonstration). In *Proceedings of ICSE 1997*, pp. 604-605, ACM Press, 1997.  
[16] Lucas, C., Steyaert, P. Modular Inheritance of Objects Through Mixin-Methods. In *Proceedings of the 1994 Joint Modular Languages Conference*, pp. 273-282.  
[17] Luckham, D., Vera, J., Meldal, S.. Three Concepts of System Architecture. Stanford University Technical Report, CSL-TR-95-674, July 1995.  
[18] Mezini, M. Variation-Oriented Programming Beyond Classes and Inheritance. *Ph.D. Thesis*. University of Siegen, 1997.  
[19] Mezini, M., Lieberherr, K.. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *Proceedings of OOPSLA, 1998*.  
[20] Seiter, L., Palsberg, J., Lieberherr, K. Evolution of Object Behavior Using Context Relations. *IEEE Transactions on Software Engineering*. Vol. 24, No. 1, January 1998, pp. 79-92.  
[21] Wieringa, R., de Jonge, W., Spruit, P. Using Dynamic Classes and Role Classes to Model Object Migration. *Theory and Practice of Object Systems*, 1995