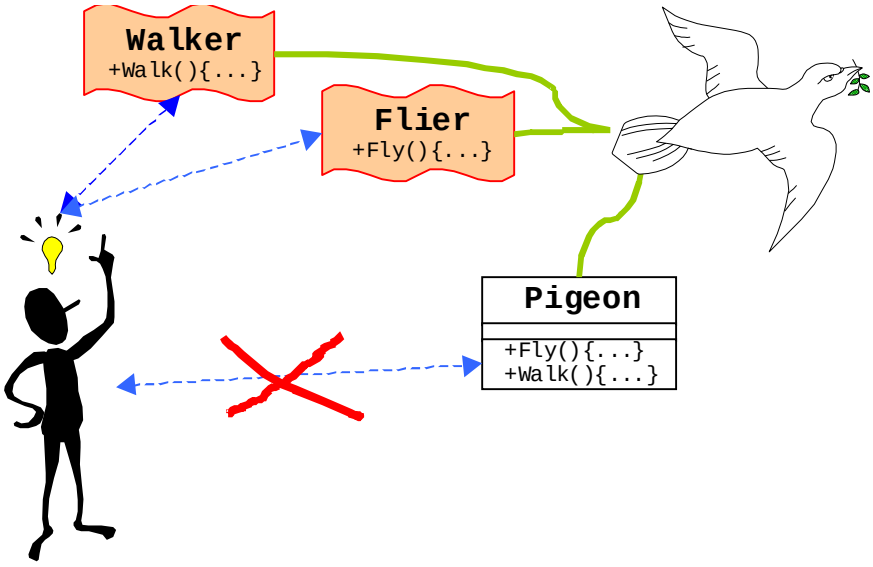
Designing and Programming with Personalities

Luis Blando



College of Computer Science

Northeastern University

(Technical report #NU-CCS-98-12)

1998

Designing and Programming with Personalities

by

Luis Blando

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science

Northeastern University

1998

Supervisory Committee: **Karl Lieberherr** **Mira Mezini**

Date: December 2nd, 1998.

Northeastern University

Abstract

DESIGNING AND PROGRAMMING WITH PERSONALITIES

by Luis Blando

Decoupling behavior modeling from a specific inheritance hierarchy is 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 helps to preserve the semantics during reuse. Furthermore, we show how Personalities can help solve the problem of *object migration* and how they can easily integrate with *frameworks*. We present two different Personalities implementations by extending the Java Programming Language.

TABLE OF CONTENTS

List of Figures 8
List of Code Samples 9
THE DIFFICULT TASK OF MODELING BEHAVIOR 11
The Functional Nature of Software Systems 11
Finding Objects and Behavior 13
When Behavior Misbehaves 15
Roles and the Application Domain Functions 15
The Problem of Mapping Application Domain Functions to the Class Hierarchy 16
Issues in Modeling Popular Functions 17
Pelicans, Whales, and The Virtual Zoo: A Running Example 17
Alternatives for Mapping Popular Functions 20
MODELING WITH PERSONALITIES 24
Analysis and Design With Roles 24
What are Personalities? 25
Syntax and Usage 26
The Law of Personalities 31
No Default Implementation Rule (the need for some class) 32
Basic Types Rule (KISS) 32
Behavioral Buffer Rule (gotta do something, after all) 33
Fixed Popular Behavior Rule (don't go second-guessing me) 34
Implementation Separation Rule (to each its own) 34
The Rule That Almost Made It 35
DYNAMIC PERSONALITIES 36
Why Do We Even Care About This? 36
What's Wrong with Personalities "As-Is" 38
Where Static Personalities Have It Right 38
Where Static Personalities Fall Short 38

Dynamic Personalities 39
What We Are Trying To Achieve 40
Indecisive Personalities. (Not Fully Dynamic, But Good Enough) 41
Fully Dynamic Personalities (The Wonders of Simplifying) 42
Method Dispatch in Dynamic Personalities 42
Class' Conformance to a Personality's DI 44
PERSONALITIES/J 46
A Few Words About the Programming Environment 46
Implementing Static Personalities 46
Java and Interfaces 46
The Mapping Process 47
Using a Class that Personifies 47
Mapping to Java 49
The Changes for Dynamic Personalities 55
Personalities' Protocol 57
Client Code Changes 57
The Generated Java Code 58
PERSONALITIES AND THEIR BIG COUSINS 62
Frameworks and Personalities 62
Adapting a Framework using Hotspots 62
Personalities as Hotspots 65
Personalities as Traffic Cops 66
Composing Frameworks using Personalities 69
Delegating Composition to the Application's Code 71
Other Collaboration-Based Work 73
FUTURE WORK 75
Performance Ranges or Guarantees 75
Mapping and Parameter Conversion 75
Inheritance of Personalities 78

Composition of Personalities 80
WHERE HAVE WE SEEN THIS BEFORE? 82
Using Delegation 82
Relaxing Inheritance 82
The Visitor Pattern 83
Subject-Oriented Programming 84
Adaptive Programming 85
The RAPIDE Connection 85
CONCLUDING REMARKS 86
What We Said We Were Going To Say 86
What We Actually Said 86
What Good It Did Us 87
Bibliography 88
APPENDIX A – A COMPLETE EXAMPLE 92
The Static Version 92
The `.pj` Files for the Animal Hierarchy 92
The Personality Files 93
The Client using Static Personalities 94
The Generated Java Code 95
Animal Hierarchy 95
Personalities 97
Client Code 98
The Dynamic Version 99
The Client using Dynamic Personalities 99
The Generated Java Code 101
Animal Hierarchy 101
Personalities 103
Client Code 105

LIST OF FIGURES

Number *Page*

Figure 1: Zoo class hierarchy 18
Figure 2: Popular behavior decomposition 19
Figure 3: Popular behavior using multiple inheritance 22
Figure 4: Relationship between client, personality, and personifying classes 26
Figure 5: A prototype and an example of a personality 28
Figure 6: The Zoo class diagram using personalities 30
Figure 7: Person, Employee, Manager hierarchy 36
Figure 8: Generated Java Files 49
Figure 9: Generated Java System's Relationships 51
Figure 10: Creation of objects and their associated `$Ego` classes 54
Figure 11: Using an upstream interface method 56
Figure 12: Dynamic Personalities' common protocol 57
Figure 13: Subclassing hotspots in framework instantiation 64
Figure 14: Delegation and object identity 65
Figure 15: ShowFramework with `Flier` as hotspot 66
Figure 16: `TakeoffAndLandFramework` with `LandGear` as hotspot 67
Figure 17: `PlaneApp`, the original application 68
Figure 18: Composing two frameworks intrusively 70
Figure 19: The plane application using the composed framework 71
Figure 20: The Plane Application using two side-by-side frameworks 72
Figure 21: Inheritance of Personalities w.r.t. abstraction and granularity 79

LIST OF CODE SAMPLES

Number *Page*

Code Sample 1: Definition of the `Flier` personality 28

Code Sample 2: A `Bat` class that personifies a `Flier` 29

Code Sample 3: A `LazyPelican` redefining what it shouldn't 34

Code Sample 4: A client using an object that personifies `Flier` 47

Code Sample 5: Knowing and Unknowing clients 48

Code Sample 6: `SeaLion.java` 52

Code Sample 7: `Swimmer$Ego.java` 52

Code Sample 8: Pseudo code for generating the `$Ego` classes 53

Code Sample 9: The Java interface `Swimmer.java` 53

Code Sample 10: Pseudo code for generating the Java interface file 54

Code Sample 11: Pseudo-code for creating the Java class files 55

Code Sample 12: A dynamic client 58

Code Sample 13: Adding `Shrink` at the correct place in the inheritance chain 59

Code Sample 14: Generating Java class files (dynamic version) 60

Code Sample 15: Generating `$Ego` files for the dynamic case 61

Code Sample 16: `SpaceShuttle` class 76

Code Sample 17: `SpaceShuttle` class with name-mapped personification 76

Code Sample 18: Two SEBSD personalities 77

Code Sample 19: Personifying two SEBSD personalities without parameter conversion 78

Code Sample 20: Personifying two SEBSD personalities with parameter conversion 78

Code Sample 21: `MechanicFlier extends Flier` 79

ACKNOWLEDGMENTS

I would like to thank Karl Lieberherr and Mira Mezini for their help and support of this work. Thanks to Tony Confrey and Johan Ovlinger for their review of early versions of this work. I am grateful to GTE Laboratories for their sponsorship of my studies. Most importantly, I would like to thank Laura for her unconditional support.

Chapter 1

THE DIFFICULT TASK OF MODELING BEHAVIOR

This thesis aims to present an alternative way of modeling behavior in software, using as much as possible of the current technology. It is important, therefore, to understand the nature of a software system, the process by which systems come into being, and the complex task of modeling application-domain behavior in software.

THE FUNCTIONAL NATURE OF SOFTWARE SYSTEMS

At a very high level, a software system is deployed to perform a *function* for its owner. When feasibility studies are conducted on whether to develop a new system or not, owners are concerned with the new functionality or capabilities a system will have. The "black-box" metaphor has its roots in the inherent functional nature of software systems: "given input **x**, it will produce output **f(x)**". In the words of Bertrand Meyer, "*a well-organized software system may be viewed as an operational model of some aspect of the world. Operational because it is used to generate practical results and sometimes to feed these results back into the world; model because any useful system must be based on a certain interpretation of some world phenomenon.*" [Meyer88, pp. 51]

Furthermore, the functional view of the system can be decomposed into a function hierarchy. Bigger-scope functions are made up of smaller-scope ones. The result is a *functional decomposition* of the application domain, which represents a kind of containment hierarchy in the sense that in order to fulfill a bigger scope function, you must "have" (i.e. execute) its smaller-scope functions. However, this is containment of relationships (i.e. links), and not of the functions themselves, as it is quite possible for several bigger-scope functions to depend on the same smaller-scope function to perform a certain task generic to the entire system (i.e. `login()`).

Following this strict functional decomposition fueled the *structured programming* approach to software development. Structured programming implies decomposing the functionality of the application into several smaller functions, each with clear interfaces. The flow of control is kept by a "privileged" function (i.e. `main()`) who yields to sub-functions as necessary. Since functionality was encapsulated, it was easy to see "what the program was attempting to do".

Unfortunately, focusing exclusively on the functional structure of the system has its disadvantages. The main problem is that the data over which all the functions operate are spread throughout the entire system. Partially because of the programming languages' requirements with respect to visibility, many times specific data items needed to be duplicated in different parts of a system, breaking *data encapsulation.* The alternative to data duplication was giving all the data elements global visibility, with the accompanying reduction of the global namespace, as well as the scalability and concurrency problems it entails. Furthermore, in both approaches it was not possible to guarantee the integrity of the data elements. Since many different functions were allowed to manipulate the data, different functions might change the data without abiding by the same intended semantics.

Object-oriented programming grew to address these problems. According to the object-oriented programming paradigm, data elements that belong together are collected in newly created abstractions, called *classes.* Access to a class' data members is closely guarded by the class itself, thus providing a mechanism for preserving data integrity through the use of *accessor methods.* Classes can themselves be ordered in a hierarchy, creating what is usually called the *class hierarchy* or *inheritance tree.* The relationship between classes in a class hierarchy is of the "IS-A" kind, and does not imply containment.

Viewed from the leaves up however, an inheritance tree does imply containment of data members. As a matter of fact, inheritance grew out of the need for providing incremental extension of data abstractions. Therefore, a class that *extends* a super-class contains all the data members of the super-class plus its own. Inheritance trees are, therefore, specially appropriate to represent relationships between real-world objects. From examining the real

world, it is apparent that the characteristics (i.e. data attributes) of many pairs of objects that would fulfill the "is-a" relationship can be strictly ordered via a superset relationship. For instance, an oversimplified version of an `Automobile` could contain the `make`, `model`, `year`, and `motor-hp` attributes, while a `Truck` that `extends Automobile` has all these attributes plus maybe `load-capacity` and `traction`, which are specific to trucks.

The same is not true, however, for the member functions in a class hierarchy. While inheriting classes must carry along all the data members of their super-class, they are in some cases allowed to override the member functions and thus might conceivably change their semantics.

FINDING OBJECTS AND BEHAVIOR

Object-oriented programming has been well received by the software engineering community at large, and is currently being used by virtually every large software development effort. The first thing that comes to mind when designing an object-oriented system is, naturally enough, finding the objects. In spite of Meyer's assertion, "*This is why object-oriented designers usually do not spend their time in academic discussions of methods to find the objects: in the physical or abstract reality being modeled, the objects are just there for the picking!*" [Meyer88, pp. 51], a number of object-oriented analysis and design methodologies have tried to help developers do just that, identify and populate the objects.

This thesis is more concerned, however, on the contents of these classes. Namely, the member functions (and the semantics) each of them will have, and how they get assigned to a particular class. As indicated in what follows, it is generally recognized that after initial analysis of the application domain, a software developer is left with two hierarchies: the data hierarchy and the function hierarchy. How these two sets get melded together to form classes is at the heart of the problem of modeling behavior with current object-oriented programming languages.

In the view of Rebecca Wirfs-Brock et. al. [Wirfs90], initial exploration yields a list of classes within the application, a description of the knowledge and operations for which each class is responsible, and a description of the collaboration between classes. Classes are loosely defined as "objects that share the same behavior". However, the steps proposed for identifying classes in the system are oriented towards following the data decomposition hierarchy. Furthermore, responsibilities are assigned to objects very early-on: "**What** *actions must get accomplished, and* **which** *object will accomplish them, are questions that we must answer right at the start*" [Wirfs90, pp. 32]

Grady Booch's method [Booch94] suggests first identifying a data dictionary and then concentrating on the "semantics" of classes and objects, their behavior. The data dictionary or class hierarchy is arrived at by a number of different paths. Booch suggests not only looking at noun-phrases in the requirements specifications, but also, as in [Wirfs90], encourages making roles, responsibilities, events, and other abstractions also part of the data dictionary, using CRC cards [Bellin97] as the "*catalyst for the brainstorming process*" [Booch94, pp.237].

In Object-Oriented Software Engineering [Jacobson92], Ivar Jacobson and his colleagues follow their own "use-case based approach". They propose roughly the same sequence of discovery as the other methods; finding the class hierarchy first and determining the operations on the objects later. In their own words, "*The object's operations come naturally when we consider and object's interface. The operations can also be identified directly from the application, when we consider what can be done with the items we model.*" [Jacobson92, pp. 79].

The work of James Rumbaugh et. al. [Rumbaugh91] states very clearly that operations and, specially, roles, should not be made into classes: "*The name of a class should reflect its intrinsic nature and not a role that it plays in an association*" [Rumbaugh91, pp. 155]. They also first concentrate on building the class hierarchy from the nouns in the requirements specification.

It is important to note that all these methods do state, very clearly, that the classes should encapsulate *normal behavior* for the object they attempt to model. Normal behavior is similar

to what [Harrison93] calls *intrinsic* behavior. We agree with their view as far as the object data members are concerned, as well as their related behaviors (`get()` and `set()` operations, for instance). However, we believe that there's no singular application domain behavior that can be considered unique and different from any other (e.g. photosynthesis in [Harrison93]). Rather, the class hierarchy designer has a particular application domain she is targeting and thus her particular *view* at the time becomes the object's intrinsic (application domain) behavior.

In practice, however, the object-oriented methods all correctly suggest a bias towards representing the real world in the class hierarchy. The intrinsic *properties* [Harrison93] are somewhat view-independent and thus they provide a good foundation on which to begin modeling. This means that software developers tend to closely follow the data decomposition of the application domain in their class hierarchy. In our experience, the rule of thumb "when in doubt, follow the data" is widely acknowledged by industry's software developers.

WHEN BEHAVIOR MISBEHAVES

Unfortunately, while decomposing an application domain into a clear hierarchy of data abstractions is relatively straightforward, giving each of those data abstractions their application-based behavior is not.

ROLES AND THE APPLICATION DOMAIN FUNCTIONS

Once the data decomposition part of the analysis is finished, we are left with a number of functions that we need to assign to the class hierarchy. In this context, these functions are not the low-level, basic functions that are easily attached to every data abstraction (i.e. accessor methods, constructors, etc.) but rather part of domain-based behaviors, such as roles and responsibilities (i.e. "owner", "employee", and "service representative"). These high-level functions are usually clustered together into what has been called a "role". For

example, an `Employee` role offers the application domain functions `JoinUnion()`, and `Quit()`.

In our experience, domain experts usually speak in terms of these high-level functions and roles when describing the system. Some object-oriented methods, such as [Andersen92] and [Wirfs90], are more conducive to the discovery of roles than others. Application domain roles can usually be found when inspecting the dynamic and operational views of your system, such as use-cases in [Jacobson92], object interaction diagrams in [Booch94], and event traces in [Rumbaugh91]. These high-level functions can also arise from the need to encapsulate commonality after a detailed analysis of the behavior of a group of data abstractions, as the example in the next section will demonstrate.

The Problem of Mapping Application Domain Functions to the Class Hierarchy

The mapping from the application functional domain onto the data decomposition domain is not one-to-one or many-to-one (which is easy), but rather one-to-many or many-to-many. In other words, there are application domain functions that might be required by more than one application domain data abstraction. In this thesis, we will call these functions *popular*.

The standard solution to this problem is to select one of the potential classes and assign the popular function to it, while using associations between the other potential classes and the one implementing the popular function. This approach works fine as long as the classes involved are naturally related in the application domain, but breaks down when the association needs to be added specifically to reuse the implementation of the popular function. Adding a spurious association, not supported by the problem domain, increases the coupling of the system and hinders its reusability by incrementing the dependencies among its components. An alternative approach to using an association is simply duplicating the function implementation. This is, in its own right, undesirable from a correctness and maintenance perspective. Duplicating the function implementation means that changes to the function need to be propagated to all the classes that host an implementation of it. It

also reduces the robustness of the system, since now there are more places where an error can wreak havoc in the semantics of the whole system.

ISSUES IN MODELING POPULAR FUNCTIONS

How then, are popular functions modeled in current programming languages? To motivate this section, we will use an example. Our application domain has to do with building a software system for an animal theme park.

PELICANS, WHALES, AND THE VIRTUAL ZOO: A RUNNING EXAMPLE

Studying the different requirements from the distinct groups of users in the system, the software developer arrives at the object model for the animals in the theme park. Since the primary goal of the system is to maintain the well being of the zoo's animals, the software architect uses the veterinary group members as the domain experts. Applying any one of the more popular OOA/D methodologies, such as [Booch94], [Jacobson92], [Meyer88], [Rumbaugh91], or [Wirfs90] will likely yield a class hierarchy that follows the *classical model* [OMG92], or the intrinsic properties of the data. The class hierarchy arrived at is shown in Figure 1.

The hierarchy in Figure 1 will support the intrinsic properties' behavior (i.e. `get()`, `set()`, etc) and the application-domain behavior required by the veterinary group. Thus, functions like `LayEggs()` and `Nurse()`, which help support a "reproductive behavior", can be correctly assigned to specific classes, as shown in the class diagram.

The veterinary group will most likely not be the only user of the system. Furthermore, each user of the system has her own view about the class hierarchy. Whereas the veterinary needs to classify the animals based on their reproductive system, the inventory representative must make sure that every animal has its own code. The public relations representative, on the other hand, is terribly concerned with the fact that each animal must have its own friendly name. The trainers would like to classify the animals according to their capabilities, whereas

the feeders only care about an animal's diet, and so on. Subject-Oriented Programming, as originally proposed by Harrison and Ossher [Harrison93] explores this problem in detail.

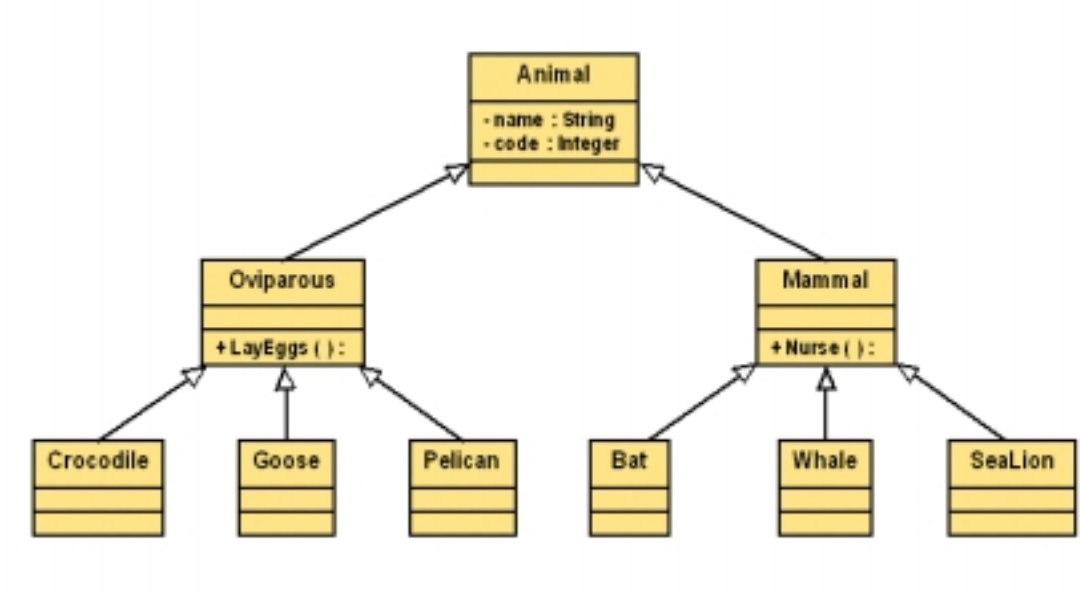


Figure 1: Zoo class hierarchy

Each of the different users of the system will require a potentially different application-domain behavior. This translates to a different set of popular functions for each. Mapping these onto the class hierarchy shown in Figure 1 will prove difficult at best. For the sake of clarity, we will consider in detail the needs of the trainers. Specifically, the task of building a virtual animal show for the theme park to be used for advertising purposes. After consulting with the domain experts, we might have a requirement statement that looks like:

> "The show shall start with the pink pelicans and the African geese flying across the stage. They are to land at one end of the arena and then walk towards a small door on the side. At the same time, a killer whale should swim in circles and jump just as the pelicans fly by. After the jump, the sea lion should swim past the whale, jump

out of the pool, and walk towards the center stage where the announcer is waiting for him."

Upon an initial analysis of the requirements, we realize that most animals perform some basic functions in the same manner. That is, there exists a set of popular functions whose semantics you can model independently based on the semantics of their sub-functions. For our running example, these popular functions could be `Fly()`, `Swim()`, `Walk()`, and `Jump()`. A simple version of the semantics of, for example, the `Fly()` function could be fixed using Java as:

```
// (x,y) is the target landing spot
void Fly(int x, int y, int altitude) {
   resetMetersFlown();
   Takeoff();
   for (int a=0; a < altitude; a++) Ascend();
   while( !ThereYet(x, y) ) FlapTowards(x, y);
   for(int a=altitude; a > 0; a--) Descend();
   Land();
}
```

Similar definitions could be made for the rest of the popular functions, making use of their respective sub-functions, as shown in Figure 2.

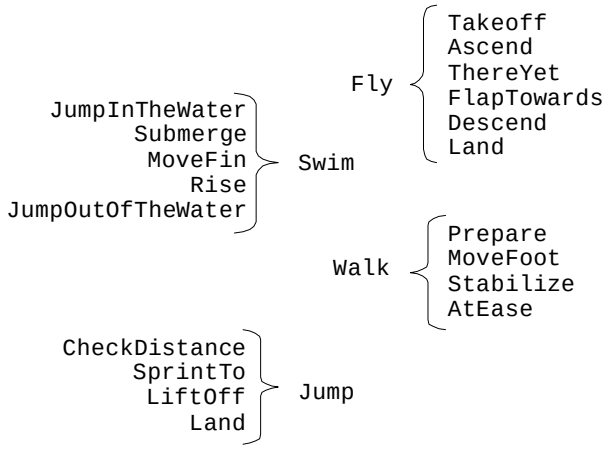


Figure 2: Popular behavior decomposition

While the software developers can, and should in the author's opinion, model these popular functions isolated from the data hierarchy, relying solely in their sub-functions, there is no easy way for mapping them onto the class hierarchy for the system, shown in Figure 1. The difficulty stems from two facts:

(a) More than one class need to provide the same function, with identical semantics

(b) The set of classes from (a) does not conform to the data decomposition, and ultimately, the class hierarchy of the system.

If we wanted to model the "reproductive behavior", on the other hand, we would be able to easily allocate its functions in the hierarchy, since it follows (somewhat) a reproductive classification. For example, the `Nurse()` function of the reproductive behavior can be correctly assigned to the `Mammal` class, since nursing is an intrinsic behavior of the "real-life" object the `Mammal` class attempts to model. Similarly, `LayEggs()` could be assigned to the `Oviparous` class without problems.

Alternatives for Mapping Popular Functions

The problem of mapping these behaviors to a class hierarchy is not new. As a matter of fact, it arises in all but the most trivial systems, as there are many stakeholders (i.e. clients) of the system, each with their own view. Unfortunately, none of the common solutions are satisfactory.

Pushing all the popular functions (i.e. `Fly()`, `Swim()`, `Walk()`, and `Jump()`) up the inheritance hierarchy is one common response to this problem. It helps in that subclasses will share a single implementation of the function, a goal we want to achieve. However, even in our simple example we find that in order to have a single implementation of the popular functions we need to map them at the root of the hierarchy (i.e. `Animal` class). For example, we can't assign `Walk()` to the `Oviparous` class, since `SeaLion` also needs that function. Besides, this approach is undesirable from a design perspective, as not all classes

perform all functions. For example, bringing `Fly()` just one step up to the `Oviparous` class immediately introduces a design mistake, since a `Crocodile` clearly does not fly.

Another alternative to moving the popular functions' implementation up the inheritance hierarchy is to simply duplicate these implementations wherever needed. This approach is clean from a design perspective, as the popular functions are allocated exactly where needed, but it is severely flawed from an engineering perspective. The duplication of code implies ever increasing maintenance costs whenever a change is required. More importantly, preserving a single semantics for the popular function becomes a daunting task, since the developer must make sure that she changes every single implementation of the function that is being worked on. It is analogous to the treatment data objects enjoyed in the early days of structured programming.

Even if you could somehow manage to have just a single implementation of each of the popular functions, your problems would be only halfway over. Each individual class still needs to advertise which ones of the popular functions they support. This is essential in strongly typed languages to enable the compiler to type check the client's source code that calls the popular function. For example, a compiler must know that `Whale` supports `Swim()` and `Jump()`, but not `Fly()` or `Walk()`, whereas `SeaLion` does support `Walk()`.

The concept of *interfaces* [Arnold97], as understood by the Java programming language, helps solve this last problem, but does not help with the others. Interfaces allow a class to advertise its compliance with arbitrary sets of method signatures. Interfaces, however, only contain method signatures, and no implementation. Thus, even if we could "advertise" the popular functions using interfaces, we would still need an implementation for each class that implements the interface, with the drawbacks that have already been mentioned. There is, however, a subtle additional problem with interfaces. To the clients of the system (i.e. the trainer or feeder objects), the fact that there is a single interface for a given popular behavior might convey the erroneous impression of fixed semantics, and thus relax explicit programmer checks or assertions.

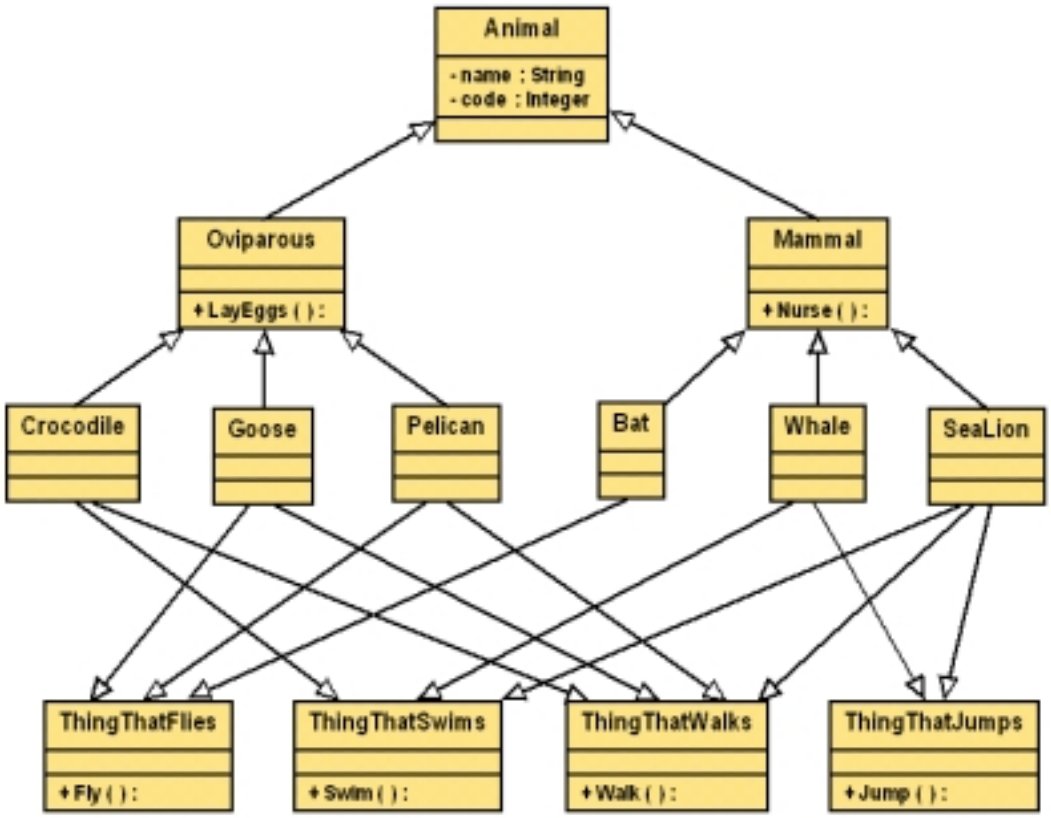


Figure 3: Popular behavior using multiple inheritance

Another approach to this modeling problem would be to turn the popular behaviors into classes and use multiple-inheritance to compose them with the data class hierarchy. The classes `ThingThatSwims`, `ThingThatWalks`, `ThingThatFlies`, and `ThingThatJumps` are created and linked to the data hierarchy as shown 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 classes that embody the behaviors (i.e. `ThingThatFlies`, `ThingThatSwims`, `ThingThatJumps`, and `ThingThatWalks`) do not have any constraints in the

implementation of their respective popular functions (i.e. `Fly()`, `Swim()`). 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 `ThingThatFlies` 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.

Chapter 2

MODELING WITH PERSONALITIES

In this chapter we present our first solution for flexibly modeling popular behavior by introducing Personalities. We first present how to do analysis and design with roles, later introduce the foundations of the Personalities idea by describing a simpler version that is static in nature but is useful to understand the concepts.

ANALYSIS AND DESIGN WITH ROLES

As has been mentioned before, there are several analysis and design methodologies that are conducive to the identification of roles in a software system. These are not, however, the only way that roles can be identified, as our running example shows.

A superficial examination of the requirements statement would not yield the different roles. A closer inspection, however, will start to show commonalties in the functions the different objects play. It is the job of the software designer, in our opinion, to pro-actively search for this commonality and extract the role descriptions in that way. The requirement statement has been purposely prepared to "hide" roles in order to demonstrate that even in this apparently adverse case, roles are easily found. Consider if not the following alternative representation of the same requirements.

> "The show shall start with every flying animal in our zoo flying across the stage. They are to land at one end of the arena and then walk towards a small door on the side. At the same time, a killer whale should swim in circles and jump just as the birds fly by. After the jump, the sea lion should swim past the whale, jump out of the pool, and walk towards the center stage where the announcer is waiting for him."

In the above representation, it is clear that the show is concerned with “flying things” as a generic behavior it expects from a subset of the Zoo’s animals. It is important to note, however, that the same cannot be said for the part of the show that involves the whale and the sea lion, since in this case the show requires those specific instances to expose a given behavior. The software designer’s job, however, is to be able to discern that both whales and sea lions (and many swimming animals) swim in a similar way, and thus be able to abstract that behavior out as a role.

The software designer can detect commonality in behavior by inspecting the different abstractions in the (evolving) system and studying how they react to similar stimuli. It is important to free yourself from the constraints of the class hierarchy at this point and look for similarities in any set of classes, even if not directly related via a common ancestor. As we have seen before, the role classification hierarchy does not necessarily conform to the class hierarchy.

WHAT ARE PERSONALITIES?

This thesis proposes a new concept, the *personality*, to encapsulate a role-specific behavior. The idea behind personalities is being able to design a set of popular functions in isolation from the class hierarchy that will eventually play the role the popular functions belong to.

In concept, a personality is a kind of *micro-framework* since its popular functions encapsulate the logic and control flow necessary for fulfilling the behavior, but depend on the implementation of certain sub-functions by the class that plays the role, or *personifies* the personality. This is analogous to the case of *frameworks* [Johnson97] where a set of cooperating classes encapsulate an entire application’s flow of control, relying on the definition of specific subclasses for customization.

From a programming language perspective, personalities are similar to abstract classes in the sense that they are also under-defined abstractions. The intent of both, however, is different. While abstract classes tend to be everything to everybody, personalities specifically target the

modeling of one (and only one) role. Furthermore, personalities impose a set of constraints both on their own definition as well as on the clients that use them to provide a greater degree of semantic guarantees than abstract classes do.

From the perspective of the clients of the personalities, they look more like Java interfaces [Arnold97], since personalities "export" the set of popular functions that correspond to a given role behavior. However, personalities are not identical to interfaces for two main reasons. First, they are conceptually more narrowly defined than interfaces. Second, they contain implementation, whereas interfaces do not.

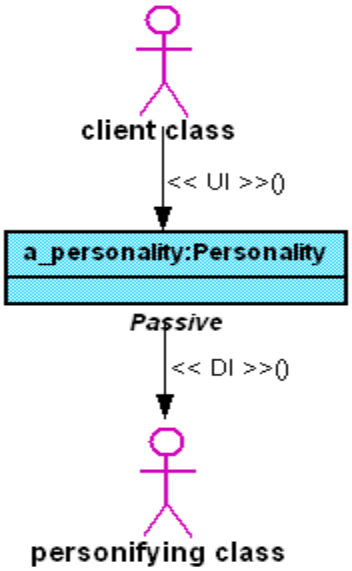


Figure 4: Relationship between client, personality, and personifying classes

SYNTAX AND USAGE

From an architectural perspective, a personality acts as a system with two interfaces to external *actors* [Jacobson92]. Being an under-defined abstraction, a personality needs to be personified by a class in order for it to be instantiable. This class is called the *personifying* class. On the other end, a personality acts as role-specific interface to the object. The systems that

make use of the personality-defined interface are called *users* or *clients* of the personality. Figure 4 depicts this relationship.

Defining a personality is very similar to defining a new class. There are a few added keywords with easy to remember semantics. A personality definition consists of five parts:

- 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" functions. 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 personality cannot access these methods.

- Any **private functions** that the personality might need to implement the upstream interface. These functions are not visible to either clients or personifying classes.

- Any **role-specific attributes** necessary for maintaining some state about the role in the personality itself.

- A **constructor** for the personality, used to initialize any personality-defined attributes.

Code Sample 1 shows the definition of the `Flier` personality, while Figure 5 shows an UML representation of a prototypical Personality as well as the `Flier` personality. New keywords are underlined in Code Sample 1.

```
// Flier.pj
personality Flier {
  // upstream interface. Must implement here.
  public
   void Fly(int x, int y, int altitude) {
     resetMetersFlown();
     Takeoff();
     for (int a=0; a < altitude; a++) Ascend();
     while( !ThereYet(x, y) ) FlapTowards(x, y);
     for(a = altitude; a > 0; a--) Descend();
     Land();
  }
  // downstream interface. Don't impl here.
  di void Takeoff();
  di void Ascend();
  di boolean ThereYet(int x, int y);
  di void FlapTowards(int x, int y);
  di void Descend();
  di void Land();
  // private functions. Must implement here.
  private void resetMetersFlown() { meters_flown = 0; }
  // attributes (specific to the role)
  private float meters_flown;
  // constructor (optional)
  Flier() { resetMetersFlown(); }
}
```

Code Sample 1: Definition of the `Flier` personality

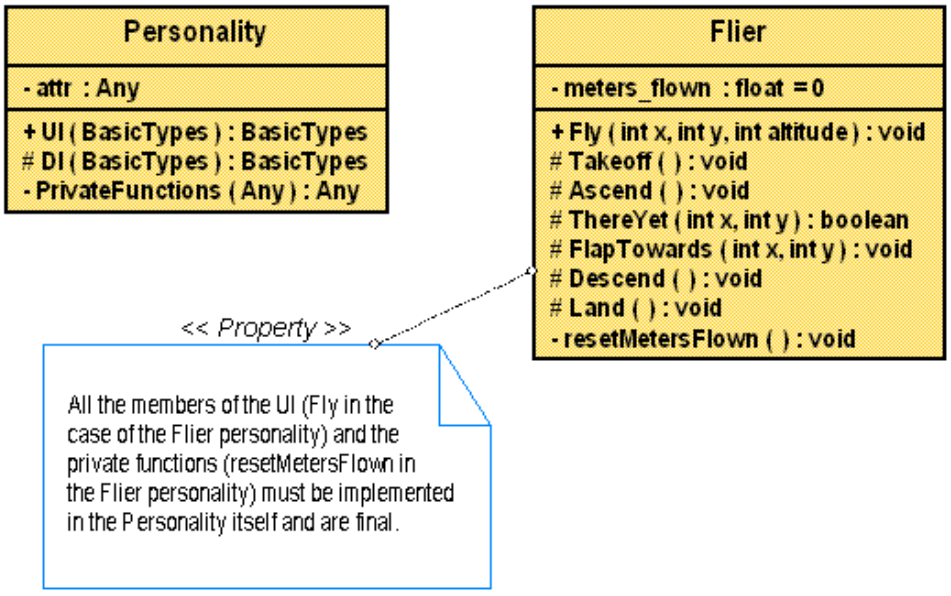


Figure 5: A prototype and an example of a personality

A class that wants to personify a given personality needs do the following:

1. Declare its intent via the `personifies` clause
2. Implement all the functions in the downstream interface.

For illustration purposes, Code Sample 2 shows a trivial definition of a "special" `Bat` class that personifies a `Flier` personality.

```
// Bat.pj
public class Bat extends Mammal personifies Flier
{
   // intrinsic properties and methods 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;
   }
   // since a Bat flies, use the Flier personality with
   // the following implementations of the DI
   Compass _compass = new Compass();
   void waitUntilInDracula() {       // sleep until
      while( !in_Dracula_mode ) {    // we can go to
         UpdateMode( new Date() );   // Dracula mode
         Thread.sleep( 5000 );       // since that's
      }                              // when we 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.unitialized() )
         _compass.set_target(x, y);
      // do whatever I need to move...
      _compass.update_position();
   }
   void Descend() { /* not shown */ }
   void Land() { /* not shown */ }
}
```

Code Sample 2: A `Bat` class that personifies a `Flier`

Figure 6 shows how the Zoo class diagram looks like with the insertion of the personalities. Notice the similarity of this diagram with Figure 3, which depicts the multiple-inheritance

solution. Personalities allow the power of expression of multiple-inheritance without many of its complications.

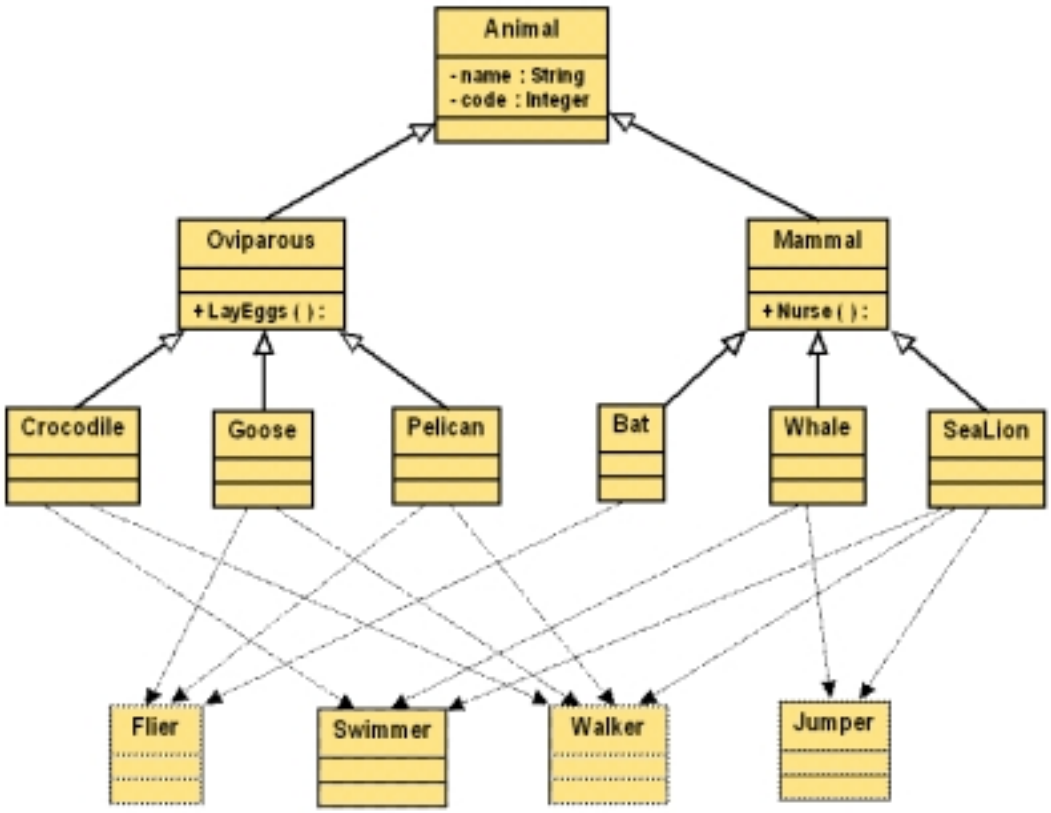


Figure 6: The Zoo class diagram using personalities

Personalities are rooted in the idea that popular functions can often be expressed in terms of smaller-granularity, class-dependant functions. The examples in this thesis, albeit somewhat simplistic, are geared towards illustrating this idea. The downstream interface is made up of these *sub-functions,* which become the link to the personifying class. From our experience in industry, large-scale framework customization is rather complex to implement, but lower-level (i.e. class-level) tailoring is many times useful and much more tractable. Personalities attempt to make this micro-framework implementation easy to model and apply.

THE LAW OF PERSONALITIES

As has been explained before, personalities are more than just a convenient way of expressing multiple-inheritance. They attempt to strengthen the semantic guarantees they provide to the client classes by conforming to a set of requirements, or rules. 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. Wherever possible, the personalities compiler needs to make sure that these are met.

1. **"No Default Implementation" Rule**: The downstream interface must be a set of pure abstract functions, with clearly identified semantics.

2. **"Basic Types" Rule**[1]: Only basic object types (i.e. `string`, `int`, `Vector<string>`, etc.) should be passed in parameters and returned from functions in the downstream interface. A reference to the Personality itself is permitted to allow for self-referential implementations.

3. **"Behavioral Buffer" Rule**: 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. **"Fixed Popular Behavior" Rule**: The implementation of the popular functions must be protected against changes by personifying classes.

5. **"Implementation Separation" Rule**: The implementation of the popular functions is allowed to use the smaller-granularity functions to communicate with the personifying class and nothing more.

---

[1] This rule has been demoted to "recommendation" to allow for the componentization of a set of personalities in an integrated fashion. We still recommend its use at the boundary of the deployable component.

No Default Implementation Rule (the need for some class)

The rationale for this rule 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. Pragmatically, this rule makes a personality uninstantiatable on its own. This much is easily enforceable by the compiler.

This rule also calls for the semantics of the downstream 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 classes. 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.

Basic Types Rule (KISS)

A compiler can easily check this requirement, which ensures an attainable minimum set of pre-required knowledge in order for any class 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, since the given personality will forever need to be deployed alongside the package that defines `MyDateClass`. Using Java's "standard" `Date` class, the following would be preferable.

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

After some debate, we decided to demote this rule to recommendation-level status. Strictly enforcing this rule makes the job of componentizing personalities too difficult, as translation to/from basic types is required at every interface. We strongly recommend the use of this rule at the "boundary" of the unit of deployment (whatever this may be) since that would ease the way for the user of the personality "component".

Behavioral Buffer Rule (gotta do something, after all)

This rule 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 fixing the client's entry point to 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 meters, 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
}
```

Fixed Popular Behavior Rule (don't go second-guessing me)

This aims at making sure that the personifying classes do not change the originally intended semantics for the personality[2]. For example, if we would allow a `LazyPelican` class to do something like what is shown in Code Sample 3 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.

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

Code Sample 3: A `LazyPelican` redefining what it shouldn't

Implementation Separation Rule (to each its own)

Finally, this rule makes personalities follow their own advice by requiring that all communication with the personifying class be restricted to the functions defined in the

[2] Personalities cannot have complete certainty that the intended semantics will be realized in the DI. We know of no way of automatically specifying and validating source-code semantics, and thus we claim that Personalities provide *some* semantic guarantees, but not *strong* guarantees.

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 the Behavioral Buffer rule 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!).

The Rule That Almost Made It

We originally [Blando98] thought about having a rule for making sure that every personality-implemented popular function actually added some behavior on top of the downstream interface semantics. The rationale was that, in an ideal world, each layer of functionality would talk to the layer right below, and no need for pass-through functions would be required. In other words, we were naïve.

We do not live in an ideal world, and even if we did, sometimes clients still need access to primitive functionality (such as that embodied by the downstream interface). After this realization, we decided to allow the clients access to the downstream interface functionality as long as there is an upstream interface method that serves it. In order to maintain a behavioral buffer we need to make certain that the clients only talk to the upstream interface. Therefore, if the personality expects its clients to need any of its low-level functionality, it needs to publicize it through an upstream interface that simply delegates to the downstream counterpart.

Chapter 3

DYNAMIC PERSONALITIES

This chapter explains the intrinsic dynamic nature of roles, critiques Personalities as presented in Chapter 2, explores the reasons and usability of the concept of Dynamic Personalities, and presents two different approaches for achieving dynamic behavior within the Personalities context.

WHY DO WE EVEN CARE ABOUT THIS?

Roles are dynamic by nature. Figure 7 shows a very simple class hierarchy. A `Person` is not born an `Employee`, much less a `Manager`. As time goes by, the `Person` "becomes" these roles. The dynamic nature of roles is perfectly consistent with the real world. If we attempt to create a software system that models the real world as closely as possible, it is only appropriate that we allow for the possibility of roles to be dynamically "attached" and "detached" from objects. This is related to a well-known problem, the *object migration* [Wieringa95] problem.

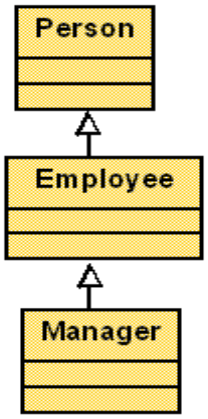


Figure 7: Person, Employee, Manager hierarchy

Traditional approaches to this problem have resorted to creating new instantiations (that is, new objects) as time goes by and the state of the original (i.e. `Person`) object changes. From a modeling perspective, creating an instance of an `Employee` object, or a `Manager` object, while keeping the original `Person` object around is counterintuitive, since all three objects are really the same real-world entity (i.e. John Smith). Also, the behavior of these three instantiations will either be very tightly coupled (i.e. using delegation) or some sort of duplication of object state will take place. In the first case, the code for the "personalized" object (i.e. `Employee`) needs to account for the fact that it needs to retrieve all `Person`-based state from the `Person` instance it is "bound" to at creation time. This, besides being error prone in the face of complex classes, is also problematic in terms of maintenance, since the `Person` class might be modified in the future and overlap with some of the state and semantics that the current version of the `Employee` class considers its own. At that time, the `Employee` class needs to be modified or state duplication will take place. Duplication of state is very error prone since keeping the states synchronized is not trivial. In the more simpler approaches, synchronization is ignored; in the most complex ones, database procedures such as two-phase commit might be necessary to guarantee the accuracy of a "split" object's state on the face of state-altering method calls.

Alternative approaches to the problem of changing roles also include re-classification schemes. In these, the `Person` object would be destroyed and a new `Employee` object will be created from the old object. This solution is costly computationally and presents the problem of not preserving object identity. Each new instantiation (i.e. `Employee`) is a brand new object, even if it is tightly bound to the original object (i.e. inherits from the `Person` class). In the face of a distributed environment, having new instantiations with different object identifiers mean that all clients of the old object need to be "refreshed" with the new reference. This is necessary if we want services which are based on the concept of immutable object identifiers (i.e. CORBA, COS Persistence State Services, etc.) to continue working as expected.

WHAT'S WRONG WITH PERSONALITIES "AS-IS"

Personalities help in modeling software with roles. There is nothing wrong with the concept of Personalities per-se. When we turn that concept into a software implementation such as the one presented in Chapter 2, however, we run into some difficulties. But before we condemn static personalities, let's take a look at how they help with the problems of the previous section.

WHERE STATIC PERSONALITIES HAVE IT RIGHT

In this section, we consider only the benefits of Static Personalities in relation to the dynamic role problem. We have explored the benefits of personalities in other contexts in previous chapters.

Static personalities lay the foundation for dynamic roles to be implemented. However, they also help in other respects. For example, static personalities help preserve object identity, since the potentially many different roles are "hidden" within the object itself. They still present very different interfaces to different systems, although they do not control which interface is active at any given time. For example, the following definition does allow different systems to interface with this single `Person` object as an `Employee` or as a `Manager` object

```
// Person will (someday) be Employee and, with
// any luck, also a Manager
class Person personifies Employee, Manager
```

Thus, static personalities can completely avoid having to re-classify an object every time its state changes.

WHERE STATIC PERSONALITIES FALL SHORT

Static Personalities impose no constraints on which Personality the class must be personifying at any given time. For instance, the moment the `Person` object is created, it can be passed to the `MIP` (Manager-Incentive-Program) system even though that `Person`

might not even be an employee yet! To some degree, static personalities resemble inheritance in this particular case. Instantiation of the above `Person` object would be similar to always instantiating a `Manager` object, which in turn inherits from an `Employee` and ultimately from a `Person` object. You can always call the object using its `Person`-defined methods, but can also use its `Manager`-defined methods at any time if you so please.

As presented in Chapter 2, the concept of Personalities is completely static. This means that Personalities are assigned to classes at compile time, and are carried with the object throughout its lifetime. This approach works fine for certain types of roles that are inherent to the object itself, although these are hard to find and usually belong to the inheritance hierarchy anyway. In general, lifelong attachment of Personalities to classes is not flexible enough to model roles accurately. As an example, a `Pelican` might be a `Flier` for only a period of its life, but it will always be an `Oviparous` animal.

Furthermore, static personalities lack commonality between themselves. That is, each Personality is its own unique entity and shares no protocol or interface with other personalities. This is disadvantageous because Personalities are supposed to be freely applicable to any class, and thus one would expect a personifying class to conform to some kind of base interface, but they don't. In short, Personalities embody micro-frameworks well, but don't go a long way in making macro-frameworks' life any easier.

DYNAMIC PERSONALITIES

To solve the problems mentioned above, we can extend the concept of Personalities to accommodate dynamic attachment and detachment at runtime. Furthermore, we extend the Personality concept with a simple protocol so that it can be more easily handled by the client code.

WHAT WE ARE TRYING TO ACHIEVE

To provide some context for what follows, we will first specify what we are hoping to achieve with Dynamic Personalities. It should be noted that this is a partial list of goals, used mostly to compare the pros and cons of the different approaches we will present later on.

1. **Runtime attachment and detachment**. That is, we should be able to have an object "act as" different Personalities at different times. In the same spirit, we need to be able to take some behavior away from an object. (i.e. a `Person` is laid off and thus he can no longer personify `Employee`)

2. **Preservation of object identity**. We would like to create an object and preserve its reference or identity throughout its lifecycle.

3. **Preservation of typing**. While preserving identity is a more important (i.e. it has operational implications), preserving type information to allow for compile-time checking is useful in the software building process.

4. **Common interface**. We would like all personalities to share some common base interface, so that systems that use classes that include personalities are able to resort to a baseline protocol with which to communicate and "discover" the characteristics of the class.

5. **Ubiquitous personification**. Ideally, we would like to be able to attach any personality to any class, provided that the class fulfills some pre-defined set of requirements. This would allow us to develop the classes in complete isolation from the personalities (possibly even at different times) and still be able to combine them together.

6. **Reasonable performance**. Ignoring this issue would be foolish, as any solution that does not take into account such a pragmatic concern is headed for failure. We expect a Personality-based implementation to be within the same order of magnitude from a multiple-inheritance based implementation with respect to execution speed. We place no

requirements or make no claims about compilation time, since that is usually not a critical factor[3].

INDECISIVE PERSONALITIES. (NOT FULLY DYNAMIC, BUT GOOD ENOUGH)

A minimalist approach to Dynamic Personalities yields *Indecisive Personalities.* Concentrating in the most important benefit, namely runtime attachment and detachment, we could imagine a scenario where a class declares its intent (and conformance) with a set of personalities at compilation time. The class, however, does not automatically get any of these personalities until they are "attached" to it. For instance, the Personalities/J[4] statement:

```
class Person personifies Employee, Manager
```

specifies that `Person` will "in the future" personify `Employee` and/or `Manager`. A third-party needs to instruct the class, at a later time, to enable or disable a given personality.

This approach has the obvious drawback that the set of all needed personalities needs to be specified at compile time, which in fact limits the freedom of the designer since she needs to "plan ahead". From a more practical perspective, however, we have found that roles themselves can be modeled into hierarchies and that usually classes personify the "abstract" versions of these roles. For instance, personifying a generic `Employee` role is broad enough to encapsulate many potential different cases.

On the positive side, though, we do expect to preserve the types of all these personalities and thus we will be able to do type checking in both the personality itself and also on the code of the different client systems. Performance should be degraded only minimally, since we expect method dispatch to be slowed only by a single check to verify the "on/off state" of the personality on the current class.

---

[3] Incidentally, with Moore's law at full-speed, nor is memory footprint that big an issue any longer. However, the "need for speed" seems to live on.

FULLY DYNAMIC PERSONALITIES (THE WONDERS OF SIMPLIFYING)

A more ambitious approach to solving the problem of the dynamic nature of roles would be to completely redesign the programming language's type system and method dispatch mechanism to allow for runtime mutation of an object's methods table. With such an approach, a class need not declare its intent of personifying anything, since all `personifies` statements will be handled at runtime.

At the conceptual level, this approach faces two immediate problems: how to manage method dispatch while preserving some degree of type checking, and how to determine and validate a given class' conformance to a personality's requirements in terms of its downstream interface. Let's investigate these in detail.

Method Dispatch in Dynamic Personalities

The purpose of specifying Personalities for objects is so that these can be used by the client systems' code to make requests that contain some specific syntax. Clients, therefore, program solely against a Personality's upstream interface. With fully dynamic personalities, however, this client's request code needs to be accepted by an object that, at class-creation time might not have known it would have had to support such a function.

For instance, in our `Flier` personality, the `Fly()` function is expected from every object that `personifies Flier`. Therefore, with static personalities class `Pelican` knows that sooner or later a `Fly()` request will come its way, and it can thus prepare for that. In the dynamic personalities case, however, `Pelican` does not know what personalities will eventually be personified by itself, and thus it cannot prepare (i.e. have the Personalities/J compiler add the `Fly()` method to the class implementation).

Since we would like to preserve the typing information at the client code, we still need to come up with a solution to the problem of method dispatch. One potential mechanism in

---

[4] Defined in the next Chapter, Personalities/J is the name we have given to the Personalities-based, Java-like programming language.

the absence of mutable dispatch tables is to mimic one by lifting all the functions in a personality's upstream interface into a generic "catch-all" function in charge of dispatching. In the example above, for instance, the Personalities/J compiler would add the `Fly()` method and its parameter into the personalities list of available functions, while at the same time modifying the client's code to call the generic catch-all function with `Fly()` as one of its parameters. Each class must then implement this catch-all function which will unmarshall the parameters and yield to the appropriate personality implementation. Notice that with this approach, we can still preserve type-checking of the client's code since that can be done before the lifting phase. Code Sample 4 shows the original code, on the left, and pseudo-code for the output of the compiler on the right, showing the lifting of the upstream interface methods.

```
// Flier.pj
personality Flier {
  void Fly(...) { ... }
  ...
}

// Pelican.pj
class Pelican {
  ...
  // no Fly() in here!
}

// SkyWorldShow.pj
class SkyWorldShow {
  void perform(Flier aFlier) {
    aFlier.Fly(10,10,10);
    ...
  }
}
```

```
// after processing by Personalities/J
class Flier {
   void Fly(...) { ... }
   ...
}

// all classes now are Personable
class Pelican implements Personable {
   ...
   void CATCHALL(String name, ...) {...}
}

// The client's code gets mutated into
class SkyWorldShow {
  void perform(Personable aFlier) {
     aFlier.CATCHALL( "Fly", 10, 10, 10 );
     ...
  }
}
```

Code Sample 4: Lifting `Flier`'s upstream interface and changing client code

The `CATCHALL` method, which is standard for all the classes in the system, plays the part of a dynamic dispatch mechanism. There could be a standard framework in which the parameters of this `CATCHALL` function are fixed and all calls are routed through it. Furthermore, each personality could define the entry points and signatures for each of the functions in its upstream interface, and these would both get added to a class' dynamic personality table at "personification" time.

Class' Conformance to a Personality's DI

Determining whether a given class can indeed personify a personality is a more complex problem. Since we are now dealing with the dynamic attachment of personalities at runtime, we need to make sure that the functions that are exported by the class do indeed fulfill the personality's downstream interface specifications.

In the absence of a computationally tractable way of specifying and validating the semantics of a piece of software, we must place our trust in the programmer. In other words, when a personality gets "attached" to an object, the programmer needs to implicitly (via identical method signatures) or explicitly (via mapping from the personality's DI signatures to the class' methods) specify how the class will fulfill the downstream interface. Thus, when the programmer attachs the `Flier` personality to the `Pelican` object we expect `Pelican` to either have methods named `Ascend()`, `Takeoff()`, etc. or the programmer to specify which of the methods that `Pelican` does have will perform each of these functions. These maps can be generated on the fly at personification time. Also, the selective dispatching of downstream interface functions from upstream method's implementation using these maps could be made part of a standard ancestor to all personalities pretty much in the similar way that the `CATCHALL` method is part of every class. In that way, every personality will know how to deal with maps at runtime.

Once we have determined that a given class does indeed have what it takes to personify some personality, we still need to make it work. From Chapter 2 we know that the personality's upstream interface implementations need to talk back to the personifying class through the downstream interface methods. Since we are working in a typeless, dynamic environment, we cannot simply expect to call these downstream interface methods on the personifying object reference since we don't have access to the object's type. This problem is solved by making sure that *all* the methods in a class are part of the `CATCHALL` scheme. In this way, the upstream methods can use the same trick to call the downstream functions.

It is important to note that the approach mentioned above would work with any standard programming language. Our purpose in this chapter is not to rely too heavily on a particular

language implementation but rather to understand the motivation behind these ideas. With Java's introspection facilities, however, these tasks become simpler and more streamlined, since we can actively inquiry any class for the methods it supports and thus the no-map personification process becomes very simple. Also, the upstream implementation becomes simpler since we can now discover the type and functions of the object and call these directly by creating invocations.

Chapter 4

PERSONALITIES/J

This chapter delves into the implementation details of the Personalities concept. Static personalities are presented, followed by dynamic personalities. Appendix A contains the complete source code for the code samples shown in this chapter.

A FEW WORDS ABOUT THE PROGRAMMING ENVIRONMENT

Our approach for implementing the Personalities/J language and compiler has been to maintain, as much as possible, the familiar Java programming language environment. However, we have made the underlying assumption that the programmer deals only with Personalities/J modules and that the Java language is merely an intermediary step between the `.pj` code and machine code.

We use the Java programming language as our target language, and rely on commercial Java compilers to create the bytecode or the native machine language code. We decided in favor of using delegation to implement Personalities/J. In retrospect, this was a good decision since it helped us in the transition from static to dynamic personalities.

IMPLEMENTING STATIC PERSONALITIES

Static personalities, although not as powerful as their dynamic counterpart, help us lay the foundations of the implementation of the Personalities concept.

JAVA AND INTERFACES

If we were relying on a language that supported multiple inheritance, such as C++, then the implementation might have been somewhat eased (with the Personalities/J compiler

concentrating mostly on enforcing the additional semantic constraints that Personalities impose over multiple inheritance semantics). Our target language, Java, on the other hand does not support multiple inheritance. It does, however, support *interfaces.*

Interfaces, as understood in the Java programming language, allow a class to advertise the implementation of selected methods by declaring its compliance with arbitrary sets of method signatures called interfaces. An interface contains only method signatures but no implementation. We use Java interfaces as a tool to implement Personalities.

THE MAPPING PROCESS

The steps to define a personality have already been presented in Chapter 2 (See Code Sample 1, for instance). Similarly, how to make a class personify a personality has also been explained (See Code Sample 2). We have not, however, described how a client system can make use of a class that personifies a given personality, or the details of the Java code that the Personalities/J compiler generates.

Using a Class that Personifies

In a nutshell, using a class that has a given personality is as simple as casting the class to the appropriate personality and simply calling its upstream interface methods. Code Sample 4 shows the prototype of a simple client that gets an object, casts it to a `Flier` "object", and calls its personality-defined `Fly()` method.

```
// ... client that uses a class that personifies the
// ... Flier personality, reference is passed as Object
// ... but the client knows that the "object" actually
// ... personifies a Flier
void DoSomethingWithAFlier(Object anObject) {
   Flier aFlier = (Flier)anObject;
   aFlier.Fly(10,10,10);
}
```

Code Sample 4: A client using an object that personifies `Flier`

It is important to notice that there is an underlying assumption about the amount of knowledge the client has about the class. In other words, even though the client is passed an `Object`, it does know that this object actually `personifies Flier`. This is not an

insurmountable constraint since clients are hopefully designed to know their objects, and delegate to other clients that know only (and are built for) a given specific Personality. We call these the “knowing” and “unknowing” clients, respectively.

```
void main() {
 // ... taken from Zoo.pj [static] in Appendix-A
 // ... knowing client sets up its data structs.
 Vector all_swimmers = new Vector();

 ...

 // ... classes are created. This “knowing” client
 // ... knows how to classify them. Notice how it
 // ... puts the object into the appropriate vector
 // ... depending on whether it personifies Swimmer,
 // ... Flier, Jumper, or Walker
 SeaLion toto   = new SeaLion();
 toto.setName("Toto");
 all_swimmers.addElement( toto ); // personifies Swimmer
 all_walkers.addElement( toto );  // personifies Walker
 all_jumpers.addElement( toto );  // personifies Jumper

  ...

 // ... when this clients needs to do work, however,
 // ... it delegates to other functions (potentially
 // ... entire systems that are built only to the
 // ... personality interface. For instance, it delegates
 // ... to PoolShow(), but notice how it casts the object
 // ... first
 for(int i=0; i < all_swimmers.size(); i++)
   PoolShow ( (Swimmer)all_swimmers.elementAt(i) );

 ...
}
// ... this “unknowing” client knows only about
// ... the personality it cares about.
static void PoolShow(Swimmer swimmer) {
    System.out.println("  PoolShow with " + swimmer);
    swimmer.Swim( 1, 1 ); // and uses its UI
}
```

Code Sample 5: Knowing and Unknowing clients

Code Sample 5, for instance, has been taken from `Zoo.pj` [static version] in Appendix A. We can see both types of uses of the object. First, the knowing client (`main()` in the sample) has full knowledge of the class it creates and its associated personalities. It can therefore cast it safely to any of the given personalities. Second, the unknowing client (`PoolShow()` in the sample) does not know about sea lions, pelicans, or whales. It has been designed and written to work only with `Swimmer` objects. While in the examples in this thesis there’s not too much of an explicit separation between these two types of clients, they are the cornerstone of the usability of the Personalities concept. We expect, therefore, for

these two “clients” to actually be entire systems, the unknowing ones built on top of the personalities’ upstream interfaces, whereas the knowing ones (i.e. the “factories”) are enterprise/domain dependent.

Mapping to Java

Each `.pj` generates at least one `.java` file. Figure 8 shows a representative set of Personalities/J files and the Java files they generate. Consult Appendix-A for the source code of a complete example.

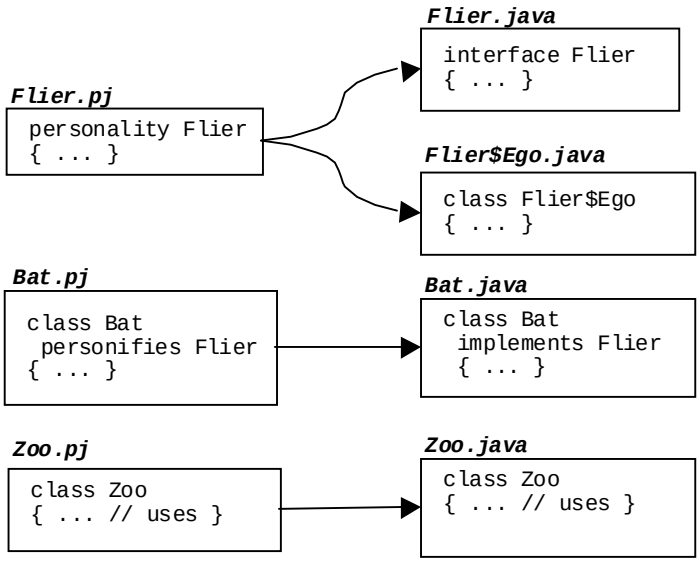


Figure 8: Generated Java files

A personality definition (i.e. `Flier.java`) generates two files. The first is simply the definition of a Java interface that encapsulates the upstream interface methods. In order to simplify the Personality infrastructure, this interface also includes the downstream interface methods. On the surface, this might appear to go against the goals of this work about separating the two interfaces. However, the Personalities/J compiler performs syntax and semantic analysis on the set of `.pj` files (which do separate between downstream and upstream interfaces). Furthermore, all of the Java files are deterministically and automatically generated by the compiler from the set of `.pj` files. We are thus guaranteed (assuming the

compiler works as advertised, of course) that clients are not accessing the downstream interface methods, even though they might look accessible via Java[5].

Once the Personalities/J compiler has finished its work, the set of generated Java files are arranged according to the diagram of Figure 9[6]. The relationship between the client system (class `Zoo`) and the personalities' interfaces are not shown to keep the diagram readable. However, bear in mind that class `Zoo` actually contains two types of client methods, one that knows about its objects (shown) and several that only know about the personalities (not shown).

The personifying classes make use of the `$Ego` classes to implement the Personality defined behavior. The compiler inserts the code for the aggregation when it generates the Java files. It also generates proxies for the personalities' upstream interface methods. This is all done automatically from the `.pj` files. Code Sample 6, extracted from Appendix A, shows the `SeaLion.java` file. Notice how the compiler has inserted the code for creating `Swimmer$Ego`, `Walker$Ego`, and `Jumper$Ego`. In addition, it has inserted the proxies for `Swim()`, `Walk()`, and `Jump()` that correspond to each of the personalities the sea lion personifies. Code Sample 11 shows the algorithm for generating these files.

The `$Ego` classes implement the personality defined behavior. Since that behavior depends on the class-defined downstream interface implementation, the `$Ego` classes need an instance of the class to be able to call its downstream interface methods. This is where defining the Java interface to contain both upstream and downstream interface methods

---

[5] In this thesis we ignore the problem presented by potential name clashes in the face of multiple `personifies` statements. While important, the solution to that problem is somewhat orthogonal to the issues being addressed in this thesis. Furthermore, a solution to such a problem would benefit not only Personalities but also other programming languages. We thus ignore the problem altogether and assume the more common model of aborting compilation when clashes exist.

[6] The `$` sign in the identifiers has been replaced by an underscore (_) because of diagramming tool inadequacies

becomes useful. We can thus use the Personality "type" in the `$Ego` classes as the `host` parameter and use it to delegate back. Code Sample 7 shows the `Swimmer$Ego` class.

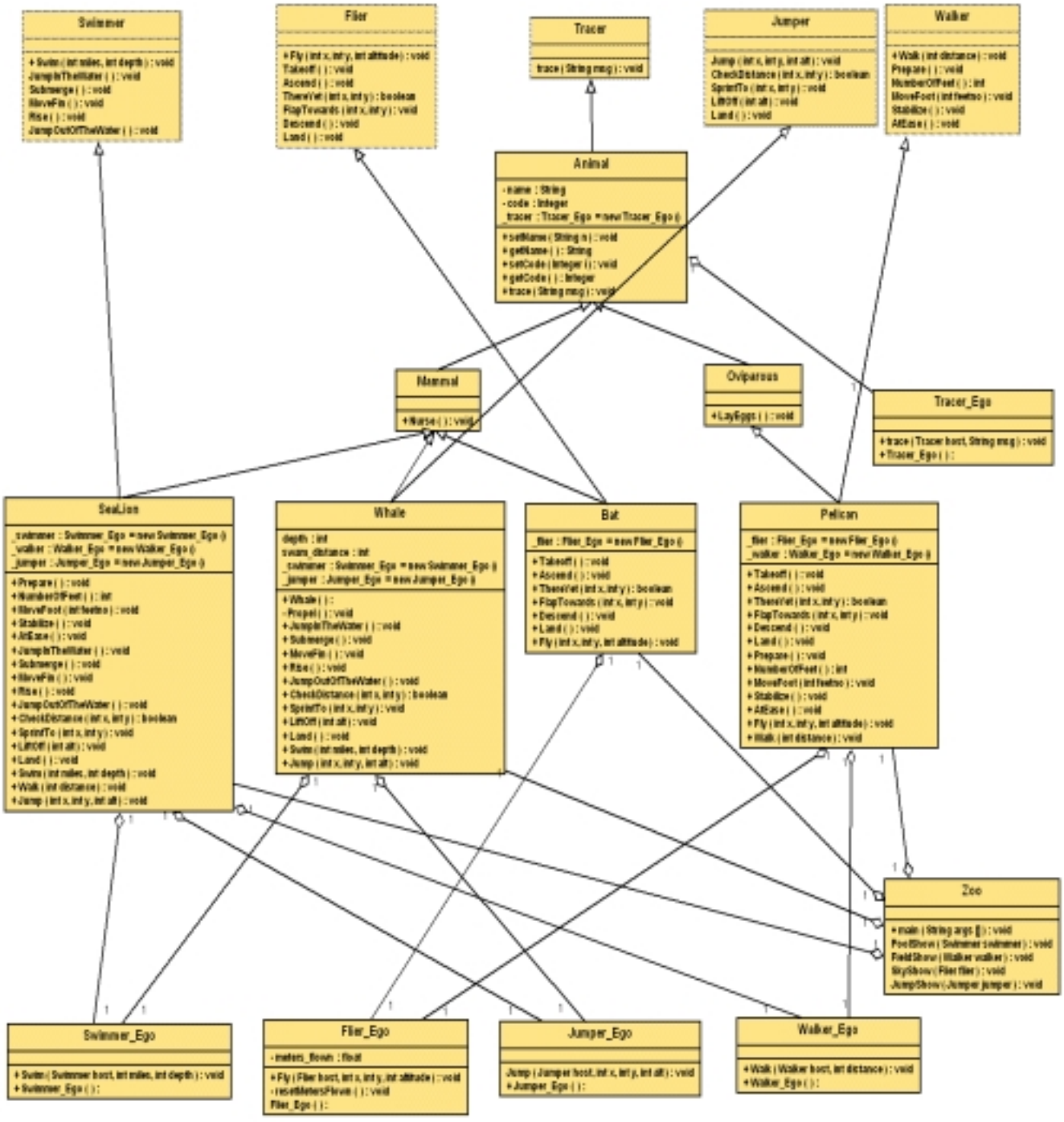


Figure 9: Generated Java system's relationships[i]

```
// SeaLion.java [static]
import java.util.*;
public class SeaLion
 extends Mammal
 implements Walker,Jumper,Swimmer
{
  // for Walker
  public void Prepare() {
    trace( "SeaLion.Prepare()" ); }
  public int  NumberOfFeet() {
    trace( "SeaLion.NumberOfFeet()" );
    return 2;
  }
  public void MoveFoot(int feetno) {
    trace( "SeaLion.MoveFoot()" ); }
  public void Stabilize() {
    trace( "SeaLion.Stabilize()" ); }
  public void AtEase() {
    trace( "SeaLion.AtEase()" ); }
  // for Swimmer
  public void JumpInTheWater() {
    trace( "SeaLion.JumpInTheWater()" );}
  public void Submerge() {
    trace( "SeaLion.Submerge()");  }
  public void MoveFin() {
    trace( "SeaLion.MoveFin()");  }
  public void Rise() {
    trace( "SeaLion.Rise()"); }
  public void JumpOutOfTheWater() {
   trace("SeaLion.JumpOutOfTheWater()");}
  // for Jumper
  public boolean
  CheckDistance(int x, int y) {
    trace( "SeaLion.CheckDistance()" );
    return true;
  }
  public void SprintTo(int x, int y) {
    trace( "SeaLion.SprintTo()" ); }
  public void LiftOff(int alt) {
    trace( "SeaLion.LiftOff()" ); }
  public void Land() {
    trace( "SeaLion.Land()" ); }
  // ============== for Swimmer
  Swimmer$Ego $swimmer=new Swimmer$Ego();
  public void Swim(int miles, int depth)
  {
    $swimmer.Swim(this, miles, depth);
  }
  // ============== for Walker
  Walker$Ego $walker = new Walker$Ego();
  public void Walk(int distance) {
    $walker.Walk(this, distance);
  }
  // ============== for Jumper
  Jumper$Ego $jumper = new Jumper$Ego();
  public void Jump(int x, int y, int alt)
  {
    $jumper.Jump(this, x, y, alt);
  }
}
```

Code Sample 6: `SeaLion.java`[7]

```
// Swimmer$Ego.java [static]
public class  Swimmer$Ego
{
  public void Swim(Swimmer host,
                   int miles, int depth) {
    host.JumpInTheWater();
    for (int d = 0; d < depth; d++)
      host.Submerge();
    while ((miles--) > 0) host.MoveFin();
    for(int d = depth; d > 0; d--)
      host.Rise();
    host.JumpOutOfTheWater();
  }
  public Swimmer$Ego()
  {  }
}
```

Code Sample 7: `Swimmer$Ego.java`[2]

The implementation of the `$Ego` classes, however, is basically identical to that present in the personality definition itself. The compiler needs to make sure it places the reference to the

[7] Code automatically inserted by the Personalities/J compiler is shown in **bold** typeface.

`host` variable in a few critical locations. The algorithm presented in Code Sample 8 can help explain the process by which the compiler determines how to create the `$Ego` files.

```
/* process_method: given personality P and method m,
   insert host wherever appropriate */
proc process_method(P, m) {
    ret ← return type of «m»
    nam ← name of «m»
    par ← parameter list of «m»
    println( "public " + «ret» + " " + «nam»+
    "(" + «P» + "host, " + «par» + ") {" )
    for each statement stm in «m» do
        for each token tok in «stm» do
            if «tok» is not a function call
                print( «tok» )
            else
                if «tok» is part of the DI of «P»
                    print( "host." + «tok» )
                else
                    print( «tok» )
                endif
            endif
    endfor
    println()
    endfor
    println( "}" )
endproc

/* Generate the $Ego file given «P» */
proc gen_ego(P)
   println( "public class " + «P» + "$Ego" )
   println( "{" )
   foreach UI method uim in «P» do
     process_method( «uim» )
   endfor
   foreach private data member mbr in «P» do
     println( "private " + «mbr» )
   endfor
   foreach private function pfn in «P» do
     println( «pfn» )
   endfor
   println( "public " + «P» + "$Ego() {" )
   con ← constructor of «P»
   process_method( «con» )
   println( "} }" )
endproc
```

Code Sample 8: Pseudo code for generating the `$Ego` classes

Last, but certainly not least, the Personalities/J compiler must generate the Java interface file. Code Sample 9 shows an example of such a file for the `Swimmer` personality, whereas Code Sample 10 presents the simple algorithm to generate these interfaces.

```
// Swimmer.java [static]
interface Swimmer
{
  public void Swim(int miles, int depth);
  void JumpInTheWater();
  void Submerge();
  void MoveFin();
  void Rise();
  void JumpOutOfTheWater();
}
```

Code Sample 9: The Java interface `Swimmer.java`[8]

[8] Code automatically generated by the Personalities/J compiler is shown in **bold** typeface.

Figure 10 shows a dynamic view of a client instantiating a new object that in turns personifies the `Swimmer` personality. The client in the diagram (`aZoo`) is not involved in creating the different `$Ego` classes for the object. The code automatically inserted by the Personalities/J compiler takes care of that and the whole process is transparent to `aZoo`, who needs to know only how to create a `SeaLion` named `toto`, and nothing else.

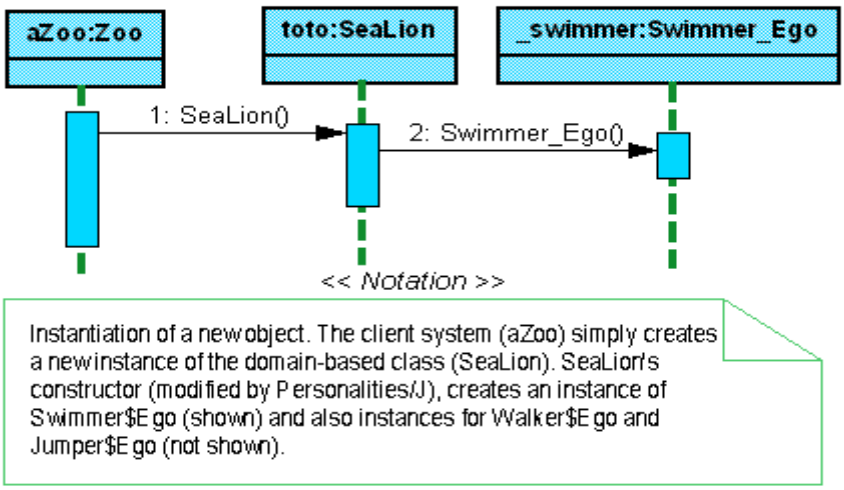


Figure 10: Creation of objects and their associated `$Ego` classes[9]

```
/* gen_interface: given personality P create the interface file */
proc gen_interface(P)
    println( "interface " + «P» + "{" )
    foreach UI method uim in «P» do
        sig ← signature of «uim»
        println( «sig» )
    endfor
    foreach DI method dim in «P» do
        println( «dim» )
    endfor
    println( "}" )
endproc
```

Code Sample 10: Pseudo code for generating the Java interface file

[9] The `$` sign in the identifiers has been replaced by an underscore (_) because of diagramming tool inadequacies

```
/* gen_class: generate the Java file for the class
   definition given the class PJ file C */
proc gen_class( C )
    ext ← parent of «C»
    println( "public class " + «C» )
    if «ext» is not empty
        println( "extends " + «ext» )
    endif
    per ← personifies list of «C»
    if «per» is not empty
        if «ext» is not empty
            print( ", " )
        endif
        print( " implements " + «per» )
    endif
    imp ← implements list of «C»
    if «imp» is not empty
        if «per» is not empty
            print( «imp» )
        else
            if «ext» is not empty
                print( ", " )
            endif
            println( " implements " + «imp» )
        endif
    endif
    println( "{" )
    bod ← body of «C»
    println( «bod» )
    foreach personality p in «per» do
        pp ← «p» with first letter's case changed
        println( «p» + "$Ego $" + «pp» +
                "= new " + «p» + "$Ego();" )
        foreach UI method uim in «p» do
            ret ← return type of «uim»
            nam ← name of «uim»
            par ← parameter list of «uim»
            println( "public " + «ret» + " " +
                    «nam» + "( " + «par» + ") {" )
            println( "$" + «pp» + "." + «nam»
                    + "(this, " + «par» + "); }" )
        endfor
    endfor
    println( "}" )
endproc
```

Code Sample 11: Pseudo-code for creating the Java class files

Figure 11 shows a sequence diagram corresponding to a client (`aZoo`) accessing our previously created `SeaLion toto`. The client uses the `Swim()` method, which is part of `Swimmer`'s upstream interface to contact `toto`. The proxy code that has been inserted by the Personalities/J compiler in the `SeaLion` class immediately delegates to the `Swimmer$Ego` instance, which was previously created at construction time. The proxy, however, adds a reference to itself (i.e. its class) on the call to `Swimmer$Ego`, so that the latter can call `toto` back for its implementation of the downstream interface methods.

THE CHANGES FOR DYNAMIC PERSONALITIES

Chapter 3 presented two approaches for dynamic personalities. In this work, we will present our implementation of Indecisive Personalities, leaving an implementation of the fully dynamic approach for future work. It is the author's belief that the constraints imposed by

indecisive personalities are not drastic, and more than offset by the simplicity of the implementation.

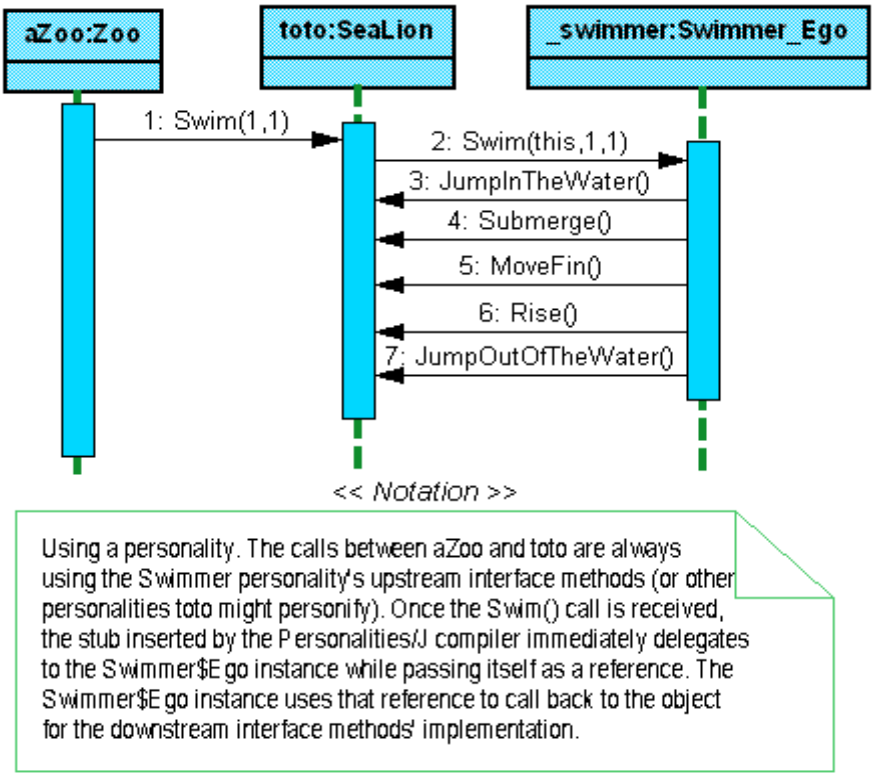


Figure 11: Using an upstream interface method[10]

When defining the implementation patterns for dynamic personalities, we strived to make the changes as small as possible, both in the user-space code (i.e. the `*.pj` files) and in the Java-space code. Thus, we do not require any changes at all in the `*.pj` files for the personality and the classes' definitions. We do, however, require changes in the client's code, since it must now deal with the additional behavior of "attaching" and "detaching" a personality.

[10] The `$` sign in the identifiers has been replaced by an underscore (_) because of diagramming tool inadequacies

PERSONALITIES' PROTOCOL

In order to bring some commonality to all personalities, we extended the concept and provided the interface shown in Figure 12, supported automatically by the system, for all personalities.

`personify( "<personality>" )`	Enable personification of `<personality>`
`forget( "<personality>" )`	Disable personification of `<personality>`
`personifies( "<personality>" )`	Returns `true` or `false` depending on whether `<personality>` is enabled in the class.
`personalities()`	Returns a `Vector` of `String`s with the names of all the personalities that are enabled in the class.

Figure 12: Dynamic Personalities' common protocol

CLIENT CODE CHANGES

In our running example (see Appendix A), we have made the assumption that the client code is the one responsible for updating the state of its classes. For example, we assume that the client code (i.e. `Zoo.pj`) will enable and disable the personalities for the animals in the Zoo. This assumption is not entirely correct, as any class with a reference to the object and knowledge of the object's personifies set could have directed the object to enable or disable a given personality. This is arguably not strong enough. We have not addressed the issue of restricting the protocol-defined operations that a client class can perform on a personifying class in this work. Creating a more secure way for updating the state of a class with respect to its list of enabled personalities is left as a future work.

The client code, therefore, needs first to create an object, and enable/disable personalities as it sees fit. Code Sample 12 shows an abbreviated version of the `Zoo.pj` client shown in Appendix A. Notice how the client enables or activates the different personalities. It also disables them through `forget()`. Last, notice how the client can check for the state of any

given personality before dispatching a call. Alternatively, it could dispatch and receive an error or exception.

```
// smallZoo.pj
void main() {
  // create a new object
  SeaLion toto = new SeaLion();
  toto.personify( “Swimmer” );  // enable Swimmer
  toto.personify( "Jumper" );   // and Jumper
  // this should print [Swimmer,Jumper]
  System.out.println(toto.personalities());
  toto.personify( "Walker" );   // enable Walker
  DoShow(toto);
  toto.forget( "Walker" );
  toto.forget( "Jumper" );
  toto.forget( “Swimmer” );
}
void DoShow(Object animal) {
  if ( animal.personifies( "Swimmer" )
    ((Swimmer).animal).Swim(10,10);
  if ( animal.personifies( "Walker" ) )
    ((Walker).animal).Walk(10,10);
  if ( animal.personifies( "Flier" ) )
    ((Flier).animal).Fly(10,10,10);
  if ( animal.personifies( "Jumper" ) )
    ((Jumper).animal).Jump(10,10);
}
```

Code Sample 12: A dynamic client

THE GENERATED JAVA CODE

The Personalities/J compiler generates pretty much the same files as for the static case. The contents of the files, however, are slightly modified to support keeping the personalities table at each class. We define a class, `Shrink`, which keeps the list of the personalities that are being enabled. This class acts as a singleton on a given inheritance chain. That is, there should be only one `Shrink` object in the chain, and it should be located at the topmost class that personifies anything. Code Sample 13 shows a simple algorithm for making sure the `Shrink` class is allocated appropriately.

```
/* add_shrink: given class definition c from c.java,
   put the Shrink object and proxies in c.java */
proc add_shrink( c )
    mbr ← data members of «c»
    if «mbr» does not contain "Shrink $shrink"
        mbr ← «mbr» +
    "protected Shrink $shrink=new Shrink();
    public boolean personify(String what)
    { return $shrink.personify(what); }
    public boolean personifies(String what)
    { return $shrink.personifies(what); }
    public boolean forget(String what)
    { return $shrink.forget(what); }
    public boolean canpersonify(String what)
    { return $shrink.canpersonify(what); }
    public Vector personalities()
    { return $shrink.personalities(); } "
    endif
    recreate «c».java using new «mbr»
endproc

/* place_shrink: utility function to keep last plausible
   Shrink placement while searching for a new one */
proc place_shrink(c, f)
    per ← personifies list of «c»
    if «per» is not empty
        f ← «c»
    endif
    ext ← extends of «c»
    if «ext» is not empty
        place_shrink( «ext», «f» )
    else
        add_shrink( «f» )
    endif
endproc

/* do_shrink: starts the process of finding the
   right place to put the Shrink object */
proc do_shrink( C )
    per ← personifies list of «C»
    if «per» is not empty
        place_shrink( «C», «C» )
    endif
endproc
```

Code Sample 13: Adding `Shrink` at the correct place in the inheritance chain

Once we have the `Shrink` instance properly located in the chain, we can modify the code for the classes in the inheritance chain. Each `$Ego` class will now use the `Shrink` object as a repository for the personalities. Thus, the chain-singleton `Shrink` is passed to the `$Ego` classes in its construction. Also, all the proxy stubs for the upstream interface functions now check to make sure that the object has the personality "enabled" before delegating. Code Sample 14 shows the algorithm for generating these classes (changes in bold).

```
/* gen_class: generate the Java file for the class
   definition given the class PJ file C  [dynamic] */
proc gen_class( C )
    ext ← parent of «C»
    println( "public class " + «C» )
    if «ext» is not empty
        println( "extends " + «ext» )
    endif
    per ← personifies list of «C»
    if «per» is not empty
        if «ext» is not empty
            print( ",  " )
        endif
        print( " implements " + «per» )
    endif
    imp ← implements list of «C»
    if «imp» is not empty
        if «per» is not empty
            print( «imp» )
        else
            if «ext» is not empty
                print( ", " )
            endif
            println( "implements " + «imp» )
        endif
    endif
    println( "{" )
    bod ← body of «C»
    println( «bod» )
    do_shrink( «C» )
    foreach personality p in «per» do
        pp ← «p» with first letter's case changed
        println( «p» + "$Ego $" + «pp» +
            "= new " + «p» + "$Ego($shrink);" )
        foreach UI method uim in «p» do
            ret ← return type of «uim»
            nam ← name of «uim»
            par ← parameter list of «uim»
            println( "public " + «ret» + " " +
                «nam» + "( " + «par» + ") {" )
            println( "if (personifies( " +
                «p» + "))" )
            println( "$" + «pp» + "." + «nam»
                + "(this, " + «par» + "); }" )
        endfor
    endfor
    println( "}" )
endproc
```

Code Sample 14: Generating Java class files (dynamic version)

The `$Ego` classes, in turn, are the living proof that this object personifies the personality. Thus, it is the perfect place where to implement the registration with the `Shrink` object. Each `$Ego` class thus needs to register itself with the `Shrink`. Personalities/J needs to modify the creation of the `$Ego` files slightly to accommodate this requirement. Code Sample 15 shows the new algorithm, with the only change being adding an input parameter to the constructor (the `shrink` object), and always generating a non-empty constructor that registers the personality with the shrink (shown in bold).

```
/* process_method: given personality P and method m,
   insert host wherever appropriate [dynamic] */
proc process_method(P, m) {
    ret ← return type of «m»
    nam ← name of «m»
    par ← parameter list of «m»
    println( "public " + «ret» + " " + «nam»+
     "(" + «P» + "host, " + «par» + ") {" )
    for each statement stm in «m» do
        for each token tok in «stm» do
            if «tok» is not a function call
                print( «tok» )
            else
                if «tok» is part of the DI of «P»
                    print( "host." + «tok» )
                else
                    print( «tok» )
                endif
            endif
    endfor
    println()
    endfor
    println( "}" )
endproc

/* Generate the $Ego file given «P» */
proc gen_ego(P)
  println( "public class " + «P» + "$Ego" )
  println( "{" )
  foreach UI method uim in «P» do
    process_method( «uim» )
  endfor
  foreach private data member mbr in «P» do
    println( "private " + «mbr» )
  endfor
  foreach private function pfn in «P» do
    println( «pfn» )
  endfor
  println( "public " + «P» +
           "$Ego(Shrink shrink) {" )
  con ← constructor of «P»
  process_method( «con» )
  println( "shrink.register_personality(" + «P» + ");" )
  println( "} }" )
endproc
```

Code Sample 15: Generating `$Ego` files for the dynamic case

Chapter 5

PERSONALITIES AND THEIR BIG COUSINS

As explained in previous chapters, personalities only claim to help in the micro-framework space. Getting more ambitious and thinking about system-wide scope, there exist a number of approaches for encapsulating behavior. Frameworks are the most common case. We explore how Personalities can help Frameworks by becoming the join points. We then briefly consider other, more advanced, collaboration-based works.

FRAMEWORKS AND PERSONALITIES

The concept of personalities turns out to be a good vehicle for embodying the *hotspots* in a framework. The hotspots are those classes that a user of a framework needs to either subclass or specialize in some way to "adapt" the framework to her application. In the general case, these hotspots are abstract classes that need to be subclassed by the application developer. We claim that using personalities as hotspots is a cleaner approach and it also overcomes certain programming language limitations.

ADAPTING A FRAMEWORK USING HOTSPOTS

A framework can be characterized as a collection of classes that embody a certain number of specific behaviors. Just as personalities encapsulate functionality in their upstream interface methods' implementation, frameworks encapsulate functionality through the interactions of several prototypical classes. Designing at the framework level is desirable because you can concentrate on the semantics of the functions you are trying to implement, without being bothered by the specific details of the class graph that will eventually embody the framework. In other words, frameworks assume an "ideal" situation and implement their behavior in that vacuum.

However, in order for frameworks to be usable, they need to be "plugged" or adapted into an application[11]. The framework developer identifies the pieces of information that are required from the application. These will be later used to customize the framework (which is supposed to be generic) to each specific application. The pieces of information that are required from the application usually take the form of classes in object-oriented programming but object-oriented programming is by no means a requirement for the existence of frameworks. As a matter of fact, some of the more successful frameworks so far are not dependent on object-oriented programming languages. For instance, The X Window System, is based in the C programming language.

Once the hotspots have been identified, there are potentially two ways in which an application "connects" itself with the framework:

1. Using inheritance: the application developer subclasses the hotspot classes and implements a number of abstract methods (only applicable with OO).

2. Using delegation: the application developer implements a set of predefined method in her own classes and then 'registers' these classes with the framework.

Using inheritance is a simpler approach since the application developer need not worry about specifically instantiating and registering the two different sets of objects (i.e. the framework objects and the application objects). However, class inheritance is not convenient when you have one of the following situations:

- There is one application-level class that could be used to embody two or more framework-level classes.

- The application-level class that can embody the framework-level class is already inheriting from another application-level class.

[11] Whether the framework is "adapted" to the application or the application "adapts" to the framework is a subject of heated dispute among practitioners ☺.

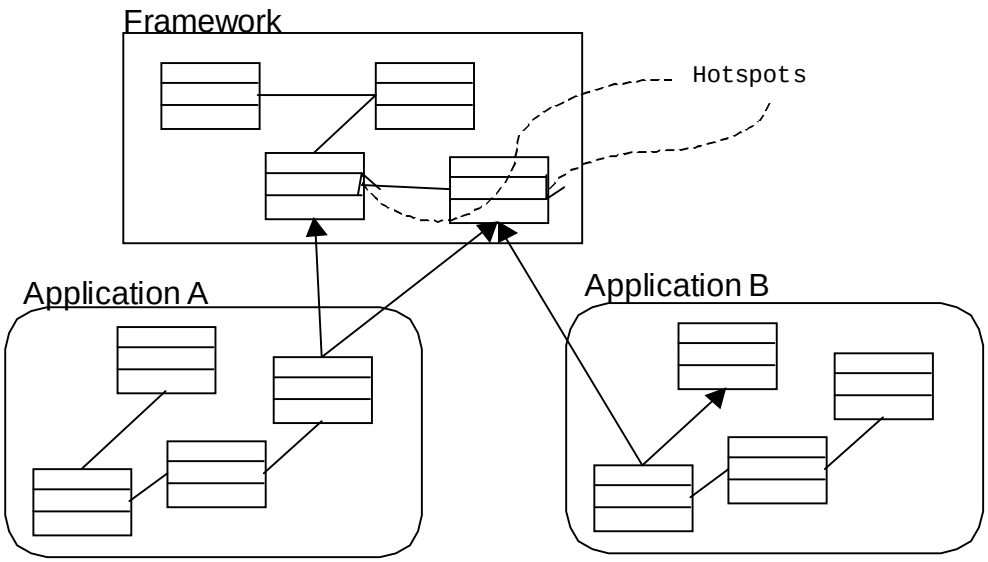


Figure 13: Subclassing hotspots in framework instantiation

Both these situations lead to the need for multiple inheritance. Alternatively, adapters and/or proxy classes [Gamma94] need to be created and maintained. Figure 13 depicts these two problems. A framework with two hotspots is adapted by two different applications. "Application A" has one class that can potentially subsume the functionality of both hotspots, leading to multiple inheritance, while "Application B" has a class that provides the functionality for one of the framework's hotspots but it is already extending another class in the application, which again leads to multiple inheritance.

Using delegation as the mechanism for joining an application with a framework avoids these problems at a considerable expense in complexity. The application developer must now make sure that the appropriate application classes are registered with the framework so they can be called by it as part of the behavior the framework encapsulates. Delegation has another side effect that might be undesirable. Since at runtime there will be two live objects, the hotspot object and the delegation object, two object references will need to be maintained. A decision needs to be made as to which object identity prevails at each layer of abstraction. Since the framework will unavoidably call its own hotspot "object" for the behavior it needs, the identity of the hotspot object seems to prevail. However, that object

reference will potentially not be enough for other application-level client objects since they might need to interact with the object through a protocol not covered in the framework's. Using delegation of protocol, as presented in [Wieringa95] is a theoretic solution to this problem. It essentially states that all methods not covered in the receiving object's repertoire should be passed on to the object associated to it. However, this solution will not work in strongly typed languages, such as Java, since the hotspot type cannot be "cast" to the application type. Figure 14 shows this problem in more detail.

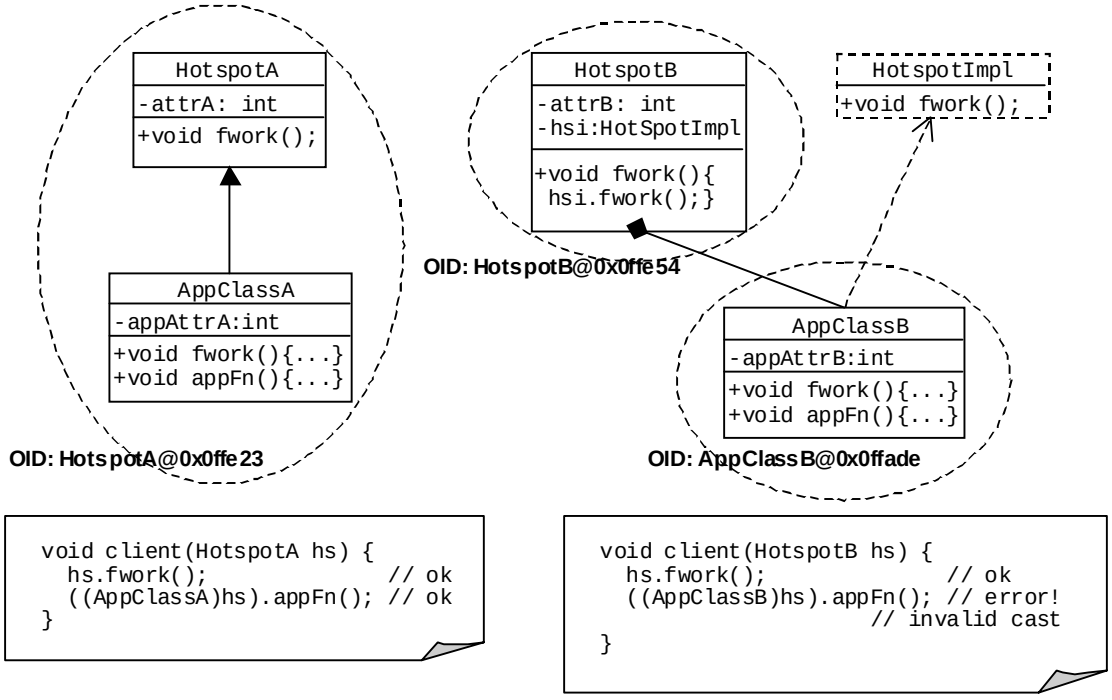


Figure 14: Delegation and object identity

PERSONALITIES AS HOTSPOTS

A better mechanism for implementing the hotspots in a framework is required. We believe the Personalities concept is ideally suited for this purpose. By making each hotspot a personality, we take advantage of all the personality's additional semantics and compile-time validation checks. Furthermore, we allow application developers to freely personify hotspots

in a clean, inheritance-like way, while still solving the different problems presented in the previous section.

Personalities do encapsulate behavior at the micro-level. Being a realization of the template-method pattern [Gamma94], 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. Framework developers, however, might have defined completely empty classes, with no implementation whatsoever. In these cases, the personality could degenerate to provide a pass-through for all its downstream interface functions. While the additional benefits of the template method pattern would be lost, the developers can still leverage the preservation of identity and the freedom from the multiple inheritance problem that personalities provide.

PERSONALITIES AS TRAFFIC COPS

Our main point in proposing personalities as the glue between any collaboration-based encapsulation of behavior (i.e. frameworks/APPCs/CGVs, etc) is that personalities' clear semantics can significantly ease the job of plugging frameworks together.

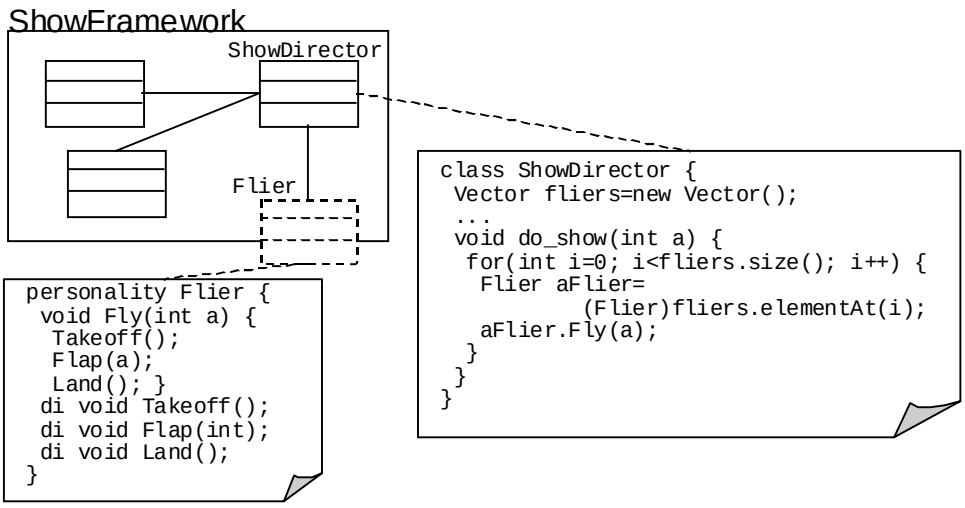


Figure 15: `ShowFramework` with `Flier` as hotspot

As an illustrative example, let's consider the case where we have decided to purchase a `ShowFramework` framework for our Zoo system. For the sake of simplicity, we'll omit most of the details and will make `ShowFramework` only deal with air shows. As such, the framework needs to know only about the `Flier` personality. `Flier` contains all the semantics that `ShowFramework` requires to perform its job. Figure 15 shows a diagram with some pseudo-code for one of `ShowFramework`'s classes and a simplified version of the `Flier` personality.

Now imagine that instead of wanting to build a flying show for the Zoo, you decide to build a flying demonstration for a plane show. What you need to do, then, is to personify `Flier` in your application (i.e. `Takeoff()`, `Flap()`, and `Land()`) and off you go. However, you early on realize that most of your planes actually takeoff and land in pretty much the same way. Thus, you decide to obtain another framework (from potentially another manufacturer) that encapsulates the behavior of mechanical planes that need to takeoff and land the way planes do. Let's call this the `TakeoffAndLandFramework.`

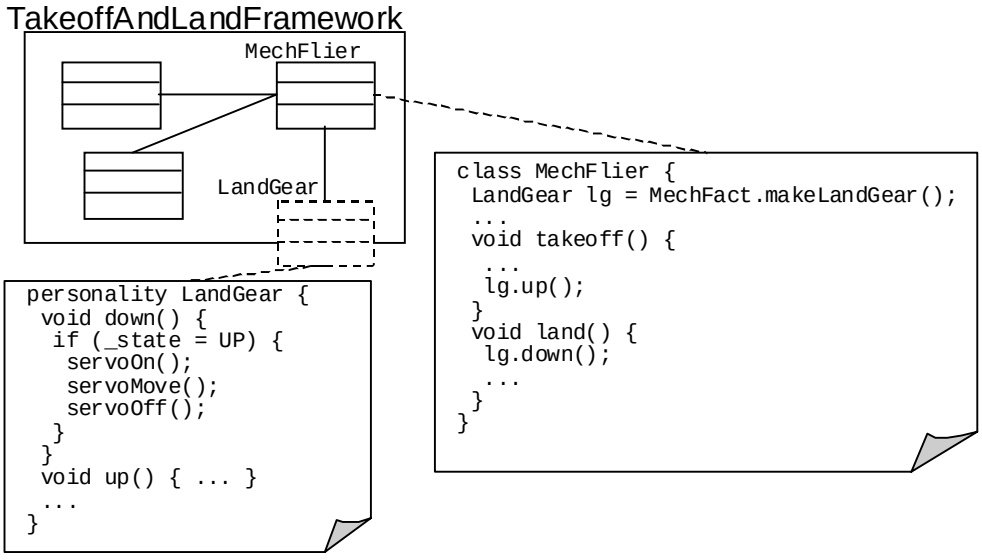


Figure 16: `TakeoffAndLandFramework` with `LandGear` as hotspot

It is quite possible that this `TakeoffAndLandFramework` provides the application developer with one (or more) functions to trigger the execution (just as the `do_show()`

function in the `ShowDirector` class is the entry point in the `ShowFramework` shown in Figure 15). For the sake of simplicity, let's assume that these functions are called `takeoff()` and `land()` respectively. Figure 16 shows a diagram of this framework. It uses personality `LandGear` as its hotspot. Some details have been omitted to keep the diagram uncluttered.

So now you have purchased two different frameworks that might ease your job as the application developer. Still, you need to glue them together. As explained in a previous section, personalities can be used as hotspots very effectively to overcome some of the programming language limitations that might arise in the use of frameworks. Personalities also provide a nice demarcation point between frameworks. What is more, this demarcation point is (usually) not shallow, but rather contains behavior in itself. This helps reinforce the semantics required of the downstream interface from the application. In keeping with the current example, you might have an application that looks like Figure 17. There are two simple associated hierarchies, one for planes and another for wheels of planes.

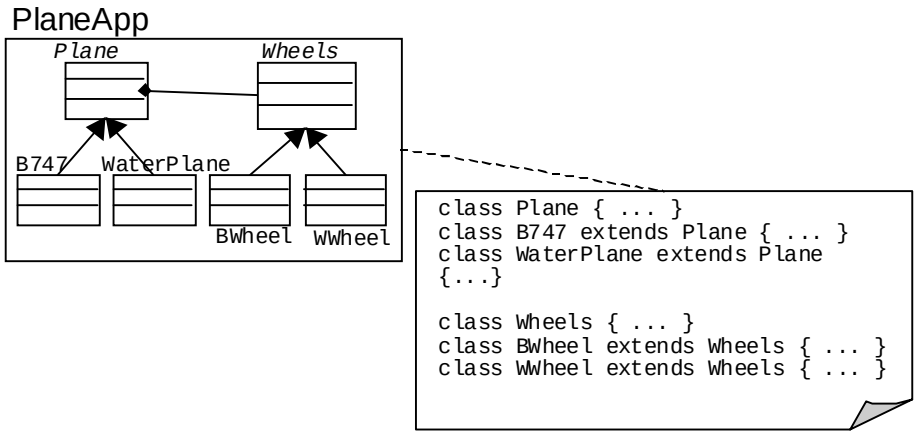


Figure 17: `PlaneApp`, the original application

The goal would be to make use of the `ShowFramework` and the `TakeoffAndLandFramework` wherever possible. For that, we need to plug into the hotspots `Flier` and `LandGear`, respectively. Furthermore, we realize that while our

"wheeled" planes can (and should) use the `TakeoffAndLandFramework`, our water planes obviously should not, since they do not even have wheels[12].

There are at least two alternatives to achieve our goals: composing frameworks automatically using personalities, and delegating the composition to the client's code. Let's examine each of these in detail.

Composing Frameworks using Personalities

The most obvious approach is to create a new framework composing both. For instance, the `Show_TakeoffAndLandFramework`, shown in Figure 18, would support the `do_show()` interface and will require `LandGear` as its hotspot. This has in effect achieved an increase of granularity plus a decrease of abstraction in the hotspot (i.e. more control) while preserving the high-level behavior (i.e. `do_show()`).

The process of composing two, already-existent frameworks can be automated through the use of a compiler that we will call a *compositor* process. Figure 18 shows the changes that the compositor needs to make to the frameworks in **bold** typeface. The reader might notice that the naming conventions used in this example are conspicuously similar. In other words, it was not by chance that the `Flier` personality requires `Takeoff()` and that the `MechFlier` class provides `takeoff()`! In other words, composing frameworks does not just "happen", but rather it should be a well thought-out process.

Another interesting point has to do with the completeness of the composition. In the example, we see that the `TakeoffAndLandFramework` will provide behavior for both `Takeoff()` and `Land()` but is clueless about what to do with `Flap()`. The compositor process therefore needs to determine which ones of the first framework's hotspot downstream interface methods will be provided and for those that will not, a new hotspot needs to be created that simply proxies the original DI signature with identical semantics.

[12] For reasons of symmetry, these "null" wheel classes (i.e. `WWheel`) are usually still modeled and given empty behavior.

That is why in our example, the final application will need to personify both `PartialFlier` and `LandGear`. The second framework class(es) that personify the hotspots of the first framework will get these "partial" personifying objects at runtime much in the same way any personifying object is bound. Also, the first framework's hotspot (i.e. `Flier`) is still available if the application decided to use only that part of the composed framework. Once the compositor has finished, we are left with a separate and complete framework that in no way depends on the other ones. It is in essence a new entity that we can use to develop systems. Changes in the original frameworks will obviously not be propagated to the new one, but then again running the compositor is trivial.

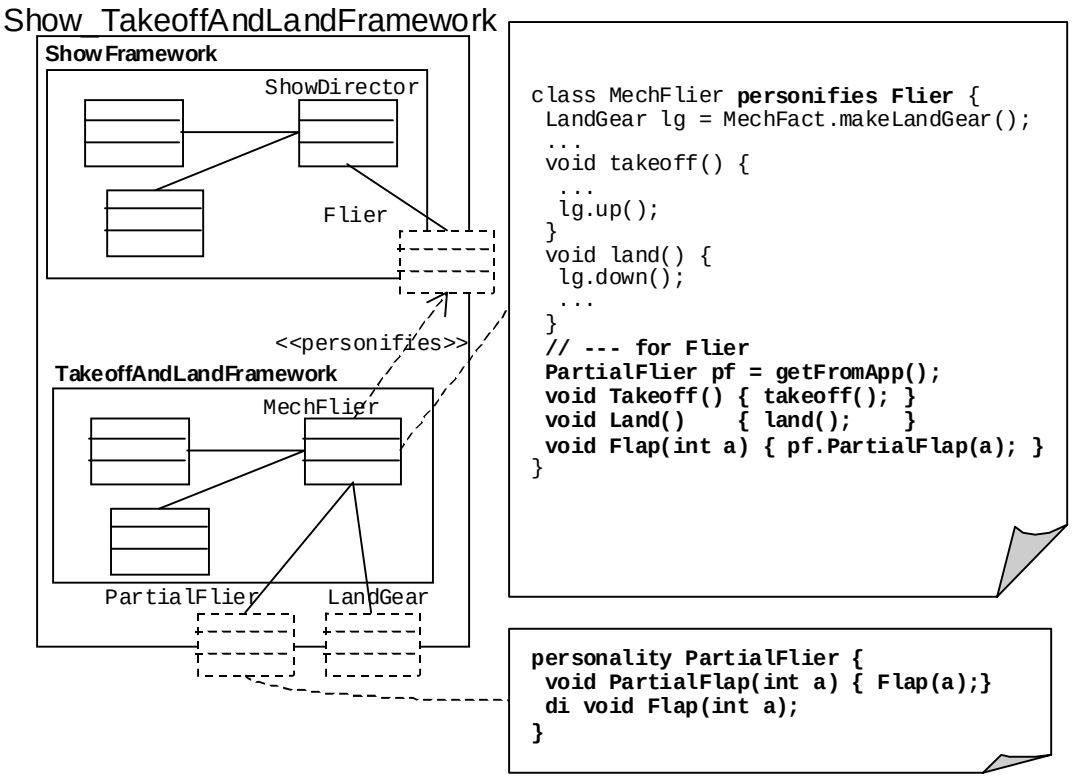


Figure 18: Composing two frameworks intrusively

Figure 19 shows our plane application using this composed framework. Notice how we make use of the fact that we can selectively use the first half of the composed framework (for our `WaterPlane`) and the entire framework for our Boeing plane. The underlying

assumption is, of course, that `B747` planes will contain `BWheel` classes. We thus need to personify `LandGear` at the `BWheel` class and take care of the "rest" of the original `Flier` interface at the `B747` class (by personifying `PartialFlier`).

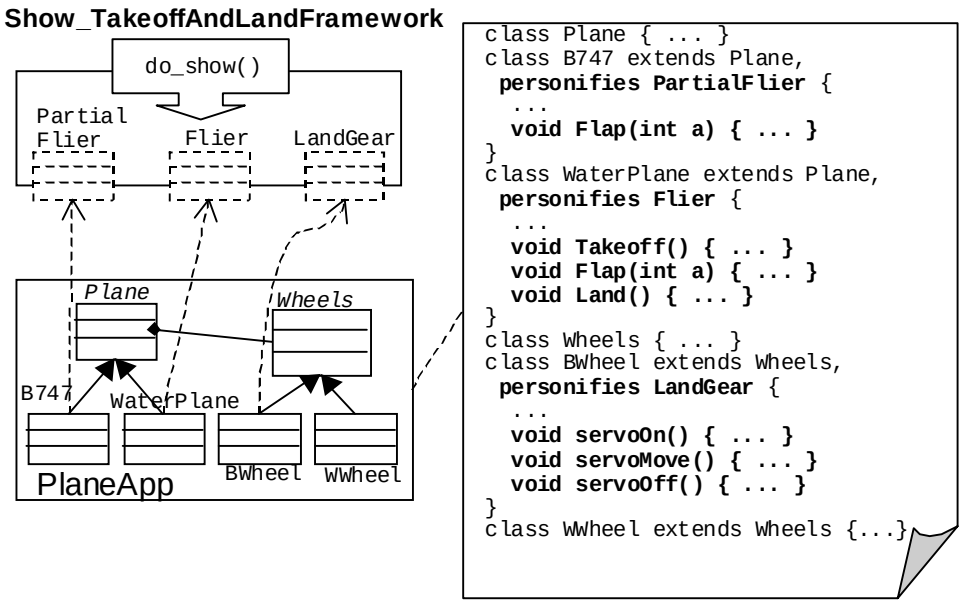


Figure 19: The plane application using the composed framework

Delegating Composition to the Application's Code

The previous solution for composing frameworks using personalities involve intrusive modifications to the original frameworks and effectively produces a brand new composed framework. For cases where changes to the original frameworks are not possible, or when it is desirable to keep the original frameworks "intact" (in order, for instance, to automatically install new versions or patches) then we can delay the composition infrastructure until the client makes use of the framework. This option will undoubtedly require more effort from the application developer, but it keeps the two (or more) frameworks being composed relatively isolated from each other.

To continue our running example, what we would need is to have all `Planes` personify `Fliers`. The `WaterPlane`, since it does not really have landing gear, will implement all the

`Flier`-required methods. The `B747`, on the other hand, can make use of the `TakeoffAndLandFramework` to implement that behavior. Since we are following a delegation model, all we really need to do is make sure that the `B747` class "has" an instance of the `TakeoffAndLandFramework` and can invoke it's high-level operations (that is, `takeoff()` and `land()`). We can then simply adapt the `Flier`'s requirements to these functions. However, our picture would be only halfway complete, since for the `TakeoffAndLandFramework` to work we need to personify its own hotspots. We do this much in the same way as we did it in the previous approach. Figure 20 depicts the situation and presents the pseudo-code.

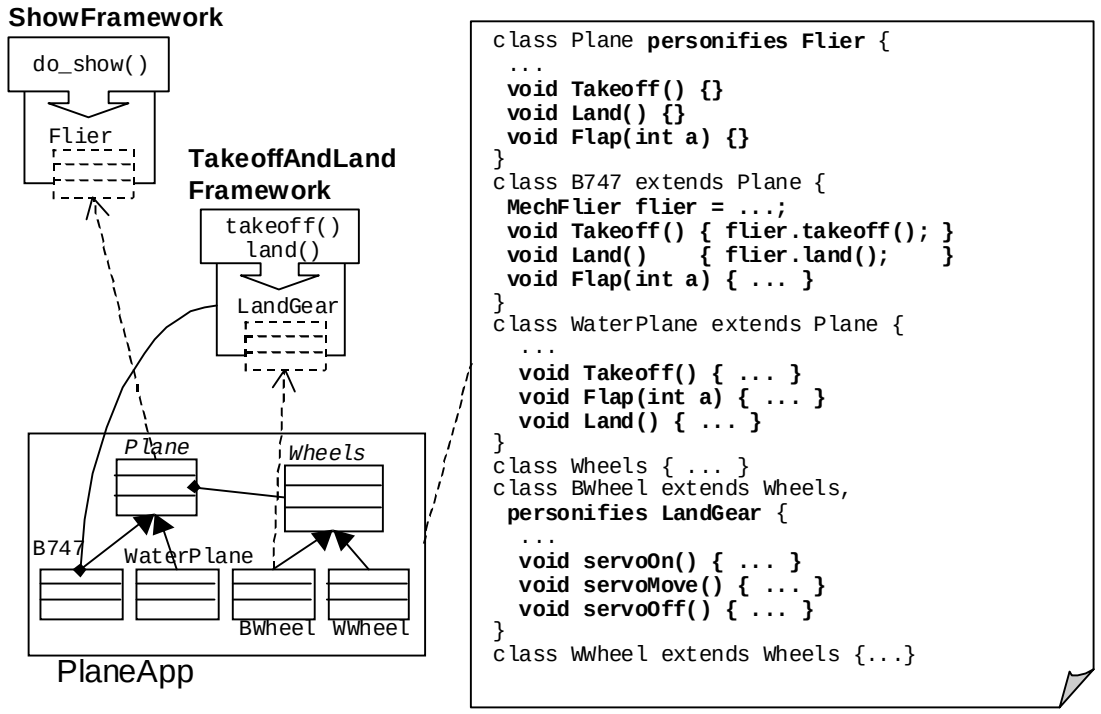


Figure 20: The Plane Application using two side-by-side frameworks

There are several items worth noticing in the above implementation. First, the two frameworks are kept separate. Second, the `B747` class contains a reference to the `MechFlier` object within the `TakeoffAndLandFramework` instance. Third, the details on how to construct the `BWheel` objects, how to pass them to the framework are not

shown. Fourth, the application developer needs to hand-write all the delegation code (in the B747 class).

While this second alternative to using multiple frameworks at different levels of abstraction in the same application might seem more problematic, it actually is a little bit cleaner from an architecture perspective. Not only are the two frameworks separated but also the connection between them is in the control of the application developer, who can solve minor "impedance mismatchs" between what one provides and what the other expects. The previous approach, on the other hand, requires an almost perfect correspondence in syntax and semantics for the automatic compositor to work. Also, as shown in the example, the previous approach might lead to a creation of artificial personalities to encapsulate the bits of behavior from the higher-level framework that were not provided by the lower-level framework and thus need to be passed along to the application for implementation.

Regardless of which method we select, the use of Personalities as the mediators between frameworks and applications allows us greater flexibility than traditional mechanisms. We believe that building a compositor as outlined in the preceding section is fairly simple and would constitute a useful tool for framework composition.

OTHER COLLABORATION-BASED WORK

Personalities are related to, and can be used in conjunction with, other ideas for encapsulating behavior at a larger granularity. Techniques such as Adaptive Plug-n-Play Components (APPCs) [Mezini98] and Class Graph Views (CGVs) [Ovlinger98] attempt to model behavior at a larger scale than that of personalities and can make use of the Personalities concept. In a sense, all these techniques are equivalent to frameworks and thus the Personalities concept can be used for embodying the contract or hotspot between them and the application code.

The purpose of an APPC is to encapsulate a slice of behavior. It essentially encapsulates one sequence interaction diagram. APPCs are similar in concept to frameworks, though they

differ substantially in their expressiveness and adaptability. An APPC has an entry point called the *main entry* method call. From that single method call, a number of interactions between different classes might take place. APPCs can be composed and they of course need to be applied to an application in order for them to “execute”.

The application ties into the APPC through something similar to the hotspot concept in framework technology [Johnson97]. We propose using Personalities as the glue between the APPC and the application, or between two APPCs. Since we will need one APPC per implementation of a downstream interface method, Personalities’ capability of easily multiply personifying several classes becomes essential.

CGVs are another approach for specifying different views on an application’s class diagram. These views encapsulate behavior much in the same way as APPCs do. CGVs define a concept that is somewhat similar, at least in structure, to a personality. It is called a *class definition* and it contains both *behavior methods* and *map methods*. These are analogous to a personality’s upstream and downstream interface methods, respectively. We believe that a good way to merge the two technologies would be to replace class definitions by personalities.

Chapter 6

FUTURE WORK

In this chapter we will present some of our ideas for future research work. We believe most of these to be attainable, and of practical use to software engineers.

PERFORMANCE RANGES OR GUARANTEES

An early version of this work was presented at a workshop about pragmatic issues in Framework technology at OOPSLA ’98. The idea of using of personalities as the interface layer between different frameworks was well received. Some suggestions were made to incorporate more information in the downstream interface specifications. More specifically, items such as memory requirements, average and worst execution speed, and scalability expectations would be very useful when application developer set out to use a framework through the use of Personalities.

While we wholeheartedly adhere to the intention of the workshop participants, we must note that we know of no way of programmatically verifying that a downstream implementation is within certain specified semantic ranges. Therefore, we must once again put our trust on the programmer.

MAPPING AND PARAMETER CONVERSION

When a personifying class wants to attach to a personality, it needs to define all the functions in that personality’s downstream interface. This is usually not a problem when we are dealing with brand-new applications but might become cumbersome if we have an existent application that we are extending or enhancing. For example, we might already have a Zoo

application but, after having added a theme park to our Zoo, we decided to purchase the `ShowFramework` for implementing the shows in our application.

In such an environment, we might have classes that can already provide the functionality required by a personality's downstream interface. However, there might be a mismatch in these functions name or, more generically, their signature. For example, if we were working for a space agency, we might have a class such as the one shown in Code Sample 16. If we wanted to have this class personify `Flier`, we shouldn't need to define `Takeoff()`, `Ascend()`, etc. so that they simply delegate to the respective `SpaceShuttle` methods.

```
class SpaceShuttle {
   void EngageRockets() { ... }
   void ExitAtmosphere() { ... }
   void Orbit(int x, int y) { ... }
   void EnterAtmosphere() { ... }
   void Land() { ... }
   boolean ThereYet(int x, int y) { ... }
}
```

Code Sample 16: `SpaceShuttle` class

A simple name mapping mechanism would remove the need for these artifices to be created. Assuming the mismatch is only in the method name, we could very easily define a mapping from each of the personality's downstream interface methods to the corresponding personifying class methods. The Personalities/J compiler could then automatically insert the delegation code as appropriate. Code Sample 17 shows one possible language extension to accommodate this mapping. Notice that neither `Land()` nor `ThereYet()` are mentioned in the map since they have the exact same signatures as the personality requires and therefore are handled by the "default" case.

```
class SpaceShuttle personifies Flier
   with EngageRockets = Takeoff,
        ExitAtmosphere = Ascend,
        Orbit = Flap,
        EnterAtmosphere = Descend
{
   void EngageRockets() { ... }
   void ExitAtmosphere() { ... }
   void Orbit(int x, int y) { ... }
   void EnterAtmosphere() { ... }
   void Land() { ... }
   boolean ThereYet(int x, int y) { ... }
}
```

Code Sample 17: `SpaceShuttle` class with name-mapped personification

Even such a simple 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.

So far, we have only considered the case where the only mismatch between the personality requirements and the existent class were the names of the methods providing the behavior. What about slight differences in parameters? We can extend this simple mapping to accommodate minor semantic differences. If, for instance, you have purchased your ordering system and your data collection system from different vendors, you might get personalities that speak about the same ideas but in slightly different terms. Code Sample 18 shows one possible situation in which two different personality providers have developed their interfaces using different measurement systems. We call these two personalities *semantically equivalent but syntactically different*, or SEBSD for short.

```
// 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);
}
```

Code Sample 18: Two SEBSD personalities

In such a case the original mapping approach would not work since there's a unit conversion that needs to take place in between. Thus, the application developer needs to both do the conversion and also delegate. Code Sample 19 shows one possible class that personifies both personalities and performs this conversion and delegation.

```
class TheRedAppleInc extends AppleCo
  personifies FruitProducer, FreezerUser
{
  ...
  public boolean CheckStockHas(double kilograms) {
    // behavior is implemented in metric system
  }
  public boolean ValidateProdQuant(double pounds) {
    return CheckStockHas( pounds * 0.454 );
  }
}
```

Code Sample 19: Personifying two SEBSD personalities without parameter conversion

The addition of some new syntax and compiler support to provide “on-the-fly” parameter conversion would allow these problems to be overcome. Code Sample 20 shows what the client code would look like with the new syntax.

```
class TheRedAppleInc extends AppleCo
  personifies FruitProducer, FreezerUser
   with CheckStockHas = ValidateProdQuant(<pounds>*2.2)
{
   ...
   boolean CheckStockHas( double kilograms ) { ... }
   // ValidateProdQuant needs not be implemented at all
}
```

Code Sample 20: Personifying two SEBSD personalities with parameter conversion

INHERITANCE OF PERSONALITIES

Providing extensibility of personalities via inheritance seems a good way to specialize a given personality and reduce the requirements on the implementing application. In such a context, we would usually have a higher-level personality and a lower-level personality that provides the downstream implementation for the first one and in turn requires an even lower-level, higher granularity implementation from the application.

Extending personalities in such a way would achieve the reduction of the level of abstraction since as an application developer you might not be ready to implement certain personality’s downstream interface (i.e. it might be too high-level for you) but you might know how to deal with lower-level problems. In such a case, we envision a middle behavior layer (that is, another personality) that would simply provide “generic” behavior for the higher level personality and reduce the abstraction or complexity on the downstream interface it requires

from clients. This extension can be viewed as triangle, with abstraction decreasing as we move down while granularity increases, as shown in Figure 21.

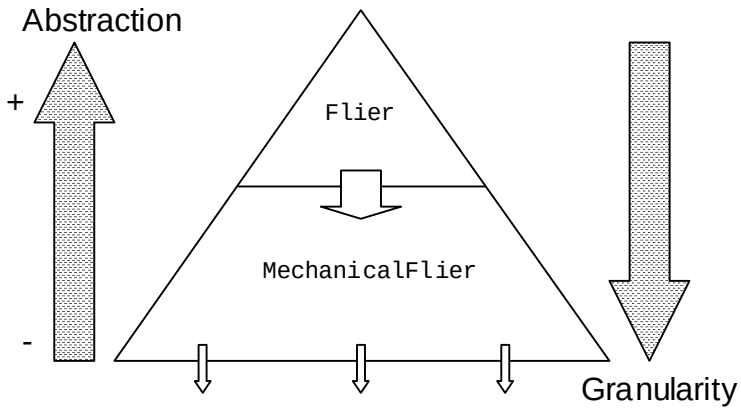


Figure 21: Inheritance of Personalities w.r.t. abstraction and granularity

Inheritance of personalities can keep the exact same semantics as in Java inheritance. A `MechanicalFlier` personality can extend a `Flier` personality as shown in Code Sample 21 (details have been omitted and the code might not be representative of a real plane, but does present the difference in abstraction and granularity that we have mentioned before).

```
personality MechanicFlier extends Flier
{
   void TakeOff() {
     gotoRunway();
     waitForClearance();
     speedUp();
     increaseFlaps();
     bringLandingGearUp();
   }
   void Ascend() { ... }
   void FlapTowards(int x, int y) { ... }
   boolean ThereYet(int x, int y) { ... }
   void Descend() { ... }
   void Land() { ... }
   di void gotoRunway();
   di void waitForClearance();
   di void speedUp();
   di void increaseFlaps();
   di bringLandingGearUp();
   ...
}
```

Code Sample 21: `MechanicFlier extends Flier`

Notice that this `MechanicFlier` is different from that we presented in our discussion about frameworks, which we termed `MechFlier`. `MechanicFlier` implements its behavior (basically `Flier`'s downstream interface) without interacting with other classes. In a sense, it is a stand-alone unit of behavior, just as `Flier`. `MechFlier`, on the other hand, is part of a bigger framework and might require complex collaborations to carry out its behavior. The names have been purposely made similar to bring attention to the fact that while they both "do the same thing" they are not similar in terms of their requirements on the software environment.

COMPOSITION OF PERSONALITIES

We have explored composition of frameworks and how Personalities help ease that process in previous sections. Along the same ideas we could think of composing just the personalities themselves. It would be useful to take two complementary personalities and generate a third one. For example, if we had our `Flier` personality and we purchase an `Acrobatic` personality, we would be interested in composing the two to generate an `AcrobaticFlier` personality that would have the `Flier` behavior but with an `Acrobatic` "flavor".

An application would use such a composed personality by providing an implementation for a subset of the union of the downstream interfaces of the original personalities. However, we still need glue code to make it all work together. The `Flier` personality has very specific downstream interface requirements. `Acrobatic`, by the same token, might require different methods from the personifying class. We believe that in order for the composition to succeed we will need a third entity (possibly a class) that would personify both personalities being composed, provide a generic implementation of one using the other, and generate a new personality for the composed behavior requiring the minimum amount of behavior from the end-user application as possible.

In our hypothetical example, this means that this “composer” class will need to personify `Flier` and make use of `Acrobatic`’s upstream interface to make sure that all the behavior of a flier can be carried out doing rolls, loops, free-falls, etc. This is not a simple problem because right away we notice an asymmetry in the composition, since one of the personalities will “dominate” the behavior of the composed one. In this particular case, the `Flier` personality dominates since `Acrobatic` is merely an “adjective” to `Flier`’s behavior. Ways for specifying these dependencies as well as for automatically generating the composition code remain a topic of current research.

Chapter 7

WHERE HAVE WE SEEN THIS BEFORE?

Role modeling in the object-oriented world has traditionally been an area of active research. Therefore, there are many works that have some similarities with the ideas presented in this thesis. We explore the different approaches and study how they differ from Personalities.

USING DELEGATION

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* [LaLonde86] and the work by Gottlob et al. on extending object-oriented systems with roles [Gottlob96]. Both approaches support two kinds of hierarchies: class and role hierarchies (called exemplars in [LaLonde86]). 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.

RELAXING INHERITANCE

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* [Bracha90, Bracha92], *contracts* [Holland93], *mixin-methods*

[Lucas94], *MixedJava* [Flatt98], *Rondo* [Mezini98], and *context relationship* [Seiter98]. These works share the fact that variations on a base behavior are modeled in stand alone artifacts called mixins in [Bracha90, Bracha92] and [Flatt98], contracts in [Holland93], mixin-methods in [Lucas94], adjustments in [Mezini98], and context objects in [Seiter98]. 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 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 fully dynamic Personalities.

THE VISITOR PATTERN

The work presented in [Krishnamurthi98] also considers the need for synthesizing object-oriented and functional decompositions. The visitor pattern [Gamma94] 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.

SUBJECT-ORIENTED PROGRAMMING

The work on *subject-oriented programming* [Harrison93] 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.

ADAPTIVE PROGRAMMING

In Karl Lieberherr's work, behavior is described by propagation patterns (in Demeter/C++ [Lieberherr96]) or adaptive methods (in Demeter/Java [Lieberherr97]), 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 enforce the laws of personalities as described in this paper.

THE RAPIDE CONNECTION

Our concept of upstream and downstream interfaces is very similar in spirit to that of *provided* and *required* interfaces in [Luckham95]. 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 [Luckham95] aims at defining an architecture whereas personalities work at a much smaller (class) granularity.

Chapter 7

CONCLUDING REMARKS

WHAT WE SAID WE WERE GOING TO SAY

The original motivation for this work was very pragmatic. After using Java for a while to build industrial-strength software applications we started to dislike the fact that some of our tried-and-true programming practices (e.g. souped-up template-method pattern [Gamma95]) were not easily expressed because of Java's lack of multiple inheritance support coupled with the constraints imposed on interfaces. We thought then about a simple extension to the language to simply "mimic" multiple inheritance. Little did we know that we were going to get into roles, frameworks, and all kinds of other neat stuff.

WHAT WE ACTUALLY SAID

We have presented a new concept, Personalities, that serves to encapsulate what traditionally has been called a Role. Personalities present several benefits to software engineers, including:

- Mimicking multiple-inheritance for behavior encapsulation
- Implementation of the template-method pattern [Gamma95], with substantial semantic additions to make certain that the behavior remains encapsulated.
- Grouping of a set of upstream methods (a.k.a template-methods) into one cohesive collection to give it a specified semantics and identity in the software developer's arsenal.
- Automatic support for object-migration when using dynamic personalities.

Finally, we contrasted Personalities with the large amount of previous works in this area, explaining the differences between those approaches and this one.

WHAT GOOD IT DID US

We consider personalities merely just an evolutionary step towards better support for role modeling in programming languages. A simple extension to the language yielded a number of interesting possibilities. We conscientiously decided to focus on the theoretical and/or practical aspects of the Personalities idea rather than in the implementation details. However, the Personalities implementation in terms of the Java programming language turned out to be a good example of the use of Design Patterns.

As expected, Personalities play well with Frameworks. Even though it would seem obvious, since Personalities are supposed to be embodiments of “micro-frameworks”, the realization that Personalities can readily replace hotspots and bring something to the mix was a welcomed one. The Personalities concept was very well received at a workshop about pragmatic issues in Framework technology at OOPSLA earlier this year, which provided us with a lot of encouragement.

The contributions of this work are manifold and are detailed earlier in this chapter. In closing, however, we would like to point out that we believe Personalities to be just the tip of the iceberg in the arena of role-modeling in object-oriented design. We have easily found a lot of synergy between personalities and other approaches which leads us to believe that the research community as a whole might be converging towards a more sensible approach to object-oriented design. We hope that Personalities have added some clarity at the micro-design level.

BIBLIOGRAPHY

[Andersen92] Egil Andersen, Trygve Reenskaug. System Design by Composing Structures of Interacting Objects. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP) 1992*. Springer-Verlag, pp. 131-152.

[Arnold97] Ken Arnold, James Gosling. The Java Programming Language, Second Edition. Addison-Wesley, December 1997.

[Bellin97] David Bellin, Susan Suchman Simone. The CRC Card Book. Addison-Wesley, 1997.

[Blando98] Luis Blando, Karl Lieberherr, Mira Mezini. Modeling Behavior with Personalities. *Technical Report: NU-CCS-98-08*, *Northeastern University*, August 1998.

[Booch94] Grady Booch. Object-Oriented Analysis and Design with Applications. The Benjamin/Cummings Publishing Company, Inc., 1994.

[Bracha90] Gilad Bracha, William Cook. Mixin-based Inheritance. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) 1990*.

[Bracha92] Gilad Bracha, Gary Lindstrom. Modularity meets Inheritance. In *Proceedings of IEEE Computer Society International Conference on Computer Languages* (Washington, DC, April 1992). IEEE Computer Society, pp. 282-290.

[Flatt98] Matthew Flatt, Shriram Krishnamurthi, Matthias Felleisen. Classes and Mixins. In *Proceedings of the 1998 Principles of Programming Languages (POPL) Conference.* San Diego, CA, January 1998.

[Gamma94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Software. Addison-Wesley, 1994.

[Gottlob96] Georg Gottlob, Michael Schrefl, Brigitte Roeck. Extending Object-Oriented Systems with Roles. In *ACM Transactions on Information Systems,* Vol. 14, No. 3, July 1996.

[Harrison93] William Harrison, Harold Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) 1993*.

[Holland93] Ian Holland. The Design and Representation of Object-Oriented Components. *PhD Thesis.* Northeastern University, 1993.

[Jacobson92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Övergaard. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison Wesley, 1992.

[Johnson97] Ralph Johnson. Frameworks = (Components + Patterns). In *Communications of the ACM*, Vol. 40, No. 10. October 1997.

[Krishnamurthi98] Shriram Krishnamurthi, Matthias Felleisen, Daniel Friedman. Synthesizing Object-Oriented and Functional Design to Promote Reuse. In *Proceedings of ECOOP '98*. Lecture Notes on Computer Science, Springer Verlag, 1998.

[LaLonde86] Wilf R. LaLonde, Dave Thomas, John Pugh. An Exemplar-Based Smalltalk. In *Proceedings of OOPSLA '86*. ACM Sigplan Notices, Vol. 21, No. 11, pp. 322-330.

[Lieberherr96] Karl Lieberherr. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, Boston, 1996.

[Lieberherr97] Karl Lieberherr, Doug Orleans. Preventive Program Maintenance in Demeter/Java (Research Demonstration). In *Proceedings of ICSE '97*. ACM Press, pp. 604-065. 1997.

[Lucas94] Carine Lucas, Patrick Steyaert. Modular Inheritance of Objects Through Mixin-Methods. In *Proceedings of the 1994 Joint Modular Languages Conference (JMLC).* Springer-Verlag, pp. 273-282.

[Luckham95] David Luckham, James Vera, Sigurd Meldal. Three Concepts of System Architecture. Stanford University Technical Report, CSL-TR-95-674, July 1995.

[Meyer88] Bertrand Meyer. Object-oriented Software Construction. Prentice-Hall International Series in Computer Science, 1988.

[Mezini97] Mira Mezini. Variation-Oriented Programming Beyond Classes and Inheritance. *PhD Thesis, University of Siegen*, 1997.

[Mezini98] Mira Mezini, Karl Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *Proceedings of ACM Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 1998*.

[OMG92] Object Management Architecture Guide. OMG Document 92.11.1, Object Management Group, 1992.

[Ovlinger98] Johan Ovlinger, Karl Lieberherr. Class Graph Views. *Technical Report: NU-CCS-98-07*, *Northeastern University*, August 1998.

[Rumbaugh91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen. Object-Oriented Modeling and Design. Prentice-Hall, 1991.

[Seiter98] Linda Seiter, Jeng Palsberg, Karl Lieberherr. Evolution of Object Behavior Using Context Relations. In *IEEE Transactions on Software Engineering.* Vol. 24, No. 1, January 1998, pp. 79-92.

[Wieringa94] Roel Wieringa, Wiebren de Jonge, Paul Spruit. Using Dynamic Classes and Role Classes to Model Object Migration. *Theory and Practice of Object Systems*, Vol 1(1), pp. 61-83, 1995.

[Wirfs90] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener. Designing Object-Oriented Software. Prentice-Hall, 1990.

APPENDIX A – A COMPLETE EXAMPLE

This appendix presents the implementation for the running example we have used in this thesis. The application itself is very simplistic, but does demonstrate the different characteristics of the Personalities concept. We present implementations for both the static and the dynamic versions.

THE STATIC VERSION

THE .PJ FILES FOR THE ANIMAL HIERARCHY

```
// Animal.pj
public class Animal
 personifies Tracer
{
  private String     name;
  private Integer    code;
  void    setName(String n) { name = n;}
  String  getName() { return name; }
  void    setCode(Integer i) { code = i;}
  Integer getCode() { return code; }
  // --- for Tracer
  //  (nothing in its DI)
}
//=====================================
// Mammal.pj
public class Mammal
  extends Animal
{
  public void Nurse() {
    trace( "Mammal.Nurse()" );  }
}
//=====================================
// Oviparous.pj
public class Oviparous
 extends Animal
{
  public void LayEggs() {
    trace( "Oviparous.LayEggs()" );  }
}
//=====================================
// Pelican.pj
public class Pelican
 extends Oviparous
 personifies Flier, Walker
{
  // --- for Flier
  void Takeoff() {
    trace("Pelican.Takeoff():"); }
  void Ascend() {
    trace("Pelican.Ascend()");   }
  boolean ThereYet(int x, int y) {
    trace( "Pelican.ThereYet()" );
    return false;
  }
  void FlapTowards(int x, int y) {
    trace( "Pelican.FlapTowards()" ); }
  void Descend() {
    trace( "Pelican.Descend():" ); }
  void Land() {
    trace( "Pelican.Land()" ); }
  // --- for Walker
  void Prepare() {
    trace( "Pelican.Prepare()" ); }
  int  NumberOfFeet() {
    trace( "Pelican.NumberOfFeet()" );
    return 2;
  }
  void MoveFoot(int feetno) {
    trace( "Pelican.MoveFoot()" ); }
  void Stabilize() {
    trace( "Pelican.Stabilize()" ); }
  void AtEase() {
    trace( "Pelican.AtEase()" ); }
}
//=====================================
// Bat.pj
public class Bat
 extends Mammal
 personifies Flier
{
  // di implementation for Flier -------
  void Takeoff() {
    trace("Bat.Takeoff():"); }
  void Ascend() {
    trace("Bat.Ascend()");   }
  boolean ThereYet(int x, int y) {
    trace( "Bat.ThereYet()" );
    return true;
  }
```

```
  void FlapTowards(int x, int y) {
    trace( "Bat.FlapTowards()" ); }
  void Descend() {
    trace( "Bat.Descend():" );    }
  void Land() {
    trace( "Bat.Land()" );        }
}
//=====================================
// Whale.pj
public class Whale
  extends Mammal
  personifies Swimmer, Jumper
{
  int depth;
  int swam_distance;
  public  Whale() {
     depth = 0;
     swam_distance = 0;
  }
  private void Propel() {
     swam_distance++; }
  // --- for Swimmer
  void JumpInTheWater() {
    trace( "Whale.JumpInTheWater()" ); }
  void Submerge() {
    trace( "Whale.Submerge()");
    depth++;
  }
  void MoveFin() {
    trace( "Whale.MoveFin()");
    Propel();
  }
  void Rise() {
    trace( "Whale.Rise()");
    depth--;
  }
  void JumpOutOfTheWater() {
    trace("Whale.JumpOutOfTheWater()"); }
  // --- for Jumper
  boolean CheckDistance(int x, int y) {
    trace( "Whale.CheckDistance()" );
    return true;
  }
  void SprintTo(int x, int y) {
    trace( "Whale.SprintTo()" ); }
  void LiftOff(int alt) {
    trace( "Whale.LiftOff()" ); }
  void Land() {
    trace( "Whale.Land()" );    }
}
//=====================================
// SeaLion.pj
public class SeaLion
 extends Mammal
 personifies Walker,Jumper,Swimmer
{
  // --- for Walker
  void Prepare() {
    trace( "SeaLion.Prepare()" ); }
  int  NumberOfFeet() {
    trace( "SeaLion.NumberOfFeet()" );
    return 2;
  }
  void MoveFoot(int feetno) {
    trace( "SeaLion.MoveFoot()" ); }
  void Stabilize() {
    trace( "SeaLion.Stabilize()" ); }
  void AtEase() {
    trace( "SeaLion.AtEase()" ); }
  // --- for Swimmer
  void JumpInTheWater() {
    trace("SeaLion.JumpInTheWater()" ); }
  void Submerge() {
    trace( "SeaLion.Submerge()");  }
  void MoveFin() {
    trace( "SeaLion.MoveFin()");  }
  void Rise() {
    trace( "SeaLion.Rise()"); }
  void JumpOutOfTheWater() {
   trace("SeaLion.JumpOutOfTheWater()");}
  // --- for Jumper
  boolean CheckDistance(int x, int y) {
    trace( "SeaLion.CheckDistance()" );
    return true;
  }
  void SprintTo(int x, int y) {
    trace( "SeaLion.SprintTo()" ); }
  void LiftOff(int alt) {
    trace( "SeaLion.LiftOff()" ); }
  void Land() {
    trace( "SeaLion.Land()" ); }
}
```

THE PERSONALITY FILES

```
// Flier.pj
personality Flier {
  // upstream interface. Must implement.
  public void Fly(int x, int y,
                  int altitude) {
    resetMetersFlown();
    Takeoff();
    for (int a=0; a < altitude; a++)
      Ascend();
    while( !ThereYet(x, y) )
      FlapTowards(x, y);
    for(int a = altitude; a > 0; a--)
      Descend();
    Land();
  }
  // downstream interface.
  // Don't implement here.
  di void Takeoff();
  di void Ascend();
  di boolean ThereYet(int x, int y);
  di void FlapTowards(int x, int y);
```

```
  di void Descend();
  di void Land();
  // private functions. Must implement.
  private void resetMetersFlown() {
    meters_flown = 0; }
  // attributes (specific to the role)
  private float meters_flown;
  // constructor (optional)
  Flier() { resetMetersFlown(); }
}
//=====================================
// Swimmer.pj
personality Swimmer
{
  public void Swim(int miles, int depth){
    JumpInTheWater();
    for (int d = 0; d < depth; d++)
      Submerge();
    while ((miles--) > 0) MoveFin();
    for(int d = depth; d > 0; d--)
      Rise();
    JumpOutOfTheWater();
  }
  di void JumpInTheWater();
  di void Submerge();
  di void MoveFin();
  di void Rise();
  di void JumpOutOfTheWater();
}
//=====================================
// Walker.pj
personality Walker
{
```

```
  public void Walk(int distance) {
    Prepare();
    while( (distance--) > 0 ) {
      for(int f=0; f<NumberOfFeet(); f++)
        MoveFoot(f);
      Stabilize();
    }
    AtEase();
  }
  // downstream
  di void Prepare();
  di int NumberOfFeet();
  di void MoveFoot(int feetno);
  di void Stabilize();
  di void AtEase();
}
//=====================================
// Jumper.pj
personality Jumper
{
  void Jump(int x, int y, int alt) {
    if ( CheckDistance(x, y) ) {
      SprintTo( x, y );
      LiftOff( alt );
      Land();
    }
  }
  di boolean CheckDistance(int x, int y);
  di void SprintTo(int x, int y);
  di void LiftOff(int alt);
  di void Land();
}
```

THE CLIENT USING STATIC PERSONALITIES

```
// Zoo.pj - [static]
//  this is a simple client program,
//  unrealistic, but shows the Show()
//  methods only dealing with the
//  appropriate personality (and its UI)
import java.util.*;
public class Zoo
{
  static public void
  main(String args[]) {
    Vector all_swimmers = new Vector();
    Vector all_fliers   = new Vector();
    Vector all_walkers  = new Vector();
    Vector all_jumpers  = new Vector();

    // some animals are born in our zoo..
    // the client or the factory must
    // know what Personalities are
    // attached to each class.
    SeaLion toto   = new SeaLion();
    toto.setName("Toto");
    all_swimmers.addElement( toto );
    all_walkers.addElement( toto );
    all_jumpers.addElement( toto );

    Whale   keiko  = new Whale();
    keiko.setName("Keiko");
    all_swimmers.addElement( keiko );
    all_jumpers.addElement( keiko );
```

```
    Bat     drac   = new Bat();
    drac.setName("Drac");
    all_fliers.addElement( drac );

    // perform the different shows
    for(int i=0; i < all_swimmers.size(); i++)
      PoolShow( (Swimmer)all_swimmers.elementAt(i) );
    for(int i=0; i < all_walkers.size(); i++)
      FieldShow( (Walker)all_walkers.elementAt(i)  );
    for(int i=0; i < all_jumpers.size(); i++)
      JumpShow ( (Jumper)all_jumpers.elementAt(i)  );
    for(int i=0; i < all_fliers.size(); i++)
      SkyShow  ( (Flier)all_fliers.elementAt(i)   );
  }

  static void PoolShow(Swimmer swimmer) {
    System.out.println("  PoolShow with " + swimmer);
    swimmer.Swim( 1, 1 );
  }
  static void FieldShow(Walker walker)  {
    System.out.println("  FieldShow with " + walker);
    walker.Walk( 1 );
  }
  static void SkyShow(Flier flier)      {
    System.out.println("  SkyShow with " + flier);
    flier.Fly( 1, 1, 1 );
  }
  static void JumpShow(Jumper jumper)   {
    System.out.println("  JumpShow with " + jumper);
    jumper.Jump(1, 1, 1);
  }
}
```

THE GENERATED JAVA CODE

Animal Hierarchy

```
// Animal.java [static]
import java.util.*;
public class Animal
 implements  Tracer
{
  private String     name;
  private Integer    code;
  public void setName(String n) {name=n;}
  public String getName() {return name;}
  public void setCode(Integer i){code=i;}
  public Integer getCode(){ return code;}
  // ============= for Tracer
  Tracer$Ego $tracer = new Tracer$Ego( );
  public void trace(String msg) {
    $tracer.trace(this, msg);
  }
}
/=======================================
// Oviparous.java [static]
public class Oviparous
 extends Animal
{
  public void LayEggs() {
    trace( "Oviparous.LayEggs()" );
  }
}
```

```
/=======================================
// Mammal.java [static]
public class Mammal
 extends Animal
{
  public void Nurse() {
    trace( "Mammal.Nurse()" );
  }
}
/=======================================
// Pelican.java [static]
import java.util.*;
public class Pelican
 extends Oviparous
 implements Flier, Walker
{
  // for Flier
  public void Takeoff() {
    trace("Pelican.Takeoff():"); }
  public void Ascend() {
    trace("Pelican.Ascend()");  }
  public boolean ThereYet(int x, int y) {
    trace( "Pelican.ThereYet()" );
    return false;
  }
  public void FlapTowards(int x, int y) {
```

```
    trace( "Pelican.FlapTowards()" ); }
  public void Descend() {
    trace( "Pelican.Descend():" ); }
  public void Land() {
    trace( "Pelican.Land()" ); }
  // for Walker
  public void Prepare() {
    trace( "Pelican.Prepare()" ); }
  public int  NumberOfFeet() {
    trace( "Pelican.NumberOfFeet()" );
    return 2;
  }
  public void MoveFoot(int feetno) {
    trace( "Pelican.MoveFoot()" ); }
  public void Stabilize() {
    trace( "Pelican.Stabilize()" ); }
  public void AtEase() {
    trace( "Pelican.AtEase()" ); }
  // ============== for Flier
  Flier$Ego $flier = new Flier$Ego( );
  public void Fly(int x, int y,
                  int altitude) {
    $flier.Fly(this, x, y, altitude);
  }
  // ============== For Walker
  Walker$Ego $walker = new Walker$Ego( );
  public void Walk(int distance) {
    $walker.Walk(this, distance);
  }
}
/=======================================
// Bat.java [static]
import java.util.*;
public class Bat
 extends Mammal
 implements Flier
{
  // di implementation for Flier -----
  public void Takeoff() {
    trace("Bat.Takeoff():"); }
  public void Ascend() {
    trace("Bat.Ascend()");  }
  public boolean ThereYet(int x, int y) {
    trace( "Bat.ThereYet()" );
    return true;
  }
  public void FlapTowards(int x, int y) {
    trace( "Bat.FlapTowards()" ); }
  public void Descend() {
    trace( "Bat.Descend():" ); }
  public void Land() {
    trace( "Bat.Land()" ); }
  // ============== for Flier
  Flier$Ego $flier = new Flier$Ego();
  public void Fly(int x, int y,
                  int altitude) {
    $flier.Fly(this, x, y, altitude);
  }
}
/=======================================
// Whale.java [static]
import java.util.*;
public class Whale
  extends Mammal
  implements Swimmer, Jumper
{
  int depth;
  int swam_distance;
  public  Whale() {
    depth = 0;
    swam_distance = 0;
  }
  private void Propel() {
    swam_distance++; }
  // --- for Swimmer
  public void JumpInTheWater() {
    trace( "Whale.JumpInTheWater()" ); }
  public void Submerge() {
    trace( "Whale.Submerge()");
    depth++;
  }
  public void MoveFin() {
    trace( "Whale.MoveFin()");
    Propel();
  }
  public void Rise() {
    trace( "Whale.Rise()");
    depth--;
  }
  public void JumpOutOfTheWater() {
    trace( "Whale.JumpOutOfTheWater()");}
  // --- for Jumper
  public boolean CheckDistance(int x,
                               int y) {
    trace( "Whale.CheckDistance()" );
    return true;
  }
  public void SprintTo(int x, int y) {
    trace( "Whale.SprintTo()" ); }
  public void LiftOff(int alt) {
    trace( "Whale.LiftOff()" ); }
  public void Land() {
    trace( "Whale.Land()" ); }
  // ============== for Swimmer
  Swimmer$Ego $swimmer=new Swimmer$Ego();
  public void Swim(int miles, int depth){
    $swimmer.Swim(this, miles, depth);
  }
  // ============== for Jumper
  Jumper$Ego $jumper = new Jumper$Ego();
  public void Jump(int x, int y, int alt)
  {
    $jumper.Jump(this, x, y, alt);
  }
}
/=======================================
// SeaLion.java [static]
import java.util.*;
public class SeaLion
 extends Mammal
 implements Walker,Jumper,Swimmer
{
  // for Walker
  public void Prepare() {
    trace( "SeaLion.Prepare()" ); }
  public int  NumberOfFeet() {
    trace( "SeaLion.NumberOfFeet()" );
    return 2;
  }
  public void MoveFoot(int feetno) {
    trace( "SeaLion.MoveFoot()" ); }
  public void Stabilize() {
```

```
    trace( "SeaLion.Stabilize()" ); }
  public void AtEase() {
    trace( "SeaLion.AtEase()" ); }
  // for Swimmer
  public void JumpInTheWater() {
    trace( "SeaLion.JumpInTheWater()" );}
  public void Submerge() {
    trace( "SeaLion.Submerge()");  }
  public void MoveFin() {
    trace( "SeaLion.MoveFin()");  }
  public void Rise() {
    trace( "SeaLion.Rise()"); }
  public void JumpOutOfTheWater() {
   trace("SeaLion.JumpOutOfTheWater()");}
  // for Jumper
  public boolean CheckDistance(int x,
                               int y) {
    trace( "SeaLion.CheckDistance()" );
    return true;
  }
  public void SprintTo(int x, int y) {
    trace( "SeaLion.SprintTo()" ); }
  public void LiftOff(int alt) {
    trace( "SeaLion.LiftOff()" ); }
  public void Land() {
    trace( "SeaLion.Land()" ); }
  // ============== for Swimmer
  Swimmer$Ego $swimmer=new Swimmer$Ego();
  public void Swim(int miles, int depth)
  {
    $swimmer.Swim(this, miles, depth);
  }
  // ============== for Walker
  Walker$Ego $walker = new Walker$Ego();
  public void Walk(int distance) {
    $walker.Walk(this, distance);
  }
  // ============== for Jumper
  Jumper$Ego $jumper = new Jumper$Ego();
  public void Jump(int x, int y, int alt)
  {
    $jumper.Jump(this, x, y, alt);
  }
}
```

Personalities

```
// Flier.java [static]
interface Flier {
  public void Fly(int x, int y,
                  int altitude);
  void Takeoff();
  void Ascend();
  boolean ThereYet(int x, int y);
  void FlapTowards(int x, int y);
  void Descend();
  void Land();
}
/=======================================
// Flier$Ego.java [static]
public class Flier$Ego
{
  // upstream interface. Must implement.
  public void Fly(Flier host, int x,
                  int y, int altitude) {
    resetMetersFlown();
    host.Takeoff();
    for (int a=0; a < altitude; a++)
      host.Ascend();
    while( !host.ThereYet(x, y) )
      host.FlapTowards(x, y);
    for(int a = altitude; a > 0; a--)
      host.Descend();
    host.Land();
  }
  // private functions. Must implement.
  private void resetMetersFlown() {
    meters_flown = 0;
  }
  // attributes (specific to the role)
  private float meters_flown;
  // constructor (optional)
  Flier$Ego( ) {
    resetMetersFlown();
  }
}
/=======================================
// Swimmer.java [static]
interface Swimmer
{
  public void Swim(int miles, int depth);
  void JumpInTheWater();
  void Submerge();
  void MoveFin();
  void Rise();
  void JumpOutOfTheWater();
}
/=======================================
// Swimmer$Ego.java [static]
public class  Swimmer$Ego
{
  public void Swim(Swimmer host,
                   int miles, int depth){
    host.JumpInTheWater();
    for (int d = 0; d < depth; d++)
      host.Submerge();
    while ((miles--) > 0) host.MoveFin();
    for(int d = depth; d > 0; d--)
      host.Rise();
    host.JumpOutOfTheWater();
  }
  public Swimmer$Ego()
  {  }
}
/=======================================
// Walker.java [static]
interface Walker
{
  public void Walk(int distance);
  void Prepare();
  int NumberOfFeet();
  void MoveFoot(int feetno);
  void Stabilize();
```

```
  void AtEase();
}
/=======================================
// Walker$Ego.java [static]
public class  Walker$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();
  }
  public Walker$Ego()
  { }
}
/=======================================
// Jumper.java [static]
interface Jumper
{
  void Jump(int x, int y, int alt);
  boolean CheckDistance(int x, int y);
  void SprintTo(int x, int y);
  void LiftOff(int alt);
  void Land();
}
/=======================================
// Jumper$Ego.java [static]
public class Jumper$Ego
{
  void Jump(Jumper host, int x, int y,
            int alt) {
    if ( host.CheckDistance(x, y) ) {
      host.SprintTo( x, y );
      host.LiftOff( alt );
      host.Land();
    }
  }
  public Jumper$Ego()
  {  }
}
```

Client Code

```
// Zoo.java - [static]
//  this is a simple client program, unrealistic,
//  but shows the Show() methods only dealing with
//  the appropriate personality (and its UI)
import java.util.*;
public class Zoo
{
  static public void main(String args[]) {
    Vector all_swimmers = new Vector();
    Vector all_fliers   = new Vector();
    Vector all_walkers  = new Vector();
    Vector all_jumpers  = new Vector();

    // some animals are born in our zoo...
    // notice that the client or the factory
    // must know what Personalities are attached
    // to each class.
    SeaLion toto   = new SeaLion();
    toto.setName("Toto");
    all_swimmers.addElement( toto );
    all_walkers.addElement( toto );
    all_jumpers.addElement( toto );

    Whale   keiko  = new Whale();
    keiko.setName("Keiko");
    all_swimmers.addElement( keiko );
    all_jumpers.addElement( keiko );

    Bat     drac   = new Bat();
    drac.setName("Drac");
    all_fliers.addElement( drac );

    // perform the different shows
    for(int i=0; i < all_swimmers.size(); i++)
      PoolShow ( (Swimmer)all_swimmers.elementAt(i));
    for(int i=0; i < all_walkers.size(); i++)
      FieldShow( (Walker)all_walkers.elementAt(i)  );
    for(int i=0; i < all_jumpers.size(); i++)
```

```
      JumpShow ( (Jumper)all_jumpers.elementAt(i)  );
    for(int i=0; i < all_fliers.size(); i++)
      SkyShow  ( (Flier)all_fliers.elementAt(i)   );
}

  static void PoolShow(Swimmer swimmer) {
    System.out.println("  PoolShow with " + swimmer);
    swimmer.Swim( 1, 1 );
  }
  static void FieldShow(Walker walker)  {
    System.out.println("  FieldShow with " + walker);
    walker.Walk( 1 );
  }
  static void SkyShow(Flier flier)      {
    System.out.println("  SkyShow with " + flier);
    flier.Fly( 1, 1, 1 );
  }
  static void JumpShow(Jumper jumper)   {
    System.out.println("  JumpShow with " + jumper);
    jumper.Jump(1, 1, 1);
  }
}
```

THE DYNAMIC VERSION

There are no changes needed in either the Animal or the Personalities hierarchy for the dynamic version. The clients of the personalities, however, do need to change as well as the generated Java code.

THE CLIENT USING DYNAMIC PERSONALITIES

```
// Zoo.pj [dynamic]
//  this is a simple client program, unrealistic,
//  but shows the Show() methods only dealing with
//  the appropriate personality (and its UI)
//  This is the dynamic version of the client, it
//  also does slightly different things.
public class Zoo
{
  static public void main(String args[]) {
    // some animals are born in our zoo...
    SeaLion toto   = new SeaLion();
    toto.setName("Toto");
    Whale   keiko  = new Whale();
    keiko.setName("Keiko");
    Bat     drac   = new Bat();
    drac.setName("Drac");
    System.out.println("TIME 0 - Zoo is created");

    toto.personify( "Tracer" );
    keiko.personify( "Tracer" );
    drac.personify( "Tracer" );

    pp(toto); pp(keiko); pp(drac);

    // toto and keiko can swim the moment
```

```
    // they are born. drac, however, can't fly
    keiko.personify( "Swimmer" );
    toto.personify( "Swimmer" );

    System.out.println("TIME 1 - animals created,
                        Whale and SeaLion swim");
    pp(toto); pp(keiko); pp(drac);

    // they've been just born, but the
    // show must go on anyways...
    PerformShow( toto  );
    PerformShow( keiko );
    PerformShow( drac  );

    // let's pretend time goes on. toto and
    // keiko are getting strong enough to jump
    // also, drac can now fly
    toto.personify( "Jumper" );
    keiko.personify( "Jumper" );
    drac.personify( "Flier" );

    System.out.println("TIME 2 - Whale and SeaLion
                        jump, Bat flies");
    pp(toto); pp(keiko); pp(drac);
    PerformShow( toto );
    PerformShow( keiko );
    PerformShow( drac );

    // finally, toto can also walk...
    toto.personify( "Walker" );

    System.out.println("TIME 3 - SeaLion walks");
    pp(toto);
    PerformShow( toto );

    // time goes on, and toto and keiko get old, not
    // strong enough to walk or jump anymore
    toto.forget( "Walker" );
    toto.forget( "Jumper" );
    keiko.forget( "Jumper" );
    System.out.println("TIME 4 - Whale and SeaLion
                        forget to walk and jump");
    pp(toto); pp(keiko); pp(drac);
    PerformShow( toto );
    PerformShow( keiko );
    PerformShow(drac);
  }

  // master of ceremonies...
  static void PerformShow(Animal animal) {
    System.out.println(" PerformShow with " +
                        animal);
    if ( animal.personifies( "Swimmer" ) )
      PoolShow( (Swimmer)animal );
    if ( animal.personifies( "Walker" ) )
      FieldShow( (Walker)animal );
    if ( animal.personifies( "Flier" ) )
      SkyShow( (Flier)animal );
    if ( animal.personifies( "Jumper" ) )
      JumpShow( (Jumper)animal );
  }
  static void PoolShow(Swimmer swimmer) {
    System.out.println("  PoolShow with " + swimmer);
    swimmer.Swim( 1, 1 );
  }
  static void FieldShow(Walker walker)  {
```

```
    System.out.println("  FieldShow with " + walker);
    walker.Walk( 1 );
  }
  static void SkyShow(Flier flier)      {
    System.out.println("  SkyShow with " + flier);
    flier.Fly( 1, 1, 1 );
  }
  static void JumpShow(Jumper jumper)   {
    System.out.println("  JumpShow with " + jumper);
    jumper.Jump(1, 1, 1);
  }
  // auxiliary function
  static void pp(Animal a) {
    System.out.println( a.getName() + ": "
                        + a.personalities() );
  }
}
```

THE GENERATED JAVA CODE

Animal Hierarchy

```
// Animal.java [dynamic]
import java.util.*;
public class Animal
 implements  Tracer
{
  private String     name;
  private Integer    code;
  public void setName(String n){ name=n;}
  public String getName() { return name;}
  public void setCode(Integer i){code=i;}
  public Integer getCode(){ return code;}
  // == SHRINK INSERTION -=-=-=-=-=-=-=-
  // == generic for all classes that
  //    "personify". The Personalities/J
  //    compiler must insert these
  //    definitions only once at the
  //    class within the hierarchy that
  //    is closest to the root, whose
  //    ancestors do NOT personify
  //    anything, but it personifies
  //    something.
  //
  //    In this example, the class that
  //    fulfills these requirements is
  //    the Animal class.
  protected Shrink $shrink= new Shrink();
  public boolean personify(String what)
  { return $shrink.personify(what); }
  public boolean personifies(String what)
  { return $shrink.personifies(what); }
  public boolean forget(String what)
  { return $shrink.forget(what); }
  public boolean canpersonify(String
                                  what)
  { return $shrink.canpersonify(what); }
  public Vector  personalities()
  { return $shrink.personalities(); }
  // == SHRINK INSERTION ENDS =-=-=-=-=-=
  // for Tracer
  Tracer$Ego $tracer=
                  new Tracer$Ego($shrink);
  public void trace(String msg) {
    if (personifies( "Tracer" ))
      $tracer.trace(this, msg);
  }
}
//=====================================
// Oviparous.java [dynamic]
public class Oviparous
 extends Animal
{
  public void LayEggs() {
    trace( "Oviparous.LayEggs()" );
  }
}
//=====================================
// Mammal.java [dynamic]
public class Mammal
 extends Animal
{
  public void Nurse() {
    trace( "Mammal.Nurse()" );
  }
}
//=====================================
// Pelican.java [dynamic]
import java.util.*;
public class Pelican
 extends Oviparous
 implements Flier, Walker
{
  // for Flier
  public void Takeoff() {
    trace("Pelican.Takeoff():"); }
  public void Ascend() {
    trace("Pelican.Ascend()");  }
  public boolean ThereYet(int x, int y) {
    trace( "Pelican.ThereYet()" );
    return false;
  }
  public void FlapTowards(int x, int y) {
```

```
    trace( "Pelican.FlapTowards()" ); }
  public void Descend() {
    trace( "Pelican.Descend():" ); }
  public void Land() {
    trace( "Pelican.Land()" ); }
  // for Walker
  public void Prepare() {
    trace( "Pelican.Prepare()" ); }
  public int  NumberOfFeet() {
    trace( "Pelican.NumberOfFeet()" );
    return 2;
  }
  public void MoveFoot(int feetno) {
    trace( "Pelican.MoveFoot()" ); }
  public void Stabilize() {
    trace( "Pelican.Stabilize()" ); }
  public void AtEase() {
    trace( "Pelican.AtEase()" ); }
  // ============== for Flier
  Flier$Ego $flier
              = new Flier$Ego( $shrink );
  public void Fly(int x, int y,
                  int altitude) {
    if ( personifies( "Flier" ) )
      $flier.Fly(this, x, y, altitude);
  }
  // ============== For Walker
  Walker$Ego $walker
             = new Walker$Ego( $shrink );
  public void Walk(int distance) {
    if ( personifies( "Walker" ) )
      $walker.Walk(this, distance);
  }
}
//=======================================
// Bat.java [dynamic]
import java.util.*;
public class Bat
 extends Mammal
 implements Flier
{
  // for Flier
  public void Takeoff() {
    trace("Bat.Takeoff():"); }
  public void Ascend() {
    trace("Bat.Ascend()");  }
  public boolean ThereYet(int x, int y) {
    trace( "Bat.ThereYet()" );
    return true;
  }
  public void FlapTowards(int x, int y) {
    trace( "Bat.FlapTowards()" ); }
  public void Descend() {
    trace( "Bat.Descend():" ); }
  public void Land() {
    trace( "Bat.Land()" ); }
  // ============== for Flier
  Flier$Ego $flier
              = new Flier$Ego( $shrink );
  public void Fly(int x, int y,
                  int altitude) {
    if ( personifies( "Flier" ) )
      $flier.Fly(this, x, y, altitude);
  }
}
```

```
//=======================================
// Whale.java [dynamic]
import java.util.*;
public class Whale
  extends Mammal
  implements Swimmer, Jumper
{
  int depth;
  int swam_distance;
  public  Whale() {
    depth = 0;
    swam_distance = 0;
  }
  private void Propel() {
    swam_distance++; }
  // --- for Swimmer
  public void JumpInTheWater() {
    trace( "Whale.JumpInTheWater()" ); }
  public void Submerge() {
    trace( "Whale.Submerge()");
    depth++;
  }
  public void MoveFin() {
    trace( "Whale.MoveFin()");
    Propel();
  }
  public void Rise() {
    trace( "Whale.Rise()");
    depth--;
  }
  public void JumpOutOfTheWater() {
    trace( "Whale.JumpOutOfTheWater()");}
  // --- for Jumper
  public boolean CheckDistance(int x,
                               int y) {
    trace( "Whale.CheckDistance()" );
    return true;
  }
  public void SprintTo(int x, int y) {
    trace( "Whale.SprintTo()" ); }
  public void LiftOff(int alt) {
    trace( "Whale.LiftOff()" ); }
  public void Land() {
    trace( "Whale.Land()" ); }
  // ============== for Swimmer
  Swimmer$Ego $swimmer
            = new Swimmer$Ego( $shrink );
  public void Swim(int miles, int depth){
    if ( personifies( "Swimmer" ) )
      $swimmer.Swim(this, miles, depth);
  }
  // ============== for Jumper
  Jumper$Ego $jumper
             = new Jumper$Ego( $shrink );
  public void Jump(int x, int y, int alt)
  {
    if ( personifies( "Jumper" ) )
      $jumper.Jump(this, x, y, alt);
  }
}
//=======================================
// SeaLion.java [dynamic]
import java.util.*;
public class SeaLion
 extends Mammal
 implements Walker,Jumper,Swimmer
```

```
{
  // for Walker
  public void Prepare() {
    trace( "SeaLion.Prepare()" ); }
  public int  NumberOfFeet() {
    trace( "SeaLion.NumberOfFeet()" );
    return 2;
  }
  public void MoveFoot(int feetno) {
    trace( "SeaLion.MoveFoot()" ); }
  public void Stabilize() {
    trace( "SeaLion.Stabilize()" ); }
  public void AtEase() {
    trace( "SeaLion.AtEase()" ); }
  // for Swimmer
  public void JumpInTheWater() {
    trace( "SeaLion.JumpInTheWater()" );}
  public void Submerge() {
    trace( "SeaLion.Submerge()");  }
  public void MoveFin() {
    trace( "SeaLion.MoveFin()");  }
  public void Rise() {
    trace( "SeaLion.Rise()"); }
  public void JumpOutOfTheWater() {
   trace("SeaLion.JumpOutOfTheWater()");}
  // for Jumper
  public boolean CheckDistance(int x,
                               int y) {
    trace( "SeaLion.CheckDistance()" );
    return true;
  }
  public void SprintTo(int x, int y) {
    trace( "SeaLion.SprintTo()" ); }
  public void LiftOff(int alt) {
    trace( "SeaLion.LiftOff()" ); }
  public void Land() {
    trace( "SeaLion.Land()" ); }
  // ============== for Swimmer
  Swimmer$Ego $swimmer
            = new Swimmer$Ego( $shrink );
  public void Swim(int miles, int depth){
    if ( personifies( "Swimmer" ) )
      $swimmer.Swim(this, miles, depth);
  }
  // ============== for Walker
  Walker$Ego $walker
             = new Walker$Ego( $shrink );
  public void Walk(int distance) {
    if ( personifies( "Walker" ) )
      $walker.Walk(this, distance);
  }
  // ============== for Jumper
  Jumper$Ego $jumper
             = new Jumper$Ego( $shrink );
  public void Jump(int x, int y, int alt)
  {
    if ( personifies( "Jumper" ) )
      $jumper.Jump(this, x, y, alt);
  }
}
```

Personalities

```
// Shrink.java [dynamic]
//  this class is only generated when the
//  Personalities/J compiler is
//  generating dynamic personality code.
//  It is a fixed implementation of a
//  helper class used to maintain the
//  personality list for the classes
//  within a particular inheritance
//  hierarchy.
import java.util.*;
public class Shrink
{
  Vector names  = new Vector();
  Vector states = new Vector();

  public void
  register_personality(String p) {
    if ( ! names.contains(p) ) {
      names.addElement( p );
      states.addElement( Boolean.FALSE );
    }
  }
  public boolean
  personify(String p) {
    if ( !canpersonify(p) ) return false;
    int idx = names.indexOf(p);
    states.setElementAt(Boolean.TRUE,
                        idx);
    return true;
  }
  public boolean
  forget(String p) {
    if ( !canpersonify(p) ) return false;
    int idx = names.indexOf(p);
    states.setElementAt(Boolean.FALSE,
                        idx);
    return true;
  }
  public boolean personifies(String p) {
    if ( names.contains(p) ) {
      int idx = names.indexOf(p);
      return ((Boolean)
               states.elementAt(idx))
               .booleanValue();
    }
    else return false;
  }
  public boolean canpersonify(String p) {
    return names.contains(p);
  }
  public Vector personalities() {
    Vector result = new Vector();
    for (int i=0; i < names.size(); i++)
      if ( ((Boolean)
             states.elementAt(i))
             .booleanValue() )
        result.addElement(names
                          .elementAt(i));
    return result;
```

```
  }
}
//=======================================
// Flier.java [dynamic]
interface Flier {
  public void Fly(int x, int y,
                  int altitude);
  void Takeoff();
  void Ascend();
  boolean ThereYet(int x, int y);
  void FlapTowards(int x, int y);
  void Descend();
  void Land();
}
//=======================================
// Flier$Ego.java [dynamic]
public class Flier$Ego
{
  // upstream interface. Must implement.
  public void Fly(Flier host, int x,
                  int y, int altitude) {
    resetMetersFlown();
    host.Takeoff();
    for (int a=0; a < altitude; a++)
      host.Ascend();
    while( !host.ThereYet(x, y) )
      host.FlapTowards(x, y);
    for(int a = altitude; a > 0; a--)
      host.Descend();
    host.Land();
  }
  // private functions. Must implement.
  private void resetMetersFlown() {
    meters_flown = 0; }
  // attributes (specific to the role)
  private float meters_flown;
  // constructor (optional)
  Flier$Ego( Shrink shrink ) {
    resetMetersFlown();
    shrink.register_personality("Flier");
  }
}
//=======================================
// Swimmer.java [dynamic]
interface Swimmer
{
  public void Swim(int miles, int depth);
  void JumpInTheWater();
  void Submerge();
  void MoveFin();
  void Rise();
  void JumpOutOfTheWater();
}
//=======================================
// Swimmer$Ego.java [dynamic]
public class  Swimmer$Ego
{
  public void Swim(Swimmer host,
                   int miles, int depth){
    host.JumpInTheWater();
    for (int d = 0; d < depth; d++)
      host.Submerge();
    while ((miles--) > 0) host.MoveFin();
    for(int d = depth; d > 0; d--)
      host.Rise();
    host.JumpOutOfTheWater();
  }
  public Swimmer$Ego(Shrink shrink)
  { shrink
    .register_personality( "Swimmer" ); }
}
//=======================================
// Walker.java [dynamic]
interface Walker
{
  public void Walk(int distance);
  void Prepare();
  int NumberOfFeet();
  void MoveFoot(int feetno);
  void Stabilize();
  void AtEase();
}
//=======================================
// Walker$Ego.java [dynamic]
public class  Walker$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();
  }
  public Walker$Ego(Shrink shrink) {
    shrink
    .register_personality( "Walker" ); }
}
//=======================================
// Jumper.java [dynamic]
interface Jumper
{
  void Jump(int x, int y, int alt);
  boolean CheckDistance(int x, int y);
  void SprintTo(int x, int y);
  void LiftOff(int alt);
  void Land();
}
//=======================================
// Jumper$Ego.java [dynamic]
public class Jumper$Ego
{
  void Jump(Jumper host, int x,
            int y, int alt) {
    if ( host.CheckDistance(x, y) ) {
      host.SprintTo( x, y );
      host.LiftOff( alt );
      host.Land();
    }
  }
  public Jumper$Ego(Shrink shrink)
  { shrink
    .register_personality( "Jumper" ); }
}
```

Client Code

```
// Zoo.java [dynamic]
//  this is a simple client program, unrealistic,
//  but shows the Show() methods only dealing with
//  the appropriate personality (and its UI)
//  This is the dynamic version of the client, it
//  also does slightly different things.
public class Zoo
{
  static public void main(String args[]) {
    // some animals are born in our zoo...
    SeaLion toto   = new SeaLion();
    toto.setName("Toto");
    Whale   keiko  = new Whale();
    keiko.setName("Keiko");
    Bat     drac   = new Bat();
    drac.setName("Drac");
    System.out.println("TIME 0 - Zoo is created");

    toto.personify( "Tracer" );
    keiko.personify( "Tracer" );
    drac.personify( "Tracer" );

    pp(toto); pp(keiko); pp(drac);

    // toto and keiko can swim the moment
    // they are born. drac, however, can't fly
    keiko.personify( "Swimmer" );
    toto.personify( "Swimmer" );

    System.out.println("TIME 1 - animals created,
                        Whale and SeaLion swim");
    pp(toto); pp(keiko); pp(drac);

    // they've been just born, but the
    // show must go on anyways...
    PerformShow( toto  );
    PerformShow( keiko );
    PerformShow( drac  );

    // let's pretend time goes on. toto and
    // keiko are getting strong enough to jump
    // also, drac can now fly
    toto.personify( "Jumper" );
    keiko.personify( "Jumper" );
    drac.personify( "Flier" );

    System.out.println("TIME 2 - Whale and SeaLion
                        jump, Bat flies");
    pp(toto); pp(keiko); pp(drac);
    PerformShow( toto );
    PerformShow( keiko );
    PerformShow( drac );

    // finally, toto can also walk...
    toto.personify( "Walker" );

    System.out.println("TIME 3 - SeaLion walks");
    pp(toto);
    PerformShow( toto );

    // time goes on, and toto and keiko get old, not
    // strong enough to walk or jump anymore
    toto.forget( "Walker" );
```

```
    toto.forget( "Jumper" );
    keiko.forget( "Jumper" );
    System.out.println("TIME 4 - Whale and SeaLion
                        forget to walk and jump");
    pp(toto); pp(keiko); pp(drac);
    PerformShow( toto );
    PerformShow( keiko );
    PerformShow(drac);
  }

  // master of ceremonies...
  static void PerformShow(Animal animal) {
    System.out.println(" PerformShow with " +
                        animal);
    if ( animal.personifies( "Swimmer" ) )
      PoolShow( (Swimmer)animal );
    if ( animal.personifies( "Walker" ) )
      FieldShow( (Walker)animal );
    if ( animal.personifies( "Flier" ) )
      SkyShow( (Flier)animal );
    if ( animal.personifies( "Jumper" ) )
      JumpShow( (Jumper)animal );
  }
  static void PoolShow(Swimmer swimmer) {
    System.out.println("  PoolShow with " + swimmer);
    swimmer.Swim( 1, 1 );
  }
  static void FieldShow(Walker walker)  {
    System.out.println("  FieldShow with " + walker);
    walker.Walk( 1 );
  }
  static void SkyShow(Flier flier)      {
    System.out.println("  SkyShow with " + flier);
    flier.Fly( 1, 1, 1 );
  }
  static void JumpShow(Jumper jumper)   {
    System.out.println("  JumpShow with " + jumper);
    jumper.Jump(1, 1, 1);
  }
  // auxiliary function
  static void pp(Animal a) {
    System.out.println( a.getName() + ": " +
                        a.personalities() );
  }
}
```

[i] On Figure 9, the following "implements" relationships are not shown to keep the diagram uncluttered: *`SeaLion implements Walker, Jumper`*; *`Whale implements Swimmer`*; and *`Pelican implements Flier`*.