Designing and Programming with Personalities

Outline

- Modeling behavior
- Personalities
- Dynamic Personalities
- Synergy with Frameworks
- Personalities/J language
- Q&A
Modeling Behavior - Outline

- The functional nature of software systems
- Functional decomposition and Structured Programming
- Data Decomposition and Object Oriented Programming
- Finding objects and behavior
- When behavior misbehaves
- Popular functions and their mapping

The functional nature of software systems

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.  

Bertrand Meyer, Object Oriented Software Construction, 1988
When Behavior Misbehaves - 1

When Behavior Misbehaves - 2

The intrinsic stuff is easy to do! :-)
Popular Functions

The popular functions in ZooSys

Fly
- Takeoff
- Ascend
- ThereYet
- FlapTowards
- Descent
- Land

Swim
- Prepare
- MoveFoot
- Stabilize
- AtEase

Walk
- CheckDistance
- SprintTo
- LiftOff
- Land

Jump
- JumpInTheWater
- Submerge
- MoveFin
- Rise
- JumpOutOfTheWater

Fly() popular function

Encapsulate this sequence. Always the same for any class...

JumpInTheWater
Submerge
MoveFin
Rise
JumpOutOfTheWater

CheckDistance
SprintTo
LiftOff
Land

Swim
Walk
Jump
Fly

Mapping Popular Functions

Animal
- name : String
- code : Integer

Oviparous
- LaysEggs()

Mammal
- Nursels()

Swim()
Fly()
Walk()
Alternatives for Mapping

- Pushing functions up the hierarchy
  - Design errors
- Duplicate function implementation
  - Maintenance nightmare
- Multiple inheritance
  - Ambiguous/not understood
  - Not supported by some languages
  - “Roles” are unconstrained (i.e. are full-fledged classes)

Personalities - Outline

- Designing with roles
- Personalities
  - Concept, Architecture, Components, Syntax, Usage
- The law of personalities
Personality - Concept

- Encapsulate popular functions independent of any specific class hierarchy
- Template Method Pattern++
  - “micro-framework”
- Not abstract classes
  - Embody one, and only one role
- Not interfaces (a-la Java)
  - More constrained
  - Contain behavior implementation

Personality - Architecture

Users of the object only deal with the Personality’s Upstream Interface

The role behavior is encapsulated here... and defined in terms of the Downstream Interface, which classes in the hierarchy must implement.
Personality - Components

- **Upstream interface**
  - popular functions go here

- **Downstream interface**
  - functions to be implemented by personifying class

- **Private functions**
  - no visibility either upstream or downstream

- **Role-specific attributes**
  - to keep the role’s state

- **Constructor**
  - to initialize the role-specific attributes

Personality - Definition Syntax

```java
// 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.
  private void Takeoff();
  private void Ascend();
  private boolean ThereYet(int x, int y);
  private void FlapTowards(int x, int y);
  private void Descend();
  private void Land();

  // attributes (specific to the role)
  private float meters_flown;

  // constructor (optional)
  Flier() { resetMetersFlown(); }
}
```

Each method in the UI is declared and defined.

Each method in the DI is declared but not defined.

Private aux. functions and attributes.

Constructor
public class Bat extends Mammal implements Flier {

    boolean inDraculaMode;

    void UpdateMode(Time time) {
        if (time > SUNLIGHTOUT) inDraculaMode = true;
        else inDraculaMode = false;
    }

    Bat() { inDraculaMode = false; }

    Boolean BiteBeautifulLady(Lady lady) {
        if (inDraculaMode) lady.BittenBy( this );
        return inDraculaMode;
    }

    // since a Bat flies, use the Flier personality with
    // the following implementations of the DI
    Compass _compass = new Compass();

    void waitUntilInDracula() {
        while(!inDraculaMode) {
            UpdateMode( new Date() );
            Thread.sleep( 5000 );
        }
    }

    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.uninitialized())
            _compass.set_target(x, y);
        // do whatever I need to move...
        _compass.updateposition();
    }

    void Descend() { /* not shown */ }

    void Land() { /* not shown */ }
}
LoP - No default implementation

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

vs.

// compute and return today’s date
// in the format “YYYYMMDD”
String Today();

LoP - Basic Types (recommendation)

personality Foo {
    ...
    // return today
    \textcolor{red}{MyDateClass} Today();
}

vs.

// return today
\textcolor{red}{java.util.Date} Today();
...
}

This small comment makes all the difference in the world!

This forces Foo to always have to be deployed with MyDateClass!
LoP - Behavioral Buffer

// SeaWorldShow() is a client
// of Swimmer personality
void SeaWorldShow(Swimmer s) {
    s.Swim(10,10); // ok, UI used
    s.Submerge(); // error, DI used
}

LoP - Fixed Popular Behavior

// 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();
        for(int a = altitude; a > 0; a++)
            Descend();
        Land();
    }
}

This LazyPelican is going to crash! What happens to all the clients of the Flier personality now?
LoP - Implementation Separation

- Popular functions’ implementation can only use DI methods to access the personifying class
  - helps make sure that the DI is complete for the intended semantics of the personality
  - keeps the personality encapsulated

- However, requiring “behavior impedance” is too constraining
  - that is, pass-through UI methods are sometimes needed (and healthy)

Keeping everybody happy (“personifies” relationship)

I can talk to the animals as Mammal and Oviparous.

And to me they are just Fliers, Swimmers, Walkers, and Jumpers!
Dynamic Personalities

- Why bother?
  - Workarounds for object migration
- Shortcomings of “static” personalities
- What we’d like
  - Partially-dynamic personalities (a.k.a. “indecisive personalities”)
  - Fully-dynamic personalities

---

Roles are very much “dynamic”

*Object migration problem*

```java
Person john = new Person();
Medsys.addPatient(john);

// john joins a company. Need to use the “Employee” interface, but:
john.yellToBoss(); // => undef!
// so we need to do:
Employee tmp = new Employee();
// copy all the state from john to tmp
// and get rid of john
delete john;
tmp.yellToBoss(); // now it works
```
Workarounds for the Object Migration problem

Using personalities (even the static version) helps, since:

```java
// Person will (someday) be Employee and,
// with any luck, also a Manager
class Person personifies Employee, Manager ...
...
Person john = new Person();
```

Whoever has the john instance, can still use methods from Manager and/or Employee’s upstream interface without any reclassification!

Shortcomings of “static” Person

- tough to do

```java
// personality code
class Person
   personifies Employee, Manager ...

// client code
Person john = new Person();
((Manager)john).yellToBoss();
```

But wait! John is still not even an Employee, much less a Manager!

No common protocol among personalities
What We Would Like

- Runtime personality attachment and detachment
- Preservation of object identity
- Preservation of typing properties
- Common interface for all personalities
- Ubiquitous personification
- Reasonable performance

Indecisive Personalities - 1

- "personifies" semantics change to mean "can" instead of "does"
- Third-party instructs the class to activate and deactivate the personality – security concerns are ignored
- Still need to declare all potential personalities at class-creation time

// client code (acts as third party)
Person john = new Person(); // born
... 
john.personify( "Employee" ); // hired
... 
john.personify( "Manager" ); // promoted
... 
john.forget( "Manager" ); // fired
john.forget( "Employee" );
Indecisive Personalities -2
Report Card

- Runtime attachment and detachment
- Preservation of object identity
  - same framework as static personalities
- Preservation of Typing properties
  - compile-time strong typing
  - possibility of "non-active" personality must be recognized by client program
- Common interface among personalities
  - "personify()", "forget()", "personifies()" etc.
- Reasonable performance
- Ubiquitous personification

Hey!
Five out of Six isn’t that bad!

Personalities/J - Outline

- The environment and assumptions
- Knowing and unknowing clients
- The mapping process
- Static PJ
- Indecisive PJ
Environment and Assumptions

- The system is built using PJ alone
- Java is only an “intermediary” step
- Semantic analysis is done on the PJ code
  - type checking

The Mapping Process (to Java)

```
interface Flier
{ ... }

class Flier$Ego
{ ... }

class Bat
implements Flier
{ ... }

class Zoo
{ ... // uses }
```

```
Flier.java
interface Flier
{ ... }

Flier$Ego.java
class Flier$Ego
{ ... }

Bat.java
class Bat
implements Flier
{ ... }

Zoo.java
class Zoo
{ ... // uses }
```
Static PJ - «class». java

// SeaLion.java [static]
import java.util.*;
public class SeaLion extends Mammal {  
    implement Walker, Jumper, Swimmer

    // Copy intact from SeaLion.pj...
    // ============== for Swimmer
    Swimmer$Ego $swimmer = new Swimmer$Ego();
    public void Swim(int miles, int depth) {
        $swimmer.Swim(this, miles);
        // ============== for Walker
        Walker$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);
            }
        }
    }
}

Static PJ - «personality». java

// Swimmer.java [static]
interface Swimmer {
    public void Swim(int miles, int depth);
    void JumpInTheWater();
    void Submerge();
    void MoveFin();
    void Rise();
    void JumpOutOfTheWater();
}
Static PJ - «pers»$Ego.java

```java
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() {
    }
}
```

Use the interface as the type of the parameter. This is how it calls back to the personifying class.

Dynamic PJ - Common Protocol

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>personify( &quot;&lt;personality&gt;&quot; )</td>
<td>Enable personification of &lt;personality&gt;</td>
</tr>
<tr>
<td>forget( &quot;&lt;personality&gt;&quot; )</td>
<td>Disable personification of &lt;personality&gt;</td>
</tr>
<tr>
<td>personifies( &quot;&lt;personality&gt;&quot; )</td>
<td>Returns true or false depending on whether &lt;personality&gt; is enabled in the class.</td>
</tr>
<tr>
<td>personalities()</td>
<td>Returns a Vector of Strings with the names of all the personalities that are enabled in the class.</td>
</tr>
</tbody>
</table>
Dynamic PJ - Client Changes

// 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(10,10,10);
}

This client is a "knowing" client which uses the common protocol to decide what to do (not very useful, I know! :-)

Personalities and Frameworks Outline

- Using Hotspots to plug-in a framework
  - through inheritance
  - through delegation
  - through personalities
- Personalities as traffic cops
  - the benefits
  - composing frameworks using personalities
  - delegating composition to the app’s code
Using Personalities for Hotspots

- Easy to adapt a framework
  - inheritance-like
- Flexible
  - multiple inheritance-like
- Robust
  - solve the object identity problem introduced by delegation
- Plus some semantic guarantees
  - encapsulated in their UI methods

Personalities as Traffic Cops

- The additional behavior encapsulation really helps…
- When we need to integrate frameworks…
- Or compose them
Other Northeastern Collaboration-Based Work

- **Mira’s APPCs**
  - one APPC per DI method
  - multiple personification becomes essential

- **Johan’s CGVs**
  - behavior methods similar to UI methods
  - map methods similar to DI methods

Inheritance of Personalities

- Useful for decreasing granularity

```java
personaity 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();
  ...
}
```

- Implement some of Flier’s DI methods...
- And add some of your own.

- Keep “standard” inheritance semantics
Related Works

- **Using delegation**
  - LaLonde86, Gottlob96
  - ‘roles’ are tightly bound to the class, not easy to apply them to different hierarchies

- **Relaxing inheritance**
  - Bracha90,92, Holland93, Lucas94, Flatt98, Mezini98, Seiter98
  - do not attempt to commit to a specific behavior (as in popular functions)

- **The visitor pattern**
  - Krishnamurthi98, Gamma94
  - no notion of DI, code duplication, strongly tied to class hierarchy

Related Works (cont’d)

- **Subject-Oriented programming**
  - Harrison93
  - similar goals to personalities, no concept of UI/DI, no enforcement of LoP

- **Adaptive programming**
  - Lieberherr96
  - no enforcement of LoP

- **Rapide**
  - Luckham95
  - provided and required similar to upstream and downstream, different scope
Personalities/J Compiler

- Built entirely using Demeter/Java 0.7
- Distributed team
  - GTE Labs
  - Catholic University of Cordoba, Argentina
    - Daniel Gandara
    - Diego Rodrigo
- Version 0.1.9
  - static personalities
  - indecisive are around the corner! :-)

Personalities/J Compiler

- Built entirely using Demeter/Java 0.7
- 63,000 chars written
- 2,047,000 chars generated
- 1000 line (18,400 chars) class dictionary
- 8 traversals
  - from CompilationUnit via ClassDeclaration to UnmodifiedClassDeclaration
  - creates 147 line traversal graph (*.trv)