Introduction to OOP & Design Principles

{OOP}

Michael P. Redlich

Who’s Mike?

- InfoQ Java Queue News Editor
- Co-Director, Garden State Java User Group
- Leadership Council, Jakarta EE Ambassadors
- Committer, Jakarta NoSQL and Jakarta Data
- “Petrochemical Research Organization”

Objectives

- Object-Oriented Programming
- Object-Oriented Design Principles
- Live Demos (yea!)

Object-Oriented Programming (OOP)

Some OOP Languages

- Ada
- C++
- Eiffel
- Java
- Modula-3
- Objective C
- OO-Cobol
- Python
- Simula
- Smalltalk
- Theta

What is OOP?

- A programming paradigm that is focused on objects and data
    - as opposed to actions and logic
- Objects are identified to model a system
- Objects are designed to interact with each other

OOP Basics (1)

Procedure-Oriented

- Top Down/Bottom Up
- Structured programming
- Centered around an algorithm
- Identify tasks; how something is done

Object-Oriented

- Identify objects to be modeled
- Concentrate on what an object does
- Hide how an object performs its task
- Identify behavior

OOP Basics (2)

- Abstract Data Type (ADT)
  - user-defined data type
  - use of objects through functions (methods) without knowing the internal representation

OOP Basics (3)

- Interface
  - functions (methods) provided in the ADT that allow access to data
- Implementation
  - underlying data structure(s) and business logic within the ADT

@mpredli

OOP Basics (4)

Class

- Defines a model
- Declares attributes
- Declares behavior
- Is an abstract data type

Object

- Is an instance of a class
- Has state
- Has behavior
- May have many unique objects of the same class

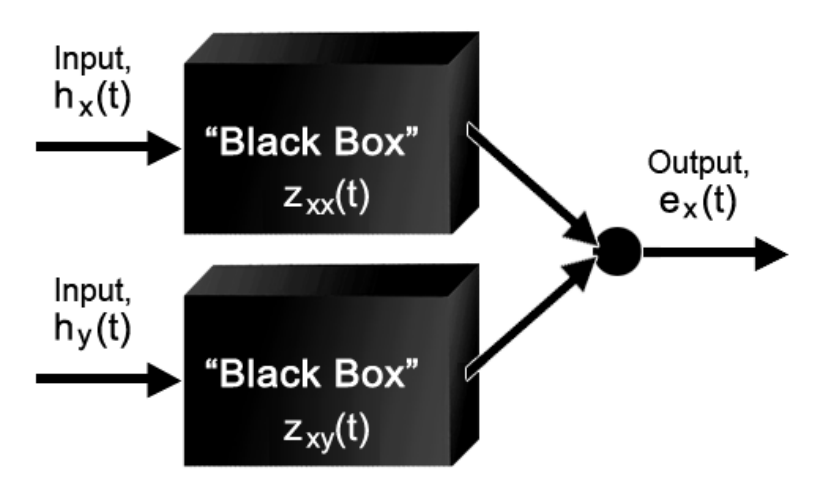
OOP Attributes

- Four (4) Main Attributes:
  - data encapsulation
  - data abstraction
  - inheritance
  - polymorphism

Data Encapsulation

- Separates the implementation from the interface
- A public view of an object, but implementation is private
  - access to data is only allowed through a defined interface

Data Abstraction



- A model of an entity
- Defines a data type by its functionality as opposed to its implementation

@mpredli

Inheritance

- A means for defining a new class as an extension of a previously defined class
- The derived class inherits all attributes and behavior of the base class
- “IS-A” relationship
    - `Baseball` is a `Sport`

Polymorphism

- From the Greek meaning "many forms"
- The ability of closely-related objects to respond differently to the same function

Advantages of OOP

- Interface can (and should) remain unchanged when improving implementation
- Encourages modularity in application development
  - Better maintainability of code
  - Code reuse
- Emphasis on what, not how

Classes (1)

- A user-defined abstract data type
- Based on the C `struct` mechanism
- Contain:
  - constructor
  - destructor
  - data members and member functions (methods)

Classes (2)

- Static/Dynamic object instantiation
- Multiple Constructors:
    - `Sports(void);`
    - `Sports(char *,int,int);`
    - `Sports(float,char *,int);`

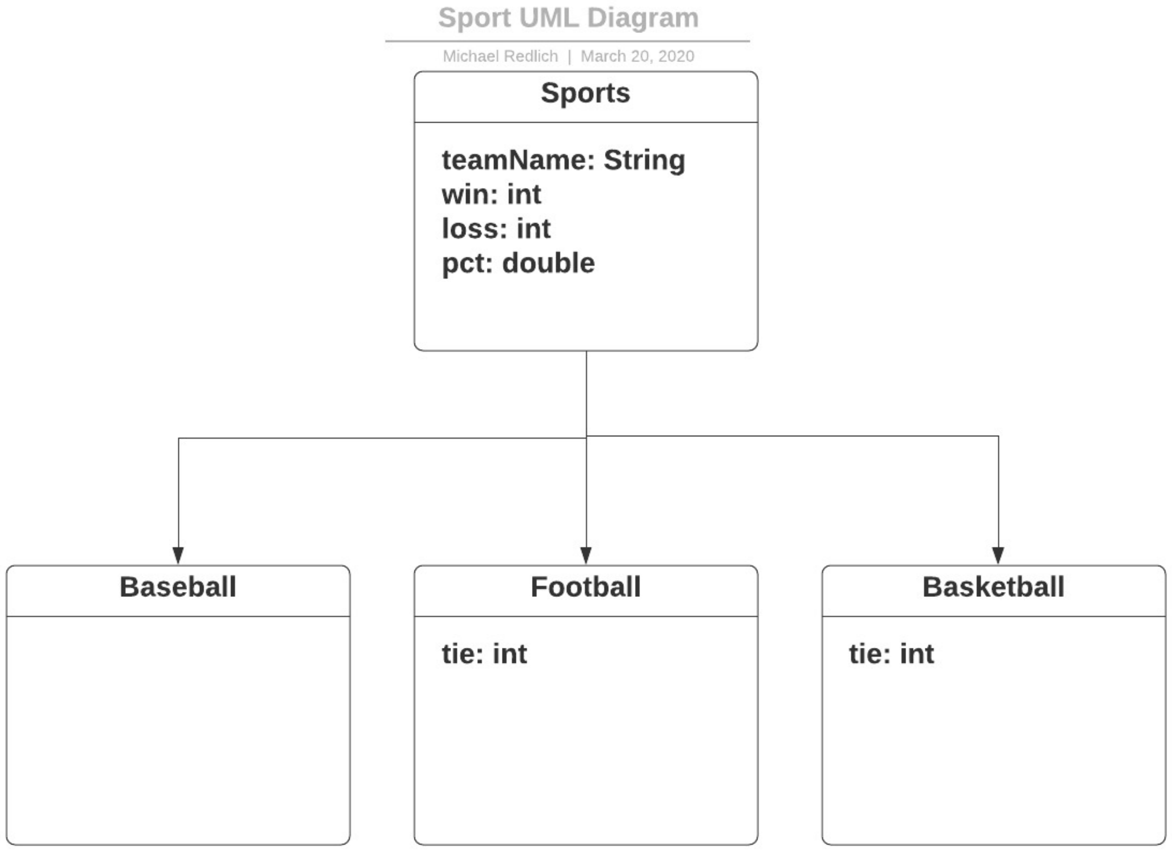
Classes (3)

- Class scope (C++)
  - scope resolution operator (: :)
- Abstract Classes
  - contain at least one pure virtual member function (C++)
  - contain at least one abstract method (Java)

Declaring Abstract Methods

- Pure virtual member function (C++)
  - `virtual void draw() = 0;`
- Abstract method (Java)
  - `public abstract void draw();`

Class Inheritance



Sport UML Diagram

Michael Redlich | March 20, 2020

Sports

teamName: String
win: int
loss: int
pct: double

Baseball

Football

tie: int

Basketball

tie: int

Static Instantiation (C++)

- Object creation:
  - `Baseball mets("Mets",97,65);`
- Access to public member functions:
  - `mets.getWin(); // returns 97`

Dynamic Instantiation (C++)

- Object creation:

  -
    ```
    Baseball *mets = new
    Baseball("Mets",97,65);
    ```
- Access to public member functions:

  -
    ```
    mets->getWin(); // returns 97
    ```

Dynamic Instantiation (Java)

- Object creation:
  - `Baseball mets = new Baseball("Mets",97,65);`
- Access to public member functions:
  - `mets.getWin(); // returns 97`

Deleting Objects (C++)

```
Baseball mets("Mets",97,65);

// object deleted when out of scope
```

```
Baseball *mets = new
Baseball("Mets",97,65);

delete mets; // required call
```

Deleting Objects (Java)

```
Baseball mets = new
Baseball("Mets",97,65);

// automatic garbage collection or:

System.gc(); // explicit call
```

Object-Oriented Design Principles

What are OO Design Principles?

- A set of underlying principles for creating flexible designs that are easy to maintain and adaptable to change
- Understanding the basics of object-oriented programming isn't enough

Some OO Design Principles (1)

- Encapsulate What Varies
- Program to Interfaces, Not Implementations
- Favor Composition Over Inheritance
- Classes Should Be Open for Extension, But Closed for Modification

Some OO Design Principles (2)

- Strive for Loosely Coupled Designs Between Objects That Interact
- A Class Should Have Only One Reason to Change

Encapsulate What Varies

- Identify and encapsulate areas of code that vary
- Encapsulated code can be altered without affecting code that doesn’t vary
- Forms the basis for almost all of the original Design Patterns

```
// OrderCars class

public class OrderCars {
  public Car orderCar(String model) {
    Car car;
    if(model.equals("Charger"))
      car = new Dodge(model);
    else if(model.equals("Corvette"))
      car = new Chevrolet(model);
    else if(model.equals("Mustang"))
      car = new Ford(model);

    car.buildCar();
    car.testCar();
    car.shipCar();
    }
  }
```

{OOP}

Demo

Program to Interfaces, Not Implementations

- Eliminates being locked-in to a specific implementation
- An interface declares generic behavior
- Concrete class(es) implement methods defined in an interface

```
// Cat class

public class Cat {
  public String meow() {
    return "meow";
    }

// Cow class

public class Cow {
  public String moo() {
    return "moo";
    }
  }
```

```
// Animals class - main application

public class Animals {
  public static void main(String[] args) {
    Cat cat = new Cat();
    System.out.println("The cat says, " + cat.meow());

    Cow cow = new Cow();
    System.out.println("The cow says, " + cow.moo());
    }
  }

// output
The cat says, meow
The cow says, moo
```

```
// Animal interface

public interface Animal {
 public void speak();
 }
```

```
// Cat class (revised)

public class Cat implements Animal {
  public void speak() {
    meow();
    }

  public String meow() {
    return "meow";
    }

// Cow class (revised)

public class Cow implements Animal {
  public void speak() {
    moo();
    }

  public String moo() {
    return "moo";
    }
  }
```

```
// Animals class - main application (revised)

public class Animals {
  public static void main(String[] args) {
    Animal cat = new Cat();
    System.out.println("The cat says, " + cat.speak());

    Animal cow = new Cow();
    System.out.println("The cow says, " + cow.speak());
    }
  }

// output
meow
moo
```

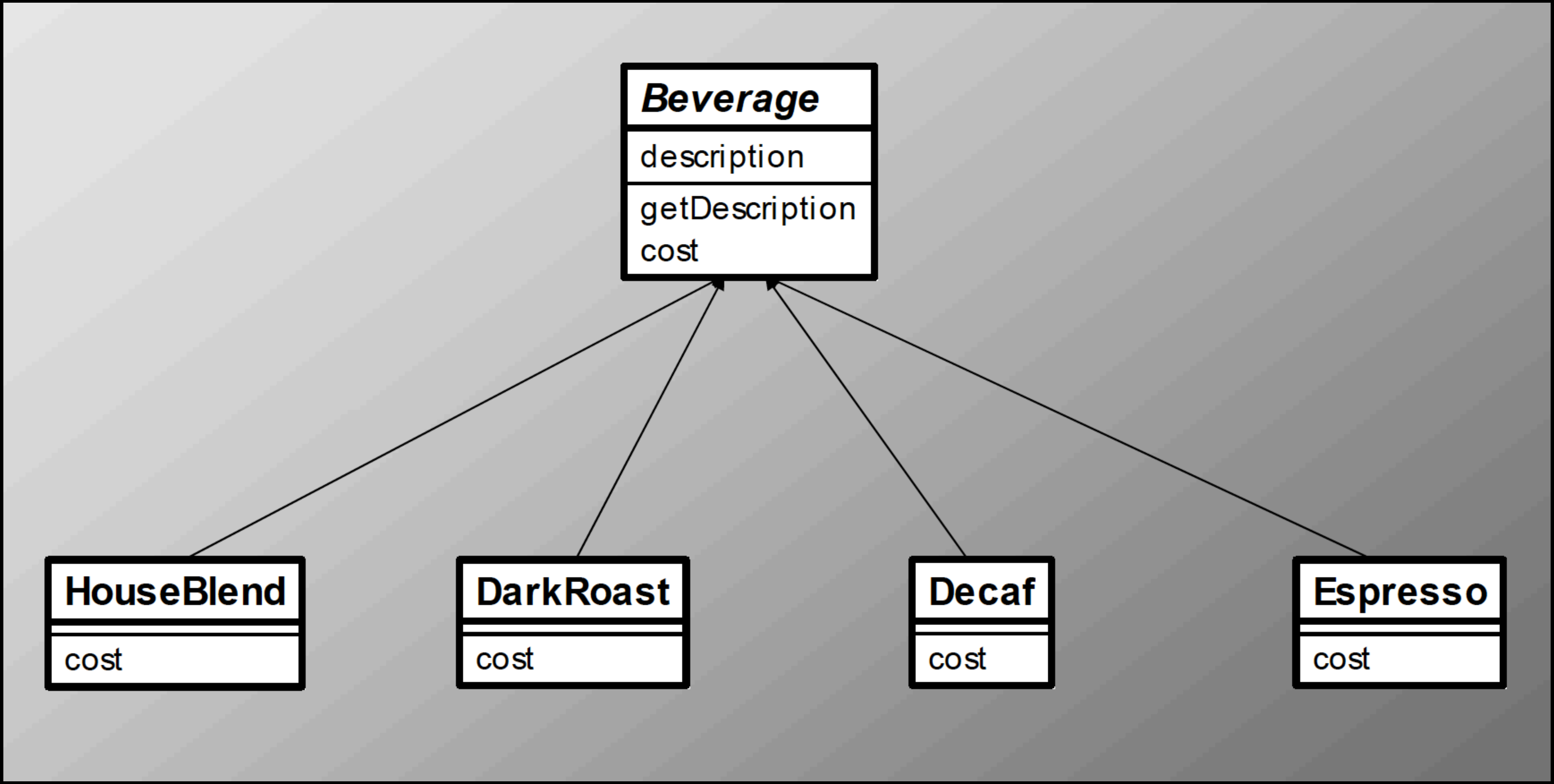
Favor Composition Over Inheritance

- “HAS-A” can be better than “IS-A”
- Eliminates excessive use of subclassing
- An object’s behavior can be modified through composition as opposed through inheritance
- Allows change in object behavior at run-time

Classes Should Be Open for Extension...

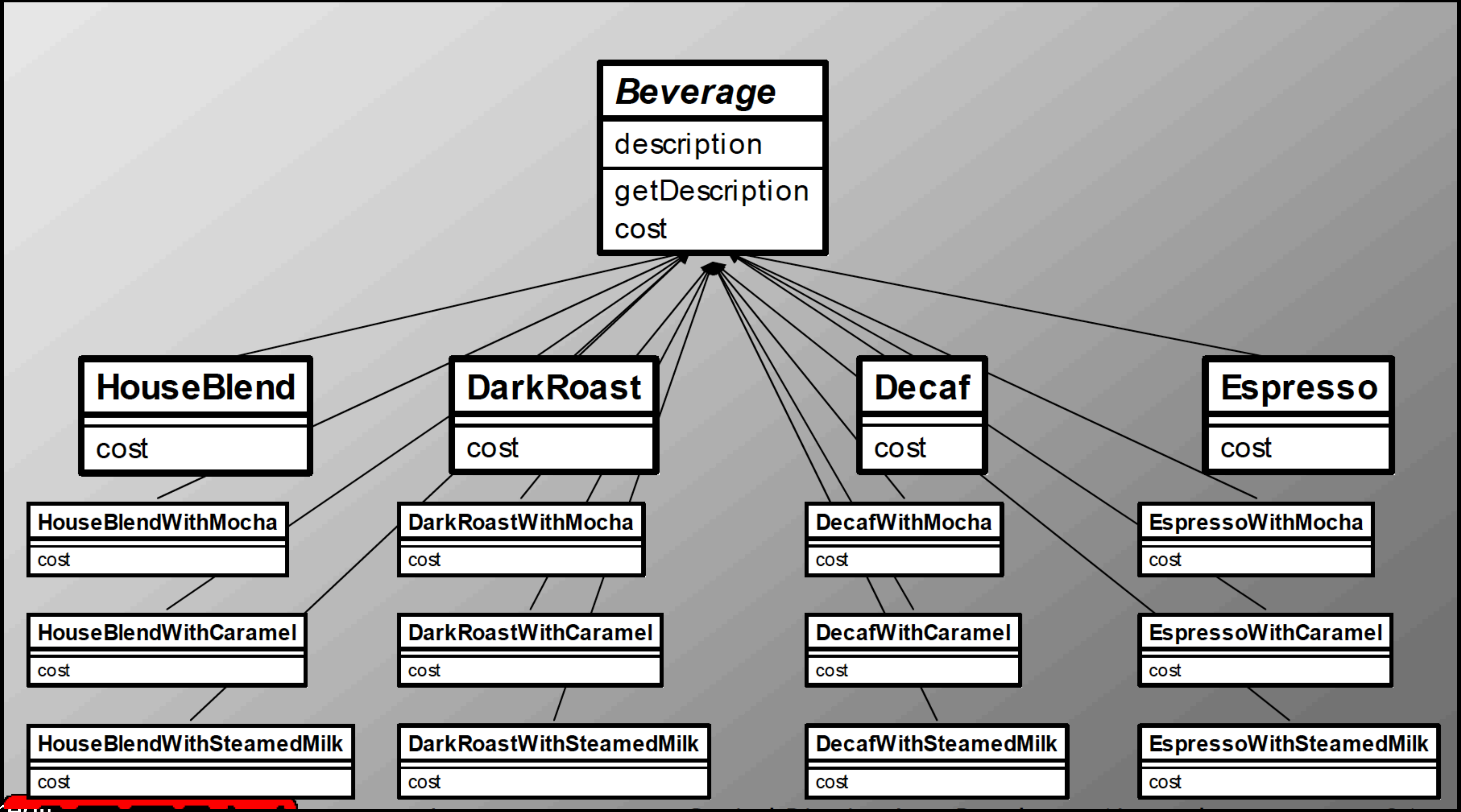
- ...But Closed for Modification
- “Come in, We’re Open”
  - extend the class to add new behavior
- “Sorry, We’re Closed”
  - the code must remain closed to modification

A Simple Hierarchy...

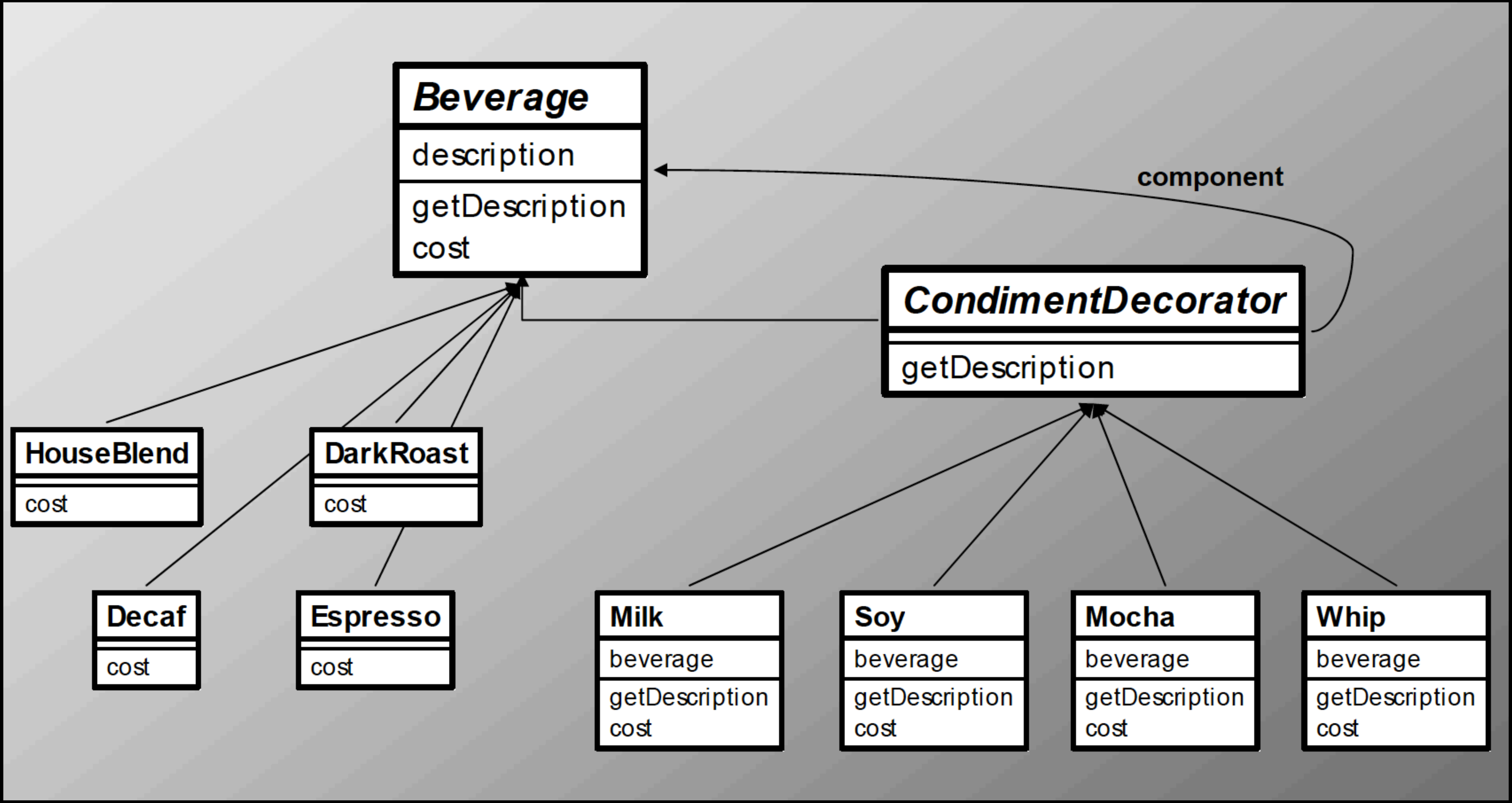


@mpredli

...That Quickly Becomes Complex!



Refactored Design

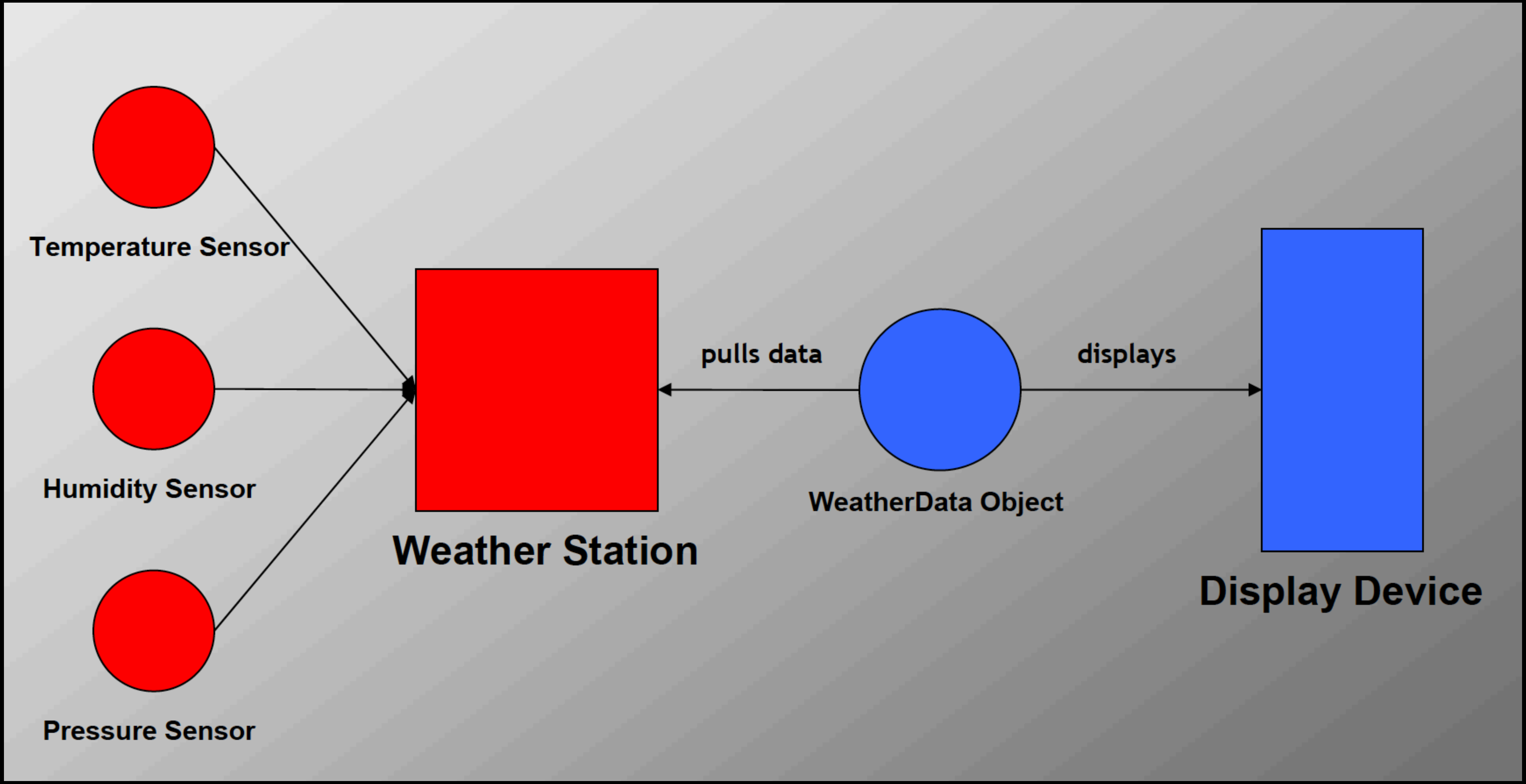


@mpredli

Strive for Loosely Coupled Designs...

- ...Between Objects That Interact
- Allows you to build flexible object-oriented applications that can handle change
    - interdependency is minimized
- Changes to one object won’t affect another object
- Objects can be used independently

Publisher/Subscriber



@mpredli

{OOP}

Demos

A Class Should Have...

- ...Only One Reason to Change
- Classes can inadvertently assume too many responsibilities
  - interdependency is minimized
  - cross-cutting concerns
- Assign a responsibility to one class (and only one class)

Local Java User Groups (1)

- Garden State Java Users Group (GSJUG)
  - facilitated by the GSJUG Leadership Team
  - `gsjug.org`
- NYJavaSIG
  - facilitated by Frank Greco, et.al
  - `javasig.com`

Local Java User Groups (2)

- PhillyJUG
  - facilitated by Paul Burton, et. al.
  - **`meetup.com/PhillyJUG`**
- Jersey City Java Users Group
  - facilitated by Amitai Schleier
  - **`meetup.com/Jersey-City-Java-User-Group-JC-JUG/`**

Local Java User Groups (3)

- Capital District Java Developers Network
  - facilitated by Dan Patsey
  - `cdjdn.com`
  - currently restructuring

Further Reading

A Brain-Friendly Guide

Head First Design Patterns

Avoid those embarrassing coupling mistakes

Learn why everything your friends know about Factory pattern is probably wrong

Discover the secrets of the Patterns Guru

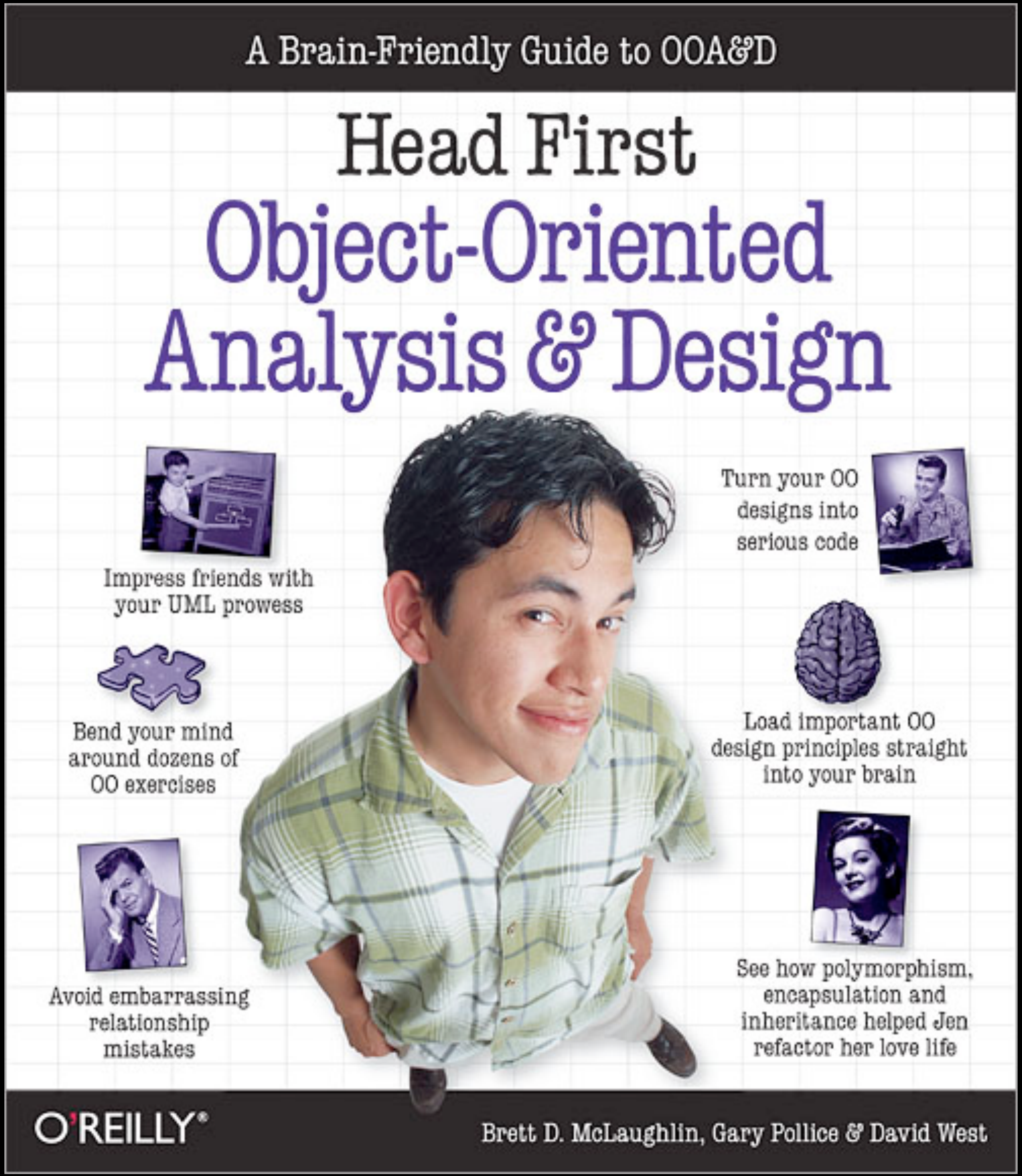
Load the patterns that matter straight into your brain

Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern

See why Jim's love life improved when he cut down his inheritance

O'REILLY®

Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates



A Brain-Friendly Guide to OOA&D

Head First Object-Oriented Analysis & Design

Impress friends with your UML prowess

Turn your OO designs into serious code

Bend your mind around dozens of OO exercises

Load important OO design principles straight into your brain

Avoid embarrassing relationship mistakes

See how polymorphism, encapsulation and inheritance helped Jen refactor her love life

O'REILLY®

Brett D. McLaughlin, Gary Pollice & David West

Resources

- `java.sun.com`
- `headfirstlabs.com`
- `themeteorbook.com`
- `eventedmind.com`
- `atmosphere.meteor.com`

Contact Info

mike@redlich.net

@mpredli

redlich.net

redlich.net/portfolio

github.com/mpredli01

Thanks!

@mpredli