Advanced Object Oriented PHP

Dylan Lane McDonald

CNM STEMulus Center
Web Development with PHP

November 11, 2015

Outline

Inheritance

Definition

Inheritance *is when a class is based on another class. An inherited (child) class gains access to all protected and public state variables and methods. Private state variables and methods are not inherited by the child.*

Inheritance

Definition

Inheritance *is when a class is based on another class. An inherited (child) class gains access to all protected and public state variables and methods. Private state variables and methods are not inherited by the child.*

Inheritance is a mechanism of **code reuse** that allows a class to share code with its parent class for free. Inheritance is a **is-a** relationship. That is, child classes are considered a subset or special case of the parent class. For instance, the **HybridCar** class would be a child class of the **Car** class.

The down side to using inheritance is that the child classes become dependent on the parent classes and are forced to contain the code propagated via inheritance.

Inheriting a Class

```
class HybridCar extends Car {
    protected $batteryCapacity;
    public function getBatteryCapacity() {
      return($this->batteryCapacity);
    }
}
```

Listing 1: Creating a Hybrid from a Car

The **HybridCar** class will have full access to the non-private state variables and methods in **Car**.

`final` Keyword

The `final` keyword prevents child classes from modifying the parent class. This has the effect of creating a read-only inheritance of `final` items. The following items can be declared `final`:

- **Method**: prevents child classes from overriding methods
- **Class**: prevents child classes from modifying anything inheriting from the parent class

Notably absent from this list are state variables. `final` state variables can be approximated by using class constants.

Contracts

Definition

A **contract** *is a collection of state variables or methods that outline the services provided by the contract and what is required to be implemented by the contract.*

Contracts

Definition

A **contract** *is a collection of state variables or methods that outline the services provided by the contract and what is required to be implemented by the contract.*

The term "contract" comes from the Java world. [1] The term can be likened to hiring a contractor to build another room in a house. In the house analogy, one is paying a professional to add something new to a house. In software development, a developer implements additional code with the added benefit of a service or feature provided to the program.

Implementing Contracts: Interfaces

```
interface BatteryCar {
    public function getBatteryCapacity();
}
class HybridCar implements BatteryCar {
    protected $batteryCapacity;
    public function getBatteryCapacity() {
      return($this->batteryCapacity);
    }
}
```

Listing 2: Creating a Hybrid Car

Traits

Definition

A **trait** *is a fragment of reusable code (complete state variables and methods) that is later incorporated into a class.*

Traits

Definition

A **trait** *is a fragment of reusable code (complete state variables and methods) that is later incorporated into a class.*

Like inheritance, traits allow entire code blocks to be injected into a class. Unlike inheritance, multiple traits can be applied to a class. Inheritance only allows one parent class per child.

Traits

```
trait BatteryCarTrait {
    protected $batteryCapacity;
    public function getBatteryCapacity() {
      return($this->batteryCapacity);
    }
}
class HybridCar {
    use BatteryCarTrait;
}
```

Listing 3: Creating a Hybrid Car with a Battery Trait

Diamond Problem

Definition

The **diamond problem** *is a problem where methods have an ambiguous source. This is caused by using multiple inheritance and is depicted in Figure 1.*

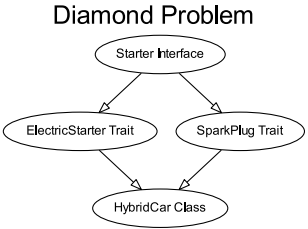

Figure 1: Diamond Problem

Diamond Problem

```
trait SparkPlug {
    public function start() {
        echo "zoom zoom";
    }
}
trait ElectricStarter {
    public function start() {
        echo "started electrically";
    }
}
```

Listing 4: Diamond Problem Using Traits

Diamond Problem (Cont'd)

```
class HybridCar {
    use ElectricStarter, SparkPlug;
}
$hybrid = new HybridCar();
$hybrid->start();
```

Listing 5: Diamond Problem Using Traits (Cont'd)

This illustrates the diamond problem because the `start()` method is ambiguously injected from two sources. The resolution is to either use one trait or specify where the trait comes from by writing `SparkPlug::start insteadof ElectricStarter;` as part of the `use` block.

Composing vs Inheriting

The Gang of Four explicitly say, "Favor object composition over class inheritance." [2] Inheritance exposes the implementation of the parent class to the child classes. This is said to "violate encapsulation" because any change to parent classes directly affect all child classes. This is often not the desired affect. One can be more selective about which behaviors to effect by using composition instead of inheritance.

The main advantage to composition is that it is dynamically decided at runtime, so respect for objects' interfaces is preserved, and hence, encapsulation is preserved. The end result is a system that is more robust and more flexible as business needs change.

Strategy Design Pattern

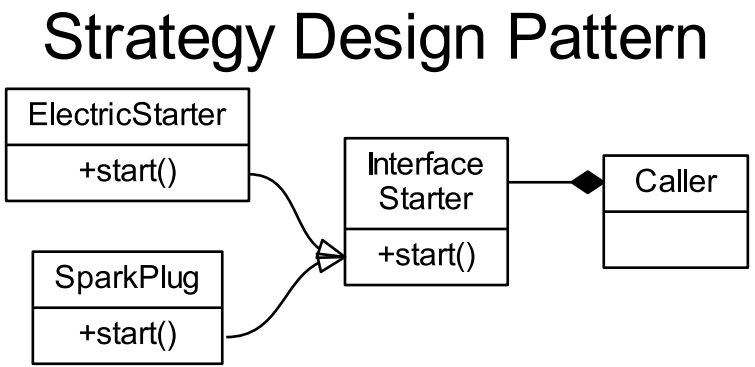

Figure 2: Strategy Design Pattern

Strategy Design Pattern

The traits in Listing 4 can be extended to implement the **Strategy Design Pattern**. The traits can be thought of as two concrete strategies to start a Car object. Next, an interface to enforce this behavior is needed.

```
interface Starter {
    public function start();
}
```

Listing 6: Building Out the Strategy Pattern: the Interface

Both traits from Listing 4 are candidates for this new interface because they completely implement the interface.

Strategy Design Pattern

The final step is the write the class to implement the interface from Listing 6 by simply using one of the traits from Listing 4.

```
class HybridCar implements Starter {
    use ElectricStarter;
}
```

Listing 7: Building Out the Strategy Pattern: the Class

The extra layer introduced by the strategy design pattern allows one to interchange programmatic parts such as traits and interfaces to dynamically create classes with little work. It also adds the advantage of standardizing how all the methods behave without being concerned for what exactly the methods are doing.

Works Cited


John O'Hanley.
Design by contract.
http://www.javapractices.com/topic/TopicAction.do?Id=194, 2015.


Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley Professional, first edition, November 1994.