Back to Basics … Three or Four OOP Pillars?

themoderndeveloper.com/the-modern-developer/back-to-basics-three-or-four-oop-pillars/

July 19, 2013

Encapsulation Featured

A New Addition to the OOP Pillar Family?

– Data Abstraction –

For years I have been asking the classic Programming interview question: "*What are the Three Pillars of Object Oriented Programming*?"

The candidate would confidently exclaim: "*Encapsulation, Inheritance and Polymorphism*!"

I would be obligated to accept that as a correct answer.

But is it really?

My Software Development Odyssey has taken me down somewhat of a different path.

The original OOP Pillars that I have embraced for years now seem to be missing an important stanchion:

BackTo Basics

The Data Abstraction

> ***Encapsulation of State in Data Transfer Objects***
>
> ***… Using Composites and Aggregates DTOs***

The Four Pillars Paradigm

Pillar Number One: *Encapsulation*

> ***Details Required to Return a Result from a Class Method should be Hidden from the Calling Method***

Encapsulation Pillar

Encapsulation provides a valuable service to your design: *It hides the implementation*. It enables compliance with the Object Oriented Principle: *Separation Of Concerns (SOC)*

The concept protects the objects within the type from being altered in a way that is unintended by the Developer.

Encapsulation also abstracts away the *Dependencies* that the Encapsulated Class Method requires from the calling Software Entity.

Encapsulation helps the Developer to comply with the “*O*” in the *S.O.L.I.D.* Design Principles: *Open / Closed Principle*.

Pillar Number Two: *Inheritance*

> ***The Ability to Enable Objects to Share Common Functionality of the Parent Object. Inheritance Promotes the Reuse of Existing Object’s Code Elements***

Inheritance Pillar

Conventional use of Inheritance is Class based.

This assumes that the relationship between the Parent Class and the Children Classes has an “*Is A*” relationship.

The Derived Child Class “*Is A*” a variation of its Parent Class.

In the *Animal Kingdom* an “*Is A*” relationship is implicit. Unfortunately in *Programming Kingdom* it is not implicit*.*

In reality, most programming inheritance structure are really a “*Behaves Like*” relationship more that an “*Is A*” relationship.

Interface Inheritance, or Composition, should be used when it is clear that the *Derived Type* is not a variation of its *Base Type.*

An improper inheritance model leads to unmanageable code over the life cycle of the code base due to inevitable change requests and feature/functionality additions.

Pillar Number Three: ***Polymorphism***

> ***The Ability to Derive Different Concrete Behaviors from a Common Abstract Type***

In *Class Inheritance* we can accomplish Polymorphic Behavior by deriving Classes from a Common Base “*Parent*” Class

Polymorphism Pillar

We then create different child variations of the parent

The variation is Object “*Typed*” as a variation of the Parent Type.

In Interface inheritance we create the “*Behaves Like*” Polymorphic Behavior by using the *Contract* of the *Interface* as the *Base Behavior.*

We implement the *Interface Behavior Definitions* in the concrete Types to Inherit the Interface behavior.

Pillar Number Four: ***Data Abstractions***

> ***The Encapsulation of Data within a Single Object that includes Primitives and other Complex Objects as Related Members***

Data Abstraction Pillar

Data is State.

State Objects are *Classes* that only contain *Properties* and *Constructors.* Properties are initialized by Constructors.

> ***These Data Abstractions are called Data Transfer Objects: A DTO***

A DTO abstracts the details of the consumable properties, and their initialization process, from the consuming method. The DTO can manage the property object through access modifiers within the *getters and setters* for the property using the three available class constructors.

The Data Abstraction can be a simple encapsulation DTO that holds a single primitive:

Public *bool* ***IsSuccess*** *{get; set}*

The DTO Data Abstraction can be a Container

… Acting as a Transport Package for other DTO Types

+Transport Package Example

This displays the *Derived Complex DTO* and its *Member Properties* with its *Collection Objects*. This Object inherits the *Error Information* from its *Base Class Error Object*.

***Public** int **CompanyId** {get; set}*

***Public** string **CompanyName** {get; set}*

***Public** List<CustomerContact> **CustomerContactsList** {get; set}*

***Public** bool **IsActive** {get; set}*

This is the *Base Class Error Object DTO* that delivers status for the entire *Complex Object*:

***Public** bool **IsSuccess** {get; set}*

***Public** List<Error> **ErrorList** {get; set}*

Composite and Aggregate Data Abstractions

Composites and *Aggregates* are State DTO that perform specific roles in Data Abstractions.

They are containers for *State Data* that are defined by their *Role* in the *Business Domain*

Domain Driven Design, Data Abstractions and the Business Domain

In Domain Driven Design (DDD) an ***Entity*** is an object that has a *Domain Identity* and *Can Stand Alone as a Domain Member*.

A ***Value Object*** is an object that has *No Domain Identity* and only has *Domain Member Value* when associated with an ***Entity*** within the *Domain*.

> ***In the Data Abstraction Pillar a DTO Performs Domain Roles***

+Domain Roles
It is important to understand the Roles and Relationships to the Business Domain that well architected State DTOs perform.

1. **An Aggregate DTO** – A *Value Object* that has identity only to itself. It can belong to a Collection of its Types and be part of a larger DTO
    1. A TIRE DTO is an Aggregate to a Collection of TIRE objects: *TIRES*
    2. A List<TIRE> can then be a member of a Complex DTO: *CHASSIS*

3. **A Composite DTO** – An *Entity Object* into itself with an Identity. This identity may or may not be represented as a *Domain Entity*.
    1. The TIRES Composite DTO identifies the collection of four tires for the VEHICLE Entity but is not a Domain Entity as the TIRES by themselves has to Value unless associated with the *CHASSIS* Entity of the *Domain*: *VEHICLE*
    2. The Composite TIRES is therefore
        1. A *Aggregate Object* to CHASSIS
        2. A *Composite Object* to TIRE

1. **A Complex Composite DTO** – A true Domain Entity that contains other Composites and Aggregates. The Complex DTO is the Transport Package for the lower level Composites and Value Object Aggregates
    1. The *CHASSIS* Complex Composite holds the *TIRES* and all other Child Composites that make up the *CHASSIS*
    2. When called as an Entity the *CHASSIS* Complex DTO Composite DTO holds the Base Object *ERROR* DTO as its *Status* object for the *Domain*: *VEHICLE*

The New Pillar of OOP: Data Abstrations

Adding this “*Fourth Pillar*” creates a *Design Model* for the separation of the *State* and the *Behavior* responsibilities.

As a bridge to Domain Driven Design Entity paradigm, Data Abstractions gives the Architect and Developer a Business tool for a better understanding the intent of the Solution Data being managed by the Application.

> ***The Forth Pillar Provides Separation Of Concerns***
>
> ***… For objects that Manage Behavior from Objects that Manage State***