Anemic vs. Rich Domain Models

‹ Previous Next ›

Anemic vs. Rich Domain Models

There are a lot of discussions about rich and anemic domain models. Some developers tend to praise the anemic model, because of it's simplicity while other blame the anemic model for it's simplicity and they pray for the rich domain model.

In this blog I want to show both models, the advantages and disadvantages that come with each model to give you a decision and discussion basis.

The anemic model

An anemic model is a domain model that doesn't contain any logic. That is why the model is called anemic. It is just a container for data that can be changed and interpreted by clients. Therefore all logic is placed outside the domain objects in an anemic model.

Let's take a look at a simple anemic model.

```
public class Order {
   private BigDecimal total = BigDecimal.ZERO;
   private List<OrderItem> items = new ArrayList<OrderItem>();

   public BigDecimal getTotal() {
      return total;
   }

   public void setTotal(BigDecimal total) {
```

Datanect

A tool for developers and tester.
Get a **free trial now**

René Link
33,847
●10 ●72 ●103

I am a codementor

Follow me on stackoverflow

Just started! Have not answered any questions.

Popular Recent

Anemic vs. Rich Domain Models
October 1st, 2011

How to fix
java.lang.ClassNotFoundExceptior

```
      this.total = total;
   }

   public List<OrderItem> getItems() {
      return items;
   }

   public void setItems(List<OrderItem> items) {
      this.items = items;
   }
}

public class OrderItem {
   private BigDecimal price = BigDecimal.ZERO;
   private int quantity;
   private String name;

   public BigDecimal getPrice() {
      return price;
   }

   public void setPrice(BigDecimal price) {
     this.price = price;
   }

   public int getQuantity() {
     return quantity;
   }

   public void setQuantity(int quantity) {
     this.quantity= quantity;
   }

...

}
```

When using anemic domain models the logic to alter and interprete the data must be placed somewhere else. Most times the logic is placed in classes named ***Service, Util, Helper or Manager***. I guess the naming depends on what sounds more freaky and is up to date. Service-oriented architectures are up to date and therefore the classes are often named

com.ibm.websphere.ssl.protocol.S
in eclipse
July 20th, 2014

The MVC pattern implemented with java swing
July 20th, 2013

Most viewed

How to fix java.lang.Clas...

13.25 views per day

The MVC pattern implement...

12.41 views per day

Anemic vs. Rich Domain Mo...

6.53 views per day

How to fix IBM Websphere...

6.50 views per day

Impact of native sql quer...

5.52 views per day

Archives

Archives Select Month ▼

Service. But these classes often have nothing in common with the services in a service-oriented architecture. Serivces in a service-oriented architecture are usually application services that encapsulate use cases. Take a look at the Service Layer, Martin Fowler for details… ok but let's get back to the anemic domain models.

From an architectural perspective the "services" that hold the business logic in an anemic domain model are transaction scripts. The only difference to plain transaction scripts is often that they use parameter objects that are named after doamin objects.

An anemic domain service for calculating the total of an order might look like:

```
public class OrderService {

   public void calculateTotal(Order order) {
      if (order == null) {
          throw new IllegalArgumentException("order must not be null");
      }

      BigDecimal total = BigDecimal.ZERO;
      List<OrderItem> items = order.getItems();

      for (OrderItem orderItem : items) {
          int quantity = orderItem.getQuantity();
          BigDecimal price = orderItem.getPrice();
          BigDecimal itemTotal = price.multiply(new BigDecimal(quantity));
          total = total.add(itemTotal);
     }
     order.setTotal(total);
   }
}
```

You might think that this is a clear, simple design and straight forward. Of course it is straight forward, because it is procedural programming. It's simplicity is also it's achilles, because an anemic model can never gurantee it's correctness at any time. An anemic model has no logic that ensures that it is in a legal state at any time. E.g. the order object does not react to changes of it's item list and can therefore not update it's total.
The main problem with anemic models is that they are contradictory with fundamental object-oriented principles like: encapsulation, information hiding.

> *Objects combine data and logic while anemic models separate them.*

The following example shows why an anemic model can never gurantee that it is in a legal state.

```
public class OrderTest {

/**
  * This test shows that an anemic model can be in an inconsistent state,
  * because it doesn't handle it's state changes. So an anemic model can
  * never guarantee that it is in a legal state. State handling of an anemic
  * model is placed outside that object in classes mostly named "Service",
  * "Helper", "Util", "Manager" etc.
  * <p>
  * <blockquote> The problem with an anemic model is that a client must know
  * in which state the object it uses is and which service methods it has to
  * call if it changes the state of an anemic model to keep the object in a
  * legal state. </blockquote>
  * </p>
  */
@Test
public void anAnemicModelCanBeInconsistent() {
     OrderService orderService = new OrderService();
     Order order = new Order();
     BigDecimal total = order.getTotal();

    /*
     * A new order has no items and therefore the total must be zero.
     */
     assertEquals(BigDecimal.ZERO, total);

     OrderItem aGoodBook = new OrderItem();
     aGoodBook.setName("Domain-Driven");
     aGoodBook.setPrice(new BigDecimal("30"));
     aGoodBook.setQuantity(5);

    /*
     * We break encapsulation here as we alter the internal state of the
     * order's item list. This is a common programming pattern when using
     * anemic models.
     */
     order.getItems().add(aGoodBook);

    /*
     * After we added an OrderItem. The Order object is in an illegal state.
     */
```

```
        BigDecimal totalAfterItemAdd = order.getTotal();
        BigDecimal expectedTotal = new BigDecimal("150");

        boolean isExpectedTotal = expectedTotal.equals(totalAfterItemAdd);

       /*
        * Of course the order's total can not be the expected total, because
        * anemic models do not handle their state changes.
        */
        assertFalse(isExpectedTotal);

       /*
        * To fix it we have to call the OrderService to re-calculate the total
        * and bring the Order object to a legal state again.
        */
        orderService.calculateTotal(order);

       /*
        * Now the order is in a legal state again.
        */
        BigDecimal totalAfterRecalculation = order.getTotal();
        assertEquals(expectedTotal, totalAfterRecalculation);
    }
}
```

Another interesting point is that the services that hold the logic are independent from a domain object instance – they are stateless. While some developer think that "stateless" is always a good idea, I want to show you now why it is not "always".

The interpretation of the data is done by the stateless service. While the service is stateless it can't know when it is time to execute the logic and when not. In more detail: The stateless service can not cache the values it calculated unlike the rich domain object. A rich domain object handles it's state changes and therefore knows when it has to recalculate a property's value.

At least an anemic domain model is not object-oriented programming and like martin fowler says:

> *At its worst business logic can be very complex. Rules and logic describe many different cases and slants of behavior, and it's this complexity that objects were designed to work with.*

> *Enterprise Application Architecture, Domain Model, Martin Fowler, http://martinfowler.com/eaaCatalog/domainModel.html*

Anemic models are procedural programming

When you look at the anemic models you will recognize that this pattern is very old. Anemic models are procedural programming.
In the early days of programming the order example would have been implemented this way:

```
struct order_item {
int amount;
double price;
char *name;
};

struct order {
int total;
struct order_item items[10];
};

int main(){
struct order order1;
struct order_item item;
item.name = "Domain-Driven";
item.price = 30.0;
item.amount = 5;
order.items[0] = item;
calculateTotal(order1);
}

void calculateTotal(order o){
int i, count;
count = 0;
for(i=0; i < 10; i++) {
order_item item = o.items[i];
o.total = o.total + item.price * item.amount;
}
}
```

If you compare source 4 with source 3 you will recognize that the style is very very similar.

Using anemic domain models means using procedural programming. Procedural programming is straight forward, but hard to understand if you want to understand the state handling of an application. Furthermore the anemic domain model moves the logic of state handling and data interpretation to the clients and this often results in code duplication or in very fine grained services. Ending up with a lot of services and service methods that are interconected in a wide spread and complicated object net. This makes it hard to find out why an object is in a certain state. To find out why an object is in a certain state means that you have to find the method call hierarchy that the object has been passed through and every place that modified the object.

The rich domain model

In contrast to the anemic domain model the rich domain model follows the object-oriented principles. Therefore a rich domain model is really object-oriented programming.
The purpose of a rich domain model or of object-oriented programming is to bring data and logic together. Take a look at the principles of object-oriented programming for details.

Object oriented means that: an object manages it's state and gurantees that it is in a legal state at any time.

The "Order" class shown in source 1 can be easily transformed to an object oriented version.

```
public class Order {

   private BigDecimal total;
   private List<OrderItem> items = new ArrayList<OrderItem>();

   /**
     * The total is defined as the sum of all {@link OrderItem#getTotal()}.
     *
     * @return the total of this {@link Order}.
     */
   public BigDecimal getTotal() {
      if (total == null) {
         /*
          * we have to calculate the total and remember the result
          */
          BigDecimal orderItemTotal = BigDecimal.ZERO;
          List<OrderItem> items = getItems();

          for (OrderItem orderItem : items) {
             BigDecimal itemTotal = orderItem.getTotal();
            /*
             * add the total of an OrderItem to our total.
             */
             orderItemTotal = orderItemTotal.add(itemTotal);
          }
          this.total = orderItemTotal;
      }
   return total;
   }

   /**
     * Adds the {@link OrderItem} to this {@link Order}.
     *
     * @param orderItem
     *            the {@link OrderItem} to add. Must not be null.
     */
    public void addItem(OrderItem orderItem) {
       if (orderItem == null) {
          throw new IllegalArgumentException("orderItem must not be null");
       }
       if (this.items.add(orderItem)) {
           /*
            * the list of order items changes so we reset the total field to
            * let getTotal re-calculate the total.
            */
            this.total = null;
       }
    }

    /**
      *
      * @return the {@link OrderItem} that belong to this {@link Order}. Clients
      *         may not modify the returned {@link List}. Use
      *         {@link #addItem(OrderItem)} instead.
      */
    public List<OrderItem> getItems() {
         /*
          * we wrap our items to prevent clients from manipulating our internal
          * state.
          */
         return Collections.unmodifiableList(items);
    }
```

```
}

import java.math.BigDecimal;

public class OrderItem {
    private BigDecimal price;
    private int quantity;
    private String name = "no name";

    public OrderItem(BigDecimal price, int quantity, String name) {
       if (price == null) {
          throw new IllegalArgumentException("price must not be null");
       }
       if (name == null) {
          throw new IllegalArgumentException("name must not be null");
       }
       if (price.compareTo(BigDecimal.ZERO) < 0) {
          throw new IllegalArgumentException(
                "price must be a positive big decimal");
       }
       if (quantity < 1) {
          throw new IllegalArgumentException("quantity must be 1 or greater");
       }
       this.price = price;
       this.quantity = quantity;
       this.name = name;
    }

    public BigDecimal getPrice() {
       return price;
    }

    public int getQuantity() {
       return quantity;
    }

    public String getName() {
       return name;
    }

   /**
```

```
     * The total is defined as the {@link #getPrice()} multiplied with the
     * {@link #getAmount()}.
     *
     * @return
     */
    public BigDecimal getTotal() {
       int quantity = getQuantity();
       BigDecimal price = getPrice();
       BigDecimal total = price.multiply(new BigDecimal(quantity));
       return total;
    }
}
```

The advantage of an object-oriented programming is that an object can garantee that it is in a legal state at any time and that no “Service” class is needed anymore.
A test case will show the difference to **source “anemic models can not gurantee that they are in a legal state at any time“.**

```
public class OrderTest {

/**
  * This test shows that a rich model gurantees that it is in a legal state
  * at any time.
  */
@Test
public void richDomainModelMustEnsureToBeConsistentAtAnyTime() {
  Order order = new Order();
  BigDecimal total = order.getTotal();

 /*
   * A new order has no items and therefore the total must be zero.
   */
  assertEquals(BigDecimal.ZERO, total);

  OrderItem aGoodBook = new OrderItem(new BigDecimal("30"), 5,
   "Domain-Driven");
  List<OrderItem> items = order.getItems();

 try {
   items.add(aGoodBook);
 } catch (UnsupportedOperationException e) {
  /*
```

```
     * We CAN NOT BREAK ENCAPSULATION, because the order object will not
     * expose it's internal state to clients. It takes care about it's
     * state and ensures that it is in a legal state at any time.
     */
  }

  /*
    * We have to use the object's mutator method
    */
   order.addItem(aGoodBook);

  /*
    * After we added an OrderItem. The object is still in a legal state.
    */
   BigDecimal totalAfterItemAdd = order.getTotal();
   BigDecimal expectedTotal = new BigDecimal("150");

   assertEquals(expectedTotal, totalAfterItemAdd);
 }
 }
```

Which model to use?

Developers often argue about the model an application should use.

My opinion is that an application should use the object-oriented approach as much as possible. The advantage of object-oriented programming is that an object can gurantee that it is in a legal state at any time. These gurantees or constraints are needed if you want to gurantee that the whole application is always in a legal (expected) state. This is a good way for an application, a web service, a library or a simple class to gurantee it's API. API constraints reduce the complexity of a client, because the client must not check all possible states, it can count on what the API declares.
If you go the other way and build up an application on anemic models your application will become unmaintainable sooner or later.
Starting with an anemic model can be easy, but refactor an application towards a rich domain model architecture later can be cumbersome and fail prone. In most cases this is not feasible and therefore will never be done.

That's why I believe in object-oriented programming and rich domain models.

References

- Source files available though github: https://github.com/link-intersystems/blog/tree/master/anemic-vs-rich-domain-model
- Anemic Domain Model, Martin Fowler
- Domain-Driven Design: Tackling Complexity in the Heart of Software, Eric Evans

Recommended Reading

Related Posts:

1. **Pros and cons of service layer designs**
2. **Make method pre-conditions explicit by design**
3. **The difference between pojos and java beans**
4. **Enums as type discriminator anti-pattern**

By | October 1st, 2011 | Architecture & Design | 3 Comments

Share This Story, Choose Your Platform!

About the Author: René Link

I'm a software developer since 2002 and founded my company Link Intersystems in 2011. My favorite programming language is Java, but I m also experienced in other languages. I focus on design and architecture of java enterprise applications with spring, hibernate or jee. Software quality and therefore clean code belong to my basic principles. ---- If you want others to take you seriously, than take your craftsmanship seriously - be a professional ---- Visit me on careers 2.0 and follow me on stackoverflow.com

3 Comments

Vijayaraja August 21, 2014 at 11:04 am - Reply

simply great article

Rombe June 13, 2016 at 12:55 pm - Reply

thanks

Dominik May 19, 2017 at 3:04 pm - Reply

Sehr schöner Artikel, der das ganze sehr grundlegend verständlich erklärt. Danke dafür 🙂

Leave A Comment

Comment...

Name (required) Email (required) Website

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

I'm not a robot

reCAPTCHA
Privacy - Terms