# Domain Logic Patterns

09/149/2020

## Overview

## Domain Logic Patterns

- Domain Logic
    - How do we structure domain logic and where do we put it?
- Why do we separate Domain Logic?
    - Business rules change rapidly
    - Business rules can be very complex
    - Business rules require significant testing

## Background Pattern Summaries

- Gateway (466): An object that encapsulates access to an external system or resource
- Row Data Gateway (152): An object that acts as a Gateway (466) to a single record in a data source. There is one instance per row.
- Table Data Gateway (144): A single instance that acts as a Gateway (466) to a database table. Once instance handles all the rows in the table.
- Record Set (508): An in-memory representation of tabular data
- Data Mapper (165): A layer of Mappers (473) that moves data between objects and a database while keeping them independent of each other and the mapper itself.
- Active Record (160): An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.

## Domain Model (116)

- An object model of the domain that incorporates both behavior and data
- One class for each type of thing we store and one instance for each one of them.
- Standard OO Strategy: nouns are objects
- Gives us all of the OO tools/patterns we are used to
- We end up with a web of interconnected objects of the things that are meaningful to our system.
- Objects mimic the data in the business and capture business rules (mingles data and process)
- The challenge is in connecting that structure across the paradigm shift to a relational database. That is what the Data Mapper (165) is responsible for.

## Types of Domain Models

- Simple Domain Model
  - One domain object per table
- Rich Domain Model
  - Uses OO techniques like inheritance, other patterns
- Rich is better as things get more complex
- Rich makes mapping to the db harder

## Interface to the Presentation Layer

- Want to keep the details as hidden as possible because business logic changes frequently
- Don't really want to expose all of the Domain layer objects
- The line of what functionality is specific to one kind of transaction and what is shared causes questions about where that code should go
    - In the presentation since it is only used one place (breaking layering rules)
    - In the domain model object (bloated, overly exposed objects)
    - In a transaction script! Or Command pattern
- Careful layering for when things are complex:
    - Command pattern between presentation and Model (MVC)
    - Commands only know how to get the objects they need and call methods in those objects (keep this layer as thin as possible)
    - Business logic goes in domain model objects and use good OO techniques to keep them from bloating

## Managing Creation/Destruction of Domain Objects

- Object Creation
  - Two objects for the same “thing” causes identity problems.
  - IdentityMap can make sure that we only have one object
  - When two sessions (threads) share the same object, we have to worry about
    - threading issues
    - consistency issues (when one thread updates the object, does the other thread notice the change)
    - persistency issues (who gets to write the changes - maybe the other thread is halfway through an update)
- Destruction
  - If an IdentityMap is referring to the object, it will never get garbage collected
  - IdentityMap could track how many references the object has
  - Everyone who asks for an object has to say when it is finished with that object.

## Transaction Scripts (110)

- Organizes business logic by procedures where each procedure handles a single request from the presentation.
- Each interaction with the system is essentially a transaction with its own transaction script encoding the logic of that interaction
- Each transaction is individually coded, but they might share routines for common functionality
- Where do you put them?
    - server page
    - cgi script
    - distributed session object

## Server Page Example

```
<!DOCTYPE html>
<html>
<body>
<form action="demo_reqquery.asp" method="get">
Name: <input type="text" name="fname" size="20" />
<input type="submit" value="Submit" />
</form>
<%
    dim fname
    fname=Request.QueryString("fname")
    If fname<>"" Then
        Response.Write("Hello " \& fname \& "!<br>")
        Response.Write("How are you today?")
    End If
%>
</body>
</html>
```

## CGI Script Example

In the html:

```
<FORM METHOD="GET" ACTION="https://SERVER/cgi-bin/USER/
    PROGRAM">
<b> Enter argument: </b>
<INPUT size=40 name=q id=q >
<INPUT TYPE="submit" VALUE="Submit">
</FORM>
```

The CGI code:

```
#!/bin/sh

echo "Content-type: text/html"
echo

echo '<html> <head> <title> CGI script </title> </head>
    <body>'

argument=`echo "$QUERY_STRING" | sed "s|q=||"`

echo "   QUERY_STRING is: <b> $QUERY_STRING </b> <br>"
echo "Actual argument is: <b> $argument      </b> <br>"
```

## Distributed Session Objects (1 of 3)

```
@Stateless
public class AccountService implements AccountServiceRemote
    {
    @PersistenceContext
    private EntityManager em;

    @Override
    public Customer createAccount(String firstName, String
        lastName) {
        Customer customer = new Customer();
        customer.setFirstName(firstName);
        customer.setLastName(lastName);

        em.persist(customer);

        return customer;
    }
}
```

## Distributed Session Objects (2 of 3)

```
@WebServlet(name = "AccountController", urlPatterns =
    {"/AccountController"})
public class AccountController extends HttpServlet {
    @EJB
    private AccountServiceRemote accountService;

    @Override
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws
        ServletException, IOException {
      response.setContentType("text/html;charset=UTF-8");
      try (PrintWriter out = response.getWriter()) {
        String firstName = request.getParameter("firstName")
            ;
        String lastName = request.getParameter("lastName");

        Customer customer = accountService.createAccount
            (firstName, lastName);
        out.println("Customer["+customer.getFirstName()+
            "] : " + customer.getId());
      }
    }
}
```

## Distributed Session Objects (3 of 3)

```
<!DOCTYPE html>
<html>
    <head>
        <title>Create Customer</title>
        <meta name="viewport" content="width=device-width,
            initial-scale=1.0">
    </head>
    <body>
        <form action="AccountController" method="post" >
            <input type="text" name="firstName" placeholder=
                "First Name" />
            <input type="text" name="lastName" placeholder="
                Last Name"/>
            <input type="submit" value="Create"/>
        </form>
    </body>
</html>
```

## Organization in Classes

- We can put more than one script in a class
    - If they are stateless, they can be static methods
    - Group by their purpose
- Command Pattern
    - One class for each script implementing a common interface
    - Each has an "execute" method containing the logic

## When to Use it

- Great when there is a small amount of business logic
- As logic gets more complicated, shared functionality can lead to complexity or duplicated code which may need to be refactored into a Domain Model (116)
- Advantages:
  - pretty simple
  - procedural, so the old folks are comfortable
  - Works nicely with either a Row Data Gateway (152) or Table Data Gateway (144)
  - Maps well to transaction boundaries
- Disadvantages
  - duplication of code when multiple user requests require similar actions
  - Even if we break those into subroutines, we end up with a lot of unorganized routines

## Table Module (125)

- A single instance that handles the business logic for all rows in a database table or view.
- Half way between the structure of the data source and domain model
    - Like Domain Model, puts data and behavior together
    - Matches the structure of the underlying DB (one class per DB table)
- Every method requires a parameter that is the ID of the individual it should use
- Usually uses Record Set (508) and gives you a method-based interface to manipulate it
- Often have to use multiple Table Modules with multiple Record Sets to get something done.
- A Table Module can be an instantiable class or just a bunch of static methods.

## How do we create it if it is instantiable?

- Factory method based on a query or a view
- If we need multiple data sets, use a Table Data Gateway (144) for each data set
- This works really nicely when the GUI environment is designed to use results of queries organized as Record Sets (508) like COM and .NET

## Table Module Example (1 of 2)

```php
<?php
class StatisticsModule{
    public function __construct(array $rows){
        $this->_rows = $rows;
    }

    public function getMostPopularBrowser(){
        $browsers = array();
        foreach ($this->_rows as $row) {
            if (!isset($browsers[$row['browser']])) {
                $browsers[$row['browser']] = 0;
            }
            $browsers[$row['browser']]++;
        }
        arsort($browsers);
        reset($browsers);
        return current(array_keys($browsers));
    }

    public function isResolutionUsed($resolution, $margin = 0.1){
        $visitors = 0;
        foreach ($this->_rows as $row) {
            if ($row['resolution'] == $resolution) {
                $visitors++;
            }
        }
        return $visitors / count($this->_rows) > $margin;
    }
}
```

## Transaction Scripts Example (2 of 2)

```
    function create_row($browser, $resolution, $page)
    {
        return array(
        'browser' => $browser,
        'resolution' => $resolution,
        'page' => $page
        );
    }
}
```

## Transaction Scripts (110)

## Transaction Scripts (110)

## Transaction Scripts (110)

## Transaction Scripts (110)

## Transaction Scripts (110)