Using Design Patterns with GRASP

**General Responsibility Assignment Software Patterns**

The “patterns” provide a representation of nine basic principles that form a foundation for designing object-oriented systems.

- Creator
- Information Expert
- Low Coupling

- Controller
- High Cohesion
- Polymorphism

- Pure Fabrication
- Indirection
- Protected Variations
Responsibility-Driven Design

Responsibilities of an Object include two types: *Knowing* and *Doing*

**Doing responsibilities of an object include:**

- Doing something itself, such as creating an object or doing a calculation
- Initiating action in other objects
- Controlling and coordinating activities in other objects

**Knowing responsibilities of an object include:**

- Knowing about private encapsulated data (know thyself, presume not God to scan)
- Knowing about related objects
- Knowing about things it can derive or calculate

**Responsibilities are an abstraction – methods fulfill responsibilities**

Responsibilities are implemented by means of methods that either act alone or collaborate with other methods and objects.
Example of RDD – Monopoly Game
Example of RDD – Monopoly Game

Domain Model (Analyze Phase)
Monopoly Game Example

Who creates the Square object?

The **Creator** pattern

<table>
<thead>
<tr>
<th>Name:</th>
<th><strong>Creator</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem:</td>
<td>Who creates an object A?</td>
</tr>
<tr>
<td>Solution:</td>
<td>Assign class B the responsibility to create an instance of class A if one of these is true</td>
</tr>
<tr>
<td><strong>This can be viewed as advice</strong></td>
<td>+ B “contains” or completely aggregates A</td>
</tr>
<tr>
<td></td>
<td>+ B records A</td>
</tr>
<tr>
<td></td>
<td>+ B closely uses A</td>
</tr>
<tr>
<td></td>
<td>+ B has the initializing data for A</td>
</tr>
</tbody>
</table>
Implementation of the Creator Pattern in Monopoly

Applying the Creator pattern in Static Model

Dynamic Model – illustrates Creator Pattern
Monopoly Game Example

Suppose objects need to be able to reference a particular Square, given its name. **Who knows about a Square object, given a key?**

**Information Expert pattern**

Name: Information Expert  
Problem: What is a basic principle by which to assign responsibilities to an object  
Solution: Assign a responsibility to the class that has the information needed to respond to it.

The player Marker needs to find the square to which it is to move and the options pertaining to that square.

The Board aggregates all of the Squares, so the Board has the Information needed to fulfill this responsibility.
Make Board Information Expert

:s = getSquare(name)

s = get(name) : Square
Alternative Design

s = getSquare(name)

sqs = getAllSquares(name)

s = get (name) : Square

Poor Design!

More coupling if Piece has getSquare()
Low Coupling

One of the major GRASP principles is Low Coupling.

Coupling is a measure of how strongly one object is connected to, has knowledge of, or depends upon other objects. An object A that calls on the operations of object B has coupling to B’s services. When object B changes, object A may be affected.

Name: Low Coupling
Problem: How to reduce the impact of change?
Solution: Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.
Key Point: Expert Supports Low Coupling

To return to the motivation for Information Expert: it guides us to a choice that supports Low Coupling. Expert asks us to find the object that has most of the information required for the responsibility (e.g., Board) and assign responsibility there.

If we put the responsibility anywhere else (e.g., Dog), the overall coupling will be higher because more information or objects must be shared away from their original source or home, as the squares in the Map collection had to be shared with the Dog, away from their home in the Board.
Controller

A simple layered architecture has a user interface layer (UI) and a domain layer. Actors, such as the human player in Monopoly, generate UI events (such as clicking a button with a mouse to play a game or make a move). The UI software objects (such as a JFrame window and a JButton) must process the event and cause the game to play. When objects in the UI layer pick up an event, they must delegate the request to an object in the domain layer.

What first object beyond the UI layer should receive the message from the UI layer?

Name: Controller

Problem: What first object beyond the UI layer receives and coordinates a System Operation?

Solution: Assign the responsibility to an object representing one of these choices:

+ Represents the overall “system” – a root object
+ Represents a use case scenario within which the system operation occurs.
The Controller Pattern

Monopoly: JFrame

Monopoly Game

UI Layer

Domain Layer

JButton

Press to play

actionPerformed

playGame

The Controller Pattern
High Cohesion

High Cohesion is an underlying Design Objective

Cohesion measures how functionally related the operations of a software element are. It also measures how much work an object is doing. Note low cohesion and bad coupling often go together.

Name: High Cohesion

Problem: How to keep objects focused, understandable, and manageable, and, as a side effect, support Low Coupling

Solution: Assign responsibilities so that cohesion remains high. Use this criteria to evaluate alternatives.
Contrasting Levels of Cohesion

**Poor (low) Cohesion in the MonopolyGame object**

- Invocation of `playGame`
- Method calls `doA`, `doB`, and `doC`

**Better Design**

- Invocation of `playGame`
- Method calls `doA` and `doB`
First Iteration of the Monopoly Game

In Iteration 1 – there is no winner. The rules of the game are not yet incorporated into the design. Iteration 1 is merely concerned with the mechanics of having a player move a piece around the Board, landing on one of the 40 Squares each turn.

Definition –

- **turn** – a player rolling the dice and moving one piece
- **round** – all players taking one turn

The game loop algorithm:

```plaintext
for N rounds
  for each player p
    p takes a turn
```
Assign Responsibility for Controlling the Game Loop

<table>
<thead>
<tr>
<th>Information Needed</th>
<th>Who Has the Information?</th>
</tr>
</thead>
<tbody>
<tr>
<td>The current round count</td>
<td>No object has it yet, but by LRG*, assigning this to the <em>MonopolyGame</em> object is justifiable</td>
</tr>
<tr>
<td>All the players (so that each can be used in taking a turn)</td>
<td>From examination of the domain model, <em>MonopolyGame</em> is a good candidate.</td>
</tr>
</tbody>
</table>

*LRG – low representational gap. Lower the gap between our mental and software models.*
Controlling the Game Loop

:MonopolyGame

playGame

loop [rndCnt < N]

playRound
# Who Takes a Turn?

<table>
<thead>
<tr>
<th>Information Needed</th>
<th>Who Has the Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current location of the player (to know the starting point of a move)</td>
<td>We observe from the domain model, a Piece knows its <em>Square</em> and a <em>Player</em> knows its <em>Piece</em>. Therefore, a <em>Player</em> software object could know its location by LRG.</td>
</tr>
<tr>
<td>The two Die objects (to roll them and calculate their total)</td>
<td>The domain model indicates that <em>MonopolyGame</em> is a candidate since we think of the dice as being part of the game.</td>
</tr>
<tr>
<td>All the squares – the square organization (to be able to move to the correct new square)</td>
<td>By LRG, <em>Board</em> is a good candidate.</td>
</tr>
</tbody>
</table>
Taking a Turn

Taking a turn means:

• Calculating a random number between 2 and 12
• Determining the location of the new square
• Moving the player’s piece from the old location to the new square.

Calculating a new face value means changing information in Die, so by Expert, Die should be able to roll itself (generate a random number) and answer its face value.

The new square location problem: Since the Board knows all its Squares, it should be responsible for finding a new square location, given an old square location and some offset (the dice total).

The piece movement problem: By LRG it is reasonable for a Player to know its Piece, and a Piece its Square location (or even for a Player to directly know its Square location). By Expert, a Piece will set its new location, but may receive that new location from its Player.
Final Design of the System Operation playGame (Iter. 1)

```
:MonopolyGame

playGame

loop

playRound

loop

takeTurn

:Player

dice[i] : Die

:Board

:Piece

roll

fv = getFaceValue

oldLoc = getLocation() : Square

newLoc = getSquare(oldLoc, fvTot) : Square

setLocation (newLoc)
```
Visibility

Make a list of the messages (with parameters) and the classes of objects that send and receive them.

<table>
<thead>
<tr>
<th>Message</th>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>playGame</td>
<td>UIlevel</td>
<td>MonopolyGame</td>
</tr>
<tr>
<td>playRound</td>
<td>MonopolyGame</td>
<td>MonopolyGame</td>
</tr>
<tr>
<td>takeTurn</td>
<td>MonopolyGame</td>
<td>Player</td>
</tr>
<tr>
<td>roll</td>
<td>Player</td>
<td>Die</td>
</tr>
<tr>
<td>getFaceValue:int</td>
<td>Player</td>
<td>Die</td>
</tr>
<tr>
<td>getLocation:Square</td>
<td>Player</td>
<td>Piece</td>
</tr>
<tr>
<td>getSquar(Square, int):Square</td>
<td>Player</td>
<td>Board</td>
</tr>
<tr>
<td>setLocation(Square)</td>
<td>Player</td>
<td>Piece</td>
</tr>
<tr>
<td>create</td>
<td>Board</td>
<td>Square</td>
</tr>
</tbody>
</table>
Visibility

From the previous Table we learn what classes must be visible to each other to implement the System Operation: *playGame*

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
<th>Visibility Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MonopolyGame</td>
<td>Player</td>
<td>Attribute (fixed)</td>
</tr>
<tr>
<td>Player</td>
<td>Die</td>
<td>Attribute (fixed)</td>
</tr>
<tr>
<td>Player</td>
<td>Piece</td>
<td>Attribute (fixed)</td>
</tr>
<tr>
<td>Player</td>
<td>Board</td>
<td>Attribute (fixed)</td>
</tr>
<tr>
<td>Board</td>
<td>Square</td>
<td>Attribute (fixed)</td>
</tr>
<tr>
<td>Piece</td>
<td>Square</td>
<td>Attribute (Transient)</td>
</tr>
</tbody>
</table>

Four kinds of visibilities for object B to send a message to object A:

A is global to B

A is part of B

A is a parameter to some operation of B

A is a locally object in some operation of B
public class Square {
    private String name;
    private Square nextSquare;
    private int index;
    public Square (String name, int index) {
        this.name = name;
        this.index = index;
    }
    public void setNextSquare (Square s) {
        nextSquare = s;
    }
    public Square getNextSquare () {
        return nextSquare;
    }
    public String getName () {return name; }
    public int getIndex () {return index; }
}

public class Piece {
    private Square location;
    public Piece (Square location) {
        this.location = location;
    }
    public Square getLocation () {
        return location;
    }
    public void setLocation (Square location) {
        this.location = location;
    }
}
public class Die {
    public static final int MAX = 6;
    private int faceValue;
    public Die() { roll(); }
    public void roll() {
        faceValue = (int) (Math.random() * Max) + 1;
    }
    public int getFaceValue() { return faceValue; }
}

public class Board {
    private static final int SIZE = 40;
    private List squares = new ArrayList(SIZE);
    public Board() {
        buildSquares();
        linkSquares();
    }
    public Square getStartSquare() {
        return (Square) squares.get(0);
    }
    public Square getSquare(Square start, int dist) {
        int endIndex = (start.getIndex() + dist) % SIZE;
        return (Square) squares.get(endIndex);
    }
    public void buildSquares() {
        for (int i = 1; i <= SIZE; i++)
            build(i);
    }
    public void build(int i) {
        Square s = new Square(“Square “ + i, i – 1);
    }
    public void linkSquares() {
        for (int i = 0; i < SIZE; i++)
            link(i);
    }
    public void link(int i) {
        Square current = (Square) squares.get(i);
        Square next = (Square) squares.get((i+1)%SIZE));
        current.setNextSquare(next);
    }
}
public class Player {
    private String name;
    private Piece marker;
    private Board board;
    private Die[] dice;

    public Player(String name, Die[] dice, Board b) {
        this.name = name;
        this.dice = dice;
        this.board = b;
        marker = new Piece(board.getStartSquare());
    }

    public Square getLocation() {
        return marker.getLocation();
    }

    public String getName() { return name; }

    public void takeTurn() {
        //roll dice
        int rollTotal = 0;
        for (int i = 0; i < dice.length; i++) {
            dice[i].roll();
            rollTotal += dice[i].getFaceValue();
        }
        Square newLoc = board.getSquare(marker.getLocation(), rollTotal);
        marker.setLocation(newLoc);
    }
}
public class MonopolyGame {
    private static final int ROUNDS_TOTAL = 20;
    private static final int PLAYERS_TOTAL = 2;
    private List players = new ArrayList( PLAYERS_TOTAL);
    private Board board = new Board( );
    private Die[] dice = { new Die( ), new Die( ) };
    public MonopolyGame( ) {
        Player p;
        p = new Player( "Dog", dice, board);
        players.add(p);
        p = new Player( "Car", dice, board);
        players.add(p);
    }
    public List getPlayers( ) {
        return players;
    }
    public void playRound( ) {
        for ( Iterator itr = players.iterator(); itr.hasNext( ); ) {
            Player player = (Player) iter.next( );
            Player player = (Player) itr.next( );
            player.takeTurn( );
        }
    }
    public void playGame( ) {
        for ( int i = 0; i < ROUNDS_TOTAL; i++ )
            playRound( );
    }
}
Principle of Separation of Command and Query

Given two solutions for obtaining the outcome of a roll of a Die:

//style # 1 -- used in the previous solution
public void roll() {
    faceValue = (int) (Math.random() * Max) + 1;
}
public int getFaceValue() { return faceValue; }

//style # 2 – do everything at once
public int roll() {
    faceValue = (int) (Math.random() * Max) + 1;
    return faceValue;
}

Command-Query Separation Principle -- Every method should be:
• A command method that performs an action, often has side effects such as changing the state of objects, and is void
• A query that returns data to the calloer and has no side effects. But not both!
Where do we go from here? 2<sup>nd</sup> Iteration

Name: **Polymorphism**

Problem: How to handle alternatives based on type.
Pluggable software components -- how can you replace one server component with another without affecting the client?

Solution: When related alternatives or behaviors vary by type (class), assign responsibility for the behavior – using polymorphic operations – to the types for which the behavior varies. In this context, polymorphism means giving the same name to similar or related services.

Now we begin to add the “business rules” to the monopoly game.
Designing for different Square actions

- Player
  - location
  - Square {abstract}
    - landedOn {abstract}
      - PropertySquare
        - landedOn
      - GoSquare
        - landedOn
      - RegularSquare
        - landedOn
      - IncomeTaxSquare
        - landedOn
      - ...


Applying Polymorphism

p:Player
dice[i]:Die
:Board
loc:Square

takeTurn

roll
fv = getFaceValue : int
loc = getSquare(currentLoc, fvTot) : Square
landedOn(p)

By Expert

By Polymorphism
The Polymorphic Cases

Polymorphic method `landedOn` directed to the (abstract) base class (interface) and is implemented by each of the concrete subclasses.
<table>
<thead>
<tr>
<th>Name:</th>
<th>Pure Fabrication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem:</td>
<td>What object should have responsibility when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for example) are not appropriate? Sometimes assigning responsibilities only to domain layer software classes leads to problems like poor cohesion or coupling, or low reuse potential.</td>
</tr>
<tr>
<td>Solution:</td>
<td>Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a domain concept.</td>
</tr>
<tr>
<td>Example:</td>
<td>Rolling the dice in a Monopoly game – Dice are used in many games and putting the rolling and summing responsibilities in Player makes it impossible to generalize this service. Also, it is not now possible to simply ask for the current dice total without rolling again.</td>
</tr>
</tbody>
</table>
Pure Fabrication

Use a *Cup* to hold the dice, roll them, and know their total. It can be reused in many different applications where dice are involved.