A defense of so-called anemic domain models

Luís Marques @ D-Lang-Silicon-Valley Meetup
January 28, 2016

Object-Oriented Programming

- What is the *essence* of OOP? There's no widely accepted answer.

- Therefore, the definitions of OOP tend to focus on the superficial feature: objects.

- Objects contain both data and the code that operates on that data.

Object-Oriented Design

- OK, OOP is kind of about programming with objects. What objects will you program?

- That is the task of object-oriented design: defining the jumble of objects that your software will have, and how they communicate.

- Based on your problem, which objects should you define?

    - Make a list of nouns and verbs?

Domain-Driven Design

- **Domain layer / Domain models**
    - “Responsible for representing concepts of the business, information about the business situation, and business rules.” [1]
    - “An object model of the domain that incorporates both behavior and data.” [3]
        - Their objects are called entities.
- **Value objects**
- **Service layer / Services**
    - “Defines an application’s boundary with a layer of services that establishes a set of available operations and coordinates the application’s response in each operation.” [4]

RDM vs ADM

- Rich Domain Model (RDM) [1][2][3]
  - Domain entities encapsulate all business logic and data.
- Anemic Domain Model (ADM) [1][2]
  - Behaviour-free classes containing business data required to model the domain.
  - Claimed as anti-pattern. [2]

RDM

- All domain rules are implemented via domain models.
- The domain service layer is extremely thin or non-existent.
- Domain entities are capable of completely enforcing their invariants.
- Standard conclusion: good object-oriented design.

ADM

- ADM model classes typically contain little or no validation of the data as conforming to business rules

- Business logic is implemented by a domain service layer.
    - Objects which process the domain models as dictated by business rules.

- The domain entities cannot enforce their own invariants.

- Standard conclusion:
  not OOP; procedural business logic == bad.

Is ADM really an anti-pattern? [5]

Anti-Pattern?

- Local vs global invariants
- SOLID principles
- Testability (not covered today)

Local vs Global

- The ability to enforce *local* invariants in domain entities is good. But is that the end-all of OO design?

- In ADM the domain logic is implemented in dedicated domain service classes.

    - Each class covers a cohesive set of domain rules, which might affect multiple domain entities.

    - A good place to enforce *global* invariants. Might also be able to enforce localized invariants.

- Overall, the implementation is more flexible and maintainable.

SOLID principles

- **S**ingle responsibility principle
- **O**pen/closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

Single responsibility principle

- “Every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.”
  - Wikipedia

- “A class should have only one reason to change”
  — Robert C. Martin

ADM Respects the SRP

- Domain entities care only about themselves. They don't go spelunking around the domain model graph to do their jobs.
    - When they do enforce some invariants, those invariants pertain to themselves only.
- It's easy to implement domain service classes with only one responsibility
    - The global invariants they enforce are associated with their responsibility.

Example?

Monopoly

- “My favourite interview question” — Reginald Braithwaite [6]

- “How might you design a program that lets people play Monopoly with each other over the internet?”

Monopoly:
Nouns and Verbs?

- Nouns
  - Game, Board, Player, Property, etc.
- Verbs
  - Buy Property, Pay Rent, Roll Dice, etc.

RDM Monopoly

```
class Player
{
    Money cash;
    Property[] ownedProperties;
    // (...)

    bool canBuyProperty(Property property)
    {
        return cash >= property.printedPrice;
    }

    void buyProperty(Property property)
    {
        property.owner = this;
        cash -= property.printedPrice;
    }
}

class Property
{
    string name;
    Money printedPrice;
    Player owner;
    // (...)
}
```

The Challenge of Monopoly

- Where do the rules live?
- “In a noun-oriented design, the rules are smooshed and smeared across the design, because every single object is responsible for knowing everything about everything that it can ‘do’. All the verbs are glued to the nouns as methods.” [6]
- Walter Bright touches upon this topic in his article about UFCS [9].

Example

- If a player owns Baltic Avenue, can the player add a house to it?
  - Can she afford it?
  - Is there a house in the bank?
  - Is it either the player's turn or between turns?
  - Does the property already have four houses?
  - Is Baltic Avenue mortgaged?
  - What if Mediterranean Avenue (same group) is mortgaged?
  - What if Baltic Avenue has one house but Mediterranean Avenue has none?

ADM Monopoly

```
class OfficialRulesHousePurchaseValidator : HousePurchaseValidator
{
    Game game;
    Player purchaser;
    Property property;

    bool canBuyHouse()
    {
        return game.currentPlayer == purchaser &&
               player.cash >= property.houseCost &&
               (…);
    }
}

class BuyHouse : Action
{
    Property property;
    HousePurchaseValidator validator; // we depend on the abstract validator interface

    override void perform()
    {
        enforce(validator.canBuyHouse);
        (…)
    }
}
```

Not convinced?

Vetted Example?

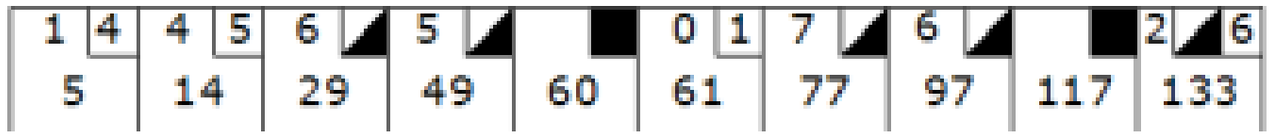
- Are the previous examples just a straw man?
- Can we find an example that was carefully considered, simple and yet interesting?
- Reviewed by hundreds of people?

Bowling for Objects

- The Bowling Game!
- “[It] has been done so many times, by so many people.” [7]
- “I’ve been doing it for probably ten years.” [7]

— Robert C. Martin

The Bowling Game Kata



- “The game consists of 10 frames as shown above. In each frame the player has two opportunities to knock down 10 pins. The score for the frame is the total number of pins knocked down, plus bonuses for strikes and spares.”

- “A spare is when the player knocks down all 10 pins in two tries. The bonus for that frame is the number of pins knocked down by the next roll. So in frame 3 above, the score is 10 (the total number knocked down) plus a bonus of 5 (the number of pins knocked down on the next roll.)”

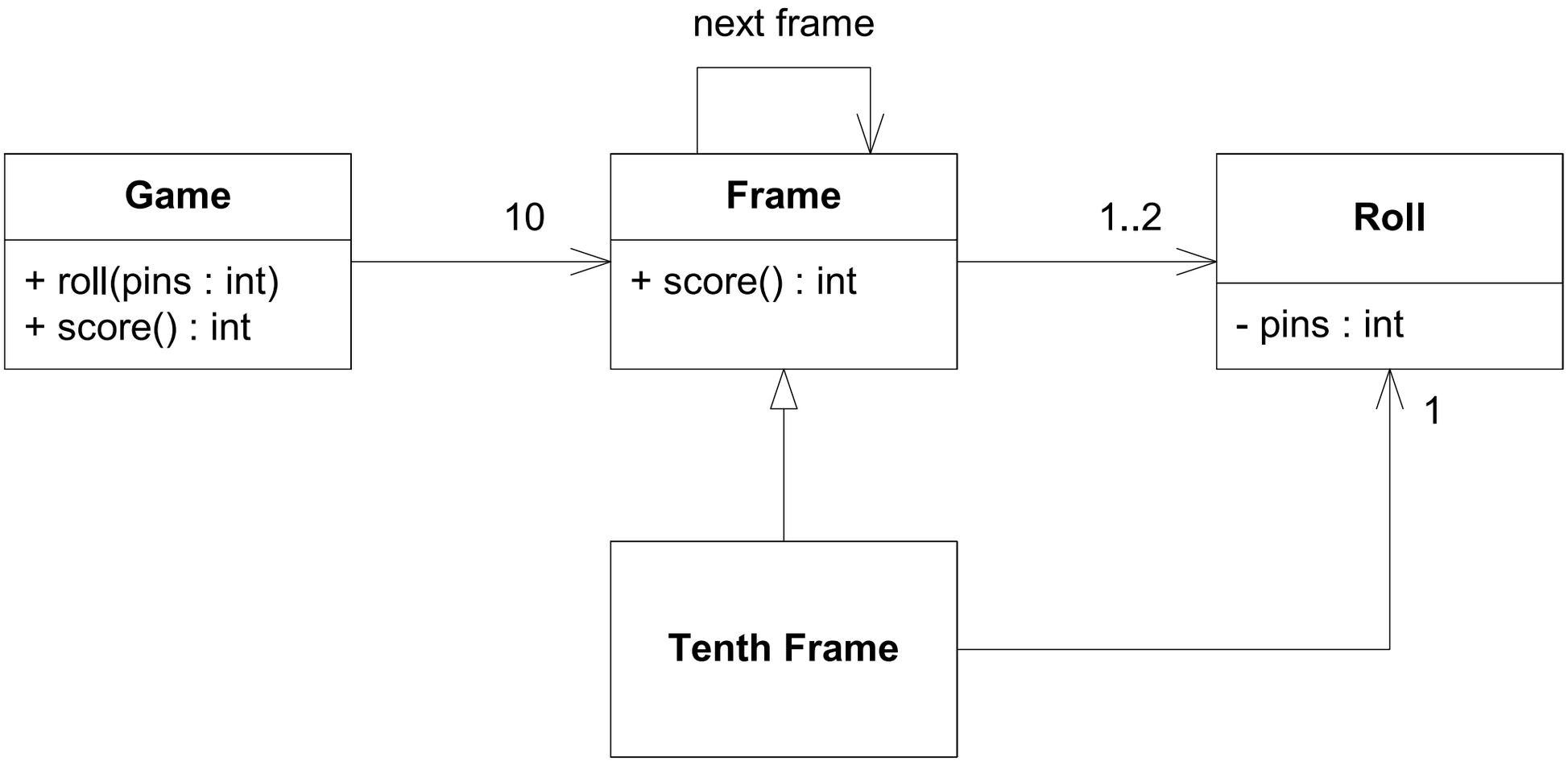
- “A strike is when the player knocks down all 10 pins on his first try. The bonus for that frame is the value of the next two balls rolled.”

- “In the tenth frame a player who rolls a spare or strike is allowed to roll the extra balls to complete the frame. However no more than three balls can be rolled in tenth frame.”

The Bowling Game Kata

- Write a class named “Game” that has two methods:
  - roll(pins : int) is called each time the player rolls a ball. The argument is the number of pins knocked down.
  - score() : int is called only at the very end of the game. It returns the total score for that game.
- The handling of invalid inputs is not implemented in the example.

Sketch Bowling Design



The First Test

```
import junit.framework.TestCase;

public class BowlingGameTest
  extends TestCase
{
  public void testGutterGame()
  {
    Game g = new Game();
    for (int i=0; i<20; i++)
      g.roll(0);
    assertEquals(0, g.score());
  }
}
```

```
public class Game {
  public void roll(int pins) {
  }

  public int score() {
    return 0;
  }
}
```

The Second Test

```
(…)
protected void setUp()
{
  g = new Game();
}

public void testGutterGame(){
  rollMany(20, 0);
  assertEquals(0, g.score());
}

private void rollMany(int n, int pins)
{
  for (int i = 0; i < n; i++)
    g.roll(pins);
}

public void testAllOnes()
{
  rollMany(20,1);
  assertEquals(20, g.score());
}
```

```
public class Game
{
  private int score = 0;

  public void roll(int pins)
  {
    score += pins;
  }

  public int score()
  {
    return score;
  }
}
```

etc…

The Final Bowling Design

```
...
public void testGutterGame() throws Exception {
  rollMany(20, 0);
  assertEquals(0, g.score());
}

public void testAllOnes() throws Exception {
  rollMany(20,1);
  assertEquals(20, g.score());
}

public void testOneSpare() throws Exception {
  rollSpare();
  g.roll(3);
  rollMany(17,0);
  assertEquals(16,g.score());
  }

  public void testOneStrike() throws Exception {
    rollStrike();
    g.roll(3);
    g.roll(4);
    rollMany(16, 0);
    assertEquals(24, g.score());
  }

  public void testPerfectGame() throws Exception {
    rollMany(12,10);
    assertEquals(300, g.score());
  }

  private void rollStrike() {
    g.roll(10);
  }

  private void rollSpare() {
    g.roll(5);
    g.roll(5);
  }
}
```

```
public class Game {
  private int rolls[] = new int[21];
  private int currentRoll = 0;

  public void roll(int pins) {
    rolls[currentRoll++] = pins;
  }

  public int score() {
    int score = 0;
    int frameIndex = 0;
    for (int frame = 0; frame < 10; frame++) {
      if (isStrike(frameIndex)) {
        score += 10 + strikeBonus(frameIndex);
        frameIndex++;
      } else if (isSpare(frameIndex)) {
        score += 10 + spareBonus(frameIndex);
        frameIndex += 2;
      } else {
        score += sumOfBallsInFrame(frameIndex);
        frameIndex += 2;
      }
    }
    return score;
  }

  private boolean isStrike(int frameIndex) {
    return rolls[frameIndex] == 10;
  }

  private int sumOfBallsInFrame(int frameIndex) {
    return rolls[frameIndex] + rolls[frameIndex+1];
  }

  private int spareBonus(int frameIndex) {
    return rolls[frameIndex+2];
  }

  private int strikeBonus(int frameIndex) {
    return rolls[frameIndex+1] + rolls[frameIndex+2];
  }

  private boolean isSpare(int frameIndex) {
    return rolls[frameIndex]+rolls[frameIndex+1] == 10;
  }
}
```

Is this a SRP Game?

- What are the responsibilities of a Game object?
  - “Being” a Bowling Game?
  - Validating inputs and maintaining a valid sequence of rolls?
  - Knowing if a frame is a strike or a spare?
  - Computing the score for a scoresheet?
  - Everything that might be related to the game play?

“Anemic” Bowling

- A game object has a single responsibility: maintaining the state of a game (the rolls).
    - It can do some local enforcement of business logic (e.g., a roll never knocks more than 10 pins)
- A scoring service receives the relevant state from a game object and produces a score.
    - The game looks much more like a “bag of getters and setters”, compared with a richer model.

ADM Bowling

```
class Game
{
  void roll(int pins) {
    …
  }

  auto scoresheet()
  {
    return cast(RandomAccessFinite!int[]) inputRangeObject(_scoresheet)
  }

  int score()
  {
    return scoresheet.score
  }

private
   int[][] _scoresheet;
}
```

ADM Bowling

```
int score(RandomAccessFinite!(RandomAccessFinite!int) scoresheet)
{
    …
}
```

- How do you implement the scoring service?
- You can continue decomposing.
- Helper “services” for scoring a single frame, determining if a frame is a strike, etc.
- These “services” can be reused outside the game entity and the scoring service.

ADM Bowling

```
enum isFrame(T) =
    isRandomAccessRange!T && isNumeric!(ElementType!T);

enum isScoresheet(T) =
    isRandomAccessRange!T && isFrame!(ElementType!T);

bool isValidFrame(Frame)(Frame frame, bool last = false)
    if(isFrame!Frame)
{
    if(frame.length > (last ? 3 : 2))
        return false;

    if(frame.any!"a > 10")
        return false;

    if(frame.sum > (last ? 30 : 10))
        return false;

    return true;
}
```

ADM Bowling

- Because we decomposed further we tested further:

```
unittest
{
    assert((cast(int[])[]).isValidFrame);
    assert([0].isValidFrame);
    assert([0,7].isValidFrame);
    assert(![7,4].isValidFrame);
    assert([1,2,3].isValidFrame(true));
    assert([10,9,1].isValidFrame(true));
}
```

ADM Bowling

- To get the score for a frame we need access to more than the rolls in that frame.

```
int firstFrameScore(Scoresheet)(Scoresheet sheet)
    if(isScoresheet!Scoresheet)
{
    auto frame = sheet.front;
    int points = frame.sum;
    bool isLast = sheet.length == 1;

    if(frame.isStrike(isLast))
    {
        points += sheet.save.dropOne.joiner.take(2).sum;
    }
    else if(frame.isSpare)
    {
        points += sheet.save.dropOne.joiner.take(1).sum;
    }

    return points;
}
```

ADM Bowling

- Then we bring it all together:

```
int score(Scoresheet)(Scoresheet sheet)
    if(isScoresheet!Scoresheet)
{
    int score = 0;

    while(!sheet.empty)
    {
        score += sheet.firstFrameScore;
        sheet.popFront();
    }

    return score;
}
```

ADM Bowling

- And we apply the same tests as the original kata:

```
unittest
{
    enum gutterGame = [0, 0].repeat(10);
    assert(gutterGame.score == 0);

    enum allOnes = [1, 1].repeat(10);
    assert(allOnes.score == 20);

    enum oneSpare = [[5, 5], [3, 0]] ~ [0, 0].repeat(8).array;
    assert(oneSpare.score == 16);

    enum oneStrike = [[10], [3, 4]] ~ [0, 0].repeat(8).array;
    assert(oneStrike.score == 24);

    enum perfectGame = [10].repeat(9).array ~ [10, 10, 10];
    assert(perfectGame.score == 300);
}
```

- No need for **`rollSpare()`**, **`rollStrike()`**, etc.

Why not ADM Bowling?

- Why didn’t anyone consider doing it like this?
  - In Java you don’t have ranges / STL-style iterators
  - You’d instead pass the internal data. “Not OOP!”
  - Or… you’d forward the rolls and maintain the state redundantly?

Alternative Bowling Design (PPP)

```
public class Game
{
  public int score()
  {
    return scoreForFrame(itsCurrentFrame);
  }

  public void add(int pins)
  {
    itsScorer.addThrow(pins);
    adjustCurrentFrame(pins);
  }

  private void adjustCurrentFrame(int pins)
  {
    if (lastBallInFrame(pins))
      advanceFrame();
    else
      firstThrowInFrame = false;
  }

  private boolean lastBallInFrame(int pins)
  {
    return strike(pins) || !firstThrowInFrame;
  }

  private boolean strike(int pins)
  {
    return (firstThrowInFrame && pins == 10);
  }

  private void advanceFrame()
  {
    itsCurrentFrame = Math.min(10, itsCurrentFrame + 1);
  }

  public int scoreForFrame(int theFrame)
  {
    return itsScorer.scoreForFrame(theFrame);
  }

  private int itsCurrentFrame = 0;
  private boolean firstThrowInFrame = true;
  private Scorer itsScorer = new Scorer();
}
```

```
public class Scorer
{
  public void addThrow(int pins)
  {
    itsThrows[itsCurrentThrow++] = pins;
  }

  public int scoreForFrame(int theFrame)
  {
    ball = 0;
    int score=0;
    for (int currentFrame = 0;
         currentFrame < theFrame;
         currentFrame++)
    {
      if (strike())
      {
        score += 10 + nextTwoBallsForStrike();
        ball++;
      }
      else if ( spare() )
      {
        score += 10 + nextBallForSpare();
        ball+=2;
      }
      else
      {
        score += twoBallsInFrame();
        ball+=2;
      }
    }

    return score;
  }

  private boolean strike()
  {
    return itsThrows[ball] == 10;
  }
  private boolean spare() { … }
  private int nextTwoBallsForStrike() { … }
  private int nextBallForSpare() { … }
  private int twoBallsInFrame() { … }

  private int ball;
  private int[] itsThrows = new int[21];
  private int itsCurrentThrow = 0;
}
```

Conclusion

- There's a thin line between fine grained decomposition and an ADM.

- Better SRP enforcement / increased decomposition implies less local business model rules to enforce.

- We still enforce the other business rules at the level of the services.

    - We have to be careful not to duplicate such validation across services.

- D makes it easy to devise simple and reusable domain services: Templates, CTFE, UFCS, ranges and algorithms, etc.

References

- [1] Evans, Eric. Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional, 2004.
- [2] Fowler, Martin. Anaemic Domain Model. <http://www.martinfowler.com/bliki/AnemicDomainModel.html>
- [3] Fowler, Martin. Domain Model. <http://martinfowler.com/eaaCatalog/domainModel.html>
- [4] Fowler, Martin. Service Layer. <http://martinfowler.com/eaaCatalog/serviceLayer.html>
- [5] “S”. The Anaemic Domain Model is no Anti-Pattern, it’s a SOLID design. <https://blog.inf.ed.ac.uk/sapm/2014/02/04/the-anaemic-domain-model-is-no-anti-pattern-its-a-solid-design/>
- [6] Braithwaite, Reginald. My favourite interview question. <http://weblog.raganwald.com/2006/06/my-favourite-interview-question.html>
- [7] Martin, Robert C. Clean Code, episode 6, part 2. <http://cleancoders.com/episode/clean-code-episode-6-p2/show>
- [8] Alexandrescu, Andrei. Three Cool Things about D. <https://www.youtube.com/watch?v=FdpaBHyQNco>
- [9] Bright, Walter. Uniform Function Call Syntax. <http://www.drdobbs.com/cpp/uniform-function-call-syntax/232700394>