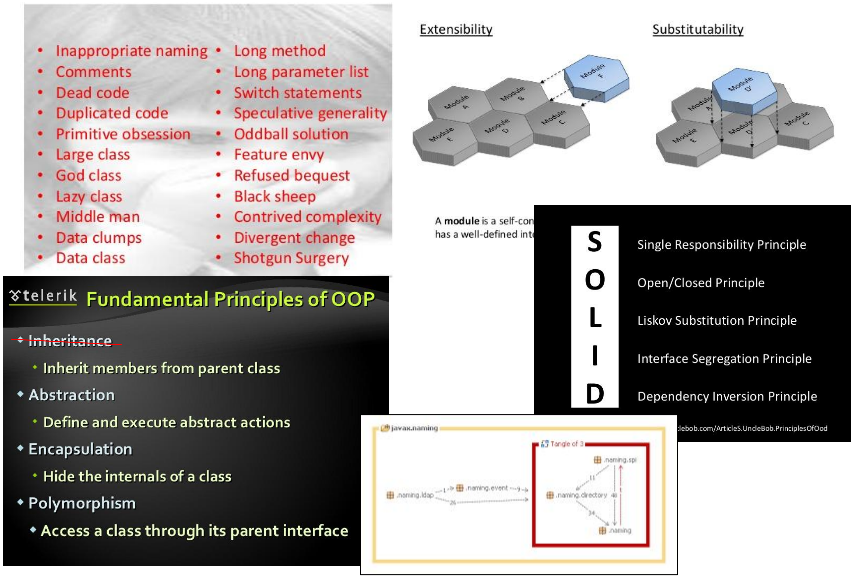
# Model Design and Implementation

Slide Series #4

# General Principles



- Inappropriate naming
- Comments
- Dead code
- Duplicated code
- Primitive obsession
- Large class
- God class
- Lazy class
- Middle man
- Data clumps
- Data class
- Long method
- Long parameter list
- Switch statements
- Speculative generality
- Oddball solution
- Feature envy
- Refused bequest
- Black sheep
- Contrived complexity
- Divergent change
- Shotgun Surgery

telerik **Fundamental Principles of OOP**

- ~~Inheritance~~
  - Inherit members from parent class
- Abstraction
  - Define and execute abstract actions
- Encapsulation
  - Hide the internals of a class
- Polymorphism
  - Access a class through its parent interface

A **module** is a self-con
has a well-defined int

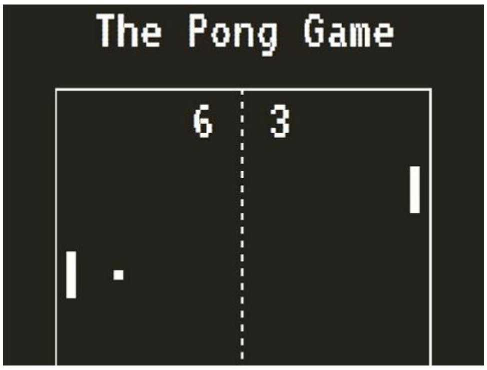
S Single Responsibility Principle
O Open/Closed Principle
L Liskov Substitution Principle
I Interface Segregation Principle
D Dependency Inversion Principle

clebob.com/ArticleS.UncleBob.PrinciplesOfOod

There are quite a few design principles and best practices

- Even more important at application design upcomming.

# Responsibilities in Models



The Pong game has a Ball and Two paddles

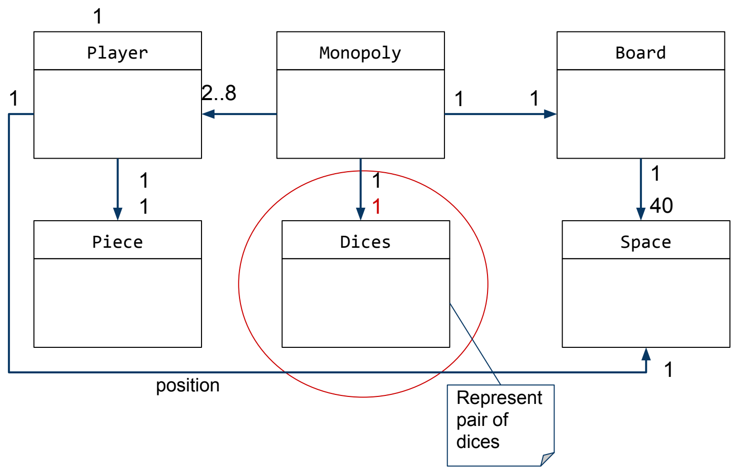
- Which will check for collision?
- .. or will (should) someone else check …?

Summary: There are levels in the model …

- … some objects are at a higher level, handling objects at lower level!
- Paddle and Ball are at the “same” level, so something higher up should handle collisions

How about MP? Anything similar?

# MP : Design Model



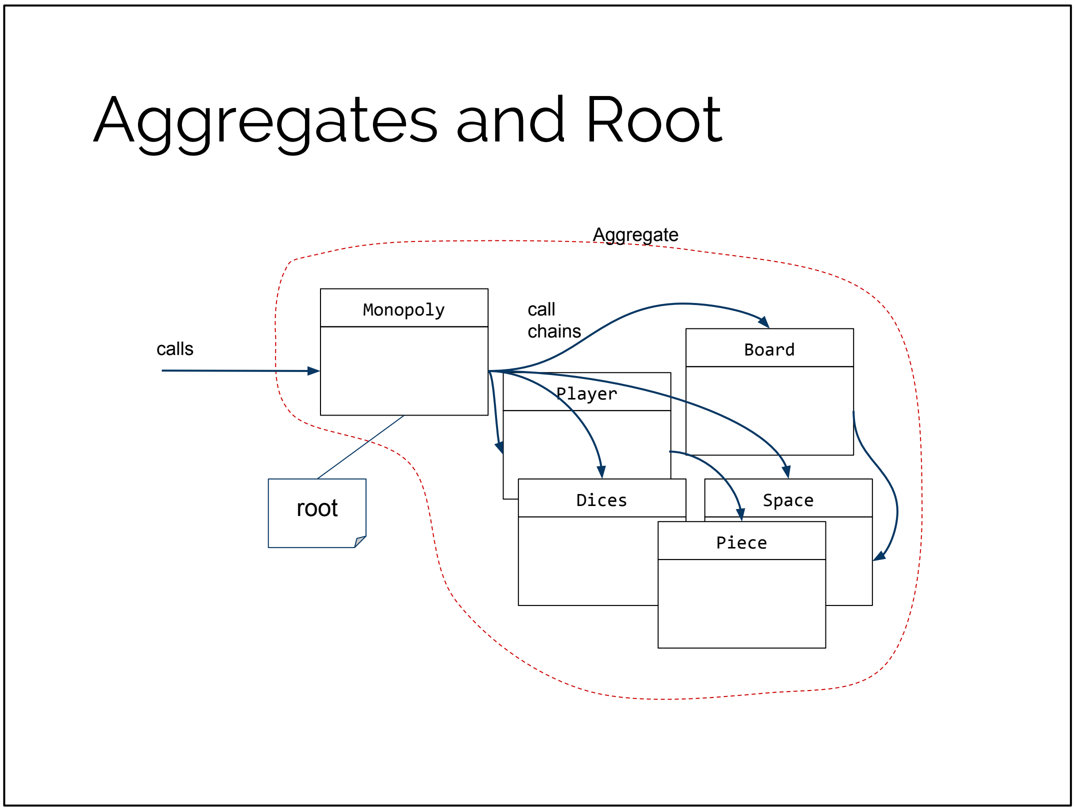
Represent pair of dices

To adapt the domain model to be suitable to implement we modify the model

- Possibly add, divide or collapse classes, and more …
- This is the design model

MP: Minor changes from domain model.

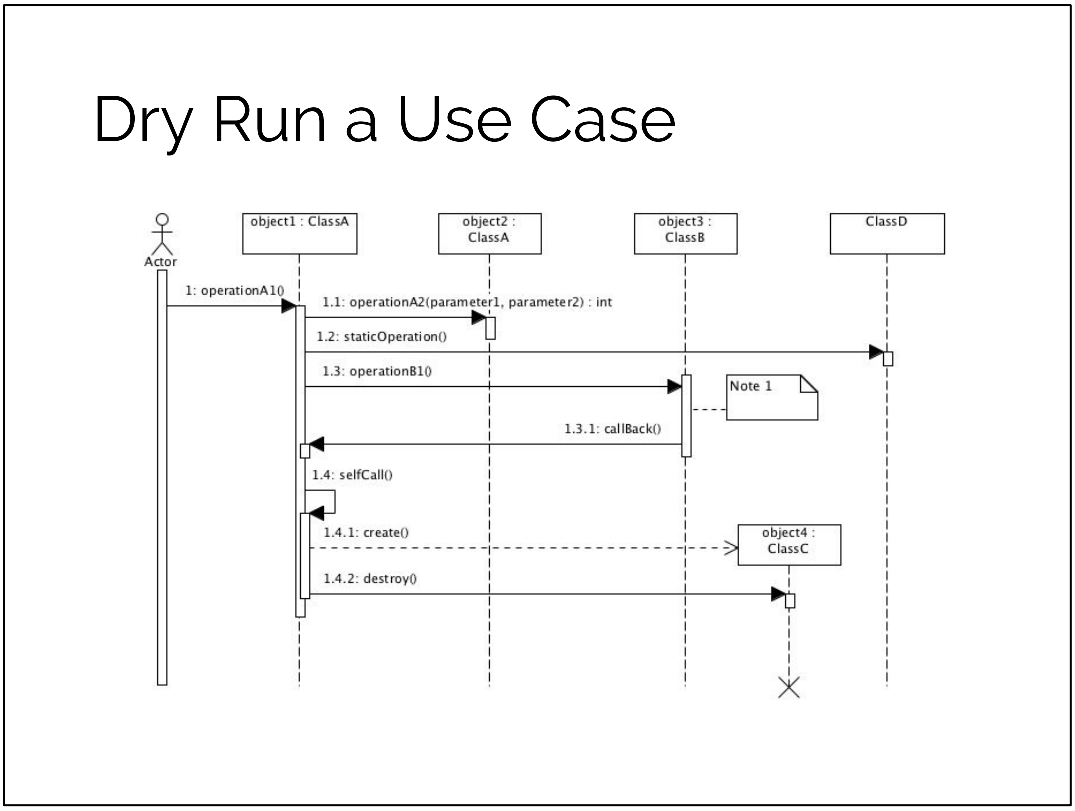
- Replaced Dice with class representing 2 dices



# Aggregates and Root

An aggregate is a cluster of classes (objects) treated as a unit
- All calls to the aggregate goes throught the aggregate root
- This will establish disciplined call chains in model
    - Will help to keep objects in a valid state

MP: We'll treat the complete model as an aggregate
- All calls will go through Monopoly object
- Not an universally valid decision, there may be ways to group (in other applications).



# Dry Run a Use Case

Dry run is last step before start implementing.

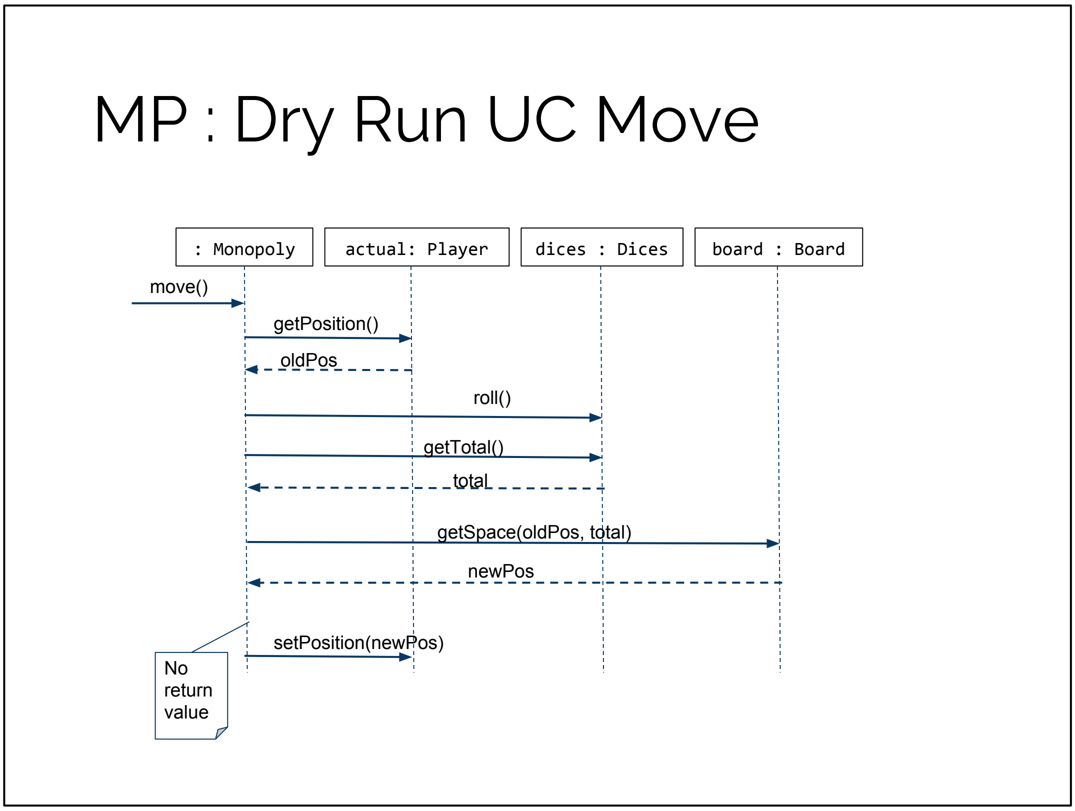
- Will must decide directions of associations (if not done before)
- Will reveal which methods in which classes!

For some use case(s) and the domain model

- Create an UML sequence diagram
  - Diagram will describe involved objects and application flow for the use case

If diagram gets very awkward/complex/messy possible have to modify domain model

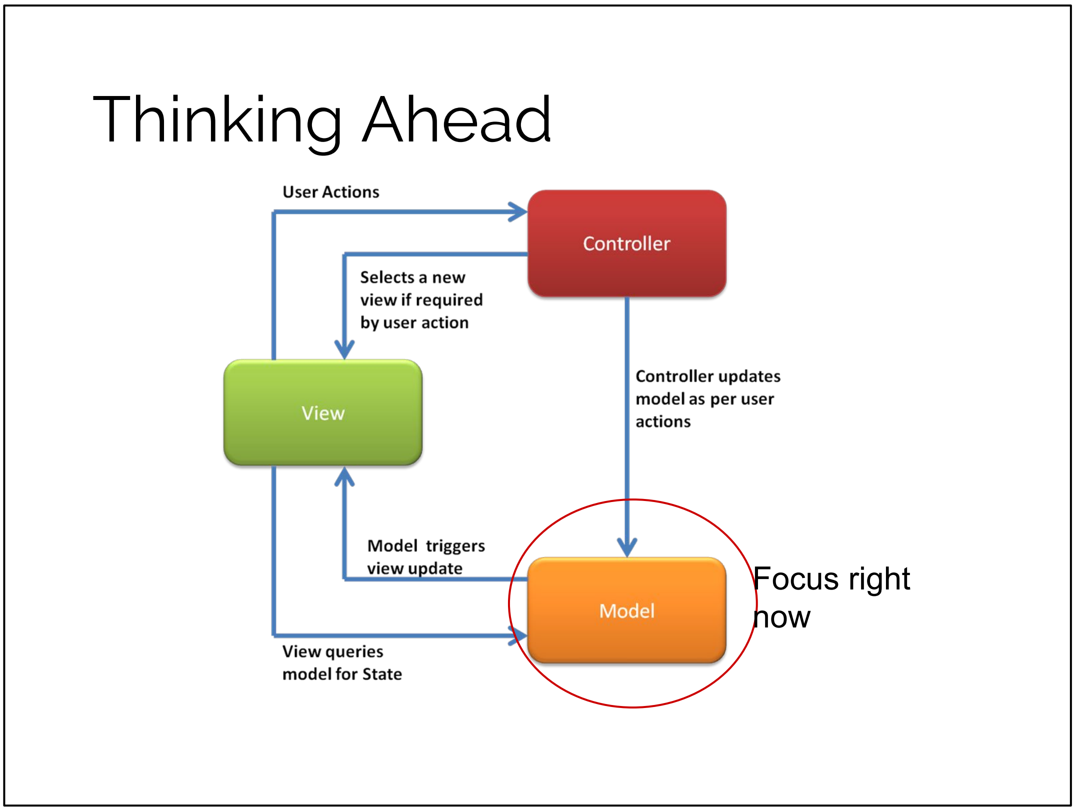
- Missing/bad association may be added/changed now
- Missing classes may show up



# MP : Dry Run UC Move

From this dry run it should be possible to implement use case move (just a simple translation from diagram to code)!

- … but in practice, … often most modify things …
- Just some few considerations before coding … (upcoming)



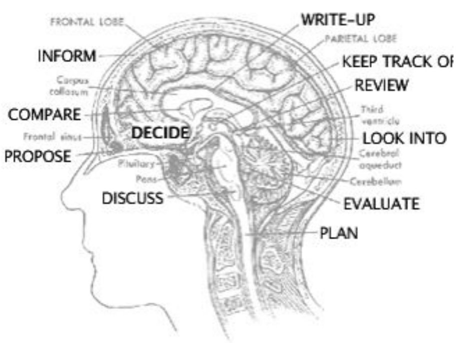
## Thinking Ahead

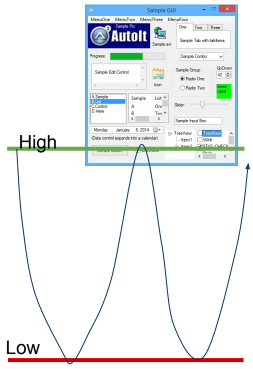
User Actions
Controller
Selects a new view if required by user action
View
Controller updates model as per user actions
Model triggers view update
Model
View queries model for State
Focus right now

Any application with a GUI will (should) use some sort of MVC architecture (pattern)

- Have to keep in mind.
  - Should methods have returnvalues or should it be handled by observer?
- Until now we have only worked with the model …
  - … except the view (GUI) sketches

# Think High and Think Low

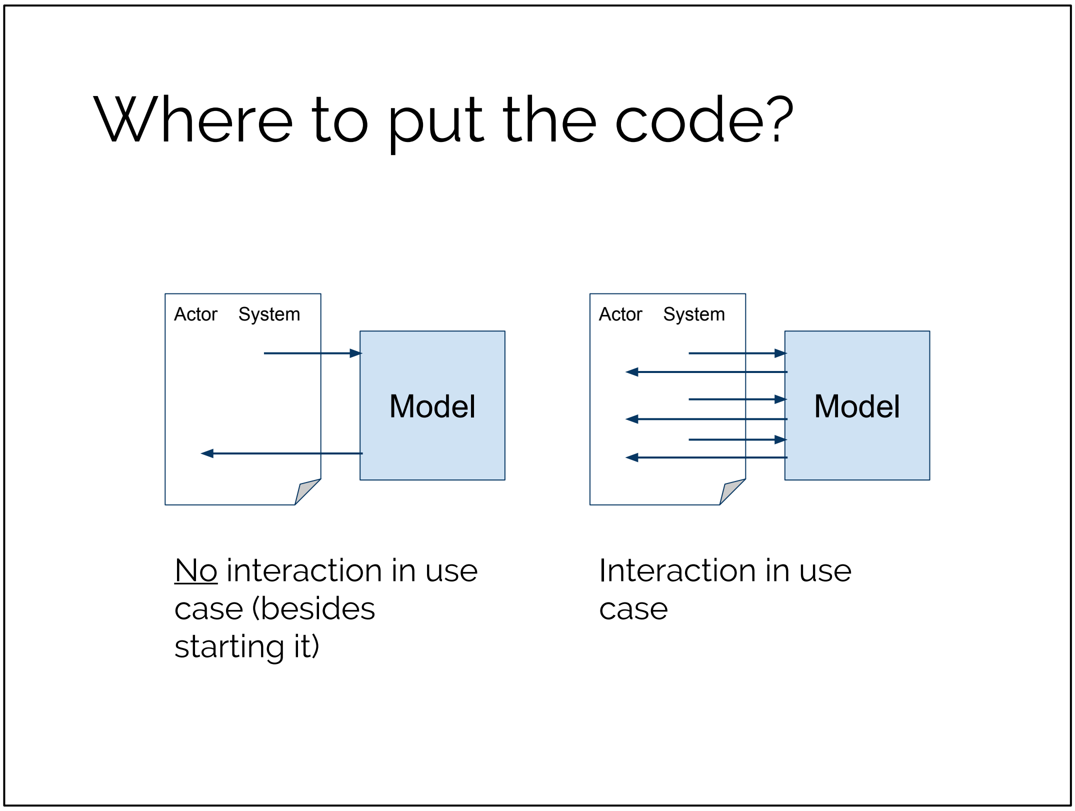




```
public void move() {
Space oldPos = actual.getPosition();
dices.roll();
Space newPos = board.getSpace(oldPos,
dices.getTotal());
actual.setPosition(newPos);
```

During implementation we must be able to switch between high and low level abstractions

- If stuck at high level (use cases, GUI ...), concretise by implement on low level to clarify (i.e. code it)
- If stuck at low level (during coding) abstract at high level
    - What is this about (how would GUI look from user perspective)?

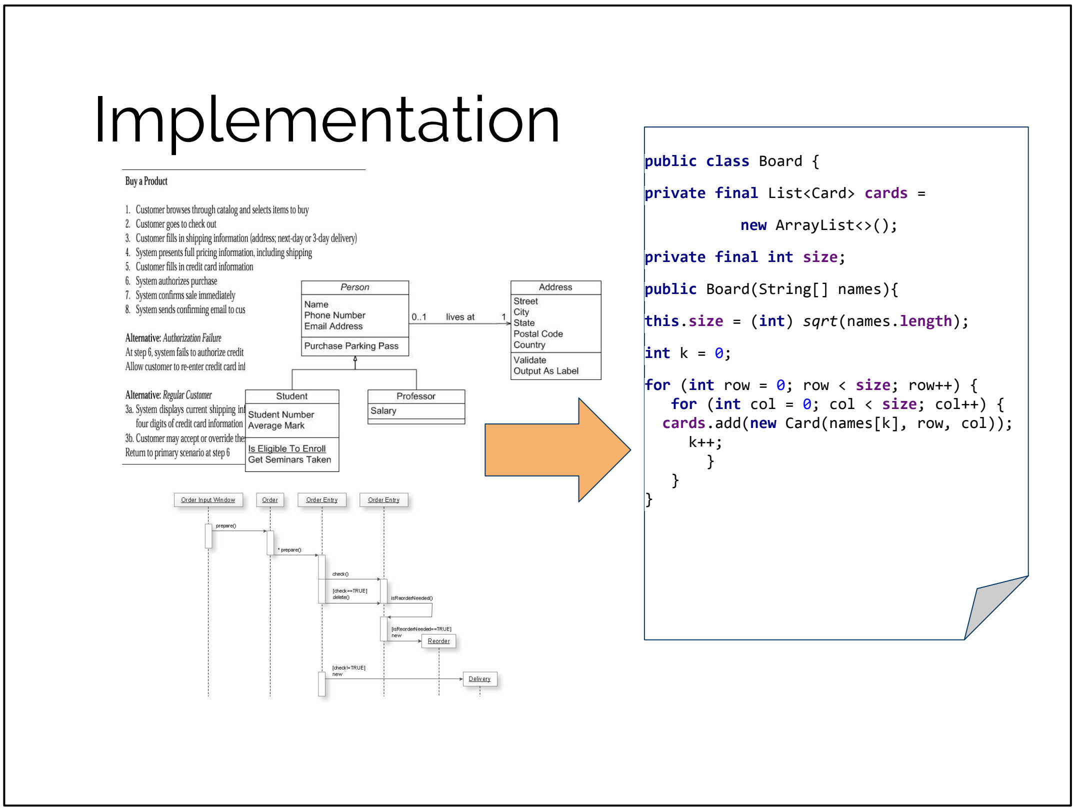


# Where to put the code?

No interaction in use case (besides starting it)

Interaction in use case

If …
- No user interaction in use case
  - Complete use case may run within single method call to model (i.e. like use case Move, normal flow)
  - Any GUI updates probably done using observer pattern (as part of MVC model) …
    - … so possibly no need for return values.
- If user interaction in use case
  - More calls to model
  - Later handled by control parts of MVC
  - More likely with returnvalues …
    - … control parts inspect returnvalues and act upon.



# Implementation

**Buy a Product**

1. Customer browses through catalog and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information (address; next-day or 3-day delivery)
4. System presents full pricing information, including shipping
5. Customer fills in credit card information
6. System authorizes purchase
7. System confirms sale immediately
8. System sends confirming email to cus

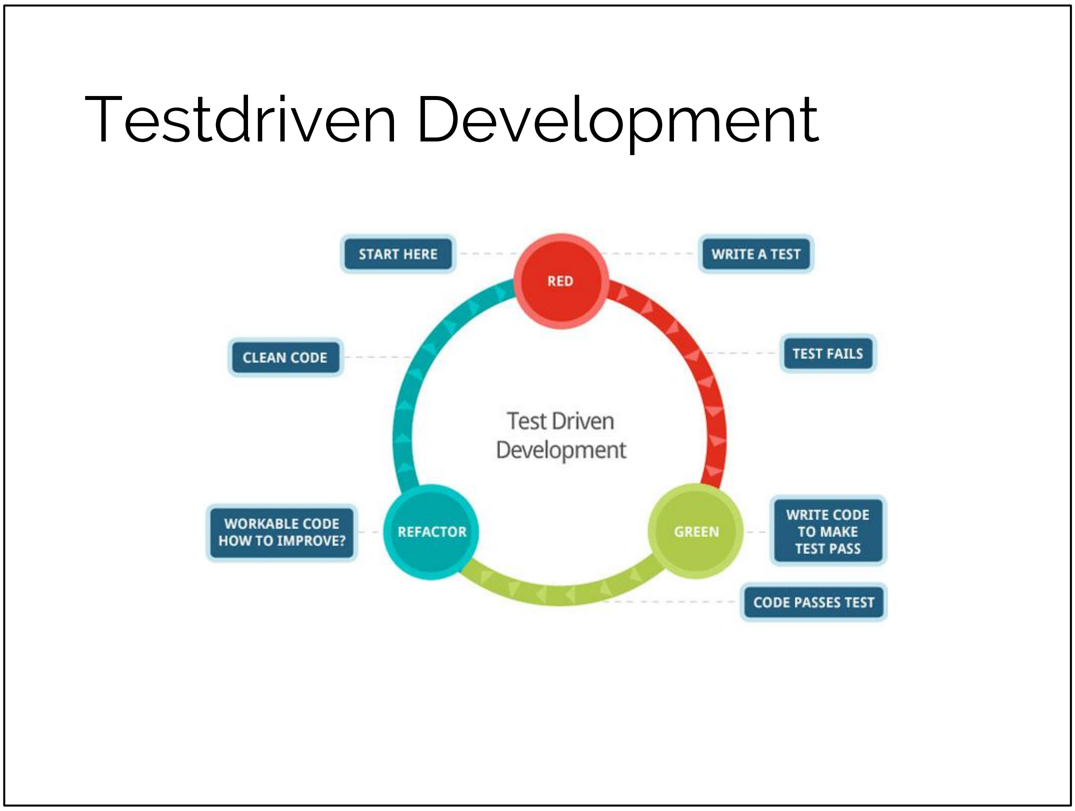
**Alternative:** *Authorization Failure*
At step 6, system fails to authorize credit
Allow customer to re-enter credit card inf

**Alternative:** *Regular Customer*
3a. System displays current shipping inf
four digits of credit card information
3b. Customer may accept or override the
Return to primary scenario at step 6

```java
public class Board {
private final List<Card> cards =
           new ArrayList<>();
private final int size;
public Board(String[] names){
this.size = (int) sqrt(names.length);
int k = 0;
for (int row = 0; row < size; row++) {
   for (int col = 0; col < size; col++) {
  cards.add(new Card(names[k], row, col));
     k++;
       }
   }
}
```

Have all information we need!

- Write the code and run …. ehhhh, run how… (upcoming)?



# Testdriven Development

Test driven development is a way to work with code inside the process

- During the implementation phase we use TDD

## How to Run?

How do we actually run the model???

- Answer: By creating tests!
  - We use test driven development!
  - We know JUnit, use it!

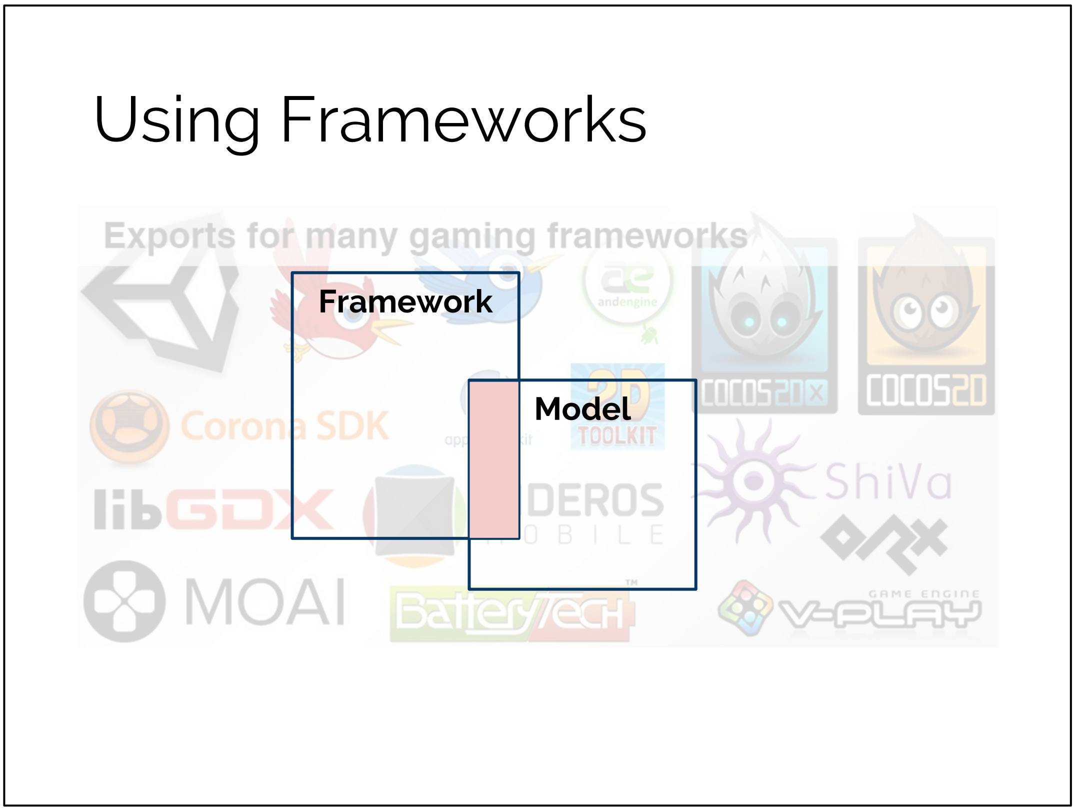
Why is this a good idea???

- We'll only produce the code we need!
  - The code needed to pass the test (the use case)!
- The code will have higher quality, because you will not implement "large" untestable methods
- Will always have something to run!
- Keeping work focused on the logic of the model
  - Great way to clarify the model logic
    - We must solve the problems (can't program them away)
  - Possibilities to discover model errors
- Debugging tests are much easier (vs full application)
- Being able to run a test suite against the model at any time if extremely useful.
  - In particular after refactoring
- Tests are very good documentation.
- Later: Being able to test certain techniques (snippets) also very useful

Tech talk

- We always keep test code separated from the application
  - In NetBeans: Test Packages

- The package structure for test should be the same as the structure for the application, more to come …



# Using Frameworks

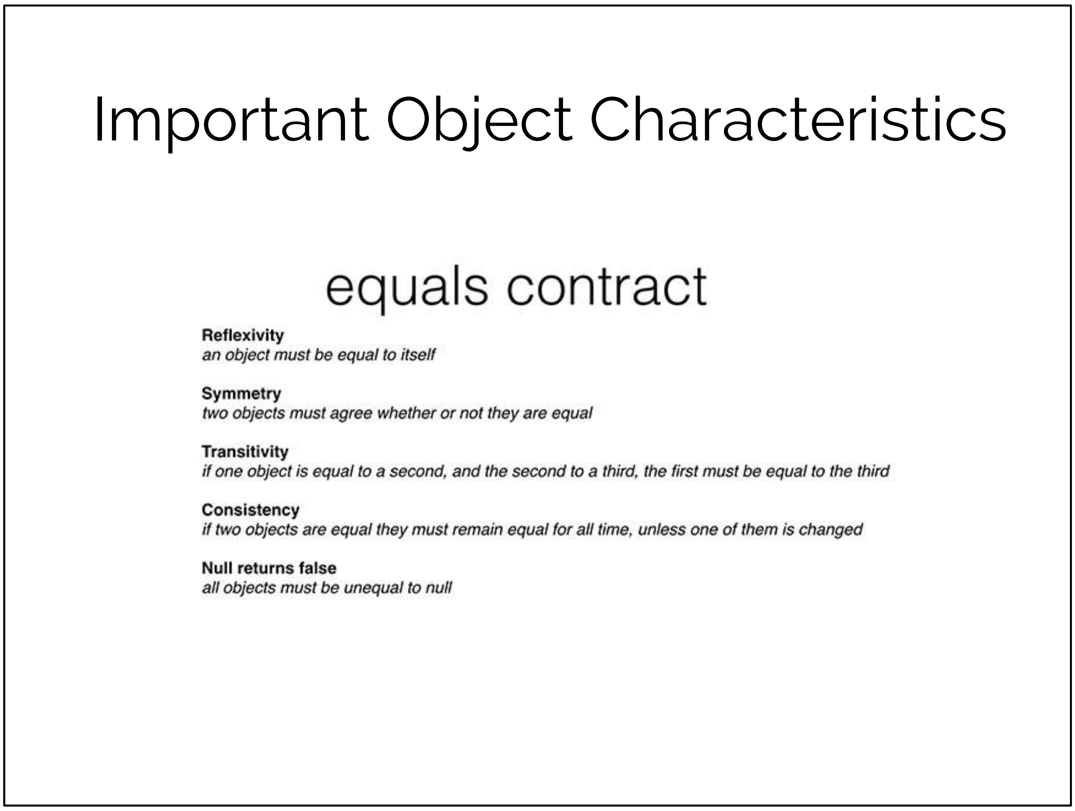
If using any framework possibly parts of model is handled by framework

- Example:
  - 2D Position (no x and y in model classes)
  - Collision detection
  - Movement/Physics/Rendering in 3D game frameworks
- Exclude from model parts handled by framework
  - If so: Can't (don't need to) test those parts
- Test what's not handled by framework
  - Possible have to mock (a lot)

NOTE: There should always be a model, using a framework doesn't mean you may skip the model

NOTE: Model may not be dependent on framework, more later at application design!

- Should be possible to switch framework



# Important Object Characteristics

## equals contract

**Reflexivity**
*an object must be equal to itself*

**Symmetry**
*two objects must agree whether or not they are equal*

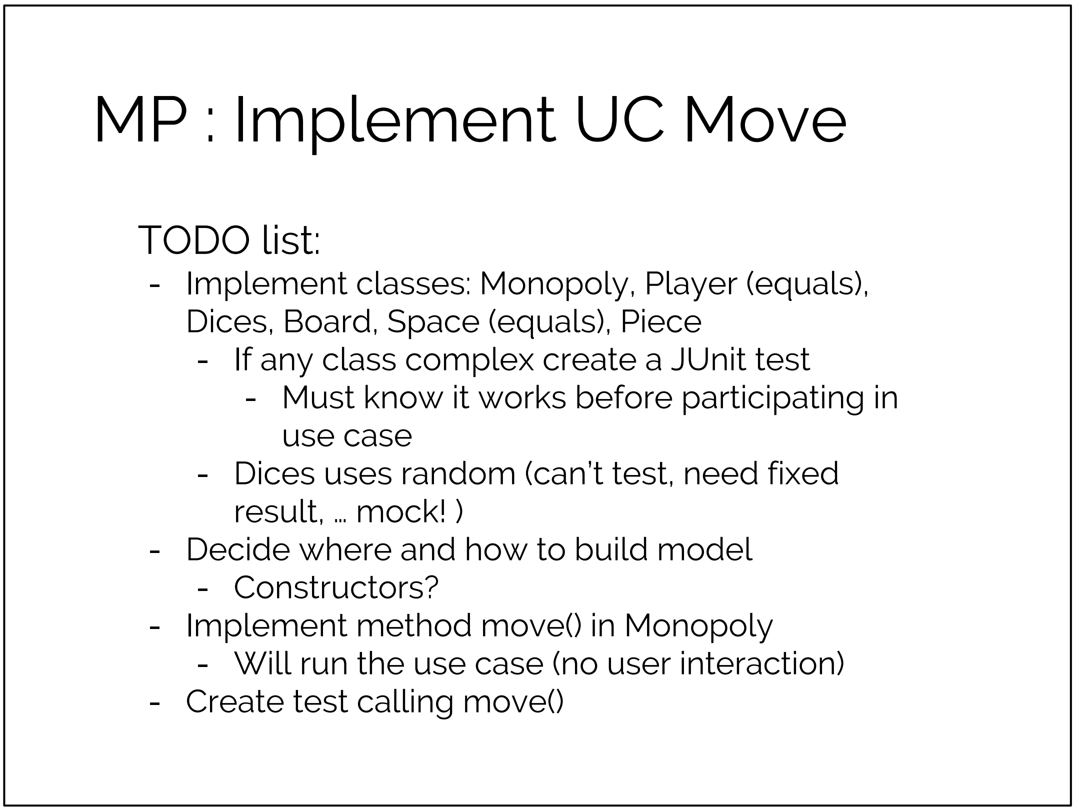
**Transitivity**
*if one object is equal to a second, and the second to a third, the first must be equal to the third*

**Consistency**
*if two objects are equal they must remain equal for all time, unless one of them is changed*

**Null returns false**
*all objects must be unequal to null*

Any class used in any Collection should implement equals() and hashCode()

- MP: Spaces, Players … (equals on name, name unique)



# MP : Implement UC Move

TODO list:

- Implement classes: Monopoly, Player (equals), Dices, Board, Space (equals), Piece
  - If any class complex create a JUnit test
    - Must know it works before participating in use case
  - Dices uses random (can't test, need fixed result, … mock! )
- Decide where and how to build model
  - Constructors?
- Implement method move() in Monopoly
  - Will run the use case (no user interaction)
- Create test calling move()

Finally ready to implement an use case!

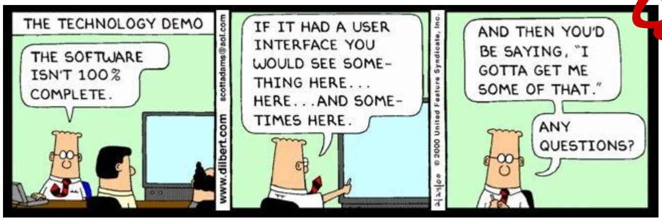
- If some class participating in the UC is complex first test it ..
- … senseless to try to implement a UC if not the "pieces" are working!

The development environment

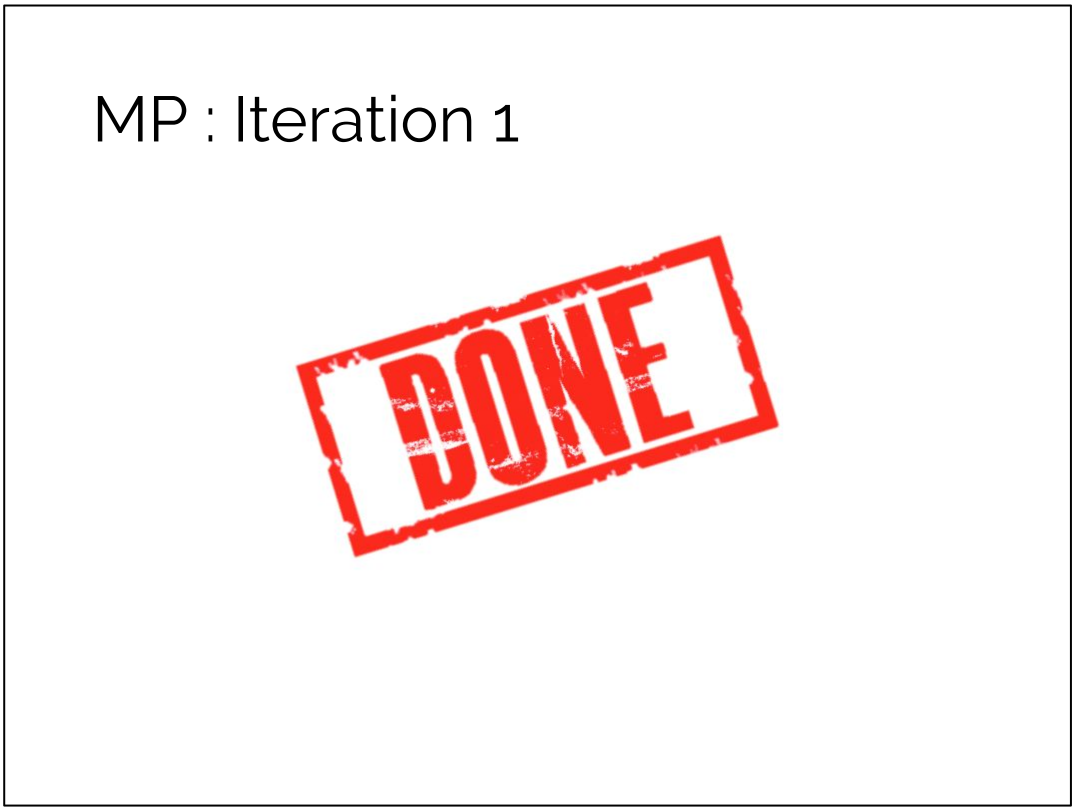
- Will use a Maven project
- Will run it using JUnit
- Version Handling using Git
- IDE: Netbeans (you use any …)

# MP : Monopoly-0.1

Demo time


THE TECHNOLOGY DEMO
THE SOFTWARE ISN'T 100% COMPLETE.
scottadams@aol.com
www.dilbert.com
IF IT HAD A USER INTERFACE YOU WOULD SEE SOMETHING HERE... HERE...AND SOMETIMES HERE.
© 2000 United Feature Syndicate, Inc.
AND THEN YOU'D BE SAYING, "I GOTTA GET ME SOME OF THAT."
ANY QUESTIONS?

Download from course page, inspect and run!



# MP : Iteration 1

Here we have done a full cycle, i.e. iteration 1

- Requirements
- Analysis
- Design (not much, just the model)
- Implementation of some high priority use case(s) ...
    - ... as JUnit tests (integration test)
    - Also test for complex classes (unit test)

# Summary

## Design and Implementation

- We got the first UC up and running!
- A small model (with some basic design)
- We only run as test for now

Next: Next iteration, more use cases (continue prototyping) real GUI and MVC.

Code sample for iteration 1: monopoly-0.1 (course page)