CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Object-oriented Programming Project

Design and implementation

Dr. Alex Gerdes

TDA367/DIT212 - HT 2018

Summary previous lecture

- Define the Idea so clear as possible
- Sketch the GUI
- Define Epics -> User Stories -> Tasks
- Prioritise -> Estimate -> Select -> Sprint (iteration)
- Start prototyping (in parallel)
- Domain model
  - Based on User Stories
  - Common language
  - Input for Design model
- Requirements Analysis Document

Seminar

- Quiz-based:
  - Try to guess the application based on domain model
- Purpose:
  - Get feedback on domain model
  - Give feedback
- Process:
  - Show domain model UML diagram (prepare a slide)
  - Leave out any class names that may give away application
  - Give some hints in case of no feedback
  - Show project idea / application
  - (3 min. quiz, 3 min. app, 3 min. feedback)

Design and implementation

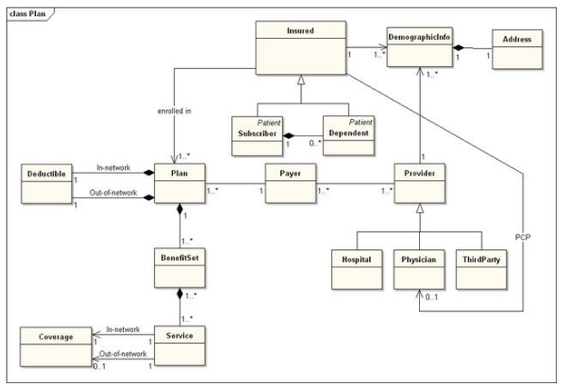
Design and implementation

- The next phase(s) of our process:
  - Create a implementable/runnable version of our domain model: the design model
  - Possibly some primitive GUI, some simplified MVC

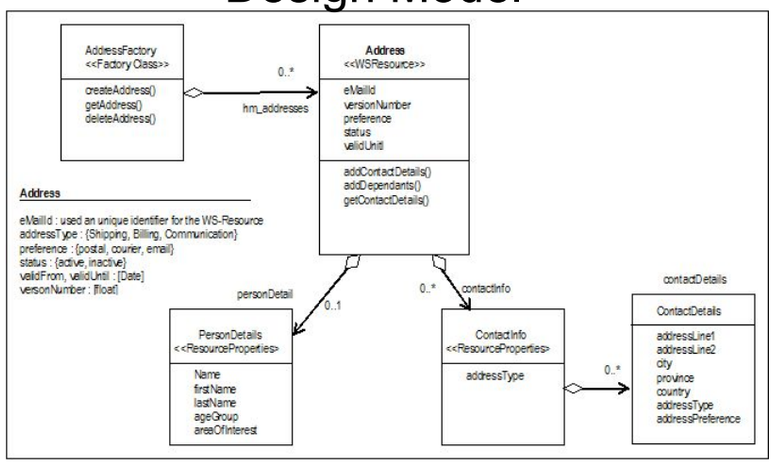
Requirement artifacts (from RE)

RAD

Domain Model



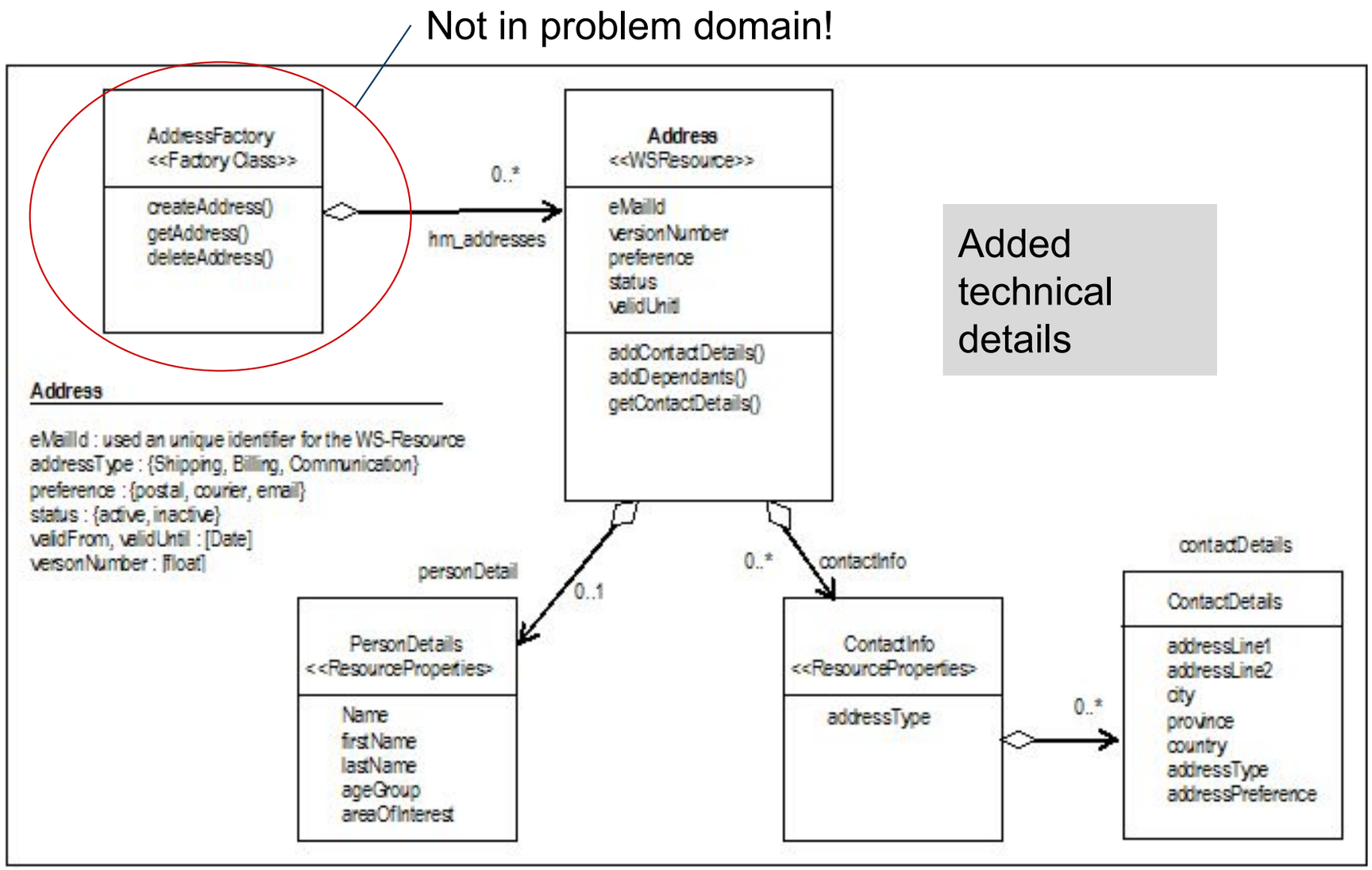
Design Model



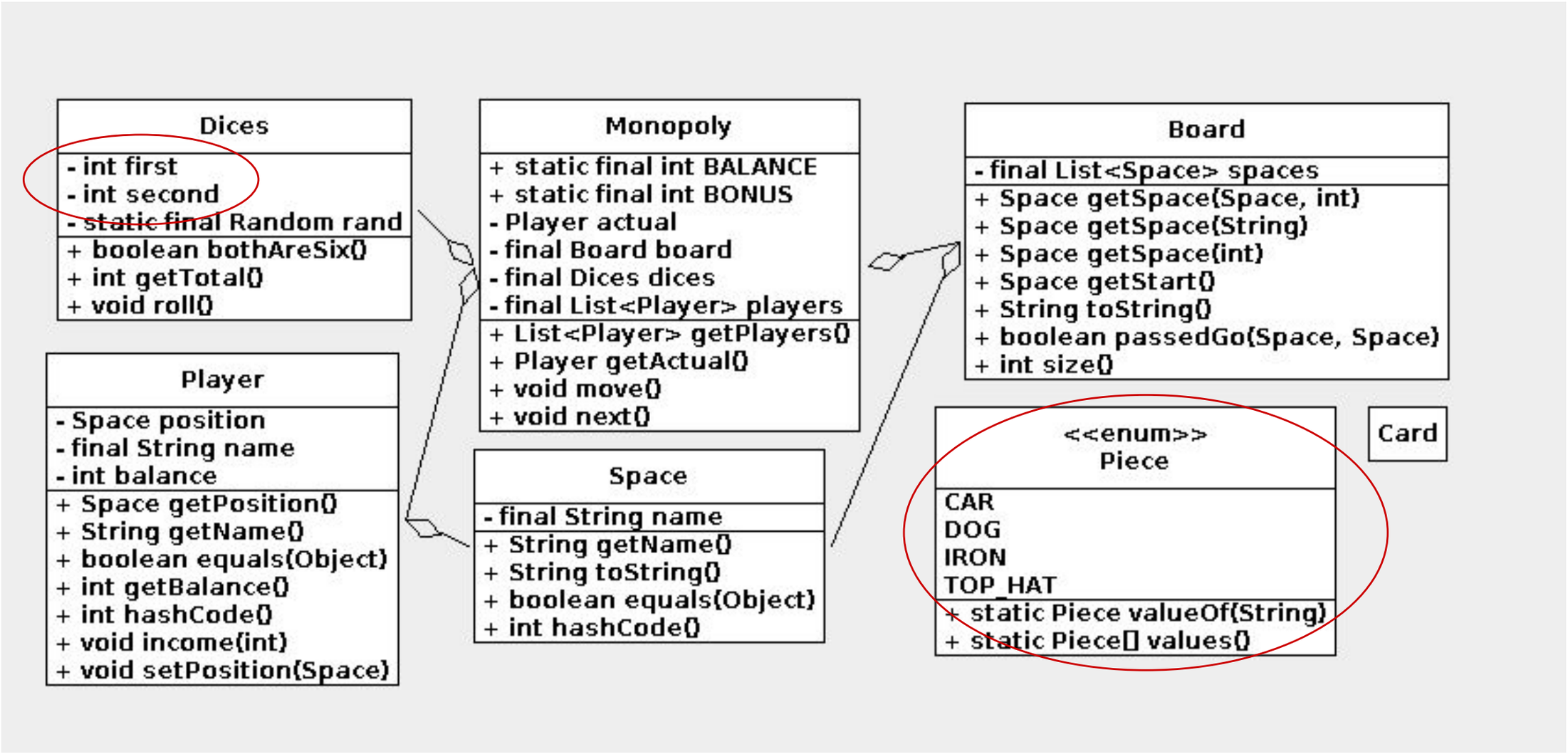
(Note: In the figure there are different domains for the models, this should not be the case)

Design model

- The *design model* is the domain model extend with technical classes and details
  - Enables us to implement the application
  - NOTE: Design model not *per se* understandable for all stakeholders
  - Must be correspond to domain model

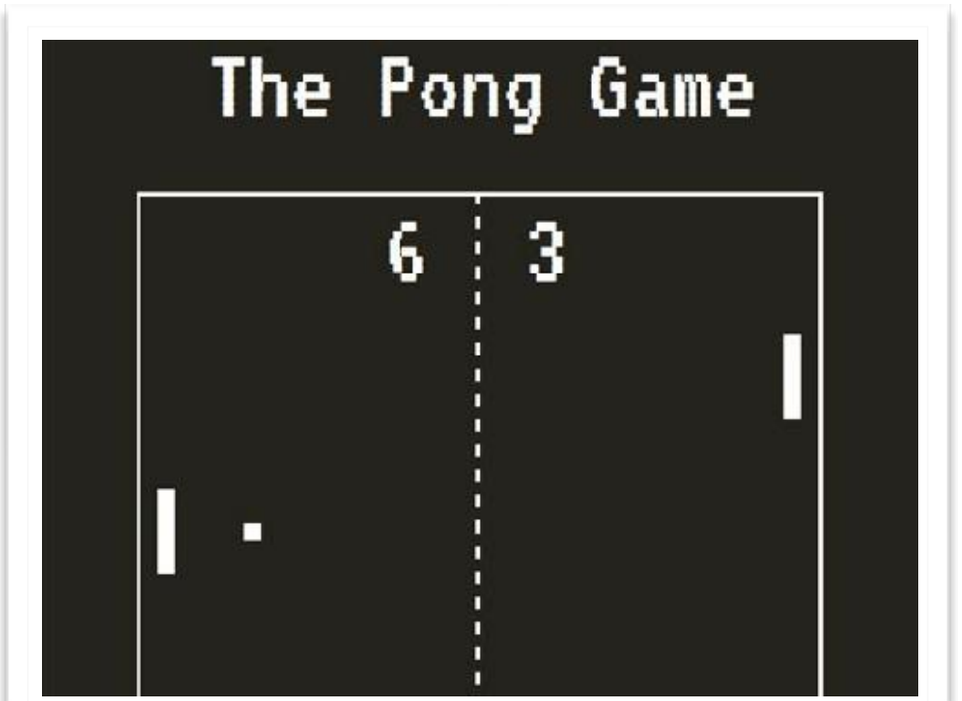


Monopoly: design model



Levels in model

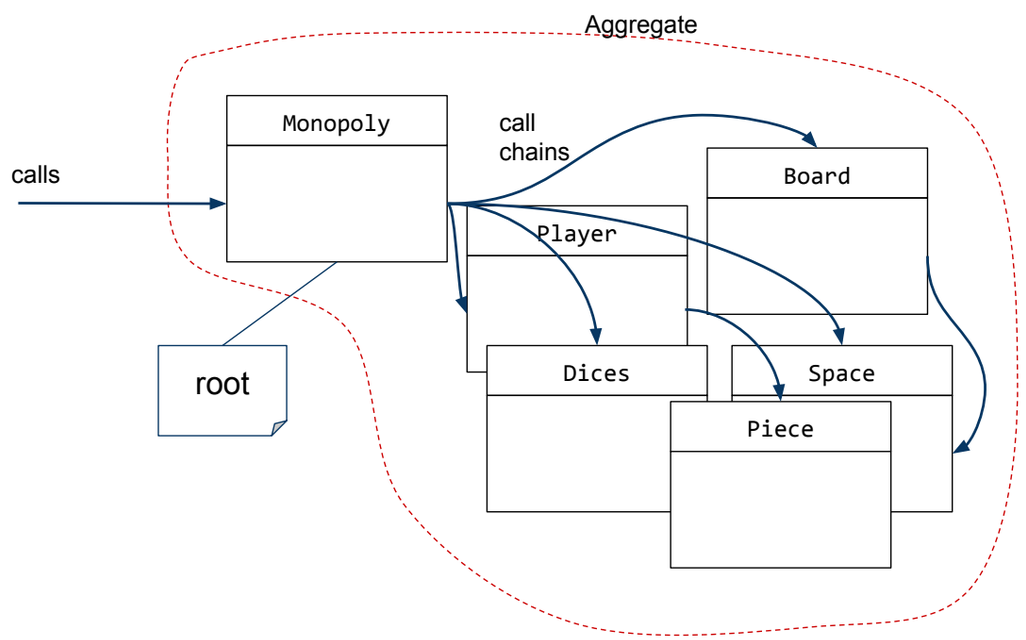
- The Pong game has a Ball and two Paddles
  - Which will check for collision:
    - Ball or Paddle?



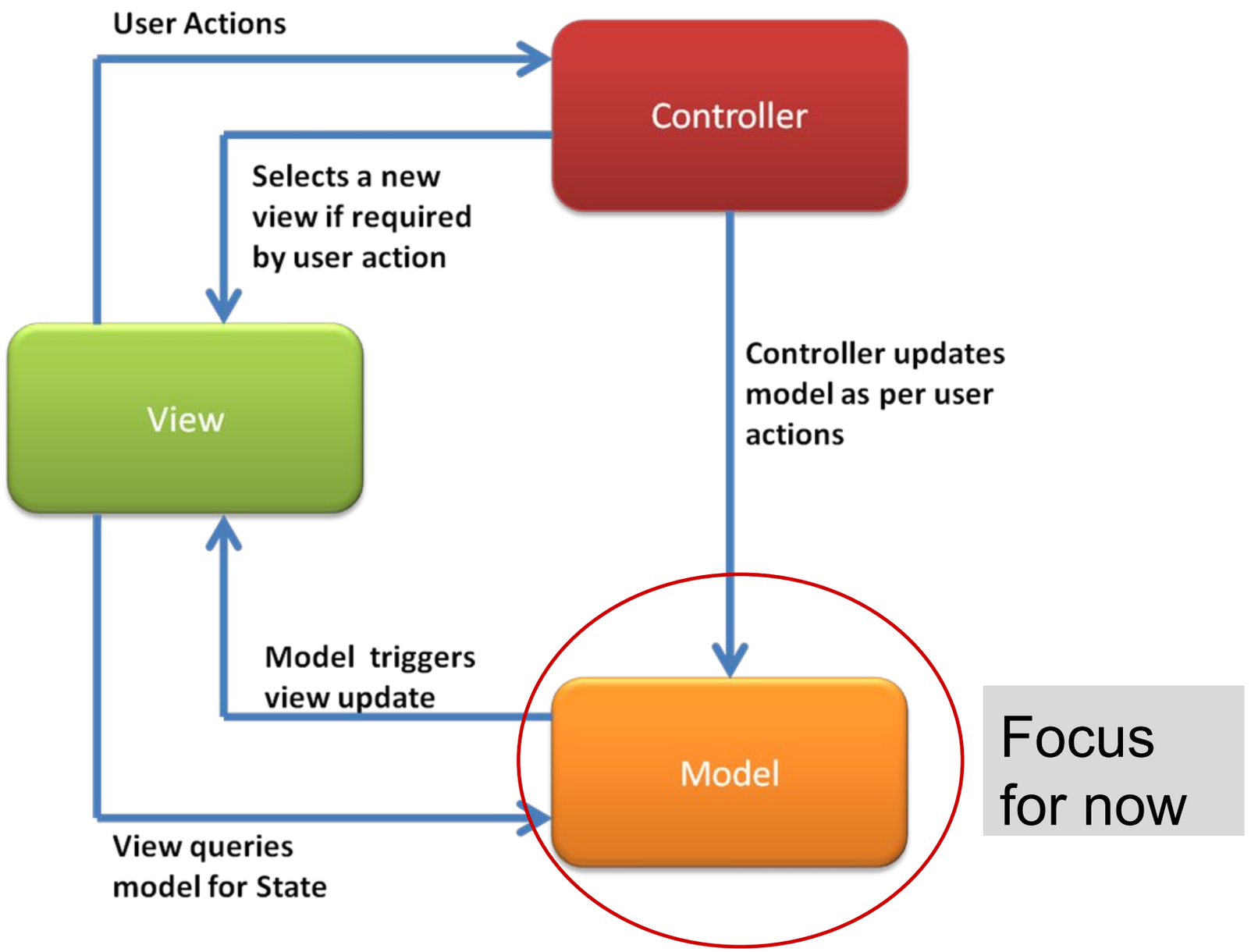
- Answer: probably none of!
  - There are levels in the model
  - Some objects are at a higher level, handling objects at lower level
  - Paddle and Ball are at the “same” level, so something higher up should handle collisions

Aggregates

- An aggregate is a cluster of classes treated as a single unit
  - All calls to the aggregate go through the *root* of the aggregate
  - This prevents unnecessary dependencies and allows for proper call chains
  - Will help to keep objects in a valid state
- If there are too many methods in root, add method to return sub-aggregate with a new root
- Monopoly: we treat the complete model as an aggregate
  - All calls will go through `Monopoly` object
- Not an universally valid decision, there may be other ways to group (in other applications)



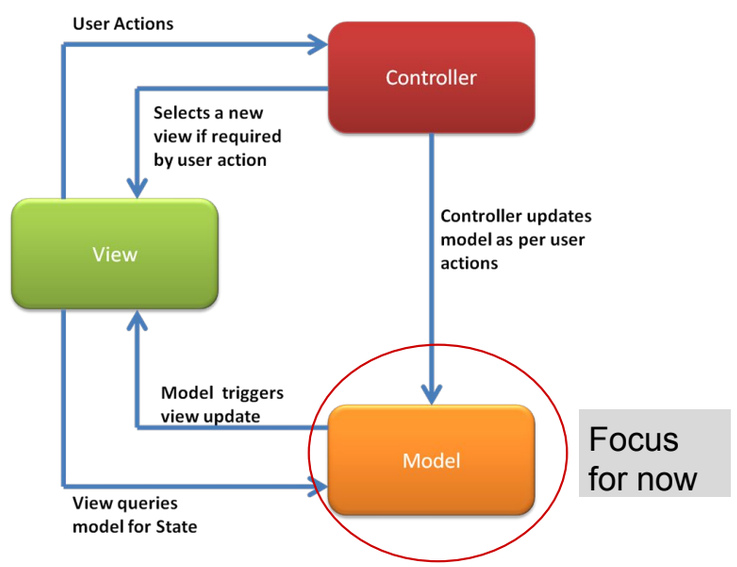
MVC recap



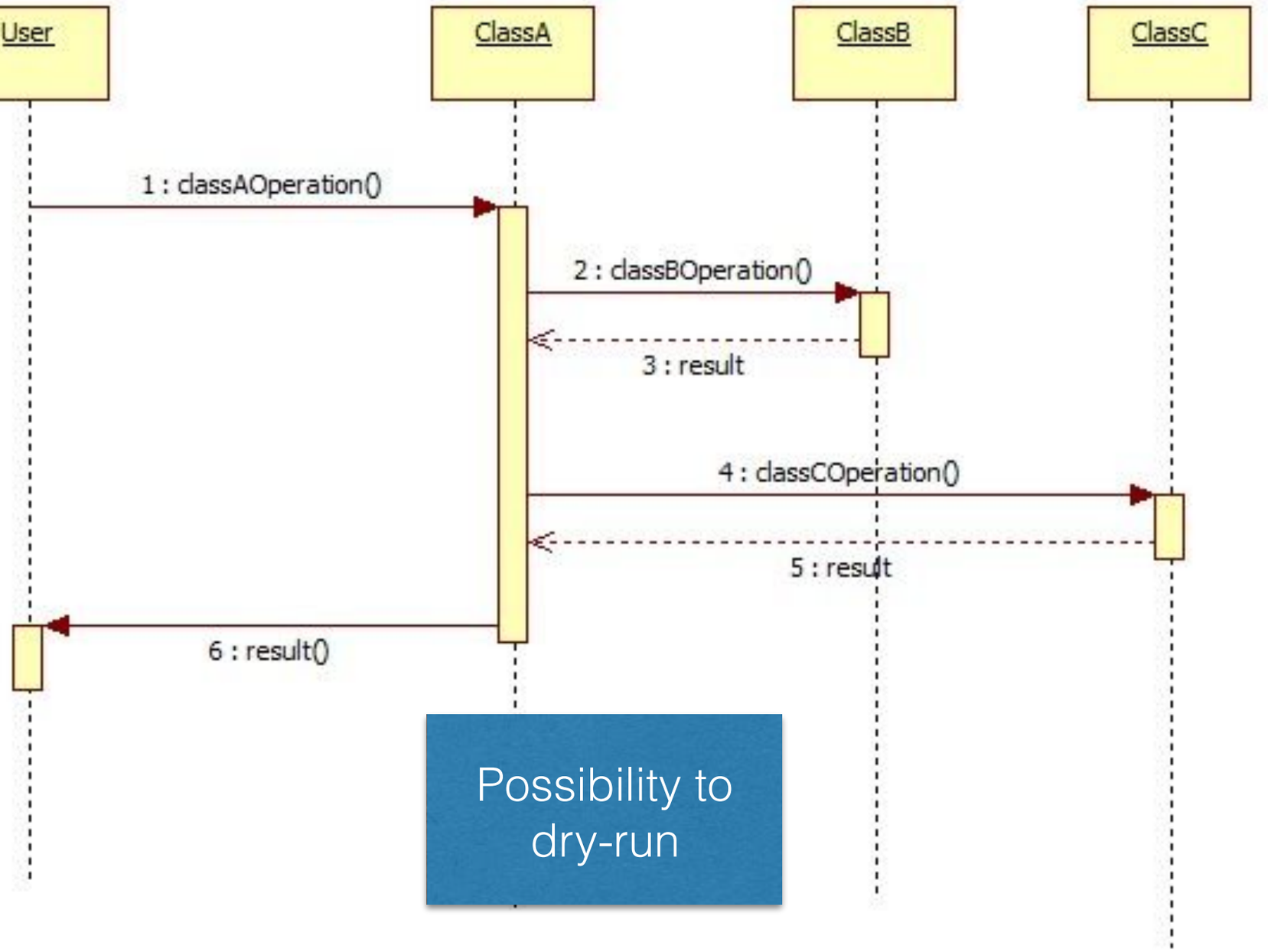
Focus for now

Return values

- Any application with a GUI should use some sort of MVC architecture
  - Should methods have return values or should it be handled by an observer?
  - Until now we have only worked with the model except the view (GUI) sketches
  - NOTE: Methods with return values easier to test
- No user interaction in user story
  - Complete user story may run within single method call to model
  - Any GUI updates probably done using observer pattern, so possibly no need for return values
- User interaction in user story
  - More calls to model
  - Later handled by control parts of MVC
  - More likely with return values
  - Controllers inspect return values and act accordingly



UML sequence diagram



UML sequence diagram

- A sequence diagram is used to describe the *dynamic behaviour* of interacting objects

- A dry-run is usually the last step before implementation
  - Decides directions of associations (possibility to reevaluate)
  - Reveals in which class a method should be placed

- Create an UML sequence diagram for some aspects of the design model

- If diagram gets very awkward/complex/messy possibly have to modify domain/design model
  - Missing/bad association may be added/changed
  - Missing classes may show up

- If diagram to big, decide on which abstraction level, factor out lower levels to separate diagrams

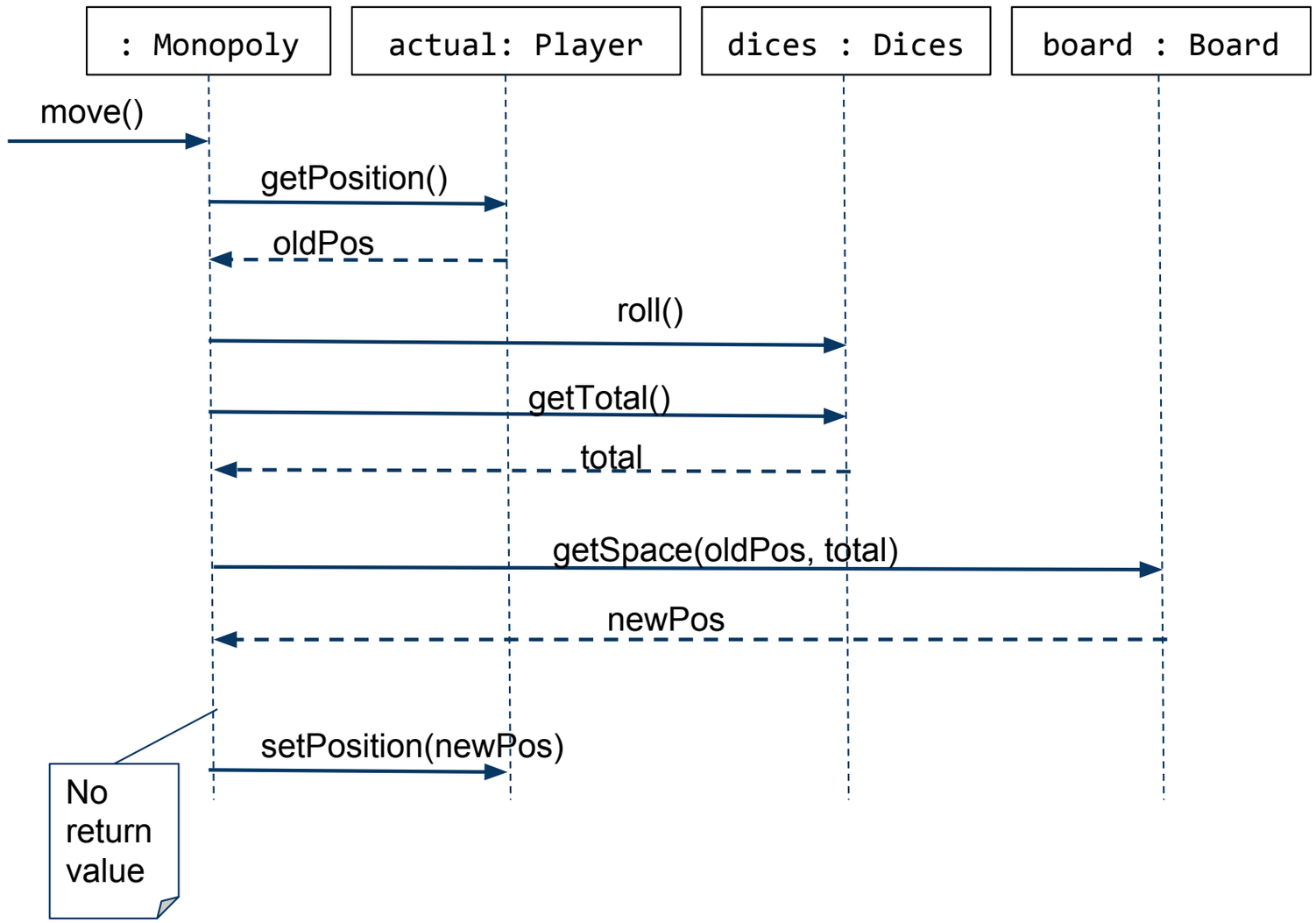
Monopoly: roll dices User Story

As as: player
I want to: roll the dices
so that: I can make a move

Acceptance:
- Player can start the roll of the dices
- All players can see the result
- After rolling the player can make a move
- The player can only roll the dices once
- …

- Show the board
- Show the players on the board
- Highlight the active player
- Allow the active player to roll the dices
- Show the resulting dice values
- Make the dices values available to other actions (next move tex)
- Change active player

Monopoly: roll dices User Story



- From this dry-run it should be possible to implement the user story ’roll dices’
  - But in practice things may turn up -> often necessary to modify

Data representation


```
Spaces[] board = …

Spaces[][] board = …

List<Space> board = …

Map<String, Space> board = …
```

Implementing classes

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

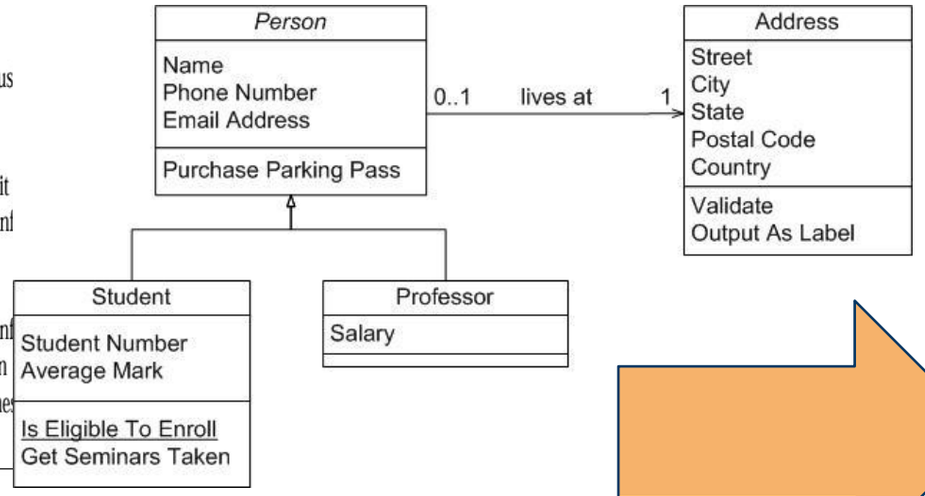
Implement!

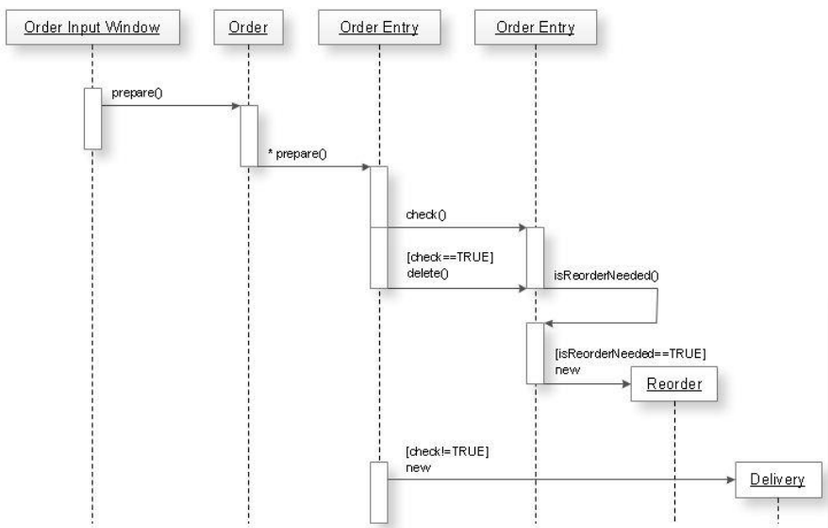
Buy a Product

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 thes
Return to primary scenario at step 6



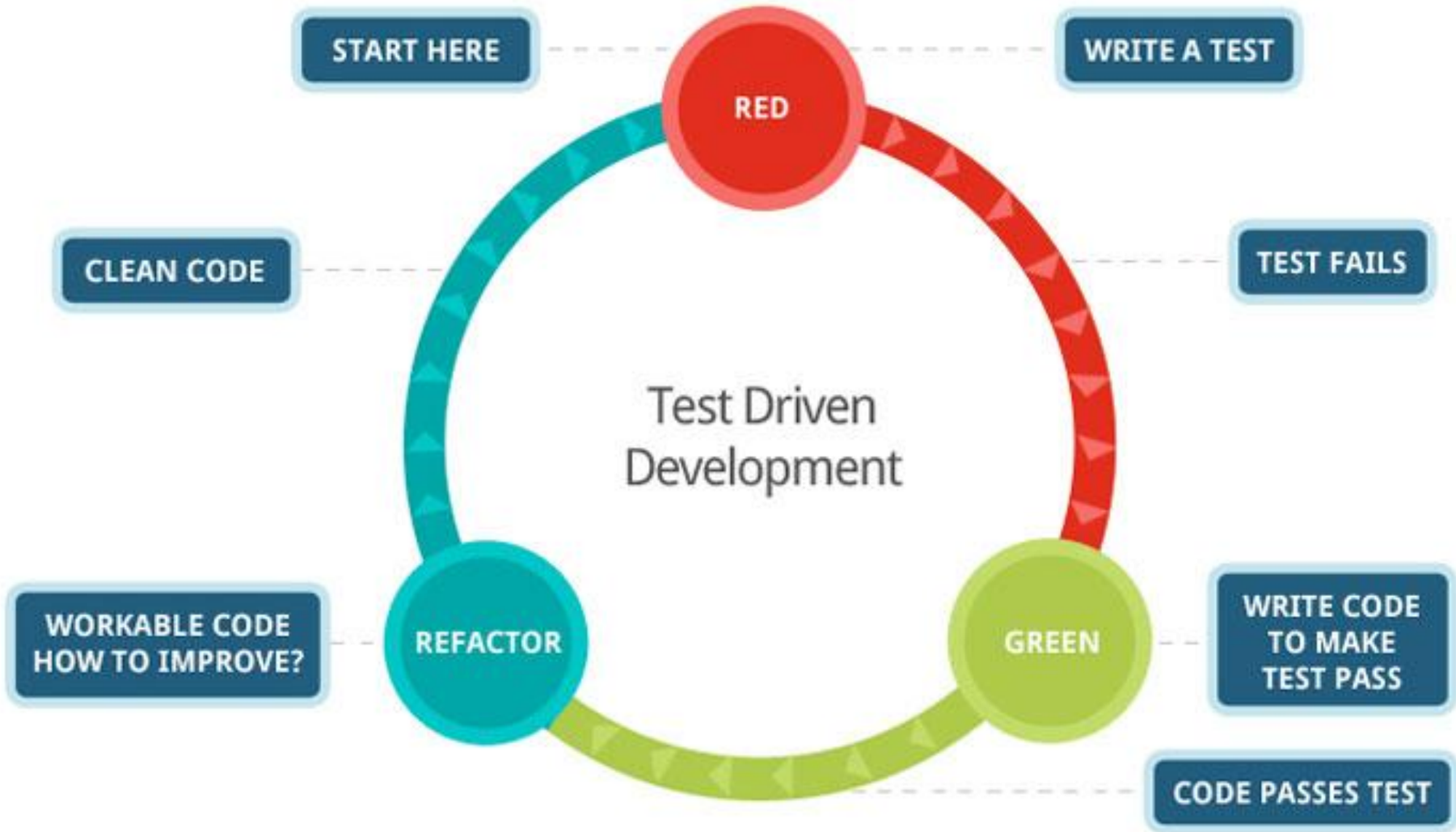



```
public class Board {
private final List<Card> cards =
          new ArrayList<>();
private final int size;
public Board(String[] names){

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?

Test driven development



How to run?

- By creating tests!
- Why is this a good idea?
  - We'll only produce the code we need!
  - 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
    - 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 is extremely useful
    - In particular after refactoring
- Keep test code separated form the application (as much as possible)

Tests as documentation



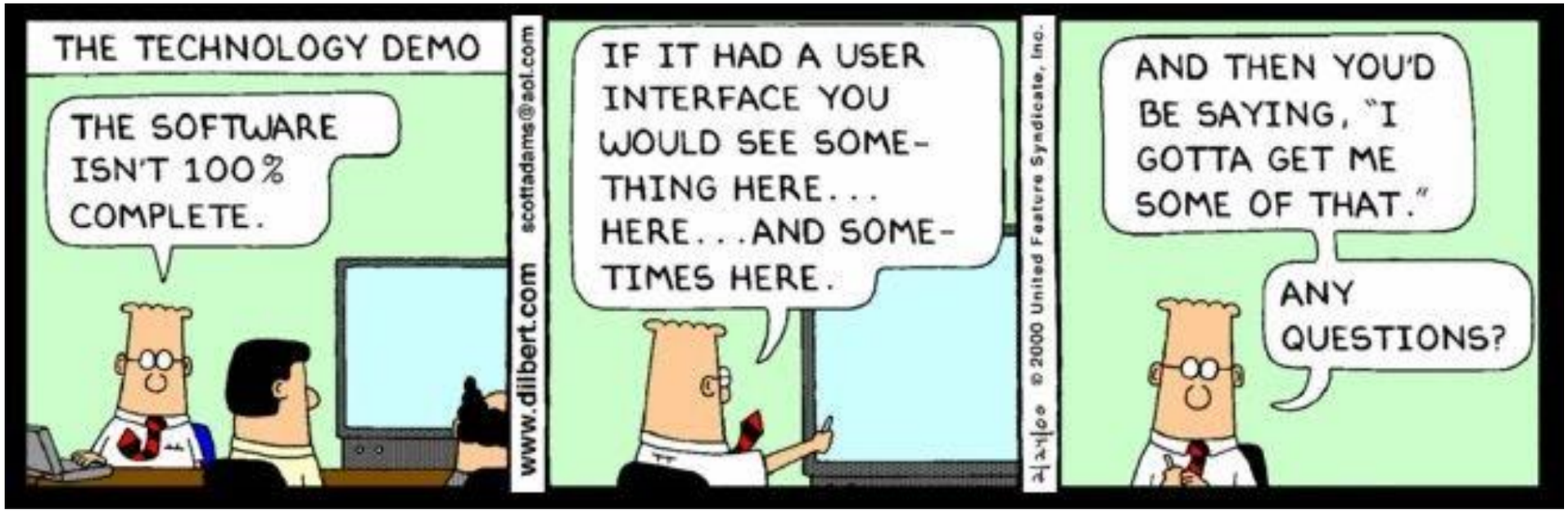
```java
@Test
public void testMoveAndPassGo() {
  Player player = m.getActivePlayer();
  int startBalance = player.getBalance();
  player.setPosition(getSpace(30));
  m.setDices(new MockDices(12)); // make sure we pass start
  m.move();
  assertEquals(player.getBalance(),
               startBalance + Monopoly.BONUS);
}
```

Monopoly: implement Roll User Story

- TODO list:
  - Implement classes: Monopoly, Player, Dices, Board, Space, Piece
    - Create JUnit tests, especially for more complex classes
  - Dices uses random, which is difficult to test, need fixed result -> *mock*!
  - Decide where and how to build model
    - Constructors?
  - Implement method `move()` in Monopoly
  - Create test calling `move()`
- The development environment
  - Will use a Maven project
  - Will run it using JUnit
  - Version Handling using Git
  - Continuous Integration with Travis

Monopoly: first version

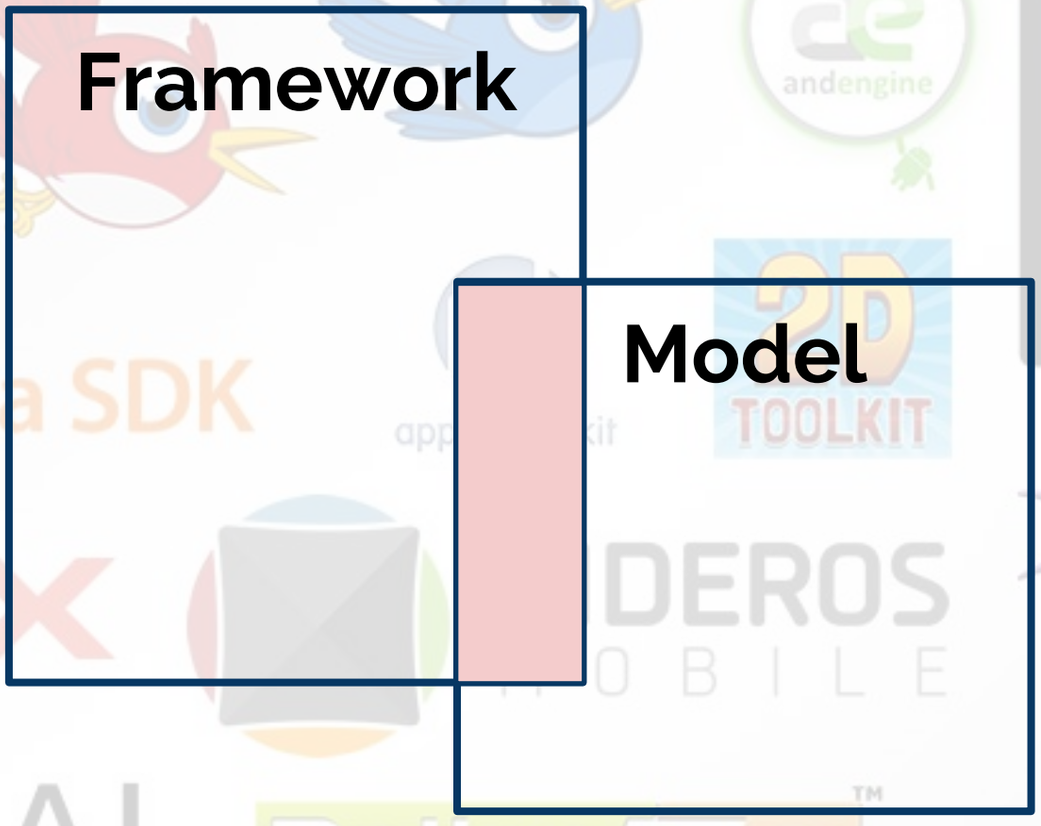
Demo time



Monopoly: first iteration

- We have done an iteration(ish):
  - Requirements
  - Analysis
  - Design
  - Implementation of a (part of) high priority user story
    - As JUnit tests
    - Also test for complex classes (unit test)
  - Now reflect and refactor!

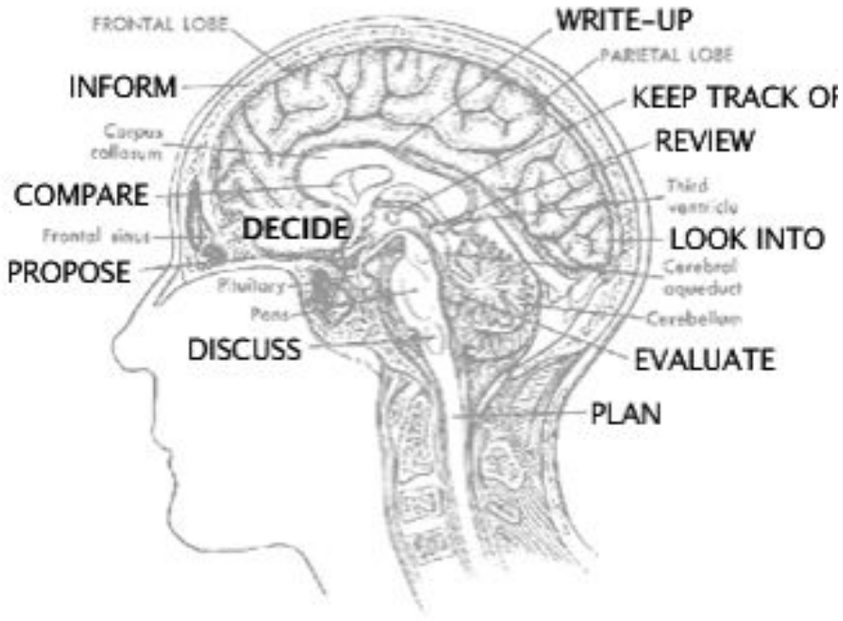
Using frameworks/libraries

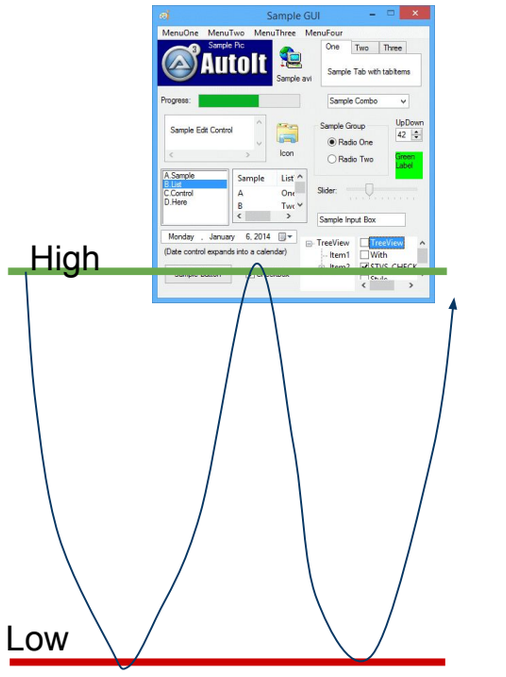


- There should always be a model!
- Look for quality
- Be as independent as possible

Thinking High and Low

- During implementation we must be able to switch between high and low level abstractions
  - If stuck at high level (user stories, GUI ...), concretise by implement on low level to clarify (i.e. code it, use prototypes)
  - If stuck at low level (during coding) abstract at high level
    - What is this about (how would GUI look from user perspective)?






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

Summary

- We have a running model
  - We got the first task/user story up and running!
  - Continue to complete chosen user stories
    - A small model (with some basic design)
    - We only run as tests for now
- Next iteration, more user stories, continue prototyping, refine design, improve GUI, etc.