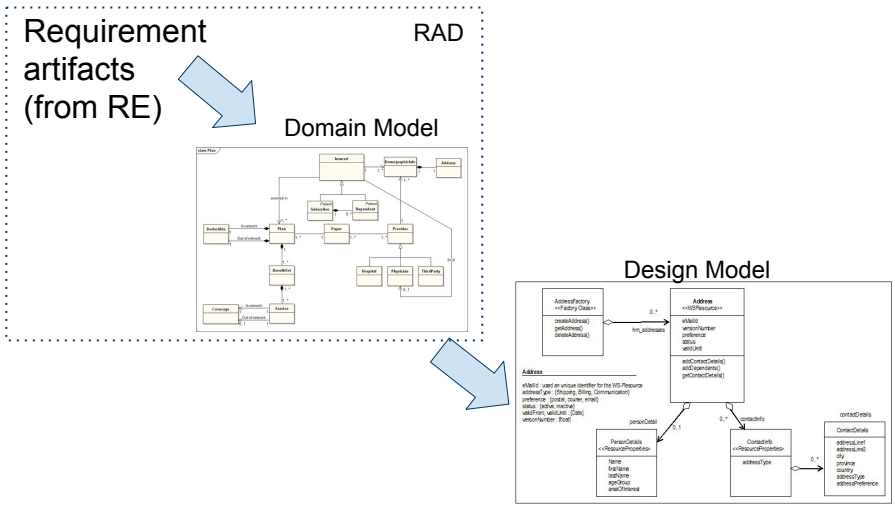
Design and Implementation

Slide Series #4

Design & Implementation

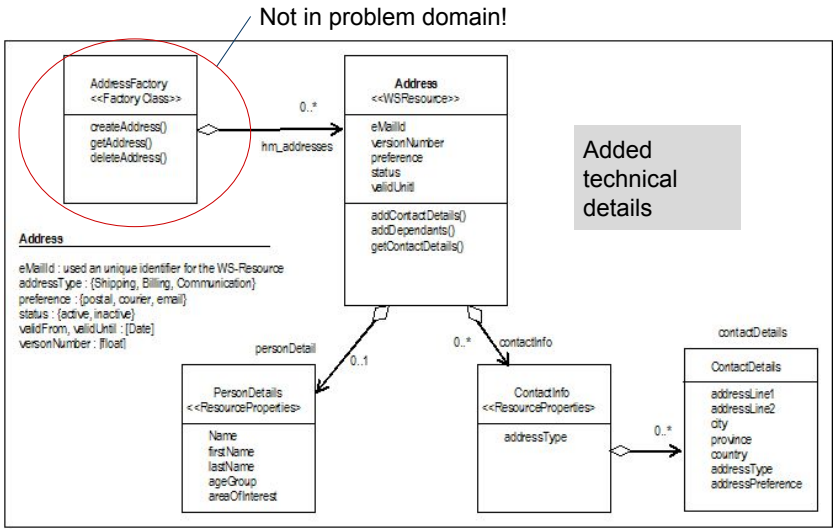


The last phase(s) of our process

- Now we'll create a runnable version of our domain model, the design model
- Possibly some primitive GUI, some simplified MVC (more on MVC later)

Note: In slide there are different domains for the models, this should not be the case.
Here I just try to show that the design model is more detailed.

Design Model



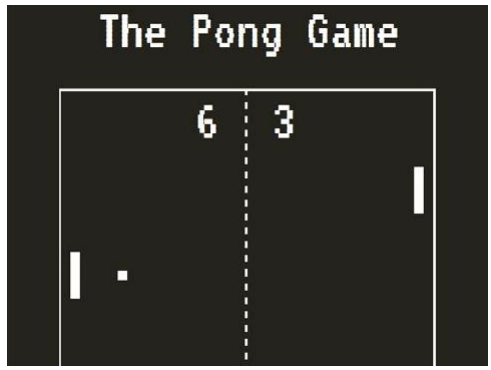
The design model is the domain model enhanced with technical classes/details, constructors, etc

- To make domain model runnable!

NOTE: Design model must not be understandable for all stakeholders (customer).

- Must be traceable form domain model.

Levels in Model



The Pong game has a Ball and Two paddles

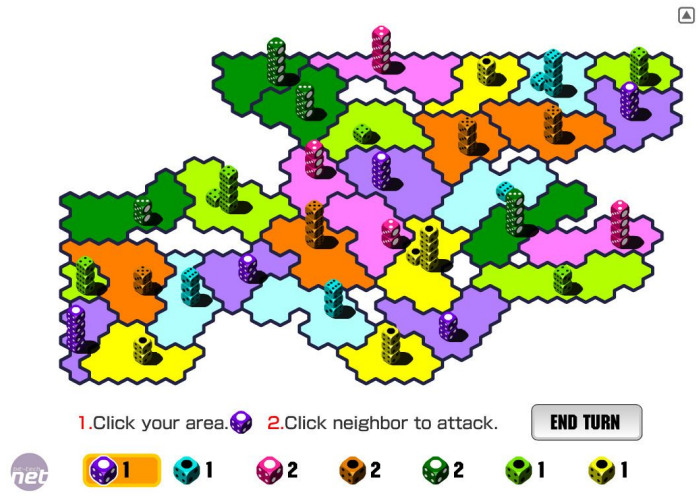
- Which will check for collision?
- .. or will (should) someone else check …?

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

How about MP? Anything similar?

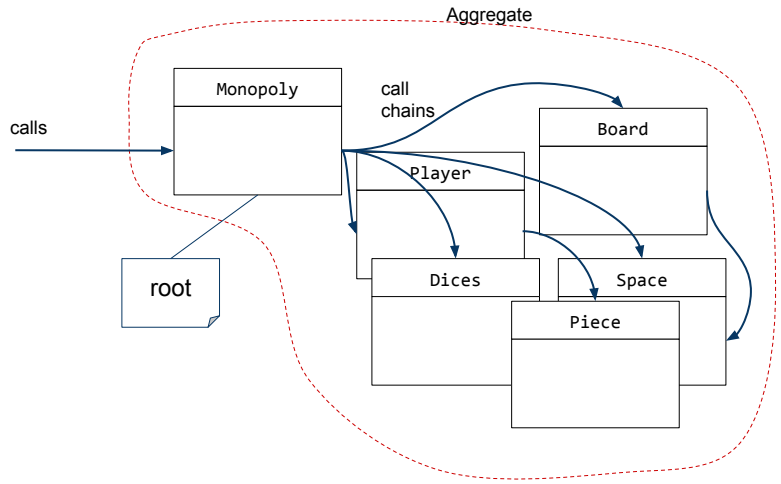
Where to Place Methods?



Assume

- Map is built out of hexagons (hexes). All hexes have a (unique) position
- Each country is a collection of hexes (and possible more, .. an owner)
- So classes are: Hex, Map and Country
    - Where to put a method to get all the neighbours of a hex (cases: hex have references to all neighbours or not?)?
    - If by value in Hex, will be parameterless method (no parameters is best possible)
    - ... if by reference in Map (must supply hex as parameter)

Aggregates and Call chains



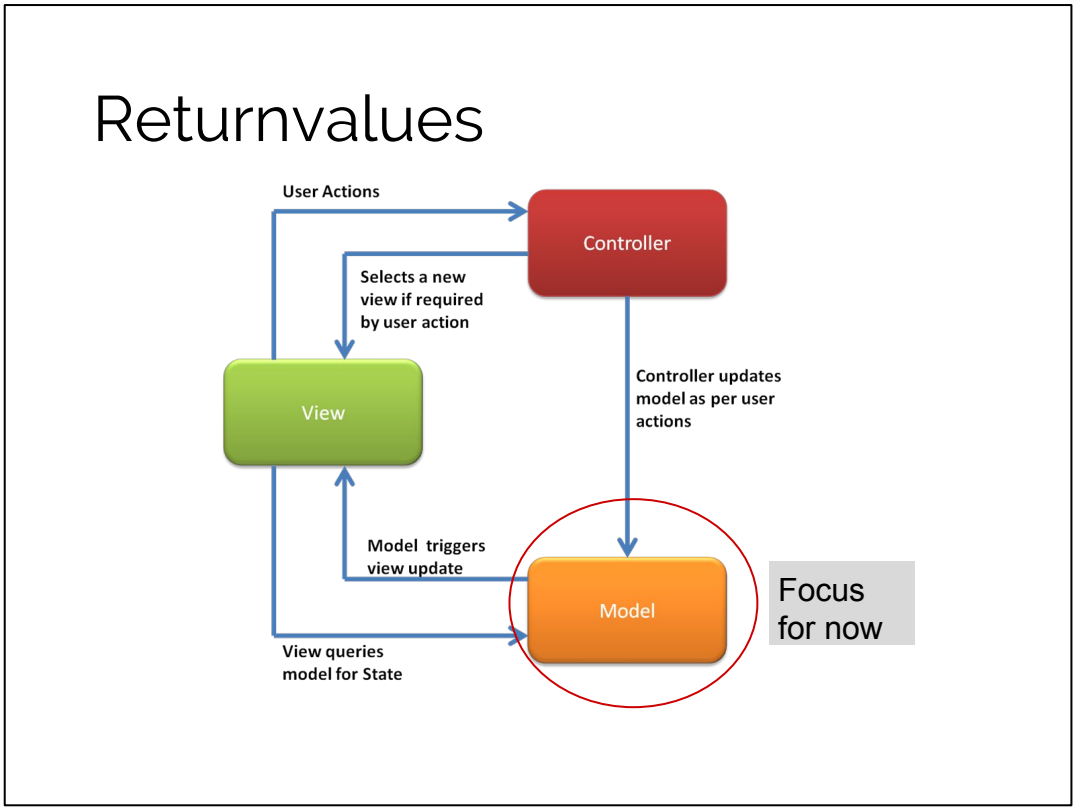
An aggregate is a cluster of classes (objects) treated as a unit

- 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

If too many methods in root, add method to return sub-aggregate with new root.

MP: We'll treat the complete model as an aggregate

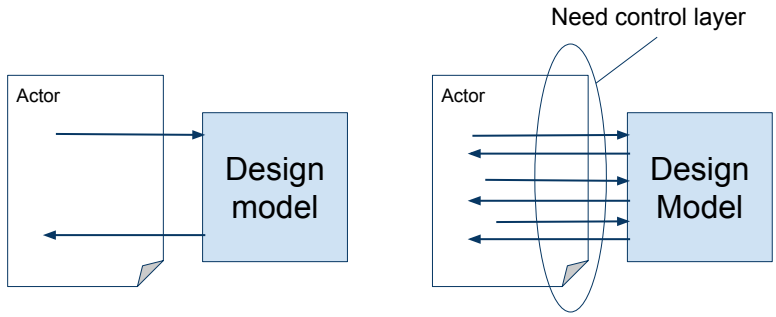
- All calls will go through Monopoly object
- Not an universally valid decision, there may be other ways to group (in other applications).



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

- Have to keep in mind.
  - Should methods have return values or should it be handled by observer?
- Until now we have only worked with the model …
  - … except the view (GUI) sketches
- NOTE: Methods with return values much easier to test!

UCs and Methods?

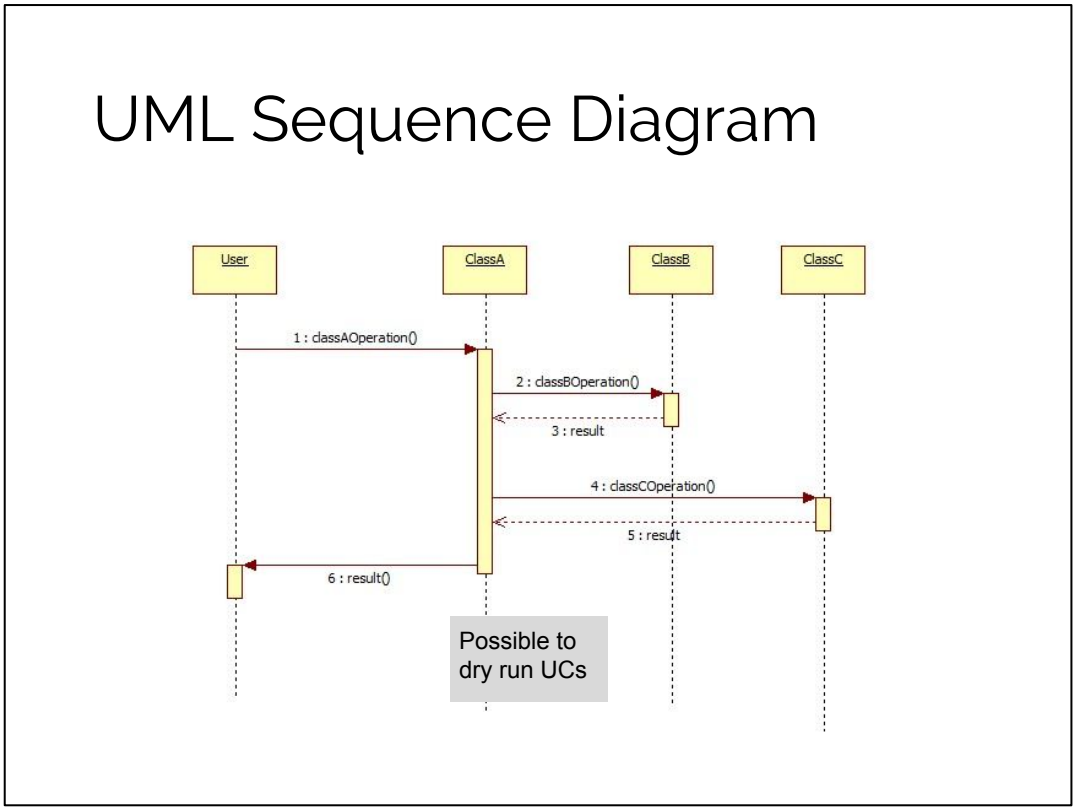


No interaction in use case (besides starting it)

Interaction in use case

If …

- No user interaction in use case (actor just starts off)
  - 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 return values and act upon.



A sequence diagram is used to describe a (the) dynamic behaviour of interacting objects

Dry run is last step before start implementing.

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

For some use case(s) and the domain model

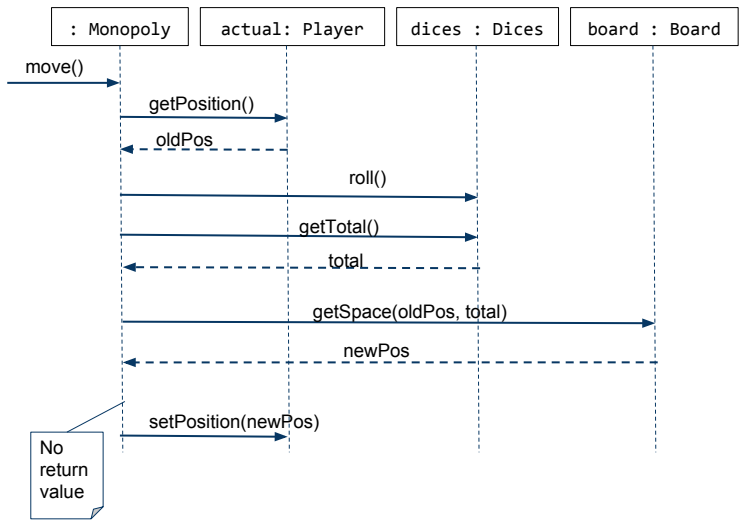
- Create an UML sequence diagram

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

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

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

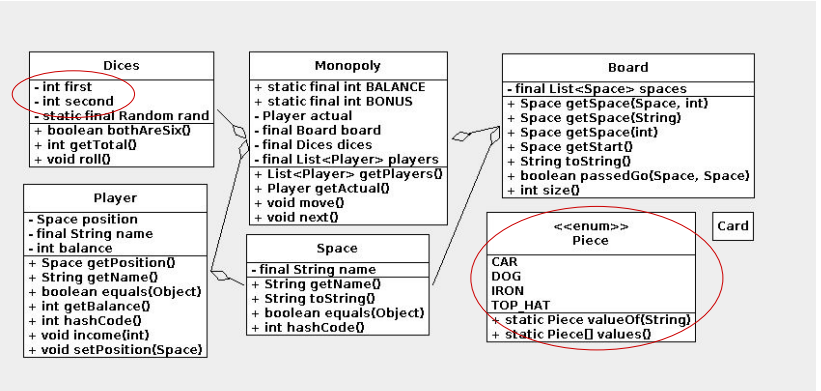
MP : Dry Run UC Move



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

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

MP : Design Model



OK to auto generate domain model (only)!

Data Representation

```
Spaces[] board = ...

Spaces[][] board = ...

List<Space> board = …

Map<String, Space> board = ...
```

In general

- Arrays only for fixed positions, else Collections

Implementing 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)

Construct Model

```
public class Monopoly {
  private final List<Player> players;
  private final Dices dices;
  private final Board board;

  public Monopoly(List<Player> players,
       Board board, Dices dices) {
      this.dices = dices;
      this.board = board;
      this.players = players;
      ...
   }
}
```

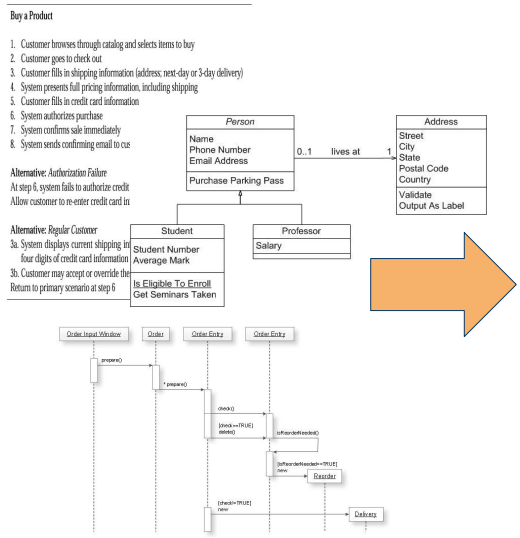
Use Constructors! Pass in dependencies

```
public class Monopoly {
  private final List<Player> players = new ArrayList<>();
  private final Dices dices = new Dices();
  private final Board board = new Board();
    ...
}
```

Build all objects needed to create model in single location

- No “new” spread out all over the application like …
- … A constructs B constructs constructs D …

Implementation

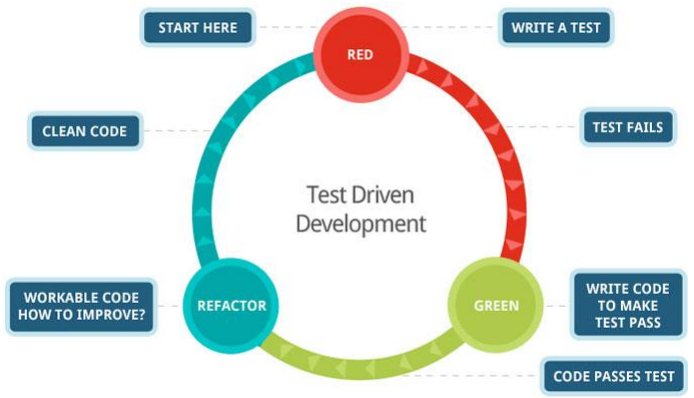



```
public class Board {

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 try to 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
- Later: Being able to test certain techniques (snippets) also very useful

Tech talk

- We always keep test code separated from the application
  - Folder: test in Maven
- The package structure for test should be the same as the structure for the

- application, more to come ...

Test as Documentation

```
@Test
public void testMoveAndPassGo() {
   MockDices md = new MockDices(35);
   //Using constructor to pass in proxy for real dices
   Monopoly m = new Monopoly(ps, b, md);

   ...

   assertTrue(m.getActual().getBalance()
           == Monopoly.BALANCE + Monopoly.BONUS);
}
```

Tests are very good documentation.

- Use long explaining names for tests

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!

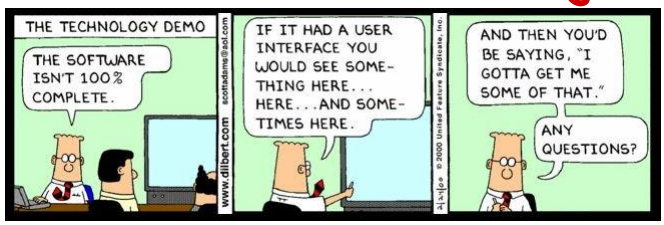
- If some class participating in the UC is complex first test it ..
- … senseless to try to implement a UC if not the "pieces" are working!

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

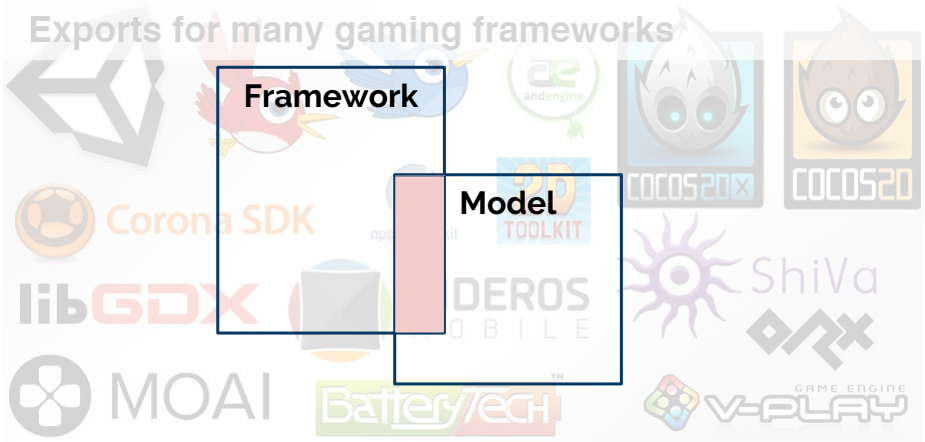


MP : Iteration 1

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)

Using Frameworks



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

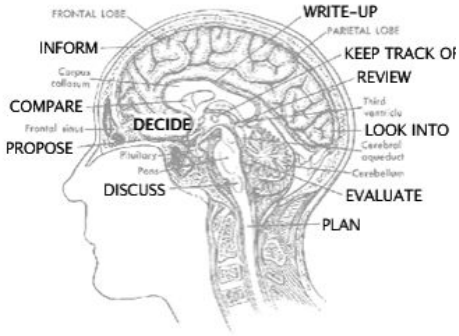
- Example:
  - 2D Position (no x and y in model classes)
  - Collision detection
  - Movement/Physics/Rendering in 3D game frameworks
- Exclude from model parts handled by framework
  - If so: Can't (don't need to) test those parts
- Test what's not handled by framework
  - Possible have to mock (a lot)

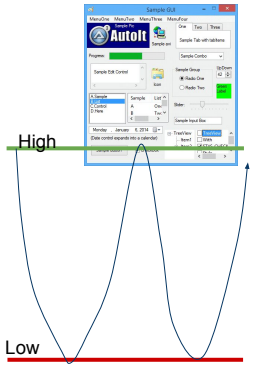
NOTE: There should always be a model, using a framework doesn't mean you may skip the model

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

- Should be possible to switch framework

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)?

Summary

We have a running model

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

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

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