TEST-DRIVEN DEVELOPMENT

TEST

PBA SOFTWAREUDVIKLING/
BSC SOFTWARE DEVELOPMENT

Christian Nielsen cnls@cphbusiness.dk

Tine Marbjerg tm@cphbusiness.dk

SPRING 2019

TODAY'S TOPICS

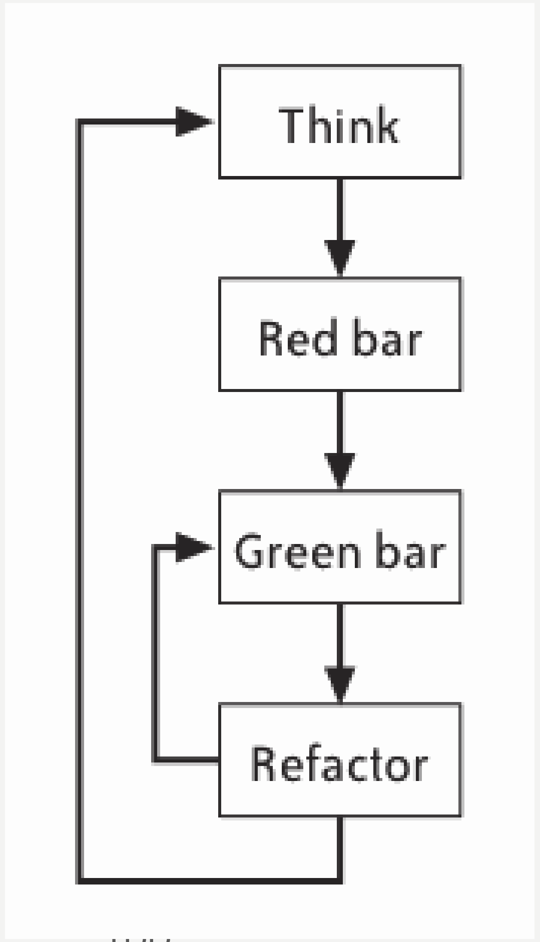
- **TDD**
  - Principles of TDD
- **TDD Classic style**
  - State-based testing
- **TDD Mockist style**
  - Behavior-based testing
- **Configuration (Managing dependencies)**

WORKSHOP GOALS

- Practice getting into the TDD mindset **by example**
- **Use mock objects** to support TDD cycle
  - Create any required mock objects
  - Create any real objects, including the target object
  - Specify how you *expect* the mock objects to be called by target object
  - *Assert* that any resulting values are valid and that all the expected calls have been made

TDD – A THREE-PHASE CYCLE (THINK PHASE NOT OFFICIAL ☺)

- Make a failing test
- Make it pass
- Refactor



TDD –THE PURPOSE

- Drive the **design** of the code
  - Make it decoupled and testable: *clean code that works!*
- **Refactoring** stage is crucial to the technique's success because principles of good design are applied:
  - Remove duplication
  - The Boy Scout Rule : *Leave your code better than you found it.*
  - Automated tests provide a safety net
- Unit test **behavior**, not methods
  - Makes object responsibility more understandable
    - A test called `testBidAccepted()` tells what it does, but not what *it's for*.
    - Better test name `testAuctionSniperMakesBid()`

TDD –THE PURPOSE

Bob Martin (Agile Software Development):

- "The act of writing a unit test is more an act of design than of verification.
- It is also more an act of documentation than of verification.
- The act of writing a unit test closes a remarkable number of feedback loops, the least of which is the one pertaining to verification of function".

TDD CLASSIC STYLE

- Order of tests
  - Degenerate case
  - One or a few happy paths test
  - Tests that provide information and knowledge
  - Error handling and negative tests

TDD CLASSIC STYLE

- Red- to Green-bar Strategies
  - Fake It
  - Obvious Implementation
  - Triangulation

TDD TECHNIQUE: FAKE IT!

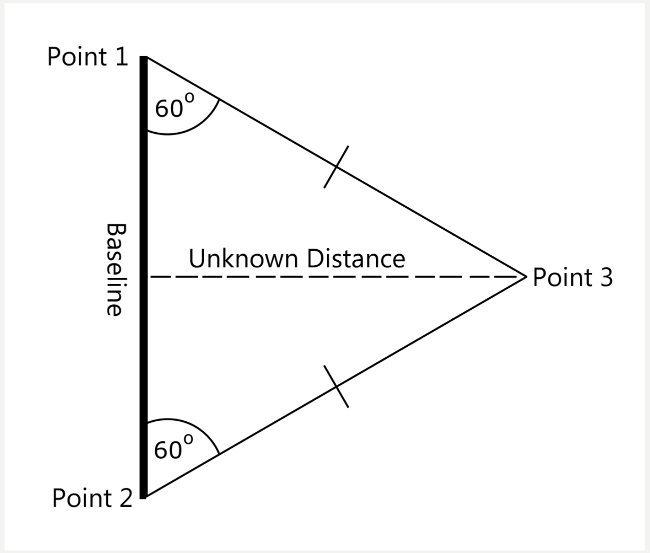
- Test:

```
@Test
public void testFibonacci() {
   assertEquals(0, fib(0));
}
```

- Fake Implementation:

```
public static int fib(int n) {
   return 0;
}
```

TDD TECHNIQUE: TRIANGULATION 1



- Use a sequence of test examples and generalize the solution until you cover just enough test cases to produce the general solution.

- Test:

```
@Test
public void testFibonacci() {
   assertEquals(0, fib(0));
   assertEquals(1, fib(1));
}
```

- Implementation:

```
public static int fib(int n) {
  if (n == 0) return 0;
  return 1;
}
```

TDD TECHNIQUE: TRIANGULATION 2

- Test:

```
int cases[][] = {{0, 0}, {1, 1}, {2, 1}};

@Test
public void testFibonacci() {
   for (int i = 0; i < cases.length; i++) {
        assertEquals(cases[i][1], fib(cases[i][0]));
   }
}
```

- Implementation:

```
public static int fib(int n) {
  if (n == 0) return 0;
  if(n <= 2)  return 1;
  return 2;
 }
```

TDD TECHNIQUE: TRIANGULATION 3

- Test:

```
 int cases[][] = {{0, 0}, {1, 1}, {2, 1}};

 @Test
 public void testFibonacci() {
    for (int i = 0; i < cases.length; i++) {
         assertEquals(cases[i][1], fib(cases[i][0]));
    }
 }
```

- Implementation:

```
public static int fib(int n) {
  if (n == 0) return 0;
  if(n <= 2)  return 1;
  return 1 + 1;
 }
```

TDD TECHNIQUE: TRIANGULATION 4

- Test:

```
int cases[][] = {{0, 0}, {1, 1}, {2,1}, {3,2}};

@Test
public void testFibonacci() {
   for (int i = 0; i < cases.length; i++) {
        assertEquals(cases[i][1], fib(cases[i][0]));
   }
}
```

- Implementation:

```
public static int fib(int n) {
  if (n == 0) return 0;
  if(n <= 2) return 1;
  return fib(n-1) + fib(n-2);
}
```

TDD CLASSIC STYLE

- Discussion
  - How small steps?
  - What should be refactored?
  - Test first or test last?

TDD MOCKIST STYLE

Verifying behavior between objects

- Arrange
- Act
- **Assert**

Check the return value, or an exception
Check the state of the object, or the state of a collaborator
Check the object correctly interacts with a collaborator

COMMUNICATION OVER CLASSIFICATION

- Specification of **interface**
  - Interfaces describes whether components *fit* together

- Definition of **communication protocol**
  - Protocol describes whether they will *work* together

TDD wih mock objects

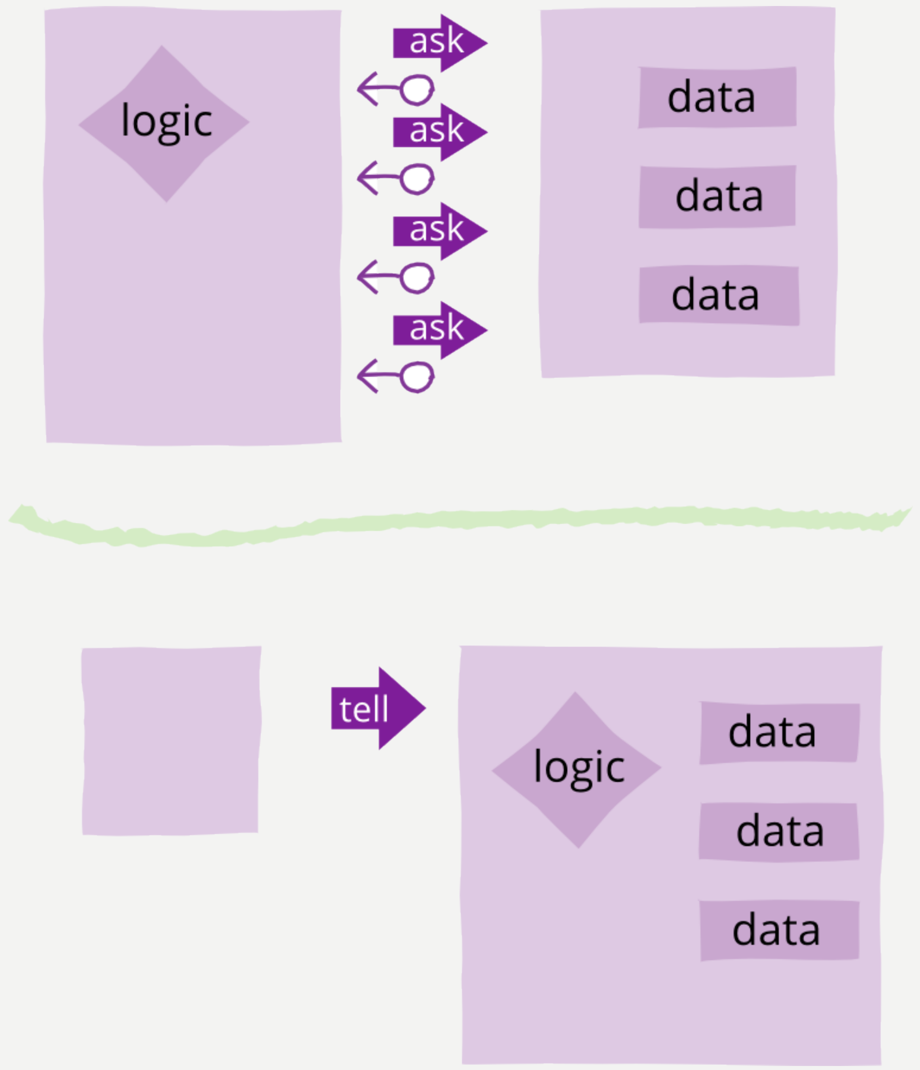
TDD MOCKIST STYLE – PRINCIPLE 1

- **"Tell, Don't Ask"** Object Oriented Design
    - Commands, i.e. instructions to do something and return result

TELL, DON'T ASK

- Principle that helps us remember that OO is about bundling data with the functions that operate on that data.

- It encourages to move behavior into an object to go with the data



IDEA IS TO AVOID ‘TRAIN WRECKS’

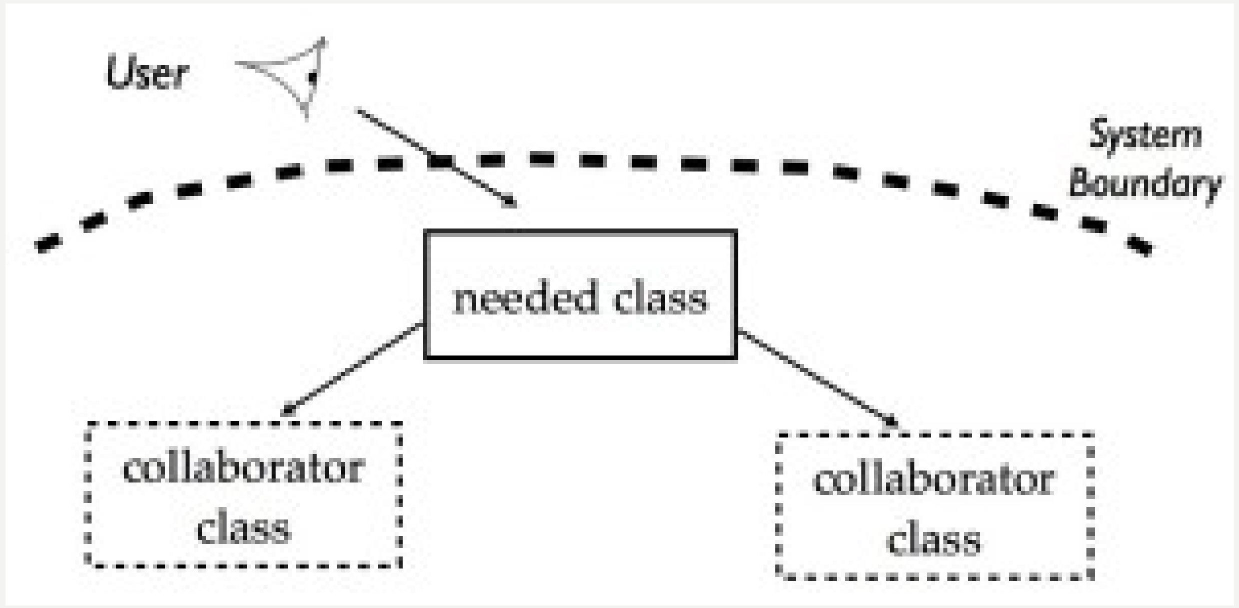
- Law of Demeter: For all classes C, and for all methods M attached to C, all objects to which M sends a message must be
    - M’s argument objects, including the self object or
    - The instance variable objects of C

🚫

```
car.getOwner().getAddress().getStreet();
```

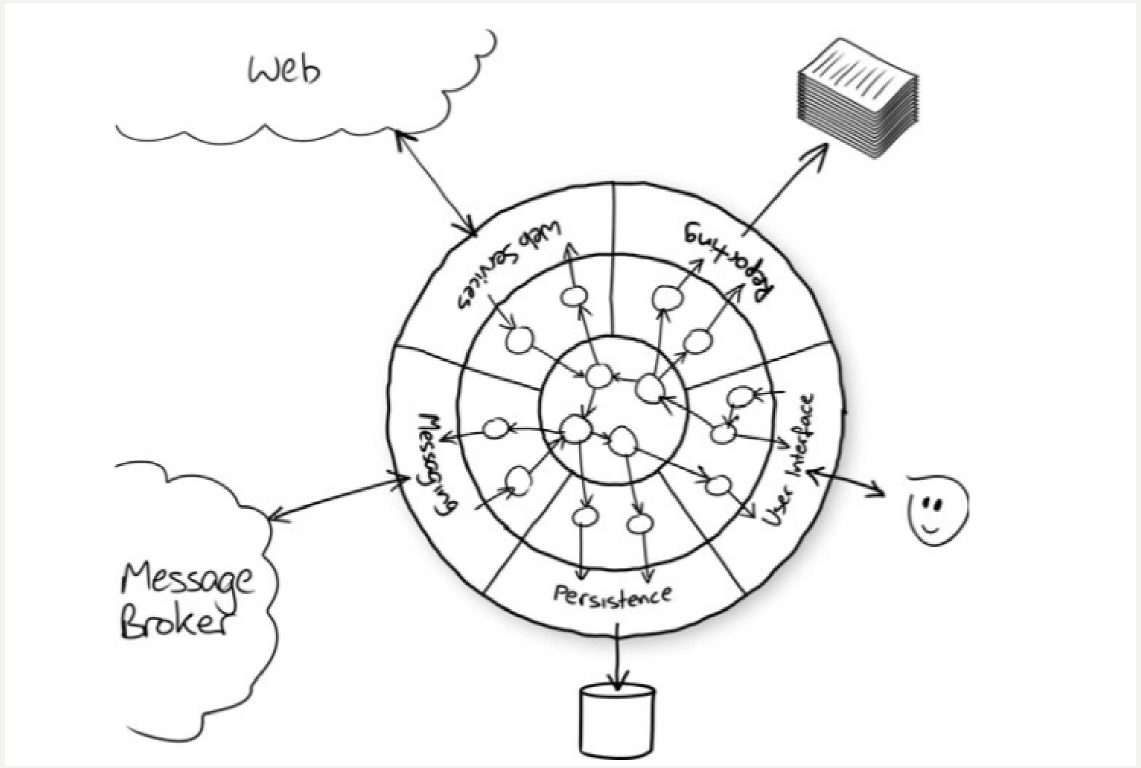
TDD MOCKIST STYLE PRINCIPLE 2

- Designing Outside-In



PRINCIPLES FOR MAINTAINABILITY

- Separation of concerns
- Higher level of abstraction



Domain model mapped onto technical infrastructure

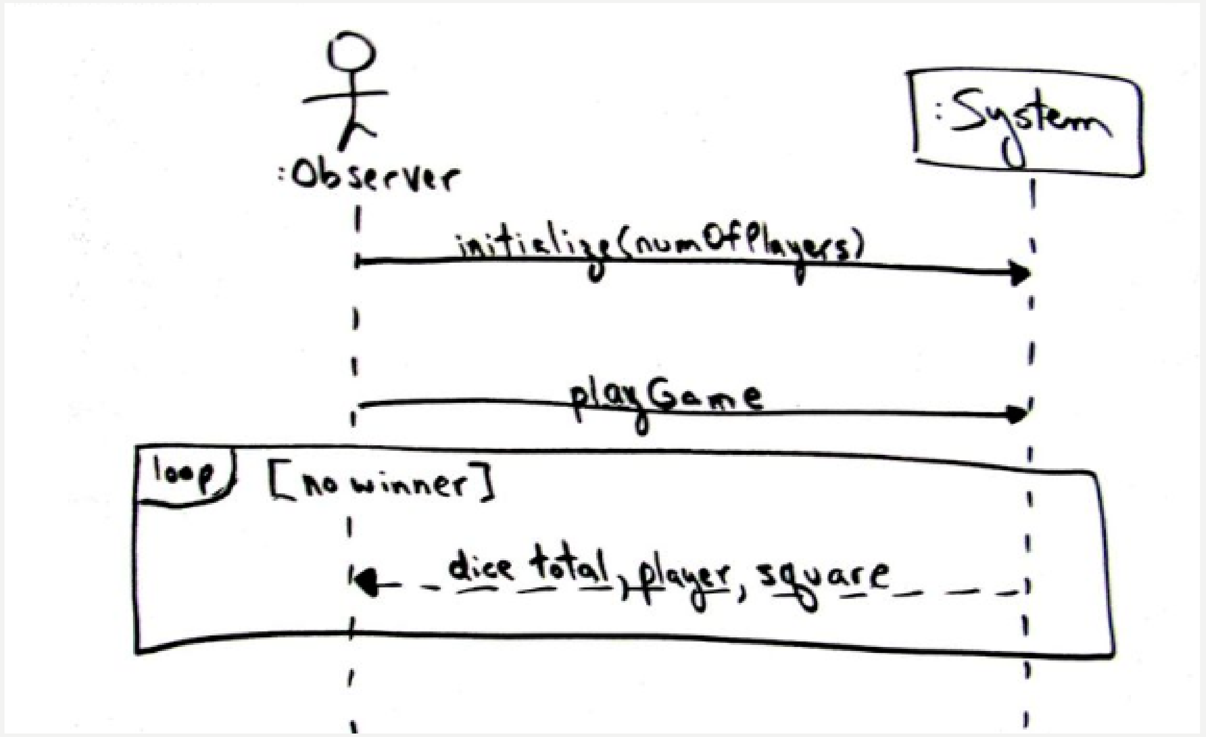
MONOPOLY –AN EXAMPLE (1)

TDD
Think first ☺

2 use cases (~ stories)

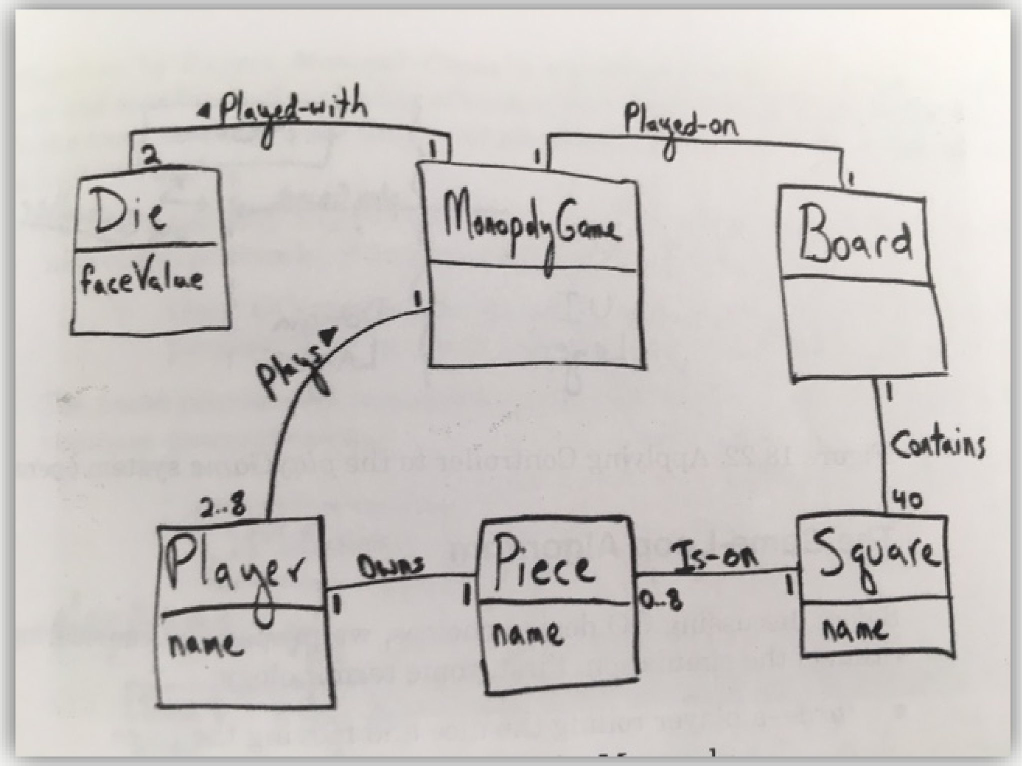
- Initialization
- Play the game

- *Game loop* Algorithm
  - *turn* – a player rolls the dice and moves the piece
  - *round* - all the players taking one turn each



MONOPOLY –AN EXAMPLE (2)

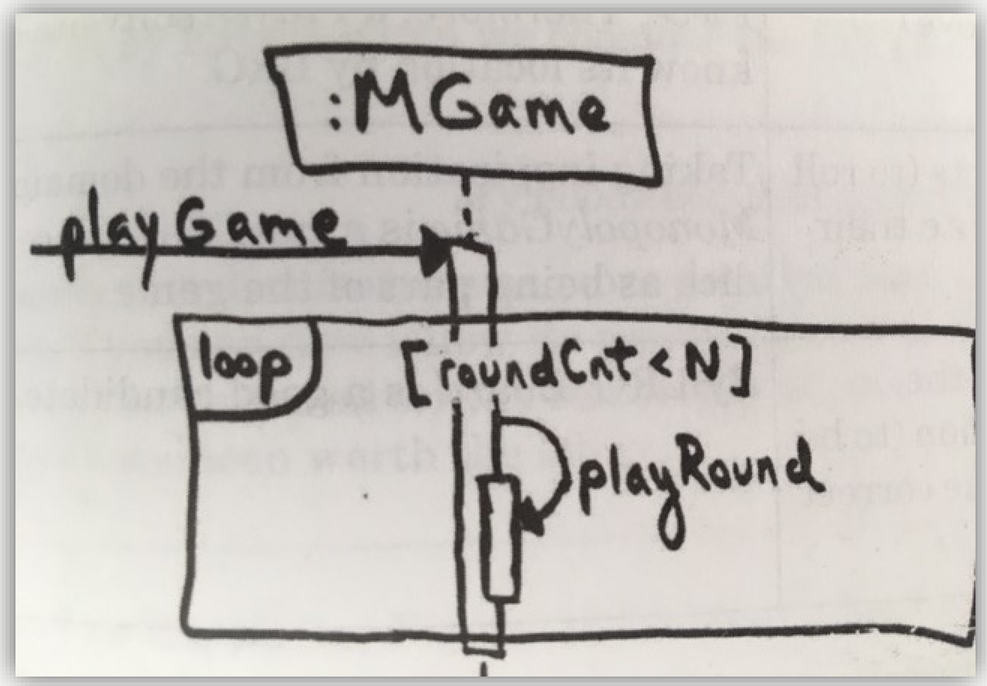
- Who should be responsible for controlling *Game loop*?
- Let's use the Domain Model to make a decision:



```
For N rounds
   For each Player p
      p takes a turn
```

MONOPOLY –AN EXAMPLE (3)

MonopolyGame knows the players, so is a good choice (is information expert)



MONOPOLY –AN EXAMPLE (4)

- Who *takes a turn*?
- Let's look at the Domain Model again

    - Candidates:
        - Player (not just because human player does the task IRL)
        - MonopolyGame
        - Board

    - Guideline
        - Sometimes we need to look ahead to make a choice
        - In that case Player seems a fit candidate. Why?

MONOPOLY –AN EXAMPLE (5)

Taking a Turn involves:

1. Calculating random number total between 2 and 12 (range of two dice)
2. Calculating the new square location
3. Moving the player's piece from old location to a new square location

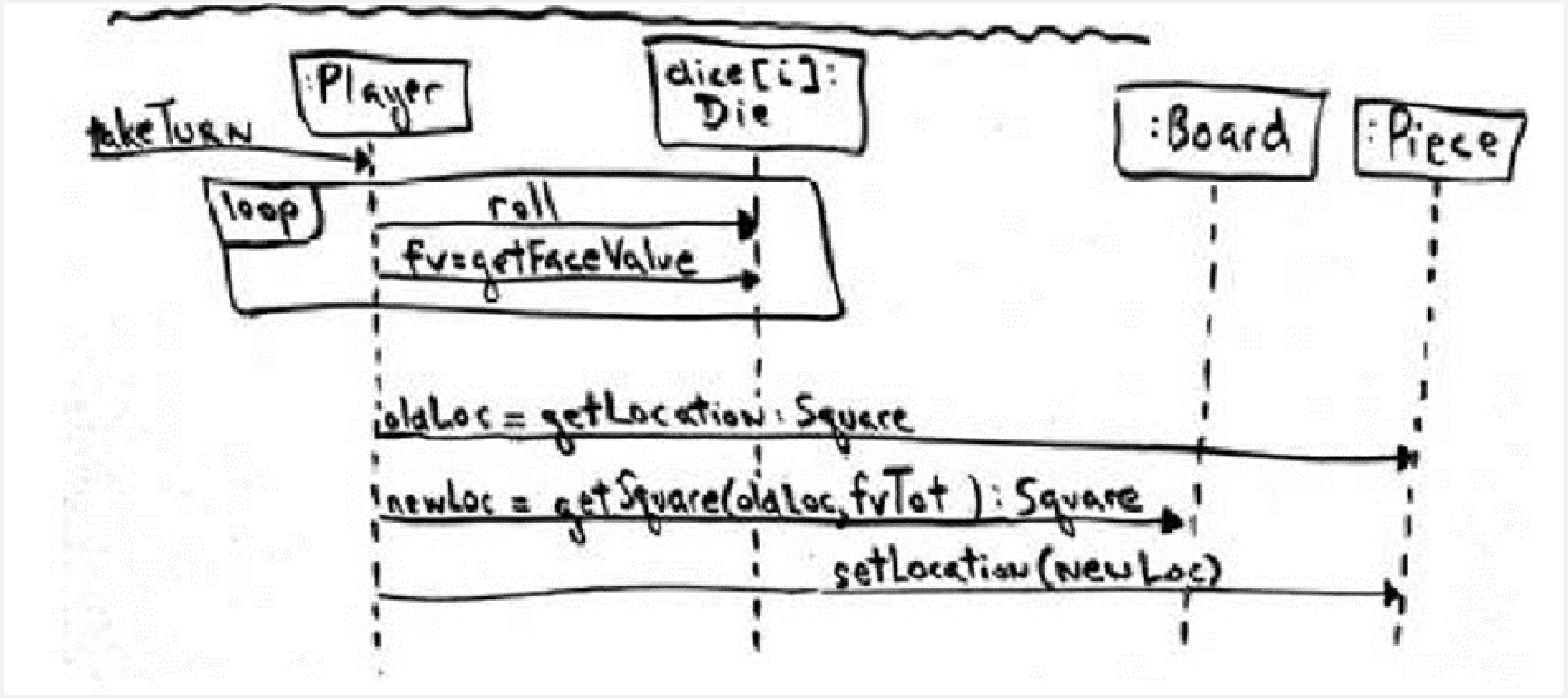
Random number problem: Die has face value and can roll

Calculate new square problem – Board knows its squares. Given an old square location + offset (dice total) → board can be responsible for knowing new location

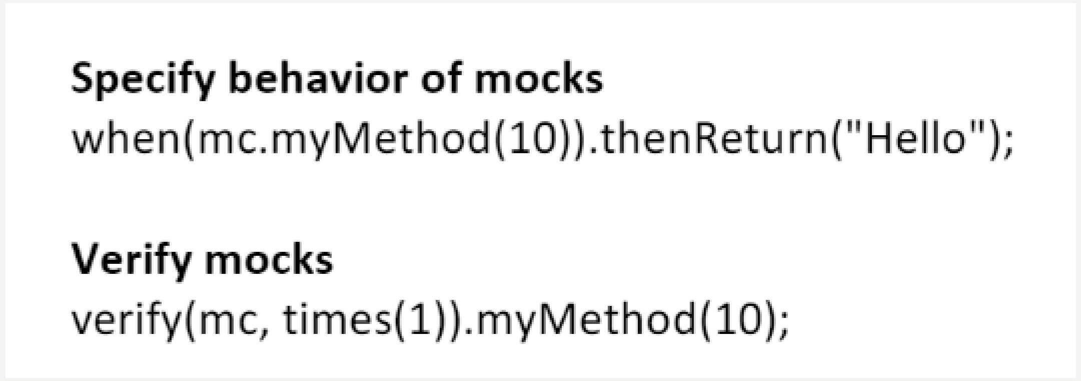
Piece movement problem – Player knows its piece, and piece knows its location. So piece will set its new location, but could receive new location from player

MONOPOLY –AN EXAMPLE (6)

How *Taking a Turn design* looks in a sequence diagram:



MONOPOLY –AN EXAMPLE (7)

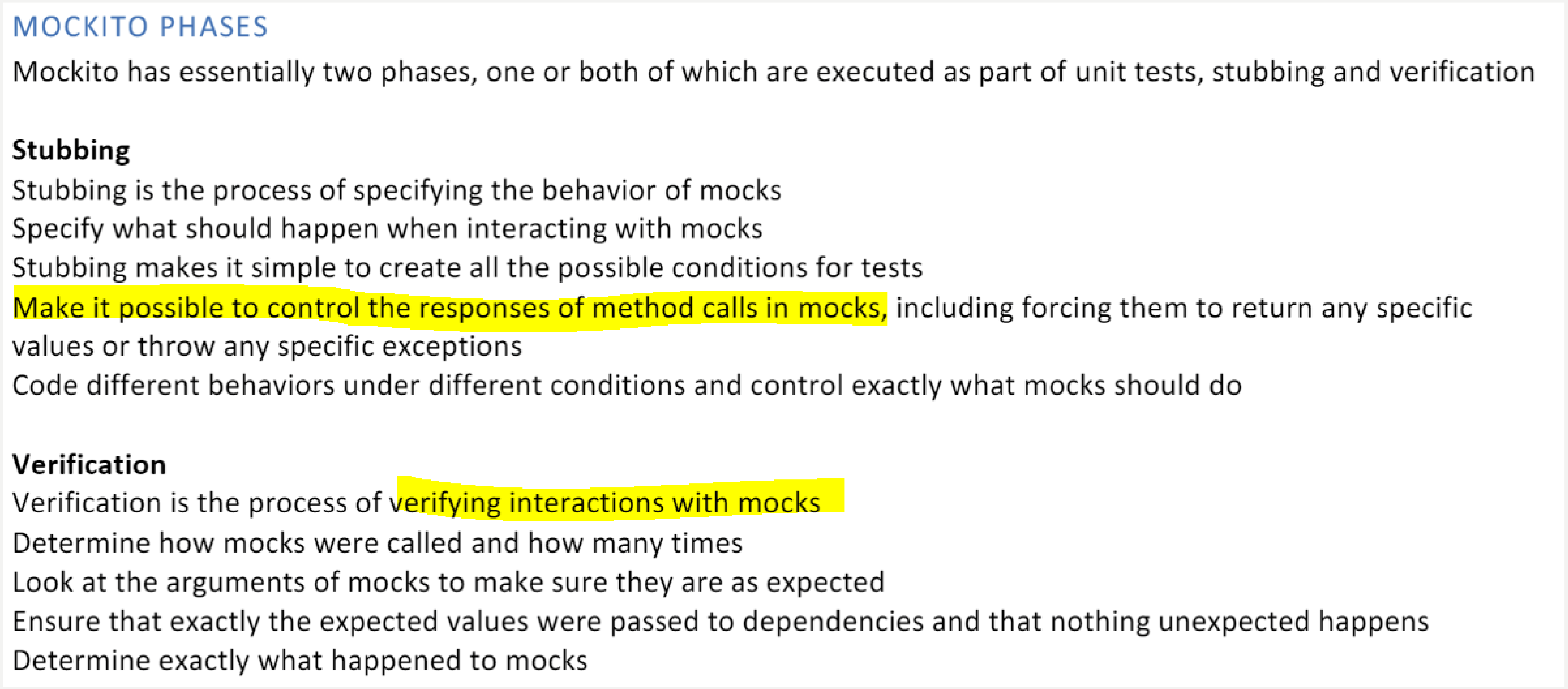

Specify behavior of mocks

```
when(mc.myMethod(10)).thenReturn("Hello");
```

Verify mocks

```
verify(mc, times(1)).myMethod(10);
```

MOCKITO PHASES



MOCKITO PHASES

Mockito has essentially two phases, one or both of which are executed as part of unit tests, stubbing and verification

Stubbing

Stubbing is the process of specifying the behavior of mocks
Specify what should happen when interacting with mocks
Stubbing makes it simple to create all the possible conditions for tests
Make it possible to control the responses of method calls in mocks, including forcing them to return any specific values or throw any specific exceptions
Code different behaviors under different conditions and control exactly what mocks should do

Verification

Verification is the process of verifying interactions with mocks
Determine how mocks were called and how many times
Look at the arguments of mocks to make sure they are as expected
Ensure that exactly the expected values were passed to dependencies and that nothing unexpected happens
Determine exactly what happened to mocks

MONOPOLY –AN EXAMPLE (8)

- We must remember The *Refactoring* step in TDD

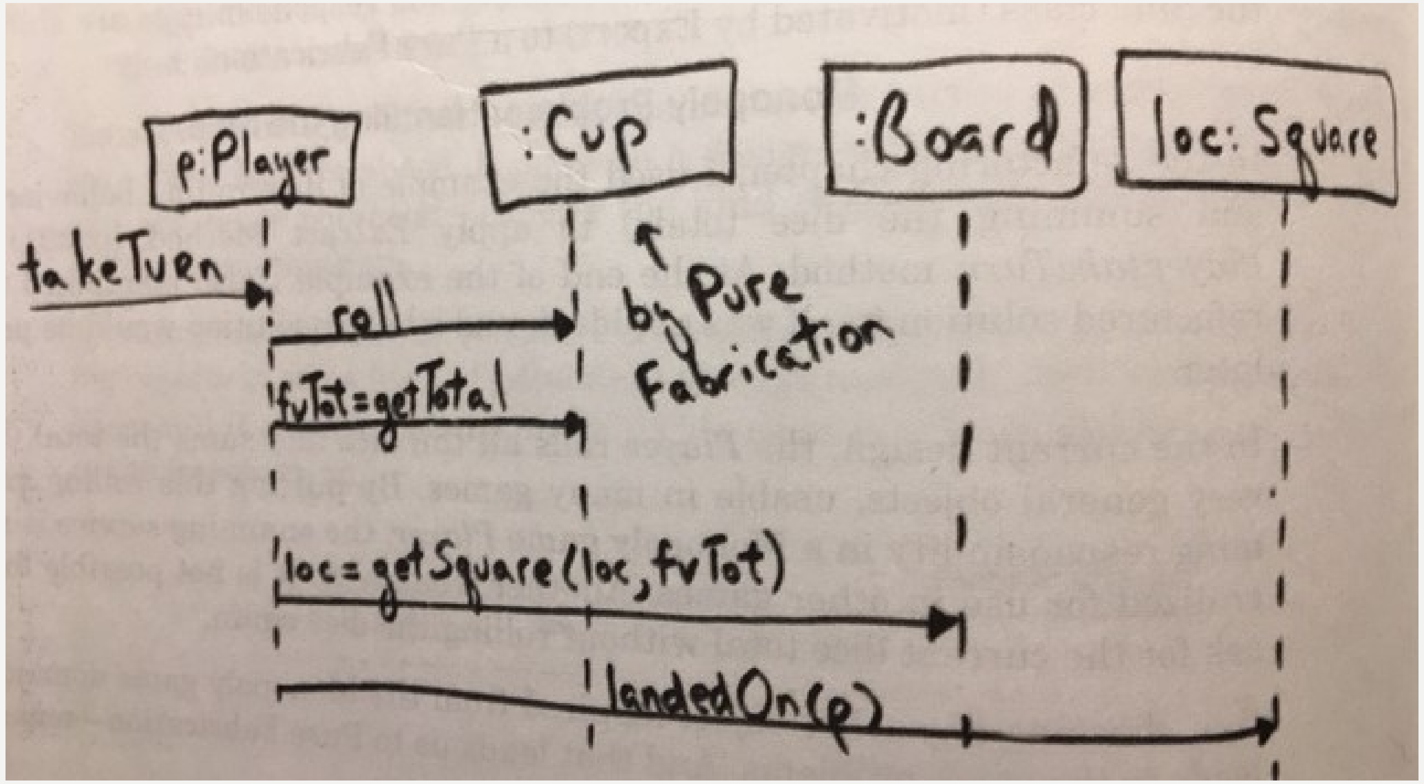
- Can we get higher cohesion by using Extract Method?

```
public void takeTurn() {
  int rollTotal = 0;

   for (Die die : dice) {
      die.roll();
      rollTotal += die.getFaceValue();
  }

  Square newLocation = board.getSquare(piece.getLocation(), rollTotal);
  piece.setLocation(newLocation);
}
```

MONOPOLY –AN EXAMPLE (9)

- Problems
  - How about reuse of the dice (in other apps)?
  - It is not possible to ask for current dice total without rolling again?

s
o
l
u
t
i
o
n



CLASSIC OR MOCKIST?

- It depends
- When using mockist style, and add more tests, you can switch strategy at the fringes

DOUBLE LOOP TDD MOCKIST STYLE

- How does the program (feature) work *as a whole?*

- We need another feedback loop of automated acceptance tests
  - unit tests are preceded by automated acceptance tests (which require the entire infrastructure and deployment process of the feature to be in place)
  - Clarifies WHAT to do with no underlying tech focus

- End-to-end tests can mean different things to different applications, for instance
  1. Start framework or container (web service)
  2. Deploy/start the endpoint
  3. Post details to the endpoint
  4. **Verify result**
  5. Shut everything down

DEPENDENCY INJECTION - CONFIGURATION

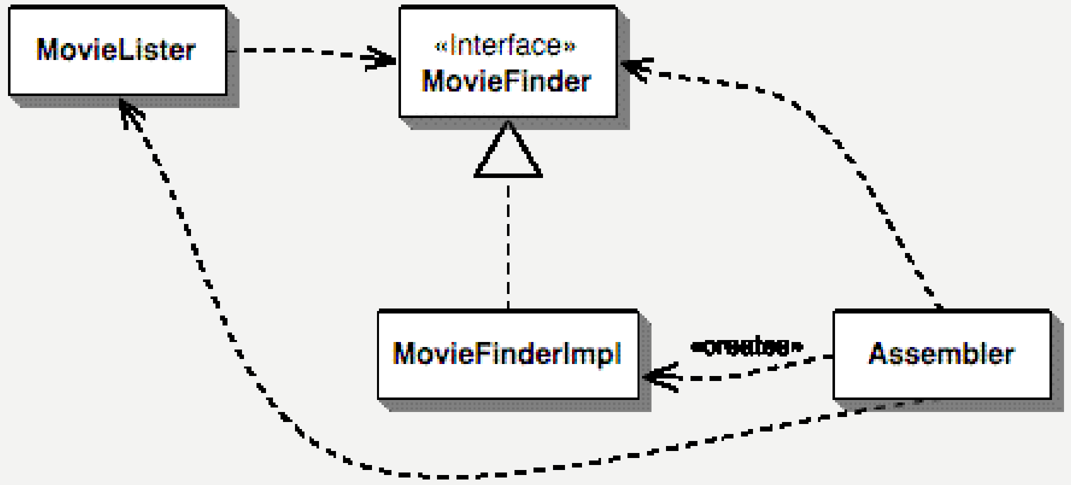
- Dependency injection can be used to **externalize** a system's **configuration** details into configuration files, allowing the system to be reconfigured without recompilation.

- Separate configurations can be written for different situations that require different implementations of components. This includes, but is not limited to, testing.

Source: https://en.wikipedia.org/wiki/Dependency_injection

DEPENDENCY CONFIGURATION – THE ASSEMBLER

- Basic idea of Dependency Injection is to have a separate object, an assembler (i.e. injector), that populates a field in a class with an appropriate implementation for the interface



CONTAINERS

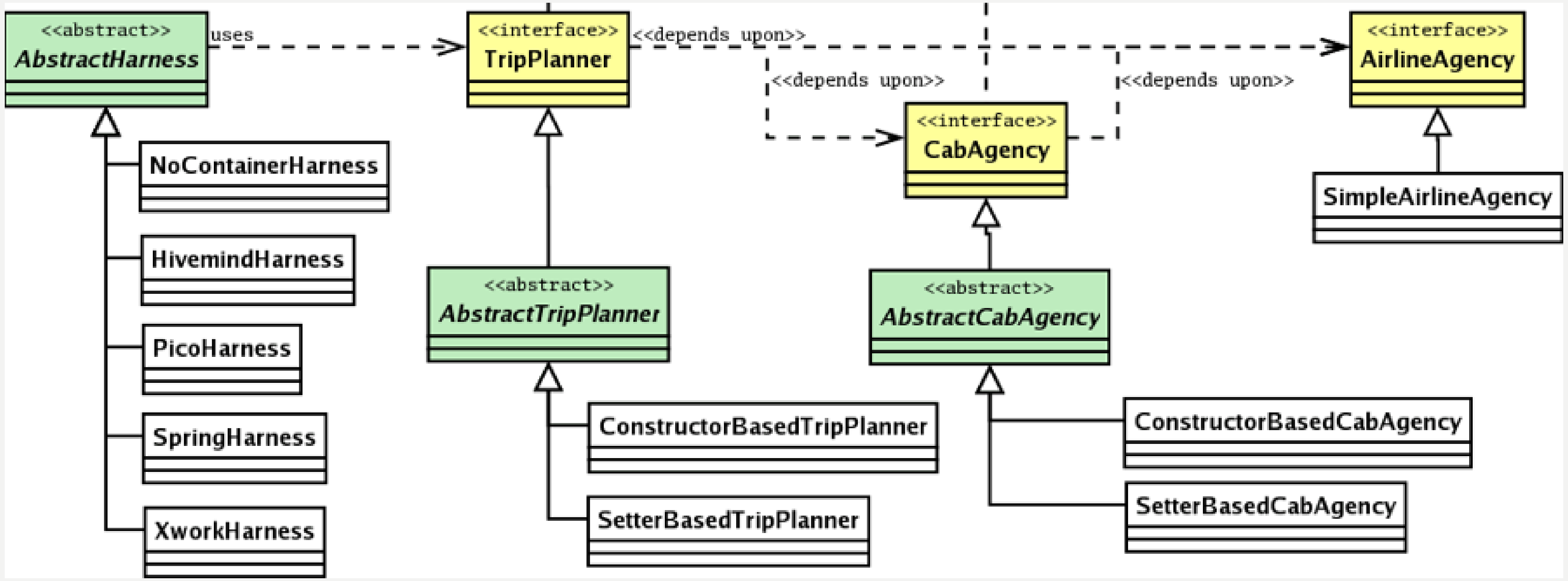
Normally you use Inversion Of Control (IOC) containers/frameworks to handle DI.

Examples:

- Spring
- PicoContainer
- Autofac (.NET)
- …

A "NO CONTAINER" EXAMPLE 1

A trip planner depends upon a cab agency and an airline agency to plan a trip



Source: https://www.theserverside.com/news/1321158/A-beginners-guide-to-Dependency-Injection

A "NO CONTAINER" EXAMPLE 2

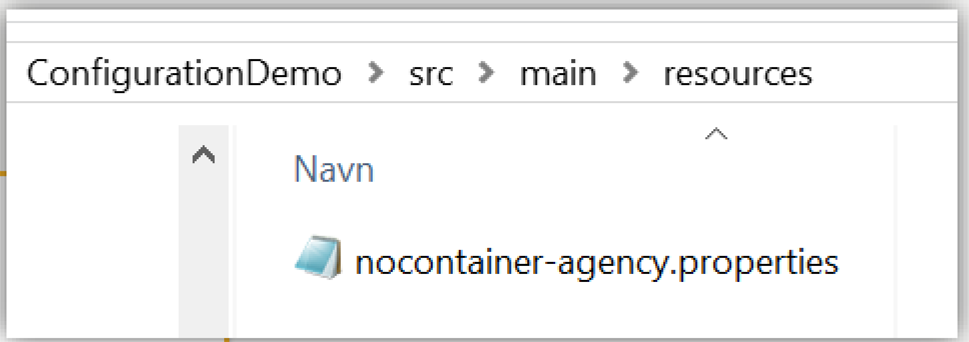
Property file defines actual implementations to be used

```
airline-agency-class = tdddemo.SimpleAirlineAgency
cab-agency-class = tdddemo.ConstructorBasedCabAgency
trip-planner-class = tdddemo.ConstructorBasedTripPlanner
```

A "NO CONTAINER" EXAMPLE 3

POM file specifies where to look for property file

```
 …
<resources>
        <resource>
            <directory>src/main/resources</directory>
            <filtering>true</filtering>
        </resource>
    </resources>
  </build>
```



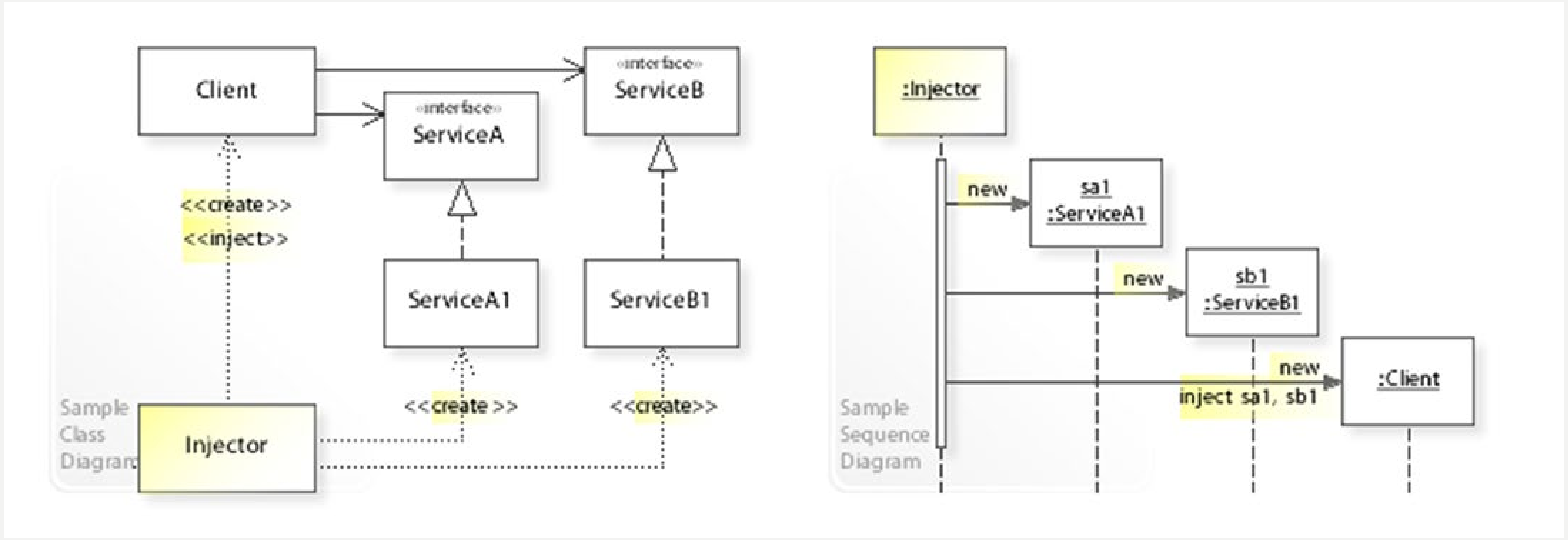
ConfigurationDemo › src › main › resources

Navn

nocontainer-agency.properties

A "NO CONTAINER" EXAMPLE 4

UML **Class diagram** & **Sequence diagram** illustrate dependency structure and run-time interactions for manual construction of dependencies (wiki)



A "NO CONTAINER" EXAMPLE 5

Assembler code

```
Properties prop = new Properties();
prop.load(this.getClass().getResourceAsStream("/nocontainer-agency.properties"));
String trip = prop.getProperty("trip-planner-class"); //get all properties
…
Class tripPlannerClass = Class.forName(trip); // get all classes
…
if (TripPlanner.class.isAssignableFrom(tripPlannerClass)) {
    Constructor constructor = tripPlannerClass.getConstructor(new Class[]{AirlineAgency.class, CabAgency.class});
    tripPlanner = (TripPlanner) constructor.newInstance(new Object[]{airlineAgency, cabAgency});
}
```