AARHUS UNIVERSITET

Software Engineering and Architecture

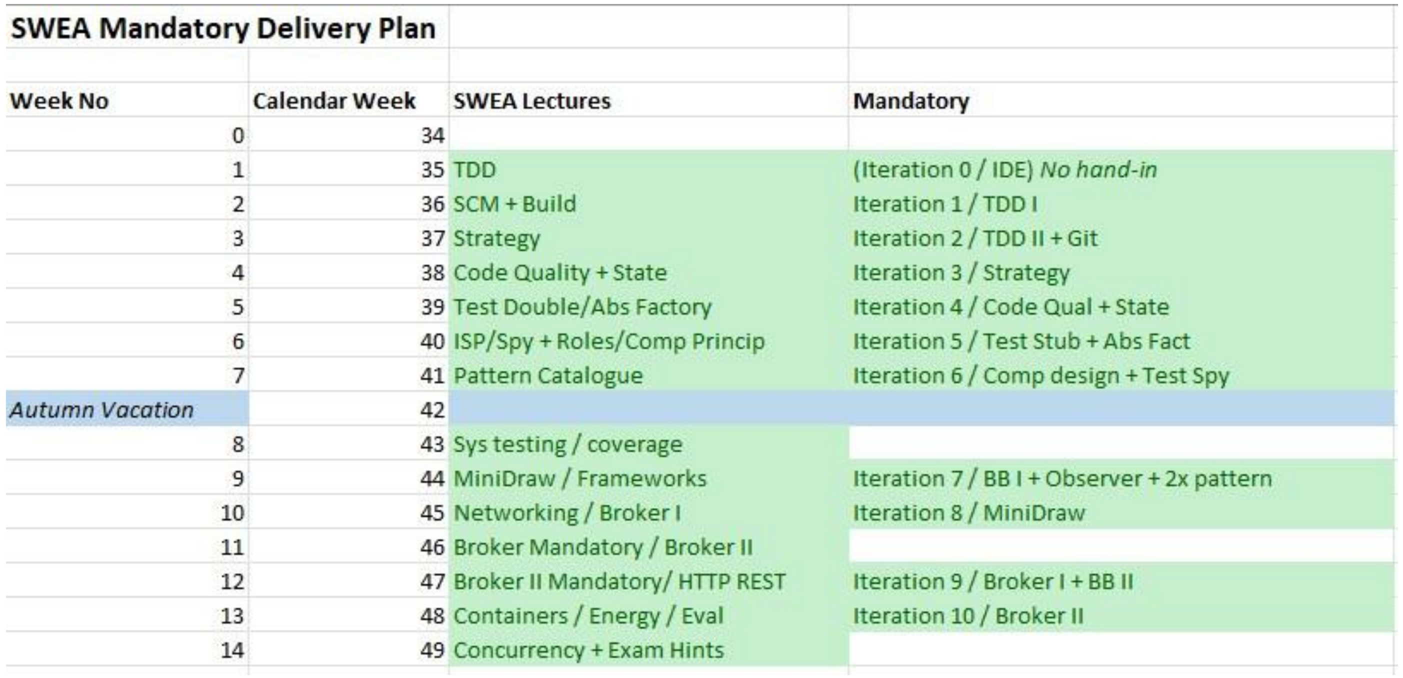
Broker I Mandatory:

First Steps on Distributed HotStone

AARHUS UNIVERSITET

Don’t Panic

- I will introduce the Broker I mandatory
- … but it is not this week’s hand-in ☺



SWEA Mandatory Delivery Plan

Week No	Calendar Week	SWEA Lectures	Mandatory
0	34		
1	35	TDD	(Iteration 0 / IDE) *No hand-in*
2	36	SCM + Build	Iteration 1 / TDD I
3	37	Strategy	Iteration 2 / TDD II + Git
4	38	Code Quality + State	Iteration 3 / Strategy
5	39	Test Double/Abs Factory	Iteration 4 / Code Qual + State
6	40	ISP/Spy + Roles/Comp Princip	Iteration 5 / Test Stub + Abs Fact
7	41	Pattern Catalogue	Iteration 6 / Comp design + Test Spy
Autumn Vacation	42		
8	43	Sys testing / coverage	
9	44	MiniDraw / Frameworks	Iteration 7 / BB I + Observer + 2x pattern
10	45	Networking / Broker I	Iteration 8 / MiniDraw
11	46	Broker Mandatory / Broker II	
12	47	Broker II Mandatory/ HTTP REST	Iteration 9 / Broker I + BB II
13	48	Containers / Energy / Eval	Iteration 10 / Broker II
14	49	Concurrency + Exam Hints	

Some Experiences

- HotStone is not difficult to ‘go distributed’, but ..
  - It was **not** designed with distribution in mind
    - But the ‘Facade’ pattern nature of Game is actually ideal
  - It contains an Observer pattern, however!
    - Which our Broker does *specifically not support !*
  - There are a lot of methods
    - Means a lot of ‘if (operationName.equals(xxx))’
  - It is multi-class and multi-object
    - Game, Card, Hero, Player, …
  - It is a non-trivial code base
    - Abstract Factory, Xstrategy, ...

Double Leap

- Making HotStone distributed is not hard...
  - ... Once you know the details in Broker
    - I solved the ‘hard parts’ in a couple of hours coding time
    - Disclaimer – I probably know Broker pretty well by now after having written the book and fumbled my way through it for a couple of years in a couple of projects...
- ... But getting into the process of *using* Broker is hard!
- We will split the work into two mandatory deliveries:

AARHUS UNIVERSITET

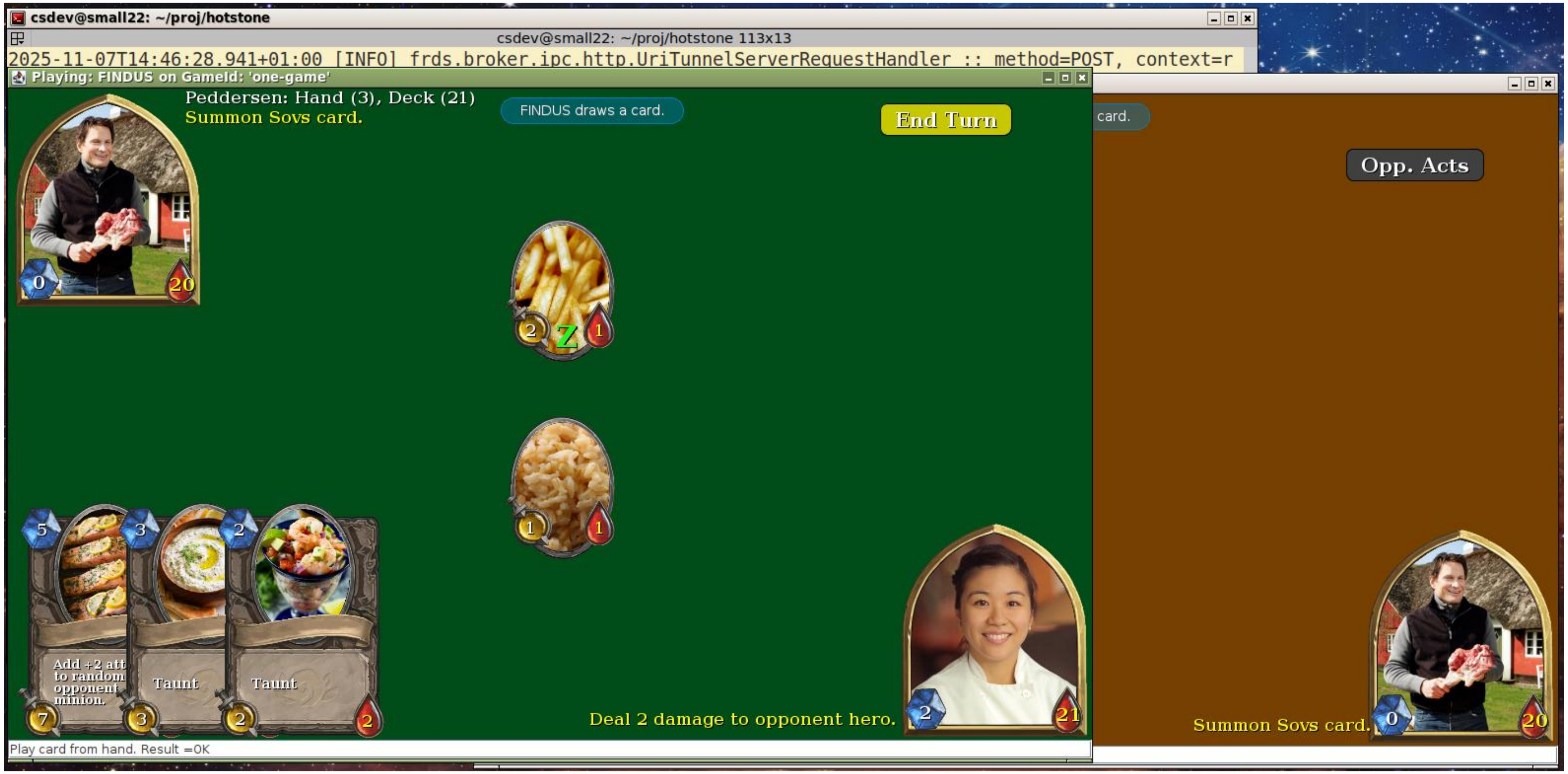
Broker I

- Learning Goal
  - Getting into the Broker pattern’s roles and implementation
  - Develop all methods that are *not handling object references*
  - Reinforced learning of using test doubles to avoid *big bang integration*
- Product Goal
  - JUnit Test suite, TDD developing ClientProxies and Invoker code
  - Integration testing, using HTTP based communication

Broker II

- Learning Goal
  - Get the *handle object reference* methods implemented
    - c = getCardInHand(…), playCard(.., c), and cousins...
  - System test: MiniDraw GUI integrated in a full client
  - Optional Refactor Invoker “Blob” into *Multi Type Dispatching*
- Product Goal
  - JUnit test suite that cover **all** broker related code
  - System testing of a *full HotStone GUI based product!*

Final HotStone System



```
csdev@small22: ~/proj/hotstone
csdev@small22: ~/proj/hotstone 113x13
2025-11-07T14:46:28.941+01:00 [INFO] frds.broker.ipc.http.UriTunnelServerRequestHandler :: method=POST, context=r
```

Playing: FINDUS on GameId: 'one-game'
Peddersen: Hand (3), Deck (21)
Summon Sovs card.
FINDUS draws a card.
End Turn
card.
Opp. Acts
Add +2 att to random opponent minion.
Taunt
Taunt
Deal 2 damage to opponent hero.
Summon Sovs card.

Play card from hand. Result =OK

AARHUS UNIVERSITET

Broker I Exercise

AARHUS UNIVERSITET

Exercises

- Broker 1.1
  - Develop much of Game’s methods
    - All those that handle simple values / no object references

```
int getTurnNumber();
```

```
int getDeckSize(Player who);
```

```
Player getPlayerInTurn();
```

- Broker 1.2
  - Develop Card and Hero methods (are all simple values)

```
int getManaCost();
```

```
String getName();
```

```
int getMana();
```

- Broker 1.3
  - Make a real *manual integration test* case using a real HotStoneGameServer – involving a client and a server

Limitations

... To lower your effort ...

You will only...

- ... Handle a *single game* on the server
  - Only one *GameServant* object
    - Thus its object id is irrelevant and no need to keep a datastructure of multiple servant objects
    - Just like the TeleMed system
  - Known as the **Singleton** design pattern
    - One, globally, accessible object, only one exists…
    - [Some consider this an anti-pattern, but…]

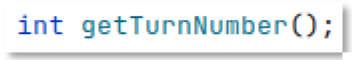
AARHUS UNIVERSITET

Broker 1.1

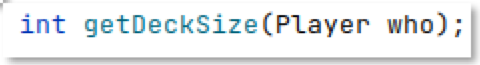
Pass by Value Game methods

Pass by Value Methods

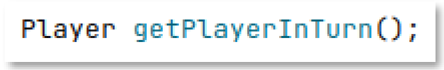
- Broker 1.1: Develop all *pass by value* methods of Game



```
int getTurnNumber();
```



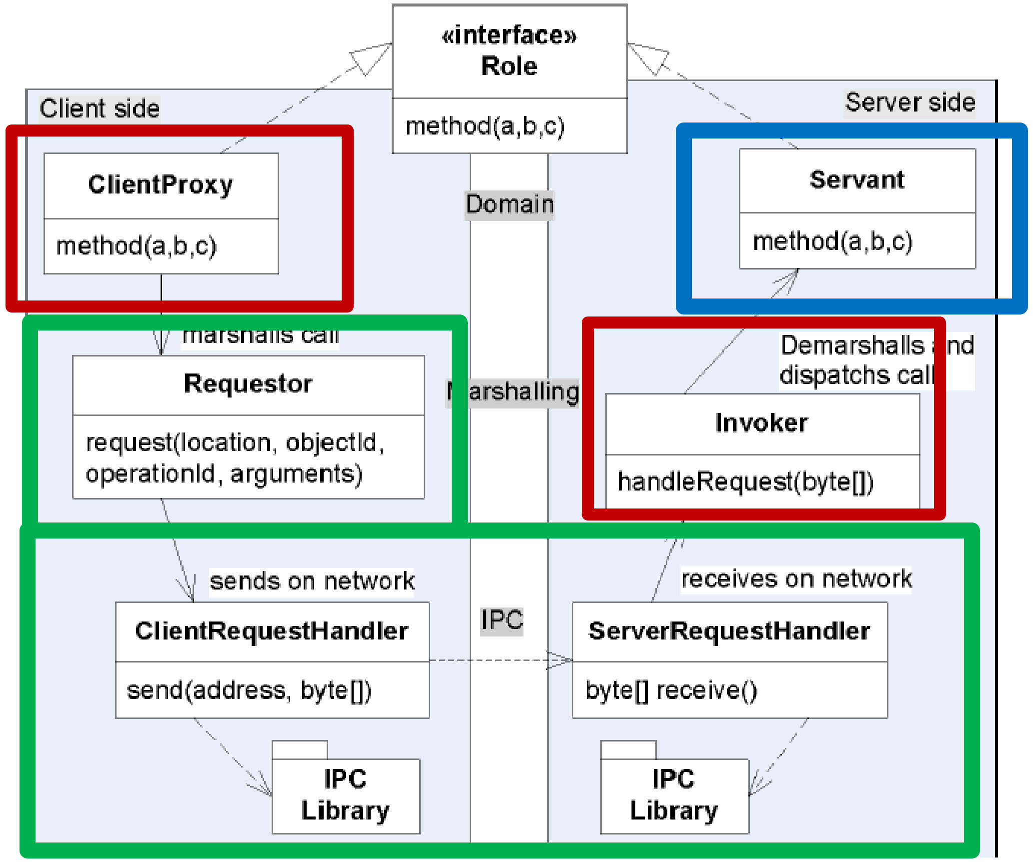
```
int getDeckSize(Player who);
```



```
Player getPlayerInTurn();
```

What to implement?

- Broker roles
  - Most are reused!
- Use the JSON marshalling (Gson)
- Use the HTTP IPC
- Use Doubles/Your Game
- Missing are
  - ClientProxies
  - Invoker(s)

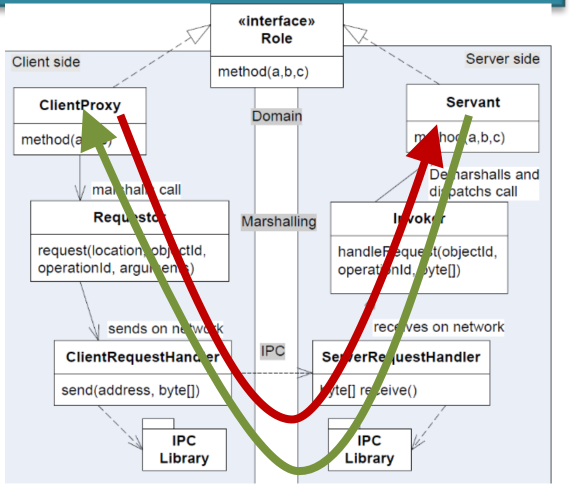


Staring at the Screen

- How the h... do you start this exercise?
  - I stared at the screen with a very blank expression for 10 minutes

- The starting point is to *establish the full broker chain of roles...*

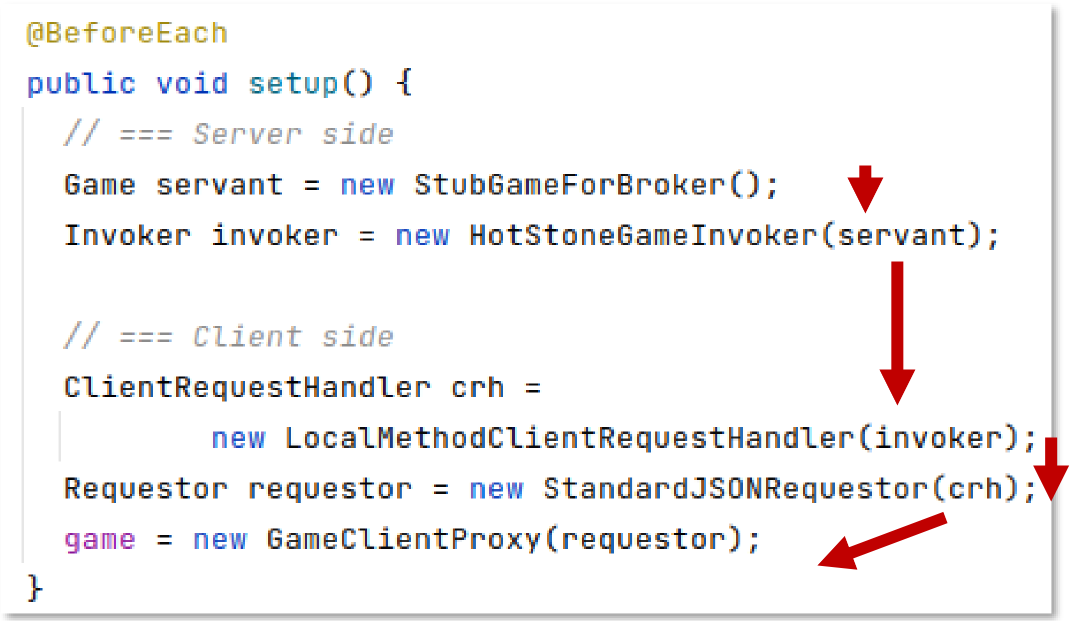
  - *(So the template code provides it for Game, make your own for Card and Hero.)*
  - *hotstone.broker.TestGameBroker*



AARHUS UNIVERSITET

So, First thing to do is...

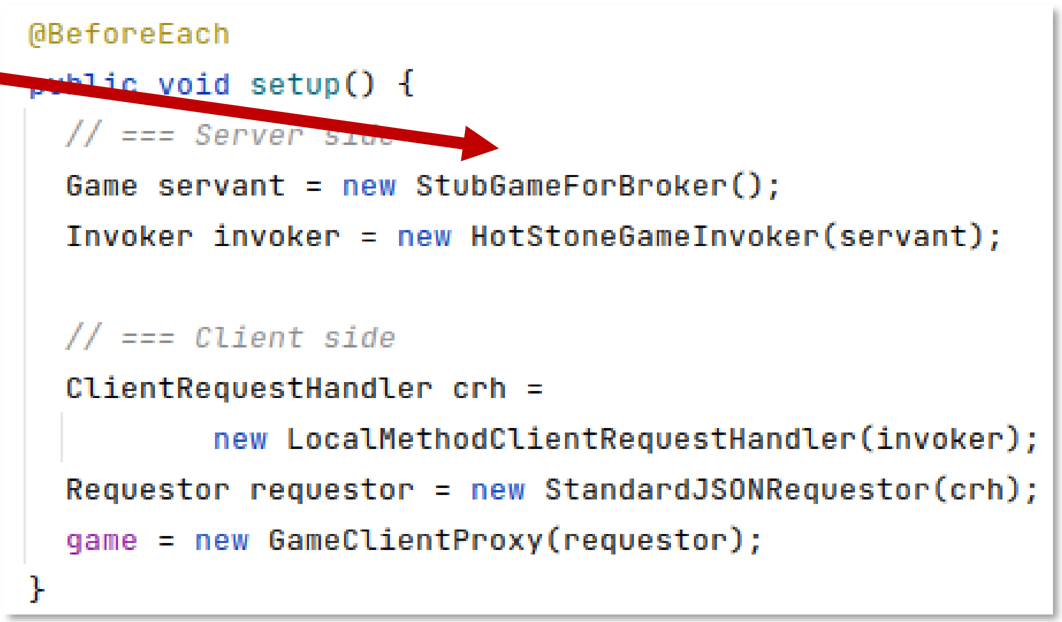
- Make the @BeforeEach method that sets up the chain / dependency inject roles
- Example:



```
@BeforeEach
public void setup() {
  // === Server side
  Game servant = new StubGameForBroker();
  Invoker invoker = new HotStoneGameInvoker(servant);

  // === Client side
  ClientRequestHandler crh =
          new LocalMethodClientRequestHandler(invoker);
  Requestor requestor = new StandardJSONRequestor(crh);
  game = new GameClientProxy(requestor);
}
```

Use Doubles ?

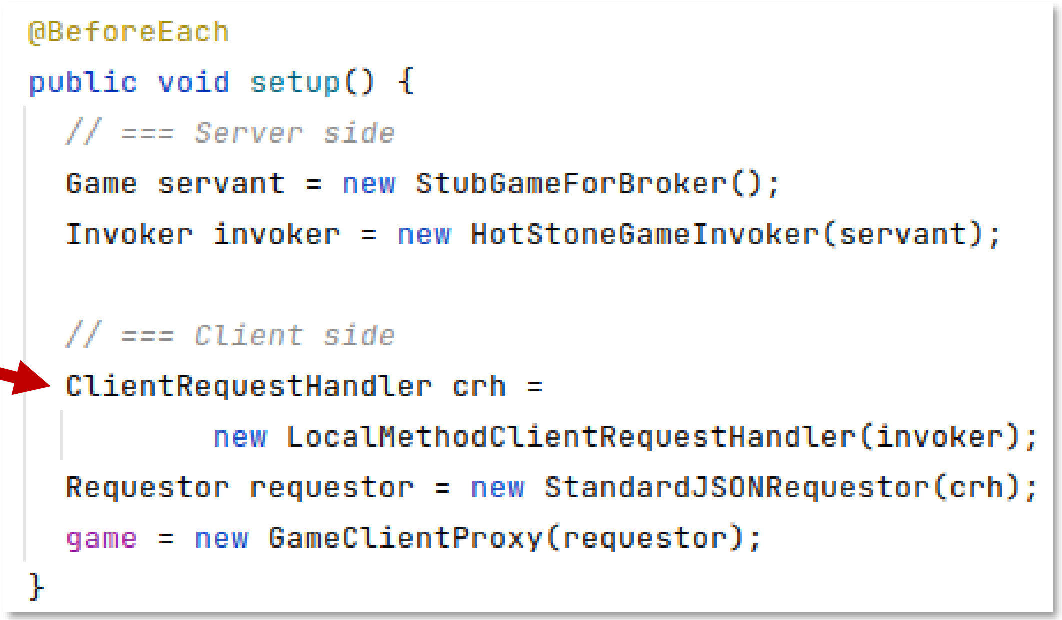
- *Use a Fake Object Game or a special stub game*
  - Create/reuse a Test fake object for the game
  - Or a simple AlphaStone ?



```
@BeforeEach
public void setup() {
  // === Server side
  Game servant = new StubGameForBroker();
  Invoker invoker = new HotStoneGameInvoker(servant);

  // === Client side
  ClientRequestHandler crh =
          new LocalMethodClientRequestHandler(invoker);
  Requestor requestor = new StandardJSONRequestor(crh);
  game = new GameClientProxy(requestor);
}
```

Do it *without IPC*

- *Take small steps and Keep Focus!*
  - TDD the ClientProxy + Invoker code first
  - Go IPC/distribution next...



```java
@BeforeEach
public void setup() {
  // === Server side
  Game servant = new StubGameForBroker();
  Invoker invoker = new HotStoneGameInvoker(servant);

  // === Client side
  ClientRequestHandler crh =
          new LocalMethodClientRequestHandler(invoker);
  Requestor requestor = new StandardJSONRequestor(crh);
  game = new GameClientProxy(requestor);
}
```

Use Doubles

- If you develop the Broker code using your full HotStone Game as *Servant* object...
- …You may run into **big bang integration** problems...
  - You have a bug, debug for hours in your ClientProxy and Invoker, only to discover the bug is in some weird strategy in the HotStone code base ☹
- Counterpoint
  - Your current AlphaStone should be pretty 'battle hardened'…

AARHUS UNIVERSITET

Use Doubles

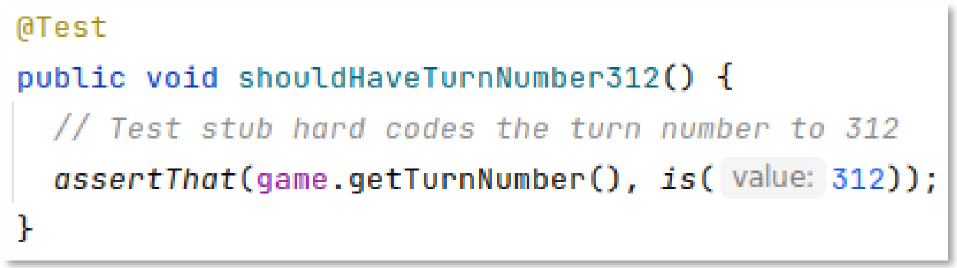
- Broker code is ‘mechanical transport of data’ and thus *no real game behavior is important for testing the broker implementation!*
  - Different from the MiniDraw exercise – there it *was* important to visually test the UI

- Thus use a test double with weird data if you can…
  - Turn number = 312
    - Broker can transport any value, and by using ‘weird’ values you are certain you talk to a test stub ☺

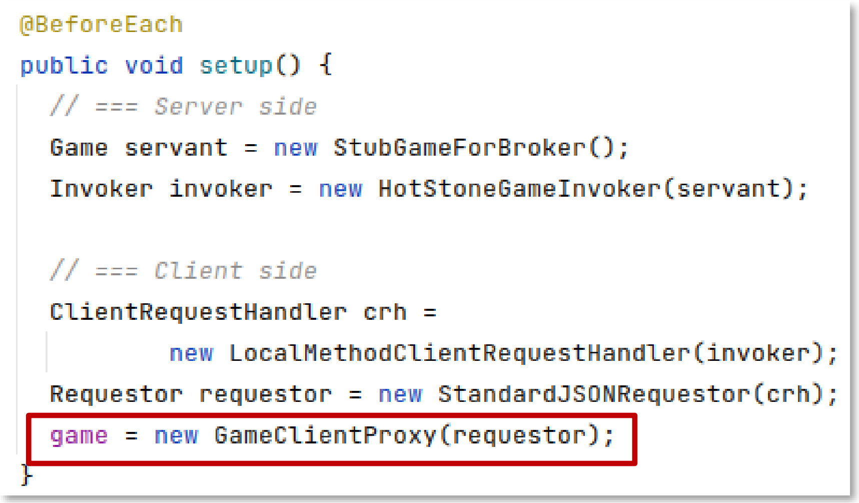
AARHUS UNIVERSITET

Go *depth first...*

- I *advice* to TDD the code *depth-first*
  - Breath-first = make all ClientProxy methods first, next all invoker
  - Depth-first = make *one* ClientProxy method and drive all code into existence, that is the Invoker, until the test case pass
- That is,
  - *Step 1: Quickly add a Test*



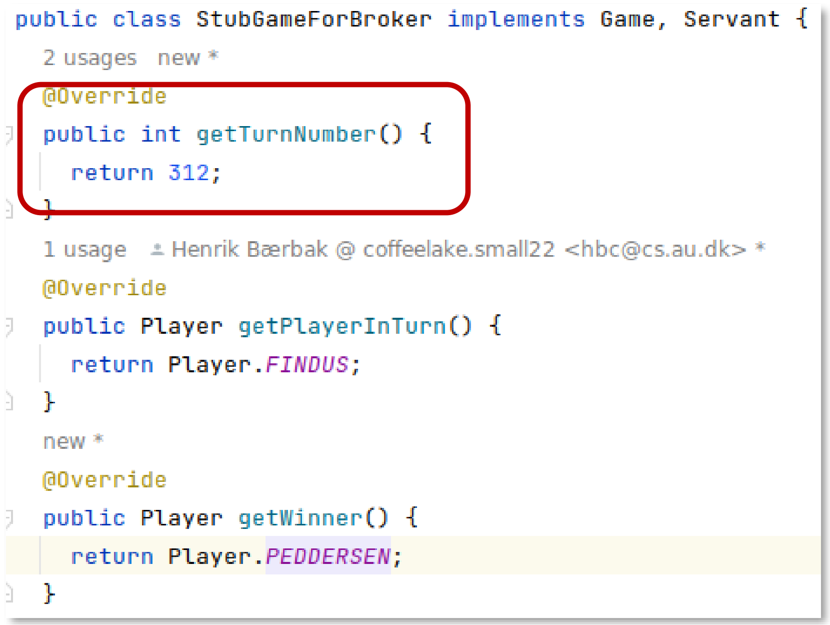
```
@Test
public void shouldHaveTurnNumber312() {
  // Test stub hard codes the turn number to 312
  assertThat(game.getTurnNumber(), is( value: 312));
}
```



```
@BeforeEach
public void setup() {
  // === Server side
  Game servant = new StubGameForBroker();
  Invoker invoker = new HotStoneGameInvoker(servant);

  // === Client side
  ClientRequestHandler crh =
          new LocalMethodClientRequestHandler(invoker);
  Requestor requestor = new StandardJSONRequestor(crh);
  game = new GameClientProxy(requestor);
}
```

The Test Stub

- *Step 1: Quickly add a Test*
  - Purpose: Develop the ClientProxy and the Invoker code
  - But the Servant must of course return turn number 312
- *Make the stub output easily recognizable output*
  - If every method returns 0, it is difficult to see if you call the right one…



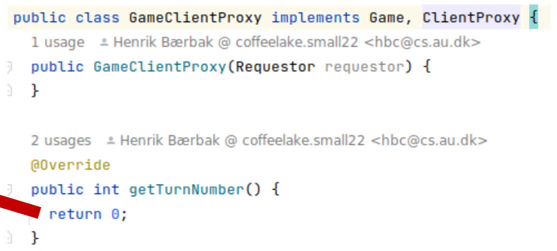
```
public class StubGameForBroker implements Game, Servant {
  2 usages  new *
  @Override
  public int getTurnNumber() {
    return 312;
  }
  1 usage  ≗ Henrik Bærbak @ coffeelake.small22 <hbc@cs.au.dk> *
  @Override
  public Player getPlayerInTurn() {
    return Player.FINDUS;
  }
  new *
  @Override
  public Player getWinner() {
    return Player.PEDDERSEN;
  }
```

Go depth first...

- *Step 2: See test fails...*

```
java.lang.AssertionError:
Expected: is <312>
     but: was <0>
```

Yeah!!!

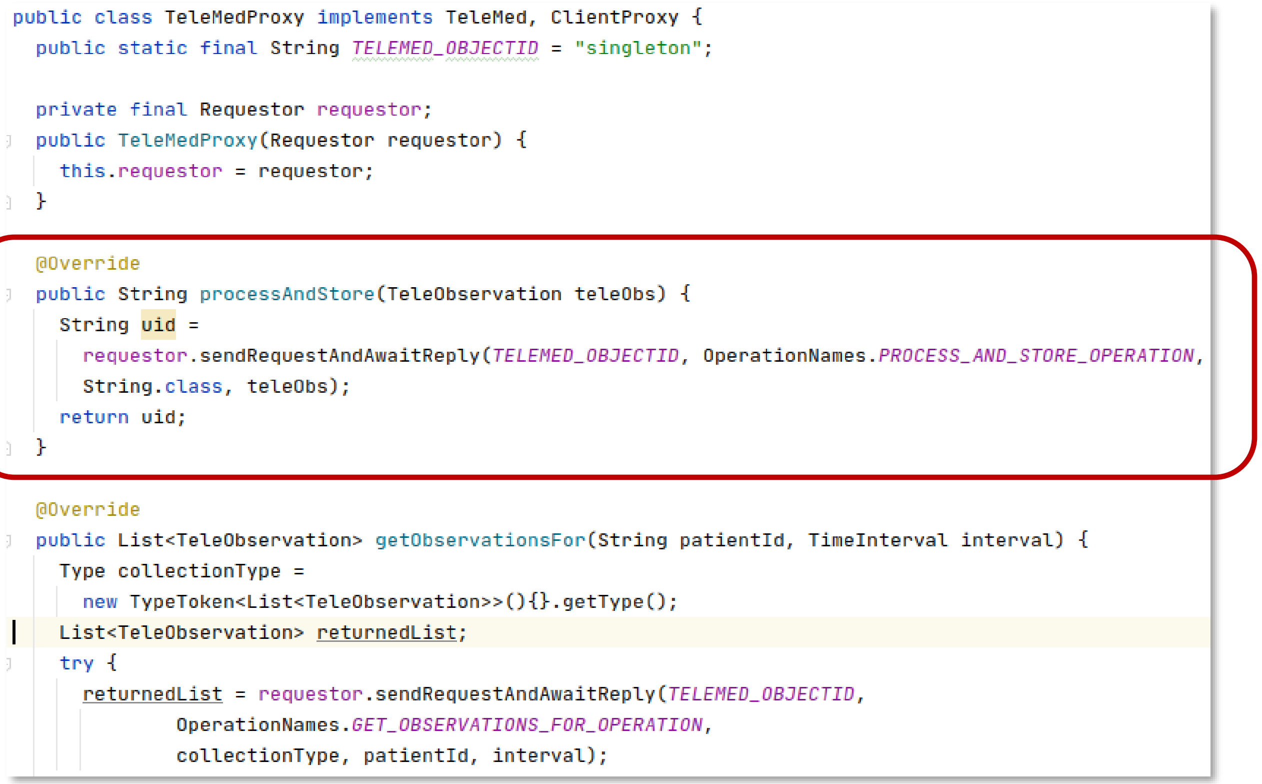


```
public class GameClientProxy implements Game, ClientProxy {
  1 usage  Henrik Bærbak @ coffeelake.small22 <hbc@cs.au.dk>
  public GameClientProxy(Requestor requestor) {
  }

  2 usages  Henrik Bærbak @ coffeelake.small22 <hbc@cs.au.dk>
  @Override
  public int getTurnNumber() {
    return 0;
  }
```

- *Step 3: Make a little change...*
  - Find *inspiration* in the TeleMed code and make the *first method of the first abstraction in the call chain work –*
  - the **GameClientProxy's getTurnNumber() method...**
    - Send the method request to the server, using the Requestor's *sendRequestAndAwaitReply()* method…

Inspiration

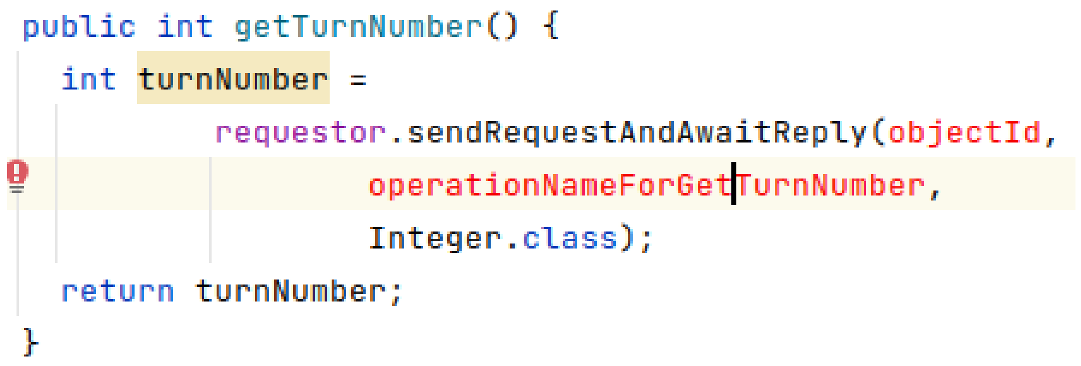
- Something inspired by the TeleMedProxy code



```
public class TeleMedProxy implements TeleMed, ClientProxy {
  public static final String TELEMED_OBJECTID = "singleton";

  private final Requestor requestor;
  public TeleMedProxy(Requestor requestor) {
    this.requestor = requestor;
  }

  @Override
  public String processAndStore(TeleObservation teleObs) {
    String uid =
      requestor.sendRequestAndAwaitReply(TELEMED_OBJECTID, OperationNames.PROCESS_AND_STORE_OPERATION,
      String.class, teleObs);
    return uid;
  }

  @Override
  public List<TeleObservation> getObservationsFor(String patientId, TimeInterval interval) {
    Type collectionType =
      new TypeToken<List<TeleObservation>>(){}.getType();
    List<TeleObservation> returnedList;
    try {
      returnedList = requestor.sendRequestAndAwaitReply(TELEMED_OBJECTID,
              OperationNames.GET_OBSERVATIONS_FOR_OPERATION,
              collectionType, patientId, interval);
```

First client proxy method

- Inspired by looking at TeleMed code I write my first attempt at a ClientProxy implementation

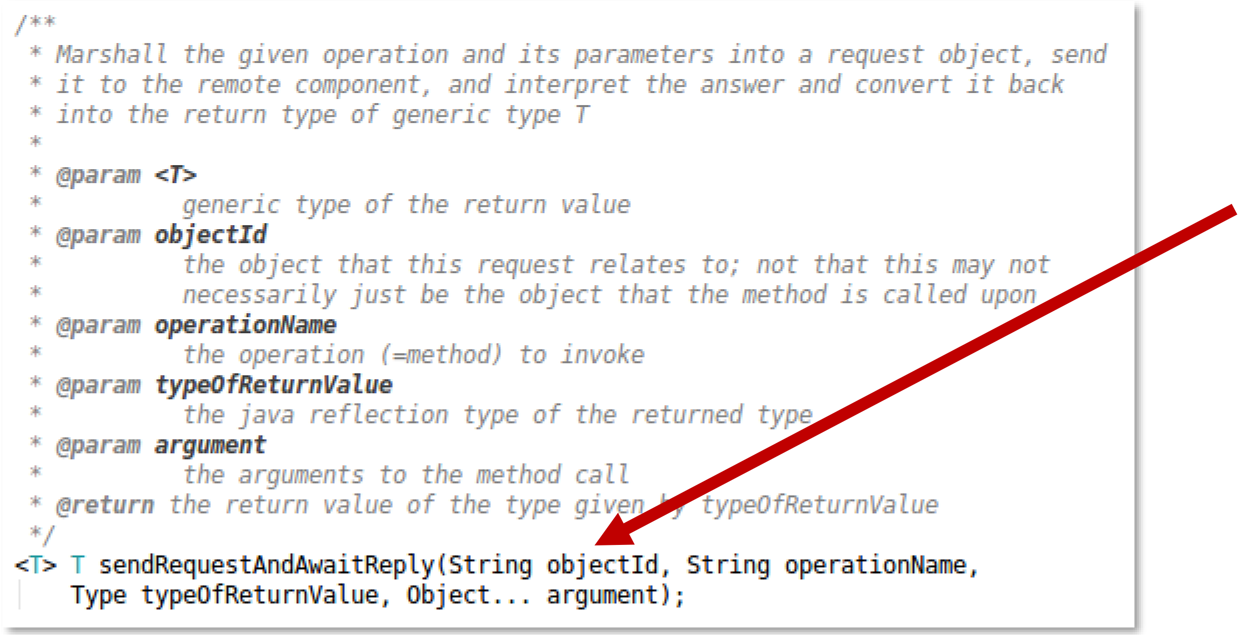


```
public int getTurnNumber() {
  int turnNumber =
          requestor.sendRequestAndAwaitReply(objectId,
                  operationNameForGetTurnNumber,
                  Integer.class);
  return turnNumber;
}
```

- OK, two issues
  - What *objectId* to use?
  - What *operation name* to use?

objectId Issue

- The objectId???
- Answer:
  - *Single* game on server
- *Exercise:*
  - *What is the objectId*

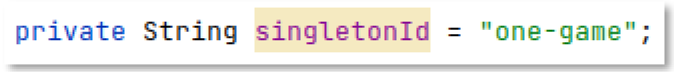


```
/**
 * Marshall the given operation and its parameters into a request object, send
 * it to the remote component, and interpret the answer and convert it back
 * into the return type of generic type T
 *
 * @param <T>
 *          generic type of the return value
 * @param objectId
 *          the object that this request relates to; not that this may not
 *          necessarily just be the object that the method is called upon
 * @param operationName
 *          the operation (=method) to invoke
 * @param typeOfReturnValue
 *          the java reflection type of the returned type
 * @param argument
 *          the arguments to the method call
 * @return the return value of the type given by typeOfReturnValue
 */
<T> T sendRequestAndAwaitReply(String objectId, String operationName,
    Type typeOfReturnValue, Object... argument);
```

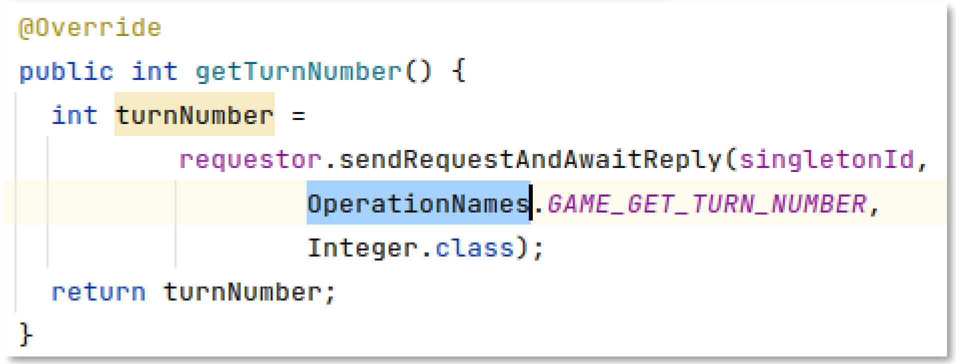
AARHUS UNIVERSITET

And Operation Name

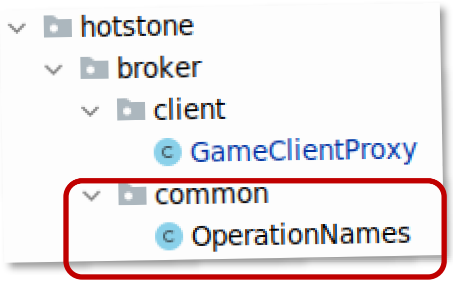
- A mangled string, uniquely identifying the method on both client and server side
  - I have provided a class 'OperationNames' with all the strings.



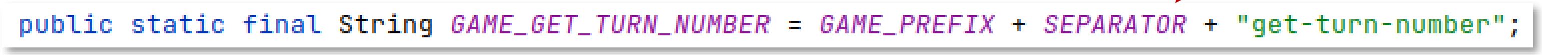
```
private String singletonId = "one-game";
```



```
@Override
public int getTurnNumber() {
  int turnNumber =
          requestor.sendRequestAndAwaitReply(singletonId,
                  OperationNames.GAME_GET_TURN_NUMBER,
                  Integer.class);
  return turnNumber;
}
```



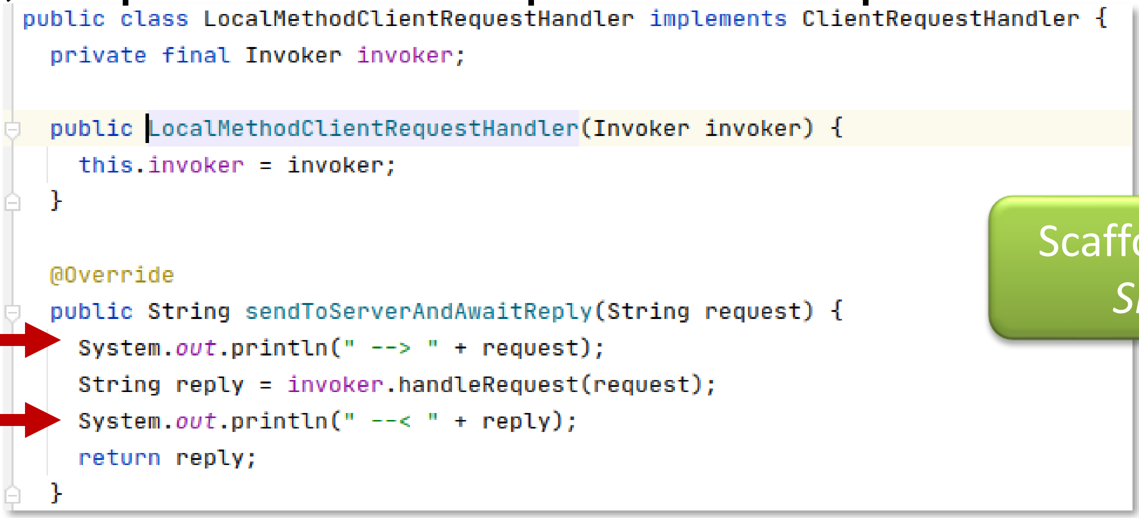
```
hotstone
  broker
    client
      GameClientProxy
    common
      OperationNames
```



```
public static final String GAME_GET_TURN_NUMBER = GAME_PREFIX + SEPARATOR + "get-turn-number";
```

Print Stuff Now, Remove Later!

- I print stuff to know 'where am I' and I can trace the call chain, inspect JSON request and replies…



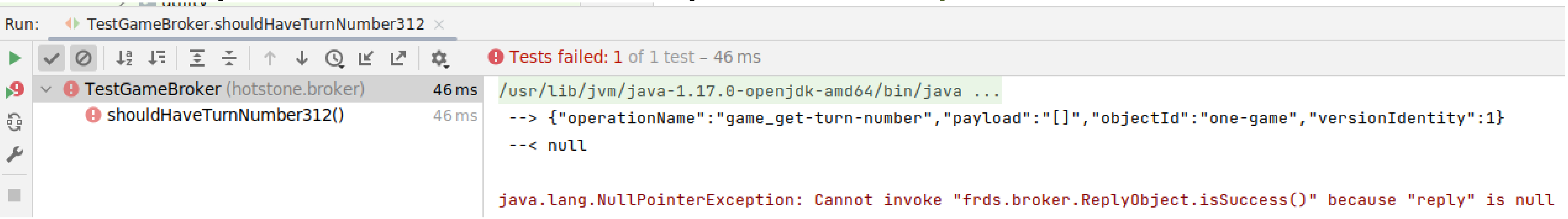
```java
public class LocalMethodClientRequestHandler implements ClientRequestHandler {
  private final Invoker invoker;

  public LocalMethodClientRequestHandler(Invoker invoker) {
    this.invoker = invoker;
  }

  @Override
  public String sendToServerAndAwaitReply(String request) {
    System.out.println(" --> " + request);
    String reply = invoker.handleRequest(request);
    System.out.println(" --< " + reply);
    return reply;
  }
```

Scaffolding code ☺.
Small steps!

- **Remove System.out again, once all test cases pass!**
  - *Or you when the output is no longer useful. Clean code!*

TDD Step 2 or 4?

- *Step 2: See it fail* or *Step 4: See it pass???*



Run: TestGameBroker.shouldHaveTurnNumber312

Tests failed: 1 of 1 test – 46 ms

TestGameBroker (hotstone.broker) 46 ms
shouldHaveTurnNumber312() 46 ms

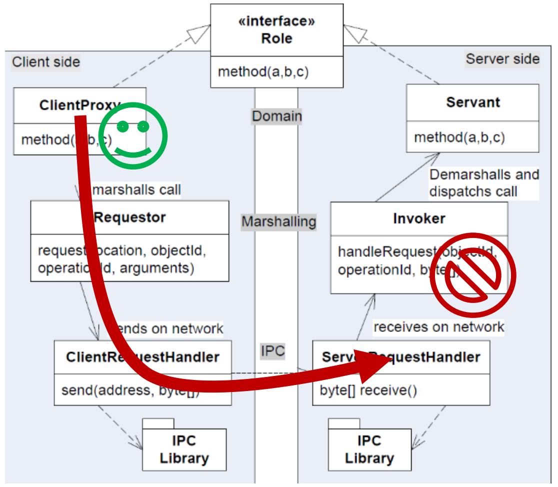
```
/usr/lib/jvm/java-1.17.0-openjdk-amd64/bin/java ...
 --> {"operationName":"game_get-turn-number","payload":"[]","objectId":"one-game","versionIdentity":1}
 --< null

java.lang.NullPointerException: Cannot invoke "frds.broker.ReplyObject.isSuccess()" because "reply" is null
```

- The printout tells me that I have
  - **Proxy at step 4**: *It works!*
    - The RequestObject looks OK

```
--> {"operationName":"game_get-turn-number","payload":"[]","objectId":"one-game","versionIdentity":1}
```

  - **Invoker at step 2**: *No code yet!*

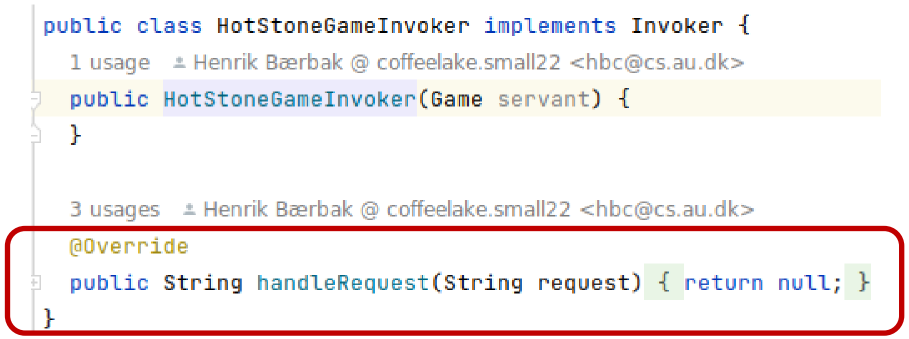
```
 --< null

java.lang.NullPointerException: Cannot invoke "frds.broker.ReplyObject.isSuccess()" because "reply" is null
```



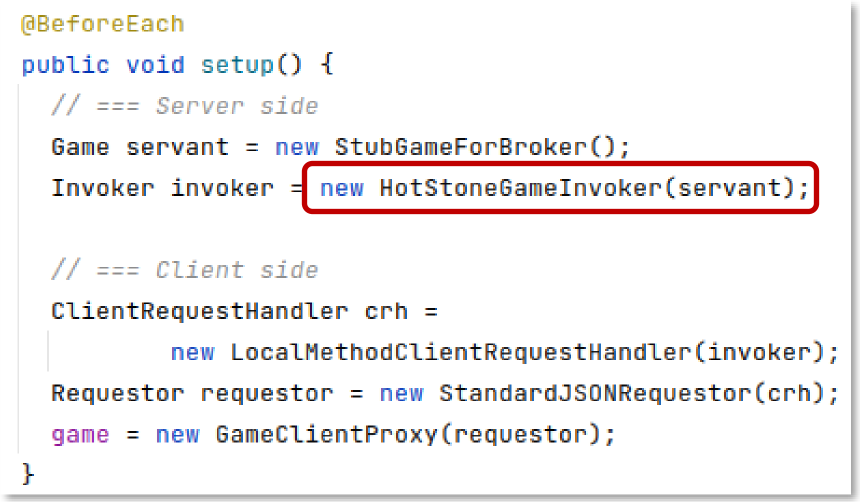
AARHUS UNIVERSITET

Step 3, part 2

- *Step 3: Make a little change...*
  - make the ***next*** *method of the first abstraction in the call chain work –*
  - the **GameInvoker's handleRequest's first switch on method name...**



```
public class HotStoneGameInvoker implements Invoker {
  1 usage  ± Henrik Bærbak @ coffeelake.small22 <hbc@cs.au.dk>
  public HotStoneGameInvoker(Game servant) {
  }

  3 usages  ± Henrik Bærbak @ coffeelake.small22 <hbc@cs.au.dk>
  @Override
  public String handleRequest(String request) { return null; }
}
```

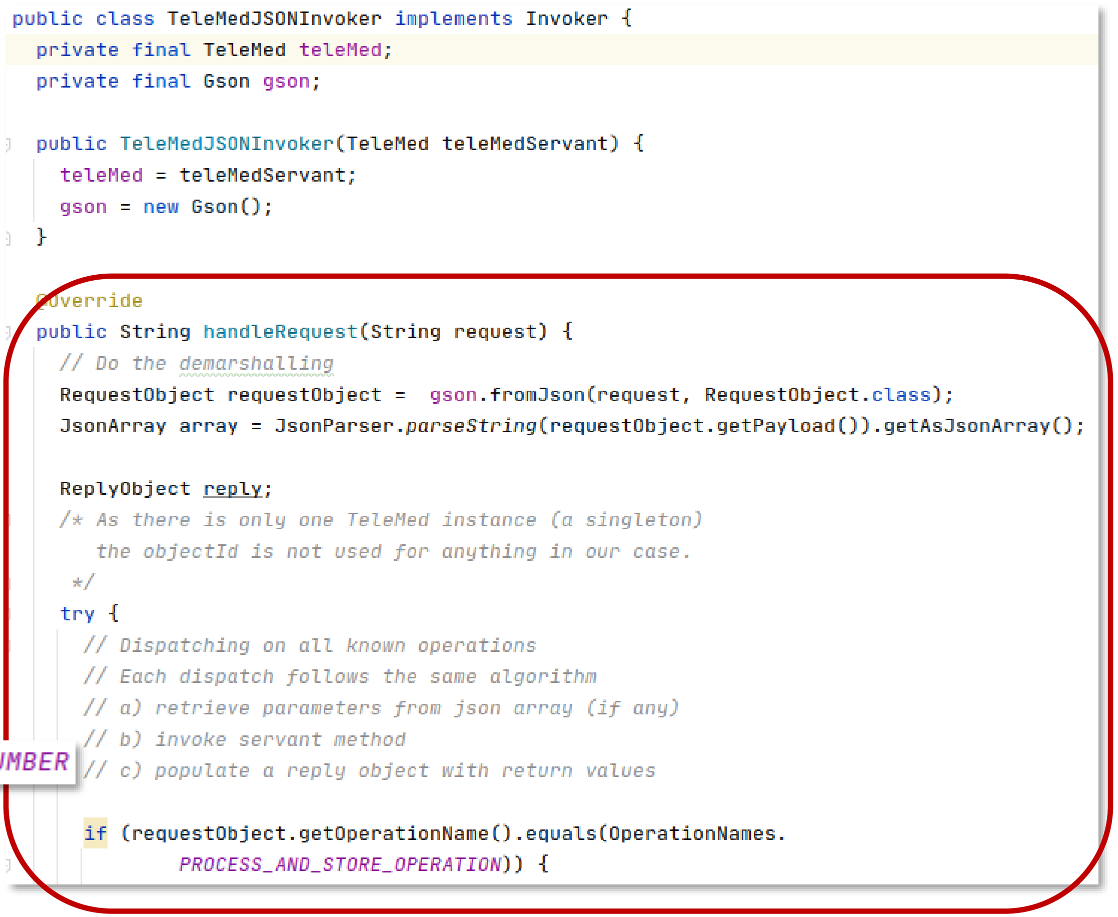


```
@BeforeEach
public void setup() {
  // === Server side
  Game servant = new StubGameForBroker();
  Invoker invoker = new HotStoneGameInvoker(servant);

  // === Client side
  ClientRequestHandler crh =
          new LocalMethodClientRequestHandler(invoker);
  Requestor requestor = new StandardJSONRequestor(crh);
  game = new GameClientProxy(requestor);
}
```

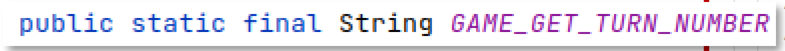
AARHUS UNIVERSITET

Inspiration

- From TeleMed’s Invoker code
- Copy a bit, change *equals(..)* to the right OpName…



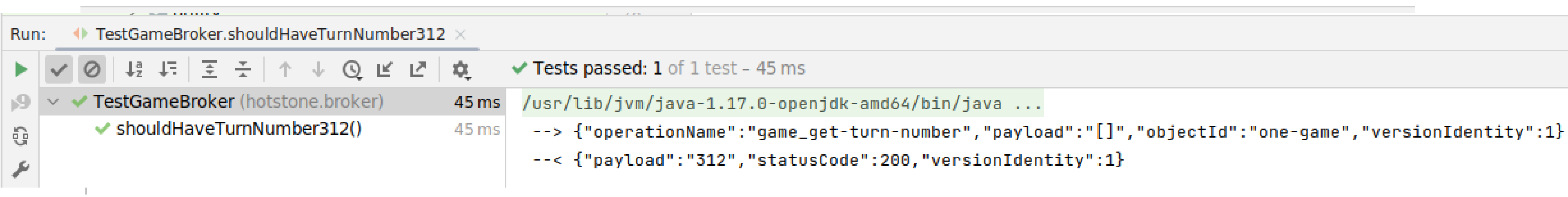
```
public class TeleMedJSONInvoker implements Invoker {
  private final TeleMed teleMed;
  private final Gson gson;

  public TeleMedJSONInvoker(TeleMed teleMedServant) {
    teleMed = teleMedServant;
    gson = new Gson();
  }

  @Override
  public String handleRequest(String request) {
    // Do the demarshalling
    RequestObject requestObject =  gson.fromJson(request, RequestObject.class);
    JsonArray array = JsonParser.parseString(requestObject.getPayload()).getAsJsonArray();

    ReplyObject reply;
    /* As there is only one TeleMed instance (a singleton)
       the objectId is not used for anything in our case.
     */
    try {
      // Dispatching on all known operations
      // Each dispatch follows the same algorithm
      // a) retrieve parameters from json array (if any)
      // b) invoke servant method
      // c) populate a reply object with return values

      if (requestObject.getOperationName().equals(OperationNames.
              PROCESS_AND_STORE_OPERATION)) {
```



```
public static final String GAME_GET_TURN_NUMBER
```

Step 4

- A bit of Invoker coding later …
  - Test pass ☺
  - Output looks OK – proper RequestObject and ReplyObject ☺



Run: TestGameBroker.shouldHaveTurnNumber312

✓ Tests passed: 1 of 1 test – 45 ms

✓ TestGameBroker (hotstone.broker) 45 ms
✓ shouldHaveTurnNumber312() 45 ms

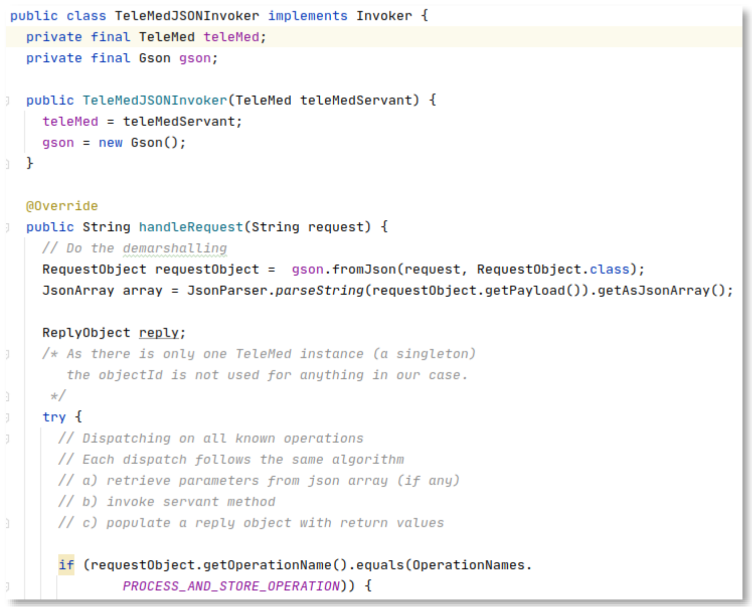
```
/usr/lib/jvm/java-1.17.0-openjdk-amd64/bin/java ...
 --> {"operationName":"game_get-turn-number","payload":"[]","objectId":"one-game","versionIdentity":1}
 --< {"payload":"312","statusCode":200,"versionIdentity":1}
```

- Conclusion
  - *One* method of Game is now correctly work through the chain
  - *All required classes are in place...*
  - Depth-first! Repeat until all pass-by-value methods in place…

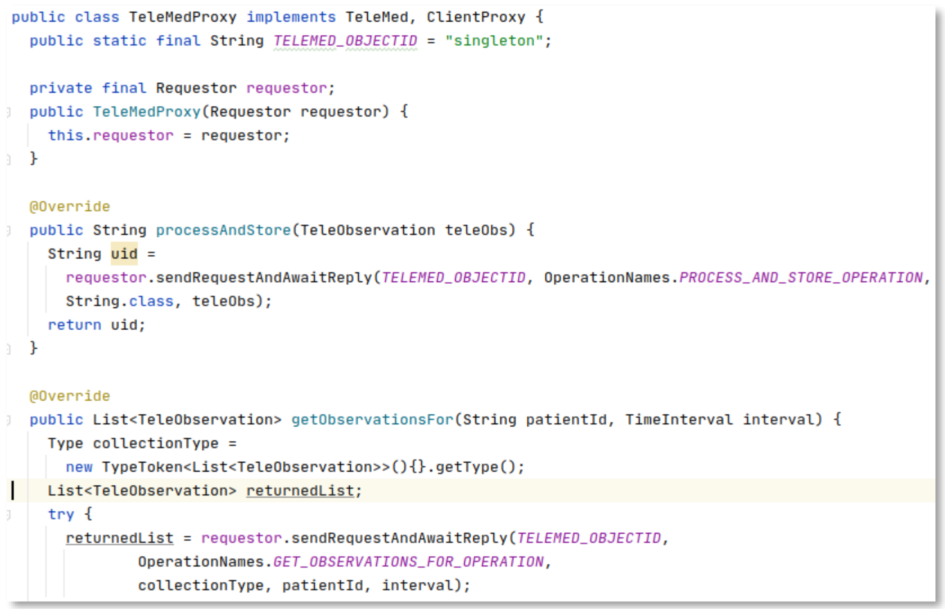
AARHUS UNIVERSITET

Steal with Pride!

- It is a learning process, this one...
  - Learn from TeleMed.
  - Have the code handy for reference ...



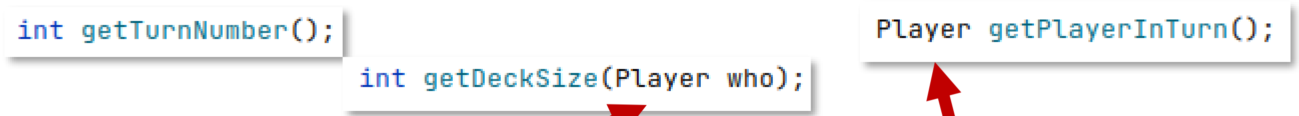
```java
public class TeleMedJSONInvoker implements Invoker {
  private final TeleMed teleMed;
  private final Gson gson;

  public TeleMedJSONInvoker(TeleMed teleMedServant) {
    teleMed = teleMedServant;
    gson = new Gson();
  }

  @Override
  public String handleRequest(String request) {
    // Do the demarshalling
    RequestObject requestObject =  gson.fromJson(request, RequestObject.class);
    JsonArray array = JsonParser.parseString(requestObject.getPayload()).getAsJsonArray();

    ReplyObject reply;
    /* As there is only one TeleMed instance (a singleton)
       the objectId is not used for anything in our case.
     */
    try {
      // Dispatching on all known operations
      // Each dispatch follows the same algorithm
      // a) retrieve parameters from json array (if any)
      // b) invoke servant method
      // c) populate a reply object with return values

      if (requestObject.getOperationName().equals(OperationNames.
              PROCESS_AND_STORE_OPERATION)) {
```



```java
public class TeleMedProxy implements TeleMed, ClientProxy {
  public static final String TELEMED_OBJECTID = "singleton";

  private final Requestor requestor;
  public TeleMedProxy(Requestor requestor) {
    this.requestor = requestor;
  }

  @Override
  public String processAndStore(TeleObservation teleObs) {
    String uid =
      requestor.sendRequestAndAwaitReply(TELEMED_OBJECTID, OperationNames.PROCESS_AND_STORE_OPERATION,
      String.class, teleObs);
    return uid;
  }

  @Override
  public List<TeleObservation> getObservationsFor(String patientId, TimeInterval interval) {
    Type collectionType =
      new TypeToken<List<TeleObservation>>(){}.getType();
    List<TeleObservation> returnedList;
    try {
      returnedList = requestor.sendRequestAndAwaitReply(TELEMED_OBJECTID,
              OperationNames.GET_OBSERVATIONS_FOR_OPERATION,
              collectionType, patientId, interval);
```

Pass by Value

- Broker 1.1: Develop all *pass by value* methods of Game



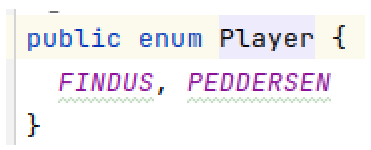
```
int getTurnNumber();
int getDeckSize(Player who);
Player getPlayerInTurn();
```

- Huh? Player???

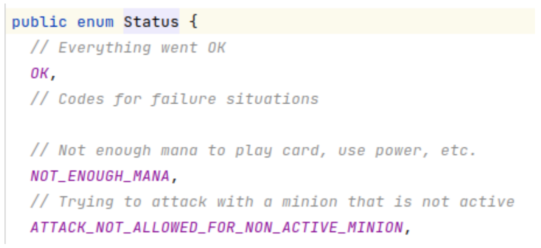
- That is an enum which is actually a class, right?

Enums *are* Values

- Enums are in Java implemented as classes *but represent values!*

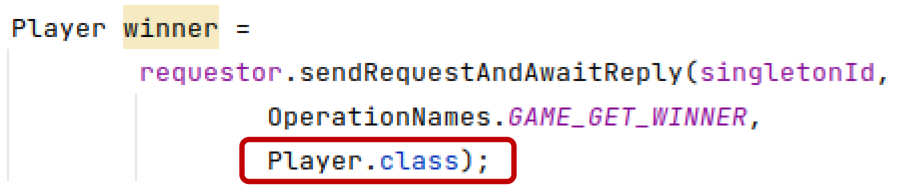


```
public enum Player {
  FINDUS, PEDDERSEN
}
```



```
public enum Status {
  // Everything went OK
  OK,
  // Codes for failure situations

  // Not enough mana to play card, use power, etc.
  NOT_ENOUGH_MANA,
  // Trying to attack with a minion that is not active
  ATTACK_NOT_ALLOWED_FOR_NON_ACTIVE_MINION,
```

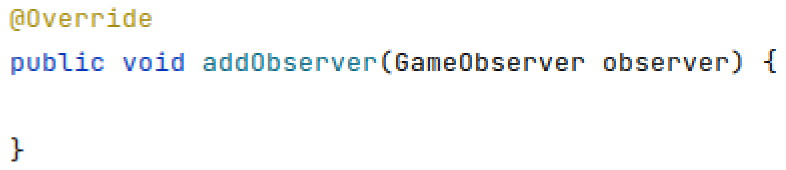
- Just pass them *as values*, that is, use Gson to marshall and demarshall
  - Gson handles it as you would expect



```
Player winner =
        requestor.sendRequestAndAwaitReply(singletonId,
                OperationNames.GAME_GET_WINNER,
                Player.class);
```

Observer

- A pure client-server architecture cannot implement Observer as outlined previously
  - Would involve the server calling the client **Not permitted**
- What then about that aspect of Game?
- Exercise:
  - In the ClientProxy, what is the observer's responsibility?
  - And what implications does that have for the client-server relation?

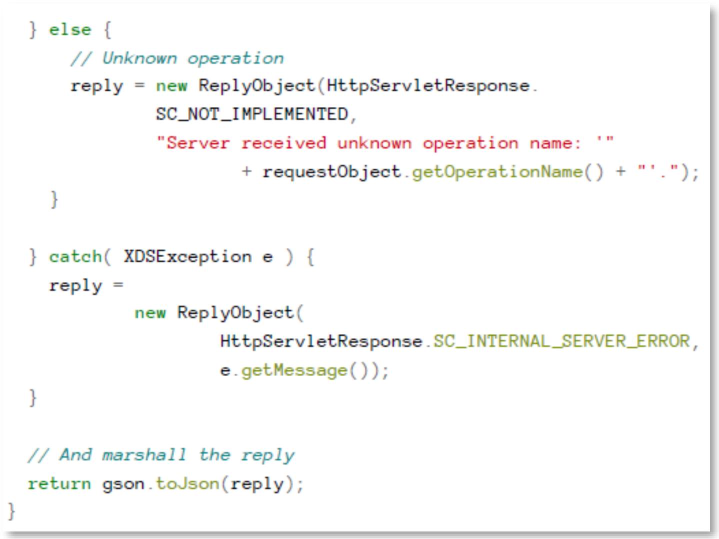


```
@Override
public void addObserver(GameObserver observer) {

}
```

AARHUS UNIVERSITET

Error Handling

- SWEA sticks to the ‘happy path’, so not required …
- However, a bit of error handling is nice (optional)
  – FRDS.Broker library *does* support transporting exceptions over the network (sort of).
  – Study the TeleMed code ☺ (page 46)
    - Handles ‘unknown method’
    - Handles ‘exception on the server’
  – If the requestor receives a Reply with errorcode >= 300, it will throw an exception in the client…



```
  } else {
    // Unknown operation
    reply = new ReplyObject(HttpServletResponse.
            SC_NOT_IMPLEMENTED,
            "Server received unknown operation name: '"
                    + requestObject.getOperationName() + "'.");
    }

  } catch( XDSException e ) {
    reply =
            new ReplyObject(
                    HttpServletResponse.SC_INTERNAL_SERVER_ERROR,
                    e.getMessage());
  }

  // And marshall the reply
  return gson.toJson(reply);
}
```

AARHUS UNIVERSITET

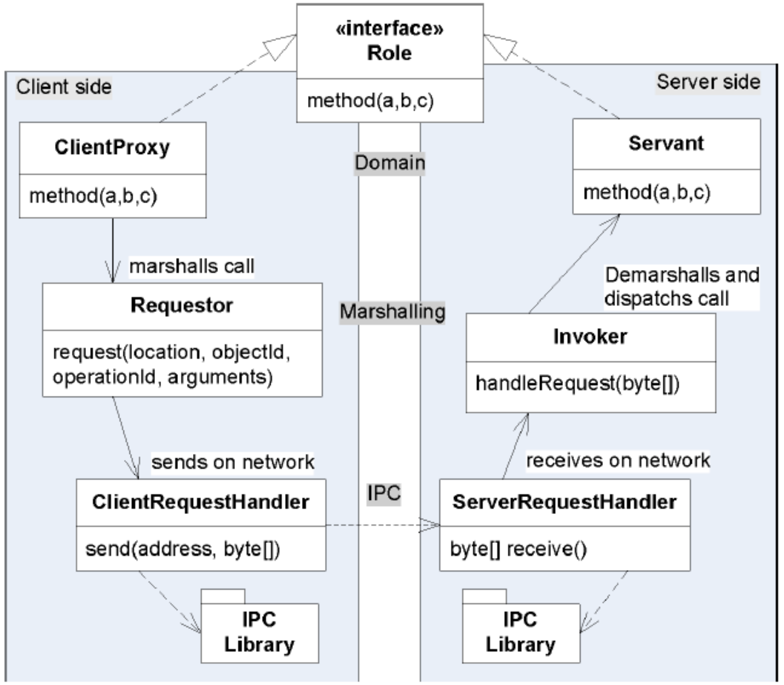
Broker 1.2

Hero and Card

AARHUS UNIVERSITET

Broker Pattern

- *There is a 'chain' for each Role!*
- Card role
  - CardClientProxy
- Hero role
  - HeroClientProxy
- What about Invoker?
  - One big invoker?
  - Invoker for each role?



AARHUS UNIVERSITET

Broker 1.2

- We will make a *stepping stone* approach in this exercise…
  - Just like in the GameLobby system in the FRDS book, we will make **one big invoker (a ‘blob’ invoker).**
  - That is: there is only one invoker, and it handles method dispatch for Game **and** Card **and** Hero
- This is “Make horrible sins and clean up later”
  - (If you make it clean from the start (= separate invokers for each type) **you will run into problems in Broker II exercises ☹**)

Hero and Card

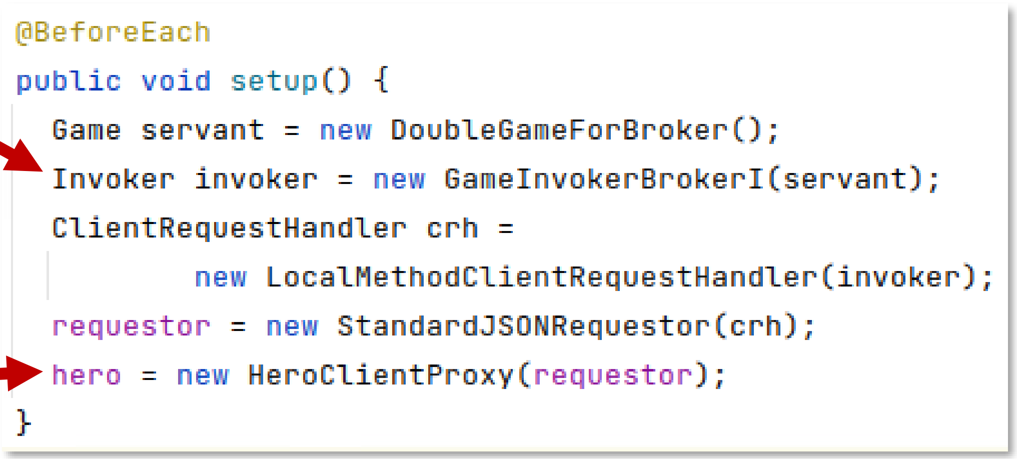
- These interfaces' methods all have value type return values
  - So, you can develop them! And you are asked to...

- Process – similar as for game… *Take small steps…*
  - Make a TestHeroBroker test class
  - Make the *broker chain* as I did in @BeforeEach for Game
  - Implement one method depth-first
    - HeroClientProxy – then update the associated dispatch in Invoker
  - Repeat until done

AARHUS UNIVERSITET

Example Hero

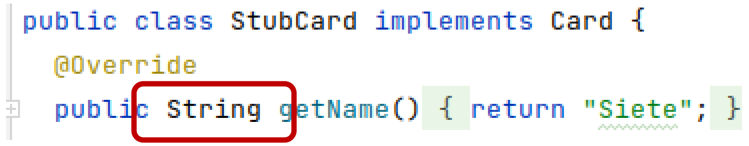
- Ala a @BeforeEach like
  - The ‘blob’ invoker
  - A Hero specific ClientProxy



```java
@BeforeEach
public void setup() {
  Game servant = new DoubleGameForBroker();
  Invoker invoker = new GameInvokerBrokerI(servant);
  ClientRequestHandler crh =
          new LocalMethodClientRequestHandler(invoker);
  requestor = new StandardJSONRequestor(crh);
  hero = new HeroClientProxy(requestor);
}
```

Strings as Value Type

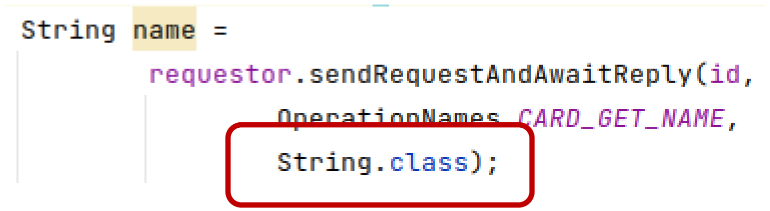
- What about String type? It is a Java class, not a primitive type ☹.



```
public class StubCard implements Card {
  @Override
  public String getName() { return "Siete"; }
```

- Treat String as a **value type**.
  - We need the characters “Siete”, not a memory reference to that string.

- Again, Gson will handle it.

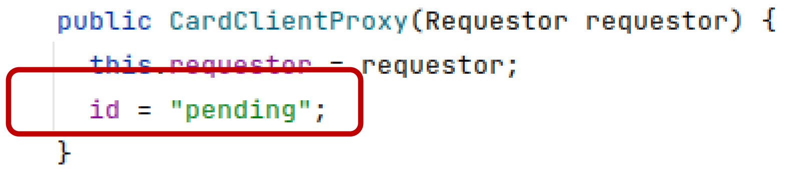


```
String name =
        requestor.sendRequestAndAwaitReply(id,
                OperationNames.CARD_GET_NAME,
                String.class);
```

Hero and Card

- The key obstacle, however, is: What is the objectId?
  - Which is the core learning goal of Broker Exercise II, next week

- For now ***Fake it till you make it... Scaffolding !***
  - Use a 'fake id' in the client proxy
  - *And…*



```
public CardClientProxy(Requestor requestor) {
  this.requestor = requestor;
  id = "pending";
}
```

Fake Id in Invoker

- Card is *not* a Singleton, there are many of them!

```java
public class GameInvokerBrokerI implements Invoker {
```

```java
@Override
public String handleRequest(String request) {
  // Do the demarshalling
  RequestObject requestObject =
          gson.fromJson(request, RequestObject.class);
  String objectId = requestObject.getObjectId();
```

Method encapsulates the lookup. Presently it is *fake-it* code, but next week can be recoded to proper impl.

```java
  // CARD methods
} else if (operationName.startsWith(OperationNames.CARD_PREFIX)) {
  // Lookup the right card to invoke the method on
  Card servant = lookupCard(objectId);
```

```java
private Card fakeItCard = new StubCard();
private Card lookupCard(String objectId) {
  return fakeItCard;
}
```

One Level of Abstraction – uncle bob

Sidebar Exercise

- Why not pass Card as a value type?
  - It is just dumb data, right?
  - Gson can marshall and demarshall it properly, right?
- Argue in favor of *pass-by-reference* and *pass-by-value*
- **In the exercise, you must implement the ClientProxy + Invoker pairs for both Card and Hero**
  - Which is ‘pass-by-reference’…

Cost of the FakeIt code

- Next week, you will actually have to modify your Invoker quite a bit – split them, replace fake-it lookup…
  - Take small steps, sometimes goes *via code that needs to be removed again once we get to the final stages of the development.*
  - Scaffolding is common in other engineering disciplines ☺

AARHUS UNIVERSITET

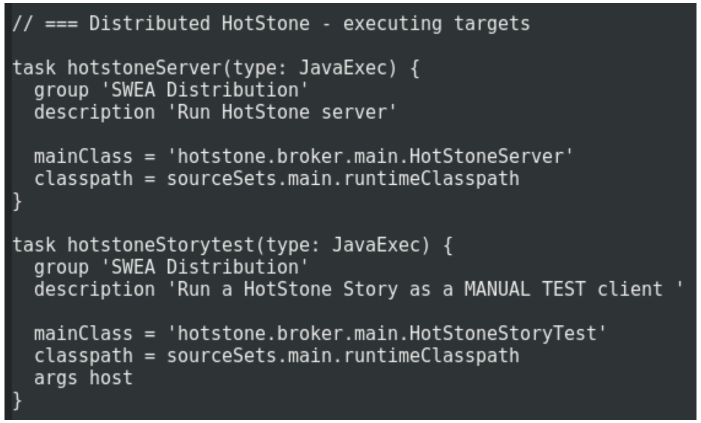
Broker 1.3

The Client and Server programs

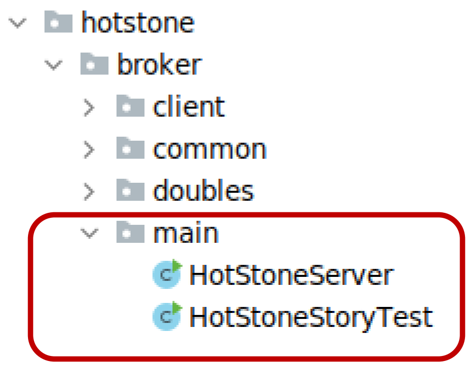
Manual integration test

The Main Methods

- TDD and Doubles will get all the core code in place.
- Still, we need *applications* to run a distributed system
  - HotStoneServer's main method
  - HotStoneClient's main method
- Provided code provides both



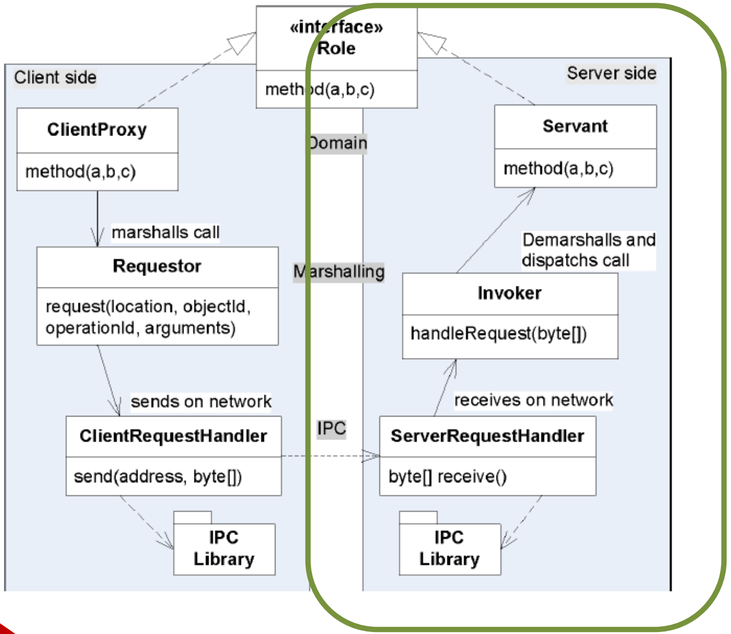
```
// === Distributed HotStone - executing targets

task hotstoneServer(type: JavaExec) {
  group 'SWEA Distribution'
  description 'Run HotStone server'

  mainClass = 'hotstone.broker.main.HotStoneServer'
  classpath = sourceSets.main.runtimeClasspath
}

task hotstoneStorytest(type: JavaExec) {
  group 'SWEA Distribution'
  description 'Run a HotStone Story as a MANUAL TEST client '

  mainClass = 'hotstone.broker.main.HotStoneStoryTest'
  classpath = sourceSets.main.runtimeClasspath
  args host
}
```



```
hotstone
  broker
    client
    common
    doubles
    main
      HotStoneServer
      HotStoneStoryTest
```

Manual Integration Test

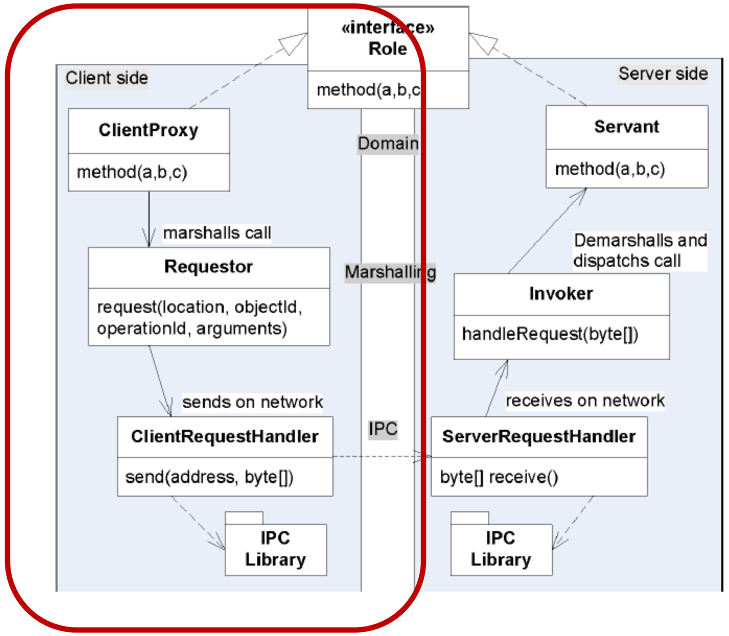
- We can build a proper server setup already this week!
- Why?
  - There is only one Servant game
    - Create 'servant'
    - Couple the invoker to it
    - Couple a UriTunnelSRH to it
      - Listen to HTTP requests from client
- And then we are done…
  - (Change the servant to your code!)
  - (The servant is not complete, but will be next week.)



```
public HotStoneServer() {
  int port = BrokerConstants.HOTSTONE_PORT;
  // Define the server side root servant
  Game servant = new StubGameForBroker();
```

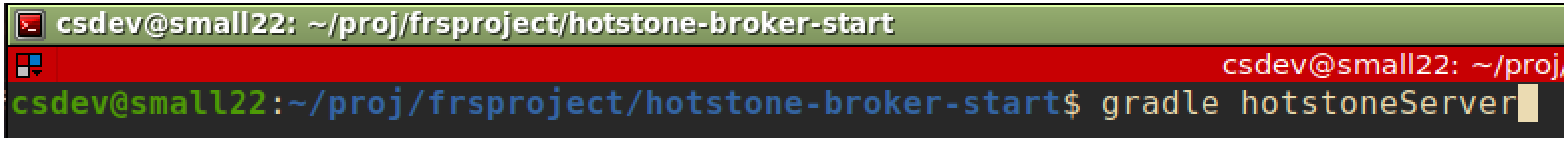
Manual Integration Test

- But we cannot build a full-blown client
- Why?
  - Game only partially done ☹
  - The Hero and Card Invokers are not coupled to the Game Invoker
  - Thus a MiniDraw GUI will fail !
- But, we *can* test the simple Game methods
- A manual integration test
  - Call a few simple Game methods over real HTTP network
    - And verify that server receives them and returns proper results…



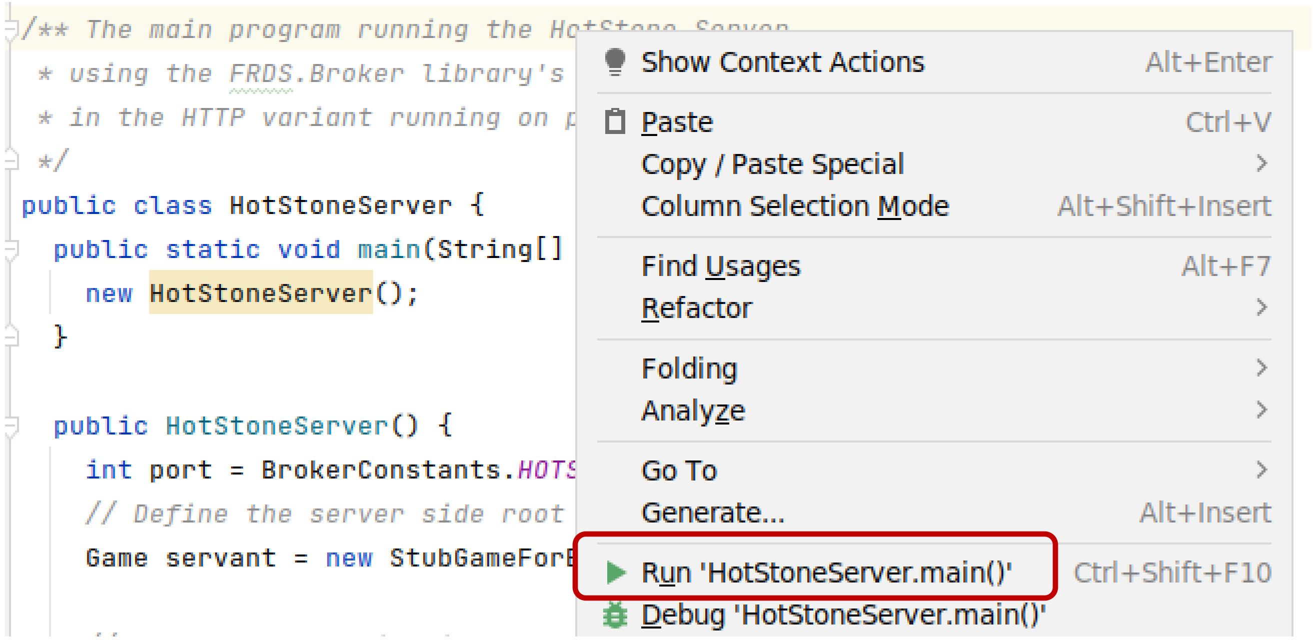
Server Side

- Start server using Gradle



```
csdev@small22: ~/proj/frsproject/hotstone-broker-start
                                                          csdev@small22: ~/proj/
csdev@small22:~/proj/frsproject/hotstone-broker-start$ gradle hotstoneServer
```

- Or in IntelliJ

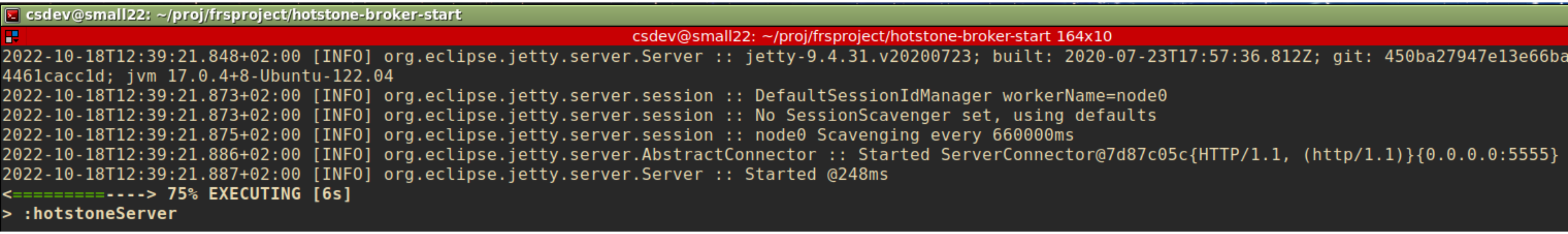


```
/** The main program running the HotStone Server
 * using the FRDS.Broker library's
 * in the HTTP variant running on p
 */
public class HotStoneServer {
  public static void main(String[]
    new HotStoneServer();
  }

  public HotStoneServer() {
    int port = BrokerConstants.HOTS
    // Define the server side root
    Game servant = new StubGameForE
```

Show Context Actions Alt+Enter
Paste Ctrl+V
Copy / Paste Special
Column Selection Mode Alt+Shift+Insert
Find Usages Alt+F7
Refactor
Folding
Analyze
Go To
Generate... Alt+Insert
Run 'HotStoneServer.main()' Ctrl+Shift+F10
Debug 'HotStoneServer.main()'

Server Running…

- The server uses a Logging framework (SLF4J) to provide server side info – a life saver in case of trouble…



csdev@small22: ~/proj/frsproject/hotstone-broker-start

csdev@small22: ~/proj/frsproject/hotstone-broker-start 164x10

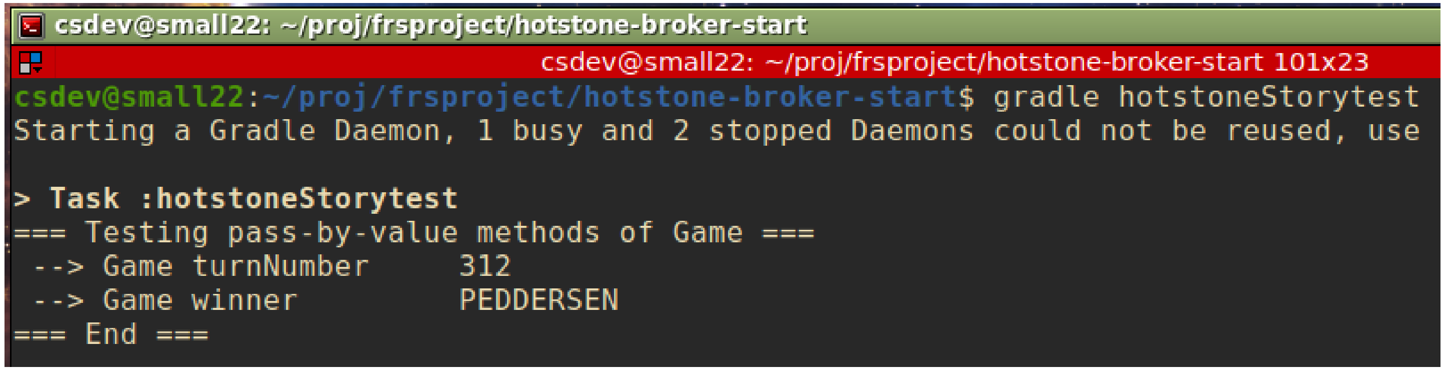
```
2022-10-18T12:39:21.848+02:00 [INFO] org.eclipse.jetty.server.Server :: jetty-9.4.31.v20200723; built: 2020-07-23T17:57:36.812Z; git: 450ba27947e13e66ba
4461caccld; jvm 17.0.4+8-Ubuntu-122.04
2022-10-18T12:39:21.873+02:00 [INFO] org.eclipse.jetty.server.session :: DefaultSessionIdManager workerName=node0
2022-10-18T12:39:21.873+02:00 [INFO] org.eclipse.jetty.server.session :: No SessionScavenger set, using defaults
2022-10-18T12:39:21.875+02:00 [INFO] org.eclipse.jetty.server.session :: node0 Scavenging every 660000ms
2022-10-18T12:39:21.886+02:00 [INFO] org.eclipse.jetty.server.AbstractConnector :: Started ServerConnector@7d87c05c{HTTP/1.1, (http/1.1)}{0.0.0.0:5555}
2022-10-18T12:39:21.887+02:00 [INFO] org.eclipse.jetty.server.Server :: Started @248ms
<=========----> 75% EXECUTING [6s]
> :hotstoneServer
```

- (Controlled by the 'log4j.properties' file in the src/main/resources folder, outside the scope of exercise)

AARHUS UNIVERSITET

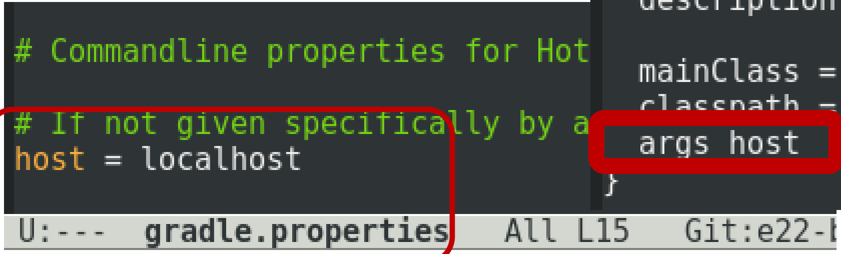
Client Side

- Start Client story test using Gradle

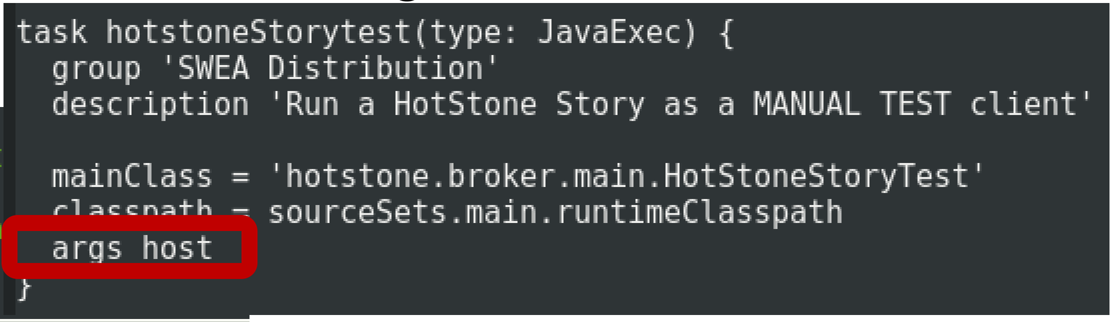


```
csdev@small22: ~/proj/frsproject/hotstone-broker-start
csdev@small22: ~/proj/frsproject/hotstone-broker-start 101x23
csdev@small22:~/proj/frsproject/hotstone-broker-start$ gradle hotstoneStorytest
Starting a Gradle Daemon, 1 busy and 2 stopped Daemons could not be reused, use

> Task :hotstoneStorytest
=== Testing pass-by-value methods of Game ===
 --> Game turnNumber      312
 --> Game winner          PEDDERSEN
=== End ===
```

- This 'main()' method needs 1 argument: which server to contact?



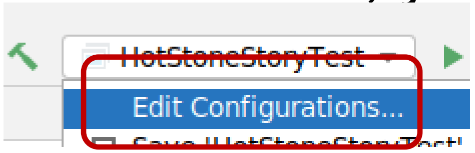
```
# Commandline properties for Hot

# If not given specifically by a
host = localhost

U:---  gradle.properties    All L15    Git:e22-
```

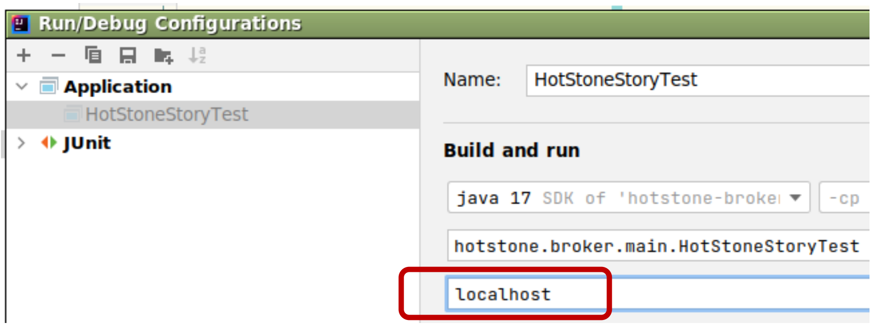


```
task hotstoneStorytest(type: JavaExec) {
  group 'SWEA Distribution'
  description 'Run a HotStone Story as a MANUAL TEST client'

  mainClass = 'hotstone.broker.main.HotStoneStoryTest'
  classpath = sourceSets.main.runtimeClasspath
  args host
}
```

Client Side

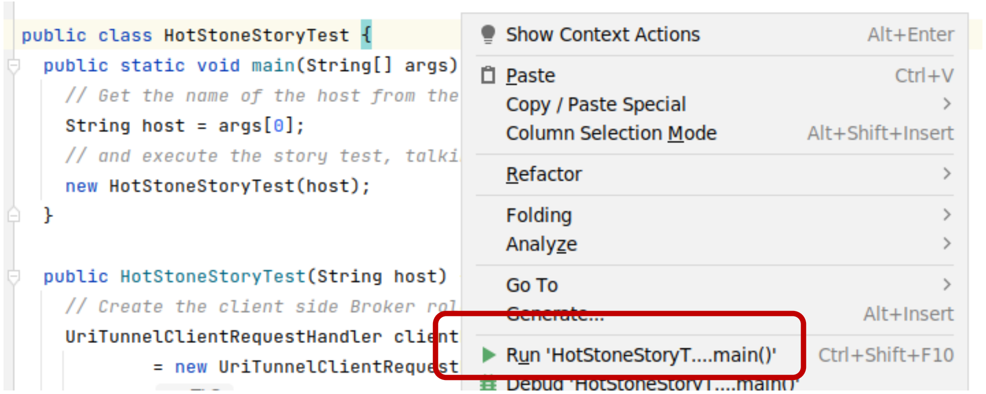
- From IntelliJ, you also need to give that parameter





Name: HotStoneStoryTest

```
hotstone.broker.main.HotStoneStoryTest
localhost
```

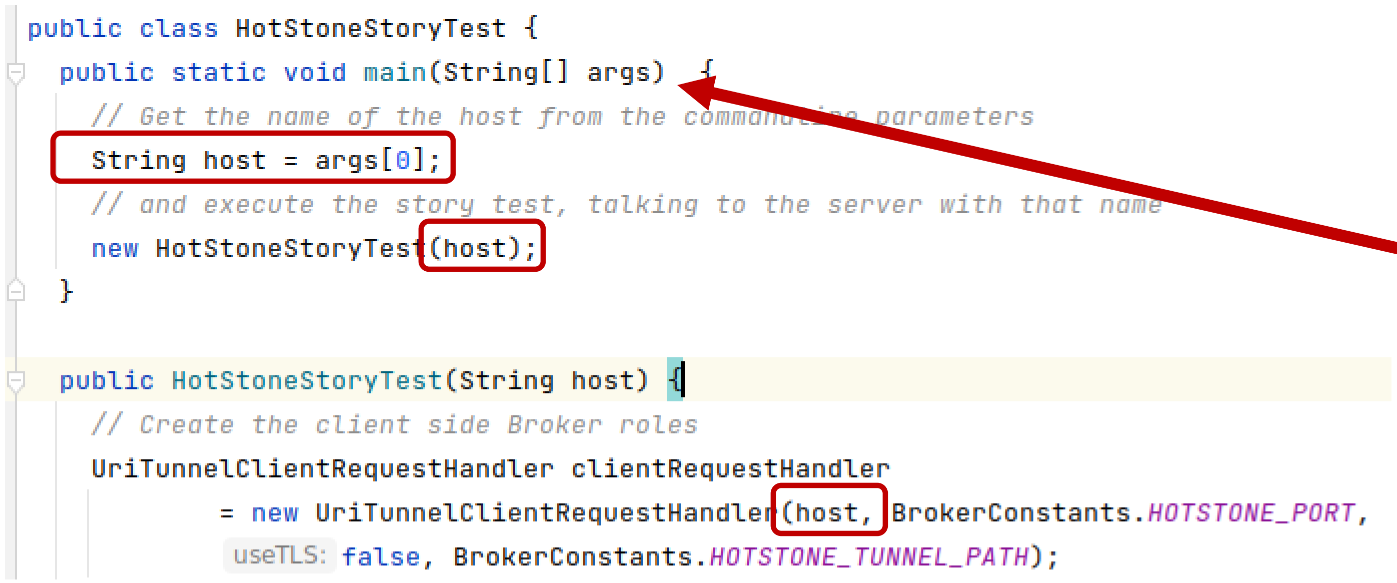


```
public class HotStoneStoryTest {
  public static void main(String[] args)
    // Get the name of the host from the
    String host = args[0];
    // and execute the story test, talki
    new HotStoneStoryTest(host);
  }

  public HotStoneStoryTest(String host)
    // Create the client side Broker rol
    UriTunnelClientRequestHandler client
          = new UriTunnelClientRequest
```

Show Context Actions Alt+Enter
Paste Ctrl+V
Copy / Paste Special
Column Selection Mode Alt+Shift+Insert
Refactor
Folding
Analyze
Go To
Generate... Alt+Insert
Run 'HotStoneStoryT....main()' Ctrl+Shift+F10
Debug 'HotStoneStoryT....main()'

Client Code

- Passing host parameter to the main method…



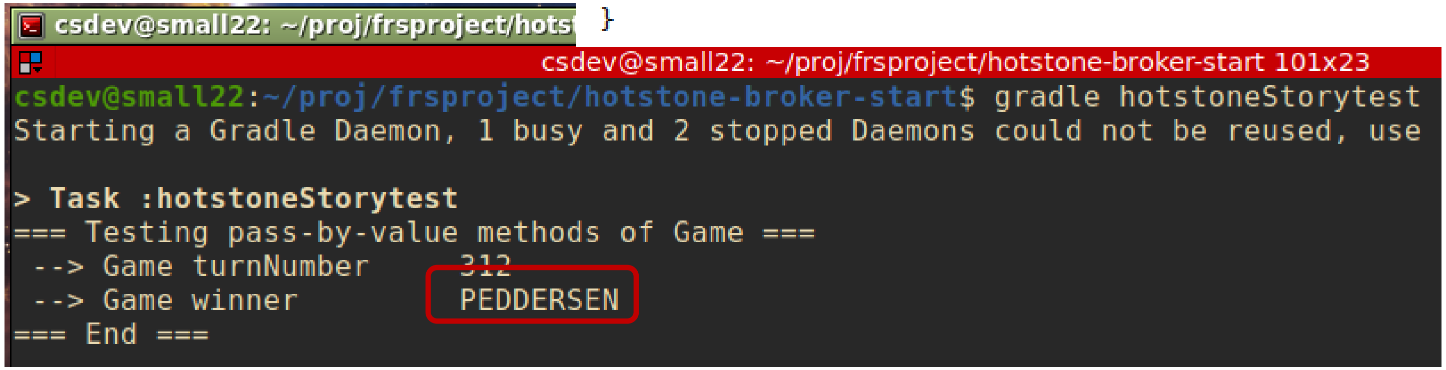
```
public class HotStoneStoryTest {
  public static void main(String[] args)  {
    // Get the name of the host from the commandline parameters
    String host = args[0];
    // and execute the story test, talking to the server with that name
    new HotStoneStoryTest(host);
  }

  public HotStoneStoryTest(String host) {
    // Create the client side Broker roles
    UriTunnelClientRequestHandler clientRequestHandler
            = new UriTunnelClientRequestHandler(host, BrokerConstants.HOTSTONE_PORT,
            useTLS: false, BrokerConstants.HOTSTONE_TUNNEL_PATH);
```

AARHUS UNIVERSITET

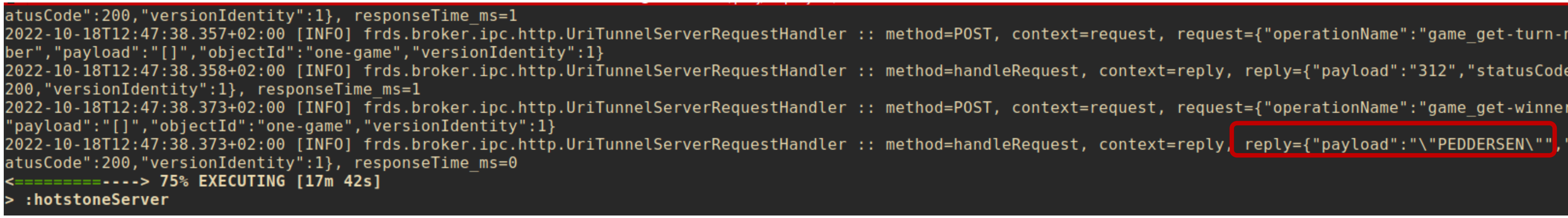
Manual Test method

- Let the client just exercise a scenario/remote calls

```
private void testSimpleMethods(Game game) {
  System.out.println("=== Testing pass-by-value methods of Game ===");
  System.out.println(" --> Game turnNumber     " + game.getTurnNumber());
  System.out.println(" --> Game winner         " + game.getWinner());
  // TODO - add calls to the rest of the implemented methods
  System.out.println("=== End ===");
}
```



```
csdev@small22: ~/proj/frsproject/hotst
csdev@small22: ~/proj/frsproject/hotstone-broker-start 101x23
csdev@small22:~/proj/frsproject/hotstone-broker-start$ gradle hotstoneStorytest
Starting a Gradle Daemon, 1 busy and 2 stopped Daemons could not be reused, use

> Task :hotstoneStorytest
=== Testing pass-by-value methods of Game ===
 --> Game turnNumber     312
 --> Game winner         PEDDERSEN
=== End ===
```



```
atusCode":200,"versionIdentity":1}, responseTime_ms=1
2022-10-18T12:47:38.357+02:00 [INFO] frds.broker.ipc.http.UriTunnelServerRequestHandler :: method=POST, context=request, request={"operationName":"game_get-turn-n
ber","payload":"[]","objectId":"one-game","versionIdentity":1}
2022-10-18T12:47:38.358+02:00 [INFO] frds.broker.ipc.http.UriTunnelServerRequestHandler :: method=handleRequest, context=reply, reply={"payload":"312","statusCode
200,"versionIdentity":1}, responseTime_ms=1
2022-10-18T12:47:38.373+02:00 [INFO] frds.broker.ipc.http.UriTunnelServerRequestHandler :: method=POST, context=request, request={"operationName":"game_get-winner
"payload":"[]","objectId":"one-game","versionIdentity":1}
2022-10-18T12:47:38.373+02:00 [INFO] frds.broker.ipc.http.UriTunnelServerRequestHandler :: method=handleRequest, context=reply, reply={"payload":"\"PEDDERSEN\"",
atusCode":200,"versionIdentity":1}, responseTime_ms=0
<=========----> 75% EXECUTING [17m 42s]
> :hotstoneServer
```

AARHUS UNIVERSITET

Summary

- Development Patterns for Iteration 9+10

- *Setup the Broker Chain first*
- *Use Test Doubles for Game and IPC*
- *Print now and remove later*
  - *Print to System.out to trace flow, remove when shit works*
- *Develop each method depth-first*
  - *Make proxy method for method x, see proper output from print, next iteration make invoker code, done…*

AARHUS UNIVERSITET

Conclusion...

Happy Coding!