AARHUS UNIVERSITET

Software Engineering and Architecture

Broker II

Object References

TeleMed Limitations

- TeleMed was a three-wheeled bicycle ☺

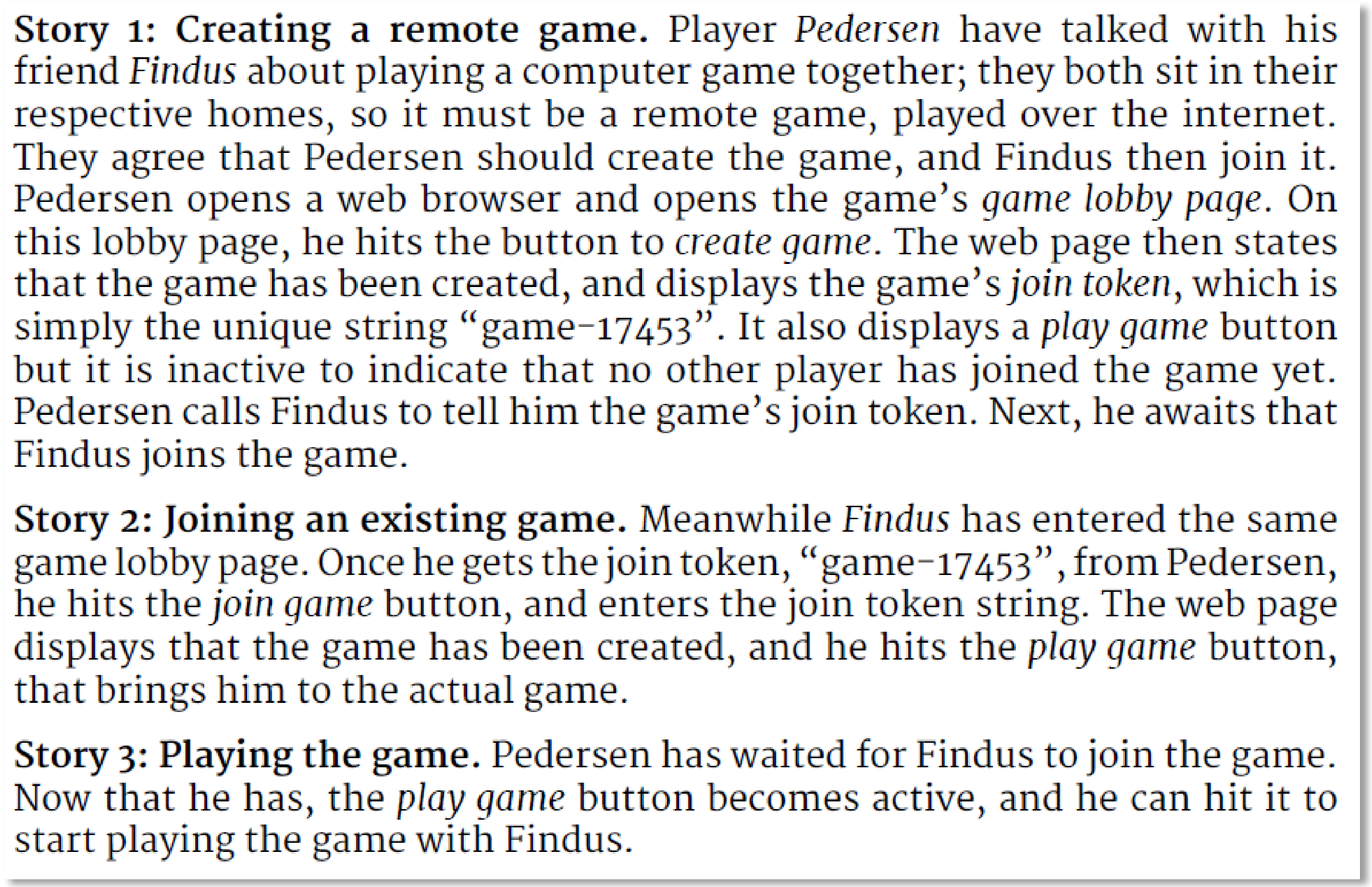
– Only *one* object	:TeleMedServant
– Only *one* class	TeleMedServant
– Object id was given by domain	Patients unique ID

  - That is, CPR is provided to TeleMed – allow getting Inger’s data…

- We need to ride an ‘ordinary bicycle’, like HotStone

– Multiple objects	Multiple Cards, Heroes
– Multiple classes	Card, Hero, ...
– ‘new StandardCard()’	On the server!

New Case: GameLobby



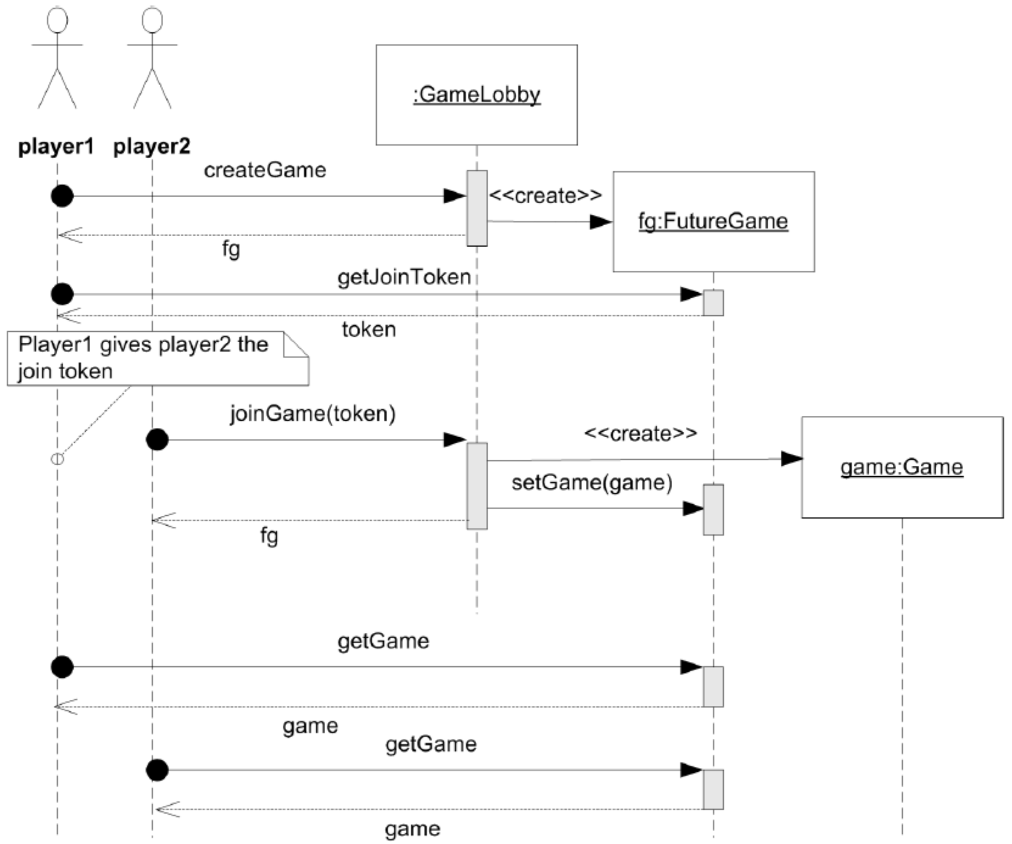
Story 1: Creating a remote game. Player *Pedersen* have talked with his friend *Findus* about playing a computer game together; they both sit in their respective homes, so it must be a remote game, played over the internet. They agree that Pedersen should create the game, and Findus then join it. Pedersen opens a web browser and opens the game's *game lobby page*. On this lobby page, he hits the button to *create game*. The web page then states that the game has been created, and displays the game's *join token*, which is simply the unique string "game-17453". It also displays a *play game* button but it is inactive to indicate that no other player has joined the game yet. Pedersen calls Findus to tell him the game's join token. Next, he awaits that Findus joins the game.

Story 2: Joining an existing game. Meanwhile *Findus* has entered the same game lobby page. Once he gets the join token, "game-17453", from Pedersen, he hits the *join game* button, and enters the join token string. The web page displays that the game has been created, and he hits the *play game* button, that brings him to the actual game.

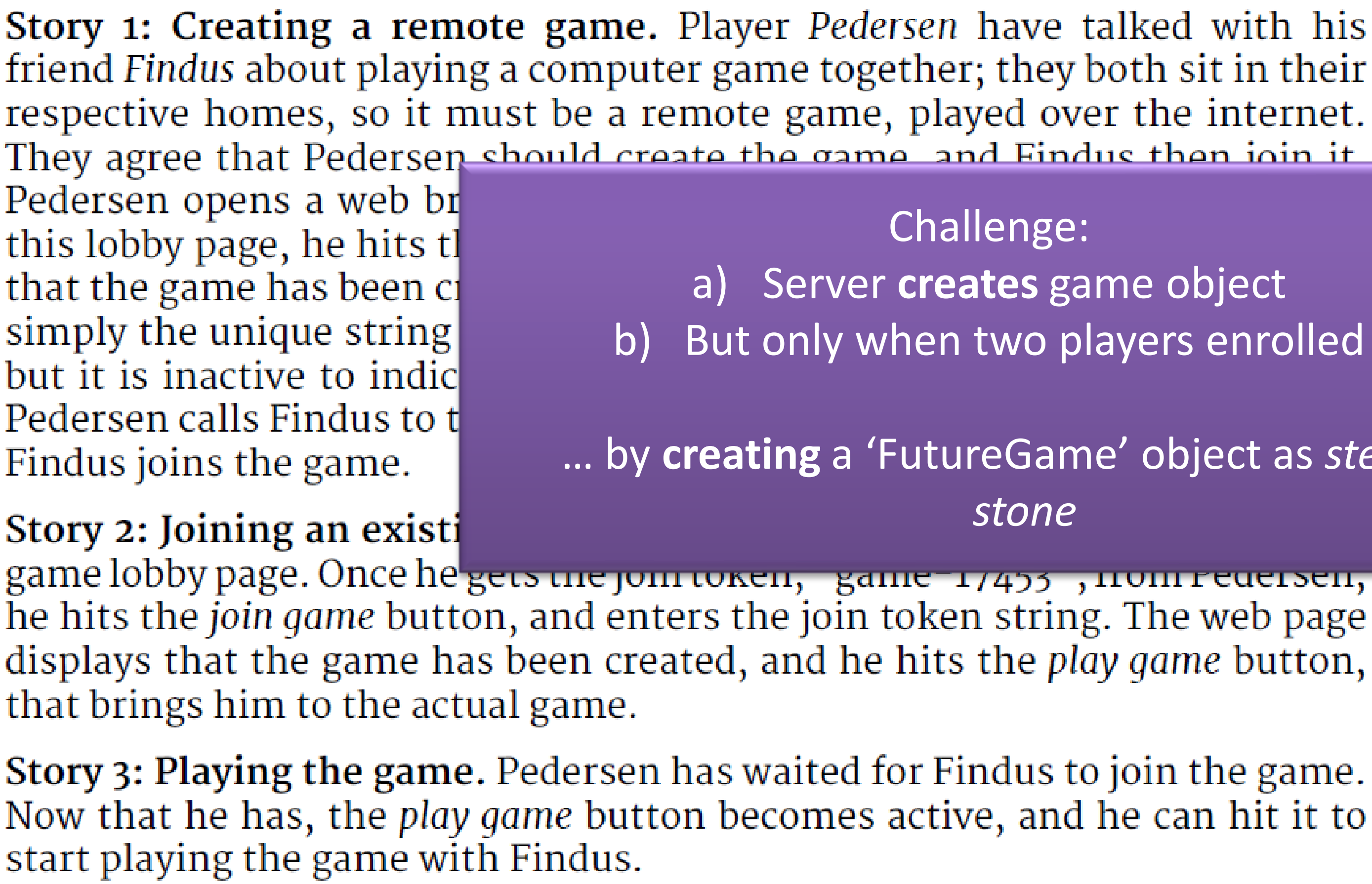
Story 3: Playing the game. Pedersen has waited for Findus to join the game. Now that he has, the *play game* button becomes active, and he can hit it to start playing the game with Findus.

Dynamics

- Side note
  - Perhaps a bit 'convoluted' design but…
  - Lots of 'remote references' to pass to the clients…



New Case: GameLobby



Story 1: Creating a remote game. Player *Pedersen* have talked with his friend *Findus* about playing a computer game together; they both sit in their respective homes, so it must be a remote game, played over the internet. They agree that Pedersen should create the game, and Findus then join it. Pedersen opens a web br
this lobby page, he hits t
that the game has been c
simply the unique string
but it is inactive to indic
Pedersen calls Findus to t
Findus joins the game.

Story 2: Joining an existi
game lobby page. Once he gets the join token, "game-17453", from Pedersen, he hits the *join game* button, and enters the join token string. The web page displays that the game has been created, and he hits the *play game* button, that brings him to the actual game.

Story 3: Playing the game. Pedersen has waited for Findus to join the game. Now that he has, the *play game* button becomes active, and he can hit it to start playing the game with Findus.

Challenge:

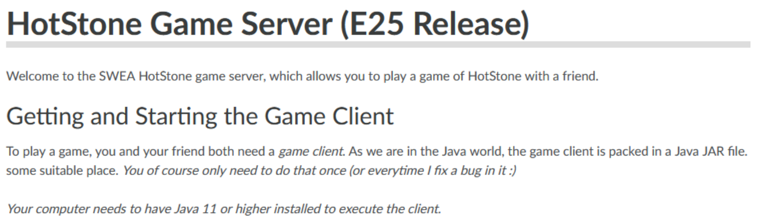
a) Server **creates** game object
b) But only when two players enrolled

… by **creating** a 'FutureGame' object as *stepping stone*

AARHUS UNIVERSITET

Side Note

- GameLobby'ish design is used in my game server on hotstone.littleworld.dk



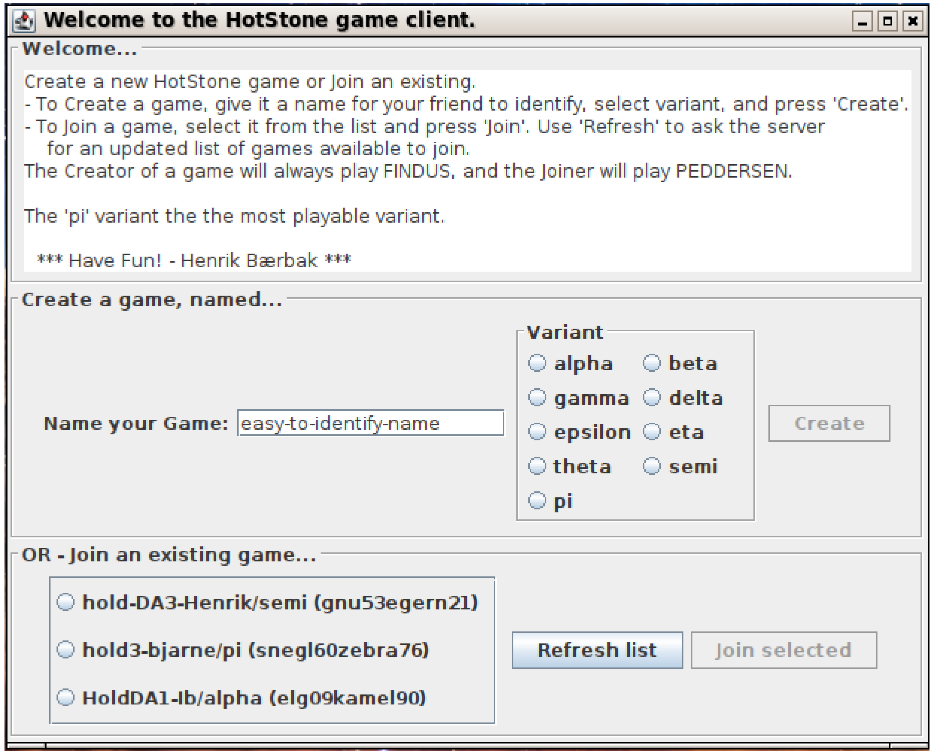
HotStone Game Server (E25 Release)

Welcome to the SWEA HotStone game server, which allows you to play a game of HotStone with a friend.

Getting and Starting the Game Client

To play a game, you and your friend both need a *game client*. As we are in the Java world, the game client is packed in a Java JAR file. some suitable place. *You of course only need to do that once (or everytime I fix a bug in it :)*

Your computer needs to have Java 11 or higher installed to execute the client.



Welcome to the HotStone game client.

Welcome...

Create a new HotStone game or Join an existing.
- To Create a game, give it a name for your friend to identify, select variant, and press 'Create'.
- To Join a game, select it from the list and press 'Join'. Use 'Refresh' to ask the server
  for an updated list of games available to join.
The Creator of a game will always play FINDUS, and the Joiner will play PEDDERSEN.

The 'pi' variant the the most playable variant.

*** Have Fun! - Henrik Bærbak ***

Create a game, named...

Name your Game: easy-to-identify-name

Variant: alpha, beta, gamma, delta, epsilon, eta, theta, semi, pi

OR - Join an existing game...

- hold-DA3-Henrik/semi (gnu53egern21)
- hold3-bjarne/pi (snegl60zebra76)
- HoldDA1-Ib/alpha (elg09kamel90)

Roles

GameLobby

- Singleton object, representing the entry point for creating and joining games.

FutureGame

- A Future, allowing the state of the game (available or not) to be queried, and once both players have joined, return the game.
- Provides an accessor method `getJoinToken()` to retrieve the join token that the second user must provide.

Game

- The actual game domain role.

AARHUS UNIVERSITET

Roles

GameLobby

- Singleton object, representing the entry point for creating and joining games.

Distributed systems always have some kind of 'singleton object' – the hardcoded 'first reference/entrypoint' of the system

FutureGame

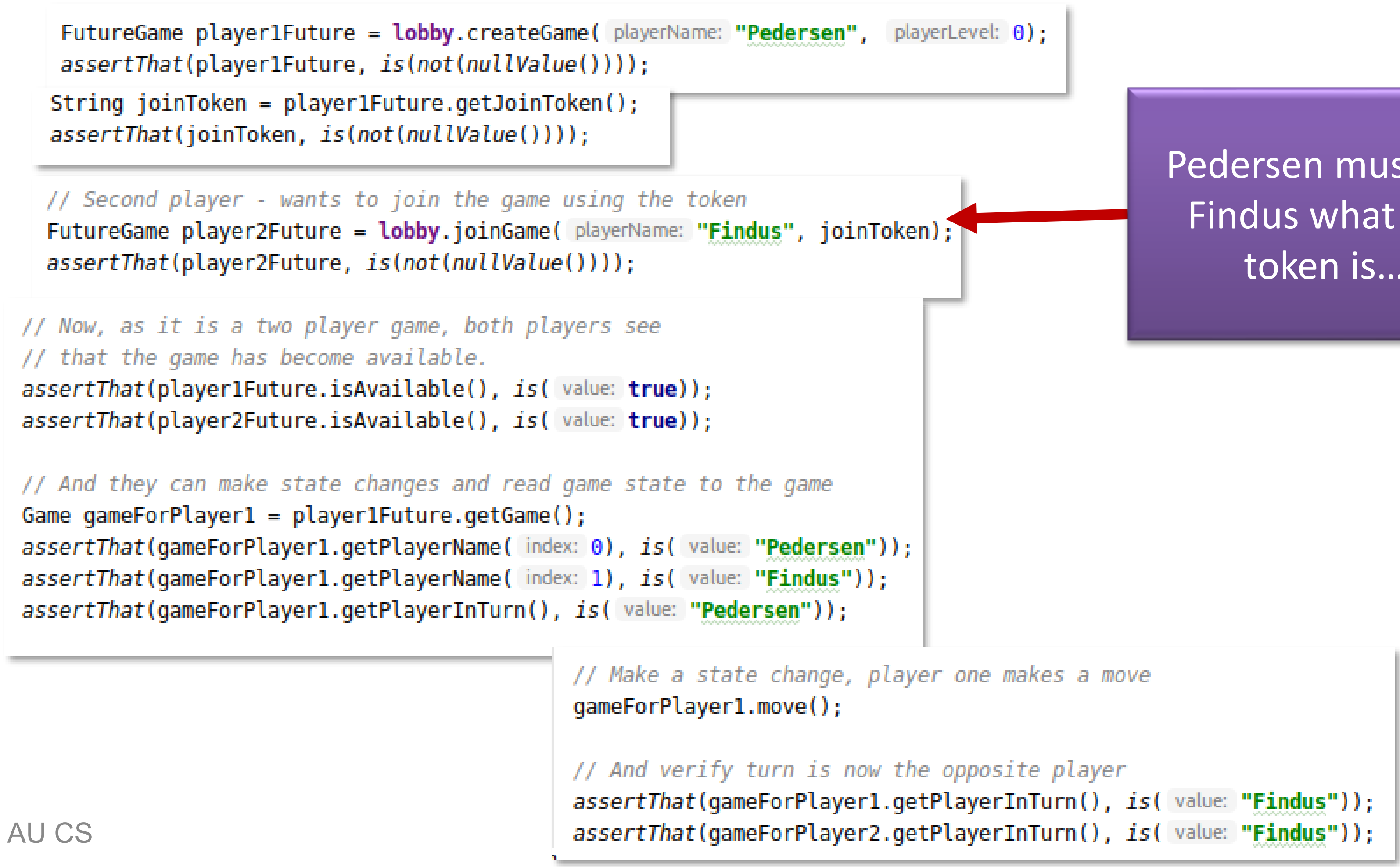
- A Future, allowing the state of the game (available or not) to be queried, and once both players have joined, return the game.
- Provides an accessor method `getJoinToken()` to retrieve the join token that the second user must provide.

Game

- The actual game domain role.

AARHUS UNIVERSITET

Test Code View (Client side)



```
FutureGame player1Future = lobby.createGame( playerName: "Pedersen",  playerLevel: 0);
assertThat(player1Future, is(not(nullValue())));
String joinToken = player1Future.getJoinToken();
assertThat(joinToken, is(not(nullValue())));

// Second player - wants to join the game using the token
FutureGame player2Future = lobby.joinGame( playerName: "Findus", joinToken);
assertThat(player2Future, is(not(nullValue())));

// Now, as it is a two player game, both players see
// that the game has become available.
assertThat(player1Future.isAvailable(), is( value: true));
assertThat(player2Future.isAvailable(), is( value: true));

// And they can make state changes and read game state to the game
Game gameForPlayer1 = player1Future.getGame();
assertThat(gameForPlayer1.getPlayerName( index: 0), is( value: "Pedersen"));
assertThat(gameForPlayer1.getPlayerName( index: 1), is( value: "Findus"));
assertThat(gameForPlayer1.getPlayerInTurn(), is( value: "Pedersen"));

// Make a state change, player one makes a move
gameForPlayer1.move();

// And verify turn is now the opposite player
assertThat(gameForPlayer1.getPlayerInTurn(), is( value: "Findus"));
assertThat(gameForPlayer2.getPlayerInTurn(), is( value: "Findus"));
```

Pedersen must tell Findus what the token is…

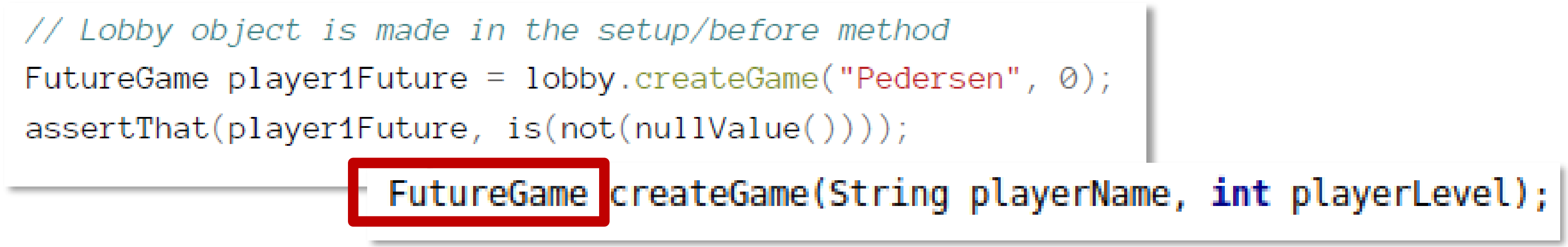
AARHUS UNIVERSITET

The Positive Viewpoint

- *Let us approach the problem from the viewpoint of status:* what ***does*** work on the ship? [*Apollo 13 Movie* ☺]
  - What can our Broker *already* handle?
- Three Java interfaces in the GameLobby system
  - Yep: We can make ClientProxies and Servants for the three roles
    - GameLobby, FutureGame, Game
  - Yep: Marshalling method names, arguments
  - Yep: IPC, nothing new here either
  - Invoker: Quite a few methods in the invoker, but Yep, we can do that – *just put more if's into the thing...*

But...

- The culprit is the method call on client:



```
// Lobby object is made in the setup/before method
FutureGame player1Future = lobby.createGame("Pedersen", 0);
assertThat(player1Future, is(not(nullValue())));
```

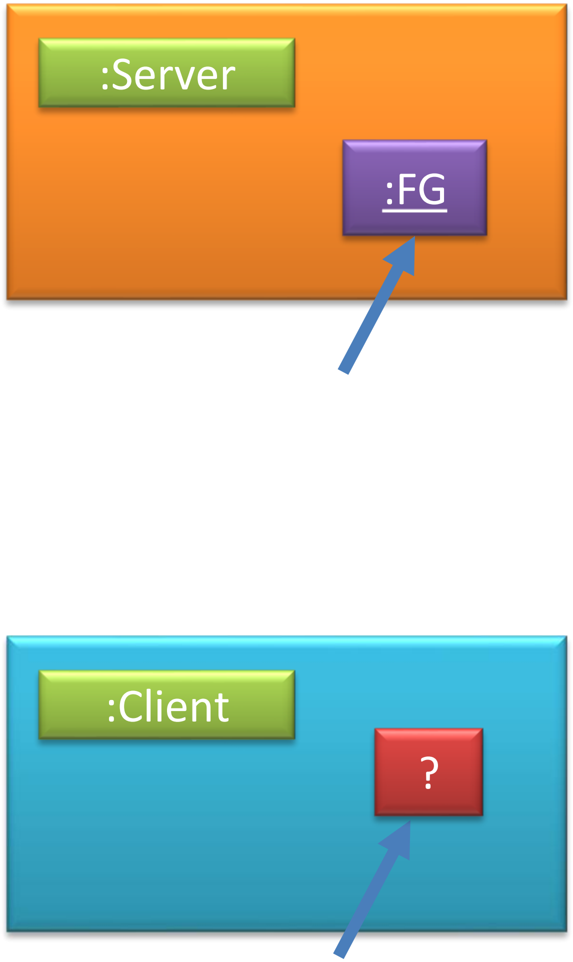
```
FutureGame createGame(String playerName, int playerLevel);
```

- On the server, *createGame()*, will create and return a FutureGame instance, but we cannot ***pass by reference***.
  - We can only **pass by value**
    - Strings, integers, DTO/POJO, Record type (json stuff)
- On the client, we only have ClientProxies, right?
  - If we pass-by-value ‘player1Future’ then no interactions across the two clients, just get a *copy-of-values* on their local machine…

AARHUS UNIVERSITET

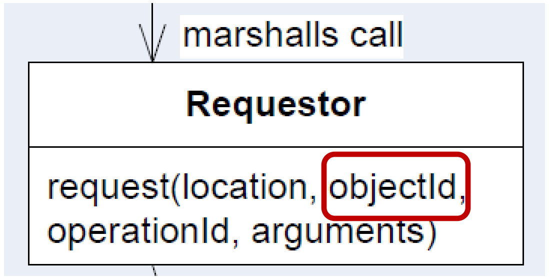
Revisit

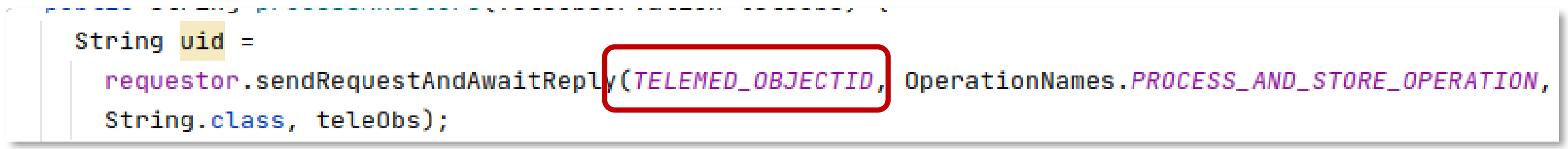
- This is the deep and hard problem
- *We create an object on the server (object reference to something on the java heap (=a memory address))...*
- But a memory address/object reference is **only valid on the server**
  - A pass-by-value of it does not make sense on the heap of the client!



The Insight

- The key insight is
  - How does ClientProxies address their associate Servant objects? *Through the 'objectId' parameter of the requestor call…*





```
  String uid =
    requestor.sendRequestAndAwaitReply(TELEMED_OBJECTID, OperationNames.PROCESS_AND_STORE_OPERATION,
    String.class, teleObs);
```

AARHUS UNIVERSITET

The Insight

- The key insight is
  - How does ClientProxies address their associate Servant objects? *Through the 'objectId' parameter of the requestor call…*

- So –
  - the server must *return a unique objectId of the object, not a reference to it*
  - the client proxy must create a *proxy*, and associate it with that particular *objectId*

If you do not see this right away...
It is because it is one of those 'aha' things ☺

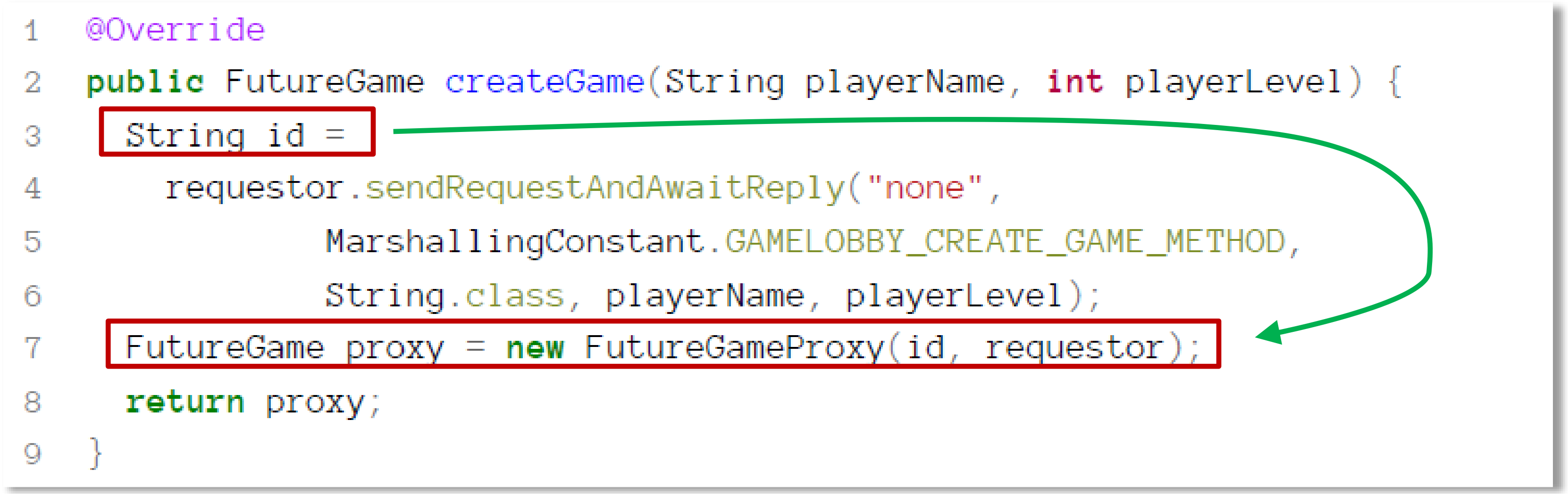
AARHUS UNIVERSITET

Client Side

How does this look?

Client: GameLobbyProxy

- So, the ClientProxy code becomes

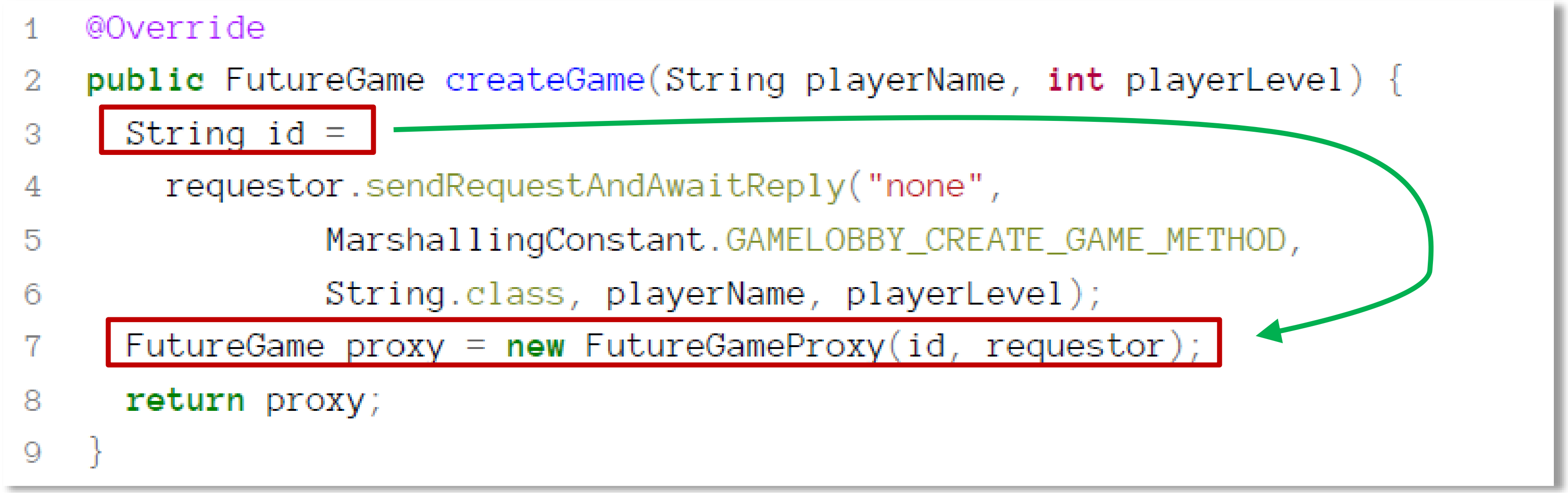


```java
@Override
public FutureGame createGame(String playerName, int playerLevel) {
  String id =
      requestor.sendRequestAndAwaitReply("none",
              MarshallingConstant.GAMELOBBY_CREATE_GAME_METHOD,
              String.class, playerName, playerLevel);
  FutureGame proxy = new FutureGameProxy(id, requestor);
  return proxy;
}
```

That is, **objectId** replaces *object reference*

The Concept

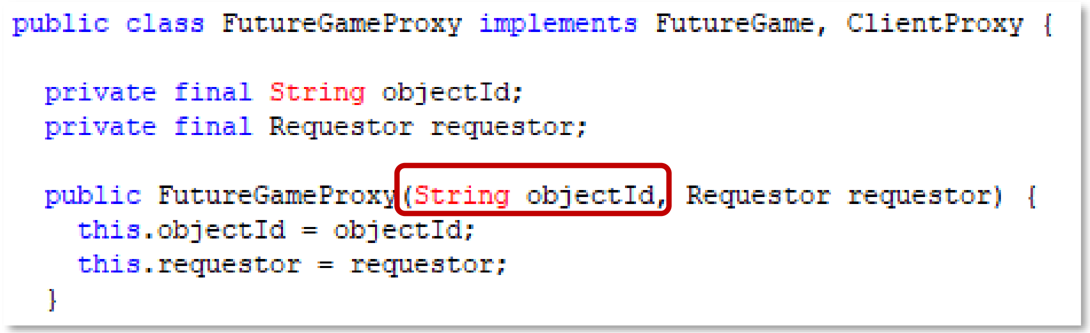
- Concept / Template
  - ClientProxy methods that return object references must
    - 1) Get an *objectId* from the server
    - 2) Create an appropriate ClientProxy, and assign that *objectId* to it



```
@Override
public FutureGame createGame(String playerName, int playerLevel) {
  String id =
    requestor.sendRequestAndAwaitReply("none",
            MarshallingConstant.GAMELOBBY_CREATE_GAME_METHOD,
            String.class, playerName, playerLevel);
  FutureGame proxy = new FutureGameProxy(id, requestor);
  return proxy;
}
```

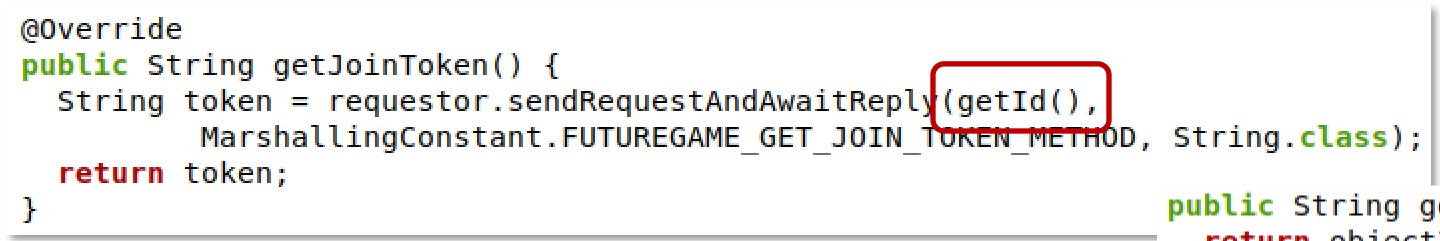
ClientProxy Constructor

- That is, the *objectId* from the server must be assigned during client proxy construction, alas, in the constructor

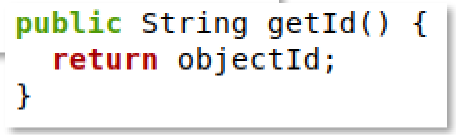


```
public class FutureGameProxy implements FutureGame, ClientProxy {

  private final String objectId;
  private final Requestor requestor;

  public FutureGameProxy(String objectId, Requestor requestor) {
    this.objectId = objectId;
    this.requestor = requestor;
  }
```

- … which is then used in all subsequent proxy methods:

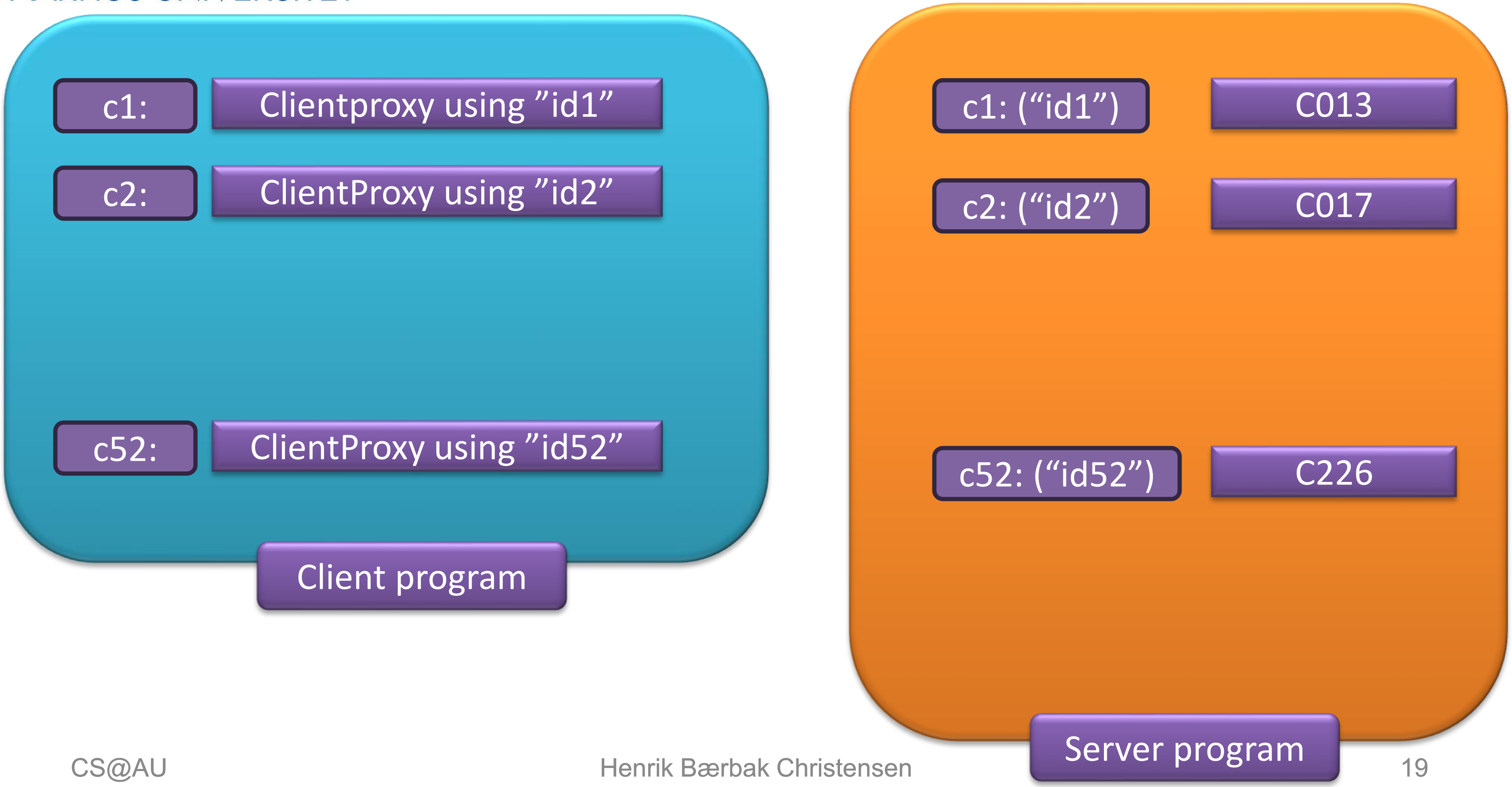


```
@Override
public String getJoinToken() {
  String token = requestor.sendRequestAndAwaitReply(getId(),
          MarshallingConstant.FUTUREGAME_GET_JOIN_TOKEN_METHOD, String.class);
  return token;
}
```



```
public String getId() {
  return objectId;
}
```

The pass-by-reference problem!



AARHUS UNIVERSITET

Server Side

How does this look?

Server: Invoker

- The Invoker’s code’s *first part* is the normal upcall...

```
if (operationName.equals(MarshallingConstant.GAMELOBBY_CREATE_GAME_METHOD)) {
  String playerName = gson.fromJson(array.get(0), String.class);
  int level = gson.fromJson(array.get(1), Integer.class);
  FutureGame futureGame = lobby.createGame(playerName, level);
```

- But, - how do we assign a unique ID to the FutureGame?

- Sigh – we have a new *responsibility someone must have*

I will assign it to...

- The Servant object

```
public FutureGameServant(String playerName, int playerLevel) {
  // Create the object ID to bind server and client side
  // Servant-ClientProxy objects together
  id = UUID.randomUUID().toString();

  [...]
}
```

Java's library for creating *unique IDs*

```
@Override
public String getId() {
  return id;
}
```

That is: Servant objects will generate a unique objectId during construction; and provide a 'getId()' method...

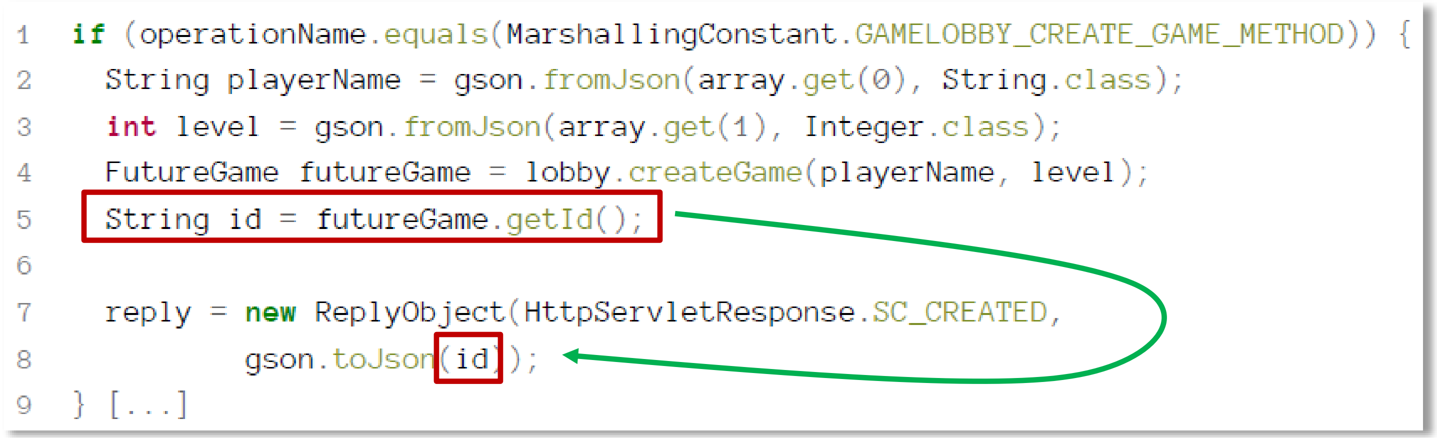
AARHUS UNIVERSITET

But...

- The responsiblity *could* have been assigned elsewhere:
  - The **invoker** that generates a unique UUID
  - A **Name Service** that we ask to assign a UUID
  - **Domain** already defines a unique ID
  - Storage tier **(RDB)** generates unique Ids / *Primary Key*
  - ...
- As usual in this course, each possible *design decision* have certain advantages and disadvantages
- Exercise:
  - What disadvantage has my decision? Advantage?

Server: Invoker

- Thus the Invoker’s code’s *last part* is different – instead of returning the servant’s return value directly (pass-by-value), it must return the return value’s *objectId* (pass-by-reference)



```
if (operationName.equals(MarshallingConstant.GAMELOBBY_CREATE_GAME_METHOD)) {
  String playerName = gson.fromJson(array.get(0), String.class);
  int level = gson.fromJson(array.get(1), Integer.class);
  FutureGame futureGame = lobby.createGame(playerName, level);
  String id = futureGame.getId();

  reply = new ReplyObject(HttpServletResponse.SC_CREATED,
          gson.toJson(id));
} [...]
```

Summary so far

1

```java
@Override
public FutureGame createGame(String playerName, int playerLevel) {
  String id =
    requestor.sendRequestAndAwaitReply("none",
            MarshallingConstant.GAMELOBBY_CREATE_GAME_METHOD,
            String.class, playerName, playerLevel);
```

2

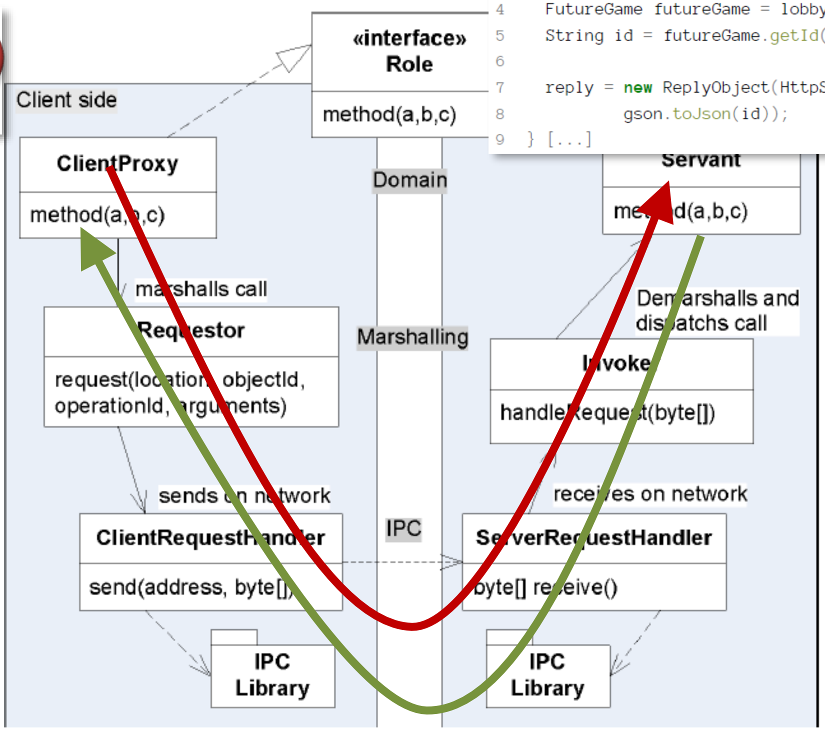
```java
if (operationName.equals(MarshallingConstant.GAMELOBBY_CREATE_GAME_METHOD)) {
  String playerName = gson.fromJson(array.get(0), String.class);
  int level = gson.fromJson(array.get(1), Integer.class);
  FutureGame futureGame = lobby.createGame(playerName, level);
  String id = futureGame.getId();
```

3

```java
  reply = new ReplyObject(HttpServletResponse.SC_CREATED,
          gson.toJson(id));
} [...]
```

4

```java
  FutureGame proxy = new FutureGameProxy(id, requestor);
  return proxy;
}
```



Next Issue

- Next…

```
// Lobby object is made in the setup/before method
FutureGame player1Future = lobby.createGame("Pedersen", 0);
assertThat(player1Future, is(not(nullValue())));
```

- But what happens when client *uses* the player1future?

```
String joinToken = player1Future.getJoinToken();
```

?

- This is now a client side *FutureGameProxy* getJoinToken() call...

Next Issue

- But what happens when client *uses* the player1future?

```
String joinToken = player1Future.getJoinToken();
```

?

- FutureGameProxy’s implementation is easy enough, the return type is just a String (pass by value)

```
@Override
public String getJoinToken() {
  String token = requestor.sendRequestAndAwaitReply(getId(),
          MarshallingConstant.FUTUREGAME_GET_JOIN_TOKEN_METHOD,
          String.class);
  return token;
}
```

AARHUS UNIVERSITET

But... In the Invoker?

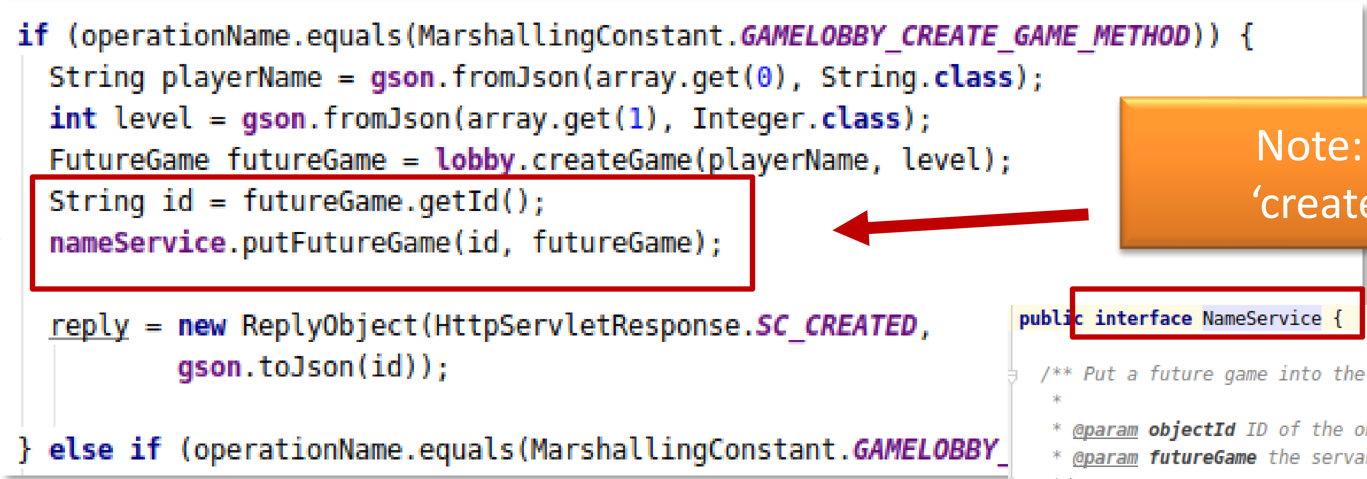
```java
if (operationName.equals(MarshallingConstant.FUTUREGAME_GET_JOIN_TOKEN_METHOD)) {
  FutureGame futureGame = ???
  String token = futureGame.getJoinToken();
  reply = new ReplyObject(HttpServletResponse.SC_OK, gson.toJson(token));
}
```

- *Someone* is *responsible* for knowing the relation between objectId and actual servant object reference!
  - *name service* ☺

- *Someone* is *responsible* for storing the relation.

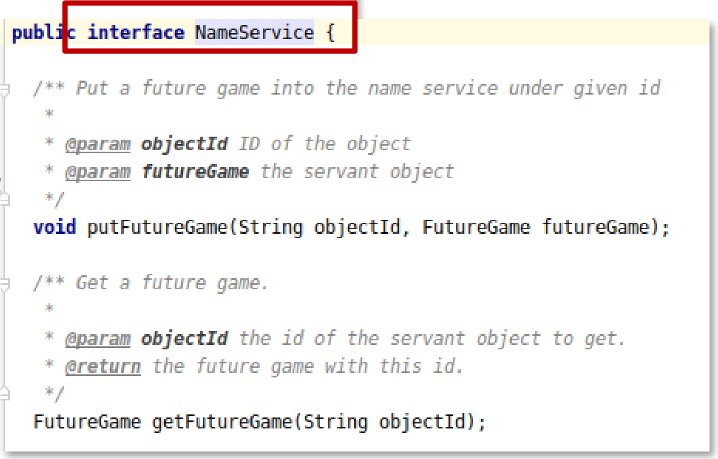
So - Update

- I let the Invoker update a *name service* at ‘create time’:



```
if (operationName.equals(MarshallingConstant.GAMELOBBY_CREATE_GAME_METHOD)) {
  String playerName = gson.fromJson(array.get(0), String.class);
  int level = gson.fromJson(array.get(1), Integer.class);
  FutureGame futureGame = lobby.createGame(playerName, level);
  String id = futureGame.getId();
  nameService.putFutureGame(id, futureGame);

  reply = new ReplyObject(HttpServletResponse.SC_CREATED,
          gson.toJson(id));

} else if (operationName.equals(MarshallingConstant.GAMELOBBY_
```

Note: this is in the ‘createGame’ code

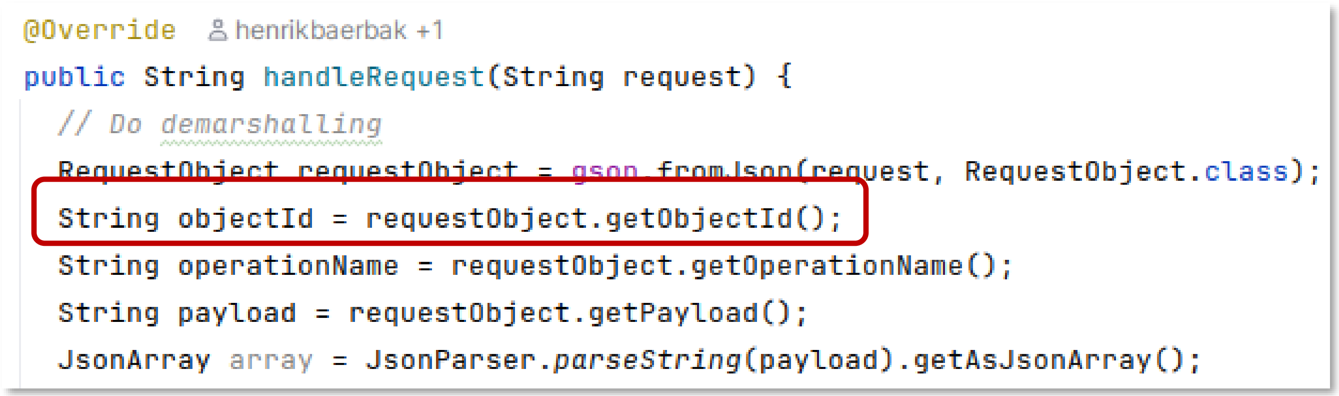


```
public interface NameService {

  /** Put a future game into the name service under given id
   *
   * @param objectId ID of the object
   * @param futureGame the servant object
   */
  void putFutureGame(String objectId, FutureGame futureGame);

  /** Get a future game.
   *
   * @param objectId the id of the servant object to get.
   * @return the future game with this id.
   */
  FutureGame getFutureGame(String objectId);
```

- A non-distributed name service

Final Piece...

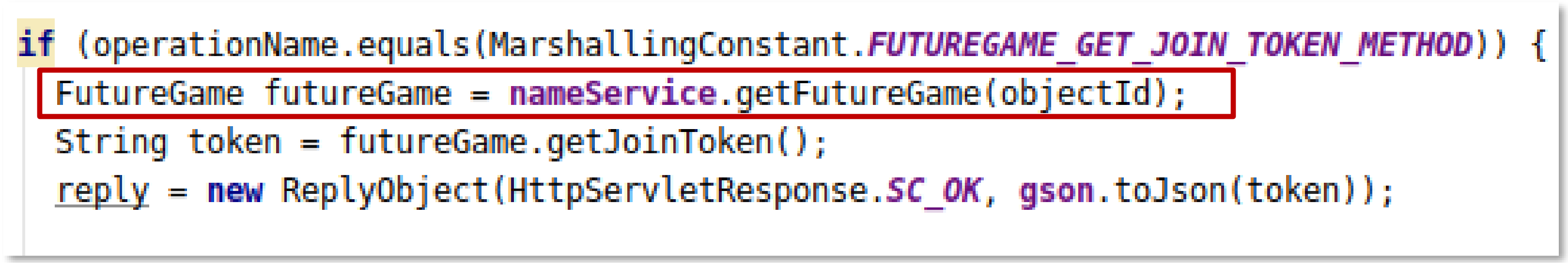
- Now Invoker code can be completed for handling the FutureGame's getJoinToken() method
  - Retrieve the object reference based upon the objectId sent by the client side proxy
    - It is part of the RequestObject that we demarshall in 'handleRequest()'



```
@Override   henrikbaerbak +1
public String handleRequest(String request) {
  // Do demarshalling
  RequestObject requestObject = gson.fromJson(request, RequestObject.class);
  String objectId = requestObject.getObjectId();
  String operationName = requestObject.getOperationName();
  String payload = requestObject.getPayload();
  JsonArray array = JsonParser.parseString(payload).getAsJsonArray();
```

Final Piece...

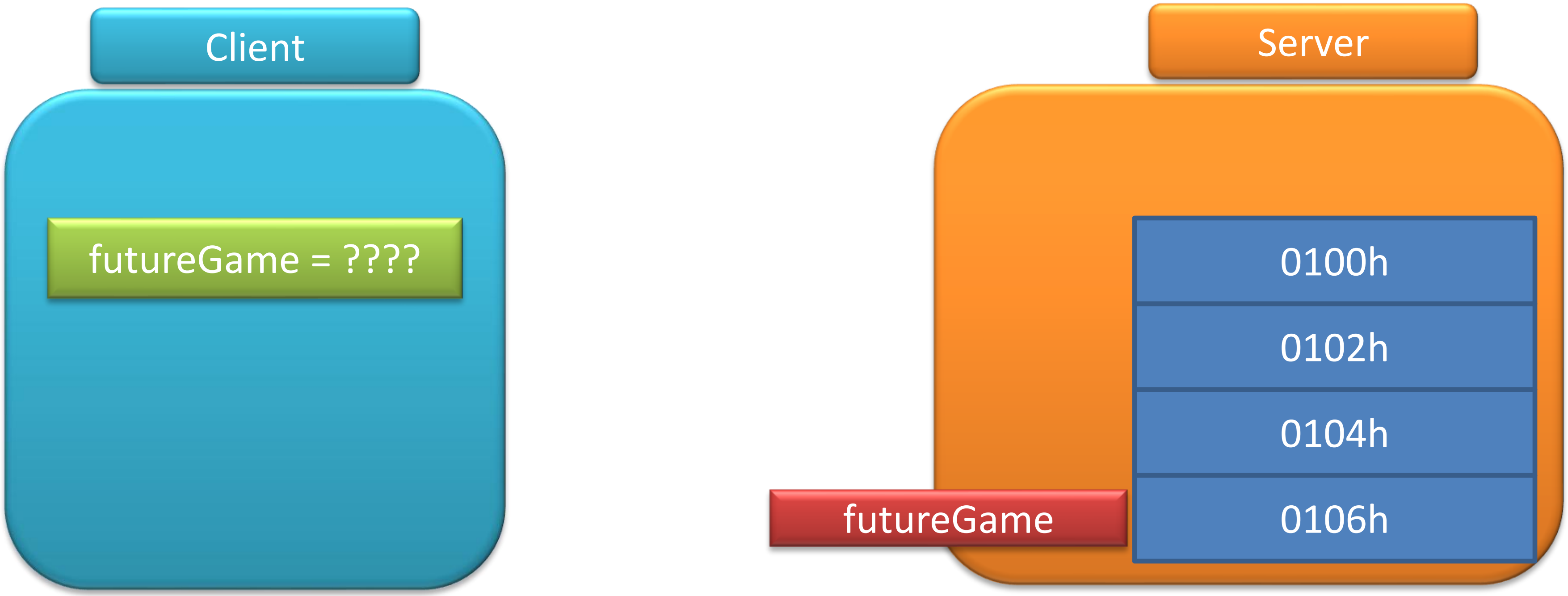
- Now Invoker code can be completed for handling the FutureGame's getJoinToken() method
  - Find the associated FutureGame object associated with 'objectId' in the name service
  - Do the upcall on that particular object
  - Marshall the return value and return that back to ServerRequestHandler



```
if (operationName.equals(MarshallingConstant.FUTUREGAME_GET_JOIN_TOKEN_METHOD)) {
  FutureGame futureGame = nameService.getFutureGame(objectId);
  String token = futureGame.getJoinToken();
  reply = new ReplyObject(HttpServletResponse.SC_OK, gson.toJson(token));
```

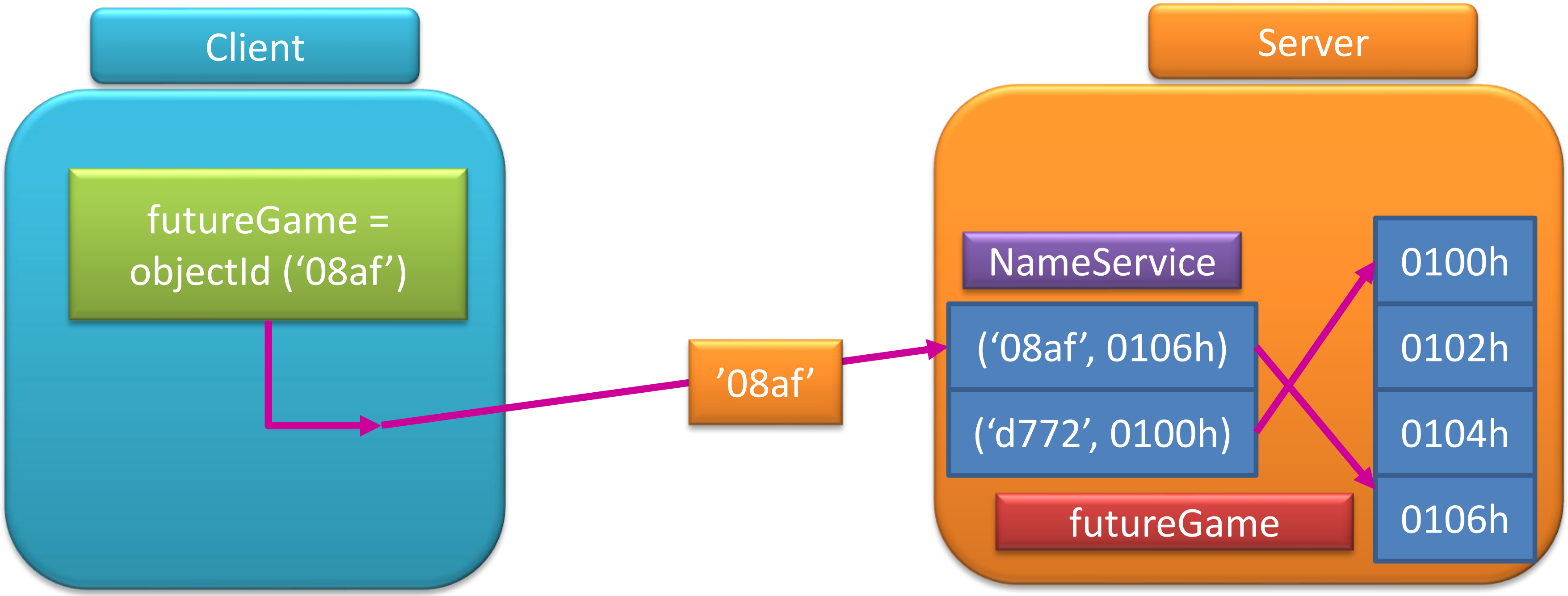
Or - in Pictures…

- Server creates a FutureGame in memory at 0106h



objectId and NameService

- The address is abstracted by a **unique objectId**



AARHUS UNIVERSITET

Discussion

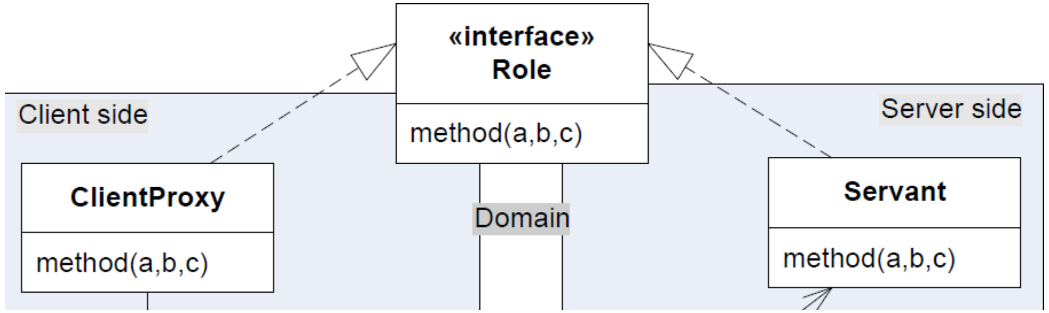
- My *name service* is an in-memory data structure
  - Does not work if server crashes ☺
  - Does not work in case of *horizontal scaling*
    - *That is: Many copies of the same game server*

- Production systems need to keep the name service directory in a tiered persistent system

– Database	Slow	RDB
– External cache	Faster	MemCached/Redis
– Internal cache	Fastest	MemCached/Redis

Discussion

- Be aware that for given a given role...



- ... Not all methods are necessarily remote calls!
- Example: method *getId()*

ClientProxy

```
@Override
public String getId() {
  return objectId;
}
```

Servant

```
@Override
public String getId() {
  return id;
}
```

Exercise

- Example: method *getId()*

ClientProxy

```
@Override
public String getId() {
  return objectId;
}
```

Servant

```
@Override
public String getId() {
  return id;
}
```

- Exercise:
  – Why is it ***extremely stupid*** to make the client side proxy ‘getId()’ into a remote method?

AARHUS UNIVERSITET

Role Interfaces Again

- Both our Servant and Proxy objects must be ‘identifiable’, ala, implement the ‘getId()’ method…
- In my HotStone I have a Role Interface for that…

```
package hotstone.roleinterface;

public interface Identifiable {
  String getID();
}
```

```
public interface Card extends Effectable, Identifiable, Attributable {
```

```
public interface Hero extends Effectable, Identifiable {
```

Pass-by-Ref – but only *one way*

- So – status… What can we do?
  - Pass by value

• From Server to Client	ala return values
• From Client to Server	ala arguments in parameter lists

  - Pass by reference

• From Server to Client	ala objectId, proxies, name service
• From Client to Server	yes and no!

Client-to-Server References

- We actually have two cases
  - A) Client creates an object, pass the ref to the server
  - B) Client pass the ref of an object *on the server* to the server
- The first case, A), is not possible with FRDS.Broker.
  - It would allow the server to call methods on client objects…
    - Generally, a bad idea (security, availability, performance)
- The second case, B), is OK.
  - … How ?

AARHUS UNIVERSITET

Client pass Object Ref

- Exercise: How do a client pass an object ref?

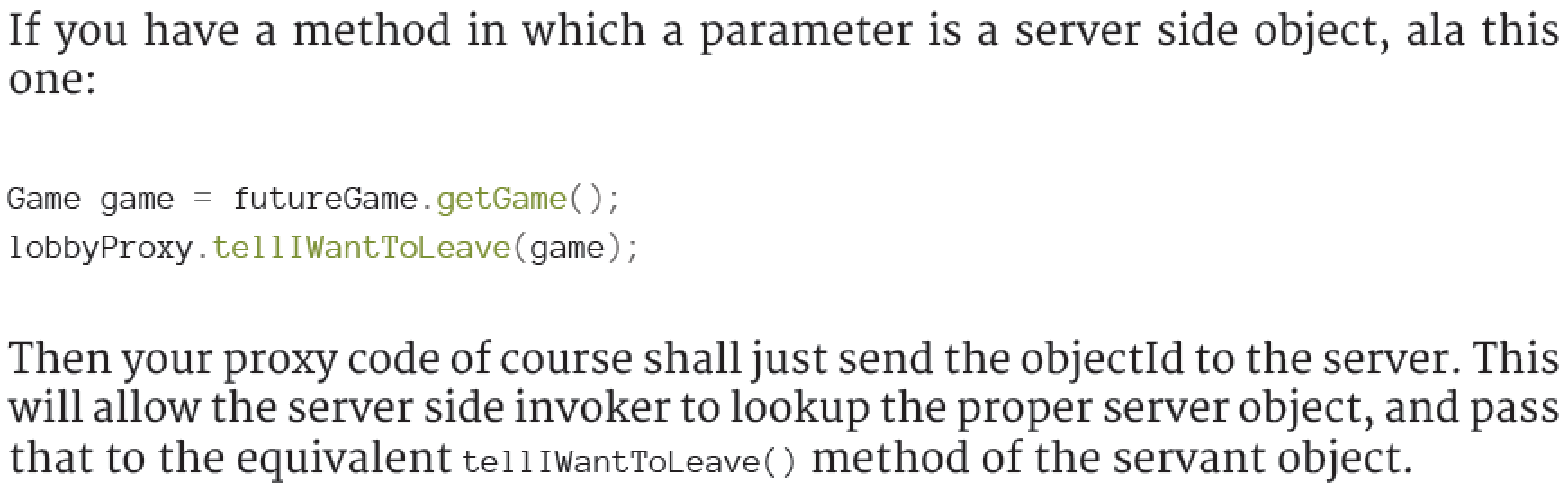
```
public Status playCard(Player who, Card card, int atIndex);
```

- The GameClientProxy must
  - ‘sendRequestAndAwaitReply(objId, opName, Status.class, who, **WHAT?**, atIndex);
  - *We cannot replace ‘WHAT?’ with ‘card’ because ‘card’ is a client side object reference, referring to a CardClientProxy.*
    - *The server has no idea what that memory adress is!*

- Exercise: What does the client need to send instead?

Client-to-Server References

- The technique is:



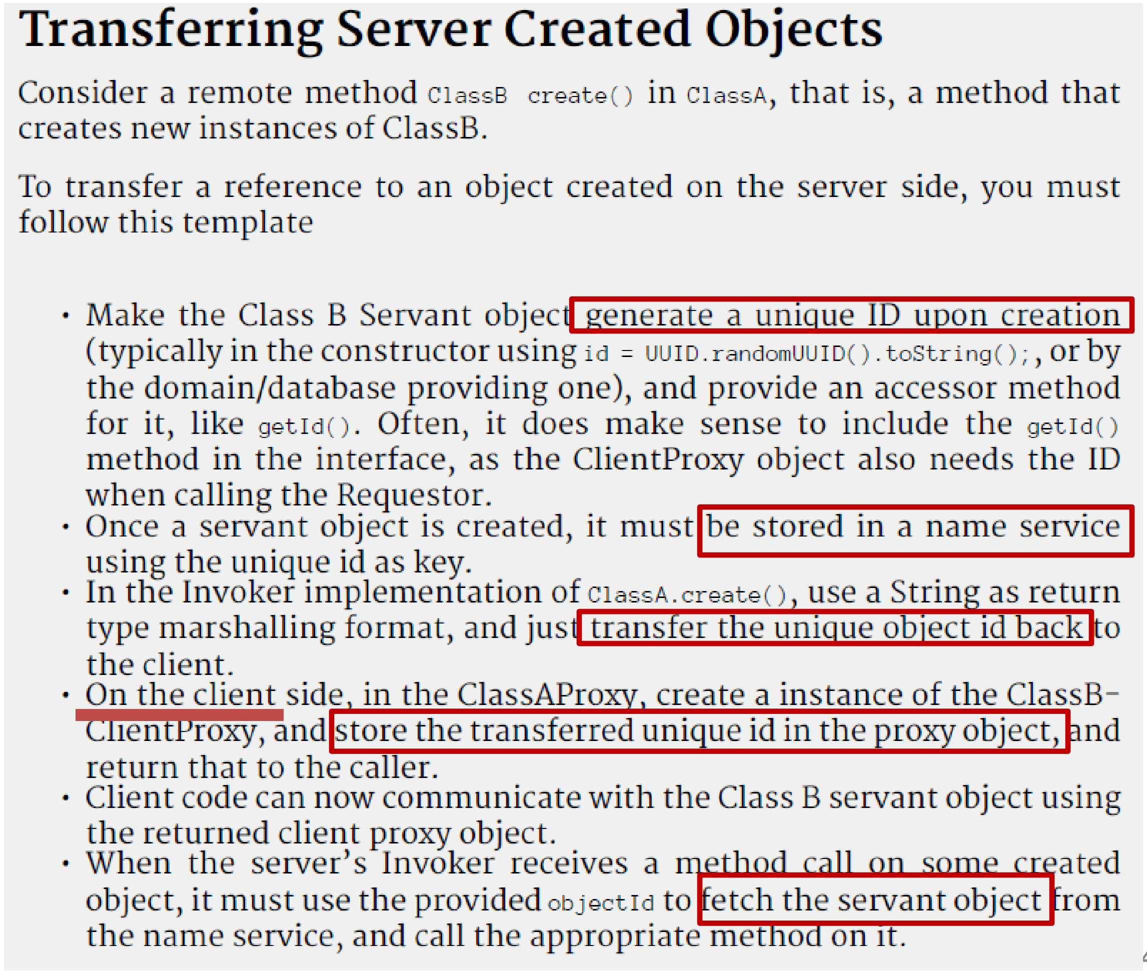
If you have a method in which a parameter is a server side object, ala this one:

```
Game game = futureGame.getGame();
lobbyProxy.tellIWantToLeave(game);
```

Then your proxy code of course shall just send the objectId to the server. This will allow the server side invoker to lookup the proper server object, and pass that to the equivalent `tellIWantToLeave()` method of the servant object.

Summary

- Recipe:
  - createObject()



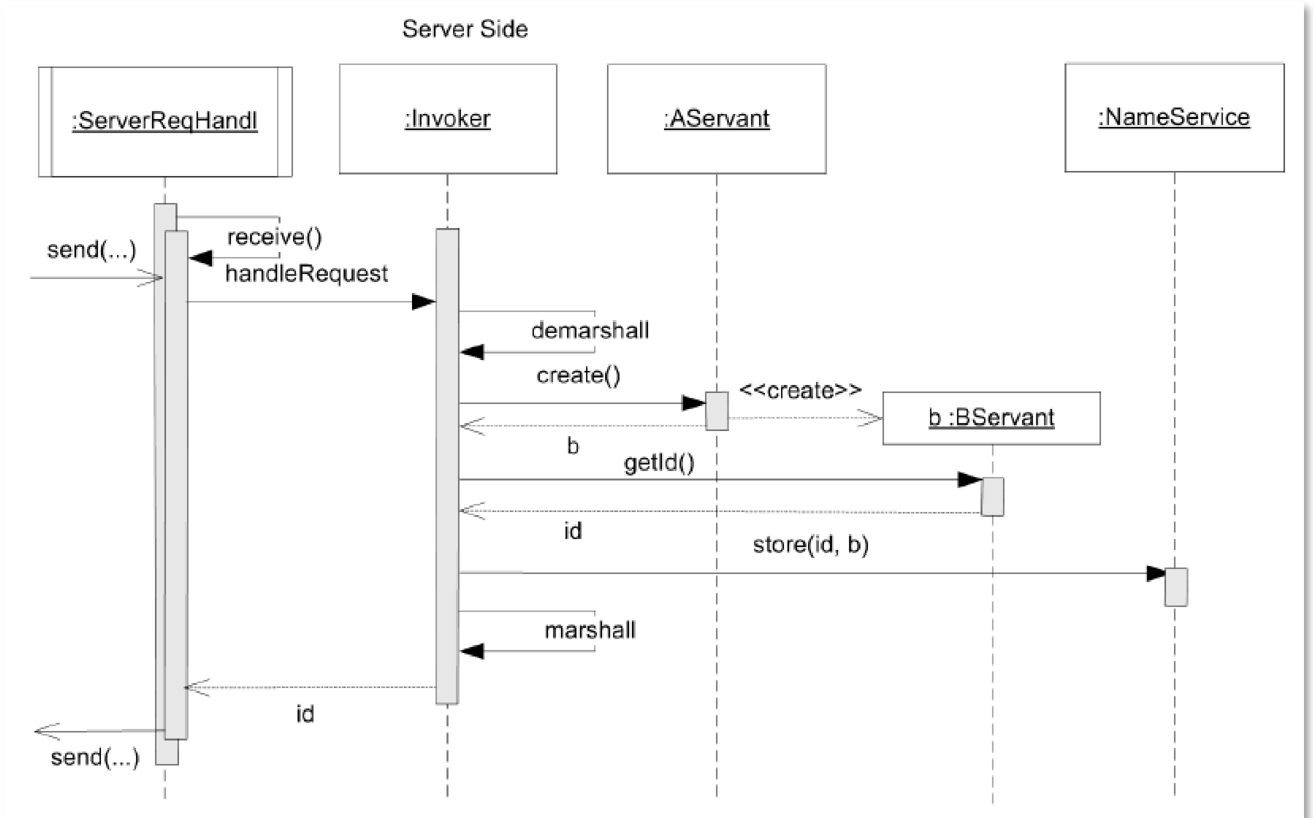
Transferring Server Created Objects

Consider a remote method `ClassB create()` in `ClassA`, that is, a method that creates new instances of ClassB.

To transfer a reference to an object created on the server side, you must follow this template

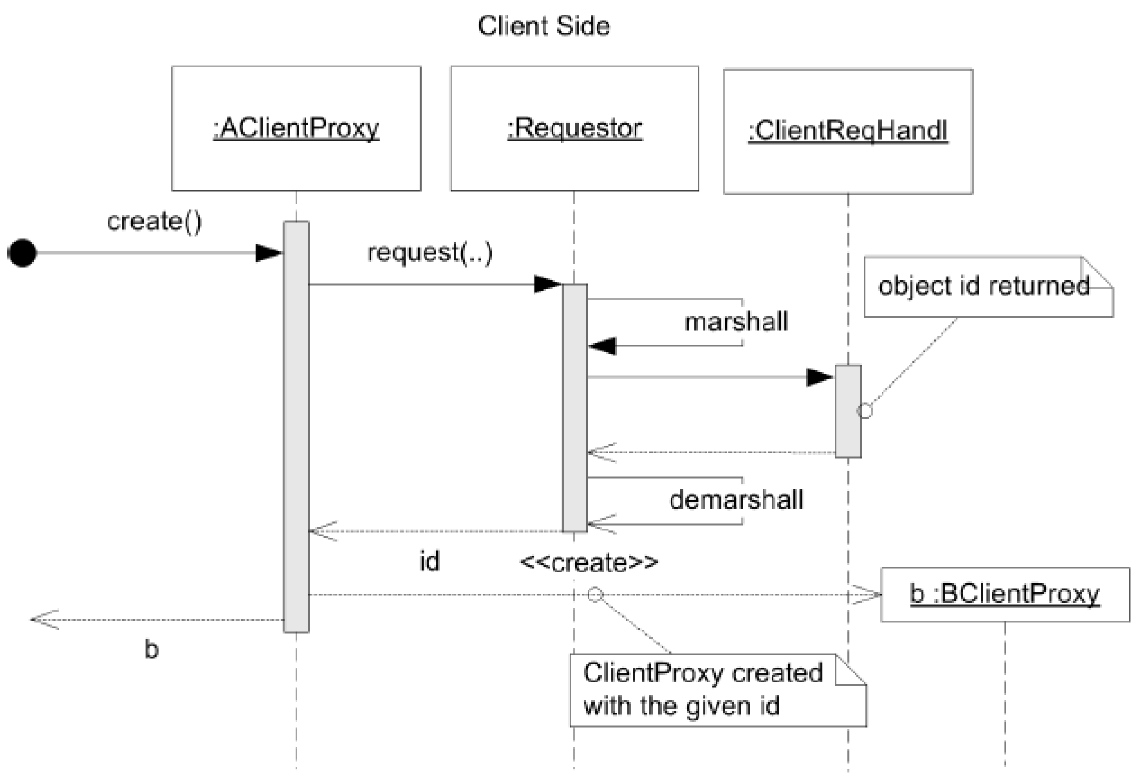
- Make the Class B Servant object generate a unique ID upon creation (typically in the constructor using `id = UUID.randomUUID().toString();`, or by the domain/database providing one), and provide an accessor method for it, like `getId()`. Often, it does make sense to include the `getId()` method in the interface, as the ClientProxy object also needs the ID when calling the Requestor.
- Once a servant object is created, it must be stored in a name service using the unique id as key.
- In the Invoker implementation of `ClassA.create()`, use a String as return type marshalling format, and just transfer the unique object id back to the client.
- On the client side, in the ClassAProxy, create a instance of the ClassB-ClientProxy, and store the transferred unique id in the proxy object, and return that to the caller.
- Client code can now communicate with the Class B servant object using the returned client proxy object.
- When the server's Invoker receives a method call on some created object, it must use the provided `objectId` to fetch the servant object from the name service, and call the appropriate method on it.

Summary



Transferring Server Created Objects – Server side

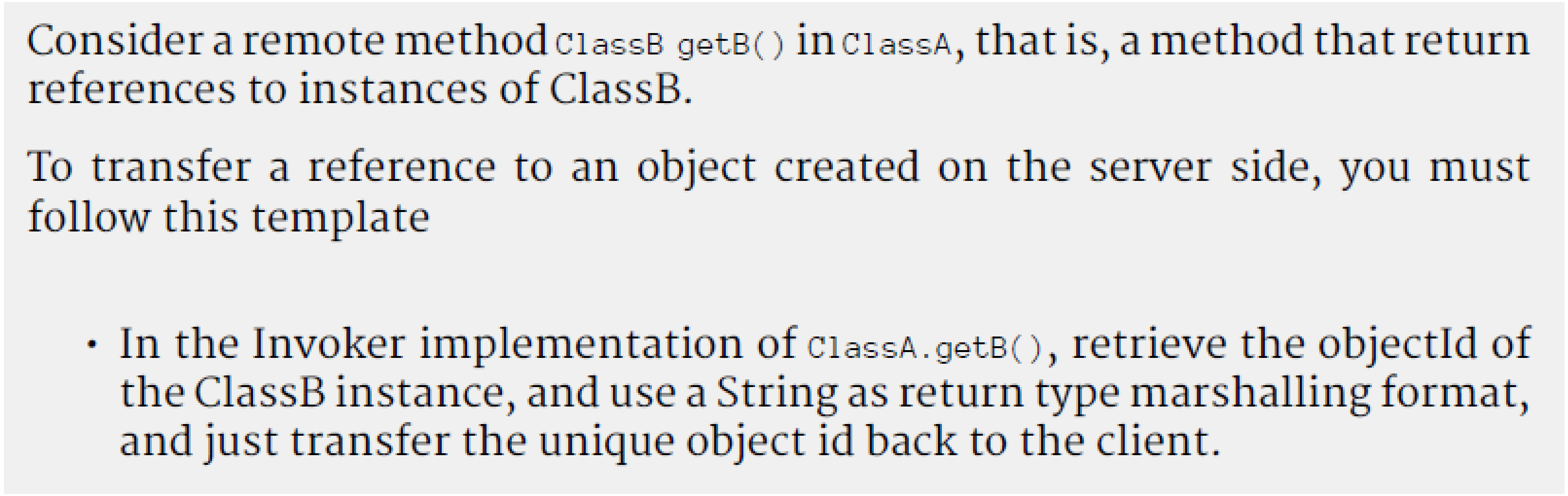
Summary



Transferring Server Created Objects – Client side

Summary

- Recipe:
  - getObject() (just getting a server side object)



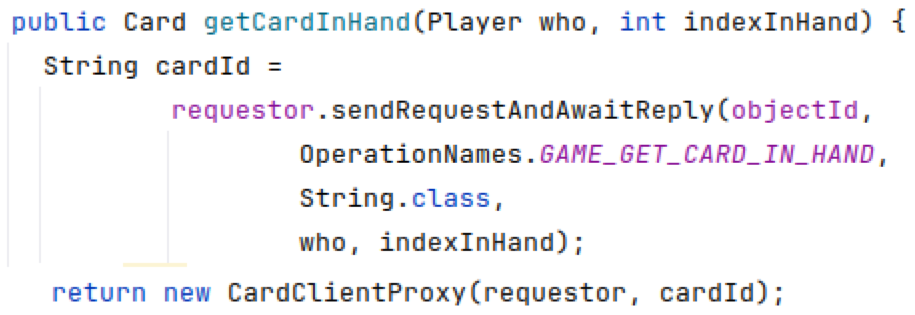
Consider a remote method `ClassB getB()` in `ClassA`, that is, a method that return references to instances of ClassB.

To transfer a reference to an object created on the server side, you must follow this template

- In the Invoker implementation of `ClassA.getB()`, retrieve the objectId of the ClassB instance, and use a String as return type marshalling format, and just transfer the unique object id back to the client.

The Proxy Explosion

- One issue:



```java
public Card getCardInHand(Player who, int indexInHand) {
  String cardId =
          requestor.sendRequestAndAwaitReply(objectId,
                  OperationNames.GAME_GET_CARD_IN_HAND,
                  String.class,
                  who, indexInHand);
  return new CardClientProxy(requestor, cardId);
```

- If I call this twenty times on the same (who, index), how many proxy objects do I create?
- What is the issue here?
  - Is it a big problem?

  - Still, can it be avoided?

AARHUS UNIVERSITET

The Client-Server Argument

Java RMI - and

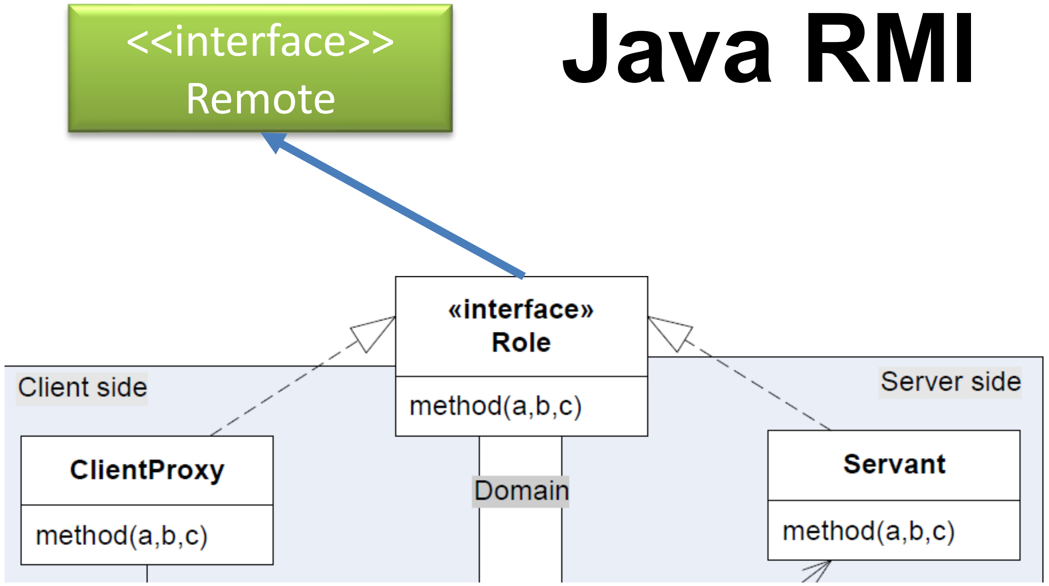
Why it was a bad idea ☺

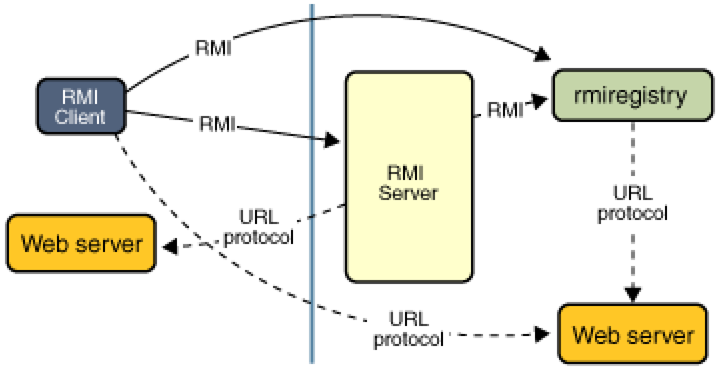
(Or a good idea that was misused)

AARHUS UNIVERSITET

Java RMI

- Idea:
  - Let Java **generate** the ClientProxy and Invoker!
- Let your Role interface extend "Remote"
  - I.e. you have already high coupling to RMI ☹
- Normal Java compile
  - Will call 'rmic' tool which will generate (see next slide…)

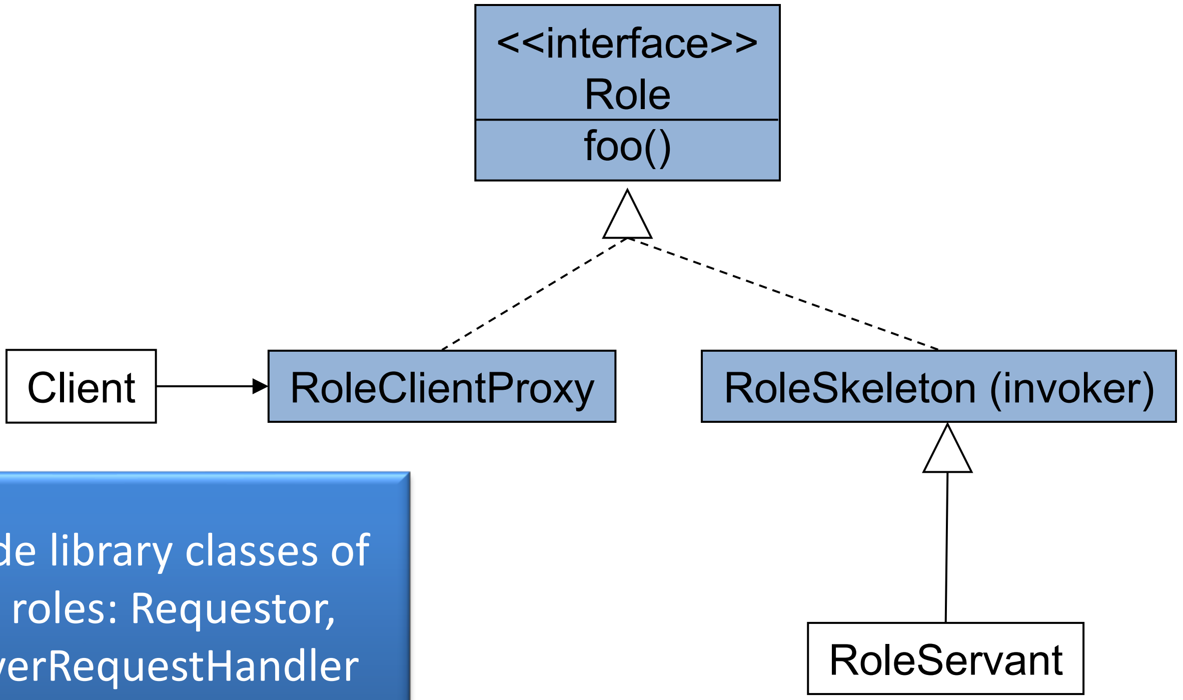




AARHUS UNIVERSITET

That is…

- The ‘rmic’ compiler will produce not one class file but two:



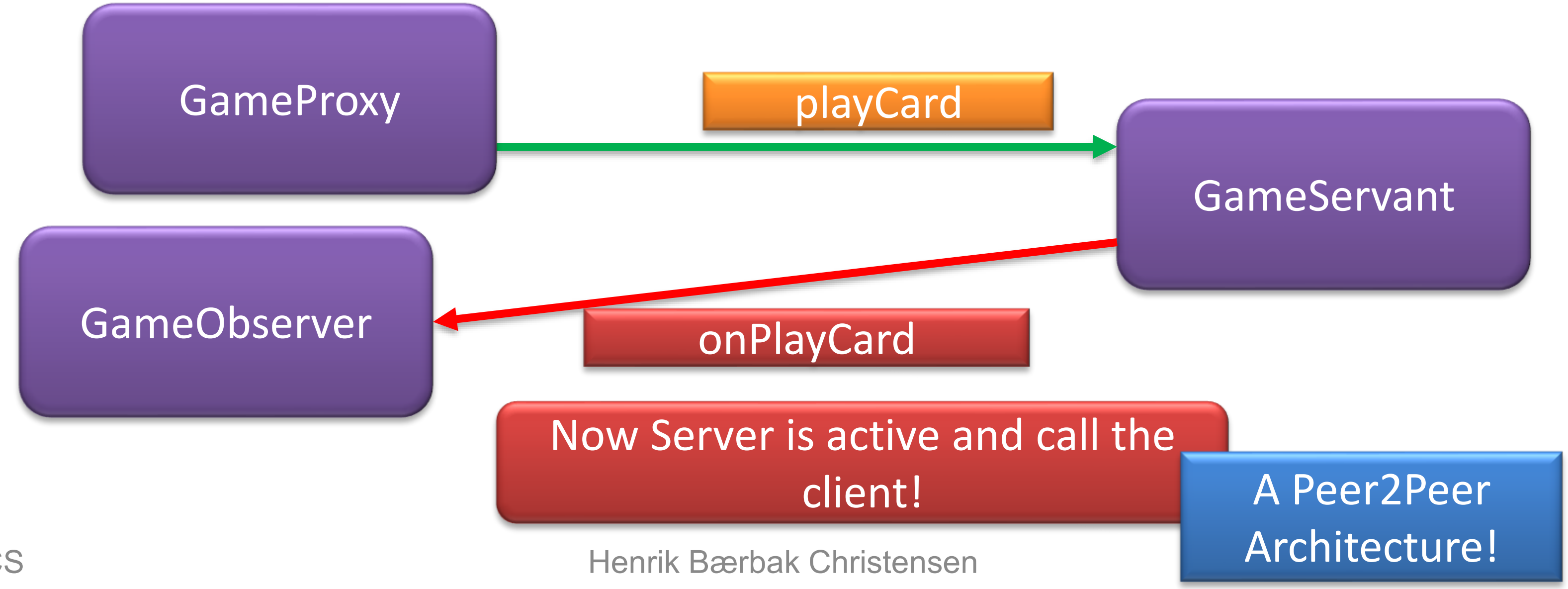
As well as provide library classes of the rest of the roles: Requestor, Client- and ServerRequestHandler

CORBA/RMI/.NET Remoting

- Early Broker systems strove to achieve one ability:
  - **Transparency**
    - **Ideally, you program as you normally do in OO, ignoring the fact that some objects where on the server**
- Thus *any (remote) object may invoke methods on any other (remote) object.*
- Observer pattern is a good case
  - Game game = new MyFantasticHotCivGame(…);
  - Drawing drawing = new CivDrawing(game);
  - game.addGameObserver(drawing);
- ***Now game and drawing can call each other, and will!***

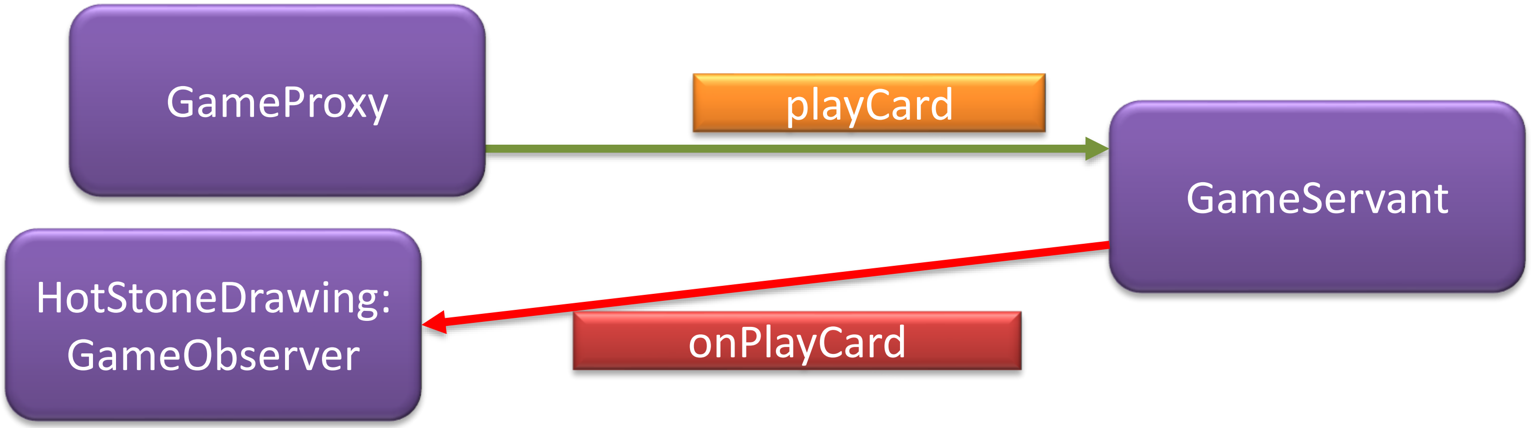
This is not Client-Server

- Client-Server Architecture
  - Many active clients that queries a single reactive server
- But the observer pattern *is two way*
  - *Both client and server are active and reactive*


GameProxy
playCard
GameServant
GameObserver
onPlayCard
Now Server is active and call the client!
A Peer2Peer Architecture!

The Peer2Peer

- Benefits
  - *Now we can actually code a server, which updates the GUI on the client!*



AARHUS UNIVERSITET

But…

- The system is more *fragile!*
  - *Clients come and go all the time while servers are much more resilient (ideally stay powered on forever, never crash).*
    - *Lots of server logic to handle disappearing clients*
  - *Servers becomes tightly coupled to clients*
    - *Server behavior relying on behavior on clients*
  - *Scaling the server is difficult*
    - *The server becomes statefull (has to know it's clients)*
    - *Horizontal scaling (more servers) is therefore harder*
  - *Performance suffers*
    - *Server calling 100.000 clients is slow!*
  - *Security is harder*
    - *The server invokes code on my machine!*

But…

- A personal belief (no scientific facts here) is that the *transparency aspect* of RMI lead architects to create ‘big ball of mud’ networks of interconnected remote objects
  - You need to be extremely aware when method invocations are remote and when not!
- Why?
  - Because the semantics is *so much different*
    - *Methods fail*
    - *Methods execute 275 times slower*
    - *Methods may be controlled by hackers…*

And Coders Then Chose REST!

- REST is ‘using HTTP as intended’

- REST architecture is *a pure client-server architecture*
  - *You always pass data purely by value*
    - *Mediatypes like XML, HTML, JSON*
      - *”No” security issue*

  - *You never see the server call back to you*

  - *ObjectId as we use is basically a ‘resource identifier’ = URL*
    - *Hotstone.littleworld.dk:5220/game/a476/card/7b534*

So

- So... To code remote systems…
- ... Software architects must *carefully and explicitly* decide which method calls/objects are remote!
- Thus, adding the extra ‘generate ID’ behavior in our remote objects are in line with this explicitly...

AARHUS UNIVERSITET

And – of course

- … You sometimes need ‘calling back to clients’
  - Games are a good example
    - Servers call back to clients when opponents do stuff!
  - Streaming vido/audio
  - Web feeds, chats fora, … WarpTalk :)
- Lot of tricks to actually do so
  - “Comet”, long polling, server-sent events, WebSockets, …
- Morale: Do it if necessary, not by accident!