AARHUS UNIVERSITET

Software Engineering and Architecture

Hints on Broker II

Mandatory

Broker II

- Learning Goal
  - Get the *return object reference* methods implemented
    - getUnitAt(p), and cousins...
  - Get the Invoker code integrated
  - Get the MiniDraw GUI integrated in a full client
- Product Goal
  - JUnit test suite that cover **all** broker related code
  - System testing of a *full HotCiv GUI based product! Wow! ☺*

Broker 2.1

- TDD the Game methods that return object references.
- That is, write JUnit tests that implement the FRDS process

Transferring Server Objects

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.

Broker 2.1

- What about Tile?

```
public interface Tile {

  /** return the tile type as a string. The set of
   * valid strings are defined by the graphics
   * engine, as they correspond to named image files.
   * @return the type type as string
   */
  public String getTypeString();
}
```

Broker 2.2

- Integrate the Invokers from last exercise, ensure that they are split into sub invokers.
- That is, remove the *fakeit* objectId code parts from last exercise, and refactor to use multi type dispatch.

Multi Type Dispatching

Consider an **Invoker** that must handle method dispatching for a large set of roles. To avoid a *blob* or *god class* invoker implementation, you can follow this template:

- Ensure your *operationId* follows a mangling scheme that allow extracting the role name. A typical way is to construct String valued *operationId* that concatenate type name and method name, with a unique seperator in between. Example: "FutureGame_getToken".
- Construct **SubInvokers** for each role. A **SubInvoker** is role specific and only handles dispatching of methods for that specific role. The **SubInvoker** implements the **Invoker** interface.
- Develop a **RootInvoker** which constructs a (key, value) map that maps from role names to sub invoker. Example: if you look up "FutureGame" you will get the sub invoker specific to the **FutureGameServant**'s methods
- Associate the **RootInvoker** with the **ServerRequestHandler**. In it's *handleRequest()* calls, it demangles the incoming *operationId* to get the role name, and use it to look up the associated **SubInvoker**, and finally delegates the *handleRequest()* call to it.

2.3 System Testing

Gui, Client, Observer???

The Grand Finale

AARHUS UNIVERSITET

Click for Drawing updates, brown blob for refresh

Unit

Moves Left:

City

Produce:

Balance:

Rfsh

State change: Moving archer to (2,2)

--> Received {"operationName":"unit_get_move_count","payload":"[]","objectId":"7a80f36d-abcb-4788-817c-028b4ca9e3c3","versionIdentity":1}
--< replied: ReplyObject [payload=1, errorDescription=null, responseCode=200]
Closing socket...
--> Accepting...
--> Received {"operationName":"unit_get_owner","payload":"[]","objectId":"30db8c31-d0c6-4d5a-8a32-0dd9b0e4846d","versionIdentity":1}
--< replied: ReplyObject [payload="RED", errorDescription=null, responseCode=200]
Closing socket...
--> Accepting...
--> Received {"operationName":"unit_get_move_count","payload":"[]","objectId":"30db8c31-d0c6-4d5a-8a32-0dd9b0e4846d","versionIdentity":1}
--< replied: ReplyObject [payload=1, errorDescription=null, responseCode=200]
Closing socket...
--> Accepting...
--> Received {"operationName":"unit_get_owner","payload":"[]","
--< replied: ReplyObject [payload="RED", errorDescription=null,

System Testing.

Story: Pedersen and Findus play a game of HotCiv

Disclaimer: Screenshot is a Unit Test ☺

Observer so far

AARHUS UNIVERSITET

- There are *two* observers
  - Both Servant and ClientProxy have one
- Last Iteration
  - Made them shut up…

```
@Before
public void setup() {
  Game servant = new StubGame3();
  GameObserver nullObserver = new NullObserver();
  servant.addObserver(nullObserver);

  Invoker invoker = new HotCivGameInvoker(servant);

  ClientRequestHandler crh =
          new LocalMethodClientRequestHandler(invoker);

  Requestor requestor = new StandardJSONRequestor(crh);

  game = new GameProxy(requestor);
  game.addObserver(nullObserver);
}
```

- It works well because
  - What is the role of the Observer?
  - *To inform the GUI about state changes in the Game domain*
  - *And – we had no GUI ☺*

AARHUS UNIVERSITET

But CivDrawing Observes Game

- From the Iteration 8 exercise, the Minidraw *Drawing* role have to *observe* on Subject Game

```
public CivDrawing( DrawingEditor editor, Game game ) {
  super();
  this.delegate = new StandardDrawing();
  this.game = game;
  this.figureMap = new HashMap<>();

  // register this unit drawing as listener to any game state
  // changes...
  game.addObserver(this);
  // ... and build up the set of figures associated with
  // units in the game.
  defineUnitMap();
  // and the set of 'icons' in the status panel
  defineIcons();
}
```

Now, CivDrawing will be notified about events in the Game, like *worldChangedAt(p)*

```
public void worldChangedAt(Position pos) {
  // TODO: Remove system.out debugging output
  System.out.println( "CivDrawing: world changes at "+pos);
  clearSelection();
  // this is a really brute-force algorithm: destroy
  // all known units and build up the entire set again
  removeAllUnitFigures();
  defineUnitMap();
}
```

AARHUS UNIVERSITET

But...

- So – requirements collide here ☹
  - GUI can only update based on Observer events from Game
  - FRDS.Broker insists on not making a remote observer possible because *it only supports passing value objects from client to server*

So What?

- *I suggest to take a crude and implementation-wise cheap route instead...*

AARHUS UNIVERSITET

Observer on the ClientProxy

- We simply implement the Subject behaviour in the GameClientProxy as usual, ala

```
@Override
public boolean moveUnit(Position from, Position to) {
  Boolean valid = requestor.sendRequestAndAwaitReply(MarshallingConstants.GAME_SINGLETON_OBJECT_ID,
      MarshallingConstants.GAME_MOVE_UNIT, Boolean.class, from, to);
  observer.worldChangedAt(from);
  observer.worldChangedAt(to);
  return valid;
}
```

Sorry, incomplete Observer here ☹

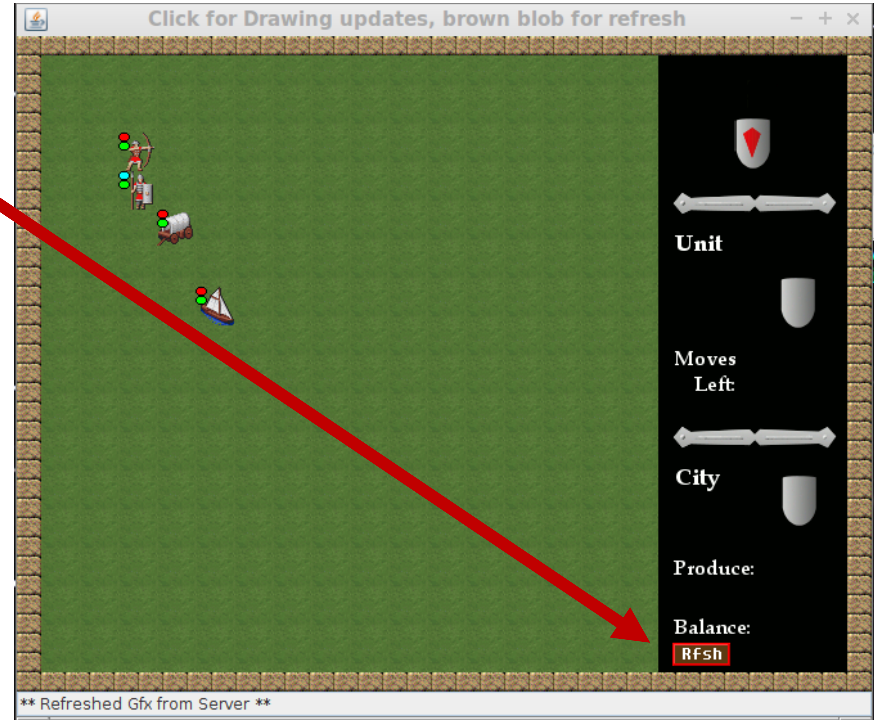
- Exercise:
  - What do we achieve by that?
  - What do we NOT achieve?

Client Manual Refresh

- Introducing the (ugly) refresh button
  - In GfxConstants

```
public static final int REFRESH_BUTTON_X = 510;
public static final int REFRESH_BUTTON_Y = 472;
```

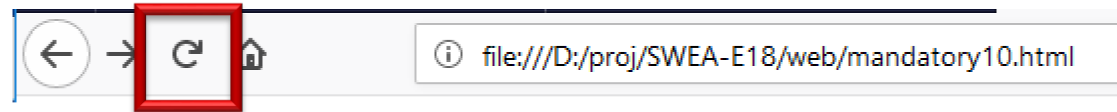
- Update your CompositionTool so *mouseUp()* events on button call the Drawing role's *requestUpdate()*
  - *editor.drawing().requestUpdate()*




```
@Override
public void requestUpdate() {
  // A request has been issued to repaint
  // everything. We simply rebuild the
  // entire Drawing.
  defineUnitMap();
  defineIcons();
  // TODO: Cities pending
}
```

Discussion

- Does this have a flavour of *being tacked on?*
- Yep! But still 'refresh' is quite common ☺:



- Alternative: Optional Exploration Exercise
  - Make server side observer that keeps log of state events with a incrementing sequence number (index in array may suffice)
  - Make a client side observer that periodically poll the server to fetch a list of all stateevents since the last fetched event
    - getEventsSinceEventWithID(47) or something like that...
  - Alternative: *long polling method call…*

AARHUS UNIVERSITET

Chatty Interface

AARHUS UNIVERSITET

We pay a (huge) penalty...

- In the MiniDraw exercise, I advocated a crude update policy

```
public void worldChangedAt(Position pos) {
  // TODO: Remove system.out debugging output
  System.out.println( "CivDrawing: world changes at "+pos);
  clearSelection();
  // this is a really brute-force algorithm: destroy
  // all known units and build up the entire set again
  removeAllUnitFigures();
  defineUnitMap();
}
```

Will rebuild by query all positions for the units on them. This is 256 remote calls!

```
for ( int r = 0; r < GameConstants.WORLDSIZE; r++ ) {
  for ( int c = 0; c < GameConstants.WORLDSIZE; c++ ) {
    p = new Position(r,c);
    Unit unit = game.getUnitAt(p);
    if ( unit != null ) {
      String type = unit.getTypeString();
      // convert the unit's Position to (x,y) coordinates
      Point point = new Point( GfxConstants.getXFromColumn(p.getColumn()),
                               GfxConstants.getYFromRow(p.getRow()) );
      UnitFigure unitFigure =
        new UnitFigure( type, point, unit );
      unitFigure.addFigureChangeListener( this);
      figureMap.put(unit, unitFigure);
```

Chatty Interface

- That is, the client is *chatty*.
  - Networks are slow so it takes awfully long time to complete...
- Build **Chunky Interfaces** for remote calls
- *Implement client side caching in* getUnitAt(p)
  - *First call:* ***bulk transfer all unit information, store in internal data structure (the cache), and make a timestamp***
  - *Next calls:* if less than N seconds since cache was populated, just return cache contents. If more than N seconds then reload from the server again.
  - *N = 2 is enough to convert 256 calls into one!*
    - *Constants do matter a lot in real computing, it is not just O(n) ☺*

AARHUS UNIVERSITET

Conclusions

Happy Coding...

Post/Mail in case if unforseen problems!!!