



AARHUS UNIVERSITET

Software Engineering and Architecture

Broker II: Dispatching

Dispatching



A **train dispatcher** (US), **rail traffic controller** (Canada), **train controller** (Australia) or **signalman** (UK), is employed by a railroad to direct and facilitate the movement of trains over an assigned territory, which is usually part, or all, of a railroad operating division. The dispatcher is also responsible for cost effective movement of trains and other on-track railroad equipment to optimize physical (trains) and human resource (crews) assets.^[1][\[full citation needed\]](#)

In computing, *dispatchers* are responsible for distributing incoming messages efficiently



Invoker Becomes *The Blob*

- Consider HotCiv's Invoker's 'handleRequest':

```
1 // === GAME
2 if (operationName.equals(MarshallingConstants.GAME_GET_PLAYER_IN_TURN)) {
3     ...
4 } else if (operationName.equals(MarshallingConstants.GAME_END_OF_TURN)) {
5
6     [lots of more if clauses removed]
7
8     // === UNIT
9 } else if (operationName.equals(MarshallingConstants.UNIT_GET_OWNER)) {
10
11     [lots of more if clauses removed]
12
13     // === CITY
14 } else if (operationName.equals(MarshallingConstants.CITY_GET_OWNER)) {
15     ...
```



Blobs do not scale

- Consider 20 remote roles
 - Thus the invoker handles twenty different *servant classes*
- ... With each 10 methods
 - Thus the invoker handles 200 methods...
- **That is an 'if () else if () else if() else if()' with 200 branches... ☹**



Composition!

- *Favor object composition*
 - *Instead of one object doing it all, delegate to specialists*
 - *Let someone else do the dirty job*
- Insight:

Let us have *one Invoker per role* in the system.
Let a 'root invoker' determine which invoker to delegate to.

- So, I have actually prepared for this in my marshalling
 - Three classes and *three prefixes on the method names*

```
public class MarshallingConstant {  
  
    // Type prefixes  
    public static final String GAME_LOBBY_PREFIX = "gamelobby";  
    public static final String FUTUREGAME_PREFIX = "futuregame";  
    public static final String GAME_PREFIX = "game";  
  
    // Method ids for marshalling  
    public static final String GAMELOBBY_CREATE_GAME_METHOD = GAME_LOBBY_PREFIX + "_create_game_method";  
    public static final String GAMELOBBY_JOIN_GAME_METHOD = GAME_LOBBY_PREFIX + "_join_game_method";  
  
    public static final String FUTUREGAME_GET_JOIN_TOKEN_METHOD = FUTUREGAME_PREFIX + "_get_join_token_method";  
    public static final String FUTUREGAME_IS_AVAILABLE_METHOD = FUTUREGAME_PREFIX + "_is_available_method";  
    public static final String FUTUREGAME_GET_GAME_METHOD = FUTUREGAME_PREFIX + "_get_game_method";  
  
    public static final String GAME_GET_PLAYER_NAME = GAME_PREFIX + "_get_player_name_method";  
    public static final String GAME_GET_PLAYER_IN_TURN = GAME_PREFIX + "_get_player_in_turn_method";  
    public static final String GAME_MOVE = GAME_PREFIX + "_move_method";  
}
```



Name Mangling

AARHUS UNIVERSITET

In **compiler** construction, **name mangling** (also called **name decoration**) is a technique used to solve various problems caused by the need to resolve unique names for programming entities in many modern **programming** languages.

It provides a way of encoding additional information in the name of a **function**, **structure**, **class** or another **datatype** in order to pass more semantic information from the **compilers** to **linkers**.

```
public class MarshallingConstant {  
  
    // Type prefixes  
    public static final String GAME_LOBBY_PREFIX = "gamelobby";  
    public static final String FUTUREGAME_PREFIX = "futuregame";  
    public static final String GAME_PREFIX = "game";  
  
    // Method ids for marshalling  
    public static final String GAMELOBBY_CREATE_GAME_METHOD = GAME_LOBBY_PREFIX + "_create_game_method";  
    public static final String GAMELOBBY_JOIN_GAME_METHOD = GAME_LOBBY_PREFIX + "_join_game_method";  
  
    public static final String FUTUREGAME_GET_JOIN_TOKEN_METHOD = FUTUREGAME_PREFIX + "_get_join_token_method";  
    public static final String FUTUREGAME_IS_AVAILABLE_METHOD = FUTUREGAME_PREFIX + "_is_available_method";  
    public static final String FUTUREGAME_GET_GAME_METHOD = FUTUREGAME_PREFIX + "_get_game_method";  
  
    public static final String GAME_GET_PLAYER_NAME = GAME_PREFIX + "_get_player_name_method";  
    public static final String GAME_GET_PLAYER_IN_TURN = GAME_PREFIX + "_get_player_in_turn_method";  
    public static final String GAME_MOVE = GAME_PREFIX + "_move_method";  
}
```

My method names includes the name of the class

So, I Delegate

- The Invoker simply looks up the associated Invoker

```
public class GameLobbyRootInvoker implements Invoker {
```

```
@Override
public ReplyObject handleRequest(String objectId, String operationName, String payload) {
    ReplyObject reply = null;

    // Identify the invoker to use
    String type = operationName.substring(0, operationName.indexOf('_'));
    Invoker subInvoker = invokerMap.get(type);

    // And do the upcall
    try {
        reply = subInvoker.handleRequest(objectId, operationName, payload);
    } catch (UnknownServantException e) {
        reply =
            new ReplyObject(
                HttpServletResponse.SC_NOT_FOUND,
                e.getMessage());
    }

    return reply;
}
```

Extract the class
name

Sorry Magic
Constant ☹️

Setting up the Lookup

- Have to initialize the *root invoker*

```
public GameLobbyRootInvoker(GameLobby lobby) {
    this.lobby = lobby;
    gson = new Gson();

    objectStorage = new InMemoryObjectStorage();
    invokerMap = new HashMap<>();

    // Create an invoker for each handled type/class
    // and put them in a map, binding them to the
    // operationName prefixes
    Invoker gameLobbyInvoker = new GameLobbyInvoker(lobby, objectStorage, gson);
    invokerMap.put(MarshallingConstant.GAME_LOBBY_PREFIX, gameLobbyInvoker);
    Invoker futureGameInvoker = new FutureGameInvoker(objectStorage, gson);
    invokerMap.put(MarshallingConstant.FUTUREGAME_PREFIX, futureGameInvoker);
    Invoker gameInvoker = new GameInvoker(objectStorage, gson);
    invokerMap.put(MarshallingConstant.GAME_PREFIX, gameInvoker);
}
```



Smaller, Type-specific, Invokers

- Achieve *high cohesion* in the type specific invokers

```
public class FutureGameInvoker implements Invoker {  
  
    @Override  
    public ReplyObject handleRequest(String objectId, String operationName, String payload) {  
        ReplyObject reply = null;  
  
        // Demarshall parameters into a JSONArray  
        JsonParser parser = new JsonParser();  
        JSONArray array =  
            parser.parse(payload).getAsJSONArray();  
  
        if (operationName.equals(MarshallingConstant.FUTUREGAME_GET_JOIN_TOKEN_METHOD)) {  
            FutureGame futureGame = objectStorage.getFutureGame(objectId);  
            String token = futureGame.getJoinToken();  
            reply = new ReplyObject(HttpServletResponse.SC_OK, gson.toJson(token));  
        } else if (operationName.equals(MarshallingConstant.FUTUREGAME_IS_AVAILABLE_METHOD)) {  
            FutureGame futureGame = objectStorage.getFutureGame(objectId);  
            boolean isAvailable = futureGame.isAvailable();  
            reply = new ReplyObject(HttpServletResponse.SC_OK, gson.toJson(isAvailable));  
        } else if (operationName.equals(MarshallingConstant.FUTUREGAME_GET_GAME_METHOD)) {  
            FutureGame futureGame = objectStorage.getFutureGame(objectId);  
            Game game = futureGame.getGame();  
            String id = game.getId();  
            reply = new ReplyObject(HttpServletResponse.SC_OK, gson.toJson(id));  
        }  
  
        return reply;  
    }  
}
```

- Conclusion:

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.