City Building Education

Moldovan Paul Andrei

30235

Contents

1. Iteration 0

The goal of the project is to create an application that deals with a layered representation of the world. Each layer represents different aspects of the world like land types, hydro types, or places types. The places layer includes tiles that represents buildings. Those tiles create a district and more district a city. Each tile from the places layer contains information about that place like address, contact info, capacity, availability, rating. Each tile has x and y coordinates and a tile type. The tile type contains a rating for that tile that will influence the district/city rating later in the project. For each layer type there are different tile types. For example for hydro layer I include river and sea tile types.

My project is structured on 4 different layers: Presentation, Business, Data Access, Common.

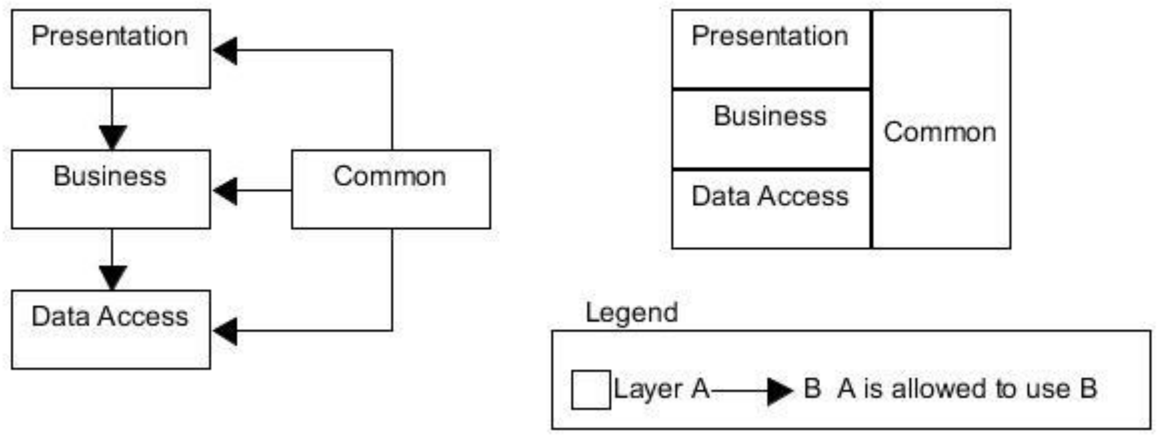


Fig.1.1. Project structure – Iteration 0

The Data Access layer manages the storage logic, the communication with the database. The Business layer contains objects and entities and this is where all the logic is handled. The Presentation layer contains what the user sees and interacts with. The Common layer is the start point of the project.

For the Business Layer I chose the Domain Model pattern. This pattern defines objects for each table from the database. Each object handles a part of the business logic.

For the Data Access Layer I chose the Data Mapper pattern because it is simple to map table columns to equivalent attributes of a business class and because this pattern it is mostly used with the Domain Model one.

1. MoSCoW Chart

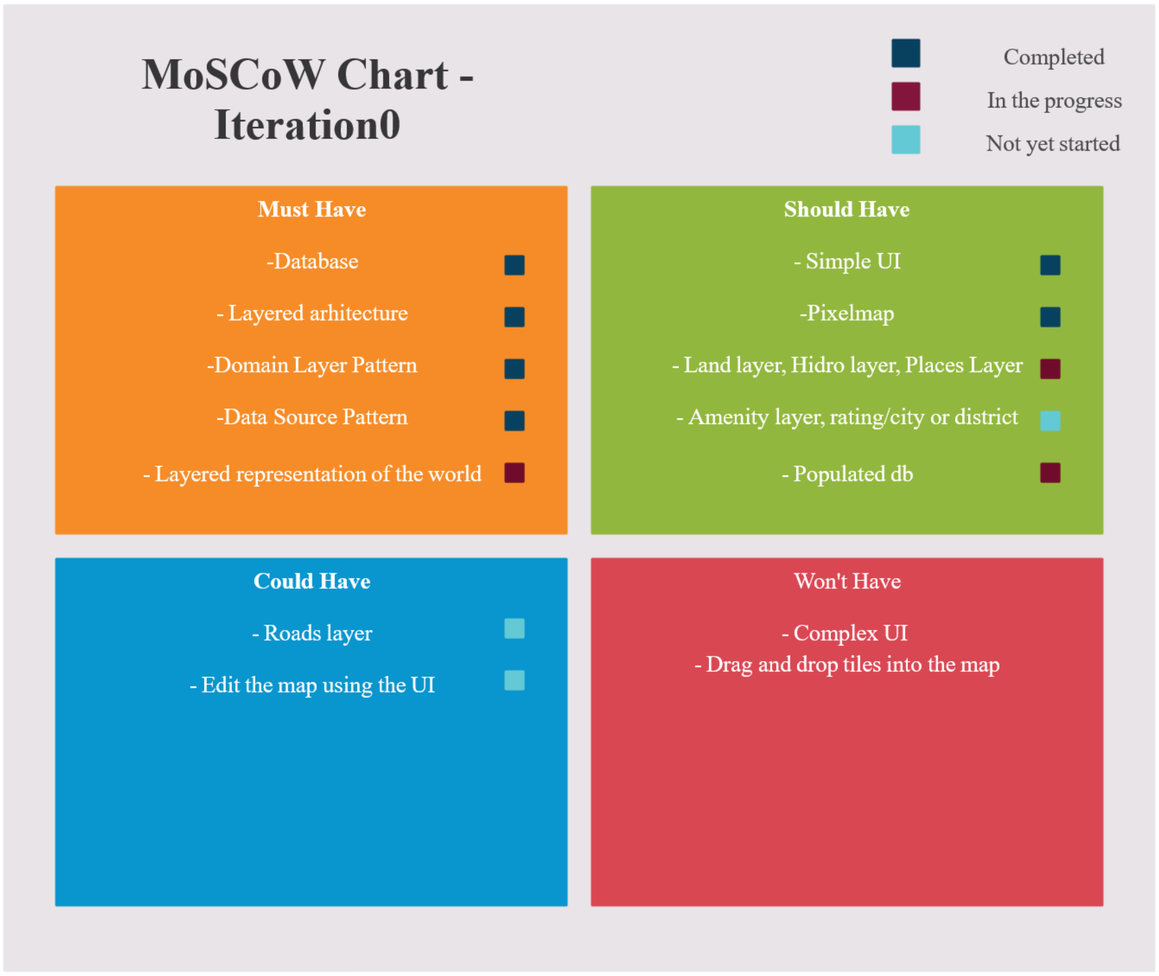


Fig.1.2 MoSCoW Chart – Iteration 0

2. Database diagram

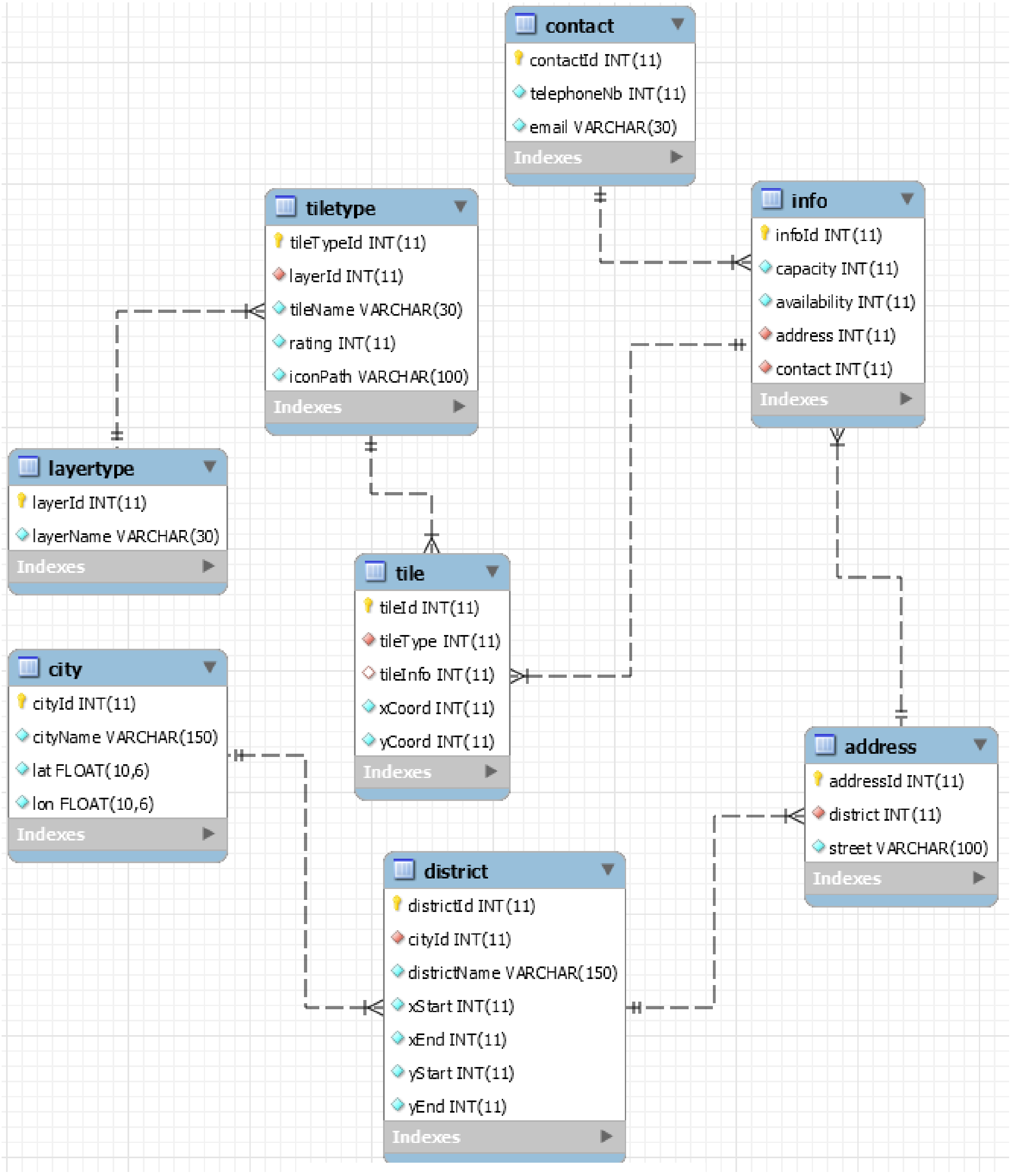


Fig.1.3 Database diagram – Iteration 0

2. Iteration 1

1. MVC Architectural pattern

MVC separates the application into three packages: Model, View and Controller. Model represents the same of the data, this is where I defined all JPA entities (Tile, City, District..), the Repository class that include operations over the database (find, findAll, persist, remove..) and RulesLogger that writes and reads from into a txt file.

View represents the user interface. Here I defined the main window (MainFrame) and the tile map panel and tile sheet panel. For the tile map panel and tile sheet panel I used spring layout from SpringUtilities to force the layout into a compact grid of labels (JLabels), where each tile is a JLabel (MapTile and SheetTile extends JLabel).

The controller initializes the view components with data from the database and handles the view listeners (ButtonsPanelListener, MapTileListener, SheetTileListener).

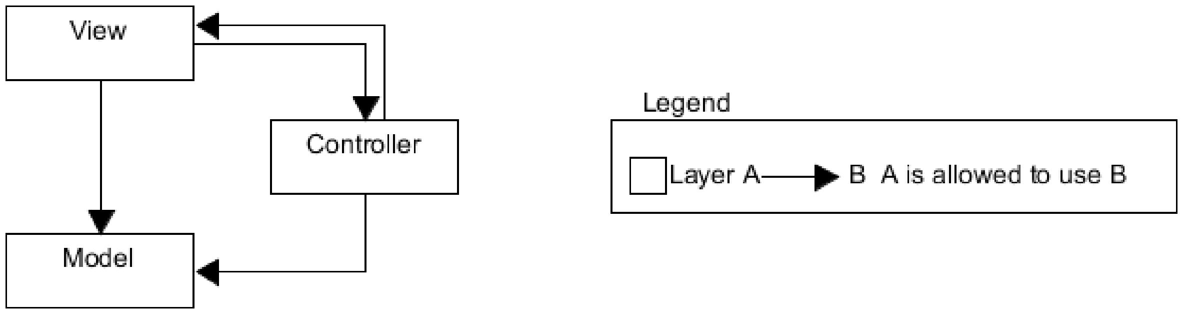


Fig.2.2 Project structure – Iteration 1

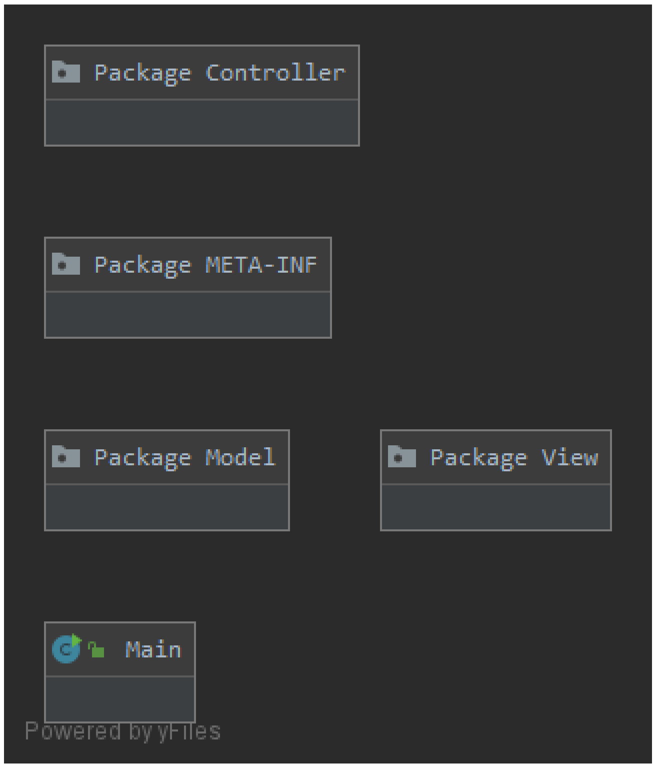


Fig.2.2 Project packages – Iteration 1

2. JPA

I used JPA (Java Persistence API) to annotate my classes and EntityManager to translate entity state transitions into SQL statements (findById, findAll, persist, merge, remove). I used wildcards so I can have only one repository and not one for each entity, the disadvantage is that everytime I use an operation I have to send the class of the entity I want to do the operation on to the repository.

3. Use cases

At the moment we can't do that much. When we start the application we can see the tile map panel and tile sheet panel. We can select what layer we view (all layers, or only one) with the JRadioButtons. We can write a new building rule into the text panel and click the Add rule button to add it to the rules txt file and we can see all rules into the logger console by clicking the View rules button.

If we hover over the map tile map we can see each tile's coordonates into the logger consoler. We can select a tile from the tile sheet panel by clicking on it and paint it into the tile map panel by double clicking on a tile.

4. Design alternatives

An alternative is that each place from the places layer could cover more than one tile. At the moment each place represents a single tile and this makes it easiser to represent the places into the UI, the disadvantage is that I need to add more conditions to the building operations.

5. Package diagrams

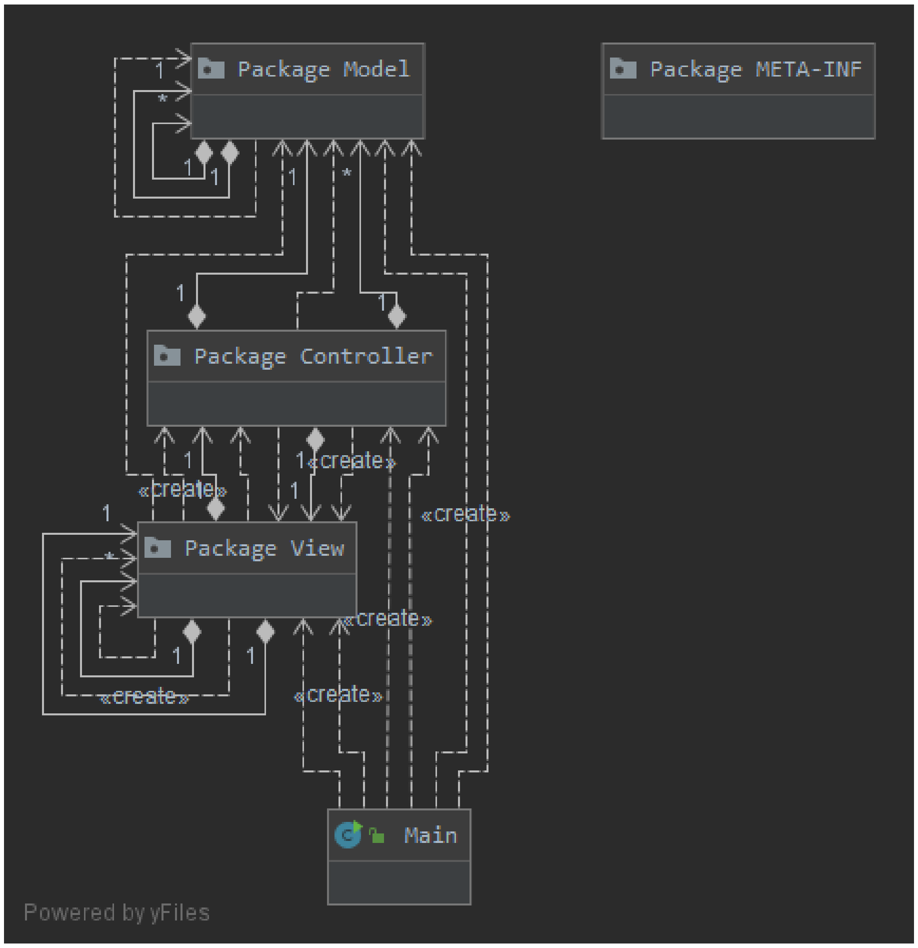


Fig.2.3 Project packages dependencies – Iteration 1

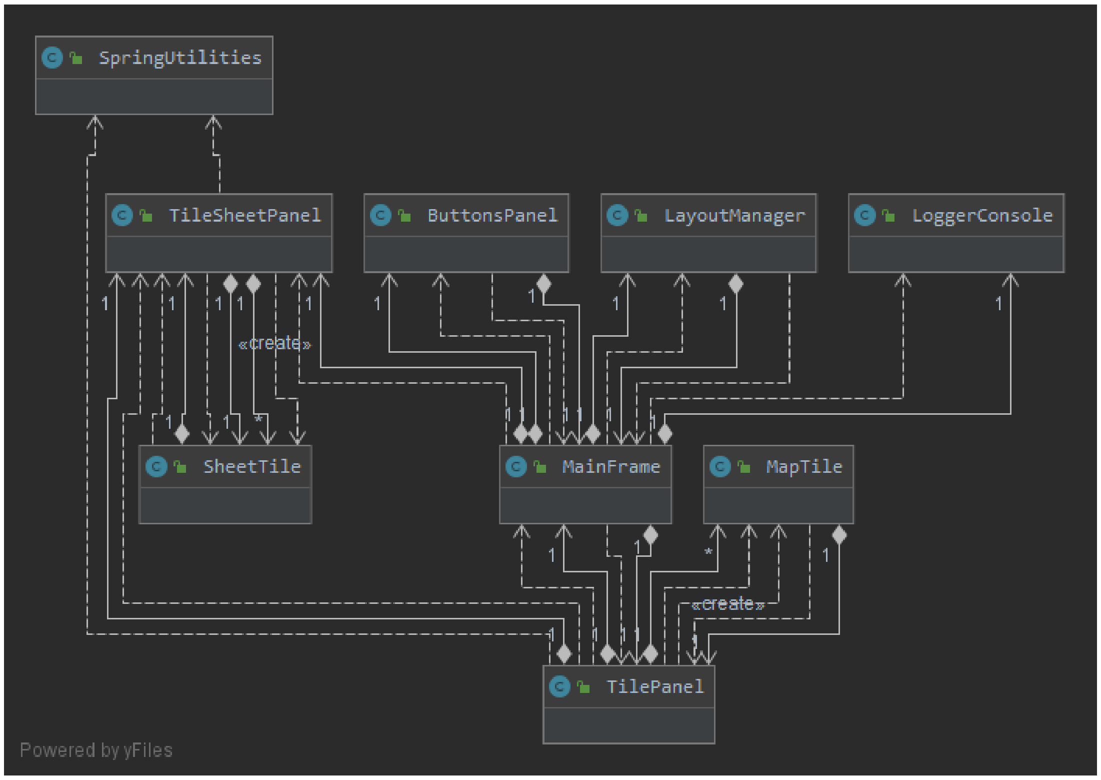


Fig.2.4 View package dependencies – Iteration 1

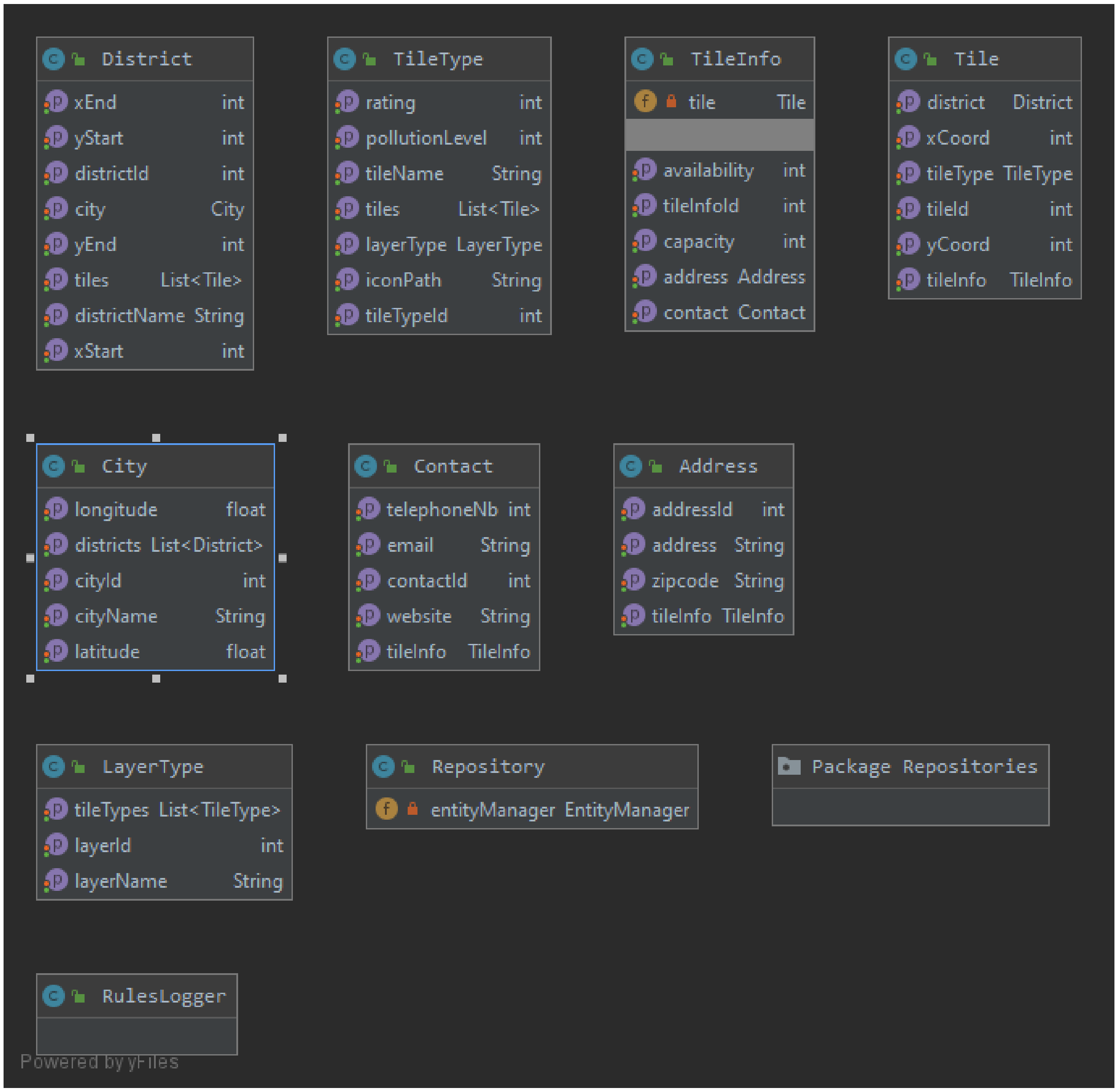


Fig.2.5 Model package fields and properties – Iteration 1

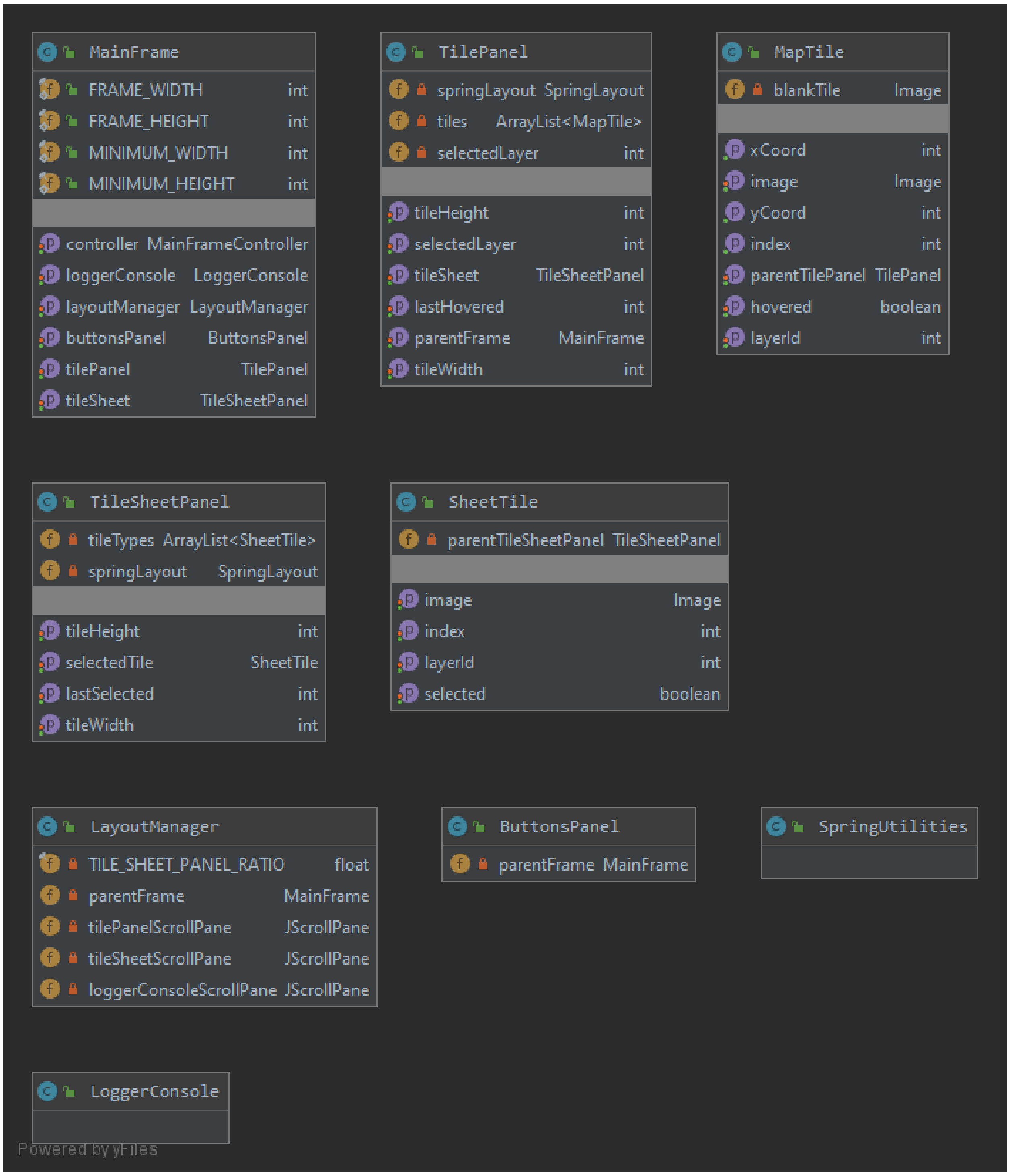


Fig.2.6 View package fields and properties – Iteration 1

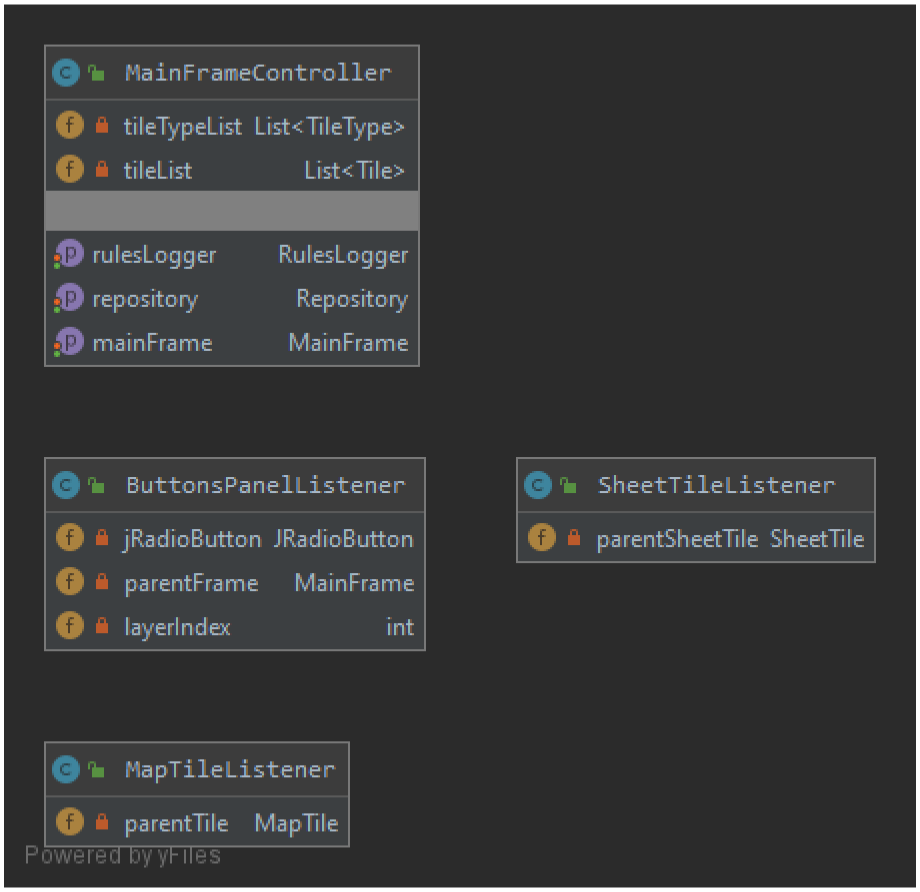


Fig.2.7 Controller package fields and properties – Iteration 1

6. Database diagram

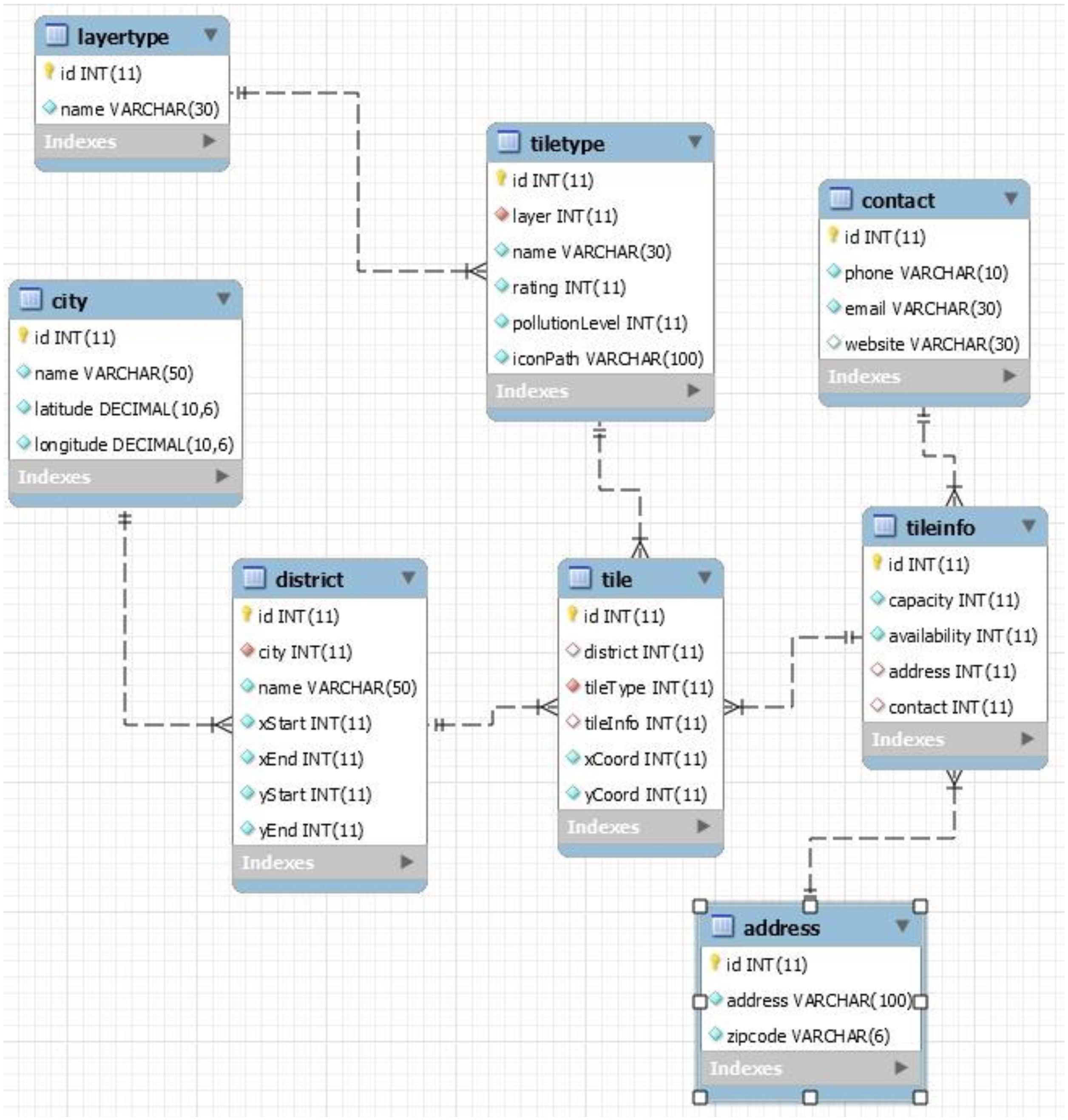
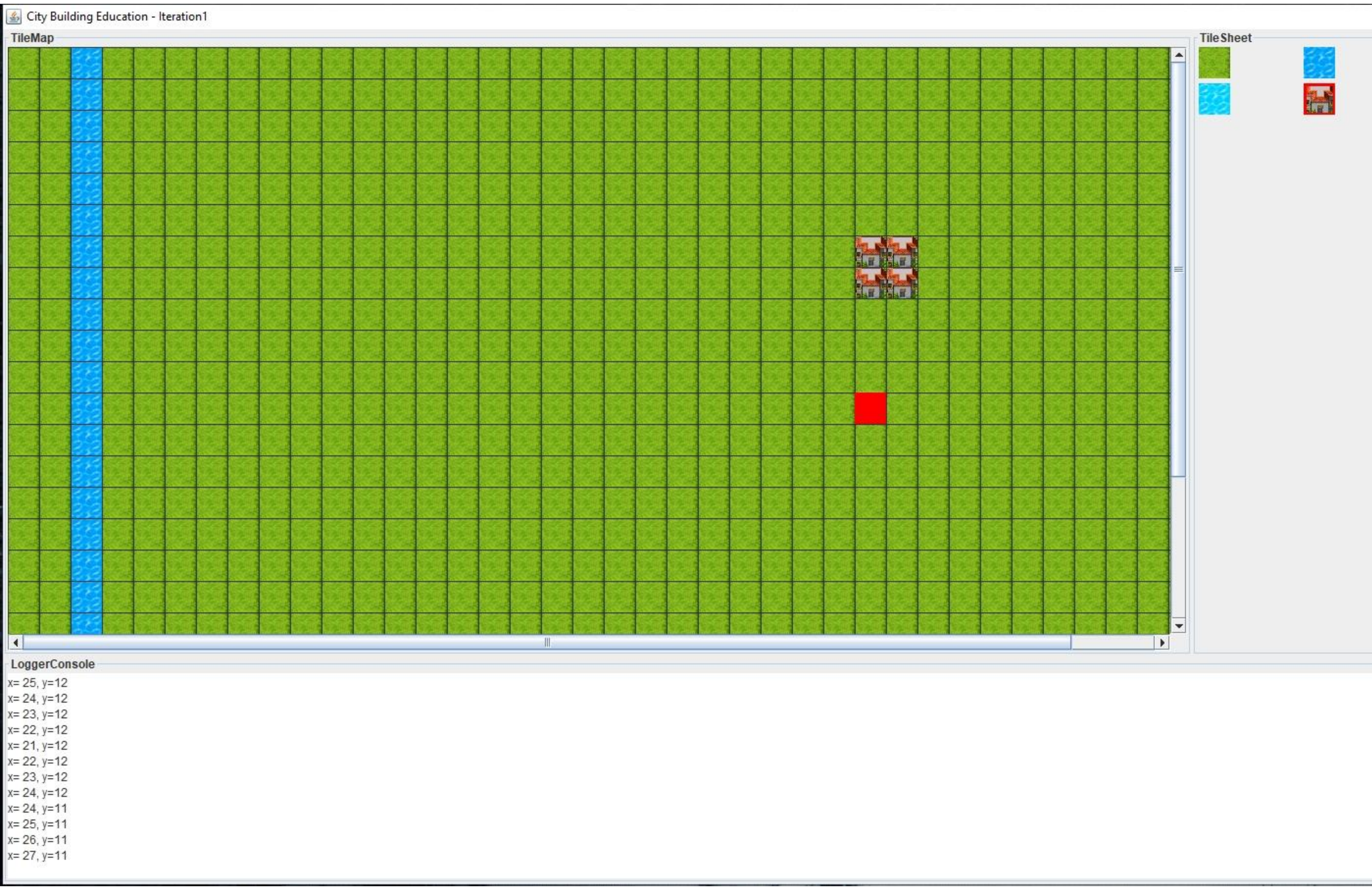


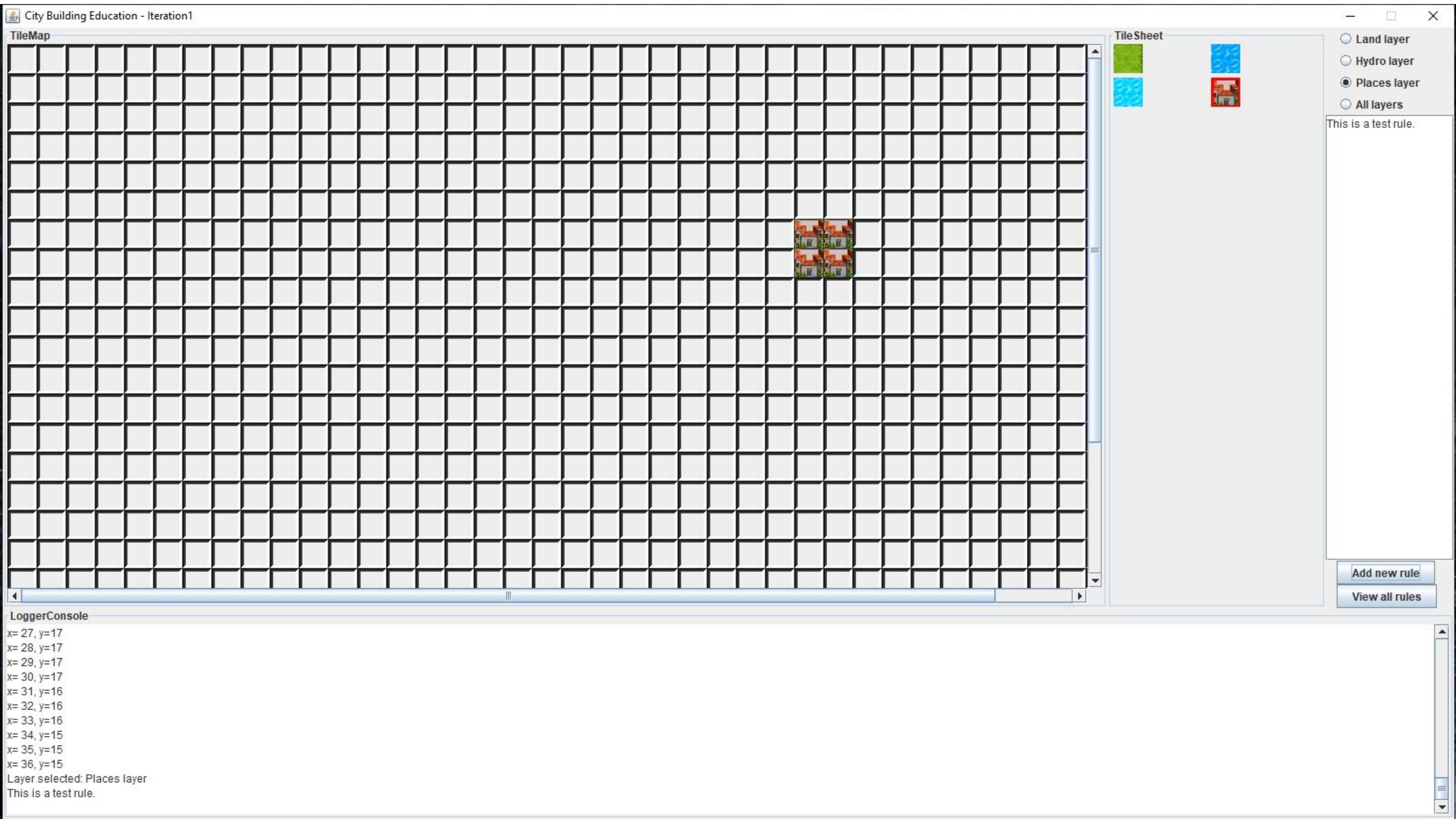
Fig.2.8 Database diagram – Iteration 1

7. UI screenshots



```
x= 25, y=12
x= 24, y=12
x= 23, y=12
x= 22, y=12
x= 21, y=12
x= 22, y=12
x= 23, y=12
x= 24, y=12
x= 24, y=11
x= 25, y=11
x= 26, y=11
x= 27, y=11
```

Fig.2.9 UI screenshot – Iteration1



This is a test rule.

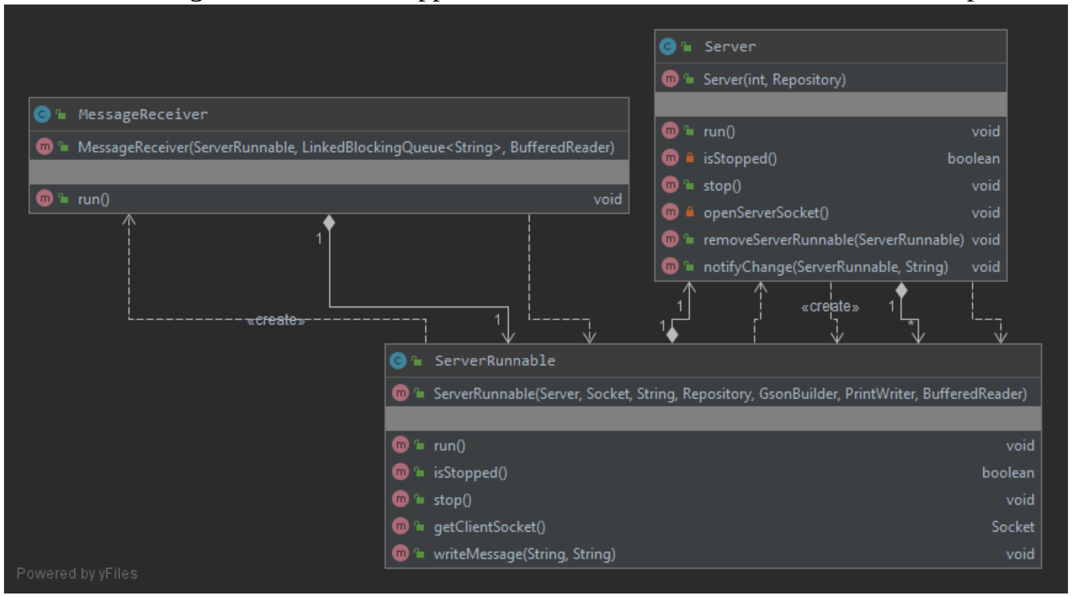
```
x= 27, y=17
x= 28, y=17
x= 29, y=17
x= 30, y=17
x= 31, y=16
x= 32, y=16
x= 33, y=16
x= 34, y=15
x= 35, y=15
x= 36, y=15
Layer selected: Places layer
This is a test rule.
```

Fig.2.10 UI screenshot – Iteration1

3. Iteration 2

1. Client-Server architecture

For this I separated the project into two projects: server and client. The server represents a thread pool. Each incoming connection is wrapped in a Runnable and handed off to a thread pool with a



Server
Server(int, Repository)
run() void
isStopped() boolean
stop() void
openServerSocket() void
removeServerRunnable(ServerRunnable) void
notifyChange(ServerRunnable, String) void
MessageReceiver
MessageReceiver(ServerRunnable, LinkedBlockingQueue<String>, BufferedReader)
run() void
«create»
1
ServerRunnable
ServerRunnable(Server, Socket, String, Repository, GsonBuilder, PrintWriter, BufferedReader)
run() void
isStopped() boolean
stop() void
getClientSocket() Socket
writeMessage(String, String) void
Powered by yFiles

Fig.3.1 Server Package – Iteration2

fixed number of threads. Each connection has a MessageReceiver that handles the incoming messages from the client. It takes the messages and puts them into a LinkedBlockingQueue of strings. And a ServerRunnable that handles the requests from the client and sends data from the database to client as json strings.

The client sends requests to the server for data to build the map for the first run. Everytime a tile is updated it sends it to the server to merge it into the database. The server notifies the other active clients that the map is updated.

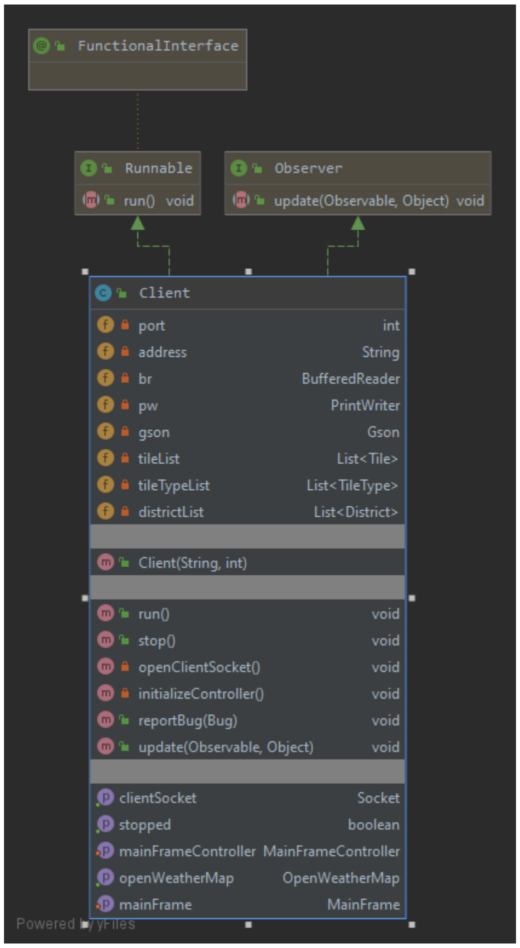
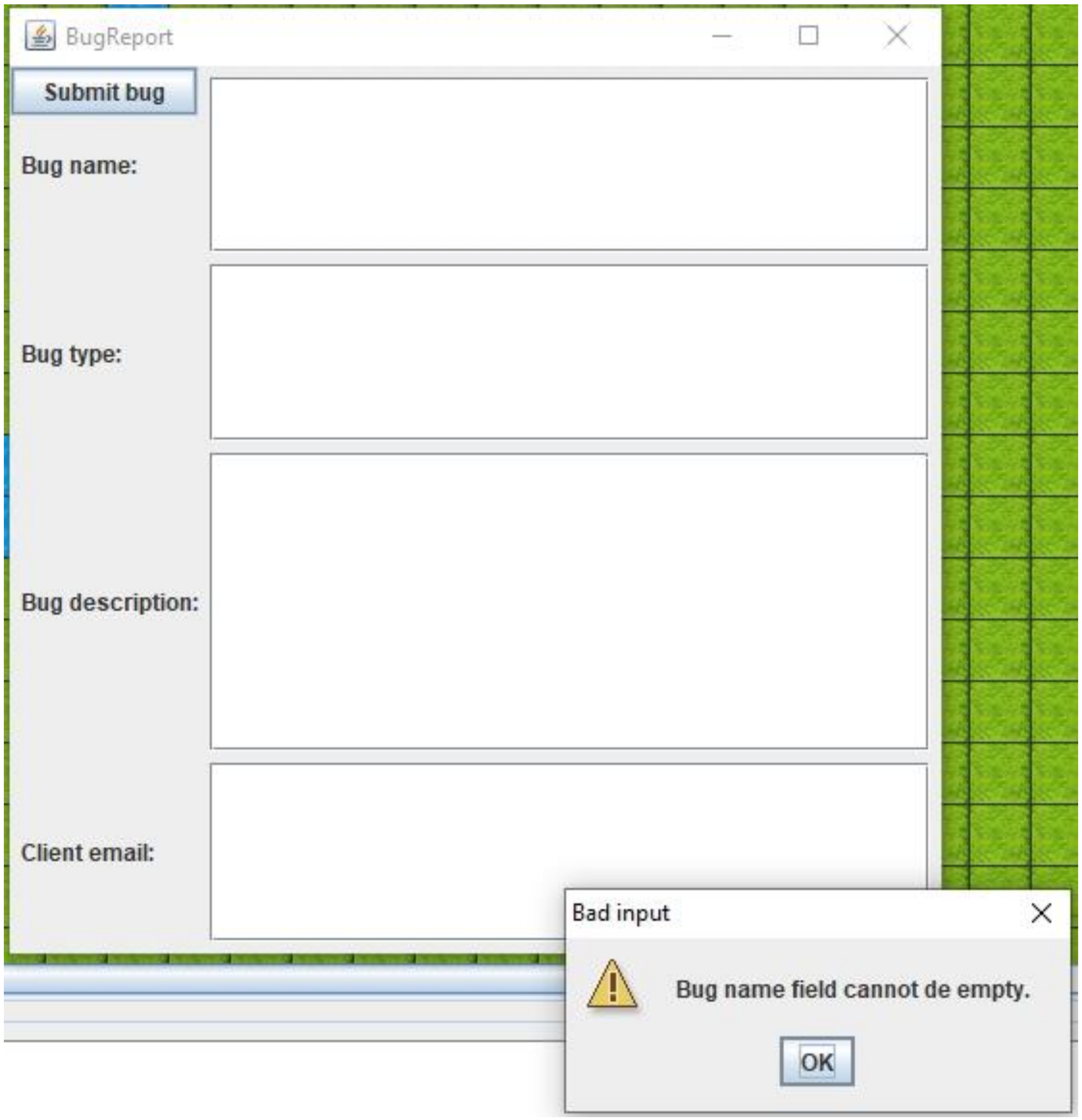


Fig.3.2 Client Package – Iteration2

2. Bug report

The bug report system takes client's input, saves it into a Bug object and then serializes it into a json. The client sends the json of the bug to the server where it is saved into a file for further processing.



Bug name field cannot de empty.

Fig.3.3 Bug report window – Iteration2

3. Use cases

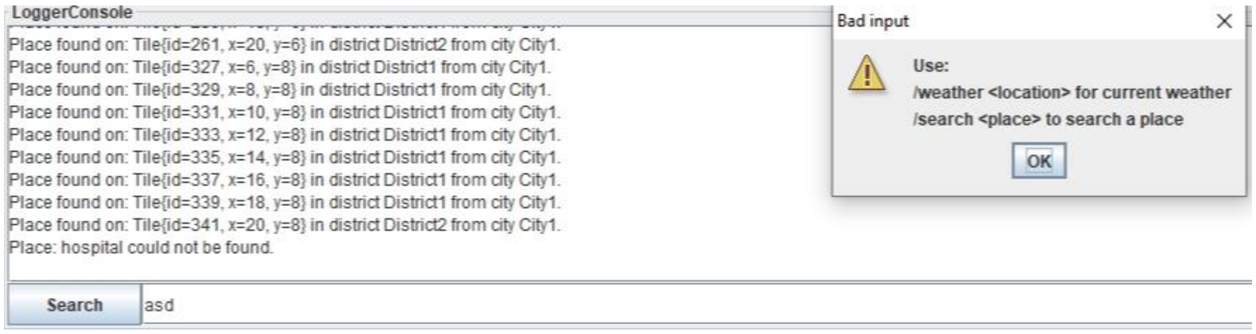
To view the info of a tile you have to right-click it and then the info will be autocompleted into the infoscrollpane where you can change/update/add and then submit it.

To use the openweathermap you have to use the input bar as following:

/weather <location>

For the search functionality :

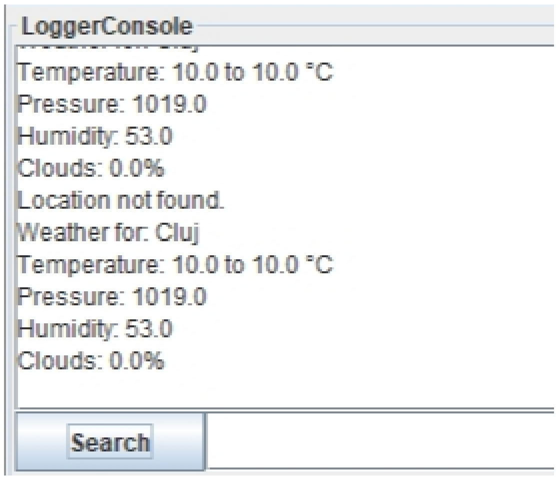
/search <place>



```
Place found on: Tile{id=261, x=20, y=6} in district District2 from city City1.
Place found on: Tile{id=327, x=6, y=8} in district District1 from city City1.
Place found on: Tile{id=329, x=8, y=8} in district District1 from city City1.
Place found on: Tile{id=331, x=10, y=8} in district District1 from city City1.
Place found on: Tile{id=333, x=12, y=8} in district District1 from city City1.
Place found on: Tile{id=335, x=14, y=8} in district District1 from city City1.
Place found on: Tile{id=337, x=16, y=8} in district District1 from city City1.
Place found on: Tile{id=339, x=18, y=8} in district District1 from city City1.
Place found on: Tile{id=341, x=20, y=8} in district District2 from city City1.
Place: hospital could not be found.
```

```
Use:
/weather <location> for current weather
/search <place> to search a place
```

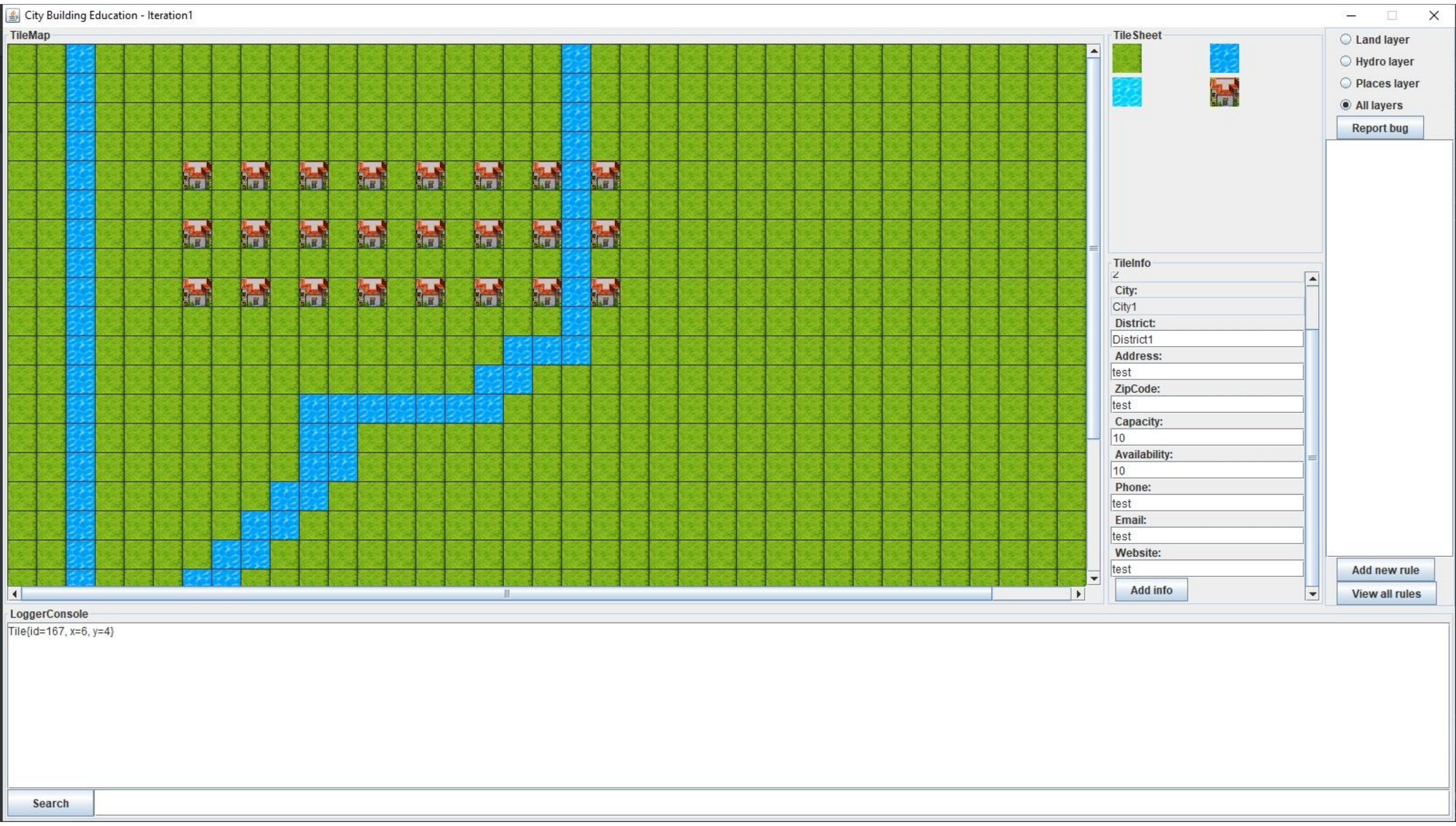
Fig.3.4 Search bar use case – Iteration2



```
Temperature: 10.0 to 10.0 °C
Pressure: 1019.0
Humidity: 53.0
Clouds: 0.0%
Location not found.
Weather for: Cluj
Temperature: 10.0 to 10.0 °C
Pressure: 1019.0
Humidity: 53.0
Clouds: 0.0%
```

Fig.3.5 Openweathermap use case – Iteration2

4. UI screenshots



[● All layers]

City:	City1
District:	District1
Address:	test
ZipCode:	test
Capacity:	10
Availability:	10
Phone:	test
Email:	test
Website:	test

```
Tile{id=167, x=6, y=4}
```

Fig.3.6 UI screenshot – Iteration2

4. Iteration 3

1. Multi-agent server

I split the server into 3 agents: RepositoryAgent, UpdateAgent, BugReportAgent. The main server accepts requests from the clients and sends them to the right agent to handle them. The RepositoryAgent takes data from the database and sends it to the main server wich sends it to the client. The UpdateAgent updates data into the database. The BugReportAgent handles bug reports, at the moment it only saves them for further processing.

The main server takes requests from the clients and based on what each request contains sends it to the right agent to handle it. It creates a ServerAgent thread that connects to the right agent server through a new socket.

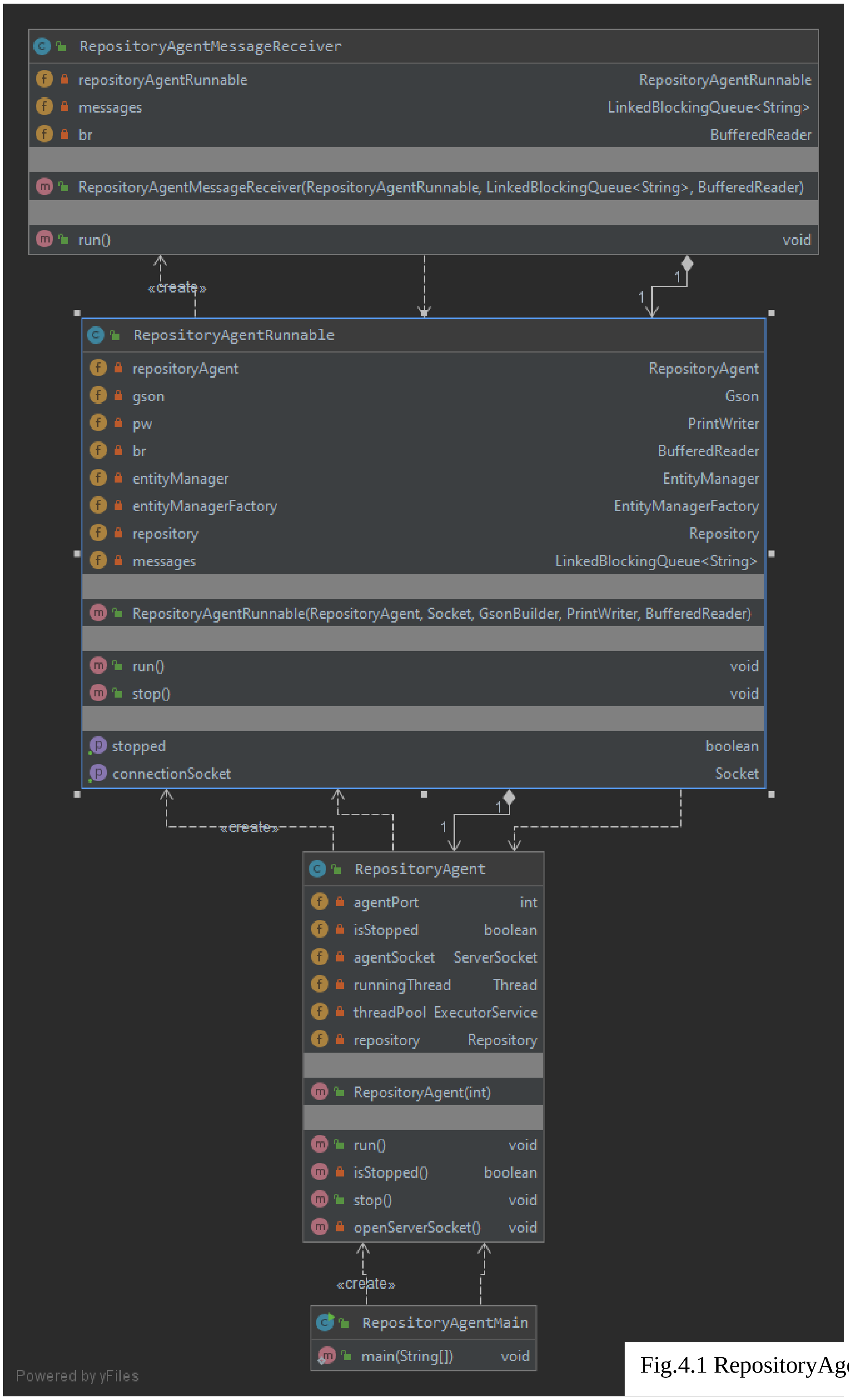

RepositoryAgentMessageReceiver
repositoryAgentRunnable RepositoryAgentRunnable
messages LinkedBlockingQueue<String>
br BufferedReader
RepositoryAgentMessageReceiver(RepositoryAgentRunnable, LinkedBlockingQueue<String>, BufferedReader)
run() void
«create»
1
1
RepositoryAgentRunnable
repositoryAgent RepositoryAgent
gson Gson
pw PrintWriter
br BufferedReader
entityManager EntityManager
entityManagerFactory EntityManagerFactory
repository Repository
messages LinkedBlockingQueue<String>
RepositoryAgentRunnable(RepositoryAgent, Socket, GsonBuilder, PrintWriter, BufferedReader)
run() void
stop() void
stopped boolean
connectionSocket Socket
«create»
1
1
RepositoryAgent
agentPort int
isStopped boolean
agentSocket ServerSocket
runningThread Thread
threadPool ExecutorService
repository Repository
RepositoryAgent(int)
run() void
isStopped() boolean
stop() void
openServerSocket() void
«create»
RepositoryAgentMain
main(String[]) void
Powered by yFiles

Fig.4.1 RepositoryAgent package

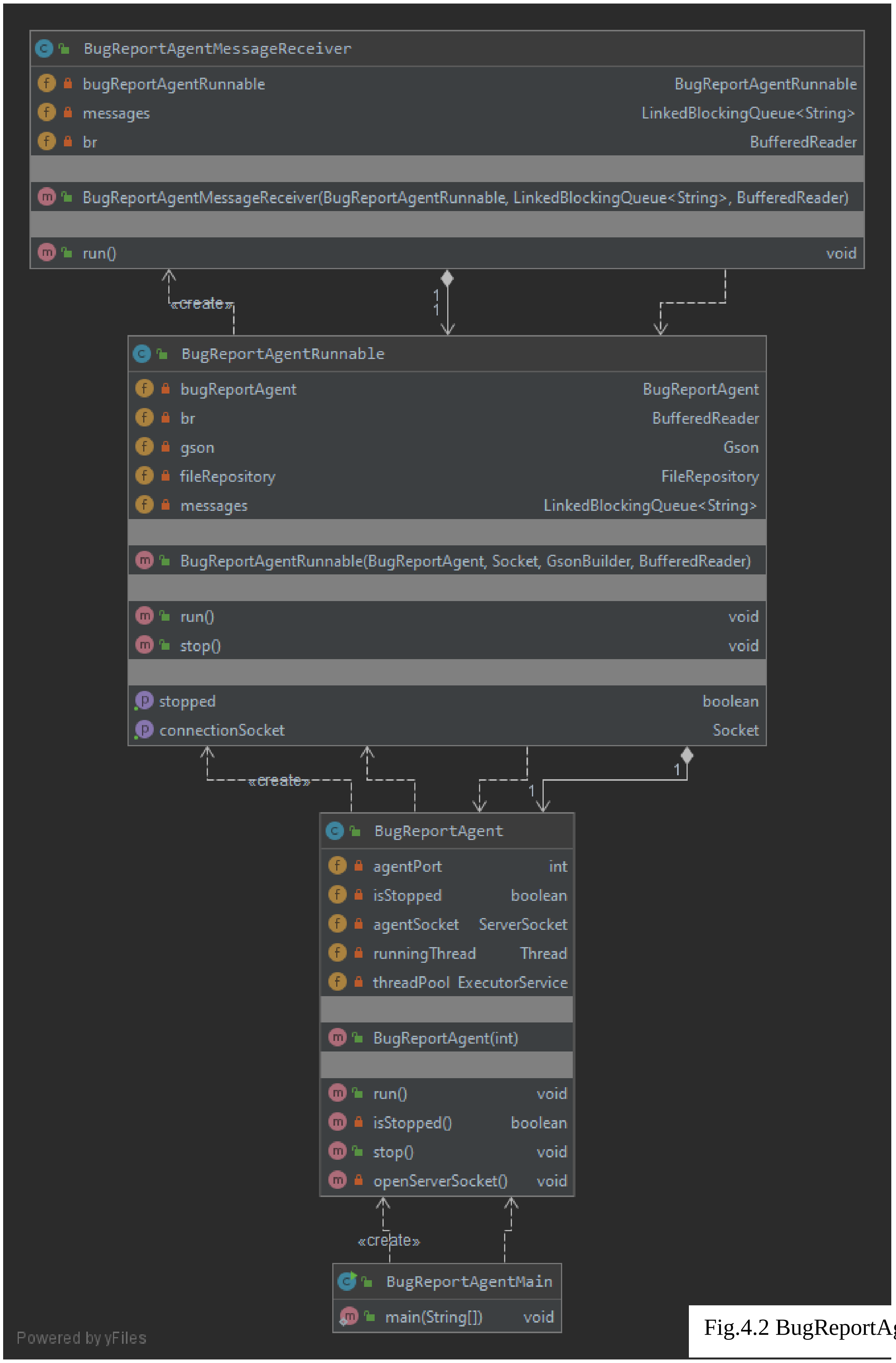

BugReportAgentMessageReceiver
bugReportAgentRunnable BugReportAgentRunnable
messages LinkedBlockingQueue<String>
br BufferedReader
BugReportAgentMessageReceiver(BugReportAgentRunnable, LinkedBlockingQueue<String>, BufferedReader)
run() void
«create»
1
1
BugReportAgentRunnable
bugReportAgent BugReportAgent
br BufferedReader
gson Gson
fileRepository FileRepository
messages LinkedBlockingQueue<String>
BugReportAgentRunnable(BugReportAgent, Socket, GsonBuilder, BufferedReader)
run() void
stop() void
stopped boolean
connectionSocket Socket
«create»
1
1
BugReportAgent
agentPort int
isStopped boolean
agentSocket ServerSocket
runningThread Thread
threadPool ExecutorService
BugReportAgent(int)
run() void
isStopped() boolean
stop() void
openServerSocket() void
«create»
BugReportAgentMain
main(String[]) void
Powered by yFiles

Fig.4.2 BugReportAgent package

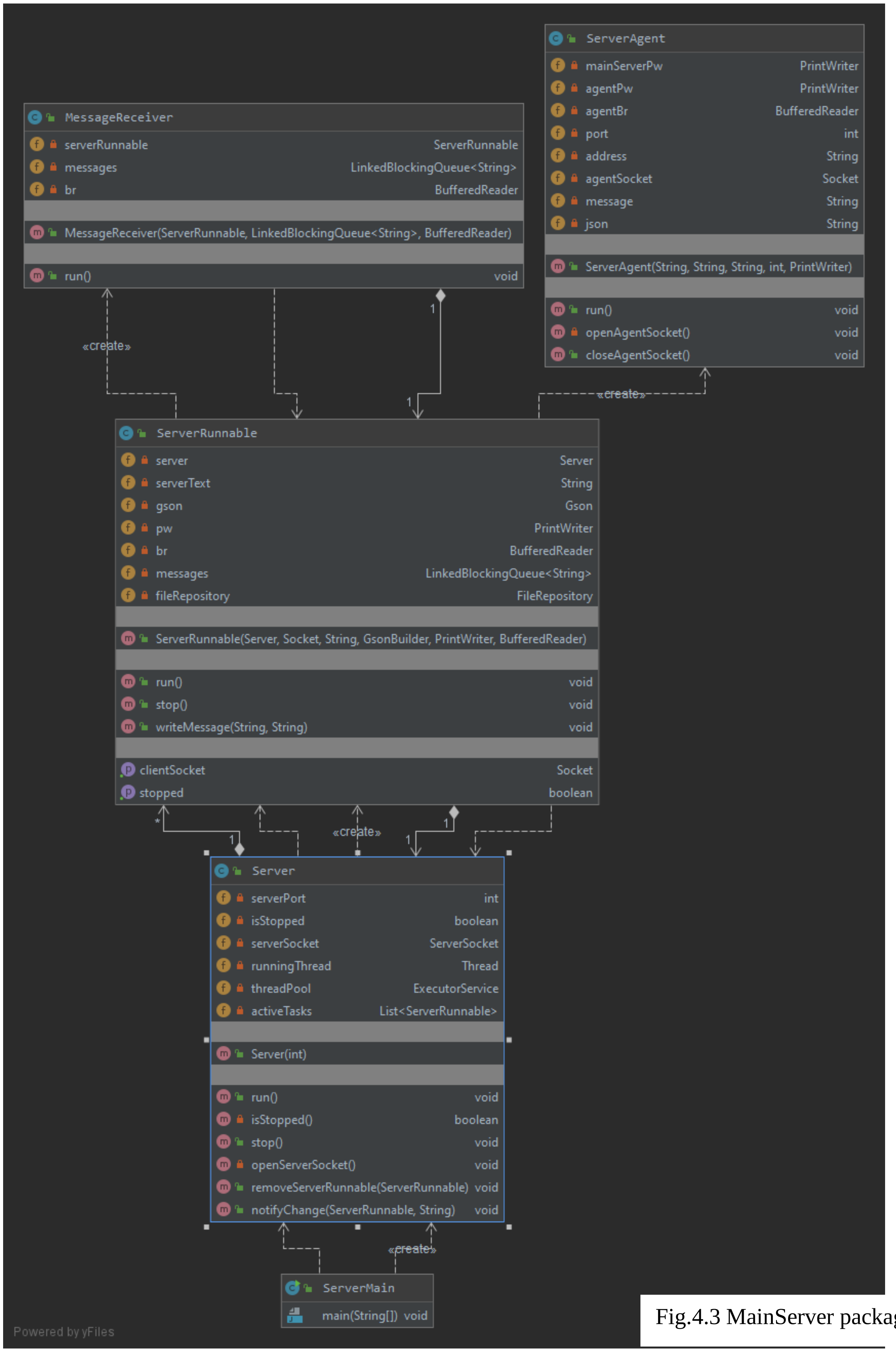


Fig.4.3 MainServer package

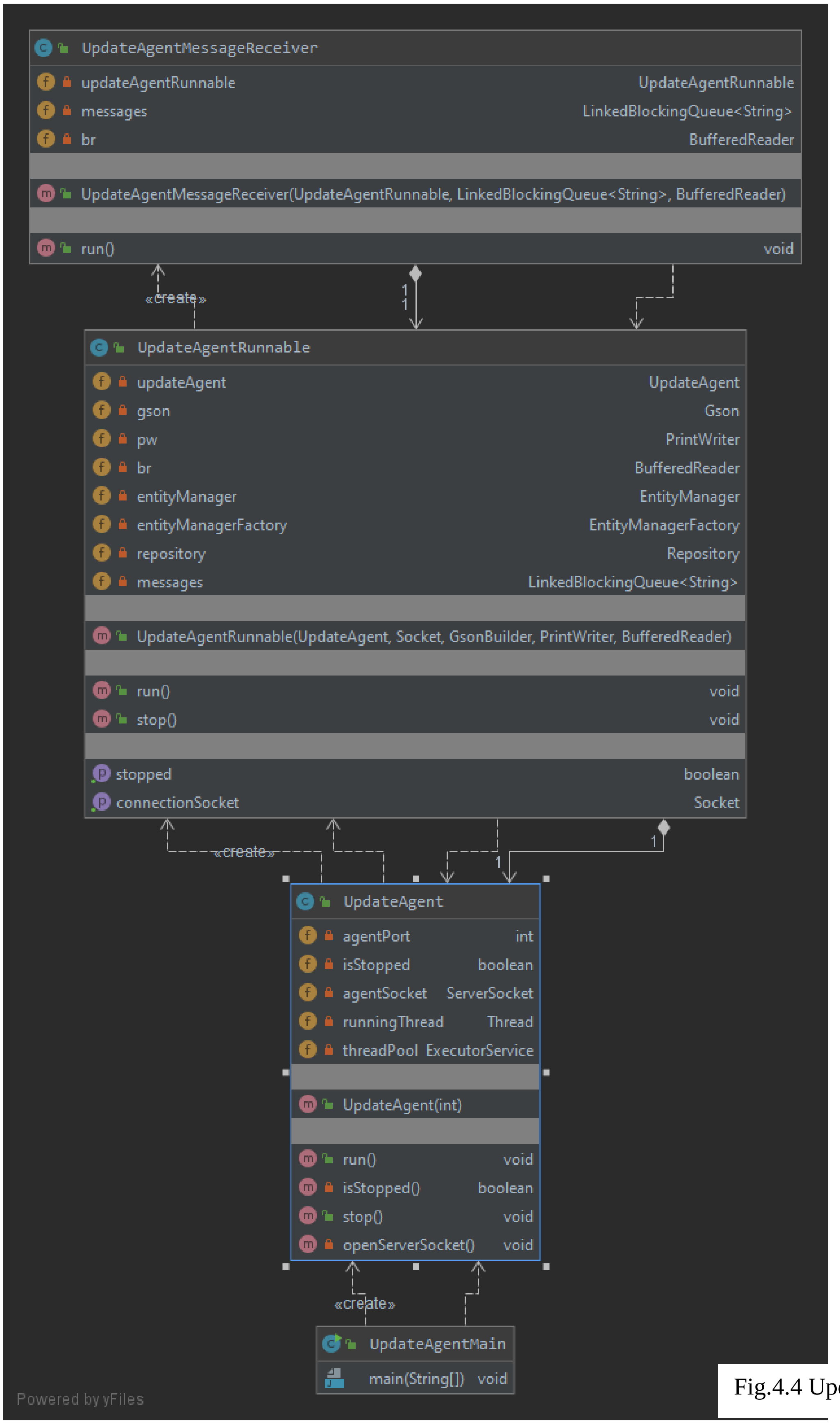


Fig.4.4 UpdateAgent package