Polytechnic University of Timisoara - Faculty of Automation and Computers
COURSE: CEBP (Concurrent and Event Based Programming)

PIXEL WARS

The PIXEL world awaits you: Grind, build, conquer!

Developed by Team CASE

DOCUMENTATION

Timișoara
2019

Contents

1. Software requirements

1.1. Language: Java

- Highlight and motivate the thread-safety/event-based concepts implemented in the application;

1.2. Program: Strategy game

- An application that simulates and implements a strategy game;
- Players are serviced by threads, and the game takes place in a virtual common space. This space provides various resources of different types that can be used to build up different objectives/buildings;
- Any building must be built using different resources, and the number of resources required can vary depending on the game's rules; Moreover, some buildings might require some other specific buildings to be owned in order to be built;
- The available resources are limited, but some objectives/buildings can produce other resources;
- Players aim to build a minimum number of objectives. The player who first meets the game-specific goal wins;
- Both the players and the resources/objectives may generate asynchronous events that bring modifications to the game's entities state;
- Everything has to be thread-safe;

Observation: The actual game specifications will be described by the students and any changes in the requirements/specification are subject to teacher's approval!

Additional requirements/specification:

- In the beginning, the user will be able to configure the virtual common space and the player's details via the GUI;
- During the gameplay, the user is able to monitor the activities/events that happen in the game via the GUI: read event messages, check every player's holdings on the fly, see details about relevant entities in the virtual common space;

2. Requirements analysis

2.1. General scenario (in detail)

The name of this prototype of a multithreaded strategy-game application is PIXEL Wars. Generally, strategy games like this do offer some configuration settings in the beginning. In our case, that configuration might be changed by the **user** that uses the app via completing the form in the intro scene of the GUI.

Thereby, when the game is launched, the **user** might:

- select the number of players (2..8);
- fill the name and select a color (red, orange, yellow, brown, green, cyan, blue or purple) for everyone;
- select the size of the map (tiny, small, medium, large or giant);
- select the density of the resources (starvation, moderate or richness);

After choosing the preferred configuration, the **user** should click the "PLAY" button in order to send the configurations and load the game based on it. In addition, there's a "QUIT" button too in the intro scene, that the user might click in order to close the program.

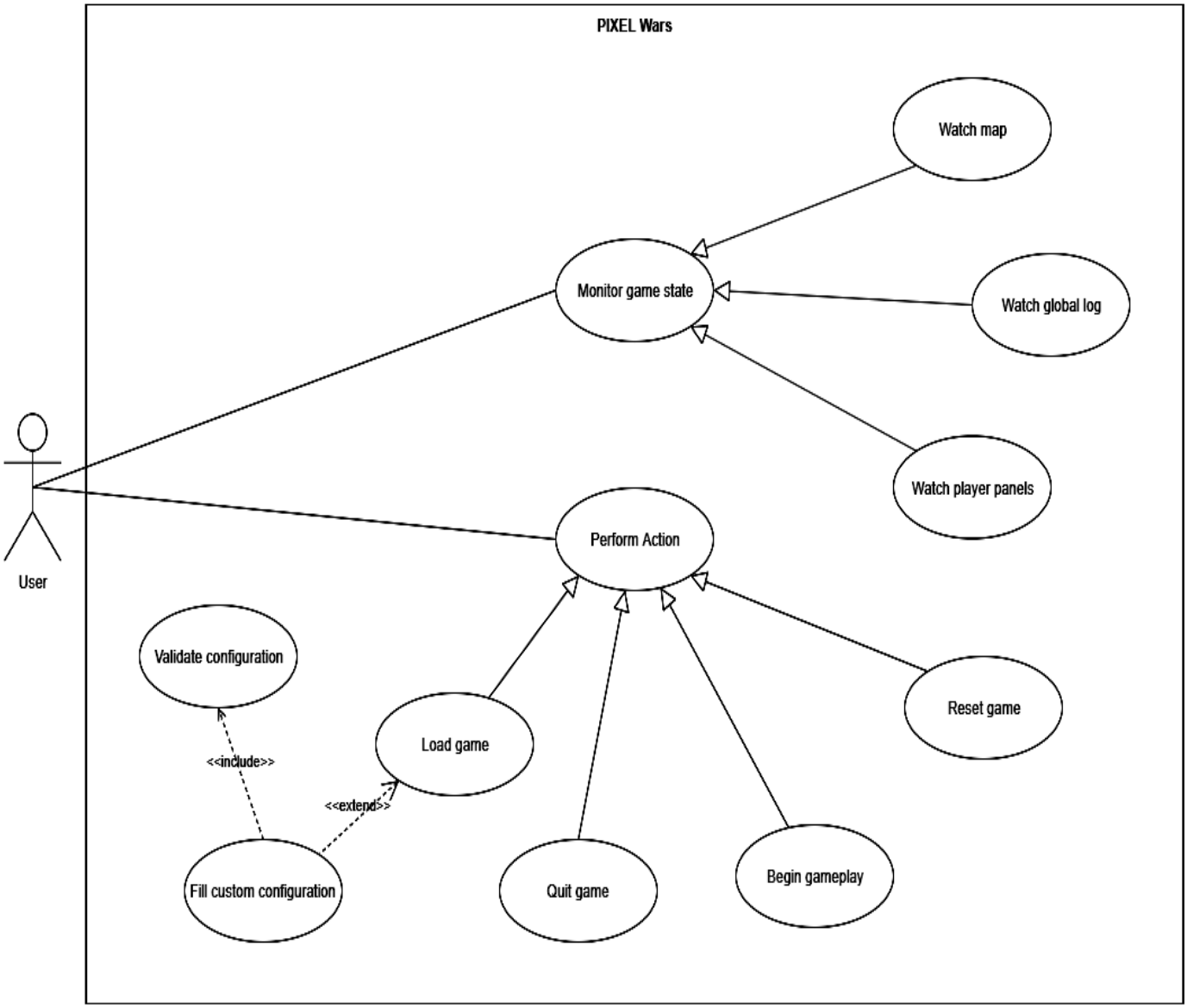
Shortly after clicking "PLAY", the game is going to be loaded causing the GUI to transit to the in-game scene that will contain: a representation of the map, a panel where to check details about the players and a global messaging log where events that occur in the game are written. The map will have already randomly spawned the players inside it, along with the desired amount of resources (both based on the configuration earlier). Next, the game will be in a waiting state, until the **user** decides to click the "BEGIN" button in order for the effective gameplay to start, or to click the "RESET" button in order for the user to be taken back to the intro/configuration scene and perform the previous steps again (maybe for another game based on some new configuration or just to be able to quit).

The **user**'s main purpose is to monitor/see at any time details about the game state/events/activities that occur (being updated in real-time by the GUI agent):

- information about every player in the game is available from a panel, that is located somewhere in the in
- game scene. Details shown there are: its name, icon, current position on the map and the player's resource bar/building bar where there are details about the owned amounts of resources/buildings (and also the costs for the buildings);
- the **user** is able to check at any time information about any entity that currently inhabits the map (position,
- color, concrete name etc.), just by hovering the mouse over their icon;
- any events that occur in the game/on the map are readable by **the user** in the Global Event Log, located somewhere in the in-game scene;

When the user clicks the "BEGIN" button, the gameplay starts. The user continues to check and see the details mentioned earlier and is also able to hit at any time the "RESET" button, if the game is wanted to be reset (even if the game is in progress). During the gameplay, the automated players build buildings and search for the needed resources to exploit (the player has to move to some neighbor coordinates of the resource in cause in order to perform such actions). Some of the buildings might also produce resources inside their owner's (a player) resource bar, once upon a cooldown time. When a player completes/achieves the game winning goal (when they build the needed-to-win set of objective buildings), they stop their actions and the game is notified. If, by any circumstance (like resources starvation), the player can't win at all, he is declared loser and forfeits the game.

2.2. UML Use Case diagram



2.3. OOA (Object Oriented Analysis)

We identify the next classes from the requirements:

- Game (the logic controller);
- Map (the virtual common space);
- Player, Resource, Building (all 3 extensions of some MapEntity abstract class as they have a place and perform actions inside the map). Both Player and Building should be serviced by their own threads.
- BuildRequirements (a wrapper class for the requirements for each building);
- EventBroadcaster class and EventCapturer interface (will use observer pattern for the game event handling);
- The GUI (Graphical User Interface) is built using the JavaFX library, and consists of the GameEngine (the main controller class) and the GameUI (the gui builder) classes, which have methods that create/update/process the GUI and create the Game object.
- Any other classes that are considered to be needed in the design are going to be described later.

3. Design and implementation

3.1. Design Patterns

3.1.1. Abstract factory

We have an abstract factory interface that is concreted by the ResourceBank factory and the Building factory. This allows simple creation of ResourceBank and Building objects by overriding the create method inside the abstract factory interface, method that needs a String parameter referring the entity's concrete name. The building factory also confers a similar method for getting the BuildRequirements of a concrete type of building.

3.1.2. Acyclic Visitor

We are using acyclic visitor in order to implement the things mentioned at 3.1. Building PS section;

Different types of building do produce different types of resources in different quantities;

Different types of buildings have different production cooldowns;

The ProductionHandler interface is extended by Building Production Handler interfaces with the name in format: <BuildingName>PH, which are implemented by the concrete Resource Production Handler classes, which are one for every type of exploitable ResourceBank on the map;

Every building has its own list of production handlers on whom it may call the requestProduction() method, this method behaving in a specific way for that specific building; Every <BuildingName>PH interface defines the requestProduction() method that accepts a parameter of the type of that specific building;

The concrete Resource Production Handler classes do implement those <BuildingName>PH interfaces, that might produce resources of that concrete type. For example, if Gold might be produced only by Mine and Parliament, and Food might be produced by House, Granary and TownHall, then the GoodPH implements the MinePH and ParliamentPH interface, while the FoodPH should implement the HousePH, GranaryPH and TownHallPH interfaces, both GoldPH and FoodPH needing to override the requestProduction() method from each of the interfaces they implement, in a specific way. The interfaces <BuildingName>PH also have a default method for getCooldown(), which return the cooldown time for building's production in terms of milliseconds.

3.1.3. Observer

This pattern is used for the event-handling part of the system that is described at section 3.4;

The classes that are "observable" contain a reference to an EventBroadcaster object inside them;

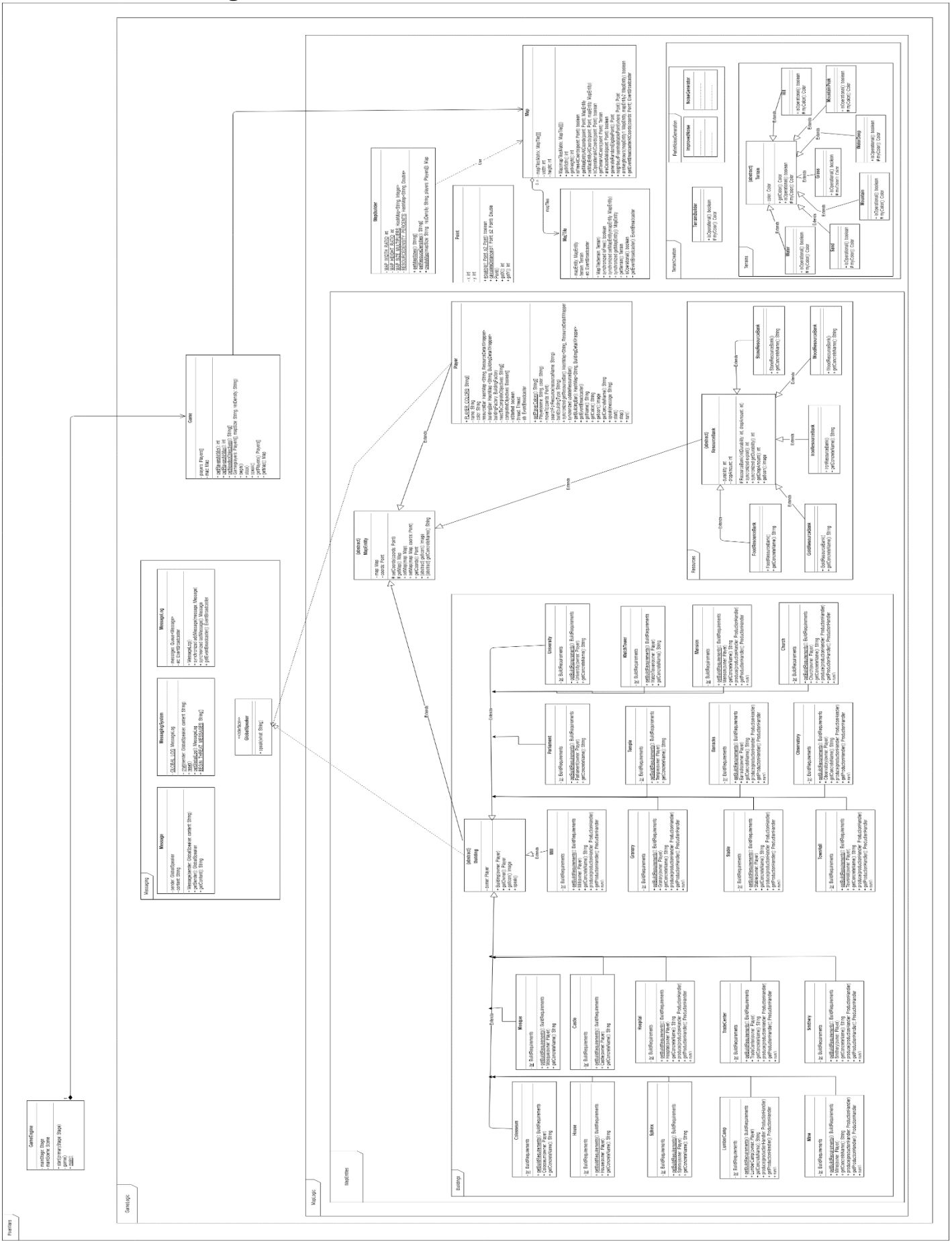
EventBroadcaster objects contain a collection of observers (here nicknamed EventCapturers), initially empty, a set of methods for manipulating this collection (add/remove) and the method notifyEventCapturers() described below;

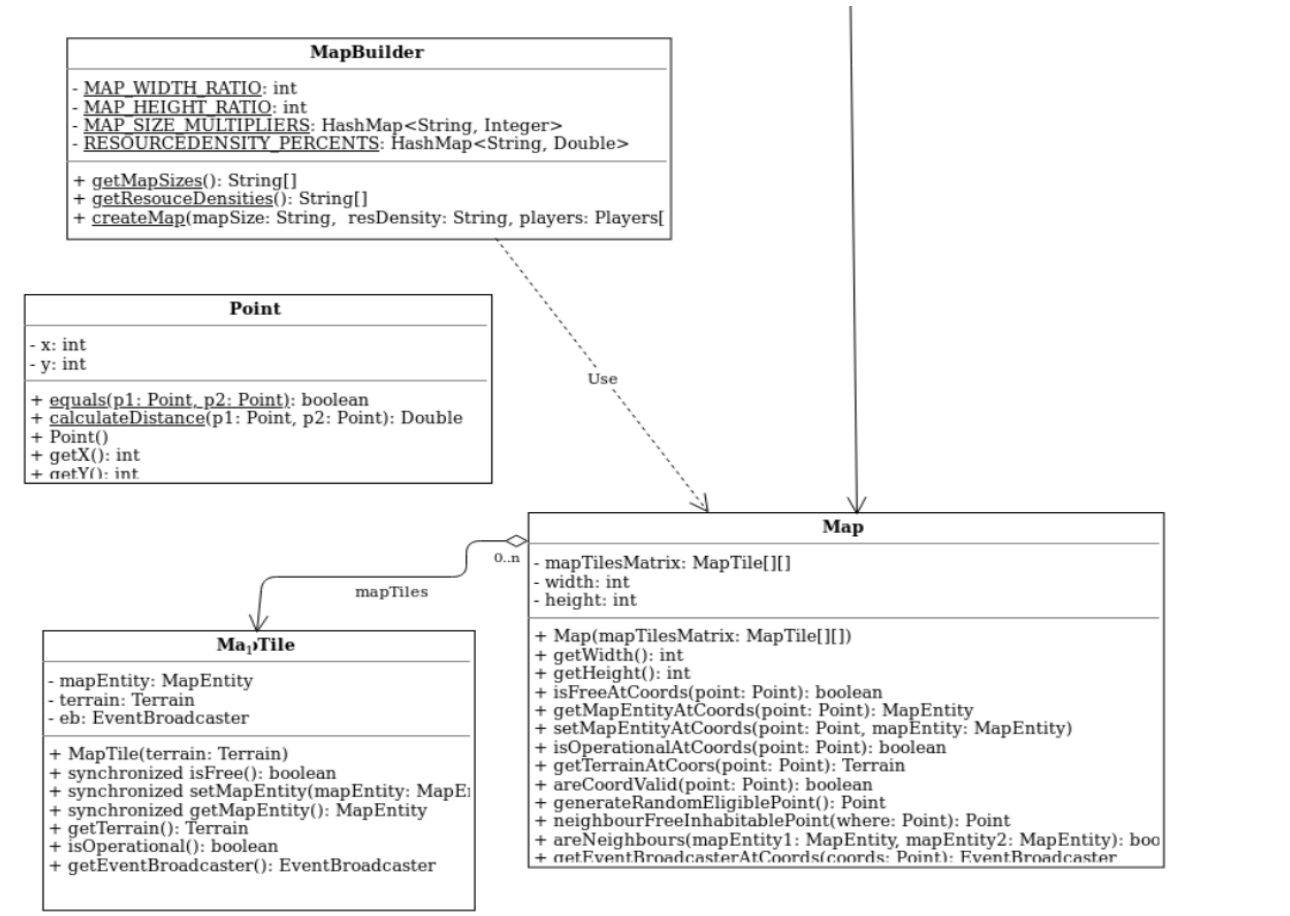
EventBroadcasters also contain a cause object, which will always be set in the constructor as the observable object itself (for example, if the Player class does broadcast events it has a field EventBroadcaster eb = new EventBroadcaster(this)). This will be used later in order to decide how should the event be handled by one specific EventCapturer's update method;

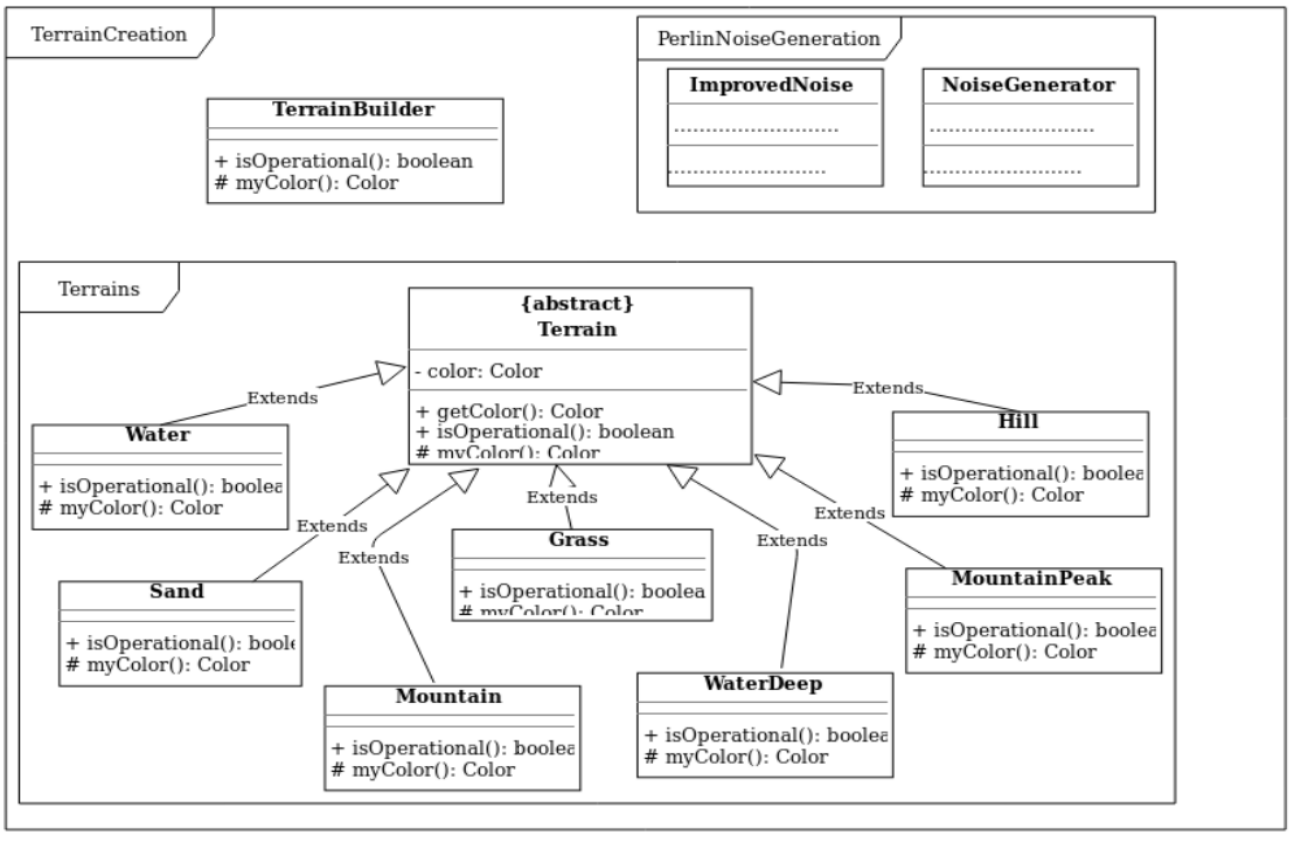
When the observable classes perform an action that is considered an event, they call the notifyEventCapturers() method on the EventBroadcaster object. This will notify all their EventCapturers inside the collection of the observable, by calling the update(observable=cause) method on them;

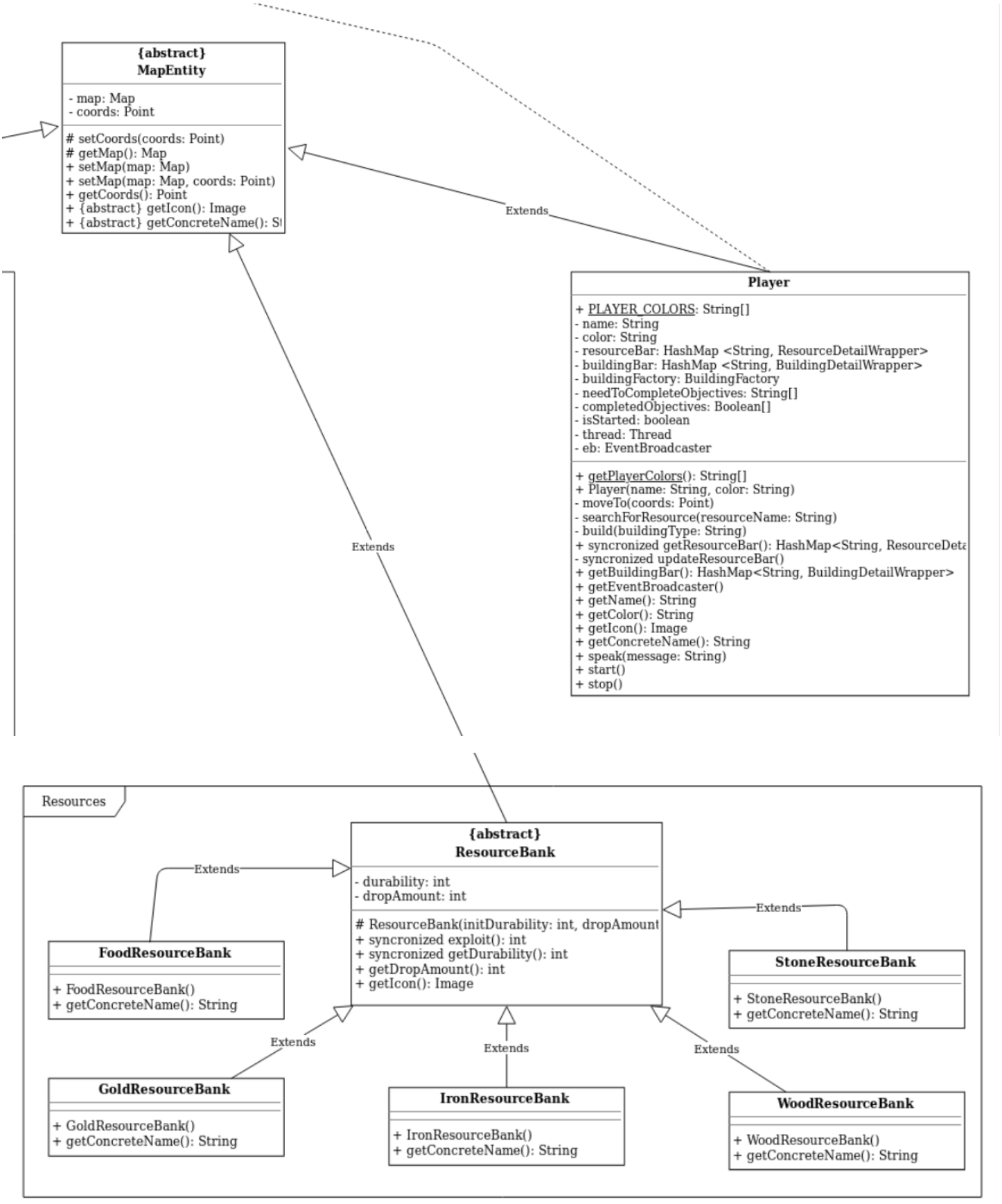
Afterwards, the EventCapturers/Observers are going to handle the event depending on the concrete type of the observable received as parameter.

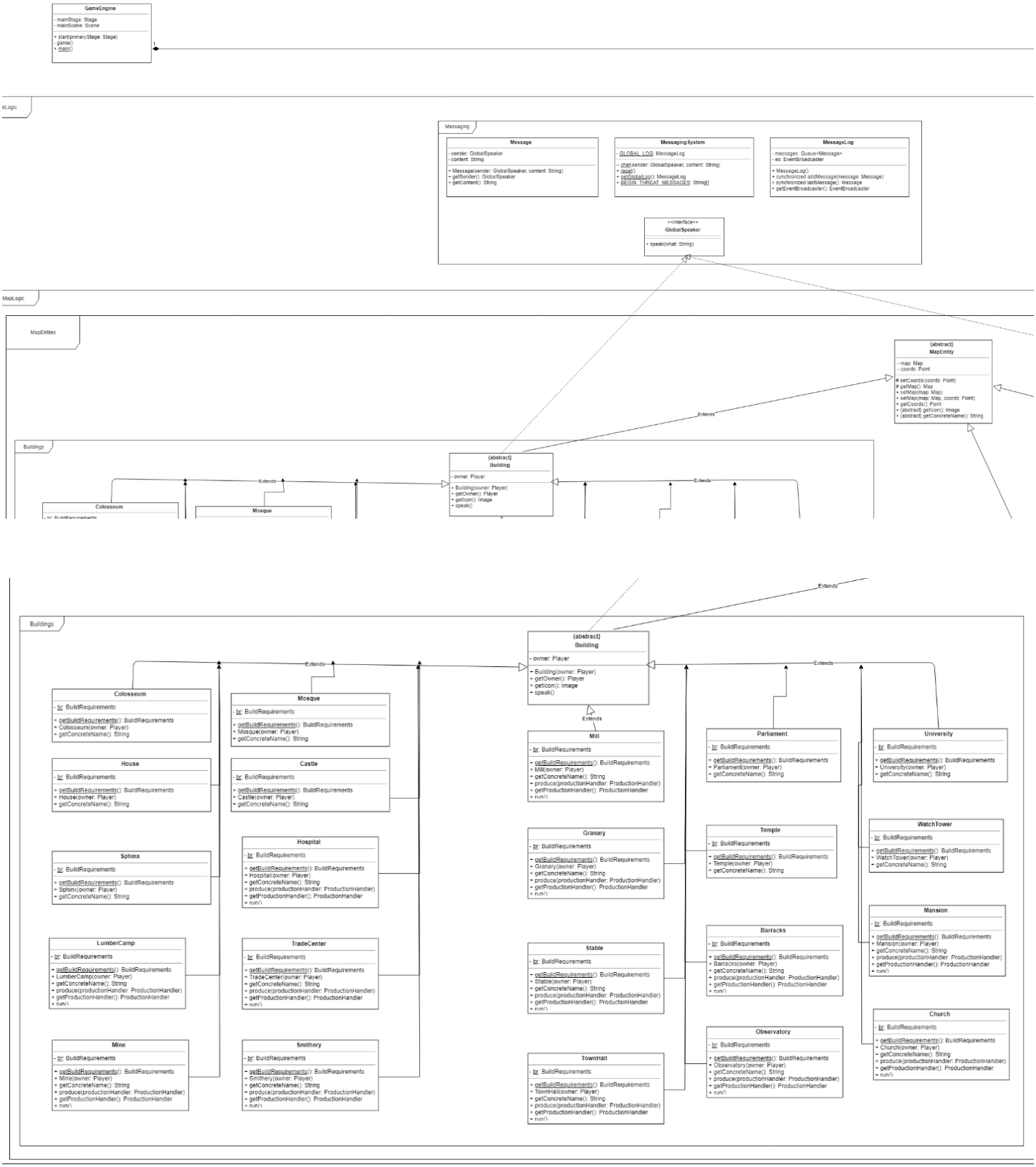
3.2. UML class diagrams




MapBuilder
- MAP_WIDTH_RATIO: int
- MAP_HEIGHT_RATIO: int
- MAP_SIZE_MULTIPLIERS: HashMap<String, Integer>
- RESOURCEDENSITY_PERCENTS: HashMap<String, Double>
+ getMapSizes(): String[]
+ getResouceDensities(): String[]
+ createMap(mapSize: String, resDensity: String, players: Players[
Point
- x: int
- y: int
+ equals(p1: Point, p2: Point): boolean
+ calculateDistance(p1: Point, p2: Point): Double
+ Point()
+ getX(): int
+ getY(): int
Use
Map
- mapTilesMatrix: MapTile[][]
- width: int
- height: int
+ Map(mapTilesMatrix: MapTile[][])
+ getWidth(): int
+ getHeight(): int
+ isFreeAtCoords(point: Point): boolean
+ getMapEntityAtCoords(point: Point): MapEntity
+ setMapEntityAtCoords(point: Point, mapEntity: MapEntity)
+ isOperationalAtCoords(point: Point): boolean
+ getTerrainAtCoors(point: Point): Terrain
+ areCoordValid(point: Point): boolean
+ generateRandomEligiblePoint(): Point
+ neighbourFreeInhabitablePoint(where: Point): Point
+ areNeighbours(mapEntity1: MapEntity, mapEntity2: MapEntity): boo
+ getEventBroadcasterAtCoords(coords: Point): EventBroadcaster
0..n
mapTiles
Ma1pTile
- mapEntity: MapEntity
- terrain: Terrain
- eb: EventBroadcaster
+ MapTile(terrain: Terrain)
+ synchronized isFree(): boolean
+ synchronized setMapEntity(mapEntity: MapE
+ synchronized getMapEntity(): MapEntity
+ getTerrain(): Terrain
+ isOperational(): boolean
+ getEventBroadcaster(): EventBroadcaster


TerrainCreation
PerlinNoiseGeneration
ImprovedNoise
NoiseGenerator
TerrainBuilder
+ isOperational(): boolean
# myColor(): Color
Terrains
{abstract}
Terrain
- color: Color
+ getColor(): Color
+ isOperational(): boolean
# myColor(): Color
Extends
Water
+ isOperational(): boolea
# myColor(): Color
Hill
+ isOperational(): boolea
# myColor(): Color
Grass
+ isOperational(): boolea
Sand
+ isOperational(): boole
# myColor(): Color
Mountain
+ isOperational(): boolea
# myColor(): Color
WaterDeep
+ isOperational(): boolea
# myColor(): Color
MountainPeak
+ isOperational(): boolea
# myColor(): Color


{abstract}
MapEntity
- map: Map
- coords: Point
setCoords(coords: Point)
getMap(): Map
+ setMap(map: Map)
+ setMap(map: Map, coords: Point)
+ getCoords(): Point
+ {abstract} getIcon(): Image
+ {abstract} getConcreteName(): St
Extends
Player
+ PLAYER_COLORS: String[]
- name: String
- color: String
- resourceBar: HashMap <String, ResourceDetailWrapper>
- buildingBar: HashMap <String, BuildingDetailWrapper>
- buildingFactory: BuildingFactory
- needToCompleteObjectives: String[]
- completedObjectives: Boolean[]
- isStarted: boolean
- thread: Thread
- eb: EventBroadcaster
+ getPlayerColors(): String[]
+ Player(name: String, color: String)
- moveTo(coords: Point)
- searchForResource(resourceName: String)
- build(buildingType: String)
+ syncronized getResourceBar(): HashMap<String, ResourceDeta
- syncronized updateResourceBar()
+ getBuildingBar(): HashMap<String, BuildingDetailWrapper>
+ getEventBroadcaster()
+ getName(): String
+ getColor(): String
+ getIcon(): Image
+ getConcreteName(): String
+ speak(message: String)
+ start()
+ stop()
Extends
Resources
{abstract}
ResourceBank
- durability: int
- dropAmount: int
ResourceBank(initDurability: int, dropAmount
+ syncronized exploit(): int
+ syncronized getDurability(): int
+ getDropAmount(): int
+ getIcon(): Image
Extends
FoodResourceBank
+ FoodResourceBank()
+ getConcreteName(): String
Extends
StoneResourceBank
+ StoneResourceBank()
+ getConcreteName(): String
Extends
Extends
Extends
GoldResourceBank
+ GoldResourceBank()
+ getConcreteName(): String
IronResourceBank
+ IronResourceBank()
+ getConcreteName(): String
WoodResourceBank
+ WoodResourceBank()
+ getConcreteName(): String


GameEngine
- mainStage: Stage
- mainScene: Scene
+ start(primaryStage: Stage)
- game()
+ main()
eLogic
Messaging
Message
- sender: GlobalSpeaker
- content: String
+ Message(sender: GlobalSpeaker, content: String)
+ getSender(): GlobalSpeaker
+ getContent(): String
MessagingSystem
- GLOBAL_LOG: MessageLog
- chat(sender: GlobalSpeaker, content: String)
+ reset()
+ getGlobalLog(): MessageLog
+ BEGIN_THREAT_MESSAGES: String[]
MessageLog
- messages: Queue<Message>
- eb: EventBroadcaster
+ MessageLog()
+ synchronized add(message: Message)
+ synchronized lastMessage(): Message
+ getEventBroadcaster(): EventBroadcaster
<<interface>>
GlobalSpeaker
+ speak(what: String)
MapLogic
MapEntities
{abstract}
MapEntity
- map: Map
- coords: Point
setCoords(coords: Point)
getMap(): Map
+ setMap(map: Map)
+ setMap(map: Map, coords: Point)
+ getCoords(): Point
+ {abstract} getIcon(): Image
+ {abstract} getConcreteName(): String
Extends
Buildings
{abstract}
Building
- owner: Player
+ Building(owner: Player)
+ getOwner(): Player
+ getIcon(): Image
+ speak()
Extends
Colosseum
Mosque
Buildings
Extends
{abstract}
Building
- owner: Player
+ Building(owner: Player)
+ getOwner(): Player
+ getIcon(): Image
+ speak()
Extends
Colosseum
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ Colosseum(owner: Player)
+ getConcreteName(): String
Mosque
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ Mosque(owner: Player)
+ getConcreteName(): String
House
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ House(owner: Player)
+ getConcreteName(): String
Castle
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ Castle(owner: Player)
+ getConcreteName(): String
Sphinx
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ Sphinx(owner: Player)
+ getConcreteName(): String
Hospital
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ Hospital(owner: Player)
+ getConcreteName(): String
+ produce(productionHandler: ProductionHandler)
+ getProductionHandler(): ProductionHandler
+ run()
LumberCamp
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ LumberCamp(owner: Player)
+ getConcreteName(): String
+ produce(productionHandler: ProductionHandler)
+ getProductionHandler(): ProductionHandler
+ run()
TradeCenter
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ TradeCenter(owner: Player)
+ getConcreteName(): String
+ produce(productionHandler: ProductionHandler)
+ getProductionHandler(): ProductionHandler
+ run()
Mine
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ Mine(owner: Player)
+ getConcreteName(): String
+ produce(productionHandler: ProductionHandler)
+ getProductionHandler(): ProductionHandler
+ run()
Smithery
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ Smithery(owner: Player)
+ getConcreteName(): String
+ produce(productionHandler: ProductionHandler)
+ getProductionHandler(): ProductionHandler
+ run()
Extends
Mill
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ Mill(owner: Player)
+ getConcreteName(): String
+ produce(productionHandler: ProductionHandler)
+ getProductionHandler(): ProductionHandler
+ run()
Granary
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ Granary(owner: Player)
+ getConcreteName(): String
+ produce(productionHandler: ProductionHandler)
+ getProductionHandler(): ProductionHandler
+ run()
Stable
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ Stable(owner: Player)
+ getConcreteName(): String
+ produce(productionHandler: ProductionHandler)
+ getProductionHandler(): ProductionHandler
+ run()
TownHall
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ TownHall(owner: Player)
+ getConcreteName(): String
+ produce(productionHandler: ProductionHandler)
+ getProductionHandler(): ProductionHandler
+ run()
Extends
Parliament
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ Parliament(owner: Player)
+ getConcreteName(): String
Temple
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ Temple(owner: Player)
+ getConcreteName(): String
Barracks
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ Barracks(owner: Player)
+ getConcreteName(): String
+ produce(productionHandler: ProductionHandler)
+ getProductionHandler(): ProductionHandler
+ run()
Observatory
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ Observatory(owner: Player)
+ getConcreteName(): String
+ produce(productionHandler: ProductionHandler)
+ getProductionHandler(): ProductionHandler
+ run()
Extends
University
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ University(owner: Player)
+ getConcreteName(): String
WatchTower
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ WatchTower(owner: Player)
+ getConcreteName(): String
Mansion
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ Mansion(owner: Player)
+ getConcreteName(): String
+ produce(productionHandler: ProductionHandler)
+ getProductionHandler(): ProductionHandler
+ run()
Church
- br: BuildRequirements
+ getBuildRequirements(): BuildRequirements
+ Church(owner: Player)
+ getConcreteName(): String
+ produce(productionHandler: ProductionHandler)
+ getProductionHandler(): ProductionHandler
+ run()

3.3. Implementation

3.3.1. The Game Logic

Before explaining the game logic there are some relevant classes that are the types of the objects that interact in the program. Thus, the concepts behind them should be understood:

Game:

The class that covers the main configuration object;

It's fields are a reference to the map object, created by the MapBuilder class and the list of players that participate;

Confers methods for starting/interrupting-joining the player threads;

The PIXEL Wars game can be won upon completing it's winning rule. For now, the rule is a concrete one:

In order to win, every player has to build and own at a time a requested collection of "special" buildings;

Theese special buildings are called "objectives", and they manifest the same as normal buildings (they've got an owner, some build requirements, and also, some of them may produce resources within their lifetime thread). The only difference is that they are required to win; Another important thing to mention is that a player may only own one specific objective at a time, so there can't be duplicates among theese objectives;

Decides who are theese objectives that need to be completed in order for the game to be won. The static method getNeededObjectives() returns a collection of strings that are the names of those objective buildings;

In this implementation, the game class requires theese objectives: Colosseum, Mosque and Sphinx to be completed by each player. The minimum number of players is 2 and the maximum number is 8;

After the "BEGIN" button is pressed, the Game starts the player threads and sends them the required-to-win objectives list.

Map:

Represents the virtual common space where the logic occurs;

Wrapper class for a 2D Matrix of MapTile objects. Defines interfaces for manipulating theese MapTiles via xOy coordinates (see Point class);

The mapTilesMatrix object defines both the terrain aspect and constraints and also refers the inhibitants of the map, which would be MapEntities;

Runnable MapEntities (like Players/Buildings) always manifest their actions according to the Map's current state, synchronizing on the MapTile where they perform the action (like movement/resources exploitation/building construction).

MapTile:

Represents a map cell, that can be accessed inside the mapTilesMatrix object by its coordinates (those must be valid of course: $x = \overline{0, width - 1}$ (columns) and $y = \overline{0, height - 1}$ (rows);

Every MapTile contains a reference to one final Terrain instance (representing the concrete type of terrain we got at at those coordinates);

Every MapTile contains a reference of type MapEntity which is null if no MapEntity currently inhabits the MapTile at those coordinates, or on the other hand, refers to the MapEntity object that currently holds that position on the map. One single MapEntity might occupy a MapTile at a time.

They are manipulated in a **thread-safe manner** (synchronized methods for getting/setting/checking the MapEntity). For example: when players read the current state of the map and want to move at some coordinates, they synchronize the MapTile at those coordinates, in order to prevent other player entity to perform an action at the same time. Another example would be when resources reach 0 durability, they auto-update the map with null at their position, now their spot being able to be inhabited by other entities (e.g. one player builds a building in that spot, after the resource is gone). One final relevant example is when players would like to read if a MapTile is free and contains no entity at the concrete time, so they might move right there. Synchronized ensures a happens-befoore relationship. This will ensure that player thread conflicts won't occur over the same MapTile.

MapBuilder:

As mentioned earlier the Map object is returned by the createMap static method inside the **MapBuilder** class

The createMap method does the following:

- creates the terrain elevation matrix using the Perlin Noise procedural generation algorithm within some constrains (width/height from mapSize parameter) and that is converted to a MapTiles matrix, that contains only the terrain and no MapEntity in the beginning.
- after that, the mapTilesMatrix is being populated with the players received as parameter, and also, a number of resources is being computed and spawned on the map object, based on the resDensity parameter;
- finally, the Map object is returned to the game class.

MapEntity:

Abstract class that represents different types of entities that inhabit the map: ResourceBanks, Players, Buildings;

Every MapEntity has some coordinates in xOy space, and of course, a reference to the map object they inhabit;

Every concrete MapEntity has a specific icon that is shown in the GUI (loaded via ImageLoader class);

Every concrete MapEntity has a String type name that is used for different purposes during the game logic (examples: the player's build method/the resource and building factories/icon loading etc);

When a MapEntity updates inside the MapTile matrix, the MapTile sends an event to the ImageView at those coordinates inside the GUI's MAP, updating it with the concrete MapEntity's icon (in a syncrhonized manner).

Player:

Concrete type of MapEntity that has a name, a color, it's own thread (implements runnable), and also has a resourceBar + buildingBar that states their resources amount/building types and counts.

During its run method, the Player tries to win the game by completing it's winning rule. Players have access to the game's map (which is shared among the runnable entities that populate the map), where they are allowed to move, build buildings, and, of course, to exploit resource banks in order to build those mentioned earlier. All theese actions must be performed according to the Map constrains/MapTiles constraints (and **synchronization**) at those coordinates.

One player's workflow would be:

1. While not completing the game's winning rule, the player is rolling for a random Game needed objective that it hasn't been completed yet by the Player and calls build(String buildingName) on it.

2. First step in the build method is to check the BuildRequirements for that objective, if they are met by the player, it will build it in the closest free neighbour MapTile (up/down/left or right, if it's free), else the player random teleports to another free MapTile and applies the same free neighbourhood rule.

3. If the BuildRequirements are not met yet, the player first looks at the Required Buildings that he doesn't own and applies the build method on every single building there, applying steps 1-3 for each of them (basically it's a top-down approach on the dependencies graph);

4. If the player meets the required Buildings or there are no required Buildings in its Build Requirements at all, the player will search on the map for the needed resources for the current building in the build method;

    4.1. The player performs a breadth-first search on the mapTilesMatrix object inside the Map until he founds the needed resource, then, he will try to teleport in a free neighbour MapTile (up/down/left or right, if it's free), else the player continues the breadth first search;

    4.2. If the needed resource is found, the player calls the exploit method on that resource (which is **synchronized**), getting the drop amount from it, then waiting cooldown to rest (just for realism and debugging purposes);

    4.3. If no resource of the type needed is found at all on the map, it means the map reached starvation, so they will FORFEIT and leave/lose the game.

5. After completing all the required resources in the BuildRequirements, the player will do step 2.

6. If the player has completed all the game's objectives, he is declared winner.

Players might obtain resources from the ResourceBanks on the map or from different Buildings (buildings implement runnable, and some of them may produce resources once upon a built-in cooldown time);

They define interfaces for updating their resourceBar/buildingBar in a **synchronized** manner (for example, two building produce resources for they're same owner at the same moment, and the player also finished exploiting a resource of that type at the same moment, all this being done in a **synchronized** manner, avoiding thread conflicts).

ResourceBank:

Concrete type of MapEntity that has a durability and a drop amount. They do not implement runnable.

they're relevant method is the exploit one, which is **synchronized** and can be called by explointant entities (for now, players). When exploit is called, the caller player updates (in **synchronized** way) his resource bar with the amount that dropped, and the resource updates its durability with minus that amount (down to a minimum of zero). When zero durability is reached, the resource object updates the map with null at her coordinates and will be collected by the Garbage Collector at some point in the near future. If players have already found a needed ResourceBank and are “on the way” to it, but the ResourceBank becomes null while player is “on the way”, there won’t be a problem, as the players perform an additional verification on that MapTile (its MapEntity!=null&&MapEntity instanceOf concrete ResourceBank that the player searches);

Up to a maximum number of 4 exploitants might exploit a resource bank at a time, all of them doing this in a synchronized manner. The exploitants need to be neighbours (up/down/left or right) of the resourcebank in order to perform the exploit it;

Synchronization of the exploit method ensures no threading conflicts between exploitants occur (one exploitant locks the ResourceBank while running the exploit code, thus ensuring happens before relationship with the next exploitant that will see the right changes in the ResourceBank’s state).

Building:

Concrete type of MapEntity that has an owner (Player), some BuildRequirements that must be met in order for the building to be built, a list of production handlers (that can be empty) and it's own thread (implements runnable).

During its run method, the Building might produce some resources for its owner once upon a built-in cooldown time. The resources and quantities that it produces are defined by the production handlers (if the building's got any);

Every building defines its BuildRequirements and its production handlers it's class's static block.

P.S: The ProductionHandler interface is designed via the Acyclic Visitor Pattern. Buildings might call requestProduction on different ProductionHandler objects, but only if the specific ProductionHandler implements that building's production interface in the Visitor hierarchy.

3.3.2. The Messaging subsystem

Consist of the interfaces and classes responsible for speaking/chatting in a global event log, that might be read by the user. Thus, the user might trace and monitor the activities/events/changes in the game state/on the map that are performed by ingame entities.

GlobalSpeaker

Interface implemented by any game entity that can speak in the global log (The game class itself, under the name of "SYSTEM", the players and the buildings for now);

The speak method's default implementation will call the Chat method inside the MessagingSystem class, sending as parameters the concrete object that spoke and the String message it spoke.

Message

Wrapper class that contains final references to the sender (GlobalSpeaker) and to the message content String. Message's fields are accessible via getters, but immutable.

MessageLog

Wrapper class for a queue of sent messages, messages being able to be stored in MessageLog objects;

GlobalSpeakers which are also serviced by their own threads (see Players/Buildings) will call the speak method, which adds their message to the static MessageLog reference of the MessagingSystem class. Thus, the addMessage method inside the MessageLog will perform **synchronization** over the messages queue object, in order to avoid threading conflicts like concurrent modification exception;

Readers of the message log (in this case, the Capturer_TextArea which is going to be described in the Event Handling section) call the lastMessage() method (again **synchronized** over the messages queue) in order to get the last added message to the queue.

MessagingSystem

This class contains a static global Message Log instance, where the GlobalSpeakers might call the speak method in order to add messages;

This log is reset every time the game is reset.

3.3.3. The GUI

The classes responsible for the GUI creation, process, update and control are:

GameEngine

Modelates the main stage (e.g. sets the window fullscreen) and scene (applies CSS to the scene from some external css file);

Responsible for the transition between the two scenes, if button events occur: clicking play would trigger the ingame scene, then clicking reset would trigger back the intro (configuration form) scene;

Responsible for starting/stopping the game.

GameUI

Creates all the UI components: layouts/texts/controls/etc. for the intro and ingame scenes;

Responsible for updating/processing theese mentioned earlier;

Responsible for extracting parameters from the intro configuration form and creating the Game object from theese.

ImageLoader

Performs an initial load of all the icon images resources and stores the links in a hashmap, in order for the ingame scene to perform faster.

3.3.4. Event Handling

As described in the 2.3. section (please read that first), the event handling of our application is mainly based on the Observer pattern. The event-handling for the GUI is done using JavaFX 8.

Observable classes:

MapTile

Triggers the notifyEventCapturers() method of its EventBroadcaster instance whenever the setMapEntity method is called on that MapTile;

Concrete EventCapturer: **Capturer_ImageView** extends ImageView from JavaFX 8

The handle method will get the MapEntity currently on the MapTile cause and will set a proper ImageView for it (or if it's null then empty).

Player.ResourceDetailWrapper

Triggers the notify method whenever that player's resourceBar is updated on some specific resource. Player's resourceBar are hashMap<String,ResourceDetailWrapper>, and every ResourceDetailWrapper has a value (integer) and an EventBroadcaster object, whose notify is called on any update of that value integer.

Concrete EventCapturer: **Capturer_TextField** extends TextField from JavaFX 8

The handle method will get the Player.ResourceDetailWrapper cause object's value and display it in the specific field for that specific resource's count in the player's panel;

Player.BuildingDetailWrapper

Performs the same as ResourceDetailWrapper, but for the buildings that the player owns. Instead of an integer value, the BuidlingDetailWrapper contains a list of the buildings of that specific type, that will be updated whenever a player builds a building. Thus, the Capturer_TextField will show the size of that list inside the specific field inside player's panel.

Player

Triggers the notify method whenever moves on the Map;

Concrete EventCapturer: **Capturer_TextField** extends TextField from JavaFX 8

The handle method will get the Player cause's coords and display them in the Position field inside of that player's panel;

MessageLog

Triggers the notify method whenever a new message has been added to the messages queue;

Concrete EventCapturer: **Capturer_TextFlow** extends TextFlow from JavaFX 8

This JavaFX 8 TextFlow behaves the same as a TextArea, but it allows rich text formats such as different fonts/colors etc;

The handle method will get the MessageLog cause object's last message and add it to the TextFlow, styling the text in a specific way, based on the message's sender (GlobalSpeaker). For example: players will have the text colored in their color, buildings the same. The Game (SYSTEM speaker) will have the text colored in white etc.

Observer/Concrete capturer classes implement the EventCapturer interface, overriding it's update method. This update method will call the handle(specific type of cause object) method;

3.3.5. More on Thread-Safety

We achieved the thread-safety by using the java "synchronized" keyword.

Java synchronized keyword marks a block or method a critical section. A critical section is where one and only one thread is executing at a time, and the thread holds the lock for the synchronized section.

Java is multi-threaded language where multiple threads runs parallel to complete their execution. It makes two kinds of errors possible: thread interference and memory consistency errors. We need to synchronize the shared resources to ensure that at a time only one thread is able to access the shared resource in order to prevent these errors.

Java provide two synchronization idioms: synchronized blocks and synchronized methods;

1. Java synchronized Blocks

If we only need to execute some subsequent lines of code not all lines of code within a method, then we should synchronize only block of the code within which required instructions are exists. The general syntax for writing a synchronized block is as follows:

Synchronized block:

```
synchronized( lockObject )
{
  // synchronized statements
}
```

When a thread wants to execute synchronized statements inside the synchronized block, it MUST acquire the lock on lockObject's monitor. At a time, only one thread can acquire the monitor of a lock object. So all other threads must wait till this thread, currently acquired the lock, finish it's execution and release the lock.

In this way, synchronized keyword guarantees that only one thread will be executing the synchronized block statements at a time, and thus prevent multiple threads from corrupting the shared data inside the block.

2. Java synchronized method

The general syntax for writing a synchronized method is as follows:

Synchronized method:

```
<access modifier> synchronized method( parameters )
{
  // synchronized code
}
```

Similar to synchronized block, a thread MUST acquire the lock on the associated monitor object with synchronized method. If a Object is visible to more than one threads, all reads or writes to that Object's fields are done through the synchronized method.

In case of synchronized method, the lock object is:

- '.class' object – if the method is static.
- 'this' object – if the method is not static. 'this' refer to reference to current object in which synchronized method is invoked.

Java synchronized keyword is re-entrant in nature it means if a synchronized method calls another synchronized method which requires same lock then current thread which is holding lock can enter into that method without acquiring lock.

Important points:

i. When thread enters into synchronized instance method or block, it acquires Object level lock and when it enters into synchronized static method or block it acquires class level lock;

ii. Java synchronization will throw null pointer exception if Object used in synchronized block is null. For example, If in synchronized(instance) , instance is null then it will throw null pointer exception;

iii. Java synchronized method run very slowly so you should synchronize the method when it is absolutely necessary because it may degrade the performance.

4. Other libraries

4.1. JavaFX 8

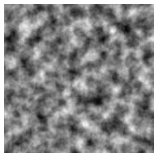
JavaFX is a Java library used to build Rich Internet Applications. The applications written using this library can run consistently across multiple platforms. The applications developed using JavaFX can run on various devices such as Desktop Computers, Mobile Phones, TVs, Tablets, etc. To develop GUI Applications using Java programming language, the programmers rely on libraries such as Advanced Windowing Toolkit and Swing. After the advent of JavaFX, these Java programmers can now develop GUI applications effectively with rich content.

Features of JavaFX

- **Written in Java** − The JavaFX library is written in Java and is available for the languages that can be executed on a JVM, which include − Java, Groovy and JRuby. These JavaFX applications are also platform independent.
- **FXML** − JavaFX features a language known as FXML, which is a HTML like declarative markup language. The sole purpose of this language is to define a user Interface.
- **Scene Builder** − JavaFX provides an application named Scene Builder. On integrating this application in IDE's such as Eclipse and NetBeans, the users can access a drag and drop design interface, which is used to develop FXML applications (just like Swing Drag & Drop and DreamWeaver Applications).
- **Swing Interoperability** − In a JavaFX application, you can embed Swing content using the Swing Node class. Similarly, you can update the existing Swing applications with JavaFX features like embedded web content and rich graphics media.
- **Built-in UI controls** − JavaFX library caters UI controls using which we can develop a full-featured application.
- **CSS like Styling** − JavaFX provides a CSS like styling. By using this, you can improve the design of your application with a simple knowledge of CSS.
- **Canvas and Printing API** − JavaFX provides Canvas, an immediate mode style of rendering API. Within the package javafx.scene.canvas it holds a set of classes for canvas, using which we can draw directly within an area of the JavaFX scene. JavaFX also provides classes for Printing purposes in the package javafx.print.
- **Rich set of API's** − JavaFX library provides a rich set of API's to develop GUI applications, 2D and 3D graphics, etc. This set of API's also includes capabilities of Java platform. Therefore, using this API, you can access the features of Java languages such as Generics, Annotations, Multithreading, and Lambda Expressions. The traditional Java Collections library was enhanced and concepts like observable lists and maps were included in it. Using these, the users can observe the changes in the data models.
- **Integrated Graphics library** − JavaFX provides classes for 2d and 3d graphics.

4.2. Perlin noise

Perlin noise is a procedural texture primitive, a type of gradient noise used by visual effects artists to increase the appearance of realism in computer graphics. The function has a pseudo-random appearance, yet all of its visual details are the same size. This property allows it to be readily controllable; multiple scaled copies of Perlin noise can be inserted into mathematical expressions to create a great variety of procedural textures. Synthetic textures using Perlin noise are often used in CGI to make computer-generated visual elements – such as object surfaces, fire, smoke, or clouds – appear more natural, by imitating the controlled random appearance of textures in nature.



It is also frequently used to generate textures when memory is extremely limited, such as in demos. Its successors, such as fractal noise and simplex noise, have become nearly ubiquitous in graphics processing units both for real-time graphics and for non-realtime procedural textures in all kinds of computer graphics.

5. Versioning system and maintenance

Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity, and support for distributed, non-linear workflows.

All the code for this prototype strategy-game with threading application we called “PIXEL Wars” is yet **open-source** for personal and non-commercial use and can be found at the following link.

6. Tools used for development

IntelliJ IDEA is an integrated development environment (IDE) written in Java for developing computer software. It is developed by JetBrains (formerly known as IntelliJ), and is available as an Apache 2 Licensed community edition, and in a proprietary commercial edition. Both can be used for commercial development.

7. Known Limitations

The JavaFX tends to run a little slow at some points on some computers, this being a common problem as it is considered not to be a really adequate framework for a game GUI. We are looking forward to migrating to some other technology.

8. Roles and contributions to the project

Team CASE members:

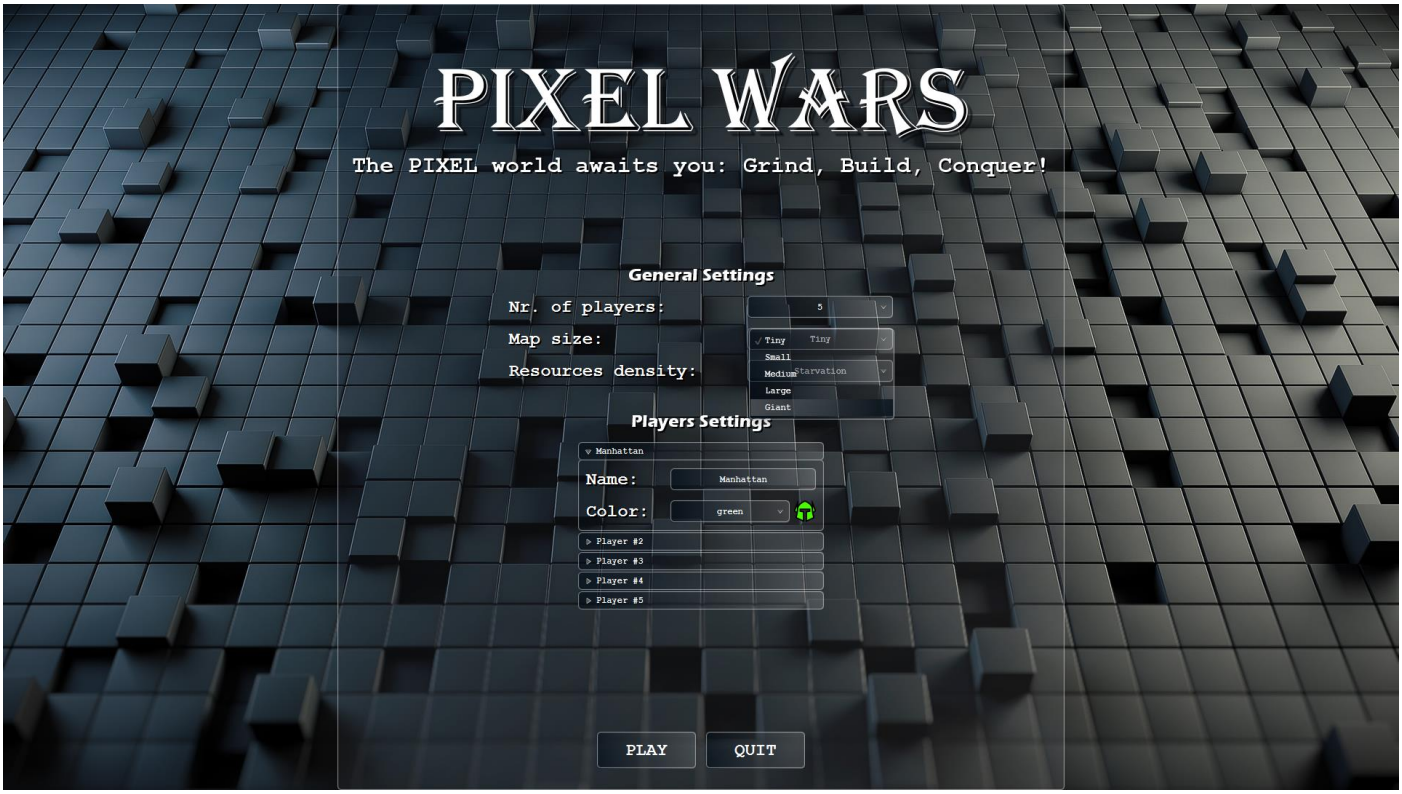
Mecheș Sebastian
Application logic, Thread-safety, Event-Handling, Developer, Object-Oriented design and implementation, Game concept idea, Graphical User Interface

Iuonac Daniel
Application logic, Thread-safety, Event-Handling, Developer, Problem solver, Software design, Breadth-first-search and building algorithms

Berlovan Adrian
Application logic, Thread-safety, Event-Handling, Developer, Map building/generation (Perlin Noise generator) and other algorithms, Software design, Game concept idea

Dobroj Klaudio
Application logic, Thread-safety, Event-Handling, Developer, Debugging, Artwork

9. The result


PIXEL WARS
The PIXEL world awaits you: Grind, Build, Conquer!
General Settings
Nr. of players:
5
Map size:
Tiny
Tiny
Small
Medium
Large
Giant
Resources density:
Starvation
Players Settings
Manhattan
Name:
Manhattan
Color:
green
Player #2
Player #3
Player #4
Player #5
PLAY
QUIT


Players Panels
Manhattan
Position:
X:30,Y:118
Resources
Food
Gold
Iron
Stone
Wood
145
125
50
260
25
Buildings
Hospital - 400 150 100 100
Owned: 3
House - 30 30
Owned: 4
LumberCamp - 40 40
Player #2
Player #3
Player #4
Player #5
Global Event Log
Player #5 (green): Exploited 15 from GoldResourceBank at 51 101. Now has 55 Gold
Player #3 (yellow): Exploited 20 from IronResourceBank at 66 23. Now has 60 Iron
Manhattan (green): Exploited 30 from IronResourceBank at 23 99. Now has 40 Iron
Player #2 (orange): Exploited 50 from FoodResourceBank at 43 74. Now has 155 Food
Player #5 (green): Exploited 15 from GoldResourceBank at 51 101. Now has 70 Gold
Manhattan (green): Exploited 20 from IronResourceBank at 23 99. Now has 60 Iron
Player #3 (yellow): Exploited 30 from IronResourceBank at 66 27. Now has 90 Iron
Player #4 (brown): Exploited 50 from FoodResourceBank at 124 100. Now has 125 Food
Player #2 (orange): Exploited 30 from IronResourceBank at 42 73. Now has 40 Iron
Player #5 (green): Exploited 15 from GoldResourceBank at 51 101. Now has 85 Gold
Manhattan's Temple: Produced resources.
Player #3 (yellow): Exploited 30 from IronResourceBank at 66 27. Now has 120 Iron
Manhattan (green): Exploited 30 from IronResourceBank at 25 96. Now has 90 Iron
Player #4 (brown): Exploited 50 from FoodResourceBank at 124 150. Now has 175 Food
Player #2 (orange): Exploited 20 from IronResourceBank at 42 73. Now has 60 Iron
PIXEL Map
BEGIN
RESET

10. Bibliography

1. *Learn JavaFX 8: Building User Experience and Interfaces with Java 8 1st ed.* by Kishori Sharan;
2. *JavaFX tutorials:* http://tutorials.jenkov.com/javafx/index.html;
3. *Perlin noise & procedural Terrain Generation:* https://www.youtube.com/watch?v=wbpMiKiSKm8&list=PLFt_AvWsXl0eBW2EiBtl_sxmDtSgZBxB3;
4. *Java concurrency*: http://tutorials.jenkov.com/java-concurrency/index.html;
5. *JavaFX Thread Confinement*: https://www.developer.com/java/data/multithreading-in-javafx.html;