

ETH ZURICH

Chair of Software Engineering

FLAT_HUNT

User & Developer Documentation

Table of Contents

Part I User Guide	2
Introduction	3
<i>About this document</i>	<i>4</i>
<i>Legal stuff</i>	<i>4</i>
What's the story?	5
How to play?	6
<i>General rules</i>	<i>6</i>
<i>Game modes</i>	<i>7</i>
<i>Other things to do...</i>	<i>8</i>
Part II Developer Guide	10
How It works	11
<i>Overview</i>	<i>11</i>
FLAT_HUNT	12
<i>Cluster Controller</i>	<i>13</i>
<i>Cluster Model</i>	<i>13</i>
<i>Cluster View</i>	<i>14</i>
The states of the game	15
<i>Game loop</i>	<i>15</i>
TRAFFIC – the library	17
<i>Cluster Graph</i>	<i>17</i>
<i>Cluster City</i>	<i>17</i>
<i>Cluster Visualization</i>	<i>18</i>
Guided “walk-through”	19

Part I

User Guide

Introduction

Welcome to FLAT_HUNT!

FLAT_HUNT is a simple adaptation of the well-known board game “Scotland Yard” (see Figure 1). Instead of some agents hunting Mr. X all around London, it is about a group of students starting off at ETH Zurich. To make their student life a bit more pleasant, they are desperately trying to find a flat in this little big city. But to get a flat, they must first meet the estate agent, who is running all around Zurich showing his flats to other people...

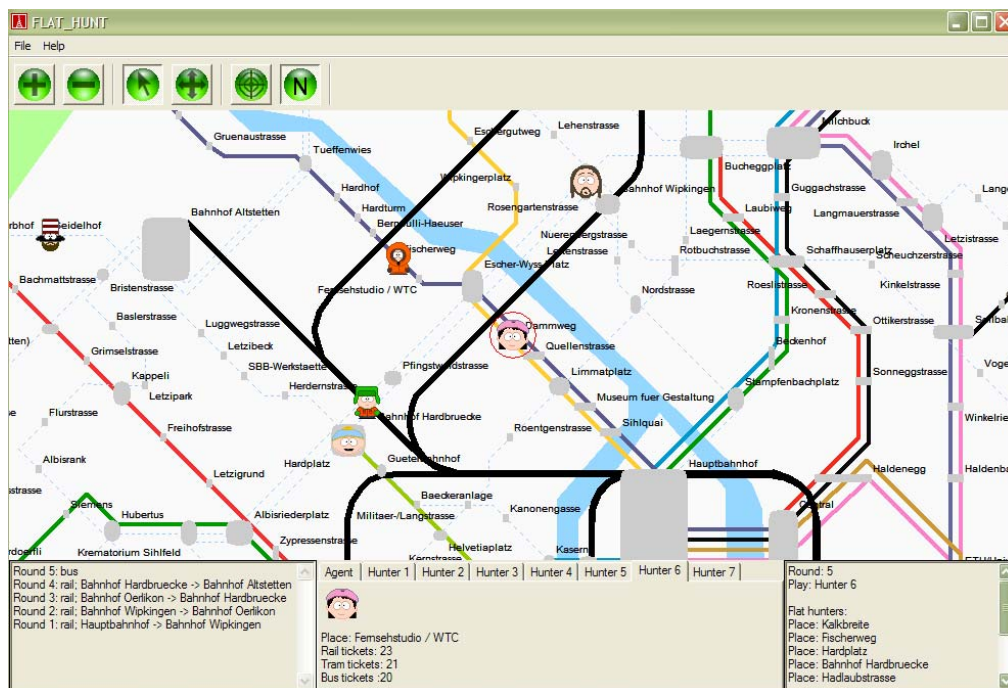


Figure 1 FLAT_HUNT screenshot

About this document

This first part is the FLAT_HUNT User Guide. The next chapter goes a bit more into the story of FLAT_HUNT. Chapter 3 tells you how this game is meant to be played.

Part II (Chapters 4 to 8) explains in detail how the game works, describes the classes used and highlights some of the features.

Words in all UPPERCASE usually refer to classes of FLAT_HUNT or TRAFFIC. Words in small letters and in *italic* describe features of classes. Cluster names are starting with uppercase in *Italic*. And `this is actual Eiffel code`.

Legal stuff

The South Park icons used in FLAT_HUNT are courtesy of Trey Parker & Matt Stone.

The idea to FLAT_HUNT is based on Scotland Yard, a board game courtesy of Ravensburger.

What's the story?

As the title suggests (and the introduction mentions), it is all about finding a flat in Zurich...

However, this is not so easy... There is this guy, the estate agent (Figure 2 a), who is renting flats. The problem is that he is always busy showing flats to other customers (e.g. other students), and even in his office they don't really always know where exactly he is. The only thing they know is what kind of transport he is moving around with. This is because the estate agent is taking part in a new VBZ¹-project called "Customer tracking".

In collaboration with ETH, they equipped some volunteers with transponders. These transponders gather information like current position and type of transport, and send it in real-time to the office. However, for privacy reasons, only the type of transport can be accessed all the time.

Once in a while, the estate agent calls his office to tell the secretary which flat he is currently visiting. So sometimes, the people there in the office can tell "you" where to look for the guy... "You" meaning yourself (Figure 2 b) and your friends (Figure 2 c), the guys you want to share the flat with...



Figure 2 Estate agent (a),



you (b)



and your friends (c)

¹ Verkehrsbetriebe Zürich (Zurich Public Transport)

How to play?

Playing FLAT_HUNT is not very difficult, especially for those that know the game “Scotland Yard”...

This chapter is structured in three sections. The first section describes the general rules of the game. Then the different modes are explained more precisely. The last section is about other stuff like how to open a new map or how to start a new game.

General rules

The game lasts for at most 23 rounds. In these 23 rounds, the flat hunters try to find the estate agent, while he tries to avoid them (this is because he would rather rent the flats to elderly couples, since presumably they make fewer parties in the middle of the night...).

In each round, every player can make one move on the public transport system². The estate agent is the first, then it's the hunters turn. One move is either

- one or two stops by tram (coloured lines),
- one stop by train (thick black lines),
- or one stop by bus (thin dashed blue lines).

A move with a certain transport can only be made if one has still enough tickets (see Figure 3), if there is a connection (obviously), and if there is no other player at that destination (and in the case of tram lines, if there is no hunter in between).

Agent	Hunter 1	Hunter 2
Place: Schwamendingerplatz		
Rail tickets: 4		
Tram tickets: 11		
Bus tickets :6		

Figure 3 Tickets left for Hunter 1

² By far the easiest way to move around in Zurich (except for bicycles)

Attention: If you are at a bus-only stop, and you run out of bus tickets, you will get stuck there forever, so be careful...

The possible places one can move to have a blue highlighted outline (see Figure 4). To make a move, just click on one of those highlighted places. The red circle centers on the player whose turn it is, and in the information panel at the bottom, the current player's tab is highlighted and you can see the player's image.



Figure 4 Highlighting of possible moves

Game over?

The game is over when

- a) the hunters could not find the estate agent within 23 rounds,
- b) one flat hunter moves onto the place where the agent currently is,
- c) or the hunters encircle the estate agent so that he cannot move anymore.

In case a), the winner is the estate agent (he does not have to rent his flat to students), whereas in b) and c) it is the hunters that win, as they get to meet the estate agent on time and thus manage to find a flat.

Game modes

There are four modes to play FLAT_HUNT: *Hunt*, *Escape*, *Versus* and *Demo*. Depending on the mode, zero (*Versus*), one (*Hunt/Escape*) or two (*Demo*) parts are taken over by the computer.

Hunt

This is probably the most typical situation; the player tries to find the agent, which is played by the computer. Thus, the player only knows about every fifth move where the agent just was... The exact route of the agent can be seen at the bottom left (see Figure 5). The agent shows himself only in rounds number 1, 3, 8, 13, 18, and 23.



Figure 5 Agent just showing himself, bottom shows the route he has taken

Escape

This is just the opposite: The agent is played by you, and the hunters are played by the computer. The hunters always move as close in your direction as possible, as they somehow manage to decode your transponder signal, and thus always know your precise location (so much for privacy...). You just have to try to avoid them as long as possible...

Versus

This is the multiplayer mode. One of the players is the agent; the other plays all the hunters. While the player of the agent is making a move, the player of the hunters is supposed to look away...

Demo

This mode is more or less the opposite of the buzzword “interactive”, but is about as entertaining as watching fish in an aquarium. The computer is playing against himself, trying to catch the agent as fast as possible.

Other things to do...

Start new game

At any time, a new game can be started. Just select File → Start... (see Figure 6) or press [Enter] and the game dialog (see Figure 7) will appear. There you can select the game mode and specify the number of hunters. You can also use this dialog to alternate between the small and the big map of Zurich.

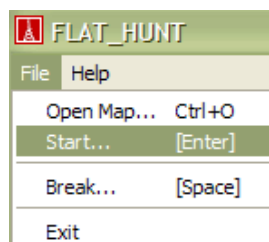


Figure 6 File menu

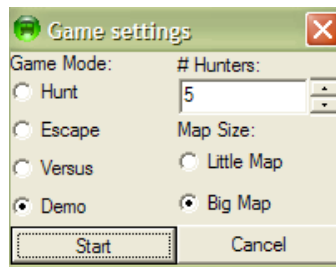


Figure 7 Game dialog

Open new map

The FLAT_HUNT game comes with two maps: “*zurich_big_city.xml*” and “*zurich_little_city.xml*”. By using the File menu (see Figure 6) or [Ctrl-O] the map can be changed.

Buttons and keyboard short-cuts

Figure 8 shows the button panel in FLAT_HUNT. From left to right, they have the following function:

- Zoom in and out (you can also use your mouse wheel for this).
- Selection mode: Normal mode to play the game.
- Move mode: If you want to move the map (can also be done by using the right mouse button). Go back to “selection mode” afterwards.
- Center map on current player.
- Switch the place names on/off.



Figure 8 Buttons in FLAT_HUNT

If you press [Space] any time during the game, the game will stop after the next players move until you press [Space] again.

Part II

Developer Guide

How it works

This and the following chapter should help you to understand how exactly FLAT_HUNT works. First you will get an overview of the whole systems organization, and then we will have look at some important classes in more detail (Chapter 5).

The different states the FLAT_HUNT game can be in are described in Chapter 6. In Chapter 7 we will have a short glance at the TRAFFIC library, which is also used in the TOUCH_APPLICATION. This developer guide finishes with a “walk-through” of a typical game in Chapter 8.

Overview

When opening FLAT_HUNT in EiffelStudio, the cluster view in the bottom right corner shows three clusters: *Dependencies*, *Flat_hunt* and *Traffic*. Figure 9 shows them already expanded.

Cluster *Flat_hunt* is described in Chapter 5 and 6, and Chapter 7 introduces the *Traffic* cluster.

The cluster *Dependencies* contains several libraries that are used in FLAT_HUNT, like *Time* to find out how many milliseconds have passed, *Gobo* which includes data structures like DS_LINKED_LIST or *Vision2* on which the graphical user interface (GUI) of FLAT_HUNT is based.

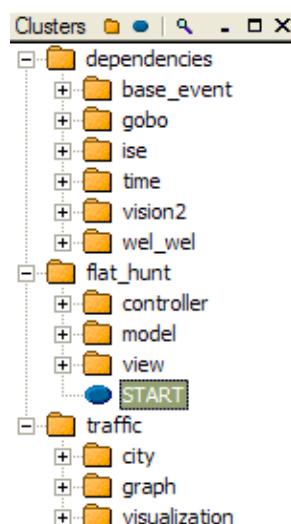




Figure 9 Cluster hierarchy

FLAT_HUNT

To remove complexity, FLAT_HUNT is structured in three clusters (see Figure 10): *Model*, *View* and *Controller*. In each cluster, there are several classes, and sometimes there are subclusters. Like in any object-oriented system, these classes are connected. This is symbolized by the red and blue arrows.

 Red arrows describe an **inheritance relationship**. For example, class BOT (7) inherits from class BRAIN (6). This has the effect that class BOT can do the same things as class BRAIN, because it inherits all the features from class BRAIN. Usually, the class that inherits (BOT) can do some additional things which are not defined in the parent class (BRAIN).

 Blue arrows stand for a **client-supplier relationship**. The class the arrow points to is the supplier, the other class is the client that makes use of the supplier class. In FLAT_HUNT, class GAME (3) is a client of class PLAYER (4). Using the client-supplier relationship allows the client to access features on the supplier.

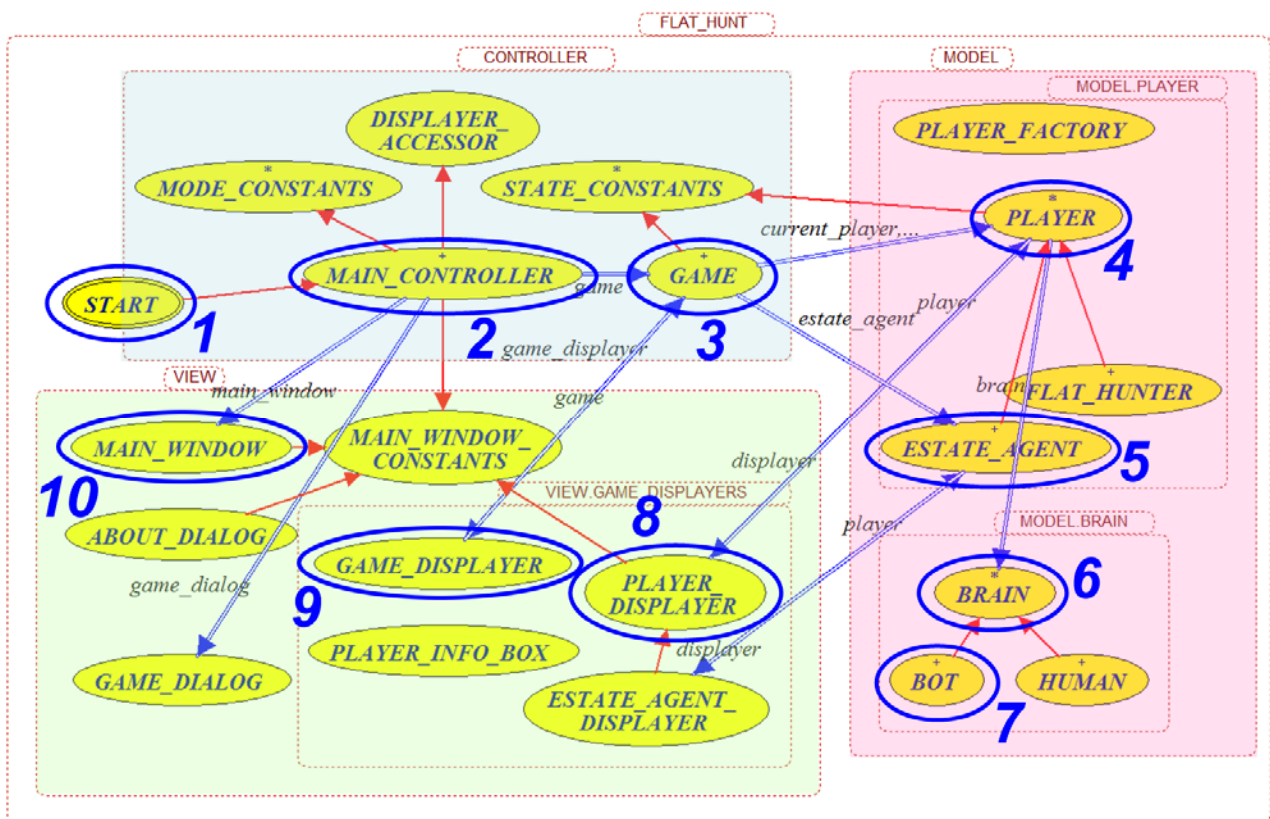


Figure 10 FLAT_HUNT class diagram

Cluster *Controller*

Cluster *Controller* is the fundamental cluster in FLAT_HUNT. Here are the classes that “control” the action. They make sure that the displayer classes in cluster *View* display the proper information, which they get from the *Model* classes. For example, feature *prepare* in class GAME controls the display update by calling `current_player.displayer.display_after_move`.

- **START (1):** This is the entry point to FLAT_HUNT. Almost all features are inherited from class MAIN_CONTROLLER. There is only one feature called *start* which is indeed the start of the whole system. When you run FLAT_HUNT, this is where it all begins.
- **MAIN_CONTROLLER (2):** The MAIN_CONTROLLER is (as the name suggests) responsible for many things. It provides access to the MAIN_WINDOW, to class GAME (3) and to the whole TRAFFIC library, which is responsible for the visualization of the city (not visible in Figure 10, see Chapter 7).
- **GAME (3):** Class GAME features the FLAT_HUNTER logic. It knows which players turn it is. And as it is an heir to class STATE_CONSTANTS, it also knows the game state (see Chapter 6).

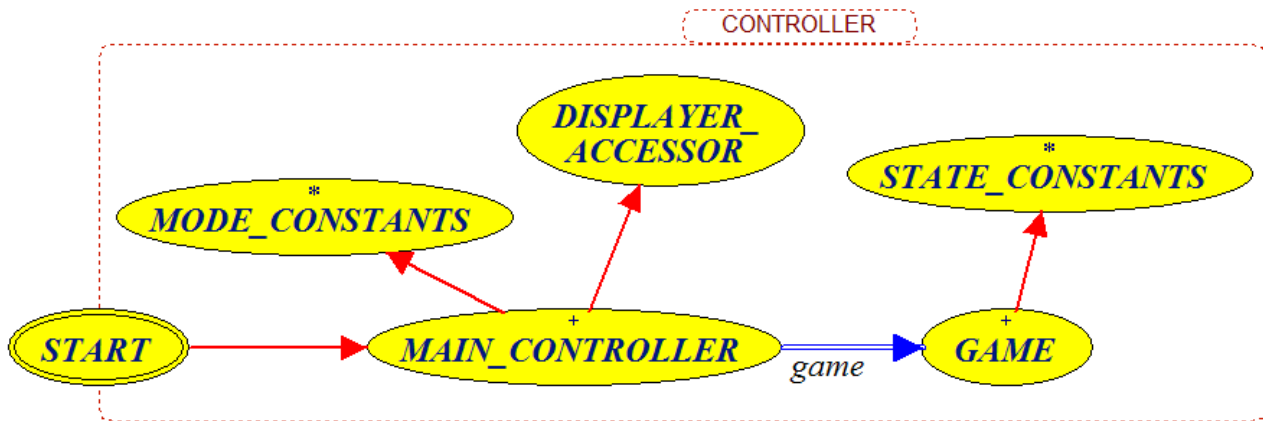


Figure 11 Classes in cluster *Controller*

Cluster *Model*

In the cluster *Model*, there are two important parent classes: Class PLAYER and class BRAIN. PLAYER is the parent of FLAT_HUNTER and ESTATE_AGENT, and BRAIN is the parent of HUMAN and BOT. These *Model* classes describe the internal representation of “real world” objects. Here’s a description of four of these classes:

- **PLAYER (4):** Class PLAYER knows the basic things one needs to know about a player of FLAT_HUNT, like how many tickets are left. It features the commands *play* and *move* (see Chapter 6) and has either a HUMAN or a BOT *brain*.
- **ESTATE_AGENT (5):** This is one of the two heirs of class PLAYER. It has some additional information that is special for an estate agent player like knowing where he last showed himself.
- **BRAIN (6):** Class BRAIN includes the intelligence to choose the next move.

- **BOT (7):** BOT is one of the two intelligences to choose a move. In contrast to class HUMAN, this is an artificial intelligence (AI).

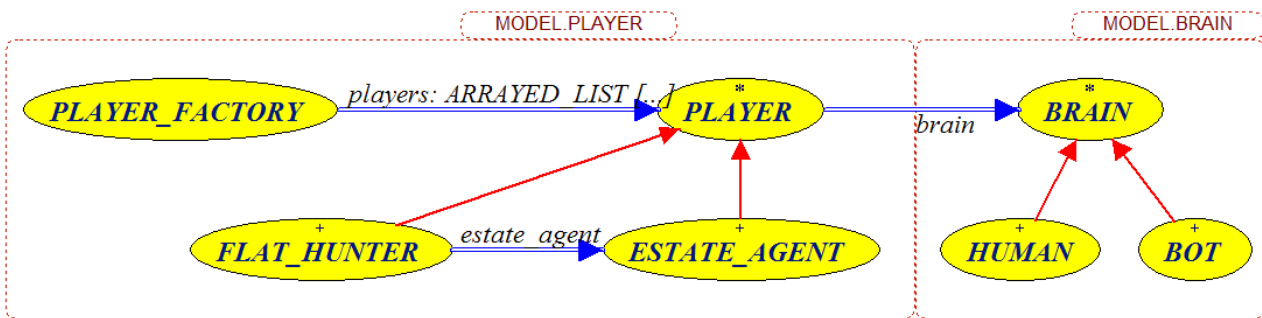


Figure 12 Classes in cluster *Model*

Cluster *View*

This clusters job is to make sure that the user sees what is going on. It includes all windows and dialog boxes, as well as displayers for the game players and the game itself:

- **PLAYER_DISPLAYER (8):** This class displays the player on the map and prints the amount of tickets left. PLAYER_DISPLAYER knows this information because of the client-supplier relationship with class PLAYER. PLAYER_DISPLAYER also features the *animate_defeat* animation.
- **GAME_DISPLAYER (9):** ... displays the game statistics, like whose turn it is and whether it is game over or not. *This class does not exist at the beginning. In assignment 4 it will be your task to create this class.*
- **MAIN_WINDOW (10):** MAIN_WINDOW includes not only the canvas on which the map is painted, but also all the info_boxes, the buttons and the menu.

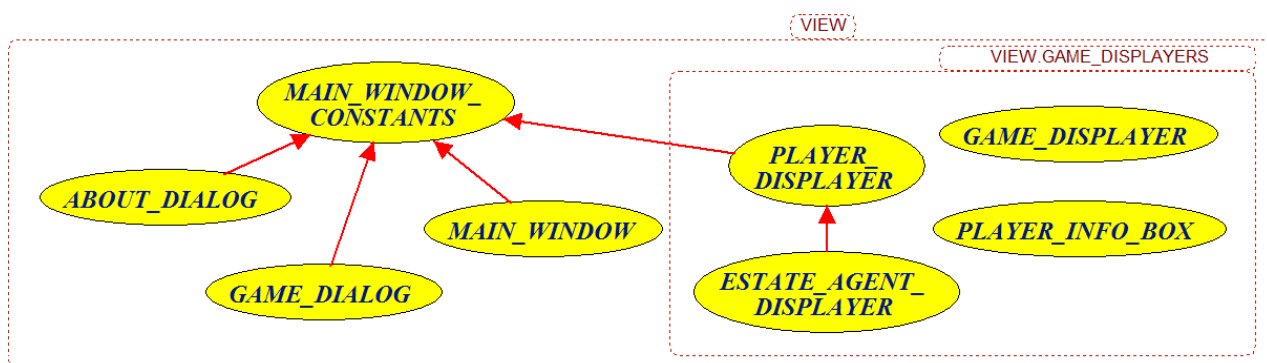


Figure 13 Classes in cluster *View*

The states of the game

Every game has at least two states: playing and game over. FLAT_HUNT has six states in total; three playing states and three game over states (see Figure 14). These game states are defined in class `STATE_CONSTANTS`:

```
Agent_stuck, Agent_caught, Agent_escapes, Prepare_state, Play_state, Move_state: INTEGER is unique
```

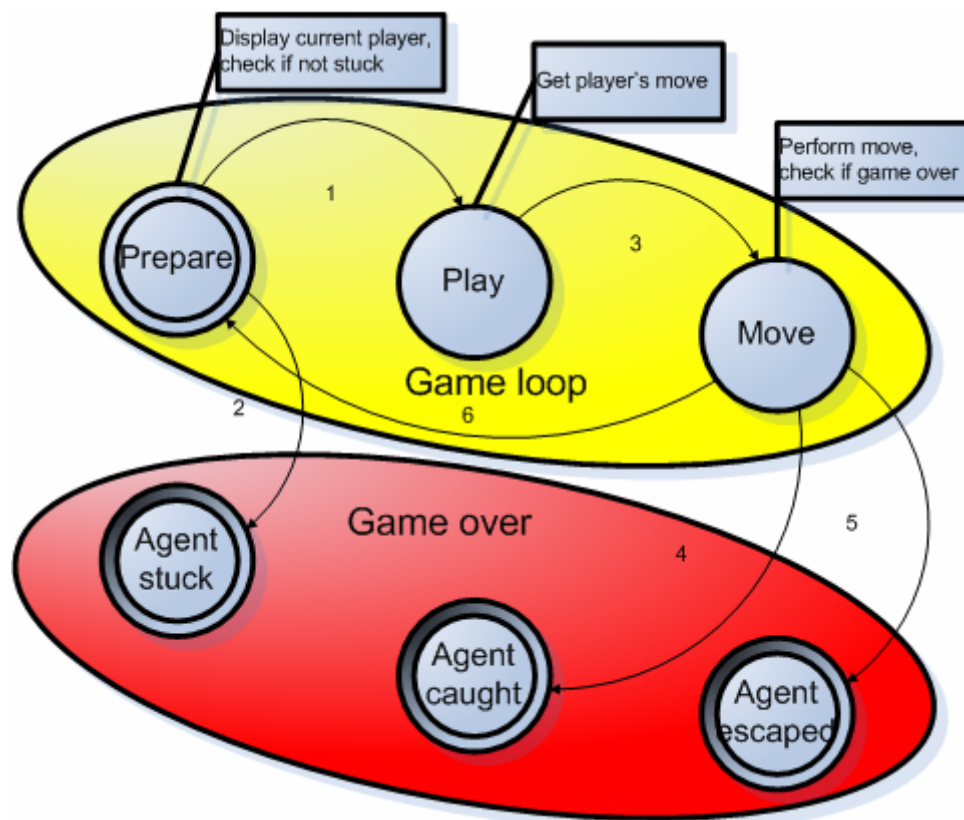


Figure 14 Game states

Game loop

For each player in each round in FLAT_HUNT, the game goes through the following states: *Prepare*, *Play* and *Move*. In addition, there are three game over states: *Agent_stuck*, *Agent_caught* and *Agent_escapes*.

- **Prepare:** If the game is in state *Prepare*, the current player gets a red circle and the possible moves are calculated and displayed. If the current player is the estate agent, and there are no possible

moves, the agent is stuck and thus the game is over (state *Agent_stuck*, 1). If that is not the case, the game goes in state *Play* (2).

- **Play:** In state *Play*, if the current player is played by a human, the game waits until the human player clicks on one of the places that are highlighted. If the player is controlled by an artificial intelligence, then the best of the possible moves is calculated. The game then goes in state *Move* (3).
- **Move:** *Move* does perform the move selected in state *Play*. After the move, the game checks if the player hits the place of the estate agent. If that is the case, the game goes into state *Agent_caught* (4). If the agent did not get caught, and the round number is greater than 23, then the estate agent is the winner and the game goes into state *Agent_escaped* (5). If none of the above is the case, then it's the next player's turn and the game loop starts again in state *Prepare* (6, see also example below).

In the classes MAIN_CONTROLLER, GAME and PLAYER, you can find the features *prepare*, *play* and *move* that deal with these game states. As an example, let's have a look at feature *move* in class GAME:

```
move is
    -- Make the chosen move.
    do
        current_player.move
        if current_player.location = estate_agent.location
        and current_player /= estate_agent then
            state := Agent_caught
        else
            state := Prepare_state
            next_turn
        end
    end
end
```

TRAFFIC – the library

Like the TOUCH_APPLICATION, FLAT_HUNT also makes use of TRAFFIC, a library of classes especially designed for ‘Introduction to programming’. This chapter will give a short overview of the TRAFFIC library.

TRAFFIC consists of three clusters: *Graph*, *City* and *Visualization*.

Cluster *Graph*

This small cluster features three classes: GRAPH, NODE and EDGE. They build the basis for the city’s transport network, as the places in the city are based on class NODE and the transportation links are implemented using EDGES. This is achieved by inheriting from class GRAPH in class NETWORK:

```
class
    NETWORK
inherit
    GRAPH [PLACE, LINK]
```

Cluster *City*

The FLAT_HUNT application uses this cluster to display a map of Zurich. Class CITY is the main entry point if access to places or lines is needed, because it contains lists of all the PLACES, LINES and LINKS:

```
links: DS_LINKED_LIST [LINK]
      -- Container in which all links of the city are stored
```

Through the feature *transport_network* of type NETWORK (see cluster *Graph* above), feature *calculate_possible_moves* in class GAME can find out the next possible moves:

```
outgoing_links := city.transport_network.outgoing_links (a_player.location)
```

Figure 15 shows the important classes in cluster *City* including all inheritance relations (red arrows).

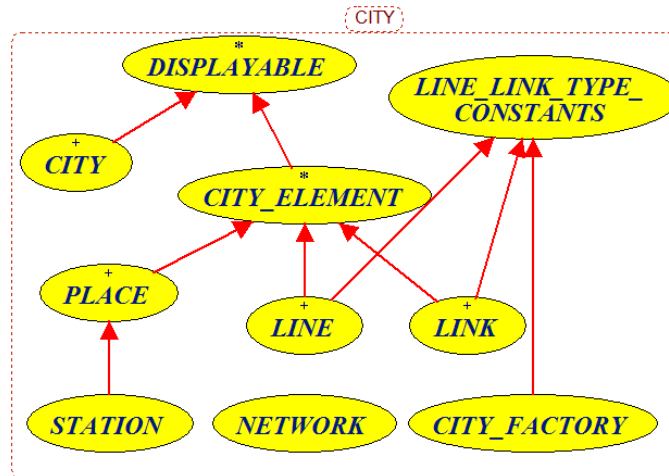


Figure 15 *City* inheritance diagram

Cluster *City* includes a subcluster called *Input*. This cluster contains classes that help reading in the map of the city, which is in XML-format. Below you can see some lines of *zurich_little_city.xml*:

```
<?xml version="1.0"?>
<city name="Zurich (little)">
  <places>
    <place name="Bahnhof Stadelhofen"/>
    <place name="Bellevue"/>
    <place name="Boersenstrasse"/>
    <place name="Buerkliplatz"/>
  </places>
  <lines>
    <line name="11" type="tram">
      <line_displayer thickness="4">
        <color red="0" green="154" blue="0" type="standard"/>
        <color red="10" green="184" blue="20" type="highlight"/>
      </line_displayer>
      <link from="Bahnhof Stadelhofen" to="Bellevue" direction="undirected">
        <point x="910" y="495"/>
        <point x="843" y="495"/>
      </link>
      <link from="Bellevue" to="Buerkliplatz" direction="undirected">
        <point x="843" y="495"/>
        <point x="750" y="495"/>
      </link>
      <link from="Buerkliplatz" to="Boersenstrasse" direction="undirected">
        <point x="750" y="495"/>
        <point x="735" y="495"/>
        <point x="710" y="520"/>
      </link>
    </line>
  </lines>
</city>
```

As you can see, the elements places, lines and links directly relate to the classes PLACE, LINE and LINK in cluster CITY.

Cluster *Visualization*

This cluster contains mostly DISPLAYER classes like CITY_DISPLAYER or PLACE_DISPLAYER. They help in displaying the city's objects. The subcluster *Graphics* features lots of DRAWABLE classes (like DRAWABLE_ROUNDED_RECTANGLE which is used in class PLACE_DISPLAYER) as well as the CANVAS class on which everything is drawn.

Guided “walk-through”

What happens when you start FLAT_HUNT? In this last chapter we will go step-by-step through a typical FLAT_HUNT game. However, because there are lots of details involved, we concentrate on the more important steps...

1. A call to *start_game* in MAIN_CONTROLLER creates a game of the proper gaming mode by calling `game.make`.
2. *make* in class GAME creates the players using class PLAYER_FACTORY and sets the game state to *Prepare*.
3. In class PLAYER_FACTORY, for example the estate agent is created using `estate_agent.make` in feature *build_players*.
4. This creates a HUMAN or BOT *brain* depending on the value of *bot_estate_agent*.
5. Back to class MAIN_CONTROLLER: Feature *idle_action* gets called whenever nothing is going on, i.e. now. *idle_action* checks whether the game is in one of the three game loop states, and calls the corresponding feature in class MAIN_CONTROLLER. In the first run, this is *prepare*...
6. ...which centers the city map on `game.current_player` and then calls `game.prepare`.
7. *prepare* of class GAME first calculates the estate agents possible moves (see also Section “Cluster City” in Chapter 7). If there are no possible moves (`current_player.possible_moves.is_empty`) then it’s either the next player’s turn or the state is set to *Agent_stuck*. Otherwise it’s `state := Play_state`.
8. With that, the call to *prepare* (Step 5) comes to an end and control goes back to feature *idle_action* of class MAIN_CONTROLLER. According to the present state, *idle_action* will now call *play* which then calls `game.play`.
9. This calls `current_player.play (place)`, where `place` is the last place the user clicked on. `place` is then passed on to class BRAIN.
10. *choose_move* in class PLAYER is deferred, which means that *choose_move* of class ESTATE_AGENT or FLAT_HUNTER gets called, depending on whether the current player is a hunter or an agent.
11. FLAT_HUNTERS *choose_move* calls *choose_estate_agent_move* on class BRAIN.

12. Again, this is a deferred feature. Let's assume that the agent is played by a human player. *choose_estate_agent_move* of class HUMAN (which inherits from BRAIN) checks whether the *place* clicked on (see Step 9) is one of the destinations of the *possible_moves* (see Step 7).
13. Back to class ESTATE_AGENT: *next_move := brain.chosen_move* selects the new move.
14. In class GAME, the state is now changed to *Move_state*.
15. Feature *idle_action* in class MAIN_CONTROLLER: After *play*, if the player is a BOT, *sleep_and_process* takes a short break...
16. Next comes *move* which calls *game.move* which in turn calls *current_player.move*.
17. *location := next_move.other_end (location)* in class PLAYER sets the player to the new position.
18. Back to *move* in class GAME: The state is either set back to *Prepare_state* and it's the next player's turn (*next_turn*) or the state is set to *Agent_caught*.
19. The last *if* in *idle_action* tests whether the game is in one of the game over states via *game.is_game_over*. Class GAME inherits this feature from class STATE_CONSTANTS.
20. If the above test yields *True*, *end_game* of class MAIN_CONTROLLER is called and displays the game over messages.

We hope that this little walk-through could give you an idea of what's going on when you run FLAT_HUNT, and that you enjoy both playing and working with FLAT_HUNT.