# Flat Hunt Redesign
# and
# ESDL Extensions

Ursina Caluori
ucaluori@student.ethz.ch

September 26, 2005

# Contents

## 1 Introduction

For two years now, the department of Computer Science at ETH Zurich has been applying a new teaching approach called "Inverted Curriculum" [1] [2] [6] to the *Introduction to Programming* course. This technique is also referred to as a "outside-in" strategy of teaching. Rather than beginning with writing the infamous "Hello World" program, the students work from the start with a big software framework, which they gradually get to know better. At first, the exercises consist of merely calling a few library functions. Later on, control structures, Design by Contract, genericity and other advanced topics are introduced, some also using the framework.
The framework consists of a game-like application called *Flat Hunt* and several libraries upon which the game is built. Namely *TRAFFIC* [4], *EiffelBase* and *EiffelVision2*.

Due to the complexity of *EiffelVision2* and the sheer unlimited multimedia capabilities of *EiffelMedia* (which is the new name of *ESDL* [7] [8] [5]), it was decided that *Flat Hunt* should be redesigned to use *EiffelMedia* for visualization rather than *EiffelVision2*. And exactly that is the goal of this semester thesis. Well, actually almost exactly - for the goal also includes *ESDL* extensions (see section 3), suggestions for student assignments (see section 4) and last but not least also the redesign from a graphical point of view (meaning not only the application code will get a refresh but also the "look and feel" of the game).

Since *Flat Hunt* is a teaching application, a great responsibilty lay upon my shoulders in producing a very clean design and code of impeccable quality (which is generally an utopia in software engineering, if you ask me - hence I don't claim to have completely accomplished that, but nonetheless endeavored to achieve).

## 2 Design Decisions

During the redesign of *Flat Hunt* I stumbled upon several tricky issues that had to be dealt with. The more memorable and important ones are described in this chapter.

### 2.1 Singleton Scenes vs. `last_scene` vs. no such thing

**Singleton scenes:** A class `SHARED_SCENES` would provide singleton access to all necessary scenes of the game.

**`last_scene`:** Each scene in the game would have an attribute `last_scene` with which you could return to the previously run scene. This obviously only works if the scenes' order of appearance has a tree structure.

**No such thing:** You have none of the above. Everytime you change to another scene, a new scene is created.

At the very beginning I was going to use singleton scenes. The main problem with this was, that when you switch to another scene and then come back to a scene already run once, its events are trying to initialize a second time which leads to assertion violations. I tried to overcome this obstacle, but to no avail. Anyway, this problem was also the reason for dropping the `last_scene` approach and finally going back to our good old "create-scenes-like-crazy"-buddy. Whereas the "like-crazy" part might be just slightly over the top, since in *Flat Hunt* you really only have three scenes and you switch very rarely. These thoughts are the reason I finally decided to be old-fashioned for once and go with the last approach.

## 2.2 Menu Design

Basically there were three options I took into consideration:

**1.** Having a class `MENU` which contains a list of strings (`EM_STRING`s to be exact, but could also be generalized to be `EM_DRAWABLE`s) that represent the entries. Along with that you would have to store which entry is currently selected. Then you would also need an `on_select` - procedure that makes a case distinction based on `selected_entry` and reacts accordingly. The code could look like this:

```
class
    MENU
...
feature
    entries : ARRAYED_LIST [EM_DRAWABLE]

    selected_entry : INTEGER
...
```

```
class
    MENU_SCENE
...
feature
    menu: MENU
    on_select is
        do
            if menu.selected_entry = 1 then
                -- Do what entry 1 says, e.g. start a new game.
            elseif menu.selected_entry = 2 then
                -- Do what entry 2 says, e.g. show the credits .
            elseif ...
                ...
            end
        end
...
```

**2.** Pretty much the same setup as in case 1, but additionally class `MENU` would contain a list of agents with each agent corresponding to a menu entry. And `on_select` were to be moved from `MENU_SCENE` to `MENU`.

```
class
    MENU
...
feature
    entries : ARRAYED_LIST [EM_DRAWABLE]

    selected_entry : INTEGER

    agents: ARRAYED_LIST [PROCEDURE [ANY, TUPLE]]

    on_select is
        do
            agents.i_th ( selected_entry ). call ([])
        end
...
```

```
class
    MENU_SCENE
...
feature
    menu: MENU

    make is
        do
```

```
            create menu.make
            menu.agents. extend (agent agent1)
            menu.agents. extend (agent agent2)
            ...
        end

    agent1 is
        do
            -- Do what entry 1 says, e.g. start a new game.
        end

    agent2 is
        do
            -- Do what entry 2 says, e.g. show the credits .
        end
...
```

**3.** Case 2 directly leads to a more beautiful solution: Having a class `MENU` which is an `EM_DRAWABLE_CONTAINER` that contains all menu entries of type `MENU_ENTRY`, whereas each menu entry has its callback (i.e. agent) as an attribute. The beauty of this approach is its pure object-orientedness as opposed to the previous two.

```
class
    MENU_ENTRY
inherit
    EM_DRAWABLE_CONTAINER [EM_DRAWABLE]
...
feature
    callback : PROCEDURE [ANY, TUPLE]

    text : EM_STRING

    call is
        do
            if callback /= Void then
                callback . call ([])
            end
        end
...
```

```
class
    MENU
inherit
    EM_DRAWABLE_CONTAINER [MENU_ENTRY]
...
feature
    selected_entry : INTEGER

    add_entry ( a_text : STRING;
    a_callback : PROCEDURE [ANY, TUPLE]) is
        local
            a_menu_entry: MENU_ENTRY
        do
            create a_menu_entry.make_with_text ( a_text )
            a_menu_entry. set_callback ( a_callback )
            extend (a_menu_entry)
        end

    on_select is
        do
            item ( selected_entry ). call
        end
...
```

```
class
    MENU_SCENE
...
feature
    menu: MENU

    make is
        do
            create menu.make
            menu.add_entry (''Entry 1'', agent agent1)
            menu.add_entry (''Entry 2'', agent agent2)
             ...
        end

    agent1 is
        do
```

```
                -- Do what entry 1 says, e.g. start a new game.
            end

        agent2 is
            do
                -- Do what entry 2 says, e.g. show the credits .
            end
    ...
```

Originally I implemented the first version, albeit it is the most abominable one, because it seemed to me to be the straight-forward approach (I don't have a very strong object-oriented programming background - or now it is perhaps more appropriate to say I didn't have, because I learned an awful lot by working on this project). It didn't take me very long to notice that this was ugly, so I thought of other solutions and came up with the second and third option. Frankly, I wasn't so sure at this point which one to use, but after discussing the matter with Michela, I finally opted for the third one (which now appears to me should have been the logical choice from the start - but that's just the beauty of hindsight, I guess. . . ).

### 2.3 Main Controller Necessary?

In the old version of *Flat Hunt* a `MAIN_CONTROLLER` controlled how the game logic and the visualization worked together. I was not so sure if the main controller was really a necessity, because it seemed to be just as reasonable and also easier to implement if the visualization was directly in contact with the logic behind and got the information on what to display when from there. And also because a `EM_SCENE` is not strictly a visualization tool but includes an event loop, which means it is some kind of visualization / control hybrid. So, did I really need some third party controlling unit if my scene can already take care of that?
After some contemplation I came to realize that I did indeed need just that, in order to maintain a clear distinction of the *View* and *Controller* clusters (see subsection B.2), i.e. between visualization and control.

### 2.4 `PLAYER` and `PLAYER_DISPLAYER`

I had a pretty rough time figuring out how exactly to handle this separation. The main problem was not dividing the model and the view features but more about the question "Who is in charge?". I didn't want to have a two-way dependency, because I was told that was bad design. So I could either have a player which has a player displayer as an attribute or the other way around. The logical choice would be to have a player displayer with a player as an attribute (which I also chose in the end), because the player displayer visualizes a player and therefore has to know him. In my first design though, I couldn't eliminate the need for the player to know the displayer. That was until I discovered the *draw* feature of `EM_DRAWABLE`s. My original displayer was an `EM_DRAWABLE_CONTAINER [EM_DRAWABLE]`, and there you don't necessarily have to write your own *draw* procedure (you can, though), you just throw everything that needs to be drawn in the container. The problem with this is that you can't make conditional draws, meaning you have to tell the displayer from the outside when to draw what, so the player needed to inform the displayer about what he wants to get drawn and what not. As I said, my attempts to eliminate the two-way dependency this way failed miserably, so I thought I'd try it the other way around - which obviously was another failure. So then I did a little more research into *EM* and discovered above-mentioned *draw* feature, which worked like a charm and finally allowed me to have my desired one-way dependency.

### 2.5 Necessity of `FLAT_HUNT_SCENE`

A `FLAT_HUNT_SCENE` inherits directly from `EM_SCENE`. This class was my prototype for a scene which supports the `last_scene` approach described in subsection 2.1, but since I decided against that option, this scene was useless for some time. Until, one day, I implemented a simple music player for *Flat Hunt* and wanted it to be a shared music player, so that the songs play on as scenes are switched, and the controls are the same in every scene. That literally called for a `FLAT_HUNT_SCENE`, which is why I dug it out again but changed it profoundly. What is left is a scene that supports a shared music player and its controls. And because that is exactly what I wanted, `FLAT_HUNT_SCENE` is a survivor after all…

## 3 ESDL Extensions

Since I initially wanted to do my semester thesis on *ESDL*, but then due to several circumstances ended up doing it on *Flat Hunt*, I wanted to keep the option open to simultaneously develop *ESDL* if needed for my project. So I named my thesis "Flat Hunt Redesign and ESDL Extensions".
As I made progress with my work it turned out that *ESDL* already sufficed for *Flat Hunt* in virtually every aspect. And then *ESDL* was transformed into a new project called *EiffelMedia* (*EM*) which would be aggressively developed over summer by a number of people.
So basically what I did with *ESDL* was use it, rather than extend it (except for

fixing minor bugs which I ran across, but that does not count as an extension in my opinion).
In the end, I was relieved that *ESDL*, or now *EM*, already had such good support for everything I needed, because redesigning *Flat Hunt* proved to be much more elaborate than I had imagined.

## 4 Assigment Suggestions

I tried to maintain most of the old exercises as well as give my own suggestions, which was quite difficult. But I hope some of them are usable nonetheless.

### 4.1 Feature calls

In class START there is a feature *start* in which the students can place calls to make game settings and start the game.

```
start is
        -- Adjust the game settings and start the game.
    do
--      POSSIBLE ASSIGNMENT (sample solution):
--      game.set_map ("./map/zurich_little.xml")
--      game.set_game_mode (Versus)
--      game.set_number_of_hunters (3)
        -- If 'start_game' is left out, you will be stuck on the
        -- start menu scene, because no game will be started,
        -- even if you hit enter on "start game".
        start_game
    end
```

### 4.2 Conditional Statements

Give the students the class PLAYER but with

```
enough_tickets (a_type: TRAFFIC_TYPE): BOOLEAN is
        -- Check if player has tickets to drive
        -- with the transportation type 'a_type'.
    require
        a_type_valid: a_type /= Void and then is_valid_type (a_type)
    do
        if a_type.name @ (1) = 'b' then
```

```
            if bus_tickets > 0 then
                Result := True
            end
        elseif a_type.name @ (1) = 'r' then
            if rail_tickets > 0 then
                Result := True
            end
        elseif a_type.name @ (1) = 't' then
            if tram_tickets > 0 then
                Result := True
            end
        else
            Result := False
        end
    end
```

instead of the current *enough_tickets* and tell them to make this procedure a bit more readable and maybe hint at the inspect-when construct. Their goal would be to achieve the current *enough_tickets* routine (naturally this class should be changed before the students get it, otherwise it will be just a copy-paste exercise for them).

## 4.3 Contracts

Remove all contracts from some given class, and let the students fill them in.

## 4.4 Loops

In class PLAYER_DISPLAYER you'll find the feature *mark_defeat* with an empty loop that the students get to fill (similar to the old exercise, but without animation, because an animation would indeed have been easily realizable with an EM_ANIMATABLE but that would have eliminated the need for a loop and hence make the whole procedure pointless for this exercise).

```
mark_defeat (a_surface: EM_SURFACE) is
        -- Mark the defeat of the player.
    require
        a_surface_exists : a_surface /= Void
    local
        circle : EM_CIRCLE
        position : EM_VECTOR_2D
        count: INTEGER
    do
        -- Build ' circle ' at ' position '.
        create position .make (player. location . position .x,
        player. location . position .y)
        create circle .make_inside_box ( picture .bounding_box)
        circle . set_line_color (white)
        circle . set_filled (False)
        circle . set_line_width (2)
        circle .draw (a_surface)

        -- POSSIBLE ASSIGNMENT
        -- With instructions for the students
        -- and sample solution:
        from
                -- Fill
--                  count := 0
        until
                -- Replace 'True' and fill .
--                  count = 5
                True
        loop
                -- Fill
--                   circle . set_radius ( circle . radius + 5)
--                   circle .draw ( a_surface )
--                  count := count + 1
        end
    end
```

## 4.5 Inheritance

A few suggestions:

- Inherit from class `BRAIN`, redefine *choose_move* and implement an own artificial intelligence (for example a `DRUNKARD` like in last year's exercise).
- Inherit from class `ESTATE_AGENT_DISPLAYER`, redefine *draw* and for example implement an agent that shows himself in random rounds.
- Inherit from class `MENU`, effect *set_entry_position* to customize the menu layout and redefine *handle_key_down_event* to change the menu's behavior.

The first option is the most suitable in my opinion (it was also used in last year's exercise), but I wanted to throw my other ideas in anyway…

### 4.6 Events

Let the students study the class `BUTTON` or have them make class `TEXT_BOX` clickable.

## 5 Thanks

Thanks to…

**Michela Pedroni** for having been a great assistant who gave me a lot freedom in all aspects and was very supportive.

**My Predecessors** for their work.

**Till G. Bay** for ESDL support and the like.

**My Friends, Family and Cats** for motivating me and creating a pleasant working atmosphere.

## References

[1] Bertrand Meyer. *The Outside-In Method of Teaching Introductory Programming*. 2003.
http://www.inf.ethz.ch/~meyer/publications/teaching/teaching-psi.pdf

[2] Michela Pedroni. *Teaching Introductory Programming with the Inverted Curriculum Approach*. ETH Zurich, 2003.
http://se.inf.ethz.ch/projects/michela_pedroni

[3] Roger Küng. *Touch User Guide*. ETH Zurich, 2005.
http://se.inf.ethz.ch/projects/roger_kueng

[4] Sibylle Aregger. *Redesign of the TRAFFIC library*. ETH Zurich, 2005.
http://se.inf.ethz.ch/projects/sibylle_aregger

[5] Rolf Bruderer. *Object-Oriented Framework for Teaching Introductory Programming*. ETH Zurich, 2005.
http://se.inf.ethz.ch/projects/rolf_bruderer

[6] Marcel Kessler. *Exercise Design for Introductory Programming. "Learn-by-doing" basic OO-concepts using Inverted Curriculum*. ETH Zurich, 2004.
http://se.inf.ethz.ch/projects/marcel_kessler

[7] Benno Baumgartner. *ESDL - Eiffel Simple Direct Media Library*. ETH Zurich, 2004.
http://se.inf.ethz.ch/projects/benno_baumgartner

[8] Till G. Bay. *Eiffel SDL Multimedia Library (ESDL)*. ETH Zurich, 2003.
http://se.inf.ethz.ch/projects/till_bay

# A Flat Hunt User Guide

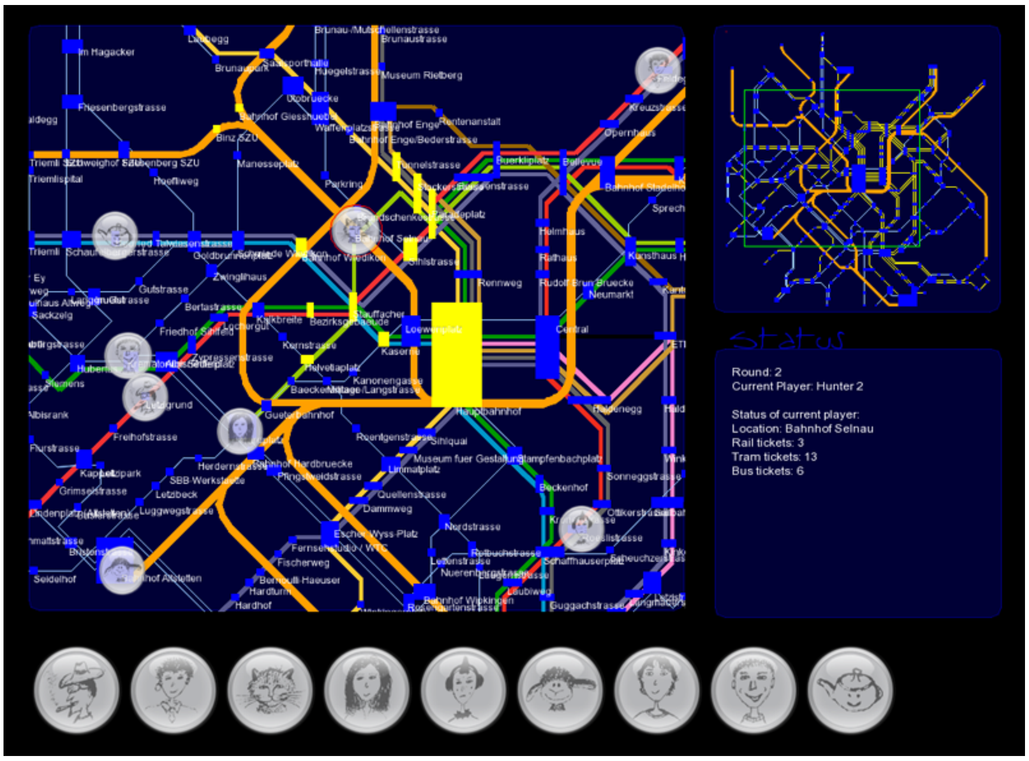
*Flat Hunt* is an application that is used to teach you programming, along with another application named *Touch* [1]. *Flat Hunt* is a "Scotland Yard"-like game that will mainly appear in the assignments for the Introduction to Programming course. It is based on *TRAFFIC* [2] for modeling the city where the game takes place and on *EiffelMedia* (formerly known as *ESDL* [3]) for visualization.

This document describes how to use *Flat Hunt*.

## A.1 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…



```
Round: 2
Current Player: Hunter 2

Status of current player:
Location: Bahnhof Selnau
Rail tickets: 3
Tram tickets: 13
Bus tickets: 6
```

Figure 1: Screenshot of *Flat Hunt* in action

## A.2 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, who is renting flats. The problem is that he is always busy showing flats to other customers, 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 VBZ-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 (Figure 2) calls his office to tell the secretary which flat he is currently visiting. So sometimes, the people there in the office can tell the flat hunters (Figure 3) where to look for him. . .

Figure 2: Estate agent

Figure 3: Flat hunters

## A.3 Getting Started

*What you need for running* Flat Hunt. . .

### A.3.1 Requirements

**EiffelStudio:** `http://www.eiffel.com/downloads/`

**TRAFFIC:** `http://se.inf.ethz.ch/traffic/`

**EiffelMedia:** `http://se.inf.ethz.ch/eiffelmedia/`

### A.3.2 Installation

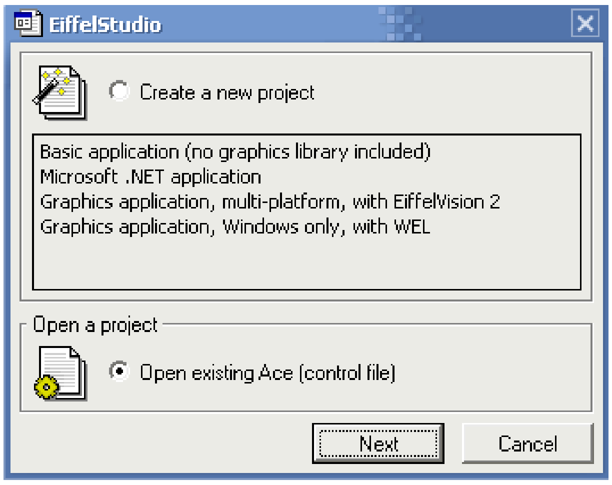
*EiffelStudio*, as well as *EiffelMedia*, come with an installer. Just follow the on-screen instructions like you would when installing any other program. No magic there..

*TRAFFIC* does not need to be installed, just download the zip-file and unzip it to a directory of your choice.
Flat Hunt is located in the directory `traffic/example/flat_hunt`. To get it to run, however, you'll have to compile it first.

For that you have to complete the following steps:

1. Start EiffelStudio

2. Click on "File ->New Project...". Choose "Open existing Ace (control file)" from the dialog (see Figure 4) and click on "Next".

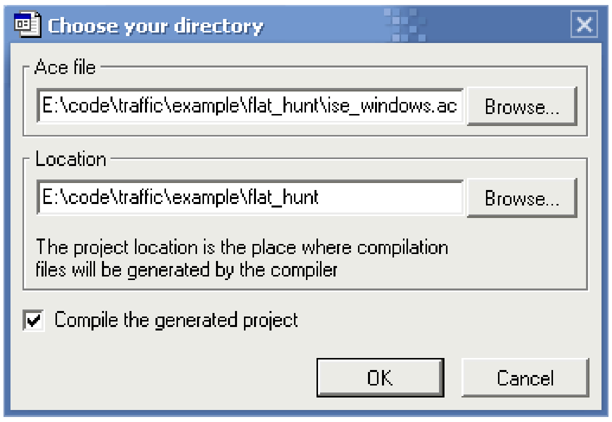


Basic application (no graphics library included)
Microsoft .NET application
Graphics application, multi-platform, with EiffelVision 2
Graphics application, Windows only, with WEL

Figure 4: New Project Dialog

3. This will open a file dialog that lets you choose the Ace file. Browse to the directory `traffic/example/flat_hunt`. Depending on the operating system you are working on, choose *ise_windows.ace* or *ise_linux.ace*. Click on "Open".

4. The dialog shown in Figure 5 lets you choose the project directory. In most cases you can leave both paths (Ace file and location) as EiffelStudio proposes. Make sure that the checkbox for compiling the generated project is selected. Click on "OK". This will start the compilation of the project.



Ace file: E:\code\traffic\example\flat_hunt\ise_windows.ace

Location: E:\code\traffic\example\flat_hunt

The project location is the place where compilation files will be generated by the compiler

☑ Compile the generated project

Figure 5: Project Directory Dialog

5. Once the project is compiled you can execute it by clicking on the "Launch" button in EiffelStudio or by hitting **F5**.

Now you are ready for playing Flat Hunt. Enjoy…

## A.4 Gameplay

*Playing Flat Hunt is not very difficult, especially for those that know the game "Scotland Yard"…*

### A.4.1 General Rules

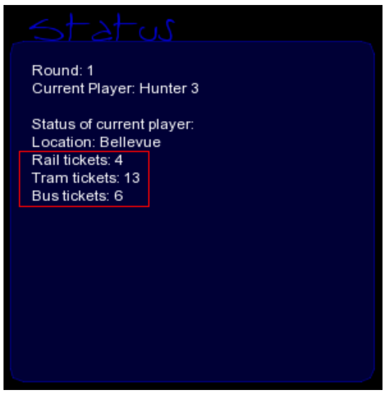
The game lasts for at most 23 rounds. In these 23 rounds, the flat hunters try to find the estate agent, while he tries to avoid them (this is because he would rather rent the flats to elderly couples, since presumably they make fewer parties in the middle of the night…).

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

- one or two stops by tram (colored lines),
- one stop by train (thick orange lines),
- or one stop by bus (thin light blue lines).

A move with a certain transport can only be made if one has still enough tickets (see Figure 6), 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).

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…



Status

Round: 1
Current Player: Hunter 3

Status of current player:
Location: Bellevue
Rail tickets: 4
Tram tickets: 13
Bus tickets: 6

Figure 6: Ticket status

The possible places you can move to are colored yellow (see Figure 7). 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 status box at the right, the game status and information about the current player get displayed. If you want to know the status of another player just click on his picture at the bottom. Click again to close the just opened status box.

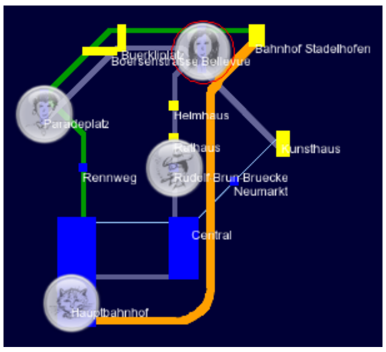


Figure 7: Highlighted places

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.

### A.4.2 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 agent shows himself only in rounds number 1, 3, 8, 13, 18, and 23. In these rounds, the exact route of the agent is displayed under *History* in the status box at the bottom right corner and in the estate agent's own status box if opened. In all other rounds you only see the detailed history up to the round the estate agent last showed himself. As soon as the agent has come out of hiding for the first time, a dimmed version of his picture will always be shown at the location he was last sighted.

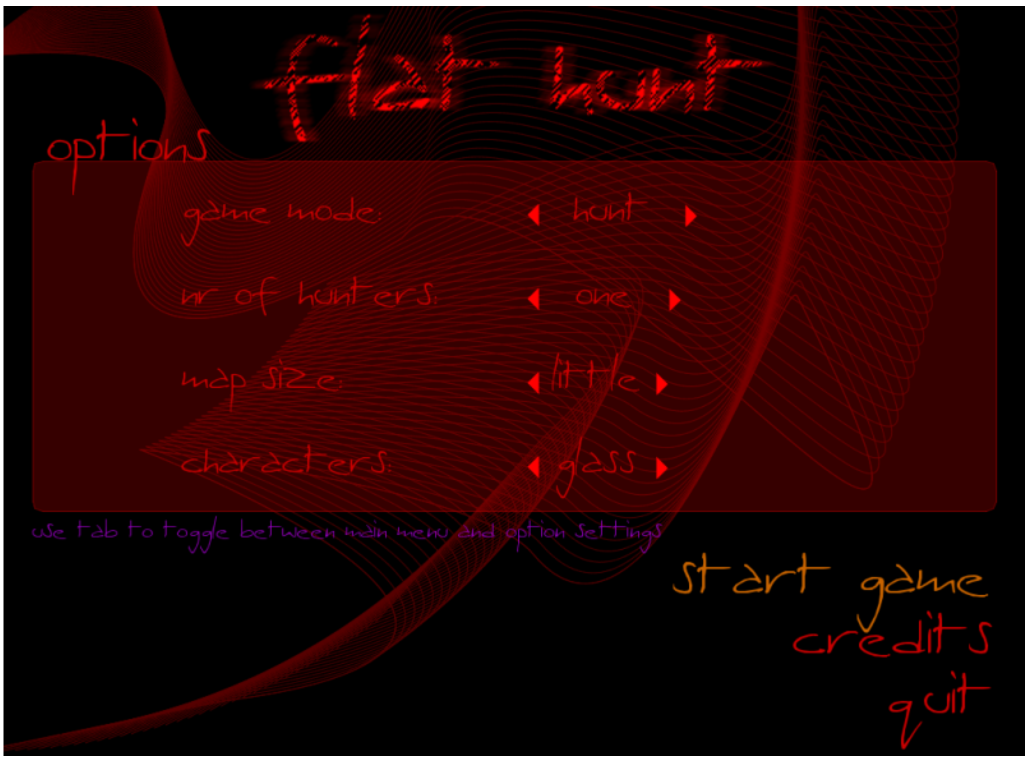
**Escape** This is the exact opposite of *Hunt* mode: The agent is played by you, and the hunters are played by the computer. The hunters always move as close in your direction as possible, as they somehow manage to decode your transponder signal, and thus always know your precise location (so much for 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.

### A.4.3 Other

When you run *Flat Hunt*, the first you'll see is a menu (see Figure 8). You can either let the default options in place and just select *start game* or you can adjust the settings to your needs. *Game mode* is explained in subsubsection A.4.2, *number of hunters* and *map size* should be self-explanatory and *characters* specifies which pictures to use for the players. To toggle between the settings menu and the normal menu press *tab*.



game mode: hunt
nr of hunters: one
map size: little
characters: glass

use tab to toggle between main menu and option settings

Figure 8: Screenshot of the start menu

During the game when you press *p*, the pause menu is shown. *Continue* makes the pause menu disappear and lets you resume the game, *new game* takes you to the start menu and *quit* quits the application.

The game over menu is similar to the pause menu, only there is no *continue* in this one.

## A.5 Special Features

### A.5.1 Map Control

Map control is fairly simple: you got two maps in a game scene, one of which only is a smaller version of the other one. The big map is on the left and that's where the action takes place, the little map on the right is meerely a navigation tool. To control the big map, use your mouse as follows:

**left click:** only has an impact if clicked on a highlighted place

**right button down + move mouse:** moves map in the direction of your mouse movement

**middle button down + move mouse up:** zoom in

**middle button down + move mouse down:** zoom out

When you **left click + move mouse** in the little map, a red rectangle is drawn between the point where your left mouse button is pressed down and the point when its released. As soon as you release the mouse button, the map segment that is inside this rectangle gets displayed on the big map.

### A.5.2 Music Player

*Flat Hunt* comes with an integrated music player and some default background music. Since not everyone likes the same sound, there is also the possibility to play your own.
Just put your `.ogg`-files in the directory `${FLAT_HUNT}/resources/sound` before you start the Flat Hunt application. Flat Hunt will then automatically load all the `.ogg`-files from this directory and play them in alphabetical order (unless you enable shuffle, obviously). Music player control: see subsubsection A.5.3.

### A.5.3 Keyboard Shortcuts

During the game, the following shortcuts are available:

**p:** pause the game and show pause menu

**s:** music player toggle shuffle

**v:** music player decrease volume

**shift + v:** music player increase volume (at startup the volume is already at maximum)

**page up:** music player next song

**page down:** music player previous song

## A.6 Legal Stuff and Thanks

This document is based upon its prior version, which was written by Michela Pedroni and Marcel Kessler (thanks!). All graphics for the game were designed by me and Photoshop.

Thanks to Michela Pedroni for her assistance, all my predecessors for their work, Till G. Bay (and others) for the *EiffelMedia* (formerly *ESDL*) Library and Bertrand Meyer for the *Eiffel* language.

# References

[1] Roger Küng. *Touch User Guide*. ETH Zurich, 2005.
`http://se.inf.ethz.ch/projects/roger_kueng`

[2] Sibylle Aregger. *Redesign of the TRAFFIC library*. ETH Zurich, 2005.
`http://se.inf.ethz.ch/projects/sibylle_aregger`

[3] Till G. Bay. *Eiffel SDL Multimedia Library (ESDL)*. ETH Zurich, 2003.
`http://se.inf.ethz.ch/projects/till_bay`

## B Flat Hunt Developer Guide

*TRAFFIC* [2], *Touch* [1] and *Flat Hunt* is software that hopefully makes learning to program more fun and more interesting for you. *TRAFFIC* is a library that supports the reading and display of public transportation systems. A library is a piece of software whose functionality can be used by other software. *Flat Hunt* is an application that uses the *TRAFFIC* library to model a city map. For the visualization the *EiffelMedia* Library (formerly known as *ESDL* [6][5]) is used. *Flat Hunt* is a strategy game, similar to Scotland Yard, but with a different background story (for more information about the story and gameplay of *Flat Hunt*, read the Flat Hunt User Guide). *Touch* is also an application that uses the *TRAFFIC* library. For more information on *TRAFFIC* and *Touch*: see [2] and [1] or visit this website: http://se.inf.ethz.ch/traffic.

This document describes how *Flat Hunt* is built, what classes are important and highlights some of their features.

## B.1 Getting Started

*What you need for running* Flat Hunt…

Please refer to the *Flat Hunt User Guide* for detailed instructions on how to get *Flat Hunt* to run.

## B.2 Design

### B.2.1 Overview

When opening *Flat Hunt* in EiffelStudio, the cluster view in the bottom left corner of EiffelStudio shows many clusters. For you only the top-level clusters *Traffic* and *Flat_hunt* are important.

To remove complexity, *Flat Hunt* is structured in four top-level clusters (see Figure 9): *Model*, *View*, *Controller* and *Util*. Some clusters contain sub-clusters and in each cluster there are several classes.

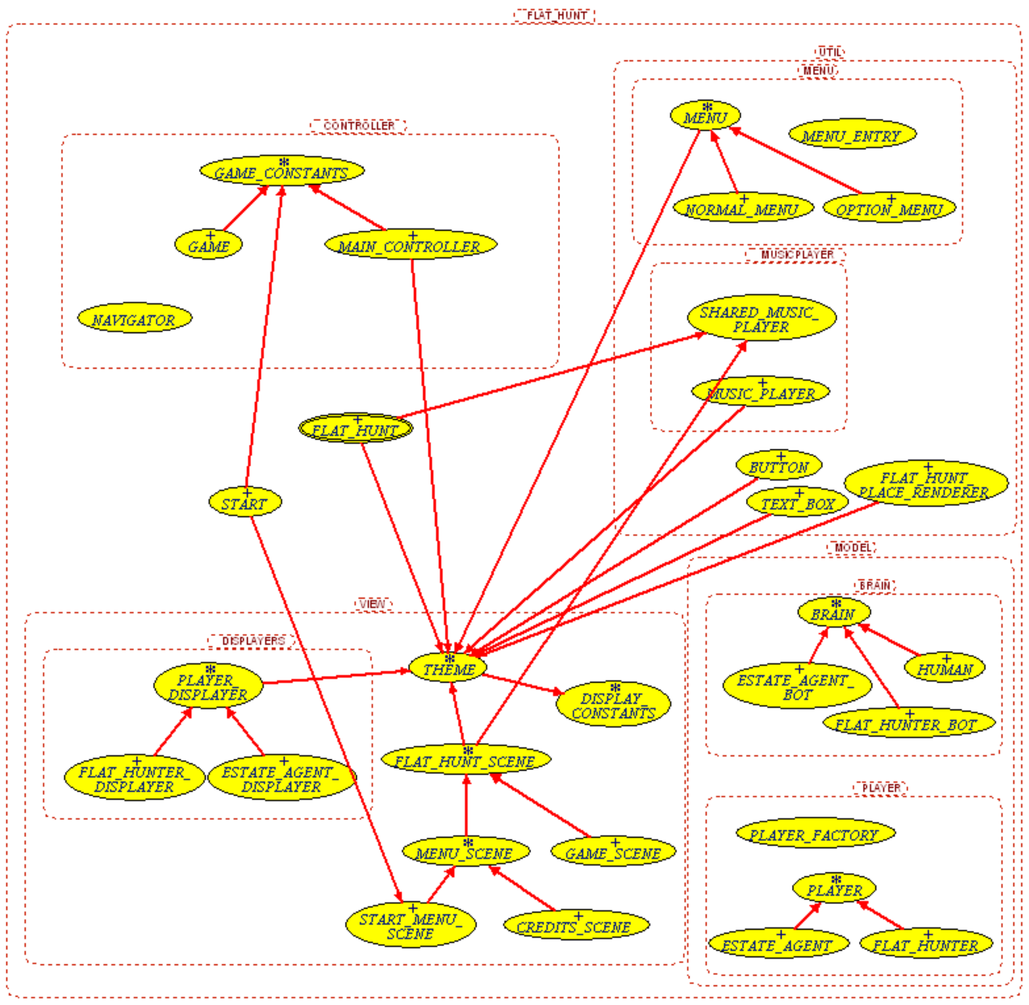


Figure 9: Flat Hunt Clusters (**Note:** The client-supplier relationship arrows are omitted for the sake of overview.)

### B.2.2 Controller cluster

Cluster **Controller** is the fundamental cluster in *Flat Hunt*. Here are the classes that "control" the actions. 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 MAIN_CONTROLLER controls the display update by calling *game_scene.center_on_player (game.current_player)*.

- **MAIN_CONTROLLER**: The MAIN_CONTROLLER is (as the name suggests) responsible for many things. It provides access to the GAME_SCENE, to class GAME and to the whole *TRAFFIC* library, which is responsible for the visualization of the map.

- **GAME**: Class GAME features the game logic. It knows which player's turn it is, and also, since it is an heir of GAME_CONSTANTS, what state the game is in.

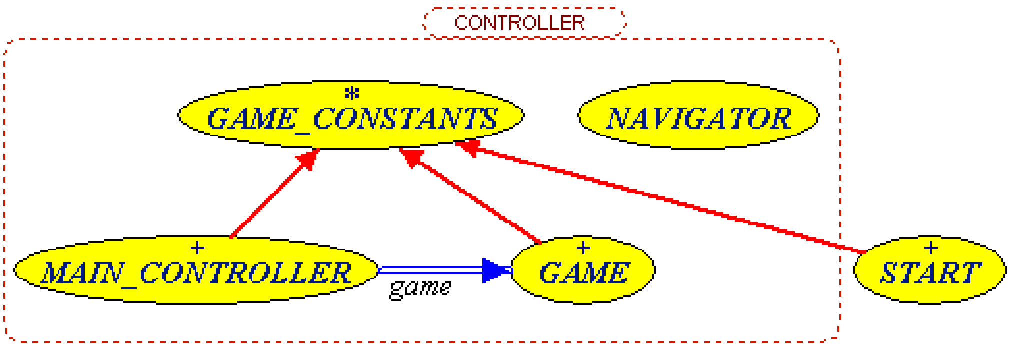


Figure 10: Diagram of the Controller Cluster

### B.2.3 Model cluster

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, FLAT_HUNTER_BOT and ESTATE_AGENT_BOT. These **Model** classes describe the internal representation of "real world" objects. Here is a description of some of these classes.

- **PLAYER**: Class PLAYER knows the basic things one needs to know about a player of *Flat Hunt*, like how many tickets he got left. It features the commands *play* and *move* and has either a HUMAN or a BOT brain.

- **ESTATE_AGENT**: 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**: Class BRAIN includes the intelligence to choose the next move.

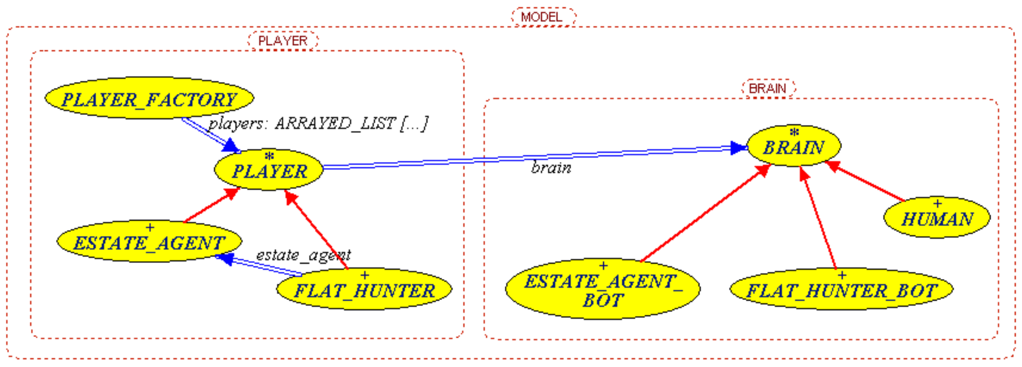


Figure 11: Diagram of the Model Cluster

### B.2.4 View cluster

This cluster's job is to make sure that the user sees what is going on. It includes all scenes and menus, as well as displayers for the game players and status information.

- **PLAYER_DISPLAYER**: 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.
- **GAME_SCENE**: Contains all the drawables of the current game scene and displays them.

### B.2.5 Util Cluster

Those classes that are not directly part of the game, but rather serve as utils, reside in the **Util** cluster. For one, there are several menu handling classes, which provide the functionality for a normal menu and an option menu. Also important are the helper classes like TEXT_BOX, which allows to comfortably display status messages in a nice translucent box. And last but not least, a basic music player with shuffle function can be found here.

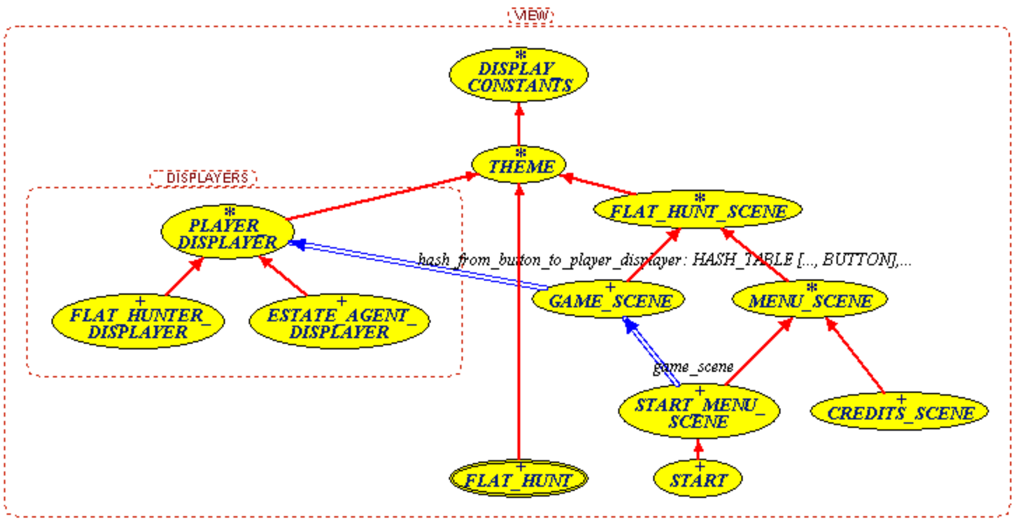


Figure 12: Diagram of the View Cluster

## B.3 The States of the Game

### B.3.1 Overview

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 x). These game states are defined in class `GAME_CONSTANTS`:

```
Agent_stuck, Agent_caught, Agent_escapes, Prepare_state , Play_state ,
Move_state: INTEGER is unique
        -- Possible states of the game.
```

### B.3.2 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_escaped`.

**Prepare** If the game is in this state, 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`). If that is not the case, the game goes in state `Play`.

**Play** In this state, 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

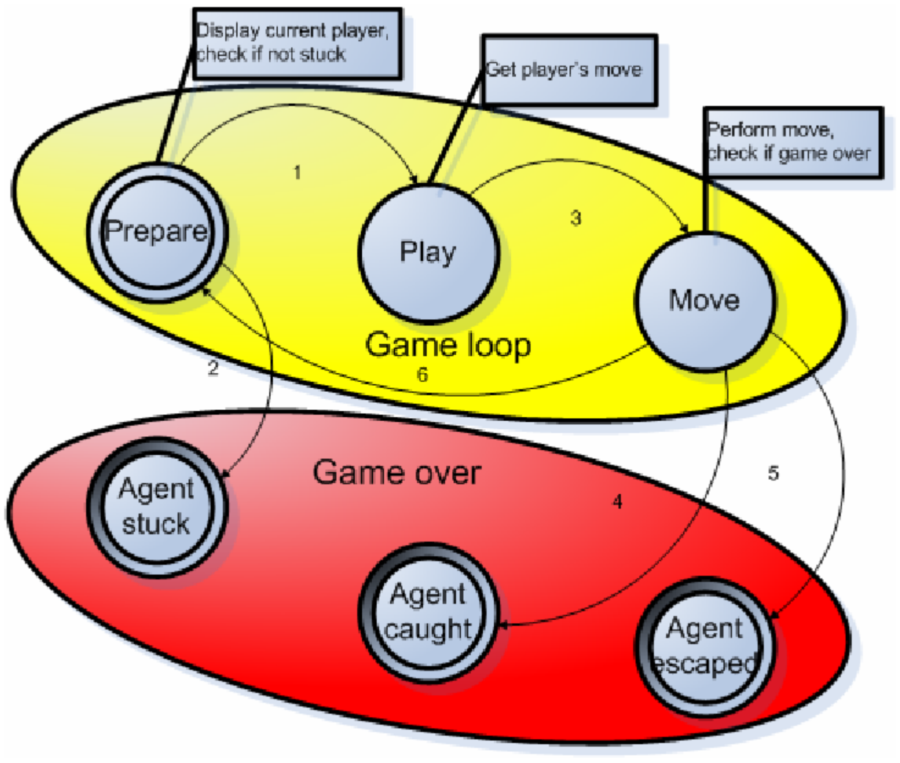


Figure 13: Game states and loop

player is controlled by an artificial intelligence, then the best of the possible moves is calculated. The game then goes in state `Move`.

**Move** In this state, the move selected in state `Play` is performed. 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`. 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`. 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`.

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
```

```
            if current_player = estate_agent then
                update_agent_visibility
            end
            current_player.move
            if current_player.location = estate_agent.location
            and current_player /= estate_agent then
                state := Agent_caught
                update_agent_visibility
            else
                state := Prepare_state
            next_turn
        end
end
```

## B.4 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. At the very beginning, the application has to be launched. By calling *make_and_launch* of the root class `FLAT_HUNT` exactly that is achieved. This feature sets the application name, resolution and several other options and then launches the first scene to be displayed, which is of type `START` and is an heir of `START_MENU_SCENE`.

2. When "start game" is selected in this scene, *start_callback* is called and creates a game with the proper settings and a game scene, whose job it is to visualize the game. *start_callback* also creates the `MAIN_CONTROLLER` and calls *main_controller.start_game*.

3. *start_game* in class `MAIN_CONTROLLER` calls *create_players* of class `GAME` which creates the players using class `PLAYER_FACTORY`. Then it calls *start_game* of class `GAME` which sets the game state to `Prepare_state` and starts the game.

4. In class `PLAYER_FACTORY`, for example the estate agent is created using *estate_agent.make* in feature *build_players*.

5. This creates a `HUMAN`, `FLAT_HUNTER_BOT` or `ESTATE_AGENT_BOT` brain depending on the value of `flat_hunters_bot` or

`estate_agent_bot` respectively, which are boolean values to indicate if a human or the computer is going to play the corresponding player(s).

6. 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*...

7. ... which centers the city map on `game.current_player` and then calls *game.prepare*.

8. *prepare* of class `GAME` first calculates the estate agent's possible moves. If there are no possible moves (i.e. `current_player.possible_moves.is_empty`) then it's either the next player's turn or the state is set to `Agent_stuck`. Otherwise the game state is set to `Play_state`.

9. With that, the call to *prepare* (Step 6) comes to an end and control goes back to feature *idle_action* of class `MAIN_CONTROLLER`. According to the present game state, *idle_action* will now call *play* which then calls *game.play*.

10. This calls *current_player.play (selected_place)*, where `selected_place` is the last place the user clicked on. `selected_place` is then passed on to class `BRAIN`.

11. *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.

12. This calls *brain.choose_move*, where brain is either a `FLAT_HUNTER_BRAIN`, `ESTATE_AGENT_BRAIN` or `HUMAN`.

13. The next move is now chosen, and thus the player moves. Control goes back to *idle_action* and we are back at step 6.

## B.5 Legal Stuff and Thanks

This document is based upon its prior version, which was written by Michela Pedroni and Marcel Kessler (thanks!). All graphics for the game were designed by me and Photoshop. The code of *Flat Hunt* is based on its prior version [6], which is mainly the work of Marcel Kessler. Major parts had to be rewritten by me though.

Thanks to Michela Pedroni for her assistance, all my predecessors for their work, Till G. Bay (and others) for the *EiffelMedia* Library (formerly *ESDL* [6][5]) and Bertrand Meyer for the *Eiffel* language.

## References

[1] Roger Küng. *Touch User Guide*. ETH Zurich, 2005.
`http://se.inf.ethz.ch/projects/roger_kueng`

[2] Sibylle Aregger. *Redesign of the TRAFFIC library*. ETH Zurich, 2005.
`http://se.inf.ethz.ch/projects/sibylle_aregger`

[3] Rolf Bruderer. *Object-Oriented Framework for Teaching Introductory Programming*. ETH Zurich, 2005.
`http://se.inf.ethz.ch/projects/rolf_bruderer`

[4] Marcel Kessler. *Exercise Design for Introductory Programming. "Learn-by-doing" basic OO-concepts using Inverted Curriculum*. ETH Zurich, 2004.
`http://se.inf.ethz.ch/projects/marcel_kessler`

[5] Benno Baumgartner. *ESDL - Eiffel Simple Direct Media Library*. ETH Zurich, 2004.
`http://se.inf.ethz.ch/projects/benno_baumgartner`

[6] Till G. Bay. *Eiffel SDL Multimedia Library (ESDL)*. ETH Zurich, 2003.
`http://se.inf.ethz.ch/projects/till_bay`