Flat Hunt Developer Guide

Ursina Caluori
ucaluori@student.ethz.ch

September 26, 2005

Contents

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.

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.

2 Design

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 1): *Model*, *View*, *Controller* and *Util*. Some clusters contain sub-clusters and in each cluster there are several classes.

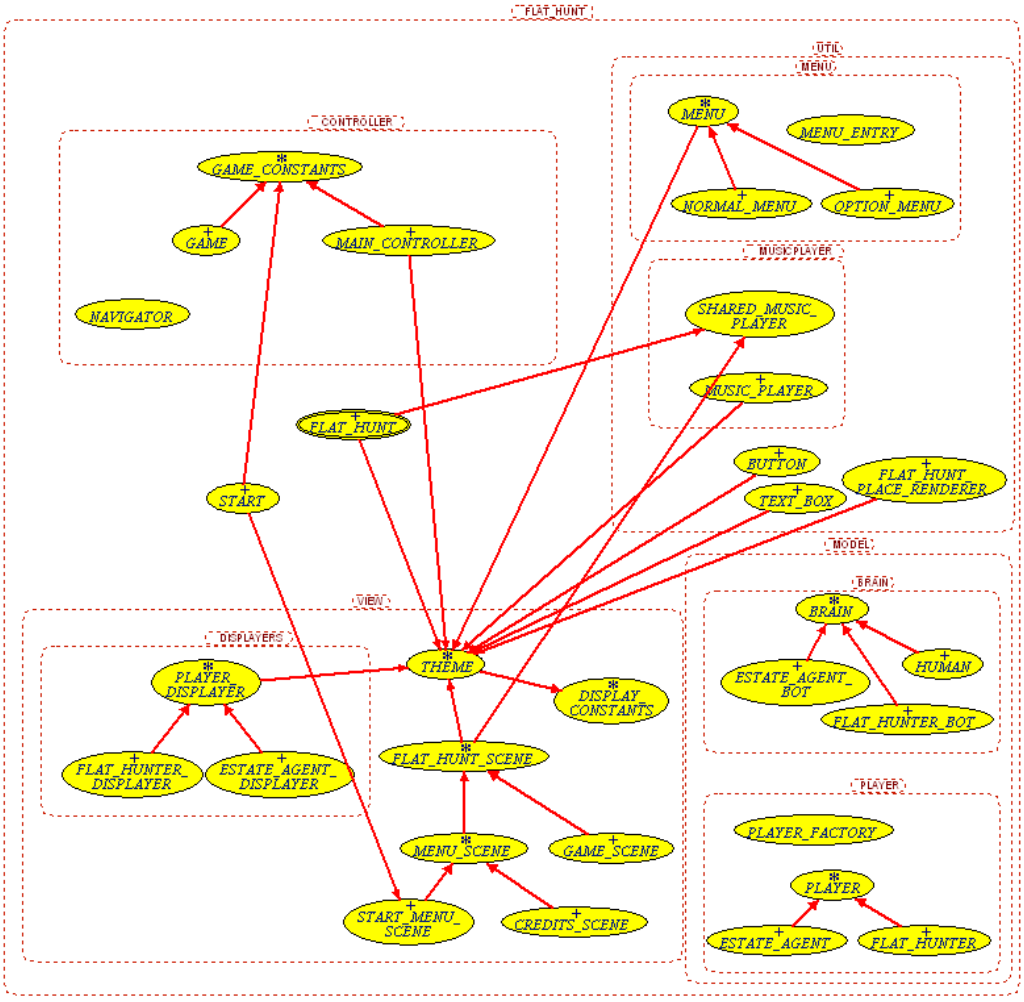

Figure 1: Flat Hunt Clusters (**Note:** The client-supplier relationship arrows are omitted for the sake of overview.)

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.

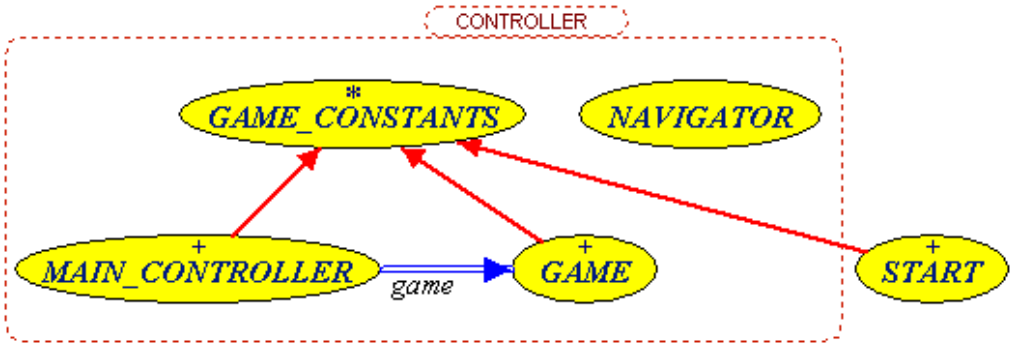

Figure 2: Diagram of the Controller Cluster

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.

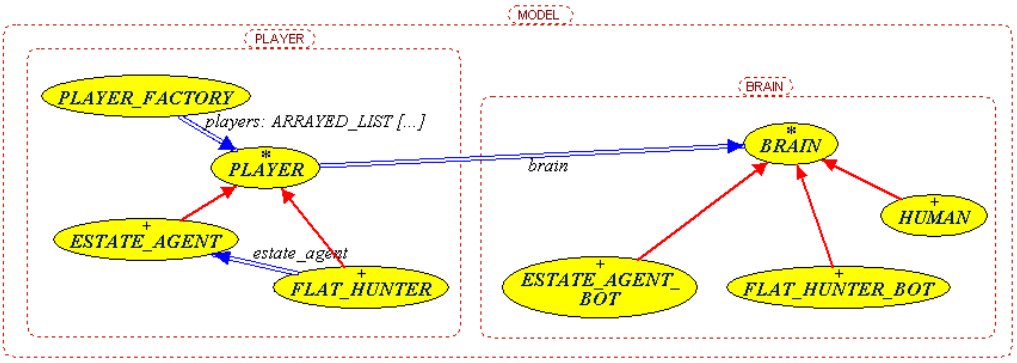

Figure 3: Diagram of the Model Cluster

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.

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.

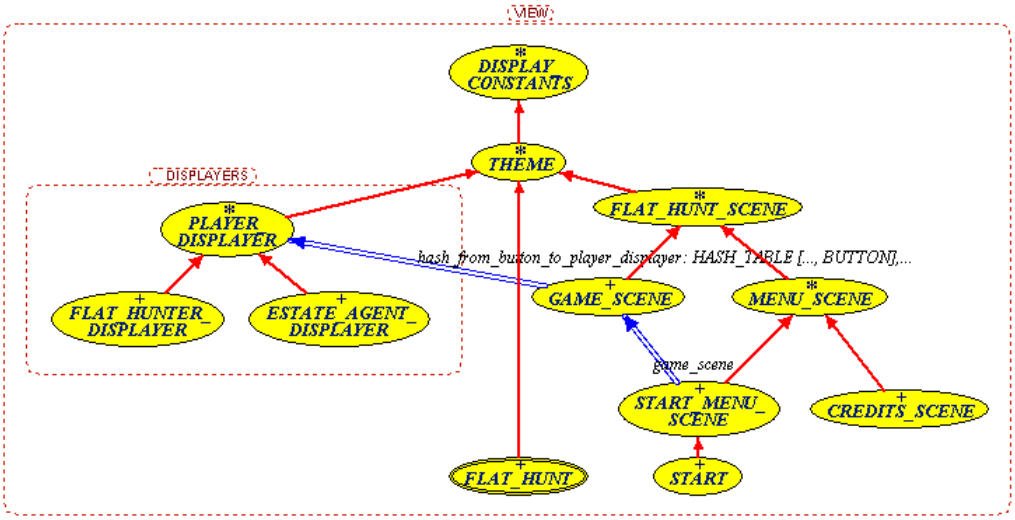

Figure 4: Diagram of the View Cluster

3 The States of the Game

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.

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`.

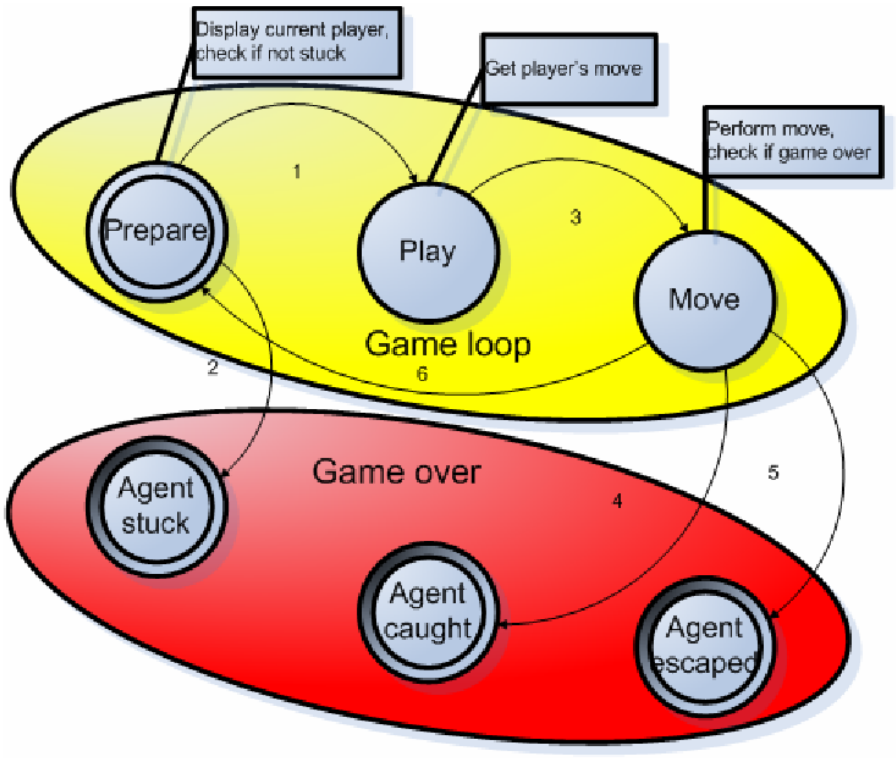

Figure 5: Game states and loop

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 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
```

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.

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 [**?**], 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