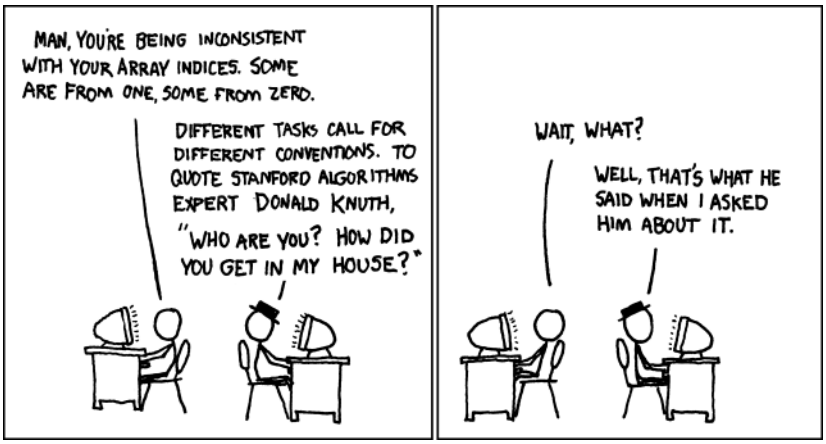
Assignment 10: Agents and board games

ETH Zurich

handout: 1 December 2014
Due: 17 December 2014



Donald Knuth © Randall Munroe (http://xkcd.com/163/)

Goals

- Practice creating and invoking agents.
- Finish the design and implementation of the board game.

1 Navigating in Zurich

In Traffic the elements of a city map can be configured to react to certain events. Each view class has, for example, an (inherited) attribute *on_left_click_no_args* of type *V_LIST* [*PROCEDURE* [*ANY*, *TUPLE*]]: a list of procedures without arguments. By default this list is empty; however if you add an agent to it, this agent will be executed every time a single left mouse button click is performed on the view. Similar lists are available for other events (right click, double click, drag, scroll). Moving the map by dragging and zooming it with the mouse wheel is implemented using this mechanism. In this task you will make the map react to other events in a certain way.

You are to implement a simple navigation application that helps the user find a way from one station in Zurich to another using public transportation. Our code skeleton provides a class *ROUTE_FINDER* with a query *shortest_route* that yields a shortest route between two stations. Your task is to provide the user interface for the application: namely, to allow the user to choose the two stations by clicking on them and then display the resulting route.

To do

1. Download http://se.inf.ethz.ch/courses/2014b_fall/eprog/assignments/10/traffic.zip, extract it and open `assignment_10.ecf`. Open class *NAVIGATOR*.

2. In the feature *add_event_handlers* add a reaction to the mouse button click events for the view of every station in Zurich, in order to achieve the following behavior:

   - left click on a station selects it as the route origin;
   - right click on a station selects it as the route destination;
   - once both origin and destination are selected, the shortest route is displayed on the map and directions are output to the console.

   Directions can be output, for example, in the following format:

   ```
   From Hardplatz to Bahnhof Wiedikon:
   Take tram 8 until Stauffacher
   Take tram 9 until Bahnhof Wiedikon
   ```

Hint: Note that *shortest_route* creates a new route but does not add it to Zurich.
Hint: The view of a station s is accessible through one of the following expressions: *Zurich_map.views* [s] or *Zurich_map.stations_view* (s).

To hand in

Hand in the code of *NAVIGATOR*.

2 Home automation

Imagine that you are participating in a project "Digital home": developing a home automation system, which lets users set up different reactions to certain events happening in their homes.

Your task is to implement a class *TEMPERATURE_SENSOR*, which manages reactions to changes in the room temperature. Assume that there is a hardware component that measures the temperature and calls the feature *set_temperature* of *TEMPERATURE_SENSOR* whenever a change occurs. You have to provide the interface for configuring the set of reactions, and make sure all the desired reactions are invoked in response to a temperature change.

Other project members have already implemented a couple of classes that provide two basic reactions to a temperature change. Class *DISPLAY* represents a digital LED display with a feature *show* (*a_temperature*: *DOUBLE*) that displays a temperature. Class *HEATING_CONTROLLER* provides a feature *adjust* (*a_temperature*: *REAL_64*) that turns heating and cooling on or off depending on the difference between the current and the goal temperature.

To do

1. Download http://se.inf.ethz.ch/courses/2014b_fall/eprog/assignments/10/digital_home.zip, extract it and open `digital_home.ecf`.

2. Create a new class *TEMPERATURE_SENSOR* with the following functionality:

   - It should store the current temperature and allow to modify it through a feature *set_temperature* (*a_temperature*: *REAL_64*).

- It should allow registering any number of observers: procedures that will be called every time the current temperature changes. **Hint:** those procedures can be defined in any class and take the new temperature as their single argument, thus the type of the corresponding agents is *PROCEDURE* [*ANY*, *TUPLE* [*REAL_64*]].
- It should allow removing observers that are already registered.
- It should be possible to register features *show* and *adjust* discussed above without changing the classes *DISPLAY* and *HEATING_CONTROLLER*.

3. Test your implementation of *TEMPERATURE_SENSOR* from within the class *APPLICATION*. Create objects of types *TEMPERATURE_SENSOR*, *DISPLAY* and *HEATING_CONTROLLER*; register features *show* and *adjust* as observers of the sensor. Make several calls to *set_temperature* on the sensor to emulate calls from the hardware component; as a result of each call the temperature should be displayed and the heating should be adjusted.

To hand in

Hand in the code of *TEMPERATURE_SENSOR* and *APPLICATION*.

3 The final project. Board game: part 4

For the final project you can do one of the following:

1. Finish the implementation of the board game following the specification given below.
2. Implement any game of your choice if your assistant agrees that the game proposed by you is suitable for the project.

If you choose option 1 you have to implement a simplified version of *Monopoly* (`http://en.wikipedia.org/wiki/Monopoly_(game)`), played by the following rules.

The game comes with a board divided into 20 squares (Figure 1), a pair of *four*-sided (tetrahedral) dice, and can accommodate 2 to 6 players.

It works as follows:

- Players have money and can own property. Each player starts with CHF 1500 and no property.
- All players start from the first square ("Go").
- One at a time, players take a turn: roll the dice and advance their respective tokens clockwise on the board. After reaching square 20 a token moves to square 1 again.
- Certain squares take effect on the player (see below), when his token passes or lands on the square. In particular it can change the player's amount of money.
- If after taking a turn a player has a negative amount of money he retires from the game. All his property becomes unowned.
- A round consists of all players taking their turns once.
- The game ends either if there is only one player left or after 100 rounds. The winner is the player with the most money after the end of the game. Ties (multiple winners) are possible.

There are the following kinds of squares on the board:

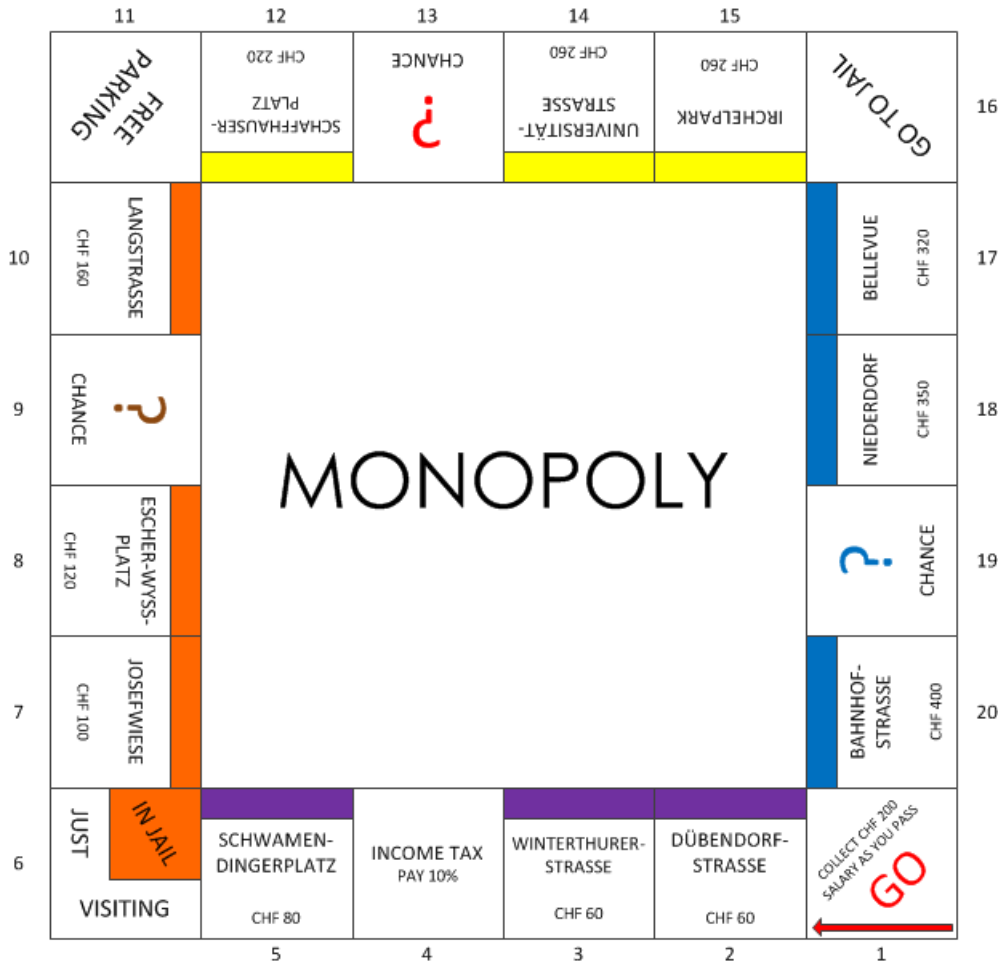


Figure 1: Monopoly board

Property squares (marked by a colored stripe). They contain the name and the price of the property and can be owned by players. If a player lands on an unowned property he can choose to buy it for the written price or do nothing. If a player lands on a property owned by another player he has to pay a rent (rent amounts are listed in Table 1).

Go. Every time a player passes through (not necessarily lands on) this square he gets CHF 200 salary.

Chance. If a player lands on one of these squares he either gains a random amount (multiple of 10) up to CHF 200 or looses a random amount (multiple of 10) up to CHF 300.

Income tax. If a player lands on this square he pays 10% of his money (rounded down to a multiple of 10) as tax.

Free parking. This square has no effect.

Go to Jail. If a player lands on this square he immediately goes to the "In Jail" part of the "In Jail/Just Visiting" square.

In Jail/Just Visiting. If a player lands on this square he is "Just Visiting": the square has no effect. However, if the player got here by landing on "Go to Jail", he is in Jail and cannot make a move. A player gets out of Jail by either throwing doubles[1] on any of his next three turns (if he succeeds in doing this he immediately moves forward by the number of spaces shown by his doubles throw) or paying a fine of CHF 50 before he rolls the dice on either of his next two turns. If the player does not throw doubles by his third turn he must pay the CHF 50 fine. He then gets out of Jail and immediately moves forward the number of spaces shown by his throw.

Table 1: Properties

Position	Name	Price	Rent
2	Dübendorfstrasse	60	2
3	Winterthurerstrasse	60	4
5	Schwamendingerplatz	80	4
7	Josefwiese	100	6
8	Escher-Wyss-Platz	120	8
10	Langstrasse	160	12
12	Schaffhauserplatz	220	18
14	Universitätstrasse	260	22
15	Irchelpark	260	22
17	Bellevue	320	28
18	Niederdorf	350	35
20	Bahnhofstrasse	400	50

To do

Implement the game using a command line user interface. Your program should ask for user input every time a player can decide whether to buy a property or to pay the fine to get out of Jail.

We recommend that you start from the master solution to assignment 7: `http://se.inf.ethz.ch/courses/2014b_fall/eprog/assignments/07/board_game_solution.zip`.

Optionally you can implement any extensions to the game, such as:

- other standard rules of Monopoly: property groups, auctions, improving property, mortgaging, etc. (see for example `http://richard_wilding.tripod.com/monorules.htm`);
- non-human players (your program will make decisions for some of the players instead of a human);
- graphical user interface.

To hand in

Hand in the code of your classes.

[1] When both dice come out the same face up.

4 MOOC: Selective exports, multiple inheritance, and agents

To do

1. Access the main MOOC course web page at http://se.ethz.ch/mooc/programming.
2. Listen to lecture number 12 "Selective exports and deferred classes" and take the corresponding quiz.
3. Listen to lecture number 15 "Multiple inheritance" and take the corresponding quiz.
4. Listen to lecture number 16 "Functional programming and agents" and take the corresponding quiz.

Your goal is to provide all correct answers to the quizzes. You can take the quizzes as many times as you want. If you succeed, you will be awarded a badge for each correctly solved quiz.