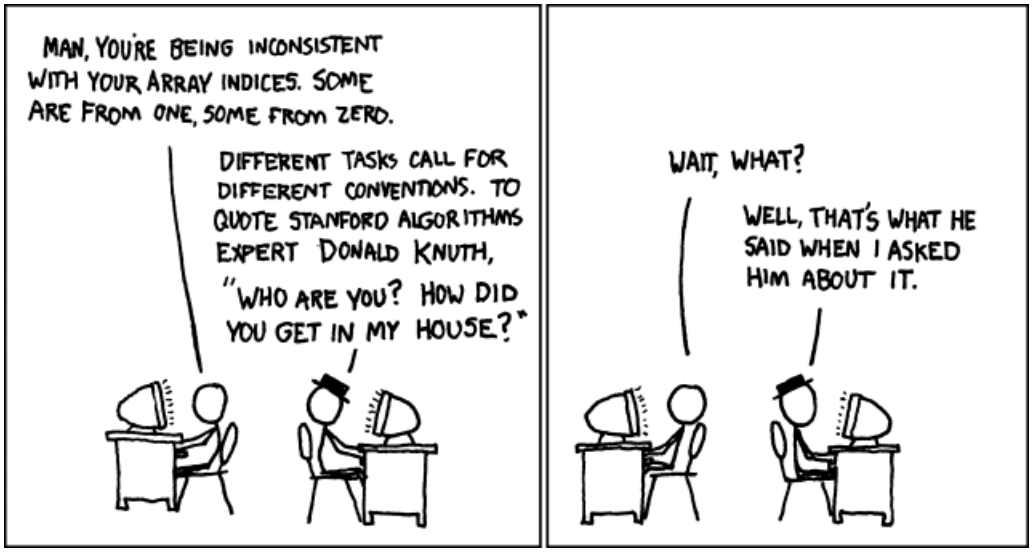
Assignment 10: Agents and board games

ETH Zurich

Hand-out: 3 December 2010
Due: 19 December 2010



Donald Knuth © Randall Munroe (xkcd.com)

Goals

- Test your understanding of agents and event driven programming.
- Test your debugging skills.
- Finish the design and implementation of the board game.

1 Air conditioning

You are implementing software for an air conditioning system. Assume that there is a hardware component that measures the temperature and calls a predefined routine from your application whenever the temperature changes.

Additionally, you are given two classes containing different reactions to changes of temperature. 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*: *DOUBLE*) that turns heating and cooling on or off depending on the difference between the current and the goal temperature.

Your task is to implement a class *TEMPERATURE_SENSOR* that receives the signal from the hardware component and makes sure that all the required reactions to a temperature change are performed.

To do

1. Create a new project in EiffelStudio with a root class *APPLICATION*. Download classes *DISPLAY* and *HEATING_CONTROLLER* from http://se.ethz.ch/teaching/2010-H/eprog-0001/assignments/10/ac.zip and add them to your project.
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*) (this feature will be called by hardware whenever the temperature changes).
   - It should allow to register agents as observers. All the observers should be called every time the current temperature changes.
   - It should be possible to register any number of observers.
   - 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*.

2 Debug me!

In this exercise you will be given a class that contains faults (bugs), as well as a test scenario that reveals those faults. Your task is to fix the faults in such a way that the given test scenario is executed without failures (contract violations or calls on void target).

To do

1. Download http://se.ethz.ch/teaching/2010-H/eprog-0001/assignments/10/debugging.zip, extract it into a directory of your choice and open the project "debugging.ecf" in EiffelStudio.
2. Class *SORTED_LINKED_LIST* represents a singly-linked list where elements are sorted. Its implementation contains 10 faults[1]. The root class *TESTER* contains a testing routine that exercises the behavior of *SORTED_LINKED_LIST*. Each of the 10 faults is revealed by the testing routine through violating either the contracts of *SORTED_LINKED_LIST* or **check**s in the routine itself, or causing a call on void target.

[1] The number of faults is approximate; the exact number depends on the definition of "fault".

3. Run the program: it will result in a failure. **Using the debugger** (step-by-step execution, "Objects" tool, "Watch" tool) try to figure out the reason of the failure and fix the implementation of *SORTED_LINKED_LIST* so that it doesn't occur. Note: modify only feature bodies in *SORTED_LINKED_LIST*; you are not allowed to change contracts, add or remove features, or modify class *TESTER*.

   Repeat this until the program executes without failures.

To hand in

Hand in the modified class *SORTED_LINKED_LIST* and one-sentence descriptions of fixed faults (these can be in a separate document or inserted as comments in the code).

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 have chosen 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 (figure 1), divided into 20 squares, a pair of *four*-sided 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 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 following kinds of squares on the 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.

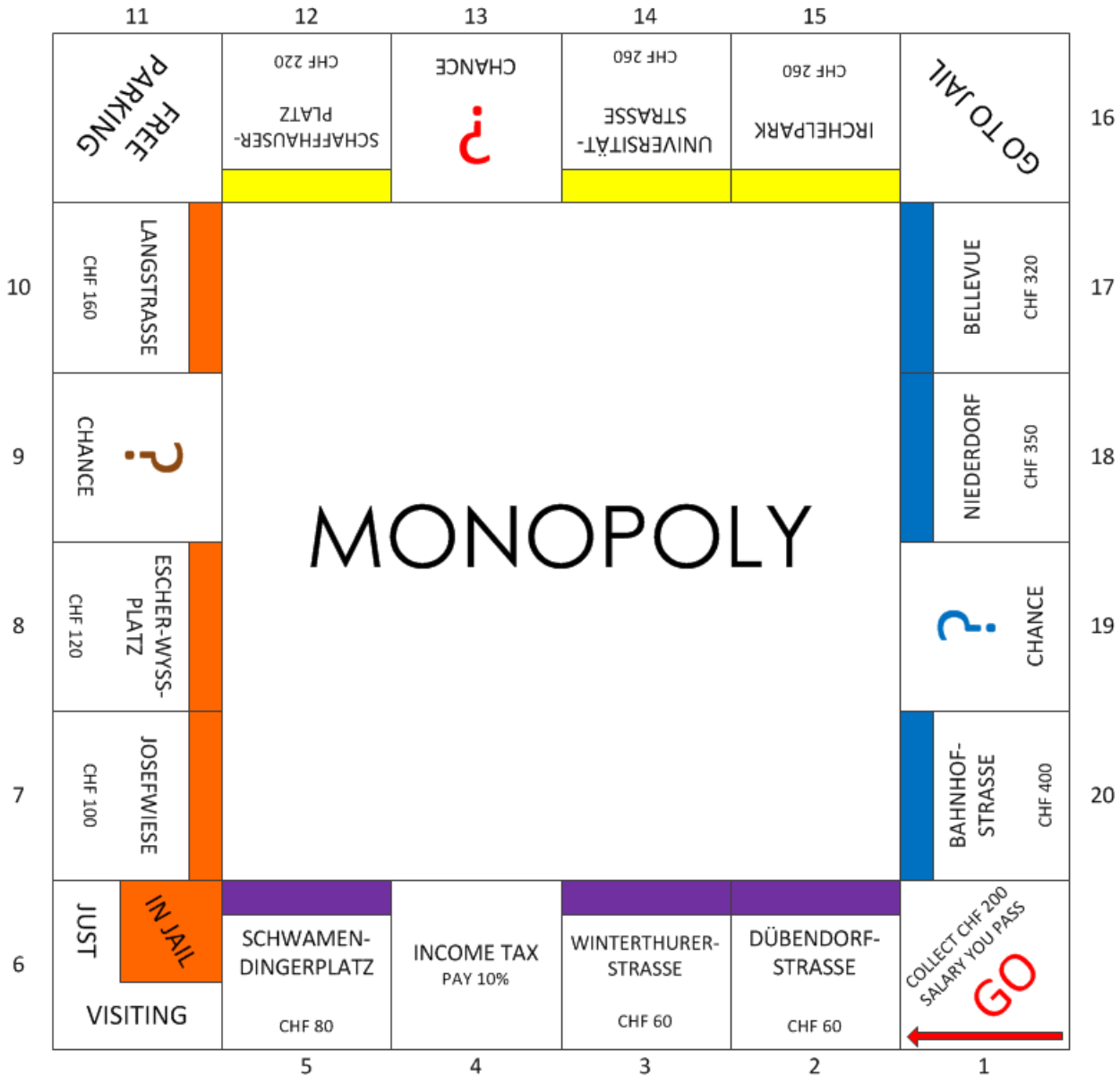


Figure 1: Monopoly board

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[2] on any of his next three turns (if he succeeds in doing this he immediately moves forward 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 with 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 the assignment 8:
http://se.ethz.ch/teaching/2010-H/eprog-0001/assignments/08/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.

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