Large Programming Assignment: *Binary Puzzle Assistant* (BPA)

Programming Methods (2IP15)

December 2012 – March 2013, *BPA* (v1.2)

Abstract

The goal of this assignment is to integrate the activities that play a role in developing a larger object-oriented program with a graphical user interface. Note: This document will receive updates (see change history at the end).

1 Introduction

This assignment concerns the development of a program that assists the user in solving and designing *Binary Puzzles*. See below for the precise requirements that the program has to fulfil.

A *Binary Puzzle* consists of a rectangular grid, partially filled with zeroes and ones (see Fig. 1). The objective is to fill the grid completely with zeroes and ones such that nowhere more than two equal symbols are horizontally or vertically adjacent, and the number of zeroes equals the number of ones, in each row and in each column. The given zeroes and ones cannot be changed when solving the puzzle, but a puzzle designer needs to change them.

0		0		1			
					1	1	
0	0			1			
0						1	0
			1				
				1			
			1				0

Figure 1: The initial state of a *Binary Puzzle*

More precisely:

1. There is a *rectangular* grid of square *cells*, with an even number of rows, and columns. Typically, but not necessarily, the grid itself is also a square.
2. Each cell is either *open* (empty), or contains a single *symbol*. The symbol is either a *zero* (0) or a *one* (1).
3. No *three horizontally or vertically adjacent symbols* are the *same*.
4. In each row and in each column, *no more than half the cells* contain the *same* symbol. Hence, in a completely solved puzzle, the number of zeroes equals the number of ones, in each row and in each column.
5. The puzzle has a unique solution.

In some variants, there are other and/or additional rules, but we will not consider such variants.

2 Requirements

There are three priority levels for requirements:

Priority 1 The requirement must be fulfilled. This is the default priority.

Priority 2 The requirement should be fulfilled, but after Priority 1.

Priority 3 This requirement is desirable, but only after Priority 2.

Note that all priority levels must be addressed in the final version. These priorities help you to choose a development order.
Concerning functionality:

1. The program incorporates the rules of binary puzzles as described above.
2. The user can load a binary puzzle from a text file via File > Open.... Each line describes a row in the grid, using the character '`.`' for an open cell, '`0`' for a zero, and '`1`' for a one. The zero or one can optionally be followed by an asterisk '`*`' to mark a value entered as part of a solution, that is, in an *unlocked* cell; other zero/one cells are *locked*, empty cells are always *unlocked*. Spaces are ignored.
3. The user can save the current puzzle state in a text file, via File > Save..., according to the same format, separating cells by one or two space characters (to compensate for an '`*`') to format the output in aligned columns. When loading a saved file, this must result in the same grid state as directly before the save operation. [Priority 3]

4. The user can change the contents of cells (also see below).
5. The program can apply simple *strategies* to help solve a puzzle:
   (a) If three adjacent cells contain two equal symbols, then the third cell, if empty, must be filled by the other symbol.
   (b) If half of the cells in a row or column contain equal symbols, then the remaining empty cells must be filled by the other symbol. [Priority 2]
   (c) If filling an empty cell by symbol s (zero or one) would lead to an inconsistency through repeated application of the preceding two strategies, then that cell must be filled with the *opposite* symbol. [Priority 3]
6. The program can apply *backtracking* to help solve a puzzle. [Priority 2]

Concerning the graphical user interface (GUI):

7. The puzzle state is shown graphically (see Fig. 1).
8. The user can change the contents of unlocked cells by *point-and-click* on a cell to cycle the contents, from empty to zero to one to empty again.
9. With the alt-modifier key, clicking a cell cycles in the opposite direction. [Priority 3]
10. The user can request feedback on rule violations in a warning message that remains visible via a Check button or menu item.
11. When enabled by the user, rule violations are automatically marked in the grid, by highlighting the violating symbols in red. [Priority 3]
12. The user gets automatic feedback when the puzzle is solved. [Priority 2]
13. The user can request the application of *strategies* via an Apply Strategies button or menu item.
14. The user can choose whether the strategies are applied until the first change, or until no further changes occur. [Priority 2]
15. Which strategies are applied is selectable by checkboxes (possibly in menu items). [Priority 3]
16. The user can initiate *backtracking* to find *one* solutions, via a Solve button or menu item. Note that the selected strategies are automatically applied after each speculation step during backtracking. [Priority 2]
17. The user can initiate *backtracking* to find *all* solutions, via a Solve All button or menu item. Note that the selected strategies are automatically applied after each speculation step during backtracking. [Priority 3]

18. The user can enable *edit mode*, to change locked cells. [Priority 2]
19. The user can repeatedly undo any change via Edit > Undo. [Priority 2]
20. The user can repeatedly redo any undone change via Edit > Redo. [Priority 3]
21. The user can undo all changes in one action (Undo All), and can redo all undone changes (Redo All). [Priority 3]
22. Cells changed in the previous action (directly by user, or indirectly by strategies or backtracking) are marked, e.g., a light blue background. [Priority 3]

Concerning the design of the program:

23. There is a clear separation between
    - *model classes* that describe (the state of) a binary puzzle; these include at least `Grid` and `Cell`;
    - *view classes* that contain additional information to render (a view of) the puzzle in the GUI;
    - *control classes* that respond to the user interface actions of the user;
    - *solver-support classes* that offer operations on puzzle states to assist in solving or designing a binary puzzle.
24. Every non-GUI class has a corresponding JUnit test class that tests at least the key functionality. In Test-Driven Development, these are created after specifying functionality, but before implementing functionality. [Priority 1]
25. The design accommodates future modifications and extensions, such as adding new strategies, through the application of appropriate *design patterns*.
26. The design conforms to the provided *checklist*, and the implementation code adheres to the *2IP15 Coding Standard*.

Wish list (not required to be implemented, but could be used to help make design decisions):

1. Provide *hints* for the user.
2. Maintain some *statistics*.
3. Determine the *difficulty level* of a puzzle.
4. *Automatically generate* (solvable) binary puzzles.

3 Submission to peach^3

Every week, starting in Week 3, you need to submit the most recent version of your project to peach^3.

Put your *name* and the *date* in every file that you create or change.

Submit your entire NetBeans project in a zip archive. Make sure you have included `package-info.java` with current status information in every package.

4 Model

Let us first analyze what the *model* for binary puzzles should provide.

1. It *stores* the state of a binary puzzle, during the process of solving it. Solving operations can be done manually by the user, or automatically by some algorithm (below referred to as "client").

   This involves storing the *state* of each cell in the grid. It would be useful to distinguish whether a cell was initially filled with a symbol (and its contents should not be changed by the user), or whether it was initially open (but later on possibly filled by the user with a symbol). This can be accomplished by an attribute `boolean locked` per cell.

2. The client can *construct* a binary puzzle in its *initial state* from a given `Scanner`. The format is described in Section 2.

   The *controller* (one of the clients of a binary puzzle model) can then provide a scanner for a text file that the user has selected (through a `JFileChooser`).

3. The client can *query* (the state of) a cell given by its coordinates. This includes whether it is *locked*, and whether it is involved in a *rule violation*. This could be accomplished through separate query methods.

4. The client can *modify* the state of a cell given by its coordinates to a given new state, provided that the cell is not locked.

5. The client can *query* the puzzle state as a string, which can then be written to a file. The format is described in Section 2.

6. The client can *query* whether the current state *violates the rules*.

This analysis is not necessarily complete. The code will evolve later on, as it is being used by actual client code.

Avoid code duplication, and follow the DRY principle: *Don't Repeat Yourself*. For instance, you can use an `Iterable<Cell>` to iterate over a line, being either a row or a column. That way, you can write the code to check the rules for a line *once*, and apply it to both rows and columns.

Note that your test cases are an important first client of this code.

5 GUI: Views + Controller

Here are some hints for the design and implementation of *view* and *controller* classes.

1. What goes where? The requirements tell us what the user is to see of the model (that is, of the puzzle). This needs to be done by the view, in particular, in its `paintComponent` method:

   - A rectangular grid of cells, partially filled with symbols 0 and 1.
   - Some indication of whether the rules are violated, preferably as localized as possible (e.g., using red symbols).
   - It might be good to visually distinguish locked cells (e.g., using grey symbols).
   - Finally, it might be nice to provide visual feedback when the puzzle is solved.

   The view also needs to offer functionality to convert mouse coordinates within the grid into cell (row and column) indices, so that the controller can handle mouse clicks.

   The controller has to offer functionality

   - to load a puzzle from a file through an Open menu item in the File menu;
   - to change the puzzle state when the user clicks in a cell (shown in the view);
   - and later do the remainder and extras (clear all unlocked cells, invoke a solver, save to file, etc.).

   Both viewer and controller need access to the (puzzle) model. Both can store a reference to the puzzle in a (private) instance variable. The controller sets it in the view (because the controller can call methods in the view directly, but not the other way round).

2. Where to start the development? With the view or the controller? A chicken-egg problem. Let's consider both options in turn.

   (a) *Start with the view*. In order to view something, a puzzle needs to be available. This puzzle could be provided via the controller, which needs to handle the Open menu item anyway. Alternatively, we could test the view without controller by feeding it some puzzles created from hard-coded strings (see your test cases for the model) or from a hard-coded file. Hard-coding is not flexible and such test cases must eventually be replaced as the controller evolves. Anyway, also in this approach with a hard-coded controller, testing of the view needs to be done manually (to check the visualization).

(b) *Start with the controller*. In order to test the loading of a puzzle from a file, the loaded puzzle must be inspected. Since the loading is invoked manually, it might as well be inspected manually. For that purpose, the puzzle needs to be visualized. This need not be graphically, but could simply be in the form of a string sent to standard output (the console), or in a *text area* (`JTextArea`).

It might be good to have a scrollable text area available anyway, so that the controller can log its activities (both for user feedback, and helpful in testing and debugging). This text area can stay inside the controller, since its purpose is not to render an up-to-date view of the model. The text area can be tested without puzzle.

3. So, in conclusion, it might be best to start with the controller, loading a puzzle from a file through the Open menu item, showing it in textual form.

   (a) Create a new JFrame Form by right-clicking in the `gui` package (you can name it `MainFrame`). This class will have a `main` method, which can be used to run the application. That is another reason to start with the controller.

   The view class is not yet needed. First, complete the loading operation in the controller, and test it.

   (b) Add a menu bar, in the Design view of `MainFrame` (drag it from the Swing Menus in the palette on the right).

   (c) If not yet present, add a menu to the menu bar (again by dragging), and change its name into File.

   (d) Add menu items Open and Quit to the File menu (you can also drag menu items).

   (e) Add a Text Area within a Scroll Pane to the controller window.

   (f) Add `actionPerformed` event handlers to the menu items, and implement these methods. Use `System.exit(0)` to quit the application. Write the loaded puzzle as string to the text area using the method `JTextArea.append(String)`.

   (g) Test the menu items.

4. The automatically generated names for GUI elements are too general. You must give generated instance variables better, application-specific names. This is done in the Design view, by right-clicking on the element and selecting "Change Variable Name…". For example, change `jMenuItem1` into `jMenuItemOpen`.

5. Next, provide a graphical view, initially without highlighting.

   (a) Create a new JPanel Form (a subclass of `JPanel`) by right-clicking in the `gui` package (you can name it `PuzzlePanel`).

(b) In the Design view of `MainFrame`, drag a `PuzzlePanel` from the (top-left) projects pane to the main window (frame); place and size it appropriately.

(c) Give the `PuzzlePanel` a private instance variable `puzzle` and a corresponding setter method, so that the controller can inform the view about the puzzle to visualize.

(d) Override the `paintComponent` method in `PuzzlePanel` to draw (the current state of) the model. Do not forget to start with

```
super.paintComponent(g);
```

(e) Note that NetBeans will introduce a `MouseAdapter` to delegate the handling of mouse events.

6. Test the view.

7. In `MainFrame`, add a `mouseClicked` event handler to the the puzzle panel. Implement this method. It will need to ask the view to convert the mouse coordinates into (a reference to) a grid cell (or its coordinates).

8. Test mouse clicking.

   Note that proper handling of mouse events could be tested in a unit test, by generating appropriate mouse events in the unit test and checking that they had the expected effect. However, this is not needed (also see below about manual testing).

9. Incorporate other functionality.

These considerations are not necessarily complete. The code will evolve further, as it is being used while the application grows.

Advice Add functionality in small increments and see to it that your program compiles and runs properly after every increment.

Testing Note that this code also needs to be tested systematically, to prevent surprises later on. However, it is harder (though not impossible) to automate the testing of graphical user interfaces.

Describe manual test cases

For this assignment, we will be satisfied with some test cases that are executed manually. Describe these manual test cases in comments in an, otherwise, empty test class. For instance, the test cases for the main controller in `MainFrame.java` reside in `MainFrameTest.java`. That will ensure that we can check them, that you can repeat them, and that you will not just do some ad hoc improvised playing instead of systematic testing.

6 Solvers

Here are some hints for the design and implementation of *solver* classes.

1. There are three pieces of functionality to design and implement:

   (a) an undo/redo facility; although the original requirements state that this has lower priority for the user, it is still important for some strategies and for the backtracker when it is using the strategies;

   (b) a facility to automatically fill open cells according to certain strategies for forced cells (see Section 2, item 5; note that requirement 5(c) is not immediately necessary);

   (c) a backtrack solver; for efficiency reasons, it also applies the selected strategies.

2. The undo/redo facility can be realized through the *Command design pattern*. Put it in a separate package `commands`.

   The first concrete command to incorporate concerns the operation to change the state of a cell. That operation takes one parameter, the new cell state, and it is called on a particular `Cell` object (the receiver). To undo it, the command needs to store the old cell state before changing the cell.

   Note that it is useful to incorporate the protocol checking as was done in the `CommandPattern` example (using exceptions and the instance variable `executed`). However, this needs some thinking when providing a compound command (see below).

3. Provide a unit test to exercise the functionality in the `commands` classes.

4. Provide an Undo menu item in the Edit menu, and also a Redo. The implementation will involve an undo and a redo stack of commands.

5. When applying strategies, it is convenient to collect all the resulting cell changes in a single *compound command*. This allows for the possibility to undo and redo all these changes in one step.

6. Provide strategy classes with methods to fill (one or more) open cells that are forced to a certain symbol by the rules. Design an appropriate strategy interface and toolkit to combine them flexibly. Here are some considerations.

   - Each strategy offers a method that applies it to a given (open) cell.
   - A decorator strategy that finds an empty cell to which a given strategy applies, and then applies it, resulting in, at most, one changed cell.
   - A decorator strategy that repeatedly applies a (possibly composite) strategy until no further changes occur.

- A composite strategy that consists of a collection of strategies that are applied in succession.

Each strategy returns a compound command of (already executed) cell changing commands that resulted from the application of the strategy. That way, it is easy to undo them (in another strategy, the backtracker, or the user interface).

The puzzle on which these methods operate, can be stored in an instance variable (provide at least a setter, so that the controller can set the puzzle after loading it).

Here, the *Strategy design pattern* can be used, as well as the *Decorator design pattern* and the *Composite design pattern*.

7. Provide a Puzzle menu with a menu item Apply Strategies to let the user select and apply strategies. The user can select which strategies will be applied and whether strategies will be applied once or until no change occurs (check box menu items).

   You can test the strategy functionality manually (no automated unit test needed; but do describe your manual testing).

8. Provide a class `Backtracker` to automatically solve a puzzle through backtracking. To make the backtracker more efficient, it starts by applying the strategies. Next, it picks an open cell (if none exist, the puzzle is solved), and tries both symbols in succession, solving the remainder through recursion.

   The puzzle on which the backtracker operates, can be stored in an instance variable (provide at least a setter, so that the controller can set the puzzle after loading it).

9. There are two modes of operation to consider:

   (a) Required: The backtracker continues searching until it has exhausted all possibilities; it reports solutions, as they are found, through a observer interface; the controller registers a simple observer that appends each solution as string to the text area.

   (b) Optionally: The backtracker stops as soon as it finds a first solution, and leaves that solution in the puzzle, so that the *view* can show it.

   This can be accomplished in various ways.

   - The recursive backtrack method returns a **`boolean`** to indicate that it found a solution; this return value is checked, and if **`true`**, the backtracker breaks off the search and returns **`true`** to propagate the termination.
   - Alternatively, it can be done by throwing a custom exception in the recursive method, and catching it outside the recursive method.

10. Provide a menu item Solve All in the Puzzle menu to let the user invoke the backtracker to find all solutions.

    You can test the backtracker manually (no unit test needed).

    If you want, you can a add a second menu item Solve One for the functionality to stop at the first solution found.

These considerations are not necessarily complete. The code will evolve further, as it is being used while the application grows.

7 Edit and Save

Some hints for the design and implementation of the *edit* and *save* facilities:

1. There are two pieces of functionality to design and implement:
   (a) ability for the user to switch to *edit mode*; in edit mode, the user can change the initial state of the puzzle; that is, the user can change the state of locked cells;
   (b) ability for the user to *save* the current puzzle state in a file; see requirements for details on the file format.
2. Incorporating the *edit* facility involves the following:
   (a) Add an Edit Mode check box menu item in the Edit menu, to let the user switch to and from *edit mode*.
   (b) When entering edit mode, all unlocked cells are cleared (if you are nice, ask for confirmation). This need not be undoable.
   (c) In edit mode, a mouse click changes any cell: the controller first unlocks it, then changes its state, and finally locks it, unless it is open.
   (d) In edit mode, the Apply Strategies and Solve menu items should be disabled, or return without doing anything.

   You can test this edit functionality manually (no unit test needed).

   The operations in edit mode must be be undoable and redoable.
3. Provide a menu item Save... in the File menu to let the user save the current puzzle state to a file.

   Reuse the `JFileChooser` that was introduced for loading puzzles from text files. Instead of the `showOpenDialog` method, the `showSaveDialog` method is called.

   It is good practice, to ask confirmation when the selected file already exists, and not to overwrite it silently. This can be accomplished with the **`static`**

`JOptionPane.showConfirmDialog` method. Also see §11.2.3 in the course book by David Eck.

You can test the save functionality manually (no unit test needed).

8 Run the Backtrack Solver in a Background Thread

Here are some hints to run the backtracker in a separate thread.

1. Without concurrency, all activity in the *Binary Puzzle Assistant* runs in a single thread, the GUI thread. This has two disadvantages:

   (a) Any calculation (for example, finding all solutions) that takes more than a fraction of a second, will block the GUI from responding to user actions. The user will have to wait until the calculation finishes. The user even does not have the ability to abort or interrupt the calculation, without terminating the entire application.

   (b) Even if the user is willing to wait, such a longer running calculation cannot produce graphical output (involving updates of the view).

   These limitations can be overcome by running such calculations in a separate thread. Here, you are required to run the *backtrack solver* "in the background" via a `SwingWorker`.

2. There are several things to take care of (also see slides and example code):

   (a) Create, configure, and start the `SwingWorker`, when the user selects the Solve menu item. This `SwingWorker` will simply invoke the backtrack solver in a separate thread.

   (b) Introduce a way for the user to abort the backtrack solver. For example, by changing the menu item that invoked the solver, into an Abort menu item. There is one event handler for this menu item, and its action then depends on whether or not the solver is running: either start or stop it.

   (c) When the user requests to abort the solver process, the controller should not just kill it, because that might leave the model in an undefined state. All that it should do is raise a flag to communicate this request to the solver. The solver will have to be adapted to honor such a request at a safe moment, preserving the integrity of the model.

   Since the backtrack solver is recursive, the body of the recursive method can start by checking whether an abort is requested, and if so, throw an `InterruptedException`, which is caught by a (non-recursive) top-level method, that then terminates (without exception).

(d) When the backtrack solver finds a solution, it can signal this to the controller, so that the view can be updated. It suffices here to provide the solver with an *observer* that updates the view and waits for half a second to let the user get at least a glimpse of the solution, before it is overwritten by the next solution.

3. You can test the `SwingWorker` functionality manually (no automated unit test needed).

Notes Concurrency is wonderful and dangerous (as are other things in life). This part of the assignment is a first practical encounter with concurrency.

The communication from the solver thread to the GUI thread brings some complications, depending on how you designed the view. If the view draws the model on-demand in a `paintComponent` method, then the view will track the solver. However, the view will be updated only when requested, either by an explicit `repaint` (or similar method) call from the application itself, or by a user action, such as resizing the window. The solver thread keeps modifying the model state, while the view is querying it. That way, the user may end up seeing an inconsistent rendering of the model state. For this assignment, that is acceptable.

The application should not crash or "damage" the model data structure through concurrent modifications. It would be best to block user action that can change the model (mouse clicks on cells, applying strategies), while a backtracker is active.

9 Grading Criteria

The *Coding Standard for 2IP15* applies, as well as the *Checklist* for developing larger object-oriented programs. Also important is that functionality works as required and has no undesirable side effects (such as crashes).

All functionality (all priority levels) must be present. The indicated design patterns must have been applied; in particular,

- Command pattern (for undo-redo, strategies, and backtracking)
- Composite pattern (for commands and for strategies)
- Strategy pattern (for strategies and backtracking)
- Decorator pattern (for strategies)
- Observer pattern (for the backtracker)

Offering the possibility to stop the backtracker at the first solution is optional.

10 Change History

v1.0 First release.

v1.1 Released to instructors only.

1. Added a note about the significance of priorities.
2. Clarified testing of mouse event handling.
3. Added advice on incremental development.
4. Removed quotes around *strategy*.
5. Clarified requirements for automatic strategies.
6. Removed drag-and-drop from Edit Mode.
7. Added undo-redo to Edit Mode.

v1.2

1. Various minor improvements in the formulation.
2. Required that automatically generated names of GUI elements are changed into application-specific names.
3. Clarified manual testing: describe it via comments in a test class.
4. Changed *listener* into *observer*.