1.What kind of a Maze will the MazeApp have before you've loaded any file?

Green one made of grass, I mean it's just plain image stating that no maze is loaded. Somehow didn't like to load anything by default. Even other applications like MS Word I like when they start with blank content. The maze doesn't need any files anyway, if you want just test it and have no files, you can generate mazes.

2.Which object should be responsible for generating the status message?

It's inside GuiDraw class and everybody calls it as it desires. If there would be too many messages and they would override each others, then maybe different mechanism would be needed. But in this case and with these messages it works fine.

3.How will you share the responsibility for actually drawing the maze? Remember that the solution-in-progress should display explored squares in grey---which data structure is keeping track of that information?

It's done inside GuiDraw, while Gui class handles all gui the GuiDraw handles all maze rendering, path coloring and updates of status bar messages. First it was drawn from objects but later i converted to bufferedImage. It wastes memory to keep RGB 16x16 image per each block but to having whole maze as one image has it's own advantages. If I wanted to improve and have artists custom draw unique maze there would be minimal change on code. I have done it past where automatic and generated graphics was enough to test something in early stages, and it was good baseline for artist to have it as background. And then he can quickly make something unique (which wouldn't be possible if tiling would be used). This way with 20% of effort we got 80% of finalised image. It's not 90's anymore and thanks to more memory and storage now maps, mazes and backgrounds do not have to be tile based. I would stay away from tiles if I can. Once I needed 32000 pixels x 32000 background image, then I managed to stream ion the fly just the visible part of the image, and no repetitive parts, everything unique.

4.How will you represent locations? Why can't you just modify Square for this?
I didn't wanted Square, I wanted my own type (Block enum) so I can have different types in the future and having own class it's easier to extend or modify.

5.How will you keep track of the locations you've explored vs those you haven't?
I never keep informations about locations I haven't explored per say (that would be all white squares). I just keep information about locations which are accessible, they were not visited and I want to visit them. This limits the size of the list because the algorithm for example wants visit just 3 forks, but list of non visited would be whole maze.

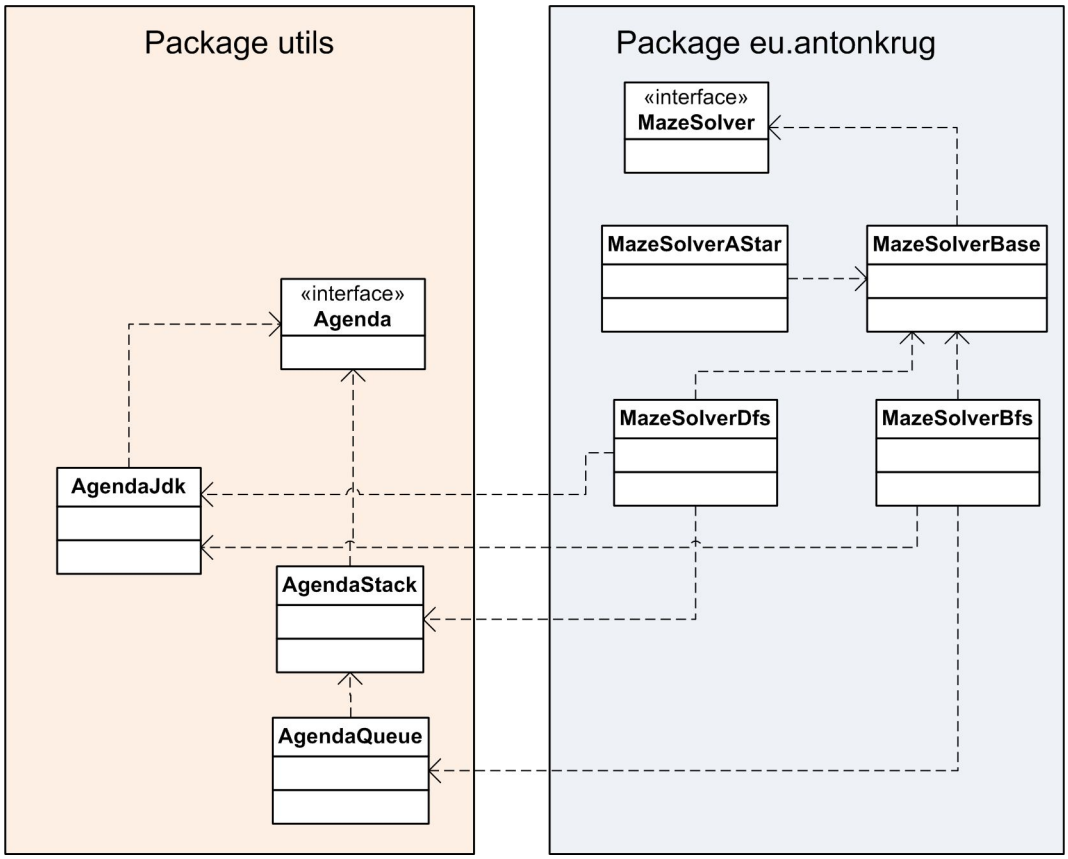
- For MazeSolverAStar both lists are kept in hash maps. Even there is memorry overhead and there is penalty till the hash is calculated or even another penalty in hash collision occurrences (in java8 performance of collisions improved a lot!). Still i like this implementation because of average speed of O(1). Because I didn't wanted modify maze lot of stuff needs to be done on these lists, many times **contains** called, often random access **get**s. Sometimes even the list is modified, A* is able to update already visited path if there was shorter path found which then updates own nodes inside the class. Being destructive on the maze could improve performance on other methods and then this one wouldn't be as dependant on hashmaps as it's now. But this way multiple independent solvers should be able to run at the same time, without any interference. If multi-threaded version is used then just thread safe hashmaps are used.

- There are 2 different versions of MazeSolverBfs, one using Linked Lists directly as Queue. One using AgendaQueue (which inside holds stuff in Linked List, but accesses it differently).

- There are 2 different versions of MazeSolverDfs, one using JDK Stack and one using AgendaStack (which holds items in ArrayList).

Some parts are so abstracted now that it doesn't affect functionality if I would change some variables for different types. I did 2 implementations for BFS because I wanted still have just wrapper for JDK and then something i was trying to do. The wrapper performs faster than mine implementations. But still is pretty slow, it could be run faster if I woudn't some runtime checking, or if I hadn't had no wrrapers or any other interfaces and inherited classes (which are overhead, for inheritence checks needs to be done on runtime). I had plain Stack and LinkedList implementation before I was experimenting with Agenda and it was performing much much faster, but I wasn't able to switch between different datastructures. Now I kept the slower version because I like the interface, ability to switch between other Agenda implementations is nice. Plus I did few things to practice some stuff, not to make it faster.

I tried to do UML class modeling in visual paradigm, but I didn't like it (i left some pictures in the root), but I did simple diagram in Visio:



As show in the picture below I defined interface Agenda and everything implementing it can be using it. AgendaQueue was very similar to AgendaStack so I desided to extend it instead of scratch implementation of Agenda. Then I have MazeSolver interface. Then abstract class MazeSolverBase which contains most of common logic and all 3 classes MazeSolverAStar, MazeSolverBfs and MazeSolverDfs are extending on this abstract one and over-riding only methods which they need. This way Gui is using MazeSolver interface and doesn't care about which implementation is used, and Dfs, Bfs can use 4 different data-structures and because all implement Agenda interface they can be swapped between each other.

6.What other data structure(s) would you need if your solver needed to be able to report the actual path from the start to finish, instead of just the fact that one existed (or not)?

For DFS stack is enough, for others which jump around a lot a map with keys and values is good. Key should contain given location and value should point to location of parent. A* has already visited list which contains this information and for BFS a extra map had to be used to store node -> parent informations so I could display actual path realtime for each step.

7.Whose responsibility should it be to actually create the MyStack or MyQueue, as appropriate?
BFS and DFS in runtime they decide if they will use mine implementations or use the JDK wrappers. (depending what arguments I sent them when calling constructor).

8.The obvious time to click the Step button is immediately after one of the Start buttons, which was probably clicked right after the Load button. What should happen if a user clicks buttons at an unexpected time? (Hint: not a NullPointerException.)

Because the more functions I was adding the more combination of possible button presses was increasing and it was starting to be confusing which functions are allowed. The conditions were increasing and when I did static code analysis it was growing fast. Realised that is not the way to go, changed the design and now the buttons which shouldn't be pressed just can't be pressed. To increase stability I was paying attention to Cyclomatic complexity and other factors (the report is in staticCodeAnalysisReport.xml) and when needed re-factored few methods to have them easy to read. Or in some cases I decided completely different approach, which in retrospective really looks like better code. Actually I'm impressed how the process got so much better, in past I wouldn't be able to do static analysis, test, re-factor so easily and with free tools (or even at all). Java8 is much more pleasant to work with than precedences. I think when you enjoy your environment it can show in the code you produce. I designed JUnit tests to test all data structures and to test solvers as well. Solvers are tested for the results for each maze (9 test mazes), have to contain correct number of iterations, correct size of solution, size of visited and opened lists (and no errors or exceptions). Or throw exception when it's supposed to give exception. And the positions inside these list are checked as well (if same length but different path was chosen it will fail the test). And I test all 6 combinations, 3 algorithms and have 2 internal data-structures each. For 9 different mazes each contains different problem done lot of automated testing with minimal effort. This is not bullet proof but it makes me a more confident than without them, without wasting time to test all 6 combinations at each 9 mazes. When I was redoing and re-factoring some parts it helped me to detect problems which occurred just in 1 instance from all 6, and manually maybe I would overlook it. I'm not saying these tests are bullet prof and now my code is bug free, I'm just saying that they saved me a lot of time and found some bugs I maybe wouldn't notice.Because there are no tests for GUI I decided to give it to people without any instructions just one "try to break it on purpose". Giving any instructions could superimpose my ways of using the application on test subject and I wanted to avoid it. These approaches helped me to eliminate many bugs (hope there are none left :)). I don't have 100% JUnit coverage, and even I see other people trying to do JUnits they do it badly, forget about false positives, true negative and other extremes and the best case test just best case scenario. There is no point testing getters when the complicated methods are badly tested. I think making flawless tests is much harder than making flawless code in the first place. So with mine approach I wanted get lot of confidence about the code, without testing every single getter and setter in all extremes and in all possible combinations (and still could miss something).

Final thoughts.

If I would continue the algorithm after it found the destination to fill whole maze, I could end up with nice lookup table which will get me to starting position no matter what point in the maze i will chose. This could be used in games where large number of mobs are having same target (your position). It could work even when you would moving, you could recalculate it every 3rd frame to save on calculations and still have pretty accurate tables.

I prefer the A* because the AStarNode can be extended in so many way and still non-destructive to the map. The map can be non-uniform (polygons instead of squares), different costs for paths and diagonals supported easily . Enemies on the map could be in different states "Fear from attack", "Hurt" where he could prioritize healing block (hospital) and cover block (forest) over fast path (going on road). There could be heigh on map and depending from which direction you will travel enemy could get bonus or penalty for attacking (going uphill, or downhill) on the very same map block (so this information can't be contained inside the map blocks). Enemy could prefer going further but finding more suitable path to attack (primitive flanking). Because all the cost and values can have different weighting it will allow different types of enemies have different intelligence and priorities, some will take any chance for better cover, some for better attack or some will go the shortest path even if it's the worst one. Nice is there so many things the nodes can support which are not meant to be stored on the map itself. Node know path and current direction and there could be penalty accumulated for changing direction. Track based vehicles like tanks should go longer distance if they don't have to change direction and travel at higher speed (still they will get to target sooner). And that just few things that crossed my mind.

Even for this maze solver BFS and DFS are just fine, still I prefer the A* because it allows to be extended with such ease and adding new features is simple where DFS and BFS would require workarounds to get some more features into them.

Anton Krug - 20062210