Homework 5: Santorini (Part 2)

In this assignment, we return to Santorini but implement the full game including some god cards and a user interface. Implementing god cards will force you to explicitly think about extensibility in your design. This is a **3-week** assignment with one intermediate milestone. It has two main parts, which interact, but which you can start in either order. Based on your previous implementation from HW3 you will:

- Make your implementation extensible to support god cards. You will update your design and provide implementations for four god cards.
- Provide a web-based user interface for people to play the game (both players share a screen).

The game has multiple challenge problems, that are entirely optional. You can decide to engage in one or multiple of these if you like.

- *Optional challenge problem:* Implement additional god cards.
- *Optional challenge problem:* Support functionality to undo steps and replay the game by making data structures of the game immutable.
- *Optional challenge problem:* Provide an AI to allow single-player games against the computer.

As part of this assignment, you will practice object-level design, you will explicitly design for extensibility, and you will learn to write a web-based user interface.

Tasks

Warmup: Fixing the Base Game

Continue working in your Santorini repository from HW3.

Update your implementation and design documents based on feedback received for HW3. Fixing design issues will make it easier to extend the program and we will grade updated versions of these again, as described below.

Extension 1: God cards

Redesign the game to support god cards according to the rules of the game. As described in the rules of the game, each player can pick a god card which modifies possible actions, winning conditions, or other aspects of the game. The game should be easily extensible for different god cards. This will require that you plan in the base code to make state available to implementations of god cards and allow god cards to change the behavior of the base game. For example, you may want to provide extension points to extend the implementation of what is considered a valid move, to modify what action a player can take next, or to store additional state. In a good extensible implementation of the game, god cards can be implemented with few lines of code each and minimal coupling.

Required god cards: Implement the following four god cards to demonstrate that your system is extensible: *Athena*, *Demeter*, *Minotaur*, and *Pan* (see appendix or original game rules).

Design requirement: In your implementation entirely decouple god cards from the base game. The base game should not refer to any concepts of specific God cards or store specific state for the god cards. For example, the base game should *not* use `if (player.hasAtlasGod)) ...` within `Board.isValidMove(...)`) and the base game should not store whether the last player moved up just for implementing *Athena*. Of course the god cards can interact with the base game, call functions, register callbacks, and so forth.

Design documentation: Revisit the design documentation to (a) fix issues from HW3, (b) reflect the god card extension, and (c) add a discussion of extensibility and design patterns to the justification.

First, we recommend that you first update your diagrams based on feedback you received on HW3, since we will regrade diagrams for qualities similar to HW3.

Second, extend the designs to include the changes you make for god cards. This should include the extension points in the base game as well as an example of a concrete card. To keep things simple, you do not need to show all cards, but it is sufficient to just include the god card *Minotaur* as a single concrete example in your design documents. Specifically, in the domain and object model a single concrete god card is sufficient; for the contract and the interaction diagrams assume that the active player has the *Minotaur* card (and the opposing player has no card).

Third, extend your `justification.md` file with two new sections: Section "Extensibility" should describe your design decision with regards to making the game extensible (including considered alternatives and using the course's design vocabulary); section "Design patterns" should list all design patterns used (if any) and explain why they were used.

Testing: Also update your test suite to test the functionality of the game logic (unit tests), of the god cards (unit tests), and of the game with and without god cards (integration tests). As before, we do not set any specific coverage goals but ask you to use your own judgement about how much testing is useful for you to have confidence in your implementation.

Extension 2: User interface

Implement a web-based user interface for the game with god cards (2-player). The user interface should be functional, allowing two players (sharing a screen) to pick god cards (see rules, first player picks two god cards, the other player selects one of them, first player picks starting player), to select the initial spots for their workers, and to play the game taking moves and build actions until one player wins.

We leave details of the user interface to you. The user interface can be very simple. A 5 by 5 grid of buttons is likely sufficient with a button panel on the side for skipping optional actions and starting a new game. In the simplest case, you can represent workers by letters and towers with symbols on each button, e.g., `[[ ]]` for an empty level two tower, `[ X ]` for a tower with a worker from player *X*, and `[[[O]]]` for a tower with a dome. Background colors might indicate the active worker or valid locations for the next move or build. Text above the grid might indicate whose turn it is and what the next action is or who won. We do not grade for how pretty the game is, but we may provide bonus points for exceptionally nice user interfaces.

Both players can play alternating on a single screen. If using TypeScript or JavaScript, you can run the entire game in the browser (e.g., using `browserify`). If using Java you will create a web server that renders the game state on request; a Java solution can also include some TypeScript or JavaScript code for the frontend. The server does not need to distinguish different simultaneous games (session management). We will show examples of these techniques in recitation.

Design requirements: Decouple the user interface from the core game. That is, the core game should not have any dependency on parts of the user interface (i.e., it should still work if all the GUI code was deleted). Decide deliberately what state should be stored in the game and what state is managed by the GUI code. Additionally, we recommend to use a template engine for rendering HTML (like in the starter code of HW4). You do *not* need to write automated tests for the user interface (but you are welcome to try).

Documentation requirements: In your `README.md` add a section "Starting a game" that explains how to start the game from the command line (e.g., what sequence of maven or npm commands do we need to call). Ideally the game's user interface is self-explanatory (e.g., explains what logos mean, has textual instructions on what to do next), but if it is not clear how to play the game for somebody who knows the rules, explain this in the README too.

Challenge Problem 1: More God Cards (optional)

Implement the other *basic* god cards (see appendix or original game rules) or any subset of them -- points will be awarded per additional card implemented. Note that supporting *Prometheus* will require a much more challenging extension mechanism than the other cards.

If you make additional changes to your design, reflect them in the design documents. Make sure the additional god cards are tested as well.

If you implement more than the four required God cards, add a section "God cards" to your `README.md` file that lists all the God cards that you implemented.

Challenge Problem 2: Undo (optional)

Extend the game to support undo and replay of the entire game. To this end, represent all state of the game using immutable classes and store the history of the game as a list or stack of game states. (This refers to the core of the game, the user interface may use mutable state.) We will cover immutability in class after the homework's release.

As part of the implementation ensure that users of the objects' public methods can indeed not mutate any state, especially when using mutable objects (e.g., lists, objects) internally.

Add an undo button to the user interface, allowing users to go back all the way to the beginning of the game. Update your design documents and tests to reflect the changes and the undo functionality.

If you implement the Undo functionality with immutable data structures, add a section "Undo" to your `README.md` file that briefly explains how you changed your implementation from HW3 to make the data structures immutable (one paragraph).

Challenge Problem 3: AI (optional)

Implement an AI that can play the game against a human player or against another AI. The AI should be able to use god cards. Using an minmax algorithm with alpha-beta pruning is sufficient and well documented online. It will yield an AI that is not entirely stupid but probably not very challenging to beat. Additional heuristics can make it stronger. You may reuse existing implementations or libraries for the AI algorithms (e.g., npm package minmax-wt-alpha-beta-pruning). You can explore other strategies too, especially if you have some background in this area, like reinforcement learning.

Update your design documents to include the AI functionality. You are not required to write automated tests for the AI. You may want to implement functionality to play AI vs AI games for testing or training, but this it is not required.

If you implement an AI, add a section "AI" to your `README.md` file that explains how to start human vs AI game and provides a brief description of how you implemented your AI.

Milestones and Deliverables

The assignment has one milestone and a final deadline. At both, you submit a link to a commit on Canvas as usual. You can use separate late days for both.

For the milestone, submit a reasonably complete version of the god cards or of the user interface. You may still change code and design documents after the milestone submission; we will only grade the final submission at the final deadline. For the milestone, we will only grade the presence of a reasonably complete solution for either extension.

Make sure that your repository has updated versions of `README.md`, `domain-model.pdf`, `system-sequence-diagram.pdf`, `interaction-move.pdf`, `interaction-build.pdf`, `object-model.pdf`, and `justification.md` (or `justification.pdf`) in the root directory of your repository, just as in HW3. For the final submission, ensure the diagrams and justification reflect extensions 1 and 2.

As in HW3, ensure the project is build and your tests are executed in Travis CI.

Evaluation

This assignment is worth 300 points, with up to 100 bonus points for the challenge problems. For the milestones we will only check whether a reasonably complete implementation was submitted. All other rubric items will be graded on the entire solution submitted for the main deadline.

Milestone (20 points):

- [ ] 20: A reasonably complete solution of one extension is ready at the milestone.

God cards (150 points)

- [ ] Design and justification (extensibility) -- 65pt
  - [ ] 10: The design allows god cards to change what move and build actions are valid the opposing player (e.g., for Athena). God cards can be added without changing the implementation of the game's core classes (i.e., no hard coding with if statements).
  - [ ] 10: The design allows god cards to allow additional (optional) actions (e.g., for Demeter). God cards can be added without changing the implementation of the game's core classes (i.e., no hard coding with if statements).
  - [ ] 10: The design allows god cards to change winning rules (e.g., for Pan). God cards can be added without changing the implementation of the game's core classes (i.e., no hard coding with if statements).
  - [ ] 10: The design must allow god cards to move other workers as part of the current player's actions (e.g., for Minotaur). God cards can be added without changing the implementation of the game's core classes (i.e., no hard coding with if statements).
  - [ ] 10: The design across all models makes reasonable decisions about responsibility assignment, avoids unnecessary coupling, and has reasonable cohesion.

- [ ] 5: The justification in file `justification.md` (or PDF) identifies all used design patterns (if any) and justifies their use or why no design patterns were used.
- [ ] 10: The justification uses suitable terminology and discusses design alternatives in a meaningful way, demonstrating an engagement with design principles and tradeoffs.

- [ ] Design diagrams (notation, consistency, completeness) -- 45pt
  - [ ] 5: The updated domain model in file `domain-model.pdf` describes the vocabulary of the problem, uses suitable notation, and is at the right level of abstraction.
  - [ ] 5: The updated system sequence diagram in file `system-sequence-diagram.pdf` is reasonably complete, uses suitable notation, and is at the right level of abstraction. It includes the setup for god cards.
  - [ ] 5: The updated behavior contract in file `contract.pdf` or `contract.md` is reasonably complete regarding pre- and post-conditions under the assumption that the active player has the *Minotaur* card.
  - [ ] 20: Two object-level interaction diagrams in files `interaction-move.pdf` and `interaction-build.pdf` describe the internal interactions for move and build and are consistent with the object model (i.e., only uses methods and accesses state to which the classes have access according to the model), uses suitable notation, and is at the right level of abstraction. Both models describe the behavior for the case where the active player has the *Minotaur* card.
  - [ ] 10: The object model in file `object-model.pdf` is reasonably complete, uses suitable notation, and is at the right level of abstraction. The model covers also all mechanisms for god cards and specifically covers the *Minotaur* card. It is consistent with the system sequence diagram.
- [ ] Implementation -- 40pt
  - [ ] 20: The four required gods cards are implemented in the game
  - [ ] 10: The game with and without god cards is well tested, using both unit tests and integration tests. The tests follow good practices. The test are executed automatically with Travis CI.
  - [ ] 10: The implementation is consistent with the design documents.

User interface (100 points)

- [ ] 20: Separation of GUI and core: The core of the game is independent of the GUI (works with GUI code deleted). Both are separated in different locations of the project. There are no GUI related concepts in the implementation of the core game and the GUI code does not reimplement the game concepts.
- [ ] 80: The game is functional, playable through the user interface:
  - [ ] 10: Instructions in the `README.md` file are sufficient to understand how to start and play the game (assuming user knows the rules)
  - [ ] 5: Game starts and can be restarted (e.g., 'new game' button)
  - [ ] 5: Allows each player to select a god card
  - [ ] 5: Allows both players to place the initial workers
  - [ ] 5: Indicates the current player
  - [ ] 10: Indicates available actions (valid move/build actions, cancel) or rejects invalid actions
  - [ ] 5: Recognizes the end of a turn and moves to the next player

- [ ] 10: Correctly updates the game after a move action
- [ ] 10: Correctly updates the game after a build action
- [ ] 5: Correctly identifies and shows the winner of the game
- [ ] 10: Correctly supports extra available actions allowed by the god card (move further, build again, etc)

General implementation quality (30 points)

- [ ] 10: Game builds and passes tests on Travis CI
- [ ] 10: Code quality and style: Code meets language's conventions, is reasonably well documented, reasonably clean (e.g., avoids dead code), avoids common programming errors (e.g., == vs equals in Java)
- [ ] 10: Reasonably cohesive commits and reasonable commit messages

Bonus points (100 points max)

- [ ] Challenge problem 1: More god cards (24pt max)
  - [ ] 4: points per additional god card listed in README.md, if it is implemented correctly, tested, and usable through the user interface
- [ ] Challenge problem 2: Undo (40pt max)
  - [ ] 10: The game and user interface have a working undo function that allows repeated undo steps until the beginning of the game
  - [ ] 10: The core game data structures are immutable (for example, we will check how the move and build operations are implemented)
  - [ ] 10: The implementation does not leak mutable state through methods or fields (for example, we will particularly check whether internal mutable lists/arrays/maps are accessible to clients)
  - [ ] 10: The immutable implementation is tested; the test are executed automatically with Travis CI; the design documents reflect the immutable design and matches the implementation.
- [ ] Challenge problem 3: AI (30pt max)
  - [ ] 18: A working AI that can play the basic game against a human can be selected in the user interface.
  - [ ] 2: The AI beats the TA who is grading the assignment in the first game.
  - [ ] 10: The AI can make use of the god cards.
- [ ] 6: The user interface is unusually well done: it looks good and is easy to use

Appendix: God Cards

You can find the original rules of Santorini including all god cards at: https://roxley.com/products/santorini

We consider only *basic* god cards. The bold ones are required:

- Apollo: Your Worker may move into an opponent Worker's space by forcing their Worker to the space yours just vacated.

- Artemis: Your Worker may move one additional time, but not back to its initial space. (UI hint: You will likely need a way to indicate that the player wants to skip the optional second move, either with a "pass" button or by clicking on the worker's current location)
- **Athena:** During opponent's turn: If one of your Workers moved up on your last turn, opponent Workers cannot move up this turn.
- Atlas: Your Worker may build a dome at any level. (UI hint: you can implement this in the user interface similar to Hephaestus, giving the player an second optional build action; this build action is interpreted as building a dome)
- **Demeter:** Your Worker may build one additional time, but not on the same space. (UI hint: You will likely need a way to indicate that the player wants to skip the optional second build, e.g., with a "pass" button or by clicking on the worker's current location)
- Hephaestus: Your Worker may build one additional block (not dome) on top of your first block. (UI hint: You will likely need a way to indicate that the player wants to skip the optional second build, e.g., with a "pass" button or by clicking on the worker's current location)
- Hermes: If your Workers do not move up or down, they may each move any number of times (even zero), and then either builds. (UI hint: Rather than allowing multiple move actions, it might be easier to indicate all possible target spaces where a worker can move too)
- **Minotaur:** Your Worker may move into an opponent Worker's space, if their Worker can be forced one space straight backwards to an unoccupied space at any level.
- **Pan:** You also win if your Worker moves down two or more levels.
- Prometheus: If your Worker does not move up, it may build both before and after moving.