SENG301: Practical Work

Lab 3 2022

Design patterns

1. The Prisoner's dilemma - Wikipedia is a well-known problem in game theory. It has been studied extensively, and with numerous variations, since the 1950s. The basic idea is to explore what happens when rational individuals are placed in a situation where they can choose to cooperate with each other or to betray each other—a payoff matrix determines the specific benefits or costs of the various combinations. We'll look at the *iterated* version, in which the participants play the same "game" repeatedly.

   Imagine we're in the backwoods of upstate New York in 1928 during the prohibition period. We've done a deal, set up by our intermediaries, with a somewhat dodgy Canadian bootleg distiller. Every Saturday, at midnight, we'll leave a wad of fake jewellery in a hole in a dead tree at Dry Creek and they'll leave a barrel of rum in Old Dave's barn. To make sure we keep everything as secret as possible, we'll wait until 2a.m. before we go to collect the rum and they go to pick up the jewellery. A barrel of rum costs them $100 to make and we can sell it for $200. Similarly, it costs us $100 to make the jewellery and they can sell it for $200.

   Everything works fine for a few months: we get to supply our speakeasy in the Big Apple and the Canadians get to finance expanding their distillery. But then things start to go wrong. Sometimes we get to the barn and there's no rum awaiting us. We sometimes decide to keep the cash for ourselves and still get the rum without paying for it. This has to stop before a bloody feud erupts — time to call in the software engineers!

   The options are to *cooperate* (leaving the cash or rum as promised) or to *defect* (not to). When we make a move, we have a potential benefit of $200 with an associated cost of $100. Thus, if we both cooperate we each end up with a net profit of $100. If we both defect, then neither of us gets anything. The most interesting case is where only one of us defects, and the other cooperates. For example, we collect the rum but "forget" to leave the jewellery. The defector then gets $200 with zero cost while the cooperator is left to suffer the cost. This variant of the prisoner's dilemma is called the *donation game*.

   In our context, "rational" means that each of our individuals ("players") will use some algorithm to determine the action they will take ("move") in each iteration. For example, maybe we should *always* trust the other party, however they behave. This could lead to an unscrupulous partner taking advantage of us. On the other hand, if we *never* trust the other party then they will soon tire of dealing with us. We could try tossing a coin to decide how we behave on each trade. Perhaps we should copy what the other party did on their last trade.

a. Design an experiment where two players ("prisoners") make a series of moves as described above. Each player will have some way of choosing the next move to make—these may be the same or different. You will need to be able to maintain the players' scores (i.e. benefits - costs) after each move and report them when the experiment is over. It should be possible for a player to adopt a different way to choose moves—perhaps because their current approach isn't working against the opponent.

   Your design should include at least one GoF pattern. Use plantuml to document your design as a UML class diagram—use stereotypes and/or notes to show how the elements of the GoF pattern map onto those of your design. Discuss your design with your neighbours and/or tutor.
b. Implement your design in Java. In particular, make sure all the elements of your GoF pattern are implemented. A "no-frills" approach is fine, but you should demonstrate more than one way for players to decide their next moves.
c. Run your code (via a simple application, JUnit or Cucumber). How do the most successful players decide their moves? Does the number of moves have any influence on the outcome?

2. We've been hired to design, and partially implement, software to control a television set. We'll keep it simple at first, while we demonstrate the feasibility and advantages of our design. In order to keep costs down, there are no physical controls (buttons, knobs, sliders, …) on the TV — everything is controlled by the remote. The remote unit has buttons which can be labelled to represent numbers or other operations — we'll write the software that handles things when the buttons are pressed. The remote also has a button labelled "undo" which reverses the effect of the previous command.
   a. What, if any, data elements will be needed to model the state of the TV? Remember the KISS principle!
   b. What, if any, methods will the TV need to have in order to manage any state variables it may have? Again, remember the KISS principle.
   c. Use plantuml to document your TV model as a UML class diagram.
   d. Now consider how the remote talks to the TV. We need a design that allows for additional operations to be added in future without significant refactoring. We suspect there's probably a GoF design pattern that will be suitable for this (hint: there is). What is it, and why is it suitable here? Discuss your choice with your neighbours and/or tutor.
   e. Extend your class diagram to illustrate how the pattern will be implemented. Use stereotypes and notes to show the correspondences between the elements of the pattern and those of your design.
   f. Implement your design in Java — make sure you include sufficient tests (JUnit and/or Cucumber) to demonstrate that it works as expected. You need not implement all the operations your TV supports, but should make sure you demonstrate how the "undo" button works.
   g. Our employer asks if we can have a button that will set the TV to a user's preferences with a single operation. Can your design support that? Discuss

your answer with your neighbours and/or tutor. Optional extra: if your design can support this extension then modify your class diagram and Java code accordingly to demonstrate how it works.

3. While listening to music, Ron likes to have access to information about the artist and recording. He is writing a Java app, RonsMusicCompanion, to meet all his needs. Although he only listens to vinyl, his sound system can play in any room and his app will run on his laptop so he can take it with him to wherever he is listening. Although his plans are ambitious, Ron has (wisely!) decided to start small and gradually add more features. At the moment, all he has implemented is a feature for displaying the cover image for the album he's listening to. Currently, he has to select the album from a menu (eventually an AI will detect what is playing and automate the process). The images he uses come from Amazon and they sometimes take a while to load. Ron has written some code to display a message while the image is loading.
    a. Obtain the source code for the initial version of Ron's app (classes RonsMusicCompanion, TemporaryImageIcon and ImageComponent). This code is a slightly-modified version of an example from the *Head First Design Patterns* book. It has almost no documentation — we'll soon fix that!
    b. Compile and run the code. When you first select an album from the menu, you should see a message until the cover image arrives in a few seconds. If you subsequently select the same album, the corresponding image should be displayed without delay.
    c. Ron is pleased with his progress on the app and shows his code to Hermione. Their conversation goes like this:

    Hermione: This looks good. It's pretty rough in places and there's no tests or other documentation. However, I really like the way you've used a GoF design pattern in your design.

    Ron: Thanks! It really works well — I'm very pleased with it.

    Hermione: Just make sure you document it properly. Otherwise, you won't remember how it works in a few months when you get around to implementing the other features.

    Ron: You're always right — I'll do it now…

    Look at the code and identify the GoF pattern used. Add comments in the source code to indicate the rôles played by elements that participate in the pattern. Use plantuml to draw a UML class diagram — include notes and/or stereotypes to show how the pattern is applied. Discuss your work with your neighbours and/or tutor — do you agree?