Design Patterns

The Memento Design Pattern

Memento
Design Pattern

The Memento Pattern

It is sometimes necessary to capture the internal state of an object (the values of all it's data members), and then have the ability to restore the object back to that state at a later time. The classic case for the memento is when you want to provide the functionality to "rollback" one or more objects within your application to a previous state (i.e., "undo" and "redo"). However, if you were to provide undo/redo functionality *within* a domain classes, that would violate the Single Responsibility Principle (SRP) because that object already has a responsibility.

The Memento pattern is a design pattern that permits the current state of an object to be "externally" stored without breaking the rules of Encapsulation or the SRP principle. The domain object can be modified as required but can be restored to the saved state at any time.

To support an *undo* operation, an application must be able to revert to the state in which it existed before a user executed an operation. There are at least a couple ways you can implement this: Operation Reversal, and State Storage.

- Operation Reversal - If you can directly reverse an operation, undo is a simple matter of storing a list of executed operations, each with an associated reversal operation. For example, in a simple calculator that supports addition and subtraction, you can revert to a prior value by reversing the operation. If a user added 5 to an initial value of 3, undo is a simple matter of subtracting the same quantity (5) from the current value of 8.

- State Storage – Some operations are too complex or not even feasible to reverse. As an example, consider a game like checkers or chess. User make moves, which changes the state of the game. From time to time, a player might want to "undo" a poor move. There is no real operation here that can be reversed by another operation. The best approach here is to store the state of the game after each move. This stored list of game states is the *memento*.

Memento
Design Pattern

Memento Design Pattern Components

The Memento design pattern defines three distinct components: the Originator , a Caretaker, and the Memento.

- The Originator is some domain object that has an internal state (Automobile, Person, Order, RoadNetwork, etc). This state will change over time. Many versions of the originator's state may need to be saved, recording the information that makes up the object at key points in time when some condition or operation will change the state, resulting in potential new behaviors.
  - To support this, the Originator includes a method (i.e., "createMemento") that is used to generate a Memento object containing a snapshot of the Originator's current state. The Originator also includes a method (i.e., "loadMemento"), which accepts a previously created memento object and uses it to restores the Originator to a previously stored state.

- The Memento is used to hold the information from an Originator's state. The amount of information held is controlled by the Originator. The memento stores a *snapshot* of the internal state of the Originator at one point in time. Memento objects are maintained by the Caretaker.
  - To support this, the Memento should provide protection against change to the stored state by including a very limited interface with no means of modifying the values it holds.

- The Caretaker is responsible for saving the memento objects for later use, and for returning a selected memento back to the Originator. A memento object is a "black box" to the Caretaker. The Caretaker does not know what is in the memento, and it should not examine or manipulate the contents of a memento object. (Memento objects should be immutable to help assure this).

Memento

Design Pattern

Memento Components – Save State

[4] Latest memento

[3]

[2]

[1]

[0] Oldest Memento

Originator ("Automobile")

- State includes data such as Location, Destination, Speed, RPMs, Direction, etc.
- **From time to time, the current state needs to be saved. The current state is saved in a Memento object and provided to the Caretaker upon request**

Memento

- **The Memento object is created by the Automobile (Originator) and will be stored/managed by the Caretaker.**
- **The Memento object holds a snapshot of the Automobile's data values (i.e., the "state" of the originator).**
- Memento objects can be used by the Originator to reset its state to a previous moment in time.

Caretaker

- **The Caretaker is responsible for saving the memento objects**
- The Caretaker should not examine or modify the content of a Memento object

---

Memento Components – Restore State

[4] Latest memento

[3]

[2]

[1]

[0] Oldest Memento

Caretaker

- **The Caretaker will return a saved Memento object upon request. Memento objects can be used by the Originator to reset its state to a previous moment in time.**
- The Caretaker should not examine or modify the content of a Memento object

Memento

- **The Memento object is retrieved by the Caretaker and given to the Originator (Automobile).**
- **The Memento object holds a snapshot of the Automobile's data values (i.e., the "state" of the originator).**
- The Memento objects is used by the Originator to reset its state to a previous moment in time.

Originator ("Automobile")

- **From time to time, a previous state needs to be restored. The previous state is saved in a Memento object and is returned to the Originator upon request**
- State includes data such as Location, Destination, Speed, RPMs, Direction, etc.

Operation Reversal vs. State Storage

Providing "undo" functionality via the Memento pattern can use more memory resources (each memento object takes up some amount of memory) than would an operation reversal technique, where the object is reverted to a previous state by reversing the operations performed on it. As a result, operation reversal can be a more attractive option for implementing "undo".

In some cases, you can successfully reverse an operation to revert an object to a previous state. In many applications, operation reversal is a simple matter, guaranteed to reproduce the objects previous state exactly. Addition is reversed with subtraction, multiplication is reversed with division, a database INSERT operation can be "undone" be DELETEing the same record, etc.

The problem with operation reversal however, is that many operations do not produce the same result when the reverse operation is applied. Creating an undo capability for a language translation application via operation reversal is an example where reversing an operation does not always generate the previous state:

Original Text (English): I love learning new design patterns, especially Memento!

Translated Test (Serbian): Волим учење нових образаца дизајна, нарочито Мементо!

If this translation is "undone" by *reversing* the translation, the English results are:

I love learning new **forms of design**, especially Memento!

As you can see above, we *almost* get the same English text back, but with some minor changes as shown in bold. In this case, reversing the translation operation as an "undo" mechanism is not going to be effective if it is important that the "undo" return the object to its *exact* original state before the operation.

Memento

Design Pattern

Language translations are particularly difficult to translate and then reverse translate to re-acquire the original text due to subtle changes (and errors) made to the original information during the translation process, as can be seen below:

Original English: ***U.S. stocks drifted higher Friday as investors look to close the books on a quiet trading week.***

Polish Translation: Zapasy w USA płynął wyższy piątek, gdyż inwestorzy patrzeć zamknięcia ksiąg na cichej tygodnia handlu.
Retranslation to English: U.S. Stocks drifted higher Friday as investors look for quiet closure of trade week.

German Translation: US-Aktien getrieben höher Freitag als Investoren, die Bücher in einer ruhigen Handelswoche zu schließen suchen.
Retranslation to English: U.S. stocks pushed higher Friday as investors, looking to close the books on a quiet trading week.

Russian Translation: Американские акции дрейфовали выше пятницу инвесторы ищут, чтобы закрыть книг на тихой торговой недели.
Retranslation to English: U.S. stocks drifted higher on Friday, investors are looking to close the books on a quiet trading week.

Greek Translation: ΗΠΑ αποθέματα παράσυρε υψηλότερη Παρασκευή, καθώς οι επενδυτές φαίνονται να κλείσει τα βιβλία σε μια ήσυχη εβδομάδα συναλλαγών.
Retranslation to English: U.S. stocks drifted higher Friday as investors look to close the books on a quiet trading weeks.

Lithuanian Translation: JAV atsargos pasitraukė didesnis penktadienis, nes investuotojai žiūri į arti knygas apie ramioje prekybos savaitę.
Retranslation to English: U.S. stocks moved higher Friday as investors look to close the books on a quiet trading week.

In this situation, implementing an "undo" feature via reversing the original operation will not return you to the *exact* original state. Using the memento pattern here would guarantee a return to the original state.

Note: The execution of reverse operations to undo an action can be more CPU and time intensive than would storing and restoring an object's state.

Memento Example

Assume that we are planning to write a simple Checkers game application.

- Checkers is played by two players (or one player if playing against a computer. The players start at opposite ends of the board. One player has dark pieces, and one player has light pieces.
- They take turns moving their pieces diagonally forward from one square to another square. When a player jumps over their opponent's (the other player's) piece, he takes that piece from the board.
- If a player's piece moves into the King Row on the other player's side, it becomes a king, which can move forward and backward.
- The first player to lose all their pieces, or who cannot make a legal move loses.

If we would like a player in our game to be able to “undo” a bad move, we will need a way to revert the state of the game back to a previous point in time. There is no real “calculation” that could be reversed in this situation, so the operation reversal technique is not applicable here. In this case, implementing the memento pattern will give us the undo (and redo) capability that we are looking for.

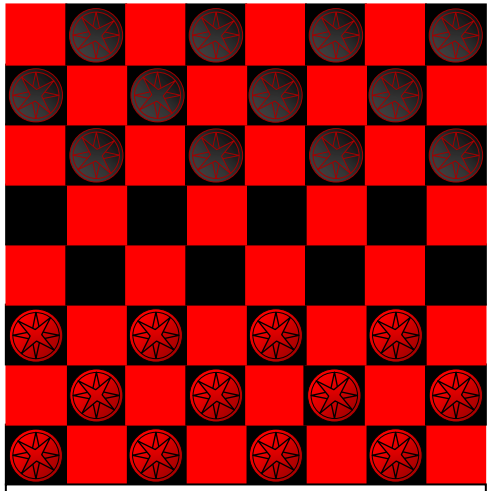
Memento

Design Pattern

Checkers Memento Example – Undo a Bad Move

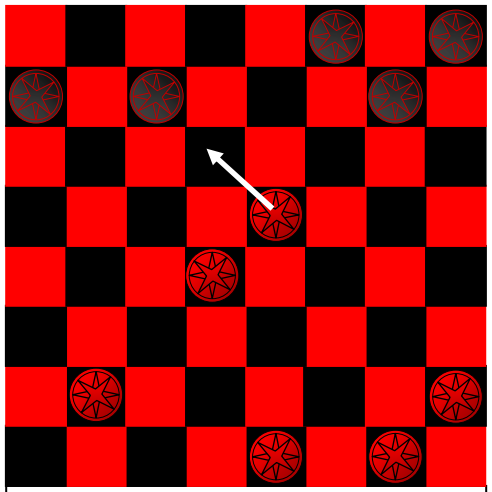
- In the first diagram below (left), you can see the initial Checker Board *(the BLACK pieces start at the top of the board, the RED pieces start at the bottom of the board).*
- The center diagram shows the state of the Checker Board after several rounds of play. At this point, the red-checker player makes a particularly bad move to the upper-left, as shown.
- The third diagram below (right) shows that as a result of the previous (bad) red-checker move, the black-checker player can now multi-jump 3 red pieces and in the process will reach the opposite side and become a King piece.

The red-checker player, realizing the mistake, would like to "undo" their last move (returning to the point just before the red move shown in the center diagram).



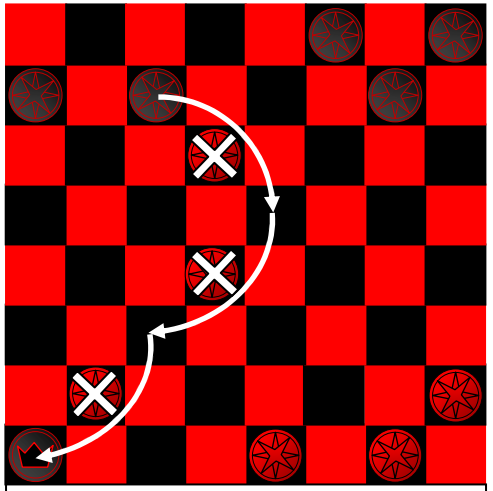
Initial Checker Game Board *(BLACK pieces on the top, RED pieces on the bottom)*

No moves made.



Checker Game Board after several turns.

At this point, the RED player makes a poor move.



In the next move, the BLACK player multi-jumps 3 RED pieces, and in the process reaches the opposite side and becomes a King!

Memento
Design Pattern

Checkers Memento Example – The Checker Game State

In this example, the state of the checker board (location of pieces in play) and the current player (red or black) make up the data that changes each time a move is made. Each time a move is made, the state of the board and the current player change. It is these changes we would like to be able to "undo". The following shows a sample of how the checker board and current player might be stored in this sample application:

```
package memento;

public class CheckerGame {

    // Use this enum to create a RED and BLACK type.
    enum CheckerColor {

        RED, BLACK
    }

    // Declare a constant for the board width (avoids hardcoding 8's everywhere)
    public static final short BOARDWIDTH = 8;

    // Define the board as a 2 dimensional array of CheckerColors (all null initially)
    private CheckerColor[][] checkerBoard = new CheckerColor[BOARDWIDTH][ BOARDWIDTH];

    // Define the board as a 2 dimensional array of CheckerColors
    private CheckerColor currentPlayer = CheckerColor.RED;

    …
}
```

Here, the CheckerGame object is the Originator – our domain object with an internal state that will change over time. After each move, the "checkerBoard" array and the "currentPlayer" value will change. To revert the game to a previous state, "checkerBoard" and "currentPlayer" must be reverted to a previous value, so it is these data elements that will need to be saved in a memento object.

Memento

Design Pattern

Checkers Memento Example – The Checker Game Memento

The below shows the Memento class for the CheckerGame. The state of the CheckerGame object at one moment in time will be stored using these Memento objects. Note that values held here can be *set and accessed* but not *modified* (i.e., immutable).

```
package memento;
public class Memento {

    // This will hold a snapshot of the checker board's state
    private CheckerGame.CheckerColor[][] boardSnapshot =
            new CheckerGame.CheckerColor[CheckerGame.BOARDWIDTH][CheckerGame.BOARDWIDTH];

    // This will store the current player (RED or BLACK)
    private CheckerGame.CheckerColor playerSnapshot;

    // Constructor accepts and sets up the shapshot state values
    public Memento(CheckerGame.CheckerColor[][] boardIn, CheckerGame.CheckerColor playerIn) {

        // Copy the information from the checker board passed in into the snapshot
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                boardSnapshot[i][j] = boardIn[i][j];
            }
        }

        // Save the current player
        playerSnapshot = playerIn;
    }

    // This method allows access to the checker board square values but does not allow modification
    public CheckerGame.CheckerColor getSquareContents(int x, int y) {
        return boardSnapshot[x][y];
    }

    // This method allows access to the current player value but does not allow modification
    public CheckerGame.CheckerColor getPlayer() {
        return playerSnapshot;
    }
}
```

Checkers Memento Example – Creating Mementos

The Memento Pattern states that the originator must be able to *create* and *load* memento objects, so the following methods need to be added to the CheckerGame class:

```
package memento;

public class CheckerGame {
    …

    // This method will create and return a Memento object reflecting
    // the current state of the CheckerGame object
    public Memento createMemento() {
        return new Memento(checkerBoard, currentPlayer);
    }

    // This method accepts a Memento object parameter and uses that
    // object to re-set the state of the CheckerGame object
    public void loadMemento(Memento data) {
        // Copy the values from the checker board snapshot "data"
        // into the "checkerboard" array.
        for (int i = 0; i < BOARDWIDTH; i++) {
            for (int j = 0; j < BOARDWIDTH; j++) {
                checkerBoard[i][j] = data.getSquareContents(i, j);
            }
        }

        // Set the "currentPlayer" to the player value held in "data"
        currentPlayer = data.getPlayer();
    }
}
```

Upon request, a CheckerGame object can return an immutable Memento object reflecting its current state (via "createMemento"). When instructed to do so, the CheckerGame object can reset its state to the values held in the provided Memento object (via "loadMemento").

Memento

Design Pattern

Checkers Memento Example – The Caretaker

The Caretaker (shown below) is responsible for saving Memento objects (in its "mementos" ArrayList) and for returning a specific Memento object upon request:

```
package memento;

import java.util.ArrayList;

public class Caretaker {

    // This ArrayList will save "n" Memento objects (in the order in which they were saved)
    private static ArrayList<Memento> mementos = new ArrayList<>();

    // A private c'tor insures no instances will be created. All data & behaviors are static.
    private Caretaker() {
    }

    // This method accepts a CheckerGame object parameter, then saves the memento object
    // generated by that CheckerGame in the "mementos" list.
    public static void addMemento(CheckerGame cg) {
        mementos.add(cg.createMemento());
    }

    // This method accepts a CheckerGame object and an int parameter. The int is used as an index
    // into the "mementos" list. The Memento object retrieved from the "mementos" list is passed
    // to the CheckerGame's "loadMemento" method.
    public static void restore(CheckerGame cg, int idx) {
        cg.loadMemento(mementos.remove(idx));
    }

    // This utility method returns the number of Memento objects currently held by the caretaker.
    // This value is useful in determining which Memento object should be returned.
    public static int numSavedStates() {
        return mementos.size();
    }
}
```

Memento
Design Pattern

Checkers Memento Example – Game Play

This (very) simple "main" shows how game play works. The user enters their next move, and that move is passed to the CheckerGame's "makeMove" method. The "makeMove" method returns a value indicating the results of the move. While the game is not over, play continues in this fashion.

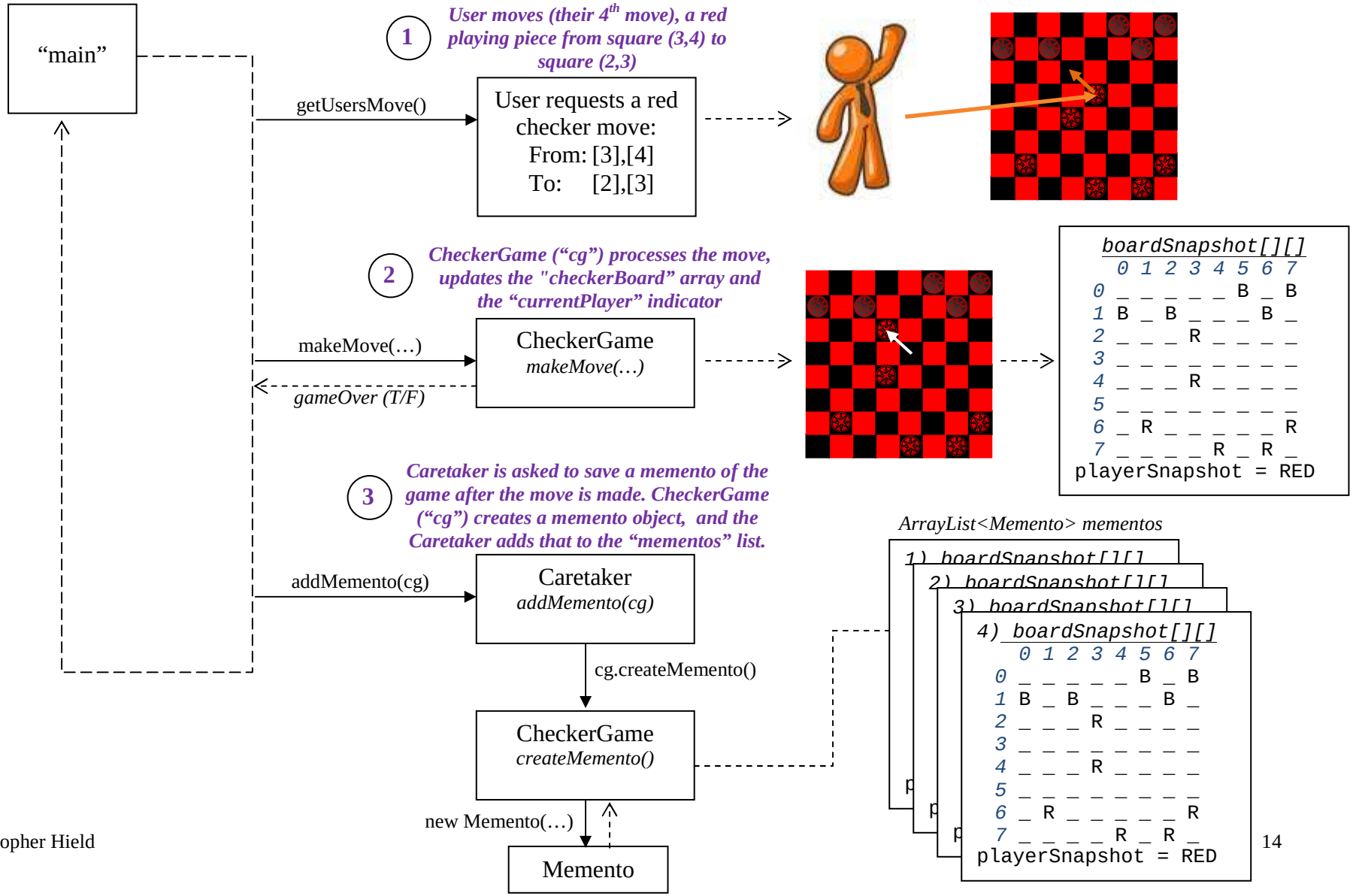
Note that the Caretaker is used to save the initial state of the checker game, and the state of the game after each move is made.

```
public static void main(String[] args) {

   // Create a new CheckerGame object
   CheckerGame myCheckerGame = new CheckerGame();

   // Pass the new CheckerGame object to the Caretaker, which will
   // save the state of the game (the start-state in this case).
   Caretaker.addMemento(myCheckerGame);

   Result moveResults = CONTINUE; // Holds move result values (defined in an enum)
   while (moveResults != GAMEOVER) {
      // Get the user's next more
      UserInput ui = getUsersMove();

      // Make the move, save the Boolean indicator
      moveResults = myCheckerGame.makeMove(ui);

      if (moveResults == UNDO)
          // The user wants to undo their last move, so ask the Caretaker
          // to "restore" the CheckerGame to a previous state. Caretaker will
          // retrieve a Memento object and pass it to the CheckerGame for loading
          Caretaker.restore(myCheckerGame);
      else if (results == GAMEOVER)
          processGameEnd();
      else
          // Pass the new CheckerGame object to the Caretaker, which will save the
          // current state of the game (after the user's move is made).
          Caretaker.addMemento(myCheckerGame);
   }
}
```

Memento

Design Pattern

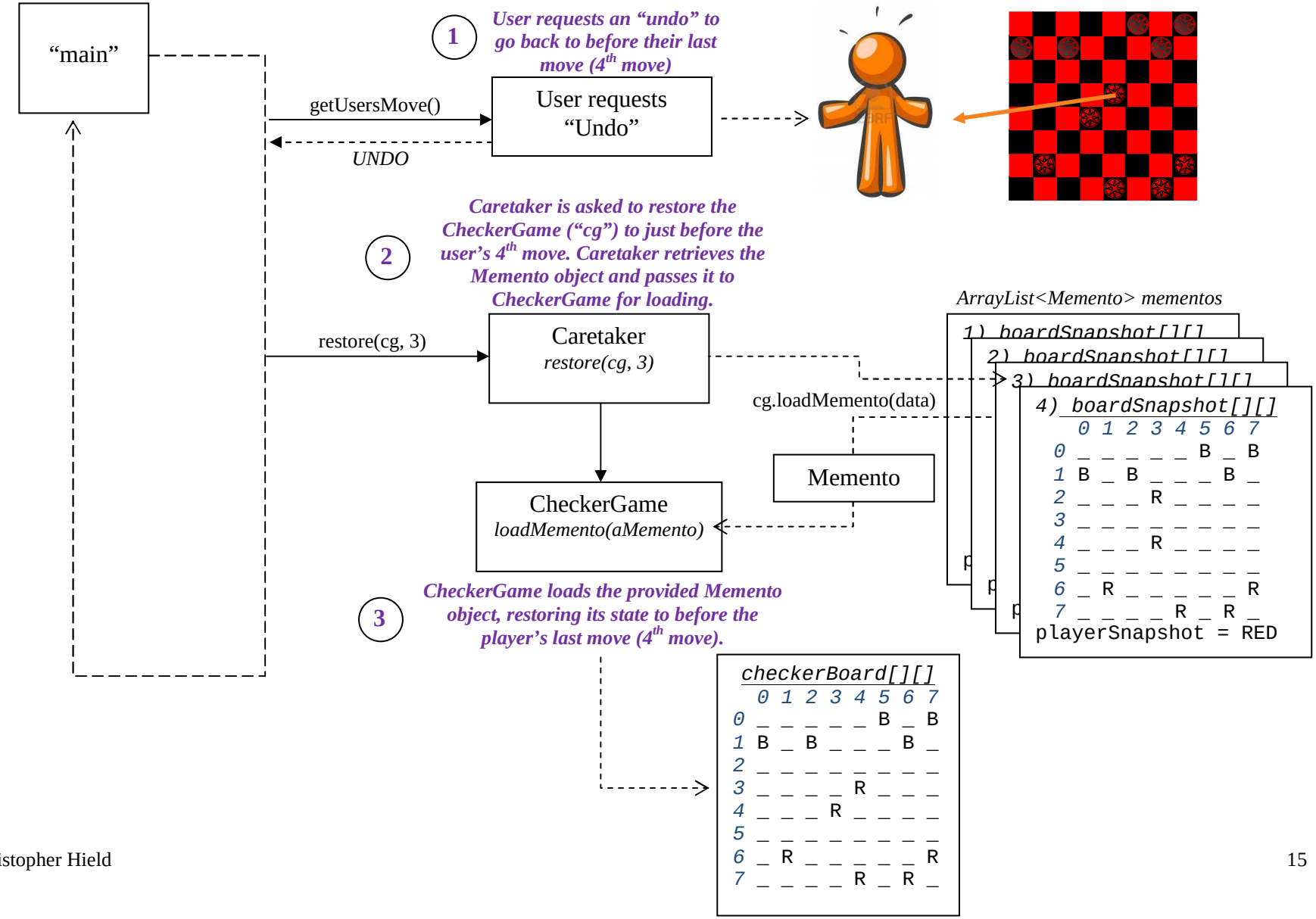
Checkers Memento Example - Game Play with *Memento Storage* Walkthrough



Memento

Design Pattern

Checkers Memento Example - Game Play with *Undo* Walkthrough



Memento

Design Pattern

Summary

- The memento design pattern provides a simple way to manage different versions of a class, thus you can implement undo/redo behavior without breaking encapsulation or violating Encapsulation or the Single Responsibility Principle.
- The Memento pattern shields other objects from potentially complex internal state of the Originator
- If the Originator is maintaining versions of its internal state, the Memento pattern keeps Originators simpler
- Creating and restoring state may be expensive. Memento may not be appropriate when state contains a large amount of information
- It is difficult to ensure that the Originator can be the only one who can access the Memento's state
- The Caretaker will be managing Mementos, but has no idea how big or small are they.