JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2009**

Vol. 8, No. 5, July-August 2009

The Application of Design Patterns to Develop Games for Mobile Devices using Java 2 Micro Edition

J. Narsoo, School of Business Informatics and Software Engineering, University of Technology, Mauritius
M. S. Sunhaloo, School of Business Informatics and Software Engineering, University of Technology, Mauritius
R. Thomas, School of Business Informatics and Software Engineering, University of Technology, Mauritius

Abstract

In this paper, we demonstrate the use of design patterns to develop games for mobile devices on the J2ME platform. We believe that the proposed idea will help J2ME game developers to write better re-usable code faster. We consider a single player Sudoku board game which is based on the model-view-controller architecture. The view is configured with a Game Controller Choice pattern so that different controllers can be selected. The view and the model implement the Game State Observer pattern. The Canvas drawing logic is created using the Drawing Template pattern, which ensures re-usability of the board size computation as well as provides a means for the programmer to implement game specific drawing functionalities. A generic undo method is provided using the Game Memento pattern. Setting of the current display is achieved through the Change Screen pattern. We note that the use of patterns in the Sudoku game makes it possible to cater for changes without breaking up the overall architecture of the game. By considering a downgraded version of the standard Soduko game, we show that patterns allow modifications to be made without opening up too many classes. We also show how the proposed design of the Sudoku game facilitates the design of other games.

1 INTRODUCTION

In the early 1970s software was developed using structural paradigms, which separated procedures or functions from the data. It was difficult to write re-usable software components using the procedural paradigm. The object oriented approach overcame this drawback [3].

J. Narsoo, M. S. Sunhaloo, R. Thomas: "The Application of Design Patterns to Develop Games for Mobile Devices using Java 2 Micro Edition", in *Journal of Object Technology*, vol. 8, no. 9, July-August 2009, pp. 153-175 http://www.jot.fm/issues/issue_2009_07/article4/

Nowadays, the object oriented paradigm has become the norm. However, with the continued increase in size and complexity of software systems, the concept of design patterns has emerged to improve software architectures [2,8]. And today, design patterns are providing generic solutions to commonly occurring problems in software development [7]. A design pattern describes a design problem, the solution and the consequences [3].

Recently, the mobile gaming industry has experienced a tremendous growth. As a result, development of new games may lead to recurring similar design problems. Thus, design patterns can be used to optimize the design and implementation of games for mobile devices [5]. For example, design patterns can provide facilities for creating a new board in a board game, for varying the board dimensions, for maintaining and saving the game state, for computing the next move, for rendering the display screen and for managing record stores. Solving these problems using first principles, by duplicating codes and solutions may not be appropriate, thus leading to the development of monolithic, inflexible, difficult to maintain, inefficient software with undesirable features [1].

While mobile games can be developed using different technologies such as Windows mobile platform, Flash, Python, among others, the Java 2 Micro Edition (J2ME) platform has been considered. J2ME is a dominant platform in game development for mobile devices. In this paper, the Gang of Four patterns [3] are used in the context of J2ME game programming to build scalable, maintainable and robust software. The Sudoku puzzle and the N-Puzzle, which are single player board games, are selected to illustrate the application of design patterns to develop games. We note that these games can be modified into multiplayer games by enabling different players to independently solve the same puzzle, for example over a wireless network. We have opted for a single player game so as to focus on the patterns in the design of the game rather than on the advanced game functionalities.

2 THE SUDOKU GAME

The Sudoku game is a single player board game with 81 square cells distributed as 9 rows and 9 columns. These 81 cells are divided into 9 boxes with each box containing 9 cells.

Random numbers ranging from 1 to 9 are placed on the board. This arrangement is what is known as the Sudoku puzzle. The aim of the game is to fill the board with numbers from 1 to 9, in such a way that, there exists only one instance of the numbers 1 to 9, in every column, row and box on the board. It is interesting to note that every Sudoku puzzle has one and only one unique solution [4].

The player is given an unsolved puzzle and is required to solve it by filling the empty cells with relevant numbers. There are various techniques to solve the Sudoku puzzle but in this paper credit is given to the design aspect of games rather than these techniques.

The important aspect of playing this game is that the player uses logical deductions to make a move. The player should not guess the number that has to be placed in a particular cell. Every puzzle number can be derived by analyzing the puzzle's current play state and applying logical deductions.

The implementation of the Sudoku game involves two stages. Firstly the generation of the Sudoku puzzle making sure that only one solution exists and secondly solving of puzzle. Sudoku puzzle generation and solving is an NP complete problem [4]. There exists no, known best algorithm for its generation. Different implementers use different techniques for generation. All known generation techniques take considerable time and so it would not be advisable to implement a generation module for the Sudoku game on a J2ME enabled device. This is because of the low processor power and memory constraints. The implemented game, has a database of generated puzzles along with their solutions, the player can solve the puzzle which will then be validated against an existing solution.

3 GAME DESIGN

The main components of the Soduku game are the model, view and controller components, which together make up the Model-View-Controller (MVC) pattern [9] as shown in Figure 1.

These components are the main parts of the application. The MVC architecture makes it possible to loosely couple the graphical representation from the core logic of the game. There could possibly be more that one view in an application. The controller uses the MenuFactory component to create Displayable components, which are objects that can be displayed on the device screen. The model makes use of the PuzzleStore and Storage classes. The PuzzleStore component contains a list of unsolved puzzles and their solutions. When a new puzzle is requested by the player, the model generates a random number and uses this to select the puzzle that is stored in the puzzle list. By decoupling the puzzle list from the model, the design ensures that the PuzzleStore can be extended for changes without affecting the other components. For example it is this design that makes it possible to extend the Sudoku game to download the puzzles via bluetooth or wireless mobile device.

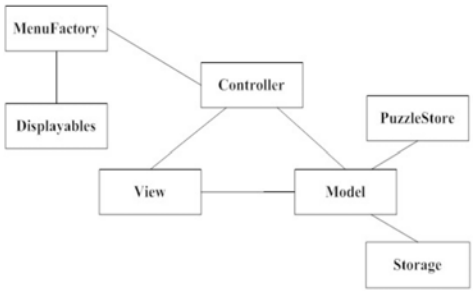


Figure 1 – The Model-View-Controller Architecture.

Model

The model is represented by the SudokuGameModel class. This class implements the GameModel interface. It is important to represent the model in an abstract form so that it can be re-used in other games. The GameModel interface declares common functionalities of single player board based games that have to be implemented by Concrete classes. The GameModel interface listing is given as follows:

```
1 public interface GameModel
2 {
3 Object createMemento ( ) ;
4 void setMemento ( Object memento ) ;
5 void r e g i s t e r (Canvas view ) ;
6 void i n i t i a l i z eMo d e l ( ) ;
7 void selectGameData (boolean savedGameExists ) ;
8 void setGameLevel (byte l e v e l ) ;
9 byte getGameLevel ( ) ;
10 byte getBoardDimension ( ) ;
11 void set PuzzleState ( int row , int col , byte value) ;
12 byte [ ] [ ] g e tPuz z l eSt a t e ( ) ;
13 void saveGame ( ) ;
14 byte [ ] getOriginalGameData ( ) ;
15 boolean isValidMove ( int indexOfDataValue ) ;
16 boolean i s S t a t e F i l l e d ( ) ;
17 boolean isGameOver ( ) ;
18 public boolean doesSavedGameExist ( ) ;
19 }
```

Lines 3 and 4 are the methods of the memento pattern. These abstract methods do not refer to any concrete class implementations. The Object class of the Java platform is used to represent concrete objects polymorphically. This enables designers to use the model interface for other games. Lines 6 to 17 are required for game playing and for the controller and view classes to interact with the model.

The concrete class SudokuGameModel stores the game state and contains game specific methods for the Sudoku game. It implements the GameModel interface.

The state of the Sudoku Game is a two dimensional byte array that represents the puzzle numbers. In addition the SudokuGameModel also encapsulates the index of the current puzzle selected from the list of stored puzzles. The model uses a number of

classes, which are required to provide functionalities to the overall game. These include the PersistentStorage, SudokuPuzzleStore and SudokuMemento classes. The PersistentStorage class provides functionality to save an unfinished game. The SudokuPuzzleStore contains a list of puzzles along with their solutions. The SudokuMemento captures the game state.

View

The GameView, BoardView and Cell classes together represent the View part. GameView class extends the J2ME abstract Canvas class.

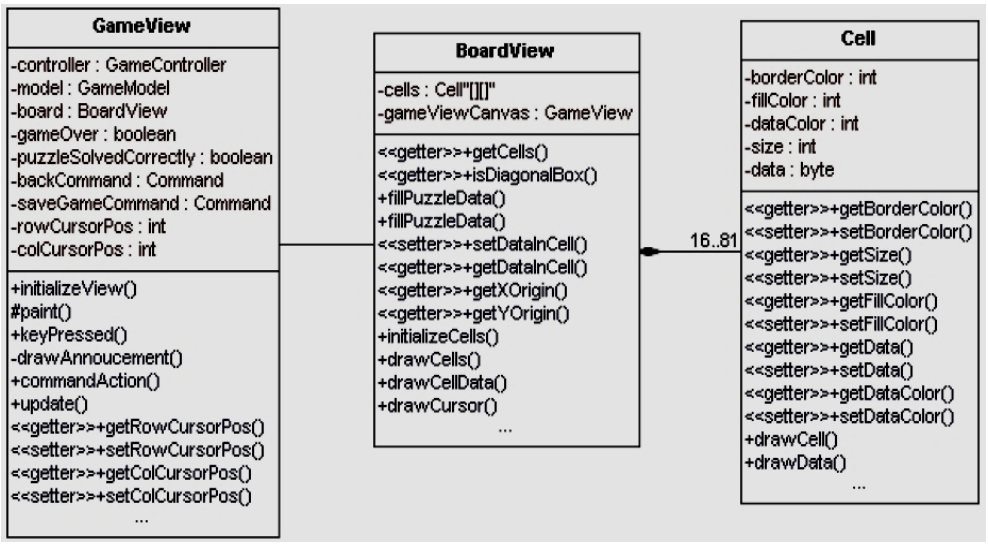


Figure 2 – View.

The GameView class is responsible for painting the screen and for capturing player moves and delegating it to controller. It contains a reference to the BoardView class. The BoardView class contains the drawing functionalities like, drawing the board, puzzle numbers and the cursor. The BoardView aggregates the cells whose number depends on the dimension of the board, which in turn is a state of the model. The relation between the BoardView class and the Cell class is illustrated in Figure 2.

The draw method of the BoardView class contains a number of steps and the design of this draw method is opened for extensions. Choosing this way of design makes the painting of the Canvas simpler, as shown in the following code.

```
1 protected void paint ( Graphics g )
2 {
3 i f ( ! isGameOver ( ) )
4 {
5 board . draw( this , g ) ;
6 }
```

```
7 el se
8 {
9 drawAnnoucement ( ) ;
10 }
11 }
```

In Board games, a board is divided into a number of cells which contains attributes that represent the border color, fill color, the size of the cell and data to be displayed inside the cell. In the Sudoku game, the data are the puzzle numbers. The GameView class registers itself with the model. When the state of the model changes the GameView class receives notifications from the model to update itself.

Controller

The GameMidlet Class, the main class or the entry point, together with the SudokuGameController class constitute the controller part of the architecture. The SudokuGameController class is programmed to the interface GameController. This has the advantage that the GameMidlet class does not need to reference a concrete implementation, meaning that if there are any changes to be done to the SudokuGameController class, it will not affect the GameController class. This adheres to the principle of loosely coupled and highly cohesive modules. The SudokuGameController class could be replaced with another concrete implementation of the GameController interface, without affecting other classes. The SudokuGameController class functionalities could have been directly included in the midlet class itself, but that would have made the design too monolithic. It would also have resulted with the midlet class being overburdened with multiple responsibilities. It is desired, in the interest of good design, that every class is responsible for one operation. Though this may lead to many number of classes in the design, but ultimately a balanced has to be reached. In this design, the SudokuGameController class was necessary so that the game control logic could be separated from the overall midlet management. Thus, the midlet class manages the game as a whole, interacting with the java virtual machine and the SudokuGameController class controls the user actions with respect to the game. The GameController interface is as shown below.

```
1 public interface GameController
2 {
3 public abstract MenuFactory getMiFactory ( ) ;
4 public abstract void handleKeyEvent (int keyCode, int rowCursorPos , int colCur
sorPos );
5 public abstract void handleSaveEvent ( ) ;
6 abstract void doCleanup ( ) ;
7 }
```

The controller classes and their relationship is shown in Figure 3.

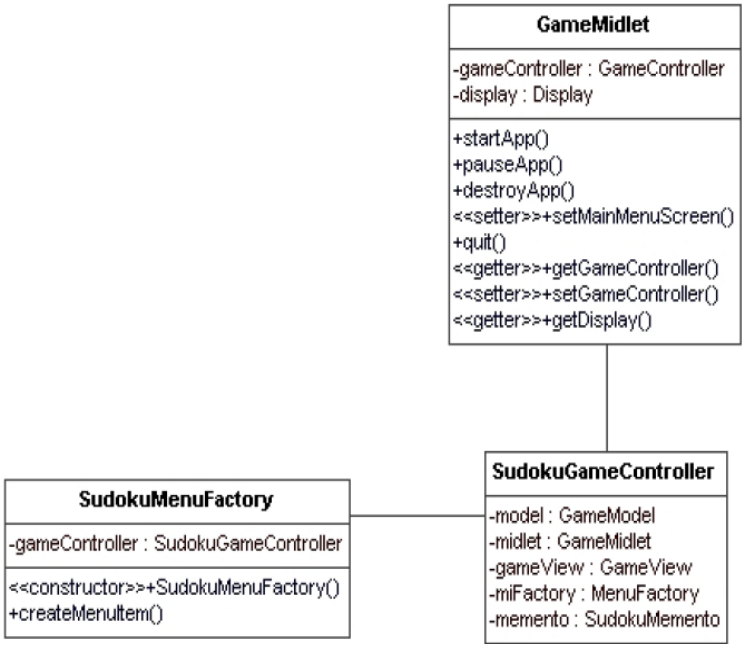
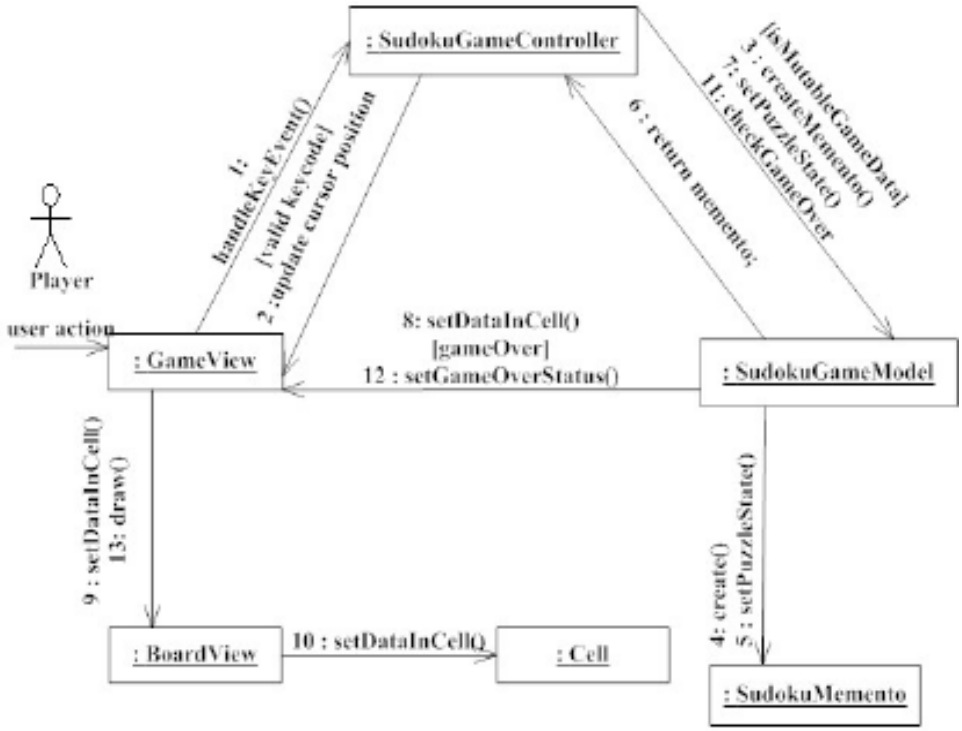


Figure 3 – The Controller Classes.

4 MVC COLLABORATIONS

The collaboration diagram for the model view and controller objects is shown in Figure 4.



An example scenario to illustrate the collaboration among the different classes is when the player enters a number in an empty board cell. The sequence of events is as follows:

1. The player presses a key and the view calls the handleKeyEvent method of the controller.
2. If it is a valid keycode, the controller updates the cursor position variables in the view class.
3. The controller then checks whether the board cell data can be modified. This is necessary to prevent the player from overwriting the original puzzle numbers. If the data can be written, the controller asks the model to create a memento object.
4. The model creates the memento object as requested by the controller. In this step, the model saves its state in the memento object.
5. After creating the memento object, the model returns it to the controller.
6. The controller asks the model to change its state. The model takes the number entered by the player and assigns it to the particular index of the array that represents the model's state.
7. The model notifies the view of the change in its state.

8. The GameView asks the BoardView to update the cell.
9. The BoardView delegates the updating of the data in the cell, to the relevant Cell object.
10. The controller checks if the game is over by checking the model's state.
11. If the game is over, the model notifies the view about it.
12. The GameView asks the board to draw itself on the screen.

5 DRAWING THE SCREEN

The repaint method is used to redraw the Canvas. The J2ME framework provides two overloaded methods for this purpose. The first one can repaint the entire screen and the second, which is more efficient can be used to repaint part of the screen by specifying the location, width and height [10]. The entire display screen does not need a repaint in this game. For example, as shown in Figure 5, if the cursor is at the top left cell, and if the player moved the cursor to the right, the only area to be repainted is that comprising of the cells where the cursor was previously and the new position of the cursor.

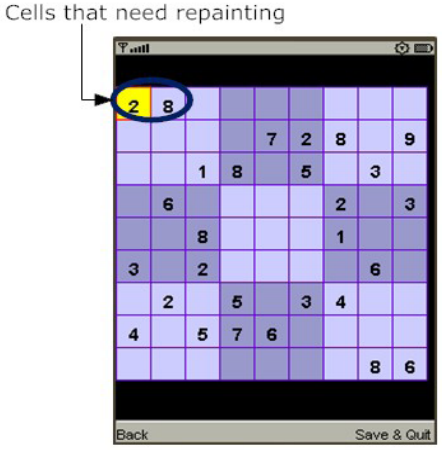


Figure 5 – Soduku Board.

The area to be repainted is specified as follows. The cell under which the cursor is currently displayed is known and the area of this cell is computed. Then, in another call to the repaint method, the area of the cell, where the cursor is to be moved is calculated. The repaint method takes the total area specified in the multiple repaint method calls. This effectively reduces the size of the screen to be refreshed.

When the state of the model changes the view is also updated by calling appropriate view methods. This should have the effect of the view being re-drawn instantaneously, to show the change in the model's state. In J2ME game programming, drawing the screen is achieved by the paint() method which is defined in the Canvas class. To be able to use it, the game developer needs to subclass the Canvas class and implement the paint() method. The platform determines, for efficiency reasons when the screen needs to be repainted [10]. Therefore it is not possible for the SudokuGameModel class to directly make a call

to the paint(), to update the screen. To tackle this problem, the model updates the view as its state changes before the second call to the repaint() method is encountered.

6 PERSISTENTSTORAGE CLASS

The Sudoku game allows saving an unfinished game and reverting back to the saved game. This functionality is implemented in the PersistentStorage class. It uses the RecordStore class. The latter implements the Singleton pattern. The RecordStore provides storage and retrieval methods that act upon a single dimensional array of bytes. The state of the Sudoku game is maintained in a two dimensional byte array. Therefore there is the issue of converting the two dimensional array into a single dimensional array, before the puzzle state is written to a record and vice-versa when the saved puzzle has to be retrieved from the record. Another consideration is that there is a need to store the puzzle number, so that it can be checked with its corresponding solution when the puzzle is being played.

Currently the game allows a single puzzle to be saved. Therefore if a saved puzzle exists, we overwrite it with the new puzzle to be saved. The puzzle number is stored as the last byte in the array.

7 RANDOM PUZZLE SELECTION

The functionality to generate a random number with an upper limit is available in MIDP 2.0. To enable a larger number of devices to use the game, the configuration is set up for MIDP 1.0.

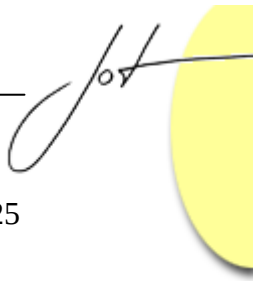
8 FILE FORMAT

The StandardSudoku class stores a number of generated Sudoku puzzles along with their solutions. The puzzles are stored in a specific format that aids adding new puzzles. In deciding on the format of the file to represent the puzzles, a simple approach is needed. This enables users of the game, to add additional puzzles or delete puzzles from the list of stored puzzles. The puzzle numbers are represented as a sequence of strings starting from number that should appear in the first cell up to the last cell of the board. For every puzzle, the unsolved puzzle, along with its solution are stored in separate arrays.

An example of an unsolved puzzle is as shown.

064020000003409000090010506400092010200000005030540002705080030000201900000030250

An example of the same puzzle with the solution is as shown.



164825379573469128892317546457692813289173465631548792725984631346251987918736254

A zero indicates an empty cell. This is later used in the model to validate for a valid player move. The game over status can be checked by comparing the puzzle state maintained by the model to that of the stored solution.

9 J2ME CONFIGURATION AND PROFILE

Applications programmed on the J2ME platform are called midlets. All applications programmed on the J2ME platform contain a class that extends from the abstract class MIDlet [12]. In the Sudoku game, the GameMidlet class provides the functionality of a MIDlet class. The Sudoku game is compatible with CLDC 1.0 and MIDP 1.0.

10 GAME PATTERNS

The following sections describe the patterns which we have used in the Soduko game [6].

Change Screen

Intent

The Change Screen pattern, as shown in Figure 6, defines an interface that enables changing of the current Display item in a MIDlet. The actual Displayable item is selected by the subclasses.

Motivation

In game development on the J2ME platform, providing a standard menu interface is very common. When the midlet starts, the user is given a series of options to choose in the form of a menu. Depending on the user's selection, it is required to change the current displayable item to what the user has selected in order to change the screen. It is first necessary to create the displayable item. The Change Screen pattern provides a solution to create these different displayable items at runtime and then changes the display screen to the user's choice.

Applicability

The Change Screen pattern is used when the actual item to be displayed is not known till runtime.

Structure

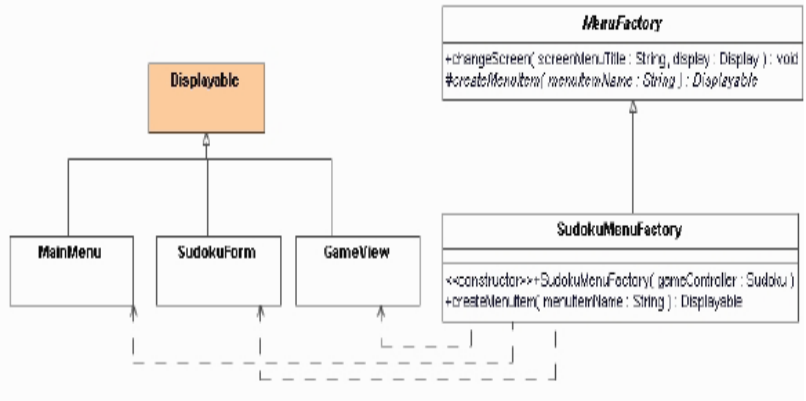


Figure 6 – Change Screen Pattern.

Participants

- Displayable
  - abstract class that groups together all Displayable items.
- MainMenu, SudokuForm, GameView
  - concrete classes that inherit from Displayable.
- MenuFactory
  - abstract class that declares a change screen method. Calls the change screen method to create a Displayable object that has to be used as the current Display object.
- SudokuMenuFactory
  - overrides the createMenuItem method to create and return an instance of a MainMenu, SudokuForm or GameView object.

Collaborations

The MenuFactory relies on the SudokuMenuFactory to return a concrete instance.

Implementation

The Displayable class represents all objects that can be displayed on the screen. This class is part of MIDP and is defined in the package lcdui. There is no need for it to be defined by the user. The name of the object to be created is passed as a String.

The pattern allows for the creation of a generic menu. By allowing MainMenu, SudokuForm and GameView classes to be known by a common abstract class, the implementation is not bound to a concrete implementation of a Displayable. The user is free to extend this pattern and to add other concrete classes. Letting the subclass decides the concrete instantiation, also allows the attachment to another MenuFactory if requirements change.

Game Momento

Intent

This pattern as shown in Figure 7, captures, stores the state of the game model and can revert the model back to the previous sate.

Motivation

In many scenarios, it is necessary to save the state of a game, so that the player can revert back to particular playing level or just undo the latest move made. This might be possible by implementing the logic into the core application itself, but this will result in high cost in the maintainability of the code. It is probable that encapsulation will be violated. The Game Memento pattern provides a way to keep track of an object's internal state and at the same time provides an undo mechanism.

Applicability

The Game Memento pattern is used when there is a need to undo changes made to an object's state or to revert back to a previous game level.

Structure

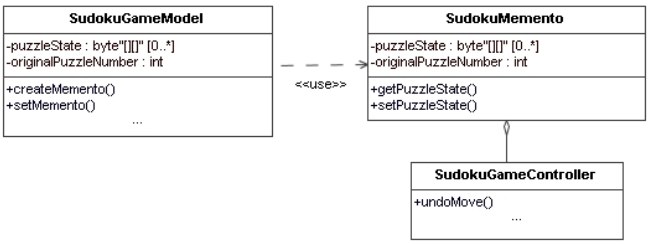


Figure 7 – Game Memento Pattern.

Participants

- SudokuMemento
  - stores the internal state of the SudokuGameModel object.
- SudokuGameModel
  - creates a snapshot of its state and also uses this stored state to restore its state.
- SudokuGameController
  - provides the undo mechanism.

Collaborations

When the user makes a move, the SudokuGameController requests a memento from the SudokuGameModel, which creates the memento and returns the instance back to the SudokuGameController. If the player requests for an undo, the SudokuGameController passes the memento object to the SudokuGameModel, which uses it to revert back to its previous state.

Implementation

Capturing the current game state is provided by the SudokuGameController. This is because the controller is responsible for decoding the player input for the model.

When using the memento pattern there is also a trade off in terms of efficiency. For every move that the user makes, a SudokuMemento object is created by SudokuGameModel, and this object must be returned to the SudokuGameController. To limit this potential overhead, the game has been implemented to support a single undo level.

The SudokuGameController holds a reference to the memento object. In the handleKeyEvent method, the key code for undo is checked before calling the undo method. Similarly, a snapshot of the originator object is created. Care should be taken to first create the snapshot of the state and then update the SudokuGameModel's state with the new value entered by the player otherwise the state captured will be the same as the values by which the model was updated.

Game Controller Choice

Intent

The Game Controller Choice pattern, as shown in Figure 8, defines a unified interface for concrete game controller classes so that they can be interchangeable.

Motivation

In the model-view-architecture, the view delegates to the controller the action that needs to be taken for any user's input. The view should be independent of the controller. There could be more than one controller or that future enhancements require building a completely new controller. By implementing the Game Controller Choice pattern, the view is decoupled from the model and it is possible to modify the controller for future changes without the need to change the view class.

Applicability

The Game Controller Choice pattern is used when there is a need to choose from different game controllers or when a decision is to be made to select an object from different objects implementing the same interface.

Structure

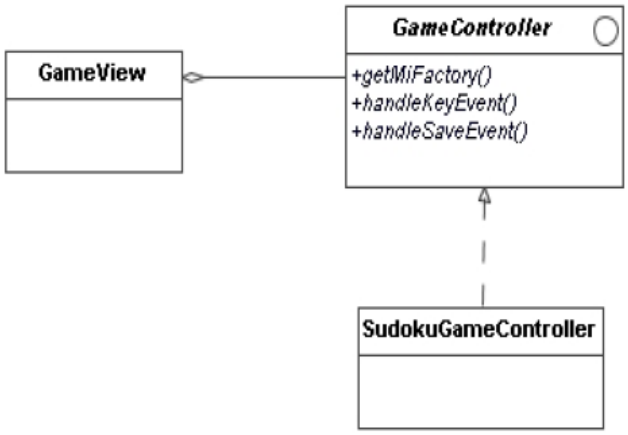


Figure 8 - Game Controller Choice Pattern Structure.

Participants

- GameController
  - the interface that defines common functionalities of controllers.
- SudokuGameController
  - concrete controller that implements the GameController interface.
- GameView
  - the context that is configured with a concrete controller object.

Collaborations

GameView and the SudokuGameController interact together to select the controller to be used to process user requests.

Implementation

The GameView class extends the Canvas class. The GameController interface is programmed to define the control actions necessary when the user does some action like moving to the next cell in the board, entering data, saving and exiting the game.

The SudokuGameController implements the GameController interface. We can substitute another game controller if requirements change. The GameView need not be aware of the changes done.

Game State Observer

Intent

The Game State Observer pattern, as shown in Figure 9, provides a means to update dependent objects when the state of the game model changes.

Motivation

In many cases it becomes necessary to separate the visual display part of an application from that of the business logic or core logic. Essentially the idea is to decouple them, so that changes in the model can be done without modifying the display classes and vice-versa. It is also possible that a business model or game logic may have more than one display elements, in which the data that is displayed relates to the same game logic. The observer pattern allows the decoupling of the display and the core game logic. When the state of the logic changes, the various display elements are notified of the changes and asked to update themselves.

Applicability

The Game State Observer pattern is used when the display part of a game has to be decoupled from the game logic.

Structure

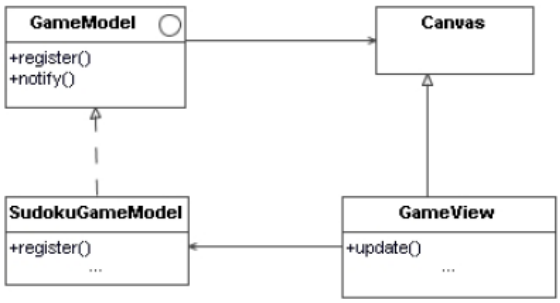
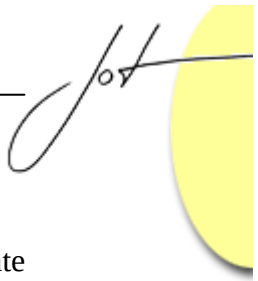


Figure 9 – Game State Observer Pattern.

Participants

- GameModel
  - defines an interface for all concrete implementations of a game model.
- SudokuGameModel
  - contains game model core state information.
  - notifies its dependents when its state changes.
- Canvas
  - defines an interface for game displays based on the Canvas class.
- GameView
  - concrete implemenation of an observer.
  - updates itself upon receiving notification from the SudokuGameModel.



Collaborations

The GameView registers itself with the SudokuGameModel. It may also request for state information. When the SudokuGameModel notifies the GameView about a change in its state, the GameView updates itself to reflect the new state.

Implementation

The Canvas class is an abstract class and need not be implemented by the programmer. The update method is user defined and is therefore not a member of the Canvas class. This implies that the programmer has to define the update method in the concrete GameView class which is used for presenting the state information. The programmer could have defined a new class and let the GameView class inherits from this new class. However this could have unnecessarily increased the total number of classes in the game, for no benefits.

Loose coupling is achieved between the model and the view due to the Game State Observer Pattern.

Drawing Template

Intent

Figure 10 shows the Drawing Template pattern. This defines a generic drawing algorithm which consists of several steps, but let subclasses define certain steps of the algorithm.

Motivation

The display screen has to be rendered with the visual representation. For example, the board has to be displayed on the screen along with the game data which could be puzzle numbers. The cursor needs to be displayed too, which indicates the current position of the player. Programmers can have their own way to draw the board, the game data and the cursor. By providing a generic algorithm for drawing purpose, programmers can extend this generic algorithm to suit the need of their particular game.

Applicability

The drawing template pattern is used to draw the visual components of the game onto the display.

Structure

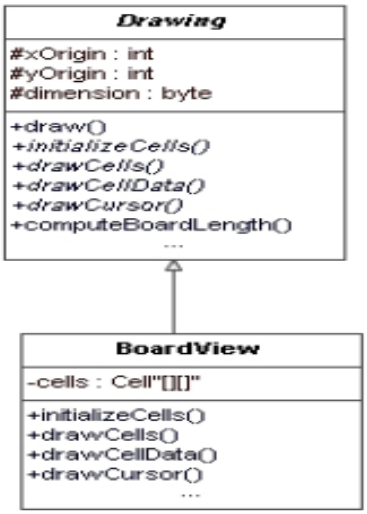


Figure 10 – Drawing Template Pattern.

Participants

- Drawing
  - implements the draw() method which defines a generic algorithm, with steps that can be overridden or implemented by subclasses.
  - contains abstract methods that subclasses implement to define the steps of the algorithm.
  - Drawing is an abstract class.
- BoardView
  - implements the abstract methods defined in the Drawing class

Collaborations

The Drawing class cannot be used on its own. The BoardView class has to implement the abstract methods, which are essentially the steps of the algorithm.

Implementation

The Drawing class requires the dimension of the board as this information is used in calculating the individual cell size.

The Drawing class provides a good reuse technique. The computeBoardLenth() method is used to calculate the usable part of the displayable screen, in a most optimal manner. This can be used in other games that require display screen size computations.

The BoardView class overrides all the abstract methods thus implementing the steps of the drawing algorithm. The GameView class acts as client and uses the draw algorithm.

Singleton

Intent

The singleton, as shown in Figure 11, provides a single instance of the PersistentStorage class.

Motivation

The player may wish to save the game state and play at a later stage. The state of the game needs to be saved to a persistent storage device. Ideally for such scenarios, there is a need for just a single connection to the filesystem or store. Opening a new connection to the store for every request from the client, is not an efficient solution as this requires more memory. The Singleton pattern provides this functionality by ensuring that there is only one connection to the storage.

Applicability

The singleton pattern is used to get access to the unique instance of an object.

Structure

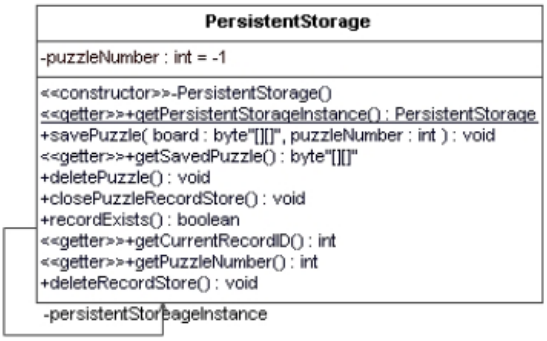


Figure 11 – Singleton pattern.

Participants

- PersistentStorage
  - defines a class method or instance operation for getting a reference to its unique instance.
  - provides a constructor to create the unique instance.

Collaborations

Clients such as the SudokuGameModel class get access to the store through the class method.

Implementation

The instance variable should be static and the constructor should be made private to prevent multiple instances. Controlled access to the persistent storage is possible as a result of the method that returns the instance and keeps track of this instance. Since one connection to the recordstore is opened, it is easy to manage compare to multiple connections.

11 JUSTIFICATION OF PATTERNS USED

In this paper, it is shown how the patterns can be used in games. The importance of design patterns has been described in the Sudoku puzzle. It is shown that, the existing design can accommodate for the change in dimension to accommodate for children's sodoku.

Since the standard Sudoku and the junior Sudoku games are virtually similar, the later is not implemented as a separate game, but is implemented as an option in the main game. Secondly, a different board game, the number puzzle game is implemented. The patterns used in the Sudoku game design are re-used in the number puzzle game. Thirdly it is shown how these patterns can be re-used if the game model changes. This is shown by explaining the design of a two player board game.

Junior Sudoku

To cater for the additional functionality of a 4×4 Sudoku board size, a state variable is maintained in the SudokuGameModel class. This variable keeps track of the dimension of the board, based on the gamer's selection. Since the stored puzzles were of 9×9 size, there has to be way to make the model independent of the puzzle size. This is achieved by incorporating the strategy pattern. The SudokuPuzzleStore defines an interface that all stores must implement. The StandardSudoku and JuniorSudoku are concrete implementations of the store. They contain puzzle sizes of 9×9 and 4×4 respectively. Depending on the user's selection, the model aggregates a different size puzzle. The option to select a differently sized puzzle is provided by means of a menu option as shown in Figure 12. The view classes adjust themselves, to either a 9x9 or a 4x4 board size.

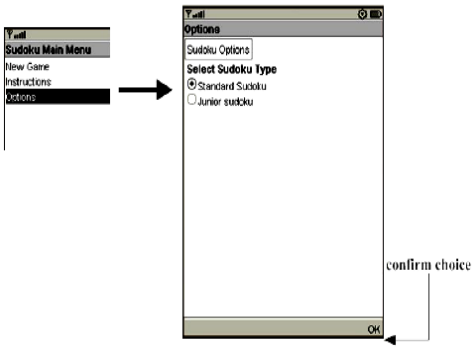


Figure 12 – Soduku menu option.

Number Puzzle Game

Based on the Sudoku game design, the number puzzle game has been designed and implemented. This is also known as the 15 puzzle game. Most of the components of the

Sudoku game remain the same in this second game too. The overall model view controller architecture is preserved. This game too, has a menu that is presented to the user. For this purpose the classes MainMenu, SudokuForm and GameView are re-used by changing the display text to reflect the N-Puzzzle game requirements. This is the only thing that has changed. The structure of these classes and their relation does not change at all. This implies that the Abstract Factory method pattern is also used in the N-Puzzle game.

An interesting thing to note is that the GameView class remains the same. The code for the paint() method remains the same and so does the key-Pressed method. The view and the model still implement the Observer pattern which was illustrated earlier in the Sudoku game.

The BoardView class and the Drawing class together implement the drawing template pattern. The model interface in the Sudoku game has abstracted the features of a single player board game and so the same interface has be used in the N-puzzle game. A concrete model class has to be implemented to take into consideration the state and behavior relevant to the new game. This technique holds true for the controller class too.

A screen shot of the N-puzzle game, with a 4x4 board is shown in Figure 13.

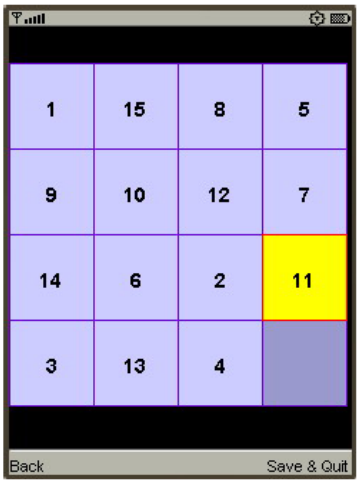


Figure 13 - N-Puzzle.

Two Player Board Game

The patterns described and implemented in this paper can also be implemented in a two player board game. The Change Screen pattern can be used to provide efficient change screen functionality. The controller would change and the model interface would change too, to reflect the abstractions of two player games. Since the earlier model interface was defined for a single player board game, it will not suit for a two player game. Therefore it is required that a new model interface be defined. Alternately the model interface could be adapted to the two player game. The drawing logic can be re-used and the programmer can define game specific drawing.

12 CONCLUSION

In this paper, we have described the usage of patterns to design and implement games on the J2ME platform. The patterns are catalogued and explained by using a template. It has been demonstrated by sample game implementations how patterns can be effectively applied to single player board based games. The design of the games has been tested for changes in requirements and it is shown how it can be modified easily to reflect these new changes.

Overuse of patterns can make the MIDlet size to grow considerably and may result in bloatable software. Some mobile devices limit the size of the jar file that can be loaded on a mobile device using the J2ME platform. This limit is usually specified in the device specifications given by the manufacturers. The games described and implemented in this paper were tested on a NOKIA n-gage gaming device with unlimited jar file size. Over use of patterns also may include additional classes to the design, which may increase the collaborations between them and likely affect the gaming performance. Developers should find a compromise between the usage of design patterns and the memory and processing power requirements for memory scarcity and low computational power mobile devices.

Finally, it can be noted that although the patterns designed and described in this paper are based on a single player, they can also be used in multi-player games and other non gaming applications built on the J2ME platform. The patterns discussed can be extended to solve commonly occurring problems in other sub-domains of computer science, like distributed computing, database applications among others [11].

REFERENCES

S. Bennett, S. McRobb and R. Farmer. Object-Oriented System Analysis and Design. McGraw Hill, 2006.

S. Bjork, S. Lundgren and J. Holopainen. Game design patterns. Proceedings of Digital Games Research Conference, 2003.

E. Gamma, R. Helm, R. Johnson and J. Vlissedes. Design Patterns Elements of Resuable Object oriented software. AddisonWesley, 1995.

Z. W. Geem. Harmony Search Algorithm for Solving Sudoku. Knowledge-Based Intelligent Information and Engineering Systems, Lecture Notes in Computer Science.

B. Hui. Big design patterns for small devices, 2002.

B. Kreimeier. The case for game design patterns, 2002.

S. MacDonald, D. Szafron, J. Schaeffer, J. Anvik , S. Bromling and K. Tan. Generative Design Patterns. Proceedings of the 17th IEEE International Conference on Automated Software Engineering, 2002.

K. Menzel, M. Keller and K. Eisenblätter. Context sensitive mobile devices in architecture, engineering and construction, ITcon Vol. 9, Special Issue Mobile Computing in Construction , pg. 389-407, 2004.

D. Nguyen and S. Wong. Design pattern for games. Special Interest Group on Computer Science Education SIGCSE, 2002.

R. Riggs, A. Taivalsaari and J. Van Peursem. Programming Wireless Devices with the Java 2 Platform Micro Edition. Addison-Wesley, second edition, 2003.

S. Stephen and A. Singh. Design Patterns for Parallel Computing Using a Network of Processors. The 6th IEEE International Symposium on High Performance Distributed Computing, 1997.

K. Topley. J2ME in a Nutshell. O'Reilly, 2002.

About the authors

J. Narsoo holds an M.S.c Computational Science and Engineering and is a Lecturer at the University of Technology, Mauritius.

M. S. Sunhaloo holds a Ph.D. degree and is a Senior Lecturer at the University of Technology, Mauritius.

R. Thomas holds an M.Sc. Computational Science and Engineering from the University of Technology Mauritius.