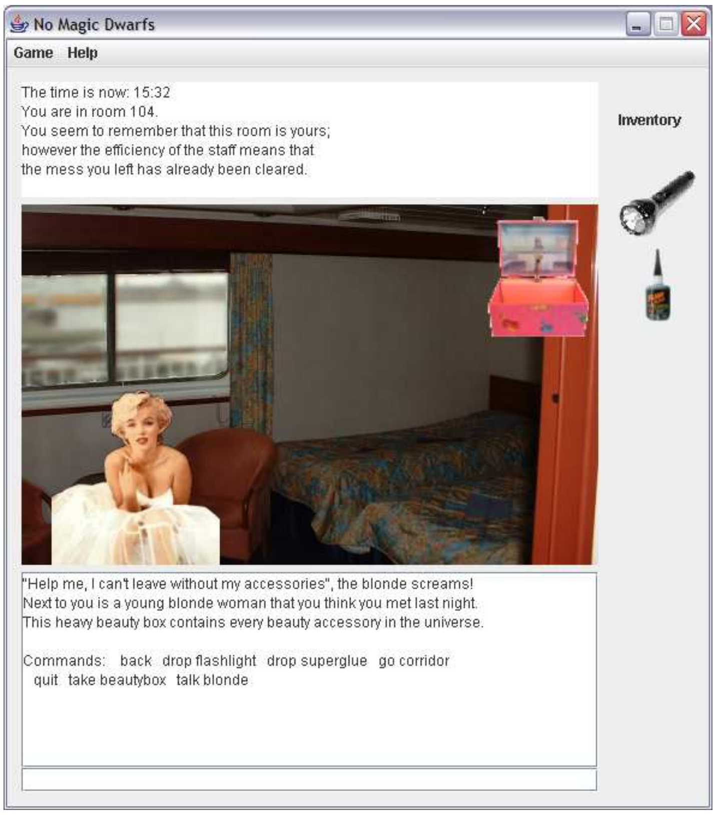
SW08 - Assignment 3
The World of Zuul - with images!



The time is now: 15:32
You are in room 104.
You seem to remember that this room is yours;
however the efficiency of the staff means that
the mess you left has already been cleared.

"Help me, I can't leave without my accessories", the blonde screams!
Next to you is a young blonde woman that you think you met last night.
This heavy beauty box contains every beauty accessory in the universe.

Commands: back drop flashlight drop superglue go corridor
quit take beautybox talk blonde

Jacob Aae Mikkelsen 191076
Jens Kristian Kræmmer Nielsen 260880
Klaus Walker 280271
Anders Brysting 110358

Due date: 15th December 2004, at 12:00

Indhold

1 Introduction

This is the third assignment and exam project for SW08. The requirement and design consideration have been made upon our own game scenery which we have named "M/S No Magic Dwarfs". The motivation for creating the game was to make a different, fun and intuitative game while still using generic codebase that would result in providing a game platform for text-based adventure games.

2 Requirements

We have implement all requirements in the assignment as well as all challenge tasks that fitted into our scenery. All in all this makes up the following feature list:

- The game has seveal locations/rooms.
- The player can walk through the locations.
- There are items in some rooms. Every room can hold any number of items. Some items can be picked up by the player, others can't.
- The player can carry some items with him. Every item has a weight. The player can carry items only up to a certain total weight.
- The player can win. There has to be some situation that is recognised as the end of the game where the player is informed that he/she has won. *(In our implementation a room can be marked as the exit)*
- Implement a command back that takes you back to the last room you've been in.
- Add at least four new commands *(our implementation allows an arbitrary number of commands in form of Tasks).*
- The application runs outside BlueJ and opens its own frame *(The game can be started thru the main method of the class Game)*
- Locations have associated images and still have text descriptions.
- The application has a menu including Quit and About.

2.1 Challenge tasks

- Add characters to your game. Characters are people or animals or monsters anything that moves, really. Characters are also in rooms (like the player and the items). Unlike items, characters can move around by themselves. *(We have implemented this by the Person class, they can move, they will follow the player as he finds them)*
- Extend the parser to recognise three-word commands. You could, for example, have a command give bread dwarf to give some bread (which you are carrying) to the dwarf. *(We have rewritten the parser to allow any number of words, in the provided scenery we use Tasks which require between 2 to 4 words)*
- Read the game specification from a file instead of hard-coding it. This way, the same program can play multiple scenerios. *(This has been done using multiple text files and images)*
- Implemented a 'Save' and 'Load' command that saves and loads the game status. *(We have implemented this by saving current location of all our objects)*

2.2 Special features

We have implemented the following extra features:

- We programmed into the game is the ability to talk to persons, in a way that you need to ask about the right topics, to get the desired information.
- We have implemented tasks that optionally require possession of items to be solved and will give you extra points.
- We have implemented an evaluation system which by looking at how many tasks and persons the player have found gives the player a matching evaluation at the end of the game.
- We have added a time limit to the game. Performing tasks, moving around and talking to charaters in the game all takes time.
- Besides showing location images, the GUI shows items and person in the room and the players inventory.

3 The Scenery of M/S No Magic Dwarfs

The scenario of the game is a cruise ship, on its way to the ocean floor. The goal of the game, is to rescue yourself and your blonde girlfriend, from a sinking ship. However, you should also rescue your partner, and your exam project, which your partner knows the location of. During the game, your blonde girlfriend demands different things, and to keep your girlfriend, you shouldn't disappoint her too much, or she will dump you in the end. You must reach the lifeboat before the ship sinks, or you will loose the game completely!

During the game you are able to pick up items, up to a reasonable weight, and use some of those items in your tasks. Choose the items carefully; they are one of the keys to success. When you meet important persons on your way, you can talk to those, but be rational, the ship is sinking, and you will only have a limited amount of time to complete the tasks, and reach the lifeboat. The time in the game is not dependent on real time, but on how many actions you go thru, (HINT: don't waste too much time on drunken Swedes).

The game is started by executing: `java -jar Dwarfs.jar`

4 Code design

One of the first decisions we made was to separate the game contents from the code. We have done this by placing all the content related text in simple text files. When then game is started all the content is then read from the files by the `TextLoader` class. Depending on how the text is being used, the class has several methods. It is possible to get the text as a string, a list or a map. These methods are made static. A good deal of the other classes makes use of this class, and thereby makes a lot of coupling. We have worked hard to minimize this, but since the class is a utility class it has been difficult to avoid.

Our goal has been to make it possible create a whole new game by only editing the text files. This is possible. But it is a lot of files (145 files for the Dwarfs game) that need editing! - It could be made much more smooth if we had all the game content stored in a XML-file, and then used Java XML-parser class to load it.

A few commands have been hard coded. They are `quit`, `back`, `bye`, `take` and `drop`. These commands will be used, no matter what game scenario is created. `take` and `drop` also have some special operation attached to them as they relate to the action of picking up or dropping an item.

When we started to add other things than rooms to the game environment, we quickly realized that we could use inheritance. The `GameObject` class is the super class for the `Player`, `Room`, `Person`, `Item` and the `Task` classes in the game. The functionality these classes have in common are related to the setting the name of the objects, and the TextLoader loading in the description and command words of the objects as well as they all have a location which have been implemented to be in any other `GameObject`. The class `GameObject` is abstract due to it forcing all subclasses to implement `GameAction` interface, this is done by design so that subclasses of `GameObject` implements an action so the player at least can do something with each game object.

When interacting with persons in the game, we thought it would be nice if the player where able to talk to them. The tech support system based on the Eliza idea seemed to be a good way to implement this. We simply changed the code of the tech support so it makes use of the `TextLoader` to load the files containing the dialogue for the specific persons. Since a dialogue is not the same as commands in the game we have implemented this is its own to avoid too much coupling to the rest of the classes.

The main `GameEngine`, which has the respondability of handling the runtime flow the game, and the dialogue system both uses the classes `GameCommand` and `CommandWords` to handle and parse all input from user. `GameCommand` implements one specific allowed command to react upon and `CommandWords` contains a set of currently allowed commands and gives both the main game and the dialogue system a way of identifying which words the user has entered. A game command can be of any number of words the users has to enter. The user can enter the words in any other in any case and can write the command as a full sentence if he/she wants to.

5 Graphical User Interface

We wanted the GUI to contain the following objects and functionality:

- A picture of the room, where the player currently is located.
- A text field showing the description of that room.
- Another text field, used to display the effect of the players action, the persons the player interacts with and the available commands.
- An input field to type in the commands.
- A panel where the items that the player has picked up is displayed.
- Further more we wanted to display the persons and items the player encounter on top of the room image. To do this we needed pictures with transparency as an option, and the png format was therefore chosen.

The frame is built up mostly with border layouts within borderlayouts so we, in that way, is able to control which elements to show where. The images are shown by adding `ImageIcon`'s to `JLabels`. The big center picture, where several images are shown on top of each others is a `JLayeredPane`. The method `setBounds` is used to control the appearance of the persons and items in the right places. The smaller item images that are added to inventory panel are also `ImageIcon`'s.

The text input field, that receives the commands, has an action listener that responds to an enter and sends the input to the game. The different elements in the GUI are then updated accordingly to the received command.

The other functionality is sited in the menu bar. Here are two menus, a game menu and a help menu. The game menu has four menu items the first one, New game, gives the possibility to choose between different game scenarios (for now there is only one real game the other is an example to illustrate this option, only consisting of two rooms, one item, one task and one person). The second is the entry for saving a game. The third gives the possibility to load a previously saved gamer. The last is the Quit option. In the Help menu we have two entries both opens a dialog. The first is the traditional About, and the second a short explanation of how to play the game.

The size of the frame is determined by the size of the elements in it. It is set so the user can't resize it. We did this as an easy way to insure that the layout doesn't get messed up by random resizing.

The GUI has one more feature. When a dialogue with one of the characters in the game takes place, it happens in a new frame. It holds a picture of the person and has a text field for showing the responses, and a field for the input. It disappears when the dialogue is terminated.

6 External maintenance

As mentioned before, it is possible to create a whole new scenery, by creating new external text files. To demonstrate the process, here is a guide to add the most complex game object - a new person.

6.1 Guide to adding a person

To add a person (with the name NEW) to the scenery (named SCENERY), you should create the following files:

`text/SCENERY/Person/NEWDefaultResponses.txt` Responses when no keyword is in the question

`text/SCENERY/Person/NEWDescripton.txt` The description of the person

`text/SCENERY/Person/NEWGoodbye.txt` Text written, when dialog is finished

`text/SCENERY/Person/NEWHello.txt` Text written, when dialog starts

`text/SCENERY/Person/NEWResponses.txt` responses, with one or more corresponding keywords

`img/SCENERY/Person/NEW.png` small picture, which is displayed on top of the room picture

`img/SCENERY/Person/NEWDialogue.png` picture showed in the dialog box

Finally, the name and initial location, should be added to the list of persons in `SCENERY/persons.txt`
To add an item, task or a new room, the sequence is rather similar, demonstrating that it can be done, without having to change the source code, and compiling over again.

7 Bugs

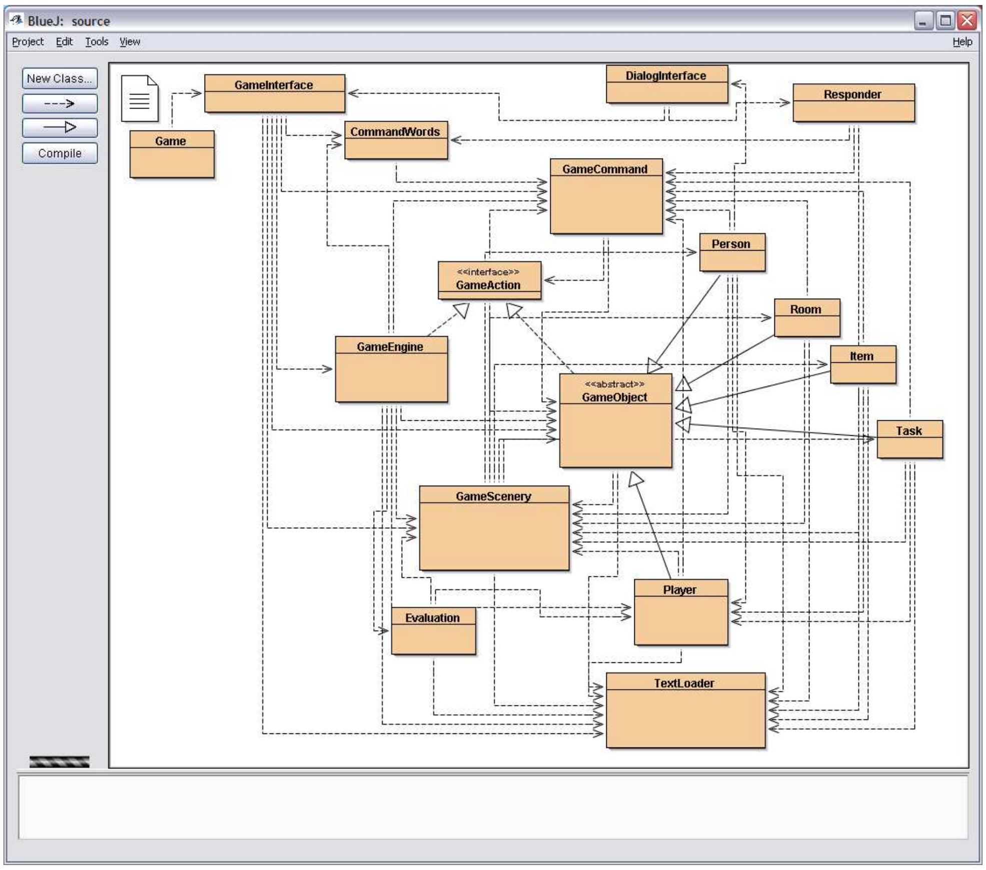
Known bugs/designflaws as of writing this document:

- If the game starttime is set, so that the time will past midnight, the game time will just continue counting up (24:01, 25:00 etc.). In the supplied scenery this is not an issue, but if any new scenery is implemented, and the time parameter is not considered, this will appear as a bug.
- A limitation exists in the load and save feature which requires every game object to have a unique name meaning that no task, room, item or person can have the same name.
- The game interface has a limit of how many items that can be shown in the inventory and a limit of how many persons and items that it can shown in the same room at the same time.
- In one case in the scenery we use that Java does not complain about missing images. This is used to avoid drawing items that are already in the main picture.

8 Conclusion

We implemented all of the requirements that we decided for and the game works as intended. We could have used even more time to construct an even better class structure, thoughts have been to distinguish between a game object and a game actor to make less coupling to the `Player` class. Also we have considered making `GameEngine` a better facade for the `GameScenery` and `Player` classes which currently is coupled to lots of places. We never got to actually code use of our possiblity for our characters to take/drop items themselfs and/or perform tasks. A lot of time went redesigning our input/output structure, which were cupled a lot with `System.out`, to be able to implement the GUI interface. The release itselfs is feature complete.

A BlueJ Class Diagram



B Source Code

B.1 Game.java

```
/**
 * This class is the main class of the "M/S No Magic Dwarfs" application.
 * "M/S No Magic Dwarfs" is a text based adventure game.
 *
 * @author Jakob Mikkelsen, Kristian Krœmmer Nielsen,
 * @author Klaus Walker and Anders Brysting
 * @version 1.0 (December 2004)
 */
public class Game
{

    /**
     * Constructor - makes no sense with an instance of this object
     */
    private Game()
    {
    }

    /**
     * Starts the game by creating interface
     */
```

```java
    public static void main(String argv[])
    {
        GameInterface gi = GameInterface.getInstance();
        gi.getFrame().setVisible(true);
    }

}
```

B.2 GameInterface.java

```java
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.Point;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Set;

import javax.swing.ImageIcon;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JLayeredPane;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.border.EmptyBorder;

/**
 * Creates a GUI for the text based adventure game "no Magic Dwarfs". This class
 * is a singleton and may only be instanciated once.
 *
 * @author Anders Brysting
 * @author Jacob Aae Mikkelsen, Kristian Kræmmer Nielsen
 * @version 1.0 (December 2004)
 */
public class GameInterface
{
    private static GameInterface gameInterface;

    private JFrame frame;
    private JLabel image;
    private JTextArea roomText;
    private JTextField textField;
    private JTextArea actionTextArea;
    private JLabel textInventory;
    private JPanel inventoryPane;
    private JPanel itemPane;
    private JLayeredPane imagePane;
    private HashMap inventory; // keeps track of the elements shown in the
    // inventory.
    private String inputLine;
    private String senario;
    private GameEngine gameEngine;
    private GameScenery gameScenery;
    private CommandWords commandWords;

    /**
     * Constructor for objects of class GameInterface
     */
    private GameInterface()
    {
        makeFrame();
        initialize("Welcome - please select a scenery in the menu.");
    }

    /**
     * Initializes the interface to defaults
     */
    private void initialize(String text)
    {
        inventory = new HashMap();
        commandWords = new CommandWords();
        gameScenery = null;
        gameEngine = null; // game is not started
        roomText.setText("");
        actionTextArea.setText(text);
    }

    /**
     * Creates the frame that holds the interface of the game.
     */
    private void makeFrame()
    {
        // The frame itself
        frame = new JFrame("No Magic Dwarfs");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Holding the content of the frame
        JPanel contentPane = (JPanel) frame.getContentPane();
        contentPane.setBorder(new EmptyBorder(6, 6, 6, 6));

        // Setting the overall layout
        contentPane.setLayout(new BorderLayout(6, 6));

        // Panel that hold the big image and the fields for in- and output text
        JPanel imgTextPane = new JPanel();
        imgTextPane.setBorder(new EmptyBorder(6, 6, 6, 6));

        imgTextPane.setLayout(new BorderLayout(6, 6));

        // The text area showing the room(location) description
        roomText = new JTextArea();
        roomText.setFocusable(false); //no user editing here
        roomText.setRows(6);
        imgTextPane.add(roomText, BorderLayout.NORTH);

        // The image showing the room (location)
        imagePane = new JLayeredPane();
        imagePane.setPreferredSize(new Dimension(480, 300));

        image = new JLabel(new ImageIcon("img/welcome.png"));
        imagePane.add(image, new Integer(0));

        // image.setVerticalAlignment(JLabel.TOP);
        // image.setHorizontalAlignment(JLabel.CENTER);
        image.setOpaque(true);
        image.setBounds(0, 0, 480, 300);
        imgTextPane.add(imagePane, BorderLayout.CENTER);

        // in- and output text
        JPanel inOutText = new JPanel();
        inOutText.setLayout(new BorderLayout());

        // output text, showing the actions from the input and the possible
        // commands
        actionTextArea = new JTextArea();
        actionTextArea.setFocusable(false); // users are not alowed to edit here
        actionTextArea.setRows(10);
        actionTextArea.setLineWrap(true);
        JScrollPane scrollActionTextPane = new JScrollPane(actionTextArea);
        inOutText.add(scrollActionTextPane, BorderLayout.CENTER);

        // to type in the that commands, reacting to the "Enter" key
        textField = new JTextField();
        inOutText.add(textField, BorderLayout.SOUTH);
        textField.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                handleInput();
            }
        });

        imgTextPane.add(inOutText, BorderLayout.SOUTH);

        // The panel showing which items the player has in the inventory.
        inventoryPane = new JPanel();
        inventoryPane.setLayout(new FlowLayout());

        itemPane = new JPanel();
        itemPane.setLayout(new GridLayout(0, 1));

        setInventoryLabel();
        itemPane.add(textInventory);

        inventoryPane.add(itemPane);

        contentPane.add(imgTextPane, BorderLayout.CENTER);

        contentPane.add(inventoryPane, BorderLayout.EAST);

        makeMenuBar();
```

```java

        // making sure stupid users don't mess up the layout :-P
        frame.setResizable(false);
        frame.pack();

        // from MK's imageviewer. Center the application on the screen.
        Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
        frame.setLocation(d.width / 2 - frame.getWidth() / 2, d.height / 2
                - frame.getHeight() / 2);
    }

    /**
     * Creates the menus for the game
     */
    private void makeMenuBar()
    {
        JMenuBar menuBar = new JMenuBar();
        frame.setJMenuBar(menuBar);

        JMenu gameMenu = new JMenu("Game");
        menuBar.add(gameMenu);

        JMenu helpMenu = new JMenu("Help");
        menuBar.add(helpMenu);

        JMenu newMenu = new JMenu("New game");
        gameMenu.add(newMenu);

        List senarios = getSceneryDescriptions();
        for (Iterator it = senarios.iterator(); it.hasNext();) {
            final String senario = (String) it.next();
            JMenuItem gameItem = new JMenuItem(senario);
            newMenu.add(gameItem);
            gameItem.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e)
                {
                    startGame(senario);
                }
            });
        }

        JMenuItem saveItem = new JMenuItem("Save game");
        saveItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                saveGame();
            }
        });
        gameMenu.add(saveItem);

        JMenuItem loadItem = new JMenuItem("Load game");
        loadItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                loadGame();
            }
        });
        gameMenu.add(loadItem);

        JMenuItem quitItem = new JMenuItem("Quit");
        quitItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                quit();
            }
        });
        gameMenu.add(quitItem);

        JMenuItem aboutItem = new JMenuItem("About 'No Magic Dwarfs'");
        aboutItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                openAbout();
            }
        });
        helpMenu.add(aboutItem);

        JMenuItem playItem = new JMenuItem("How to play 'No Magic Dwarfs'");
        playItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                openHelp();
            }
        });
        helpMenu.add(playItem);
    }

    /**
     * Sets the picture of the room.
     */
    private void paintRoom()
    {
        imagePane.removeAll();
        image = new JLabel(new
            ImageIcon((gameEngine.getPlayerLocation()).getImage()));
        imagePane.add(image, new Integer(1));
        image.setOpaque(true);
        image.setBounds(0, 0, 480, 300);
    }

    /**
     * Sets the pictures of persons in the room.
     */
    private void paintPersonsInRoom()
    {
        Set persons = gameScenery.getPersons((GameObject) gameEngine
                .getPlayerLocation());
        Iterator it = persons.iterator();
        Point origin = new Point(25, 150);
        int layerNumber = 2;
        while (it.hasNext()) {
            image = new JLabel(new ImageIcon(((GameObject)
                it.next()).getImage()));
            imagePane.add(image, new Integer(layerNumber));
            image.setOpaque(false);
            image.setBounds(origin.x, origin.y, 140, 200);
            origin.x += 120;
            layerNumber++;
        }
    }

    /**
     * Paints the pictures of items in the room.
     */
    private void paintItemsInRoom()
    {
        Set items = gameScenery.getItems((GameObject) gameEngine
                .getPlayerLocation());
        Iterator it = items.iterator();
        Point origin = new Point(380, 10);
        int layerNumber = 50;
        int numberHorizontally = 0;
        while (it.hasNext()) {
            image = new JLabel(new ImageIcon(((GameObject) it.next())
                    .getImage()));
            imagePane.add(image, new Integer(layerNumber));
            image.setOpaque(false);
            image.setBounds(origin.x, origin.y, 100, 100);
            origin.y += 90;
            numberHorizontally++;
            if (numberHorizontally % 3 == 0) {
                origin.x -= 100;
                origin.y = 10;
            }
            layerNumber++;
        }
    }

    /**
     * Quit function: quit the application.
     */
    private void quit()
    {
        System.exit(0);
    }

    /**
     * Load game function
     */
    private void loadGame()
    {
        JFileChooser chooser = new JFileChooser();
        int returnVal = chooser.showOpenDialog(frame);
        if (returnVal == JFileChooser.APPROVE_OPTION) {
            gameEngine = GameEngine.loadGame(chooser.getSelectedFile());
            gameScenery = gameEngine.getGameScenery();
            actionTextArea.setText("");
            updateStatus();
        }
    }

    /**
     * Save game function
     */
    private void saveGame()
    {
        if (gameEngine != null) {
            JFileChooser chooser = new JFileChooser();
            int returnVal = chooser.showSaveDialog(frame);
            if (returnVal == JFileChooser.APPROVE_OPTION) {
                File selected = chooser.getSelectedFile();
                boolean save = false;
                if (selected.exists()) {
                    if (JOptionPane.showConfirmDialog(frame,
                            "Are␣you␣sure␣you␣want␣to␣overwrite␣'"
                                    + chooser.getSelectedFile() + "'?",
                            "Save␣game", JOptionPane.WARNING_MESSAGE,
                            JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION)
                                {
                        save = true;
                    }
                }
                else {
                    save = true;
                }
                if (save) {
                    gameEngine.saveGame(chooser.getSelectedFile());
                }
            }
        } else {
            JOptionPane.showMessageDialog(frame, "You␣need␣to␣be␣playing␣a␣game␣
                before␣you␣can␣save␣it!", "Save␣game", JOptionPane.ERROR_MESSAGE);
        }
    }

    /**
     * Displays an item that has been picked in the inventory display.
     */
    private void paintItems()
    {
        itemPane.removeAll();
        setInventoryLabel();
        Set items = gameEngine.getPlayerInventory();
        for (Iterator it = items.iterator(); it.hasNext();) {
            JLabel item = new JLabel(new ImageIcon(((GameObject)
                it.next()).getSmallImage()));
            item.setBorder(new EmptyBorder(4, 0, 4, 0));
            itemPane.add(item);
        }
        frame.pack();
    }

    /**
     * Creates the text label the names the inventory
     */
    private void setInventoryLabel()
    {
        textInventory = new JLabel("Inventory");
        textInventory.setPreferredSize(new Dimension(64, 60));
        textInventory.setBorder(new EmptyBorder(0, 0, 4, 0));
        itemPane.add(textInventory);
    }
```

```java

    /**
     * Appends a string of text to the text already displayed.
     *
     * @param text The text to be placed as a string.
     */
    private void appendActionText(String text)
    {
        if (actionTextArea.getText().length() != 0) {
            actionTextArea.append("\n");
        }
        actionTextArea.append(text);
        // scroll to bottom:
        actionTextArea.setCaretPosition(actionTextArea.getDocument()
                .getLength());
    }

    /**
     * Loads the names of avialable scenearies
     */
    private List getSceneryDescriptions()
    {
        return TextLoader.getTextList("text/sceneries.txt");
    }

    /**
     * Starts a new game
     *
     * @param sceneryName The name of the scenery.
     */
    private void startGame(String sceneryName)
    {
        if (gameEngine != null) { // confirm to quit existing game
            if (JOptionPane.showConfirmDialog(frame,
                    "You are already playing - are you sure you want to start a new game?",
                    "New game",
                    JOptionPane.WARNING_MESSAGE,
                    JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION) {
                gameEngine = null;
            }
        }

        if (gameEngine == null) {
            gameScenery = new GameScenery(sceneryName);
            gameEngine = new GameEngine(gameScenery);
            actionTextArea.setText("");
            updateStatus();
        }
    }

    /**
     * Updates game status in the GUI
     */
    private void updateStatus()
    {
        if (gameEngine.isStopped()) {
            paintGameEnd();
```

```java
                initialize(gameEngine.getStatus());
            }
            else {
                appendActionText(gameEngine.getStatus());
                commandWords = gameEngine.getCurrentCommandWords();
                appendActionText(commandWords.getTextList());
                roomText.setText(gameEngine.getLocationDescription());
                paintItems();
                paintRoom();
                paintItemsInRoom();
                paintPersonsInRoom();
            }
        }

        /**
         * Perform an inputted command. Executes an user-entered command
         */
        private void handleInput()
        {
            String cmdText = textField.getText().trim();
            if (cmdText.length() > 0) {
                Set cmdWords = new HashSet(Arrays.asList(cmdText.split(" ")));
                if (commandWords.isCommand(cmdWords)) {
                    GameCommand cmd = commandWords.getCommand(cmdWords);
                    String actionText = gameEngine.handleCommand(cmd);
                    if (actionText == null) {
                        actionTextArea.setText("");
                    }
                    else {
                        actionTextArea.setText(actionText + "\n");
                    }
                    gameEngine.updateStatus();
                    updateStatus();
                }
                else {
                    appendActionText("I am sorry, that is not a valid command - please
                        try again.");
                }
                textField.setText("");
            }
        }

        /**
         * Get the instance of the GameInterface
         *
         * @return Instance of class
         */
        public static GameInterface getInstance()
        {
            if (gameInterface == null) {
                gameInterface = new GameInterface();
            }
            return gameInterface;
        }

        /**
         * Get the main frame of the GameInterface This can be used to instance
         * dialogs
```

```
     *
     * @return JFrame
     */
    public JFrame getFrame()
    {
        return frame;
    }

    /**
     * opens the about text in a new frame.
     */
    private void openAbout()
    {
        String aboutText = TextLoader.getTextString("text/about.txt");
        JOptionPane.showMessageDialog(frame, aboutText, "About",
            JOptionPane.INFORMATION_MESSAGE);
    }

    /**
     * opens the help text in a new frame.
     */
    private void openHelp()
    {
        String aboutText = TextLoader.getTextString("text/help.txt");
        JOptionPane.showMessageDialog(frame, aboutText, "Help",
            JOptionPane.INFORMATION_MESSAGE);
    }

    /**
     * paints the game end image.
     */
    private void paintGameEnd()
    {
        imagePane.removeAll();
        if (gameEngine.isCompleted()) {
            image = new JLabel(new ImageIcon("img/" + gameScenery.getPath()
                    + "/gamewon.png"));
        }
        else {
            image = new JLabel(new ImageIcon("img/" + gameScenery.getPath()
                    + "/gameover.png"));
        }
        imagePane.add(image, new Integer(1));
        image.setOpaque(true);
        image.setBounds(0, 0, 480, 300);
    }

}
```

B.3 GameAction.java

```
/**
 * This defines the interface for a GameAction
 * A implementation of a GameAction provides definition of how to execute
 * any behavour for one or more GameCommand objects.
 *
 * @author Kristian Kræmmer Nielsen
 * @version 2.0 (December 2004)
 */
```

```

public interface GameAction
{
    /**
     * Handles an action
     *
     * @param player The player that performed the action
     * @param cmd The command performed
     * @return Returns action text or null if nothing happens
     */
    public String performCommand(GameObject player, GameCommand cmd);

}
```

B.4 GameObject.java

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

/**
 * Class GameObject - an object in the "M/S No Magic Dwarfs" adventure game.
 *
 * This class is a superclass for various classes in game.
 *
 * @author Kristian Kræmmer Nielsen and Anders Brysting.
 * @version 2.0 (December 2004)
 */
public abstract class GameObject implements GameAction
{
    // instance variables
    private String name;
    private String description;
    private GameScenery scenery;
    private GameObject location;
    private Set objects; // Set of other objects which this object holds
    private boolean pickable; // can the object be picked up
    private int weight; // the Objects weight

    /**
     * Constructor for objects of class GameObject. The descriptions and the
     * command words are loaded from text files.
     *
     * @param scenery Scenery of which this object belong
     * @param name The name of the game object.
     */
    public GameObject(GameScenery scenery, String name)
    {
        this.scenery = scenery;
        this.name = name;
        this.location = null;
        this.objects = new HashSet();
        description = TextLoader.getTextString(getFilePrefix() +
            "Description.txt");
        pickable = false;
        weight = 0;
    }

    /**
```

```
     * Returns prefix for filenames used by this object
     *
     * @return Prefix for files used by this object
     */
    protected String getFilePrefix()
    {
        return "text/" + scenery.getPath() + "/" + this.getClass().getName() +
            "/" + getName();
    }

    /**
     * Returns image filename
     *
     * @return image filename
     */
    public String getImage()
    {
        return "img/" + scenery.getPath() + "/" + this.getClass().getName() + "/"
            + getName() + ".png";
    }

    /**
     * Returns image filename for the small image
     *
     * @return image filename
     */
    public String getSmallImage()
    {
        return "img/" + scenery.getPath() + "/" + this.getClass().getName() + "/"
            + getName() + "Small.png";
    }

    /**
     * Returns name of object
     *
     * @return The name of the object.
     */
    public String getName()
    {
        return name;
    }

    /**
     * Returns description for object
     *
     * @return The description of the object.
     */
    public String getDescription()
    {
        return description;
    }

    /**
     * Returns content description for the object
     *
     * @return Returns description of the objects this object contains, e.g.
     *     tasks, items, persons,...
     */
    public String getContentDescription()
    {
        String returnString = "";
        for (Iterator i = objects.iterator(); i.hasNext();) {
            String desc = ((GameObject) i.next()).getDescription();
            if (desc != null) {
                returnString += desc + "\n";
            }
        }
        return returnString;
    }

    /**
     * Returns current location of the object
     *
     * @return Returns object
     */
    public GameObject getLocation()
    {
        return this.location;
    }

    /**
     * Returns Game Scenery of which this objects belongs
     *
     * @return Game Scenery
     */
    protected GameScenery getGameScenery()
    {
        return scenery;
    }

    /**
     * Move object to a new location
     *
     * @param location Object to place in (null allows the object to disappear)
     */
    public void setLocation(GameObject location)
    {
        if (this.location != null) {
            this.location.objects.remove(this);
        }
        this.location = location;
        if (location != null) {
            location.objects.add(this);
        }
    }

    /**
     * Returns a set of all game objects contained in this object
     *
     * @return Set of gameObjects
     */
    public Set getObjects()
    {
        return this.objects;
    }

    /**
     * Returns a set of commands which this object allows anything to perform
     * with it.
     * Default implementation returns an empty set.
     *
     * @return Set of Commands
     */
    protected Set getCommands()
    {
        return new HashSet();
    }

    /**
     * Returns a set of commands which this object allows a specifik object, e.g.
        player
     * to perform with it.
     * Default implementation returns the same as getCommands()
     *
     * @param player Player object that may perform commands
     * @return Set of Commands
     */
    public Set getCommands(GameObject player)
    {
        return getCommands();
    }

    /**
     * Returns a set of all avialable commands on this object and objects
     * contains in this object
     *
     * @param player Player object that may perform commands
     * @return Set of Commands
     */
    public Set getAllCommands(GameObject player)
    {
        Set cmds = new HashSet(getCommands(player));
        for (Iterator i = getObjects().iterator(); i.hasNext();) {
            GameObject go = (GameObject) i.next();
            cmds.addAll(go.getAllCommands(player));
        }
        return cmds;
    }

    /**
     * @return A boolean value expressing whether or not the item is pickable
     */
    public boolean isPickable()
    {
        return pickable;
    }

    /**
     * Sets the boolean value for pickability
     *
     * @param b The boolean value to set.
     */
    public void setPickable(boolean b)
    {
```

```java
            pickable = b;
        }

        /**
         * @return The weight of the Object
         */
        public int getItemWeight()
        {
            return weight;
        }

        /**
         * Sets the objects weight
         *
         * @param w The weight given as an int.
         */
        public void setWeight(int w)
        {
            weight = w;
        }

}
```

B.5 Item.java

```java
import java.util.HashSet;
import java.util.List;
import java.util.Set;

/**
 * This class represent items in the
 * game. it defines the weight of the item, and sets whether or not it is
 * possible to pick up the item.
 *
 * @author Klaus Walker and Anders Brysting and Kristian Kræmmer Nielsen.
 * @version 0.1 (November 2004)
 *
 */
public class Item extends GameObject
{

    /**
     * Constructor for objects of class Item
     *
     * @param scenery Scenery of which this object belong
     * @param name The name of the item, to pas on the the super class
     */
    public Item(GameScenery scenery, String name)
    {
        super(scenery, name);
        List weightList = TextLoader.getTextList(getFilePrefix() + "Weight.txt");
        setWeight(Integer.parseInt((String) weightList.get(0)));
        setPickable(Boolean.valueOf((String) weightList.get(1)).booleanValue());
    }

    /**
     * Returns either the "take" or "drop" prefixed command depending upon if
     * the item is hold by a player or not
     *
     * @return Set of GameCommands
     */
    public Set getCommands()
    {
        Set cmds = new HashSet();
        if (getLocation() instanceof Player) { // only players can do this at this
             time
            cmds.add(new GameCommand(this, "drop " + getName()));
        }
        else if (isPickable()) {
            cmds.add(new GameCommand(this, "take " + getName()));
        }
        return cmds;
    }

    /**
     * Perform take or drop action
     *
     * @param holder The player that performed the action
     * @param cmd The command performed
     * @return Returns action text or null if nothing happens
     */
    public String performCommand(GameObject holder, GameCommand cmd)
    {
        if (holder instanceof Player) {
            Player player = (Player) holder;
            player.addTime(1);
            if (getLocation().equals(player)) {
                setLocation(player.getLocation()); // drop and place me where
                                                    // holder are
            }
            else if (player.canCarry(this)) {
                setLocation(player); // taken by player
            }
            else {
                return "You cannot carry that much weight!";
            }
        }
        else { // others like Person - FIXME: this feature is not used of the
            current avialable sceneries
            if (getLocation().equals(holder)) {
                setLocation(holder.getLocation()); // drop and place me where
                                                    // holder are
            }
            else {
                setLocation(holder); // taken
            }
        }
        return null;
    }

}
```

B.6 Person.java

```
import java.util.HashSet;
import java.util.Set;

/**
```

```java
 * Class Person - a person in the "M/S No Magic Dwarfs" adventure game.
 *
 * This class is part of the "M/S No Magic Dwarfs" application. "M/S No Magic
 * Dwarfs" is a very simple, text based adventure game.
 *
 * A "Person" represents one character that the player can interact with in the
 * game. It is has a dialog and/or some action that the player can act and
 * respond to.
 *
 * @author Kristian Kræmmer Nielsen and Anders Brysting.
 * @version 2.0 (December 2004)
 */
public class Person extends GameObject
{
    /**
     * Constructor for objects of class Person
     *
     * @param scenery Scenery of which this object belong
     * @param name Name of person
     * @param room Initial room
     */
    public Person(GameScenery scenery, String name, GameObject room)
    {
        super(scenery, name);
        setLocation(room);
    }

    /**
     * Returns image filename for the small image
     *
     * @return image filename
     */
    public String getDialogueImage()
    {
        return "img/" + getGameScenery().getPath() + "/"
                + this.getClass().getName() + "/" + getName() + "Dialogue.png";
    }

    /**
     * Return goodbye text after a dialog
     *
     * @return text
     */
    private String getGoodbyeText()
    {
        return TextLoader.getTextString(getFilePrefix() + "Goodbye.txt");
    }

    /**
     * Returns the "talk xxx" command
     *
     * @return Set of Commands
     */
    public Set getCommands(GameObject player)
    {
        Set cmds = new HashSet();
        if (player instanceof Player) {
            cmds.add(new GameCommand(this, "talk␣" + getName()));
        }
        return cmds;
    }

    /**
     * Handles an action
     *
     * @param player The player that performed the action
     * @param cmd The command performed
     * @return Returns action text or null if nothing happens
     */
    public String performCommand(GameObject player, GameCommand cmd)
    {
        DialogInterface dialog = new DialogInterface(getFilePrefix(),
                getDialogueImage(), getName());
        ((Player) player).addTime(5); // takes five minutes
        return getGoodbyeText();
    }

}
```

B.7 Task.java

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;

/**
 * Tasks in the "no Magic Dwarfs" game.
 *
 * @version 1.0 (November 2004)
 * @author Anders Brysting
 */
public class Task extends GameObject
{
    private Map triggers;
    List requirements;

    /**
     * Constructs a Task object.
     *
     * @param gameScenery The curret game senario.
     * @param name The name of the task. To pass on to the super class.
     */
    public Task(GameScenery gameScenery, String name)
    {
        super(gameScenery, name);
        requirements = TextLoader.getTextList(getFilePrefix()
                + "Requirements.txt");
        makeTriggerMap();
    }

    /**
     * Checks if the player holds the item(s) that may be required to solve the
     * task.
     *
     * @return A boolean value expressing whether the task can be solved or not.
     */
    public boolean solveable(GameObject player)
    {
        List missingObjs = new ArrayList(requirements);
        for (Iterator i = player.getObjects().iterator(); i.hasNext();) {
            GameObject go = (GameObject) i.next();
            if ((go instanceof Item) && (missingObjs.contains(go.getName()))) {
                missingObjs.remove(go.getName());
            }
        }
        return missingObjs.isEmpty();
    }

    /**
     * Returns a set of commands which this object allows an user to do any of
     * to solve the task
     *
     * @return Set of Commands
     */
    private void makeTriggerMap()
    {
        Map textMap = TextLoader.getTextMap(getFilePrefix()
                + "CommandWords.txt");
        triggers = new HashMap();
        for (Iterator i = textMap.keySet().iterator(); i.hasNext();) {
            String cmd = (String) i.next();
            triggers.put(new GameCommand(this, cmd), textMap.get(cmd));
        }
    }

    /**
     * Returns a set of commands which this object allows an user to do any of
     * to solve the task
     *
     * @param player The object that may do these things
     * @return Set of Commands
     */
    public Set getCommands(GameObject player)
    {
        if (this.solveable(player)) {
            return triggers.keySet();
        }
        else {
            return new HashSet();
        }
    }

    /**
     * If a task is solved it is moved to be located in the player object
     *
     * @param player The player that performed the action
     * @param cmd The command performed
     * @return Returns action text or null if nothing happens
     */
    public String performCommand(GameObject player, GameCommand cmd)
    {
```

```java
        if (player instanceof Player) { // only a player can perform tasks
            // Takes away the used items from player
            Set mustRemove = new HashSet();
            for (Iterator i = player.getObjects().iterator(); i.hasNext();) {
                GameObject go = (GameObject) i.next();
                if (requirements.contains(go.getName())) {
                    mustRemove.add(go);
                }
            }
            // must be done afterwards since it changes the player inventory
            for (Iterator i = mustRemove.iterator(); i.hasNext();) {
                GameObject go = (GameObject) i.next();
                go.setLocation(null); // nowhere
            }
            ((Player) player).addTask(this);
            return (String) triggers.get(cmd);
        }
        return null;
    }

}
```

B.8 Room.java

```java
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;

/*
 * A room in the "No Magic Dwarfs" adventure game.
 *
 * A "Room" represents one location in the scenery of the game. It is connected
 * to other rooms via exits. For each existing exit, the room stores a reference
 * to the neighboring room. The room can also hold items.
 *
 * @author Anders Brysting, Kristian Kræmmer Nielsen
 *
 * @version 1.0 (November 2004)
 */

public class Room extends GameObject
{
    Map exits; // holds map of commands to move to another room from here

    /**
     * Creates new room
     *
     * @param scenery Scenery of which this object belong
     * @param name The name of the room. To pass on to the super class.
     */
    public Room(GameScenery scenery, String name)
    {
        super(scenery, name);
        exits = TextLoader.getTextMap(getFilePrefix() + "CommandWords.txt");
        loadItems();
        loadTasks();
    }

    /**
     * Loads items that is in the room
     */
    private void loadItems()
    {
        List items = TextLoader.getTextList(getFilePrefix() + "Item.txt");
        for (Iterator i = items.iterator(); i.hasNext();) {
            String itemName = (String) i.next();
            getGameScenery().getItem(itemName).setLocation(this);
        }
    }

    /**
     * Loads tasks that belong to the room
     */
    private void loadTasks()
    {
        List newTasks = TextLoader.getTextList(getFilePrefix() + "Task.txt");
        for (Iterator i = newTasks.iterator(); i.hasNext();) {
            String taskName = (String) i.next();
            getGameScenery().getTask(taskName).setLocation(this);
        }
    }

    /**
     * Returns possible exit commands from this room.
     * Overrides method in superclass.
     *
     * @return Set of commands
     */
    public Set getCommands()
    {
        Set cmds = new HashSet();
        for (Iterator i = exits.keySet().iterator(); i.hasNext();) {
            String cmd = (String) i.next();
            Room exitRoom = getGameScenery().getRoom((String) exits.get(cmd));
            cmds.add(new GameCommand(exitRoom, cmd));
        }
        return cmds;
    }

    /**
     * Moves the player to this room
     *
     * @param player The player that performed the action
     * @param cmd The command performed
     * @return Returns action text or null if nothing happens
     */
    public String performCommand(GameObject player, GameCommand cmd)
    {
        player.setLocation(this);
        return null;
    }

}
```

B.9 Player.java

```java
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
import java.util.Stack;

/**
 * Keeps track of the inventory, and in which room the player is.
 *
 * Keeps track of the time spent in the project, and keeps track of different
 * tasks is accomplished or not.
 *
 * @author Jacob Aae Mikkelsen
 * @version 1.1 (December 2004)
 */
public class Player extends GameObject
{
    private int startingTime;
    private int totalTimeUsed;
    private int useableTime;
    private Stack history;
    private Set completedTasks;
    private int maxItemWeight; // maximum weight the player can carry.

    /**
     * Constructor for objects of class player
     *
     * @param room Initial room
     */
    public Player(GameScenery scenery, String name)
    {
        super(scenery, name);
        setStartingTime();
        totalTimeUsed = 0;
        setUseableTime();
        history = new Stack();
        maxItemWeight = 20;
        super.setLocation(scenery.getEntrance());
        completedTasks = new HashSet();
    }

    /**
     * Changes location (overrides superclass) In addition to moving the player,
     * this increases the time spend in the game and stores the current location
     * in history
     *
     * @param location New room player enters
     */
    public void setLocation(GameObject location)
    {
        history.push(getLocation());
        addTime(3);
        super.setLocation(location);
    }

    /**
     * Move player back one room
     */
    public void goPreviousLocation()
    {
        if (history.size() > 0) {
            super.setLocation((GameObject) history.pop());
            addTime(1);
        }
    }

    /**
     * Sets the time of start of the game, the default value is 1800
     */
    private void setStartingTime()
    {
        startingTime = Integer.parseInt((String) TextLoader
                .getTextString("text/" + getGameScenery().getPath()
                        + "/startTime.txt"));
    }

    /**
     * Sets the number of minutes, the player can use, before the game is over
     */
    private void setUseableTime()
    {
        useableTime = Integer.parseInt((String) TextLoader
                .getTextString("text/" + getGameScenery().getPath()
                        + "/timeToUse.txt"));
    }

    /**
     * Sets the number of minutes, the player can use, before the game is over
     *
     * @param time The new time in minnutes the player is allowed to use before
     *             "GAME OVER"
     */
    public boolean isAlive()
    {
        if (useableTime > totalTimeUsed) {
            return true;
        }
        else {
            return false;
        }
    }

    /**
     * Can carry item
     *
     * @return A boolean expressing wether or not the player can carry anymore.
     */
    public boolean canCarry(GameObject thing)
    {
        int itemWeight = 0;
        // calculate how much player is carrying.
        for (Iterator i = getInventory().iterator(); i.hasNext();) {
            GameObject item = (GameObject) i.next();
            itemWeight += item.getItemWeight();
        }
        return (itemWeight + thing.getItemWeight() <= this.maxItemWeight);
    }

    /**
     * Adds the completed task, but only if it is not already present in the
     * collection.
     *
     * @param task the task completed
     */
    public void addTask(GameObject task)
    {
        task.setLocation(this);
        addTime(10);
    }

    /**
     * Checks if the collection contains a task
     *
     * @param task the task to check whether the collection contains it
     * @retrun True or false for a tasks persens.
     */
    public boolean containsTask(GameObject task)
    {
        return getObjects().contains(task);
    }

    /**
     * @return the number of tasks completed
     */
    public int numberOfTasksCompleted()
    {
        int numberOfTasks = completedTasks.size();
        return numberOfTasks;
    }

    /**
     * Adds time to the total time used in the game
     *
     * @param minutes the number of minutes the total time should be incremented
     */
    public void addTime(int minutes)
    {
        totalTimeUsed = totalTimeUsed + minutes;
    }

    /**
     * Returns total time used
     *
     * @retrun minutes used
     */
    public int getTotalTimeUsed()
    {
        return totalTimeUsed;
    }

    /**
     * Returns invetory filters out Item objects by checking if the object is
     * pickable
     *
     * @return Set of Items
```

```java
     */
    public Set getInventory()
    {
        Set inventory = new HashSet();
        for (Iterator i = getObjects().iterator(); i.hasNext();) {
            GameObject go = (GameObject) i.next();
            if (go.isPickable() == true) {
                inventory.add(go);
            }
        }
        return inventory;
    }

    /**
     * @return Returns the time of the game
     */
    public String getTime()
    {
        int hours = (startingTime / 100) + (totalTimeUsed / 60);
        int minutes = (startingTime % 100) + (totalTimeUsed % 60);
        String timeString = "The␣time␣is␣now:␣";
        if (minutes < 10) {
            timeString += hours + ":0" + minutes;
        }
        else {
            timeString += hours + ":" + minutes;
        }
        return timeString;
    }

    /**
     * Returns the "back" command that the player always can do.
     *
     * @return Set of Commands
     */
    public Set getCommands()
    {
        Set cmds = new HashSet();
        cmds.add(new GameCommand(this, "back"));
        return cmds;
    }

    /**
     * Overrides getDescription() Players currently does not have any
     * description
     *
     * @return String
     */
    public String getDescription()
    {
        return null;
    }

    /**
     * This handles the "back" command
     *
     * @param player The player that performed the action (normally myself)
     * @param cmd The command performed
     * @return Returns action text or null if nothing happens
     */
    public String performCommand(GameObject player, GameCommand cmd)
    {
        goPreviousLocation();
        return null;
    }
}
```

B.10 GameScenery.java

```java
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;

/**
 * Class GameScenery - contains the game scenery.
 *
 * When constructing a new instance of a GameScenery, the entire scenarie is
 * loaded by creating all available Rooms, Items, Persons and Tasks in the
 * provided scenery.
 * This class collects all elements that makes up the scenery and is used to
 * later receive the objects.
 *
 * @author Kristian Kræmmer Nielsen
 * @version 1.0 (22 November 2004)
 */
public class GameScenery
{
    private String path;
    private String goodbyeText;
    private String welcomeText;
    private String gameOverText;
    private Map rooms;
    private Map persons;
    private Map items;
    private Map tasks;
    private Set followers; // Persons that will follow the Player when they see
                           // him.

    /**
     * Constructs a new game scenery
     *
     * @param path Base directory of text files making up the scenery
     */
    public GameScenery(String path)
    {
        this.path = path;
        // load texts
        this.welcomeText = TextLoader.getTextString("text/" + path
                + "/sceneryWelcome.txt");
        this.goodbyeText = TextLoader.getTextString("text/" + path
                + "/sceneryGoodbye.txt");
        this.gameOverText = TextLoader.getTextString("text/" + path
                + "/gameOver.txt");
        // tasks and items are loaded on demand
        this.tasks = new HashMap();
        this.items = new HashMap();
        // load rooms
        this.rooms = new HashMap();
        List roomNames = TextLoader.getTextList("text/" + path + "/rooms.txt");
        for (Iterator i = roomNames.iterator(); i.hasNext();) {
            String name = (String) i.next();
            rooms.put(name, new Room(this, name));
        }
        // load persons
        this.persons = new HashMap();
        Map personNames = TextLoader
                .getTextMap("text/" + path + "/persons.txt");
        for (Iterator i = personNames.keySet().iterator(); i.hasNext();) {
            String name = (String) i.next();
            Person person = new Person(this, name, getRoom((String) personNames
                    .get(name)));
            persons.put(name, person);
        }
        // load followers
        this.followers = new HashSet();
        List followersNames = TextLoader.getTextList("text/" + path
                + "/followers.txt");
        for (Iterator i = followersNames.iterator(); i.hasNext();) {
            String name = (String) i.next();
            followers.add(getPerson(name));
        }
    }

    /**
     * Returns base path of scenery files
     *
     * @return Base path
     */
    public String getPath()
    {
        return this.path;
    }

    /** Returns greeting for when the game is started */
    public String getWelcomeText()
    {
        return this.welcomeText;
    }

    /** Returns goodbye message for when leaving the scenery */
    public String getGoodbyeText()
    {
        return this.goodbyeText;
    }

    /** Returns game over message for when time is up */
    public String getGameOverText()
    {
        return this.gameOverText;
    }

    /**
```

```
     * @return Room given by name
     * @param name Name of Room
     */
    public Room getRoom(String name)
    {
        return (Room) rooms.get(name);
    }

    /**
     * Returns a Person by name
     *
     * @param name Name of person
     * @return Person
     */
    public Person getPerson(String name)
    {
        return (Person) persons.get(name);
    }

    /**
     * Returns Item given by name
     *
     * @param name Name of item
     * @return Item
     */
    public Item getItem(String name)
    {
        Item item = (Item) items.get(name);
        if (item == null) {
            item = new Item(this, name);
            items.put(name, item);
        }
        return item;
    }

    /**
     * Returns a Task by name
     *
     * @param name Name of task
     * @return Task
     */
    public Task getTask(String name)
    {
        Task task = (Task) tasks.get(name);
        if (task == null) {
            task = new Task(this, name);
            tasks.put(name, task);
        }
        return task;
    }

    /**
     * Returns followers
     *
     * @return Set of followers
     */
    public Set getFollowers()
    {
        return followers;
    }

    /**
     * Returns Person that are currently in the given room
     *
     * @param room Room
     * @return Set of Persons
     */
    public Set getPersons(GameObject room)
    {
        Set inRoom = new HashSet();
        for (Iterator i = room.getObjects().iterator(); i.hasNext();) {
            GameObject person = (GameObject) i.next();
            if (person instanceof Person) {
                inRoom.add(person);
            }
        }
        return inRoom;
    }

    /**
     * Returns Items that are currently in the given room
     *
     * @param room Room
     * @return Set of Persons
     */
    public Set getItems(GameObject room)
    {
        Set inRoom = new HashSet();
        for (Iterator i = room.getObjects().iterator(); i.hasNext();) {
            GameObject item = (GameObject) i.next();
            if (item instanceof Item) {
                inRoom.add(item);
            }
        }
        return inRoom;
    }

    /**
     * Returns the first room in the scenery
     *
     * @return first room
     */
    public Room getEntrance()
    {
        return (Room) rooms.get(TextLoader.getTextString("text/" + path
                + "/entrance.txt"));
    }

    /**
     * Returns the last room in the scenery
     *
     * @return first room
     */
    public Room getExit()
    {
        return (Room) rooms.get(TextLoader.getTextString("text/" + path
```

```java
                    + "/exit.txt"));
    }

    /**
     * Returns all movable objects.
     * Used to save/load scenery.
     *
     * @return returns all items
     */
    public Map getAllMovableObjects()
    {
        Map objs = new HashMap();
        objs.putAll(items);
        objs.putAll(tasks);
        objs.putAll(persons);
        return objs;
    }

}
```

B.11 GameEngine.java

```java
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;

/**
 * GameEngine controls the runtime flow of the adventure game.
 *
 * Providing a GameScenery to the GameEngine and the engine will take care of
 * the runtime of the game.
 *
 * @author Kristian Kræmmer Nielsen
 * @version 1.0 (22 November 2004)
 */
public class GameEngine implements GameAction
{

    private GameScenery gameScenery;
    private Player player;
    private boolean gameStopped;
    private boolean gameCompleted;
    private String currentStatus;

    /**
     * Constructor Takes a GameScenery and takes care of game runtime
     */
    public GameEngine(GameScenery gameScenery)
    {
        this.gameScenery = gameScenery;
        this.player = new Player(gameScenery, "player");
        this.gameStopped = false;
        this.gameCompleted = false;
        this.updateStatus();
        this.currentStatus = gameScenery.getWelcomeText() + "\n"
                + currentStatus;
    }

    /**
     * Assembles current available commands
     *
     * @return commands
     */
    public CommandWords getCurrentCommandWords()
    {
        CommandWords cmdWords = new CommandWords();
        cmdWords.addCommand(new GameCommand(this, "quit")); // static always
                                                            // available command
        cmdWords.addCommands(player.getLocation().getAllCommands(player));
        // since the player is contained in the room this will add all available
        // commands
        return cmdWords;
    }

    /**
     * Handles command Returns action text from executing the command or null if
     * nothing happens
     *
     * @param cmd The command to be handled.
     */
    public String handleCommand(GameCommand cmd)
    {
        String out;
        Set persons = gameScenery.getPersons((GameObject) player.getLocation());
        out = cmd.performCommand(player);
        handleFollowers(persons);
        return out;
    }

    /**
     * Handles followers that sticks to the users as they seem him.
     *
     * @param persons Persons currently together with user
     */
    private void handleFollowers(Set persons)
    {
        for (Iterator i = persons.iterator(); i.hasNext();) {
            GameObject person = (GameObject) i.next();
            Set followers = gameScenery.getFollowers();
            if (followers.contains(person)) {
                person.setLocation(player.getLocation());
            }
        }
    }

    /**
     * Print status, like room description, time, scores, etc..
     */
    public String getStatus()
    {
        return currentStatus;
    }

    /**
     * Get location description and time left in game
     *
     * @return Returns a description of the players location
     */
    public String getLocationDescription()
    {
        return player.getTime() + "\n" + player.getLocation().getDescription();
    }

    /**
     * Gets the current content of the players inventory.
     *
     * @return a set of the items in the inventory.
     */
    public Set getPlayerInventory()
    {
        return player.getInventory();
    }

    /**
     * Returns Scenery object
     *
     * @return scenery
     */
    public GameScenery getGameScenery()
    {
        return gameScenery;
    }

    /**
     * Returns current location of Player
     *
     * @return GameObject Location of Player
     */
    public GameObject getPlayerLocation()
    {
        return player.getLocation();
    }

    /**
     * Returns the Persons currently following the Player
     *
     * @return Set of Persons
     */
    public Set getFollowers()
    {
        Set persons = gameScenery.getPersons((GameObject) player.getLocation());
        Set fellows = new HashSet();
        for (Iterator it = persons.iterator(); it.hasNext();) {
            GameObject person = (GameObject) it.next();
            Set followers = gameScenery.getFollowers();
            if (followers.contains(person)) {
                fellows.add(person);
            }
        }
        return fellows;
    }

    /**
     * Returns weather the game has stopped or not
     */
    public boolean isStopped()
    {
        return gameStopped;
    }

    /**
     * Check whether the game is won or lost
     * @return true if won, false if lost or not completed
     */
    public boolean isCompleted()
    {
        return gameCompleted;
    }

    /**
     * Update status of game, is player alive or have he/she completed the game
     * successfully
     */
    public void updateStatus()
    {
        currentStatus = "";
        if (!player.isAlive()) {
            gameStopped = true;
            currentStatus = gameScenery.getGameOverText() + "\n";
        }
        else if (player.getLocation().equals(gameScenery.getExit())) {
            gameStopped = true;
            Evaluation evaluation = new Evaluation(gameScenery, player);
            currentStatus = evaluation.getCompleteEvaluation() + "\n";
            gameCompleted = true;
        }
        if (isStopped()) {
            currentStatus += gameScenery.getGoodbyeText();
        }
        else {
            currentStatus += player.getLocation().getContentDescription();
        }
    }

    /**
     * Handles the "quit" command
     *
     * @param player The player that performed the action
     * @param cmd The command performed
     * @return Returns action text or null if nothing happens
     */
    public String performCommand(GameObject player, GameCommand cmd)
    {
        gameStopped = true;
        return null;
    }

    /**
```

```
     * Save scenery state to file
     *
     * @param file file
     * @param player GameObject which is the player
     */
    public void saveGame(File file)
    {
        try {
            FileWriter fw = new FileWriter(file);

            fw.write(gameScenery.getPath() + "\n");
            // save player location
            fw.write(player.getLocation().getName() + "\n");
            // save time used
            fw.write(Integer.toString(player.getTotalTimeUsed()) + "\n");

            Map objs = gameScenery.getAllMovableObjects();
            for (Iterator i = objs.keySet().iterator(); i.hasNext();) {
                String name = (String) i.next();
                GameObject go = (GameObject) objs.get(name);
                fw.write(name + "\n");
                if (go.getLocation() == null) {
                    fw.write("NULL\n");
                }
                else if (go.getLocation().equals(player)) {
                    fw.write("INVENTORY\n");
                }
                else {
                    fw.write(go.getLocation().getName() + "\n");
                }
            }
            fw.close();

        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }

    /**
     * Load scenery state from file
     *
     * @param filename Name of file
     * @param player GameObject to place inventory items inside.
     */
    public static GameEngine loadGame(File file)
    {
        List inp = TextLoader.getTextList(file.getAbsolutePath());
        GameScenery gameScenery = new GameScenery((String) inp.get(0)); // scenery
                                                                         // loaded
        GameEngine gameEngine = new GameEngine(gameScenery);
        Map objs = gameScenery.getAllMovableObjects();

        // Set player location
        GameObject playerLocation = gameScenery.getRoom((String) inp.get(1));
        gameEngine.player.setLocation(playerLocation);

        // Set time used
        gameEngine.player.addTime(Integer.parseInt((String) inp.get(2)));
```

```java

        for (int i = 3; i < inp.size();) {
            String objName = (String) inp.get(i++);
            GameObject ob = (GameObject) objs.get(objName);
            String locationName = (String) inp.get(i++);
            if (locationName.equals("INVENTORY")) { // in players inventory
                ob.setLocation(gameEngine.player);
            }
            else if (locationName.equals("NULL")) { // object not in used anymore
                ob.setLocation(null);
            }
            else {
                GameObject obLocation = (GameObject)
                    gameScenery.getRoom(locationName);
                // FIXME: does not support being hold by anything else, e.g.
                    "Person"
                ob.setLocation(obLocation);
            }
        }
        gameEngine.updateStatus();
        return gameEngine;
    }

}
```

B.12 GameCommand.java

```java
import java.util.Arrays;
import java.util.List;

/**
 * This class holds information about a command that was issued by the user. A
 * command consists of a String that can contain one or many words that the user
 * must enter to execute the command.
 * A GameCommand can be associated with a GameAction which allows the command to
    be
 * directly executed.
 *
 * @author Kristian Kræmmer Nielsen
 * @version 1.0 (December 2004)
 */

public class GameCommand
{
    private GameAction action; // action to perform
    private String words; // required word for action

    /**
     * Create a command object from the provided string with the associated
     * object
     *
     * @param action GameAction to associate
     * @param word The command word
     */
    public GameCommand(GameAction action, String word)
    {
        this.action = action;
        this.words = word;
    }

    /**
     * Create a command object from the provided string with no associated
     * object
     *
     * @param word The command word
     */
    public GameCommand(String word)
    {
        this.words = word;
    }

    /**
     * Returns the set of words needed to execute this command
     *
     * @return Set of words
     */
    public List getWords()
    {
        return Arrays.asList(words.split(" "));
    }

    /**
     * Get command as a string
     */
    public String toString()
    {
        return this.words;
    }

    /**
     * Perform action
     * Notice that calling this method requires that the GameCommand in question
     * is in fact associated with a GameAction - if this is not the case the call
     * will fail terriblely.
     *
     * @param player The player that performed the action
     * @return Returns action text or null if nothing happens
     */
    public String performCommand(GameObject player)
    {
        return action.performCommand(player, this);
    }

}
```

B.13 CommandWords.java

```
import java.util.Set;
import java.util.TreeSet;
import java.util.HashSet;
import java.util.Iterator;

/**
 * This class is part of the "M/S No Magic Dwarfs" application. "M/S No Magic
 * Dwarfs" is a text based adventure game. This class holds an enumeration of
 * current avialable command words known to the game. It is used to determind
 * which commands to executed based on input from the user.
 *
 * @version 1.0 (November 2004)
 * @author Kristian Kræmmer Nielsen, Jacob Aae Mikkelsen
 */

public class CommandWords
{
    // Set of avaliable commands.
    // structure is Set of Command objects.
    private Set commandSet;

    /**
     * Constructor - initialise the command words.
     */
    public CommandWords()
    {
        commandSet = new HashSet();
    }

    /**
     * Make the specified commands available
     *
     * @param newCommands the new commands to add.
     */
    public void addCommands(Set newCommands)
    {
        commandSet.addAll(newCommands);
    }

    /**
     * Make the specified command available
     *
     * @param command the new command to add.
     */
    public void addCommand(GameCommand command)
    {
        commandSet.add(command);
    }

    /**
     * Returns the command object based on a given HashMap. This is done by
     * finding the command which has all its words contained in the given input
     * HashMap. Commands are priorites so that the commands with most words are
     * rated higher than commands with less words.
     * Examples:
     * <ul>
     * <li>input: "pickup" (will match the command "pickup")</li>
     * <li>input: "pickup lightbulb" (will match the command "pickup lightbulb"
         and not "pickup")</li>
     * </ul>
     *
     * @param input The input which to map to a command
     * @return the command object or null if it is not a valid command.
     */
    public GameCommand getCommand(Set input)
    {
        GameCommand closestCommand = null;
        int numberOfWordsUsed = 0;

        // go through all available commands
        for (Iterator iCmd = this.commandSet.iterator(); iCmd.hasNext();) {
            GameCommand command = (GameCommand) iCmd.next();
            Set words = new HashSet(command.getWords());
            // we will only look at the command if it uses more words that the
            // command we already found as a match
            if (words.size() > numberOfWordsUsed) {
                // go through the words that has to be in the sentence to match
                // this command, removing the words found.
                for (Iterator iCmdWord = words.iterator(); iCmdWord.hasNext();) {
                    String cmdWord = (String) iCmdWord.next();
                    for (Iterator iInputWord = input.iterator();
                        iInputWord.hasNext();) {
                        String inputWord = (String) iInputWord.next();
                        if (inputWord.equalsIgnoreCase(cmdWord)) {
                            iCmdWord.remove();
                            break;
                        }
                    }
                }
                if (words.size() == 0) {
                    // found all needed words => found a possible command
                    numberOfWordsUsed = command.getWords().size();
                    closestCommand = command;
                }
            }
        }
        return closestCommand;
    }

    /**
     * Check whether a given HashSet contains a valid set of command word(s).
     * Return true if it is, false if it isn't.
     */
    public boolean isCommand(Set input)
    {
        return getCommand(input) != null;
    }

    /*
     * Get all valid commands as String.
     */
    public String getTextList()
    {
        // sort commands
        TreeSet ts = new TreeSet();
        for (Iterator it = commandSet.iterator(); it.hasNext();) {
            ts.add(it.next().toString());
        }
        // print commands
        StringBuffer sb = new StringBuffer();
        sb.append("Commands:␣");

        StringBuffer line = new StringBuffer();
        boolean hasContent = false;
        for (Iterator it = ts.iterator(); it.hasNext();) {
            line.append("␣␣␣" + it.next());
            hasContent = true;
```

```java
            if (line.length() > 55) {
                sb.append(line + "\n");
                line = new StringBuffer();
            }
        }
        if (hasContent) {
            sb.append(line);
        }
        return sb.toString();
    }
}
```

B.14 DialogInterface.java

```java
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

import javax.swing.ImageIcon;
import javax.swing.JDialog;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.border.EmptyBorder;

/**
 * This class implements a GUI based dialog system. The dialog communicates via
 * text input/output in a textarea.
 *
 * This class uses an object of class CommandWords to parse input from the user,
 * and an object of class Responder to generate responses.
 *
 * @version 1.0 (December 2004)
 * @author Kristian Kræmmer Nielsen, Anders Brysting
 */
public class DialogInterface
{
    private JDialog dialog;
    private JPanel contentPane;
    private JTextArea actionTextArea;
    private JTextField textField;
    private Responder responder;
    private String imagePath;
    private String name;

    /**
     * Constructor for objects of class DialogInterface
     *
     * @param prefix prefix
     * @param imagePath image filename
     * @param name name of person
     *
     */
    public DialogInterface(String prefix, String imagePath, String name)
    {
        this.imagePath = imagePath;
        this.name = name;
        makeDialog();
        responder = new Responder(prefix);
        actionTextArea.setText(responder.getHello());
        dialog.setVisible(true);
    }

    /**
     * Creates the frame that holds the interface of the game.
     */
    private void makeDialog()
    {
        // The frame itself
        dialog = new JDialog(GameInterface.getInstance().getFrame(),
                "Dialog with " + name + " (type 'bye' to end dialog)", true);

        // Holding the content of the frame
        JPanel contentPane = (JPanel) dialog.getContentPane();
        contentPane.setBorder(new EmptyBorder(6, 6, 6, 6));

        // Setting the overall layout
        contentPane.setLayout(new BorderLayout(6, 6));

        // Panel that hold the big image and the fields for in- and output text
        JPanel imgTextPane = new JPanel();
        imgTextPane.setBorder(new EmptyBorder(6, 6, 6, 6));

        imgTextPane.setLayout(new BorderLayout(6, 6));

        // The image showing the person we are talking to
        JLabel image = new JLabel(new ImageIcon(imagePath));
        imgTextPane.add(image, BorderLayout.CENTER);

        // in- and output text
        JPanel inOutText = new JPanel();
        inOutText.setLayout(new BorderLayout());

        // output text, showing the actions from the input and the possible
        // commands
        actionTextArea = new JTextArea(8, 50);
        actionTextArea.setFocusable(false);
        actionTextArea.setLineWrap(true);

        JScrollPane scrollActionTextPane = new JScrollPane(actionTextArea);
        inOutText.add(scrollActionTextPane, BorderLayout.CENTER);

        // to type in the that commands, reacting to the "Enter" key
        textField = new JTextField();
        inOutText.add(textField, BorderLayout.SOUTH);
        textField.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                handleInput();
            }
        });
        imgTextPane.add(inOutText, BorderLayout.SOUTH);

        contentPane.add(imgTextPane, BorderLayout.CENTER);

        // making sure stupid users don't mess up the layout :-P
        dialog.setResizable(false);
        dialog.pack();

        // from MK's imageviewer. Center the application on the screen.
        Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
        dialog.setLocation(d.width / 2 - dialog.getWidth() / 2, d.height / 2
                - dialog.getHeight() / 2);
    }

    /**
     * Appends a string of text to the text already displayed.
     *
     * @param text The text to be placed as a string.
     */
    private void appendActionText(String text)
    {
        if (actionTextArea.getText().length() != 0) {
            actionTextArea.append("\n");
        }
        actionTextArea.append(text);
        // scroll to bottom:
        actionTextArea.setCaretPosition(actionTextArea.getDocument()
                .getLength());
    }

    /**
     * React on input.
     */
    private void handleInput()
    {
        String text = textField.getText().trim();
        if (text.length() > 0) {
            Set words = new HashSet(Arrays.asList(text.split(" ")));
            appendActionText(responder.generateResponse(words));
            textField.setText("");
            if (responder.isStopped()) {
                dialog.dispose();
            }
        }
    }

}
```

B.15 Responder.java

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Random;
import java.util.Set;

/**
```

```
 * The responder class represents a response generator object. It is used to
 * generate an automatic response, based on specified input. Input is presented
 * to the responder as a set of words, and based on those words the responder
 * will generate a String that represents the response.
 *
 * Internally, the reponder uses a HashMap to associate words with response
 * strings and a list of default responses. If any of the input words is found
 * in the HashMap, the corresponding response is returned. If none of the input
 * words is recognized, one of the default responses is randomly chosen.
 *
 * @version 1.2 (November.2004)
 * @author Anders Brysting, Kristian Kræmmer Nielsen
 * @author adapted from Michael Kolling's and David J. Barnes's tech support
 *         system.
 */
public class Responder
{
    private Map responseMap; // used to map key words to responses
    private ArrayList defaultResponses; // default responses to use if we don't
                                        // recognise a word
    private Random randomGenerator;
    private String prefix;
    private boolean dialogStopped;

    /**
     * Construct a Responder
     *
     * @param prefix String that determins which person to talk to.
     */
    public Responder(String prefix)
    {
        this.prefix = prefix;
        responseMap = new HashMap();
        defaultResponses = new ArrayList();
        fillResponseMap();
        fillDefaultResponses();
        randomGenerator = new Random();
        dialogStopped = false;
    }

    /**
     * Generate a response from a given set of input words.
     *
     * @param words A set of words entered by the user
     * @return A string that should be displayed as the response
     */
    public String generateResponse(Set words)
    {
        CommandWords cmdWords = new CommandWords();
        GameCommand bye = new GameCommand("bye");
        cmdWords.addCommand(bye);
        cmdWords.addCommands(responseMap.keySet());
        if (cmdWords.isCommand(words)) {
            GameCommand cmd = cmdWords.getCommand(words);
            if (cmd.equals(bye)) {
                dialogStopped = true;
                return "";
            }
```

```
                return (String) responseMap.get(cmd);
            }
            // If we get here, none of the words from the input line was recognized.
            // In this case we pick one of our default responses (what we say when
            // we cannot think of anything else to say...)
            return pickDefaultResponse();
        }

        /**
         * Enter all the known keywords and their associated responses from a text
         * file into the response map.
         */
        private void fillResponseMap()
        {
            Map textMap = TextLoader.getTextMap(prefix + "Responses.txt");
            responseMap = new HashMap();
            for (Iterator i = textMap.keySet().iterator(); i.hasNext();) {
                String trigger = (String) i.next();
                responseMap.put(new GameCommand(trigger), textMap.get(trigger));
            }
        }

        /**
         * Build up a list of default responses from which we can pick one if we
         * don't know what else to say.
         */
        private void fillDefaultResponses()
        {
            defaultResponses.addAll(TextLoader.getTextList(prefix
                    + "DefaultResponses.txt"));
        }

        /**
         * Randomly select and return one of the default responses.
         *
         * @return A random default response
         */
        private String pickDefaultResponse()
        {
            // Pick a random number for the index in the default response list.
            // The number will be between 0 (inclusive) and the size of the list
            // (exclusive).
            int index = randomGenerator.nextInt(defaultResponses.size());
            return (String) defaultResponses.get(index);
        }

        /**
         * Returns true if dialog ended The user has entered "bye"
         *
         * @return true or false
         */
        public boolean isStopped()
        {
            return dialogStopped;
        }

        /**
         * Return hello text
```

```
     *
     * @return welcome text
     */
    public String getHello()
    {
        return TextLoader.getTextString(prefix + "Hello.txt");
    }

}
```

B.16 Evaluation.java

```
import java.util.List;
import java.util.Iterator;

/**
 * The class Evaluation does what the name says, evaluates the player at the end
     of the game.
 *
 * @version 1.0 (November 2004)
 * @author Jacob Aae Mikkelsen
 */
public class Evaluation
{
    private GameScenery gameScenery;
    private Player player;
    private int totalPoints;
    private String path;

    /**
     * Constructor
     * @param gameScenery Scenery to evaluate
     * @param player Player to evaluate
     */
    public Evaluation(GameScenery gameScenery, Player player)
    {
        this.gameScenery = gameScenery;
        this.player = player;
        totalPoints = 0;
        this.path = gameScenery.getPath();
    }

    /**
     * The complete evaluation, of required tasks, required persons found etc.
     * The evaluation text is then printed from the file coresponding to the
     * obtained number of points
     * @return evaluation text.
     */
    public String getCompleteEvaluation()
    {
        foundPersons();
        completedRequiredTasks();
        completedOtherTasks();
        String evaluation = TextLoader.getTextString("text/" + path
                + "/Evaluation/" + totalPoints + "points.txt");
        return evaluation;
    }

    /**
```

```java
     * Increments the totalPoints by 100, for each person found, defined in the
     * requiredPersons file
     */
    private void foundPersons()
    {
        List personsToFind = TextLoader.getTextList("text/" + path
                + "/Evaluation/requiredPersons.txt");
        Iterator it = personsToFind.iterator();
        while (it.hasNext()) {
            String temp = (String) it.next();
            if (gameScenery.getPerson(temp).getLocation().equals(
                    player.getLocation())) {
                totalPoints += 100;
            }
        }
    }

    /**
     * Increments the totalPoints by 1000, for each required task that has been
     * completed, defined in the requiredTasks file
     */
    private void completedRequiredTasks()
    {
        List requiredTasks = TextLoader.getTextList("text/" + path
                + "/Evaluation/requiredTasks.txt");
        Iterator it = requiredTasks.iterator();
        while (it.hasNext()) {
            String taskName = (String) it.next();
            if (player.containsTask(gameScenery.getTask(taskName))) {
                totalPoints += 1000;
            }
        }
    }

    /**
     * Increments the totalPoints by the number of not specifically required
     * tasks, that the player has accomplished. However only incremented by the
     * value defined as a success limit defined in the requiredNoOtherTasks file
     */
    private void completedOtherTasks()
    {
        int numberFromFile = Integer.parseInt((String) TextLoader
                .getTextString("text/" + path
                        + "/Evaluation/requiredNoOtherTasks.txt"));
        int completedTasks = player.numberOfTasksCompleted();
        List requiredTasks = TextLoader.getTextList("text/" + path
                + "/Evaluation/requiredTasks.txt");
        Iterator it = requiredTasks.iterator();
        while (it.hasNext()) {
            String taskName = (String) it.next();
            if (player.containsTask(gameScenery.getTask(taskName))) {
                completedTasks--;
            }
            if (completedTasks >= numberFromFile) {
                totalPoints = totalPoints + numberFromFile;
            }
        }
    }
```

```
}
```

B.17 TextLoader.java

```
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

/**
 * The TextLoader reads a text file one line at the time seperating them at \n
 * or \r or both. The strings can be stored in a list. Only ASCII-characters can
    be
 * handled.
 *
 * @version 1.0 (November 2004)
 * @author Anders Brysting
 *
 */
public class TextLoader
{

    /**
     * Construct a TextLoader
     */
    private TextLoader()
    {

    }

    /**
     * Reads lines as strings from a text file. The readFile method uses the
     * RandomAccessFile() method from the java.io to create a List containing
     * strings. The original file is spilt at 'newline' or 'carriage return' and
     * only handles ASCII-characters.
     *
     * @param fileName The name of the the file to be read. Path most be
     *                 included if the file is not in the same libery as the game
     *                 files.
     * @return A list containing strings read from the file.
     */
    public static List getTextList(String fileName)
    {
        List strings = new ArrayList();
        if (new File(fileName).isFile()) {
            try {
                RandomAccessFile accessFile = new RandomAccessFile(fileName, "r");
                long offset = accessFile.getFilePointer();
                while (accessFile.readLine() != null) {
                    accessFile.seek(offset);
                    strings.add(accessFile.readLine());
                    offset = accessFile.getFilePointer();
                }
                accessFile.close();
            } catch (IOException e) {
                    System.out.println(e);
                }
            }
            return strings;
        }

        /**
         * Converts a list of strings to one continous string. The getTextString
         * method calls the getTextList method and converts the list of strings to
         * one continous string.
         *
         * @param fileName The name of the the file to be read. Path most be
         *              included if the file is not in the same libery as the game
         *              files.
         *
         * @return A String containing strings read from the file.
         */

        public static String getTextString(String fileName)
        {
            List description = new ArrayList(getTextList(fileName));
            String strings = "";
            Iterator it = description.iterator();
            while (it.hasNext()) {
                if (strings.length() > 0) {
                    strings += "\n";
                }
                strings = strings + it.next();
            }
            return strings.trim();
        }

        /**
         * Converts a list of strings a to a map. The getTextMap method calls the
         * getTextList method and converts the list of strings to a map containing
         * the strings.Strings with an even index number will be the keys, and
         * strings with an odd index number the values.
         *
         * @param fileName The name of the the file to be read. Path most be
         *              included if the file is not in the same libery as the game
         *              files.
         *
         * @return A Map containing strings read from the file.
         */
        public static Map getTextMap(String fileName)
        {
            List description = new ArrayList(getTextList(fileName));
            Map strings = new HashMap();
            Iterator it = description.iterator();
            while (it.hasNext()) {
                strings.put(it.next(), it.next());
            }
            //System.out.println(descript);
            return strings;
        }
}
```