System design document for the Morf project (SDD)

Contents

Version: 2

Date: 2015-05-31

Author: Lage Bergman, Gustav Bergström, Harald Brorsson, Christoffer Karlsson

This version overrides all previous versions.

1 Introduction

1.1 Design goals

All modules of the application must be interchangeable and able to stand alone. The model must be completely free of any framework components and only contain raw types or objects from the Java standard API.

1.2 Definitions, acronyms and abbreviations

- GUI, graphical user interface.

- Java, platform independent programming language.
- JRE, the Java Runtime Environment. Additional software needed to run a Java application.
- Host, a computer where the game will run.
- Player, the person who is playing the game.
- Player Character, the character that the player controls.
- Level, a stage where the goal is to take the character from one point to another.
- Level grid, a grid for representing positions in a level.
- Block, a type of environment placed on a specific position in the grid.

2 System design

2.1 Overview

The application will consist of a main menu leading to the level selection screen among others. From the level selection screen the game itself can be started which is the bulk of the application.

The application will use a MVC model, modified to fit with the project framework.

The application consists of six packages: model, view, handlers, screens, main and files (see Appendix).

2.1.2 Rules

The objective of the game is to water the flower located at a specific, but different position in each level. Along the way there are obstacles such as ridges and spikes, making it difficult to reach the flower. To get around these obstacles the player character has a water can, a container containing liquid nitrogen and a torch at his disposal, making it possible to pour, heat and cool water. By doing this the player can change the water's physical state between water, ice and vapor. The player character can move through water, walk on ice and fly on steam.

2.1.3 Unique identifiers, global lookups

Most factories and handlers will be accessible globally to ensure communication is possible from menus to the actual game. When one level is completed, that level's score needs to be available in the LevelPreview. Therefore the HighScoreHandler is a singleton available throughout the application, providing high score information.

2.1.4 Event handling

Collisions between physics objects in the game world are handled by Box2D. Box2D makes sure that objects that should collide do, and that new forces are assigned to all objects in the collision. This is done automatically.

Whenever two objects collide or lose contact with each other a Contact event is created containing the two fixtures that collided. The CollisionListener class implements

ContactListener and recieves every Contact event that is created. This means that the CollisionListener has access to two fixtures and should from them be able to do things like enable flying or kill the player as appropriate. To differentiate between two fixtures every fixture has an instance of CollisionData as their UserData from Box2D. UserData can be any object and is used to contain information about the fixture.

The CollisionData used as UserData has a type known as CollisionType which is an Enum. The CollisionType can be for example PLAYERCHARACTER, WATER or even GHOST_BOTTOM. The CollisionType essentially keeps track of what kind of fixture the CollisionData is attached to and whether it is a sensor or not.

Beyond CollisionType the CollisionData also contains how many other fixtures its fixture is in contact with. This is relevant when two fixtures lose contact with each other and the fixtures belong to for example a Water block in Gas state and the PlayerCharacter. When the PlayerCharacter is no longer in contact with water vapor, the flyingEnabled variable should be set to false.

However, without CollisionData it is impossible to know whether the PlayerCharacter is still in contact with water vapor as it may have been in contact with two different water vapor blocks before. With CollisionData a contact can be added whenever the fixture collides with another fixture and a contact removed whenever it loses contact with another fixture. In the above case with PlayerCharacter losing contact with water vapor the CollisionListener can check if the PlayerCharacter fixture’s number of contacts is equal to 0, in which case flyingEnabled can be set to false.

2.1.5 Frameworks

The application will use a collection of frameworks called libGDX. LibGDX contains frameworks that are made specifically to create games in Java. The application uses in particular two frameworks, Box2D and Scene2D. Box2D is responsible for all in-game physics simulations, including handling events whenever two objects (i.e. the player character and a spike block) collide. Scene2D is used for the different menu screens and, providing functionality for image buttons, for example.

2.2 Software decomposition

2.2.1 General

The application will be composed of the following modules (see Appendix):

- main, a package containing the Main class holding the main-method.
- model, core of the application. Model part of MVC.
- view, GUI handling drawing of the game. View part of MVC.
- handlers, factories for generating different objects accessible globally, a level generator, event handlers and constants.
- screens, different screens to be shown in the libGDX window (see Appendix).
- file, classes handling file writing and reading

The model package is in accordance with the MVC model responsible for holding all information about the game itself. It contains information about where all the blocks in the world are for example.

The view and screens package both handle GUI and visuals but in different ways. Screens contain framework specific classes implementing the libGDX Screen interface. A libGDX application window requires a screen for anything to be drawn in it. By making each screen responsible for a single area or menu, such as a menu or the game itself, and switching between screens it is easy to navigate through the project structure and work with the code.

The view package is responsible for drawing the game itself on one of the screens called PlayScreen. It retrieves data from the model and draws it on the screen.
The handlers package is responsible for managing settings and highscores and building the game world through the use of factories. The handlers package also contains the package controllers, which is in accordance with the MVC model responsible for updating the model and handling user input.

The file package is responsible for handling all files that are saved between sessions. For example settings and highscores.

The handlers contains factories, mostly being Singleton, for creating different objects. It also contains event handlers for handling contacts in the world and input from the user. It also contains a class for generating a level and some globally used constants.

2.2.2 Decomposition into subsystems

To create new maps or levels for the game, a third party software called Tiled Map Editor is used. The software creates a .tmx file which the Morf application can read.

The in-game physics simulations are handled by the framework Box2D.

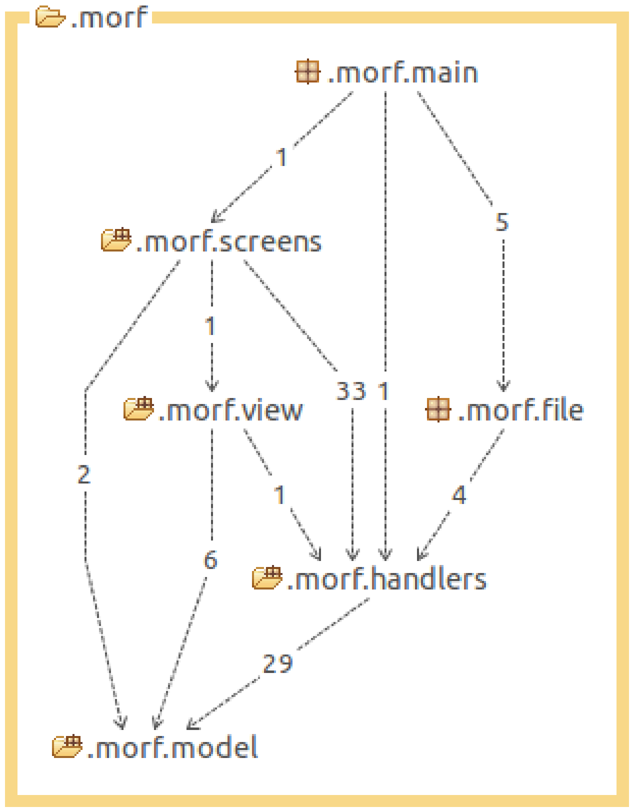
LibGDX is rendering the in-game graphics and Scene2D is responsible for displaying the menus.

2.2.3 Layering

The layering is as indicated in figure below. Higher layer are at the top of the figure (see appendix).

2.2.4 Dependency analysis

Dependencies are shown in figure. There are no circular dependencies.



2.3 Concurrency issues

NA. This is a single threaded application. Everything will be handled by the rendering thread in the Main class which is extending libGDX's Game class.

2.4 Persistent data management

All persistent data is stored in raw text files located in a hidden folder in the user home directory. If the system can not find these files on startup it generates them. If they are found they are read and the data is stored in maps in the corresponding class. On exit the maps are written back into each of their files respectively. This is handled by classes extending the abstract FileHandler class in the handlers package.

2.5 Access control and security

NA.

2.6 Boundary conditions

NA. Application launched and exited as normal desktop application.

2.7 Active Block

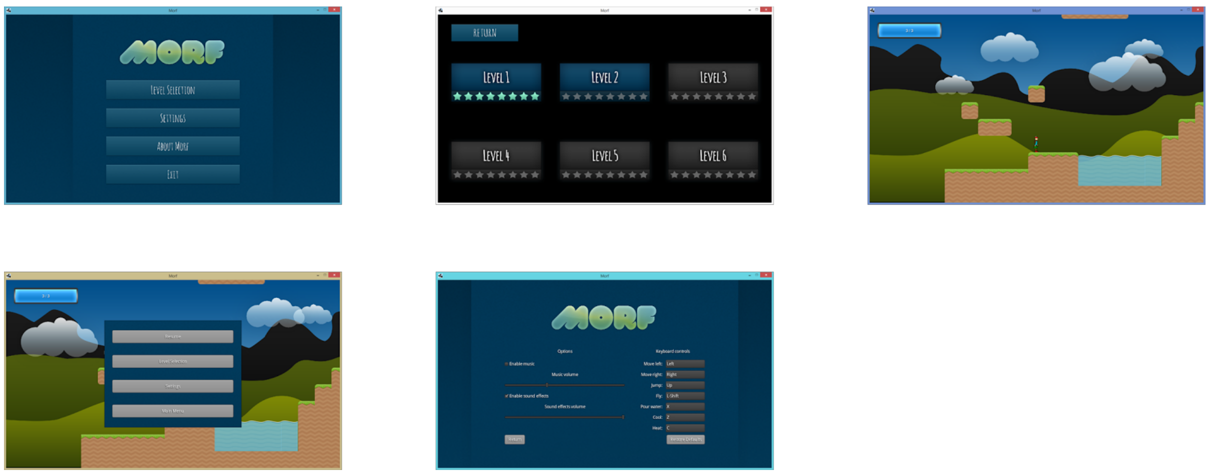
Apart from placing new water blocks the player can interact with the world through heating and cooling water blocks. The block that can currently be manipulated by the player is represented by the activeBlock variable which is stored in the PlayerCharacter class and is of the type Block.

When the PlayerCharacter heatActiveBlock or coolActiveBlock methods are called the same methods are executed in the activeBlock as the class Block implements an interface with methods heat and cool.

The activeBlock variable is updated through two other variables of type Block, activeBlockRight and activeBlockLeft. Whenever the sensors responsible for activeBlockRight registers contact with a block in the controller CollisionListener, the CollisionListener sets the activeBlockRight variable in PlayerCharacter to the block it received contact with. The same is true for activeBlockLeft. Whenever a move command is issued by the player the move method in PlayerCharacter is executed. The move method updates the activeBlock to the same as activeBlockRight if the player is moving right and vice versa.

The PlayerCharacter class contains a variable activeBlock which is manipulated when the PlayerCharacter heatActiveBlock or coolActiveBlock methods are called. This block is set by the controller holding the Box2D physics engine world whenever one of the sensor fixtures on the player character body in the world collides with another block.
Most of the time the active block heat and cool methods are empty and therefore not affected by any player inputs, but when the active block is a water block it is heated or cooled depending on what the player does.

3 Appendix



Options

Enable music

Music volume

✓ Enable sound effects

Sound effects volume

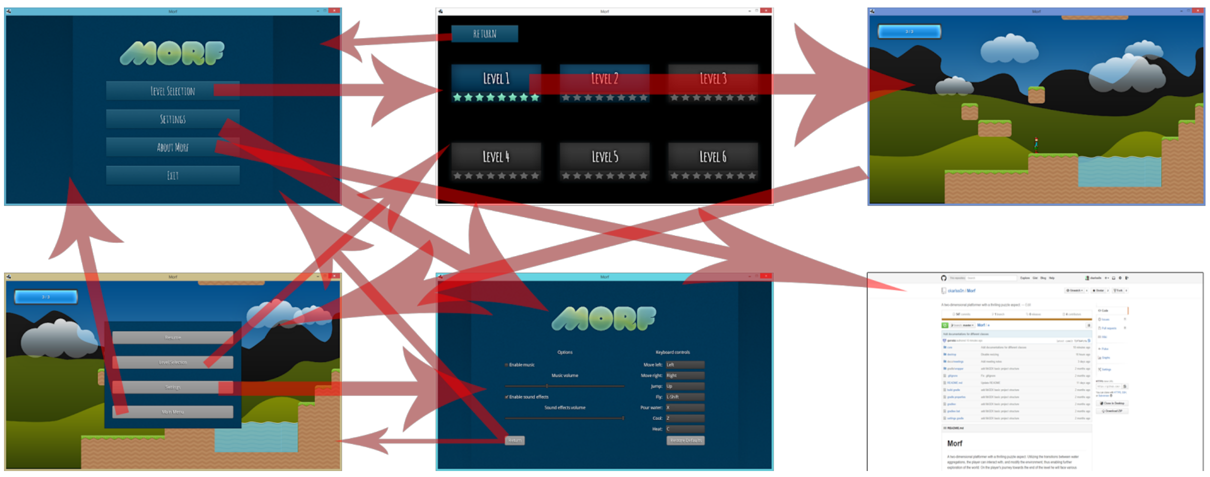
Keyboard controls

Move left:	Left
Move right:	Right
Jump:	Up
Fly:	L-Shift
Pour water:	X
Cool:	Z
Heat:	C

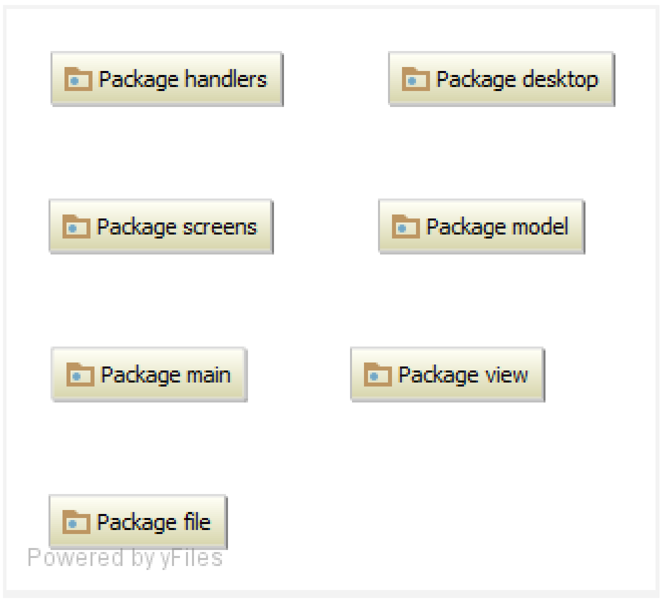
Return

Restore Defaults

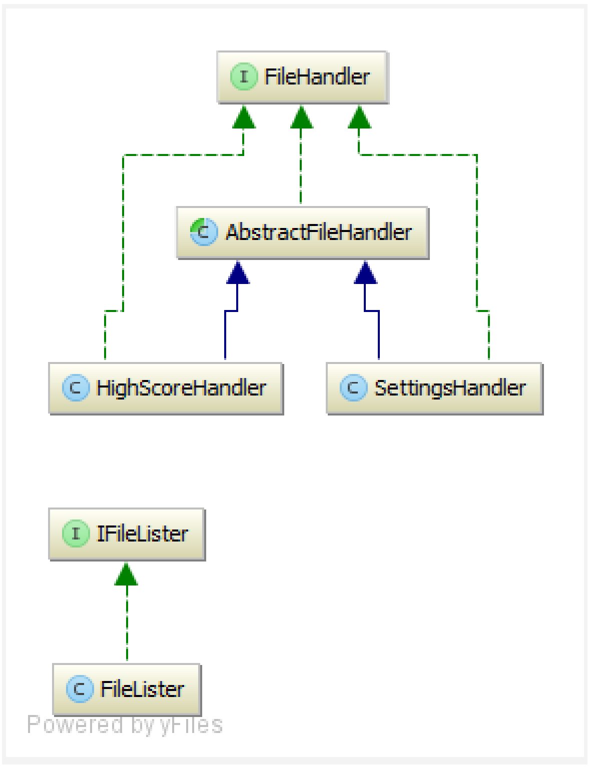
The different screens in Morf



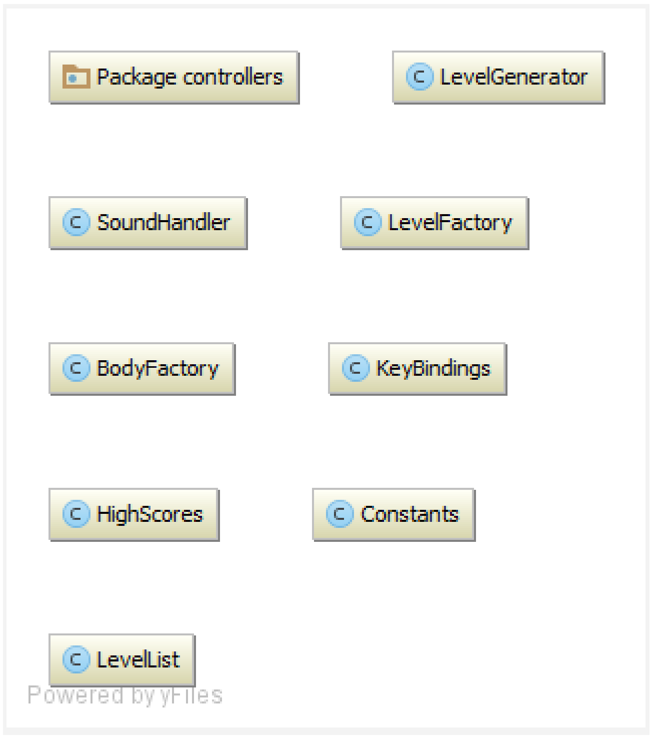
Call chains between the different screens in Morf



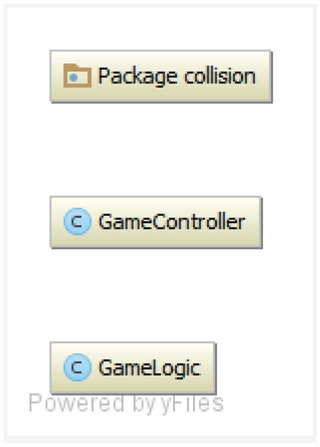
edu.chl.morf



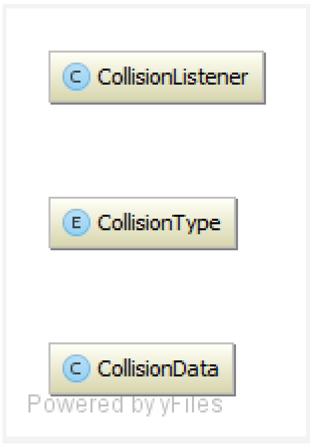
edu.chl.morf.file



edu.chl.morf.handlers



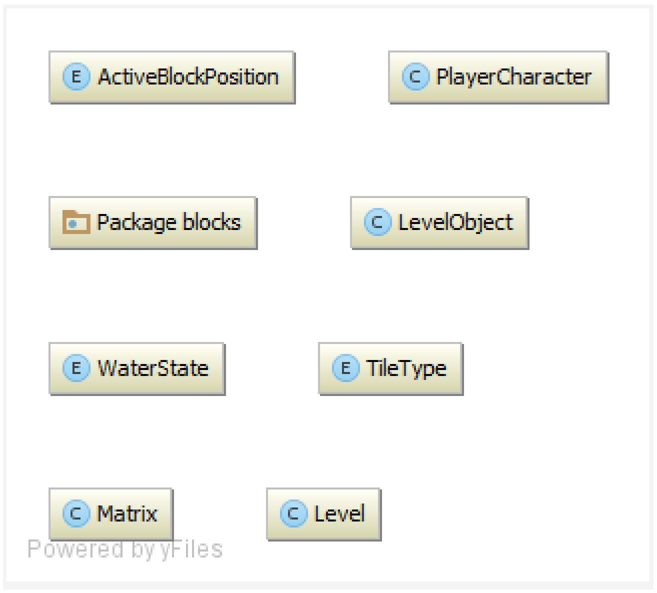
edu.chl.morf.handlers.controllers



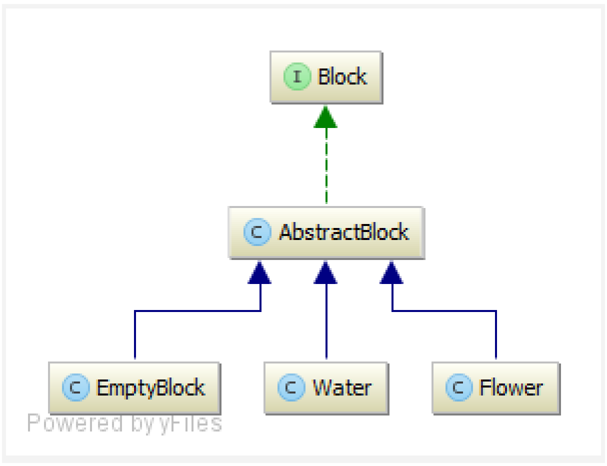
edu.edu.chl.morf.handlers.controllers.collision

---

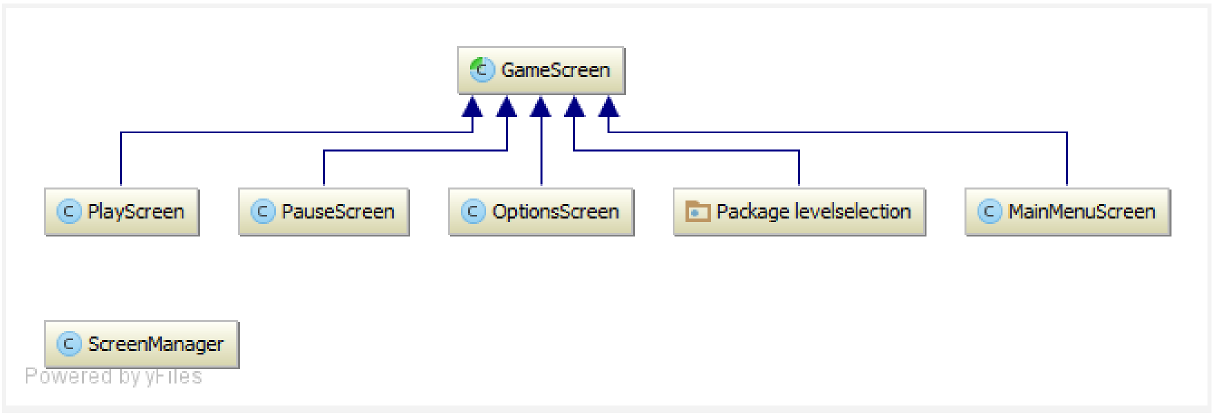
edu.chl.morf.main



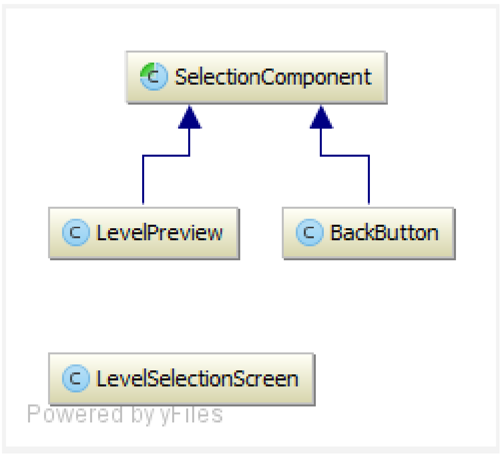
edu.chl.morf.model



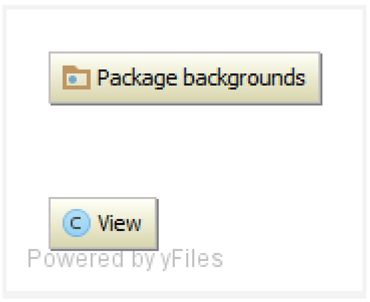
edu.chl.morf.model.blocks



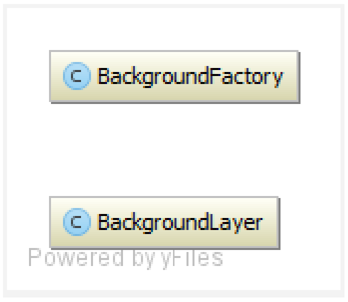
edu.chl.morf.screens



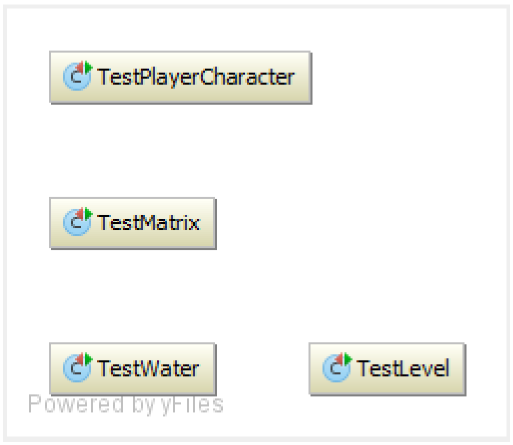
edu.chl.morf.screens.levelselection



edu.chl.morf.view



edu.chl.morf.backgrounds



test.edu.chl.morf.model