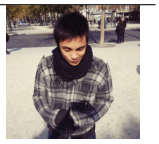
Emergent Architecture Design

Project members:		
Derk-Jan Karrenbeld	4021967	
Joost Verdoorn	1545396	
Steffan Sluis	4088816	
Tung Phan	4004868	
Vincent Robbemond	4174097	

Last edited: June 14, 2013

Contents

1 Introduction

This document contains the architectural design for the application created during the Context Project 'Programming Life: Synthetic Biology'. The application is targeted at synthetic biologists and its main purpose is to easily model, simulate and validate the complex workings of a cell.
The design of this application is explained first in terms of its design goals. Then a subsystem decomposition follows, which serves to uncover the inner workings of the application, along with a description of the mapping of subsystems to processes and computers, a hardware/software mapping. After this, the management of data and shared resources is discussed.

1.1 Design goals

The application can model cells and the processes within. The main design goal is to make this task as easy and intuitive as possible. Further, as requested by the client, it should:

- give an indication how changes to the cell will influence other parts of the cell
- have the ability to easily import and export datasets from and to frequently used data-formats e.g. CSV, HTML, PDF and Excel.
- give feedback to the user about mistakes and when possible provide a solution. An example is telling the user components are missing for a correctly working cell.
- simulate the cell by integrating the corresponding system of ODEs
- compare simulations by plotting data on top of other simulations.
- be available offline, once the application is loaded once.
- have the functionality to share data with other users.
- have an easy way to undo changes.

1.2 Additional goals

To broaden their knowledge and expand their experience, the developers agreed to use programming languages and frameworks previously unknown to them.

2 Architecture

2.1 Overview

Because the application is available offline, the server-client relation is decoupled. This means that the main functionality resides on the client. The server is merely used for administrative purposes, storing application and user data and providing the client-side functionality. A back end on the server also allows for data manipulation on the database.

2.2 Server

The primary function of the server is to provide the client-side application. After the application is downloaded the first time, it will reside on the client. Each time a client is online the application is automatically updated.
The secondary function of the server is to provide storage for everything created by the user. Sharing data is made possible by the storing functionality.
The figure below illustrates this concept. The server provides all client side assets [1] and outputs these assets and html files to the client. The clients have a local copy of the database, which is synchronised when the client is able to connect to the server.

[1] In this context assets are images, Javascript files and style sheets

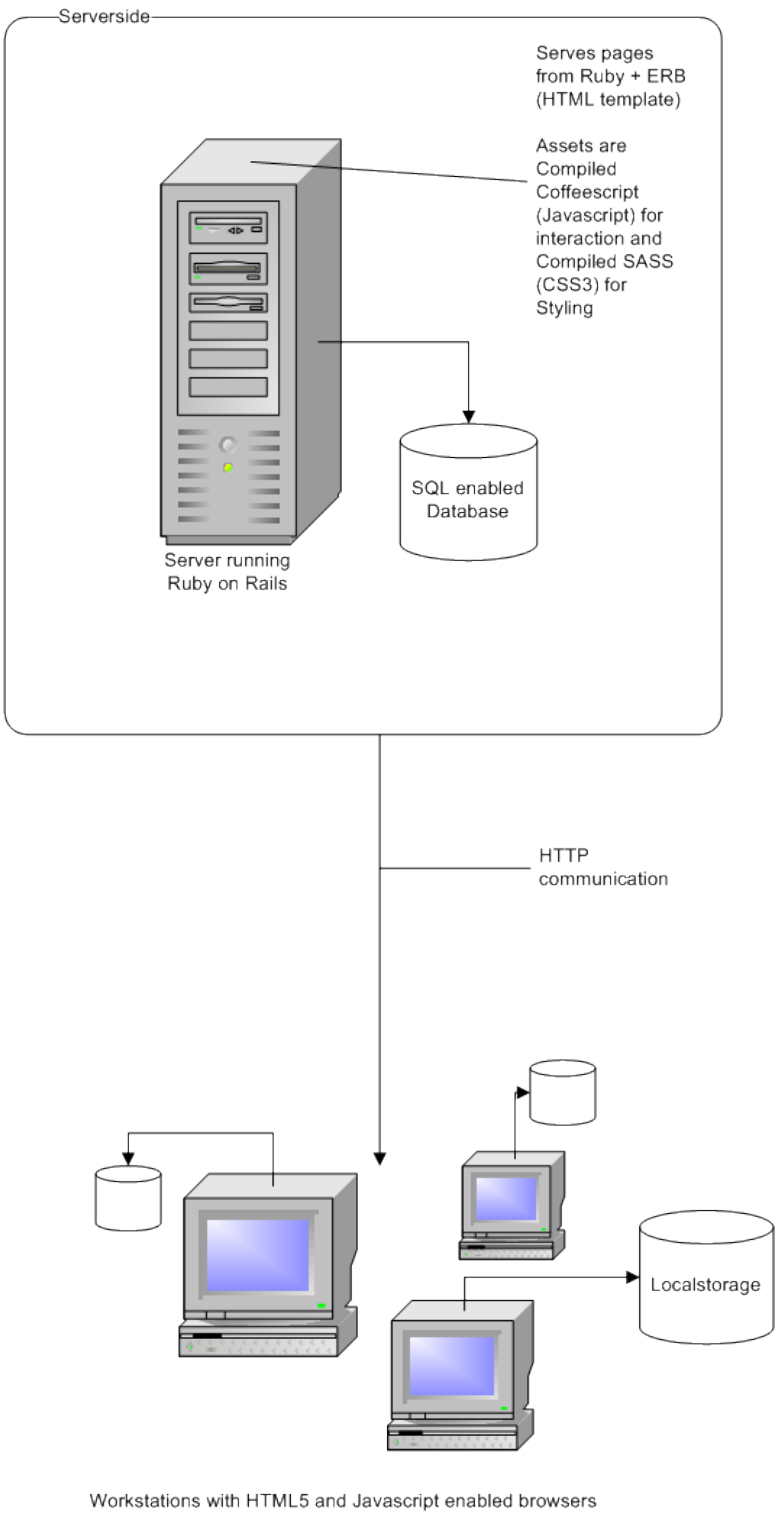


Figure 1: Deployment Diagram

2.2.1 Rails

The server runs on Ruby on Rails [1] . This is a fairly new but very stable platform that is not only free but also open source. This means it is being actively developed by a large number of people. Problems are easily fixed and this should ease the use of the application that is being designed. The language itself is written from a standpoint where you should just be able to write code and not worry about breaking the interpreter [2] . This makes it easy to write comprehensible code that executes complex tasks.

2.2.2 MVC on the server

The pages served are HTML5 for markup with CSS3 for styling and Javascript for interaction. Pages are built by a comprehensive and solid Model-Viewer-Controller system. This keeps data separated from the representation, further increasing maintainability. Interaction is illustrated in Figure 3. Rails uses MVC at its core. It makes it easy to create the back end views and client side representations because displaying data, the interaction and the actual data are separated.

2.2.3 MVC Models and Database Design

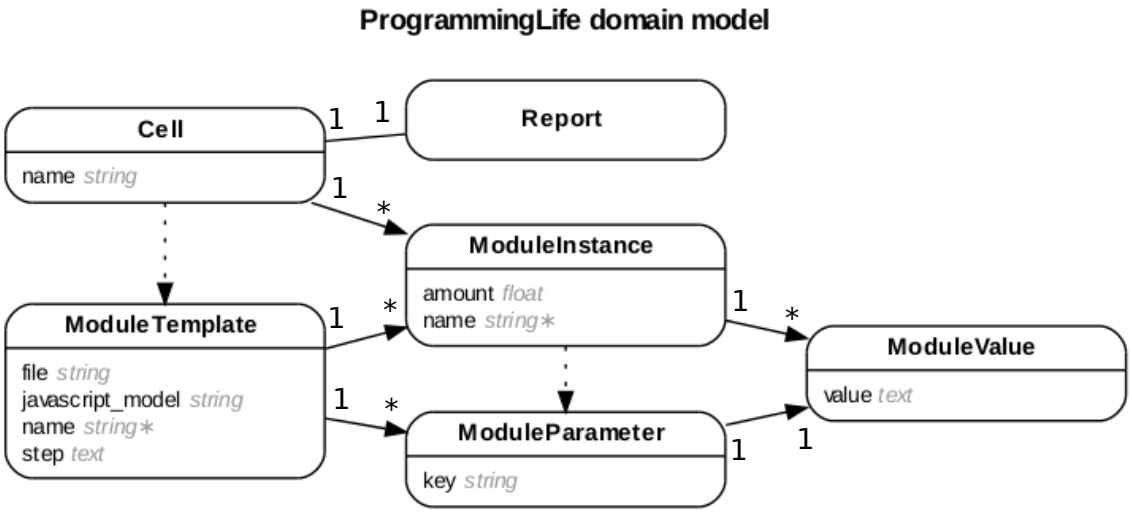


Figure 2: Database and Ruby Models

A report always shows the current cell data, so there is a one to one relation between reports and cells. The component definitions are separated from the values. A cell contains components, called modules (module_instance). These components are instances of blueprints (module_template). Each component blueprint can provide multiple parameters (module_parameter). Each instance has a value for each parameter (module_value).
The database design was devised this way, so templates are separated from instances, resulting in no data redundancy.

2.2.4 Active Database

Ruby models are mapped to any Relational Database Management System (RDBMS [3]) such as SQLite, MySQL and PostgreSQL. By not restricting the server database technology, switching systems, servers or extending their capacity should be fairly simple and easy. Database operations are done through Active Database, part of Rails, which transparently maps the query to the technology's language.

2.2.5 Views and ERB

The views are in ERB[4] which is an HTML template system. It comes with the Rails framework, so it does not require any extra software. Rails controllers written in Ruby can expose data to these views. Using a templating system such as this allows for quick translation from model to display.

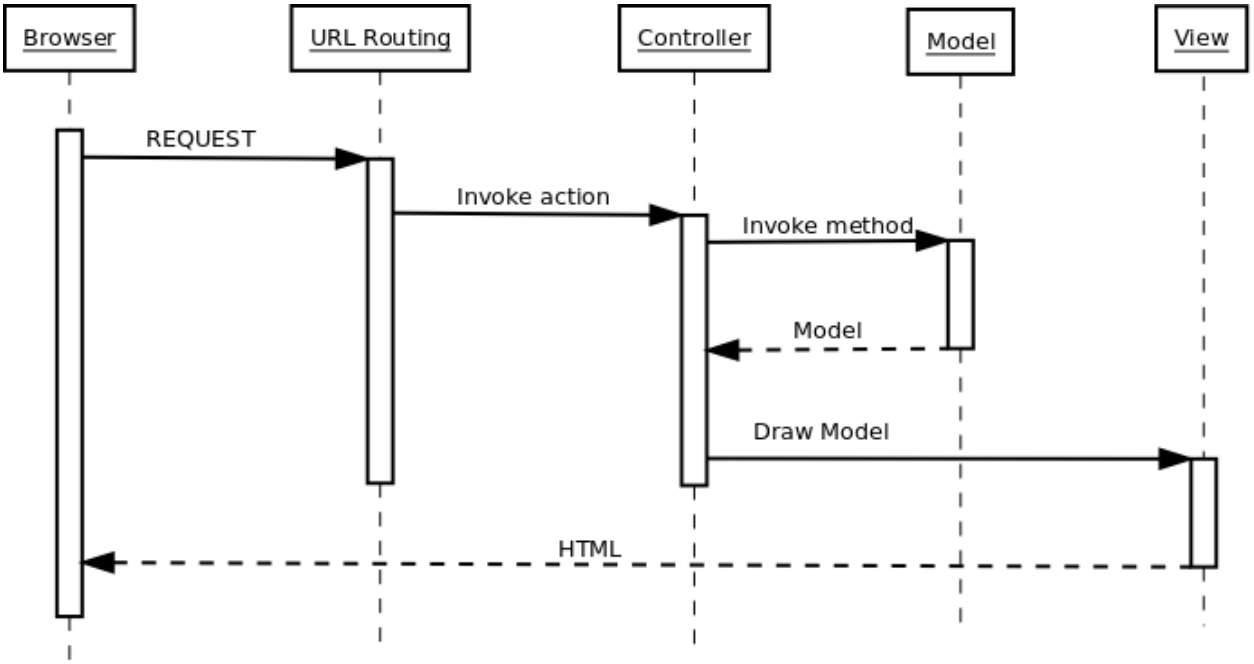


Figure 3: Sequence Diagram

2.3 Client

The server compiles the client application from Coffeescript to Javascript. This application is completely seperated from the server side functionality except for saving, loading, sharing and the production of reports. The client application is also built upon MVC.

2.3.1 Models

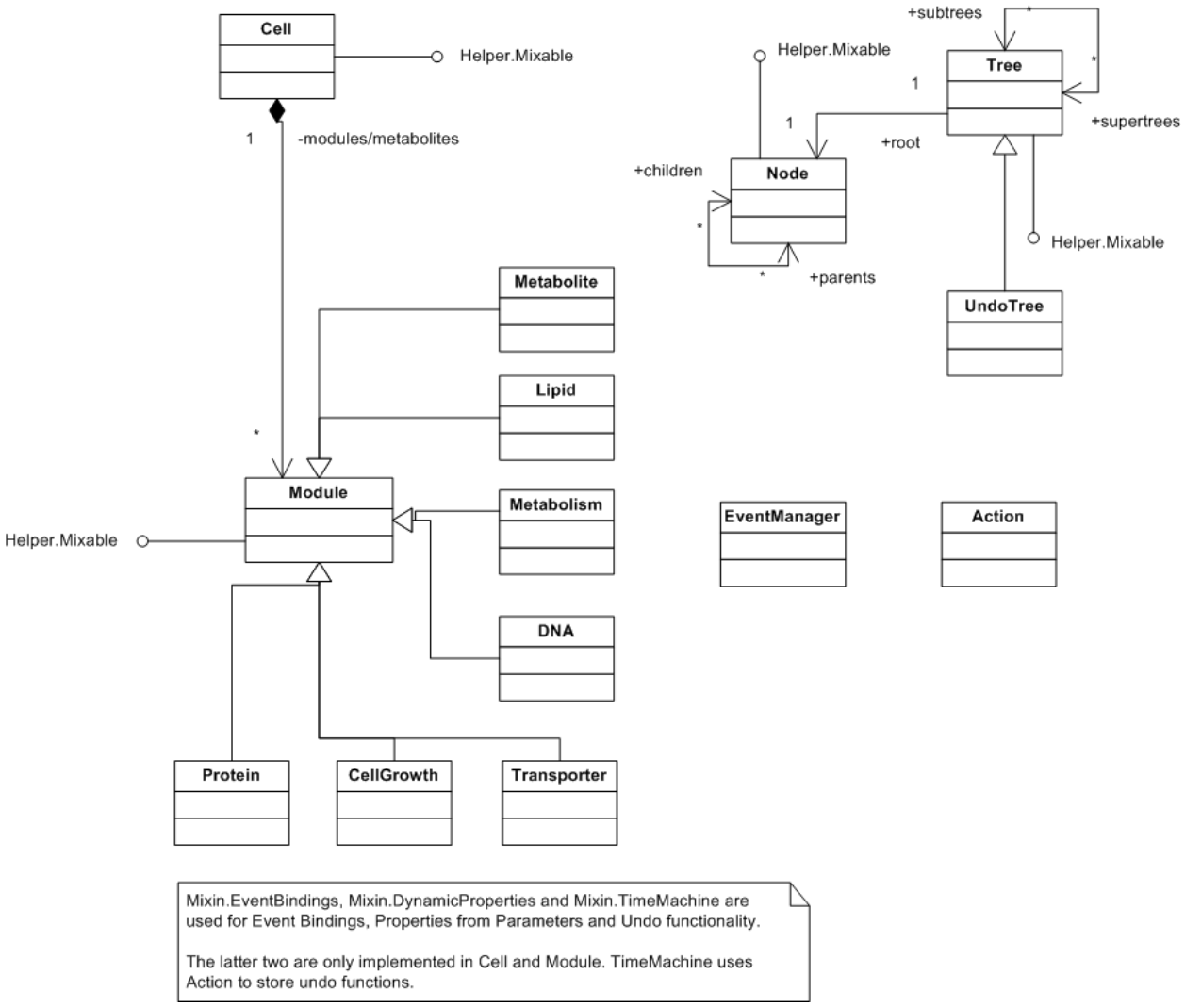


Figure 4: Client Models

There are two data models and a few meta models. The cell has it's own model and so do the modules, cell components, such as transporters and metabolites. The JavaScript classes can be seen as the ModuleTemplate and ModuleParameter classes on the server. The contents of the classes, the actual data, are the ModuleInstance and ModuleValue classes. Each model has a bit of functionality but only to ensure data consistency and correctness. Each of these data models can load and save itself.

To provide for undo functionality, the tree, node, undo tree and action classes are present. The former three are pretty transparent. The latter is a closure class used as node data in the tree.
The Event Manager adds communication between arbitrary parts of the application. Objects can trigger events or listen on it.

2.3.2 Views

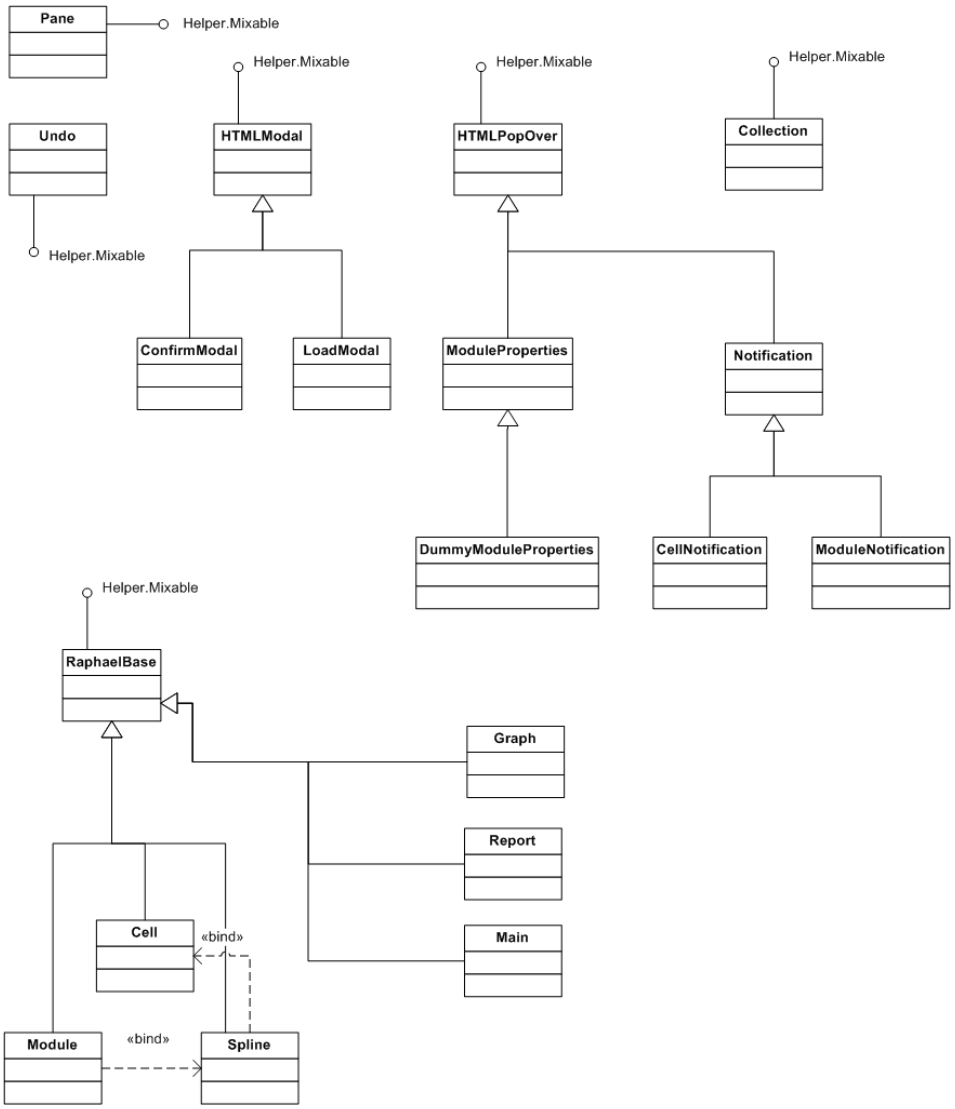


Figure 5: Client Views

The views are of two basic types: Raphael or HTML. There is a meta class Collection which just holds views. All views can have children and display a single model. Raphael typed views use SVG to display their contents. To display a model differently, new views can be added. The views only have display functionality. They do not add interaction.
Events on the models ensure views to be able to update their display. This way a view is notified of changes instead of needing to poll for changes.

2.3.3 Controllers

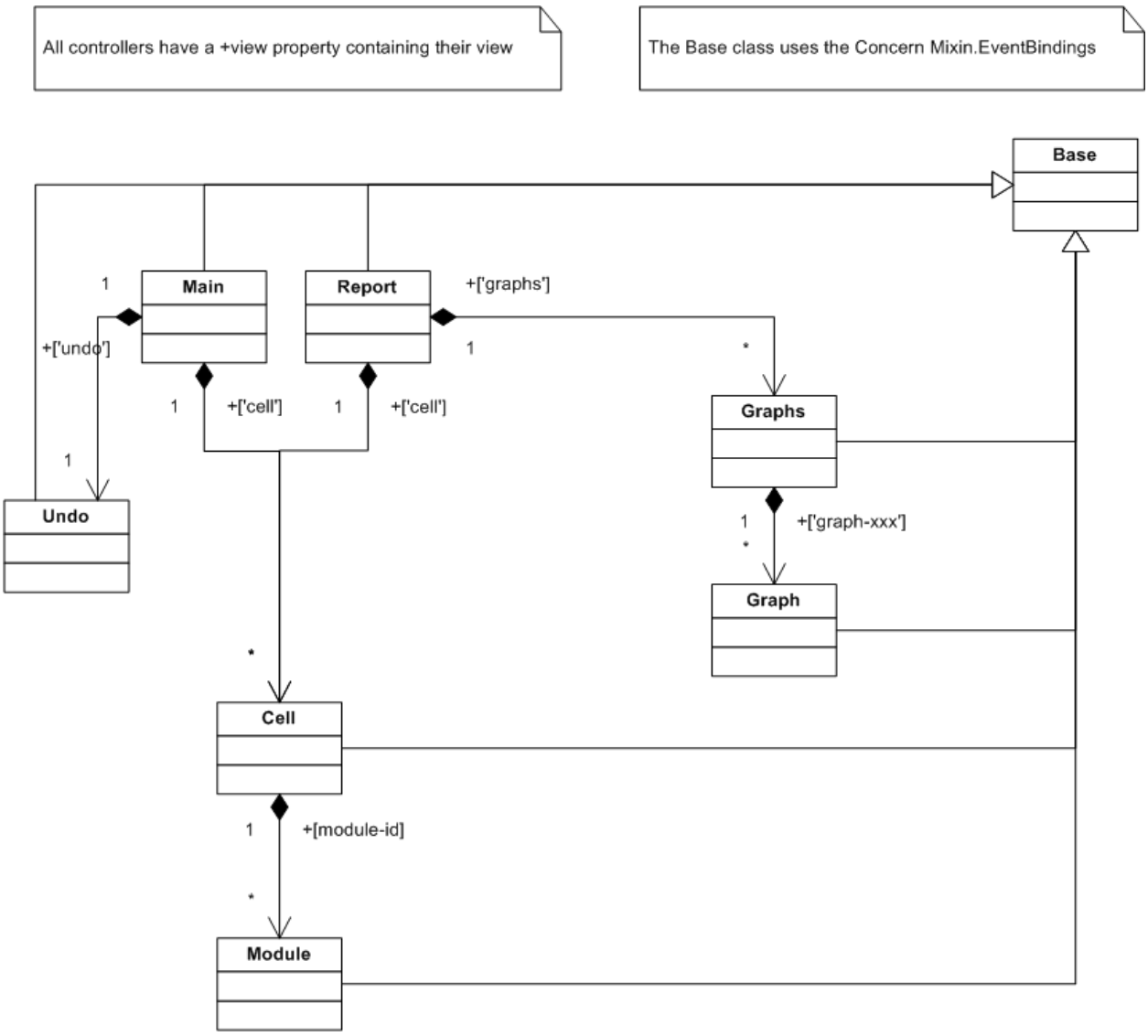


Figure 6: Client Controllers

Controllers add interaction to views and models. Controllers can have children. Parent controllers can alter a child controllers behaviour. This way there is no need for a large set of controllers, e.g. a MainCell controller and a ReportCell are now combined into one Cell controller.
For collections of elements, views can use unobtrusive JavaScript to enlist those elements for interaction. A view might add data-xxx attributes to elements. Controllers may or may not bind Javascript to elements with these attributes. Keeps the HTML clean and the interaction in the controllers hand.

2.3.4 Helpers

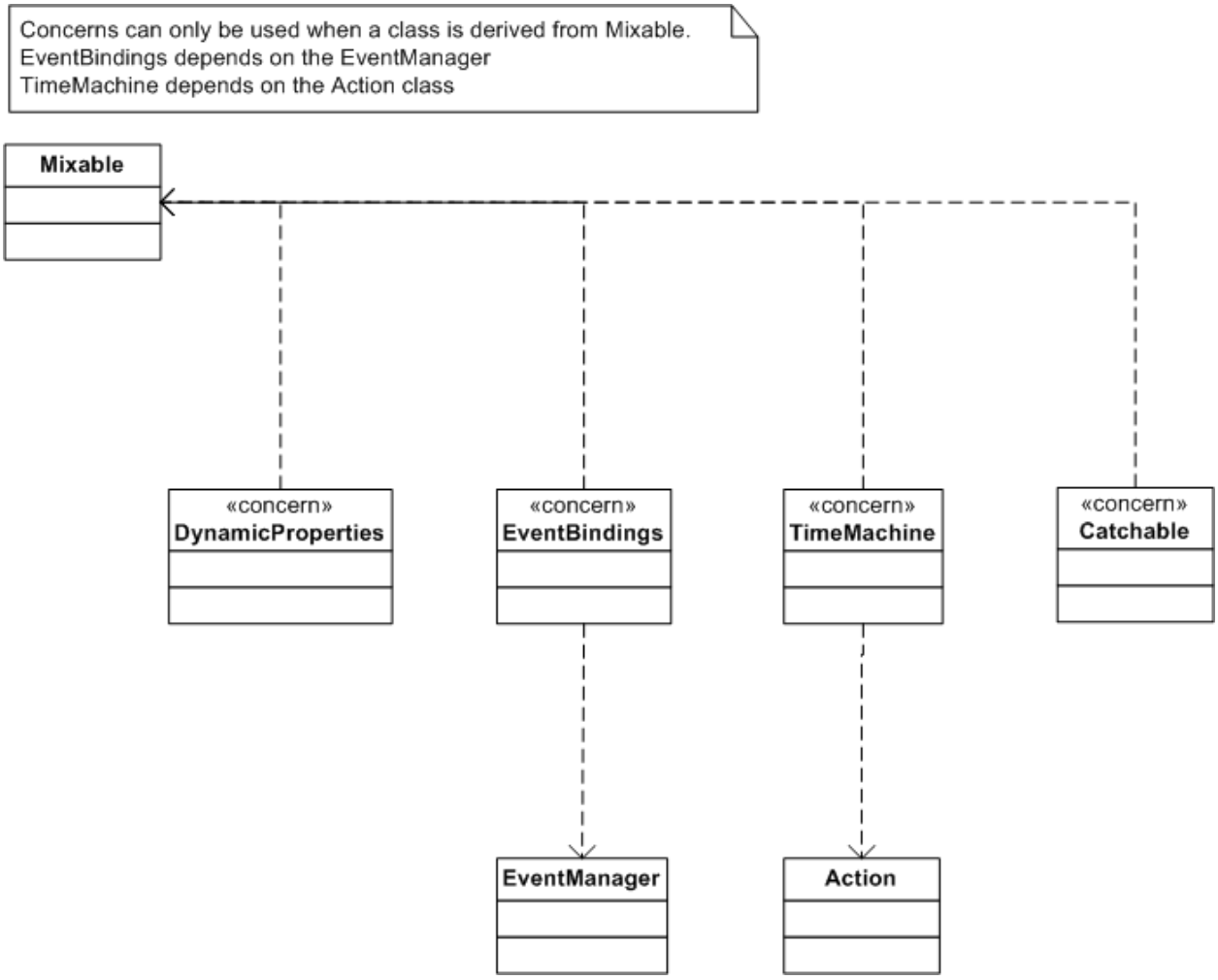


Figure 7: Client Helpers

There is a set of mixins used by many models, views and controllers. DynamicProperties adds the ability to dynamically create Javascript properties. Define getters, setters, non enumerable values and so forth and on. The Cell and Module models use this extensively. TimeMachine adds undo functionality to a class. EventBindings enable listening and triggering events, used by almost all classes. Catchable can enclose functions into try catch blocks and trigger callbacks and events instead of breaking the application. Finally there is the Mixable base class. To use mixins, this class needs to be in the prototype chain.

2.3.5 Local storage

Javascript determines the local storage engine that is available and uses that to maintain an offline copy of the application and the created cells/modules. This makes it very easy to design new cells or edit saved cells even when the user isn't connected to the internet. This is communicated to the user, but requires no further interaction. Synchronization is completely transparent.

2.4 Data management

When the user has access to the server, it can export simulation results to a report in multiple standardised format, such as *HTML*, *Excel* and *PDF* which can then be shared. Maintaining a standardised format allows for consistency in the reports as requested by the user.

2.5 Concurrency

Each client runs independently and uses asynchronous communication with the server through *REST/CRUD* and *AJAX*. Because of this, concurrency issues are very improbable. The client-side application is web-based, and uses only one process. Shared resources are retrieved atomically from and synchronised atomically with the database. Upon failure, the user is notified and possible solutions are provided. At this point the system does not allow concurrent edits on the same data, but this functionality was not requested by the user. Such functionality could be added in the future.

3 Software Technology

This section contains all the information about the software architecture of the application. The section is divided into several subsections to group together interesting information and improve readability.

3.1 Framework and Development

This section describes the key technologies used during a release cycle. It is divided into three subsections; development, testing and production. The first subsection is about the development phase, where features are realised. The second section is about the testing phase, in which the features mentioned before are tested. The third section describes the production/release phase, in which a tested feature is implemented.

3.1.1 Development

During development we are making use of Coffeescript[5] for writing interaction assets, SASS[6] for styling and SVG[7] with the Raphael[8] framework for visualising part of the application. Coffeescript and SASS are preprocessors for respectively Javascript and CSS3. Using these preprocessors gives us the advantage of cleaner syntax. By making use of SVG to render the cell every graphical object is also a DOM object, this enables us to attach Javascript event handlers and modify the objects itself.

3.1.2 Testing

We test the application in a Behaviour Driven Development (BDD [9]) and Test Driven Development (TDD [10]) way. The Javascript assets are tested using the Jasmine framework [11]. To run all tests in the browser we are making use of the Teabag framework [12]. To keep track of code coverage we are making use of the Istanbul framework [13]. The Ruby assets (database, MVC architecture - models, controllers, views - and Rails) are tested using the Rspec framework [14]. All tests run automatically using Continuous Integration, a practice in which tests are ran every time work has been done within critical sections of the code. Tests and coverage are used to ensure maintainability, consistency and quality of the code.

3.1.3 Release

During the Production release the Coffeescript assets are compiled to Javascript[15] and the SASS assets are compiled to CSS, after which they are minified and bundled using uglifier[16]. Minified assets take up less space and increase page performance which results in a better user experience and less demand on the web server. SVG is still used for creating visualisations of the cell models.

3.2 Ordinary Differential Equation Solver

All computations run client-side and are performed by our ODE Solver, which is built upon the numericjs[17] library. This is a library which makes it possible to perform sophisticated numerical computations in pure JavaScript and thus is able to run in the browser. The ODEs are solved using the Dormand-Prince RK method[18], also known as the ode45-function in MATLAB (previously used by our client).

3.3 API

The API can be found online here: `http://coffeedoc.info/github/Derkje-J/programming-life/`
In the future there will be a Rubydoc online for server-side assets and is updated after each commit.

4 Summary

The application is a lightweight, cross-platform graphical design tool with a centralised storage database. The architecture ensures its functionality on different kinds of machines as well as the ease of simulating complex cell models. It not only offers stability, ease and intuitive design, it offers it on every modern machine. During development of this application the design goals are continuously used as a mandate for the functionality and dictates the production flow.

5 Glossary

This section explains any and all terms that may be ambiguous or unclear:

REST: Representational State Transfer, or REST, is an architectural style of large-scale networked software that takes advantage of the technologies and protocols of the World Wide Web.
See http://goo.gl/Lfwhs for more information.

5.1 List of abbreviations

- **AJAX**: Asynchronous Javascript and XML
- **CSS**: Cascading Style Sheets
- **CRUD**: Create Read Update Delete
- **CSV**: Comma Separated Values
- **DOM**: Document Object Model
- **ERB**: Embedded Ruby
- **HTML**: HyperText Markup Language
- **MVC**: Model-View-Controller
- **PDF**: Portable Document Format
- **RDBMS**: Relational DataBase Management System
- **SASS**: Syntactically Awesome Style Sheets
- **SVG**: Scalable Vector Graphics
- **SQL**: Structured Query Language

Bibliography

[1] Ruby on Rails, *A web-application framework to create database-backed web applications according to the Model-View-Controller (MVC) architecture.* http://api.rubyonrails.org/

[2] Bill Venners, *The Philosophy of Ruby.* 2003. http://www.artima.com/intv/rubyP.html

[3] Relational Database Management System, *a database management system that is based on the relational model.* http://en.wikipedia.org/wiki/Relational_database_management_system

[4] ERB, *a simple but powerful templating system.* http://ruby-doc.org/stdlib-2.0/libdoc/erb/rdoc/ERB.html

[5] Coffeescript, *a language that compiles into JavaScript.* http://coffeescript.org/#language

[6] SASS, *an extension of CSS3.* http://sass-lang.com/docs/yardoc/file.SASS_REFERENCE.html

[7] SVG, *an XML-based vector image format for two-dimensional graphics that has support for interactivity and animation.* http://www.w3.org/Graphics/SVG/

[8] Raphael, *a JavaScript library that should simplify working with vector graphics on the web.* http://raphaeljs.com/reference.html

[9] Behaviour Driven Development, *a software development process based on test-driven development (TDD).* http://en.wikipedia.org/wiki/Behaviordriven_development

[10] Test Driven Development, *a software development process where tests are written first* http://en.wikipedia.org/wiki/Testdriven_development

[11] Jasmine, *a behavior-driven development framework for testing JavaScript code.* http://pivotal.github.io/jasmine/

[12] Teabag, *a Javascript test runner built on top of Rails.* https://github.com/modeset/teabag

[13] Istanbul, *a Javascript code coverage tool written in Javascript.* https://github.com/gotwarlost/istanbul

[14] Rspec framework, http://rubydoc.info/gems/rspecrails/frames

[15] Javascript https://developer.mozilla.org/en-US/docs/JavaScript/Reference

[16] Uglifier, *Ruby wrapper for UglifyJS JavaScript compressor.* http://rubydoc.info/gems/uglifier/2.1.0/frames

[17] NumericJS, *a library which allows you to perform sophisticated numerical computations in pure Javascript in the browser and elsewhere.* http://www.numericjs.com/

[18] Dormand-Prince RK method , *a method for solving ordinary differential equations.* http://en.wikipedia.org/wiki/Dormand%E2%80%93Prince_method