UPTEC X07 010

Examensarbete 20 p
Februari 2007

Design and implementation of a laboratory information system for cellular pharmacology

Jonathan Alvarsson

UPPSALA
UNIVERSITET

Bioinformatics Engineering Program

Uppsala University School of Engineering

UPTEC X 07 010	**Date of issue 2007-01**
Author **Jonathan Alvarsson**	
Title (English) **Design and implementation of a laboratory information system for cellular pharmacology**	
Title (Swedish)	
Abstract A laboratory information system for handling high-throughput drug screening and low- to medium-throughput bioassays in cancer research performed by small to medium-sized academic groups was designed and partly implemented. All parts except for the graphical user interface have been implemented. The system has functionality for keeping track of when who did what, and it provides an annotation system for the objects of the system. The system was implemented in Java using the object relational manager Hibernate and the lightweight framework Spring.	
Keywords Laboratory information system, cancer research, high-troughput screaning, Hibernate, Spring	
Supervisors **Rolf Larsson** **Uppsala University, Clinical Pharmacology**	
Scientific reviewer **Mats Gustafsson** **Uppsala University, Signals and Systems**	
Project name	Sponsors
Language **English**	Security
ISSN 1401-2138	Classification
Supplementary bibliographical information	Pages **47**

Biology Education Centre Biomedical Center Husargatan 3 Uppsala
Box 592 S-75124 Uppsala Tel +46 (0)18 4710000 Fax +46 (0)18 555217

Design and implementation of a laboratory information system for cellular pharmacology

Jonathan Alvarsson
jonathan.alvarsson@gmail.com

February 28, 2007

Svensk sammanfattning

Inom dagens cancerforskning är en vanlig metod för att hitta intressanta läkemedelskandidater att man söker igenom stora bibliotek av molekyler genom att testa dem emot cancerceller. Den här genomsökningen är ofta automatiserad och genererar stora mängder data. I detta examensarbete designades och påbörjades implementeringen av ett informationssystem för att hantera den mängd av data som genereras. Som bas användes det öppna källkodsprojektet *Bioclipse* som innehåller funktioner för att hantera bioinformatisk data.

Systemet har funktionalitet för att se vilken användare som har gjort vad och när. Användare kan även skapa egna annoteringar till objekten i systemet.

För att underlätta genomförandet testades varje komponent i systemet i en separat miljö; på det viset kunde många fel upptäckas tidigt i projektet. En komplett uppsättning tester underlättar även vid förändringar, då tappad funktionalitet direkt kan upptäckas med testerna.

Projektet kunde inte slutföras inom ramarna för detta examensarbete. Vid examensarbetets slut var design och implementation av kärnstrukturen för hela systemet utom de grafiska delarna, däribland inmatning av data och presentation av data och resultat, klart.

Examensarbete 20p, Civilingenjörsprogrammet Bioinformatik

Uppsala Universitet

Contents

1 Introduction

The main aim of this project was to design and initiate the implementation of a laboratory information system (LIS) able to handle data from both the high-throughput drug screening and the low- to medium-throughput bioassays in cancer research performed by small to medium-sized academic groups. A LIS is a data system that receives, processes, stores and delivers information generated by medical laboratories. This project has been carried out at the department of medical sciences at Uppsala University. The program was written in *Java* and use the functionality of the open source project *Bioclipse* as a base.

1.1 Overview

In current cancer research, large amounts of chemical compounds are examined with regard to effects on cancer cells. This process consists of highly automated high-throughput processes as well as focused biological evaluation of more limited extent. A few different approaches exist which constitute the core activities at the department. These are discussed in the following sections.

1.1.1 Fluorometric microculture cytotoxicity assay

The first instrument to be handled in the system is an automated machine, an Optimised Robot for Chemical Analysis (ORCA; Beckman Coulter) equipped with a multipurpose reader (FLUOstar Optima, BMG Labtech GmbH, Offenburg, Germany), for fluorometric microculture cytotoxicity assay (FMCA). FMCA estimates cell death by measuring the amount of non-fluorescent fluorescein diacetate (FDA) having been transformed into a fluorescent by cells with intact cell membranes[1]. The measurements are performed on microtiter plates with 96 or 384 wells.

First, the wells on the plates are prepared with the drugs of interest; then they are seeded with cells and incubated. After incubation, the plates are centrifuged and washed and FDA is added, and after further incubation, the fluoroscence is measured. See figure 1 for a picture of the complete process.

1.1.2 Screening using an annotated compound library

Large amounts of compounds are collected in libraries and tested on tumor cells. Highly potent drugs can be identified through screening, and hypotheses about biological mechanisms can be generated from the pattern of activity in different model systems and by combining these data with gene-expression[2]. The current library consists of more than 6500 compounds. Screening data consist of fluoroscence measurements for a few data points (often of one concentration) for each compound.

1.1.3 Dose response

Interesting compounds are subsequently tested at different concentrations, and survival of the cells are plotted as a function of the concentration in order to produce a dose response curve. From this curve, the EC_{50} value can be calculated. EC_{50} is defined as the statistically estimated concentration needed for 50% effect, in this case 50% cell death.

1.1.4 Compound combination effects

It is also of interest to study the effects of combinations of identified active compounds. An additional compound may counteract drug resistance. That is, a cell resistant to drug A may be sensitive to a combination of drug A and drug B, since drug B neutralises the cell's protection against drug A.

1.1.5 Information systems

At the moment, no satisfactory solution exists for dealing with all data generated in these processes. A few separate systems are in use at the lab, neither of which fully meets everyday requirements.

SLIMS Small Laboratory Information System[3] is an open source product very suitable for screening data but it does not support operations such as constructing dose response curves. It contains some very nice tools such as the self-organising maps functionality, which is a way to generate a visual representation of the chemical space spanned by the compounds. It is thus possible to see if interesting compounds lie close together or far apart. Not much seems to be happening with it, since in the news section of the project's webpage, the last update is from August 2004, although the latest version is from March 2006. *SLIMS* is implemented using the programming language *Python*.

Accord (Accelrys Software Inc.) is a complex system that turned out to be quite difficult to adopt to the daily work at the laboratory. It seems to be able to do many tasks but is not user-friendly. It is a powerful tool that does not really make up for the

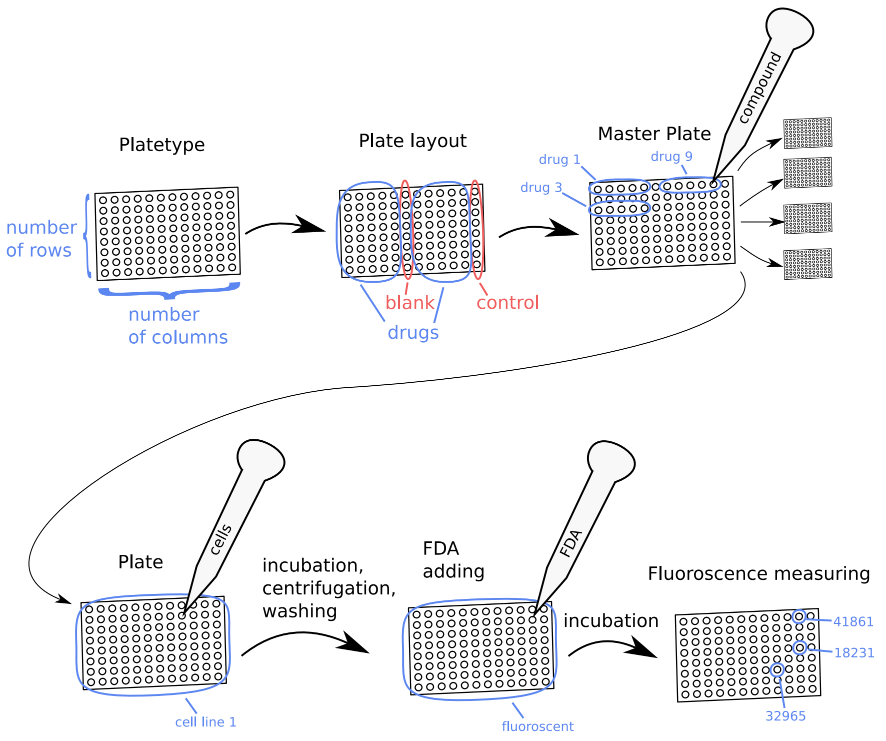


Figure 1: *Introduction to a few objects in the system and a brief description of the FMCA process. In the system a plate type defines the size, number of columns and rows, of a plate. A plate layout defines where on the plate the controls and the compounds are to be placed. Based upon this plate layout a number of equal plates are made, conforming to a so-called "master plate" that defines which drugs are placed in which wells. Each one of these plates is seeded with one cell type and incubated. After incubation, the plate is centrifuged and washed, and fluorescein diacetate is added. Then the plate is washed again, and incubated once more, before fluoroscence generated from fluorescein diacetate transformed into a fluorescent by cells with intact cell membranes is measured with a microplate fluorometer.*

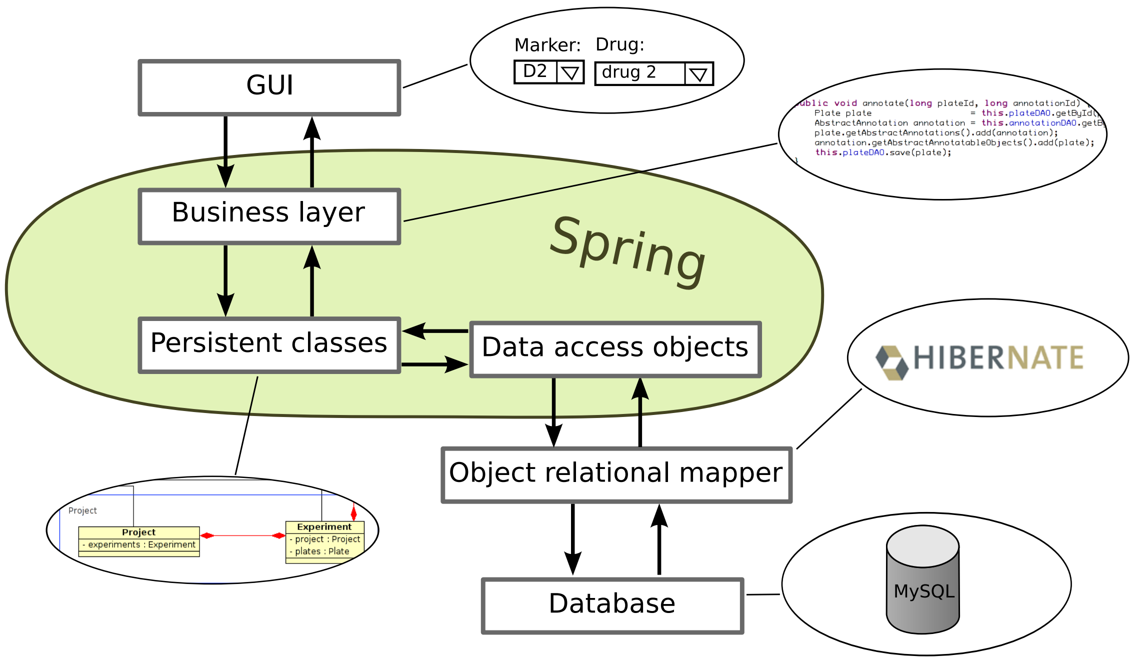
learning and configuration cost with its functionality.

Some in-house *Matlab* code also exists for interpreting measurements on plates and for colour-coding the results. In conclusion, it has become apparent that a tailor-made laboratory information system (*LIS*) is needed, and the *Bioclipse* project (see section 3.1.7) already contained a lot of the needed functionality, so it was deemed appropriate as a foundation. An advantage of the tailor-made solution is that the lab would keep the source code and be able to extend and perform changes to the implementation when needed in the future.

2 Specification and design

The first thing to do when starting a project like this is not to sit down with a computer and write code, but rather to sit down with a pen and paper, specify what the program should be able to do, and design a structure able to do it.

It is not really suitable to speak about a special software development process approach — such as the waterfall model[6] — for this project, since the number of developers during this project has been one. But an iterative process[7] supported by tests and somewhat inspired by the programming approach known as extreme programming[8] might



```
ublic void annotate(long plateId, long annotationId) {
    Plate plate                         = this.plateDAO.getById(
    AbstractAnnotation annotation = this.annotationDAO.getB
    plate.getAbstractAnnotations().add(annotation);
    annotation.getAbstractAnnotatableObjects().add(plate);
    this.plateDAO.save(plate);
}
```

Figure 2: *The overall structure of the program. See section 2.2 for a short description of all the components.*

be a fair description of the approach.

This section deals with the definition of the graphical user interface, the functionality of the program, and data modelling.

2.1 Graphical user interface

As a start, a sketch-like specification of the graphical user interface (GUI), containing components for working with data about the plates used in FMCA, was constructed (see appendix A). The development of the GUI specification was an iterative process were the program took form during interviews and discussions with future users. The specification is not to be regarded as an exact representation of how the system is going to look, but rather a specification of the future functionality of the program illustrated by examples to help the reader get a feeling for how it may look. The GUI specification is not a finished document, but a document that has been used on a daily basis throughout the project, and thus contains parts that are likely to change.

2.2 General structure of the system

When building programs above a certain level of complexity, it is desirable to use standard components in order to be able to focus on the uniqueness of the problem at hand instead of already solved standard problems. It is a bad thing to become stuck with one solution and not be able to switch to another, hence, the components of this kind of program tend to be ordered in layers, and in a perfect world it would be possible to swap a component in a layer for another one. These layers also have the benefits of taking care of a smaller part of the problem, and being able to concentrate on that in a divide-and-conquer manner.

Figure 2 shows a graphical representation of the overall structure of the program. At one end is the graphical user interface (GUI) with buttons, text-fields and so on, and at the other end is the database.

There are mainly three different sorts of objects in the system, *managers*, *persistent objects* and *data access objects*. This follows the standards from the book *Pro Spring*[4] about implementing software using the *Spring* framework. *Spring* is a framework containing standard code helpful when building *Java* programs above a certain level of complexity, for example programs that work with databases. The GUI contains text fields, buttons and similar components that call methods of manager-objects in the business layer. These managers provide functionality for operations such as creating a plate with wells and all associated information objects. The managers have names such as AnnotationManager, SampleManager and PlateLayoutManager. They work with many smaller persistent classes that contain the actual data. The persistent classes repre-

Component	Description
Plate type	Defines the size (number of wells) of a plate
Plate layout	Defines in which wells for example dilution series and controls are to be placed, also defines calculation functions
Master plate	Defines which drugs are placed and by which concentrations (when working with dilution series) in which wells
Plate	Defines which cell type has been seeded on the plate

Table 1: *The different components stepwise constructed when creating a plate.*

sent the data that are being stored in the database and have names such as Plate, Well and CellSample. The actual storing and retrieving of data from the database is done by data access objects (DAOs). The DAOs work on one or a few persistent object each, saving it and some related persistent objects, as specified in *Hibernate*'s mapping files. Hence, there are more persistent objects than there are DAOs. The DAOs have names such as PlateDAO, UserDAO, and ProjectDAO. The DAOs communicate with the object relational mapper (ORM), *Hibernate* in this case, when saving and loading the persistent objects from the database. Lastly, there is a relational database, saving and retrieving the data from disk.

In this project the *Spring* framework provides important help in implementing the business layer and the data access objects. It contains standard code which eases the implementation of these layers.

2.3 Data structure specification

As the GUI was being sketched during discussions with the staff, in the background the process of constructing a data model able to handle all that functionality took place. Finding a structure for the data is also an iterative process. Hopefully, the model will tend to evolve towards something more and more obvious. But the path towards this "obvious" goal is all but clear and many ideas are constantly set aside for better ones during the lifespan of the program. The goal is that when new kinds of data need to be saved, adding them to the system should be possible without too much of a problem. The resulting class diagram for the persistent objects is shown in figure 3. If we study the classes Project and Experiment, handled by the ProjectManager, which can be found in the lower right corner of figure 3 we see among other things that both have a DAO. The red line with two diamonds at each end means that a Project has Experiments and the Experiments correspond to a Project. If we follow the black line coming out on top of them we see that both of the classes extends AbstractAuditableObject that extends AbstractAnnotatableObject which extends AbstractBaseObject. This means that Projects and Experiments are both annotatable and auditable. The dashed lines appearing to the right of figure 3 symbolise the way the objects are created. For example a PlateLayout is created from a PlateType.

Plates with a number of different layouts are used, and a certain layout with a set of drugs are used many times but with different cells. To reduce the repetitive work when creating plates, a plate is created in a couple of steps, and each step is saved. So when creating a new plate similar to one already created, the whole process will not have to be repeated. The components corresponding to these different steps are shown in table 1.

During the discussions, it became apparent that a system for specification of calculations to be performed at the plate and well level is needed. Two sorts of functions can be defined in the system, *plate functions* and *well functions*. Plate functions are functions acting on data from the whole (or a part of the) plate and well functions are calculated result values for a well. These are stored in the database as text strings containing mathematical formulas such as $(a4 - a3)/a2$. The variables are references to the raw data for the wells. More complicated functions that can be called from the calculation functions, such as a sum function that does not count wells marked as outliers, are planned.

2.4 Data input

Both compound data and result data are imported to the system from file. It should be possible both to type in drug data manually and to insert complete libraries with drug data from file into the system. The actual results will always be imported from file.

2.5 Managers

During the design of the managers, the ideal which was strived towards was a natural division into groups of persistent objects where each persistent object is managed by one manager. The blue boxes in figure 3 correspond to the responsibility area of each manager.

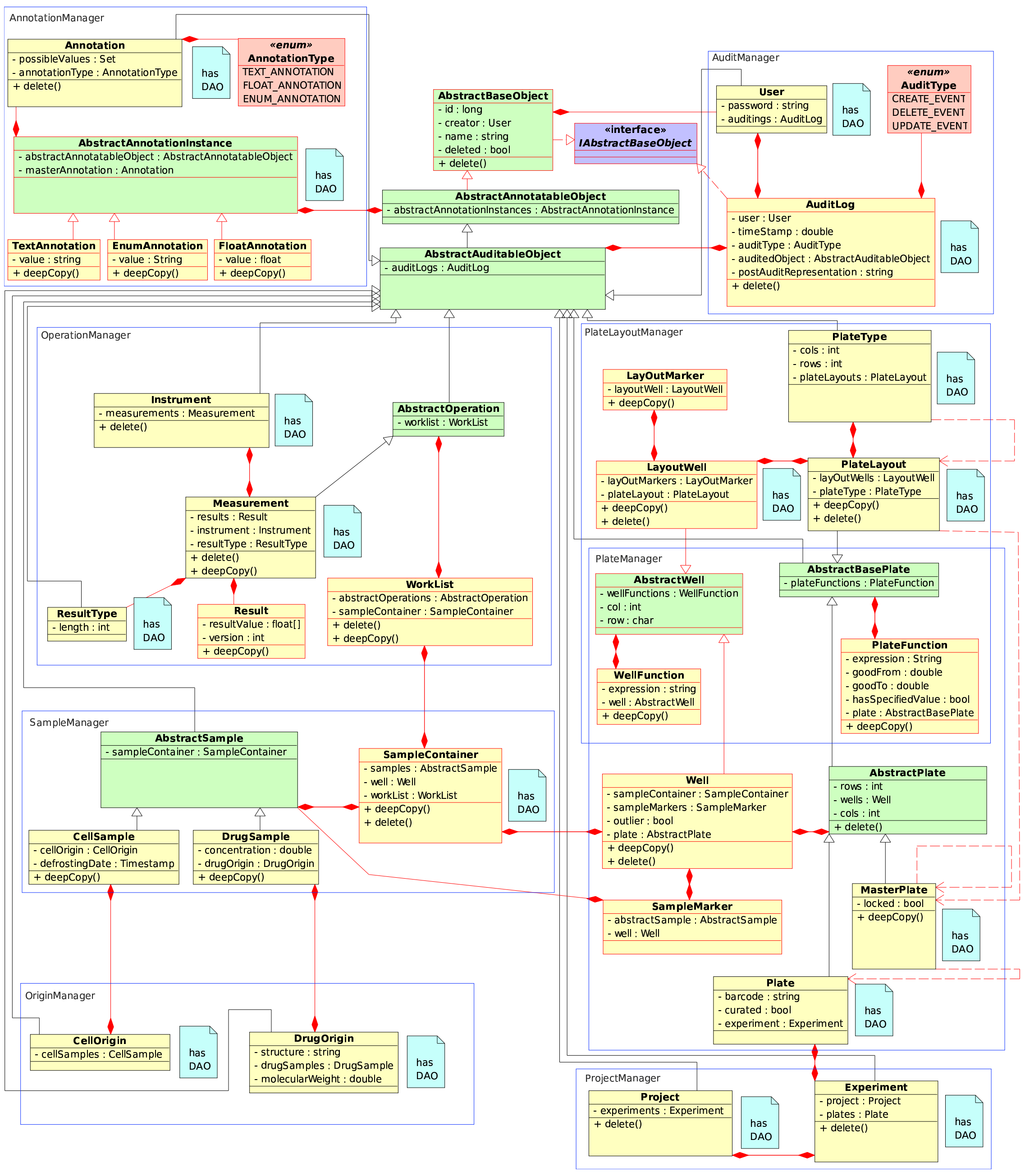
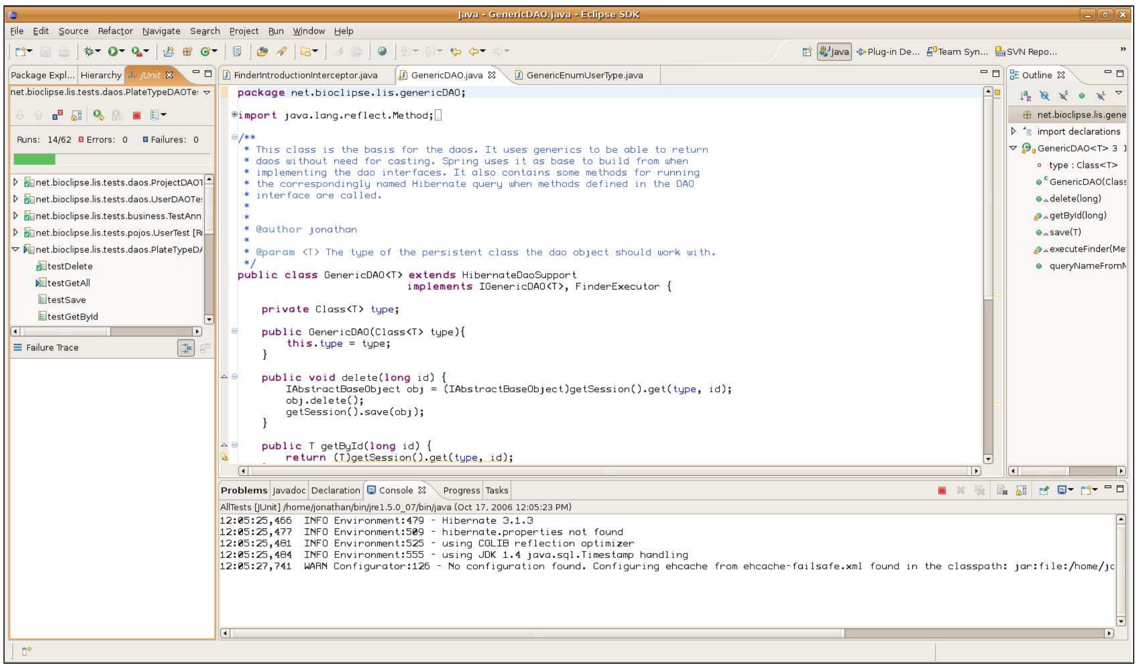


Figure 3: *The class diagram of the persistent classes. The blue boxes represent the managers and show which classes are handled by which managers, abstract classes are green, interfaces are purple, enums*[a] *are red and instatiated classes are yellow. All the persistent classes implement the interface* `IAbstractBaseObject` *which is not shown in the diagram in order to clean it up a little and some (represented with black frames in the figure) also extend* `AbstractAuditableObject`*. This diagram was created by the means of the software Umbrello[5].*

[a] *enums* where introduced in Java 5. They are types with a predefined set of values. A prototypical example of an enum is `dayOfWeek`, which can take on the values Monday, Tuesday, . . . , Sunday.



```
package net.bioclipse.lis.genericDAO;

import java.lang.reflect.Method;

/**
 * This class is the basis for the daos. It uses generics to be able to return
 * daos without need for casting. Spring uses it as base to build from when
 * implementing the dao interfaces. It also contains some methods for running
 * the correspondingly named Hibernate query when methods defined in the DAO
 * interface are called.
 *
 *
 * @author jonathan
 *
 * @param <T> The type of the persistent class the dao object should work with.
 */
public class GenericDAO<T> extends HibernateDaoSupport
                           implements IGenericDAO<T>, FinderExecutor {

    private Class<T> type;

    public GenericDAO(Class<T> type){
        this.type = type;
    }

    public void delete(long id) {
        IAbstractBaseObject obj = (IAbstractBaseObject)getSession().get(type, id);
        obj.delete();
        getSession().save(obj);
    }

    public T getById(long id) {
        return (T)getSession().get(type, id);
```

```
AllTests [JUnit] /home/jonathan/bin/jre1.5.0_07/bin/java (Oct 17, 2006 12:05:23 PM)
12:05:25,466  INFO Environment:479 - Hibernate 3.1.3
12:05:25,477  INFO Environment:509 - hibernate.properties not found
12:05:25,481  INFO Environment:525 - using CGLIB reflection optimizer
12:05:25,484  INFO Environment:555 - using JDK 1.4 java.sql.Timestamp handling
12:05:27,741  WARN Configurator:126 - No configuration found. Configuring ehcache from ehcache-failsafe.xml found in the classpath: jar:file:/home/jc
```

Java - GenericDAO.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package Expl... Hierarchy JUnit
net.bioclipse.lis.tests.daos.PlateTypeDAOTe:
Runs: 14/62 Errors: 0 Failures: 0
net.bioclipse.lis.tests.daos.ProjectDAOT
net.bioclipse.lis.tests.daos.UserDAOTe:
net.bioclipse.lis.tests.business.TestAnn
net.bioclipse.lis.tests.pojos.UserTest [R
net.bioclipse.lis.tests.daos.PlateTypeD/
testDelete
testGetAll
testSave
testGetById
Failure Trace
FinderIntroductionInterceptor.java
GenericDAO.java
GenericEnumUserType.java
Outline
net.bioclipse.lis.gene
import declarations
GenericDAO<T> 3 1
type : Class<T>
GenericDAO(Class
delete(long)
getById(long)
save(T)
executeFinder(Me
queryNameFromN
Java Plug-in De... Team Syn... SVN Repo...
Problems Javadoc Declaration Console Progress Tasks

Figure 4: *The development environment Eclipse running JUnit tests.*

3 Software tools and techniques for solutions

In order to complete this sort of program, a number of software tools for system building are useful. A database system is needed that can save and retrieve data and support querying. Although code can be written in any editor, it is nice with a development platform which is able to really help the programmer. Here follows a list of software used or just evaluated during this project. This section contains more details about what tools and techniques where used for the implementation. A big part of this project has been to look for and learn different tools and techniques.

3.1 General solutions

3.1.1 MySQL

MySQL is an open source database management system (DBMS) developed by MySQL AB. *MySQL* is a relational DBMS, that is, it presents the data in relational form as tables, and it provides relational operators to manipulate the data[9, 10].

In this project, *MySQL* was chosen mainly because it is well known, easy to install and well supported. Not many other systems were ever seriously considered.

3.1.2 Eclipse

Eclipse is, in the words of its own webpage,

> "an open source community whose projects are focused on providing a vendor-neutral open development platform and application frameworks for building software. The Eclipse Foundation is a not-for-profit corporation formed to advance the creation, evolution, promotion, and support of the Eclipse Platform and to cultivate both an open source community and an ecosystem of complementary products, capabilities, and services."[11]

Eclipse is not only a development platform. Its community also produces a plugin based platform, the *Eclipse Rich Client Platform*, that can be used to build any kind of program. Since *Eclipse* is open source and there are lots of people writing plugins for it a wide variety of plugin programs exists for it.

There are a few big development platforms for *Java*. Except for *Eclipse*, *Borland*'s *JBuilder* and *Sun*'s *NetBeans* are two that are commonly used. Figure 4 shows *Eclipse* running *JUnit* tests.

3.1.3 Hibernate

Most well-developed DBMSs are relational databases (e.g. *MySQL*, *Oracle*, *IBM DB2*), but many commonly used programming languages are object oriented (e.g. *Java*, *C++* and to different degrees also the scripting languages such as *Perl*, *Python* and *Ruby*). This makes for some problems in mapping from the objects in the programming language to the tables in the relational database[12]. A few of the problems are, in no particular order[13]:

- The relational database has no good way of dealing with type polymorphisms[1]. One of many ways to solve this need to be choosen. Neither of them comes without potential drawbacks.
- Relational databases have bi-directional connections between entities, but objects normally have uni-directional references.
- Two identical objects can exist at the same time in an object oriented program, but two identical rows in a relational table make no sense.
- The sheer amount of work related to implementing and managing this mapping can by itself be a problem.

An object-relational mapping (ORM) system takes care of the mapping between objects and relational tables, and lets the developer spend time developing the program instead of writing standard code for this mapping. Many of the underlying problems are still there, but the amount of work becomes manageable. *Hibernate* is an ORM for *Java*. The programmer writes mapping files in *XML*[2], which *Hibernate* uses to make the connection between the relational database and the programs objects[14].

3.1.4 Hibernate synchronizer

Hibernate synchronizer[15] is a plugin for *Eclipse* that can help during implementation by generating code for DAOs and persistent objects from the *Hibernate* mapping files. *Hibernate synchronizer* was tested in the beginning of the project, but turned out to lack built-in support for *Spring*, and the needed configuration was deemed too complicated to be worth the trouble. Instead, the persistent objects and the DAOs were manually constructed.

3.1.5 Spring

Spring is mostly referred to as a lightweight framework for building *Java* programs. It is a collection of usable tools in many different areas. It contains helpful code when dealing with databases and web applications, but also many other things. It is called a lightweight framework because the developer is not supposed to have to do any heavy rewriting if a decision is made late in the development to use *Spring*. Since *Spring* contains tools for many things, it can be somewhat difficult to quickly get a good overview of its advantages.

Spring provides a container that, when asked for a specific class, instantiates and delivers it according to the instructions declared in *XML*. The instructions include how the different classes fit together, but also more complex things, such as how a special method in one of these classes should be handled as a database transaction. A database transaction is a series of database events where either all of the events or none of the events take place. Transactions are used in order to guarantee that the database is always kept in a consistent state — that a set of changes does not finish halfway.

Spring AOP In this project, *Spring* is mostly used for its *aspect oriented programming* (*AOP*) functionality when dealing with transactions, and because it contains classes which take care of a lot of standard code when dealing with *Hibernate*. *AOP* is a programming mechanism developed to better deal with some situations that *object oriented programming* (*OOP*) does not do well[16]. It does not discard the *OOP* methods, but extends them. The differences between *AOP* and *OOP* can be described this way:

> "While the tendency in *OOP* is to find commonality among classes and push it up in the inheritance tree, *AOP* attempts to realize scattered concerns as first-class elements, and eject them horizontally from the object structure."[16]

In other words, when different parts of a system need to perform the same things, such as transaction management (as is the case in this program), *AOP* enables a way of writing this functionality once, outside of the object structure, and weave it in at every place[17]. In *Spring*, this weaving takes place when *Spring* constructs the object that is to be delivered by the container. See figure 5 for a description of this process for transaction management.

An example of AOP are *introductions* which can be used to add functionality to an object by "implementing" an interface during runtime so that undeclared methods can be redirected to some handler.

[1] *type polymorphism* refers to the way one definition can be used with different types of data in some programming languages. For example, a method could work on chess pieces meaning that both a pawn and a queen would be applicable input for that method.

[2] *XML* is a strict text format designed for easy parsing by computers, while still being humanly readable. It is mainly used for handling data. *XML* is somewhat related to *HTML*, the standard used for webpages.

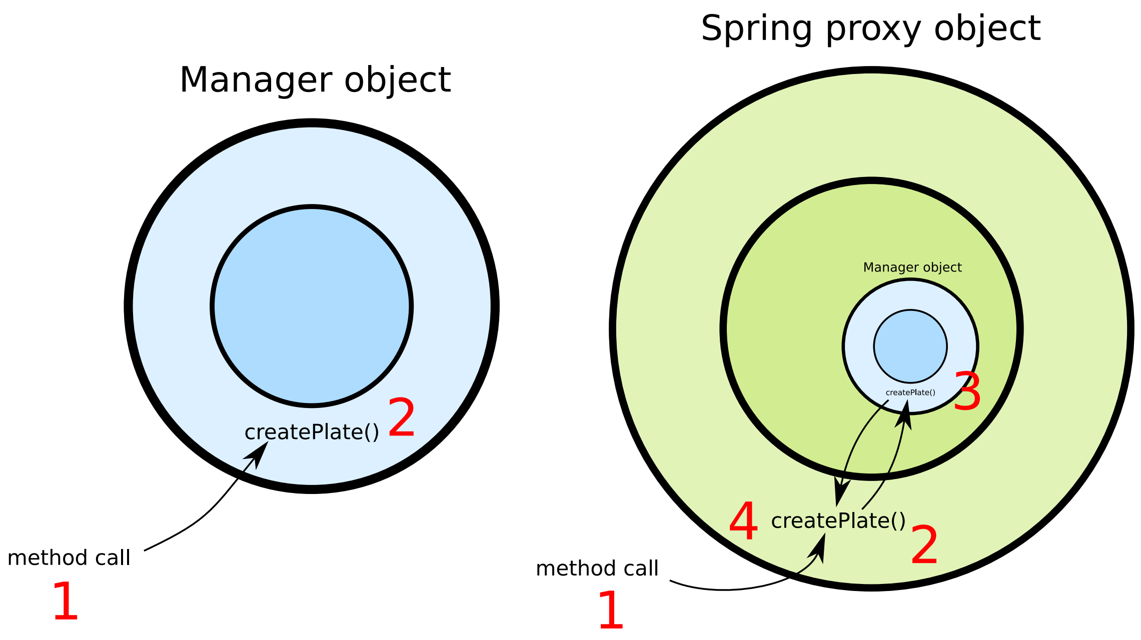


Figure 5: *Transaction management.* ***Left:*** *Without the Spring proxy a call (point 1 in figure) to a method, createPlate, of a manager (2 in figure) works as usual and the createPlate method would have to begin a database transaction, create the plate, audit the change and commit the transaction. Changes to the isolation level of the transaction would have to be performed programmatically in the method of the Manager.* ***Right:*** *With a Spring proxy a call to a method of a manager goes by a method of the Spring proxy which creates the database transaction(at point 2 in figure) and then calls the method (point 3 in figure) which creates the Plate and makes the auditing. Afterwards the method of the Spring proxy gets focus again (point 4) and can commit the transaction. With this system changes of transaction isolation level can be made in the Spring config file where all of the transactions are listed closely toghether and without need to recompile the code.*

The interface is not really implemented during runtime, but a proxy containing the real object implements it and catches the method call. If the real object has the method, it is run, otherwise some other handler can be configured to take over.

3.1.6 JUnit

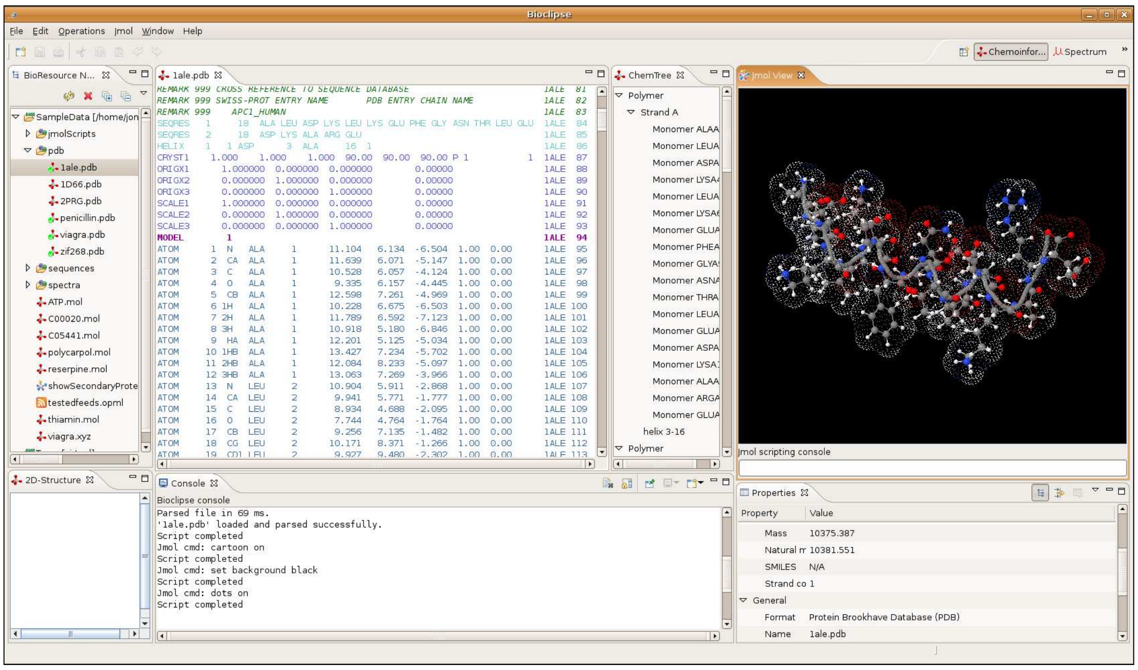
When writing large programs, it is important to be able to test the code before completion. There are always a multitude of minor errors in the code, and a way of finding them without having to run the complete program is needed. This is why programmers write tests. Unit testing means that each code component is tested alone in a controlled environment. *JUnit* is a framework to ease the writing of such unit tests in *Java*. It contains functionality for setting up the test environment, performing the tests and gathering the result.

3.1.7 Bioclipse

The *Bioclipse* project aims at creating a visual platform for chemo- and bioinformatics. Instead of a set of programs with different functionalities (not always speaking the same language) having to be used sequentially in a project, as is often the case today, *Bioclipse* aims at supplying all these functionalities in one application. Figure 6 shows *Bioclipse* running. Bioclipse can handle for example sequence data, structure data and spectrum data. There are many plugin projects under development for *Bioclipse* including such wide spread fields as energy calculations and phylogenetic analyses. *Bioclipse* is implemented using the *Eclipse Rich Client Plattform*.

3.1.8 JEP

JEP - Java Math Expression Parser - is a java library for working with mathematical expressions, supporting definable variables and customised functions[18].



1ale.pdb

```
REMARK 999 CROSS REFERENCE TO SEQUENCE DATABASE                      1ALE  81
REMARK 999 SWISS-PROT ENTRY NAME      PDB ENTRY CHAIN NAME           1ALE  82
REMARK 999    APC1_HUMAN                                             1ALE  83
SEQRES   1     18  ALA LEU ASP LYS LEU LYS GLU PHE GLY ASN THR LEU GLU  1ALE  84
SEQRES   2     18  ASP LYS ALA ARG GLU                                 1ALE  85
HELIX    1   1 ASP      3  ALA     16  1                              1ALE  86
CRYST1    1.000    1.000    1.000  90.00  90.00  90.00 P 1          1 1ALE  87
ORIGX1      1.000000  0.000000  0.000000        0.00000              1ALE  88
ORIGX2      0.000000  1.000000  0.000000        0.00000              1ALE  89
ORIGX3      0.000000  0.000000  1.000000        0.00000              1ALE  90
SCALE1      1.000000  0.000000  0.000000        0.00000              1ALE  91
SCALE2      0.000000  1.000000  0.000000        0.00000              1ALE  92
SCALE3      0.000000  0.000000  1.000000        0.00000              1ALE  93
MODEL        1                                                       1ALE  94
ATOM      1  N   ALA     1      11.104   6.134  -6.504  1.00  0.00    1ALE  95
ATOM      2  CA  ALA     1      11.639   6.071  -5.147  1.00  0.00    1ALE  96
ATOM      3  C   ALA     1      10.528   6.057  -4.124  1.00  0.00    1ALE  97
ATOM      4  O   ALA     1       9.335   6.157  -4.445  1.00  0.00    1ALE  98
ATOM      5  CB  ALA     1      12.598   7.261  -4.969  1.00  0.00    1ALE  99
ATOM      6 1H   ALA     1      10.228   6.675  -6.503  1.00  0.00    1ALE 100
ATOM      7 2H   ALA     1      11.789   6.592  -7.123  1.00  0.00    1ALE 101
ATOM      8 3H   ALA     1      10.918   5.180  -6.846  1.00  0.00    1ALE 102
ATOM      9  HA  ALA     1      12.201   5.125  -5.034  1.00  0.00    1ALE 103
ATOM     10 1HB  ALA     1      13.427   7.234  -5.702  1.00  0.00    1ALE 104
ATOM     11 2HB  ALA     1      12.084   8.233  -5.097  1.00  0.00    1ALE 105
ATOM     12 3HB  ALA     1      13.063   7.269  -3.966  1.00  0.00    1ALE 106
ATOM     13  N   LEU     2      10.904   5.911  -2.868  1.00  0.00    1ALE 107
ATOM     14  CA  LEU     2       9.941   5.771  -1.777  1.00  0.00    1ALE 108
ATOM     15  C   LEU     2       8.934   4.688  -2.095  1.00  0.00    1ALE 109
ATOM     16  O   LEU     2       7.744   4.764  -1.764  1.00  0.00    1ALE 110
ATOM     17  CB  LEU     2       9.256   7.135  -1.482  1.00  0.00    1ALE 111
ATOM     18  CG  LEU     2      10.171   8.371  -1.266  1.00  0.00    1ALE 112
ATOM     19  CD1 LEU     2       9.927   9.480  -2.302  1.00  0.00    1ALE 113
```

Console

Bioclipse console

```
Parsed file in 69 ms.
'1ale.pdb' loaded and parsed successfully.
Script completed
Jmol cmd: cartoon on
Script completed
Jmol cmd: set background black
Script completed
Jmol cmd: dots on
Script completed
```

Properties

Property	Value
Mass	10375.387
Natural m	10381.551
SMILES	N/A
Strand co	1
General	
Format	Protein Brookhave Database (PDB)
Name	1ale.pdb

Figure 6: *Bioclipse displaying a* `pdb` *file.*

3.2 Special solutions

Some of the sought-after functionality for the program needed special solutions.

3.2.1 Auditing

Being able to see who did what and when, is an important feature in this sort of system. A few ideas were considered, such as an extra table for each table containing everything that has happened and populated by triggers in the database, but that solution seemed a little bit more complex than needed. Instead, the system has one table containing string representations of the object after the event, and a timestamp telling when the event occured. It also discriminates between create, delete, and update events.

Annotation type	Description
Text annotation	any free text
Float annotation	a decimal number
Enum annotation	one of a set of predefined text strings

Table 2: *The different annotation types in the program.*

3.2.2 Annotations

The program will contain a system for creating annotations and annotating objects with annotations of the types seen in table 2.

First, an annotation is created setting a name and type. In the case of an enum annotation, a set of allowed values will also be defined in this step. This annotation can then be used to annotate an annotatable object with some value, and it should be possible to search for objects with a given annotation.

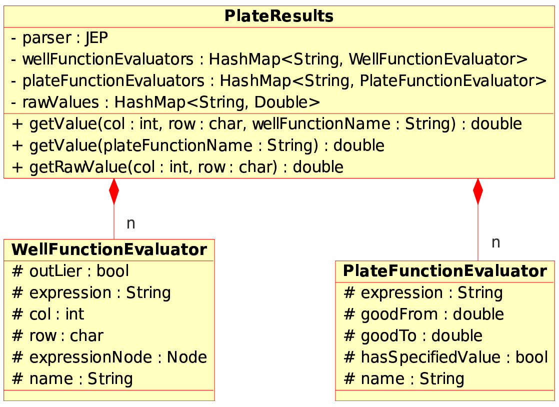

PlateResults
- parser : JEP
- wellFunctionEvaluators : HashMap<String, WellFunctionEvaluator>
- plateFunctionEvaluators : HashMap<String, PlateFunctionEvaluator>
- rawValues : HashMap<String, Double>
+ getValue(col : int, row : char, wellFunctionName : String) : double
+ getValue(plateFunctionName : String) : double
+ getRawValue(col : int, row : char) : double
n
n
WellFunctionEvaluator
# outLier : bool
# expression : String
# col : int
# row : char
# expressionNode : Node
# name : String
PlateFunctionEvaluator
# expression : String
# goodFrom : double
# goodTo : double
# hasSpecifiedValue : bool
# name : String

Figure 7: *The classes used for parsing and evaluating plate functions and well functions.*

3.2.3 Calculations

In order to be able to parse and evaluate the calculation functions *JEP* is used. Since the evaluation of these functions requires data from all around the class diagram, a separate layer of a few classes gathering all these data is first constructed and then the data is parsed from these classes. The classes in this extra layer can be seen in figure 7. There is one *JEP* instance for each plate. It is working with all plate functions and well functions connected to that plate. The parser knows the values of the variables corresponding to the raw value of each well on that plate.

3.3 Persistent objects

The persistent objects are the objects that contain the actual data. They can be stored and loaded from the database using the data access objects. They do not implement any interface of some *Java* framework (such as *Spring*). This sort of objects are commonly called *Plain Old Java Objects* (POJOs).

3.4 Data Access Objects

The data access objects (DAOs) are responsible for storing and loading objects from the database. A number of approaches for writing these were tested during the beginning of this project. The autogenerated DAOs made by *Hibernate Synchronizer* were not used because it was too complicated to get them to work with *Spring* since that would have required rewriting the rules used for the automated code generation. *Spring* has a very good base class that can be extended when writing DAOs. As the next approach tested, this class was extended and a new class was written for each DAO. This included a lot of code duplication, since every DAO looked almost the same but handled objects of different classes. One day, the idea of using the generics concept introduced in *Java 5* to write just one generic DAO class came up. Generics are a way to write a parametrised class that, when instantiated, is given a class to work with. This means that type checking can be performed at compile time instead of at runtime. After some searching on the internet, it turned out that using generics to solve this problem was not a completely new idea. Per Mellqvist had already thought of this, and presented a way of doing it[19] that has been used in this project. The solution uses *Java 5* generics to deliver the correct object without the need for type casting[3], as well as *Spring* AOP introductions to connect methods declared in an interface to *Hibernate* queries, and reduces code duplication.

Creating a new DAO consists of the following steps:

- Write an interface for the class where methods that will refer to a *Hibernate* query begin with a keyword that can be recognized, so the correct *Hibernate* query can be called.
- Write a *Spring* bean that defines how *Spring* should handle the DAO class in *Spring*'s config file.
- Write the *Hibernate* queries used by the DAO in a mapping file.

3.5 Managers

The managers containing the code called by the GUI are implemented with much help from *Spring*. Each manager object is wrapped within a *Spring* proxy — a special *Spring* wrapper object — that handles transaction management. Each method in a manager is handled as one transaction. When a manager is needed, *Spring* delivers one put together as specified in the *Spring* config file. The *Spring* proxy object implements the interface declaring all methods of the manager. It intercepts a method call and delegates it to the method with the same name on the manager object. It can perform operations before and after the method execution, and can thus take care of all the standardised transaction management. This means that the code in the managers does not have to deal with transaction management at all, since it is handled by *Spring*[4].

3.6 Tests

Having a complete test suite comes in very handy when making changes in the code. It is good to see immediately if a change breaks some other functionality somewhere else. This is a thing not always trivial to realise. It is a good practice when implementing a piece of functionality to first write a test, see that it fails, and then implement the functionality and make sure that the test passes. This practice has been the goal during this implementation, but sometimes it can be difficult to write a test for something that is not yet implemented. In these cases, the tests have been written afterwards as a confirmation of the functionality (and as a way to find bugs). It is also nice to use named but unimplemented tests as "to-do lists". A big part of the code written for this project consists of *JUnit* tests. Working with tests gives early feedback and is very satisfying.

[3]*type casting* means transforming a general object into a more specific object assuming that type casting is applicable. If not, the program will throw an exception and if that is not handled the program will crash.

4 Summary

At present, the structure for all the program's parts except the GUI is present. Although the implementation is somewhat of an iterative process, those parts are also close to finished, and the future changes to them are likely to be minor fixes and addition of new functionality. More than 250 tests have been written. They cover such things as creating, storing and retrieving all different persistent objects, creating annotations, checking that auditing only takes place if the event actually occurs and not if something breaks down before it is completed, and much more. Although a lot of tests have been written there probably exists many more not thought about, needed to be written. Time does not admit more for this master project, but the software project will continue.

5 Future

The next step is the implementation of the GUI during continous beta version releases. There are many more instruments at the department that produce data that would be interesting to incorporate into the system in the future. For example, phenotype microarray data describing cellular respiration over time by measuring generated ATP, and data from high-content screening — an automated microscope with image analysis software. There is also an interest for incorporating more advanced analysis tools such as different machine learning approaches, either by using a *Java* library, or by interfacing with *Matlab*.

6 Acknowledgements

First of all, I wish to thank my supervisor Rolf Larsson for the opportunity to perform this project and my technical supervisor Ola Spjuth for his helpful instructions and for his patience with me while I methodically explored every solution, and refused to stop until I had convinced myself that a good solution had been found. I would also like to thank Claes Andersson for his patient help with just about anything in the daily work and Mats Gustafsson for thorough and fruitful discussions about this report. Finally, Carl Mäsak and Daniel Edsgärd; thanks for your detailed examinations of this report.

References

[1] Elin Jonsson, **Application of New Methodology for Preclinical Development of Anticancer Drugs**, Uppsala University 2000, ISBN 91-554-4686-8

[2] L. Rickardson, M. Fryknäs, C Haglund, H. Lövborg, P Nygren, MG Gustafsson, A Isaksson, R. Larsson **Screening of an annotated compound library for drug activity in a resistant myeloma cell line**, Cancer Chemother Pharmacol, 2006

[3] `http://slims.sourceforge.net/`, December 2006

[4] Harrup & Machacek (2005), **Pro Spring**, Apress

[5] `http://uml.sourceforge.net/index.php`, January 2007

[6] `http://en.wikipedia.org/w/index.php?title=Waterfall_model&oldid=98196377`, January 2007
see also
W. W. Royce, **Managing the development of large software systems**, Proceedings of IEEE WESCON, vol. 26, no. August 1970, p. 1-9

[7] `http://en.wikipedia.org/w/index.php?title=Iterative_and_incremental_development&oldid=90269871`, January 2007

[8] M. Marchesi **The new XP**
`http://www.agilexp.org/downloads/TheNewXP.pdf`

[9] `http://www.mysql.com/`, October 2006

[10] `http://en.wikipedia.org/w/index.php?title=Relational_database_management_system&oldid=96972649`, January 2007

[11] `http://www.eclipse.org`, October 2006

[12] Hemrajani (2006) **Agile Java Development with Spring, Hibernate and Eclipse**, Sams Publishing.

[13] T. Neward, **The Vietnam of Computer Science**,
`http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx`, December 2006

[14] Peak & Heudecker (2006), **Hibernate Quickly**, Manning Publications Co

[15] `http://hibernatesynch.sourceforge.net/`, December 2006

[16] T. Elrad, R. E. Filman, A. Bader, **Aspect oriented programming**, Communications of the ACM, October 2001/Vol. 44. No. 10, 29-32

[17] G. Kiczales, J. Irwin, J. Lamping, J.M. Loingtier, C. V. Lopes, C. Maeda, A. Mendhekar, **Aspect-Oriented Programming**, ACM Computing Surveys 28(4es), December 1996

[18] `http://www.singularsys.com/jep/index.html`, December 2006

[19] P. Mellqvist, **Don't repeat the DAO!**,
`http://www-128.ibm.com/developerworks/java/library/j-genericDAO.html`, December 2006

A Appendix – Graphical user interface specification

Software specification
LIS – Graphical User Interface

Jonathan Alvarsson

February 27, 2007

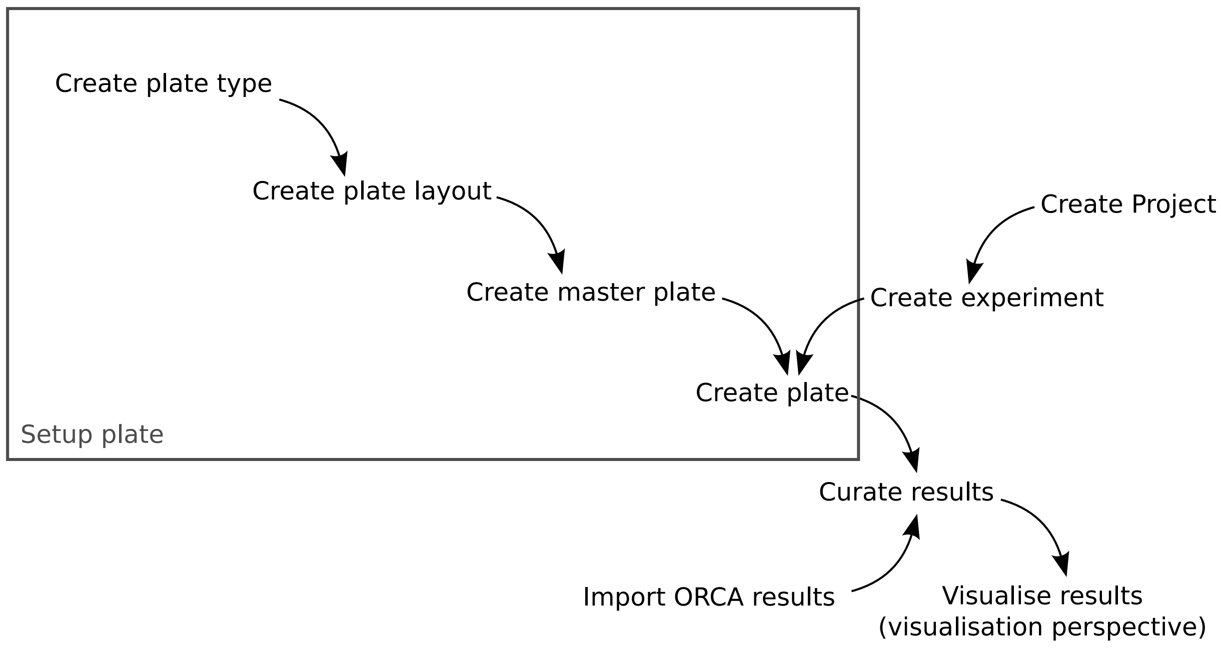
Contents

1 Introduction

Here follows a specification of the functionality of a program meant to be a help in data storing and data browsing for drug research. The GUI pictures do not represent a final version but should be seen as examples representing what should be possible to do.
A few general things that goes for all objects in the system:

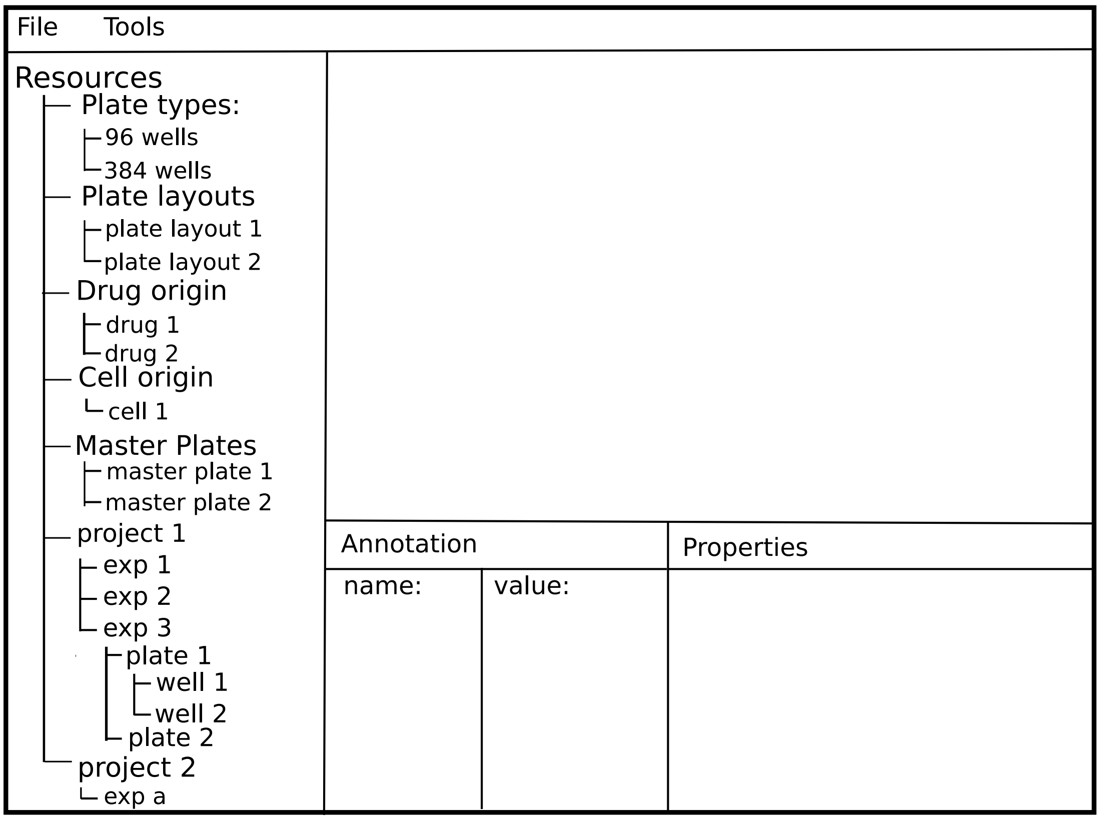
- They can be marked as deleted and thus only be viewed in a special view.
- They can be created by clicking in the menu found by right-clicking in the tree view.
- There is going to be an annotation interface easily accessible from the create dialogs for the annotatable objects and also from the views of those objects.

1.1 Basic workcycle



Normally the first thing created is a plate type defining the size of a plate. From the plate type a plate layout, defining how drugs and controls are placed on the plate, is created. The plate layout is used when creating a master plate that defines which drugs and which concentrations are to be placed in the different wells. Then a plate is constructed based upon a master plate defining which cell line the plate has been seeded with.

2 Setup perspective



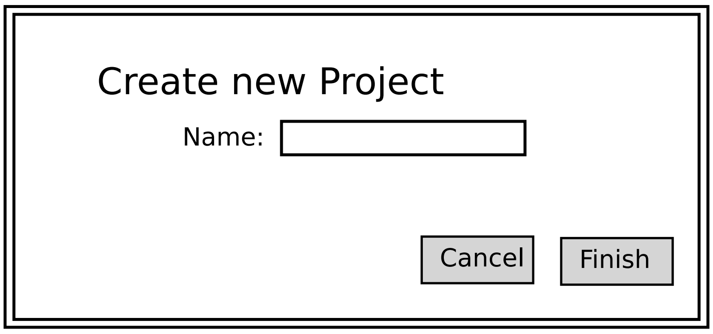
It should be possible to:

- Left-klick objects in the tree-structure and see the relevant information in the views to the right.
- Right-klick in the tree and choose `new` to create new objects.
- Left-click `Tools` → `Import from SD-file` to open the import from SD-file dialog (see: section 4.1).
- Left-click `Tools` → `4 x 96 wells to 1 x 384 wells` to open the dialog for converting from four 96 wells plates to one 384 wells plate (see: section 4.3).

2.1 Project

Definition: A *project* contains *experiments*

2.1.1 Create Project



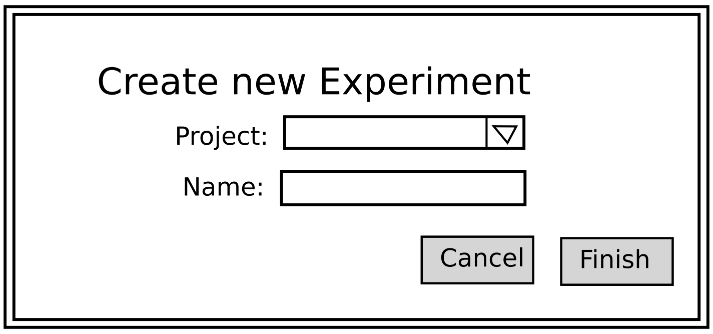
It should be possible to:

- Enter a name for the *project*.
- Click `Finish` and create the new project.
- Click `Cancel` and discard the new project.

2.2 Experiment

Definition: An *experiment* contains *plates*

2.2.1 Create experiment



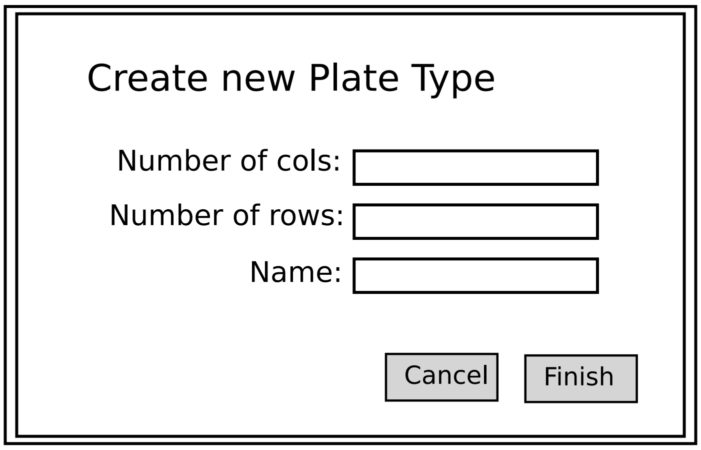
It should be possible to:

- Select a *project* for the *experiment*
- Enter a name for the *experiement*.
- Click `Finish` and create the new experiment.
- Click `Cancel` and discard the new experiment.

2.3 Plate type

Definition: A *plate type* is a base for creating a *plate layout*. It defines the size of a plate.

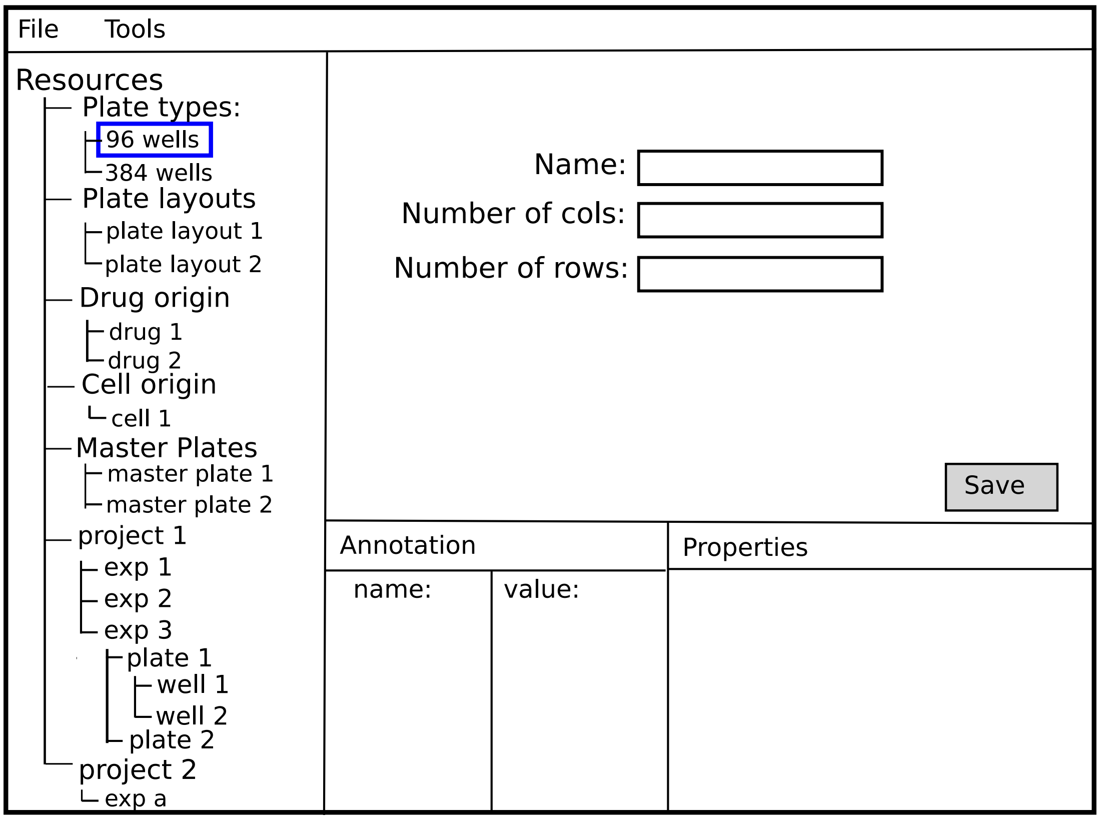
2.3.1 Create plate type



It should be possible to:

- Enter a name and the number of rows and columns (cols) of the *plate type*.
- Click `Finish` and save the new *plate type* to the database.
- Click `Cancel` and discard the changes.

2.3.2 Viewing and editing a plate type



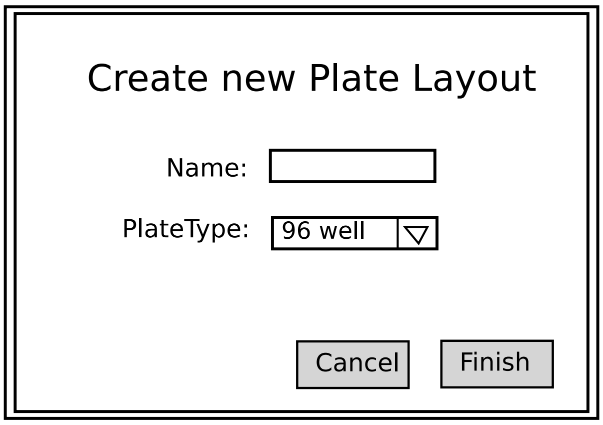
It should be possible to:

- Change the name and the number of rows and columns (cols) of the *plate type*.
- Click `Save` and save the changes to the database.

2.4 Plate layout

Definition: A *plate layout* specifies where on a plate markers, like *drug markers control markers*, are put.

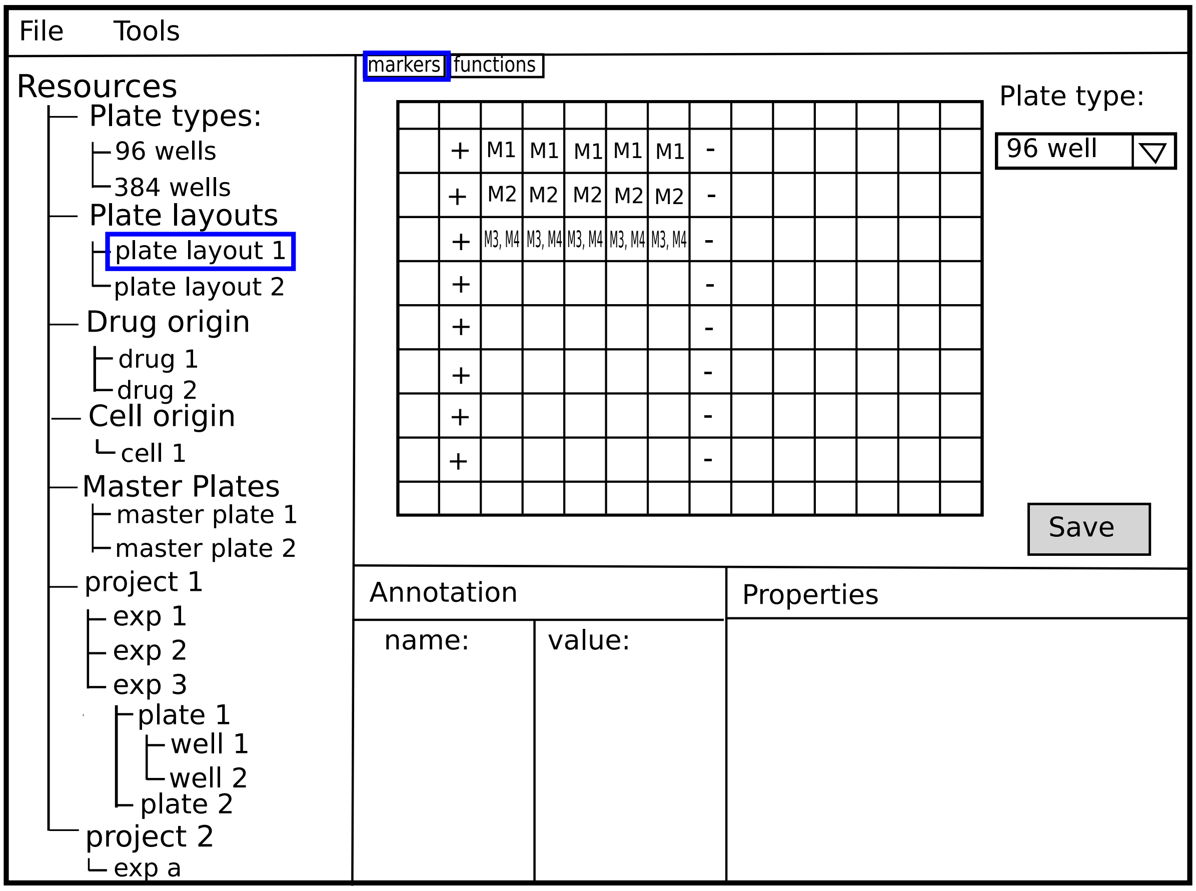
2.4.1 Create Plate Layout



It should be possible to:

- Choose a *plate type* to base the *plate layout* on
- Enter a name
- Click `Finish` and create an empty *plate layout.*
- Click `Cancel` and discard it.

2.4.2 Markers

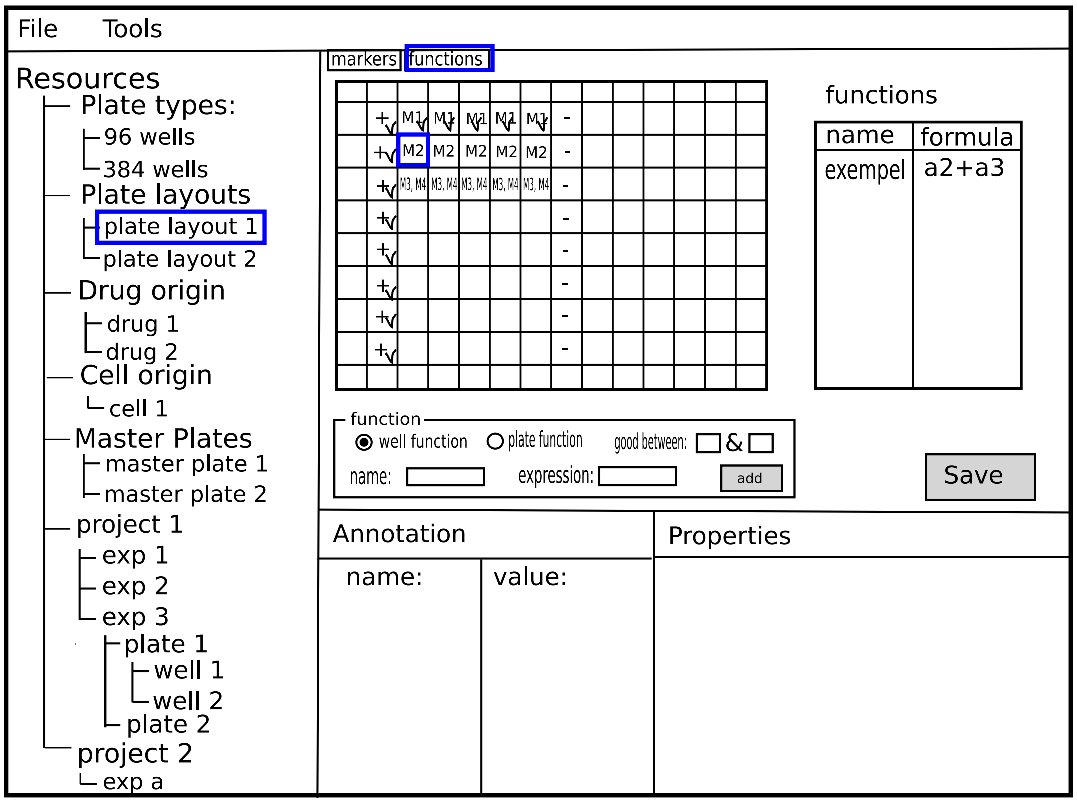


Definition: A *drug marker* labels a well. All wells with the same *drug marker* will contain the same drug. They are named with increasing numbers starting with M1.
Definition: A *control marker* labels a well. All wells with the same *control marker* will contain the same control. The control markers are: blank, control and positive control. It is also possible to leave a well unmarked which symbolises an empty well.
Example: M1 is put on well A1-A5, M2 is put on A6-A10, C+ is put on A11-A12

It should be possible to:

- Right-click and choose what *drug marker* should be on what well.
- Add more than one drug to a well.
- Mark a *well* as blank, control, positive control or empty.
- Click `Save` and save the changes to the database.

2.4.3 Functions



Definition: These are the *function tokens*:

$+$	an addition
$-$	subtraction
$/$	a division
$*$	a multiplication
$avg(a1, a2, a3 \ldots)$	average of wells
$stddev(a1, a2, a3 \ldots)$	standard deviation of wells
(or)	parantheses.

Definition: A *plate function* is a property of a plate with a name, an interval specifying a good value (can be empty) and a mathematical expression, built up of the *function tokens*, which is calculated when the functions is viewed.

Definition: A *well function* is a property of a well with a name and a mathematical expression, built up of the *function tokens*, which is calculated when the functions is viewed.

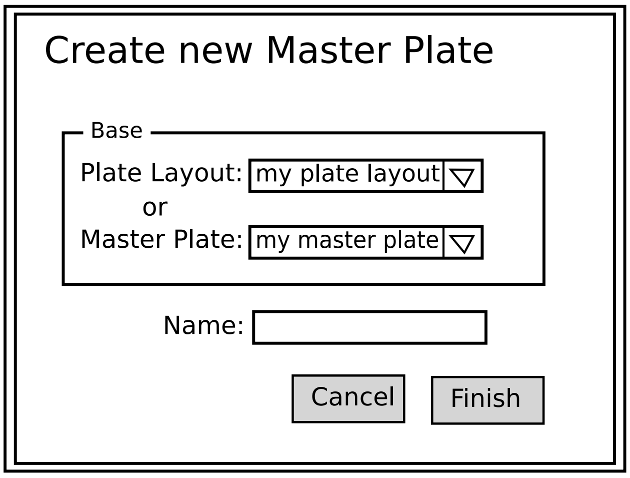
It should be possible to:

- Choose a *well* and write a multitude of *well functions* for that *well*. (the choice "good between" turns inactive when `well function` is chosen)
- Add a multitude of *plate functions* to the *plate*.
- Click `Save` and save the changes to the database.
- See on the graphical plate which *wells* have what markers and which *wells* have what *calculation functions* defined.

2.5 Master plate

Definition: A *master plate* is a *plate* used when creating plates. It specifies what drug every *drug marker* symbolise and the concentration of that drug in each well. A *master plate* can only be changed until a *plate* or another *master plate* has been made based upon it.

2.5.1 Create Master Plate

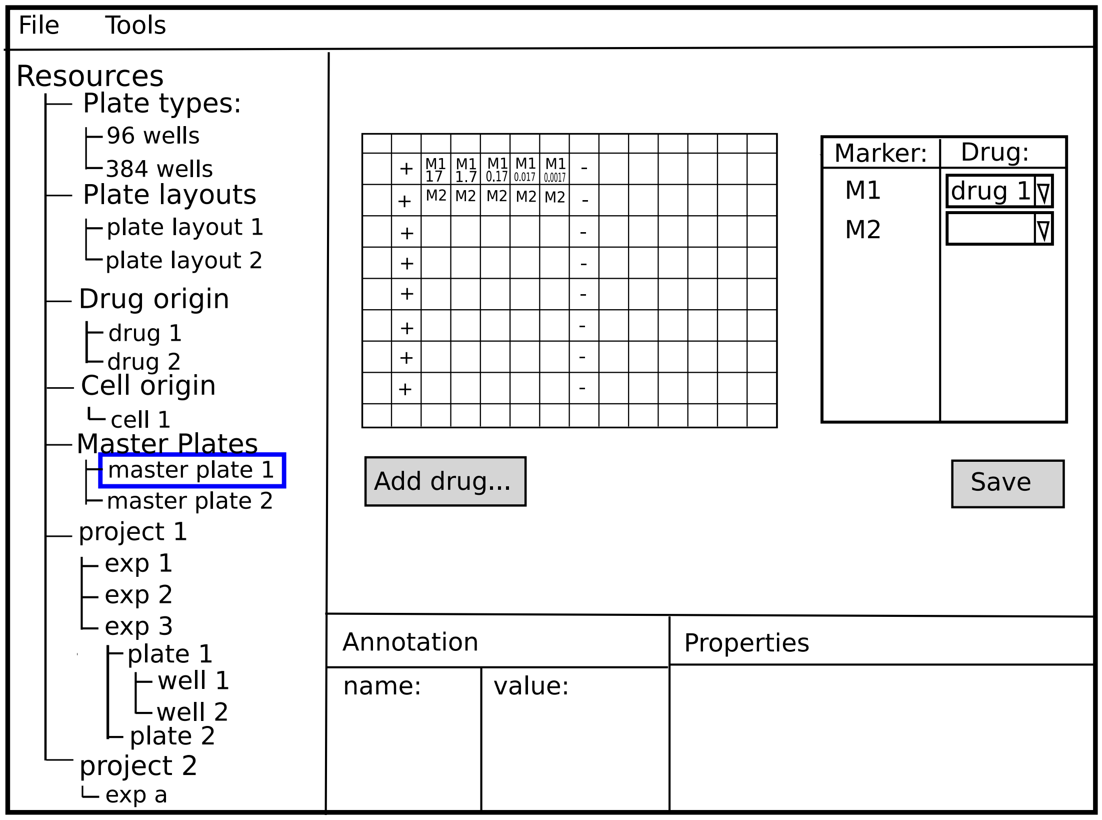


A dialog for creating a new *master plate*.

It should be possible to:

- Choose a *plate layout* or a *master plate* to base the new *master plate* on.
- Enter a name.
- Click `Finish` and save the changes to the database.
- Click `Cancel` and discard the changes.

2.5.2 Viewing and editing a master plate

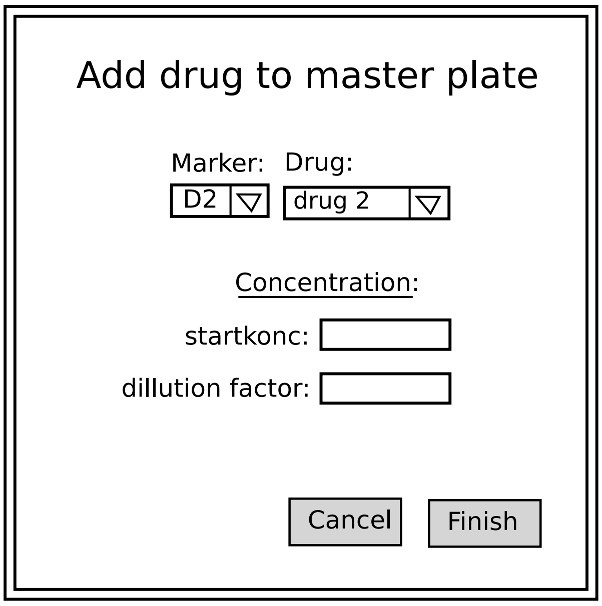


Example: $D1$ =morphine. Well $A1 = 0.5$ nm, Well $A2 = 0.9$ nM

It should be possible to:

- Decide which actual drug the *drug marker* corresponds to.
- Set the concentration by writing it for each drug in each well on the *master plate.*
- Click on `Add drug` to get to a dialog where it is possible to connect drug to *drug marker* by giving *drug marker*, *drug sample origin*, start concentration, and dilution factor.
- Click `Save` and save the changes to the database.

2.5.3 Add drug to master plate



A dialog for adding a drug to a *master plate* by giving start concentration and dilution factor.

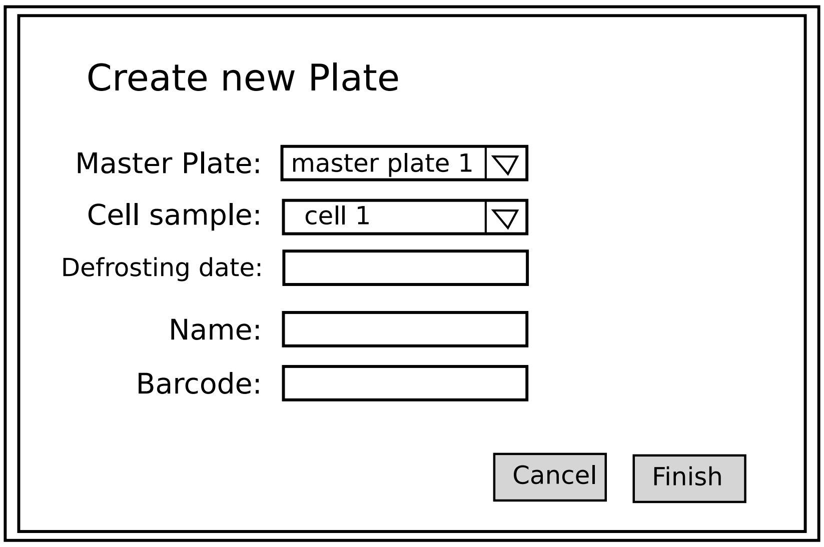
It should be possible to:

- Choose a *drug marker*.
- Choose a *drug sample origin*.
- Give start concentration
- Give dilution factor
- Click `Finish` and save the changes to the database.
- Click `Cancel` and discard the changes.

2.6 Plate

Definition: A *plate* is an instance of a *master plate* with a *cell sample* added.

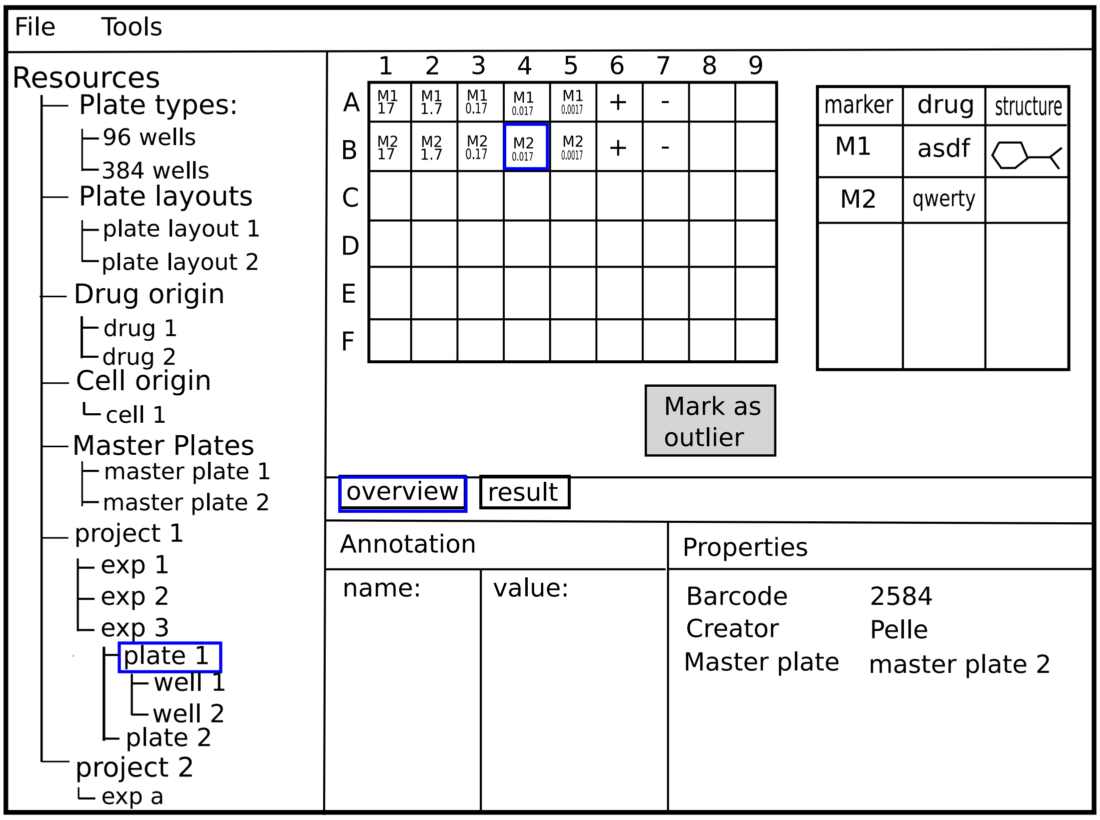
2.6.1 Create plate



It should be possible to:

- Choose an existing *master plate*.
- Choose an existing *cell sample*.
- Enter a defrosting date for the *cell sample*.
- Enter a name for the *plate*.
- Enter the barcode for the *plate*.
- Annotate the *plate*
- Click `Finish`, create a *plate* and write it to the database.
- Click `Cancel` and discard the changes.

2.6.2 Plate overview - a tab in plate view



File Tools

```
Resources
├─ Plate types:
│    ├─96 wells
│    └─384 wells
├─ Plate layouts
│    ├─plate layout 1
│    └─plate layout 2
├─ Drug origin
│    ├─drug 1
│    └─drug 2
├─ Cell origin
│    └─cell 1
├─ Master Plates
│    ├─master plate 1
│    └─master plate 2
├─ project 1
│    ├─exp 1
│    ├─exp 2
│    └─exp 3
│       ├─plate 1
│       │  ├─well 1
│       │  └─well 2
│       └─plate 2
└─ project 2
     └─exp a
```

	1	2	3	4	5	6	7	8	9
A	M1 17	M1 1.7	M1 0.17	M1 0.017	M1 0.0017	+	-		
B	M2 17	M2 1.7	M2 0.17	M2 0.017	M2 0.0017	+	-		
C									
D									
E									
F									

marker	drug	structure
M1	asdf	
M2	qwerty	

Mark as outlier

overview result

Annotation

name:	value:

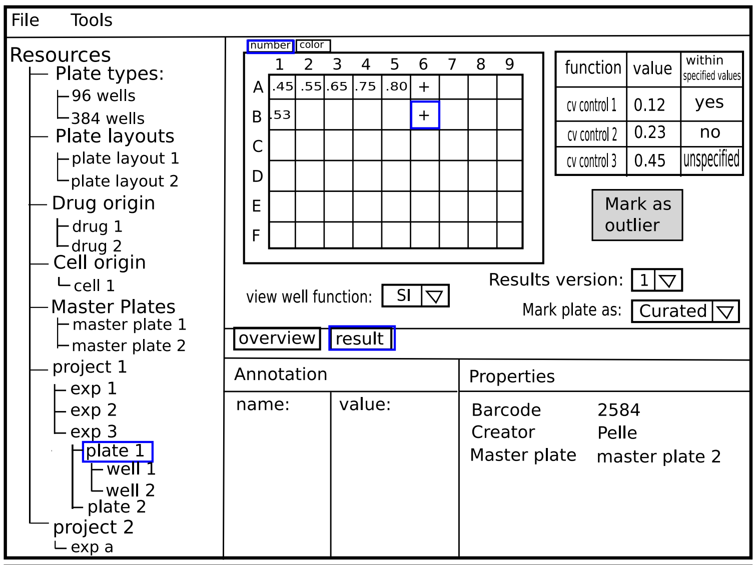
Properties

Barcode	2584
Creator	Pelle
Master plate	master plate 2

It should be possible to:

- See for each well all its samples and their concentration.
- Click on a well and see all its properties in the properties view.
- Mark a well as outlier so that it will not be used in any calculations.

2.6.3 Plate results - a tab in plate view



	1	2	3	4	5	6	7	8	9
A	.45	.55	.65	.75	.80	+			
B	.53					+			
C									
D									
E									
F									

function	value	within specified values
cv control 1	0.12	yes
cv control 2	0.23	no
cv control 3	0.45	unspecified

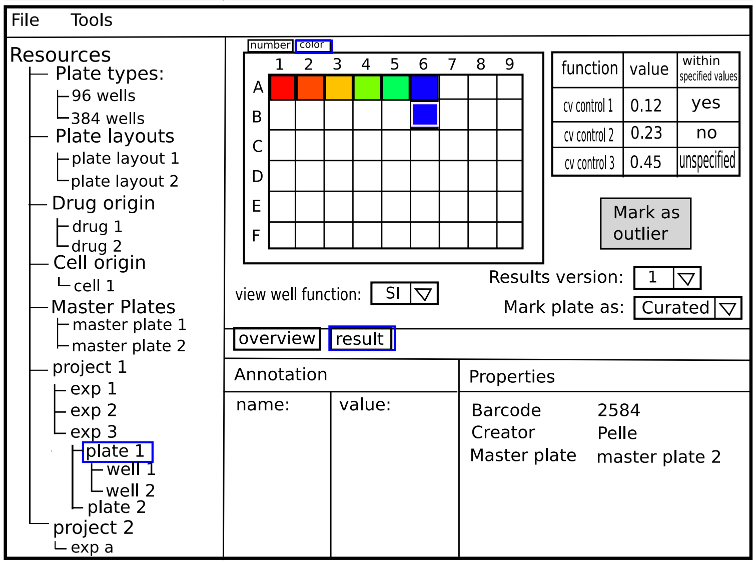
view well function: SI

Results version: 1

Mark plate as: Curated

name:	value:

Barcode	2584
Creator	Pelle
Master plate	master plate 2



function	value	within specified values
cv control 1	0.12	yes
cv control 2	0.23	no
cv control 3	0.45	unspecified

view well function: SI

Results version: 1

Mark plate as: Curated

name:	value:

Barcode	2584
Creator	Pelle
Master plate	master plate 2

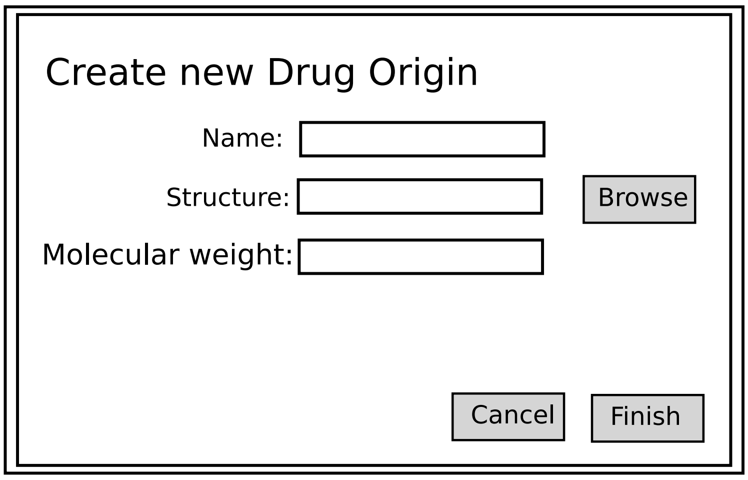
It should be possible to:

- Select which result version to view. Normally one version exists (see section: 4.2).
- Mark *wells* as outliers to keep them out of calculations.
- Change *plate* status from unknown to curated (handled and considered okey) or failed.
- Check the values of a couple of predefined functions for the *plate*. Some which may have predefined good values. Notice that it is possible to mark a plate as curated although these functions have values not within specified values.
- Choose which functions to be viewed on the *wells* (e.g si, pi, raw)
- Choose to see the wellfunctions values either as numbers or as colors.

2.7 Drug origin

Definition: A *drug origin* says something about what kind of a drug a certain drug-sample is.

2.7.1 Create drug origin

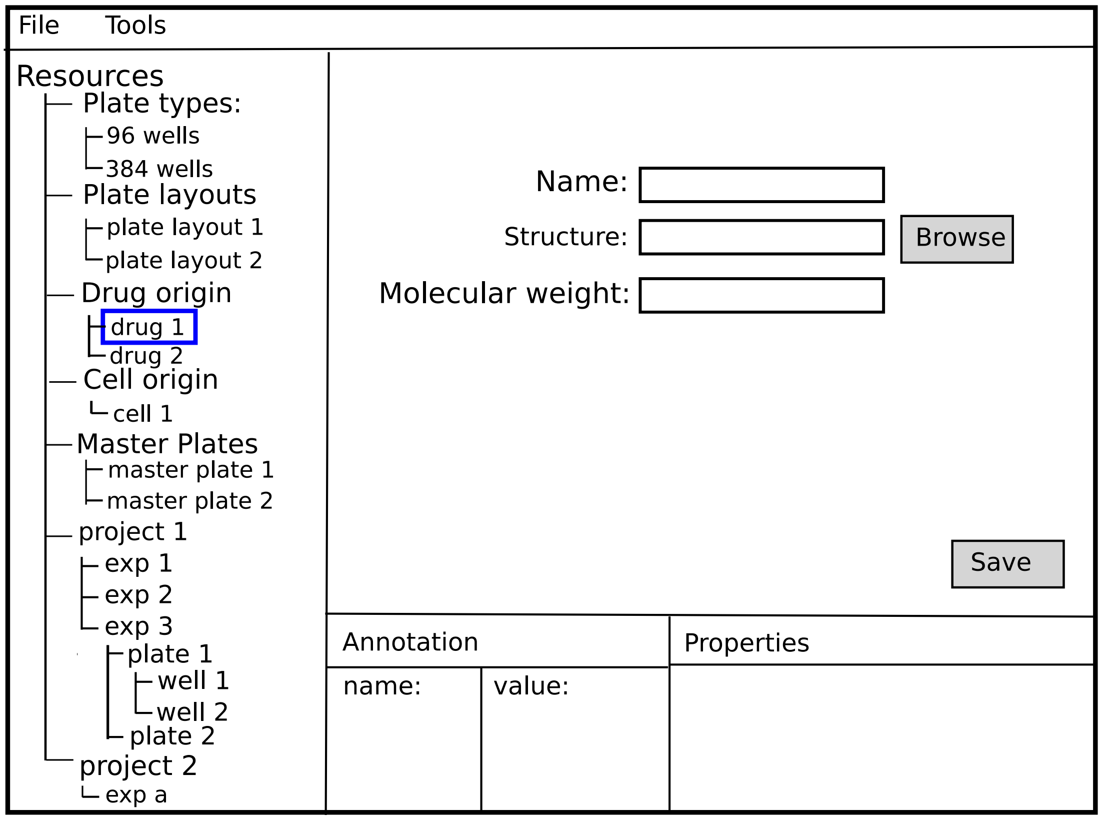


Example: Name=nystatin, structure=nystatin.pdb, molecular weight=926.1

It should be possible to:

- Enter name and molecular weight.
- Choose a file containing the structure.
- Click `Finish`, create a *sample origin* and write it to the database.
- Click `Cancel` and discard the changes.

2.7.2 Viewing and editing a drug origin

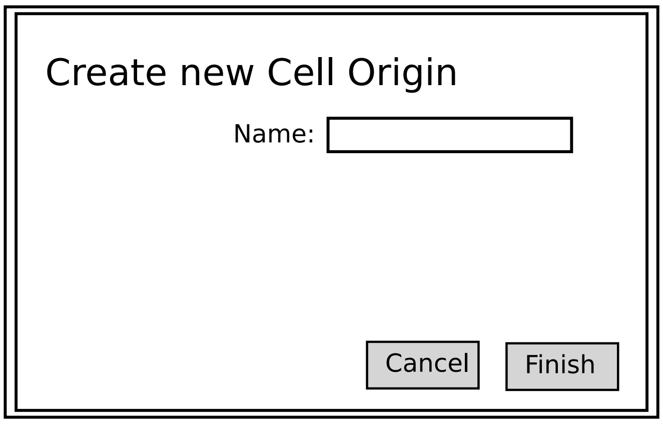


It should be possible to:

- Change name, structure and molecular weight.
- Click `Save` and save the changes to the database.

2.8 Cell origin

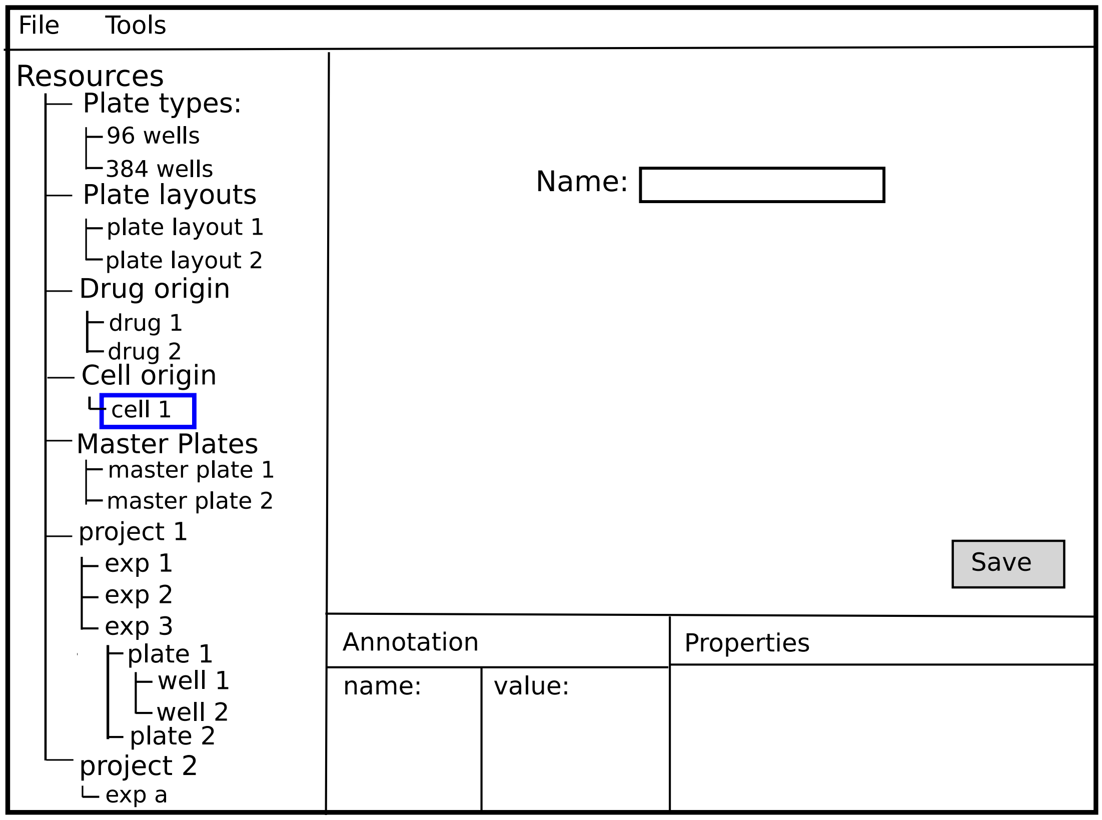
2.8.1 Create cell origin



It should be possible to:

- Enter name and patient sample code.
- Click `Finish`, create a *cell origin* and write it to the database.
- Click `Cancel` and discard the changes.

2.8.2 Viewing and editing a cell origin



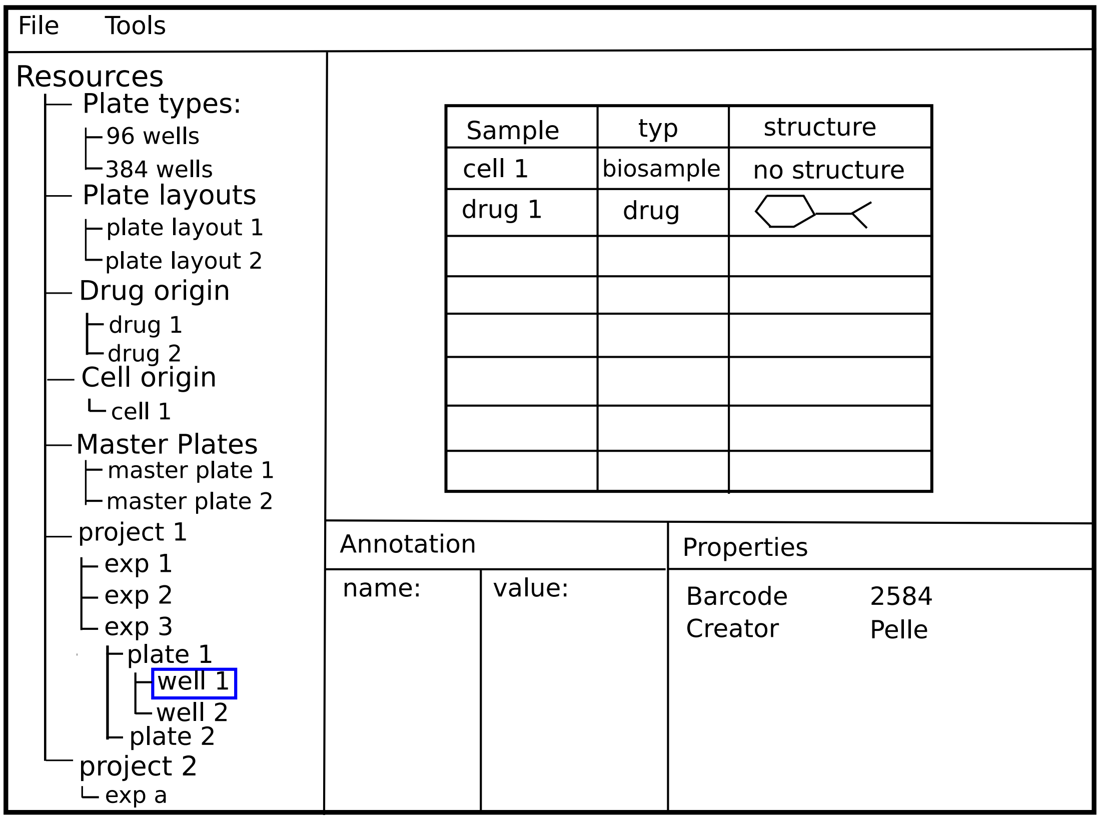
Definition: A *cell origin* says something about what kind of a cell a certain *cell sample* is.
Example: Name=8226s

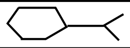
It should be possible to:

- Enter a name.
- Click `Save` and save the changes to the database.

2.9 Well

2.10 Viewing a Well



Sample	typ	structure
cell 1	biosample	no structure
drug 1	drug	

Barcode	2584
Creator	Pelle

It should be possible to:

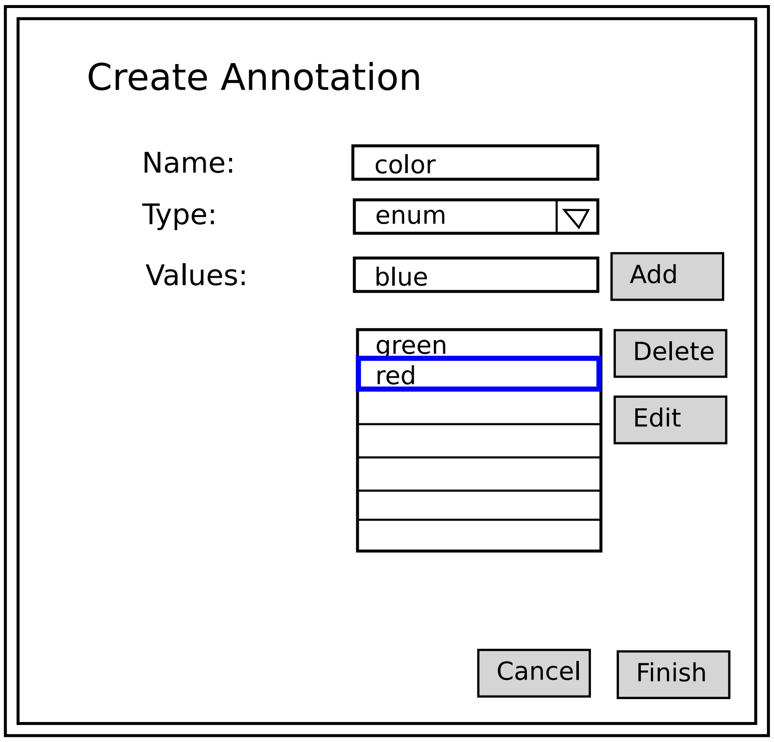
- See all the samples in the well.

This may be redundant with the plate view

2.11 Annotations

Definition: An *annotation* is a way to add new categorisations and properties to the different objects in the system.

2.11.1 Create annotation


Create Annotation

Name: color

Type: enum

Values: blue

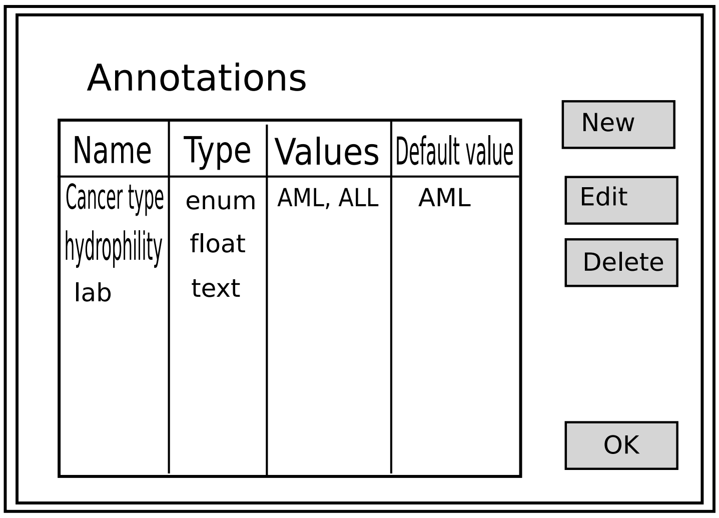
green

red

Example: Name=patient code, Type=text

It should be possible to:

- Enter a name for the *annotation*
- Choose one of these *annotation* types:
  - `text`: a text string.
  - `float`: a float number.
  - `enum`: one out of a set of predefined values.
- If `enum` has been choosen, define a set of values for it.
- Click `Finish`, create an *annotation* and write it to the database.
- Click `Cancel` and discard the changes.



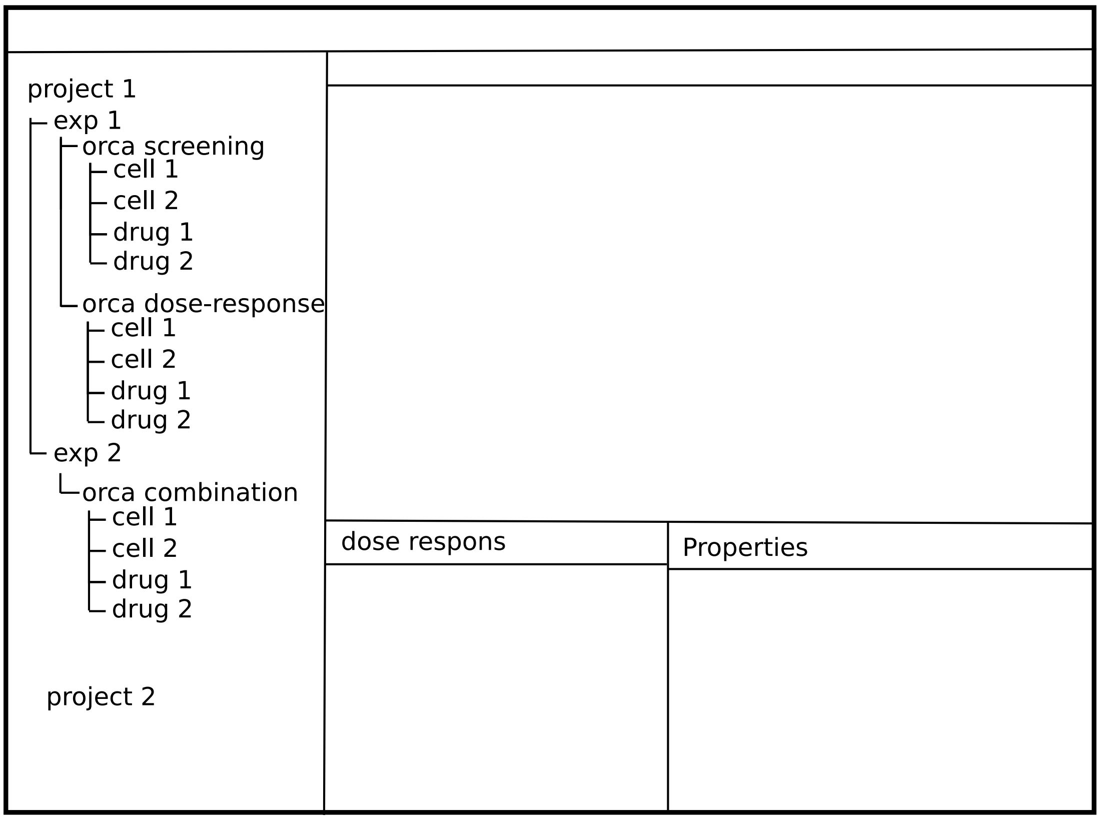
Name	Type	Values	Default value
Cancer type	enum	AML, ALL	AML
hydrophility	float		
lab	text		

Example: Annotate that a sample is from a healty or diagnosed patient to later use screening data for classifying.

It should be possible to:

- See all the *annotations* in the system
- Click `New` and in a dialog create a new *annotation*.
- Click `Edit` and get to edit an *annotation* in a dialog.
- Click `Delete` to delete an *annotation*.

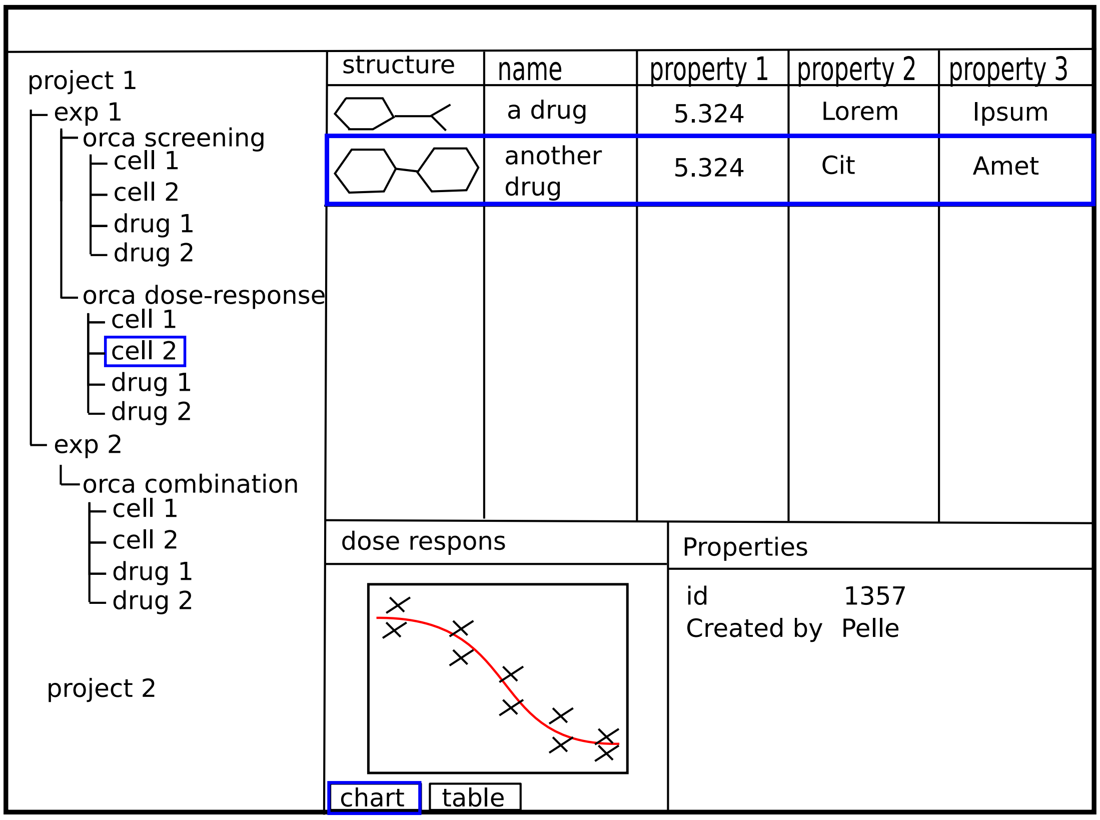
3 Data visualisation perspective



Consider everything in this section very vague and only a presentation of some ideas

It should be possible to:

3.1 Dose response cell



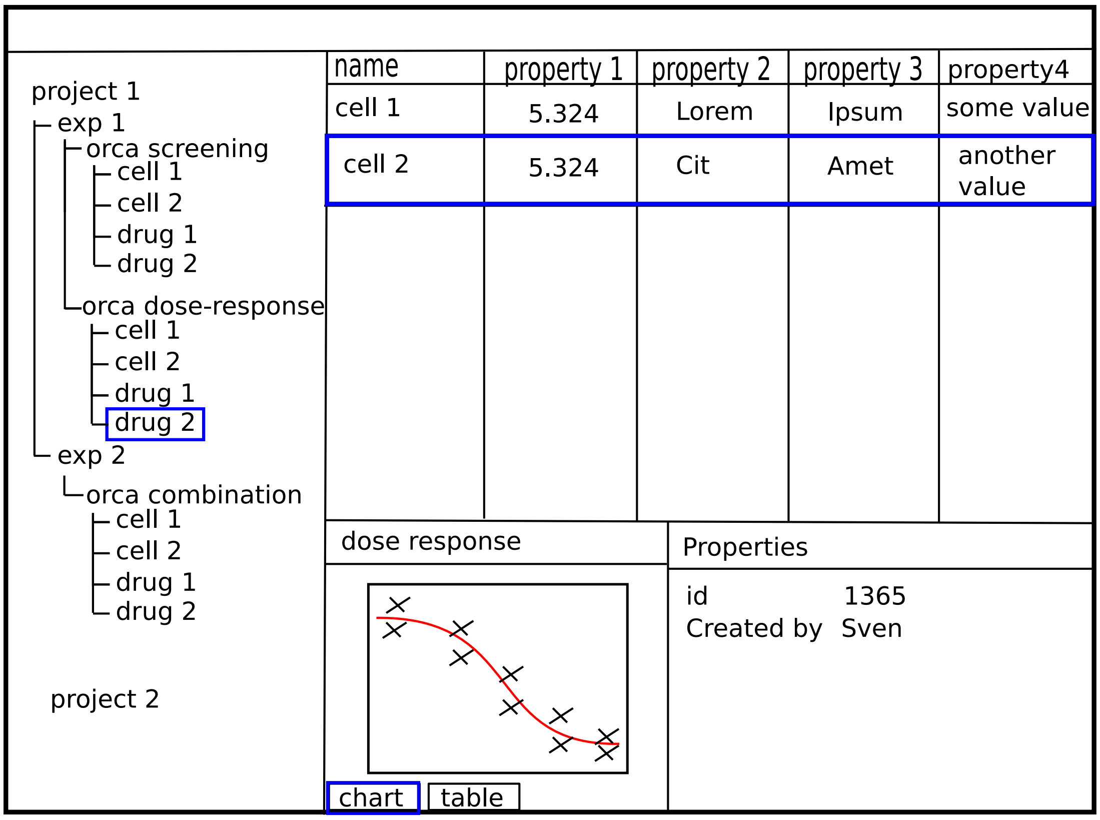
structure	name	property 1	property 2	property 3
	a drug	5.324	Lorem	Ipsum
	another drug	5.324	Cit	Amet

id	1357
Created by	Pelle

It should be possible to:

- See the dose response results browsed first by cell and secondly by drug.

3.2 Dose response drug



```
project 1
├ exp 1
│ ├orca screening
│ │ ├cell 1
│ │ ├cell 2
│ │ ├drug 1
│ │ └drug 2
│ └orca dose-response
│   ├cell 1
│   ├cell 2
│   ├drug 1
│   └drug 2
└ exp 2
  └orca combination
    ├cell 1
    ├cell 2
    ├drug 1
    └drug 2

project 2
```

name	property 1	property 2	property 3	property4
cell 1	5.324	Lorem	Ipsum	some value
cell 2	5.324	Cit	Amet	another value

dose response

chart table

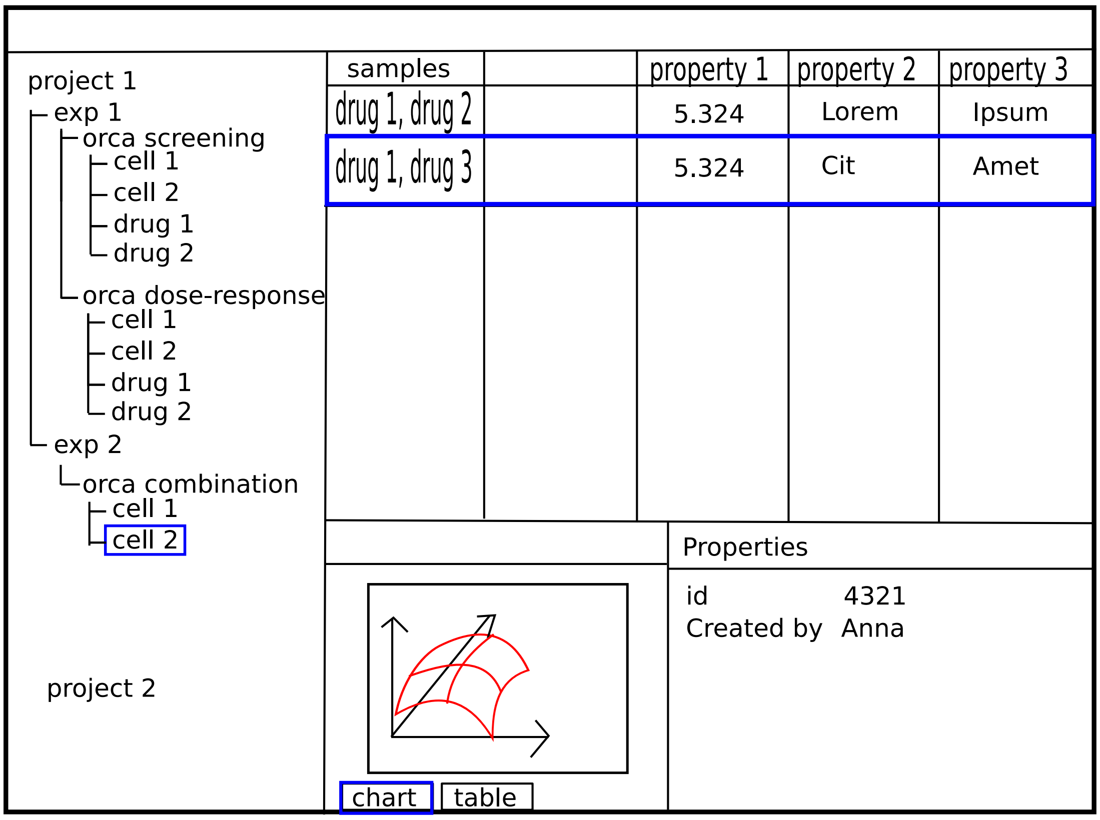
Properties

id	1365
Created by	Sven

It should be possible to:

- See the dose response results browsed first by drug and secondly by cell.

3.3 Combination



samples		property 1	property 2	property 3
drug 1, drug 2		5.324	Lorem	Ipsum
drug 1, drug 3		5.324	Cit	Amet

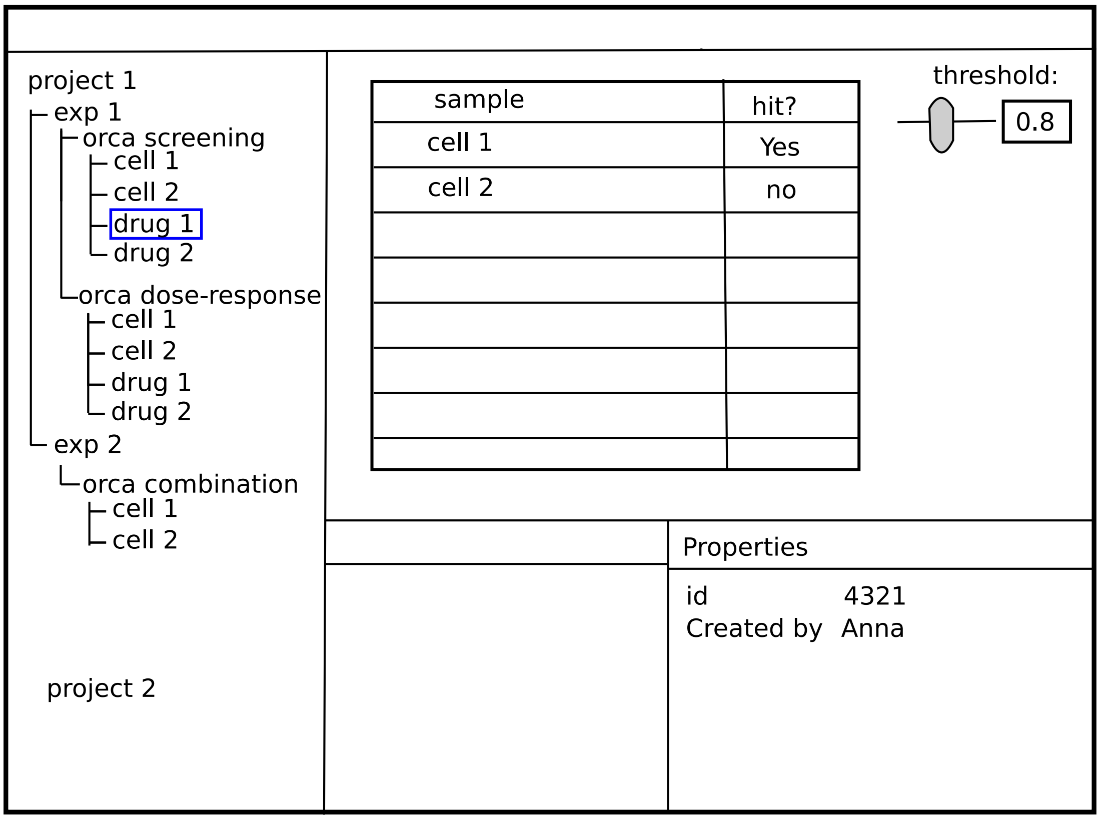
Properties

id	4321
Created by	Anna

It should be possible to:

- Browse the results of an *experiment* with multiple drugs.

3.4 Screening



sample	hit?
cell 1	Yes
cell 2	no

id	4321
Created by	Anna

Is should be possible to:

- Browse the results of a screening *experiment*.
- Change the threshold by dragging the marker left or right or enter a value in the field.

4 Tools

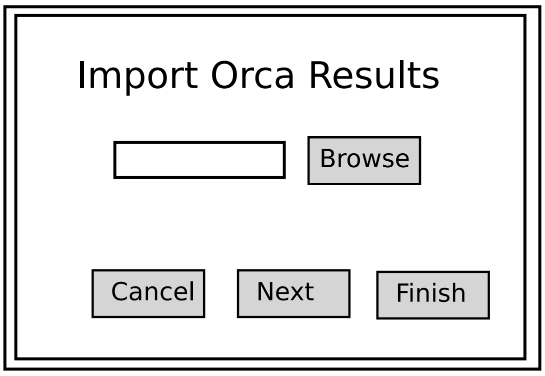
A tool is a general functionality that is not directly associated with a certain view. They are gathered in the tools menu for easy access.

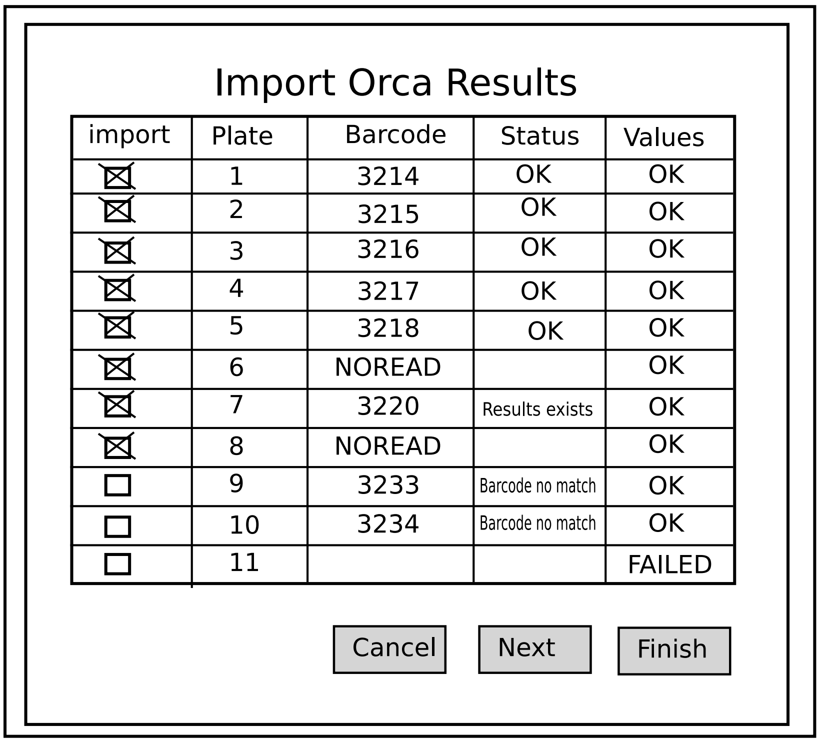
4.1 Import from SD-file

Reachable from the Tools menu the import-from-SD–file command lets the user browse for an SD-file and then imports data from it. Creates one to many new drug origins and possibly new plates etc.

4.2 Import Orca results

Reachable from the Tools menu the import-Orca-result command opens a wizard where the user first browses for a `FluoOptima_384.log` file or if manual reading has been performed (or 96 wells plate) the textversion of the generated *Excel*-file and couples the results in it to plates already in the system by the barcode. (If the barcode reading has not worked or if the reading has been done manually the barcodes have to be entered manually)





Import Orca Results

import	Plate	Barcode	Status	Values
☒	1	3214	OK	OK
☒	2	3215	OK	OK
☒	3	3216	OK	OK
☒	4	3217	OK	OK
☒	5	3218	OK	OK
☒	6	NOREAD		OK
☒	7	3220	Results exists	OK
☒	8	NOREAD		OK
☐	9	3233	Barcode no match	OK
☐	10	3234	Barcode no match	OK
☐	11			FAILED

It should be possible to:

- Select which *plates* to import.
- Results for a *plate* without a matching barcode in the database can not be imported.
- If a *plate* already has registered results: import new results for that plate and mark them with a higher version number. (The old ones will still be in the database)
- If the values for some *plates* can not be parsed (read, e.g. truncated file) the other ones should still be importable.

4.3 Transforming from four 96 wells plates to one 384 wells plate

Reachable from the Tools menu the transform-four-96-wells-to-one-384-wells-plate asks the user for four 96 wells plates and creates one 384 wells plate following the same pattern that the robot uses for such a transformation.

5 Possible features for the future

- A drug palete – something like the color picker in drawing programs with favourites and such things.
- Maybe a possibility to add *cell samples* from more than one cell-sample-origin to the different wells on a *plate*.
- Perhaps an advanced well function that is correcting for systematic errors by some not yet defined algorithm.