Chiasma Technical Description

Document version 2.3

Mats Jonsson, Uppsala University

Björn Karlsson, Uppsala University

Edvard Englund, Uppsala University

[Introduction 3](#_Toc383429773)

[Code Overview 3](#_Toc383429774)

[Namespace Molmed.Chiasma 3](#_Toc383429775)

[Namespace Molmed.Chiasma.Data 3](#_Toc383429776)

[Molmed.Chiasma.Data.Exception 4](#_Toc383429777)

[Namespace Molmed.Chiasma.Database 4](#_Toc383429778)

[Namespace Molmed.Chiasma.IO 5](#_Toc383429779)

[Namespace Molmed.Chiasma.UI 5](#_Toc383429780)

[Molmed.Chiasma.UI.Component 6](#_Toc383429781)

[Molmed.Chiasma.UI.Controller 6](#_Toc383429782)

[Molmed.Chiasma.UI.Dialog 6](#_Toc383429783)

[Molmed.Chiasma.UI.View 6](#_Toc383429784)

[Confusing stuff 1, plate positions 6](#_Toc383429785)

[Confusing stuff 2, Aliquots and Tube Aliquots 6](#_Toc383429786)

[Bar codes 6](#_Toc383429787)

[New event handling between UI and data 7](#_Toc383429788)

[Old event handling between UI and data. 8](#_Toc383429789)

[How to subscribe to data changes in the UI. 8](#_Toc383429790)

[How to publish and fire data event changes 8](#_Toc383429791)

[Lab and office mode 9](#_Toc383429792)

[Transaction handling 9](#_Toc383429793)

[Type of transaction handling 10](#_Toc383429794)

[Transaction example 10](#_Toc383429795)

[Object history information 11](#_Toc383429796)

[Background processing 11](#_Toc383429797)

[Object description 12](#_Toc383429798)

[Organization of container data objects 17](#_Toc383429799)

[Transferring of information to QC databases 18](#_Toc383429800)

[Processing of genotyping results 19](#_Toc383429801)

[Storing allele variation in forward strand notation and in plus strand notation 20](#_Toc383429802)

[Version handling 21](#_Toc383429803)

[Installation of the Chiasma client program 21](#_Toc383429804)

[Testing of the Chiasma client program 21](#_Toc383429805)

[Pooled dilution 21](#_Toc383429806)

[Handling of rounded and excluded volumes 22](#_Toc383429807)

[Relation between key variables 23](#_Toc383429808)

[Calculations for fixed concentration tab 24](#_Toc383429809)

[Calculations for the fixed amount tab 28](#_Toc383429810)

[Max concentration for concentration-locked samples 29](#_Toc383429811)

[Max volume for volume-locked samples 29](#_Toc383429812)

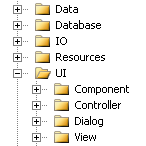
[Appendix A: Document history 29](#_Toc383429813)

# Introduction

Chiasma is a database system for managing laboratory information in the area of genetics. The Chiasma client program, which this document describes, constitutes the main part of the Chiasma system and provides functionality for storing information about samples and results, arrange samples on microtiter plates, print labels and much more.

The program is written in Microsoft C#. Chiasma also uses two web services in order to perform certain tasks which are executed on the server side. The web services run under IIS and communicate with the client program using the HTTP protocol. The first web service is called *ChiasmaResultService* and communicates with instruments in order to retrieve genotype results. The second web service is called *ChiasmaBulkFileService* and writes large result sets in text files on the database server. Both these web services are described in their own technical description documents and therefore not further described here, but they are called from the Chiasma program.

# Code Overview

The code is divided into namespaces. See below for further information about each namespace. The top namespace is *Molmed.Chiasma*. Developers should try to restrict dependencies between namespaces to the order in the following list. Classes in a namespace should only know about themselves and the classes in the namespaces below in the list.

1. *Molmed.Chiasma.UI* (including sub namespaces to UI)
2. *Molmed.Chiasma.IO*
3. *Molmed.Chiasma.Data*
4. *Molmed.Chiasma.Database*
5. *Molmed.Chiasma*

There are several base classes in Chiasma that other classes may or should (differ between base classes) inherit from. Information about some base classes is shown below together with information about the namespace in which the base class belongs.

## Namespace Molmed.Chiasma

*Molmed.Chiasma* contains the *Program* class and configuration information that is used in the whole Chiasma program.

**Base class ChiasmaBase : Object**

Contains general methods that may be of use to any class.

## Namespace Molmed.Chiasma.Data

*Molmed.Chiasma.Data* contains classes whose instances represent the data handled by Chiasma. Most types of data have two associated classes. One class whose instances represents existing data in the database and one class (with static methods) that handles creation of new objects and retrieval of existing objects, e.g. the classes *Species* and *SpeciesManager*. All logic for the data handling should be located in the data classes. All error handling should be done with the exceptions that are defined in *Molmed.Chiasma.Data.Exception*.

**Data cache:**

Some manager classes (e.g. PlateTypeManager) caches data for efficiency reasons. Only data that is small, used a lot and doesn’t change to often should be cached. These Manager classes that caches data should contain a method *public static new void Refresh()* that release old data and read new data from the database. A call to all Manager classes *Refresh* methods are done in *ChiasmaData:Refresh().* This makes it easy to refresh all cached data when needed. *ChiasmaData:Refresh()* is called each time a user log in to Chiasma and each time a user presses the key F5 in a MDI window.

**Base class ChiasmaData : ChiasmaBase**

Contains general data handling methods. All classes that represent data that is stored in the database and all manager classes should inherit from *ChiasmaData*. Many of the methods in *ChiasmaData* are public and static and are used directly by classes in other namespaces. *ChiasmaData* holds a reference to an instance of the *DataServer* class.

Examples of types of methods in *ChiasmaData*:

* Central refresh of cached data
* Check if data is valid
* Comparing data objects for equality
* Retrieve max length of data from the database
* Subscription on data changes
* Transaction handling

### Molmed.Chiasma.Data.Exception

*Molmed.Chiasma.Data.Exception* contains exception classes. All exceptions that explicitly are thrown in the data layer should be defined in this namespace.

**Base class DataException : ApplicationException**

Should be used as super class (directly or indirectly) for all exceptions defined in the namespace. *DataException* can be used directly without defining any sub class.

## Namespace Molmed.Chiasma.Database

All information which Chiasma handles is stored in an SQL Server database. To access this database, Chiasma uses ADO.NET and the SQL Server .NET Data Provider.

The database operations are encapsulated in the *DataServer* class. *DataServer* has functions for connecting to the database, initiating and committing transactions and a load of methods for retrieving and storing different objects in the database. These methods often return an instance of the *DataReader* class. It is the *DataServer* that opens the *DataReader* but it is the method that uses the *DataReader* that must make sure to close the *DataReader*.

Most access with the database is handled with stored procedure. Stored procedures are more efficient than using pure SQL and it works as an abstraction layer which hides table information in the database. Information from the database is loaded with the *SqlDataReader* class. The reason for using *SqlDataReader* is efficiency. *DataSets* are more powerful, flexible and results in better program design but efficiency has been chosen as a more important aspect.

## Namespace Molmed.Chiasma.IO

*Molmed.Chiasma.IO* contains classes that communicate with the outside world (outside of the program) except for the database which has it own namespace. This namespace contains for example a class *ZPLGenerator* used for generating scripts for the label printer, and a class *FileServer* which facilitates common file operations.

## Namespace Molmed.Chiasma.UI

*Molmed.Chiasma.UI* contains the *MainForm* class which is a MDI container class. It has a Login method which either prompts the user with a login dialog and sends the entered information to the LoginDatabase method, or calls the LoginDatabase method with no login information, meaning that the windows account should be used to authenticate the user. Apart from the MDI container, the namespace contains all forms that may be used as MDI child windows.

Sub namespaces to *Molmed.Chiasma.UI* are used to further structure the UI related classes into logical groups. The design pattern MVC (Model – View – Controller) has influenced this namespace structure.

**Base class ChiasmaForm : Form**

May be used as super class to any form and should be used as super class to all MDI children. *ChiasmaForm* catches F5 keystrokes, makes a refresh of cached data in the data layer and calls the virtual method RefreshChiasmaForm. MDI children should override RefreshChiasmaForm and add code for updating all information in the form.

Examples of types of methods in *ChiasmaForm*:

* Copies of some of the methods in *ChiasmaBase* since multiple inheritance is not available in C#.
* Handling of errors (exceptions).
* Methods for showing different kinds of messages to the user.
* Refresh of data and form.
* Show properties of different kind of data.
* Support for handling a *ContextMenuStrip*.

### Molmed.Chiasma.UI.Component

*Molmed.Chiasma.UI.Component* contains classes that have both presentation (View in MVC) and behavior (Controller in MVC). These components are used as building blocks in the other UI related classes.

### Molmed.Chiasma.UI.Controller

*Molmed.Chiasma.UI.Controller* contains classes that only have behavior (Controller in MVC) and no visual presentation. These controllers are used to add behavior to other UI related classes.

### Molmed.Chiasma.UI.Dialog

*Molmed.Chiasma.UI.Dialog* contains forms that only are used as dialogs and never as MDI children.

### Molmed.Chiasma.UI.View

*Molmed.Chiasma.UI.View* contains classes that only have presentation (View in MVC) but no behavior. These views are used as visible building blocks in other UI related classes.

# Confusing stuff 1, plate positions

Sample/aliquot position in plates is stored in database as 1-based indices, table aliquot, tube\_aliquot and sample. In code, there is a mishmash use of 1-based and 0-based indices, namely classes Aliquot, TubeAliquot and Sample use 1-based indices and class Well and its under-classes use 0-based indices. This is also commented in the code in a number of places.

BeadChip and FlowCell positioning are 0-based indexed in database, tables bead\_chip\_well and flow\_cell\_well. Indexing in code is 0-based.

# Confusing stuff 2, Aliquots and Tube Aliquots

Of historical reasons a TubeAliquot represents an aliquot for sequencing, and contains all fields for sequencing information. A TubeAliquot can be placed in a plate.

Aliquot represents an aliquot for genotyping, and can only be placed in a plate. Tubes for genotyping still use the class TubeAliquot (it contains all fields required for genotyping). If necessary in future, Aliquot should be used in tubes as well.

# Bar codes

In the Chiasma system, bar codes can be used to identify objects such as test tubes or plates. The bar codes are number sequences of a certain length which in the database are connected to the objects they should identify.

There are two separate tables for bar codes in the database; one for in-house generated codes and one for externally generated codes. These are combined in one view in the database. The benefit of splitting the codes on two separate tables, is that new codes can be generated using the MAX SQL command to get the highest existing internal code so far (and then only if MAX+1 exists as external code will there be looping to find the next free code). Triggers are utilized to make sure that no external and internal code can be the same, and that an identifiable can only have one bar code (either internal or external).

The Chiasma program is also prepared to use bar codes for triggering actions such as a user login. The latter demands that bar code reading events can be distinguished from normal keystroke events even though the bar code reader provides the same input in the same port as the keyboard. This is solved by measuring the time it takes to input number sequences followed by a carriage return. If a number sequence and return keystroke is received completely within two seconds, it is regarded as a bar code reading. The class *BarCodeController* handles bar code reading and fires the event *BarCodeController:BarCodeReceived(String barCode)* each time it detects a bar code. An instance of *BarCodeController* can be attached to any form. This makes it possible to give different meanings to entered bar codes depending on which form in Chiasma that is currently active.

# New event handling between UI and data

Event handling is about updating one/many instances of an object that are loaded locally in one or more forms when user makes an update (e.g. sample, aliquot, pool or container), and then make changes in the data representation in the User Interface as well. The old implementation, described in the section following this, didn’t worked well for *updating* and *deleting* samples, aliquots and containers (it slowed down the system), although the *create* event for the same objects still works fine. Hence, there are now (unfortunately) two systems for update handling for different kinds of events. Here comes a short description of the different parts of the new event handling system.

**How to include a new dialog into the update handling**

Let the new dialog inherit from one or more of the interfaces *ISampleForm*, *IAliquotForm*, *ITubeAliquotForm*, *IGenericContainerForm*, *IPoolInfoForAliquotsForm*, and implement the (three) functions in it. Take a look in the *ChiasmaForm* class for any suiting interface. **Important:** if the form contains lists that have to be updated, update only the particular rows that needs to be updated, not the entire list (unless virtual lists are implemented). Otherwise the application will be very slow. Also, as a general principle, don’t update more fields than necessary, because the function may be called a number of times in a row.

**ChiasmaEventHandler class**

Events and delegates for updating samples, aliquots, tube\_aliquots, containers and pools are defined in this class. This class is very small with the only purpose of communicating an event from the above five classes to the UI classes.

**ChiasmaData class**

Has a static instance of the ChiasmaEventHandler, called CEventHandler that is called from all classes.

**MainForm class**

Implements the On<object>Update events (e.g. OnSampleUpdate). Loop through all open forms, check if the form is a member of a certain interface and calls the interface update function.

**Raise event**

Are done right after a call to a Database.<update-procedure> function, typically in the Set() function in data-classes.

# Old event handling between UI and data.

Updates of the UI when data changes should, if possible, be handled with events. This is in accordance with the design pattern MVC (Model – View – Controller). There are two problems with this event handling in Chiasma.

* Several types of objects don’t have unique instances, e.g. Sample. This means that we can’t subscribe to changes with the help of an event in an instance of a class.
* We want to update the UI after the transaction has been committed and not when the data is updated.

Classes, methods, structure and patterns have been added to solve these problems.

## How to subscribe to data changes in the UI.

All subscription to changes in data is done with the help of the class *DataEventController*. *DataEventController* has two public constructors:

public DataEventController(Form form,

IDataIdentifier data,

Delegate dataDelegate)

public DataEventController(Form form, Delegate dataDelegate)

The first one is used to subscribe to changes for a data object. The second one is used to subscribe for a specified type of event (e.g. GenericContainerCreatedEventHandler)

Both take a reference to a form and a delegate as argument. The form reference is used to automatically remove the subscription if the form is closed. The delegate reference contains the method that is called when data is changed and it also defines which type of event that the subscription is used for. The first constructor also takes a reference to *IDataIdentifier* that specifies which object that the subscription comprises. The *IDataIdentifier* reference can be null and the method *SetData(IDataIdentifier data)* can be used to change which data instance that is used. If a form needs to end a subscription before the form is closed, call the method *RemoveEventHandler()*.

## How to publish and fire data event changes

Define the delegate.

Example:

public delegate void GenericContainerCreatedEventHandler(GenericContainerList containers);

When the event should be fired call on of the ChiasmaData:FireEvent methods.

Example:

ChiasmaData:FireEvent(typeof(GenericContainerCreatedEventHandler), containers);

# Lab and office mode

Chiasma can be run in two different modes depending on if mostly one person will be using the computer on which the program runs, or many different users will use the same computer. In the first case (office mode), the user is identified by the windows account, whereas in the second case (lab mode), a user logs in to the program by entering a user name and a password. In lab mode, the user is automatically logged out of the program after a certain time of inactivity, but not in office mode. The mode is determined by the *ApplicationMode* setting in the “Chiasma.exe.config” file and can have the value “Lab” or “Office”. If no setting is found, office mode will be used.

Chiasma users have a defined LOGIN in the Chiasma server (EVENTUS) and a corresponding USER for the Chiasma database (GTDB2) in order to connect and to read/write into the database. There are also defined user accounts internally in the database (table authority) for access rights management and to track changes to a specific user. Upon creation, the LOGIN is chosen as either ‘windows authentication’, where the login is authenticated from the windows user account, or ‘SQL Server authentication’, where the user has to enter a username and a password. In order to let a user run Chiasma in lab-mode, it has to be an extra LOGIN that is created with ‘SQL Server authentication’, an extra database USER and an extra user account internally in the Chiasma database for that user. In authority management in Chiasma, there will then be two user account associated with the same name for each of the user that have a lab-user account, although they have different identifiers corresponding to the database LOGINs. The convention for the username in lab mode is first the normal windows user account name (i.e. engled for Edvard Englund) plus the letters ‘lab’, so that the lab user account name for Edvard Englund would be engledlab (without MOLMED\ in the beginning).

# Transaction handling

Transactions should generally be used when data is created, deleted or updated in the database. It is only classes defined in the namespace *Molmed.Chiasma.UI* (and its sub namespaces) that should handle transactions. The reason for this is that it is the user interface that knows what the user is doing.

The UI classes should use the following methods to perform transactions:

* ChiasmaData.BeginTransaction() Start a transaction. Nested transactions are not allowed.
* ChiasmaData.CommitTransaction() Commit a transaction. Committing non-existing transactions will result in an exception.
* ChiasmaData.HasPendingTransaction() Check if we are in a transaction.
* ChiasmaData.RollbackTransaction() Rollback a transaction. If we are not in a transaction nothing is done.

All these methods call corresponding methods in the *DataServer* class that performs the actual work.

The reason for using transaction handling is to avoid inconsistent data in database as a consequence of e.g. an exception during the transaction handling. Without transaction handling there is a risk that only part of a data set would be uploaded if an exception occurred during transaction. Note that the involved database tables are locked for other users (other connections from the same user as well) even if they are only used for reading. This has become a problem when it comes to handling of large infinium data (uploading infinium results, creating infinium internal reports) which may take several days to finish. In these cases, the decreed database transaction handling above has been disabled. As a consequence, the actual database table is still locked during a single transaction, although not all database tables involved in a procedure at the same time. As long as the procedure is not finished, the internal report or result plate is explicitly set as *not finished* (uploading\_flag). If an exception is thrown during transaction, or that the user is pressing cancel, the internal report or result plate is deleted by an explicit call. With this design, key tables like individual or sample is not locked during transaction handling and it’s possible to upload more than one internal report or result plate at the same time.

## Type of transaction handling

ADO.NET transactions and Transact-SQL transactions are not compatible and should never be used together in the same database call. We have decided that the Chiasma program should only use ADO.NET transactions. This decision guaranties that we avoid the multiple transaction type’s problem.

## Transaction example

private void CreateDevice(String deviceIdentifier)

{

try

{

ChiasmaData.BeginTransaction();

DeviceManager.CreateDevice(deviceIdentifier);

ChiasmaData.CommitTransaction();

}

catch (Exception exception)

{

ChiasmaData.RollbackTransaction();

HandleError(“Failed to create device”, exception);

}

}

# Object history information

For most objects in the database, there is a history table corresponding to the actual table. The history tables are composed of the same columns as the actual table, but also have three columns called changed\_date, changed\_authority\_id and changed\_action which store information about the date and time, the user which carried out the changed and the type of change (insert, update or delete). When a row is inserted, updated or deleted, a copy of the row is inserted into the history table. It is thus possible to see the “state” of the object as it was at any one time. Since objects can be deleted from their actual tables, a field in a history table may refer to a row in an actual table which has been deleted. A reference to another object is therefore often displayed in the user interface as a link which the user can click and retrieve the history information for that other object.

Example: A user XYZ is given access rights to the project ABC. The project ABC is then deleted. Now the access history information for user XYZ points to nowhere with the project id of the ABC project. This “pointer” will be displayed in the user interface as a link to a deleted object, and when the user clicks on the link the history of the ABC project will be displayed and reveal that it was the ABC project that the user had been given access rights to.

# Background processing

Since the Chiasma program sometimes processes quite a lot of data, the program needs to perform certain tasks in the background so that the user interface can be kept responsive and the user can be notified about the progress during such tasks. One example of a situation where this is necessary is when the user loads information about markers into the database, because sometimes there can be several hundred thousand markers which should be loaded.

In order to facilitate the implementation of background processing, the Chiasma program code contains a form called *BackgroundWorkerDialog* which in turn contains a BackgroundWorker .NET component. To utilize the BackgroundWorkerDialog, set the *DoWork* event of its BackgroundWorker to call the method which should be executed in the background and call the *Start* method of the BackgroundWorkerDialog. The method which executes in the background can report its progress directly to the BackgroundWorker, and the progress reports are automatically detected by the BackgroundWorkerDialog which displays the progress to the user. Since the BackgroundWorkerDialog form is showed as a dialog, there is no risk that the user can interfere with the background processing by e.g. clicking on buttons in the user interface. The only thing the user can do is click the Cancel button in the BackgroundWorkerDialog which raises a flag on the BackgroundWorker indicating that the user wishes to cancel the operation. The method which performs the background processing should be able to detect this and abort the operation.

Note that an exception which is thrown from the background thread cannot be caught directly in the main thread. Instead, any exceptions are stored in the BackgroundWorkerDialog and the calling methods need to check the BackgroundWorkerDialog’s *ThrownException* property to make sure that no exceptions have been thrown by the background thread.

# Object description

The following table provides descriptions for the objects in Chiasma which (more or less) correspond to real-world objects. Those objects which are only programmatic constructions have been left out.

|  |  |  |
| --- | --- | --- |
| **Object** | **Description** | **Database tables** |
| Aliquot | A diluted or by other means changed version of a master/original sample. There can be many aliquots from one sample, but any aliquot can only originate from one sample. | Stored in the *aliquot* table. History information in *aliquot\_history*. |
| Assay | A combination of primers used to detect the variants of a certain marker. There can be many primers in one assay. The same primers can be part of different assays. An assay can only be for detection of one marker. | Stored in the *assay* table. History information in *assay\_history*. Which primers that belong to the assay is stored in the *primer\_set* table. History information for the latter is stored in *primer\_set\_history*. |
| BeadChip | A chip on which samples (or actually aliquots) are applied for analysis in the Illumina instrument. | Stored in the *bead\_chip* table. History information in *bead\_chip\_history*. |
| BeadChipType | Different layouts of BeadChips, i.e. how samples (aliquots) are arranged on the chip. BeadChipType is also valid to define FlowCells, although the chip\_type field in database has to be set manually to ‘FlowCell’. | Stored in the *bead\_chip\_type\_table*. The names of the different well positions for a BeadChip type is stored in bead*\_chip\_label*. |
| BeadChipWell | The position for one sample (aliquot) on a BeadChip. | Stored in the *bead\_chip\_well* table. History information in *bead\_chip\_well\_history*. |
| FlowCell | A chip with 8 lanes designed to run Solexa samples (Illumna GenomeAnalyser). | Flow\_cell  Flow\_cell\_history |
| FlowCellWell | (i.e. FlowCellLane). Contains information about a specific lane on a FlowCell. The delivery status of a specific lane is stored in flow\_cell\_well\_disc\_  Link. It’s a separate table for this to render possible to have results on more than one hard disc, with different delivery status. | Flow\_cell\_well  Flow\_cell\_well\_history  Flow\_cell\_well\_disc\_link  Flow\_cell\_disc\_project\_link\_  history |
| FlowCellDisc | Represent a hard disc that contains results and that is delivered to customer. A flowCellDisc is associated with one or more projects. | Flow\_cell\_disc  Flow\_cell\_disc\_project\_link |
| Contact | A customer (or possibly other form of person, company or institution). | Stored in the *contact* table. |
| ContactCategory | A collection of contacts. | Stored in the *contact\_category* table. Members for a *contacat\_category* is stored in the *contact\_mapping* table. |
| Device | An instrument or method. | Stored in the *device* table. |
| GenotypingMethod | A method for performing genotyping. This can sometimes correspond to an instrument, but on some instruments more than one method can be used. | Stored in the table *gt\_method*. |
| Individual | A person, animal or plant. There can be many samples from the same individual. An individual can be related to other individuals by having a father and a mother, and it can be set as a control (meaning that its genotype results will be treated separately during quality control procedures). | Stored in the *individual* table. History information in *individual\_history*. |
| InternalReport | The final report for a project issued by a lab technician. The report contains both performance statistics as well as the actual genotype results. | The main table is *internal\_report* and history information for the main entry is stored in *internal\_report\_history*. Quality statistics for each individual included in the internal report is stored in *internal\_report\_indv\_stat*. Quality statistics for each marker included in the internal report is stored in *internal\_report\_marker\_stat*. Genotype results for each individual and marker is stored in *internal\_report\_result* for normal internal reports and in separate text files for bulk internal reports. The result plates (see ResultPlate for a definition) which were included in the internal report are stored in *internal\_report\_result\_plate*. |
| Marker | A position in the DNA which can vary between individuals. | The main entries for markers are stored in the *marker* table, and additional information is stored in the *marker\_details* table. History information for fields from both the main and details tables are stored in *marker\_history*. |
| MasterPlate | A microtiter plate containing wells which in turn contain original samples. | Stored in the *plate* table. History information in *plate\_history*. |
| MasterTube | A test tube containing one original sample. | Stored in the *tube* table. History information in *tube\_history*. |
| Measurement | A measurement such as a concentration determination for a certain sample/aliquot using a certain device. | Stored in the table for the type of object for which the measurement was made, such as in the *sample* or *aliquot* tables. |
| PlateType | The layout of a microtiter plate (such as 12 x 8 wells). | Stored in the *plate\_type* table. The names of the different well positions for a plate type is stored in *plate\_label*. |
| Primer | A short fragment of DNA used in the genotyping process. Many primers can be combined into one assay. The same primer can be part of different assays. | Stored in the *primer* table. Valid types of primers are stored in *primer\_type\_code*. History information is stored in *primer\_history*. Which assays a primer belongs to is stored in *primer\_set*, and history information for that is stored in *primer\_set\_history*. |
| Project | A request to perform genotyping of certain markers on certain samples. | Stored in the *project* table. History information is stored in *project\_history*. |
| ProjectGroup | A collection of projects. | Stored in the *project\_group* table. Members of a group is stored in *project\_mapping*. |
| QualityControlDatabase | The database used for detailed analysis of genotype data in the projects. | Stored in the *qcdb* table. Logs for building up the information in the quality control database is stored in *qcdb\_rebuild\_log* and *qcdb\_rebuild\_batch*. |
| ResultPlate | A collection of genotype results. Sometimes a result plate corresponds to a physical microtiter plate, and sometimes it corresponds to a file with genotype results. | Stored in the *result\_plate* table. History information is stored in *result\_plate\_history*. The actual genotype results are stored in the *genotype* table for normal genotyping methods and in separate text files for bulk genotyping methods. |
| Sample | The master/original sample in untouched form as it arrives to the lab. A sample is taken from one individual. Many aliquots can be generated from one sample. | Stored in the *sample* table. History information in *sample\_history*. |
| SampleSeries | A collection of samples, usually from one customer and sent in one batch. | Stored in the *sample\_series* table. History information in *sample\_series\_history*. |
| SampleSeriesGroup | A collection of sample series. | Stored in the *sample\_series\_group* table. Members of the group are stored in *sample\_series\_mapping*. History information for the latter is stored in *sample\_series\_mapping\_history*. |
| SentrixPlate | A certain type of plate used in the genotyping process for one method on the Illumina instrument. | Stored in the *plate* table. History information in *plate\_history*. |
| Sex | A gender definition. | Stored in the *sex\_code* table. |
| Species | A certain kind of animal or plant. | Stored in the *species* table. |
| State | The form which a sample or aliquot is in. | Stored in the *state* table. |
| TubeRackType | The layout of a tube rack, such as 4 x 6. | Stored in *the tube\_rack\_type* table. The names of the different tube positions for a tube rack type is stored in *tube\_rack\_label*. |
| User | A user of the program. | Stored in the *authority* table. History information is stored in *authority\_history*. |
| UserGroup | A collection of users. | Stored in the *authority\_group* table. Members of the groups are stored in *authority\_mapping*. History information for the latter is stored in *authority\_mapping\_history*. |
| VBPlate | A certain type of plate used in the genotyping process for one method on the Illumina instrument. It has (at present date) the same functionality as the SentrixPlate. | Stored in the *plate* table. History information in *plate\_history*. |
| Well | A position on a microtiter plate. | The plate positions correspond to rows in the *sample* or *aliquot* tables, and the well names are defined in the *plate\_label* table. |
| WorkingPlate | A microtiter plate containing aliquots (not original/master samples). | Stored in the *plate* table. History information in *plate\_history*. |
| WorkingSet | A general collection of objects such as samples or markers. Used for grouping objects together during the quality control procedure. | Stored in the *wset* table. Valid working set types are stored in *wset\_type\_code*. Members of a working set are stored in *wset\_member*, and history information for the latter is stored in *wset\_member\_log*. |

# Organization of container data objects

In the Chiasma program there are several types of data objects which represent real world containers such as rooms and boxes, which can contain most other types of objects, but also microtiter plates and test tubes, which can only contain samples. Since these container types share some properties but can also be quite different from one another, containers are organized in the following manner:

GenericContainer

Container

Room

Box

BeadChip

Plate

Tube

MasterPlate

WorkingPlate

...etc

...etc

MasterTube

FlowCell

The corresponding organization of database tables is the following:

generic\_container

container

bead\_chip

plate

tube

When a container should be created in the database, a new row is first inserted into the generic\_container table where the container gets its identity number, and then inserted into the corresponding sub-table with the identity number it received from the generic\_container table. Hence, no containers will have the same identity number even though they reside in different database tables.

# Transferring of information to QC databases

Chiasma utilizes stored procedures to copy data to the QC databases of the SNP Quality Analysis Tool program, and also to transfer changes back to the original database. There are two types of data transfer actions. The first type updates the QC database of a certain project with new results and some other data on top of the already existing data. This update is controlled by the user from within the Chiasma client program.

Result database

QC database 1

Updating QC database

New data

The second type rebuilds the QC databases from scratch in order to make sure that all data in the QC databases is current. Before the QC databases are rebuilt, any changes to the data are transferred back to the original database. This latter type of data transfer runs as a scheduled job every night on the database server. One reason why it is important to do this complete refresh is that data inserted into the QC databases do not get their unique ID number until they are inserted into the result database.

Result database

QC database 1

New data

Rebuilding QC database

Result database

QC database 1

All data

1.

2.

3.

All QC database transfer tasks are implemented in stored procedures with a name which starts with “p\_QC…”, for example “p\_QCRebuildAll”. The stored procedures which copy data are written so that they first read data from the source database into temporary tables, and then write all data to the target database in a sequence. The reason for this is to keep the writing transaction time down in order to minimize locking.

# Processing of genotyping results

Genotyping data are stored in different stages in the Chiasma database, and in the QC database. At the first step (import result), results from all non-infinium methods are stored in the table genotype, referring to the same strand as the assay, while infinium results are stored in bulk files at the server (EVENTUS), also referring to the same strand as the infinium assay. There is a separate column in the infinium bulkfiles (GenotypesForResultPlate\_xxx.txt) with a T/B designation for the TOP or BOT strand. As genotypes are transferred to the QC-database (non-infinium methods), they are transformed to illuminas TOP strand notation. See external document about the algorithm. Here the flanking sequences are used to determine TOP notation for A/T and G/C SNPs. Flanking sequences are stored in table marker\_details as fiveprime\_flank and threeprime\_flank. Observe that the flanking sequence for a marker may be stored in either the TOP strand or the BOT strand, and is uncorrelated with the referred strand in the assay. In deciding the strand designation for an assay result, the orientation (‘R’ or ‘F’) of the extension primer belonging to the assay is used together with the flanking sequence of the marker. (function fTopPolarizeAlleleResult in the GTDB2 databases).

As it comes to genotyping results for infinium assays, the flanking sequences are not used. The TOP/BOT designation is red directly from the locusByDNA result file and is stored as ‘T’ or ‘B’ in the result bulk file.

Genotyping results in internal reports are stored in illuminas TOP notation (all methods). Non-infinium genotyping results are stored in table internal\_report\_result, and infinium genotyping results are stored in bulkfiles (GenotypesForInternalReport\_xxx.txt).

The difference between raw data results (table genotype and GenotypesForResultPlate\_xxx.txt) and internal report results (table internal\_report\_result and GenotypesForInternalReport\_xxx.txt) are then:

* Internal report results are transferred to illuminas TOP strand reference
* Some of the genotypes in internal reports are rejected (N/A) because of either duplicate errors, inheritance test errors, or manually rejected in SQAT.

# Storing allele variation in forward strand notation and in plus strand notation

The forward strand originally is referring to the sequence presented in dbSNP (it’s defined on one out of two strands). In Chiasma, as well as in Illumina manifest files, forward strand refers to the reference sequence in whatever source database is used (e.g. kgp). The forward strand is problematic since the reference strand in dbSNP for some snps are swopping between versions. When it comes to kgp-snps, which have no versions, some of them are changing the allele variations over time, e.g. from A/G to A/C.

The plus strand refers to the reference genome, i.e. the same physical strand throughout the entire chromosome. The tables for storing plus strand allele-variations are similar to the tables for forward allele variations.

Allele variation in forward strand is stored in three database tables:

*Allele\_variant\_forward*

(Marker\_id, allele\_variant\_id, is\_top\_in\_forward, version)

This table stores the allele variant in forward notation at the latest version. When importing markers from infinium manifest files, there is one allele variant forward for each marker.

*Allele\_variant\_forward\_extra\_version*

(Allele\_variant\_forward\_extra\_version\_id,Marker\_id, allele\_variant\_id, is\_top\_in\_forward, version)

It has been found that allele variation for the same snp from the kgp database is changing between infinium chips, and sometimes the two chip-types are also run in parallel. This table stores cases of different allele variations of the same snp. While the table allele\_variant\_forward typically has millions of entries, this table is for the exceptions. The performance will slow down considerably if this table had as many entries as allele\_variant\_forward. This table is linked together with table problem\_snp\_mapping.

*Allele\_variant\_forward\_old\_version*

(Allele\_variant\_forward\_old\_version\_id,Marker\_id, allele\_variant\_id, is\_top\_in\_forward, version)

This table is for old snp-versions and is never red from the Chiasma client. They are stored in case of back tracing and error-searching

# Version handling

Version number can be manually changed in file AssemblyInfo.cs, object AssemblyVersion.

The version of the Chiasma program is verified against the database in order to prevent users from connecting to the database with an old version of the program. When the program is started, the currently valid version number is retrieved from the database and compared to the version number of the executable file. If the numbers do not match, an error message is displayed and the program immediately closes.

# Installation of the Chiasma client program

In order to install Chiasma, simply execute the bat file “InstallChiasma.bat” on the target machine. All files needed by the installation script must be present in the same folder as the script itself.

# Testing of the Chiasma client program

Automatic testing is used as a complement to the manually testing. Automatic testing is done with the version 2.5.10 of the tool NUnit. Information about NUnit is available on <http://www.nunit.org>. A separate project ChiasmaTest has been created and is used by NUnit to perform the automatic tests. A reference to the NUnit framework has to be added into the C# solution for Chiasma by manually browsing to the nunit.framework.dll file which is located within the NUnit installation folder. The goal is to let NUnit test the interfaces to the data-, database and IO-classes. Manually testing will still be used to test the UI-classes.

# Pooled dilution

A pooled tube is a mix of many samples in one single tube. As the user enters a target volume and concentration (a typical dilution at fixed concentration tab in TubeDiluteDialog), an equal amount (in molar) is taken from each of the checked samples in the list. As a consequence, the partial concentrations from each of the samples are equal as well. Additionally, there is a possibility to lock selected samples to a fix volume or fix concentration. Here are the variables and formulas that are used in pooled tube dilutions.

Variables:

Kmax2 The maximum possible concentration when volume in the tube equals the targeted volume

Kmax The maximum possible concentration at max or min volume. Differ from Kmax2 only when there are samples with fixed volume

Kmin2 The minimal possible concentration when volume in the tube equals the targeted volume

Kmin The minimal concentration at max volume

Kt Targeted concentration

Vt Targeted volume

V\_0 Volume from non-locked samples and relative amount-locked samples

A\_0 Amount (in molar) from non-locked samples and relative amount-locked samples

V\_V Volume from volume-locked samples

A\_V Amount from volume-locked samples

V\_C Volume from concentration-locked samples

A\_C Amount from concentration-locked samples

V\_max Maximum volume, a very large number (10 000 ul)

### Handling of rounded and excluded volumes

Currently there are different implementations for pooling tubes in the TubeDilute and PlateDilute classes, in respect of excluded and rounded volumes. A rounded volume refers to that no volume less than the multiprobe minimum volume is permitted (2 ul). If ‘Volume in destination fixed’ is true (user option), small volumes are rounded up to 2 ul. As a consequence, the partial concentrations for these samples are higher than for other samples. If ‘Volume in destination fixed’ is false, small volumes are rounded up to 2 ul, and volumes from the other samples are rounded up in equal proportion. The resulting partial concentrations are then equal as well.

*Rounded and excluded volumes in TubeDilute*

* Volumes less than 0.05 are rounded to zero. Since they are still counted as a sample (N is not modified), there will be errors because of this.
* Volumes less than the multiprobe minimum volume (2 ul) are marked as gray in the list, and a warning message is shown.

*Rounded and excluded volumes in PlateDilute*

* Calculate sample volumes according to formulas below. If any volume from a sample is less than 0.05 ul, set volume to zero and mark the sample as ‘excluded’. Do the calculations again (from scratch) and do not count the excluded samples.
* If ‘Volume in destination fixed’ is true
  + Calculate sample volumes according to formulas below. Any volume less than 2 ul is rounded up to 2 ul, and the sample is marked as ‘rounded’. As any new rounded sample is encountered, the calculations are starting over from scratch, and any sample already marked as rounded is treated in the same way as it was locked in volume.
* If ‘Volume in destination fixed’ is false (UpscaleSmallVolumes)
  + Calculate sample volumes according to the formulas below. Finish the calculations regardless if any volume is less than 2 ul. Thereafter, determine if there is any small volume, if so, determine a scaling factor. The scaling factor is different for non-locked samples and concentration-locked samples. Depending on if the small volume was from a concentration-locked sample or not, the corresponding formula is used below.
  + If the smallest volume is from a non-locked sample
    - Determine a scale factor for all non-locked samples, and update volumes for non-locked samples
    - Determine total volume V:
    - Update volume for concentration locked samples:
  + If the smallest volume is from a concentration locked sample
    - Determine a scale factor *gamma* for all conc. locked samples, and update the corresponding volumes
    - Modify the total volume so that it’s scaled up with the value *gamma*.
    - Determine a new value of Kmax2 based on the modified target volume. Calculate the rest of the non-locked volumes from the standard formulas below with the new target volume. An alternative would be to scale up non-locked volumes according to the formula

### Relation between key variables

**A\_0, V\_0 and α**

The volume taken from each sample in a pool is by default calculated so that the amount, in ng or pmol, should be equal from each of the samples. The user has also the choice to lock samples to a specific volume, amount, relative amount or concentration in destination. If the relative amount for a sample is specified to x, the amount from that sample is then x times the amount from any non-locked sample.

relative amount from sample *i*

Each sample is initialized with = 1.

A\_0 The total amount from non-locked and relative amount locked samples

V\_0 The total volume from non-locked and relative amount locked samples

Then follows

In the class PlateDilute, rounded samples are excluded from the above formula.

**Volume from concentration locked samples**

V The resulting volume in the pooled tube

k\_Cti = the target partial concentration for concentration-locked samples

k\_Ci = the initial concentration for concentration-locked samples

### Calculations for fixed concentration tab

#### Max and min concentration values

##### Kmax2

is the concentration at the targeted volume when no TE is added ():

k\_i = concentration from non-locked samples

k\_Cti = the target partial concentration for concentration-locked samples

k\_Ci = the initial concentration for concentration-locked samples

N\_NL = number of un-locked samples

*Rounded samples (PlateDiliute)*

In the class PlateDilute, the Kmax2 formula is adjusted for samples marked as rounded

A\_R = Amount from samples marked as rounded

(v\_i is 2 ul in the standard case) In addition, the minimum possible volume for Vt is calculated from volume-locked samples, concentration-locked samples and assuming volumes from non-locked samples are at least 2 ul (different formulas are used whether ‘Volume in destination fixed’ is true or false).

##### Kmin2

is the concentration at the targeted volume when volumes from all non-locked samples are zero (TE is used instead)

##### Kmax

First decide either if max concentration is set at max volume, or if it’s set when volume from non-locked samples are zero.

If , then

max concentration is set at max volume:

If , then

Max concentration is set when volumes from non-locked samples are zero:

*Note*! Current implementation do not have volumes from non-locked samples to zero, instead lowest volume from non-locked are set to 2 ul, (= multiprobe min volume). Equations are not yet documented here.

*Rounded samples (PlateDilute)*

In class PlateDilute, the max concentration is in this case calculated when for the smallest non-locked volume is set to the multiprobe minimum volume (2ul), and all other volumes for non-locked samples are adjusted for equal partial concentrations.

A\_NLMin = minimum amount from non-locked samples

V\_NLMin = minimum volume from non-locked samples

##### Kmin

(volumes for all non-locked are zero, tube is filled with maximum volume of TE)

#### Volume determination for non-locked and concentration-locked samples (fixed concentration tab)

v\_i – volume from non-locked samples

k\_i – initial concentration in non-locked samples

N\_NL – Number of non-locked samples

Case 1:

Non-locked samples:

Concentration-locked samples:

Case 2: , Fixed volume in destination = false

First calculate the resulting volume required for the targeted concentration. Then get the volumes from the samples (v\_i). Modify the targeted volume in destination to the value of V. Below equation is also valid when all samples are locked, at least one volume-locked sample, but in that case all terms involving N\_NL has to be omitted to avoid division with zero.

V – Volume required for the targeted concentration Kt

Non-locked samples:

Concentration-locked samples:

Case 3: , fixed volume in destination = false

Set , and use the above formulas. This case also covers when no locked samples are selected. If so, the volume in the above formula is set to the targeted volume.

Case 4: , fixed volume in destination = false

Set all , calculate the required volume for the targeted concentration. Modify the targeted volume in destination to the calculated V. Amount from TE will come automatically from that.

V – Volume required for the targeted concentration Kt

Concentration-locked samples:

Case 4: , fixed volume in destination = false

Set , and use the preceding procedure.

Case 5: or , and fixed volume in destination = true

Set 2 or , then use the procedure in Case 1.

*Truncated or rounded values (PlateDilute)*

Volumes that are less than 0.05 are truncated to zero. If that happens, all other volumes have to be recalculated. The same holds if any sample volume is less than 2 ul (multiprobe min volume) and ‘Volume in destination fixed’ = true. Therefore, the function calculating the sample volumes are looped until no volumes are truncated or rounded (UpdateDiluteInformationPooled\_Inner() return true or false). In case when ‘Volume in destination fixed’ = false, volumes less than 2 ul (but more than 0.05) is handled in a separate function after the main loop (UpscaleSmallVolumes()).

### Calculations for the fixed amount tab

#### Volume determination for non-locked and concentration-locked samples

This calculation is essentially the same as the ‘normal’ case in the fixed concentration tab calculations.

v\_i – volume from non-locked samples

k\_i – initial concentration in non-locked samples

k\_Cti = the target partial concentration for concentration-locked samples

k\_Ci = the initial concentration for concentration-locked samples N\_NL – Number of non-locked samples

A\_t – targeted amount given by user

V\_t – the resulting volume in destination (targeted volume in fixed concentration, notion is copied, but user don’t choose destination volume in this tab)

Non-locked samples:

Concentration-locked samples:

Derived notations:

V\_t has to be determined when there are concentration-locked samples

#### V\_t determination when at least one non-locked and non-rounded sample

#### V\_t determination when all samples are locked, and at least one concentration locked sample

### Max concentration for concentration-locked samples

When user select samples and is about to enter a fixed concentration, there is a maximum possible concentration for the selected samples.

N\_C – Number of already concentration-locked samples. If an already concentration-locked sample is marked, it is regarded as released.

M – Number of selected samples that are about to be concentration-locked

Kmax\_C - The maximum concentration for the marked samples

### Max volume for volume-locked samples

When user select samples and is about to enter a fixed volume, there is a maximum possible volume for the selected samples.

V\_V – Volume of already volume-locked samples

V\_M – Max volume for the selected samples

### Making pools from existing pools

In cases when existing pools are the constituent parts when creating a pool, it’s desirable that the amount from each of the sample in the existing pool is equal with the amount from other samples added to the pool. Therefore, the default relative part for the pool is set to the number of samples in the pool.

= Number of samples in pool

# Appendix A: Document history

|  |  |  |
| --- | --- | --- |
| **Version** | **Comment** | **Date** |
| 1.0 | Created | 2005-01-24 |
| 1.1 | Updated when the mode setting was moved to the configuration file. | 2005-06-17 |
| 1.2 | Updated when the remote service for reading genotype results was introduced. | 2006-03-09 |
| 1.3 | Visual Basic is no longer used and namespaces are described instead of dll’s. | 2006-08-10 |
| 1.4 | Sub namespaces to Molmed.Chiasma.UI added. | 2006-09-20 |
| 1.5 | Minor changes about automatic testing. | 2006-12-04 |
| 1.6 | Information added about DataReader handling. | 2007-02-20 |
| 1.7 | More information about namespaces. Transaction, data cache and event handling added. | 2007-06-12 |
| 1.8 | Added a paragraph about historical information for objects. | 2007-09-28 |
| 1.9 | Added information about transferring of data to QC databases and did some other minor updates. | 2007-12-18 |
| 2.0 | Added information about the new web service ChiasmaBulkFileService. | 2008-12-10 |
| 2.1 | Revised a few paragraphs. Added information about background processing and organization of container objects. Also added the object description table. | 2009-02-12 |
| 2.2 | Added information about FlowCells | 2009-10-12 |
| 2.3 | Added information for pooled tube dilutions | 2011-05-13 |