# Clinical Decision Support for OpenMRS

Bierman, Robert, *Group Lead*
bierman@mail.sfsu.edu

Woeltjen, Victor, *Group Lead*
woeltjen@mail.sfsu.edu

Choi, Kay  Gimeno, Steven  Lum, Jason  Ng, Ying Kit
Uy, Bianca

May 7, 2013

*Group 1:* Final Project for CSC 668-868 Spring 2013
https://code.google.com/p/sp2013-csc668-868-group1/

# Contents

# 1 Contributions

## 1.1 Contributions by Robert Bierman

## 1.2 Contributions by Victor Woeltjen

Implemented DSS Rule Service, including rule storage and conversion to and from XML (Extensible Markup Language). Responsible for implementation of flow control, execution context, and integration of value types into DSS Interpreter. Authored sections 6 and 8 of this document, except for subsections otherwise noted.

## 1.3 Contributions by Kay Choi

## 1.4 Contributions by Steven Gimeno

## 1.5 Contributions by Jason Lum

## 1.6 Contributions by Ying Kit Ng

## 1.7 Contributions by Bianca Uy

# 2 Introduction

## 2.1 Problem statement

OpenMRS Clinical Decision Support System It is one thing to have information readily available, it is another to understand and make the best use of that information. The OpenMRS system provides a repository of data on patients but it is still up to the physician to decide the course of treatment, tests that need to be run, and medications to be administered. The Decision Support System (DSS) is designed to assist the physician by providing alerts based on correlation of data and programmatic rules. Because of the vast amounts of data on patients, the number of medications and tests available and the variability of patient behavior, the DSS is designed to provide rules to avoid mistakes, speed patient care and optimize resources.

DSS allows doctors to create rules to alert them if they prescribe a medication that may interact with other medications that the patient is taking or may be allergic to. Or, it can suggest running tests that may be due or alert to the fact that prior tests need to be redone. Decision support is about correlating the data is a manner useful to the physician.

To create a DSS, a method must exist to specify these rules, and should be of a nature to allow non-technical individuals to create them. Once created a system to store and interpret those rules needs to be devised and finally the results of the rules need to be displayed back to the physician in an intuitive and meaningful way.

The task here is given the presented language grammar (see appendix A), create the interpreter that can store and process rules and display the results on the patient summary and dashboards in OpenMRS.

## 2.2 Software and platform used

Most of the programming was done in Java, with client pages in Javascript JSP and HTML, and extensions declared in XML.

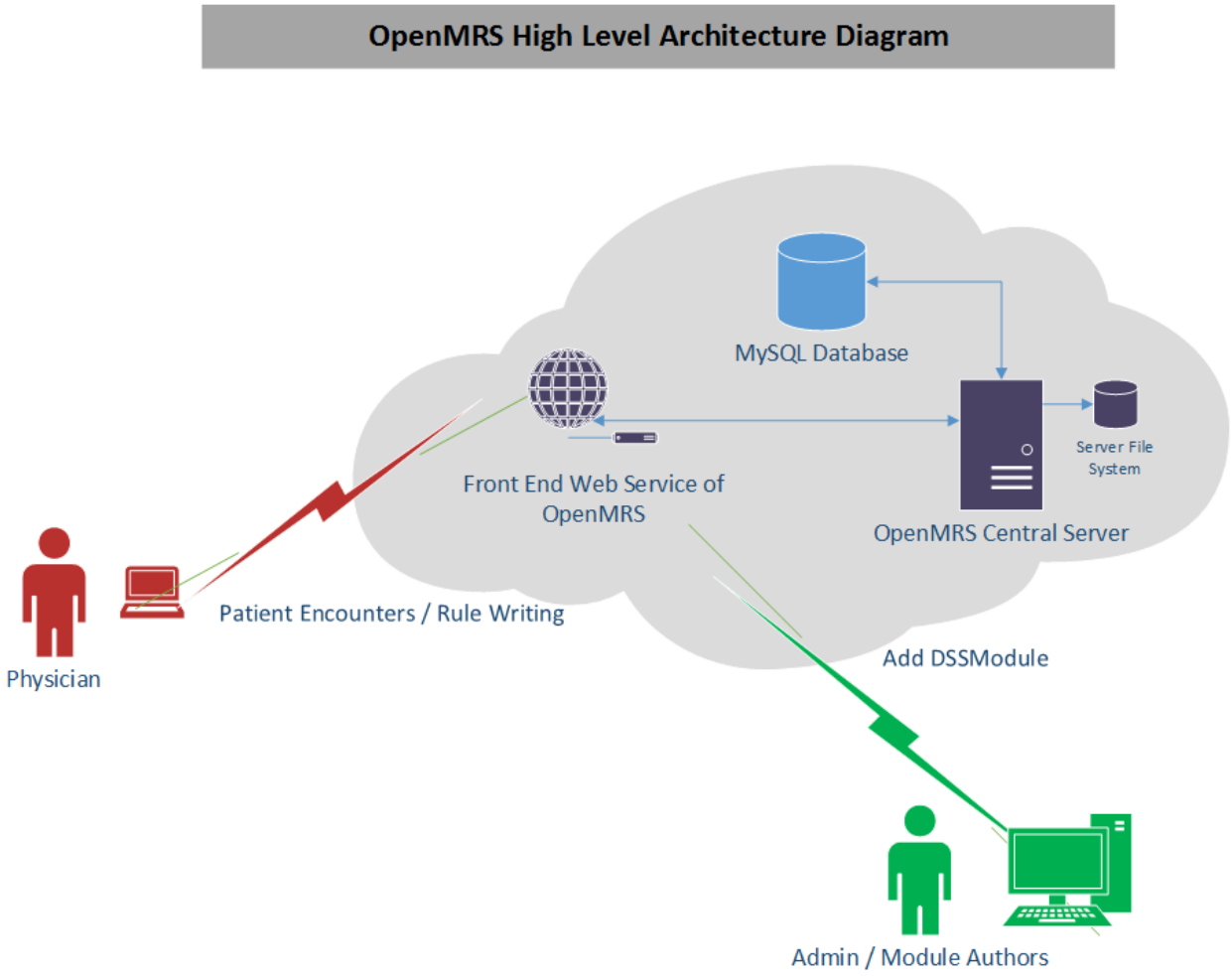


Figure 1: Architecture Diagram

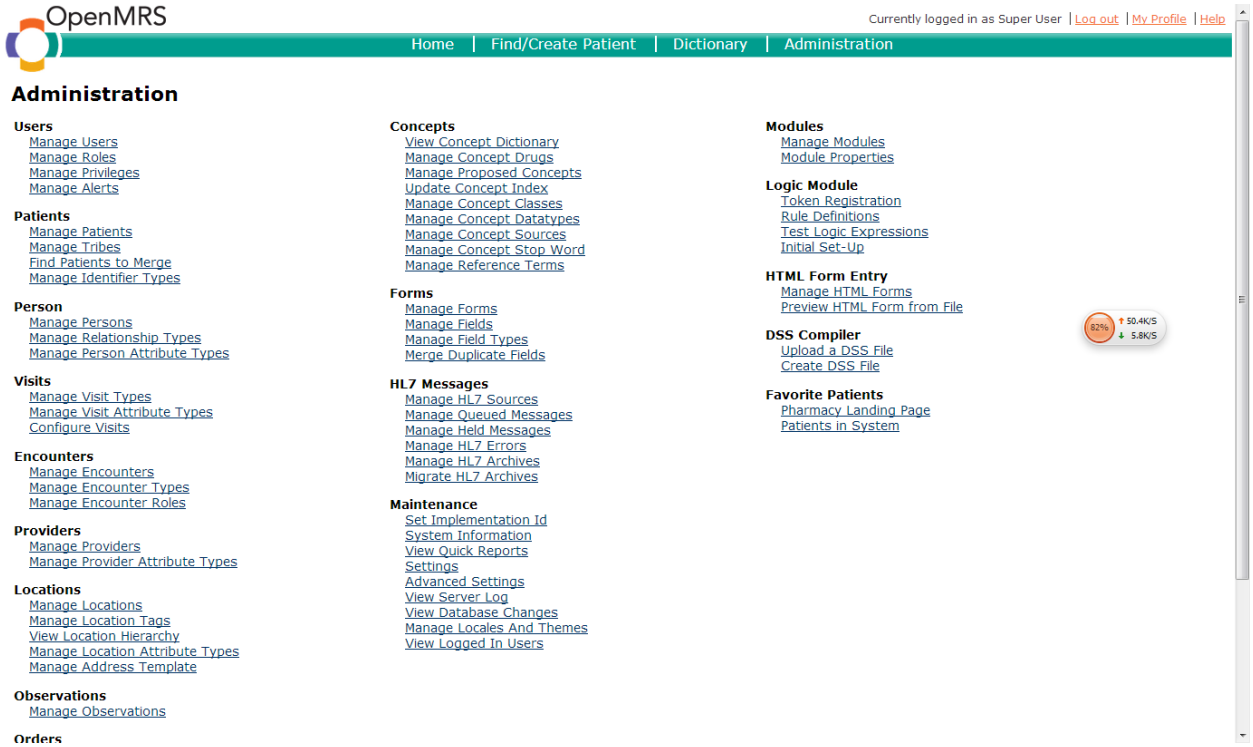


Figure 2: OpenMRS Administration

# 3 User guide

## 3.1 Rules

A DSS rule is a program which examines patient data and, as appropriate, may issue alerts to specific pages within OpenMRS.

Links for manipulating DSS rules are found in the "DSS Compiler" section of the OpenMRS Administration page, as seen in Figure 7.

### 3.1.1 Creating a rule

An OpenMRS user may create new rules using the "Create DSS Rule" link found on the Administration page. This page is shown in Figure 3. The rule's author may enter a name and source code to the rule; pressing "Save" stores the rule to the DSS rule module.

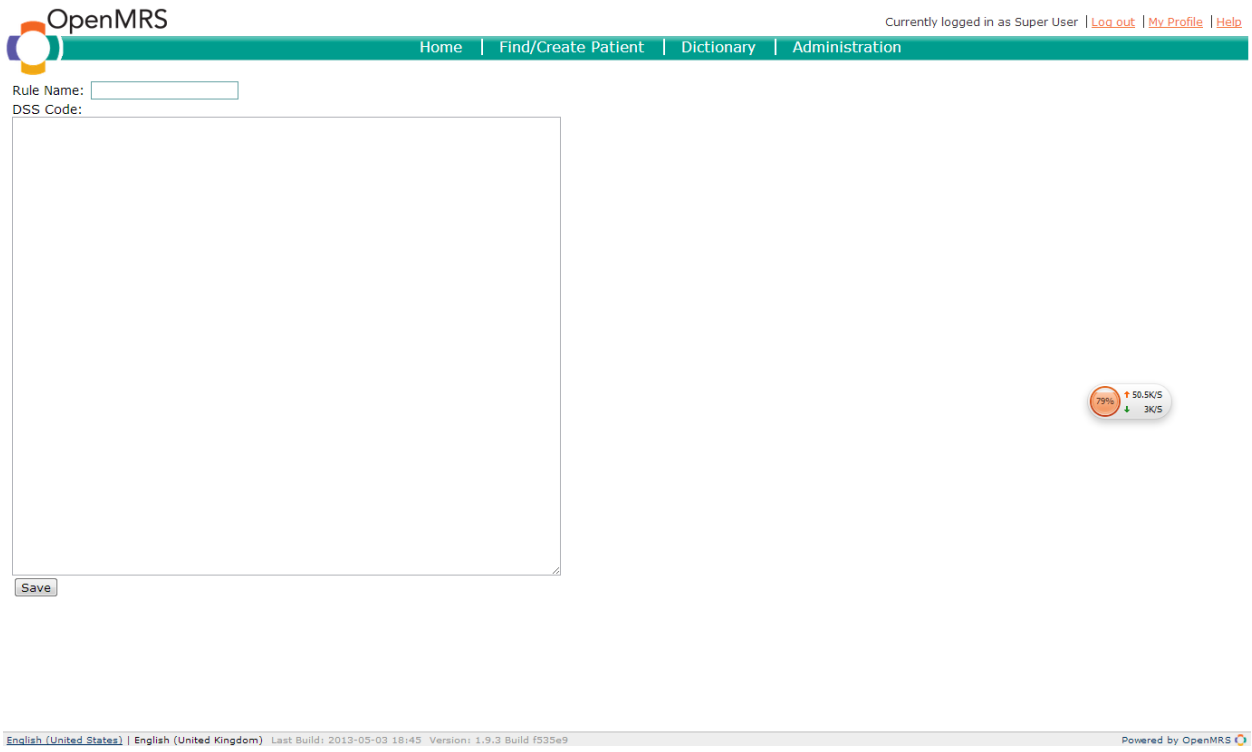


Figure 3: Creating a DSS Rule

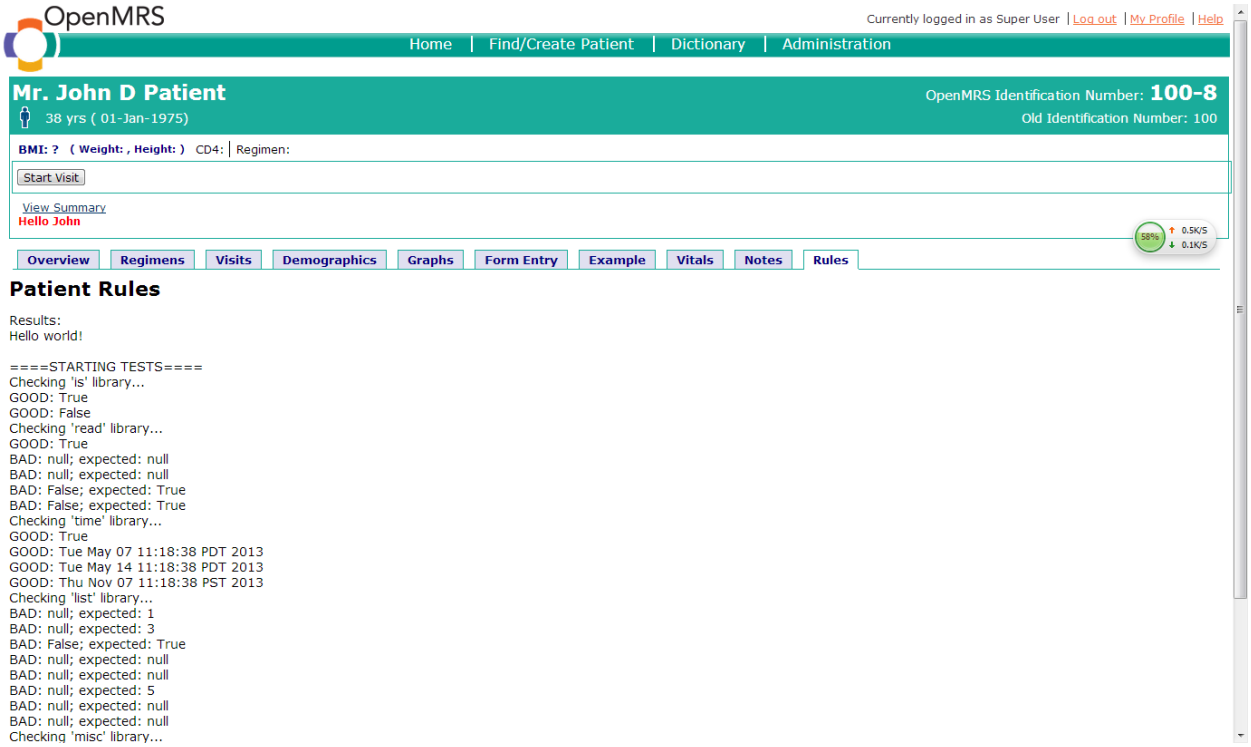


Figure 4: Patient Dashboard

### 3.1.2 Modifying an existing rule

## 3.2 Alerts

### 3.2.1 Patient dashboard

When viewing the Patient Dashboard, shown in Figure 4, any relevant alerts for the patient appear in the top left. There is also a "View Summary" link to access the Patient Summary.

### 3.2.2 Patient summary

The Patient Summary, shown in Figure 5, shows certain relevant patient information, followed by any relevant alerts that are generated by the rule service.

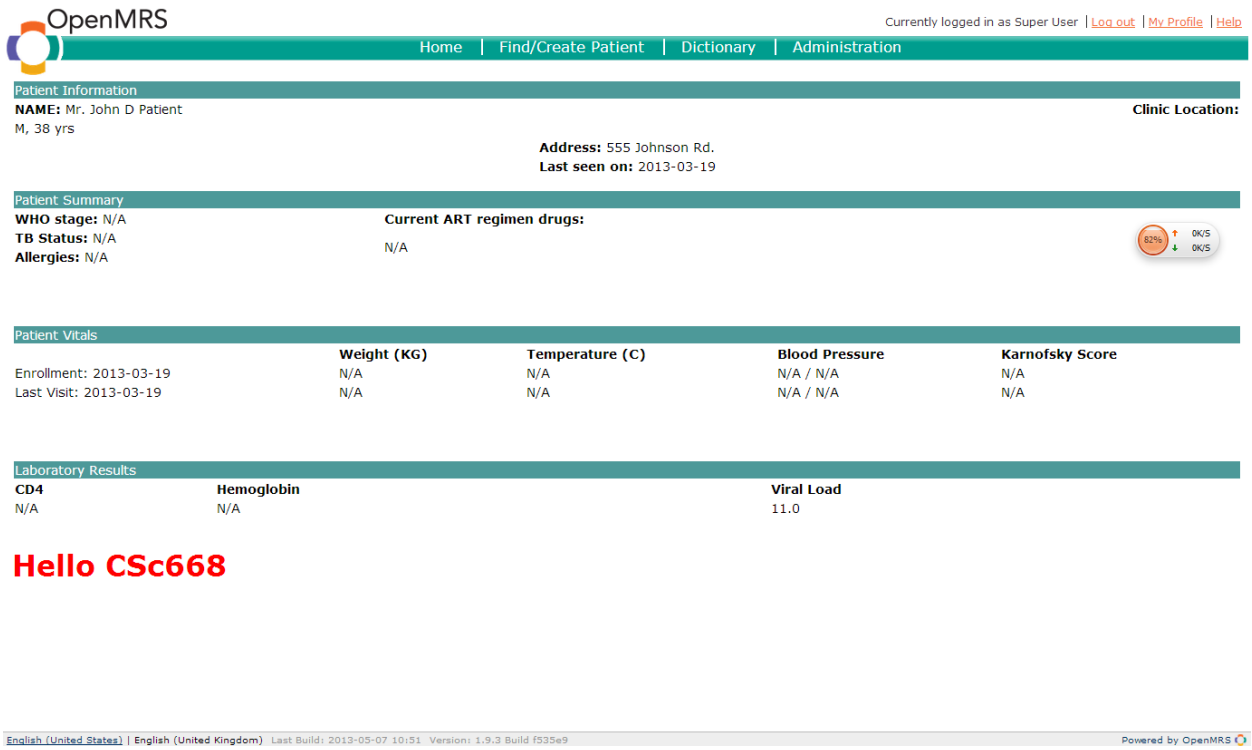


Figure 5: Patient Summary

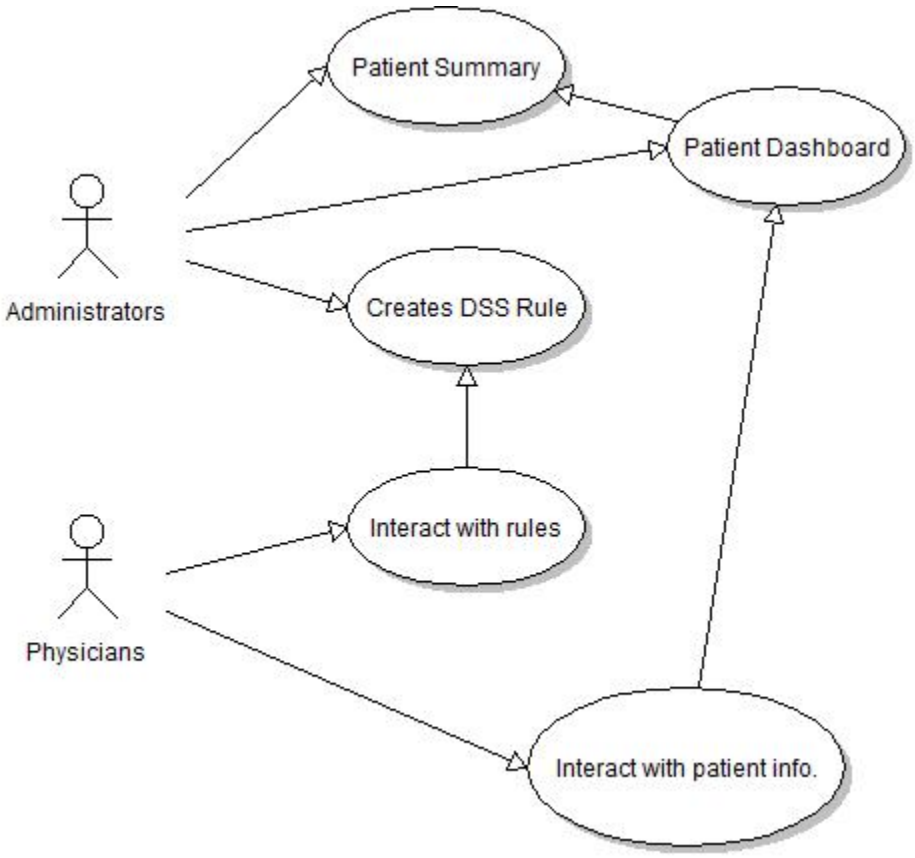


Figure 6: Use case overview

# 4 Use cases

## 4.1 Rule administration

## 4.2 Alerts

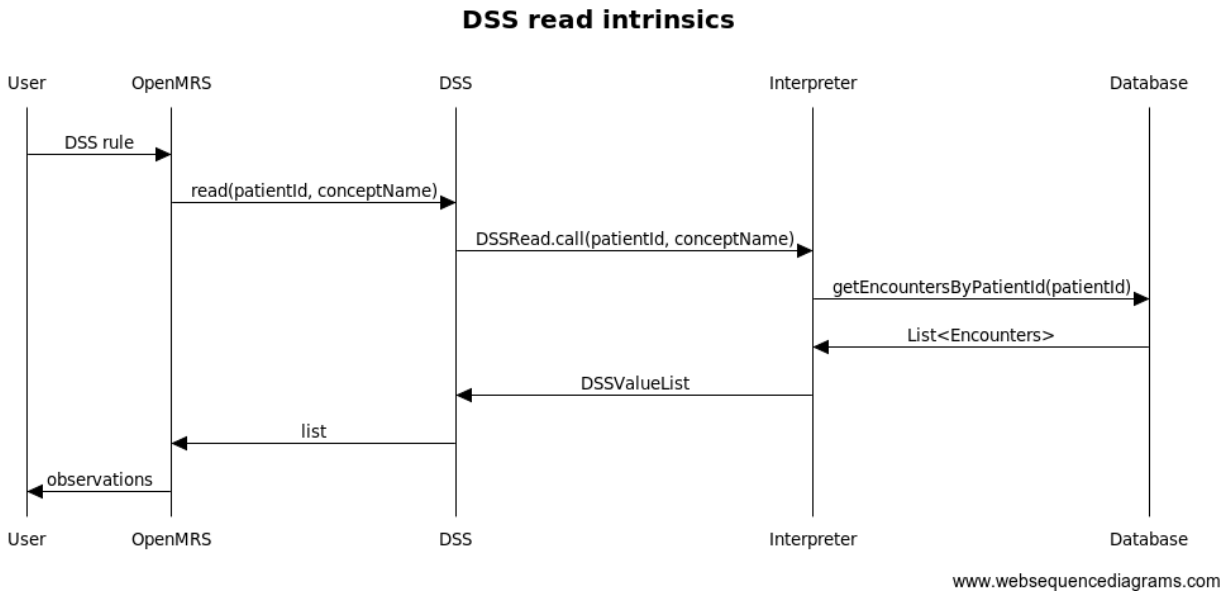


Figure 7: Sequence diagram for read operation

# 5 Sequence diagrams

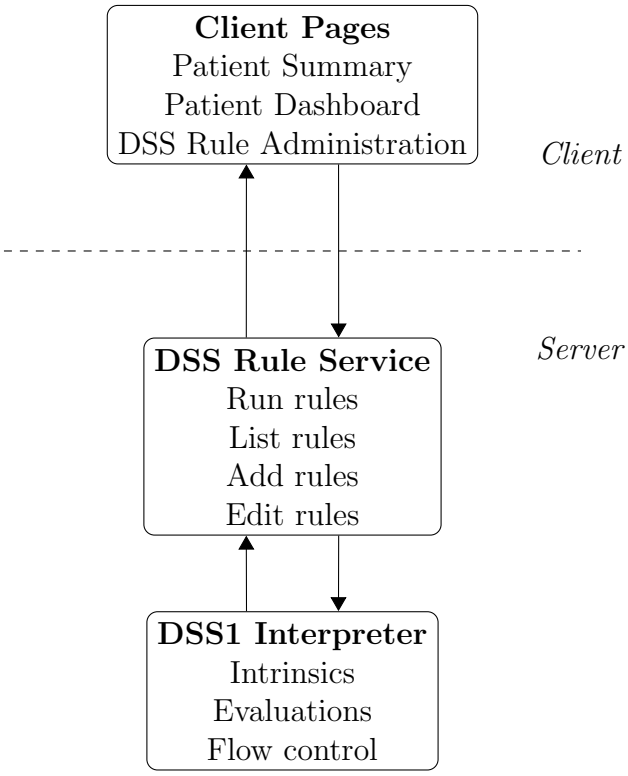


Figure 8: Architecture overview

# 6 Design overview

The DSS1 rule subsystem is incorporated into OpenMRS in a simple Client-Server fashion. The target implementation will feature client-side web pages which interact with the DSS1 rule subsystem on the server by way of DSSRuleService. Figure 8 illustrates this interaction.

The DSS Rule Service, in turn, utilizing the DSS1 Interpreter subsystem to run rules and report results.

While the DSS Rule Service runs on the server, its interface is exposed to client-side JavaScript code via DWR (Direct Web Remoting).

## 6.1 Client pages

Multiple client web pages interact with the DSS Rule Service.

### 6.1.1 Patient summary

The Patient Summary (`patientsummary.jsp`) is stand-alone page, reachable from a link on the Patient Dashboard. It is used primarily to contain major information about a patient (gender, age, WHO stage, etc.) The Patient Summary invokes the rule service via DWR to retrieve all alerts for the named target `summary` and displays them below other patient information.

### 6.1.2 Patient Dashboard

The Patient Dashboard is the primary landing point for viewing patient information, and contains multiple tabs for this purpose. This extension inserts a link to the Patient Summary on the Patient Dashboard, and accompanies this with relevant alerts by invoking the rule service for the `dashboard` target.

See `org.openmrs.module.basicmodule.extension.html.PatientSummaryExtension`

### 6.1.3 DSS Rule Administration

Create DSS Rule (`dssRules.form`) provides a form where DSS source code can be entered and submitted to the rule service with a specific rule name. Consolidates the ability to create new rules, load existing rules, and edit rules in one form. Made accessible through an extension to the Administration menu.

## 6.2 Rule service

The DSSRuleService follows the facade design pattern to expose important functionality to clients. The high-level tasks that are relevant to client code are defined using a few simple methods which hide the details of compiling, interpreting, and managing the storage of rules. Specific functionality is detailed in table 1.

### 6.2.1 Rule storage

On upload, rules are compiled to an Abstract Syntax Tree (AST) form using the provided Parser class. Once compiled successfully, the rule is stored to the OpenMRS application data directory.

Each rule is stored in two formats: Plain text source code, and an XML (Extensible Markup Language) representation of the compiled AST. The source code is subsequently used only to support user interactions (for instance, if an administrator wants to load or

| Method | Description | Details |
|---|---|---|
| `store(rule, code)` | Stores a rule (either as a new rule, or replacing an existing rule) with the given source code. | Invokes the Parser to convert source code to AST; Invokes the XMLBuilder to convert AST to DOM and save; Saves the original source to file system for subsequent retrieval; Stores the AST in memory for subsequent running. |
| `load(rule)` | Load the source code for an existing rule. | Reads stored source code from the file system. |
| `listRules()` | List all existing rules. | Returns a list of all stored rule names. |
| `runRules(patientId, target)` | Get all alerts for the given target (summary or dashboard) as appropriate to the given patient. | For each rule: Construct interpreter; Install intrinsics, including alert function which stores to a map; Predefine patientId for DSS1 program; Run the interpreter on the rule. Thereafter, pull all alerts appropriate to the target from the map. |

Table 1: Methods exposed by the DSS Rule Service

modify source for an existing rule). The XML form is used when the DSS Rule Service is first initialized to load any existing rules from the file system. After initialization, rules are stored in memory as AST objects.

The utility class XMLBuilder is used for conversion between AST and XML. Internally, the class maintains a Document Object Model (DOM) representation of the AST. This can be either as loaded from an XML file, or as formed by traversing an AST. Likewise, XMLBuilder provides methods for both producing AST objects or writing XML files.

## 6.3 Interpreter

The Interpreter is implemented with four distinct sub systems, as depicted in Figure 9. At the top level, *flow control* is provided by the InterpreterVisitor, which is responsible for traversing the Abstract Syntax Tree. An *execution context* is maintained to describe the running state of the system, including defined variables and functions. While tree traversal coordinates complex expressions, the actual *evaluation* of expressions is itself implemented in a distinct set of classes representing the types available under DSS1. Finally, a library of *intrinsic functions* is provided in order to mediate interactions with OpenMRS from running DSS1, as well as to provide certain convenience functions to DSS1 rule programmers.

### 6.3.1 Flow control

Flow control in the interpreter is implemented using the Visitor design pattern, traversing the Abstract Syntax Tree (AST) produced by the existing Compiler using an implementation of the provided ASTVisitor interface, performing computation as appropriate at every given node in the tree.

The Visitor design pattern leverages double dispatch to decouple a data structure from the operations which can be performed while traversing this data structure. The Visitor calls an `accept` method on a node within the data structure, which is itself overloaded to call a more specific method on the Visitor itself; `visitBlockTree`, for example. This permits the external object the Visitor to implement behavior using the data structure's type hierarchy, without adding that specific behavior to those types directly.

In the case of the Interpreter, the data structure is the AST, which describes a DSS1 program as a tree of elements block (BlockTree), if statements (IfTree), et cetera. The Visitor is the IntepreterVisitor, which manages and performs the computation described by this program. This is done with the support of other underlying subsystems to describe variable state and perform type-specific evaluations, as described in the Architecture section.

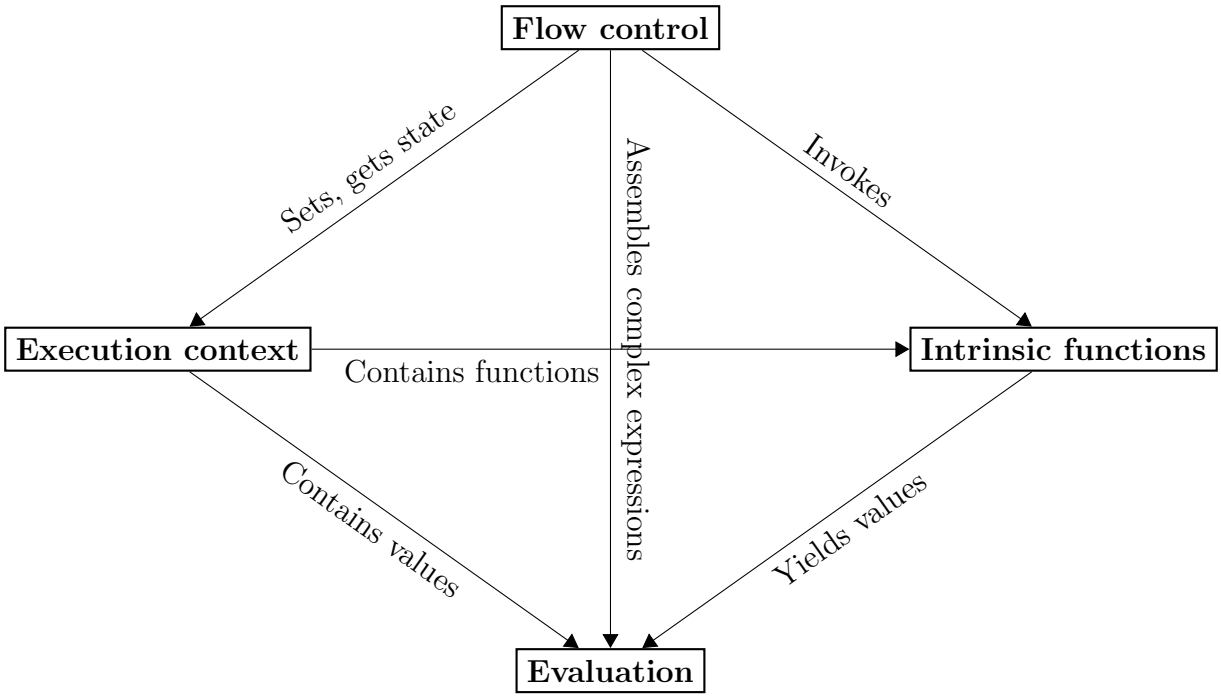


Figure 9: High-level overview of the DSS1 Interpreter.

The class InterpreterVisitor acts as the center of a subsystem responsible for high-level interpretation of the program, including flow control, and coordinating complex expressions.

### 6.3.2 Execution context

The ExecutionContext class provides a means to store and retrieve return values, variable states, and named functions. It also handles rules of scope to hide variables during function calls, and exposes the Evaluator. Note that this may be populated with functions or even variables before being given to the InterpreterVisitor, allowing the definition of intrinsics and constants (such as `patientId`).

Functions stored in the ExecutionContext are of type DSSFunction, which is an interface used to describe any function called from DSS1 (either intrinsic or user-defined). This permits function calling to be implemented identically for both categories of function. Additionally includes a method for testing if a given argument should be passed as a raw identifier instead of evaluated directly (as used by some intrinsics.).

Similarly, variables and return values are stored as DSSValue objects, with their specific implementation defined within the evaluation subsystem.

### 6.3.3 Evaluation of expressions

The abstract class DSSValue describes a set of operations which can be performed on values in DSS1 as methods, as well as the common state (the potential to store time stamps). Its concrete sub-classes, such as DSSValueInt, DSSValueFloat, et cetera, provide specific implementations of these operations in order to define the behavior of their DSS1 type. Additionally, concrete subclasses of DSSValue typically are defined with some field to maintain their specific value (for instance, DSSValueBool has an underlying Java `boolean` field to describe its value.)

The Evaluator interface and DSSEvaluator implementation exposes methods to perform operations upon DSS1 values, to interpret literals, allocate DSS objects, and perform conversions between DSS1 values and similar Java objects. The Evaluator serves as intermediary between flow control and the specific semantics implemented in DSSValue types; this facilitates separation of concerns, allowing the gradual introduction of new DSS1 data types while avoiding changes to flow control.

Finally, a DSSValueFactory class is provided to aid in the instantiation of DSSValue objects. This class utilizes the Factory design pattern to allow new values to be created (for instance, as the return values of intrinsic functions) without requiring users of those values to have specific knowledge of the DSSValue subclasses actually used.

### 6.3.4 Intrinsic functions

DSSLibrary defines an interface for delivering or generating intrinsic functions in related groupings. Each function is returned in a map where the name should be used to call the function from a DSS program, and the function object is used as a Java object that extends the DSSFunction. These functions may then be easily installed into the ExecutionContext used by the Intepreter before running rules. A list of libraries used in this implementation is presented in Table 2.

This approach supports extensibility of the DSS rule module. Rather than being built into the DSS1 Interpreter at the language level, intrinsic functions can be contained and communicated as DSSLibrary objects. Adding intrinsics is then as simple as defining a new DSSLibrary and installing it to the execution context before running rules.

The ReadLibrary serves as an interesting example case, at it illustrates interaction with the OpenMRS platform without requiring specific knowledge of this platform from other elements of the interpreter. The read functions retrieve a list of observations associated with a patient. The first parameter of the functions, patientId, is a numeric identifier unique to each patient. The second parameter, conceptName, is the word or phrase used by the OpenMRS dictionary to refer to a concept.

The three functions are nearly identical, save for one difference: while read() returns a list containing all observations that match the function parameters, readInitialEncounter() and readLatestEncounter() filter out results based on the timestamp of the observations. Calling readInitialEncounter() retrieves only the observations from the patient's earliest encounter on record, while readLatestEncounter() retrieves only the observations from the patient's most recent encounter on record.

When the functions are called, they retrieve a list of all encounters associated with patientId from the OpenMRS database. The functions iterate through these lists, and in the case of readInitialEncounter() and readLatestEncounter(), the timestamp for each encounter is checked. If the timestamp does not meet the criteria, the encounter is discarded. Once an encounter has been verified as valid the function shall retrieve all observations associated with the encounter. Each observation shall have its concept name checked against conceptName, and matches are added to the list of observations that each function shall return.

Observations consist of three pieces of data: the value of the observation, the data type of the observation value, and the time of the observation. Internally, observations are represented as DSSValue objects, which store the value of the observation and the time of the observation. The data type of the observation value is stored as part of the DSSValue class type itself. Both the time and data type of the observation can be retrieved using the time() and type check intrinsics, respectively.

Note that the `alert` intrinsic is treated as a special case. Rather than being contained within a library class, it is installed directly by the DSS Rule Service into the Interpreter before running rules. This facilitates retrieval of results issued via `alert` calls.

| **Library** | **Functions implemented** |
|---|---|
| IsLibrary | `isString(var)`<br>`isFloat(var)`<br>`isInt(var)`<br>`isBoolean(var)`<br>`isList(var)`<br>`isObject(var)`<br>`isDate(var)` |
| LengthAndWithinLibrary | `length(var)`<br>`within(v,a,b)` |
| ListLibrary | `merge(a,b)`<br>`sortTime(list)`<br>`sortData(list)`<br>`first(list)`<br>`last(list)` |
| ReadLibrary | `read(patientId, concept)`<br>`readInitialEncounter(patientId, concept)`<br>`readLatestEncounter(patientId, concept)` |
| DateLibrary | `currenttime()`<br>`recentTimeItem(list)`<br>`oldestTimeItem(list)`<br>`before(a,b)`<br>`time(var)`<br>`addDays(v,days)`<br>`addMonths(v,months)` |

Table 2: Libraries of intrinsics

# 7 Package diagrams

# 8 Class diagrams

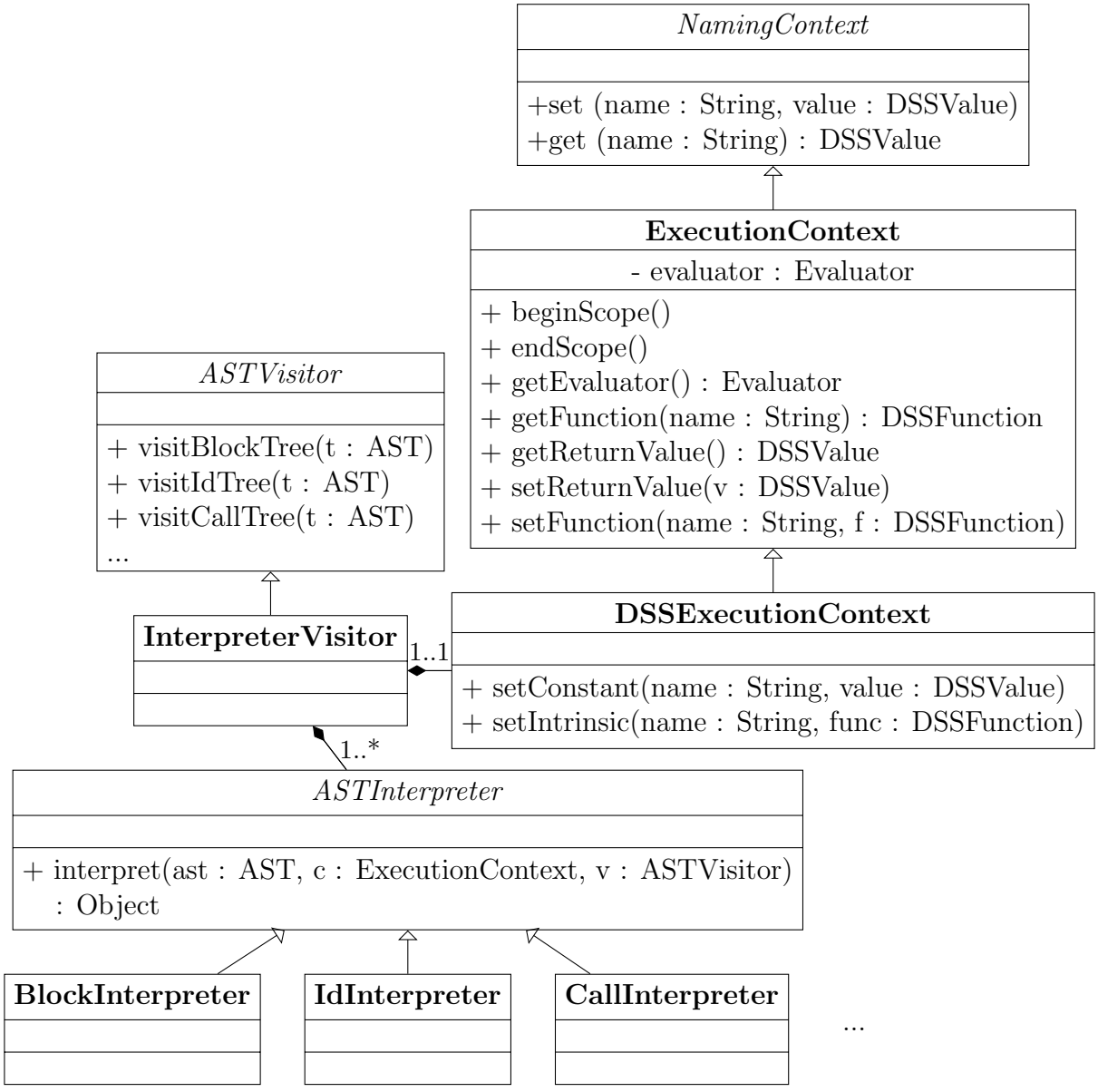


Figure 10: Interpreter visitor

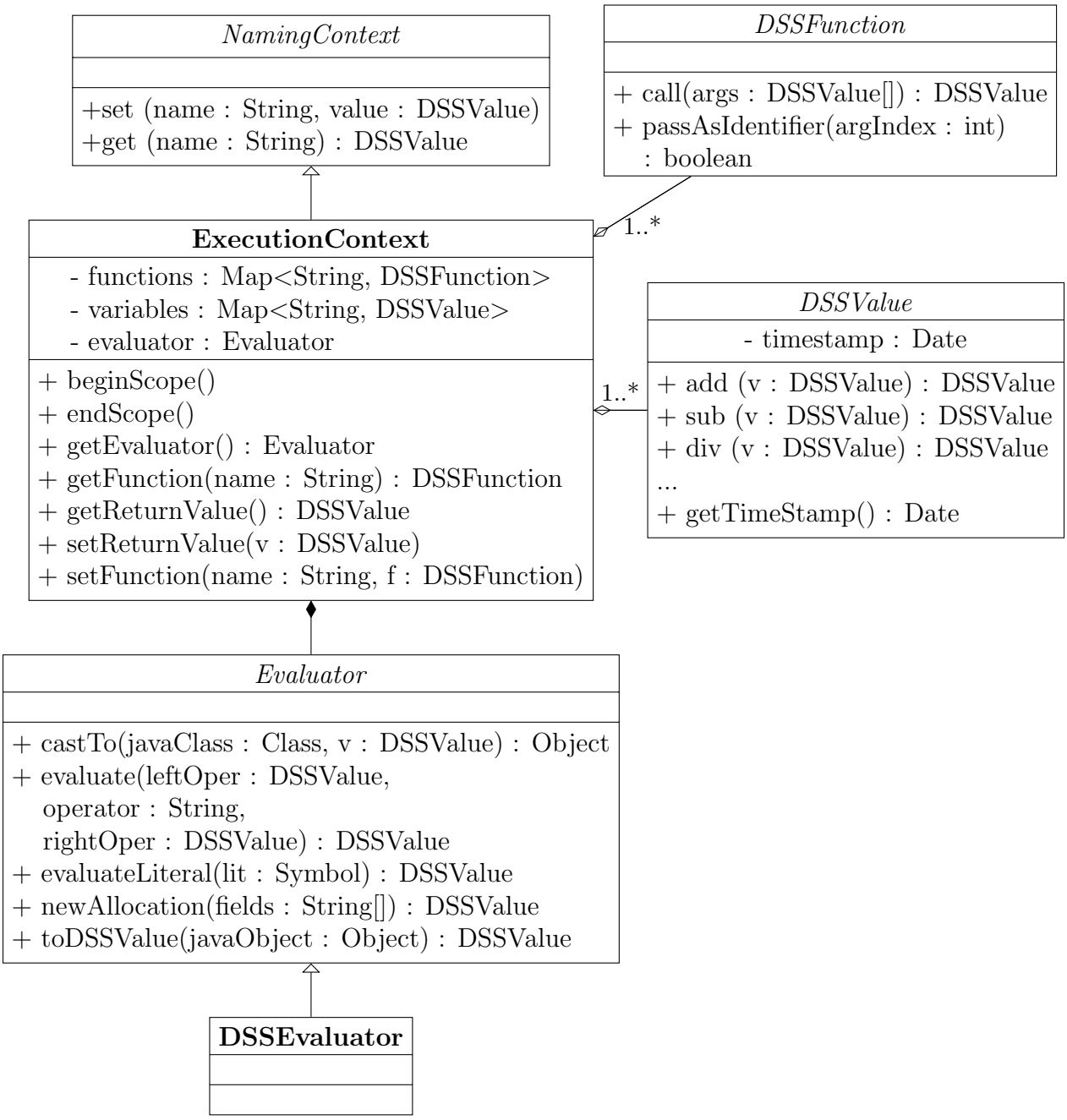


Figure 11: Execution context class diagram

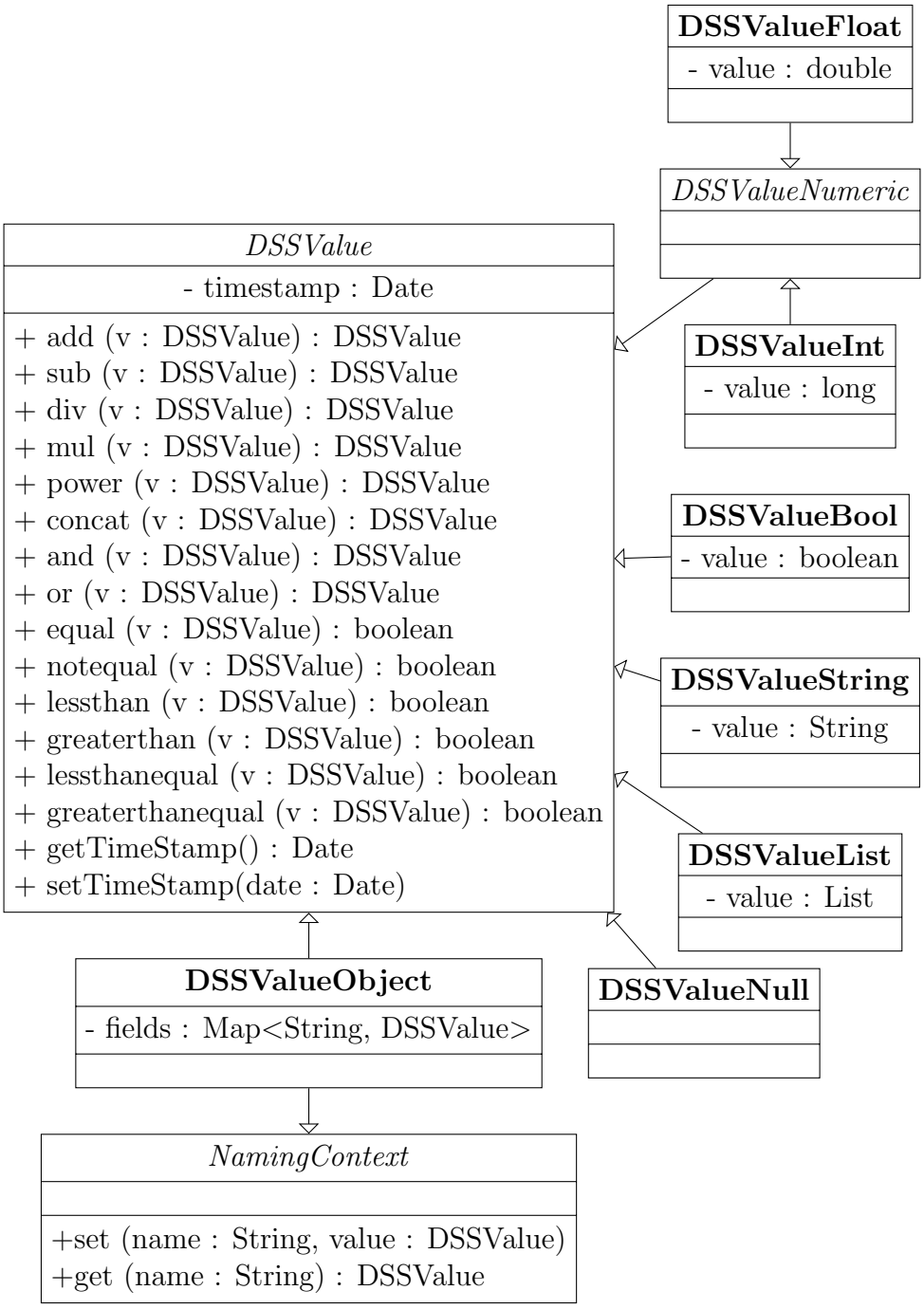


Figure 12: Value class diagram

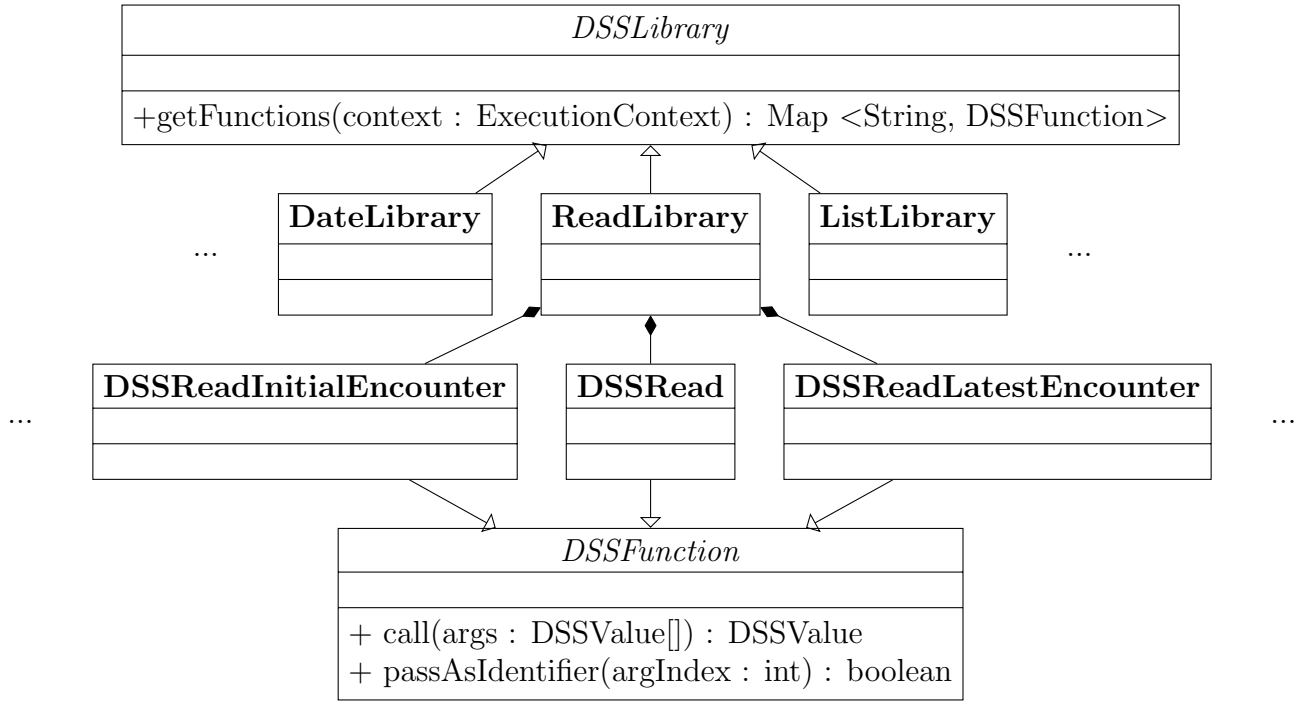


Figure 13: Intrinsic class diagram

# 9 API documentation