Polina Gohstein, David James, Jeff Magder, Olga Rodimina
CSC408: Software Engineering
Professor Reid
February 2, 2005

JCVSReport

Assess the Progress of a Java Project in CVS

Polina Gohstein
David James
Jeff Magder
Olga Rodimina

Table of Contents

Chapter 1: Design Document for JCVSReport

Abstract

JCVSReport is designed to help software development teams to assess the process of Java projects that are maintained in a CVS repository. JCVSReport will use its data gathering component to collect data from both the CVS metadata and the Java source files. The gathering component will in turn interact with the data storage component and will create a representation of the collected data in a project database. Once JCVSReport finishes gathering the required data, the presentation component will take over. The presentation component will rely on a configuration file provided to JCVSReport by the user. The user will use the configuration file to specify what types of metrics and statistics should be displayed in the final report. The presentation component will query the database for the specified metrics and statistics and will use a report component to output the final HTML report.

In this document together with the Analysis and Estimation documents for each system component we will try and provide you with a complete picture of the design and architecture of JCVSReport.

System Requirements

Functional Requirements

Data Collection

- User must be able to specify what metrics should be tracked in the data storage component.
- Users must be able to customize metrics that are tracked in the storage component.
- Statistical dataset that is kept in the data storage component should not be rebuilt every time the system runs. Instead the system must be able to store and add to previously collected data.
- Data Storage component should be able to regenerate the data from scratch if any configuration option change.
- User should be able to compare progress of one project with the progress of another.
- The user must be able to extract the following statistics from projects CVS metadata:
    - number of changed/added/deleted lines (per developer, per group of files, for specific period of time)
    - the number of files

    - the number of commits/updates/checkouts
- The user must be able to extract the following statistics from projects source files:
    - number of lines of code
    - number of lines of test code
    - number of unit tests
    - lines of comments
    - lines of documentation
    - number of methods
    - number of classes
    - number of imports per source file

Presentation

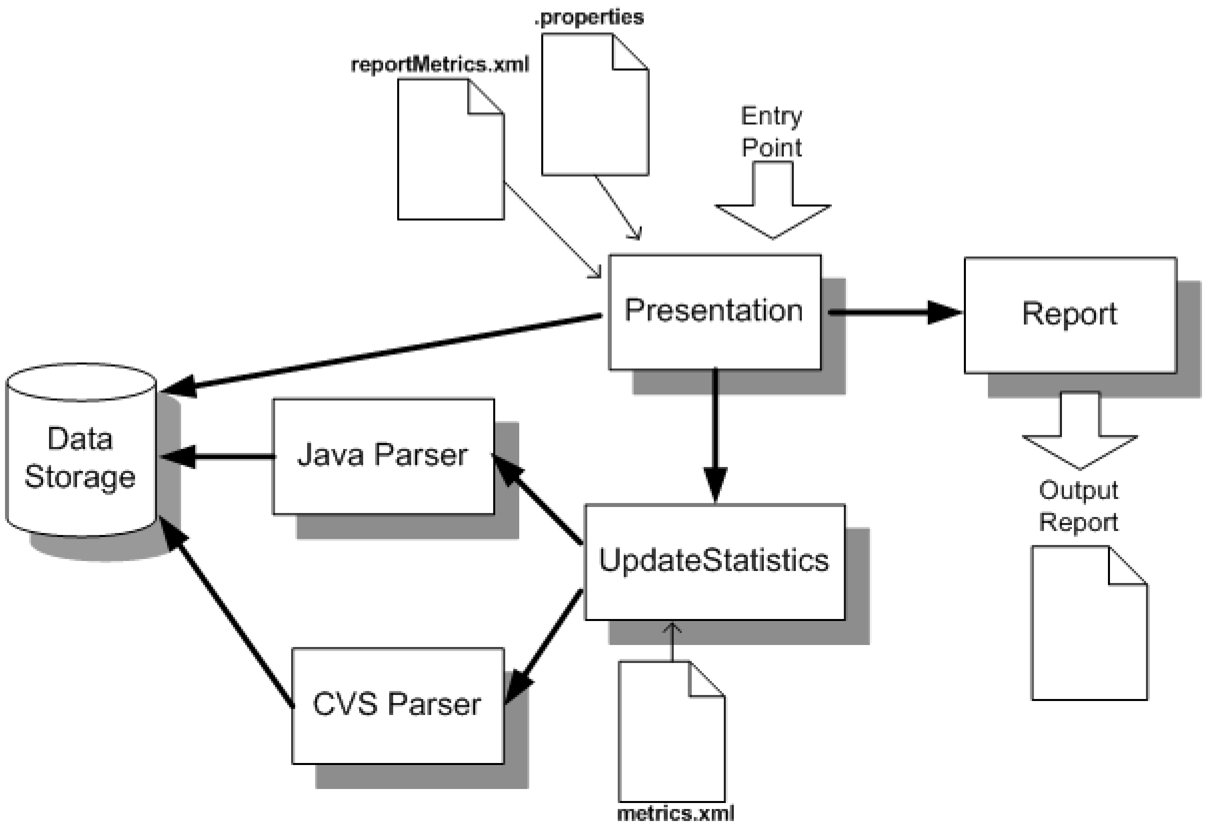
- User should be able to configure which data will be presented in the HTML report.
- User should be able to configure how the data should be presented.
- User should be able to configure granularity of presented data.
- System should output the report as static HTML pages.

Non-Functional Requirements

- The system must be extensible (developer level).
- The system should be easy to install (it should take less than 10 minutes for the TA to install).
- The system should work on CDF.
- The system can make use of third party tools/libraries only if they are open source or available on CDF.

System Architecture

JCVSReport can be broken down into the following components: Presentation, UpdateStatistics, CVS Parser, Java Parser, Data Storage and Report. The following diagram captures the individual components and the communication between the components. Moreover, it also captures the configuration files, the entry point and the output of JCVSReport.



.properties

This is a user configuration file. The user will use this file to specify the following information to JCVSReport: the CVS repository location, CVS login details, the project module, database location/name, database login details, the list of metrics to be included in the report. Here is an example of a .properties file:

```
cvs repository location=:pserver:anonymous@cvs.sourceforge.net:/cvsroot/bloof
cvs user=anonymous
cvs password=
cvs project module=bloof
database=bloofdb
database user=anonymous
database password=
report metrics=commitsPerDeveloper, linesOfCodePerDeveloper,
numberOfFunctionsPerFile
```

Presentation Component

The Presentation component will be the entry point of JCVSReport, i.e. this component will contain the main class that will be called when the user invokes JCVSReport via command line. The Presentation component will parse the .properties file to get the CVS information and the database information specified by the user. It will then pass this information to the UpdateStatistics component asking it to update the database. Once the UpdateStatistics component brings the database up to date with the CVS repository, the Presentation component will call the Report component to generate the output report. For every report metric specified by the user in the .properties file, the Presentation component will do the following:

- Find the definition of this report metric in reportMetrics.xml.
- Run the corresponding SQL query on the database.
- Graph or display the results of the query using the Report component.

reportMetrics.xml

JCVSReport will use the reportMetrics.xml file to define metrics that the user might want to include in their report. More specifically, for every such metric, the file will define the SQL query and type of graph. Here is a sample reportMetrics.xml file:

```
<reportMetrics>
    <reportMetric name=”commitsPerDeveloper” >
        <graph type=“BarGraph”  title=“Number of Commits”
axisLabel=”Developer,Number Of Commits” />
        <query>SELECT developer, COUNT(UNIQUE revision_id) FROM revisions GROUP
BY developer</query>
    </reportMetric>
    <reportMetric name=”linesOfCodePerDeveloper” >
        <graph type=“PieChart”  title=“Lines of Code Added Per Developer”
axisLabel=”Developer,Number Of Commits” />
        <query>SELECT developer, lines_add FROM  revisions GROUP BY developer<
/query>
    </reportMetric>
    <reportMetric name=”numberOfFunctionsPerFile” >
        <graph type=“BarGraph”  title=“Functions Per File”
axisLabel=”File,Number Of Functions” />
        <query>SELECT developer, metricValue FROM  metric WHERE metricName =
“numFunctions” GROUP BY developer< /query>
    </reportMetric>
</reportMetrics>
```

UpdateStatistics Component

The UpdateStatistics component will be in charge of bringing the statistics in the database up to date with the CVS repository. UpdateStatistics will first make a call to the CVSParser component (Bloof) asking it to update the database with the CVS metadata. Once the database contains up-to-date information of all the files and their revisions, the UpdateStatistics component will start updating the database with information that comes from the individual files. UpdateStatistics will parse the metrics.xml file and for every metric in this file, it will do the following:

- Dynamically load the Metric class specified in the xml file and create a metric object.
- Call Metric.update(file, revision) for every new revision of every file in the CVS.

The notion of a metric object belongs to the UpdateStatistics component. metrics.xml is a power user configuration file. It provides a convenient way of defining how the individual metrics should be gathered and updated. There are two types of metric objects: the Java AST metric and the Regular Expression metric. Both types implement the same Metric interface. Java AST metrics are metrics that can be gathered by parsing an Abstract Syntax Tree (AST) generated from a Java source file. These metrics will make use of the Java Parser component to perform this task. Regular Expression metrics are metrics that can only be generated by matching lines of a Java source file with regular expressions. For more information on these types of metrics, please refer to the Java Parser A&E.

metrics.xml

It is possible for each statistic collection module to collect different statistics at different times. For example, the Java Parser may only collect statistics on the number of methods in a file, but later it is decided we need the number of classes as well. This can be easily added without making code changes, simply by modifying the configuration file. Here is a sample configuration file:

```
<metric name="numMethods">
  <description>Number of Functions</description>
  <fileExtension>.java</fileExtension>
  <metricClass>JavaASTMatcher</metricClass>
  <match>MethodDeclarator</match>
</metric>
```

The `<metric name = “numMethods”>` tells the software that the metric to be updated in the database is called numMethods. This is important for both information storage and retrieval at a later time. That is, *numMethods* would fill in SQL query string after the *FROM* field for retrieving statistics. Similarly, it would be the assigned *VALUE* to the SQL query string during database updates. (See the Data Component A&E for details).

The `<description>Number of Functions</description>` line is simply a way of storing useful human readable information for this metric.

The `<fileExtension>.java</fileExtension>` line tells what file types this metric will work on. For now this is only .java, but could be expanded to work on other file types, such as .cpp.

The `<metricClass>JavaASTMatcher</metricClass>` line tells us which module this statistic belongs to. This is very relevant for metric gathering and updating, as it is the one line that tells each module that this XML tidbit applies to it. Here, `JavaASTMatcher` refers to the Java Parsing module. The `RegExpLineCounter` also refers to that module but to the group A statistics. (See the Java Parser A&E document for more details on group A and group B metrics).

The final `<match>MethodDeclarator</match>` line, tells us specifics about how the statistics gathering module is to find its metric. In the case of the AST Matcher section of the Java Parser module (Group B Metrics), the module simply counts the number of occurrences of `MethodDeclarator` in the AST tree. If we wanted to match the number of if, or else statements, we'd use `IfStatement|ElseStatement` in the place of `MethodDeclarator`, since those statements are what would be generated by the AST generator.

A more complete configuration file is shown below:

```
<metrics>
   <metric name="numMethods">
      <description>Number of Functions</description>
      <fileExtension>.java</fileExtension>
      <metricClass>JavaASTMatcher</metricClass>
      <match>MethodDeclarator</match>
   </metric>
   <metric name="complexity">
      <description>Number of If, Else, Case, While, For, or Try
Statements</description>
      <fileExtension>.java</fileExtension>
      <metricClass>JavaASTMatcher</metricClass>

    <match>IfStatement|ElseStatement|CaseStatement|WhileStatement|ForStatement|
TryStatement</match>
   </metric>
   <metric name="comments">
      <description>Number of Lines of Comments</description>
      <fileExtension>.java</fileExtension>
      <metricClass>RegExpLineCounter</metricClass>
      <match>//|/*(.+?)*/</match>
   </metric>
   <metric name="JavaDoc">
      <description>Number of Lines of JavaDoc</description>
      <fileExtension>.java</fileExtension>
      <metricClass>RegExpLineCounter</metricClass>
      <match>///|/**(.+?)*/</match>
   </metric>
</metrics>
```

To assist in parsing this file, all modules should use the standard Java XML parser. They should only use the information in a particular metric tag if its `<metricClass>` tag matches its module type. Such an approach allows each module to extend itself, without worrying about impacting the other modules or statistics.

CVS Parser Component

The CVS Parser component will parse the CVS log and the CVS history of a specified project. As it parses the metadata, it will store it into a relational database which is a part of the Data Storage component. Please refer to the A&E for the Data Storage component for information on database schema. Note that the CVS Parser will simply create a database representation of the metadata, with no regards to user specified statistics. Creating a representation of all the metadata in a relational database makes future operations on the data more flexible and convenient. We will use the core component of a third party solution, called Bloof, to implement the majority of the CVS Parser component. The advantage of using Bloof is the fact that it already is integrated with a database, more specifically the McKoi database. Please refer to the CVS Parser A&E for a more detailed description of this component.

Java Parser Component

Gathering metrics or statistics from the Java source files is a more complicated task than gathering CVS metadata. The Java Parser component will use an Abstract Syntax Tree (AST) generator to remove the complexities of the code by transforming it into a tree of easily recognizable tokens. It will then operate on the tree to gather a particular statistic for that Java file. Please refer to the Java Parser A&E for a more detailed description of this component.

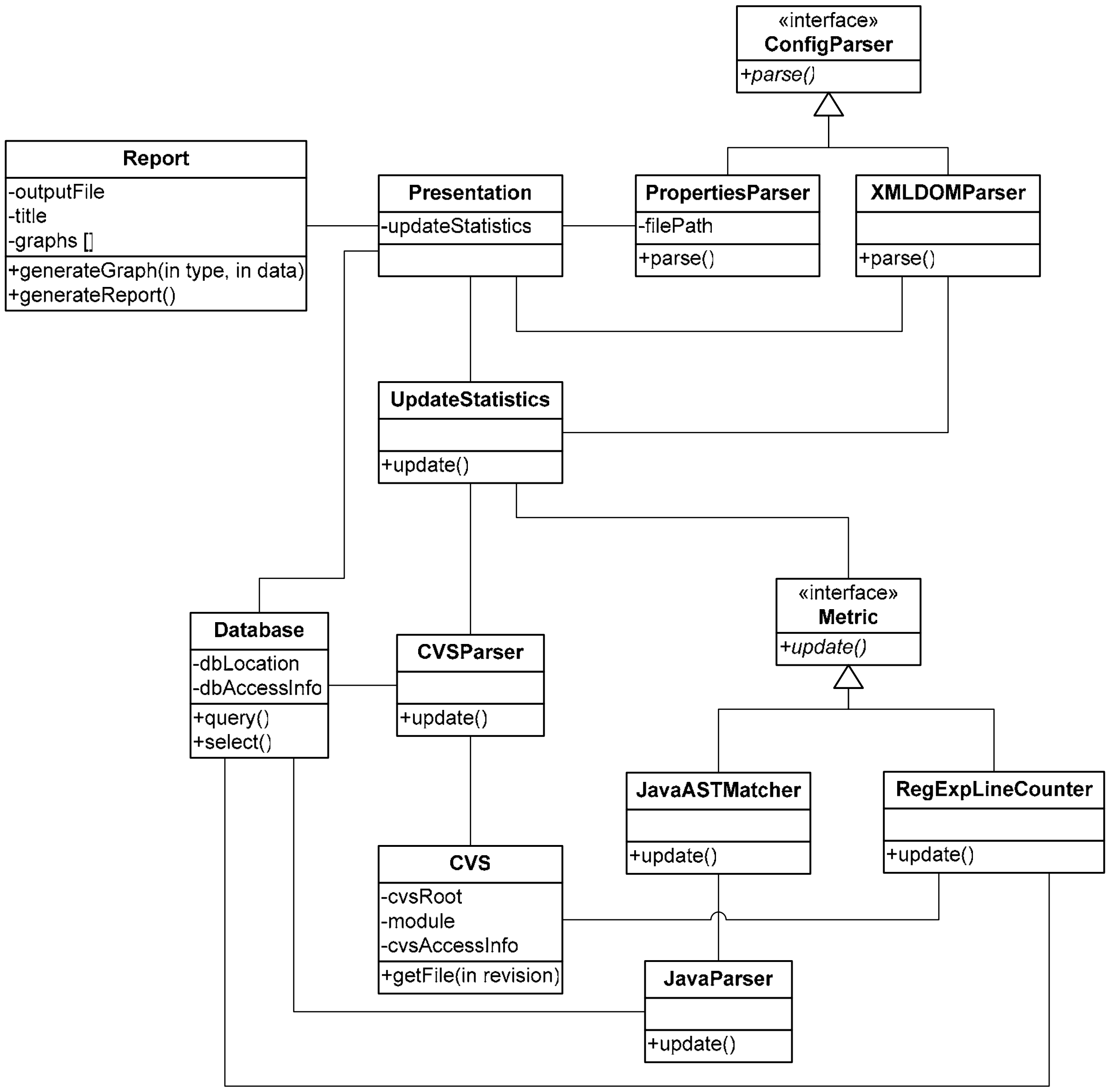
Report Component

As the name suggests, the Report component will be used for generating the final report. The final report will consist of a series of HTML pages with graphs. Please refer to the Report Component A&E for a more detailed description of this component.

Diagrams

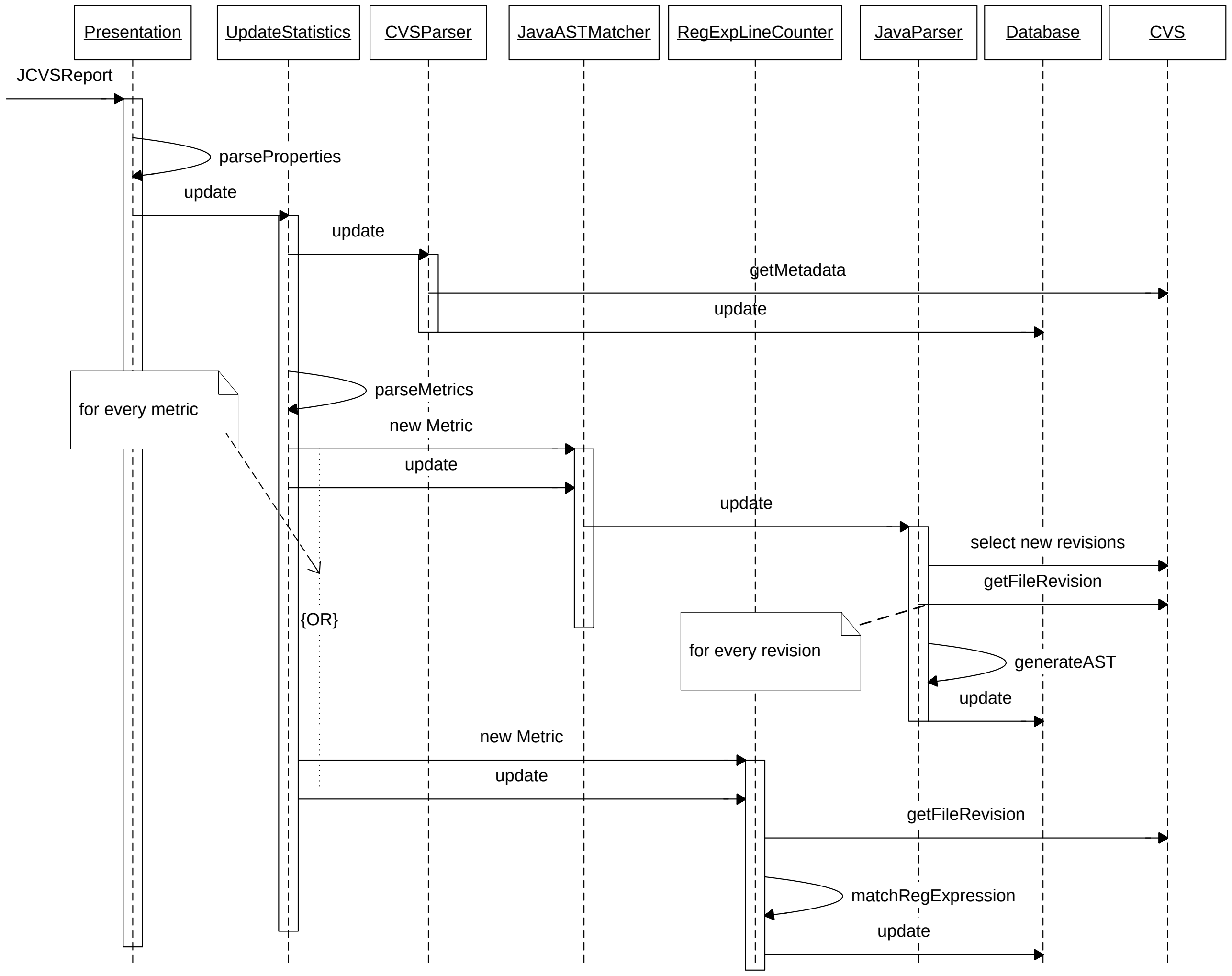
Class Diagrams

The following diagram illustrates the breakdown of the JCVSReport into its main classes and the associations between these classes.



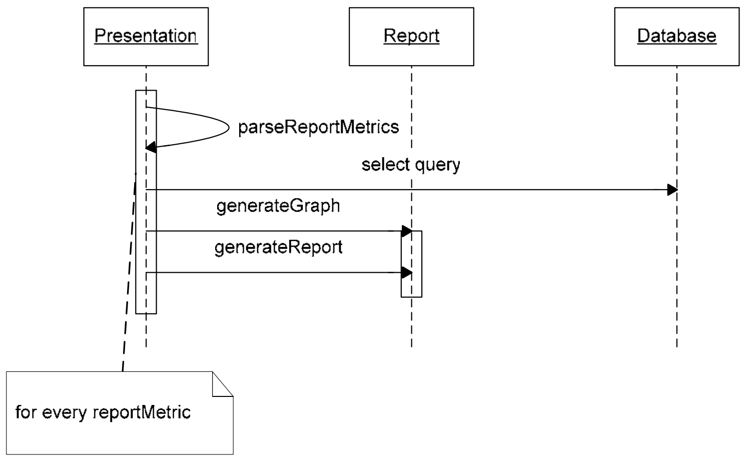
Sequence Diagram: Statistics Gathering

The following diagram illustrates the process of JCVSReport that gathers or updates the statistics.



Sequence Diagram: Report Generation

After JCVSReport completed gathering the statistics and updating the database, it can start generating the report. The following diagram illustrates the process of querying the report statistics from the database, and of generating the report.



References

Please refer to the individual Analysis and Estimation documents for every one of the system components for further details. Here is the list of all the related A&E documents:

- Chapter 2: Data Storage A&E.
- Chapter 3: CVS Parser A&E.
- Chapter 4: Java Parser A&E.
- Chapter 5: Report A&E.

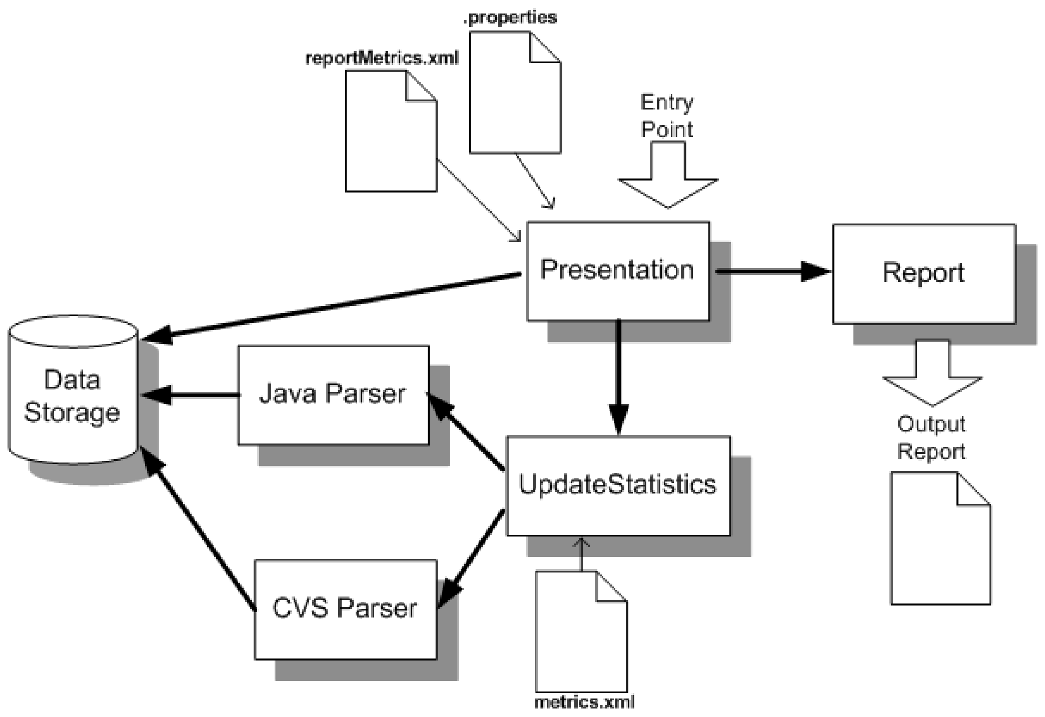
Chapter 2: Data Storage Module

Goal

To allow for the easy storage and retrieval of statistics related to a Java project in CVS

Background

JCVSReport is a lightweight framework for gathering and reporting statistical metrics related to a Java project in CVS. Because the process of gathering these metrics can potentially be quite lengthy for a large project, we want to be able to *store our results on disk* so that they can be retrieved and updated later without rebuilding the complete historical dataset for the entire CVS repository. Consequently, the Java and CVS data gathering modules will store their results directly in this database.



Once our data has been gathered, the presentation component will need to be able to request revisions and metrics that match a specific set of files, versions, authors, dates, classes, and/or functions. We must create a simple way for *specifying and responding to these queries*.

Requirements

Statistics must be stored on disk
We want to store our results on disk so that they can be retrieved and updated later without rebuilding the complete historical dataset for the entire CVS repository.

Flexible queries must be simple and possible
Our presentation layer will need to be able to gather revisions and metrics that match a specific set of files, versions, authors, dates, classes, and/or functions. We must create a simple way for specifying these queries.

Design Decisions

We must choose a simple way of representing our data that will allow us to easily meet our above requirements. Two approaches are possible:

1) **Traditional SQL Database:** Collect all of the statistics ahead of time and store them in tables in a SQL database. These results can be retrieved, collated and compared at any subsequent time using simple SQL queries.

2) **Cache Queries on Disk:** Alternatively, we can gather the statistics upon request by simply forwarding queries directly to the relevant functions in the UpdateStatistics module. As each statistic is collected, we can store the result in a hash-like database which maps each query to its cached results. When the same query is repeated later, we can simply return the cached results.

The cache-based approach has several benefits. For one, it allows us to keep all of our database-related code in a single, simple subroutine. Furthermore, it can be implemented very easily, as demonstrated by its pseudo-code:

```
if (query is cached in database):
      Output result from database
else:
      Ask UpdateStatistics module for query result
      Save query result in database
```

While this seems attractive at first, it actually adds to the software's overall complexity. That is, since our data is not stored, we need to have a defined procedure for retrieving all related statistics from every component. This means that both this module and the statistic collecting modules with be much thicker with interfaces. Furthermore, any new future statistics may require changes in all modules.

While the SQL-based system has slightly more complex implementation, it vastly simplifies our remaining modules. All requests for statistics and metrics can be made in the form of a standard SQL query string to the SQL database. We can retrieve our data in any format or order we want using a simple SQL string. Furthermore, we get to reduce the complexity of all the statistic collection modules. This is because they collect all their statistics on their own, as specified through their own configuration files, and then blast off their results into the central database, again using standard SQL. So all statistic collection modules no longer have to worry about their interfaces, as they all have an extensible SQL communication standard.

However, one of the most notable benefits to an SQL database lies in the power of the SQL query string. It allows one to query new statistics using a simple syntax. While we could implement this through our own proprietary system, SQL databases have already solved this complexity for us. Let us say for example that we have a query string which retrieves the number of lines of code by a certain author. If we wanted to add the extra constraint that we were only interested in the lines of code added between a certain date, we could do so by adding that constraint to the standard SQL query string. But if we weren't using SQL, we'd have to write more complex code. By this point, it should be clear that an SQL database should be the central part of our database.

Assumptions and Dependencies

For the approach outlined above to work, it is assumed that some other process will call the statistic collection modules on a regular basis. Since all metric and statistic queries are to an SQL database directly, the returned results can only be as recent as the last database update. Fortunately, the Presentation module always calls the updateStatistics module before sending queries to the database; therefore we can safely assume that the database is up to date.

Database Schema

Up to this point we have talked about an SQL database in the general sense, and given little details on which database we will use.

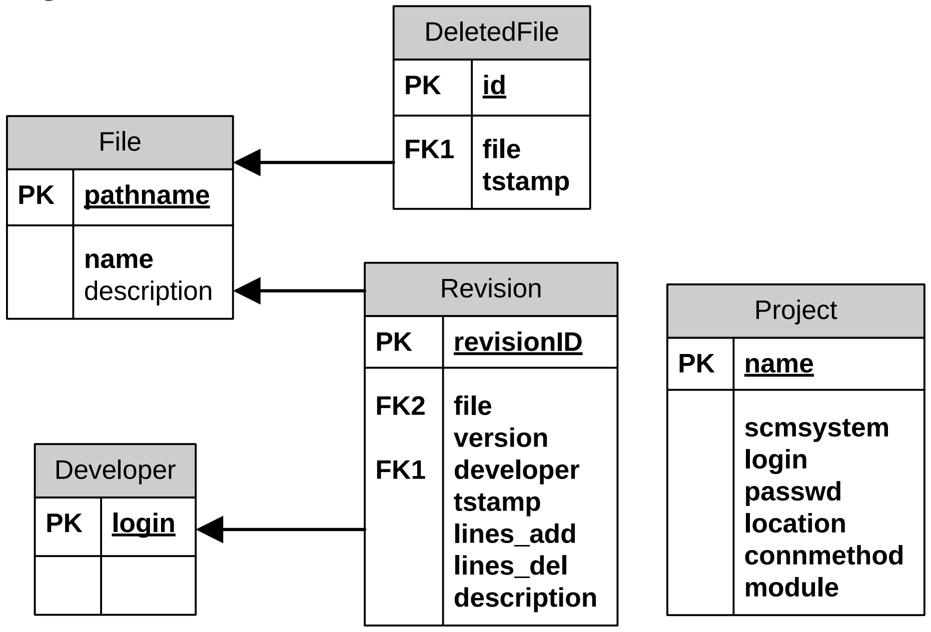
For this project, we will make use of the open-source Bloof package. Our reasons for doing so are as follows:

1) Bloof has a rich set of functionality for retrieving data directly from CVS repositories. This means that it implements a huge subset of our required functionality with respect to dealing with CVS databases.

2) Bloof stores this information directly to its McKoi SQL database. This McKoi database can be updated, extended, and expanded so as to contain all future metrics.

Bloof Schema

In order to better understand Bloof, we constructed a diagram of the Bloof database. This diagram is shown below.



The Project Table

Each CVS repository monitored by Bloof has an entry in the Bloof **Project** table:

- scmsystem: The type of repository (typically CVS)
- login: Our login name to allow us to access the repository
- passwd: Our password to allow us to access the repository
- location: The location of the CVS repository
- module: The module we would like to monitor in that CVS repository

The Developer Table

Each developer in each project has an entry in the Bloof **Developer** table:

- login: The developer's login name

The File Table

Each file in each project has an entry in the Bloof **File** table:

- pathname: The full path (including the directory and filename) for the file
- name: The filename for the file
- description: A description of the file if a user has created one using the cvs annotate command (optional)

The Revision Table

Each version of each file has an entry in the Bloof **Revision** table:

• revisionID:	An autogenerated unique identifier for this table entry
• file:	The full path (including the directory and filename) for the file. (*Refers to an entry in the File table.)*
• version:	The version number of this revision of this file
• developer:	The developer who committed this revision (*Refers to an entry in the Developer table.)*
• tstamp:	The time and date of the commit
• lines_add:	The number of lines added in this commit
• lines_del:	The number of lines deleted in this commit
• description:	The message the developer attached to describe this commit

The DeletedFile Table

When a file is deleted from the repository, an entry is added in the Bloof **DeletedFile** table:

• id:	An autogenerated unique identifier for this table entry
• file:	The full path (including the directory and filename) for the file (*Refers to an entry in the File table)*
• tstamp:	The time and date that the file was deleted

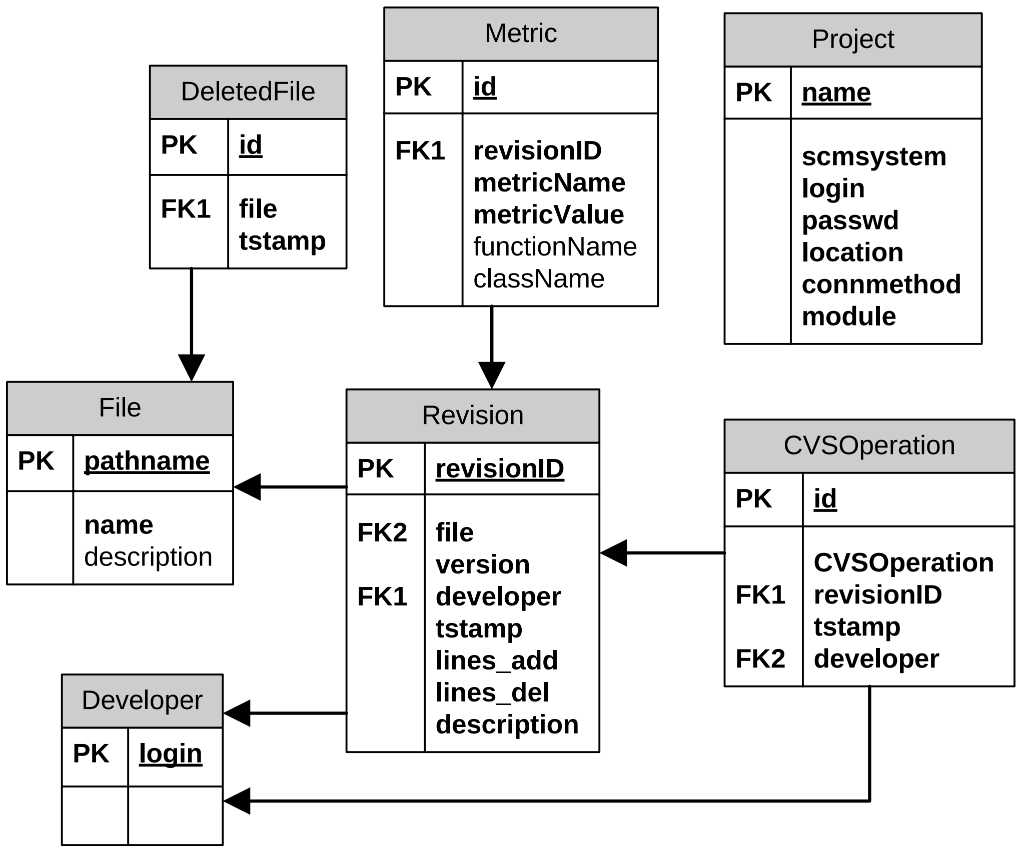
Storing CVS and Java-related information

In addition to the data provided by Bloof, our application will need to store two more types of information:

- Entries from the history file
- Metrics related to specific revisions of Java files in the repository

(For more information on the history file and on the Java-related metrics, please see the A&E documents for the Java and CVS Parsers.)

To meet these additional requirements, we have extended the Bloof database as shown below:



The **Project**, **Revision**, **Developer**, **File** and **DeletedFile** tables are identical to the related Bloof tables. We define two new tables: **CVSOperation** and **Metric.**

The CVSOperation Table

Each entry in the CVS history file has an entry in the **CVSOperation** table:

• id:	An autogenerated unique identifier for this table entry
• CVSOperation:	The type of CVS operation being performed (E.g. update, checkout, commit, etc.)
• revisionID:	Refers to an entry in the **Revision** table
• tstamp:	The time and date of the operation
• developer:	The developer who performed the operation (*Refers to an entry in the Developer table)*

The Metric Table

Each metric gathered by our module for updating statistics has an entry in the Metric table.

• id:	An autogenerated unique identifier for this table entry
• revisionID:	Refers to an entry in the **Revision** table

• metricName:	The name of the metric (e.g. lines of code)
• metricValue:	The gathered value (E.g. 100 lines of code)
• functionName:	If this metric is associated with an individual Java function, the name of the function. (Optional)
• className:	If this metric is associated with an individual Java class, the name of the class. (Optional)

An Extensible Schema

Because developers will often need to add new metrics, we have created the above metric table in a way that ensures that it is not specific to any one set of metrics. Any metric that applies to a specific version of a specific file can be stored in the Metric table with an appropriate metricName and metricValue.

Usability Shortcuts

Based on our use cases (below), we have determined that it is very common that the presentation module will want to request metrics that apply only to a specific file or a specific date range. As such, we have decided to construct a ‘view’ constructed from the natural join of the Metric and Revision tables. We labeled this view ‘MetricRevision.’ In McKoi databases, views are read-only; therefore this view should only be used in SELECT statements.

Use Cases

To better understand this system, we will go through two examples. The first will be from the viewpoint of the modules which request certain metrics and statistics. The second example shows how a statistics collection and updating module such as the Java Parser updates the actual database.

Requesting Statistics

Let us assume that the presentation module needs to collect an arbitrary metric, say in this case ‘linesOfCode’, from a particular file. The presentation module must therefore construct the following query:

```
SELECT date, metricValue FROM MetricRevision WHERE MetricName =
      ‘linesOfCode’ AND FILE='path/aFile.java’
```

In the above query, a table will be returned with two columns, one for the *date* and one for the *metricValue*, which is literally the result of the query. These results have the *constraints* that the name of the Metric (a column in the database table), must be *linesOfCode*, and that the file must be *path/aFile.java*. Furthermore, these results are obtained from the *MetricViewTable*, our databases main table of statistics.

Let us say that they now decided they only wanted the results for 2004. They could form a new query string with new constraints as follows:

```
SELECT date, metricValue FROM MetricRevision WHERE MetricName =
      ‘linesOfCode’ AND FILE='path/aFile.java’ AND date IS BETWEEN
       DATE(‘01/01/2004’) AND DATE(‘12/31/2004’)
```

In Java, the same query can be expressed as follows.

```
ResultSet results =
      db.query("SELECT date, metricValue FROM MetricRevision WHERE
                MetricName = ‘linesOfCode’ AND date IS BETWEEN
                DATE('01/01/2004') AND DATE('12/31/2004')")
```

Here, the db.query() is Java’s way of querying the database. It returns a `ResultSet` object, which allows one to easily iterate over the returned data. How these results are used is up to the rest of the program. One possibility is to just call a `DisplayStats` library and generate a pie graph directly from the results

```
DisplayStats.GenerateGraph(“PieGraph”, results, title, etc);
```

Storing Results in the Database

Let us say that the java parser just finished running, and that it wants to update the database with its newly collected statistics. To do so, it will need to follow the same steps as above, except this time with an `UPDATE` query string. Each query will need to specify which table to update, and what values to insert. Therefore it will have to perform multiple queries for different types of data. For our purposes, we will demonstrate a single update for the ‘*numberOfMethods*’ metric.

```
INSERT INTO Metric (revisionID, metricName, metricValue,
      functionName, className) VALUES
(1,’numberOfMethods’,3,NULL,’myClassesName’).
```

Or

```
INSERT INTO Metric (revisionID, metricName, metricValue) VALUES
(1,2,3).
```

In the above queries, *Metric* refers to the table of metrics. The first set of parentheses contain the columns to be updated, and the second set the values assigned to those columns. The last two columns, *functionName* and *className* are optional, in that all statistics do not necessarily have any use for them. In this case, NULL can be passed in the second set of parentheses, or the second syntax can be used.

Tasks

- Create SQL database schema
  - …for table for storing CVS history (1 hour)
  - …for table for storing Java-related metrics (1 hour)
- Replace Bloof schema with updated schema (2 hours)
- Create simple example in Java for sending SELECT, UPDATE and INSERT queries to the Bloof/McKoi database. Check this example in to CVS repository, and explain it to the rest of the team so as to make their work with the database easier (4 hours)

Task Matrix

DIFFICULTY	**Hard**			
	Moderate			• Create example Java file
	Easy			• Create SQL database schemas • Replace Bloof schema
		Low	**Medium**	**High**
		IMPORTANCE		

Schedule

Since the database component is the central information hub of JCVSReport, it is difficult to write the other components without having access to the database. Therefore the deadline for the database tasks will have to be very soon. As such, we plan to finish all three tasks by Wednesday, February 9.

Resources

Bloof:	http://bloof.sourceforge.net
Mckoi SQL database:	http://www.mckoi.com/database/

Chapter 3: CVS Parser Component

Goal

Our goal is to implement a component that will collect CVS metadata for a project maintained in a CVS repository.

Abstract

The CVS repository stores metadata about the projects that it maintains. More specifically, it stores the history of operations performed by the developers on the project repository. Moreover, for every commit operation, the CVS stores information such as the developer's comment, the files affected by the commit, the number of lines added/deleted, the date and the revision. The CVS metadata, thus, can be a major contributor to analyzing and tracking the progress of each project. The CVS Parser component will access the CVS metadata for a given project, will parse it and pass it on to the Data Storage component (which in our case will be a Relational Database). This document outlines the approach that we will take in implementing the CVS Parser component, and provides a tentative timeline for its implementation.

Background

CVS Metadata

CVS stores metadata regarding the change history of the repository. CVS metadata consists of information such as what files have changed, when, how, and by whom. There exist several ways for viewing CVS metadata. The two mechanisms we will use in our system are CVS log and CVS history.

One can generate a CVS log file by issuing the 'cvs log' command. The output of 'cvs log' contains the following information for every file:

- The location of the RCS file.
- The head revision (the latest revision on the trunk).
- All symbolic names (tags).

It also contains the following information for every revision:

- The revision number.
- The time (displayed in Coordinated Universal Time).
- The author.
- The number of lines added/deleted.
- The log message.

Here is a sample output of 'cvs log':

```
RCS file:
/cvsroot/checkstyle/checkstyle/src/checkstyle/com/puppycrawl/tools/checkstyle/A
ttic/ClassResolver.java,v
Working file: src/checkstyle/com/puppycrawl/tools/checkstyle/ClassResolver.java
head: 1.2
branch:
locks: strict
access list:
symbolic names:
       v2-branch: 1.2.0.2
       release2_4: 1.2
keyword substitution: kv
total revisions: 2;  selected revisions: 2
description:
----------------------------
revision 1.2
date: 2002/06/14 13:52:20;  author: oburn;  state: Exp;  lines: +0 -1
Removed an extra debugging statement
----------------------------
revision 1.1
date: 2002/06/14 13:23:18;  author: oburn;  state: Exp;
First cut at a class to resolve class names. It has some limitations (inner
classes), but I think it is a good enough start. Still need to do the unit
tests.
```

The second mechanism we will be looking at is CVS history. CVS history provides a summary of the repository activity, i.e.: it provides the list of all the operations performed on the repository. For every operations, it provides information such as time, revision, author and files affected by the operation. Below is a sample output of the ‘cvs history -e –a’ command (note: ‘-e -a’ means show every kind of event that happened for all users).

```
O 07/25 15:14 +0000 qsmith  myproj =mp=     ~/*
M 07/25 15:16 +0000 qsmith  1.14 hello.c    myproj == ~/mp
U 07/25 15:21 +0000 qsmith  1.14 README.txt myproj == ~/mp
G 07/25 15:21 +0000 qsmith  1.15 hello.c    myproj == ~/mp
A 07/25 15:22 +0000 qsmith  1.1  goodbye.c  myproj == ~/mp
M 07/25 15:23 +0000 qsmith  1.16 hello.c    myproj == ~/mp
M 07/25 15:26 +0000 qsmith  1.17 hello.c    myproj == ~/mp
U 07/25 15:29 +0000 qsmith  1.2  goodbye.c  myproj == ~/mp
G 07/25 15:29 +0000 qsmith  1.18 hello.c    myproj == ~/mp
M 07/25 15:30 +0000 qsmith  1.19 hello.c    myproj == ~/mp
O 07/23 03:45 +0000 jrandom myproj =myproj= ~/src/*
F 07/23 03:48 +0000 jrandom        =myproj= ~/src/*
F 07/23 04:06 +0000 jrandom        =myproj= ~/src/*
M 07/25 15:12 +0000 jrandom 1.13 README.txt myproj == ~/src/myproj
U 07/25 15:17 +0000 jrandom 1.14 hello.c    myproj == ~/src/myproj
M 07/25 15:18 +0000 jrandom 1.14 README.txt myproj == ~/src/myproj
M 07/25 15:18 +0000 jrandom 1.15 hello.c    myproj == ~/src/myproj
U 07/25 15:23 +0000 jrandom 1.1  goodbye.c  myproj == ~/src/myproj
U 07/25 15:23 +0000 jrandom 1.16 hello.c    myproj == ~/src/myproj
U 07/25 15:26 +0000 jrandom 1.1  goodbye.c  myproj == ~/src/myproj
G 07/25 15:26 +0000 jrandom 1.17 hello.c    myproj == ~/src/myproj
M 07/25 15:27 +0000 jrandom 1.18 hello.c    myproj == ~/src/myproj
C 07/25 15:30 +0000 jrandom 1.19 hello.c    myproj == ~/src/myproj
M 07/25 15:31 +0000 jrandom 1.20 hello.c    myproj == ~/src/myproj
M 07/25 16:29 +0000 jrandom 1.3  whatever.c myproj/a-subdir == ~/src/myproj
```

The general format of the history command output is:

```
     CODE DATE USER [REVISION] [FILE] PATH_IN_REPOSITORY
ACTUAL_WORKING_COPY_NAME
```

Requirements for the CVS Parser Component

As the name suggests, the CVS Parser component should be able to parse CVS metadata. The CVS Parser component should meet the following requirements:

- Access the CVS metadata, more specifically data captured by CVS log and CVS history.
- Allow one to collect metadata for the whole lifetime of a given project.
- Allow one to collect only the data that has been changed since the last time the component has been run on this project, without rebuilding historical dataset.
- Store the metadata that has been parsed in the database (the Data Storage component).

Dependencies on other Components

The CVS Parser component has a dependency on the Data Storage component. The CVS Parser component collects the data from CVS metadata and has to communicate to the Data Storage component in order to store the data.

User Stories

The user in the case of the CVS Parser component is its client application/component. The client should be able to interact with the CVS Parser component by invoking methods defined by its API.

User Story
A client component can ask CVS Parser to bring the database of a particular project up to date with its CVS repository. It will make a call to an updateProject() method defined by CVS Parser API. The client component will also provide the CVS parser with access information to the CVS repository as well as the access information for the project database. The CVS Parser will then run a log command and a history command on the given CVS repository, parse the output of these commands and update the project database accordingly.

Potential Solutions

Home Grown Solution

One of the approaches is to implement the CVS Parser component from scratch without making use of any third party solutions. Since CVS metadata is represented in plain text, we could extract the desired data from it with the use of regular expressions.

Disadvantages

The problem with this solution is that implementing a whole component that will be able to parse CVS metadata extracting different type of data is time consuming and takes a significant amount of effort. Moreover, the format of the CVS metadata is well defined and is fixed. There already exist tools that successfully parse CVS metadata and extract information from it. Therefore, re-implementing this functionality is simply unnecessary.

StatCVS and StatCVS-XML

StatCVS is an open source software that takes a CVS log file as an input, extracts relevant information, and generates various tables and charts describing the project development. StatCVS outputs the final report as a series of HTML pages.

Disadvantages

- Does not store the data in a database. This is a real disadvantage because, as mentioned in the earlier in this document, our system should be able to perform incremental updates.
- Not easy to extend/integrate.

Bloof

Bloof is an open source Java based infrastructure for analytical processing of version control data. Bloof uses version control data for analysing the evolution of software projects. Bloof is build for being integrated into other applications, providing a Java API and a scripting interface for access and a XML output format.

Advantages:

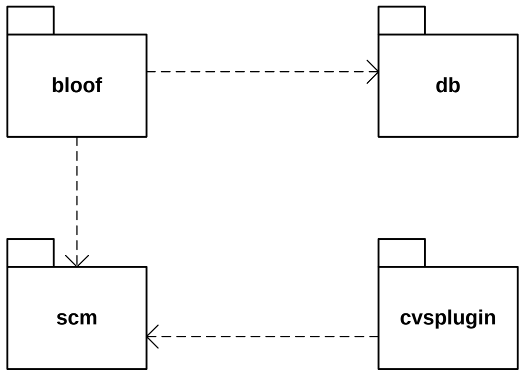
- A third party solution that for the most part follows our requirements will save us time and effort. There is no need to reinvent the wheel.
- The main advantage of Bloof is that it is designed for being integrated into other applications. Bloof has a modular architecture, which makes it easy to integrate and extend.
- If we can come up with good ways to extend Bloof, we can contribute our source code to this open source project.

Disadvantages:

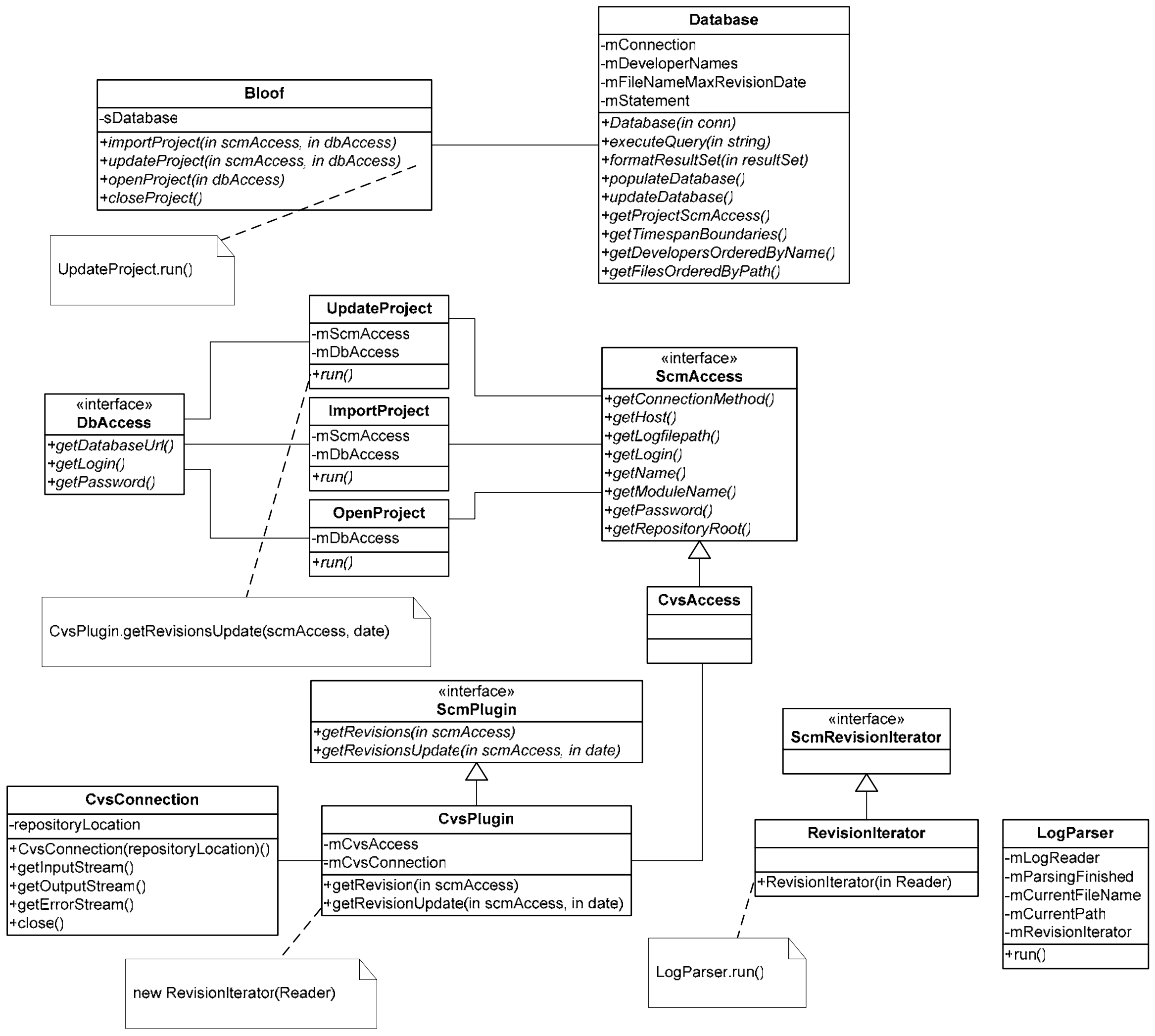
- Bloof is a relatively new project. It has not been used very much yet.
- The documentation that comes with it is not very detailed.

Decision and Justification

We decided that we will use Bloof to implement the CVS Parser component. The Bloof project is conveniently divided into a number of subprojects. The Bloof project consists of a core component and of user tools. The core component already implements a lot of the functionality that we need to fulfill the requirements of the CVS Parser component. More specifically, it can access a CVS repository, run the ‘cvs log’ command on the repository, parse the log metadata and create representation of this data in a database.



Bloof consists of the main `bloof` package, the db package that contains classes that deal with Database connection and information storage and retrieval, the `scm` package that contains classes that deal with access to an source control management system, and finally `cvs` package that contains classes that extend the `scm` package classes and that are tailored specifically towards working with CVS. The following class diagram shows a more detailed overview of the classes and the interaction between them. It seems that we could make a good use of the existing Bloof architecture in other places but the CVS Parser component. For instance, Bloof already provides us with a number of classes that facilitate interaction with a Database. In fact, for our Storage Module we will be using the McKoi database, which is the primary database currently supported by Bloof. Moreover, Bloof provides us with classes that facilitate interaction with CVS.



Features

Most of the functionality that we need is already implemented in the Bloof project.

Input Methods

Bloof supports the following input methods:

- Online access to a CVS repository via pserver, for example using `":pserver:anonymous@cvs.sourceforge.net:/cvsroot/bloof"`.
- Online access to your CVS repository via ssh, rsh.
- Reading cvs log file from the local disk.

If the specified method requires online access to a CVS repository, then Bloof uses the specified method (one of pserver, ssh, ext) to establish a connection and uses CVS protocol to send a request for the log information. Here is an example of a request that asks for CVS log data for a particular module starting from a specified date:

```
Root <repository root>
Argument –d
Argument > <date>
Argument <module name>
rlog \n
```

Updating and Importing a Project

One of the functional requirements for our overall system was that the customer should be able to use the system in the following two ways:

- To monitor and ongoing project, which implies that the system should be ablto incrementally update the data by adding new data to previously collected data.
- To generate the complete history of the project from start to finish.

Bloof currently supports both of these methods. Bloof has two API calls: `importProject(ScmAccess, DbAccess)` and `updateProject(ScmAccess, DbAccess)`. `importProject` collects CVS log information for the entire history of the project, creates a new database for this project and stores the data in a newly created database. `updateProject` on the other hand, collects CVS log information from the date the database was last updated, and adds the new data to the previously collected data in the same database.

Extensible Interface for Dealing with Source Configuration Management Programs

The Bloof project has a package `net.sf.bloof.scm`. This package defines interface classes for dealing with a source configuration management program (a version control program, in other words). It then has another package `net.sf.bloof.scm.cvsplugin`. This package contains classes that implement the interfaces from `net.sf.bloof.scm` and are specifically implemented for dealing with CVS. This architecture ensures that Bloof can be further extended to work with configuration management programs other than CVS, if required. Note that, most likely we will use the classes in these two packages outside of the CVS Parser component, because other system components will need to interact with the CVS as well.

Parsing of the CVS Log

The most convenient things about Bloof is that not only does it parse the CVS log file but it also stores the collected data into a database. The `net.sf.bloof.scm.cvsplugin` package in Bloof contains a `LogParser` class. This defines methods that parse the log data and store it in the database.

Parsing of the CVS History

Unlike the features outlined above, this feature is not currently supported by Bloof. As mentioned in the background section of this document, CVS log does not contain the entire information about the activity in the CVS repository. One cannot find out, for instance, how many update/check out/commit operations have been made to the repository by every developer. Information about the operations performed on the repository appears in the CVS history file. Thus, we would like to extend Bloof so that it parses the CVS history output and stores it in the database, just the same way it parses and stores the CVS log data.

Tasks

Setup the Bloof project and the environment

- Set up Java ™ runtime environment version 1.4 or greater.
- Check out the Bloof project from its CVS repository.
- Import the project to a development environment such as Eclipse.
- Download the following JAR dependencies and add then to the projects Java build path:
  gnu-regexp-1.1.4.jar
  activation-1.0.2.jar
  junit-3.8.1.jar
  mindterm-1.2.1.jar
  nanoxml-2.2.3.jar
  mckoidb-0.94.jar
  blooftestdata-0.1.jar
  jargs-0.3.jar

Run JUnit tests on Bloof

- Run the JUnit tests that come with the project source code and make sure that the project passes them.
- Write and run new JUnit tests if required.

Verify that Bloof is gathering and storing the log data properly

Here is an example of a method that uses Bloof to gather metadata from a CVS repository accessed via pserver and store the data in the database. Verify that the data is gathered and stored properly.

```
    public static void main(String[] args) {
        try {
            // Specify the location and the login information of the CVS
repository of the project to be analysed.
            RepositoryLocation repo = new
RepositoryLocation(":pserver:anonymous@cvs.sourceforge.net:/cvsroot/bloof");
            LoginDetails loginDetails = new LoginDetails("anonymous", "");
            ScmAccess cvs = new CvsAccess(repo, loginDetails, "bloof", "example
cvs module");

            // Specify the login information and the url of the project
database.
            DbAccess dbaccess = new DefaultDbAccess("Default internal
database", "bloof", "bloof", "/bloof/bloofdb");
```

```
            // Import this new project – parse CVS log for the lifetime of the
project and store it in a newly created database.
            Bloof.importProject(cvs, dbaccess);

            // Once the CVS log data is in the database, get the database object
and run an SQL query on it.

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Test the different input methods
Test that Bloof can successfully access CVS repositories via the different methods: pserver, ssh, ext.

Extend Bloof to parse CVS history
This involves implementation of a HistoryParser class, which would be along the lines of the LogParser class that Bloof uses for CVS log parsing purposes.

Write JUnit tests to test the history parser functionality
Write and execute JUnit tests for the new HistoryParser class.

Task Matrix

DIFFICULTY	**Hard**			
	Moderate			- Extend Bloof to parse CVS history - Write JUnit tests to for the history parser
	Easy			- Setup Bloof - Run JUnit Tests on Bloof - Verify that Bloof is gathering and storing the log data properly - Test the different input methods
		Low	**Medium**	**High**
		IMPORTANCE		

Schedule

Task	**Hours**
Setup Bloof	1 hour
Run JUnit Tests on Bloof	2 hours

Verify that Bloof is gathering and storing the log data properly	2 hours
Test the different input methods	2 hours
Extend Bloof to parse CVS history	8 hours
Write and run JUnit tests to for the history parser	2 hours

Because the Report module depends on our CVS Parser module, we want to finish the CVS Parser module as early as possible so as to allow our work on the Report module to begin. We will budget to finish 50% of our tasks for this section by February 9 (9 hours), and to finish the other 50% of our tasks by February 17 (another 8 hours).

Resources

CVS - https://www.cvshome.org
Bloof - http://bloof.sourceforge.net/
StatCVS - http://statcvs.sourceforge.net/

Chapter 4: Java Parser Module

Goal

To collect statistics from Java files in a CVS repository

Background

The Java Parser module is a small but important part of our entire system. Its sole purpose is to collect all metrics and statistics as required from Java text files. The number of lines of code in a file from a specific date or revision, or the number of methods or classes for a particular file, are all examples of valid metrics that one may wish to retrieve.

The module will be called with a path to a file as well as its revision ID. From there it will collect all statistics specified in its configuration file, and then update the Database Storage Module with the newly collected statistics. There are many approaches to collecting this data, as described below.

Potential Approaches to Data Collection

1) One could write customized Java parsers combined with regular expressions to collect all statistics. While this does allow the greatest amount of customization, it would take a considerable amount of time to implement each feature.

2) Another approach would be to use existing libraries to collect many of our statistics. This requires the least amount of time to implement, but also offers the least flexibility. For example, expansion for new statistics may not be possible.

3) As a middle approach, we could make use of certain utility software such as PMD. PMD takes static Java source code, and generates a parse tree of it. This allows one to write small pieces of software to flag certain parts of the tree traversal in Java or XPath. However, the kinds of statistics that can be generated are only those that are related to tree traversal of an abstract tree. So other non-traversing statistics such as the number of lines of code, could not be collected with this approach.

4) We could take the Abstract Syntax Tree that PMD uses, and store the entire tree in a database, for each revision. From there, simply counting the number of abstract tokens such as `UnmodifiedClassDeclaration` (for counting the number of classes), and `MethodDeclarator` (for counting the number of methods). The disadvantage to this is that the database would get somewhat

larger. That is, it takes *much* more storage to contain an entire parse trees (or even their subsets) than it does to store summaries of their individual tokens. The obvious benefit is that we retain a lot more data, and can do much more advanced SQL queries on that data.

Design Decisions

In order to decide on which approach to use, we must first recognize what kinds of statistics are expected to be needed by the users of our software. A short preliminary list is as follows:

A) Simple Counts of lines of code, comments, test code, etc.
B) Counts of the number of methods, classes, unit tests, etc.

All metrics in group A seem simple enough that we just need to count the number of matches to a simple regular expression, which can be easily written in Java. So for group A, any other approach seems to be a bit of overkill. We will write a Metric class called `RegExpLineCounter` to handle the Group A metrics and count the number of lines that match a regular expression.

Group B statistics are more complicated. It would be much more difficult to write a regular expression for these types of metrics. At first it seems the PMD approach is probably best. But after further examination, we see that PMD relies on a Java parser to produce Abstract Syntax Trees (AST), which contain all the information we need. Specifically, the complexities of the code are already removed by the AST generator, and leave us with easily recognizable tokens. Both `UnmodifiedClassDeclaration` and `MethodDeclarator` are examples.

So for group B, we are left with two good choices.

1) Count the number of occurrences of every token type (such as `MethodDeclarator`) that we are interested in, and store their totals in the SQL database. Such an approach allows quick queries into our database.

2) Store the entire parse tree (or subsets of it) in the database, and use standard SQL queries to do the counting for us.

While 2) does seem like an attractive option for its expressive power, it is overkill. Any user requirements found thus far do not need this expressive power. The computational costs for performing these queries would be much higher, and as such this approach is not advised.

Therefore the implementation should use a standard Java parser to generate AST trees which remove complexities. From there, the trees can be traversed and a tally can be made of the number of occurrences of all tokens encountered such as

`MethodDeclarator.` That is, all matches should increment a count of a state variable representing this token-type. At the end of the parse, the module will update the data storage component with a subset of its collected statistics, comprising of those specified in the Java Parsers supplied configuration file. This becomes a huge benefit because all metrics can be found in a single pass.

We will call our class for handling group B metrics `JavaASTMatcher`.

Dependencies and Risk Factors

Since all group B metrics rely on an abstract syntax tree, the Java Parser module is dependent upon the module which produces these trees. For example, our code would no longer recognize the number of methods in a file if a new version of our AST generator was released which renamed `MethodDeclarator` to `ProcedureDeclarator`. Fortunately this issue could be easily resolved by simple modifications to the configuration file, in a single place. Specifically, the '*NumberOfMethods*' metric which is set to match the `MethodDeclarator` in its configuration file, could be updated to match `ProcedureDeclarator`. Furthermore, we do not expect the AST generator to change.

Matching Java code is more complex than matching a simple syntax tree. The Java spec is not simple, and it is often true that many Java compilers accept code that doesn't exactly match the specification. We are confident that this will not be much of a hindrance however, so long as we carefully follow the Java spec and test against a wide variety of code from many different authors.

As a final note, our statistics will only be collected if the file can be properly compiled. If it can't be compiled, then that means no AST tree can be generated. However, one would not expect someone to submit a file to a repository which does not build. Regardless, we recognize this as a limitation to our system.

Use Cases

We will now describe a typical user scenario to better understand how this module fits into the overall system.

When a user of the system wants to retrieve a certain statistic, they retrieve it from a database. Therefore they do not call our module directly. However, when it comes time to generate a new batch of statistics (due to a nightly system update, for example), the Java Parser module will be called to generate new statistics, as a separate program from the command line.

For the module to begin work, it must be provided a file to collect statistics on. It should also be told what revision this file is, so that the Java Parser can update the appropriate sections of the database once its statistics collection is complete.

Note that the parser is not told what statistics to collect on the command line. The actual statistics to be collected from source files are specified in a configuration file, as described in the design document. This is because most of its statistics can be collected in one or two passes through the specified file. For example, as it goes through each line of the java files AST, it reads in a token. For each token X encountered, a counter located at HashMap(X) is incremented. At the end of this pass, whatever statistics are requested in the configuration file, are blasted off to the database. That is, if the config file contains only the two metrics `MethodDeclarator` and `UnmodifiedClassDeclaration`, then at the end of the file, The database will be updated with HashMap(MethodDeclarator), and HashMap(UnmodifiedClassDeclaration), with the constraint of the supplied file ID, and file revision. The format of this configuration file is left to the design document.

Example Java Syntax Tree

Below we show an example of a Java Source Code file and its resultant syntax tree.

Application:
```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
    }
}
```

Syntax Tree:

```
CompilationUnit
 TypeDeclaration
  ClassDeclaration:(package private)
   UnmodifiedClassDeclaration(HelloWorldApp)
    ClassBody
     ClassBodyDeclaration
      MethodDeclaration:(public)(static)
       ResultType
       MethodDeclarator(main)
        FormalParameters
         FormalParameter:(package private)
          Type
           Name:String
          VariableDeclaratorId(args)
       Block
        BlockStatement
         Statement
```

```
StatementExpression
 PrimaryExpression
  PrimaryPrefix
   Name:System.out.println
  PrimarySuffix
   Arguments
    ArgumentList
     Expression
      PrimaryExpression
       PrimaryPrefix
        Literal:"Hello World!"
```

Tasks

To gather our metrics from a Java Syntax tree, the following tasks must be completed:

1) Manually collect statistics on numerous Java files from varying sources so as to have a basis to compare the performance of our software.
2) Set up the PMD Abstract Syntax Tree parser and test it on a simple example
3) Implement the `JavaASTMatcher` class
4) Implement the `RegExpLineCounter` class
5) Test that our Java Parser module must be shown to generate proper Group A and B type statistics on a wide variety of files, based on the manually collected statistics above.
6) The config file method must be tested so as to make sure that the program runs in all cases. For example, if there are no occurrences of a particular statistic.

The following pseudo-code describes the JavaASTMatcher class.

```
HashMap hash = new HashMap();

For each token in AST {
  hash.set(token.getName(), hash.get(token.getName())+1);
}

// After we reach the end of the file:
For each desired node_name listed in the config file {
  Int result = hash.get(node_name);

  // Update the database with the statistic, for the
  // file we are collecting statistics on, its revision,
  // and the result.
  InsertIntoDB(node_name, filename, revision, result)
}
```

The following Pseudo-code describes the RegExpLineCounter class.

- Read the file from CVS into memory into a variable called “file_contents”

- Replace all instances of the specified regular expression in “file_contents” with the empty string
- Count how many lines are in “file_contents” and compare it to the number of lines in the original file to find out how many lines were deleted by the specified regular expression

Time Estimates

- Install and examine the AST generator from PMD: 2 hours
- Implement JavaASTMatcher:
  - Implement data collection: 4 hours
  - Implement inserting results into database: 2 hours
- Implement RELineCounter:
  - Implement data collection: 4 hours
  - Implement inserting results into database: 2 hours
- Collect manual statistics from a few files: 4 hours
- Compare manual to automatically gathered statistics to ensure accuracy: 2 hours
- Create user documentation which explains how to use the aforementioned statistics

Task Matrix

		Importance		
		Low	**Medium**	**High**
Difficulty	**High**			
	Medium		• Collect Manual Statistics	• Implement data collection for JavaASTMatcher • Implement data collection for RELineCounter
	Low		• Test Program on Manual Stats	• Install AST Parser • Implement inserting JavaASTMatcher results into database • Implement inserting RELineCounter results into database

Schedule

Because the Report module depends on our Java Parser module, we want to finish the Java Parser module as early as possible so as to allow our work on the presentation module to begin. We will budget to finish 50% of our tasks for this section by February 9 (10 hours), and to finish the other 50% of our tasks by February 17 (another 10 hours).

Chapter 5: Report Module

Goal

To implement a component that will generate requested statistical HTML reports in the form of charts, graphs, tables, etc.

Background

The report component should prepare HTML reports tailored to specific data, requested by its users. The specified data can be any valid data points that can be represented in the graph or a table. In our case the data points will be statistics about CVS metadata gathered from the project's cvs repository or any statistics about the project's source files that are of interest to the user. The reports generated by this component will be used to make observations about the progress of the project, and thus they must be easy to read and comprehend.

Functionality

The main purpose of this module is to present collected statistics relevant to the software's users, in an easy to understand, graphical form. To facilitate in this task, this module must be easily customizable so as to allow users to specify which statistics are to be presented, as well as the format of how these charts are displayed. For example, perhaps a statistic is better represented by a pie chart, than a histogram. Other style information should also be customizable.

Requirements

To allow this functionality, the following requirements are to be met:

- Options for configuring display features, such as the title of the report, its graphs, their legends, etc, must be provided.
- The user must be able to specify where the generated HTML reports are to be placed.
- The format and style of all charts (such as a pie chart, histogram, etc.) must be configurable.
- Each report may contain different statistics, and as such it must be easy for a user to change data is sourced.

Dependencies on other Components

The Report component depends on the Presentation component of the system. This is because it is the Presentation component that passes this component its required data for document generation.

User Stories

The user of this component will be the Presentation component of our system. The Presentation component will ask the Report to generate an HTML report with specific graphs and title.

User Story 1: The Presentation component needs to generate a report to compare the number of lines of code for two projects: say, project 1 and project 2. The generated report should have the title "Comparisons of Lines of Code - Project1 vs. Project2 ", and should convey this data in the form of a simple line graph. The HTML report is to be outputted to a directory called "/home/boss/reports" and the file name should be "ComparisonP1vsP2.html"

User Story 2: The Presentation component requests a report containing the number of commits and lines of code in relation to the user BOB. The report is further constrained in that its statistics must only present data between January 1, 2004 to January 1, 2005. The title of the report should be "Progress of Bob" and its graphs titles should be "Commits" & "Lines of Code Added". Line graphs are the preferred method of visualization. The directory where the reports will be outputted should be called "/home/boss/reports/bob/" and file should be named "Bob's Progress.html".

Potential Solutions

Approach #1 – Implement this component in Java

This component can be written in Java and can use any free software for generating charts.

Some of the free graphing software packages are:

- **PtPlot2D:** This package is a data plotting and histogram tool implemented in Java. It can be used as a standalone applet or application, and can also be embedded in a larger piece of software.

- **JChart2d:** A java programming library for visualizing quantitative data using two-dimensional charts.

- **JFreeChart:** An open source Java class library for generating many different types of charts. Support is included for pie charts, bar charts, line charts, scatter plots, time series charts, candlestick charts, high-low-open-close charts and more. Generated graphs can be exported to PNG, JPEG, PDF, SVG and HTML image maps.

Advantages of JFreeChart:

JFreeChart is a widely used software solution. As such, it is proven to be well written, tested and documented. This is always an important consideration if one wants to use another software package to help implement their own. Another major advantage lies in its easy of use for feature work and application integration.

For example, there are only 2 steps to create a chart using JFreeChart:

1) Create a dataset object containing the statistics that we want to be displayed.
2) Create a JFreeChart object by specifying the charts type, and passing it the dataset object created in the first step..

Thus, it seems that one of the better approaches to help develop this module, involves the integration of JFreeChart into a Java based solution. That is, since a large part of our application (that is, its other components), are to be implemented in Java, integration with JFreeChart should be simple and straightforward. Furthermore, since the application is already well developed, using it will save us a great deal of development time.

Approach 2: XML statistics + Bloof-Websuite

Bloof-Websuite is a user application that generates a suite of webpage's from a set of Bloof XML result files.

The following is a format of the xml file:

```
XML file format:
-------------------------------------------------------------------------------------
<?xml version="1.0" encoding="UTF-8"?>
   <result type="timeline">
      <parameter files="/junit"/>
      <data>
```

```
         <datapoint time="2000-12-03 9:36:14" value="0.0"/>
         <datapoint time="2000-12-03 9:36:15" value="0.0"/>
         <datapoint time="2000-12-03 9:36:17" value="0.0"/>
         <datapoint time="2000-12-03 9:36:22" value="0.0"/>
          ....
      </data>
   </result>
```

Advantages:

Bloof-websuite implements most of the required functionality for the Report component. This makes it appear that using it would require less work on our part.

Furthermore, since our CVS parser will be implemented using Bloof, and web-suite is already designed specifically to work with the statistics that are generated by Bloof, this would cut down on our development time even further.

Disadvantages:

Bloof-Websuite is currently in its alpha version, and as such likely has many bugs and unimplemented features. Also, since the package has to parse an XML file with statistics in order to generate reports, then parsing might take substantial amount time, which will reduce the efficiency of the system. Furthermore, extending the Bloof-Websuite to better suit our required features would be quite difficult. This is because doing so involves understanding the packages existing code base. It is also always a possibility that it has a poor architecture, and would be much too difficult to extend.

As a final disadvantage, our Java Parser component is not implemented by Bloof. This means that the Java Parser module would have to be extended to output its statistics to an XML format that is useable by this suite.

Decision and Justification

We decided that we will go with the first approach in which the component will be implemented in Java and will use JFreeChart for producing different types of charts. Implementing this component in Java will ensure easy integration with main module and using JFreeChart chart for graphing functionality will save substantial amount of development time. This approach will make Report component efficient, easy to use and extend.

Features

- Generate graphs with specified graph type, title and graph legends
- Generate comparison graphs that will display two sets of data points with specified graph type, title and graph legend
- Generating Report containing specified graphs
- Output report to the directory specified by the user

Tasks

Setup the project and the environment

- Set up Java TM runtime environment version 1.4 or greater
- Check out the JFreeChart project
- Download the following JAR dependencies and add them to the projects Java build path:
  jfreechart-0.9.21.jar

The features described above will be implemented in the class called **Report**
Below is **API for Report** class.

```
class Report {

    String title;   // title of the report
    String outputDirectory;  // directory where the report will be created
    JFreeChart[] charts;   // charts that will be included in the report

   /**
    * Construct report object
    * @param reportTitle title of the report
    * @param outputDirectory path to directory where report will be created
    */
    public Report(String reportTitle, String outputDirectory);

   /**
    * Add graph to the report document
    * @param dataPoints  data points that should be graphed
    * @param xlabel  label of the x-axis
    * @param ylabel  label of the y-axis
    * @param typeOfGraph type of graph that will be produced
    * @param graphLegend legend of the graph, can be null
    */
    addGraph(ResultSet dataPoints, String xlabel, String, ylabel, String
typeOfGraph, String graphTitle, String graphLegend)
```

```
   /**
    * Add graph to the report document
    * @param dataPoints1  first set of data points that should be graphed
    * @param dataPoints2  second set of data points that should be graphed
    * @param xlabel  label of the x-axis
    * @param ylabel  label of the y-axis
    * @param typeOfGraph type of graph that will be produced
    * @param graphLegend legend of the graph, can be null
    */
    addComparisonGraph(ResultSet dataPoints1, ResultSet dataPoints2, String
xlabel, String ylabel,
                                    String typeOfGraph, String graphTitle, String
graphLegend);

    /**
     * Creates report in the report's output directory
     */
    generateReport();

}
```

<u>Feature1: Generate graph with specified graph type, title and graph legends</u>
must implement addGraph(ResultSet dataPoints, String xlabel, String, ylabel, String typeOfGraph, String graphTitle, String graphLegend)

<u>Feature2: Generate comparison graphs that displaying two sets of data points with specified graph type, title and graph legend</u>
must implement addComparisonGraph(ResultSet dataPoints1, ResultSet dataPoints2, String xlabel, String ylabel,
String typeOfGraph, String graphTitle, String graphLegend);

To implement this methods the following actions have to be performed:
- o Extract data from the dataPoints and create appropriate dataSet object depending on the type of graph that has to be created.
- o Create JFreeChart with the specified features and add it to the array of charts in the report.

Here is sample code for creating XYSeries using JFreeChart

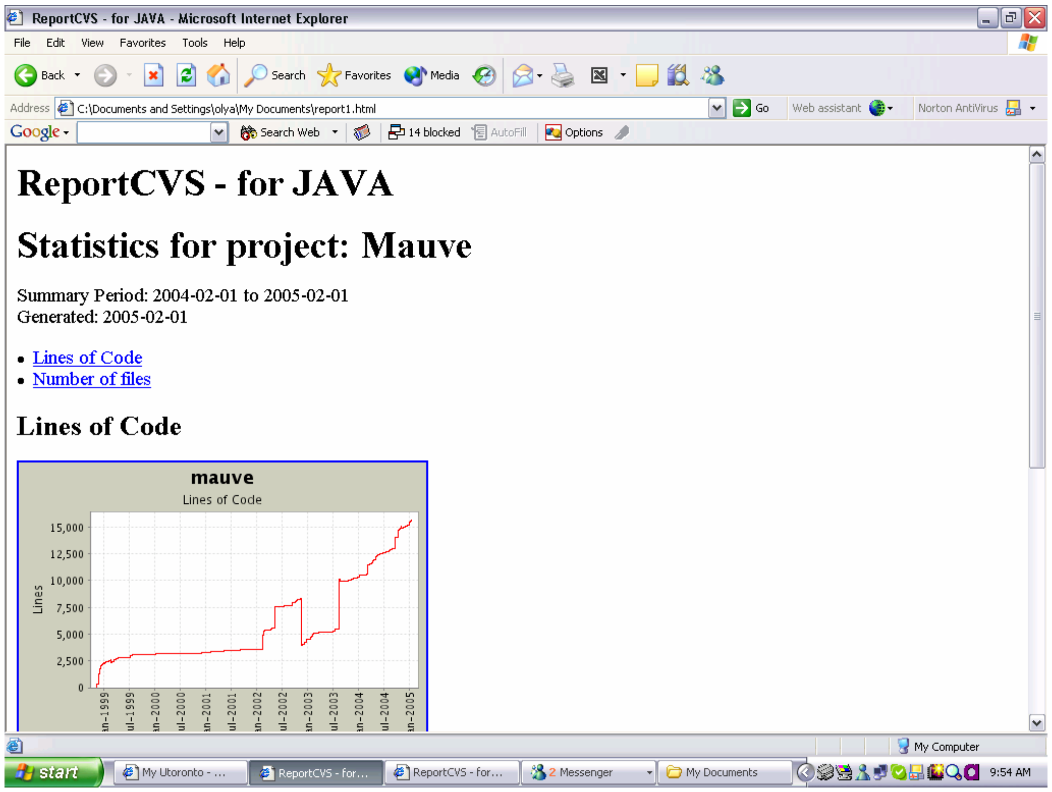
```
XYSeries data = new XYSeries("Title");
data.add(20.0, 10.0);
data.add(40.0, 15.0);
XYDataset dataSet = new XYSeriesCollection(data);

JFreeChart chart = ChartFactory.createAreaXYChart
                    ("Sample XY Chart",    // graph title
                      "dimension1",             // xlabel
                      “dimenstion2",            // ylabel
                      xyDataset,                  // data points
                      true                             // include graph legend
                    );
```

<u>3. Generating Report for specific project</u>

Every report will include “title of the project” and then links to graphs for each metric requested by the user. The graphs will be listed
one below the other, however the links will be on top of the report if the user wants to go right away to the a specific graph. The HTML report will be created using org.w3c.dom Interface Document API.

Below is a sample report for Mauve project. The report was required to contain “Lines of Code” and “Number of files”.
As you can see there is two links at the top available for each graph


ReportCVS - for JAVA - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\Documents and Settings\olya\My Documents\report1.html
Go
mauve
Lines of Code
Lines
15,000
12,500
10,000
7,500
5,000
2,500
0
start
My Computer
9:54 AM

ReportCVS - for JAVA

Statistics for project: Mauve

Summary Period: 2004-02-01 to 2005-02-01
Generated: 2005-02-01

- Lines of Code
- Number of files

Lines of Code

Features Matrix

	Hard			
Difficulty	**Moderate**	Setup the project and the environment Create report including all the graphs in the specified directory	Generating different types of graph with specified features Generating different types of comparison graphs with specified features	
	Easy		Output report to the specified	

			directory by the user	
		High	**Medium**	**Low**
		Importance		

Schedule

Features	**Time (Hours)**	
	Development	**Testing**
Setup the project and the environment	1	1
Generate graph with specified graph type, title and graph legends	5	4
Generate comparison graphs that displaying two sets of data points with specified graph type, title and graph legend	6	3
Generating Report with specified title and graphs	6	3
Output report to the specified directory by the user	2	2
Total time(hours)	20	13

References

www.jfree.org/jfreechart