MODULAR QUALITY ASSURANCE TOOLSET

OPERATIONAL CONCEPT DOCUMENT

CSE681 – SOFTWARE MODELING AND ANALYSIS
JOHN WALTHOUR
MAY 10, 2010

CONTENTS

EXECUTIVE SUMMARY

In a large project, quality assurance can become extremely difficult to manage. Managers want to understand the project status, programmers want to spot defects as they appear, and test personnel want to scrutinize high-risk areas. All of these needs can easily form a cacophony of manual tasks, which Quality Assurance Toolset aims to organize and automate.

ARCHITECTURE

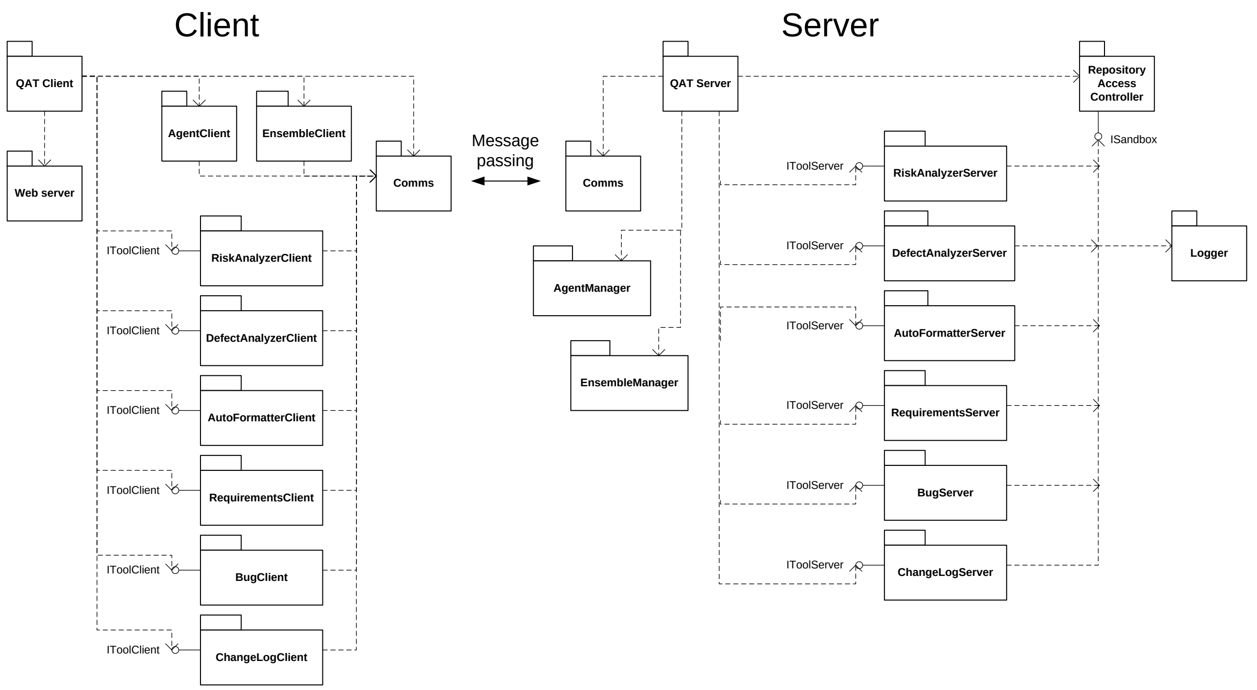

Figure 1 – Overall Architecture

The Quality Assurance Toolset (QAT) is designed to follow a plugin-based architecture, where each tool is plugged into the larger system. A plugin contains two parts – one part plugs into the client, and one part plugs into the server. The client-side plugin is responsible for generating the GUI to present to the user in HTML, and for forming messages representing the user's instructions to send to the server. The server-side plugin is responsible for the tools actions, and interacts with the files on the repository server. Each of these modules is discussed in further detail below.

SERVER SIDE TOOL HOLSTER

CONCEPT

The Server Side Tool Holster hosts the server components of all the tools in the Toolset. It runs them as required by the Client, and allows them controlled access to the filesystem. It also coordinates communications with the Client, allowing the entire toolset to listen on a single socket, and routes received messages to the appropriate tool.

The Tool Holster is designed to use a pluggable architecture to accept a wide variety of code analysis, development and processing tools. Each tool is packaged into a DLL, and implements a common interface that the Holster uses to communicate with it. Each DLL is given a pointer to a sandbox object that facilitates controlled access to the filesystem. This sandbox exposes file access functions including directory listing, file reading, and file writing, but only within the code repository.

USES

- **Easy tool administration –** The Tool Holster creates one unified place from which to configure and manage all the tools in the Quality Assurance Toolset. Administrators no longer need to learn each tool's setup individually, or set up each tool's access rights. The Holster provides a unified configuration scheme, and one access control system common to all tools.

- **Enforcing tool security –** The Tool Holster sandboxes the tools' access to the file system, preventing them from accidentally or intentionally reaching outside of the code repository. This allows administrators to install additional tools with a measure of confidence that the tool will not corrupt operating system files or access other files co-located on the server.

- **Expansion –** This Holster allows for easy development of additional tools by creating a unified pluggable architecture that many tools can operate on. Furthermore, these tools can all be controlled by a single Client, removing the need for a clutter of tools on each user workstation. Third party developers can contribute tools to the toolset, delivered as simple packaged DLLs for installation.

ACTIVITIES

The Holster primarily facilitates communications between tools, the client, and the repository. It also initializes the tools via object factories, and terminates them upon server shutdown. Some messages from the client are expected to trigger analysis of the codebase, in which case the sandbox functions of the Holster will be called by the active tool, and the holster will provide the requested filestreams and listing information.

STRUCTURE

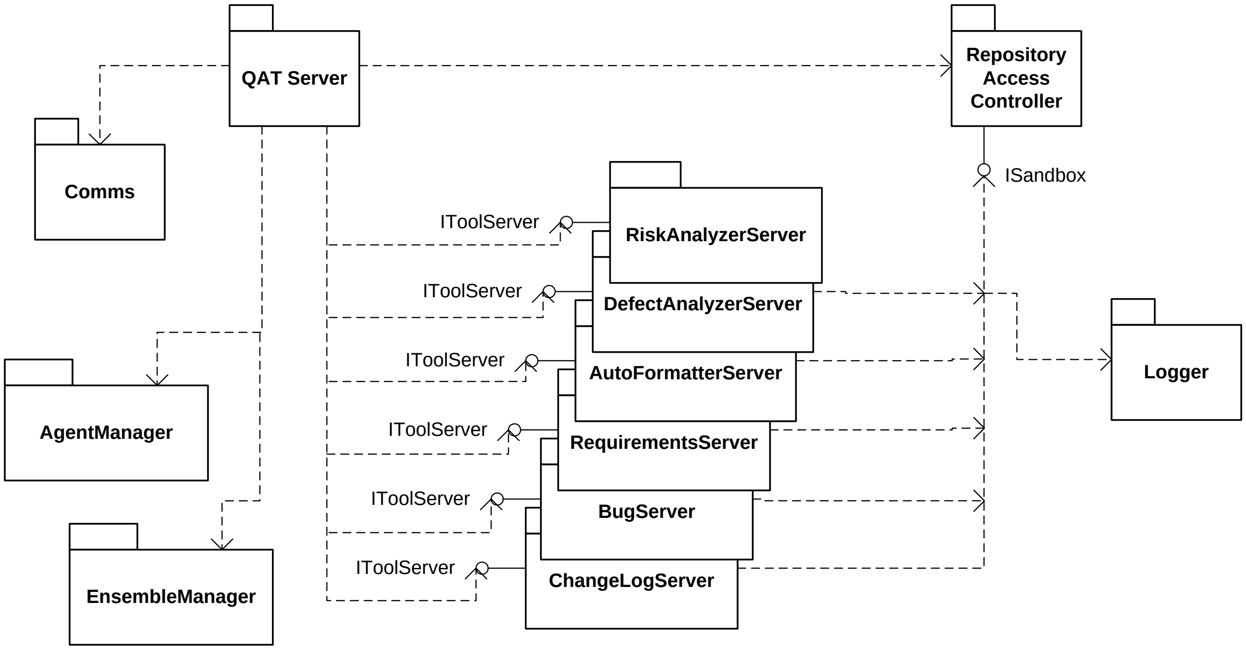

Figure 2 – Package overview for Tool Holster

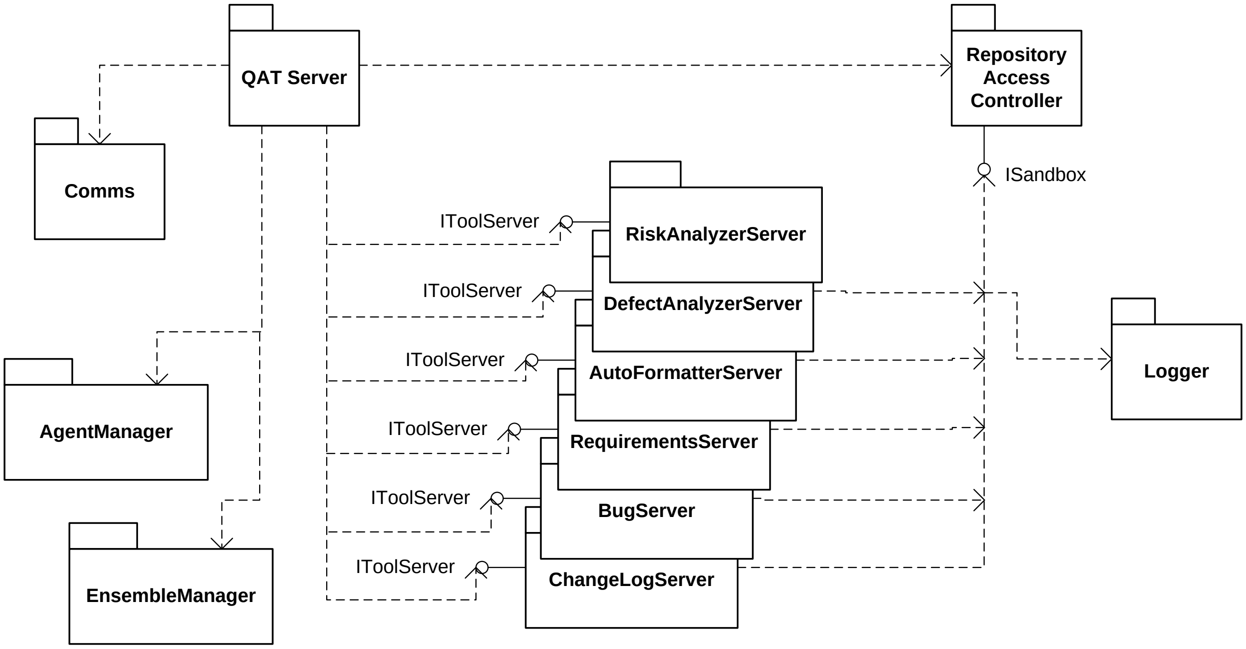

Figure 2 illustrates the overall architecture of the Holster. The specific packages have the following responsibilities:

- **QAT Server –** This is the main executive package, and coordinates the actions of all the other server-side packages. It locates the configuration files for all the tools, and initializes all tools, passing them in as open filestreams. It also communicates with the client via message passing, and routes the messages appropriately.

- **Comms –** This package communicates with the Client. It is responsible for opening a socket, listening on the socket, and firing events to the QAT Server when messages arrive. It is also responsible for sending messages back to the client, and for queuing messages when traffic is too heavy to handle immediately. All messages are encoded in XML for universal ease in translation.

- **Repository Access Controller –** This package is responsible for listing and opening files within the repository. Plugged-in tools access these functions through the ISandbox interface, through which they can request files and file information within the repository. This module controls all tool accesses to the filesystem, and does not permit access outside the configured repository area. Administrators set the repository area through this module's configuration file.

- **Logger –** This package accepts event records from tools, and logs them to a file in XML format. It makes use of queuing to improve performance, and is discussed in more detail later in this document.
- **EnsembleManager –** This package manages descriptions of groups of files within the repository known as Ensembles. It supplies the client with descriptions of the current file structure in the repository and accepts and stores sets of files specified by the user. Ensembles are stored in XML. This package also contains a parsing class so that tools can share the code that reads stored ensembles.
- **AgentManager –** This package both configures and executes small customizable items known as Agents that allow users to personalize the Toolset. Agents are akin to macros, and can be scheduled to run at preconfigured times and run specified tools on specified ensembles. This package is responsible for supplying the client with information about existing agents, as well as accepting information from the client to reconfigure agents, delete agents, or create new agents.

CRITICAL ISSUES

- **Security –** Executing many pluggable tools, some of which may be supplied by a third party, raises a distinct security challenge. This design approaches the issue by funneling all file accesses into one module, which will prevent tools from accessing outside the repository.

- **Modularity –** The entire Holster must be very thoroughly isolated from the tools it runs. This allows for consistency, as well as easing the process of involving third party plugins. For this reason, tools are created using standard object factories, only receive information from the Holster via a tool interface, and only communicate back to the Holster via other interfaces. This isolation keeps the tool developers from needing any knowledge of the inner implementation of the Holster.

CLIENT

CONCEPT

The Client is a web-based thick client that interacts with the user, conveys commands to the server, and relays information about the results to the user. Each tool has a pluggable module in the Client to match its pluggable module on the Server, and all client modules share a common connection back to the server. The Client also serves the web pages to the user, eliminating the need for a separate web server setup.

USES

- **Tool access –** The primary use of the Client is to access one of the quality assurance tools contained in the Toolset. These tools will have user interfaces available through the web interface.

- **Agent control –** Power users may also wish to configure Agents to conduct customized operations at specified times. Normally these agents are directed to operate around ensembles, which are also defined from the Client. See page 20 for more about agents.

- **Ensemble selection –** Some users will wish to mark out areas in the repository to consider as Ensembles – subsets of the larger codebase that require analysis separate from the rest. See page 22 for more about ensembles.

ACTIVITIES

The Client serves as a gateway to the server-side components of the plugin. Its actions include:

- **Communicate user actions to server –** The web interface will receive direction from the user by form submittals, and will convey this information to the client-side component of the plugin. In turn, this component will use the shared Comms module to contact the server, and communicate the user's wishes.

- **Report results to the user –** When the server completes a report, the user will be notified that a report has arrived and is ready for viewing. These reports are sometimes generated by a user, but could be generated by an agent acting on the user's behalf.

STRUCTURE

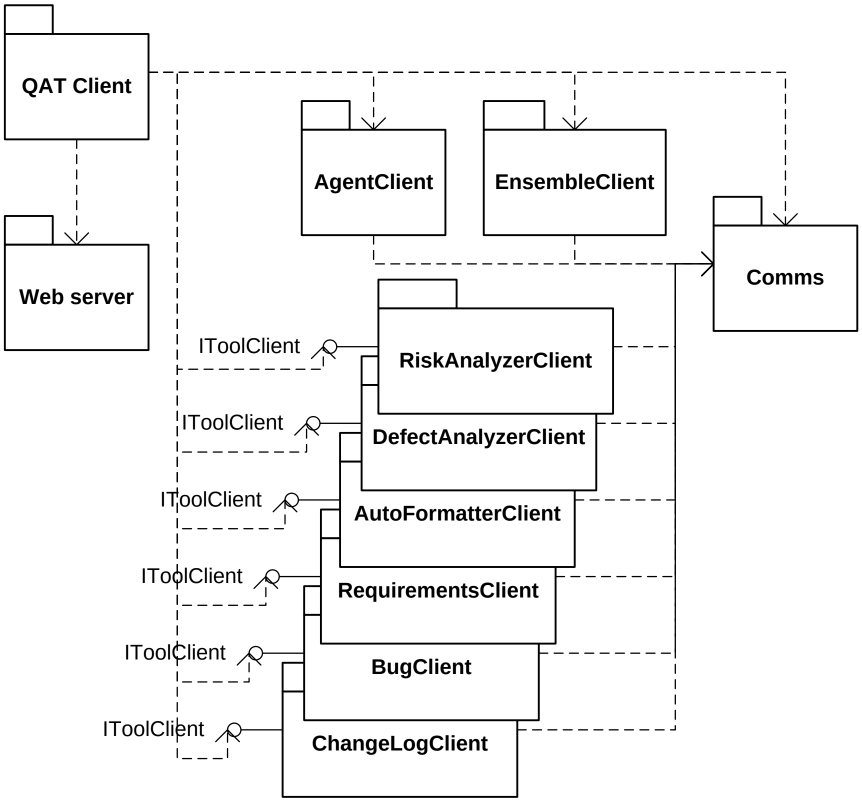

Figure 3 – Package overview for the Client

Figure 3 illustrates the overall architecture of the Client. The specific packages have the following responsibilities:

- **QAT Client –** This is the main executive package, which coordinates the actions of all other packages. Its responsibilities include instantiating the client plugins, establishing communications with the server, and communicating data between the client plugins and the web interface. When reports arrive from the server, this package processes and displays them, rather than the associated tool plugin. This is to enforce uniform display of reports.

- **AgentClient –** This package is responsible for coordinating the creation and editing of Agents on the server.

- **EnsembleClient –** This package is responsible for coordinating the creation and editing of Ensembles on the server.
- **Web Server –** The Web Server component is responsible for final display of content to the user's web browser.
- **Comms –** This package communicates with the Server. It is responsible for opening a socket, sending messages on the socket, and firing events to the QAT Client when reports arrive. It is also responsible for queuing messages when traffic is too heavy to handle immediately. All messages are encoded in XML for universal ease in translation.

CRITICAL ISSUES

- **Usability –** With many diverse tools under one GUI, it becomes vital to organize the options and controls cleanly and effectively. The Client is intended to be a unifying, organizing component, so care must be taken to prevent it from obscuring or impeding the users.
- **Portability –** The more workstations can run the Client, the more useful the tool will be to a wider audience. Cross-platform languages such as Java could be considered, as they afford use on many operating systems. However, a great deal of labor is required to allow a piece of software to use networking between platforms, so Java was discarded as too time consuming. However, this does not absolve the developers from a need to keep the software capable of running on many variants of Windows, including machines in many states of patching and .NET installation, without the added support becoming prohibitive.

RISK ANALYZER

CONCEPT

When a project contains thousands of files, it can be difficult to determine which ones need improvement. The Risk Analyzer discovers files that are a high risk, and deserve more test focus. For example, if a file has a high fan-in and a high fan-out, combined with a bad smell, the file should be subjected to additional testing, and measures taken to reduce its risk.

USES

- **Risk management –** Developers can run the Risk Analyzer when deciding how to allocate their time improving existing code. For example, when desiring to improve stability, a developer could run the tool on his codebase to identify potential weak points, and investigate further to improve high-risk code segments.
- **Test Management –** Testers can run the Risk Analyzer when planning tests, particularly unit tests. High risk code segments can be tested more thoroughly, so that test time is spent efficiently. Extra unit tests can be developed and carried out on risky modules, while more basic tests can be conducted on lower-risk modules. Likewise, full system tests can be arranged to exercise higher-risk modules in more ways, while still conducting basic tests on less risky modules.
- **Code Metrics –** Managers desiring to measure the progress of a project could use input from the Risk Analyzer to enhance their reports. For example, when 100 man-hours are spent on Module A while 300 man-hours are spent on Module B, some explanation may be required. A team lead who can identify Module B as a higher risk module could justify extra budget spent on Module B instead of Module A.

STRUCTURE

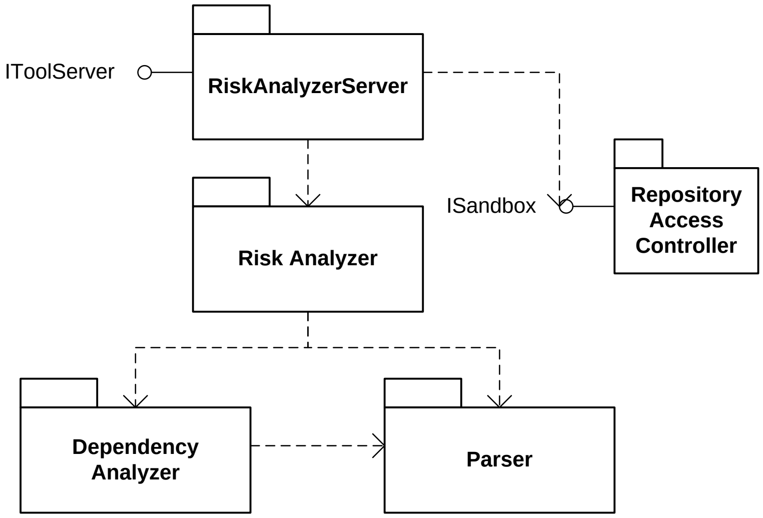

Figure 4 – Package overview for the Risk Analyzer

- **RiskAnalyzerServer –** This package acts as the Risk Analyzer's gateway to the rest of the server. It accepts information from the Holster and the Client through the IToolServer interface, and accesses the repository through the ISandbox interface. It also initiates analysis using the Risk Analyzer package, and reports the results to the Client.

- **Risk Analyzer –** This package contains the core risk analysis logic. It begins by determining the dependency graph of the specified ensemble using the Dependency Analyzer, then computes the smell of each file in the ensemble using the Parser. After another round of computations, each file's risk is reported back to the RiskAnalyzerServer for communication to the client.

- **Dependency Analyzer –** This package uses the Parser to determine the dependency graph of the specified ensemble. It scans through the files to discover all type definitions, then scans through to discover all references to the defined types. This two-pass approach is used to generate a directed graph indicating the dependencies.

- **Parser –** The Parser provides a versatile polymorphic framework with many uses. It scans through files, seeking out patterns of tokens that match specified rules. Each rule is given a list of actions, which will be executed when the parser finds a pattern of tokens that match. Rules and actions all inherit from abstract classes, allowing for a fully polymorphic parser.

CRITICAL ISSUES

- **Performance –** Since the Risk Analyzer is a tool designed to reside primarily on the repository server, it is expected that the body of the tool will execute on a server with a modest amount of computing power, most likely a mult-core machine. To take advantage of this, the Analyzer works in a multi-threaded mode, allowing for much faster analysis of the codebase.

- **Sharing –** Since the tool works on the repository server, there is a very real possibility that a developer could update a file while it is being analyzed. This is a significant concern, as it could potentially invalidate the analysis. The server relies on the local operating system's write locking, and opens each file in read-only mode, so other applications can read from the file under analysis but not write to it.

DEFECT ANALYZER

CONCEPT

The Defect Analyzer's role is to keep toxic code out of the baseline based on various code smells. This bad smelling code can be produced by a variety of things, including hurried design, hurried implementation, and inexperienced programmers. These smells generally highlight code that will impede future development – perceived by most programmers as simply hard to work with. A variety of code types smell bad, including classes with excessively long methods, classes with too many methods, and duplicated code.

USES

- **Highlighting attention-deprived code –** Sometimes code is forgotten when it does not express bugs, depriving it of its deserved refactoring. If an Agent is configured to analyze the codebase for potential defects periodically, these areas can be brought to a team lead's attention, and a developer dispatched to clean this area.

- **Encouraging programmers to write clean code –** Programmers can easily be tempted, or pushed by deadlines, to quickly write code that is good enough, but could become a maintenance nightmare. In a similar vein, sometimes code designed to be a throw-away prototype can easily become part of the codebase without a necessary refactor, resulting in code that is far from ideal. If developers know their team lead is checking their code for bad practices on a regular basis, they will be more likely to keep good code in the codebase.

STRUCTURE

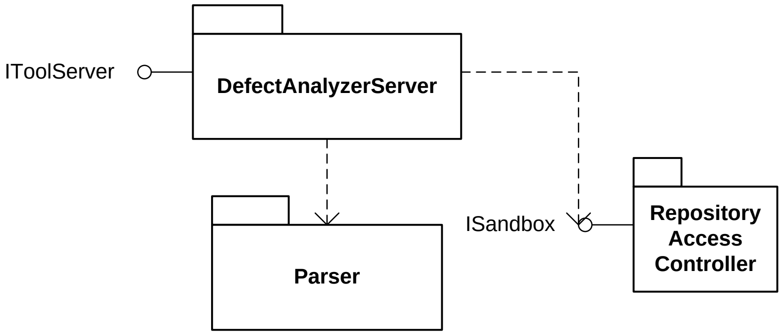

Figure 5 - Package overview for the Defect Analyzer

- **DefectAnalysisServer –** The Defect Analyzer operates in a relatively simple fashion, allowing for most logic to reside in a single package. The main DefectAnalysisServer package sets up the Parser with rules and actions appropriate to detect various code smells. This package then parses specified files, using the actions to note smelly code sections as they are discovered.

- **Parser –** This parser is the same parser as found in the Risk Analyzer. It provides a versatile polymorphic framework with many uses, in this case detecting smelly code. It scans through files, seeking out patterns of tokens that match specified rules. Each rule is given a list of actions, which will be executed when the parser finds a pattern of tokens that match. Rules and actions all inherit from abstract classes, allowing for a fully polymorphic parser.

CRITICAL ISSUES

- **Versatility –** While many smells are indicators of non-ideal programming, some organizations may find them distracting, or disagree with them. For this reason, smells can be disabled in the plugin's configuration file. A possible future expansion to the software would be to attribute a weight to each code smell, allowing managers to apply their own priorities to the analysis.

- **Expandability –** Some developers, managers, or team leads may desire to add additional code smells to the ones that come with the Analyzer. For example, some senior engineers may have discovered a high defect count associated with a certain code pattern on a specific project, and wants to detect use of this code pattern in similar projects. Since smells are based on combinations of rules and actions, more can be added to the toolset by creating more rules and actions. A possible future expansion to the toolset would be to create a scripting language that allows new smells to be defined by advanced users.

TRACKING FACILITIES

During development and maintenance, three types of databases are generally used: A requirements database, an issue tracker, and a change logger. These tools share the same structure and very similar implementation, and are therefore grouped together.

REQUIREMENTS DATABASE

CONCEPT AND USES

The Requirements Database is intended to facilitate vetting of requirements with the customer. Since requirements are something that must be very carefully controlled and understood, they merit having a separate storage area.

CRITICAL ISSUES

- **Access Control –** While most of the Quality Assurance Tools can be safely run by any developer, the Requirements should be more carefully controlled. Companies may choose to give only managers or team leads access to this database, or to designate a developer as the requirements collector. In any case, access should not be granted to all users of the QAT.

ISSUE TRACKER

CONCEPT AND USES

Every project needs some kind of tool to track bugs and shortcomings, and the Issue Tracker is designed to fill this niche. When a user discovers an issue, he records it in the Issue Tracker, which then notifies the lead for that project. The lead then assigns the issue to a developer, who investigates and solves the issue. Once the issue is solved, the developer marks it as solved, and the Issue Tracker notifies the original reporter of the issue. The issue can only be finally closed when the reporter confirms that the issue is truly solved.

The Issue tracker also reports statistics on the quantity and nature of the issues for managers' use. For example, a manager may want to know which part of the code generates the most complaints, and investigate the reason for the extra issues. The manager may also wish to reward the developer who solves the most problems, and the issue tracker would provide a good starting point. Finally, a well-implemented issue tracker can aid Configuration Management staff greatly in discovering what changes were made between specific versions of a specific tool, without diving into the code information stored in the Change Logger.

CRITICAL ISSUES

- **Logical organization –** While it may seem like a trivial task, the issues kept in the Issue Tracker must have enough metadata to keep them organized in a logical fashion. While this data varies somewhat between design houses, the basic important data includes:
    - **Affected project(s)**
    - **Affected requirement(s)**
    - **Reporter**
    - **Assignee**
    - **Links to sub-issues**
    - **Links to commits in Change Logger when fixed**

  If all this information is correctly stored, the issues stored in the Issue Tracker can be easily sorted and searched in a logical manner.

CHANGE LOGGER

CONCEPT AND USES

The Change Logger is used to record, observe, and summarize changes to the codebase. It interfaces with the code repository software to detect changes, and records the commit number, developers commit comment, and the code changes associated with the commit. If the developer specified an issue number in the comment, the log entry will be updated to include a link to this issue number.

This Logger is useful for team leads to quickly understand changes that are made to the codebase as they are made across a wide development team, and for managers to obtain metrics for the project. For example, managers can measure commitment volume get an approximation of effort spent to resolve a particular issue.

CRITICAL ISSUES

- **Interoperability –** This tool's usefulness is inextricably linked to its ability to communicate with the code repository software. For this reason, this plugin must be written with support for a wide variety of repository servers, such as SVN, MKS, and Perforce.

STRUCTURE

The three tracking tools share a common structure, illustrated in Figure 6:

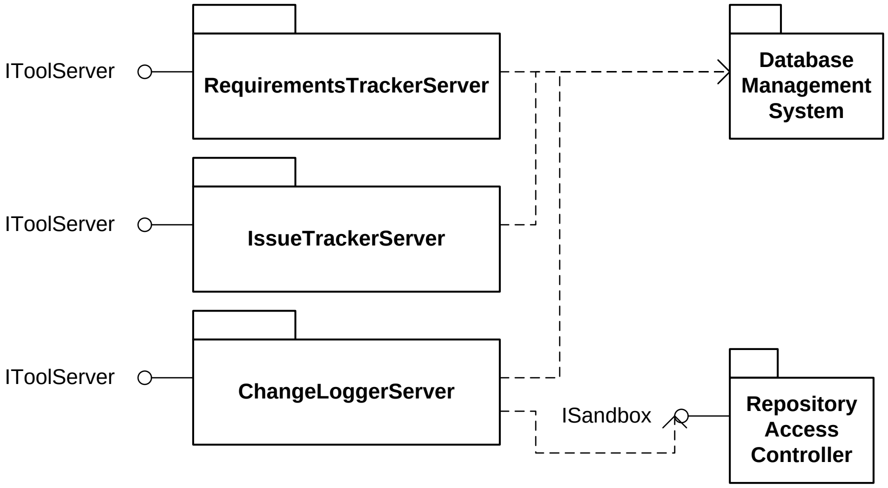
IToolServer
RequirementsTrackerServer
Database Management System
IToolServer
IssueTrackerServer
IToolServer
ChangeLoggerServer
ISandbox
Repository Access Controller

Figure 6 - Package overview for Tracking Tools

- **RequirementsTrackerServer, IssueTrackerServer, ChangeLoggerServer –** These packages represent the main packages for each of the tracking tools. Their responsibilities include communicating with the user through the client, and communicating with the database through the Database Management System. The ChangeLoggerServer also communicates with the repository through the Respository Access Controller.

- **Database Management System –** The Database Management System handles database-related functions, and does not understand the data stored within the database.

AUTOMATED FORMATTER

CONCEPT

The Automated Formatter is what many programmers informally refer to as a “Pretty Printer.” The goal of such a formatter is simple – to make all code in the project look like it came from the same company. Many styles of code in each language exist, and the differences between them are only cosmetic. For example, some hold that opening braces should be on the line following an `if` statement, while some hold that it should be on the same line. Developers can argue for hours over which style is the most readable, leading many managers and team leads to dictate mandatory standards, which developers must conform to manually. The Automated Formatter aims to save time and prevent mistakes by automatically processing the code and cosmetically modifying it to become consistent.

USES

This tool is primarily designed for developers to run on their code after committing to the repository. Developers can set up Agents to run the Formatter for them, or they can run it manually. This tool is not designed for managers who may not understand the consequences of running this tool on various source items.

STRUCTURE

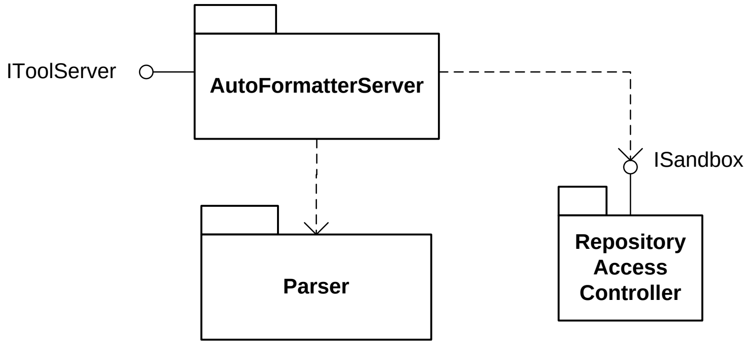

Figure 7 - Package overview for Automated Formatter

- **AutoFormatterServer –** This is the main executive for the tool. It opens each specified file and uses the parser to identify sections of code that it knows the desired formatting for – for example, a function declaration. It then outputs to an intermediate file the correctly formatted version of this section. If an unrecognized code segment is discovered, it outputs without modification. After the file's processing is complete, it copies the intermediate file over the original file.

- **Parser –** This parser is the same parser as found in the Risk Analyzer and Defect Analyzer. It scans through files, seeking out patterns of tokens that match specified rules. Each rule is given a list of actions, which will be executed when the parser finds a pattern of tokens that match. Rules and actions all inherit from abstract classes, allowing for a fully polymorphic parser. The rules used in this case simply detect notable components, and the actions notify the AutoFormatterServer for further processing.

CRITICAL ISSUES

- **Cosmetic accuracy –** This tool must only make changes that are truly cosmetic, and take pains to not alter the code's critical structure. This tool will be tested with a variety of input files to ensure it does not edit code it does not understand.
- **Performance –** A great deal of files must be processed, without affecting each others' processing, making this an ideal tool for parallelization. Many threads can be spun off, each processing a file at a time. The number of threads should be chosen based on the host's number of cores and the write speed of the host's hard disks.

AGENTS

CONCEPT

Agents are small, customizable units that can be used to personalize the toolset. Agents work off three pieces of information:

- A trigger – can be a time, an event from another tool, or a manual trigger
- A tool to run – can be any of the tools in the toolset
- An ensemble to run the code on – generated with the Ensemble Builder

When the trigger event occurs, the agent runs the specified tool on the specified ensemble.

USES

- **Nightly status report –** A manager or team lead, desiring to remain knowledgeable, could setup an agent to run on the repository nightly to generate a status report available in the morning. This manager would become very informed. For example, each morning he could learn how many issues had been solved the previous day, how many issues had been reported, and what had been done to decrease risk on the project.

- **Risk notification –** A developer is likely to want notification when he submits code that significantly increases the risk of a project. He can set up an agent to run the Risk Analyzer on his area of the codebase whenever code is committed to that area, and know immediately when changes increase risk. This is one of the ideal uses of the Risk analyzer, and allows for prompt correction when high-risk code is introduced. The same setup can be used with the Defect Analyzer to allow for quick detection and correction of code toxicity.

STRUCTURE

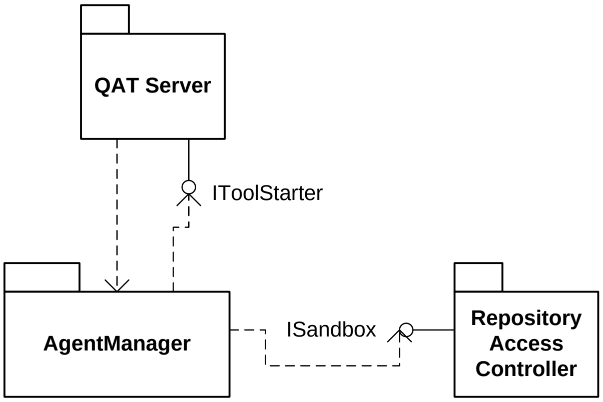

Figure 8 – Package overview of Agent Manager

AgentManager – This package is responsible for creation, editing, and invoking of Agents. It relies on user input from the Client to create and edit Agents, and invokes based on either time or events from the QAT Server. When an Agent needs to invoke a tool, it does so through the QAT Server's IToolStarter interface. This interface is used to reduce the chances that a change in the QAT Server will trigger a change in the AgentManager. The AgentManager needs access to the filesystem to read ensemble files and to load and save agent data, which it accomplishes through the ISandbox interface.

CRITICAL ISSUES

- **Security –** Since Agents can be set up by any user and by run by the server, care must be taken to ensure no Agent runs at a higher privilege level than the instigating user. For example, if Alice sets up an Agent to run whenever a change is committed to Project Alpha, this agent should only ever run at Alice's security level. If Bob submits a change to Project Alpha, the Change Logger will detect the change and fire off Alice's Agent as a result of Bob's actions. Agents must be designed to check that they are running under the owner's credentials before executing any tools.

ENSEMBLE BUILDER

CONCEPT

The Ensemble Builder is used to define subsets of the main code repository for examination by other tools in the toolset. It is often desirable to run analyses on a single project or subproject at a time, rather than looking at the baseline as a whole. A GUI is made available to the user to select files to include in the ensemble, as well as to load from or save to an XML file.

USES

- **Monitoring sub-projects –** On a large project, sub-projects may have individual team leads. These team leads are more interested in measuring the health of the subset of the codebase they are responsible for, and would want to define an ensemble describing the sub-project. This would allow the team lead to focus the Risk and Defect analyzers on their sub-project, leaving the main project health measurements to the project lead.

- **Monitoring individual contributions –** Individual developers may also be interested in defining ensembles for their own code, as they may have worked on many modules. They may also be interested in cleaning up their code to match the company standard, as enforced by the Automated Formatter. They may also desire to use the Change Logger tool to observe changes to other modules – for example, to receive notification when an issue affecting them has been solved.

STRUCTURE

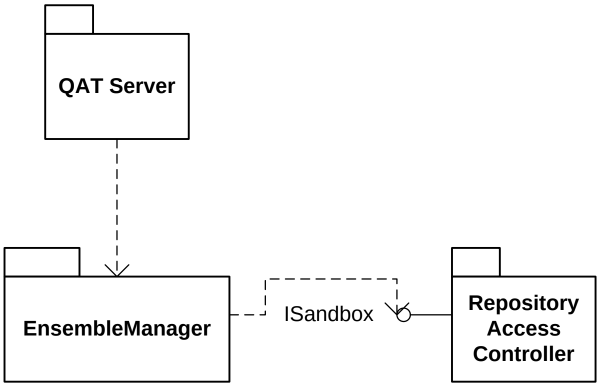

Figure 9 – Package overview for the Ensemble Manager

EnsembleManager – Since the Ensemble Manager's tasks are relatively simple, they can be contained in one package. This package is responsible for accepting and acting upon direction from the user regarding creating, loading, and saving of Ensembles.

CRITICAL ISSUES

- **Portability –** Not all repositories remain in the same place through their whole lifespan, so a list of files contained within the ensemble must be flexible. To account for the possibility of changed or branched codebases, ensemble paths are all stored relative to the directory in which they currently reside.

SYSTEM-WIDE LOGGING FACILITY

CONCEPT

Since multiple QAT tools may be running at any given moment, all accomplishing very different tasks, the Toolset needs a coherent logging facility that all modules can use to log events. This logging facility will be accessed by any of the other tools in the toolset, and can be viewed through its client. The output will be in XML, for easy viewing.

USES

- **Use metrics –** Managers may want to know how much use the various tools incur, and use these metrics to decide which tools to purchase in the future.
- **Ease of administration –** Errors are bound to occur in the Toolset, and having a clear log of events can help debug issues. Administrators can use the log to submit clear bug reports.

STRUCTURE

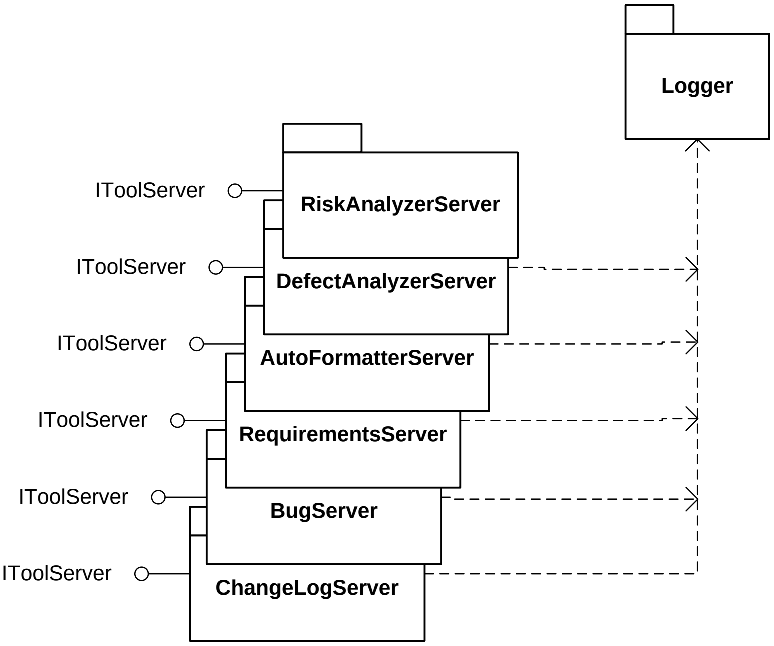

Figure 10 – Package overview for the Logger

Logger – This feature is simple enough to merit a single package. This package is responsible for receiving input from all tools in the toolset, and logging them efficiently in clear XML.

CRITICAL ISSUES

Performance is a major issue that must be considered in the design of the logging facility. With optimized tools such as the Risk Analyzer and the Defect Analyzer combing through thousands of files, there's great potential for logged events to come in at a rapid rate. If these events are written to the disk every time they are received, the logger could bog down and become inoperable. To investigate the significance of this lag, a prototype logger was written to measure the difference between writing immediately to disk and using a queue to store log entries, and dumping the queue to disk when it reaches a certain size. The prototype was built with a high-resolution timer around the code accomplishing the logging, as follows:

```
private void Log(string data)
{
    System.Diagnostics.Stopwatch timer = new System.Diagnostics.Stopwatch();
    timer.Start();

    if (filewriter != null)
    {
        // under one scenario, keep in the queue until full, then flush queue
        if (enable_buffer)
        {
            log_buffer.Enqueue(data);
            if (log_buffer.Count > BUFFER_SIZE)
                Flush();
        }
        // under one scenario, write directly to file
        else
        {
            filewriter.WriteLine(data);
            filewriter.Flush();
        }
    }

    num_log_events++;
    timer.Stop();
    time_spent_logging += timer.Elapsed;
}

// flush the buffer
public void Flush()
{
    string str = "";
    for (str = log_buffer.Dequeue(); log_buffer.Count > 0; str = log_buffer.Dequeue())
    {
        filewriter.WriteLine(str);
    }
    filewriter.WriteLine(str);
    filewriter.Flush();
}
```

Buffering was enabled or disabled using the `enable_buffer` member variable, allowing the same timer code to operate both types of logging for consistency.

This was tested under the following scenarios:

1. **Single utility using unbuffered log** – one utility was run, dumping events into the log at its maximum rate. The logger dumped directly to disk and flushed its write buffer upon each logging event. The following performance was observed:

Run	Events logged	ms/event	
1	1000	0.029769	
2	1000	0.010579	
3	1000	0.013244	
4	1000	0.010557	
5	1000	0.010891	
6	1000	0.010649	
7	1000	0.010507	
8	1000	0.010366	
9	1000	0.010308	
10	1000	0.010707	
	Average:	**0.012758**	**ms**

2. **Single utility using buffered log** – one utility was run, dumping events into the log at its maximum rate. The logger queued events, and dumped its buffer every 10 items. The following performance was observed:

Run	Events logged	ms/event	
1	1000	0.002058	
2	1000	0.002038	
3	1000	0.001925	
4	1000	0.0019	
5	1000	0.001809	
6	1000	0.001911	
7	1000	0.001783	
8	1000	0.001868	
9	1000	0.001899	
10	1000	0.001863	
	Average:	**0.001905**	**ms**

3. **Single utility using large buffered log** – one utility was run, dumping events into the log at its maximum rate. The logger queued events, and dumped its buffer every 100 items. The following performance was observed:

Run	Events logged	ms/event	
1	1000	0.003667	
2	1000	0.002414	
3	1000	0.001943	
4	1000	0.002063	
5	1000	0.002053	
6	1000	0.001907	
7	1000	0.001881	
8	1000	0.001891	
9	1000	0.001889	
10	1000	0.001905	
	Average:	**0.002161**	**ms**

4. **Multiple utilities using unbuffered log** – twenty instances of the test utility were activated, creating significant traffic density. The logger dumped directly to disk and flushed its write buffer upon each logging event. The following performance was observed:

Run	Events logged	ms/event	
1	20000	0.013722	
2	20000	0.013248	
3	20000	0.013512	
4	20000	0.014017	
5	20000	0.014076	
6	20000	0.013575	
7	20000	0.013487	
8	20000	0.0131	
9	20000	0.013531	
10	20000	0.014042	
	Average:	**0.013631**	**ms**

5. **Multiple utilities using buffered log** – twenty instances of the test utility were activated, creating significant traffic density. The logger queued events, and dumped its buffer every 10 items. The following performance was observed:

Run	Events logged	ms/event
1	20000	0.002948

2	20000	0.002908	
3	20000	0.00305	
4	20000	0.002778	
5	20000	0.002943	
6	20000	0.00293	
7	20000	0.002605	
8	20000	0.002798	
9	20000	0.003218	
10	20000	0.002894	
	Average:	**0.002907**	**ms**

6. **Multiple utilities using large buffered log** – twenty instances of the test utility were activated, creating significant traffic density. The logger queued events, and dumped its buffer every 100 items. The following performance was observed:

Run	**Events logged**	**ms/event**	
1	20000	0.003097	
2	20000	0.003172	
3	20000	0.00298	
4	20000	0.003031	
5	20000	0.003	
6	20000	0.003214	
7	20000	0.002948	
8	20000	0.003296	
9	20000	0.002961	
10	20000	0.002901	
	Average:	**0.00306**	**ms**

From these results, it appears obvious that queuing log entries will improve performance dramatically. First, the addition of logging reduces the time per logging event by a factor of approximately five – from 12.8μs in Scenario 1 to 1.9μs in Scenario 2, and from 13.6ms in Scenario 4 to 2.9ms in Scenario 5. Scenarios 3 and 6 illustrate that the queue size has an effect on performance, but require more experimentation to understand fully. It would appear that each configuration of hard drives has an ideal burst length, and that each system should be configured such that the logger's queue flushes match the ideal burst length.