# Bicyclerepairman Design

This information is correct at 27th Jan 2004.

## Bike Component Interface

import bike  
  
ctx = bike.init()  
ctx.renameByCoordinates("/home/joe/src/foo/bah.py",52,8,"bah")  
ctx.save()  
ctx.undo()

## File System Dependency:

Bicyclerepair man operates off of the file system. Modules are loaded from the file system, and saved back after a refactoring. Integration with IDEs currently relies on the IDE ensuring that the files on disk are up-to-date before performing a query or refactoring, and is responsible for reloading the modified files after the operation.

## Parsing and Fastparsing - the ***parser*** package

BRM originally did all its parsing using the Python compiler package (see python docs). This parser generates very accurate AST. Unfortunately this proved to be far too slow for interactive use, and so I built a seperate parser which does a basic job of parsing, but very fast. BRM uses this basic parser to locate and navigate round the code, and then applies the compiler package to small bits of code where necessary.

The fastparser module in the parsing package operates on a single module at a time. It generates a simple tree containing Class and Function nodes, with a Module node at the root. This tree is called the 'fastparserast', and it's nodes (Module,Class,Function) are defined in the fastparserast module. Each fastparserast node contains a pointer to the start and end lines of the node in the source text, and basic information (e.g. name). It also contains methods for navigating between nodes (e.g. getChildNodes(), getParent()

### BRM Tricks - masking the code

The fastparser is very fast because it leverages the power of the regex package, and has been carefully profiled and optimised. Before parsing a file it applies a number of regexes over the source to mask out all the comments and strings. This then allows it to scan the source very quickly for functions and classes using python keywords ('class', 'def') without fear of getting confused by occurences of keywords in strings. It builds up a tree of functions and classes in a module, storing their start and end positions (lines) in the source text.

### BRM Tricks2 - logical lines and doctored lines

As mentioned above, the compiler module is very accurate but very slow. Because of this, BRM uses it sparingly, choosing only to parse lines of code when it is reasonably confident that the line contains something important.

In order to parse a single line of source code, the line must sometimes be doctored to make it parsable. Take for example the following:

def foo(a, b, c):

This line cannot be parsed on its own, but may contain some interesting code. BRM overcomes this constraint via the makeLineParseable() function in the parserutils module. This function decorates the line with noops that make it parsable. For the above example it generates the following code:

def foo(a, b, c):  
 pass

Which parses fine.

The other problem with parsing single lines of code is that python statements often span multiple lines. e.g.

myref = foo(arg1, arg2,   
 arg3, arg4)

To combat this problem, BRM has a neat function called generateLogicalLines() (also on the parserutils module). This function takes a list of (masked) physical lines as input, and generates complete 'logical' lines as output.

## Querying the code - The ***query*** Package

As mentioned above, the relative slow speed of the compiler module prohibits BRM from building up an AST of the entire file and then navigating it. Instead the query package uses text searches to narrow down the search and then parses single 'logical lines' using the compiler.

### logical lines and doctored lines

In order to parse a single line of source code, the line must sometimes be doctored to make it parsable. Take for example the following:

a = myfunc(arg1, arg2,  
 arg3, arg4)

def foo(a, b, c):

The first problem is multiline lines.

### Example - findDefinition

For example, when searching for the definition of a reference (see findDefinition.py), BRM first takes the input coordinates and locates the fastparserast node denoting the class/function scope containing the reference (by checking the coords against the start/end coords of the fastparserast nodes). This provides the initial search scope for the query.

BRM then obtains the sourcecode of the scope with strings/comments masked out. It searches

[Phil Dawes](mailto:pdawes@users.sourceforge.net) Last modified: Mon Jan 26 18:40:51 GMT 2004