# Assisting the Code Review Process Using Simple Pattern Recognition

Eitan Farchi
I.B.M. Research Laboratory, Haifa, Israel

Bradley R. Harrington
I.B.M. Systems and Technology Group, Austin, U.S.A

October 21, 2005

**Abstract**

We present a portable bug pattern customization tool defined on top of the Perl regular expression support and describe its usage. The tool serves a different purpose than static analysis, as it is easily applicable and customizable to individual programming environments. The tool has a syntactic sugar meta-language that enables easy implementation of automatic detection of bug patterns. We describe how we used the tool to assist the code review process.

## 1 Introduction

Formal reviews in general and specifically formal code reviews are one of the most effective validation techniques currently known in the industry. Code reviews are known to find a defect every 0.2 hours and a major defect every 2 hours [3]. The down sides of formal reviews, especially formal code reviews, are that they are skill sensitive and often tedious.

Bug patterns and programming pitfalls are well known and effective programming tools that aid development, review, testing and debugging [5][1]. Bug patterns and pitfalls typically aid the review process through code review checklists or question catalogs (see appendix five in [4] for an example). As pointed out in [4], code review question catalogs must be kept small to be effective. As a result, a customization process is required - projects should create their own customized question catalog.

The developers of compilers and static analysis tools have attempted to address the skill sensitiveness, tediousness and need for customization of code reviews. Traditionally compilers could be directed to look for programming

pitfalls. Notable examples of static analysis tools are lint [1], Beam[2], FindBugs[3] and the Rational code review feature. The user scenario in effect when applying these tools is to go over a list of warnings given by the tool and determine if a warning is actually a problem. The difficulty with this approach is false alarms, i.e., warnings which are not actually problems.

While all of the above tools provide customization options, these options are limited, as they only have a fixed list of pitfalls to choose from. Notably, Jtest[4] takes this a step further. Jtest for Java and for C has a GUI based language for specifying pitfalls.

Another difficulty with the static analysis approach is portability - static analysis tools require parsing of the programming language and additional information, such as symbolic tables, to perform their task. Such information is language specific. Today's organizations might work with a multitude of programming languages (we have seen instances of up to five different programming languages having different abstraction levels from assembly to C++ in the same organization). In addition, off the shelf tools will not always work as is, in specific environments and additional work will be required (up to 1 PY in one instance in our experience) to adapt an off the shelf tool that requires parsing of a language to a complex industrial development and testing environment.

When designing the language, we had several requirements in mind. It was important to use an easy to learn, easy to implement grammar, using off the shelf tools. Additionally, we felt that a declarative language will be better than an imperative language. The ready availability of a well known, portable, pattern recognition language for use as a building block was the most attractive option. The Perl language regular expression support perfectly fits this requirement. However, our example set, based on known review checklists [1] showed that the Perl regular expression support will not be sufficient to declaratively specify many bug patterns (see section 2.2 for some examples). Consequently, we identified that the following is required

- Ability to find other patterns which follow or precede an identified pattern
- A hierarchical construct, enabling the developer to verify if a pattern does not match a line prior to checking the following line
- Identify repeated equal occurrences of a given pattern within an identified pattern

These requirements lead to the definition of a minimal syntactic sugar extension built on top of the Perl regular expression facility that enables the declarative definitions of patterns conforming to the above requirements.

[1] See also $http://www.pdc.kth.se/training/Tutor/Basics/lint/indexframe.html$.
[2] See $http://www-cad.eecs.berkeley.edu/cad-seminar/spring02/abstract/krohm.htm$
[3] See $findbugs.sourceforge.net$
[4] See $www.parasoft.com$

The portable bug pattern customization tool serves a different purpose than static analysis, as it is easily applicable and customizable to individual programming environments. A given project has certain programming pitfalls that may not be applicable to other environments and may use a range of programming languages. Also, even though there is some overlap with static analysis, the tool catches different types of bugs by using specific project related information. Thus, the tool is meant to be used in concert with static analysis.

This work addresses the customization and portability requirements. We have prototyped a portable bug pattern customization tool to assist the Rephrase review process[5]. Bug patterns are defined using a simple pattern recognition language, building on top of Perl regular expressions. Thus, the solution is portable and can be applied to any programming language or for that matter even to design documents. Once the tool is applied to a project, code lines are annotated with review questions. Next, during the review meeting, the annotated code is visible to all code reviewers, as the readers rephrase the code.

Whereas regular expression based tools have certain advantages, we recognize static analysis based tools (e.g., Jtest and Beam) may find problems regular expression based tools cannot. Specifically, static analysis tools operate on data flow (e.g., define-use graph) graphs and the control flow graph of the program. This type of information is not available when applying our regular expression based approach thus limiting its effectiveness. For example, data flow analysis enables aliasing analysis. As a result, two pointers that point to the same memory location can be identified and functions that have hazardous side effects can be identified. Another example is identifying variables that are initialized along some paths and are not initialized along some other paths in the program. This analysis requires both data flow and control flow information and is difficult to achieve using the regular expression approach.

The bug pattern customization tool is intended to be used as a supplement to existing source code review tools. The tool allows for the automatic detection of possible bug patterns during reviews.

This paper is organized as follows. First, we describe our simple pattern recognition review assisting tool. Next, typical bug pattern identification using the tool is discussed. We then describe how the tool is used to assist the review process. Finally, experience of applying the tool to real life software projects is presented.

[5]See *www2.umassd.edu/SWPI/NASA/figuide.html* for details on the Rephrase review technique

# 2 A Simple Pattern Recognition Review Assisting Tool

In this section, an overview of our simple pattern recognition review tool is given. The user defines a set of patterns, (further explained in 2.1 and formally defined in A), applicable to the project at hand. The project code is then searched for any occurrence of the patterns defined by the user. When a pattern is found, a review question is inserted above the pattern instance. Thus, creating a new version of the code, annotated with review questions. The annotated code is then reviewed, instead of the original code.

## 2.1 Defining a pattern

A bug pattern (see appendix A for a formal definition), or simply a pattern, is defined on top of the Perl regular expression facility [6]. Bug patterns are written by the project developers identifying possible problematic parts of the code under review. A pattern is implemented by using a set of Perl regular expressions. Using the syntactic sugar pattern language we have defined, it is feasible to check for a certain sequence of Perl regular expression matches and non-matches. We have found this syntactic sugar pattern language helpful in facilitating the definition of project specific bug patterns lists.

The bug pattern definition associates review questions with a bug pattern. Thus, a bug pattern definition includes the following elements.

- A definition of an interesting pattern to review
- A review question to be associated with each pattern instance found during the search (short question)
- A more detailed review question that is associated with each searched file for which at least one instance of the bug pattern was found

An example of a pattern definition follows. The pattern definition attempts to find a line that is not a trace line (a trace line logs information on the program to enable filed analysis), is probably a for loop line and has an assignment in its condition. Note that we are not too concerned if we will "find" instances that are not assignment in loop conditions as this will be easily ignored by the reviewers during the review process. Also note that in defining the pattern below we use project specific information - namely that traces are done using the $DEBUG\_TRACE\_PRINTF$ macro. The example is annotated to make it self explanatory.

```
pattern
#The line is not a print statement
```

```
   nmatch
   DEBUG_TRACE_PRINTF
   next {
#The line contains an assignment within two semi column.
#It is probably something like for(i = 0;  i = 3; i++)
      match
      ;.*=.*;
      this {
#The part that was previously matched, e.g.,  ;i = 3; , is matched.
# .*\s+=\s+.* below means possibly an identifier (.*) followed by some
#spaces (\s+), an assignment (=), some additional spaces (\s+) and
#then another possible identifier (.*) which if matched looks like it is an
#assignment
         match
         .*\s+=\s+.*
      }
    }
short question: ASSIGNMENT IN CONDITION?
long question: Double check conditions to determine if = is used instead of ==
\pattern
```

We found writing bug patterns using this syntactic sugar pattern definition language easy. Within IBM it was quickly adopted by programmers from different organizations, with different skills and backgrounds and required a negligible learning curve. Three developers constructed a useful catalog of approximately 50 bug patterns within approximately 20 person hours.

## 2.2 Using the Tool to Search for Bug Patterns and Focus the Review

In preparation for the review meeting, the tool is applied to the project under review. Bug patterns can be grouped based on concerns. For a specific concern, (e.g., concurrency,) the tool identifies a subset of the project under review, (typically a set of files,) in which the occurrence of bug patterns related to the concern were found. The review can be further focused by briefing the reviewers to focus on the specific concern when reviewing the identified subset of the project.

## 2.3 The Annotated Code Used in The Review

After application of the tool, a new version of the code under review is produced with review questions to assist the review process. To illustrate, the pattern from the previous subsection will annotate the following code segment:

```
for(i = 2; i = j); i++)
for(i == 2; i = j); i++)
for(i = 2; i== j); i++)
for(i == 2; i== j); i++)
```

as follows:

```
/*REVIEW QUESTION(0) -  ASSIGNMENT IN CONDITION?  */
for(i = 2; i = j); i++)
/*REVIEW QUESTION(0) -  ASSIGNMENT IN CONDITION?  */
for(i == 2; i = j); i++)
for(i = 2; i== j); i++)
for(i == 2; i== j); i++)
```

Thus, warning of a possible assignment in a condition, in the proceeding code segment. Note that $for(i = 2; i == j); i++)$ and $for(i == 2; i == j); i++)$ are not flagged as they do not contain an assignment in the for condition.

# 3 Identification of Typical Bug Patterns

In this section we explain how to use the bug pattern customization tool to identify typical bug patterns. A detailed description of the bug pattern definition meta-language is formally defined in appendix A, briefly described in the previous section, and expanded upon here.

The meta-language consists of ten keywords built on top of Perl regular expressions. Perl regular expressions were chosen for their well known ability to allow for easy pattern matching, and grouping.

The meta-language is also source code independent, and could possibly be used to detect patterns in non-source code files such as design documentation. Using this language, we have been able to successfully detect bug patterns in the C, C++, Java and FORTH programming languages. A simple explanation of the ten keywords is as follows:

**pattern, \pattern** indicates the beginning and end of a bug pattern.

**{}** indicates a block of pattern detection code.

**$dollarX** where X is a number starting at 1, corresponds to the $<digits>Perl keywords.

**this** corresponds to the $MATCH and $& Perl keywords.

**before** corresponds to the $PREMATCH and $‘ Perl keywords.

**after** corresponds to the $POSTMATCH and $' Perl keywords.

**match** indicates the following line is a regular expression to be matched.

**nmatch** indicates the following line is a regular expression to be inversely (not) matched.

**window(X)** indicates the following pattern should check for a match on the next X source lines of code.

**code** indicates a free form block of Perl source code to be evaluated as a Boolean expression.

The following illustrates a few simple examples on how the language is used.

In a simple example, a project under review had a history of bugs in which the programmer unnecessarily uses floating point variables. The following pattern prints a simple statement above the variable declaration during the code review.

```
pattern
        match
        float
short question: Does the program unnecessarily use float or double?
long question:
\pattern
```

An original source code file being reviewed with this line:

```
float foobar;
```

will display this during the code review:

```
/* REVIEW QUESTION(0) Does the program unnecessarily use float or double ? */
float foobar;
```

In another example, a well known bug pattern occurs when resources are not released along error paths. General error path identification using static analysis is difficult, especially outside of specific programming constructs, such as the exception handling mechanism. In contrast, we can leverage a given project's conventional methods to easily identify error paths by identifying project specific constructs. On one program we worked with, the error paths occur when the return code from a function call stored in a variable named *rc* is not zero. Using this project specific information, team members where able to define the following bug pattern and use it effectively during review sessions.

```
pattern
        match
#Error path looks something like - if (rc != 0) {
        if\s*\(.*rc.*\)
short question: RESOURCE RELEASED ALONG ERROR PATH?
long question: Are obtained resources released along error paths?
\pattern
```

Another example, demonstrating the use of a sequence of matches and non-matches of Perl regular expressions, follows. During the code review session, the reviewers should ensure all dereferenced pointers are always verified to be non-null, at runtime. A review question associated with dereferencing pointers would remind reviewers of the risk of a null pointer exception. However, the tool should only identify sections of code, and not comments. This was important for the project under review, as there were many comments that included dereference statements. The bug pattern below ignores C++ style comment lines and then checks for a pointer dereference. Only then is the review question added to the code under review.

```
pattern
#check that this is not a comment line
        nmatch
        \/\/
        next {
#this is probably a dereference line
   match
   ->
}
short question:  NULL DEREFERENCE?
long question: When dereferenced, can a pointer ever be null
\pattern
```

One obvious problem with this pattern is the possible presence of C-style comments. Due to the inherent limitations of regular expressions, it may be necessary for the project enforce a slightly more restrictive coding standard, or else risk false posititves. Conversely, the tool may be well suited for the enforcement of coding standards in the project itself. This is a topic for future exploration.

In a more complex example, perhaps the program has recently had a problem with C++ copy constructor methods.

```
pattern
        match
        (.*)::(.*)\(const (.*)\&
```

```
        code {
               dollar1 eq dollar2 && dollar1 eq dollar3
        }
short question:  COPY CONSTRUCTOR
long question:  Is a copy constructor method really required here?
\pattern
```

This pattern will match the following code:

```
Employee::Employee(const Employee& t)
```

Thus flagging the copy constructor method for special notice during the code review, giving the code the desired added attention.

In a more advanced example, perhaps the project team wanted to ensure all copy constructor methods are closely followed by corresponding destructor methods.

```
pattern
        match
        (.*)::(.*)\(const (.*)\&
        code {
               dollar1 eq dollar2 && dollar1 eq dollar3
        }
        window(20) {
                nmatch
                (.*)::~(.*)
                code {
                        dollar1 eq dollar2
                }
        }
short question: Is a matching destructor present?
long question:
\pattern
```

This pattern will alert that a matching destructor of the form:

```
Employee::~Employee()
```

might not be present.

It may also be useful to incorporate some or all of the contents of industry standard code inspection checklists into the tool. These are often very useful code review and programming guides, and are readily available. A bug pattern would be written to describe each item on the checklist, as applicable. One such

checklist is Baldwin's Abbreviated C++ Code Inspection Checklist[2]. As part of our pilot, we have incorporated most of the patterns from Baldwin's list, to provide a baseline of bug patterns, which should be viable for most projects.

# 4 Using the Bug Pattern Customization Tool to Assist the Code Review Process

In preparation for adoption of the bug pattern customization tool by a project, a set of project specific bug patterns are created. As a starting point, general code review checklists, like the one provided in [4], are used. Project members use the meta-language to create a set of bug pattern definitions. The bug patterns are defined using knowledge of project specifics, generally previous problems, and are run against the project as sanity checks to see if the results make sense. Sometimes, at this stage, issues are revealed but this is not the objective of the stage, but a side benefit. Based on previous experience we estimate this work to be approximately one person week of work for a non-trivial middleware component (which could be amortized over a period of a few months).

Typically, code review checklists and pitfalls are categorized according to review concerns - for example, declarations, data items, initializations, macros, and synchronization constructs[4]. Code reviews should be performed with a mindset similar to testing, and should be driven by these concerns. When a certain concern drives a specific code review, the subset of bug patterns related to that concern are used, thus further focusing the review process. For example, if we are interested in problems related to the use of macros, then only the part of the project annotated with review questions related to macros is actually reviewed.

Next, as mentioned in previous sections, during the review meeting the annotated source code is reviewed instead of the original source code. When an issue is revealed during the review, the review team might write a bug pattern to search the code for additional occurrence of the same problem.

A concrete example follows. In a middleware project that extensively uses macros, it was found that a set of macros, $CA\_QUERY\_BIT\_SET$ being one of them (see below), should be used in such a way that the first parameter contains the word Flags and the second parameter contains the word FLAG. Once this is identified, a bug pattern can be written to check this for the entire code base.

```
if (!CA_QUERY_BIT_SET(pCsComTCB->BM.bmExtendedFlags,
   CS_BM_NVBM_EX_FLAG__CALLED_BY_DP)) {
```

# 5 Experience

In this section we report empirical instances of usage of the bug patterns customization tool. The initial results are encouraging and indicate that the tool is useful in assisting the code review process.

## 5.1 ConTest Code Reviews

ConTest[6] is a concurrent program testing tool, used extensively within IBM. As an intrusive test tool that modifies the object code of the program under test, it should meet high quality standards in order to be used safely and reliably. We found out that users have very little tolerance for false alarms that turn out to be bugs in ConTest itself.

As a result of the high quality requirements, the ConTest development team conducts regular code reviews. For the last two months, the code review sessions were conducted on code annotated by the bug pattern customization tool. The process of the code review included the owner going over the annotated source code projected on the screen and rephrasing the code.

Obtaining the annotated code turned out to be quick and did not require additional effort on the review team. In addition, the annotated review questions obtained, based on a customized set of bug patterns did not distract the reviewer. Finally, actual problems were identified with the help of the annotated review questions.

For example in one review instance it was decided to change the interface used in a function, $findTargetString()$, and pass it a pointer to one structure instead of passing it three separate pointers that are logically connected. The following is the annotated code segment with the review question that started the discussion and eventual code change.

```
/*REVIEW QUESTION(3) -  STORAGE INSTEAD? */
nTargetIndex = findTargetString(strNew.c_str(), &nIndexAfterTarget, &callType);
```

## 5.2 Avoiding Field Escapes Through Guided Code Review

The following string buffer overflow bug, recently surfaced during an embedded software stack bringup. The offending code had been in place for several years,

[6] See *www.alphaworks.ibm.com/tech/contest* for details

with no issues. The routine takes place at the hand-off point between two different embedded software components, where one component is about to terminate, and the other is about to begin. Thus, the component interaction is similar to an interprocess communication (IPC) scenario, as one component does not have much knowledge of the other's internal data structures.

The variable *glob_fw_vernum* was defined as a 16 byte character array. The *fw_version_string* variable points to a string of unknown length, because of the IPC-like interaction. The following is executed by the offending code:

```
strcpy(glob_fw_vernum, (char *)of_data_stackptr->fw_version_string);
```

The length of the string actually originated from the embedded software image, where it was originally generated by the build team. The build process changed because of new developments on the platform. Thus, the length of the string grew from 16 characters, to 40 characters. This increase in length created a buffer overflow situation, which was not detected until a long time after the offending code had been executed. Eventually, the embedded software stack crashed due to a data storage interrupt, where the platform was attempting to read from an address that no longer made sense, because it was overwritten by the new longer version string.

A simple bug pattern could have easily warned reviewers of this situation. Also, there is reason to believe the pattern would have been flagged, because of a long history of string problems within this project. The relevant bug pattern follows.

```
pattern
        match
        strcpy
short question: Is the buffer large enough?  Is overflow handled?
long question: Has enough space been allocated to hold the size of
the string to be copied, and if not are there an overflow precautions?
```

Had the bug pattern review tool been available during the development of this code, or during subsequent reviews of this portion of the firmware, there is reason to believe the bug would not have escaped the code review.

Another bug, this time a search routine error, was a unit test escape from an embedded software component. In this situation, a data structure search routine designed to be used, similar to the Standard C Library *fread*() routine, is not properly utilized.

The search routine will only search the data structure beginning from the location of the most recent search. If the entire structure is to be searched the search routine must explicitly be instructed to do begin its search from the beginning. This is done with the *set_start*() routine. However, the bug would

not be caught with a simple unit test, as the test would most likely succeed on at least the first pass, and would only fail after repeated runs of the test. The offending source code bug follows.

```
set_filter(&filter,(uint8 *) "PCI-DEV" , 1,
                S_Offset(PCI_DEV_NODE,uniqueId), drc_index);
node_ptr = find_node(&iter, &filter);
```

The repaired routine:

```
set_start(&iter, NULL);
set_filter(&filter,(uint8 *) "PCI-DEV" , 1,
                S_Offset(PCI_DEV_NODE,uniqueId), drc_index);
node_ptr = find_node(&iter, &filter);
```

A similarly simple bug pattern could have prevented this unit test escape, as this is a common pitfall in using this API. The relevant bug pattern is as follows.

```
pattern
        match
        set_filter
short question: Is set_start() called before set_filter()?
long question:
```

# 6 Conclusion

We have prototyped a portable bug pattern customization tool to assist the rephrase review process. Bug patterns are defined using a simple pattern recognition language, built on top of Perl regular expressions. Thus, the solution is portable and can be applied to any programming language or for that matter even to design documents. Once the tool is applied to a project, code lines are annotated with review questions. Next, during the review meeting, the annotated code is visible to all code reviewers, as the readers rephrase the code.

Empirical experience indicates that the bug pattern customization tool can be effectively used to assist the code review process.

Further research will focus on gaining a comprehensive experience in the usage to the tool and enhancing the pattern definition syntactic sugar language to meet different user's requirements.

# References

[1] Eric Allen. *Bug Patterns in Java.* Apress, 2002.

[2] John T. Baldwin. An abbreviated c++ code inspection checklist, 1992.

[3] Daniel Galin. *Software Quality Assurance*. Addison Wesley.

[4] Brian Marick. *The Craft of Software testing*. Prentice Hall, 1995.

[5] Scott Meyers. *Effective C++*. Addison-Wesley, 1997.

[6] Randal L. Shwartz and Tom Phoenix. Learning perl. O'REILLY, 1993.

# A Pattern Language Semantics

Below is a description of the language used to define a bug pattern. For simplicity the semantics of the *window* and *code* blocks are not given. The description below assumes familiarity with Perl regular expressions[6]. Specifically, the meaning of `$‘, $&, $’, $1, $2,...` is assumed and used.

## A.1 Match Node (N)

A match node (N) is a node that contains a regular expression, N.R, to be matched,

`A N.$ ‘ (‘‘before’’), N.$& (‘‘this’’) and N.$’ (‘‘after’’),`

fields and a variable length list of optional fields `N.$1, N.$2,...` all pointing to either a match node or a don't match node.

The actual implementation of the language refers to

`N$‘ as before, N$& as this, N$’ as after, N$1 as dollar1.`

## A.2 Don't Match Node (N)

A node containing a regular expression, N.R, that should not match and a pointer to either a match node or a don't match node N.next.

## A.3 Pattern Definition

A pattern is a rooted tree of either match or don't match nodes. The Context of a Pattern P wrt a String S is next defined.

The context of each node in a pattern, P, is defined wrt to a string S. The context of the root tree of the pattern P is S. Given that the context of some match node, N, in the pattern P is defined to be the string S1 then the context of `N.$‘, N.$&, N$’, N.$1, N.$2,..` are defined if N.R matches S1. They are defined exactly according to the semantic of Perl for these operators (i.e., `N.$‘`

is the part of S1 before the match, `N.S&` is the matched string, `N$'` is the part of S1 that is after the match, `$1` is the string that matched the first sub regular expression that was enclosed in a parentheses in N.R, `$2` matched the second one, etc). Given that the context of a don't match node N is S1 the context of N.next is defined only if S1 does not match N.R. In this case the context of N.next is S1.

The context of a pattern P wrt to S is the association of a context to each of its nodes as defined above.

## A.4 A match

A string S matches a pattern P if the association of each node with a context is a complete function to the set of strings (each node has a defined context).