Chapter 3

Identification of Crosscutting Concerns to Measure Software Complexity

3.1 Crosscutting Concerns

Understanding of concerns plays an important role in successful software development. From software development perspective concern may represent any feature, functional or non functional which is important for stakeholders. A requirement in requirement specification document represents a concern [93].

The modularization of concerns has been emphasized by software engineering principles for better understanding of concerns. Object oriented paradigm supports the modularization of concern. Improper implementation of features and limitations of programming language construct leads to scattered and tangled implementation of the concerns [18]. Every concern might not get modularized into a separate module. Such concerns whose implementation is scattered over more than one module are called crosscutting concerns. Such concerns lead to tangling of code. Empirical studies have revealed that scattered and tangled code degrades the code quality [44]. The negative impact of scattered and tangled code is reflected not only by internal quality metrics but also by the external quality metrics [44]. Poor modularization of crosscutting concerns results
in source code which has more defects and is difficult to maintain[103]. For development of good quality software, it is essential to identify crosscutting concerns [3].

3.2 Techniques for Identification of Crosscutting Concerns

Identification of crosscutting concerns at different stages of software development is essential for 3 reasons. First, for refactoring of legacy system to aspect oriented system. Secondly, for modularized implementation of concern, its crosscutting nature needs to be identified at analysis and design level. Thirdly, for appropriate distribution of testing effort, error prone crosscutting code needs to be identified. Several research studies have resulted in number of techniques and tools for identification of crosscutting concerns. Each of these techniques and tools has its own pros and cons. These techniques are applicable either on source code or on requirement document. ConcernMapper, Fan-in analysis, Theme/Doc, Prune dependency rule, Formal concept analysis, event traces, line co-change and clone detection are some of the techniques used for identifying crosscutting concerns.

3.2.1 Prune Dependency Rule

It is a manual method of assigning code to the specific concern. Prune dependency rule states that if there is a program element related to a concern then its removal or alteration is guided by removal or change of concern. There are number of sub-rules related to scenarios where the prune dependency can be established. Consider a scenario where the program element P is related to a concern C [44].

- A reference sub rule states that is a program element P’ references P then P’ is also related to C. The reference can be of any type related to method, field, class, interface or enum.

- Dominates sub rule states that if all elements that references P’ are related to C then P’ is also related to C.

- Element Containment sub rule states that if all elements of P’ are related to concern C then P’ is also related to C.
• Concern Containment sub rule states that if a program element $P$ is related to a concern then its sub element should be related to the descendants of the said concern.

• Inheritance sub rule states that if the inherited elements of a program element $P'$ are related to a concern then $P'$ is related to the concern. If on application of prune dependency rule, a removal or change in concern results in the removal or change of more than one program element then the concern is considered to be crosscutting.

### 3.2.2 Concern Mapper

It is eclipse based plugin for mapping the program elements to related concern. The tool requires the seed element, on the basis of which the code is searched. The seed element can be a word or a phrase related to a requirement for which the code needs to be mapped. For example, if there is an implementation of word editor and undo feature is one of the requirement then seed element can be undo. The concern mapper maps the program element implementing the functionality related to undo requirement [92].

### 3.2.3 Fan-in Analysis

This is a semi automatic approach for finding crosscutting concern in the existing system. This approach takes into consideration the fan-in analysis of a method, since high fan-in of a method is an indicator of crosscutting functionality. Fan-in analysis of a system involves three steps: Computing the fan-in for each method, filtering the methods to obtain the smaller set of methods which have high probability of implementing the crosscutting requirements and manual analysis of the filtered method for finding out specific patterns in the call site. The last step involving manual analysis requires the domain knowledge and is more important from aspect mining perspective [78].

### 3.2.4 Theme

This approach of identifying crosscutting concerns works at requirement and design level. The theme is an element of design that represents the structure and behavior of a feature or requirement. This approach is divided into two parts: Theme/Doc and Theme/UML.
Theme/Doc works at analysis phase and Theme/UML works at design phase. The approach requires an action element to be identified from the requirement documents and the action view is created using requirements and actions. The aim is to relate each requirement to one action. The actions which are related to more than one requirement are the probable candidates of crosscutting nature. That is requirements that contain action which is common to other requirements are the one responsible for bringing the scattered code in the implementation. Such requirements are crosscutting requirements [9].

3.2.5 Line co-change

This is the manual technique of identifying the crosscutting concerns in the source code of the existing system. The method involves identification of lines of source code that are changed together using a versioning system. If lines of code, belonging to more than one program element, is changed in the process of bug removal or requirement change implementation then such lines of code represents the scattered implementation of a requirement and therefore, is an indicator of crosscutting concern. The approach has been studied to be effective in identifying the crosscutting concern in JhotDraw, a 2D graphical framework [27].

3.2.6 Clone Detection

This is one of the automated approaches for finding the crosscutting concern. The basic assumption behind this approach is that the presence of code clones in the existing system represents the requirements that are not properly modularized during the implementation phase. Text based techniques, token based techniques, Abstract Syntax Tree (AST) based approach, Program Dependency Graph (PDG) based approach are some of the examples. The tool CCFinder, a token based technique for clone detection, was developed by T. Kamiya et al. The use of AST as a clone detection technique has been done in Project Bauhaus. The PDG based clone detector was developed by Komondooor. The clone detection as a techniques for finding crosscutting concern has been studied by Maginel Bruntink [23].
3.2.7 Formal Concept Analysis and Execution Trace

This approach is applied on execution trace of an application for which the crosscutting concern needs to be identified. The requirements of an application are represented as use cases. For each identified use case the execution trace is generated. Concept analysis is performed on elements of implementation units and the execution trace for each use case. The generated lattice is used for detecting the crosscutting concerns. If the implementation unit for a use case belongs to different classes then the functionality reflected by the use case is considered to be a crosscutting concern [103].

3.2.8 Event Trace

The approach uses the program execution trace to find the patterns in method calls. The patterns of method calls that are recurring are indicators of crosscutting concerns. The approach has been studied on Graffiti, editor for graphs, and a specific recurring pattern of method call was observed in logging concerns in Graffiti. The approach is supported with a tool for analyzing the execution trace of the program [18].

3.3 Comparison of Techniques for Identification of Crosscutting Concerns

We compared the techniques for identification of crosscutting concerns using five different parameters: Stage of software development when the technique is applicable, whether the technique is supported by a tool, number of input parameters or the quantum of information required by a technique, format of output generated and software on which a technique have been applied or tested is given in Table 3.3.

We identified and presented the limitations of techniques for identification of crosscutting concerns.

*Prune dependency rule:* This is a manual method of detecting crosscutting concerns. This approach suffers from two limitations. First, being a manual method the approach cannot be applied on large applications and secondly, person who intends to find the crosscutting concern should have the knowledge of the code. The knowledge related to
<table>
<thead>
<tr>
<th>Identification Technique</th>
<th>Software Development Stage</th>
<th>Input Parameters</th>
<th>Output Format</th>
<th>Tool Support</th>
<th>Software on which Technique has been tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prune dependency rule</td>
<td>After implementation</td>
<td>source code/implementation unit</td>
<td>Mapping of requirements with the related code</td>
<td>Manual</td>
<td>Medium sized projects 13 – 44 KLOC</td>
</tr>
<tr>
<td>ConcernMapper</td>
<td>After implementation</td>
<td>Requirements</td>
<td>Concern graph</td>
<td>Concern mapper</td>
<td>N.A.</td>
</tr>
<tr>
<td>Fan-in analysis</td>
<td>After implementation</td>
<td>Source code</td>
<td>N.A.</td>
<td>Semi-automatic tool: FINT</td>
<td>PET store, Jhot Draw and Tomcat</td>
</tr>
<tr>
<td>Theme</td>
<td>Analysis and design</td>
<td>List of key actions identified from requirements</td>
<td>Action view</td>
<td>Theme/Doc tool</td>
<td>Crystal game having 85 requirements</td>
</tr>
<tr>
<td>Line co-change</td>
<td>Maintenance</td>
<td>Source code and bug report</td>
<td>Mapping of lines of code changed together when a bug is corrected</td>
<td>Manual</td>
<td>Jhot Draw</td>
</tr>
<tr>
<td>Clone detection</td>
<td>After implementation</td>
<td>Source code</td>
<td>The code clones are identified and the code clones are reflected with highlights</td>
<td>CC finder</td>
<td>ASML</td>
</tr>
<tr>
<td>Formal concept analysis using execution trace</td>
<td>After implementation</td>
<td>Execution trace or use cases and source code</td>
<td>Concept lattice which relates the use cases with the program elements</td>
<td>Dynamo for tracing method execution</td>
<td>Java implementation of Dijkstra Algorithm</td>
</tr>
<tr>
<td>Event trace</td>
<td>After implementation</td>
<td>Execution trace of method calls</td>
<td>Recurring patterns of method calls</td>
<td>Dynamic aspect mining tool: DynAMiT</td>
<td>Graffiti</td>
</tr>
</tbody>
</table>

Table 3.1: Comparison of techniques for identification of crosscutting concerns
concerns or the features of an application is also required.

*Concern mapper:* This technique is supported with the tool and therefore, can be used with large applications. In this approach an important term from the concern or feature is selected as a seed. This seed acts as initial basis for searching the code related to a concern from which the seed is chosen. The effectiveness of the concern mapper depends on type of seed selected. The seed is considered to be good if it is leading to the mapping of the concern with the respective implementation. The selection of good seed requires the domain knowledge.

*Fan-in analysis:* This technique has only limited tool support. The approach involves three steps and the tool is applicable for only one step. Moreover the application of all the three steps involved in this approach requires domain knowledge of the application for which the crosscutting concerns are to be identified.

*Theme:* The approach is very effective in detecting crosscutting concerns at an early stage of software development. This approach requires the selection of actions from the requirements. This step needs to be done manually. If the number of requirements is very large then finding actions in the requirement becomes tedious job. Again this approach requires domain knowledge for finding out whether an action can be attributed to a particular requirement or not.

*Line co-change:* This approach requires the Concurrent Version System (CVS) report. The technique is not supported by a tool and therefore, applicability on large applications is difficult.

*Formal concept analysis of execution trace:* The approach is only semi-automatic and is dependent on the way use cases are designed. The effectiveness of approach is dependent on granularity of use cases.

*Clone detection:* The technique is very easy to use as it requires the source code as input and is supported by a tool. The tool is able to detect the clones existing in the code. Clone detection as a technique for identification of crosscutting concern is dependent definition of clone. If the size of clone to be searched for, in the source code, is very large then the probability of finding the code clone is very low and if the size of clone is taken to be very small then large number of clone will be detected and this will
not find the actual crosscutting concerns.

Event trace: The approach is quite effective as it is easy to use and requires the execution trace of the programs. The basic problem with this approach is the size of data pool of execution trace required to identify the patterns in method call relationship. It is not clear how many execution traces are required to find all the patterns in the method calls. The approach is also supported by an automated tool.

3.4 Conclusions

The existence of crosscutting concerns adds to the complexity of software design and therefore, it is responsible for fault proneness. The number of techniques for identification of such concerns have been proposed in literature. This chapter provides a study on the comparison of techniques for identification of crosscutting concerns. As it is apparent that not all techniques can be applied to all the scenarios therefore, the knowledge of applicability of techniques is required. The work presented in this chapter was published in [68].