Modeling, Measuring, and Modularizing
Crosscutting Concerns

A Dissertation Proposal
Presented to
The Faculty of the Department of Computer Science at
Columbia University

In Partial Fulfillment of
The Requirements for the Degree of

Doctor of Philosophy in
Computer Science

By Marc Eaddy
eaddy@cs.columbia.edu

Alfred Aho, Advisor
aho@cs.columbia.edu

Department of Computer Science
Columbia University
1214 Amsterdam Avenue
Mailcode 0401
+1 212 939 7077

November 28, 2006
Abstract

Modularity is essential for creating evolvable software. Traditional programming languages limit the level of modularity that can be achieved because they provide only one way to structure the program, e.g., as a hierarchy of types. Regardless of how a program is structured, some requirements, features, properties, or other types of concerns of the program cannot be modularized because they cut across the program’s structure. These are called crosscutting concerns. The presence of crosscutting concerns results in programs that are difficult to understand and reason about, to break into manageable pieces, to reuse, and to evolve.

The nature of crosscutting and how it affects modularity is poorly understood, making it difficult to validate and compare modularity techniques. A contribution of my thesis will be a formal set theoretic model for crosscutting concerns and a collection of software metrics and methodology for measuring crosscutting. I will use the metrics in a case study to quantify the degree of crosscutting present at each stage of refinement of a medium-sized program.

The results from this case study will clarify the strengths and weaknesses of existing modularization approaches. I will use this insight to design Wicca#, a new general-purpose programming language that better supports modularizing crosscutting concerns. Current modularization approaches are biased towards modularizing some crosscutting concerns but not others, limiting their potential. Paradoxically, these approaches may actually hinder extensibility, understandability, and modular reasoning. Wicca# addresses this by supporting side classes, which provide powerful extensibility without sacrificing modularity, and statement annotations, which improve the modularity of so called heterogeneous concerns.
Contents

1. Introduction ..................................................................................................................... 1
  1.1 Problem Statement ....................................................................................................... 2
  1.2 Thesis Statement ......................................................................................................... 2
  1.3 Proposed Solution ....................................................................................................... 2
  1.4 Evaluation .................................................................................................................. 3
  1.5 Anticipated Contributions ......................................................................................... 4
  1.6 Dissertation Proposal Organization ......................................................................... 5

2. Concern Model .................................................................................................................. 5
  2.1 Basic Terminology ..................................................................................................... 5
  2.2 Scattering and Tangling ............................................................................................. 5
  2.3 Program Specification Domains ................................................................................ 6
  2.4 Concerns ................................................................................................................... 6
  2.5 Crosscutting Concerns .............................................................................................. 7
  2.6 Component Aggregation ............................................................................................ 8
  2.7 A Running Example .................................................................................................. 8

3. Crosscutting Metrics ....................................................................................................... 9
  3.1 Notation ....................................................................................................................... 9
  3.2 Concentration (CONC) ............................................................................................. 10
  3.3 Degree of Scattering (DOS) ...................................................................................... 11
  3.4 Average Degree of Scattering (DOS) ....................................................................... 12
  3.5 Dedication (DEDI) .................................................................................................. 13
  3.6 Degree of Focus (DOF) ............................................................................................ 13
  3.7 Average Degree of Focus (DOF) .............................................................................. 14
  3.8 Degree of Tangling (DOT) ....................................................................................... 15
  3.9 Average Degree of Tangling (DOT) ........................................................................... 16
  3.10 Validity of the Crosscutting Metrics: An Extended Example ................................ 16
      Refactoring #1: Submodule Refactoring .................................................................... 17
      Refactoring #2: Virtualization Refactoring .............................................................. 18
  3.11 Summary .................................................................................................................. 19

4. Case Study: Goblin ......................................................................................................... 19
  4.1 Goblin – A Platform for 3D Applications ................................................................ 20
  4.2 Evolutionary Phases ................................................................................................. 20
  4.3 Data Collection and Analysis Procedure ............................................................... 20
  4.4 Threats, Verification, and Validation ....................................................................... 21
  4.5 Preliminary Results ................................................................................................... 21

5. Wicca# ............................................................................................................................. 22
  5.1 Motivation .................................................................................................................. 22
  5.2 Side Classes: Powerful and Disciplined Class Refinement ................................... 22
  5.3 Statement Annotations: Fine-Grained Metadata for C# ........................................ 23
  5.4 The Wicca System .................................................................................................... 23

6. Related Work ................................................................................................................ 24
  6.1 Requirements Engineering ....................................................................................... 24

  Concern Modeling ............................................................................................................. 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concern Identification and Assignment</td>
<td>25</td>
</tr>
<tr>
<td>6.2 Software Metrics</td>
<td>26</td>
</tr>
<tr>
<td>6.3 Programming Languages</td>
<td>27</td>
</tr>
<tr>
<td>7. Research Plan</td>
<td>28</td>
</tr>
<tr>
<td>7.1 Statement of Work</td>
<td>28</td>
</tr>
<tr>
<td>7.2 Timeline</td>
<td>28</td>
</tr>
<tr>
<td>8. Summary</td>
<td>29</td>
</tr>
<tr>
<td>Bibliography</td>
<td>29</td>
</tr>
<tr>
<td>Appendix A. Definitions</td>
<td>33</td>
</tr>
<tr>
<td>Appendix B. Detailed Measurement Example</td>
<td>35</td>
</tr>
</tbody>
</table>
1. Introduction

Modularization is the act of breaking down a software program into “logically self-contained and discrete parts” (i.e., modules) [3] to achieve the following:

To make complexity manageable;
   to enable parallel work; and
   to accommodate future uncertainty.

   – Baldwin and Clark [6]

There is abundant evidence that well-modularized programs are easier to understand, implement, test, maintain, and evolve, thus leading to a reduction in overall software cost [34]. In recognition of this fact, programming languages have supported modular programming in various forms since the very first languages. Each language provides its own module system, which allows a software developer to create, use, and extend modules, and compose modules to create programs. Each module system supports a particular primary decomposition technique (structural, hierarchical, functional, etc.) for partitioning a program into modules, where a module may have many different incarnations (functions, procedures, subprograms, subroutines, coroutines, abstract data types, classes\(^1\), virtual types, modules ala Module-2, aspects, features, traits, mixins, packages, libraries, etc.).

It has long been observed that for a particular language and decomposition technique, some changes to a program are easy to make and others hard [47]. The canonical example of this is the expression problem [49] [47] [45] [14]. Consider a parser program that parses a simple programming language consisting of data types and operations over those types. Depending upon the programming language that the parser is written in and how the parser is organized, it may be easy to add new data types to our simple language but hard to add new operations, or vice versa. The expression problem also commonly manifests when a program needs to satisfy nonfunctional requirements such as performance, adaptability, debuggability, extensibility, maintainability, reliability, scalability, security, and supportability [46].

Interestingly, it does not matter which language or decomposition technique is used\(^2\): a feature, requirement, or more generally, a concern\(^3\), of the program that is hard to change will always exist. The concern may be hard to change because it requires multiple locations in the program to be modified, in which we call it a crosscutting concern, i.e., the implementation of the concern crosscuts the program. Another reason the change may be hard is that it negatively impacts other concerns.

The implementation of a crosscutting concern is not “logically self-contained” or “discrete” and therefore cannot be modularized. Indeed, the implementation is distributed, which makes other parts of the system less self-contained and therefore less modular. Consequently, crosscutting (i.e., the presence of crosscutting concerns) reduces a program’s overall modularity, making it harder and more costly to understand, develop, maintain, and evolve. I refer to this as the crosscutting problem\(^4\).

---

\(^1\) I use the terms “type” and “class” interchangeably.

\(^2\) This phenomenon is sometimes referred to as the “tyranny of the dominant decomposition” [45].

\(^3\) Informally, a concern is “any matter of interest in a software system” [44].

\(^4\) The expression problem is just one example of the crosscutting problem.
1.1 Problem Statement

Modularity is significantly limited by the presence of crosscutting concerns. Many new languages and modularization approaches have been proposed to address the crosscutting problem. Their main goal is to “modularize crosscutting concerns” [41]. While there is a growing body of (mostly anecdotal) evidence that suggests that these alternative decomposition techniques improve modularity [51] [15] [10], current techniques have significant drawbacks including the inability to modularize some crosscutting concerns (see my paper, [20]), complex language syntax and semantics, and the tendency to reduce a program’s understandability [13] [4] and its ability to evolve [48] [42] [28].

The first drawback stems from a lack of understanding of the crosscutting phenomenon and how it affects modularity. Indeed, no widely-accepted definition for “concern” or “crosscutting concern” exists [36]. While metrics have been proposed to measure crosscutting, they are too coarse-grained to provide a comprehensive picture of crosscutting or to distinguish the subtle differences between modularization approaches. The end result is that the research community is fractured over what is or is not a crosscutting concern, different modularization approaches are unintentionally biased towards modularizing some concerns but not others [20], quantitative comparison of approaches are scarce (for an exception, see [10]), and most importantly, it is impossible to quantify the impact of crosscutting on modularity [22].

The remaining drawbacks are (somewhat subjective) issues of language design. Many of the languages that support alternative decomposition techniques are unnecessarily complex and introduce concepts and constructs that are unnecessarily inconsonant with traditional language concepts. The use of these constructs may violate assumptions and make the program harder to reason about [13] [4] or may create tight couplings between modules that make the program harder to evolve [48] [42].

1.2 Thesis Statement

The main claim of this thesis is that crosscutting concerns reduce modularity (the crosscutting problem) and that approaches for modularizing crosscutting concerns are unnecessarily limited. A higher level of modularity means that the program is less complex, easier to break into pieces that can be developed and tested independently and in parallel, and more extensible.

I claim that the expression problem is a special case of the crosscutting problem and that solutions to the former are not necessarily applicable to the latter.

I claim that a relation can be established between software requirements and software artifacts and that crosscutting can be quantified with respect to this relation.

I claim that nontrivial programs are likely to exhibit significant crosscutting and that refactoring these programs to reduce crosscutting will significantly improve their modularity.

I claim that current popular alternative decomposition techniques, such as AspectJ-like languages [27], modularize a limited amount of crosscutting [20] [30] and are therefore limited in the level of modularity they can achieve. Furthermore, these languages exhibit language design issues that limit modularity. These limitations must be overcome if we hope to achieve higher levels of modularity than can be achieved using traditional programming languages.

1.3 Proposed Solution
For my thesis, I will develop a formal model, a set of crosscutting metrics, and a rigorous methodology for quantifying crosscutting. **This is 90% complete.**

The centerpiece of my thesis will be an evolving case study of a medium-sized program. The case study will help validate my crosscutting metrics and quantification methodology, as well as provide insight into the nature of crosscutting, and concrete evidence that links crosscutting to reduced modularity. I will take the program through a series of restructurings, using three different modularization approaches (a traditional OO refactoring, an aspect-oriented refactoring, and refactoring using Wicca#), to attempt to improve its modularity. The insights and measurements I obtain will allow me to qualitatively and quantitatively compare the approaches. **The initial phase is 50% complete.**

My prior research has helped clarify the language deficiencies efficiencies I outlined in the previous section. With the help of this research and the insights gained from the case study, I will develop a new general-purpose programming language called Wicca#, which will attempt to address these deficiencies and provide a better solution for modularizing crosscutting concerns. **The design and implementation of Wicca# is 50% complete.**

### 1.4 Evaluation

This section explains how I plan to prove my claims and measure success. I have already completed an unpublished experiment where I solved the expression problem using AspectJ™ [27], Wicca#, and Sing#. This allowed me to verify what others have observed [47], that parallel development and extensibility can be improved by using alternative decomposition techniques. My case study will provide empirical and quantitative evidence that a considerable amount of heretofore undetected crosscutting code exists in the subject program and that significantly erodes modularity.

I have already published a paper arguing that AspectJ-like languages ineffectively modularize so-called “heterogeneous concerns” and that statement annotations are an elegant approach to solving this problem [20]. Since then I have implemented support for statement annotations in Wicca#. During the initial phase of my case study, I proved the worthiness of statement annotations by using them as a basis for my novel crosscutting quantification methodology.

I have almost completed the development of my crosscutting model and metrics and have used them to paint an initial picture of the crosscutting present in the subject program. I have garnered several insights from these preliminary results including:

- **Crosscutting metrics reveal more insight into the cost of change than traditional software metrics.** For example, whereas traditional metrics can things like “class A hard to change”, crosscutting metrics can say “feature A is hard to change”. The reason is that traditional metrics measure the physical properties of a program whereas crosscutting metrics measure the relationship between logical and physical properties of a program. Since changes to a program often originate as change requests to logical entities (e.g., requirements), crosscutting metrics can more directly measure their cost.

- **The average degree of focus metric (DOF) precisely summarizes separation of concerns.** Traditional metrics only measure physical instead of logical entities and therefore cannot measure separation of concerns. As the concept of separation of concerns is central to modular programming, it is possible that $\overline{\text{DOF}}$ can be used as a novel measure of modularity.
• The more a concern is likely to change, the more important it is to modularize that concern. This echoes the modularity goal, “the ease of making a change … should bear a reasonable relationship to the likelihood of the change being needed” [35]. This is especially important for crosscutting concerns because it appears that researchers may not appreciate that a concern may be intentionally crosscutting because it is never expected to change (“what if” scenarios aside).

• The primary concerns of a program are often crosscutting. Thus a new programmer can better understand the program by examining its crosscutting concerns. A corollary observation is that nonfunctional requirements are not necessarily crosscutting.

• The concept of crosscutting is subtly dependent on several factors including what classifies as a concern, concern formality and granularity, what kind of software artifact the concern can crosscut, and the implementation language. Many different and contradicting conclusions about the amount of crosscutting in a program can be arrived at by making subtle changes to these factors. This underscores the importance of formalizing a model, metrics, and rigorous methodology for quantifying crosscutting. It also highlights some of the limitations of current approaches in the literature.

I will consider the initial phase of the case study to be a success when the paper that details the above results is accepted for publication. Success of my Wicca# language hinges on empirical evidence, i.e., anecdotes from refactoring the subject program; quantitative evidence obtained using the crosscutting metrics; qualitative arguments; and ultimately, on a paper about Wicca# being accepted for publication.

1.5 Anticipated Contributions

I anticipate the following contributions and follow on work:

• New object-oriented language constructs that provide direct support for modularizing crosscutting concerns and are more effective and elegant than current approaches. Researchers will hopefully see how their languages can be simplified using my approach, or how their language constructs can be seamlessly integrated with existing programming languages (see Steimann [41] for a convincing argument).

• A formal model, metrics, and rigorous methodology for quantifying crosscutting concerns, and guidelines on how to profitably apply them to diagnose structural problems, guide refactoring decisions and compare refactoring alternatives. Researchers will hopefully begin using these to clarify their arguments on crosscutting, justify language design choices, and compare alternate approaches. In addition, interesting future work would be to correlate the crosscutting metrics with external quality metrics such as cost (using the Net Option Value [43] technique, for example) and error proneness [33].

• A rigorous case study that measures crosscutting concerns, reveals insights into the nature of crosscutting, and helps justify the need for better ways to separate concerns. Researchers will hopefully try to replicate my results.

• [Possible] New metrics for measuring modularity and separation of concerns, and new insights to better understand and manage both. My research will hopefully illuminate the relationship between crosscutting and modularity.
1.6 Dissertation Proposal Organization

In Section 2, I provide a model for crosscutting concerns and formally define the terms “concern”, “crosscutting concern”, “scattered”, and “tangled”. The model serves as a basis for the crosscutting metrics I define in Section 3. I will use the metrics to measure the crosscutting concerns for an evolving case study, which I describe in Section 4. In Section 5, I explain how the insights gained from the case study will help complete the design of a new language called Wicca#. Related work is reviewed in Section 6. Section 7 outlines my research plan and provides a detailed work breakdown. Section 8 concludes.

2. Concern Model

In this section, I present a formal model for concerns based on set theory. I formally define the terms “concern”, “crosscutting concern”, “scattered” and “tangled”, which are standard fare in the aspect-oriented programming literature but have only been informally defined.

2.1 Basic Terminology

Consider two arbitrary domains, $S$ (the source domain) and $T$ (the target domain), and a relation $R$ (the ST relation) that relates elements of $S$ to elements of $T$ ($R:(s,t), s \in S, t \in T$). There may be multiple source and target domains (these sets are represented by $S$ and $T$ respectively) and $S$ and $T$ represent elements from this set ($S$: $S$ and $T$: $T$). We call $R$ a total relation if every element of $S$ and $T$ appears in $R$ and a partial relation otherwise.

The selection function $\sigma_{\text{source}}^Q(R)$ selects all pairs from $R$ whose first component is in the query set $Q$. Similarly, $\sigma_{\text{target}}^Q(R)$ selects all pairs from $R$ whose second component is in $Q$. When the relation $R$ and the component position are obvious from the context, we use the abbreviated form $\sigma_q$, or $\sigma_q$ (when $Q$ has only one element $q$).

The projection function extracts the set of source or target elements from $R$ or a subset of $R$, written as $\pi_D(R)$, where $D$ is the domain (e.g., $S$ or $T$).

2.2 Scattering and Tangling

While the concepts “scattering” and “tangling” are vital to aspect-oriented programming, they are loosely defined (although see [22]). Using the model above, I define these concepts formally as follows.

Element $s$ of $S$ is localized with respect to $T$ if it is related to only one element of $T$.

Element $s$ of $S$ is scattered with respect to $T$ if it is related to multiple elements of $T$. 


Elements $s_j$ and $s_k$ of $S$ are tangled if they are related to the same element in $T$.

If an element of $S$ or $T$ is not present in $R$ (i.e., $R$ is a partial relation) it is absent.

In Section 3, I define a set of metrics for measuring the degree of scattering and tangling.

2.3 Program Specification Domains

I will now explain how the model applies to software programs. In this context, the source or target domain is a description of a program, i.e., a program specification (or simply specification). For example, a program’s software requirements specification (SRS) is a domain whose elements consist of individual requirements, e.g., "the program must display 3D objects." Another example is a program’s C# class specification, which is a domain whose elements consist of individual classes and their corresponding C# source code, e.g., the Display class. (My thesis relies heavily on a program’s statement specification, which consists of every statement from the source code of the program.)

A specification is called an implementation specification if it is executable, i.e., can be compiled and executed directly. For example, the C# class specification is executable. In contrast, an abstract specification, such as an SRS, is not executable.

It is not necessary for a specification, or a particular element of that specification, to be physically realized [11], i.e., the specification may be undocumented, incomplete, or out-of-date. However, the specification should be well defined. This is discussed further in Section 4.4: Threats, Verification, and Validation.

2.4 Concerns

I use the following working definitions for the term “concern”:

**Informal Definition (also see Appendix A)**

A concern is “a human-oriented expression of computational intent.” [9]

**Formal Definition**

A concern is an element from a program’s abstract specification.

We will call the elements from a program’s abstract specification, concerns, and the elements from a program’s implementation specification, components. When $R$ relates concerns to components it is called an “implemented by” relation or a concern assignment [9]. A concern slice for concern $s$ is the set of pairs from $R$ that have $s$ as their first component (i.e., $\sigma_s$).

All the components related to a concern may be referred to loosely as the components of the concern or the concern’s components. All the concerns related to a component may be referred to loosely as the concerns of the component or the component’s concerns. When discussing a particular concern $s$ and related component $t$ (e.g., $t \in \pi_s \sigma_s$), we may describe their relationship informally as “component $t$ implements concerns $s$” or “concern $s$ is implemented by component $t$” (the modifier “partially” is assumed).
A program may have many abstract specifications and there may be a specialized version of the term “concern” for each. For example, a “feature”, “requirement”, “goal”, “design rule”, “coding guideline”, “policy”, etc., can all be considered concerns from their respective specification. Combining the two terms, e.g., “feature concern”, “requirement concern”, clarifies the kind of concern being referred to.

Concerns can be general or specific, or anywhere between. General concerns such as “performance”, “logging”, and “debugging”, may exist in the abstract specifications of many programs. Unless otherwise specified, the reader should assume a concern applies only to a specific program.

2.5 Crosscutting Concerns

The central tenant of aspect-oriented programming is to “modularize crosscutting concerns”. However, there is no widely accepted formal definition for a “crosscutting concern”. I offer the following definition:

A crosscutting concern is a scattered concern; i.e., a concern implemented by multiple components.

This definition equates crosscutting and scattered. Some authors equate crosscutting with tangled and scattered. However, there are examples of concerns that are generally considered crosscutting which are not tangled (e.g., the “tracing concern”). Gregor Kiczales extends my definition by defining a crosscutting concern as a concern that is scattered and hard to modularize using the programming language that the program was written in. My definition is less restrictive because it is difficult to quantify “hard to modularize” and I do not want to preclude traditional refactorings that can reduce scattering. My case study aims to give credence to this viewpoint.

To rephrase the example from Section 2.3, a requirement concern is crosscutting with respect to the class implementation specification if it is related to more than one class. Another way of saying this is: “the requirement is crosscutting at the class implementation level.” It is critical to specify the target specification that the concern crosscuts. For example, a requirement may crosscut at the class level but not the method level. Or a requirement that is crosscutting in a Java implementation specification may not be crosscutting in an AspectJ implementation specification.

Consider a requirement concern such as “the program must run on Windows and Unix,” a.k.a. the portability concern, and suppose that it was added after the program was written. If the program was originally implemented in C++, the code associated with the portability concern will likely be scattered because multiple parts of the program would need to be modified to port the program to another platform. This is because C++ is not portable in general.\(^5\) If the same program was originally written in Java, where portability is a language feature, less portability code is required which leads to less scattering of the portability concern. Clearly, it would be easier to satisfy the new portability concern if the program had been written in Java as opposed to C++. This underscores that crosscutting is defined only with respect to a concrete target specification.

\(^5\) Consider porting a 32-bit program to 64-bit, or a program that uses threads, graphics, or networking.
This example brings up another issue: if the portability concern was known \textit{a priori}, one can rightly argue that it is possible to modularize it in C++, using the \textit{façade design pattern} \cite{25}, for example. This is exactly what one would expect from a well-designed C++ program. This illustrates that one implementation of a program may contain more crosscutting code than another, even when the language is the same. My hypothesis is that a good deal of crosscutting concerns, once identified, can be modularized without resorting to special languages or constructs; but that constructs that directly support modularizing crosscutting concerns can do so more effectively and elegantly. This is the same argument for why C++ is better than C, even though one can write any C++ program in C: C++ directly supports object-orientation, C does not. I will give a detailed example of this in Section 3.10.

Crosscutting is a general phenomenon that is present in many non-computer science domains. For example, consider $S$ to be the domain of \textit{business functions} of a company, $T$ to be the set of \textit{departments} of the company, and $R$ to be a “required by” relation. In this model, the \textit{accounting business function} would be crosscutting because it is required by multiple departments. That is, each department requires accounting of its expenditures and profits. In contrast, the \textit{marketing business function} is required by very few departments (namely, the \textit{sales department}). Consider another example when $S$ is a set of \textit{municipal services} (sewage, gas, electrical, cable, air conditioning, etc.), $T$ a set of \textit{building floor plans}, and $R$ is a “required by” relation. In this context, all municipal services are crosscutting.

The definition of crosscutting concerns above captures the generality of the crosscutting problem and encourages solutions to be shared across domains. This is in contrast to other models proposed (for example, \cite{40} \cite{50}), which apply only to the software domain.

2.6 Component Aggregation

So far we have focused on the case when $R$ is a concern assignment. Consider the “aggregates” relation $R_{\text{Super-Sub}}$ between components in source domain $\text{Super}$ and components in target domain $\text{Sub}$. A component $\text{super}$ is said to aggregate a component $\text{sub}$ if the software artifact represented by $\text{super}$ aggregates the one represented by $\text{sub}$. For example, when $\text{Super}$ is the method specification domain and $\text{Sub}$ is the statement specification domain, a method aggregates a statement if the statement is in the source code of that method. We refer to $\text{sub}$ as a \textit{subcomponent} of $\text{super}$. Therefore, a statement is a subcomponent of a method. Unlike a concern assignment which allows a many-to-many relationship between concerns and components, the aggregates relation is a one-to-many relation where each subcomponent must be associated with exactly one super component ($|\pi_{\text{Super}}(\sigma_{\text{sub}}(R_{\text{Super-Sub}}))| = 1$).

The aggregates relation allows a concern assignment for a super component to be inferred from the assignment for its subcomponents. In other words, \textbf{a component aggregates the concern assignments of its subcomponents}.

2.7 A Running Example

Consider a program written in C that allows the user to manipulate shapes on a screen. Some of the requirement concerns for the program are the following:

- \textit{Display Shapes} – Shapes are drawn on the screen ($r1$)
- \textit{Manipulate Shapes} – Shapes can be created ($r2$), moved ($r3$), and rotated ($r4$),
• **Allowed Shapes** – The shapes allowed are circles \((r5)\) and rectangles \((r6)\)

The original program is composed of a single file, `main.c`, which consists of 10 functions including the functions named `main (f1)`, `create_shape (f2)`, `draw_shape (f3)` `translate_shape (f4)`, and `rotate_shape (f5)`. The remaining functions handle initializing the graphics library and interacting with the user. The original implementation has shape-specific creation code in the `create_shape` function, drawing code in the `draw_shape` function, etc.

Figure 1 shows a visualization of the concern assignment relation \(R\) between the requirements specification domain (the concern domain) \(S\) and the function specification domain \(T\).

\[
R = \{(r1, f3), (r2, f2), (r3, f4), (r4, f5), (r5, f2), (r5, f3), (r5, f4), (r5, f5), (r6, f2), (r6, f3), (r6, f4), (r6, f5)\}
\]

Figure 1. Concern assignment example.

The concern slice for requirement \(r5\) (support for circles) is \(\sigma_{r5} = \{(r5, f2), (r5, f3), (r5, f4), (r5, f5)\}\). The functions that implement \(r5\) are \(\pi_T(\sigma_{r5}) = \{f2, f3, f4, f5\}\). By definition, requirements \(r5\) and \(r6\) are crosscutting at the function level because they are (partially) implemented by multiple functions. Requirements \(r1\)–\(r4\) are localized. Target component \(f1\) (the main function) is absent because it does not implement any requirement (it exists only to satisfy an implicit programming language requirement).

The file specification domain \(P\) contains one element (\(main.c\)) which aggregates all the functions in the function specification domain \(T\), so \(\pi_P(\sigma_{main.c}) = \{f1, f2, f3, f4, f5, \ldots\}\). The aggregation relationship \(R_{PT}\) allows us to infer the concern assignment \(R_{SP}\) for the file specification: \(R_{SP} = \{(r1, main.c), (r2, main.c), (r3, main.c), (r4, main.c), (r5, main.c), (r6, main.c), \ldots\}\).

### 3. Crosscutting Metrics

In this section I will specify a set of crosscutting metrics. The equations in Section 2 provide a formal interpretation of scattering and tangling. However, they are not useful as software metrics because they do not indicate degree, making it difficult to quantify changes to the scattering and tangling of a particular concern and to compare concerns. To address this issue, I will leverage Wong and colleagues’ concentration and dedication metrics, which were initially designed to aid component comprehension [50].

#### 3.1 Notation

Consider a program with concern domain \(S\) and two target implementation specifications: \(C\), an unspecified implementation specification (the component specification), and \(A\), the assignment implementation specification (or simply, assignment specification), where \(C\) aggregates \(A\). \(C\) is a formal parameter of the measurement equation; the actual component specification must be
specified before the measurement is performed. This allows us to measure scattering at different abstraction levels such as file-, class-, and method-level.

The assignment specification is the target implementation specification where the concern assignment was performed, the elements of which are referred to as assigned elements. For example, Wong et al. assigned concerns at the basic block level [50] which is common when concern assignment is done using execution tracing; Revelle et al. [36], Carver and Griswold [11], and Painter and Coppit [37] assigned concerns at the character level; and Lai and Murphy [29] assigned concerns at the token level. For my case study I assign concerns at the statement level. The assignment specification represents the highest level of detail obtainable by the scattering and tangling metric.

One requirement is that all elements of the assignment specification $A$ be aggregated by elements of the component specification $C$. For example, choosing a method specification for $C$ and a statement specification for $A$ satisfies this requirement because every statement is a subcomponent of a single method. If an assigned element is not part of any component in $C$, then $R_C$ is not a total relation and the scattering and tangling metrics will not be meaningful. A similar argument applies to $R_A$.

We assume the following exist: a suitable concern assignment $R_{SA}$, which relates concerns in $S$ to target elements in $A$; a containment relation $R_{CA}$, which relates components in $C$ to target elements in $A$; an aggregation relation $R_{CA}$ which relates components to assigned elements; and a concern assignment $R_{SC}$, derived from $R_{CA}$ and $R_{SA}$, that relates concerns in $S$ to components in $C$. $t_c$ is a component from the component specification $C$. I define the following notation:

- $S_{t_s}$ is the set of concerns related to assigned element $t_s$, i.e., $\pi_S(\sigma_{t_s}(R_{SA}))$
- $A_{s}$ is the set of assigned elements (e.g., statements) related to concern $s$, i.e., $\pi_A(\sigma_s(R_{SA}))$
- $A_{t_{c}}$ is the set of assigned elements contained by component $t_{c}$, i.e., $\pi_A(\sigma_{t_c}(R_{CA}))$
- $S_{t_{c}}$ is the set of concerns related to component $t_{c}$ by aggregating the concerns related to the assigned elements ($A_{t_{c}}$) contained by $t_{c}$, i.e., $\pi_S(\sigma_{t_c}(R_{SC}))$
- $A_{s \cap t_{c}}$ is the intersection of $A_{s}$ and $A_{t_c}$, i.e., the set of assigned elements contained in $t_{c}$ that implement $s$.
- $V_{t_{c}}$ is the set of virtual assigned elements related to component $t_{c}$, i.e., $\sum_{s \cap t_{c}} A_{s \cap t_{c}}$. For example, if $t_{c}$ is a method with one statement ($A_{t_c} = \{t_1\}$) related to three concerns ($|S_{t_1}| = 3$), then $V_{t_c} = \{t_1, t_1, t_1\}$.

### 3.2 Concentration (CONC)

Informally, concentration measures how much of the implementation of a concern is contained by a particular component [50]. Formally, we define concentration as follows:

$$\text{CONC}_{st_c} = \frac{|A_{s \cap t_{c}}|}{|A_{s}|}$$

Concentration (1)
The mean concentration ($\overline{\text{CONC}}_s$) is the average of the concentration over all the components $C$:

$$\overline{\text{CONC}}_s = \frac{1}{|C|} \sum_{t_c}^C \text{CONC}_{st_c}$$  \hspace{2cm} (2)

$$\overline{\text{CONC}}_s = \frac{1}{|C|}$$  \hspace{2cm} \text{Mean Concentration}  \hspace{2cm} (3)

### 3.3 Degree of Scattering (DOS)

Degree of scattering (DOS) (also, degree of crosscutting) is a measure of how distributed the implementation of a concern $s$ is across all the components $C$ of the program. We look at how closely the actual distribution approximates a baseline uniform distribution (this is the worst case). The baseline uniform distribution is one where concern $s$ is distributed uniformly across all the components, i.e., the mean concentration $\overline{\text{CONC}}_s$ is $1/|C|$. We then normalize the variance by scaling it to the variance of the ideal distribution, where the concentration is 1 for a single component and 0 for the rest. The result is the normalized degree of localization. Subtracting from one provides the degree of scattering, which is the complement of degree of localization:

$$\text{DOS}_s = 1 - \frac{\text{VAR}_{\text{ACTUAL}}}{\text{VAR}_{\text{IDEAL}}}$$  \hspace{2cm} (4)

where,

$$\text{VAR}_{\text{ACTUAL}} = \frac{1}{|C|} \sum_{t_c}^C \left( \text{CONC}_{st_c} - \overline{\text{CONC}}_s \right)^2$$  \hspace{2cm} (5)

$$= \frac{1}{|C|} \sum_{t_c}^C \left( \text{CONC}_{st_c} - 1 \right)^2$$  \hspace{2cm} (6)

For $\text{VAR}_{\text{IDEAL}}$, $\text{CONC}_{st_c} = 1$,

$$\text{VAR}_{\text{IDEAL}} = \frac{1}{|C|} \sum_{t_c}^C \left( \text{CONC}_{st_c} - \overline{\text{CONC}}_s \right)^2$$  \hspace{2cm} (7)

$$= \frac{1}{|C|} \left( \frac{|C|-1}{|C|^2} + \left( 1 - \frac{1}{|C|} \right)^2 \right)$$

$$= \frac{|C|-1}{|C|^2}$$  \hspace{2cm} (8)

Substituting, we have
The DOS metric satisfies the following properties:

- It is normalized (i.e., $0 \leq \text{DOS}_s \leq 1$) so that values lie along a continuum that ranges from completely centralized (0) to completely decentralized (1). The normalized values can be compared in a meaningful way.
- It is somewhat proportional to the number of components that implement the concern. Roughly, the more components that participate in the implementation of a concern (participating components), the more scattered the concern is. For example, a concern whose implementation is spread across three components is more scattered than when spread across two components, in general.
- Assuming the same number of participating components, DOS is somewhat inversely proportional to the concentration. That is, the more centralized the implementation of a concern is to a subset of the participating components, the less scattered the concern. For example, a concern whose implementation is spread across three components with a concentration of $\{0.40, 0.30, 0.30\}$ is more scattered than if the concentration were $\{0.90, 0.05, 0.05\}$.
- It is somewhat proportional to the number of assigned elements $A_s$ of the concern. That is, the more statements related to the concern, the more likely it is that they will be scattered.
- The value 1 is assigned if and only if the concern is uniformly distributed across all the components, that is, the concentration for each component is $1/|C|$. This indicates that the concern’s implementation is completely decentralized.
- The value 0 is assigned if and only if the concern is completely centralized (i.e., localized) in one component, that is, $\text{CONC}=1$ for one component and 0 for the rest.

It is important to consider how some common refactorings will affect DOS. The metric rewards refactorings that consolidate redundant code into functions. A high degree of scattering implies increased coupling. Standard techniques to reduce coupling, such as employing information hiding or reengineering the class hierarchy, have the desirable effect of also reducing scattering. Adding nonparticipatory components to the program decreases DOS, removing them has the opposite effect. Consolidating participatory components decreases DOS. The extreme case is to consolidate the entire program into one component, thus ensuring that the implementation of all the concerns is centralized ($\text{DOS}=0$). This is a great example of “gaming the equations”, and is counteracted by the degree of focus metric.

### 3.4 Average Degree of Scattering ($\overline{\text{DOS}}$)

Average degree of scattering ($\overline{\text{DOS}}$) is the average of the degree of scattering across all concerns:
\[
\text{DOS} = \frac{1}{|S|} \sum_{s} \text{DOS}_s \\
\text{Average Degree of Scattering} \quad (10)
\]

This metric is useful for comparing the amount of scattering in two different versions of the same program, e.g., the original and refactored versions, as I will do in Section 3.10. However, as DOS is heavily dependent on the number of program components, \(\text{DOS}\) is not as useful for comparing programs with vastly different numbers of component.

**3.5 Dedication (DEDI)**

Informally, dedication measures how much of a component is related to a particular concern [50]. It is defined formally as follows:

\[
\text{DEDI}_{t,s} = \frac{|A_{c \cap t}|}{|V_t|} \\
\text{Dedication} \quad (11)
\]

(Note: I made a slight improvement to the original DEDI metric from [50] to support the situation when a statement is related to multiple concerns.)

The *mean dedication* \(\overline{\text{DEDI}}_t\) is the average of the dedication over all the concerns \(S\):

\[
\overline{\text{DEDI}}_t = \frac{1}{|S|} \sum_{s} \text{DEDI}_{t,s} \quad (12)
\]

\[
\overline{\text{DEDI}}_t = \frac{1}{|S|} \\
\text{Mean Dedication} \quad (13)
\]

**3.6 Degree of Focus (DOF)**

*Degree of focus* (DOF) is a measure of the number of concerns related to (partially implemented by) a component and the degree to which they are related. We look at how closely the actual dedication approximates a *baseline uniform dedication* (this is the worst case). Recall that component \(t_c\) is composed of its assigned elements \(A_{i_c}\). The baseline uniform dedication is one where every assigned element of component \(t_c\) is related to every concern, i.e., the mean dedication \(\overline{\text{DEDI}}_t\) is 1/\(|S|\). We then normalize the variance by scaling it to the variance of the *ideal dedication*, where every assigned element is dedicated to only one concern. The result is the degree of focus:

\[
\text{DOF}_t = \frac{\text{VAR}_{\text{ACTUAL}}}{\text{VAR}_{\text{IDEAL}}} \quad (14)
\]

where,
\[
\text{VAR}_{\text{ACTUAL}} = \frac{1}{|S|} \sum_{s} \left( \text{DEDI}_{t,s} - \overline{\text{DEDI}}_{t} \right)^2 \\
= \frac{1}{|S|} \sum_{s} \left( \text{DEDI}_{t,s} - \frac{1}{|S|} \right)^2
\] (15)

For \( \text{VAR}_{\text{IDEAL}} \), \( \text{DEDI}_{t,s} = 1 \),

\[
\text{VAR}_{\text{IDEAL}} = \frac{1}{|S|} \sum_{s} \left( \text{DEDI}_{t,s} - \overline{\text{DEDI}}_{t} \right)^2 \\
= \frac{1}{|S|} \left( \frac{|S| - 1}{|S|^2} + \left( 1 - \frac{1}{|S|} \right)^2 \right) \\
= \frac{|S| - 1}{|S|^2}
\] (16)

Substituting, we have

\[
\text{DOF}_t = \frac{|S| \sum_{s} \left( \text{DEDI}_{t,s} - \frac{1}{|S|} \right)^2}{|S| - 1} \\
\text{Degree of Focus}
\] (17)

The DOF metric satisfies the following properties:

- It is normalized so that values lie along a continuum that ranges from completely unfocused (0) to completely focused (1).
- It is inversely proportional to the number of concerns related to the component. Assuming the number of assigned elements is unchanged, the more concerns the component implements the less focused the component is.
- Assuming the number of concerns related to a component stays the same, DOF is proportional to the dedication. For example, a method that implements two concerns with a dedication of \( \{ .9, .1 \} \) is more focused than one with a dedication of \( \{ .5, .5 \} \).
- A value of 1 is assigned if and only if the component implements only one concern. We say that the component is completely focused on the concern.
- A value of 0 is assigned if and only if the component implements every concern uniformly, i.e., every assigned element is related to every concern. We say that the component is completely unfocused.

### 3.7 Average Degree of Focus (\( \overline{\text{DOF}} \))

Average degree of focus (\( \overline{\text{DOF}} \)) is the average of the degree of focus across all components:
\[
\text{DOF} = \frac{1}{|C|} \sum_{i} \text{DOF}_i \quad \text{Average Degree of Focus} \quad (20)
\]

The DOF directly measures the separation of concerns in a program, that is, to what degree each concern is implemented by a separate component. When DOF is 1, every component implements only one concern, implying that the separation of concerns is absolute.

3.8 Degree of Tangling (DOT)

Degree of tangling (DOT) measures the number of target elements of \( s \) that are shared with other concerns. We define the following notation:

- \( D_s \) is the set of target elements related to concern \( s \) that are not shared by any other concern, i.e., \( \forall \sigma_x \neq \sigma_s, \pi_T(\sigma_s) \cap \pi_T(\sigma_x) = \emptyset \). In other words, the statements in \( D_s \) are dedicated to concern \( s \).

The degree of tangling then becomes the number of target elements dedicated to concern \( s \) divided by the total number of target elements that implement \( s \):

\[
\text{DOT}_s = \frac{|D_s|}{|\sigma_s|} \quad \text{Degree of Tangling} \quad (21)
\]

The DOT metric satisfies the following properties:

- It is normalized by the total number of target elements for \( s \), so that it ranges from 0 to 1, inclusively. Values lie along a continuum that ranges from completely untangled (0) to completely tangled (1).
- It is proportional to the total number of target elements for \( s \), so it is a relative measure as opposed to an absolute measure with respect to the whole program. This corresponds with the intuition that scattering is a global phenomenon of the program (e.g., “\( s \) crosscuts the program”) whereas tangling measures how much one concern interferes with other concerns.
- The value 1 is assigned when every target element of \( s \) is shared by another source element. This could mean that the two elements are actually the same or are tightly coupled.
- The value 0 is assigned when no target element of \( s \) is shared.
- DOS and DOT are orthogonal but not completely independent metrics. By orthogonal, I mean a source element can be scattered but not tangled and vice versa. However, as the example below shows decreasing tangling may result in increasing scattering and vice versa, i.e., the adjustment of one metric may give rise to an inversely proportional effect on the other.

DOT can be artificially increased or decreased by splitting or combining assigned elements. For example, the conditional expression of an \( \text{if} \) statement may be composed of multiple subexpressions, each related to a separate concern:
if (s1 || s2) { body1 }

Here, s1 and s2 correspond to Boolean variables associated with concerns $s_1$ and $s_2$, respectively. These concerns are tangled together at the statement level. However, the tangling is accidental since the if statement can be broken up into separate if statements thereby eliminating the tangling:

if (s1) { body1 }
else if (s2) { body1 }

Notice that tangling has been reduced at the expense of increasing DOS, especially since $body_1$ must be duplicated. Also notice that concerns are still tangled at the method level.

### 3.9 Average Degree of Tangling ($\overline{\text{DOT}}$)

Average degree of tangling ($\overline{\text{DOT}}$) is the average of the degree of tangling across all concerns:

\[
\overline{\text{DOT}} = \frac{1}{|S|} \sum_{s} \text{DOT}_s \quad \text{Average Degree of Tangling} \quad (22)
\]

This metric is useful for comparing the amount of scattering in two different programs or two different versions of the same program.

### 3.10 Validity of the Crosscutting Metrics: An Extended Example\(^6\)

One test of the goodness of a metric is that it meets our intuitive expectations; it should corroborate refactorings that are obviously good. Consider $C$ to be the function specification and $A$ to be the statement specification for the example from Section 2.7. A naïve implementation would result in decentralized and tangled logic for drawing, rotation, etc. different shapes, as illustrated by the $\text{draw}_\text{shape}$ function in Table 1.

<table>
<thead>
<tr>
<th>STATEMENTS</th>
<th>STATEMENT ID</th>
<th>IMPLEMENTED REQUIREMENT IDS ($S_{ids}$)</th>
</tr>
</thead>
</table>

\(^6\) This example is simply a restatement of the expression problem [49] with shapes instead of expressions.
Void draw_shape(Shape* s) {
    if (s->IsCircle) {
        Circle* c = (Circle*) s;
        ...10 more statements that draw a circle...
    } else if (s->IsRect) {
        Rect* r = (Rect*) s;
        ...8 more statements that draw a rectangle...
    }
}

Table 1 indicates that all 22 statements in draw_shape are associated with the drawing requirement (r1), 12 with the circle requirement (r5), and 10 with the rectangle requirement. Assume that create_shape, move_shape, and rotate_shape have the same makeup, except replace r1 with r2, r3, and r4 respectively. The average degree of scattering (DOS) is then 0.28 and the average degree of focus (DOF) is 0.24 (see Appendix B for detailed calculations).

DOS is low because r1, r2, r3, and r4 are localized (DOS=0) in functions draw_shape, create_shape, move_shape, and rotate_shape, respectively. However, the shape-specific code (r5 and r6) for drawing, creation, etc. is highly scattered as indicated by a DOS of 0.83 for each. The degree of scattering measurement agrees with the intuition that the shape-specific functionality is not centralized, but instead cuts across several functions. The high value indicates that adding support for a new shape would require changes to several functions.

Low DOF indicates the real problem: the attention of the four functions is each divided amongst three concerns. The measurement agrees with intuition because it is obvious that each function must implement shape-specific code for each supported shape. Adding more supported shapes would make the functions even more unfocused.

Now I will take the drawing program through a series of obviously good refactorings and show how the metrics correlate with our intuition.

Refactoring #1: Submodule Refactoring

An obviously good refactoring is to move the shape-specific code into separate shape-specific modules so that all that is left is the dispatch logic (i.e., if <shape-specific-type> then <shape-specific-function-call>). I will call this the submodule refactoring. This improves modularity because the shape-specific code is centralized in dedicated modules. This is an example of a refactoring that improves modularity by using the modularization mechanisms already available in the language. Table 2 shows the corresponding change to the draw_shape function.

<table>
<thead>
<tr>
<th>STATEMENTS</th>
<th>STATEMENT ID</th>
<th>IMPLEMENTED REQUIREMENT IDS (S_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s115</td>
<td>r1, r5</td>
<td></td>
</tr>
<tr>
<td>s116</td>
<td>r1, r5</td>
<td></td>
</tr>
<tr>
<td>s117-s126</td>
<td>r1, r5</td>
<td></td>
</tr>
<tr>
<td>s127</td>
<td>r1, r6</td>
<td></td>
</tr>
<tr>
<td>s128</td>
<td>r1, r6</td>
<td></td>
</tr>
<tr>
<td>s129-s136</td>
<td>r1, r6</td>
<td></td>
</tr>
</tbody>
</table>
void draw_shape(Shape* s) {
    if (s->IsCircle) {
        Circle* c = (Circle*) s;
        circle_draw(c);
    } else if (s->IsRect) {
        Rect* r = (Rect*) s;
        rect_draw(r);
    }
}

Table 2. After submodule refactoring.

The circle.c file contains the circle_draw, circle_create, etc. functions, and similarly for the rect.c file. Notice that the implementation of general requirement r1 is now split into three functions: draw_shape, circle_draw, and rect_draw. The new measurements are DOS = 0.76 and DOF = 0.35.

It appears that the degree of focus has improved at the cost of worsening the degree of scattering. The reason is that the general drawing, creation, etc. functionality is now decentralized because it is encapsulated and hidden by each shape file. Another cause for the increase is that the shape-specific functionality is now split into two parts: the majority is now in a dedicated shape-specific function while a small portion, i.e., the dispatch logic residue, remains in the original function. The improved DOF indicates that all the functions are more focused than before. The improvement in focus is even more pronounced at the file level.

Refactoring #2: Virtualization Refactoring

The lack of direct support for virtual dispatch makes it difficult to change the set of allowed shapes, and this highlights the real limitation of C. A perfectly reasonable solution for reducing this scattering is to refactor the program using C++, which has the virtual dispatch language construct that elegantly modularizes the dispatch logic (see Table 3).\(^7\)

<table>
<thead>
<tr>
<th>STATEMENTS</th>
<th>STATEMENT ID</th>
<th>IMPLEMENTED REQUIREMENT IDS (S(S_h))</th>
</tr>
</thead>
</table>
| void draw_shape(Shape* s) {
  s->draw();
} | s115          | r1, r5, r6                            |

Table 3. After virtualization refactoring.

This refactoring, which I will call the virtualization refactoring, is quite common when transitioning from an imperative to an object-oriented language. It is an example where modularization was improved using new modularization constructs, e.g., inheritance and the

\(^7\) I purposely rule out emulating virtual dispatch in C because this misses the point: while C may enable object-orientation, C++ directly supports it with constructs that are inarguably more elegant and easier to understand.
virtual keyword, provided by an extension to the original language. The program is now more extensible, because shape-specific implementation code can be modified and new shapes can be added easily without affecting the main program or the logic related to other shapes. The program is easier to develop because boilerplate dispatch logic is now handled automatically by the language. The shape operations are abstracted and shape-specific implementation details hidden, making the problem easier to understand and enabling parallel work.

The crosscutting metrics agree with this intuition. The new measurements are $\text{DOS} = 0.76$ and $\text{DOF} = 0.60$. Compared to the original program, the virtualization refactoring resulted in a 2.5 fold increase in DOF, indicating that the separation of concerns has improved.

Interestingly, these object-oriented refactorings have had the opposite effect of an aspect-oriented refactoring because concerns are now much more scattered than before ($\text{DOS}$ increased from 0.28 to 0.76). The degree of focus metric validates that the refactoring is actually beneficial which underscores the importance of using both metrics.

### 3.11 Summary

The degree of scattering metric measures the degree to which a concern’s implementation crosscuts a program. However, this metric alone is not enough to judge whether a particular implementation is more modular than another. The reason is that scattering can be artificially reduced by consolidating components, at the expense of defocusing the components and thereby reducing the overall modularity. The degree of focus metric provides the proper counterbalance to avoid these nonsensical refactorings.

A downside of these metrics is that they assume an accurate and complete concern assignment ($R$) exists. As I have mentioned, concern assignment is a hard problem. Manual approaches are tedious and error-prone and automated approaches are inaccurate. I discuss this problem in more detail in Section 6: Related Work.

I have shown how these metrics pass a sanity check for obviously good object-oriented refactorings, such as the submodule and virtualization refactoring. I hypothesize that these metrics will help quantify the benefits of more exotic modularization techniques including aspect-oriented decomposition. My case study will evaluate this hypothesis.

(Note: I am currently unsure of the importance of the degree of tangling metric, which is why I did not include it in the previous example. However, I will measure tangling in my case study and determine if the metric is effective for quantifying crosscutting concerns.)

### 4. Case Study: Goblin

The Goblin case study is central to my thesis. Early in my thesis work, it will provide me with a better understanding of crosscutting by answering the following questions: How much of a program’s source code implements a crosscutting concern (e.g., crosscutting code)? What impact does crosscutting code have on understandability, parallel development, maintainability, and evolvability? Later phases will allow me to determine how well AspectJ-like languages modularize crosscutting, helping me to motivate the design of Wicca#. A final phase will allow me to assess how well Wicca# modularizes crosscutting concerns.

A key to the validity of the case study is the rigorous quantification of crosscutting. This requires complete and accurate concern coverage.
4.1 Goblin – A Platform for 3D Applications

Goblin is a software platform for developing three-dimensional, augmented reality, and virtual reality applications and games [16]. It is written in roughly 14,400 C# source lines of code (SLOCs). Goblin was designed by me and developed by a team consisting of Hrvoje Benko, Erik Petterson, John Waugh, and myself, while I was a member of the Columbia Graphics and User Interfaces Lab.

Goblin is an ideal case study candidate because it

- Is medium-sized, which is representative of the programs I hypothesize are negatively affected by crosscutting.
- Implements a wide variety of concerns from user-level features such as a graphical user interface, to behavioral features such as 3D object motion and collision detection, to nonfunctional requirements such as logging, media acquisition, and performance.
- Has an up-to-date formal software requirements specification (SRS) [2] with numbered requirements. This should dramatically improve concern identification consistency [36] and legitimacy.
- Is a source code base that I am intimately familiar with. I wrote or rewrote about 30% of the source code. The remainder consists of a boilerplate 3D graphic framework distributed as source code by Microsoft. Reveille et al. list “understanding of the program” as the most important criteria for ensuring consistent concern location [36].

4.2 Evolutionary Phases

The case study will undergo several phases. During the initial phase, I will establish a baseline by identifying the concerns of the original program and measuring scattering, focus, and tangling. The implementation of feature and requirements concerns will be tagged using source code annotations. (This is 50% complete.) The goal of the normalization phase is to understand the limitations of traditional OO techniques for improving scattering and focus and modularizing crosscutting concerns. For the aspect-oriented (AO) refactoring phase, I will attempt to overcome the limitations of OO techniques by refactoring the program using an AspectJ-like language. A goal of this phase is to better understand the limitations, strengths, and weakness of existing AO techniques. Finally, for the Wicca# refactoring phase I will refactor the program using Wicca#. This will allow me to evaluate the effectiveness of Wicca# for modularizing crosscutting concerns.

4.3 Data Collection and Analysis Procedure

My goal is to determine the degree to which the implementation of Goblin’s features and software requirements (i.e., concerns) are scattered across the source base. I have the following hypotheses:

1. Concerns whose implementation is highly scattered (i.e., crosscutting) and tangled are harder to evolve,
2. Tangled crosscutting code is harder to modularize (e.g., using advanced separation of concerns techniques) than non-tangled,
3. The implementations of orthogonal (non-functional) requirements are less likely to be tangled, and therefore easier to modularize, than those of functional requirements.

4. A significant amount of crosscutting code cannot be effectively modularized using leading AOP approaches.

I will manually label each program element with the name of the concern(s) that it implements using C# attributes, hereafter referred to as concern annotations. One novel aspect of my approach is that I have modified a C# compiler to allow annotations to be attached to individual statements. Here is an example of a statement annotation for the “Input” concern:

```csharp
void OnFrameMove(Device device, float elapsedTime) {
   // [Concern("Input")]
   inputMapper.Update(elapsedTime);
}
```

(See Section 5.3 for a detailed description of statement annotations.)

My modified compiler, wsc, performs rudimentary annotation processing and statistics gathering of concern annotations. To ease the annotation burden, wsc supports annotation inheritance [31]. Program elements can be annotated at any level of abstraction and these annotations can be overridden by lower levels. For example, if most of the statements inside a method body are associated with one concern, the method can be annotated with the name of that concern. Individual exceptions can be annotated explicitly at the statement level. The annotation processor also performs some rudimentary consistency checks.

I plan to compare the statistics for the informal feature specification (80% complete) and the formal requirements specification. This will mark the first time that two different concern source specifications are compared.

### 4.4 Threats, Verification, and Validation

In this section I briefly describe some of the threats that can confound my experiment. Determining measurement validity means determining that 1) the crosscutting metrics measure what they purport to measure, 2) concerns are correctly identified and associated with the code appropriately, and 3) the instrument measures the data correctly. I address the first threat by hand checking many of the measurements to ensure they agree with expectations by adding consistency checks to the annotation processor. The second threat is addressed by using a formal requirements specification for S and by my familiarity with the program, both of which should improve concern assignment consistency and accuracy [36] and reduce the possibility of experimenter bias where the experimenter only identifies the concerns that can be easily modularized using their technique. I also plan to exclude certain features and requirements from the program using only the concern assignment as my guide. Successfully compiling and running the modified program will lend confidence in the accuracy of the concern assignment.

### 4.5 Preliminary Results

Further research on this subject has revealed improvements that I need to make to my metrics and methodology, including using a formal requirements specification as the source for the
concerns to improve consistency [36], specifying the concern hierarchy [36], following existing guidelines [31] for annotation inheritance, ensuring member annotations are consistent at both declaration and use [40], and using more reliable criteria for feature concern identification based on the notion of minimal subsets and minimal increments [11].

5. Wicca#

This section describes the motivation and design decisions behind Wicca#, an extension to the C# language, and gives an informal description of the language. The language features I present here are based on my current work on side classes and statement annotations, which is still evolving. I expect to learn more about the efficacy of these features, and possibly discover the need for changes or new features, as part of my ongoing case study.

5.1 Motivation

Advanced Separation of Concerns (ASOC) is a general category that encompasses the various domains of computer science that advocate advanced (post-OO) modularization techniques, including aspect-oriented programming (AOP), feature-oriented programming (FOP) [8], subject-oriented programming, open classes [14], module systems, runtime reflection, compile-time reflection, metaprogramming, generative programming, metaobject protocols, mixins, traits, and hyperdimensional separation of concerns. The AOP community was the first to coin the term “crosscutting”, recognize that it hinders modularity, and offer language support for modularizing crosscutting concerns.

The design of Wicca# has been influenced by many of these approaches. Specifically, Wicca# seeks to unify the powerful “single class” extensibility features of subtyping, feature-oriented programming, and open classes with the “multiple classes” extensibility features that characterize aspect-oriented programming. Wicca# also seeks to address several of the deficiencies of these approaches including lack of modular reasoning.

5.2 Side Classes: Powerful and Disciplined Class Refinement

A side class is a class that extends one or more base classes. Unlike traditional subclasses whose combination requires subclasses to know about each other and which require the client to instantiate the proper subclass, side classes can be combined anonymously and do not require the client to be updated in order to use the side class. This makes side classes ideal for adding or removing functionality orthogonal to the base class. However, side classes can specify composition order to resolve method override ordering issues, thus overcoming a limitation of open classes.

Side classes may extend multiple classes using an aspect-oriented (AO) quantification construct. However, side class extensions must be explicitly enabled by the base class and the AO quantification is bounded, thus preserving modular reasoning. Method calls to side classes may include implicit context parameters (similar to the implicit this parameter). Wicca# extends C#’s method lookup mechanism to take implicit parameters into account, thus improving method reusability and extensibility without sacrificing type safety.

While side classes borrows features from several ASOC techniques, it presents a consistent, easy-to-use, and unified method dispatching concept to the programmer, and addresses some of
the limitations of those techniques. A more detailed review of related work is provided in Section 6.3.

5.3 Statement Annotations: Fine-Grained Metadata for C#

Standard C# allows annotations, also called attributes, to be attached to assemblies, namespaces, types, type members, and method parameters and return values, but not individual statements. In contrast, Wicca# allows annotations to be attached to statements and statement blocks [20]. Statement annotations look like regular annotations except they are preceded by an in-line comment so they are ignored by a standard C# compiler.

Statement annotations are useful for a variety of purposes including providing optimization and parallelization hints to a compiler, specifying contracts for a theorem prover, and marking injection points for a generative programming tool. Statement annotations are ideal for associating auxiliary information with statements, as opposed to using a separate file or database, because they are co-located with the source code, making them easy to keep in sync with code changes, are a natural extension of regular annotations, and are accessible via standard metadata APIs. In [20], we showed how statement annotations overcome a limitation of AspectJ-like languages for modularizing so-called heterogeneous concerns.

5.4 The Wicca System

The Wicca System, or simply “Wicca”, implements the Wicca# language. It consists of static and dynamic components that coordinate to support the features of Wicca# (see Figure 1). Static components include the Wicca# compiler (wsc) and a postprocessor/weaver (Phx.Morph). The Wicca Runtime and debugger (wdbg) form the dynamic components. It was developed by me along with Columbia University students Boriana Ditcheva, Rajesh Ramakrishnan, and Adam Vartanian.
The Wicca System provides a host of innovative features including dynamic software updating, dynamic weaving, and noninvasive breakpoint weaving. However, at this time it is not clear which features will be needed by the Wicca# language. In the interests of space, I will not discuss them further in this proposal. More details can be found in [18], [19], and [17].

6. Related Work

My thesis spans several areas including requirements engineering, software metrics, and programming languages.

6.1 Requirements Engineering

My work on concern modeling and assignment can be classified broadly as requirements engineering activities. My concern model consists of the source domain $S$, target domain $T$, and concern assignment $R$, each of which are the focus of separate subfields of requirements engineering. Concern identification is the activity of determining what is or is not a concern of the program based on informal criteria (i.e., determining $S$). Concern assignment\(^8\) [9] is the act of assigning concerns to the program elements that implement them (i.e., $R$). A testament to the

---

\(^8\) I use this term “concern assignment” instead of the official term, “concept assignment” [9], for consistency.
broad applicability of my model is that normally unrelated fields such as concern identification, concern assignment, aspect mining and requirements tracing can be characterized by finding S, R, or T.

**Concern Modeling**

A concern model is a formal description of the relationship between concerns and source code. Robillard and Murphy [40] formally model this relationship using graph theory. A convenient aspect of their model is that it directly represents the organization of a program, whereas my model requires a somewhat clumsy aggregation relation. A significant advantage of my model is that it is simple and easily measurable. Furthermore, my model provides a more fine-grained and complete concern assignment, which leads to a more accurate quantification of crosscutting.

Coppit et al [37] use a specialized notation to describe the relationship between concerns and source code and provide a set of source code refactoring operations that operate directly on the description. They require the same level of detail as my case study and for a similar purpose: concerns can be selectively excluded from the program. However, their research is focused on improving program comprehension and software editing, whereas my research is focused on quantifying crosscutting and refactoring decision support.

**Concern Identification and Assignment**

Most concern identification case studies are incomplete or informal. By incomplete, I mean that only some concerns or requirements are identified, making it hard to quantify how crosscutting impacts modularity overall. Incomplete identification characterizes many papers on aspect-oriented programming and is somewhat self-serving, that is, crosscutting concerns are only singled out if they can be separated easily by the proposed technique. While these positive results are suggestive of the benefit of aspect-orientation (see for example [15]), I believe they only scratch the surface of the potential for understanding and modularizing crosscutting. My case study will exhaustively identify and characterize all the concerns in the program, both crosscutting and localized, which will allow me to clearly explain the limitations of current approaches, and motivate my own solution.

Other researchers have used manual methods to exhaustively identify concerns [11] [36] [7, 21, 42, 50] [29] [37]. In these studies, the study participants are asked to reverse engineer the source code of an unfamiliar program with at most an informal understanding of the concerns to locate. By informal, I mean that the concerns are not based on any formal criteria, i.e., they are ad hoc, which confounds identification accuracy and consistency. I plan to use more formal criteria from a well-defined requirements specification to obtain a more complete picture of crosscutting concerns than previous studies.

There are several automated or semi-automated techniques for concern assignment [50] [21] [40] as well as tools that make it easier to manually assign concerns [37] [29] [36]. While automated techniques are out of the scope of this thesis, they are essential for making concern assignment feasible for large programs. Interesting follow-on work would be to create a concern assignment automatically based on treating each class as a separate concern, and therefore observe how my crosscutting metrics relate to traditional OO metrics like class

9 “Large” being the point at which the program cannot be fully understood by a single person. Revelle et al. showed that manual concern assignment by multiple people is bound to be inconsistent. [36]
coupling and cohesion. A review of the state of the art for automated techniques convinced me that a labor-intensive, manual technique was required to ensure consistency, completeness, and accuracy.

6.2 Software Metrics

Most software metrics measure physical properties of a program. This is useful for determining that a class has high coupling and is therefore complex and error prone, for example [33]. A significant difference with my work is that it measures the relationship between logical entities (concerns) and physical entities. This enables the measurements to be interpreted in the context of the problem domain. For example, my crosscutting metrics can determine that a specific feature will be hard to change or that concerns are not well separated. This kind of insight is not obtainable by measuring physical properties alone.

Several researchers have attempted to measure crosscutting. However, their metrics are too coarse-grained to provide a detailed picture of crosscutting or to distinguish the subtle differences between refactoring alternatives. Degree of scattering (DOS) is an improvement over the class-level scattering metric proposed by Zhang and Jacobsen [51], the spread metric proposed by Lai and Murphy [29] and later extended by Revelle et al. [36], and the concern diffusion metrics (also called separation of concerns metrics) specified by Garcia and coworkers [22]. These metrics only measure the participating components (e.g., CONC ≠ 0), ignoring how the implementation is distributed across components. For example, a concern whose concentration distribution is \{0.40, 0.30, 0.30\} has the same spread, diffusion, etc. as one with a distribution of \{0.90, 0.05, 0.05\}. In contrast, DOS clearly indicates that the later concentration distribution is less scattered than the former (because more of the implementation is centralized in one component). DOS is more sensitive to variations in the distribution of the implementation of a concern, and thus provides a more accurate summary of the results of a particular refactoring.

The work of Alessandro Garcia and colleagues is very similar to mine. In [22], they provide definitions for “tangled” and “scattered” that are essentially the same as mine, although they are not based on a formal model.\(^{10}\) They also adapt some OO metrics such as coupling and cohesion to the AO domain and perform several case studies [10], both of which are likely to be beneficial for me as well. As mentioned in the previous paragraph, my crosscutting metrics are more sensitive and informative than their concern diffusion/separation of concerns metrics.

My crosscutting metrics are adapted from the closeness metrics defined by Wong, Gokhale, and Horgan [50]; however, my metrics provide more useful information. For example, DOS measures the degree to which a concern’s implementation is spread across all components, providing a more holistic picture of crosscutting than the CONC metric it is based on. While CONC is useful for understanding each component of a program, DOS helps developers to find concerns that may be costly to modify.

The DEDI closeness metric helps to determine if a component is dedicated to any one concern in particular, which is helpful when reverse engineering a program. In contrast, DOF summarizes whether the component is dedicated to one or many concerns and to what degree. In addition, DOF is more helpful than metrics that simply measure how many concerns are related.

\(^{10}\) The incorrectly state that scattering implies tangling. However, this depends on the granularity level of concern and component. For example, two classes may be entirely focused on implementing a single high-level concern; the concern is scattered but not tangled.
to a component. For example, 90% of the component’s code may be dedicated to one primary concern and the remaining 10% are dedicated to secondary concerns such as error handling, logging, etc. DOF captures this subtlety whereas a metric that just looks at the total number of concerns implemented by a component would not.

6.3 Programming Languages

Wicca# borrows features from object-oriented programming (subtyping), feature-oriented programming (method refinement) [8] [5], aspect-oriented programming (quantification, implicit parameters), and open classes [14]. Wicca#’s lasting contribution will hopefully lie in its simple and elegant approach (side classes) to unify these normally disparate features and modularize crosscutting concerns. The main advantage of side classes is that it unifies subtype, open class, FOP-style, and AOP-style method overriding and dispatching.

Most AOP languages extend a traditional language by introducing new constructs to support AOP concepts like “aspects”, “pointcuts”, and “advice”. This requires the programmer to learn a new set of constructs and keywords, and sometimes even a whole new programming methodology, to take advantage of aspect-orientation. In addition, AO class extensions are usually not symmetric with respect to classes [26]. This unnecessarily complicates design because it forces programmers to switch back-and-forth between aspect- and object-oriented viewpoints. In contrast, side classes unifies and simplifies the different method overriding and dispatching mechanisms. This coincides with Steimann’s view that AOP researchers should focus on blending the key concepts of AOP with those of other programming paradigms [41].

While AOP have risen in popularity over the last decade, the drawbacks of the more prominent AOP languages are numerous and well documented [41], even making it difficult for experts to achieve modular design [22]. One of the primary concerns is that AOP languages allow the abuse of the obliviousness principle [41] [24] [32] and therefore sacrifice modular reasoning [13] [4]. Modular reasoning refers to the ability to reason about an entire program by reasoning about each module in isolation. Citing obliviousness as justification, AOP languages, as well as languages that support open classes, allow unfettered access to private methods and fields and allow methods explicitly specified as non-overridable to be overridden. This makes modular reasoning difficult, defeats information hiding (a class’ member variables +essentially become global variables [41]), and has led to the impression that AOP is synonymous with “hacking”. In contrast, Wicca# takes a straightforward approach by extending normal access control mechanisms to enforce restricted11, explicit, and well-defined extension points.

Popular AOP languages provide a powerful mechanism for extending multiple classes and methods called quantification [24]. Unfortunately, when the program evolves, existing extensions may be broken or new ones created inadvertently. This is referred to as part of the evolution paradox [48] [42], whereby the use AOP techniques actually makes it harder to evolve a program, which is contrary to the stated intentions of AOP.

Part of the problem is that AOP languages allow fragile couplings (e.g., based on naming conventions) to be made between the extensions and the classes and methods being extended [48] [28] [41]. I argued in [20] that statement annotations provide an elegant partial solution. Wicca# builds upon the work of [a]C# [12] to support statement annotations for robust fine-grained aspect weaving and concern assignment.

---

11 “restriction is the key to success” [41]
Another problem occurs when an AOP language supports *unbounded quantification*, for example, by allowing the programmer to specify the set of classes and methods to extend using a regular expression. This makes it easy to extend a class or method inadvertently or silently break an existing extension due to an otherwise insignificant change (e.g., changing the name of a method). In my thesis I will propose a mechanism for Wicca# to support *bounded quantification* [5].

7. Research Plan

My research plan spans 13 months and is centered on three activities: developing a model, metrics, and methodology for quantifying crosscutting concerns (90% complete), measuring crosscutting concerns in an evolving case study (the initial phase is 50% complete), and the design, implementation, and evaluation of the Wicca# language (50% complete). A good deal of technology infrastructure has already been developed for Wicca#. Remaining work includes the design and implementation of side classes.

7.1 Statement of Work

I plan to complete the initial phase of the case study by the end of Fall 2006. This is an important milestone because it will allow me to calibrate my metrics and methodology. I plan to complete the remaining phases in sequential order. My hope is that each phase of the case study will reveal more insight into crosscutting and effective ways to modularize it. Therefore, I will be evolving the design and implementation of Wicca# alongside the case study. Obviously, the design of Wicca# must be complete before the Wicca# Refactoring Phase begins.

The AO Refactoring Phase of the case study requires refactoring Goblin using a C# compatible AspectJ-like language. I plan to enlist the help of one or more graduate students in the Spring and Fall of 2007 to work on Phx.Morph to add support for the necessary AspectJ features, and to implement necessary features in Wicca#.

7.2 Timeline

<table>
<thead>
<tr>
<th>Timeline</th>
<th>Work Item</th>
<th>Progress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan 2006</td>
<td>Wicca (design); build team</td>
<td>DONE</td>
</tr>
<tr>
<td>Feb 2006</td>
<td>Wicca (annotation matching)</td>
<td>DONE</td>
</tr>
<tr>
<td>Mar 2006</td>
<td>Wicca; AOSD</td>
<td>DONE</td>
</tr>
<tr>
<td>Apr 2006</td>
<td>Wicca (breakpoint weaving); Statement Annotations paper</td>
<td>DONE</td>
</tr>
<tr>
<td>May 2006</td>
<td>Wicca (statement annotations; alpha release)</td>
<td>DONE</td>
</tr>
<tr>
<td>June 2006</td>
<td>Intern; Open Classes</td>
<td>DONE</td>
</tr>
<tr>
<td>July 2006</td>
<td>Intern; Open Classes; RAM-SE</td>
<td>DONE</td>
</tr>
<tr>
<td>Aug 2006</td>
<td>Intern; Open Classes</td>
<td>DONE</td>
</tr>
<tr>
<td>Sept 2006</td>
<td>Thesis proposal writing</td>
<td>50%</td>
</tr>
<tr>
<td>Oct 2006</td>
<td>Thesis proposal writing; Case Study (Phase One)</td>
<td>50%; 50%</td>
</tr>
<tr>
<td>Nov 2006</td>
<td>Thesis proposal writing; Case Study (Phase One)</td>
<td>95%; 55%</td>
</tr>
<tr>
<td>Dec 2006</td>
<td>Thesis proposal distribution and defense; Case Study (Initial)</td>
<td></td>
</tr>
<tr>
<td>Jan 2007</td>
<td>Case Study (Normalization); Wicca# design</td>
<td></td>
</tr>
</tbody>
</table>

28
8. Summary

Over the past decade, crosscutting concerns have become recognized as a major obstacle to modularity, thereby increasing the cost of software development and maintenance. I develop a new model, set of metrics, and methodology to formalize and rigorously quantify crosscutting, to measure the benefits of a refactoring and to allow existing modularization alternatives to be compared. I perform a rigorous case study to better understand crosscutting and quantitatively compare my Wicca# language with leading languages that support advanced separation of concerns. Finally, I apply the insights gained from the case study to design extensions to the C# language, embodied by my Wicca# language, to directly support effectively modularizing crosscutting and to unify normally disparate constructs, such as method dispatching and fine-grained advising, to make them easier to understand and apply.

Bibliography


Appendix A. Definitions

module – a program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; a responsibility assignment; aka modular artifact

modularization – the design decisions which must be made before the work on independent modules can begin; dividing the program into a number of modules with well-defined interfaces to achieve parallelism in design and implementation, abstraction from complexity, and ease of evolution

modular design – a design or part of a design is modular if its elements do not depend on each other (although they may depend on external design rules that serve to decouple them); two design dimensions X and Y are "modularized" if decisions can be made or changed in the X dimension without having a significant influence on the Y dimension and vice versa

modularity – the degree to which a program is modularized: a well-modularized program has a high-level of modularity

dependency – a thing X (e.g., module, concern, design decision), is dependent on another thing Y (X -> Y) if a change to Y may require a change to X; also "coupling"

decomposition – the methods and mechanisms used to modularize a program; breaking down a larger problem into a set of smaller problems which may be tackled individually

hierarchical decomposition – decomposing (part of) the program into a hierarchical structure of modules where modules are related to each other via a "uses" or "depends on" relation and the relation is a partial ordering

composition – the integration of multiple modular artifacts into a coherent whole

scattering – the occurrence of elements that belong to one concern in modules encapsulating other concerns

tangling – the occurrence of multiple concerns mixed together in one module

concern – a design decision, requirement, issue, and/or feature of the program; anything of interest to the programmer [1]; any coherent issue in the problem domain [11]; “any consideration … about the implementation of a program” [38]; “something [about the software system] one cares about” [23]; “any matter of interest in a software system” [44]; “a human-oriented expression of computational intent” [9]; “a conceptual area of interest or focus for a stakeholder of a software project” [39]

feature – a concern that adds functionality to the software [1]

concern code – code related to a concern [1]
crosscutting concern – a concern which cannot be effectively modularized within the selected decomposition. Consequently, the elements of crosscutting concerns are scattered among and tangled with elements of other concerns. A common issue that applies across a range of otherwise unrelated classes; “a concern that spans several objects or functions” [1]

aspect – unit for modularizing an otherwise crosscutting concern

aspectual decomposition – decomposing (part of) the program into aspects

weaving – same as composition except some modular artifacts may be aspects

join point – a point of interest in the execution of a program

join point model – defines the kinds of join points available and how they are accessed and used

pointcut – identifies a set of join points; a quantification of join points; a predicate that matches join points; a relationship f: JP → [ T, F ] where the domain is all possible join points; also, "pointcut expression"

advice – an element of an aspect; augments or constrains other concerns at join points matched by a pointcut

aspect-oriented programming (AOP) – programming methodology that directly supports aspects and aspectual decomposition

aspect-oriented programming language – programming language that directly supports aspects and aspectual decomposition

heterogeneous concern – a crosscutting concern whose join points are difficult to express using a particular AOP language or result in fragile pointcuts

homogeneous concern – a crosscutting concern whose join points are easy to express using a particular AOP language and result in robust pointcuts

open classes – the ability to modify a class without editing the class directly
Appendix B. Detailed Measurement Example

This appendix is meant as supplemental material designed to help the reader understand the crosscutting metrics proposed in Section 3. To that end, I explain how I arrived at the measurements for the example in Section 3.10. I have reproduced the tables from that section.

<table>
<thead>
<tr>
<th>STATEMENTS</th>
<th>STATEMENT ID</th>
<th>IMPLEMENTED REQUIREMENT IDS (S&lt;sub&gt;i&lt;/sub&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Void draw_shape(Shape* s) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (s-&gt;IsCircle) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Circle* c = (Circle*) s;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...10 more statements that draw a circle...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>else if (s-&gt;IsRect) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rect* r = (Rect*) s;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...8 more statements that draw a rectangle...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Original C program.

The table indicates that all 22 statements in `draw_shape` are associated with the drawing requirement (r1), 12 with the circle requirement (r5), and 10 with the rectangle requirement. Assume that `create_shape`, `move_shape`, and `rotate_shape` have the same makeup, except replace r1 with r2, r3, and r4 respectively. The total number of assigned statements for each concern is A<sub>r1</sub> = 22, A<sub>r2</sub> = 22, A<sub>r3</sub> = 22, A<sub>r4</sub> = 22, A<sub>r5</sub> = 48, and A<sub>r6</sub> = 40. The concentration for each concern in `draw_shape` is as follows:

CONC<sub>r1-draw_shape</sub> = 22 / 22 = 1.00
CONC<sub>r2,r3,r4-draw_shape</sub> = 0 / 22 = 0
CONC<sub>r5-draw_shape</sub> = 12 / 48 = 0.25
CONC<sub>r6-draw_shape</sub> = 10 / 40 = 0.25

The concentration for the other functions will be similar after substituting for r1. Notice the concentration is 0 if no statements related to the concern are part of the function. The degree of scattering for each concern is (with |C| = 10):

\[
\text{DOS}_{r_1,r_2,r_3,r_4} = 1 - \frac{10 \left( (1.0 - 0.1)^2 + 9(0.0 - 0.1)^2 \right)}{9} = 0
\]

\[
\text{DOS}_{r_5,r_6} = 1 - \frac{10 \left( 4(0.25 - 0.1)^2 + 6(0 - 0.1)^2 \right)}{9} = 0.83
\]
Draw_shape’s dedication to each of the concerns is as follows (with \(|V_r| = 22 + 12 + 10 = 44|):

\[
\begin{align*}
\text{DEDI}_{\text{draw\_shape-r1}} &= 22/44 = 0.50 \\
\text{DEDI}_{\text{draw\_shape-r5}} &= 12/44 = 0.27 \\
\text{DEDI}_{\text{draw\_shape-r6}} &= 10/44 = 0.23
\end{align*}
\]

Draw_shape’s degree of focus is (with \(1/|S| = 1/6 = 0.17|):

\[
\text{DOF}_{\text{draw\_shape}} = \frac{6 \left( (0.50 - 0.17)^2 + (0.25 - 0.17)^2 + (0.21 - 0.17)^2 + 3(0 - 0.17^2) \right)}{5} = 0.24
\]

The DOF for create_shape, move_shape, and rotate_shape are the same after substituting for r1. Therefore, the average degree of focus (DOF) is also 0.24.

After the Submodule Refactoring

Table 2 shows the draw_shape function after the shape-specific code is moved to separate files.

<table>
<thead>
<tr>
<th>STATEMENTS</th>
<th>STATEMENT ID</th>
<th>IMPLEMENTED REQUIREMENT IDS ((S_t))</th>
</tr>
</thead>
</table>
| void draw_shape(Shape* s) {
  if (s->IsCircle) {
    Circle* c = (Circle*) s;
    circle_draw(c);
  }
  else if (s->IsRect) {
    Rect* r = (Rect*) s;
    rect_draw(r);
  }
} | s115 | r1, r5 |
| | s116 | r1, r5 |
| | s117 | r1, r5 |
| | s118 | r1, r6 |
| | s119 | r1, r6 |
| | s120 | r1, r6 |

Table 2. After submodule refactoring.

The circle.c file contains the circle_draw, circle_create, etc. functions, and similarly for the rect.c file. Thus the number of components \(|C|\) has increased by 8. The number of assigned statements for concerns decreases for r1-r4 but increases by one for r5 and r6 due to the new function call statements (s117 & s120). The total number of assigned statements for each concern is \(A_{r1} = 24, A_{r2} = 24, A_{r3} = 24, A_{r4} = 24, A_{r5} = 52, A_{r6} = 44\). The concentration for each concern in draw_shape is as follows:

\[
\begin{align*}
\text{CONC}_{\text{r1-draw\_shape}} &= 6/24 = 0.25 \\
\text{CONC}_{\text{r5-draw\_shape}} &= 3/52 = 0.06 \\
\text{CONC}_{\text{r6-draw\_shape}} &= 3/44 = 0.07
\end{align*}
\]
The concentration for the other functions will be similar after substituting for r1. The concentration in the newly added functions is as follows (similarly for the other new functions):

\[
\begin{align*}
\text{CONC}_r1\text{-circle\_draw} & = \frac{10}{24} = 0.42 \\
\text{CONC}_r5\text{-circle\_draw} & = \frac{10}{52} = 0.19 \\
\text{CONC}_r1\text{-rect\_draw} & = \frac{8}{24} = 0.33 \\
\text{CONC}_r6\text{-rect\_draw} & = \frac{8}{44} = 0.18
\end{align*}
\]

Notice that the implementation of general requirement r1 is now split into three functions: \textit{draw\_shape}, \textit{circle\_draw}, and \textit{rect\_draw}. The degree of scattering for each concern is (with $|C|$ now being 18):

\[
\begin{align*}
\text{DOS}_{r1} &= 1 - \frac{18 \left( (0.25 - 0.06)^2 + (0.42 - 0.06)^2 + (0.33 - 0.06)^2 + 15(0 - 0.06)^2 \right)}{17} \\
\text{DOS}_{r5} &= 1 - \frac{18 \left( 4(0.06 - 0.06)^2 + (0.19 - 0.06)^2 + 10(0 - 0.06)^2 \right)}{17} \\
\text{DOS}_{r6} &= 1 - \frac{18 \left( 4(0.07 - 0.06)^2 + (0.18 - 0.06)^2 + 10(0 - 0.06)^2 \right)}{17}
\end{align*}
\]

The degree of focus for \textit{draw\_shape}, \textit{create\_shape}, \textit{move\_shape}, and \textit{rotate\_shape} stay about the same (0.24). The dedication for the new \textit{circle\_draw} function is as follows (similarly for the other seven new functions) (with $|V_c| = 10 + 10 = 20$):

\[
\begin{align*}
\text{DEDI}_{\text{circle\_draw}-r1} &= \frac{10}{20} = 0.50 \\
\text{DEDI}_{\text{circle\_draw}-r5} &= \frac{10}{20} = 0.50
\end{align*}
\]

\textit{Circle\_draw}’s degree of focus is

\[
\text{DOF}_{\text{circle\_draw}} = \frac{6 \left( 2(0.50 - 0.17)^2 + 4(0 - 0.17)^2 \right)}{5} = 0.40
\]

The average degree of focus for the four general functions and eight new functions is now 0.35. This indicates that, overall, the functions are now more focused than before.

\textit{After the Virtualization Refactoring}

Table 3 shows the \textit{draw\_shape} function after the virtual dispatch logic is removed due to the use of virtual C++ methods.
void draw_shape(Shape* s) {
    s->draw();
}

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s15</td>
<td>r1, r5, r6</td>
</tr>
</tbody>
</table>

Table 3. After virtualization refactoring.

The total number of assigned statements for each concern is $A_{r1,r2,r3,r4} = 19$, $A_{r5} = 40$, and $A_{r6} = 32$. The concentration for each concern in `draw_shape` is as follows:

CONC $r1$-draw_shape = $1 / 19 = 0.06$
CONC $r5$, $r6$-draw_shape = $0 / 40 = 0$

The concentration for the other functions will be similar after substituting for $r1$. The concentration in the newly added functions is as follows (similarly for the other new functions):

CONC $r1$-circle_draw = $10 / 19 = 0.53$
CONC $r5$-circle_draw = $10 / 40 = 0.25$
CONC $r1$-rect_draw = $8 / 19 = 0.42$
CONC $r6$-rect_draw = $8 / 32 = 0.25$

The degree of scattering for each concern is (with $|C|$ now being 18):

$$DOS_{r1} = 1 - \frac{18 [(0.06 - 0.06)^2 + (0.53 - 0.06)^2 + (0.42 - 0.06)^2 + 15(0 - 0.06)^2]}{17} = 0.54$$

$$DOS_{r5,r6} = 1 - \frac{18 [4(0.25 - 0.06)^2 + 14(0 - 0.06)^2]}{17} = 0.87$$

DOF remains the same (0.40) for the new functions. However, the degree of focus for `draw_shape`, `create_shape`, `move_shape`, and `rotate_shape` are now 1 (100% focused on one concern). The average degree of focus for the twelve functions is now 0.60, a 2.5 fold improvement over the original program.