

An Agenda for Concern-Oriented Software Engineering

Nicolas Lopez and André van der Hoek

Department of Informatics
University of California, Irvine
Irvine CA, 92627, USA

{nlopezgi, andre}@uci.edu

ABSTRACT

The principle of separation of concerns has certainly stood the test of time in guiding the field of software engineering, leading to an amazing variety of approaches available to programmers to actually separate and manage concerns in their software. In this paper, we provide a novel perspective on these approaches, a perspective that is guided by the observation that the underlying goal of any approach should not be to always separate concerns, but instead to minimize the impact of concern scattering and tangling. Reframed as such, we survey and relate existing work, highlight fundamental limitations of the four canonical approaches to minimizing the impact of concern scattering and tangling, and provide an agenda for future work – at both the code level and beyond.

Categories and Subject Descriptors

D.2.3 [Software]: Software Engineering – Coding Tools and Techniques. D.1.m [Software]: Programming Techniques – modularization. D.2.7 [Software]: Software Engineering – Distribution, Maintenance, and Enhancement – *restructuring, and reverse engineering*

General Terms

Design, Languages.

Keywords

Software concerns, separation of concerns, modularization, software maintenance and evolution, scattering and tangling

1. INTRODUCTION

Dijkstra in 1974 [9] proposed his principle of separation of concerns, which has had a profound influence on the field, so much so that it is now a standard practice in programming to leverage the facilities of the programming language to modularize concerns. Parnas' work of course was equally important in achieving this impact, as it was his early modularization ideas [20] that set the tone for how one should go about separating concerns and that also implicitly identified several requirements for programming languages to support this practice.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.

Copyright 2010 ACM 978-1-4503-0427-6/10/11...\$10.00.

While modularization has clearly been the predominant approach to supporting separation of concerns (whether via traditional programming language constructs, aspect orientation [15], or subject-oriented programming [10], to name a few), other approaches have emerged as well. Perhaps the most different approach is that of concern modeling. Instead of embedding support for separating concerns in the programming language, a separate model is maintained that describes and relates concerns, and indexes them to the programming language. Such a model may be explicitly kept (e.g., CME [11], ArchEvol [18]), or implicitly constructed with the help of queries or heuristics (e.g., FEAT [21], Mylyn [13]).

One way to examine this body of work as it has emerged to date is to treat each approach as a separate inroad into supporting a concern-based view of software development. While to some degree we do that in this paper, we also seek to establish a deeper understanding, one that is based on the observation that separating concerns is just a means to an end, rather than a goal in and of itself. Dijkstra already hinted at this, noticing that a perfect separation of concerns is impossible to achieve and highlighting that the reason to separate them was to be able to understand each in isolation as one deals with it [9].

We see two problems with this goal of understanding a concern in isolation, however:

- One does not always want to deal with just one concern, as much of the difficulty of concerns lies in dealing with them when they represent related views. For instance, we may want to know how, if we modify some security code, privacy is affected. Here, it is crucial to know where both concerns are addressed in the code.
- Given that a perfect separation is impossible to achieve, we must begin to ask which concerns are worth separating and why. The driver here, of course, is evolution: it should be possible to make future changes with the least amount of “hassle” given the concerns that those future changes address.

The issue, thus, is not necessarily separating concerns. Rather, the key objective is mitigating the long-term impact of scattering and tangling of concerns to reduce the complexity of understanding, maintaining, evolving, and reusing code. It may be perfectly fine to leave some concerns scattered, and in some cases it is even desirable to not spend the extra effort of refactoring, as that code may never be visited again. Some other code, however, may need to definitely be untangled and put into its own modules, since it is likely the subject of significant future work. By the same token, modularization is just one way in which the impact can be mitigated. In some cases, the decision may be made to use a concern model instead for some of the code, as it is easier to construct and use, and does not involve expansive refactoring.

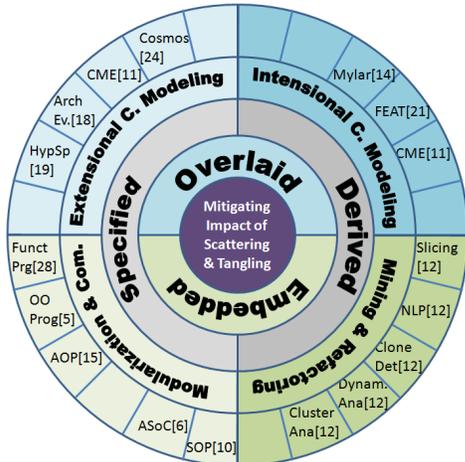


Figure 1. Concerns Framework.

With all these choices, concern-oriented software engineering in effect becomes an optimization problem: which techniques should be used when, so the long-term impact of concern scattering and tangling is minimized? It is this optimization problem that drives the research agenda that we define in the rest of the paper.

2. CONCERNS FRAMEWORK

The concept of a concern has been defined numerous times, with a relatively broad variety of definitions emerging. We adopt a definition authored at a recent workshop that highlights the dual nature of a concern [22]: (1) a concern is a conceptual area of interest or focus for a stakeholder of a software project, and (2) a concern refers to the concrete manifestation of conceptual concerns in software artifacts. Both perspectives are important. The first stipulates that concerns can be broad in nature, and stem from a variety of individuals with an interest in the system. The second ties this interest to concrete lines of code in the system.

With this definition in mind, we now examine existing approaches to mitigating the long-term impact of concern scattering and tangling using the framework presented in Figure 1. The framework identifies four canonical approaches by separating them along two orthogonal dimensions: embedded versus overlaid and specified versus derived.

The center circle represents the overarching goal we identified in our discussion thus far, towards which all advances must work. In order to do so, it clearly is necessary that concerns are represented in one way or another. The first circle surrounding the core highlights that this can be done in one of two ways: (1) by embedding the concern representation in the target language, or (2) by overlaying a concern model that relates concerns and artifacts.

The second dimension, as represented by the next circle, identifies the two alternative ways in which a concern representation can be constructed: (1) concerns are explicitly specified by the developer, or (2) concerns are derived from the artifacts, ideally in an automated or semi-automated way.

Combining these two dimensions, how concerns are represented and how this representation is constructed, identifies four distinct quadrants that group approaches with similar characteristics.

Modularization and composition (bottom left). Approaches in this category require a developer to manually decide upon a par-

ticular modularization, with each module ideally representing one concern. The idea is to physically separate concerns by leveraging programming language constructs that hide information pertaining to a concern, with tools supporting the composition of individual modules into a system. This encompasses the classic approach of classes and interfaces (composed with a compiler and linker), but also more advanced approaches such as aspect orientation [15] and subject oriented programming [10]. The assumption underlying these approaches is that, to mitigate the impact of concern scattering and tangling, the best solution is to attempt to avoid the problem altogether, at least for some concerns, by physically separating them.

Extensional concern modeling (top left). Approaches in this category require a developer to, separately from the code, specify one or more concern models that capture individual concerns and their relations. This concern model is associated to the code via explicit links between the concerns and the development artifacts, sometimes at a coarse-grained level (e.g., entire artifacts), sometimes at a fine-grained level (e.g., individual lines of code or even just sets of characters). With respect to mitigating the impact of concern scattering and tangling, these approaches recognize that such scattering and tangling is a fact of life, regardless of which modularization is applied (the tyranny of dominant decomposition [25]). The assumed best way to mitigate their impact is to be inclusive of all concerns and to always enable a precise trace to where a concern is implemented in the code. The use of visualization tools enables strong insights into actual scattering and tangling in a code base.

Intensional concern modeling (top right). Approaches in this category also have a concern model but evaluate concerns in a different way. In an extensional concern model, a concern is linked to its code once, after which it remains static unless it is explicitly updated—the links are actively managed. In an intensional concern model, where a concern resides in the code is dynamically determined; the links are not managed but derived. Intensional modeling approaches identify the code that relates to a concern through queries or heuristics that construct a temporary concern model overlaying the code. Analyses of code artifacts and changes, as well as developers’ historical interactions with the development environment, enable (semi-)automatic derivation of the relevant links. The hypothesis underlying these approaches is that the best way to mitigate the impact of concern scattering and tangling is to reduce the workload on the developer by not requiring them to maintain the concerns themselves. They instead are provided with ways of finding them when needed.

Mining and refactoring (bottom right). Approaches here parallel intensional concern modeling in using (semi-)automated analyses, but differ in their objective in seeking to derive an improved modularization as compared to how a set of concerns presently is embedded in the system. Some approaches mine for new concerns, others seek to refactor the code to yet better separate concerns from one another. The hypothesis underlying these kinds of approaches is that concern scattering and tangling can be mitigated by continuously pushing for the “ideal” modularization with each change that developers make to the system.

3. STRENGTHS AND WEAKNESSES

Given the framework, one may ask which approach is better to use in a given situation. In general, this is a difficult question to answer, because we do not yet have the right depth of understand-

ing in terms of the absolute and relative merits of each of the canonical approaches. As a first step towards building this understanding, Table 1 summarizes key strengths and weaknesses of each.

Some tradeoffs are visible immediately. Modularization and composition approaches, for instance, provide for independent design and implementation of concerns, effectively insulating developers from one another so they can work in parallel. Moreover, the code associated with a concern can be easily reused – if it was properly modularized in the first place. This is in contrast to extensional concern modeling, which does not have those benefits but allows for the management of a broader range of concerns, and can more effectively deal with making visible those concerns that are tangled. Similar tradeoffs can be derived from the framework with respect to other pairs of approaches.

Some of these weaknesses can be expected to be overcome in the future with improved language and tool support. Others, however, will not, as each quadrant has its own key fundamental limitation inherent to its underlying approach. For modularization and composition, this limitation has been widely discussed in the literature and is known as the tyranny of dominant decomposition [25]. In shorthand, this limitation states that it is impossible to modularize every concern, and scattering and tangling inevitably arise.

Similar fundamental limitations exist for the other quadrants. Any extensional approach faces the human inability to deal with scale, as it cannot be expected that developers can precisely maintain a complex mapping from concerns to code with each change they make. In the case of intensional modeling, the limitation is that only humans can interpret what a given concern means; automated queries and heuristics can only approximate such meaning. Finally, for mining and refactoring, the barrier is that those refactorings that ultimately would yield the greatest benefit in a given situation are one of a kind, and thereby unspecifiable as a generic program.

4. RESEARCH AGENDA

Given the strengths and weaknesses we presented in Table 1, and given the fundamental limitations identified in the previous section, it should be clear that none of the approaches is best under all circumstances. Worse yet, each will exhibit its weaknesses whenever it is applied; programmers will have to be aware of and deal with those weaknesses as part and parcel of their work. Further, with the ever-increasing scale and complexity of the systems we build, the weaknesses and limitations will be more obvious.

This is not to say that no progress has been made, or can be made again. Rather, we believe our framework helps identify a number of different ways in which the field can move forward and work towards the goal of reducing the impact of concern scattering and tangling.

Extending approaches within each quadrant. The most obvious step forward is to continue work within each quadrant, and indeed this work is happening and continues to happen. New modularization languages are being invented regularly, new extensional modeling tools are emerging, intensional modeling is still only in its infancy, and more advanced mining algorithms and refactoring heuristics continue to be developed. Particularly approaches in the top half of the framework have a lot of room for improvement, simply because they are still relatively new. We mention ArchEvol [18], which takes an incremental heuristics-based ap-

Table 1. Strengths (+) and Weaknesses (-).

	Modularization & Composition
+	Independent design and maintenance; effective reuse
-	A modularization can become stale when new and changing concerns demand different structures; scale of concerns can lead to very complex modularizations; advanced languages have steep learning curves
	Extensional Concern Modeling
+	Arbitrary types of concerns can be modeled; granularity is very flexible; scattered and tangled concerns easily identified
-	Requires high buy-in to consciously track all concerns, identify new concerns as they emerge, and update concerns as programming continues; tool support to update concerns is immature, leaving traceability largely as a manual task
	Intensional Concern Modeling
+	Very low barrier to use as it can be applied post-hoc; many different analyses and heuristics can be applied; other artifacts than code can be used in identifying concerns
-	Typically supports only one concern at a time; imprecise results, requiring further human filtering
	Mining and Refactoring
+	Can gradually improve scattering and tangling; very low barrier to use as it can be applied post-hoc; many different analyses and heuristics can be applied
-	Most mining research only identifies candidate concerns, it does not support restructuring the code; refactoring tends to make local improvements, not global

proach to updating concern links, and Mylyn [13], which attempts to actively provide a focused set of artifacts for a given task by mining past actions and artifacts, as particularly promising in this regard. The weaknesses listed in Table 1 should help guide the field’s research efforts; for instance finding more global refactorings, reducing the entry barrier to new modularization languages, or improving the precision and recall of intensional concern modeling approaches.

Crossing quadrant boundaries. A more complex step forward, and one that to date has been attempted only in a limited fashion, is to build connections across the quadrants, so that selected combinations of approaches can be applied to a software system. First, this might simply involve being able to switch which approach to use for a certain concern. Modularization may be favored early on for some concern, but as the relevance of that concern decreases, it may be beneficial to move it to a concern model instead, or even to not track it at all since its potential long-term impact might be less than the effort to convert and track it. Another example might be a concern that is first found using an intensional approach, then explicitly tracked in a concern model, and eventually modularized in the source code. These kinds of transitions should be supported flexibly. Hybrid approaches can even be imagined, where a given concern is modularized at first, but as it slowly scatters, a concern model is used to track where it scatters.

Second, different approaches can benefit from one another directly as well. Imagine an intensional query or heuristic that is particularly strong at identifying certain kinds of concerns. If the preferred approach is to use a concern model, however, newly found concerns using the query or heuristic must be placed in the concern model and evolved using the mechanisms provided by

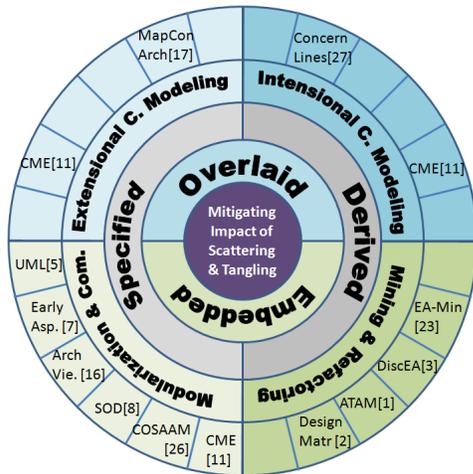


Figure 2. Concerns Framework Applied Beyond Code.

generic tools surrounding this concern model. It would be much better to build such tools in an extensible manner, allowing the original query or heuristic to be plugged in and matched to certain concerns, to improve the precision and recall of tracking the evolution of the concern. As another example, imagine applying refactoring techniques to a concern model rather than code, in an attempt to find a better “modularization” of the conceptual concerns that govern the code.

Good science. Underpinning any and all advances should be good science, that is, the use of informed research methods and consequential evaluations. We should understand how concerns really evolve, how they become scattered and tangled, and when scattering and tangling becomes problematic. Is there a predictable life cycle of some sort for concerns in which they come about, grow, and then become stable; or is the situation more erratic? Are there certain classes of concern evolution patterns that we can find? A small number of researchers are beginning to look into these kinds of questions, but clearly much more empirically grounded work is needed.

From such studies, we would expect grand challenges and benchmarks to be developed that can be used by any new approach or combination of approaches to evaluate its success in reducing the impact of concern scattering and tangling, both in absolute terms with respect to the benchmarks and challenges and in relative terms as compared to other proposed approaches. This means that we must find ways of quantifying this impact, too, which will be difficult as the ultimate impact of scattering and tangling can only be directly measured in the complexity of debugging, maintaining and evolving a system over a long time.

Beyond code. Our discussion thus far has exclusively focused on code, but other kinds of artifacts are equally subject to the notions of concerns, scattering, and tangling, and the desire to minimize the impact of concern scattering and tangling. Figure 2 presents our concern framework, but this time surveying approaches that are capable of addressing artifacts other than code. We note that the top half is more sparsely populated, but that work certainly is beginning to fill out the four canonical approaches.

The research agenda here is analogous to that of code: continue to work within each quadrant, build bridges among quadrants, and in

so doing use good science. Two additional key research questions arise, however.

First, to what degree are the four canonical approaches applicable to other kinds of artifacts, and do the strengths, weaknesses, and fundamental limitations stay the same? On the one hand, one can argue “yes”, as the only assumptions we made in formulating our framework are that there are representations for artifacts and concerns and that the two must be connected somehow (using any of the four canonical approaches). On the other hand, some representations are more difficult to deal with. How might one mine, refactor, and maintain concerns in a requirements document, or formulate an intensional query over an architecture specification involving multiple views? We believe the answer, therefore, will likely be much more nuanced than initially assumed, and involve careful study of the approaches and the underlying representations.

Second, concerns are certainly not compartmentalized to individual life cycle phases; the impact of scattering and tangling must be addressed in full, across the development life cycle. This requires an ability for concerns to persist across the life cycle, but it would be naive to assume that the same set of concerns simply overlays the development activities in each phase. Rather, their nature will change, as early high-level concerns break apart into large sets of detailed concerns regarding the code, or, vice-versa, certain low-level concerns aggregate. Some work has begun to address these issues, with CME [11] and SOD [8] offering promising ideas.

5. CONCLUSIONS

The principle of separation of concerns has long guided the field of software engineering. This paper is similarly influenced by its underlying idea, but provides a shift in the foundation upon which future work can build. Particularly, we believe the goal should not be to always separate, but rather it should be to minimize the impact of scattering and tangling – the phenomena that arise since a perfect separation of concerns simply cannot be achieved.

Our concerns framework is built upon this observation; it allows us to compare a number of approaches that to date often have been considered separate. For concern-based software engineering to succeed, these approaches must be brought together, not just in addressing concerns in code, but also in taking a concern-centric view of the entire software life cycle.

6. REFERENCES

- [1] Bass, L., P. Clements, et al. 2003. Software Architecture in Practice, Addison-Wesley Longman Publishing Co., Inc.
- [2] Baldwin, C. Y. and K. B. Clark. 1999. Design Rules: The Power of Modularity Volume 1. MIT Press.
- [3] Baniassad, E., P. C. Clements, et al. 2006. Discovering Early Aspects. IEEE Software: 23(1): 61-70.
- [4] Booch, G., J. Rumbaugh, et al. 1999. The Unified Modeling Language user guide. Addison Wesley Longman Publishing Co., Inc.
- [5] Booch, G. 1986. Object-oriented development. IEEE Trans. Softw. Eng. 12(2): 211-221.

- [6] Brichau, J., M. Glandrup, et al. 2002. Advanced Separation of Concerns. Proceedings of the Workshops on Object-Oriented Technology, Springer-Verlag: 107-130.
- [7] Chitchyan, R., M. Pinto, et al. 2009. Report on early aspects at ICSE 2009: workshop on aspect-oriented requirements engineering and architecture design. SIGSOFT Softw. Eng. Notes: 34(5): 30-35.
- [8] Clarke, S., W. Harrison, et al. 1999. Subject-oriented design: towards improved alignment of requirements, design, and code. *14th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, 325-339.
- [9] Dijkstra, E. W. 1974. EWD 447: On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, 60-66.
- [10] Harrison, W. and H. Ossher. 1993. Subject-oriented programming: a critique of pure objects. *SIGPLAN Notes*: 28-10, 411-428.
- [11] Harrison, W., H. Ossher, et al. 2005. Concern modeling in the concern manipulation environment. *2005 Workshop on Modeling and Analysis of Concerns in Software*, 1-5.
- [12] Kellens, A., K. Mens, et al. (2007). A survey of automated code-level aspect mining techniques. *Transactions on aspect-oriented software development IV*, Springer-Verlag: 143-162.
- [13] Kersten, M. and G. C. Murphy. 2006. Using task context to improve programmer productivity. *14th ACM SIGSOFT international symposium on Foundations of Software Engineering*, 1-11.
- [14] Kersten, M. and G. C. Murphy. 2005. Mylar: a degree-of-interest model for IDEs. *Proceedings of the 4th international conference on Aspect-oriented software development*. Chicago, Illinois, ACM: 159-168.
- [15] Kiczales, G., J. Lamping, et al. 1997. Aspect-oriented programming. *11th European Conference Object-Oriented Programming*, 220-242.
- [16] Kruchten, P. 1995. The 4+1 View Model of Architecture. *IEEE Software*, 12(6): 42-50.
- [17] Liu, J., R. R. Lutz, et al. 2005. Mapping concern space to software architecture: a connector-based approach. *SIGSOFT Softw. Eng. Notes*: 30(4): 1-5.
- [18] Nistor, E. C. and A. van der Hoek 2009. Explicit Concern-Driven Development with ArchEvol. *Automated Software Engineering (ASE 09)*, 185-196
- [19] Ossher, H. and Tarr, P. Multi-dimensional separation of concerns and the hyperspace approach. 2001. *In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer.
- [20] Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*: 15-12, 1053-1058.
- [21] Robillard, M. P. and G. C. Murphy. 2002. Concern graphs: finding and describing concerns using structural program dependencies. *24th International Conference on Software Engineering*, 406-416.
- [22] Robillard, M. P. 2005. Workshop on the Modeling and Analysis of Concerns in Software. *SIGSOFT Software Engineering Notes*: 30-4, 1-3.
- [23] Sampaio, A. and A. Rashid. 2008. Mining early aspects from requirements with ea-miner. *Companion of the 30th international conference on Software engineering*. Leipzig, Germany, ACM: 911-912.
- [24] Stanley M. Sutton, J. and I. Rouvellou. 2002. Modeling of software concerns in Cosmos. *Proceedings of the 1st international conference on Aspect-oriented software development*. Enschede, The Netherlands, ACM: 127-133.
- [25] Tarr, P., H. Ossher, et al. 1999. N degrees of separation: multi-dimensional separation of concerns. *21st International Conference on Software Engineering*, 107-119.
- [26] Tekinerdogan, B., F. Scholten, et al. 2009. Concern-oriented analysis and refactoring of software architectures using dependency structure matrices. *Proceedings of the 15th workshop on Early aspects*. Charlottesville, Virginia, USA, ACM: 13-18. Treude, C. and M.-A. Storey (2009).
- [27] ConcernLines: A timeline view of co-occurring concerns. Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society: 575-578.
- [28] Wadler, P. 1992. The essence of functional programming. *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Albuquerque, New Mexico, United States, ACM: 1-14.