An Experimental Aspect-Oriented Design Methodology

Mario Pavlov[1] and Daniela Gotseva[2]

**[1] Computer Systems Department, Technical University of Sofia
Sofia, Bulgaria**

**[2] Computer Systems Department, Technical University of Sofia
Sofia, Bulgaria**

Abstract

This article discus the software design methodologies as an unexplored area of the aspect oriented software design. It analyzes the properties of software design methodologies and proposes an experimental aspect-oriented design methodology. The article contains a hypothetical high level run of the proposed experimental aspect-oriented design methodology and a concise analysis of the properties of the methodology as well as what it would mean to execute it in the real world.

Keywords: *Software Design Methodology, Aspect-oriented Software Design, Object-oriented Software Design, Aspect-oriented Design Methodology, Crosscutting Concern, Core Module, Software Design Methodology, Composition, Software Design Methodology Properties.*

1. Introduction

A software design methodology provides guidelines to creating software designs. It is an ordered collection of rules, procedures and processes that help drive the creation of software designs. Software design methodologies began to emerge and develop in the 1960s. Now we have a good number of methodologies that are well-defined and proven in practice. Different methodologies address different problem domains. The idea of software design methodologies is to provide a systematic approach to translate real world problems into software designs. Usually, such problems can be described using specific requirements and use cases. A methodology is executed to help develop a software design that satisfies all requirements and use cases defined initially, plus any requirements and use cases that can potentially arise at some future point. An important property of software design methodologies is that they employ logical and systematic steps and are not inspired on the spur of the moment, which leads to more robust designs. Some of the most popular and widely used methodologies include [1]:

- top-down design
- bottom-up design
- structured design
- object-oriented design

The top-down design methodology starts with a top-level description of the system and refines it systematically to progressively lower levels. Through a process of decomposition, the system is broken down into smaller functions until individual modules can be defined, as shown in Fig. 1.

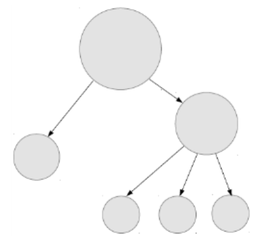

Fig. 1 Top-down Design Methodology.

The top-down design works well when the problem is thoroughly defined beforehand.

The bottom-up design methodology is analogous to the top-down design methodology but instead of starting from the top level down, it starts from the bottom level up. In other words, the basic modules of a system are identified first, along with their relationships. Then higher level concepts are composed based on the lower level concepts. The key benefit of the bottom-up approach is that modules can be evaluated during the system design. It is also suitable for problems that are not well-defined initially.

The structured design methodology is a data flow-oriented approach. It uses a notational scheme called the data flow diagram. The data flow diagram has two perspectives, the flow of data and the transformations the data undergoes. The structured design methodology has three main concepts:

- composition and refinement of the design

- separation of issues into abstraction and implementation
- evaluation of the results

The structured design starts by identifying the data inputs, outputs and a description of the functional modules that transform the data. The next step in the design is to identify the overall interrelationships of the transformations and data flows. As a result, we can define the modules and their relationships. The structured design methodology is widely used in data processing systems.

The object-oriented design methodology has three main concepts:

- modularity
- abstraction
- encapsulation

The first step in object-oriented design is to map the real world problem into classes. The main property of the classes is that they can be organized into a hierarchical inheritance structure. The instances of these classes are called objects and they encapsulate data and operations. The fact that the object-oriented design methodology creates a model of the real world and maps it to a software structure, makes object-oriented design a general purpose methodology. At the same time object-oriented design also becomes dependent on the programming language.

Although different software design methodologies have different mechanics they all aim to provide a means to create good designs to solve real world problems. It is the designers' responsibility to select the right methodology for the right problem and execute it correctly, so that the result is an optimal solution to the problem.

2. Properties of Design Methodologies

Software design methodologies have three main properties that we can analyze, namely applicability, effectiveness and complexity. These properties have specific relationships with one another and the extent to which they exist within a methodology is very important for its internal equilibrium. The applicability of a software design methodology determines the domains where it can be used, the problems that it is able to solve, and in what environment it can be executed. The effectiveness of a software design methodology is the quality of the resulting solution. It is usually measured by the robustness and extensibility of the output software design. The complexity of a software design methodology is the amount of resources it costs the designer. That is the amount of time, computer, and human resources needed to successfully execute the methodology. Therefore, a good software design methodology is one that maximizes applicability and effectiveness, but minimizes complexity. As stated above, each of these properties contributes to the equilibrium within a software design methodology as described in Table 1.

Table 1: Software Design Methodology.

	applicability	***effectiveness***	***complexity***
applicability		As applicability increases, effectiveness decreases proportionally. The more problems a methodology is able to address, the less effective the solution. In this case, the extensibility of the resulting design usually suffers.	As applicability increases, complexity is likely to increase as well. This is because more resources may be required to address a higher number of problems.
effectiveness	As effectiveness increases, applicability decreases proportionally. It is a lot easier to address a single or a few similar types of problems. Also a solution to a lower number of problems, tends to be simpler, which leads to higher effectiveness.		When effectiveness is high, complexity is likely to be low. The key here is that effectiveness implies simplicity, which means that less resources are likely to be required.
complexity	When complexity is high, applicability is usually also high. This can happen if the problem domain is broad or more than one problem domains are addressed. In this case a high amount of resources is required.	Maintaining low complexity contributes to effectiveness because it is cheaper to execute the methodology.	

It is essential for a good design methodology to maintain a reasonably good balance between these properties. Sometimes different methodologies are combined or modified to achieve a good balance. This is the main driver for methodology development. In all fields of engineering it is almost always better for a tool to do one thing and do it well. In software engineering, a software design methodology is a tool for creating software designs. So it should be more practical for methodologies to address specific and well-defined problem domains. As long as the problem domains contain problems that are closely related, the applicability of a methodology remains high and it still "does one thing". When a problem exists and there are no methodologies that exactly apply to it, it is a good idea to consider using a combination of methodologies to address the problem or compose a new methodology for the problem.

3. Composing a Design Methodology

Composing a software design methodology is not a simple task. It requires a very good understanding of the problem domain as well as overall proficiency in software design and software design methodologies. However, these requirements should not serve as an excuse not to experiment with ways to compose design methodologies. As already discussed, a software design methodology is an ordered collection of rules, procedures, and processes to solve software design problems. This high level definition seems to have resemblance with the definition of an algorithm. Therefore, it should be possible to derive the following analogy: a software design methodology is to a design problem as an algorithm is to a mathematical problem. This leads to the question: Can algorithm design paradigms be used to compose software design methodologies? A good place to start looking for an answer would be to conduct experiments. There are a number of algorithm design paradigms, such as "divide and conquer", "dynamic programming" and "greedy algorithm", which could serve as likely candidates for the experiment.

The "dynamic programming" [2] design paradigm seems the most intuitive to start experimenting with. We can describe it as follows: to solve a given problem, first the problem is broken down into subproblems that are simpler to solve and the solutions to the sub-problems are combined into an overall solution to the original problem. Often many of the sub-problems are the same, so once a sub-problem is solved, its solution could be looked up later on when the same sub-problem arises, see Fig. 2.

Following the key idea of the dynamic programming design paradigm it should be possible to compose a software design methodology for an arbitrary problem domain. Let us experiment with creating an aspect oriented design methodology using the dynamic programming design paradigm.

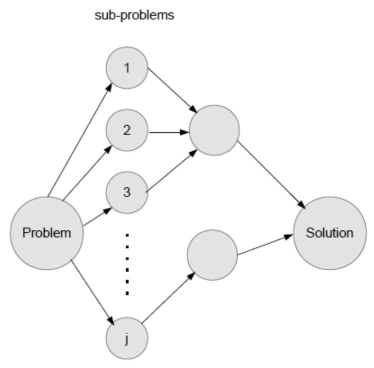

Fig. 2 Sub-problems Design Methodology.

4. An Experimental Aspect-Oriented Design Methodology

First, let us try to model what a good aspect-oriented design methodology would look like. One of the main properties of such a methodology is that at some point the core modules should be identifiable in a way that allows the object-oriented design methodology to be applied to them. This implies that the starting point of the methodology should be either the communication between the core modules (whatever they might be) or the crosscutting [2] modules. It seems more natural for an aspect-oriented design methodology to start from the crosscutting concerns.

Given the problem description, it should be possible to just infer some of the crosscutting concerns, such as tracing, caching, security, and transaction management. Therefore, the first step of the methodology would be to define the crosscutting concerns that are obvious from the problem description. In order to define the rest of the crosscutting concerns we have to start defining and refining the levels of the system. So the next step of the methodology would be to start defining the levels of the system. This should be an iterative process. After each iteration we should try to identify crosscutting concerns. When no more crosscutting concerns can be identified and the system levels are in place, the iterative step is over. At this point almost all crosscutting concerns should be clear and we can start the refinement of the core modules as the next step. For the core modules, the object-oriented design methodology can be applied directly. Because most crosscutting concerns are already clear, it should be rather easy to refine the core modules as we do not have to worry about crosscutting functionality. During this step it is possible for more crosscutting concerns to arise. If that happens we should stop, go back to step two and adjust crosscutting concerns and the system levels accordingly, similarly to the concept of backtracking [4-6]. To a certain extent, this step can also be viewed as iterative. After the core and crosscutting modules are refined, the system should be reviewed and verified as a whole. This means that the designer has to prove, for instance through use cases, that the resulting design is able to satisfy the requirements or in other words solve the problem.

The methodology described above can be summarized in four steps:

1. Identify crosscutting concerns that are immediately obvious from the problem description.
2. Define system levels iteratively and identify crosscutting concerns after each iteration.
3. Apply the object-oriented design methodology with an option to go back to step two if more crosscutting concerns arise.
4. Review and verify the resulting design to prove its legitimacy.

For a visual representation of the methodology, see Fig. 3.

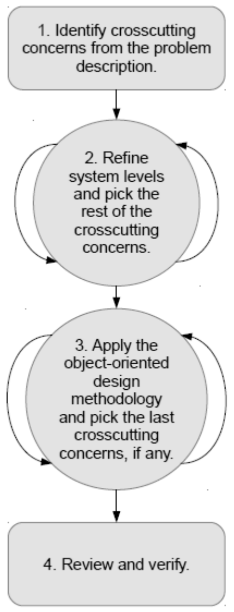
1. Identify crosscutting concerns from the problem description.
2. Refine system levels and pick the rest of the crosscutting concerns.
3. Apply the object-oriented design methodology and pick the last crosscutting concerns, if any.
4. Review and verify.

Fig. 3 Visual Representation of the Methodology.

The best way to test the aspect-oriented design methodology is to apply it to a problem.

5. High-Level Test Drive

For the purpose of this article, it would be impractical to try to apply the methodology to a real full scale problem. However, we can attempt to execute it on a high level to give an idea of what a real-world execution would look like. Let us use a classical problem, such as a bank account management system. For this high-level run we do not need a strict definition of the problem. So let us define it in a more general way as follows:

Design a multi-user bank account management system that supports balance check, deposit and withdrawal. Every operation must take no more than "n" seconds in a single-user scenario.

Given the problem above we can initiate the execution of the experimental aspect-oriented design methodology.

1. In the first step, we infer the crosscutting concerns from the problem description. The first part of the problem description states that the system should be multi-user. This means that we are going to have security and transaction support. The second part of the problem description states that there is a maximum limit to the execution time of each operation. This means that we are going to have performance monitoring and/or tracing. At this point we cannot pick more crosscutting concerns, so we move forward.
2. Here we refine the system's levels and pick the unclear crosscutting concerns leftover from the previous step. From the problem description we can immediately divide the system in the three classical parts: persistence, business logic and user interface. Because this is a multi-user system and we separated the system in three parts, we should have some sort of caching on the user interface level to avoid round-trips to the persistence level for the same data multiple times. To keep it simple, we stop here and declare this step as done. We already have a high level definition of the system parts and several crosscutting concerns (security, transaction support, performance monitoring and caching).
3. In this step, we should execute the object-oriented design methodology on the system levels we defined in the previous step. Executing the object-oriented design methodology is out of the scope of this article so we just declare this step as done and move forward.

4. After we have the entire design in place, we can conduct a review. First, we need to check if the system satisfies the functional requirements, such as user access control, balance check, deposit and withdrawal. This can be done through use-cases. Every major functionality should be checked against the problem description. Then we need to verify the non-functional requirements, such as responsiveness and data staleness. When we are satisfied with the design as a whole we are ready to proceed with implementation.

This high-level run shows that the experimental aspect-oriented design methodology can be used as a tool to develop software designs. Of course, a real execution of the methodology on a real problem will be a lot more detailed and can have multiple iterations in step two and three. The above high-level description of the execution of the methodology is meant to serve only as an illustration of what it would mean to execute the methodology in the real world. For a graphical representation of this illustration, see Fig. 4.

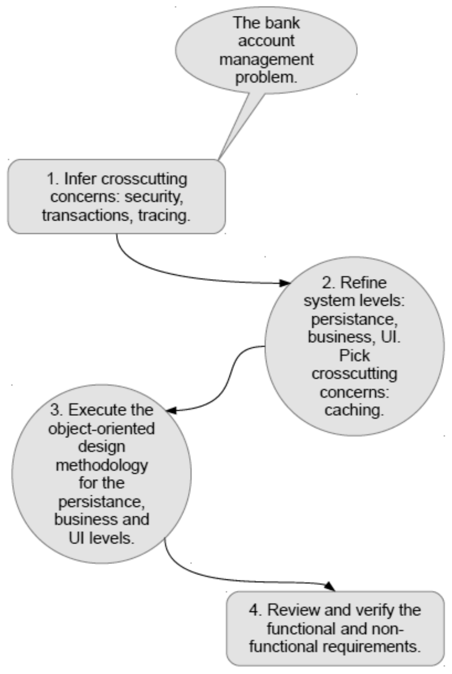
The bank account management problem.
1. Infer crosscutting concerns: security, transactions, tracing.
2. Refine system levels: persistance, business, UI. Pick crosscutting concerns: caching.
3. Execute the object-oriented design methodology for the persistance, business and UI levels.
4. Review and verify the functional and non-functional requirements.

Fig. 4 Graphical Representation of the Methodology.

6. Conclusion

The aspect-oriented programming paradigm is still not used as widely as the object-oriented programming paradigm. Therefore, a lot of areas in the aspect-oriented software design are still left unexplored. The subject of this article is to propose an experimental aspect-oriented software design methodology that can help drive further research in this area. In a sense the aspect-oriented design methodology extends from the object-oriented design methodology in a way parallel to the way the aspect oriented programming paradigm extends from the object-oriented design paradigm. The key property of the proposed experimental methodology is that it picks up the crosscutting concerns before the object-oriented design methodology is applied. In this way the object-oriented design methodology is applied only to the core modules of the system, which greatly simplifies the execution of the object-oriented design methodology and we expect that the resulting designs would be more robust. As a result, the entire aspect-oriented design methodology becomes easier to understand and simpler to execute. To sum up, the entire aspect-oriented design methodology is at most as complex as the object-oriented design methodology.

References

[1] B. Kok Swee Khoo, A Survey of Major Software Design Methodologies, http://userpages.umbc.edu/~khoo/survey2.html
[2] R. Bellman, The theory of dynamic programming, Bulletin of the American Mathematical Society, 1954.
[3] D. Gotseva, M. Pavlov, Aspect Orientation: The Path to Highly Modular Software Design, Computer and Communication Engineering, 2012.
[4] E. Gurari, Backtracking algorithms "CIS 680: DATA STRUCTURES: Chapter 19: Backtracking Algorithms", 1999.
[5] R. Laddad, AspectJ in Action, Enterprise AOP with Spring, Manning Publications, 2010.
[6] J. D. Gradecki and N. Lesiecki, Mastering AspectJ: Aspect-Oriented Programming in Java, Wiley, 2003.

Daniela Gotseva is associate professor, PhD and Vice Dean of Faculty of Computer Systems and Control, Technical University of Sofia, from 2008 with primary research interest of programming languages and fuzzy logics. She is a member of the IEEE and the IEEE Computer Society.

Mario Pavlov is PhD student at Faculty of Computer Systems and Control, Technical University of Sofia, from 2010, with primary research interest of aspect oriented programming and programming languages.