A problem-oriented theory of pattern-oriented analysis and design

Conference or Workshop Item

A Problem-Oriented Theory of Pattern-Oriented Analysis and Design

Jerry Overton[1] Jon G. Hall[2] Lucia Rapanotti[3]
[1] Computer Sciences Corporation, joverton@csc.com
[2] The Open University, j.g.hall@open.ac.uk
[3] The Open University, l.rapanotti@open.ac.uk

Abstract

The overall goal of this work is to provide problem-oriented support for Pattern-Oriented Analysis and Design (POAD) so that (i) we may construct a better understanding of the relationship of POAD to other software development approaches and (ii) we can extend the reach of problem-orientation to design patterns. This paper extends our previous contributions to show how both high-level and detailed design phases can be given a problem-oriented encoding.

Keywords: POAD, Problem Oriented Engineering, design patterns, software requirements, validation.

1 Introduction

The work described in this paper is intended to make two unique contributions. First, it introduces a framework for relating practical tools for software design, such as Pattern-Oriented Analysis and Design (POAD, [1]) and automated testing, together by codifying them within a problem-oriented meta-design framework in terms of which their interrelation can be seen and reasoned about. Second, it makes a practical theory of design available within the software practitioner's toolset by introducing a systematic and rigourous approach that can be used in combination with existing software development methods, instead of attempting to replace them.

In [2], we began a problem-oriented encoding of POAD in Problem Oriented Engineering (POE, [3]), showing how *design refinement* can take the form of refactoring existing software through design patterns. In this paper, we extend our codification of POAD in POE to: (i) consider in more detail the nature of the validation of a solution in POAD and its relation to the testing of requirements; and (ii) include *high-level design* ([1]), in addition to *design refinement*.

This paper proposes a formal model of building software systems using software design patterns. The model is applicable to problems where the requirements can be expressed in simple conjunctive form and where suitable patterns exist that can be interpreted as instructions for satisfying the problem's quality requirements (also known as non-functional requirements) through the suggestion of an architecture.

Our approach differs in a number of ways from pattern formalisation approaches, such as those presented in [4]. Instead of addressing the application of patterns to source code either as resusable software components (for instance, [5], [6]) or aspects ([7]), we focus our description on software system design.

1.1 The Portal Problem

We use the following example for illustration throughout; the example is representative of the class of development problems the first author, an experienced software engineer with specific pattern expertise, encounters within his organisation. The problem concerns the design of a corporate customer portal system for a bank. The system is required to satisfy the following requirements and will operate within a context including system end-users and content designers:

- *Portal Function:* the system should secure, display, and allow the creation of content;
- *Modifiable:* the system should be easily maintained and updated;
- *Composable:* the system should integrate various commercial off-the-shelf (COTS) components;
- *Centralized Security:* the system should secure content using a single, centralized function;
- *Decentralized Content Creation:* the system should allow content to be created concurrently by different developers;

- *Uniform Presentation:* the system should present all content with a uniform look-and-feel.

Given our encoding of POAD in POE, our task in this paper will be to design a solution to this problem that exercises that encoding.

1.2 Paper Structure

The paper has the following structure. In Section 2, we describe, briefly, the type of software patterns we consider in our work, as well as POAD and POE, the two approaches we combine. Section 3 presents our codification of POAD into POE, while Section 4 applies it to our example. We offer a discussion and some conclusion in Section 5.

2 Background

After an introduction to design patterns and POAD, we give a (necessarily) brief introduction to the part of Problem Oriented Engineering that underpins our theory.

2.1 Software Design Patterns

A design pattern is a template for solving a commonly occurring software design problem. Typically, a design pattern will be recognised only after software engineering knowledge in a particular area has reached the stage of maturity where software developers no longer need to solve the problem from scratch [8].

A design pattern is said to be *well-documented* when it consists of [9] a *Problem* that describes a particular problem treated by the pattern; a *Context* that characterises the situations in which the problem occurs; *Forces* that list the quality requirements of an acceptable solution; a *Solution* that describes alternatives for solving the *Problem*; and *Consequences* that explain the characteristics and trade-offs of each solution alternative. We restrict ourselves to well-documented patterns in this paper.

2.2 Pattern-Oriented Analysis and Design

POAD [1] is a pattern-oriented framework that allows designers to reuse software design experience and provides the scope for the reuse both of designed artefacts, such as existing components, and for extant justifications of fitness-for-purpose. POAD is *pattern oriented* in the sense of [10], i.e., POAD considers design patterns as basic building blocks for the design of software.

POAD sees development as consisting of 3 phases [1]. First, in *analysis*, patterns are selected from a domain-specific library. Next, in *high-level design*, patterns are composed to produce a high-level design representation, such as a package and/or class diagram. Last, in *design refinement*, the high-level design representation is refined to be more detailed.

Let us consider an application of POAD to the portal problem of Section 1.1. In *analysis*, the problem's requirements, in combination with the first author's knowledge of the domain and his pattern engineering experience, allows us to determine, in the POAD *analysis phase*, the patterns to be used: they are (from [9]) *Layers*, *Container*, *Inteceptor*, *Shared Repository*, *Template View* and (from [11]) *Private Workspace*.

In the *high-level design* phase, we compose the identified patterns and existing software components into a high-level design. Consider the system as *Layers* for security, the portal's content, and its display. Within each layer, use *Containers* to host the relevant portal system's components: in the security container, place an *Inteceptor* that blocks any unauthorised event; make the content container a *Shared Repository* for all content and have it accessed by both the presentation container and content development container; put *Template Views* of selected content in the presentation container and make the content development container a *Private Workspace* where content is created. From this analysis, a Unified Modeling Language (UML) diagram, such as that shown in Figure 1, would be constructed as a record of the high-level system design.

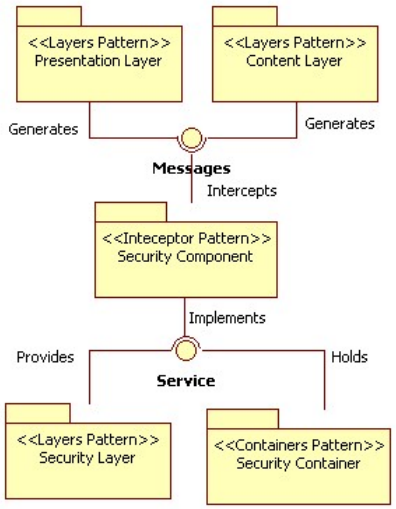

Figure 1. Sketch, in UML, of the Partial Portal Problem Solution

Our high-level design is no more than a sketch, albeit a sketch that relates the known functionalities and qualities of the system; it has grown to the completed version of the figure through the incremental consideration of those features and qualities. The incremental nature of its development is already accepted in the literature and is not a characteristic of our approach; see, for instance, [10]. We note that detailed justifications of the choices, and of their ramifications for the target system, are yet to be made. With the design experience available to us, the choices we made during analysis and high-level design *feel* right. Our confidence in them will be detailed and justified to our and/or the stake holders' satisfaction during detailed design.

2.3 Problem-Oriented Engineering

Problem Oriented Engineering is a general framework for engineering design, a generalisation of the Problem Oriented Software Engineering (POSE, [3]) framework for software engineering. It has a principled basis, with the notion of (design) problem at its centre: a design problem requiring a designed solution. POE prescribes no particular method for development (although heuristics are offered for risk, resource and communications management in [3]). Rather, the intent of POE, and part of the motivation of this paper, is to encompass existing method and methodology within the framework.

2.3.1 Problems and Problem Transformations

In POE, a problem has 3 elements: a real-world environment, W; a requirement, R; and a (to-be-found) solution, S. We write $W, S \vdash R$ to indicate that we wish to find a solution S that in the context of W satisfies R.

What is known of a problem's elements are captured in that element's description; descriptions can be written in any appropriate language from formal to natural language including, for our codification, the UML. The Portal Problem of Section 1.1 is captured in POE as:

Bank Community, *Portal System* $\vdash$ *Portal Requirement*

where:

- *Bank Community* is the intended community of users and content developers of the system to be designed (the problem's context);
- *Portal System* is the solution to be designed; and
- *Portal Requirement* is the conjunction of requirements to be satisfied, that is *Portal Function* $\wedge$ $\ldots$ $\wedge$ *Uniform Presentation* (from Section 1.1)

Engineering design under POE proceeds in a stepwise manner with the application of problem transformations, examples of which appear below. Formally, problem transformations conform to the following rule form: a *conclusion* problem $P(:\ W, S \vdash R)$, *premise* problems $P_i(:\ W_i, S_i \vdash R_i)$, $i = 1, ..., n$, $(n \geq 0)$ and *justification* J combine in a rule named N to give transformation step $\dfrac{P_1 \ldots P_n}{P}\ {}^{[\mathrm{N}]}_{\langle\langle \mathrm{J} \rangle\rangle}$ which means that S is a solution of $W, S \vdash R$ whenever $S_1, ..., S_n$ are solutions of $W_1, S_1 \vdash R_1, ..., W_n, S_n \vdash R_n$ and justification J is validated by all relevant stake-holders. J collects the evidence of adequacy of the transformation step. Essentially, through this rule, problems are transformed into others that may be easier to solve[1] until only problems with a known fit-for-purpose solution are left. Our task in this paper is to encode POAD as transformations that comply with this rule.

An initial problem, such as the Portal Problem above, forms the root of a *development tree* with problem transformations applied to extend the tree upwards towards its leaves. A problem is solved if the development tree is complete and fully justified to the satisfaction of all stake-holders.

Steps can be of arbitrary size in POE, large-steps being composed of smaller ones; indeed whole sub-development trees can be folded together into a single step. In the general POE setting, then, the goal of problem-solving is to find transformations, arguments and evidence of adequacy that arrive at a solution. In this paper, our goal is more specific: through POE we wish to encode a POAD-style development that allows us to solve problems like the Portal Problem. To do this we will define large-step transformations that allow a high-level design to be produced in an incremental fashion and then detail each large step as a sequence of small steps so that an adequacy argument attributable to POAD is produced.

Two more features of POE are needed for us to complete the technical basis of our encoding: in POE, an *AStruct* (short for *Architectural Structure*) is used to introduce an architecture into the solution. An *AStruct*, $AS[g](c_1, ..., c_n)$ has name AS, and combines a known structure, the *glue* g (of arbitrary complexity), together with the c_i which are elements of the solution yet to be designed. The solution is modified through the *solution interpretation* transformation SolInt[2] which requires the architecture to be justified as adequate:

$$\frac{W, AS[g](c_1, ..., c_n) \vdash R}{W, S \vdash R}\ {}^{[\textsc{SolInt}]}_{\langle\langle \text{Explain and justify the use of } AS[g](c_1, ..., c_n) \rangle\rangle}$$

[1] Or that lead to others that are easier to solve.

[2] There is a *requirements interpretation* step, ReqInt, too.

Finally, once an *AStruct* has been applied, the *Solution Expansion* transformation SolExp expands the problem context with the glue whilst simultaneously refocussing on the problems to find the c_j that remain to be designed. The requirement and context of the original problem are propagated to all sub-problems. In this paper, n is either 1 or 2, in which case SolInt is either

$$\frac{W, g, c_1 \vdash R}{W, S{:}AS[g](c_1) \vdash R}\ [\text{Sol-Exp}] \quad \text{or} \quad \frac{W, g, c_2, c_1 \vdash R}{W, S{:}AS[g](c_1, c_2) \vdash R}\ [\text{Sol-Exp}]$$

Given POE's notation agnosticism, we can see Figure 1 as an *AStruct* in which the glue is captured in the overall class diagram, and the components are the packages whose design remains to be detailed.

3 Codifying POAD in POE

In this section we present our approach to codifying POAD within POE. In particular, Section 3.1 briefly recalls our previous work [2] towards such an encoding, while Section 3.2 builds on these foundations to provide the novel contribution of this paper.

3.1 From patterns to pattern problems

In [2] we encoded the application of a POAD pattern as a sequence of POE transformations. This was achieved through the following observation. Although POAD is motivated by practice while POE is motivated by theory, there is a good mapping in their view of context, requirement and solution. This led us to the codification of the components of a design pattern (recall Section 2.1) in POE as summarised in the following table:

Name	*Pattern*
Context	context description, $\mathcal{W}$
Forces	quality requirement, $\mathcal{Q}$
Problem	problem matching $P_{Pattern} : \mathcal{W}, \mathcal{S} \vdash \mathcal{R} \wedge \mathcal{Q}$
Solution	$PA_1[g_1](c_1)$, $PA_2[g_2](c_2, c_2')$, *etc*
Cons.	characteristics of each PA

In detail, we associate with a well-documented pattern a POE problem template $P_{Pattern} : \mathcal{W}, \mathcal{S} \vdash \mathcal{R} \wedge \mathcal{Q}$ in which the pattern Context matches the problem Context and the pattern Forces are captured by a quality requirement Q separate from any functional requirement R. Also, the pattern Solution is captured by a collection of *AStructs*, i.e., the solution architectures

$$\cfrac{\cfrac{\cfrac{W, g, c \vdash R}{W, PA[g](c) \vdash R}\ [\text{SolExp}]}{W, PA[g](c) \vdash R \wedge Q}\ [\text{ReqInt}]\ \langle\langle \text{Q-Test} \rangle\rangle}{W, S \vdash R \wedge Q}\ [\text{SolInt}]\ \langle\langle \text{Solution assessed against Consequences, } PA \text{ chosen} \rangle\rangle$$

Figure 2. Application of a well-documented pattern to a problem $W, S \vdash R \wedge Q$ that matches its problem template $P_{Pattern}$

suggested by the pattern application[3], while the pattern Consequences are used to guide our choice of which of the solution architecture are most appropriate based on the various trade-offs the PA_i address, their contribution to functional requirements (i.e., which contribution PA_i makes to R), *etc.*

Following on from [2], after pattern choice, we reduce the problem of meeting $R \wedge Q$ to that of demonstrating that the solution will satisfy a validation condition, here called a *Quality Assurance (or Q-)Test* [4] as illustrated in Figure 2. The application of the *Pattern* to a problem-matching $P_{Pattern}$ suggests the architecture, encoded as the *AStruct* $PA[g](c)$, and assessed against the pattern's Consequences. Introducing the architecture through *Solution Interpretation* transforms the problem with justification recorded as the engineering expertise encoded in the pattern and the consequences of its application. Further application of the design pattern justifies a *requirement interpretation* ([3]) that removes Q. This is a valid transformation as long as the *Q-Test* demonstrates the *AStruct*'s satisfaction of the quality requirement in a way adequate for all stake holders. We complete the transformation with the SolExp shown that allows the focus to move to the component c that remains to be found.

3.2 Full pattern encoding

In this section, we extend our POAD codification of [2] in two ways. First, we add to the condification the clear distinction between high-level design, where the details of context and solution are not known and transformations are not fully justified, and design refinement, where the transformations' justifications are made adequate as are the details of context and solution. Second, we recognise that, in addition to satisfying quality requirements, a pattern can contribute to the satisfaction of functional requirements. This leads

[3] This may be described, perhaps, as UML class diagrams, such as that of Figure 1.

[4] Generalised from the *stress test* in [2].

us to consider the more general pattern of application we discuss below.

We know from [12] that the definition of a quality requirement can be made operational by using *concrete quality attribute scenarios* to characterize the requirement. These scenarios, specific to both the quality requirement they describe and the context to which they apply, map an understanding of a quality requirement onto specific system stimuli and observable behaviour. The *Q-Test* we have introduced simply verifies a concrete quality attribute scenario under this scheme.

Given that a problem might have many requirements and qualities, and an arbitrarily complex context, we generalise the form described in Figure 2 as follows[5]:

$$\frac{W, g, c \vdash R_1' \wedge R_2 \wedge Q_2 \wedge \cdots \wedge Q_n}{W, PA[g](c) \vdash R_1' \wedge R_2 \wedge Q_2 \wedge \cdots \wedge Q_n}\ [\textsc{SolExp}]$$

$$\frac{W, PA[g](c) \vdash R_1' \wedge R_2 \wedge Q_2 \wedge \cdots \wedge Q_n}{W, PA[g](c) \vdash R_1 \wedge R_2 \wedge Q_2 \wedge \cdots \wedge Q_n}\ [\textsc{ReqInt}]\ \langle\langle Q\text{-Test}\rangle\rangle$$

$$\frac{W, PA[g](c) \vdash R_1 \wedge R_2 \wedge Q_1 \wedge \cdots \wedge Q_n}{W, S \vdash R_1 \wedge R_2 \wedge Q_1 \wedge \cdots \wedge Q_n}\ [\textsc{SolInt}]\ \langle\langle \text{Template matched, Solution assessed, } PA \text{ chosen}\rangle\rangle$$

Within the context W, *Q-Test* invokes R_1 as stimulus to PA and checks for a response that represents an operationalized form of Q_1. The satisfaction of *Q-Test* sufficiently demonstrates that PA satisfies Q_1, with the solution contributing functionality to $R1$ leaving $R1'$. Assuming that suitable design patterns exist, under repeated application of this general transformation, we can discharge all quality requirements.

In POE it is possible to create big steps from small ones, and to apply a step without having completed its justification on the understanding that, subsequently, we will complete that justification. Without the choice of *Q-Test* being made (and a test strategy should, in general, be chosen with reference to the external stakeholders) we can forge the big step that is Figure 2, thus:

$$\frac{W, g, c \quad \vdash R_1' \wedge R_2 \wedge Q_2 \wedge \cdots \wedge Q_n}{W, S \quad \vdash R_1 \wedge R_2 \wedge Q_1 \wedge \cdots \wedge Q_n}\ [\textsc{SolInt}]\ \langle\langle \text{Application of Pattern } P\rangle\rangle$$

on the understanding that justification for the pattern and the definition and justification of the *Q-Test* will be provided later. This apply-and-justify-later idea delimits high-level design from detailed design.

Given this big step, we can consider using it to solve problems knowing that, after a candidate design is found, we can go back and justify it through expanding the step, applying the pattern definitions and completing all *Q-Tests*. The justification will be refined through this reconsideration of the high-level design to a justification that either adequately discharges the design through the collection of *Q-Tests* collected, or fails to discharge. For the latter, another high-level design might be sought or, if none such exists, the acceptance that no design can be made (at least based on our initial choice of pattern language).

[5]For brevity, only the case of a single component c to be designed is considered.

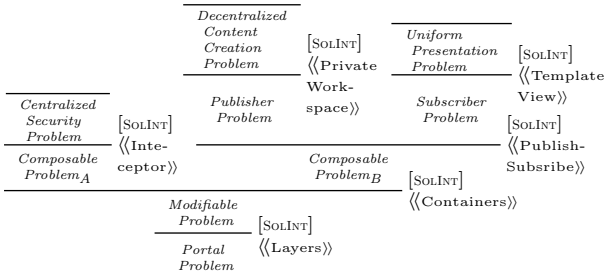

Figure 3. Pattern-Oriented Solution to the Portal Problem, using the general POE transformation.

4 Solving the Portal Problem

We now turn to the application of our codification of POAD in POE by addressing the Portal Problem of Section 1.1.

We will assume that the pattern language defined in Section 3 consists of well-documented patterns. Here, for example, is the *Inteceptor* pattern:

Name	*Inteceptor*
Context	any software environment whose behavior derives from a network service connected by a common, message-based infrastructure [9].
Forces	*Centralized Security*
Problem	a problem in which ...
Solution	*IntArch*[*SecurityInteceptor*](*Services*)
Cons.	characteristics of *IntArch*

By allowing a software engineer and POAD expert free reign, we may apply the big step transformation of the previous section to arrive at the high-level design candidate shown in Figure 3.

The high-level design indicates which patterns were applied and in which order, but provides no justification for the choices, nor which *Q-Tests* will satisfy external stake holders. Expanding the big step transformation allows us to fill in these justification components in the following way. For brevity's sake, we focus

on the *Inteceptor* pattern's application to Figure 3's $Composable\ Problem_A$.

The details are:

$$\dfrac{\dfrac{\dfrac{\dfrac{Centralized\ Security\ Problem : Bank\ Community', SecurityInteceptor, PortalServices \vdash Security\ Function'}{Bank\ Community', IntArch[SecurityInteceptor](PortalServices) \vdash Security\ Function'}\ [\textsc{SolExp}]}{Bank\ Community', IntArch[SecurityInteceptor](PortalServices) \vdash Security\ Function \wedge Centralized\ Security \wedge \ldots}\ [\textsc{ReqInt}]\ \langle\langle \text{Security Test} \rangle\rangle}{Composable\ Problem_A : Bank\ Community', S \vdash Security\ Function \wedge Centralized\ Security \wedge \ldots}\ [\textsc{SolInt}]\ \langle\langle \text{Inteceptor} \rangle\rangle$$

in which the *Q-Test* Security Test should be chosen so as to relate a stimulus to *IntArch* of both authorized and unauthorized operations and checks for an appropriate response.

5 Discussion and Conclusions

Our presented solution to the Portal Problem is pattern-oriented in that it was solved using the composition of design patterns. It is problem-oriented in that it was constructed as a (software) problem and solved using problem transformations. This synthesis of approaches is interesting for a number of reasons. First, we have presented a codification of the steps of building a software system with patterns in POE, without stepping outside of the pattern language or the system in which they are applied (always accepting that a problem-oriented encoding may be unfamiliar to the reader). In an extended presentation we could, we conjecture, express the *AStructs* in any appropriate notation, including as fully detailed UML diagrams—POE is accepting of various notations. Second, our theory is complementary to other approaches that address the application of patterns to source code as reusable software components [6], [5] or aspects [7]. Third, our supporting theory is a formal approach, but we have evidence from other work that other software development approaches combine with it. Our developing expression of POAD within POE holds promise that we will be able better to understand the relationship of POAD to other approaches; more than this, however, we have some evidence that engineering design approaches map to POE understanding how POAD fits within, say, socio-technical engineering.

Acknowledgements

We would like to thank our colleagues in the Computing Department at The Open University for their continuing support, especially Yijun Yu, for his insightful reading and comments on this paper.

References

[1] S. M. Yacoub and H. H. Ammar. *Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems*. Addison-Wesley Professional, 2004.

[2] Jerry Overton, Jon G. Hall, Lucia Rapanotti, and Yijun Yu. Towards a problem oriented engineering theory of pattern-oriented analysis and design. In *Proceedings of 3rd IEEE International Workshop on Quality Oriented Reuse of Software (QUORS)*, 2009.

[3] Jon G. Hall, Lucia Rapanotti, and Michael Jackson. Problem-oriented software engineering: solving the package router control problem. *IEEE Trans. Software Eng.*, 2008. doi:10.1109/TSE.2007.70769.

[4] T. Taibi and O. Now. *Design Patterns Formalization Techniques*. IGI Pub., 2007.

[5] B. Meyer and K. Arnout. Componentization: The visitor example. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(7):23, 2006.

[6] K. Arnout and B. Meyer. Pattern componentization: The factory example. *Innovations in Systems and Software Engineering*, 2(2):65–79, 2006.

[7] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. *ACM SIGPLAN Notices*, 37(11):161–173, 2002.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns. elements of reusable object-oriented software. *Addison-Wesley Professional Computing Series, Reading, Mass.: Addison-Wesley,| c1995*, 1995.

[9] F. Buschmann, K. Henney, and D. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing (Wiley Software Patterns Series)*, volume 4. John Wiley & Sons, 2007.

[10] D. J. Ram, KN Raman, and KN Guruprasad. A pattern oriented technique for software design. *ACM SIGSOFT Software Engineering Notes*, 22(4):70, 1997.

[11] S. P. Berczuk and B. Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley Professional, 2003.

[12] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2003.