Developing adaptable software architectures using design patterns: an NFR approach

Lawrence Chung*, Kendra Cooper, Anna Yi

Department of Computer Science, University of Texas at Dallas, Dallas, TX, USA

Abstract

Just about everything changes, and so should a software system accordingly in order to survive and succeed. But how can we develop such a software system? Lately, a growing number of practitioners have shown great interests in using design patterns towards the development of an adaptable system, since design patterns represent high-level abstractions that reflect the experience of no other than skilled practitioners themselves. According to a given format, design patterns describe the context, problems, solutions and consequences of making specific design decisions. This paper presents, Proteus—a framework which is intended to support the development of adaptable software architectures using design patterns. The main concepts of Proteus are illustrated by way of a home appliance control system.

© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Design pattern; Nonfunctional requirements; Adaptability; Software architecture

1. Introduction

Just about everything changes, and so should a software system accordingly in order to survive and succeed. This is especially true in spirit of Brooks [1] who states that success brings change to most software systems, and often, the greater the success, the greater the changes.

But how can we develop such a software system? Lately, a growing number of practitioners have shown serious interest in using design patterns towards the development of an adaptable system, perhaps because design patterns represent high-level abstractions that reflect the experience of no other than skilled practitioners themselves. In a nutshell, design patterns are high-level abstractions that describe, according to a given format, the context, problems, solutions, and consequences of making specific design decisions [6].

Due to their anticipated pragmatic utility, design patterns are drawing a growing commercial interest, as manifested through their use in commercial frameworks, notably, the Java 2 Platform, Enterprise Edition (J2EE)—an open standard for implementing and deploying component-based enterprise applications, and its baby version, the Java 2 Platform, Mobile Edition (J2ME) to run on mobile devices such as PDAs, Palm-tops, etc.

This paper presents, Proteus—a framework which is intended to support the development of adaptable software architectures using design patterns. Proteus
is an ongoing project \cite{4,5} which takes a goal-oriented approach augmenting the object-oriented approach underlying design patterns. Proteus uses the NFR Framework \cite{3,10}, in which software adaptability, as a nonfunctional requirement (NFR), is treated as a softgoal to be satisficed (i.e., achieved not absolutely but within acceptable limits). Formally speaking, this notion of “satisficing” is critical in understanding what it is really being attempted during software development. This is because on the one hand the meaning of adaptability as part of the problem statement is more often than not context-dependent, unclear and conflicting with other NFRs, and the space of design space to meet the adaptability requirements is potentially huge, if not infinite, on the other. Quite appropriately, Proteus then explores both definitional and design alternatives to satisfice the adaptability softgoal, analyzes tradeoffs among the alternatives, estimates the impact of the alternative design patterns upon software adaptability, and selects among the alternatives in relation to the domain characteristics (i.e., context and problem).

Proteus also benefits from other related work, including \cite{14} which also uses the NFR Framework for developing adaptable software architecture, but not in relation to design patterns and additionally in embedded systems setting. Another source of ideas for Proteus is of course the various work on design patterns, including \cite{2,6,8,9,11–13}. Proteus’ emphasis in adaptability is complementary to \cite{7} which looks at design patterns with agent-orientation as an emphasis.

1.1. Running example

Throughout the rest of the paper, we use as our running example a home appliance control system (HACS). Using a HACS, a user controls, monitors and coordinates home appliances such as air conditioner, microwave ovens, TV sets, indoor/outdoor lights, water sprinklers, and even home security devices and spas. A main part of the system is a controller located typically inside a house, which on the one hand communicates with home appliances, and also with a remote device such as a mobile phone or a palmtop, on the other. There could be various types of changing needs for this kind of system due to changes in the underlying technology (e.g., advancements in communication protocols) and also due to changes in customer needs (e.g., You could be stuck in a traffic jam, when your favorite dish might be just about to start burning according to your setup at the time you left your office).

Section 2 presents an overview of Proteus, in particular the steps to be taken in developing adaptable software architectures using design patterns. Section 3 presents the requirements phase of the Proteus process. Section 4 presents macroscopic architectural design, while Section 5 presents how to analyze and use, or not use, those design patterns that have the potential to enhance software adaptability at a more microscopic level. A summary of the contributions and future directions are described in Section 6.

2. Proteus: overview

Proteus aims to support the development of adaptable software architectures using design patterns. Proteus is an ongoing project \cite{4,5} which takes a goal-oriented approach by adopting the NFR Framework \cite{3,10}, hence augmenting the object-oriented approach underlying the representational mechanisms of design patterns. In Proteus, software adaptability as a nonfunctional requirement (NFR) is treated as a softgoal to be satisficed (i.e., achieved not absolutely but within acceptable limits). Formally speaking, this notion of “satisficing” is critical in understanding what it is really being attempted during software development. This is because on the one hand the meaning of adaptability as part of the problem statement is more often than not context-dependent, unclear and conflicting with other NFRs, and the space of design space to meet the adaptability requirements is potentially huge, if not infinite. Quite appropriately, Proteus then explores both definitional and design alternatives to satisfice the adaptability softgoal, analyzes tradeoffs among the alternatives, estimates the impact of the alternative design patterns upon software adaptability, and selects among the alternatives in relation to the domain characteristics (i.e., context and problem).
Using Proteus, then, an architectural design process would proceed as follows:

1. Post-adaptability requirements, along with any other important NFRs, as well as functional requirements.
2. Refine the NFRs and prioritize them, taking into consideration any particular characteristics of the intended domain.
3. Consider architectural concepts and alternatives, at a macroscopic level, to meet the requirements stated, both functional and nonfunctional.
4. Consider design patterns, at a more microscopic level, to satisfice the architectural concepts and alternatives being considered. This consideration should be done in terms of the context and problems associated with each of the design patterns.
5. Analyze tradeoffs among the alternatives of architectures and their corresponding design patterns, in relation to the adaptability and whatever other NFRs stated, while carrying out impact analysis.
6. Select among the alternatives of architectures and their corresponding design patterns that best satisfice the adaptability and other NFRs in the context of the intended application domain.
7. Compose the selected design patterns into parts of the selected architectural design.

The steps in the entire process are interleaving and iterative, and carried out in terms of a visual representation, called softgoal interdependency graph (SIG), where each node represents a softgoal and each link between a parent goal and its descendants represents the degree to which the descendants (positively or negatively) contribute to the satisficing of the parent softgoal. Accommodating informal, semiformal and formal representations, an SIG acts not only as a means to arrive at the end (an architectural design) but also as a development history, which can later be used to understand the rationale behind various design decisions and make improvements.

3. Refinements of adaptability requirements

In Proteus, the first step is the post-adaptability requirements, along with any other important NFRs, as well as functional requirements (FRs). In this paper, we assume that FRs are stated in an object-oriented (and agent-oriented) manner.

Unfortunately, however, the notion of adaptability is often times unclear whose meaning seems to depend rather heavily on the scope and nature of the particular software development project for which the requirement has been stated. Hence, without an adequate clarification of what adaptability requirements might really mean, any attempt to achieve the requirements is most likely going to be futile. In Proteus, the first step towards developing an adaptable software system is refinements of adaptability requirements.

For example, Fig. 1 shows a SIG for developing an adaptable home appliance control system. The adaptability requirement “The HACS shall be adaptable” is represented as a softgoal (a thin cloud), Adaptability [HACS], which is then refined in terms of an AND-
decomposition (a single arc) into two descendant goals: Detectability [Change in environment] and Transformability [HACS]. Here, an AND-decomposition means that in order to satisfice Adaptability [HACS], we should satisfice both of its descendant softgoals: maximizing the capability to detect changes in the environment and maximizing the capability to transform the system. The latter softgoal, Transformability [HACS], in turn is further AND-decomposed into two subgoals: Recognizability [Change in HACS] and Enactability [Change in HACS], respectively, for maximizing the capability to recognize the needed change to be made in the system and maximizing the capability to make the recognized change. The top goal is also OR-decomposed (a double arc) into Automatic Adaptability [HACS] and Manual Adaptability [HACS], respectively, for making run-time adaptation (for example, using library routines) and for making development-time adaptation. When a goal is multiply decomposed, there can be a Cartesian product of subgoals.

Fig. 2 shows how this is done. Transformability [HACS] is connected to two dark clouds through bolded, green lines (+): Low Coupling [Architecture, HACS] and High Cohesion [Architecture, HACS] (a module which carries out a single function has a high cohesion, and vice versa), which represent architectural-level design techniques for positively satisficing Transformability [HACS]. Fig. 2 also shows the consideration of Indirect Connection [Architecture, HACS] (e.g., connection not based on any explicit procedure invocation) and Mediated Connection (e.g., connection based on a third-party or even multiple parties) as a couple of architectural-level techniques to satisfice Low Coupling [Architecture, HACS].

On the functional side, we note that HACS needs mechanisms for communication between devices and coordination among processes. We also note that HACS needs mechanisms for adapting to different...
needs of the user. For example, when the user is very hungry, the microwave oven may need to respond to the user’s request that it operate maximally to cook the food as fast as it can.

5. Operationalizing architectures using design patterns

Once macroscopic architectural design considerations have been given, the next step in Proteus is to consider design patterns, at a more microscopic level, to satisfice the architectural alternatives being considered. This consideration should be done in terms of the context and problems associated with each of the design patterns.

Several patterns have been identified as (potentially) enhancing the adaptability of real-time software systems [13]. Some of these that seem to fit the nonfunctional and functional needs of the architectural considerations in the previous section include the Wrapper, Reactor, and Strategy Patterns.

The intent of the Wrapper Pattern (refer to Fig. 3) is to encapsulate lower-level functions within type-safe, modular, and portable class interfaces. It helps to (1) avoid tedious, error-prone, and non-portable programming of low-level IPC mechanisms (e.g., through pipes, shared memory, semaphores, sockets, rims, etc.) and (2) combine multiple related, but independent, functions into a single cohesive abstraction. We note here that this pattern uses a third-party, i.e., a wrapper, between the client and the server, hence a mediated connection, in handling IPC mechanisms (for example, between the controller and a home appliance), and enhances cohesion.

The intent of the Reactor Pattern (refer to Fig. 4) is to decouple event demultiplexing and event handler dispatching from the services performed in response to events. It helps to (1) demultiplex multiple types of events from multiple sources of events efficiently within a single thread of control and (2) extend application behavior without requiring changes to the event demultiplexing/dispaching framework. We note here that this pattern does not use any explicit invocation but an event mechanism, hence indirect connection, but at the same time a rather heavy use of events potentially leading to speed performance degradation.

The intent of the Strategy Pattern (refer to Fig. 5) is to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. It helps to extend the policies for advertising, listing, creating, accepting, and executing a service handler without modifying the core algorithm. We note...
here that this pattern also uses "Strategy" as an intermediate object.

Fig. 6 shows how the SIG in Fig. 2 has been further expanded with the three design patterns. Although not shown in the SIG, the description of each design pattern can be associated with its corresponding softgoal (cloud). Note that these design patterns are still treated as softgoals, since their meanings seem still not very clear and also there seems to be no guarantee that they will be absolutely achieved.

Fig. 6 also shows how these potential adaptability-enhancing design patterns are now traded off, in relation to the adaptability and whatever other NFRs stated in the context of the intended application domain. This tradeoff analysis leads to the selection among the competing design patterns, hence among the alternative architectures.

Fig. 6 shows such considerations involving Detectability [Change in environment] and Reactor Pattern [Architecture, HACS]. Detecting changes in the environment is usually a very time-consuming task, hence hurting speed. This negative relationship is shown with a red, bold line (-). Similarly, detection of events for the Reactor Pattern could induce significant performance penalty. For the HACS system, where speed is considered a key NFR, adaptability may need to be compromised in favor of speed. Although omitted here, this process would involve carrying out analysis of the effect of choosing, or not choosing a design pattern upon its parent softgoals and all the way up to the top-level softgoals, using the label propagation algorithm as described in Ref. [3].

The last step in Proteus is to compose the selected design patterns into parts of the selected architectural design. Assume strategy and wrapper patterns are chosen for the system. Now, how would these be used in the target architecture? Some of the possibilities are: no interaction between the two, no overlap between the two but with some direct interactions, some overlap, at least with interfaces, etc.

Fig. 7 shows how the designer has composed the two patterns, the wrapper pattern to activate home appliances and the strategy pattern to allow for different ways of cooking. For example, if the user is hungry, tired and may come home late, then the system may be asked to fully cook the meal by the
expected arrival time, and periodic warming up every 10 min afterwards.

6. Conclusions

Improving software adaptability is a formidable task as much as software adaptation is inevitable in reality. One rather intriguing concept that strongly promotes adaptability is design patterns, which recently has been drawing increasing interests in both academia and industry. This paper has presented Proteus which aims to help software architects develop adaptable software architectures using design patterns. In particular, this paper has presented how to analyze and use design patterns as potential adaptability enhancers in developing software systems.

Future work includes use of the Proteus approach for J2EE/J2ME and other types of system applications. Another line of research concerns building and populating the Proteus knowledge bases towards a tool support. Proteus’ knowledge base of design patterns will be organized along a set of hierarchies, capturing both general and adaptability-specific patterns (e.g., [6,9,11,13]). A more fundamental research would be considering methods for mapping the pattern language categories into the SIG representational categories.

Acknowledgements

We would like to thank former members of the ADAPT group for their dedicated surveys of both real-time and other types of design patterns, as well as
their installation of JBoss. We also appreciate the insightful comments and questions from the audience at the Adaptable Software Architectures (ASA) session in SERP’02 (International Conference on Software Engineering Research and Practise).

References


Dr. Lawrence Chung is currently an Associate Professor in the Department of Computer Science at the University of Texas at Dallas. He received his PhD from University of Toronto, and is currently on the Editorial board of Requirements Engineering Journal. He is the principal author of a book, “Non-Functional Requirements in Software Engineering”, which is being adopted in extending object-oriented analysis to goal-oriented and agent-oriented analysis. His research interests include Requirements Engineering, Software Architecture, Electronic Business Systems and Conceptual Modeling.

Dr. Cooper is an Assistant Professor in Computer Science at the University of Texas at Dallas. She holds a B.A.Sc., M.A.Sc. and a PhD, all in Electrical and Computer Engineering from The University of British Columbia, Canada. Her research interests include the use of COTS components, metrics, empirical analysis, and formal methods in requirements engineering and software architecture. Dr. Cooper also has industrial experience in the requirements specification and architectural design of complex, large-scale, software intensive systems. Before joining UTD, she worked as a senior systems engineer on Motorola’s GPRS core network project developing system requirements and extending the product architecture.

Ms. Anna Yi is currently a PhD student in the Department of Computer Science at the University of Texas at Dallas. She received a BA degree from the University of Texas at Austin. Her experience includes taking a webmaster position for a professional organization and participating in a distance learning course development. Her research interests include model-driven dynamic GUI development, business and design patterns, and nonfunctional requirements.