Static and Dynamic Structure in Design Patterns

Eric Eide eeide@cs.utah.edu
Alastair Reid reid@cs.utah.edu
John Regehr regehr@cs.utah.edu
Jay Lepreau lepreau@cs.utah.edu

University of Utah, School of Computing
http://www.cs.utah.edu/flux/

Technical Report UUCS–01–014
November 1, 2001

ABSTRACT

Design patterns are a valuable mechanism for emphasizing structure, capturing design expertise, and facilitating restructuring of software systems. Patterns are typically applied in the context of an object-oriented language and are implemented so that the pattern participants correspond to object instances that are created and connected at run-time. This paper describes a complementary realization of design patterns, in which the pattern participants are statically instantiated and connected components.

Our approach separates the static parts of the software design from the dynamic parts of the system behavior. This separation makes the software design more amenable to analysis, enabling more effective and domain specific detection of system design errors, prediction of run-time behavior, and more effective optimization. This technique is applicable to imperative, functional, and object-oriented languages: we have extended C, Scheme, and Java with our component model. In this paper, we illustrate this approach in the context of the OSKit, a collection of operating system components written in C.

1. INTRODUCTION

Design patterns allow people to understand computer software in terms of stylized relationships between program entities: a pattern identifies the roles of the participating entities, the responsibilities of each participant, and the reasons for the connections between them. Patterns are valuable during the initial development of a system because they help software architects outline and plan the static and dynamic structure of software before the structure is implemented. Documented patterns are useful for subsequent system maintenance and evolution because they help maintainers understand the software implementation in terms of well-understood, abstract structuring concepts and goals.

The conventional approach to realizing patterns [12] primarily uses classes and objects to implement participants and uses inheritance and object references to implement relationships between participants. The parts of patterns defined using classes and inheritance are static and therefore easier to understand and analyze. However, they are less flexible because their role in patterns and in the whole system is hardwired into their implementation. In contrast, parts of patterns that are defined using objects and references are more dynamic, and therefore more flexible but harder to understand and analyze.

This paper describes a complementary approach to realizing patterns based on separating the static parts of a pattern from the dynamic parts. The static participants and relationships in a pattern are realized by component instances and component interconnections that are set at compile- or link-time, while the dynamic participants continue to be realized by objects and object references. Expressing static pattern relationships as component interconnections provides more flexibility than the conventional approach while retaining much of the ease of understanding and analysis.

To illustrate the tradeoffs between these approaches, consider writing a network stack consisting of a TCP layer, an IP layer, an Ethernet layer, etc. The usual implementation strategy, used in mainstream operating systems, is for the implementation of each layer to directly refer to the layer above and below it except in cases where the demand for diversity is well-understood (e.g., to support different network interface cards). This approach commits to a particular network stack when the layers are being written, making it hard to change decisions later (e.g., adding low-level packet filtering in order to drop denial-of-service packets as early as possible).

An alternative implementation strategy is to implement the layers using the *Decorator*[1] pattern with objects: each layer is implemented by an object that invokes methods in objects directly above and below it. The objects implementing each layer provide exactly the same interface (e.g., methods for making and breaking connections, and for sending and receiving packets on connections) allowing the designer to build a large variety of network stacks. In fact, network stacks can be reconfigured at run-time, but that is more flexibility than most users require.

Our design and implementation approach offers a middle ground. Having identified the decorator pattern and having decided that the network stack may need to be reconfigured, but not at run-time, each decorator would be implemented as a component that imports an interface for sending and receiving packets and exports the same interface. The choice of network stack is then statically expressed by connecting different sets of components together. The basis of our approach is to permit system configuration and realization of design patterns at *compile-* and *link-time* (i.e., before software is deployed) rather than at *init-* and *run-time* (i.e., after it is deployed).

By matching the expected need for reconfiguration against the degree of abstraction, we achieve the following. (1) We are able to build a range of different network stacks meeting both our current and anticipated needs. (2) Network stacks are configured using a separate language that hides the implementation details of each component. This makes it possible for the system to be reconfig-

This research was supported by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory, under agreement number F33615–00–C–1696. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

[1]Unless otherwise noted, the names of specific patterns refer to those presented in Gamma et al.'s *Design Patterns* catalog [12].