# Increasing Reusability through Architectural Design

Kim Harris, Software Reuse Program Manager
Hewlett Packard Company, Corporate Engineering
1801 Page Mill Rd., MS 18DG, Palo Alto, CA 94304
Telephone 415-857-7771, FAX 415-857-2631

email: kimh@hpcea.ce.hp.com

**Abstract**

The design of good, reusable architectures is essential to effective software reuse among applications in a product family. The goals of an architecture improvement process and good architectures are presented. An example of a reusable architectural design is described. Both the process and artifacts of the architecture are included.

**Keywords:** domain analysis, reusable architectures, object-oriented analysis, object-oriented design

## 1 The Hewlett Packard Software Reuse Program

The objective of the Software Reuse program at Hewlett Packard is to institutionalize software reuse in selected business areas within the company. The program has a small staff of people at a central Corporate Engineering organization, and it works with several pilot projects in operating divisions. We are emphasizing reuse across a vertical span of domains from the application domain through low-level implementation domains. We are also focusing on external library, black box reuse of complete workproducts which have been explicitly designed for reuse. The design of software architectures for specific application domains and product families is key to a successful reuse program in a large company like HP.

## 2 Software Architecture Objectives

To improve the state of architecture designs, we need:

- quality objectives for a software architecture
- metrics and review processes for evaluating architectures

- graphical notation systems for documenting and communicating architectures
- processes for designing architectures.

Some quality objectives of good software architectures include:

- complete functional coverage of application domain(s)
- consistent treatment of functions within the architecture
- flexibility to adapt to different products in an application domain
- clean separation of responsibilities between components to ensure that side effects are minimized and documented, protocols are functionally focused, and the coupling between modules is loose and cohesion is strong
- loose constraints that do not assume or preclude lower domain decisions such as: concurrency, data structures and algorithms, i.e., allows such decisions to be made, evaluated and changed without changing the architecture
- measurable and predictable performance, effort and generality.

There are several approaches to achieving these objectives. One is a top-down process where architectural principles are derived from the objectives. Another is a bottom-up approach which captures, classifies and catalogs architectural styles, cliches and notations and then synthesizes common abstract principles with reproduce the cataloged artifacts. Both approaches are important.

# 3 An OO Architecture for Instrument Measurement and Control

An example of an interesting software architecture comes from the ATUX project recently designed at HP [Harr 91]. This paper will describe both the key elements of the architecture and the process used to design it.

First, the project team performed a domain analysis on the applications developed by the Instrumentation Department of the Optoelectronics Division at Hewlett Packard Co. The department builds Test and Measurement (T&M) systems that test optoelectronic products manufactured in HP factories. A scope of 80

The domain analysis consisted of analyzing three examples in the application family: the simplest (a single LED), a representative middle case (a multicharacter display assembly), and the most complex case (an LED replacement for a laser engine in a "laser printer"). The engineers in the project team had only partial domain knowledge. Additional domain experts were interviewed to complete the domain analysis.

An example of a product in the middle category is described in Figure 1.

Entity Relationship (ER) diagrams were used as domain models. An object-oriented analysis and design process was followed which was adapted from the OOA process by Mellor and Shlaer

A 4 digit LED display assembly. Each digit has 7 LED segments.

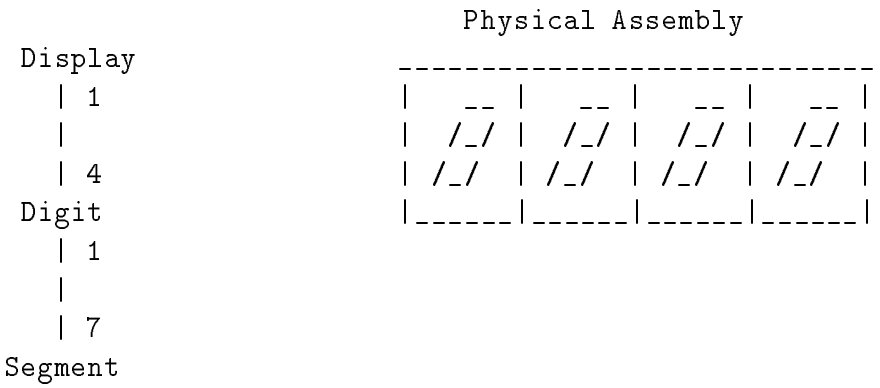


Figure 1

[Shla 88]. In the ER diagrams, the entities are objects (instances) and most of the relations are "uses-a." The main goal of the modeling was to organize solution-space functions into layers in order to facilitate interoperability of modules in each layer and the reuse of components. See Figures 2 and 3 for the domain models for the simplest and most complex cases.

# 4 Responsibility-based layers

From the domain models, class hierarchies for the application domain classes were identified. Each layer had a different purpose (functionality), temporal duties, and persistence of its data. These properties are shown in Figure 4. The abbreviation UUT stands for "Unit Under Test" (i.e., the device on which measurements are being made).

The lowest-level layer contains the instrument drivers. There is a 1:1 correspondence between drivers and physical instruments. The next higher layer is the result of a design abstraction: the virtual instrument layer. It allows application domain measurements to be made by combinations of more primitive hardware instruments. The UUT Topology layer is actually a variable number of sub-layers which have the responsibility of agglomerating measured data. The classes in these layers mirrored the topology of the products being measured. Refer back to Figure 1 for an example. The top layer(s) provided management of the measurement sequencing and communication to other tasks. The interfaces between layers were defined as architecture standards for this application, and they were implemented using Abstract Base Classes (ABC's). An example of this class design is shown in Figure 5.

Following the design of the application domain classes, lower domain classes (such as service and computer science classes) were added to the design. They were mapped onto a client-server architecture containing three nodes as shown in Figure 6.

| Layer | Purpose | Data persistence |
|---|---|---|
| Test management | Test sequence<br>Pass/fail decision | Whole UUT<br>Fetch/store data base |
| UUT Topology | Electrical Whole UUT connectivity | |
| | Hierarchical composition | Local to component of UUT |
| | Geometry, location | |
| | Comparing limits | |
| Virtual Instrument | Set measurement conditions | Each measurement |
| | Read results | |
| | Timing: trigger, intervals | |
| | Instrument topology | |
| | Calibration: engineering units | Calibration: whole session |
| Instrument Drivers | Instrument modes, limits, controli | Each measurement by instrument |
| | Communication with each instrument | |
| | Error detection/ recovery | |
| | Calibration: engineering units | Calibration: whole session |
| Hardware Instruments | | |

Figure 4

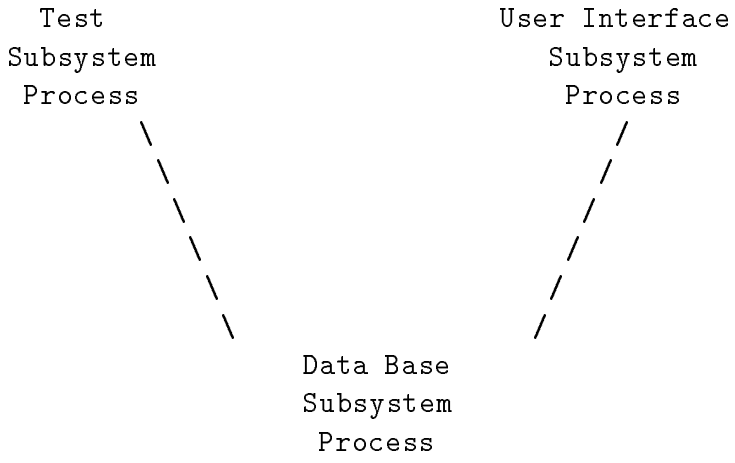


Figure 6

# 5 Parasite Classes

Another architectural technique used was the design and application of parasite classes. Parasite classes were used to perform functions such as persistence, real-time control and tracing and debugging. Parasite classes have the same interface and protocol as the main application classes they can be attached to. They were derived from the ABC's that defined the layer interfaces. They allowed optional functionality to be included without changing the application classes. For example, to add Unix real-time control to an application class, the appropriate parasite class was dynamically linked in front. Any messages sent to the application class were first received by the parasite. It performed its function (i.e., change the real-time priorities), then passed the messages along to the application class.

To facilitate rapid configuration of all classes including the optional parasites, a glue language utility was created that controlled the construction of all instances and initialized attributes and pointers to complete a running system.

# 6 Conclusions

The first application product developed met expectations, and most of the architectural goals were archieved. The interfaces between most layers were fixed for all products in the application family. However, the team was unable to define a single interface between the lowest two layers because of performance requirements. The reusability of the architecture has not been verified because the software platform has not yet been used to develop multiple application products.

# References

[Harr 91] Harris, K.R., Using Object-Oriented Methods to Develop Reusable Software for Test &

Measurement Systems: a Case Study, Proceedings of the First International Workshop on Software Reusability, SWT Memo # 57, Universitat Dortmund, June 1991

[Shla 88] Shlaer, Sally and Mellor, Stephen, "Object-Oriented Systems Analysis, Modeling the World in Data", Yourdon Press, 1988.

# 7 About the Author

Kim Harris is the manager of Hewlett-Packard's corporate-wide software reuse program. He has over 20 years experience in the areas of computer software, hardware, architecture, training and management. Mr. Harris has worked with microcomputers, minicomputers and supercomputers. He has developed scientific applications, real-time applications, operating systems, compilers and graphics software. He has published over 20 papers in the technical and popular press. He received an M.S. degree from Purdue University and a B.S. degree in Physics from Louisiana State University.

His current interest and research areas include: software engineering, process, architecture, metrics, reuse, management, education, object oriented programming, and domain analysis.