Recursive Design of an Application-Independent Architecture

Sally Shlaer
and
Stephen J. Mellor

Project Technology, Inc.
10940 Bigge Street
San Leandro, CA 94577-1123
tel: 510 567-0255
fax: 510 567-0250
http://www.projtech.com

As published in *IEEE Software*, January 1997.

Recursive Design of an Application-Independent Architecture

Sally Shlaer
Stephen J. Mellor
Project Technology, Inc.
http://www.projtech.com

The authors development method focuses on a systematic process using precise specification of system components to create application–independent architectures with a high degree of automation–achieved through large-scale code generation-and broad applicability, focusing on real-time systems.

At the turn of the nineteenth century, the gun manufacturing industry was having difficulty supplying product sufficient to the demand. Guns were then being built by forging parts to relatively rough tolerances; hence armies of fitters were required to file down the parts until they fit together snugly. Many people searched for ways to improve the situation, and they found several solutions that each yielded incremental improvements in fitters' performance.

Eli Whitney made an order-of-magnitude change by rejecting the assumption that increasing the effectiveness of the fitters was the problem. He chose instead to forge the parts to much tighter tolerances. This recast the problem from the time-consuming job of fitting to the simpler job of assembly. Whitney did not make an incremental change; he moved the industry from handcrafting toward automation by looking at the problem of gun manufacturing in a wholly different way.

With our Recursive Design method, we seek to do for software development what Whitney did for gun manufacturing. Recursive Design does not attempt to increase the effectiveness of the programmer as we define this role today. Instead, it calls for far greater precision in the specification of all system components and relies on automation to produce and assemble these components into the final system.

We propose, then, not an incremental change but a wholly different alignment of the roles and relationships between object methods, patterns, and architectures. This requires that we re-examine some widely held assumptions.

Current View

Although practitioners within our industry hold diverse views, we believe that the following five widely held assumptions constitute the core perspectives on object-oriented methods, patterns, and architectures.

1. Analysis treats only the application—the subject matter of interest to the system's end user, such as air traffic control, a telephone switch, an automated warehousing application, and the like.
2. The analysis must be couched in terms of the conceptual entities of the design. This assumption appears in much of the work on domain-specific software architectures,[1-3] as well as in many object-oriented analysis and design methods.[4,5] Hence a developer

employing an object-oriented analysis method is necessarily constrained or biased toward an object-oriented design.

3. Despite the lack of a universally accepted definition for architecture, most people would agree that an architecture provides a view of the entire system. Because this view emphasizes the major elements of the system, many details are necessarily omitted.
4. Patterns are generally thought of as small units, consisting of a few objects with well-worked-out interactions.
5. Programmers expect to select, adapt, and modify patterns according to taste and experience. Patterns are seen as useful but advisory in nature: Use of patterns in general or in particular is not required.

An Alternative View

Our Recursive Design method is founded on a fundamentally different approach to this topic that makes five very different assumptions:

1. An analysis can be performed on any domain, not just the subject matter of interest to the system's end user.
2. Use of an OOA method does not imply anything about the fundamental design of a system.
3. Because we treat the architecture domain just like any other domain, it can be modeled in complete detail using an object-oriented analysis method.
4. The OOA models of the architecture provide a comprehensive set of large-scale, interlocking patterns.
5. Because all the code other than purchased packages is automatically generated, use of the patterns is absolutely required.

Of course, these assumptions depend on the definitions of several key words.

Domains and analysis. In building virtually any software system, developers must deal with several different subject matters: the application itself as well as, typically, the user interface; various purchased packages (such as a database or an inventory control package); and—in real-time systems—an alarm service and the sensors and actuators that comprise the interface to the external world. We call each such subject matter a domain.[6] A *domain* is a separate real, hypothetical, or abstract world inhabited by a distinct set of conceptual entities that behave according to rules and policies characteristic of the domain. An *analysis* consists of work products that identify the conceptual entities of a single domain and explain, in detail, the relationships and interactions between these entities.

We assert that an object-oriented method is well suited to building analysis work products because its central components represent the conceptual entities in the problem. Because our intention is to generate all of the code for the system, we require that the OOA method employed be a "complete detail" method. In particular

- The formalism underlying the method must specify the conceptual entities of the method and the relationships between these entities. This is typically done by producing a metamodel of the method.[7,8]

- Any processing elements shown in the analysis work products must be fully defined within the semantics of the domain under analysis.
- The dynamic aspect of the formalism—the virtual machine that describes its operation—must be well specified. This is required so that a set of analysis work products can specify all legal orders of execution of the processing elements. However, according to

our second assumption, we can use OOA to analyze an application domain and then build a design that does not rely on concepts such as encapsulation, inheritance, or polymorphism.

Architecture. In our approach, an *application-independent architecture* is a separate domain that defines rules, mechanisms, and policies that apply to all the software elements of the entire system, regardless of the domain from which they are derived. An architecture deals with the following topics in complete detail:

- policies and mechanisms for organizing and accessing data;
- control strategies, including synchronization, concurrency, interactions in a distributed system, and the like;
- structural units—the tasking strategy for the system, strategies for allocating tasks to processors and processing units to tasks, standard structures used within a task; and
- time—mechanisms for keeping time and for creating delayed transfers of control.

Equally important is what the architecture does not itself define: While it does provide for allocation of elements from the application and other domains to the structural units and data structures, it does not specify the allocation to be used. It is this property that gives rise to the term "application-independent." Hence, a pipes-and-filters architecture would not state the application content of the filters, a thread-based architecture would provide mechanisms for causing threads to be executed but would make no statement regarding the application content of the threads, and an object-oriented architecture would employ encapsulation and inheritance over unstated application entities.

According to our third assumption, we can use an OOA method to model any style of architecture, whether object-oriented or not. So, in a thread-based architecture, we may see objects such as Chore List, Chore, and Chore Input Parameter. These objects support a concept of a thread as a list of chores to be executed, each of which takes input parameters. These objects do not model an architecture based on classes, encapsulation, or inheritance.

Patterns. Our approach to patterns operates from our last two assumptions. According to our fourth assumption, the OOA models of the architecture provide a comprehensive set of large-scale, interlocking patterns, which can be rendered as archetypes. This is conceptually equivalent to defining macros for each element of the patterns. We then extract elements from the repository containing the application (and other domain) OOA models. Using these elements, we expand the archetypes to generate the code for the system.

Our fifth assumption asserts that, because all the code other than purchased packages is automatically generated, use of the patterns (archetypes) is absolutely required. If a pattern must be modified, the change is made and all affected system components are regenerated.

In summary, you can model an application-independent architecture using an OOA method. From these models, you can derive and express a set of large-scale patterns in a manner that permits completely automatic code generation.

A Systematic Process

The construction of an architecture is today an expert process. Skilled architecture builders, also called system designers, generally work to familiarize themselves with certain kinds of requirements imposed by the application and other domains, matching these requirements against a mental repertoire of architectural patterns and schemes that they have seen or built before. However, most architects cannot explain exactly what aspects of the application and other domains they find especially important or how they come to decisions about what kind of an architecture is required for a particular problem.

Although we do not understand this expert process in detail, we can partition it into key activities. Such partitioning lets us identify any well-understood activities and develop automation to carry them out, either entirely or in part, and lets us focus on each expert step to improve our understanding of the expert component. Our process, which we call Recursive Design, comprises seven steps:

1. Characterize the system.
2. Define conceptual entities.
3. Define theory of operation.
4. Collect instance data.
5. Populate the architecture.
6. Build archetypes.
7. Generate code.

We examine each step in turn.

Characterize the system. This step elicits those characteristics of the system that should shape the architectural design. There are many issues you may need to consider, and we have found that a questionnaire, as described in the “System Characterization Survey” box helps to focus the activity and ensure that all issues are addressed. This questionnaire emphasizes fundamental design considerations regarding size, memory usage, data access time, throughput, identification of critical threads, response time, and the like. Although this information comes from the application and other domains, it is stated in the semantics of system design and not in the vocabulary of the application.

The system characterization is developed as a report, often containing numerous tables, hardware configuration drawings, and similar exhibits. You can do this work after the analysis is complete for all domains, or after the application domain is reasonably well understood, or even before any analysis is done at all. Clearly, the more analysis done on each domain, the more precise the characterization. However, when it is advantageous to produce the architecture as early as possible, you can often complete the questionnaire surprisingly well based only on your knowledge of related, previously developed systems.

The process of collecting and reporting characterization information helps you assimilate and prioritize the issues and understand the interactions between them. What is not well understood is how expert architects identify which pieces of information are significant for development of the architecture and which are not. In general terms, the architects work with the characterization information to achieve Christopher Alexander's concept of "fit": a mutual acceptability between context—that part of the world over which we have no control (the system characterization, including purchased or legacy components)—and form—that part of the world over which we do have control (the architecture). In his book, *Notes on the Synthesis of Form,*[9] Alexander provides a metaphor for achieving fit:

> *It is common practice in engineering, if we wish to make a metal face perfectly smooth and level, to fit it against the surface of a metal block, which is level within finer limits than we are aiming at, by inking the surface of this standard block and rubbing our metal face against the inked surface. If our metal face is not quite level, ink marks appear on it at those points that are higher than the rest. We grind away at these high spots.…*

Hence, by the time the system characterization has been completed, you have mentally ground away at the high spots and so developed ideas of the architecture type required.

Define conceptual entities. This step defines and describes precisely the conceptual entities of the architecture and the relationships that must hold among them. You do this by building an object information model.[6] We use the Timepoint architecture as a running example throughout this article. Derived from an application-independent architecture built at Lawrence Berkeley Laboratory in 1977, the project used several key ideas, including separation of domains and a system construction engine, that have since been incorporated into our Recursive Design method. The original implementation was developed in assembly language and Fortran on a PDP-11/45 running RSX-11M. It has subsequently been ported to more modern operating systems and processors throughout its 15-year production life. It has even been ported to a new accelerator with a completely different timing structure.

Which objects appear on an OIM depend on the concepts inherent in the architecture under construction. Hence, the Timepoint architecture has objects Kernel Task and Chore List, but not Processor or Class—although these objects would appear in a multiprocessing, object-oriented architecture. Timepoint's architects selected the Kernel Task and Chore List objects based on system characterization information and their own architectural experience. They had to make conscious choices here: these objects are not derived directly from the application analysis. They come about because the architect defines them. The "Origin of the Timepoint Architecture" box explains the architects' thinking as they defined the objects of this particular architecture.

Figure 1 shows a Shlaer-Mellor Object Information Model for the Timepoint architecture. In the model, a box represents an object, a typical but unspecified instance abstracted from a set of things in the domain that all have the same characteristics and conform to the same set of rules and policies. Each object is described by attributes; each attribute is an abstraction of a single characteristic. Attributes annotated with an asterisk are required to identify a specific instance of the object. Each line represents an abstraction of a systematic pattern of association, called a relationship, that exists between things that were abstracted as objects. Each relationship has a multiplicity value, which indicates how many instances participate (one arrowhead for one instance, two arrowheads for many instances), and a conditionality value indicating whether the instance is required to participate (a "C" indicates that it is conditional, so it is not required to participate). Finally, the structure marked "is a" shows a subtype/supertype relationship, in which the set of all instances of the supertype (Chore List) may be completely partitioned into the subsets Unpacking Chore List, Conversion Chore List, and so on. The object and relationship numbers and letter codes have no semantic content, save the construct Rx = Ry + Rz, which indicates that an instance participating in the relationship R*x* is constrained to refer to the same instance as that given by following the chain R*y* and R*z*.

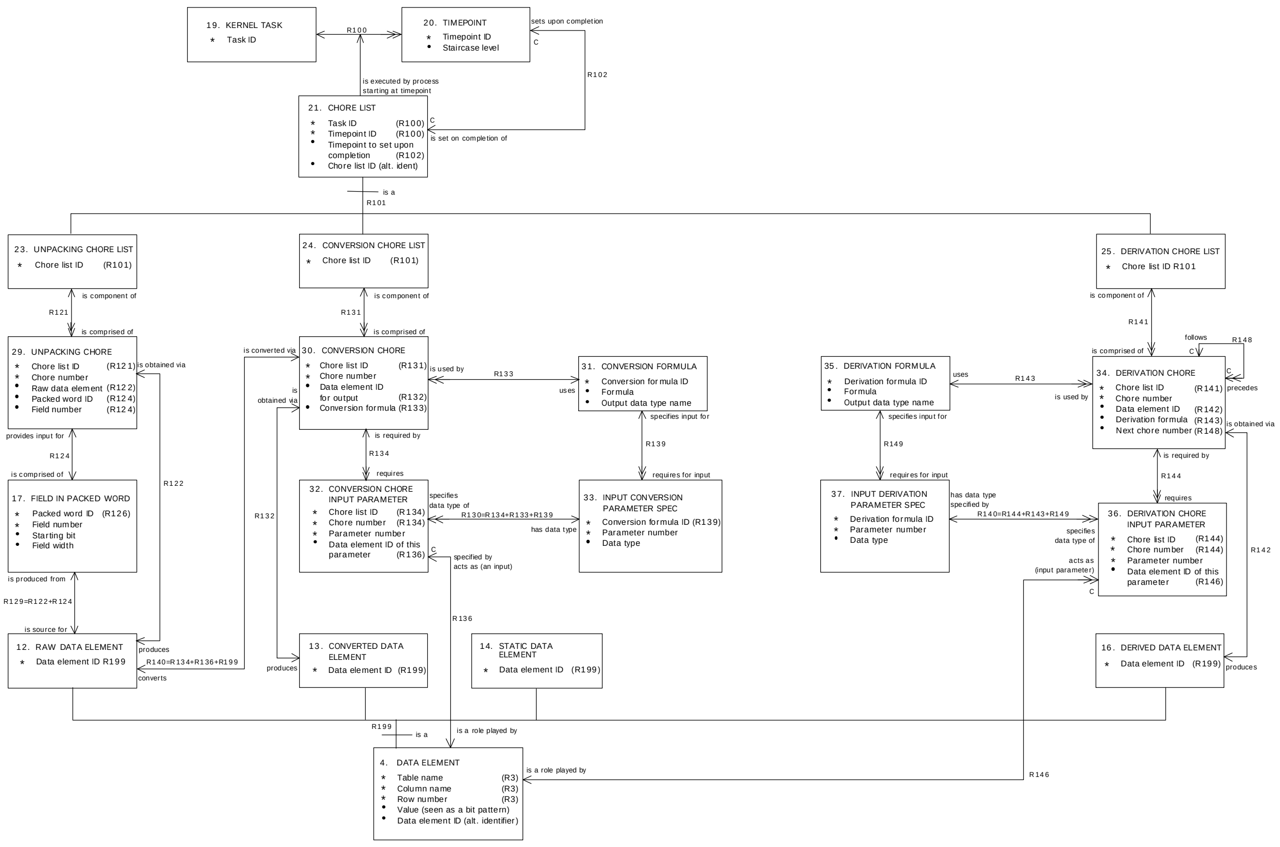

Figure 1: Shlaer-Mellor Object Information Model for the Timepoint Architecture.

Define theory of operation. This step describes precisely how the system works as a whole. We have found that an informal but comprehensive theory-of-operation document or technical note[10] works well to develop the appropriate concepts. This document should describe the threads of control that run through the architecture domain, covering all modes in which the system operates, such as

- normal system operation;
- cold-start and initialization procedures;
- warm-start, restart, and failover operation, as required for the delivered system; and shutdown.

You can then render the architecture operation as a set of state models, as prescribed by the OOA method. These models describe the dynamic operation of the architecture in complete detail.

The combination of the OIM and the state models provides a precise description of the architecture, which ensures that all the components fit together into an integrated whole. The end result is a set of interlocking patterns on a grand scale that, when taken together, implement the responsibilities of the architecture in a manner consistent with the information gathered during the system characterization step.

Collect instance data. Every set of analysis models presupposes a particular analysis time frame or time scope. For example, in a manufacturing application, you normally assume that the lathes, milling machines, and the like simply exist: they are not created by the manufacturing process being analyzed. These are called pre-existing instances.

This step simply collects the data that describes the pre-existing instances. The information will be used by the system construction engine, developed in our process's final step, to generate populated data structures or initialization code as prescribed by the architecture.

The size of this step varies. An electrical power distribution or inventory control problem may require that thousands of pieces of data be collected. For the architecture domain, typically only a few items are needed, such as processor names, port numbers, and so on. In any case, our standard procedure is to use commercially available database software to record the instance data, deriving the schema for each domain from the OIM for that domain. Each such database is known as that domain's *instance database*.

Populate the architecture. This step completes the instance database for the architecture domain. To do this, you use a "populator" program to extract elements from the repository containing the application models, then use this information to create additional instances in the architecture's instance database. For example, for the Timepoint architecture, the populator would create

- an instance of Conversion Formula for every transformation process in the application, and
- instance of Input Conversion Parameter Spec for every type of every input to a transformation in the application.

A populator is also used to transfer information from the instance databases of all other domains to the instance database of the architecture.

At the completion of this step, the architecture's instance database contains all the information about the system to be built: information from the application, such as the conversion formulas, and information specific to this particular usage of the architecture, such as the names of the Kernel Tasks and Time Points.

Build archetypes. For clarity, we present archetype building as a two-domain problem (application and architecture domains only), although the process is easily generalized to include all the domains that make up a system.

An archetype combines text, written in the target programming language, with placeholders used to represent information from the architecture's instance database. We write one or more archetypes—essentially macros—for each object in the architecture.

We have not yet defined a standard archetype language to be used with Recursive Design. The current generation of tools supporting Recursive Design use different archetype languages, all of which are similar in spirit to the description given here:

- An archetype begins with its name followed by a colon.
- Text in the target language appears inside single quotes.
- Optional expressions appear in square brackets.

- An asterisk indicates zero or more repetitions.
- A name appearing in the definition of an archetype is either the name of another archetype or an attribute of the object for which the archetype is being defined.

Consider the following simple archetype for the Input Conversion Parameter Spec object of the Timepoint architecture:

```
InputConversionParameterSpec:
 DataType ‘P’ParameterNumber
```

This archetype, when expanded, generates a string representing a datatype, followed by a parameter name of the form P1, P2, P3, and so on, for an `InputConversionParameterSpec`. Both `DataType` and `ParameterNumber` are attributes of the object Input Conversion Parameter Spec. Each expansion of the archetype causes it to substitute the values of the appropriate attributes and produce, for example, `int P1` on its first expansion, `real P2` on its second, and so on.

Archetypes may be nested in the conventional manner used for macros. For example, the archetype for Conversion Formula is:

```
ConversionFormula:
 OutputDataType
   FunctionName`(‘
   InputConversion
   ParameterSpec
 [ `,’ InputConversion
   ParameterSpec ]* `)’ =
   Formula
```

When this archetype is expanded for an instance of Conversion Formula, the `InputConversion ParameterSpec` archetype will be expanded for all related instances of Input Conversion Parameter Spec. The related instances are found by tracing the relationships (in this case, R139 in Figure 1) on the architecture’s OIM.

Generate code. The purpose of this step is to develop a script—known as the system construction engine—that will generate the code for the system from the analysis models, the archetypes, and the instance databases of all domains. This work is generally assigned to “toolmakers”: computer scientists with expertise in general-purpose tools such as macro processors, databases, lex, yacc, and so on, and in extraction of information from the CASE tool’s repository.

The central component of the system construction engine is the code generator: the component that expands the archetypes to produce the code. For a final example, consider the archetype shown in Figure 2 for Kernel Task. When this archetype is expanded using the architecture instance database shown in Figure 3, we obtain the code shown in Figure 4.

```
KernelTask:
        KernelTask-Prolog                 // Includes data element definitions and other include files
        KernelTask-StartTimePoint         // Opening code for the task in the target language
                                          // Expand archetype for TimePoint (which gets name of TimePoint & outputs staircase)

                                          // Show the Conversion Chores only, for brevity

        ConversionFormula*                // Produces definitions for the Formulae; refers to Parameter Specs
                                          // This produces all the Conversion Formulas in all the ChoreLists in the Kernel Task

        KernelTask-SwitchtoChoreList      // generates logic that tests the Timepoint to determine ChoreList

         'Switch TimePoint into { '       // Real code to produce a 'switch' statement
        (  'Case ' TimePointID ':'        // Now produce each case--note the parentheses
        ChoreList                         // Note that each 'case' gets only one ChoreList, but the expansion is repeated.
                                          // KernelTask traverses to the appropriate ChoreList, then to the Chores
                                          // The Chore produces the calls and, in turn, refers to the Input Parameters
         'Endcase' ) *                    // Repeats the contents of the parentheses--thus making each case
         ' };'                            // Finish the switch


        KernelTask-Epilog                 // Set next TimePoint, if any, and release control
```

Figure 2: Code archetype for Kernel Task.

To complete the system construction engine, the toolmakers prepare a script that sequences the various system-building activities in the following order:

1. population of the architecture’s instance database
2. code generation
3. compilation
4. linking

Finally, we execute the system construction engine to generate the system code.

Kernel Task
Task ID
Task2
Task3

TimePoint	
TimePointID	*Staircase Level*
Ta1	A
Ta2	B
Tb1	C
Tb2	D

ChoreList			
Task ID	*TimePointID*	*TP to set*	*ChoreListID*
Task2	Ta1	Ta2	CL1
Task3	Ta2	None	CL2
Task2	Tb1	Tb2	CL3
Task3	Tb2	None	CL4

Conversion Chore List
Chore List ID
CL1
CL3

Derivation Chore List
Chore List ID
CL2
CL4

Conversion Chore			
Chore List ID	*Chore Number*	*DE for O/P*	*Conv Formula*
CL1	CN1	CDE1	CF1
CL1	CN2	CDE2	CF1
CL1	CN3	CDE3	CF3
CL3	CN1	CDE4	CF4
CL3	CN2	CDE5	CF2

Conversion Formula		
Conv Form ID	*Logic*	*O/P Data Type*
CF1	p1 + p2	int
CF2	(p1 ** 3) * PI	real
CF3	p1 - p2 * p3	real
CF4	p1 ** 2 + 3	int

Conversion Chore Input Parameter			
Chore List ID	*Chore Number*	*Parm Num*	*DE ID for Parm*
CL1	CN1	1	DE1
CL1	CN1	2	DE2
CL1	CN2	1	DE3
CL1	CN2	2	DE4
CL1	CN3	1	DE5
CL1	CN3	2	DE6
CL1	CN3	3	DE7
CL3	CN1	1	DE8
CL3	CN2	1	DE9

Input Conversion Parameter Spec		
Conv Form ID	*Parm Num*	*Data Type*
CF1	1	int
CF1	2	int
CF2	1	real
CF3	1	real
CF3	2	real
CF3	3	int
CF4	1	int

Derivation Chore			
Chore List ID	*Chore Number*	*DE for O/P*	*Deriv Formula*
CL2	CN1	DDE1	DF3
CL2	CN2	DDE2	DF2
CL2	CN3	DDE3	DF1

Derivation Formula		
Deriv Form ID	*Logic*	*O/P Data Type*
DF1	(p1 ** 2) / PI	real
DF2	p1 / p2	real
DF3	p1 ** 2	int

Derivation Chore Input Parameter			
Chore List ID	*Chore Number*	*Parm Num*	*CDE ID for Parm*
CL2	CN1	1	CDE4
CL2	CN2	1	CDE2
CL2	CN2	2	CDE3
CL2	CN3	1	CDE5

Input Derivation Parameter Spec		
Deriv Form ID	*Parm Num*	*Data Type*
DF1	1	real
DF2	1	int
DF2	2	real
DF3	1	int

Figure 3: Instance database for the Kernel Task code archetype.

Code Generated for Task 2 From the Instance Database

```
(expansion of KernelTask-Prolog)               // Includes data element definitions and other include files
(expansion of KernelTask-StartTimePoint)       // Opening code for the task in the target language
                              // Expand archetype for TimePoint (which gets name of TimePoint & outputs staircase)

int CF1( int p1, int p2) = p1 + p2             // These are the definitions for all the Conversion Formulae related to
real CF2(  real p1 ) = ( p1 ** 3 )  * PI       //  the current Kernel Task through Chore List (there are several),
real CF3( real p1, real p2, int p3 ) = p1 - p2 * p3   // to the Conversion Chore List, to many  Conversion Chores to a
int CF4(  int p1 ) = p1 ** 2  + 3              // smaller number of Conversion Formulae

Switch TimePoint into {                        // Just code
Case Ta1:                                      // The first case statement
        CDE1 = CF1( int DE1, int DE2 )         // All the chores in the Chore List
        CDE2 = CF1( int DE3, int DE4 )
        CDE3 = CF3( real DE5, real DE6, int DE7 )

Case Tb2:                                      // The second case statement
        CDE4 = CF4( int DE8 )                  // All the chores in the second Chore List
        CDE5 = CF2( real DE9 )
Endcase };


(expansion of KernelTask-Epilog)               // Set next TimePoint, if any, and release control
```

Code Generated for Task 3 From the Instance Database

```
(expansion of KernelTask-Prolog)               // Includes data element definitions and other include files
(expansion of KernelTask-StartTimePoint)       // Opening code for the task in the target language
                              // Expand archetype for TimePoint (which gets name of TimePoint & outputs staircase)

real DF1(  real p1 ) = ( p1 ** 2 )  / PI       // These are the definitions for all the Derivation Formulae related to
real DF2( int p1, real p2 ) = p1 / p2          //  the current Kernel Task through Chore List (there are several),
int DF3(  int p1 ) = p1 ** 2                   // to the Derivation Chore List, to many  Derivation Chores to a
                                               // smaller number of Derivation Formulae

Switch TimePoint into {
Case Tb1:                                      // The first case statement
        DDE1 = DF3( real DE5 )                 // All the chores in the Chore List
        DDE2 = DF2( int DDE2, real DDE3 )
        DDE3 = DF1( real DDE5 )

Case Tb2:                                      // The second case statement
                                               // There are no chores in Chore List 4
Endcase };


(expansion of KernelTask-Epilog)               // Set next TimePoint, if any, and release control
```

Figure 4: Code generated for Tasks 2 and 3 from the instance database shown in Figure 3.

Implications

We have proposed that developers construct an architecture-independent application analysis expressed in complete detail in an OOA method, an application-independent architecture also expressed in complete detail in both conceptual and archetype terms, and a system-construction engine that lets us generate and regenerate the system at will.

New approaches. Recursive Design lets us approach a software development strategy in totally new ways.

- We can now build architectures without knowing anything about the semantics of the application. This enables production of commercially available architectures. Hence a project may soon approach the architecture problem as a build-or-buy decision—with potentially highly favorable economic and schedule consequences.
- Companies that build many products of a similar type—say intelligent laboratory instruments or medical electronics—often find that a single architecture (with possible minor variations) can support a whole range of products and applications that share the same characteristics. The use of an application-independent architecture lets the organization amortize the architecture investment across multiple projects by reusing the entire domain, not merely a set of classes.
- Other organizations concentrate on a product line that spans a wide range of performance and platform requirements. Here, a single set of application analysis models can be used to build multiple implementations using different architectures.
- A starter architecture supports the early delivery of application prototypes. Although the starter architecture may be lacking in response time, multiprocessor support, or the application size it can support, it is sufficient to generate and demonstrate the application as soon as each portion of the analysis is complete. Development of an architecture that meets performance needs can then proceed concurrently.
- The nature of long-term maintenance is entirely changed. Because this approach delivers application and architecture models, together with a construction engine, we do not maintain the production code. Instead, when the requirements change, we adjust the models and reconstruct the system. Consequently, maintenance is never carried out on the production code, but only on the models and archetypes that generate the code. To modify the production code would be akin to modifying the code generated by a compiler.

Focused developer roles. Recursive Design shifts the traditional skill profiles required on a software development project. The roles developers must play are now easily identified with the method's steps.

- *Analysts* are particularly important on any project that takes the approach presented here. Because all the application code will be generated directly from the analysis models, these models must be correct and complete.
- *Architects* take responsibility for the system design concept and the analysis models for the architecture. Because the architecture must appear as if it were produced by a single mind, it is best to staff an architecture team with at most three or four experienced architects.
- *Programmers* develop the archetypes. Far fewer programmers per se are required on a project using Recursive Design than on one using conventional methods, since no application code is written by hand.
- *Toolmakers* are needed on every project. However, in a Recursive Design project, the need is well identified, and the resources needed can therefore be more easily predicted.

Finally, this shift in skill profiles parallels the shift that occurred when Whitney recast gun manufacturing from a low-tolerance fitting problem to a high-tolerance assembly problem. Although a higher skill level was required to build the factory itself, machines then took over the rote work of production.

The systematic process we have outlined contains only one truly expert problem: getting from the system characterization to the architecture's conceptual entities. We see two opportunities for making inroads in this area in the near term.

First, we can expand the characterization step. Although a great deal of information is collected during system characterization, we are by no means confident we are collecting the right information. In particular, experience indicates that a better quantitative understanding of the control threads is significant in making numerous architectural decisions. Hence, we are now looking into models and automation that would provide a practical way to collect this information.

Second, we can build a collection of sample architectures. We believe that such a collection would provide an expanded repertoire of designs for consideration by system architects. Each architecture could be more closely linked to the characterization work by providing quantitative information showing the range of applicability as it is understood today.

Eventually, a comprehensive catalog of this nature could put system architects in a position similar to that enjoyed by electronics engineers. They would have a book that defines all the chips you may purchase, together with their performance characteristics, such as speed, heat dissipation, and so on. We would have a book of application-independent software architectures, together with their performance characteristics on a range of platforms. This would truly produce an order-of-magnitude change in the way we develop software.

References

1. D. Garlan and M. Shaw, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, Englewood Cliffs, N.J., 1996.

2. M. Shaw, "Comparing Architectural Design Styles," *IEEE Software*, Nov. 1995, pp. 27-41.

3. F. Buschmann et al., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley, Chichester, UK, 1996.

4. J. Rumbaugh et al., *Object Modeling and Design*, Prentice Hall, Englewood Cliffs, N.J., 1990.

5. P. Coad and E. Yourdon, *Object-Oriented Analysis*, Prentice Hall, Englewood Cliffs, N.J., 1989.

6. S. Shlaer and S.J. Mellor, *Object Lifecycles: Modeling the World in States*, Prentice Hall, Englewood Cliffs, N.J., 1992.

7. M. Lloyd, *Shlaer-Mellor Method: OOA Metamodel*, Project Technology, Berkeley, Calif., 1996.

8. G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language, version 0.8,* Rational Software Corp., Santa Clara, Calif., 1995.

9. C. Alexander, *Notes on the Synthesis of Form,* Harvard Univ. Press, Cambridge, Mass., 1964.

10. L.J. Starr, *How to Build Shlaer-Mellor Object Models*, Prentice Hall, Englewood Cliffs, N.J., 1996.

Sally Shlaer is director of research at Project Technology, where she works in close collaboration with Stephen J. Mellor on continuing development of the Shlaer-Mellor method and the codification of real-time architectures. Prior to founding Project Technology, Shlaer worked at Los Alamos National Laboratory and Lawrence Berkeley Laboratory, where she gathered over 20 years of experience in operating systems and real-time system design and implementation.

Shlaer received a BS in mathematics from Stanford University. She is a member of the IEEE Computer Society, ACM, and ACM Sigsoft.

Stephen J. Mellor teaches and does research at Project Technology. He began his career at CERN in Switzerland, where he supported accelerator control systems. Subsequently, at Lawrence Berkeley Laboratory, he led a team responsible for providing control systems for several real-time applications.

Mellor received a BA in computer science from the University of Essex in the United Kingdom.

Address questions about this article to Shlaer or Mellor at Project Technology, 10940 Bigge Street, San Leandro, CA 94577; sally@projtech.com or steve@projtech.com.

SYSTEM CHARACTERIZATION SURVEY

1. Domains

The purpose of this section is to understand the technical content of the system.

- How many domains are there in the system?
- Of these domains, how many domains are already built and need no change? How many will need some work either to build a clean interface or to change to the content of the domain?
- How many domains are yet to be built? For each domain to be built: How many active objects, passive objects, and assigner objects do you expect? How many instances of the active and passive objects do you expect for each domain?

2. Application

The purpose of this section is to characterize the system that the architecture will execute so as to make tradeoffs between the size and execution speed in the architecture.

2.1 Application type

- Which (one or more) of these terms best describes the system: batch-oriented; data acquisition; process control; event/control-driven; thread-based; database manipulation; other?
- Is this system independent of existing software or integrated with it?
- Are there any industry or internal standards that this system must comply with?
- What are the requirements for failover or error recovery? How fault tolerant will the system need to be?
- Will the application be required to run continuously for long periods of time? Are online code updates required?

2.2 Processing requirements

- Will the system to be I/O or CPU-bound? Why?
- How much dynamic instance creation is required?
- What is the execution profile of the system? Is it periodic or event driven?
- If the system is periodic, what are the required sampling frequencies?
- If the system is event-driven, what are the burst and average rates for event generation?
- Can the event stream be throttled? Can any events be thrown away? Are there hard deadlines for processing events?
- Must resources be allocated "fairly"? Is it acceptable to starve any component?
- Will the architecture have to execute within specific time periods, such as rate monotonic scheduling, or quantum scheduling? Another scheduling algorithm?
- Is priority-based preemption required? What is the priority associated with? Instances? Event types? What is the longest time an action can execute without any interruption?

2.3 Memory usage

- Will memory be a limited resource?
- How much data will the system need to store and process?
- What is the lifetime of the data?
- Does data persist between system invocations? Is there a requirement to archive runtime data?
- What kind of offline access is required to persistent data? Is it used only for initialization? Does the data have any structure? Are ad hoc queries needed?

2.4 Interface to the outside world

- Is there a user interface? How is it specified?
- Will the system be implemented using different platforms? How will platforms communicate? Shared memory? LAN? WAN?
- Is there a process input/output (PIO) domain on your domain chart?

3. Software environment

These questions help understand the environment for the architecture. The capabilities of the environment may enable more efficient or easily translated architectures.

- What operating system(s) will be used?
- What languages (and compiler versions) will be used?
- What networking protocols will be used?
- What databases will be used?
- What third-party software will be used?

4. Hardware environment

These questions characterize the hardware environment and help determine of performance levels and distribution capability.

- Is the CPU dedicated or will other tasks also be executing on the CPU?
- Is there a requirement for concurrent processing? Will the system operate on one or several CPUs? What forms of interprocessor communication are available?
- Will there be specialized hardware (DSPs, custom processors)? How will this hardware be used? What are the data rates to and from the hardware? What is the mechanism? DMA? Interrupts?

This survey is excerpted from “Architecture Characterization Questionnaire,” a Project Technology internal report by Gregory Rochford, John Wolfe, Kent Mingus, Michael Lee, and Stephen J. Mellor.

ORIGIN OF THE TIMEPOINT ARCHITECTURE

The Timepoint architecture example used throughout this article was modeled on the control software for a therapy machine used to irradiate certain kinds of tumors. The machine consists of a computer control system together with a particle accelerator that provides a beam of energized particles. The computer system controls and monitors the beam once it has left the accelerator proper.

As a result of the system characterization work, the architects identified four issues that were key to determining the architecture:

- External sensors and actuators are interfaced to a small outboard processor connected via a DMA channel to the central computer.
- The operating system used on the central computer limits each task to a small address space.
- The accelerator provides a pulsed beam, with a hardware signal supplied before and after each pulse. These two signals initiate all real-time data acquisition and computation.
- The facility provides a large and ever-increasing inventory of equipment to measure and control irradiations. Only a subset of this equipment is used for any particular irradiation, the selection being dependent on the shape of the volume to be irradiated, the energy of the beam, and similar factors. This is evidenced in the application analysis models, which show a very large number of short control threads, only a few of which are to be activated during any particular irradiation.

From the two hardware-provided signals—T_a and T_b in Figure A—the architects abstracted the concept of a *timepoint*. Whenever a timepoint occurs, a Kernel Task runs, and—upon completion—sets another timepoint. The newly set timepoint can, in turn, cause another kernel task to run.

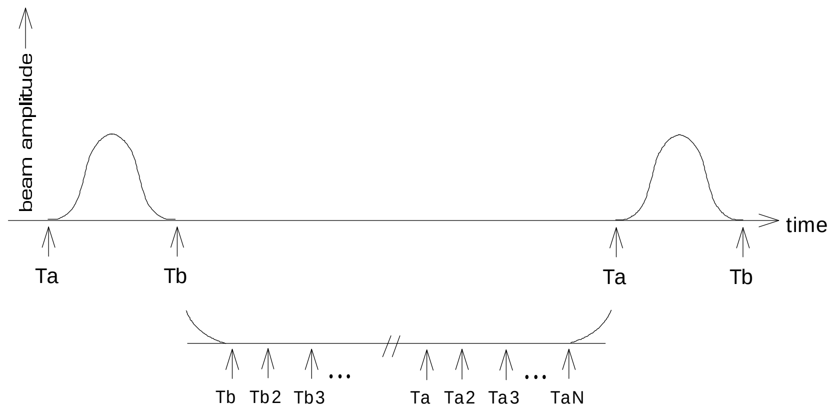

Figure A: System sketch from which architects generated the timepoint concept.

The real-time computation is divided across several kernel tasks to meet the address space restriction. Each kernel task has a specific purpose:

- acquisition of raw data from the outboard processor;
- unpacking raw data where required and converting it to engineering unit data items;
- computation of derived values, many of which are used to verify the stability of the beam and associated equipment;
- checking of data items to see that they fall in the required range; and
- running diagnostics to ensure the integrity of the computer system itself.

With regard to data item checking, any deviation of a data value causes the irradiation to terminate. Data checking accomplishes normal termination of an irradiation (measured dose _ required dose) as well as ensuring that the control and monitoring equipment is operating properly and that the beam is maintaining prescribed characteristics.

To deal with the ever-changing configuration of measuring and control equipment, the architects abstracted the idea of a *chore list*. Chores in this definition are small operations performed by a specific kernel task when initiated by a specific timepoint. For any particular irradiation, developers prepare two sets of chore lists. One set is executed at the end of the pulse to compute dose and beam-related quantities, while the other is executed before the pulse to collect background (electrical noise) measurements used to correct the dose calculation. The chore lists are prepared offline by assembling snippets of the pertinent control threads from the application analysis models.