The Shlaer-Mellor Method

Sally Shlaer
Stephen J. Mellor

Project Technology, Inc.
10940 Bigge Street
San Leandro, California 94577-1123
(510) 567-0255
http://www.projtech.com

1. Introduction

Background. The Shlaer-Mellor Method is a well-defined [1, 2, 3] and disciplined approach to the development of industrial-grade software. It is based on the object-oriented paradigm, and has developed over the past dozen years in the pragmatic environment of real-world projects. These projects have included manufacturing and process control applications, intelligent instruments and peripherals, banking operations, telecommunications, and defense applications.

Characteristics. The hallmark characteristics of the Shlaer-Mellor Method are an integrated set of models that can be executed for verification, and an innovative approach to design that produces a system design through a translation of the analysis models. This approach leverages the initial investment in a formal analysis phase; it provides smooth and predictable transitions between analysis, design, and implementation; and it enhances the reuse characteristics of the system by segregating application and implementation issues.

Framework. The framework for applying the Shlaer-Mellor Method is an initial identification of separate and independent subject matters, called *domains*. Domains can be further partitioned into subsystems to provide coherent and manageable pieces of work. The method prescribes explicit mechanisms for dealing with this layered partitioning of the system. This framework allows the Shlaer-Mellor Method to:

- scale successfully to address both large- and small-sized projects,
- provide structure for project planning,
- be the basis for large-scale reuse of domain-level components.

Process. The general process of applying the Shlaer-Mellor Method is as follows:

- Partition the system into domains.
- Analyze each domain.
- Verify the analysis through execution.
- Specify the translation of the analysis models.
- Build the translation components.
- Translate the models of each domain.

While there is a natural order to these steps, based on inherent dependencies, there is typically significant overlap in the work being done on them. Some iteration also occurs between the steps, though the Shlaer-Mellor Method is careful to limit the scope of iterations in the interest of efficiency. This process most closely resembles a transform-based lifecycle, as described in [4].

The process can also be modified to accommodate project-specific needs for variations such as prototyping and incremental development, though these are not essential elements of the Shlaer-Mellor Method.

2. Process Steps

The Shlaer-Mellor Method provides comprehensive coverage for the analysis, design, and implementation phases of the software development lifecycle. This section briefly describes each of the major steps in the Shlaer-Mellor Method and its contribution to the development lifecycle. It characterizes the activities in each step and introduces the fundamental concepts on which the activities are based.

In order to describe the Shlaer-Mellor process in detail, the process is broken down into the steps listed below. These steps distinguish between the work according to different types of domains, namely, the application domain, the service domains, and the architectural (translation) domain. These types of domains are described in the next section.

1. Partition the system into domains.
2. Analyze the application domain.
3. Confirm the analysis through static verification and dynamic verification (simulation).
4. Extract the requirements for the service domains.
5. Analyze the service domains.
6. Specify the components of the architectural domain.
7. Build the architectural components.
8. Translate the models of each domain using the architectural components.

2.1 Partition into Domains

The system to be built is first divided into distinct, independent subject matters, or *domains*. There are four types of domain that we distinguish for the purpose of explaining the method. They are:

- an *application domain*, that is the subject matter of concern to the end user of the system,
- several more-general domains, called *service domains*, such as a user interface or a sensors-and-actuators domain,
- an *architectural domain*, that is the organization of data, control and algorithm within the system as a whole
- and some *implementation domains*, such as operating system and programming language.

The domains are organized in client-server relationships, so that a domain acting as client can rely on a server domain to provide commonly-needed mechanisms and services.

The results of this initial step are depicted in a *Domain Chart* as shown in Figure 1. The ovals represent domains, and the thick arrows represent client-server relationships with the arrow pointing to the server. These relationships are called *bridges*. The domain chart is supported by textual descriptions for each domain and each bridge.

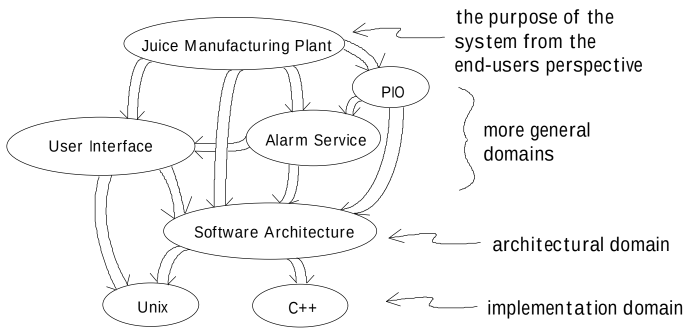

Figure 1: Sample Domain Chart

The domain chart is the first level of partitioning on the system as a whole, because each domain is a separate subject matter that can be analyzed separately from others. However, a domain may be too large to be analyzed as a unit by a team of analysts. In this case, it is necessary to partition a domain into subsystems. Each subsystem must be small enough to be analyzed effectively by a small team of analysts. There is a set of models that show relationships between subsystems within a single domain. The models in this set are the Subsystem Relationship Model, the Subsystem Communication Model and the Subsystem Access Model. These models show the data relationships, event communications and data accesses between subsystems respectively.

There is a separate management structure, the *Project Matrix*, which is used to track work, to track progress, and to provide a structure for naming the elements produced by the project team. The Project Matrix, and its uses, are described in [7]. We do not describe it here because it is not a part of the method itself, but rather the structure for managing the project using the method.

2.2 Analyze Application Domain

The next step is to analyze the application domain using Shlaer-Mellor OOA [1, 2]. Shlaer-Mellor OOA models are made up of three separate, but integrated, parts:

- First, we build an *Object Information Model* that defines the objects (conceptual entities) of the domain and the relationships between the objects.
- Second, we build *State Models* that prescribe the lifecycles (behavior) of each object. We generally build one state model for each object on the Object Information Model. We do not build state models for objects that have no dynamics, and we may build an additional state model for objects that manage contention.
- The third model is the *Action Specification* that factors and formalizes the processing required in the State Models. There is one action specification for each state of each object's lifecycle. Action specification can be done in either Action Data Flow Diagrams or a well designed action language.

This produces a set of formal models shown in Figure 2.

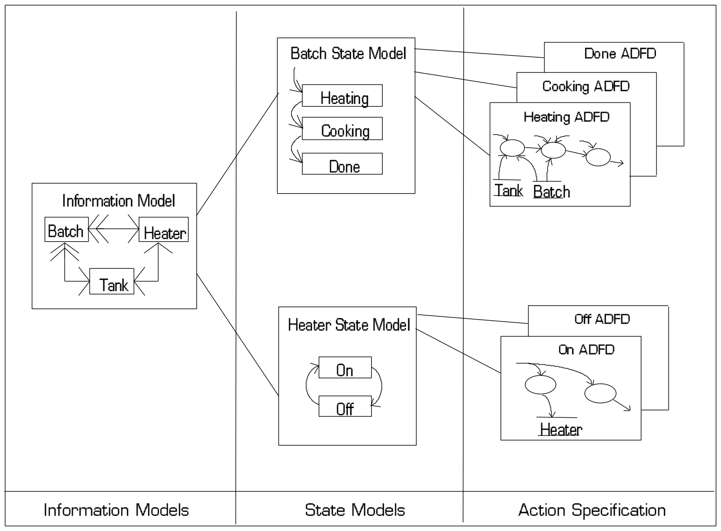

Figure 2: Structure of the primary models of Object-Oriented Analysis

The models of OOA are based on formal structures with strict rules. The Object Information Model is based on the relational model of data, augmented so that the selected abstractions also account for behavior. The State Models are deterministic finite state automata of the Moore variety [9]. The Action Specification shows sequence and conditionality, and each process can be defined in a formal language. The strict rules for these structures force certain decisions, such as placement of data, placement of responsibilities, and placement of functions. These forced decisions tend to reduce choices that the analyst can make, freeing the analyst to focus on the abstractions required in the domain. These decisions also help make models made by different analysts comparable: there is the concept of a 'right' way to build the models. This reduces unnecessary discussion about issues of taste. The strict rules also tend to expose certain types of issues very early in the analysis process, including, for example, what data is required to support the abstraction, how abstractions interact dynamically and so on. There is no way for the analyst to avoid facing fundamental issues about how things work in a domain.

The rules and formality permit the models to be mathematically transformed into other forms. For example, the relational structure of the data definition can be transformed into a linked list, a tree, a set of list based on equivalence classes, or we could even not store the data at all, instead computing the values as required at run time. Similarly, the state models can be joined together, split apart, and repackaged in a variety of ways. It is entirely possible to implement the state models in completely different ways: a set of coroutines, one for each action; as a block of code with break points; or as linear code modules that follow a thread of control. Partitioning rules on the construction of the processes in an action specification allow accessor processes to be implemented as encapsulating functions of a class, as SQL operations in a database, or as simple data accesses. Some processes may even have no physical manifestation at all because of the choice of data structure. All of these issues are, however, design and implementation issues, and they should not be considered at analysis time. The analysis formalism requires only that the analyst say what data is required, when something is to be done, and what exactly must be done. The designer will decide how to represent this in the most efficient manner.

The process of building the OOA models is similarly proscribed. The analyst proceeds first to define the objects on the Object Information Model, then to describe their behavior over time, and finally to define the processing. Figure 2, above, indicates that the work must be done in this order because the decision about which action specifications to build is determined by the states in the state models, and which state models to build is determined by the objects on the object information model. In practice, project teams discover that relatively little iteration is required between the three models, once a certain level of experience is reached.

In addition to the three fundamental models, there are also a number of *derived* models, as shown in the table below. The first group of derived models exists at the subsystem level, and act as an overview of the various relationships between subsystems. The second group of derived models exist for each subsystem.

Model Name	**Purpose**
Subsystem Relationship Model	Shows the relationships between objects in different subsystems.
Subsystem Communication Model	Shows the asynchronous event communications between objects in different subsystems.
Subsystem Access Model	Shows the synchronous data accesses between objects in different subsystems. The accesses are derived from the action specifications.
Object Communication Model	Shows the asynchronous event communications between objects. Objects that have state models, and events between those state models, are shown.
Object Access Model	Shows the synchronous data accesses between objects. The accesses are derived from the action specifications.

All these models can be built using CASE tools[1], and need not be built by the analyst directly. However, it is often helpful to build these models early in the analysis process to guide the work. This is especially true for the Object Communication Model, because this model shows the communication between objects. It is helpful to build the Object Communication Model to visualize that objects send events to that other objects. Once the fundamental models are built, the derived models can be rederived from the fundamental models, guaranteeing consistency between them.

[1]For example, BridgePoint Model Builder can derive all of these models.

2.3 Confirm the Analysis

Because of the integrated, complete, and formal nature of the Shlaer-Mellor OOA models, it is possible to confirm the specified behavior of the system both statically and dynamically.

The three models form a single, integrated unit that, when taken together, define precisely the data, behavior and processing required in the domain. There is a set of rules, described in [8], for the syntax of the models that define consistency of an OOA model set, and these rules form the basis for a static verification. Each one of the rules can be checked, and the elements that appear in several models can be cross-checked against one another. There are two basic approaches to such checking: One approach is to allow the user to enter a set of models into a CASE tool, and then check each rule later. The alternative and preferred technique is for a CASE tool to take the approach of allowing the analyst to enter only legal models. The second technique dramatically improves analysts' productivity.

The formalism for the models is sufficiently well defined that the models can be executed. In this execution, the *data* that is processed is defined in the Object Information Model, the *sequence* in which the processing takes place is specified in the State Models, and the *processing* is specified by the Action Specification. The execution of the models can be simulated using the following process:

1. Establish the desired initial state of the system in data values in the Object Information Model.
2. Initiate the desired behavior with an event sent to a State Model.
3. Execute the processing as specified by the Action Specification and as sequenced by the State Models.
4. Evaluate the outcome against the expected results.

When the processing in the Action Specification is specified in an executable manner, the execution process can be automated. Note that this approach answers the age-old question of when analysis is complete: a Shlaer-Mellor OOA is complete when it executes properly, that is, exhibits the required behavior.

2.4 Extract Requirements for Service Domains

During the analysis of a domain, we abstract away and ignore certain issues, such as how we present information to an operator, or how we intend to store data. We assume that these facilities will be provided by other domains. Each bridge between domains represents the set of assumptions made by the client domain on a particular server domain. These assumptions are then viewed as requirements by the server domains, as illustrated in Figure 3. In this example, the application domain assumes that mechanisms will be provided for storing persistent data, for transport of events, and for communicating with an operator. These requirements are assigned to the various service domains in the system. These become the basis for proceeding with analysis on the remaining domains.

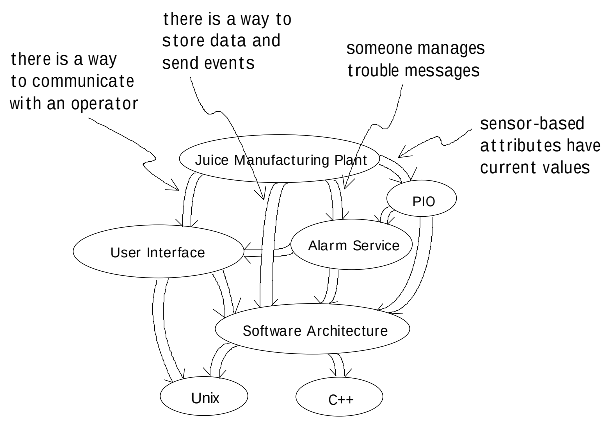
there is a way to communicate with an operator
there is a way to store data and send events
someone manages trouble messages
sensor-based attributes have current values
Juice Manufacturing Plant
PIO
User Interface
Alarm Service
Software Architecture
Unix
C++

Figure 3: Requirements generated by the application domain.

In addition to these qualitative requirements, we may also specify quantitative requirements. For example, we assume that we have a mechanism for storing up to two thousand object instances, drawn from two dozen classes. We expect to create and delete instances in the application domain quite slowly, perhaps one instance of each of twelve classes in response to an operator request. We expect the operator to be one initiator of external events; the longest thread is twelve objects long. We require a response in less than one quarter of a second. We require that we know the precise status of physical entities in the application every second. Consequently, the position of various entities in the physical plant is read each half second, and the corresponding icons are updated on the screens. We require an iconic interface, with a maximum of 100 icons visible at any time, etc.

These qualitative and quantitative requirements feed the analysis of the service domains.

2.5 Analyze Service Domains

Given the qualitative and quantitative requirements from all the client domains, we can proceed to analyze each of the remaining domains separately using Shlaer-Mellor OOA. The requirements will shape the abstractions that we choose for the server domain. The fact that the client domain requires an iconic interface, for example, will cause the analysts of the user interface domain to choose conceptual entities such as icon, and to choose to distinguish between updateable icons and unupdateable icons for the sake of efficiency. The unupdateable icons are thought of differently from updateable icons in order to meet the performance constraint. As the analysis of each domain is completed, its behavior is verified through an execution of the OOA models.

The order of the work is imposed by the client-server relationships: one must not begin analysis of a server (lower) domain until sufficient analysis has been done of its clients to ensure that all requirements on the server have been identified. Because of the form of the qualitative and quantitative requirements as described above, it is frequently the case that work can begin on analyzing server domains very early in the process. In fact, in organizations that tackle similar systems repeatedly, work often begins on all domains concurrently. However, beginning the analysis of a server domain too early in the project will increase risk. When the quantitative requirements are particularly stringent, it is sometimes advisable to delay starting work on a server domain until all the requirements imposed by the client domain are well understood.

This process continues downwards until we reach domains that are implemented and already provide the required capabilities. These typically include such domains as operating system, programming language and communication network.

2.6 Specify the Architectural Domain

The final domain to be treated is the architectural domain. The subject matter of this domain is system design, and it

- specifies generic mechanisms and structures for managing data, function and control for the system as a whole and
- defines the translation of the OOA models into these mechanisms and structures.

Issues that must be addressed by the architecture domain include: available combinations of languages, operating systems, and platforms; performance requirements for data access and event transmission; multitasking and synchronization requirements; distribution across platforms; type and degree of flexibility desired; and an appropriate translation of the OOA models.

A defining feature of the Shlaer-Mellor Method is that the statement of the architecture is *application-independent*. Instead of elaborating the analysis models, adding in design information and reorganizing the analysis, we make a statement of the mechanisms and structures required to support the OOA formalism. The architecture must therefore provide ways to store data, ways to transmit events, ways to access data, and the like. Of course, the approach taken must be efficient for the type of application involved.

The primary purpose of specifying a system-wide set of mechanisms and structures is to impose *uniformity on the construction of the software*. This is accomplished by stating policies covering such things as: how data is organized and accessed, how threads of control are managed, and how application and service code is to be constructed for the *entire system.* This uniformity reduces the complexity of the resultant system by providing standard mechanisms and structures for accomplishing widely-needed functions.

The architectural domain can be specified in OOA. The conceptual entities (objects) in this domain are defined to support the type of design desired: In a fast process control system, the conceptual entities will include periodic tasks and a universally accessible run-time database, whereas an object-oriented design (OOD) will be based on object-oriented concepts and include entities such as container classes and classes that implement mechanisms such as finite state machines, object communication, and timers.

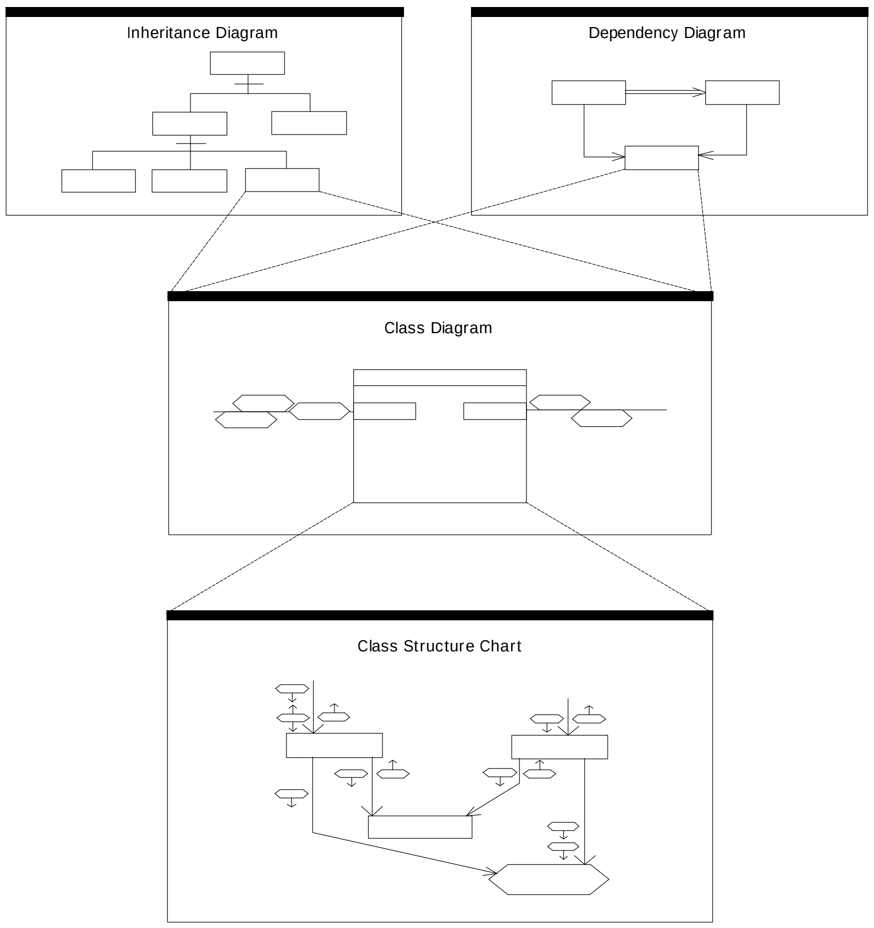

Figure 4: Object-Oriented Design Diagrams

Object-Oriented Designs. For object-oriented designs we use a notation supporting OOD constructs [5]. Four distinct diagrams are used in this notation for depicting the design of an object-oriented program, library, or environment, as shown in Figure 4. In this figure, there is:

- a *Class Diagram,* which shows the external view of a single class,
- An *Inheritance Diagram,* which shows the inheritance relationships between classes,
- a *Dependency Diagram,* which depicts the invocation and 'friend' relationships that hold between classes,
- and a *Class Structure Chart*, which shows the internal structure of the code of the operations of a class.

2.7 Build Architectural Components

The architectural domain is specified in two types of components: mechanisms and structures. *Mechanisms* represent architecture-specific capabilities that must be provided to realize the system. Examples of architectural mechanisms in an object-oriented design include classes to implement such things as the finite state machines, object communication events and the timers of OOA. Mechanisms are built in a traditional manner by designing and implementing code. This code takes the form of tasks and class libraries.

Structures represent a prescription for translating the OOA models of the client domains. An example of an architectural structure is to prescribe that all client analysis objects are realized as classes that inherit the ability to traverse their state models from an architectural class. Structures are built as a combination of code and data that needs to be completed by adding elements from the client domains such as variable declarations, operation declarations and operation code. These built structures are called *archetypes*. Adding elements from the client domains, or "populating the archetypes" as it is called, takes place prior to source code compilation. With proper CASE support and relatively straightforward tools, much of this work--up to 100%--can be automated using the OOA models as the source for client domain elements.

In summary, two different types of architectural components are built to realize the specifications for the architectural domain. Mechanisms are realized as traditional software tasks and library components. Structures are realized as archetypes containing code and data that need to be populated with elements from the client domains. Archetypes are highly reusable and their construction offers many opportunities for automation.

2.8 Translate the OOA Models

The details of the activities for this step are very dependent on the exact nature of the chosen design and the resulting architectural components. What can be described for this step are the general nature of the activities and the associated support that the Shlaer-Mellor Method provides. In the most general case of multitasking and multiprocessor systems, the basic activities are to:

- allocate instances of objects to tasks, and tasks to processors, and
- create each task by a translation of the OOA models.

The allocation of objects to tasks takes into account such issues as the communication between objects (shown in the Object Communication Model of the OOA [2]), the inherent concurrency of the problem (shown in the Thread of Control diagrams of the OOA [2]) and the ability of the operating system to simulate concurrency. The allocation of tasks to processors takes into account such issues as the communication between tasks, the inherent concurrency of the problem, and the capabilities of the inter-processor communication mechanism. The results of these allocations are shown in the Task Communication diagram [6].

The second major activity of this step is to create the tasks identified by the above allocations. In a strong architecture, the archetypes prescribe virtually the entire design of these tasks. This design is realized by populating the replaceable segments of the archetypes with elements from the OOA models. This design activity may be carried out by a combination of automation and orderly manual procedures as previously described. The final step in creating the tasks is to write the code for the algorithms specified in the Action Specification. Note that the granularity of code being built at this step is class and instance operations. To illustrate this approach, an example of a partial code archetype and the populated result are shown in Figure 5 for a particular object-oriented architecture.

Archetype

```
//      Angle brackets indicate a name to be replaced by the archetype  processor,
//      which may be a programmer, an editor, a special purpose program linked
//      to a CASE tool, or a preprocessor.
//
//      <active class name> is the name of the class being defined.
//      <active class type> is the type of the class being defined.
//
//      Subscripted variables indicate the ith element as follows:
//
//              attri = the ith declared attribute of the source object
//              attri type = the type of the ith attribute
//              eventi = an event number handled by the class
//              dataseti = event data for the ith event
//              dataseti attrj = jth attribute of the ith dataset
//
//      Declare the class
class <active class name> : public active instance
{
//      Declare the instance components
private:
        <attr1 type> <attr1>;
        <attr2 type> <attr2>;
                                . . .
public:
//      First the class operations
//      <active class type> Institute( <attr1 type>,  . . .) is written as:
type>,  . . .) is written as:
        <active class name> (<attr1 type>, <attr2 type>, . . .);
        static void Load_FSM();
//      Now the class event takers
        static void Create_Event_<event1>(<dataset1 attr1 type>,<dataset2 attr2 type>, . . .);
customer, money);
        static void Create_Event_<event2>(<dataset2 attr1 type>,<dataset2 attr2 type>, . . .);
        . . .
//      Now the instance operations, event takers first
        void Take_Event_<eventn+1>(<datasetn+1 attr1 type>,<datasetn+1 attr2 type>, . . .);
        void Take_Event_<eventn+2>(<datasetn+2 attr1 type>,<datasetn+2 attr2 type>, . . .);
        . . .
//      Now the read accessors
        <attr1 type> Read_<attr1>();
        <attr2 type> Read_<attr2>();
        . . .
};
```

Populated Result

```
//      Declare the class
        class account: public active_instance
{
        //      Declare the instance components
        private:
        int account_ID;
        money balance;
        customer customer_ID;
        public:
        //      First the class operations
                //      <active class type> Institute( <attr1

                        account ( int, money, customer );
                static void Load_FSM();
        //      Now the class event takers
                        static void Create_Event_Init(


        //      Now the instance operations, event takers first
                        void Take_Event_Debit(money);
                        void Take_Event_Deposit(money);
        . . .
        //      Now the read accessors
        money Read-balance();
        customer Read_customer_ID();
        . . .
};
```

Figure 5: Archetype and Populated Result

In this example, the archetype prescribes the translation of an OOA object into a C++ class. The replaceable segments in the archetype, as indicated by the angle brackets, have been replaced with elements from the OOA models. In this case, a banking example, the OOA contained an object `Account` with attributes `Account ID, Balance, and Customer ID,` which responded to events `Init, Debit,` and `Deposit`, and allowed read access, via encapsulated functions, to its `Balance` and `Customer ID` attributes. The archetype also prescribes a mechanism for loading the account state model, `void Load_FSM()`. Another archetype would be specified in this architecture to define the structure and content of the main module for the task in which this object would reside. This particular architecture is described in more detail in reference [2].

2.9 Benefits of Shlaer-Mellor Method

Verification. The ability to simulate the execution of the Shlaer-Mellor OOA models clearly establishes the completion criteria for analysis and allows the functional behavior of the system to be verified early, at analysis time.

Integrated Approach. The Shlaer-Mellor Method provides extensive lifecycle coverage with smooth transitions between the analysis, design, and implementation phases using Recursive Design.

Iteration. This approach reduces and controls iteration in analysis by confining it to a single domain at a time. Iteration in design is similarly controlled: Modifications to the design are made entirely in the architectural domain and propagated to the entire system through the archetypes.

Reuse. The approach systematizes and supports reuse of entire domains. Because domains are kept completely separate from one another until the final construction steps, they can be transported intact to other systems. This applies particularly to the architectural domain: This domain, including the mechanisms and archetypes, is commonly reused for other systems that have basically the same loading and performance characteristics. Mappings unite the domains into a system at the end of the development lifecycle.

Automation. Because the process is based on translation of analysis models, it is highly susceptible to automation. The most recent generation of CASE tools have created the capability for 100% code generation.

3. Deliverables

The major deliverables for the Shlaer-Mellor Method are listed below by development phase. A more detailed listing and explanation of these deliverables is available in reference [2].

<u>Analysis Phase</u>

System Level:
- *Domain Chart*
- *Project matrix*

Domain Level:
- *Subsystem Relationship Model*
- *Subsystem Communication Model*
- *Subsystem Access Model*

Subsystem Level:
- *Object Information Model*
- *Object Communication Model*
- *Object Access Model*

Object Level:
- *Object State Model*
- *State Action Specification*

<u>Design Phase</u>

System Level
- *Task Communication Diagram*

Task Level
- *Inheritance Diagram*
- *Dependency Diagram*
- *Task Archetypes*

Class Level
- *Class Diagram*
- *Class Structure Chart*
- *Class Archetypes*

<u>Implementation Phase</u>
- *Populated Task Archetypes*
- *Populated Module Archetypes*

4. Pragmatics

The Shlaer-Mellor Method is available to the public in the form of books, published articles, training courses and consulting, all supplied by Project Technology, Inc. of Berkeley, CA. The books and some of the articles have already been referenced in this description of the method. A variety of training courses are also available for various audiences. For engineers who plan to apply this method, there is a series of two, one-week courses that cover the entire method. Each of these courses includes extensive work on a realistic case study to provide as much "hands on" experience as possible. A two-day Fundamentals course is also offered for first level managers and related engineering roles such as hardware development and testing.

The method is supported by a variety of CASE tools, including BridgePoint® tools by Project Technology.

The interested reader is encouraged to contact Project Technology, Inc. at **(510) 567-0255** or visit our web site at **http://www.projtech.com** for a complete and current listing of references and courses.

References

[1] Sally Shlaer and Stephen J. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, Prentice Hall, 1988.

[2] Sally Shlaer and Stephen J. Mellor, *Object Lifecycles: Modeling the World in States*, Prentice Hall, 1992.

[3] Sally Shlaer and Stephen J. Mellor , *Recursive Design*, in Computer Language, March 1990, Volume 7, Number 3, Miller Freeman, Inc.

[4] Carma McClure, *Software Automation in the 1990's and Beyond*, in Proceedings of Fall 1989 CASE Symposium. Andover, MA: Digital Consulting, September 1989.

[5] Sally Shlaer, Stephen J. Mellor and Wayne Hywari, *OODLE: A Language-Independent Notation for Object-Oriented Design*, in Journal of Object-Oriented Programming Focus on Analysis and Design (special issue), SIGS Publications, 1991.

[6] Sally Shlaer, Stephen J. Mellor and Wayne Hywari, *Real-Time Recursive Design*, Course Notes, Project Technology, Inc.

[7] Sally Shlaer, Stephen J. Mellor and Diana Grand, *The Project Matrix: A Model for Software Engineering Project Management*, in Proceedings of the Third Software Engineering Standards Application Workshop (SESAW III), IEEE, October 1984.

[8] Neil Lang, *Shlaer-Mellor Object-Oriented Analysis Rules*, Software Engineering Notes, ACM Press, Volume 18, Number 1, January 1993.

[9] E. F. Moore, *Gedanken-experiments on Sequential Machines*, Automata Studies, Princeton University Press, Princeton, N.J., 1956