How and Why to Use the Unified Modeling Language

Lynn Sanderfer
TecMasters, Inc.

This article addresses the Unified Modeling Language and its purpose, constructs, and application to defense software development applications.

The Unified Modeling Language (UML) is a notation that can be applied to the software development process. UML, in itself, is not a software development process, but a system modeling language developed by Grady Booch, James Rumbaugh, and Ivar Jacobson of Rational Software. It provides a design notation whereby the scenarios for which the system requirements can be utilized are depicted as well as subsequent design notation for the chosen implementation. This methodology was a merging of these men's separate practices in object-oriented (OO) software design notation[1]. The primary goal of UML is to model systems using OO software (also referred to as architectural-based software).

UML: The Why

Whenever something is built, drawings are made to describe the look and feel of the entity being built. These drawings work as a specification of how we want the finished product to look. The drawings are handed over to *builders* or are broken into more detailed drawings necessary for the construction. A well-architected product pays off in the end, but high quality does not just happen. Software quality is a result of correctly understanding the requirements, a solid design that is readily implemented into code, quality user documentation, and is committed to satisfying the user's needs.

Software design is the equivalent of these construction drawings, and is developed simply to produce a software solution to a problem. The process of software design can be described as an activity in which the designer develops a highly abstract model of a solution and then transforms it into a very detailed design. OO, or architectural-based design, guides the designer into thinking about the decomposition of problems into a collection of autonomous agents or objects that can be mapped as closely as possible to a physical representation of the system. The designer can then resolve the system in terms of behaviors and responsibilities of objects.

By reducing the interdependency among software components, architectural-based programming permits the development of reusable software. Such software components can be created and tested as independent units in isolation from other portions of a software application. Programming then becomes the simulation of the model spectrum. Keep in mind that the designer's ultimate requirement is to meet the *fitness of purpose* of the user's needs: Does it work and does it do the required job as well as possible[2]?

Note that it takes less time to build a system by making *instructions* (and following them) than it would take to start from scratch to build a system without directions. This is because documenting the specifications to the desired product allows the analyst/developer to (1) verify his understanding of the task at hand, (2) visualize and identify the most crucial components of the system, and (3) identify goals that keep the team focused to the system objectives. The time considered gained by omitting the development of concrete plans is paid for many times over in misinterpreted requirements, inefficient code implementation, faulty software, and unhappy end-users.

UML is a standardized design language that provides a mechanism for creating this design, particularly when the software being developed is done by using OO principles.

UML uses foundation pieces that describe the abstraction of system components called classes, and the instantiation of those classes into system objects that are later used to provide management of component functionality and data. For further details on OO techniques, see the *Notes* section at the end of this article.

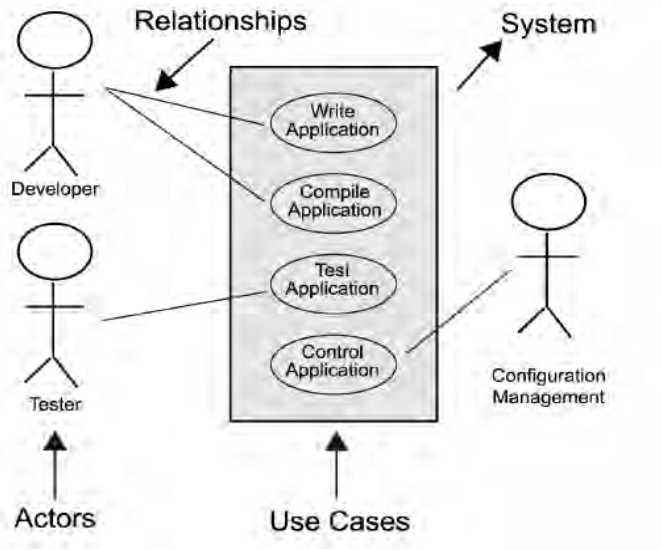
UML: The How

UML is defined by depicting the software from various aspects or views. Each view can be displayed using a variety of diagrams that detail the contents of the views. Each diagram is composed of OO concepts to include classes, objects, messages, relationships, and dependencies. These concepts or elements are persistent throughout the design process, or rather they do not change meaning or symbology throughout the design.

By looking at a physical system from different views, a developer or user can concentrate on one aspect of the system at a time. The UML views are the following:

- **Use Case View:** Depicts the functionality of the system as perceived by the external actors.
- **Logical View:** Depicts how the functionality is designed inside the system.
- **Component View:** Depicts the organization of the code components.
- **Concurrency View:** Depicts concur-

Figure 1: *Use Case Diagram*

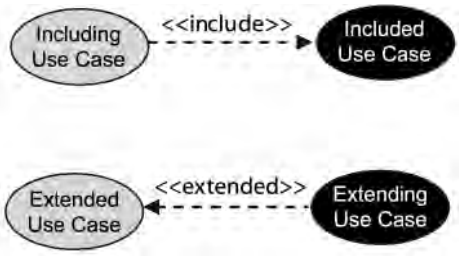


rency in the system, addressing the problems with communication and synchronization that are present in a concurrent system.

- **Deployment View:** Depicts the deployment of the system into the physical architecture with computers and nodes.

To display these views, UML introduces nine different types of diagrams: Use Case Diagrams (see Figure 1), Class Diagrams, Object Diagrams, Sequence Diagrams, State Diagrams, Collaboration Diagrams, Activity Diagrams, Component Diagrams, and Deployment Diagrams.[3]

Figure 2: *Extended/Include Use Case Notation*



Use Case Diagrams

Use case modeling is a design modeling technique that uses both verbal and graphic descriptions to reveal what a new system should do or what an existing system already does via Use Case Diagrams. These diagrams are the starting point when designing a new system using UML. This notation or diagram is designed to communicate exactly what is expected of a system upon completion to management, customers, and other interested parties.

There are four basic components of Use Case Diagrams:

- System.
- Actors.
- Use cases.
- Relationships.

The system is the entity that performs the function. Actors are entities, people, or other systems that use the system to be developed. Use cases are the actions that a user takes on a system. Relationships depict how actors relate to use cases.

If additional functionality is required in a use case description, this can typically be handled by *include* and *extend* relationships (see Figure 2). The include relationship is used to indicate that a use case will *include* functionality from an additional use case to perform its function. Similarly, the extend relationship indicates that a use case may be extended *by* another use case.

To describe use cases, a software designer would first identify the actors of the system. Next, designers would ask the providers of the requirements for more information about what they want. A designer should always keep these questions in mind as this information exchange progresses: "What will the system do?", "What input/output does the system need?" and "What are the major problems with the current implementation of the system?"

Figure 3: *Sample Class Diagram Notations*

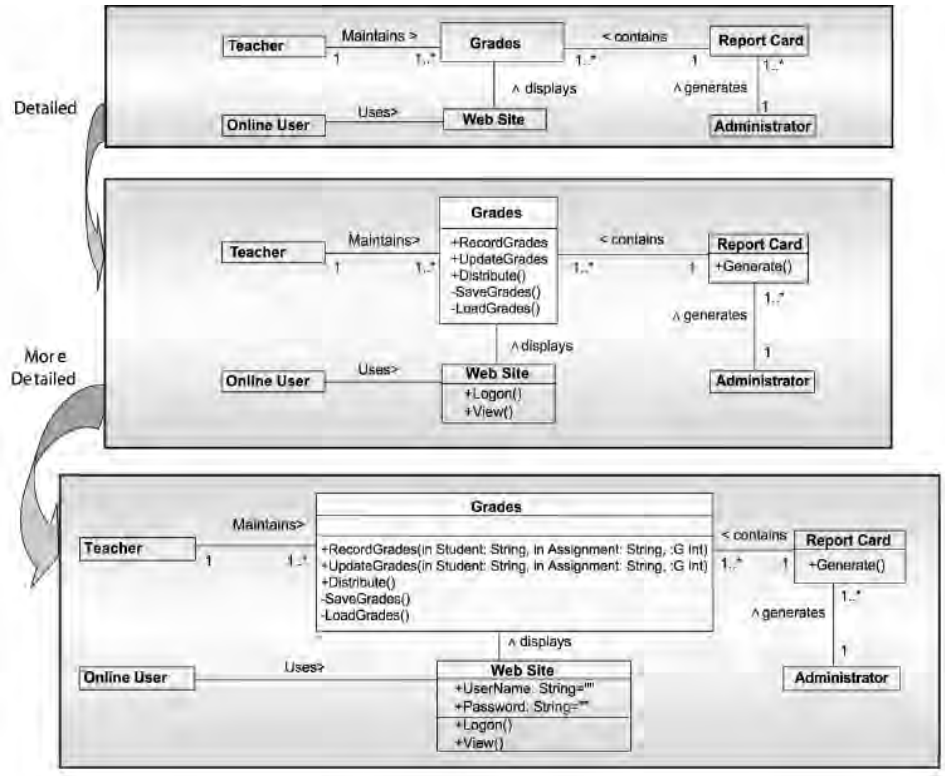
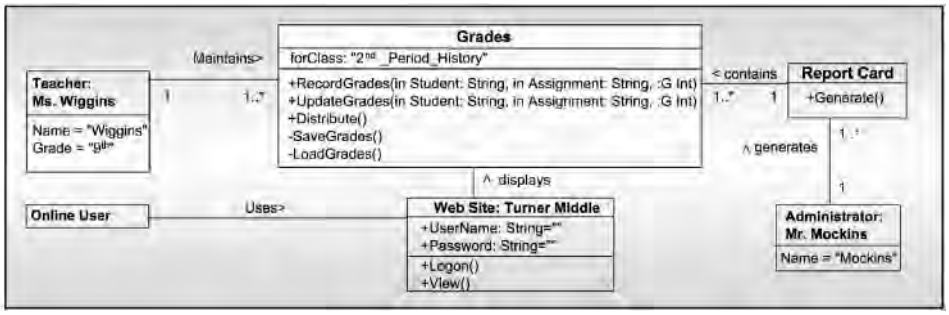


Figure 4: *Sample Object Diagram*



indicate the direction of service, *how many* relationships called multiplicity, and repetitiveness.

Sequence Diagrams

A Sequence Diagram describes the dynamic view of the system (see Figure 5). This diagram is important in that it shows the sequence of messages sent between the objects with respect to time. The Sequence Diagram consists of a number of objects shown with vertical lines. The objects are activated from left to right with an indication of the exchange of messages between the objects. Messages themselves are shown as lines with message arrows between the vertical object lines.

Each object in a Sequence Diagram is represented by an object rectangle with the object/class name underlined. A vertical dashed line down from the object, called the object's lifeline, indicates the object's execution during the sequence. Communication between objects is represented as horizontal message lines between the object's lifelines. The arrows indicate whether the message is synchronous (flow is interrupted until the message has completed), asynchronous (active object does not wait on a response), or simple (flat flow depicting control is passed without indicating details). Sequence Diagrams can also show branching, iteration, recursion, creation, and destruction of objects.

State Diagrams

State Diagrams are dynamic in nature and depict how an individual object changes state when a behavior is invoked (see Figure 6, next page). It also indicates the invoking event. A state diagram should be attached to all classes that have clearly identifiable states and complex behavior.

A state-chart diagram is composed of states, transitions, and events. A basic state is shown as a rectangle with rounded corners. The name of the state is placed within the rectangle. The first state in a model, or the start state, is simply a solid dot. The last state in a model (the end state) is a solid dot with a circle around it. Transitions are used to show flow from one state to another. A transition is modeled by an open arrow from one state to another.

Collaboration Diagrams

Collaboration is another depiction of the dynamic behavior of a system (see Figure 7, next page). Collaboration Diagrams, showing both a context and an interaction, can be thought of as a combination of the Class Diagram and Sequence Diagram. Collaboration Diagrams actually model either objects or roles and their sequenced communication between each other. Message sequence in Collaboration Diagrams is identified by numbering the messages with labels.

Message type in Collaboration Diagrams is the same as in Sequence Diagrams: synchronous, asynchronous, and simple. Messages sent in parallel can be described using letters in the sequence number expression. For example, the sequence numbers 1.1a and 1.1b of two messages in the collaboration diagram indicate that those messages are sent in parallel.

Activity Diagrams

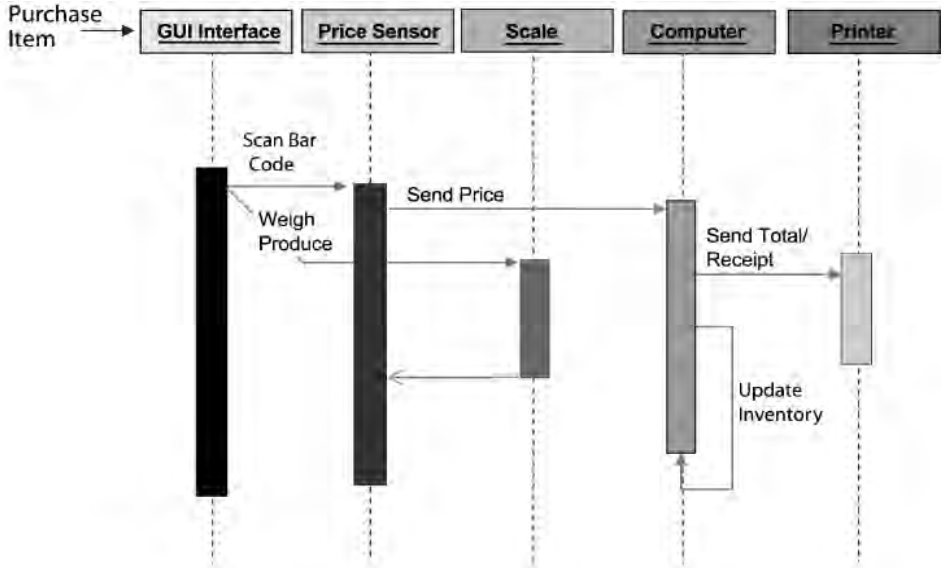
The Activity Diagram captures actions as the states of a system change (see Figure 8, page 17). Activity Diagrams focus on work performed in the execution of a function. The states in the Activity Diagram transition to the next stage directly when the action in the state is performed without specifying any event. Activity Diagrams also have swimlanes, or a grouping of activities according to the responsible software entity. Activity Diagrams show the following:

Object-Oriented or Architectural-Based Concepts

Architectural-based design models the problem in terms of a set of particular entities or objects that can be recognized in the problem itself, together with a description of the relationships that link these entities. The initial description of the system is defined in an abstract top-down process. A complete and detailed model of the solution is then developed by elaborating the descriptions of the entities and the interactions occurring between them. The strategy then is composed of grouping elements together in the design that can be described with the same functionality. The implementation is done as a bottom-up development process. Programming from an architecturally based design is comprised of the following concepts:

- Object is the encapsulation of data values (or states) and operations (or behaviors) into one executable entity.
- Class is the abstract or generic description of a concept in terms of data types and operations.
- Inheritance is the principle that knowledge of a more general category is applicable also to the more specific category. In this way classes can be organized into a hierarchical inheritance tree.
- Information hiding is the principal by which a client sending a request or invoking a process need not know the actual means by which the request will be honored.
- Abstraction is the ability to describe a type or functionally in generic terms thereby isolating design and execution information. Information hiding is a specific type of abstraction.

Figure 5: *Sample Sequence Diagram*



- The work that will be performed when an operation or method is executing.
- The internal work of an object.
- How a set of related actions may be performed, and how they affect objects around them.
- An instance of a use case in terms of object state changes.

Activity Diagrams can have a start and an end point. A start point is shown as a solid filled circle; the end point is shown as a circle surrounding a smaller solid circle. The actions in an Activity Diagram are drawn as rectangles with rounded corners. Within the action, a text string is attached to specify the action(s) taken. Transitions between actions are shown with an arrow, to which guard-conditions, a send-clause, and an action-expression can be attached. A diamond shaped symbol is used to show a decision point. Swimlanes group activities, typically, with respect to their responsibility. Swimlanes are drawn as vertical rectangles. The activities belonging to a swimlane are placed within its rectangle with a name at the top.

Component Diagrams

A Component Diagram is a type of implementation diagram (see Figure 9); it shows where the physical components of a system are going to be placed in relation to each other. In the Component Diagram the physical *pieces* are the actual software entities.

A component is shown in UML as a rectangle with an ellipse and two smaller rectangles to the left. The name of the component is written below the symbol or inside the large rectangle. A software dependency is shown as a dashed line with an open arrow and indicates that one component needs another to be able to have a complete definition.

Deployment Diagrams

The deployment view is the second type of implementation diagram and shows the physical layout of the software system after being installed on the actual hardware system and how the software components will interact with each other. Deployment Diagrams (see Figure 10) are described in terms of nodes, connections, and software components executing on nodes.

Nodes are physical objects that have some kind of computational resource. This includes computers with processors, as well as devices such as printers, communication devices, card readers, and so on. A node is drawn as a three-dimensional cube with the name inside it.

Nodes are connected via a communication association. The communication type is represented by a stereotype that identifies the communication protocol or the network used. Executable component instances may be contained within node instance symbols, showing their physical residence and execution on the hardware.

Figure 6: *State Diagram Example*

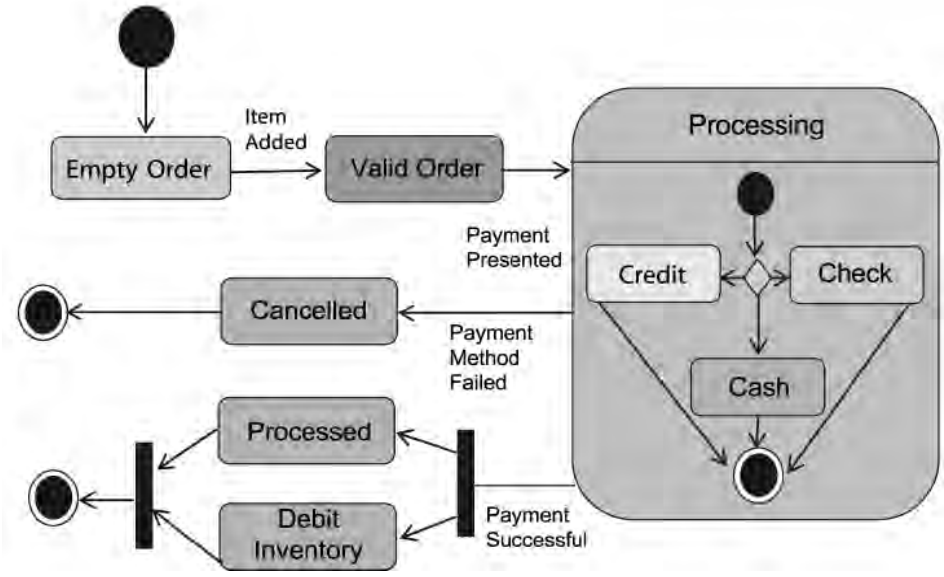
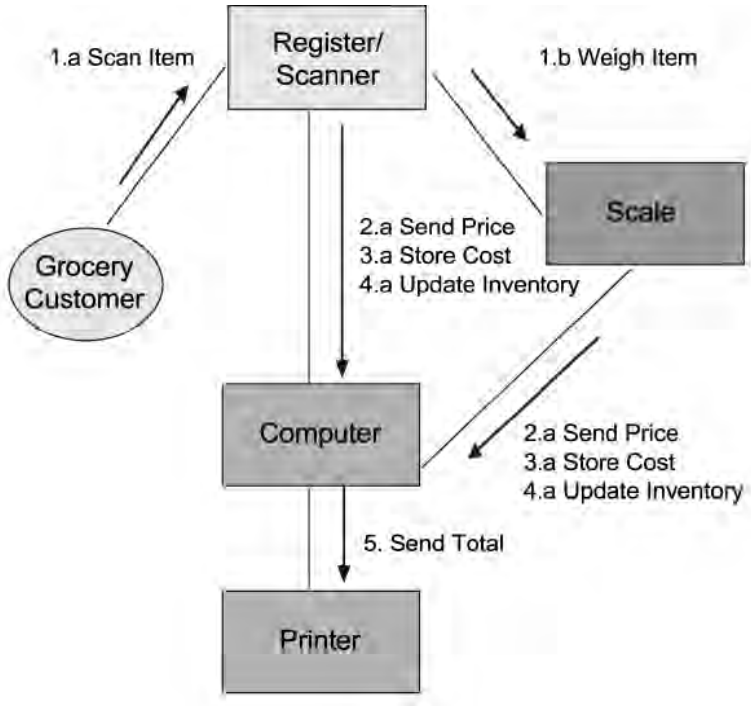


Figure 7: *Collaboration Diagram Example*



Conclusion

UML is a standard practical methodology used in representing OO systems, regardless of the procurement or financial aspects of the program, i.e., commercial or government. Advanced software projects or projects that are committed to leading-edge software technologies and concepts typically find value in designing their software systems to the physical systems that they represent. This allows non-software personnel to grasp the concept of operations of the system without having to understand software terminology. This also lends itself to reducing software life-cycle costs because it produces modular code with well-defined interfaces.

Historically, it can be shown that the more time rendered on requirements and design, the better the product when in the code and test phase. It does, in fact, improve the quality of your product, as well as other development factors, when the design of the system is clearly con-

veyed with respect to the intended requirement. However, as Doug Rosenberg stated [1], "The cold, hard reality of the world of software development is that there is simply never enough time for modeling."

The methodologies provided by UML are very rarely utilized exhaustively. There is simply not enough time or money to do so. In fact, I have seen excellent defense software efforts that implemented OO design with design techniques other than UML. However, to the designers and developers of software systems, UML offers very sound techniques in describing systems. Furthermore, if you are new to OO software development, UML provides a road map by which, if you take the time to depict the design in the various UML software design views, it is easier to produce a good product than not!◆

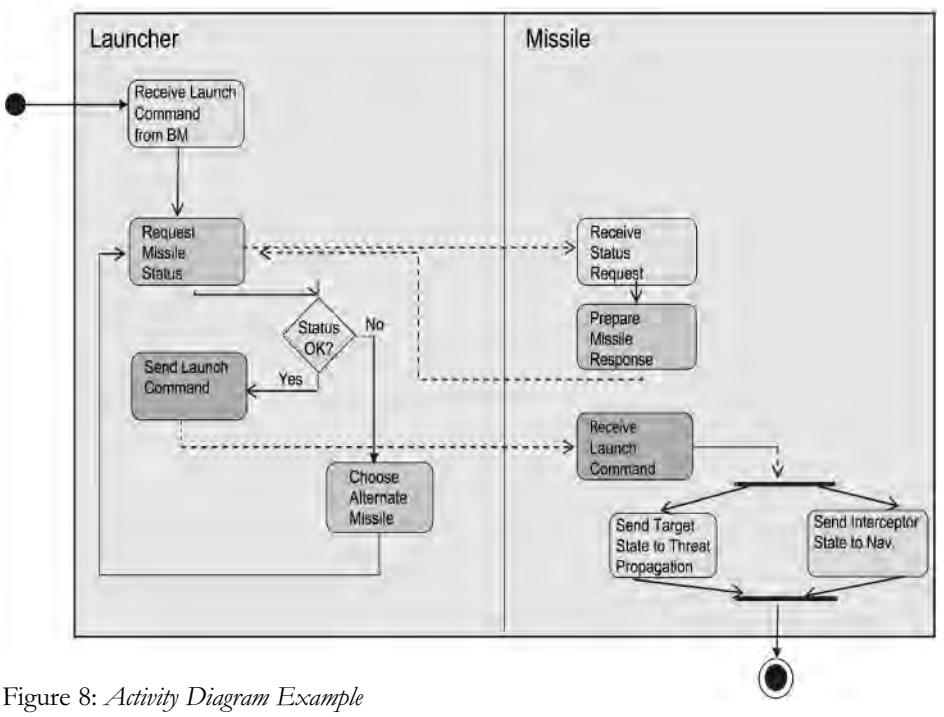


Figure 8: *Activity Diagram Example*

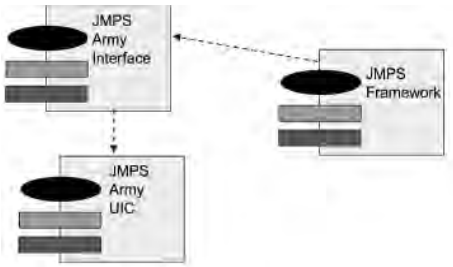


Figure 9: *Component Diagram Example*

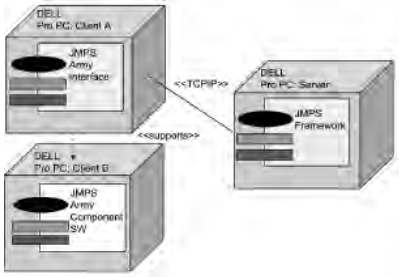


Figure 10: *Deployment Diagram*

Reference

1. Rosenberg, Doug, and Kendall Scott. Use-Case Driven Object Modeling With UML: A Practical Approach. Addison-Wesley Professional, Mar. 1999.

Notes

1. Grady Booch developed Booch diagrams. Booch defined the notion that a system is analyzed as a number of views where each view is described by a number of diagrams. The principals in his methodology were sound, based on depicting several views of the software; however, the Booch diagram notation that was based around clouds symbology was cumbersome to implement.
2. James Rumbaugh developed the Object Modeling Technique while employed at General Electric. His methodology was founded in expressing the software in various methods: the object model, the dynamic model, the functional model, and the use-case model.
3. Ivar Jacobson developed the Object-Oriented Software Engineering/ Objectory design methodology. His system is based on use-cases, which define the initial requirements on the system as seen by an external actor.

Additional Reading

1. Roff, Jason T. UML A Beginner's Guide. McGraw-Hill, 2003.
2. Eriksson, Hans-Erik, and Magnus Pinker. UML Toolkit. John Wiley & Sons, Inc., 1998.
3. Kroll, Per, and Philippe Druchten. The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP. Addison-Wesley, 2003.
4. Budgen, David. Software Design. Addison-Wesley, 1994.
5. Budd, Timothy. An Introduction to Object-Oriented Programming. Addison-Wesley, 1991.
6. Pressman, Roger S. Software Engineering: A Practitioner's Approach. 3rd ed. McGraw-Hill, 1992.
7. Stephen Prata. The Waite Group's C++ Primer Plus. 3rd ed. Indianapolis, IN: Sam's Publishing, 1998.

About the Author

Lynn Sanderfer is a software analyst for TecMasters, Inc. and is currently contracted on the Army Mission Planning Software Program. She has been in software development/software engineering for 18 years. Sanderfer has completed the Software Life Cycle Development Certification, the Software Engineering Management Certification, and the Advanced Software Development Certification offered by the Air Force Institute of Technology Software Professional Development Program. She is also certified as a Capability Maturity Model® Integration Auditor with CSSA, Inc., a licensed partner of the Software Engineering Institute. She has a bachelor's degree in engineering from the University of Alabama in Huntsville and is currently working on her master's degree in software engineering there.

TecMasters, Inc.
1500 Perimeter PKWY
STE 200
Madison, AL 35758
Phone: (256) 830-4000
Fax: (256) 830-4093
E-mail: theresa.sanderfer@us.army.mil