



Object Oriented Methodologies

Chapter 4

Chapter Objectives

- Object Oriented Methodologies
 - The Rumbaugh et al. OMT
 - The Booch Methodology
 - Jacobson's Methodologies
- Patterns
- Frameworks
- Unified Approach

Introduction

- In the 1980s , many methodologies were developed.
 - 1986. Booch developed the object oriented design concept, the Booch Method.
 - 1987. Sally Shlaer and Steve Mellor created the concept of the recursive design.
 - 1989. Beck and Cunningham produced class responsibility- collaboration cards.
 - 1990. Wrifts-Brock, Wilkerson, and Wiener came up with responsibility driven design.

Introduction

- 1991. Jim Rumbaugh led a team at the research labs of General Electric to develop the object modeling technique(OMT).
- 1991. Peter Coad and Ed Yourdon developed Coad lightweight and prototype-oriented approach to methods.
- 1994. Ivar Jacobson introduced the concept of the use case and object oriented software engineering(OOSE)



Rumbaugh et al.'s Object Modeling Technique

- The object modeling technique(OMT) presented by Jim Rumbaugh and his co-workers describes a method for analysis, design, and implementation of a system using an object –oriented technique.
- OMT is a fast and intuitive approach for identifying and modeling all the objects making up a system.



Rumbaugh et al.'s Object Modeling Technique

- Details such as class, attributes, method, inheritance, and association also can be expressed easily.
- The dynamic behavior of objects within a system can be described using OMT dynamic model.
- Finally a process description , a producer-consumer relationships can be expressed using OMT's functional model.

Rumbaugh et al.'s Object Modeling Technique

- OMT consists of four phases, which can be performed iteratively:
 - **Analysis:** The results are objects and dynamic and functional models.
 - **System design:** The results are a structure of the basic architecture of the system along with high-level strategy decisions.
 - **Object Design:** This phase produces a design document, consisting of detailed object static, dynamic and functional models.
 - **Implementation:** This activity produces reusable, extendible and robust code



Rumbaugh et al.'s Object Modeling Technique

- OMT separates modeling into three different parts:
 - An **object model**, presented by object model and the data dictionary.
 - A **dynamic model**, presented by state diagram and event flow diagrams.
 - A **functional model**, presented by data flow and constraints.

The Object Model

- The object model describes the structure of objects in a system: their identity, relationships to other objects, attributes and operations.
- The object diagram contains classes interconnected by association lines.
- Each class represents a set of individual objects.
- The association lines establish relationships among the classes.

The OMT Dynamic Model

- OMT provides a detailed and comprehensive dynamic model, in addition to letting you depict states, transitions, events, and actions.
- The state transition diagram is a network of states and events.
- Each state receives one or more events, at which time it makes the transition to the next state.
- The next state depends on the current state as well as the events.

The OMT Functional Model

- The OMT data flow diagram(DFD) shows the flow of data between different processes in a business.
- An OMT DFD provides a simple and intuitive method for describing business processes without focusing on the details of computer systems
- Data flow diagram use four primary symbols:
 - The **process** is any function being performed
 - The **data flow** shows the direction of the data element movement.
 - The **data store** is a location where data are stored.
 - An **external entity** is a source or destination of a data element.

The Booch methodology

- The booch methodology is a widely used object-oriented method that helps you design your system using the object paradigm.
- It covers analysis and design phases of an object-oriented system.
- You start with class and object diagram in the analysis and refine these diagrams in various steps.
- Only when you are ready to generate code the Booch method shines and you can document your object oriented code.

The Booch methodology

- The booch method consists of the following diagrams:
 - Class Diagram
 - Object Diagram
 - State transition Diagram
 - Module Diagrams
 - Process Diagrams
 - Interaction Diagrams
- The booch method prescribes a macro development and a micro development process.



The Macro Development Process

- The macro process serves as a controlling framework for the micro process and can take weeks or even months.
- The primary concern of macro process is technical management of the system.
- In macro process the traditional phases of analysis and design to a large extent are preserved.
- The macro development consists of the following steps:

The Macro Development Process

- **Conceptualization:**
 - During Conceptualization , you establish the core requirements of the system.
 - Establish a set of goals and develop a prototype to prove the concept
- **Analysis and Development of the model:**
 - In this step, you use the class diagram to describe the roles and responsibilities objects carry out in performing the desired behavior of the system.
 - Then, you use the object diagram to describe the desired behavior of the system in terms of scenarios, or alternatively use interaction diagrams.



The Macro Development Process

- Design or create the system architecture:
 - In the design phase, you use the class diagram to decide what classes exist & how they relate to each other.
 - Next you use the object diagram to decide what mechanisms are used to regulate how objects collaborate.
 - Then, you use the module diagram to map out where each class and object should be declared.
 - Finally, you use the process diagram to determine to which processor to allocate a process.



The Macro Development Process

- Evolution or Implementation
 - Successively refine the system through many iterations.
 - Produce a stream of software implementations(or executable releases), each of which is a refinement of the prior one.
- Maintenance:
 - Make localized changes to the system to add new requirements and eliminate bugs.

The Micro Development process

- Each macro development process has its own micro development processes.
- It is a description of the day to day activities by a single or small group of software developers.
- The micro development process consists of the following steps:
 - Identify classes and objects
 - Identify class and object semantics
 - Identify class and object relationships.
 - Identify class and object interfaces and implementation.

JACOBSON ET AL. METHODOLOGIES

- The Jacobson et al. methodologies (e.g., object-oriented Business Engineering (OOBE), object-oriented Software Engineering (OOSE), and Objectory) cover the entire life cycle and stress traceability between the different phases, both forward and backward.
- This traceability enables reuse of analysis and design work, possibly much bigger factors in the reduction of development time than reuse of code.
- At the heart of their methodologies is the use-case concept, which evolved with Objectory (Object Factory for Software Development).

THE JACOBSON ET AL. METHODOLOGIES

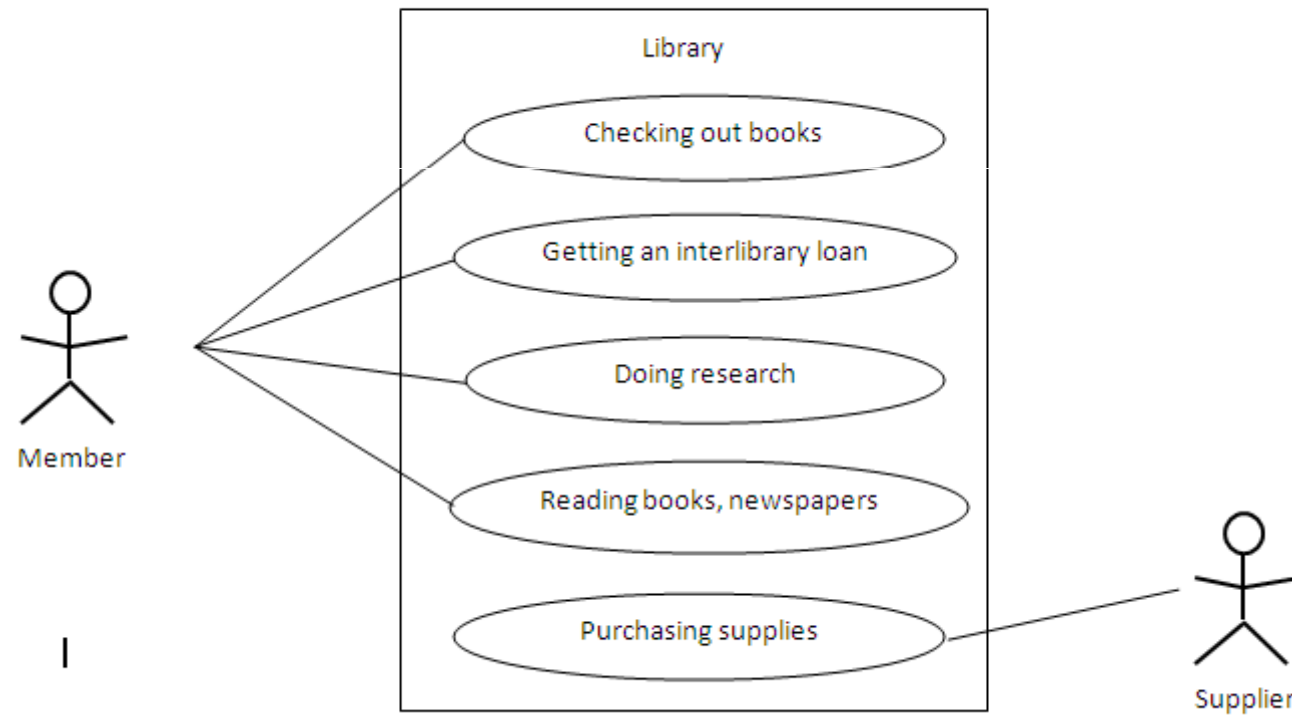
- **Use cases**

- Use cases are scenarios for understanding system requirements. A use case is an interaction between users and a system. The use-case model captures the goal of user and the responsibility of the system to its users

- **FIGURE 4-6:**

- Some uses of a library. As you can see, these are external views of the library system from an actor such as a member. The simpler the use case, the more effective it will be. It is unwise to capture all of the details right at the start; you can do that later.

THE JACOBSON ET AL. METHODOLOGIES




THE JACOBSON ET AL. METHODOLOGIES

- In the requirements analysis, the use cases, are described as one of the following :
 - Nonformal text with no clear flow of events.
 - Text, easy to read but with a clear flow of events to follow (this is a recommended style).
 - Formal style using pseudo code.

THE JACOBSON ET AL. METHODOLOGIES

- The use case description must contain:
 - *How* and *when* the use case begins and ends.
 - The interaction between the use case and its actors, including when the interaction occurs and *what* is exchanged.
 - *How* and *when* the use case will need data stored in the system or will store data in the system.
 - *Exceptions* to the flow of events.
 - *How* and *when* concepts of the problem domain are handled.



THE JACOBSON ET AL. METHODOLOGIES

- Use cases could be viewed as concrete or abstract.
- An ***abstract use case*** is not complete and has no actors that initiate it but is used by another use case.
- This inheritance could be used in several levels.
- Abstract use cases also are the ones that have uses or extends relationships.



Object-Oriented Software Engineering: Objectory

- Object-oriented software engineering (OOSE), also called *Objectory*, is a method of object-oriented development with the specific aim to fit the development of large, real-time systems.
- The development process, called *use-case driven development*, stresses that use cases are involved in several phases of the development , including analysis, design, validation, and testing.



Object-Oriented Software Engineering: Objectory

- The use-case scenario begins with a user of the system initiating a sequence of interrelated events.
- The system development method based on OOSE, Objectory, is a disciplined process for the industrialized development of software, based on a use-case driven design.
- It is an approach to object-oriented analysis and design that centers on understanding the ways in which a system actually is used.



Object-Oriented Software Engineering: Objectory

- Objectory is built around several different models:
 - *Use case-model*. The use-case model defines the outside (actors) and inside (use case) of the system's behavior.
 - *Domain object model*. The objects of the "real" world are mapped into the domain object model.
 - *Analysis object model*. The analysis object model presents how the source code (implementation) should be carried out and written.
 - *Implementation model*. The implementation model represents the implementation of the system.
 - *Test model*. The test model constitutes the test plans, specifications, and reports.



Object-Oriented Business Engineering

- Object-oriented business engineering (OOBE) is object modeling at the enterprise level.
- Use cases again are the central vehicle for modeling, providing traceability throughout the software engineering processes.



Object-Oriented Business Engineering

- ***Analysis phase:***
 - The analysis phase defines the system to be built in terms of the problem-domain object model, the requirements model, and the analysis model.
 - The analysis process should not take into account the actual implementation environment.
 - This reduces complexity and promotes maintainability over the life of the system, since the description of the system will be independent of hardware and software requirements.
 - The analysis process is iterative but the requirements and analysis models should be stable before moving on to subsequent models.
 - Jacobson et al. suggest that prototyping with a tool might be useful during this phase to help specify user interfaces.



Object-Oriented Business Engineering

- ***Design and implementation phases.***
 - The implementation environment must be identified for the design model.
 - This includes factors such as Database Management System (DBMS), distribution of process, constraints due to the programming language, available component libraries, and incorporation of graphical user interface tools.
 - It may be possible to identify the implementation environment concurrently with analysis.
 - The analysis objects are translated into design objects that fit the current implementation environment.



Object-Oriented Business Engineering

- ***Testing phase.***
 - Finally, Jacobson describes several testing levels and techniques.
 - The levels include unit testing, integration testing, and system testing.



Patterns

- An emerging idea in systems development is that the process can be improved significantly if a system can be analyzed, designed, and built from prefabricated and predefined system components.
- One of the first things that any science or engineering discipline must have is a vocabulary for expressing its concepts and a language for relating them to each other.
- Therefore, we need a body of literature to help software developers resolve commonly encountered, difficult problems and a vocabulary for communicating insight and experience about these problems and their solutions.

Patterns

- The use of design patterns originates in the work done by a building architect named Christopher Alexander during the late 1970s.
- *A Timeless Way of Building*, that, in addition to giving examples, described his rationale for documenting patterns.
- The main idea behind using patterns is to provide documentation to help categorize and communicate about solutions to recurring problems.
- The pattern has a name to facilitate discussion and the information it represents.



Pattern

- A ***pattern*** is [an] instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces.
- A pattern involves a general description of a solution to a recurring problem bundle with various goals and constraints.
- But a pattern does more than just identify a solution, it also explains why the solution is needed.



Pattern

- The documentation of a pattern, in essence, provides the contexts under which it is suitable and the constraints and forces that may affect a solution or its consequences.
- Communication about patterns is enabled by a vocabulary that describes the pattern and its related components such as name, context, motivation, and solution.
- By classifying these components and their nature (such as the structural or behavioral nature of the solution), we can categorize patterns.

Pattern

- A good pattern will do the following:
 - *It solves a problem.* Patterns capture solutions, not just abstract principles or strategies.
 - *It is a proven concept.* Patterns capture solutions with a track record, not theories or speculation.
 - *The solution is not obvious.* The best patterns generate a solution to a problem indirectly—a necessary approach for the most difficult problems of design.



Pattern

- *It describes a relationship.* Patterns do not just describe modules, but describe deeper system structures and mechanisms.
- *The pattern has a significant human component.* All software serves human comfort or quality-of life; the best patterns explicitly appeal to aesthetics and utility.



Generative and Nongenerative Patterns

- Generative patterns are patterns that not only describe a recurring problem, they can tell us how to generate something and can be observed in the resulting system architectures they helped shape.
- Nongenerative patterns are static and passive: They describe recurring phenomena without necessarily saying how to reproduce them.

Patterns Template

- Currently, several different pattern templates have been defined that eventually will represent a pattern.
- Despite this, it is generally agreed that a pattern should contain certain essential components.
 - *Name*. A meaningful name.
 - *Problem*. A statement of the problem that describes its intent
 - *Context*. The *preconditions* under which the problem and its solution seem to re-cur and for which the solution is desirable.

Patterns Template

- *Forces*. A description of the relevant forces and constraints and how they interact or conflict with one another and with the goals we wish to achieve (perhaps with some indication of their priorities).
- *Solution*. Static relationships and dynamic rules describing how to realize the desired outcome.
- *Examples*. One or more sample applications of the pattern that illustrate a specific initial context; how the pattern is applied to and transforms that context; and the resulting context left in its wake.

Patterns Template

- • *Resulting context*. The state or configuration of the system after the pattern has been applied, including the consequences (both good and bad) of applying the pattern, and other problems and patterns that may arise from the new context. It describes the *postconditions* and side effects of the pattern.
- *Rationale*. A justifying explanation of steps or rules in the pattern and also of the pattern as a whole in terms of how and why it resolves its forces in a particular way to be in alignment with desired goals, principles, and philosophies.

Patterns Template

- *Related patterns.* The static and dynamic relationships between this pattern and others within the same pattern language or system.
- *Known uses.* The known occurrences of the pattern and its application within existing systems.

Antipatterns

- A pattern represents a "best practice," whereas an antipattern represents "worst practice" or a "lesson learned."
- Antipatterns come in two varieties:
 - Those describing a bad solution to a problem that resulted in a bad situation.
 - Those describing how to get out of a bad situation and how to proceed from there to a good solution.

Antipatterns

- Antipatterns are valuable because often it is just as important to see and understand bad solutions as to see and understand good ones.

Capturing Patterns

- Writing good patterns is very difficult, explains Appleton.
- Patterns should provide not only facts (like a reference manual or users' guide) but also tell a story that captures the experience they are trying to convey.
- A pattern should help its users comprehend existing systems, customize systems to fit user needs, and construct new systems.
- The process of looking for patterns to document is called pattern mining (or sometimes reverse architecting).

Capturing Patterns

- Guidelines are summarized from Buschmann et al.
- *Focus on practicability.*
 - Patterns should describe proven solutions to recurring problems rather than the latest scientific results.
- *Aggressive disregard of originality.*
 - Pattern writers do *not* need to be the original inventor or discoverer of the solutions that they document.
- *Nonanonymous review.*
 - Pattern submissions are shepherded rather than reviewed. The shepherd contacts the pattern author(s) and discusses with him or her how the patterns might be clarified or improved on.

Capturing Patterns

- *Writer's workshops instead of presentations.*
 - Rather than being presented by the individual authors, the patterns are discussed in writers' workshops, open forums where all attending seek to improve the patterns presented by discussing what they like about them and the areas in which they are lacking.
- *Careful editing.*
 - The pattern authors should have the opportunity to incorporate all the comments and insights during the shepherding and writers' workshops before presenting the patterns in their finished form.

Frameworks

- A **framework** is a way of presenting a generic solution to a problem that can be applied to all levels in a development .However, design and software frameworks are the most popular.A definition of an object-oriented software framework is given by Gamma et al:
- A framework is a set of cooperating classes that make up a reusable design for a specific class of software.
- A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations.



Frameworks

- A developer customizes a framework to a particular application by subclassing and composing instances of framework classes.
- The framework captures the design decisions that are common to its application domain. Frameworks thus emphasize design reuse over code reuse, though a framework will usually include concrete subclasses you can put to work immediately.

Frameworks

- Gamma et al. describe the major differences between design patterns and frameworks as follows:
- *Design patterns are more abstract than frameworks.*
 - Frameworks can be embodied in code, but only examples of patterns can be embodied in code.
 - A strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly.

Frameworks

- *Design patterns are smaller architectural elements than frameworks.*
 - A typical framework contains several design patterns but the reverse is never true.
- *Design patterns are less specialized than frameworks.*
 - Frameworks always have a particular application domain. In contrast, design patterns can be used in nearly any kind of application.
 - While more specialized design patterns are certainly possible, even these would not dictate an application architecture.

The Unified Approach

- The unified approach (UA) establishes a unifying and unitary framework around their works by utilizing the unified modeling language (UML) to describe, model, and document the software development process.
- The idea behind the UA is not to introduce yet another methodology.
- The main motivation here is to combine the best practices, processes, methodologies, and guidelines along with UML notations and diagrams for better understanding object-oriented concepts and system development.

The Unified Approach

- The unified approach to software development revolves around (but is not limited to) the following processes and concepts :
- The processes are:
 - Use-case driven development
 - Object-oriented analysis
 - Object-oriented design
 - Incremental development and prototyping
 - Continuous testing

The Unified Approach

- The methods and technology employed include
 - Unified modeling language used for modeling.
 - Layered approach.
 - Repository for object-oriented system development patterns and frameworks.
 - Component-based development (Although, UA promote component-based development, the treatment of the subject is beyond the scope of the book.)



The Unified Approach

- The UA allows iterative development by allowing you to go back and forth between the design and the modeling or analysis phases.
- It makes backtracking very easy and departs from the linear waterfall process, which allows no form of backtracking.



Object-Oriented Analysis

- Analysis is the process of extracting the needs of a system and what the system must do to satisfy the users' requirements.
- The goal of object-oriented analysis is to first understand the domain of the problem and the system's responsibilities by understanding how the users use or will use the system.

Object-Oriented Analysis

- OOA Process consists of the following Steps:
 1. Identify the Actors.
 2. Develop a simple business process model using UML Activity diagram.
 3. Develop the Use Case.
 4. Develop interaction diagrams.
 5. Identify classes.



Object-Oriented Design

- Booch provides the most comprehensive object-oriented design method.
- Ironically, since it is so comprehensive, the method can be somewhat imposing to learn and especially tricky to figure out where to start.
- Rumbaugh et al.'s and Jacobson et al.'s high-level models provide good avenues for getting started.
- UA combines these by utilizing Jacobson et al.'s analysis and interaction diagrams, Booch's object diagrams, and Rumbaugh et al.'s domain models.


Object-Oriented Design

- OOD Process consists of:
 - Designing classes, their attributes, methods, associations, structures and protocols, apply design axioms
 - Design the Access Layer
 - Design and prototype User interface
 - User Satisfaction and Usability Tests based on the Usage/Use Cases
 - Iterate and refine the design



Iterative Development and Continuous Testing

- You must iterate and reiterate until, eventually, you are satisfied with the system.
- Since testing often uncovers design weaknesses or at least provides additional information you will want to use, repeat the entire process, taking what you have learned and reworking your design or moving on to reprototyping and retesting.
- Continue this refining cycle through the development process until you are satisfied with the results.
- During this iterative process, your prototypes will be incrementally transformed into the actual application.



Modeling Based on the Unified Modeling Language

- The unified modeling language was developed by the joint efforts of the leading object technologists Grady Booch, Ivar Jacobson, and James Rumbaugh with contributions from many others.
- The UML merges the best of the notations used by the three most popular analysis and design methodologies: Booch's methodology, Jacobson et al.'s use case, and Rumbaugh et al.'s object modeling technique.



Modeling Based on the Unified Modeling Language

- The UML is becoming the universal language for modeling systems; it is intended to be used to express models of many different kinds and purposes, just as a programming language or a natural language can be used in many different ways.

The UA Proposed Repository

- In modern businesses, best practice sharing is a way to ensure that solutions to process and organization problems in one part of the business are communicated to other parts where similar problems occur.
- The idea promoted here is to create a repository that allows the maximum reuse of previous experience and previously defined objects, patterns, frameworks, and user interfaces in an easily accessible manner with a completely available and easily utilized format.

The UA Proposed Repository

- Everything from the original user request to maintenance of the project as it goes to production should be kept in the repository.
- The advantage of repositories is that, if your organization has done projects in the past, objects in the repositories from those projects might be useful.
- You can select any piece from a repository—from the definition of one data element, to a diagram, all its symbols, and all their dependent definitions, to entries—for reuse.

The UA Proposed Repository

- The same arguments can be made about patterns and frameworks.
- Specifications of the software components, describing the behavior of the component and how it should be used, are registered in the repository for future reuse by teams of developers.
- The repository should be accessible to many people.
- Furthermore, it should be relatively easy to search the repository for classes based on their attributes, methods, or other characteristics.

The Layered Approach to Software Development

- A better approach to systems architecture is one that isolates the functions of the interface from the functions of the business.
- This approach also isolates the business from the details of the data access.
- Using the three layered approach, you are able to create objects that represent tangible elements of your business yet are completely independent of how they are represented to the user (through an interface) or how they are physically stored (in a database).
- The three-layered approach consists of a view or user interface layer, a business layer, and an access layer

The Business Layer

- The business layer contains all the objects that represent the business (both data and behavior).
- This is where the real objects such as Order, Customer, Line item, Inventory, and Invoice exist.
- Most modern object-oriented analysis and design methodologies are generated toward identifying these kinds of objects.
- The responsibilities of the business layer are very straightforward: Model the objects of the business and how they interact to accomplish the business processes.

The Business Layer

- When creating the business layer, however, it is important to keep in mind a couple of things.
- These objects should not be responsible for the following:
 - *Displaying details.* Business objects should have no special knowledge of how they are being displayed and by whom.
 - *Data access details.* Business objects also should have no special knowledge of "where they come from." It does not matter to the business model whether the data are stored and retrieved via SQL or file I/O. The business objects need to know only to whom to talk about being stored or retrieved.

The User Interface (View) Layer

- The user interface layer consists of objects with which the user interacts as well as the objects needed to manage or control the interface.
- The user interface layer also is called the *view layer*.
- This layer typically is responsible for two major aspects of the applications:
 - *Responding to user interaction.* The user interface layer objects must be designed to translate actions by the user, such as clicking on a button or selecting from a menu, into an appropriate response
 - *Displaying business objects.* This layer must paint the best possible picture of the business objects for the user.

The Access Layer

- The access layer contains objects that know how to communicate with the place where the data actually reside, whether it be a relational database, mainframe, Internet, or file.
- Regardless of where the data actually reside, the access layer has two major responsibilities:
 - *Translate request.* The access layer must be able to translate any data-related requests from the business layer into the appropriate protocol for data access
 - *Translate results.* The access layer also must be able to translate the data retrieved back into the appropriate business objects and pass those objects back up into the business layer.