Information Engineering *

Steven A. Demurjian
Computer Science and Engineering Department
The University of Connecticut
Storrs, Connecticut 06269-2155

January 27, 2008

Contents

What is information engineering? Why is it so important? How can it be successfully used to solve problems of today and tomorrow? What is object-oriented design? Where and how do object-oriented analyses fit into the overall design and development process? How is the object-oriented paradigm related to information engineering? What are the key features of the paradigm for supporting information engineering? What is unique about our approach to information engineering? What role will a database play in application/system development and for information engineering? What are future problems? Can we use information engineering with object-oriented design and analyses as a solution?

All of the above are excellent motivational questions, that collectively provide a rather ambitious set of requirements for the information engineering process. The purpose of this initial chapter is to set the context and tone for the book, by providing a perspective on our philosophy, and why this philosophy is important to solving both today's and tomorrow's problems. There is no way that we can answer all of those questions in a single chapter. There is also the possibility that a single book could not do them justice! However, we do hope to provide answers to a set of fundamental questions, namely: What is information engineering? What's available to support information engineering? What is the object-oriented paradigm? Why should the object-oriented paradigm be chosen for information engineering? What is necessary and sufficient for information engineering? Thus, we begin to meet, in part, the goals outlined in the preface.

The remainder of this chapter contains six major sections, the first five of which are titled with questions, that provide the motivation, background, and context for the remainder of the book. First, we investigate the definition of information engineering, presenting material and rationale that transcends the previous discussion in the preface. Next, we examine the available approaches for information engineering, as a step in a progression that leads to sections on the basic concepts of the paradigm and its advantages for information engineering. Then, we provide a framework of the necessary and sufficient characteristics that are critical for information engineering. Finally, to serve as a consistent theme throughout the book, we provide a summary section that reflects on the goals, purpose, and content of the chapter. This section also includes, when appropriate, a discussion of further readings and/or related research efforts.

1 What is Information Engineering?

To understand our definition for information engineering, which in turn, provides the foundational background on the central theme that underlies our approach, we take the reader through a progression that mirrored our own investigatory process. There are four parts to the progression. First, we investigate information engineering from a perspective that examines the individual concepts (information and engineering) to reconcile contradictions in terminology and intent. Next, using this material and discussion, we define *information engineering* and provide a further examination of its motivation, importance, advantages, and overall approach. Then, we detail the indispensable role of a database in the information engineering process, which upgrades an often overlooked aspect of the application-design process to a first-class citizen. Lastly, we introduce the umbrella concept of *information consistency*, a successor of the traditional data consistency concept, to provide a multi-dimensional perspective for viewing and dealing with information.

1.1 Information and Engineering

The terms data, information, and knowledge are prevalent in all areas of computer science and engineering, sometimes being used interchangeably, and other times being used as clearly different concepts. When we use the term *information*, what do we mean? Clearly, we have replaced "software" engineering, but how does information compare to both data and knowledge? To provide one perspective, consider the following definitions of the three terms, taken from the *American Heritage* [1] and *Webster's Ninth New Collegiate* [29] dictionaries, respectively:

Data 1. Information, esp. information organized for analysis or used as the basis for a decision. **2.** Numerical information in a form suitable for processing by computer.

Data : factual information (as measurements or statistics) used as a basis for reasoning, discussion, or calculation.

Information 1. The act of informing or the condition of being informed; communication of knowledge. **2.** Knowledge derived from study, experience, or instruction. **3.** Knowledge of a specific event or situation; news. **4.** *Law.* A formal accusation of a crime made by a public officer rather than by indictment by a grand jury. **5.** A nonaccidental signal used as an input to a computer or communications system. **6.** A numerical measure of the certainty of an experimental outcome.

Information 1: The communication or reception of knowledge or intelligence. **2 a (1):** knowledge obtained from investigation, study, or instruction **(2):** INTELLIGENCE, NEWS **(3):** FACTS, DATA **b:** the attribute inherent in and communicated by one of two or more alternative sequences or arrangements of something (as nucleotides in DNA or binary digits in a computer program) that produce specific effects **c (1):** a signal or character (as in a communication system or computer) representing data **(2):** something (as a message, experimental data, or a picture) which justifies change in a construct (as a plan or theory) that represents physical or mental experience or another construct **d:** a quantitative measure of the content of information; *specif:* a numerical quantity that measures the uncertainty in the outcome of an experiment to be performed **3:** the act of informing against a person **4:** a formal accusation of a crime made by a prosecuting officer as distinguished from an indictment presented by a grand jury.

Knowledge 1. The state or fact of knowing. **2.** Familiarity, awareness, or understanding gained through experience or study. **3.** The sum or range of what has been perceived, discovered, or learned. **4.** Learning; erudition: *men of knowledge.* **5.** Specific information about something.

Knowledge 1 *obs*: COGNIZANCE **2 a (1):** the fact or condition of knowing something with familiarity gained through experience or association **(2):** acquaintance with or understanding of a science, art, or technique **b (1):** the fact or condition of being aware of something **(2):** the range of one's information or understanding **c:** the circumstance or condition of apprehending truth or fact: COGNITION **d:** the fact or condition of having information or of being learned **3:** the sum of what is known: the body of truth, information, and principles acquired by mankind.

There are both overlaps and conflicts in the different definitions. Clearly, the spirit of each term's definition from corresponding dictionaries is similar; but, what is more interesting is that their conflicts are also similar. Both sets of definitions define data using the term information, information using the term knowledge, and knowledge using the term information. This cross referencing is troubling and points to a general lack of precision in defining each term. Moreover, the circularity between information and knowledge points towards the possibility that the two terms are equivalent, which will definitely cause disagreement within a room full of computer scientists and engineers.

In fact, in the forewords of the first issue of *IEEE Transactions on Knowledge and Data Engineering* [11], the issue of knowledge versus data was also discussed, without a resolution with respect to definition, intent, and usage. Thus, we are left with three terms that, while clearly different, are also dependent on one another. We have chosen information since it seems to occupy a middle ground. Data is closely associated with low-level processing and database systems; while knowledge is heavily utilized when referring to data (or information) that is semantically inferred, as is the case in many artificial intelligence applications. In our minds, information is the unifying concept, and is also in some sense a term that supersedes software, since without information (data/knowledge), software is not meaningful (i.e., software refers to a process and product that deals with information).

While there is clearly confusion over the previous three terms, there is less of a problem in differentiating between science and engineering, as seen in the following two sets of definitions, taken once again from *American Heritage* [1] and *Webster's* [29]:

Science 1. a. The observation, identification, description, experimental investigation, and theoretical explanation of natural phenomena. **b.** Such activity restricted to a class of natural phenomena. **c.** Such activity applied to any class of phenomena. **2.** Methodological activity, discipline, or study. **3.** An activity that appears to require study and method. **4.** Knowledge, esp. knowledge gained through experience.

Science 1: The state of knowing: knowledge as distinguished from ignorance or misunderstanding **2 a:** a department of systemized knowledge as an object of study **b:** something (as a sport or technique) that may be studied or learned like systematized knowledge **c:** one of the natural sciences **3 a:** knowledge covering general truths or the operation of general laws esp. as obtained and tested through scientific methods **b:** such knowledge concerned with the physical world and its phenomena: NATURAL SCIENCE **4:** a system or method reconciling practical ends with scientific laws.

Engineering 1. The application of scientific and mathematical principles to practical ends such as the design, construction, and operation of efficient and economical structures, equipment, and systems. **2.** The profession of or the work performed by an engineer.

Engineering 1: The activities or functions of an engineer: as **a:** the art of managing engines **b:** calculated manipulation or direction (as of behavior) **2:** the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to people in structures, machines, products, systems, and processes.

It is arguable that the two terms of science and engineering are complementary, since engineering methods and tasks must have a solid scientific or mathematical basis, while without engineering, scientific theory is not easily tested. However, within the computing field, there is an active debate regarding the appropriate name of the discipline, with different educational and research leaders taking conflicting positions [9, 18]. Computer science, computer engineering, computer science and engineering, information science (engineering), computer and information science, and so on, have all been offered as the "best" way to describe the field.

While we make no claims regarding the best way to characterize computing, in a specific sense, this issue has a great impact on our choice of information engineering rather than software engineering. Why is this the case? Basically, and controversially, we believe that software engineering is an oxymoron. When one examines what typically is referred to as software engineering, it is our opinion that it is difficult, if not impossible, to identify any rigorous engineering actions and techniques (at least from the perspective of the more traditional engineering disciplines). Software engineering is often presented as a collection of disparate techniques (e.g., data-flow diagrams, ER diagrams, finite state machines, petri nets, etc.), that tend to function as stand-alone methods for designing a solution. None of these techniques offer any idea to the designer as to when a "complete" or "final" state has been reached. Moreover, with the exception of program-proving techniques, there is a lack of analyses against a design to understand what the design represents and to determine if the design is complete and correct.

However, this is not the case in other engineering disciplines. For example, a civil engineer when designing a bridge cannot simply assemble different components haphazardly, but must carefully specify individual spans of a bridge, including the sizes, dimensions, weights, and so on. This information is carefully analyzed via mathematical techniques to engineer the structural integrity of the bridge. A civil engineer would never put together a set of components (e.g., a set of entities and relationships in an ER diagram), examine them superficially, and then decide that they look okay and seem to be a reasonable solution. However, that is what most software engineers do with an ER diagram or other techniques. Our position is that this is not acceptable, and we must upgrade the discipline to one that has a true basis in engineering rigor and discipline.

1.2 Definition and Motivation of Information Engineering

The definition of *information engineering* must be general enough to indicate its utility across a wide range of diverse application domains, yet specific enough to convey a set of concrete concepts that, when integrated, can provide a basis for comprehensive design and development. We offer the following definition:

Information Engineering is the incorporation of an engineering approach and **discipline** to the **generation** of information and the **promotion of the better use** of information and resources.

The key concepts of this definition, as highlighted, involve the essential concepts of information engineering. Discipline, an important part of the engineering process, makes available a set of analysis techniques that can be used to evaluate a design of an application for precision and accuracy, with respect to the specification, and may also provide a basis for attaining completeness. A civil engineer follows a rigorous discipline when designing a bridge, and a significant part of his/her actions involve analyses that are applied throughout the design process. Thus, this uniform application of discipline emphasizes cohesion during the information engineering process. Generation dictates that an application that is engineered often creates or synthesizes (aggregates) information that is then viewed and utilized by end-users or the application's tools that they use. A designed 'bridge' is not presented in its blueprint form to all interested parties. Instead, different representations such as a scale model (for the customer), materials (for the contractor/suppliers), and wire-frame or solid model CAD (for upstream/downstream engineers), are all necessary. Together, discipline and generation promote the better use of information for an application. As previously described, the actions of civil engineers represents only a small (but, important) portion of the engineering process; but their actions clearly dictate the overall design/development lifecycle that governs the transition from the proposal of a new bridge to its opening.

As a concept, information engineering is intended to unify and combine the different requirements that must be engineered in any complex system or application. This includes requirements for a database (data engineering), for insuring controlled access (security engineering), and for binding all application components into a single system (software engineering). Data engineering focuses on the required *information* (input and/or output) for a given application, to meet the needs of software engineers (who 'build' the system) and users (who utilize the system). Software engineering refers to the organized process of producing a software application, from the original idea to the final deliverable product. Software engineers utilize the data engineering results (information), and apply methodologies in the design and construction of an application. Security engineering refers to the access of information, both by the software engineers and the end user, clearly defining what each individual can do with what information at which times. The key issue is that applications cannot and must not be engineered in a vacuum.

Information engineering is motivated by a number of factors. First, management and control of information will be a primary concern as we continue through the 1990s and progress into the 21st century. We are clearly in an age of information, with the need to deal with its large volumes (gigabytes or even more), its diverse types (ranging from text to structured to unformatted to graphical and so on), and its complex interdependencies (among different "chunks" of information that must be maintained as changes are made across an application). Second, a variety of critical systems, that span a spectrum, heavily depend on information. Consider the following examples and their descriptions:

- Health Care Systems: At the forefront of American society as national health care is proposed and discussed. Moreover, initiatives in Europe also seek to dictate the requirements for medical and health care information systems.
- CAD/CAM/CIM: A critical need as American society attempts to diversify from a DoD-based manufacturing sector.
- Air Traffic Control: Commissioned in recent years by the FAA in the U.S. to replace an aging and outdated system.

- Airline/Hotel/Auto Reservations: The continued unification of individual systems to more readily support business and tourist travel.
- Telecommunications: High-definition and cable television along with fiber optics will offer services that were once only available in the writings of science fiction authors.
- Banking/ATMs/Credit and Debit Cards: The continued proliferation of computerization into our society, where groceries and gasoline can be paid for with more and more ease.
- Biotechnology/Bioengineering: An emerging field that offers much promise in the future across many of the aforementioned systems.
- Computing: The hardware and, more importantly, software aspects of the computing industry must also be improved in order to react and support the needs of the aforementioned application domains.

We must not only support these applications as they exist today, but must be prepared to evolve and extend their capabilities in the coming years to meet ever-changing needs and requirements. However, as these traditional boundaries continue to disappear, we must increase care regarding the confidentiality and sensitivity of information. For example, credit bureaus today electronically collect information on all consumers from many diverse sources; if this information is not correct, the potential impact is wide-ranging, and could result in consumers being unfairly denied credit. Also, consider that an innocent visit to the doctor for a cold when coupled with an error on your medical record (via a misdiagnosis or an invalid test result) could impact on your insurability when trying to qualify for life insurance. More recently, in the spring of 1993, in Manchester, Connecticut, there was the nationally celebrated ATM fraud, where a group of individuals posing as bank employees installed an ATM machine in the middle of a busy shopping mall. They utilized modified software in the ATM to record both account and PIN numbers, and subsequently used that information to fabricate new ATM cards, which were then used to illegally access funds. While credit-card fraud is not new, this case also involves confidentiality issues.

Therefore, it becomes even more apparent, that information engineering is vitaly important. On the one hand, the timely and efficient utilization of information must be ensured since this significantly impacts on productivity, will play a major role if industry is to move successfully into the future, can support and promote collaboration as the key to maintaining a competitive advantage, and will allow individuals and companies to use information in new and different ways. On the other hand, the collection, synthesis, analyses of information can lead to: a better understanding of processes, sales, productivity, etc.; and, the dissemination of only relevant and/or significant information, which can effectively reduce the overload of information on end-users. The underlying theme for both of these perspectives and our previous examples is that confidentiality must be insured and misuse must be both prohibited and prevented. Moreover, it should be clear that information engineering must span the entire business process, and not be limited to the design and development of an application.

While we have spent many paragraphs dealing with the definition and motivation of information engineering, a remaining factor must focus on the actual *engineering of information*, which is envisioned in a five phase process:

- Careful Thought to its Definition: The purpose of information in the application must be clearly delineated and understood. For example, the civil engineer must thoroughly discuss the intended project (bridge) with the customer to clearly identify needs and requirements.
- Thorough Understanding of its Intended Usage and Potential Impact: The justification and demonstration of our approach requires the garnering of "real-world" problems from industry. In the bridge example, parameters such as the number of lanes, expected traffic patters, weight limitations for trucks, tolls, impact on aquatic life, etc., must all be addressed.

- Insure and Maintain its Consistency: To effectively deal with information, we must be sure of its quality, correctness, and relevance. Once again, for the bridge, the impact of changes in requirements must be constantly evaluated, e.g., if the span length is changed, what is the impact on the weight limitations?

- Protect and Control its Availability: Who uses information? When is it used? and Where is it used? all deal with the ability to increase productivity while still insuring confidentiality and limiting misuse. During the bidding process for a transportation project such as a bridge, information exchange between contractors and design engineers are needed to provide details on requirements. However, it is also expected that any information on a particular bid is not inadvertently (or intensionally) made available to other contractors. In addition, changes to the requirements must be conveyed to all contractors in a timely fashion.

- Long-Term Persistent Storage/Recoverability: In this case, the cost to maintain information so that it may be reused is important, since through longitudinal and cumulative experience we can learn how to more effectively use information in the future. While the usefulness of persistence is not automatically clear for a bridge, in fact, historical data on 'old' designs and 'successful past' bids are critical to promote reuse to streamline the overall process.

To support these five phases, we must adopt a design and development approach for an application that focuses on information and its behavior. This is most easily quantified via the following list of design-level questions. An example using software-development environments (SDEs) is employed to illustrate the utility of each question.

∗ What are different kinds or types of information for an application?

∗ Is the "same" information stored in different ways?

In SDEs, for a given module we might maintain multiple representations for the source code (ascii file), the object code, a parse tree, a symbol table, and a data-flow graph. All of these representations offer different perspectives of a portion of a program that are important and provide different views to different users.

∗ How will the types of information be manipulated to maximize usage?

Actions or operations on source code (e.g., edit, compile, etc.) are different than ones for a parse tree (e.g., create node, scan node, get token, etc.). Information engineering requires the definition of specific manipulation techniques for all of the different representations of an application.

∗ What are the information interdependencies?

In SDEs, a given source file must be bound to its object code, parse tree, symbol table, and data-flow graph. Moreover, consistency across representations is needed to insure that a change to one representation is realized in all other appropriate representations.

∗ Are there any performance requirements related to the access and/or use of information?

SDEs are dynamic, with respect to the constant changes being made to the source code. In order to insure that all other representations (parse tree, symbol table, etc.) are consistent whenever the source is changed. Such an activity is likely computationally intensive, and if so, there would be response-time requirements regardings its completion so that the user interface performance does not suffer.

∗ What are the allowable values for the different types of information?

From a programming perspective, allowable values involves typing; from a database perspective, integrity constraints. Both are critical for SDEs. In the latter, constraints might be simple (e.g., object code corresponds to the most recent version of the source) or complex (e.g., changes in the source must be realized in all relevant representations).

* Will information persist over time?

  In SDEs, to support incremental design and development, the different representations must be stored in a long-term repository like a database. In addition, to promote reuse, a software or information engineer (IE) must be able to access "old" software to use in "new" designs/implementations.

* Are multiple versions of information available?

  This question transcends traditional approaches via source code control systems, since it must offer new and different capabilities that can take advantage of longitudinal data that are not simply ascii files. Versions of all information associated with design and development in SDEs must be recorded, thereby allowing the restoration of a project to an earlier (and hopefully, consistent,) time-point.

* Who needs access to what information when?

  In SDEs, there are many different types of IEs (e.g., IEs for project management, design, development, testing, maintenance, or IEs new to a company or project). All of these IEs need different access at different times to the representations that comprise an application in an SDE (e.g., IEs for development need write access to modules, while IEs that are new to a company may only need read access initially). Further, their access might change over time (e.g., once a new IE has improved his/her skills, changes to privileges that yield wider access can be made).

* Is some information sensitive?

* Must some information be protected?

  The last two questions are related, since once information has been identified as sensitive (e.g., employee evaluations of IEs by a project manager), it must be protected from access by all unauthorized individuals.

Overall, these eleven questions form a solid framework for an initial exploration of the information engineering requirements for an application. Exercise 6 at the end of this chapter involves the exploration of a analogous set of answers for bridge design and other applications.

1.3 The Important Role of a Database

In the previous section, the need for persistence was clearly introduced, since engineering information and its availability are critical to long-term development issues. Persistence deals with the ability to archive or store information over a period of time, and is traditionally achieved via a database system (DBS). Essentially, one of the major assumptions that we are making in our approach is that the existence of a database in future applications will be a norm rather than an exception. Databases already pervade in many application areas. For example, most CAD systems employ a database to store designs for mechanical parts. However, these same CAD systems rarely offer the ability to support simultaneous access by multiple users, i.e., they are single-user systems. That is, while there is a database, there is **not** a DBS.[1] Consequently, there are no provisions for the many features offered by a DBS such as transaction processing, concurrency control, recovery, and so on, that are critical for promoting the safe and consistent use and sharing of information. We are often provided with stories from industry about the multiple copies of designs that often exist for

[1] Note that we are not singling out CAD systems and vendors specifically; this is a problem that is true in other application domains.

a given application (like a mechanical part), where each copy is slightly different. Resolving all of these copies into a "final" or "complete" version of the design is a time-consuming and difficult task.

Clearly, from the examples of the previous section (Telecommunications, Banking, Health Care, etc.), a database must be an integral part of an application's design and development. If this is to be the case, then the usage of information by an application must be tightly linked to its storage. For some applications (like Banking), the use of a relational DBS is appropriate. However, for others (Health Care, CAD/CAM/CIM, etc.), another solution must be sought, since these applications do not fit the rigid restrictions of a relational database. As we noted for the SDE example, there is a need to support many representations of the "same" information for an application (e.g., source and object code, parse tree, etc.). While researchers have tried to use a relational DBS to store such representations [10, 15], even in a strictly read-only system (which is unrealistic), the performance of a relational DBS significantly degrades when attempting to reconstruct complex data such as parse trees and data-flow graphs via join operations across multiple relations. Instead, the database and DBS that is an integral part of future applications must promote retrieval of information that is geared towards user responsibilities, providing information to users in a form that suits their needs. To do so, we must bridge the gap that currently exists between standalone programming applications and database systems.

To understand this gap or *impedance mismatch*, it is important to understand the similarities and differences of a DBS and a programming language system (PLS), which refers to the set of tools (OS, compiler, libraries, debugger, etc.) that are used to support a particular programming language. At the system level, a PLS and a DBS are very similar, since the PLS must support shared resources (e.g., file system, I/O devices, etc.) while the DBS allows shared data (e.g., simultaneous access of the same data with consistency insured). However, the major differences are at a granularity level; a PLS deals with programs and files, while a DBS uses transactions and instances, respectively. To adequately support the database needs for information engineering, these differences must be reconciled!

To understand these differences, consider Figure 1.1, where two implementation views of the overlap between PLSs and DBSs are shown. In yesterday's view (which is still available and widely used today), interaction between a PLS and DBS is supported via embedded query languages. For example, in Ingres [25],

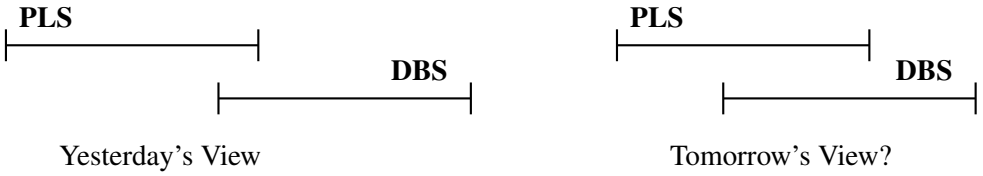


Figure 1.1: The Interplay Between a PLS and a DBS.

`Quel` query language statements are embedded into `C` files using a `##` to indicate these special statements. These statements are then preprocessed using `equel` into `C` function calls to appropriate Ingres database system routines. After preprocessing, the `C` compiler is then invoked to create the running application. However, to support information interchange, data from Ingres must be loaded into `C` structures before `C` program statements (e.g., conditionals, loops, function calls, etc.) can work on the information from the database. This was referred to earlier as an impedance mismatch. But, as noted in the other view of Figure 1.1, the overlap between a PLS and a DBS is currently undergoing an evolution, due to the advent of object-oriented database systems. At this time, it is unclear what the breadth of the eventual overlap will be; but intuition leads us to believe that there will still be dedicated and unique responsibilities that remain in each system. Note that we will discuss these issues in more depth in Section 7 of Chapter 2.

1.4 Information Consistency and Open-Ended Design

In the past, when an application was interfaced with a DBS, one important benefit was the attainment and guarantee of data consistency, thereby allowing simultaneous access to shared data by multiple users. Since we have upgraded to information engineering, then *information consistency* must be supported. The realization of information consistency is a result of the addition of engineering-like analyses for the design and development of software applications, thereby providing automatic and on-demand feedback as an application is created over time. As an initial, starting definition for this book, information consistency is gauged with respect to the following four characteristics:

- **Usage of Information:** Involves the different ways that an application will use information to its advantage, in a form that suits and supports its tasks and functionalities. In this context, analyses are utilized to insure that actions taken by the designer to formulate the application are consistent and in sync with the specification of the intended domain.
- **Persistency of Information:** The characteristic that information must be long-lived, so that it can be more effectively utilized and reused. Analyses is also critical with respect to persistence, since the definition and usage of persistent information must be consistent with all other relevant system data, which only exists during the run-time environment of the application.
- **Integrity and Security of Information:** Integrity focuses on insuring that information has correct values, and is often achieved via typing in a PLS and integrity constraints in a DBS. Security deals with who can use what information when, and is an aspect of information consistency, since by enforcing security, misuse is reduced (eliminated?). There is a definite need for analyses, to insure that correct information and access are provided. Both may fall victim to problems since often indirect information or inferred access may cause undesired results.
- **Validity or Relevance of Information:** This is the hardest aspect of information consistency to quantify, since it requires that the information that is produced by an application have a semantic meaning or usefulness to end-users. This is very difficult to achieve. Analyses in this situation is nearly impossible to define in any useful context, since it is difficult to analyze if one has the 'correct' information for an application.

All of these characteristics involve how information is used from different perspectives. While we have discussed what information consistency is and why it is important, the more significant purpose of the book to examine, via our approach, how information consistency can be attained.

Even if we acknowledge that information consistency is important for information engineering, in practice, we can never guarantee that it is totally and completely realized. This is mainly due to one indisputable fact: the design process (in software engineering or information engineering) is open-ended with a final completion point or state that is impossible to identify. This concept has strong ties to the bridge example. First, even if construction is ongoing, changes to the bridge's design can occur. After construction has been completed, maintenance, improvements, etc., are likely. 'Open-ended' refers to both situations; there is not an identifyable, discrete time point to indicate if or when the design is finished. No matter how 'finished' it may appear, there is always the chance that the bridge will need design modifications if changes to the requirements occur.

To feasibly promote design in information engineering, we must support a process that is cyclical, incremental, and iterative. Through this process, as subsequent increments are defined, one often assumes that each later increment is more correct and complete than its immediate predecessor. However, traditional software engineering offers us no formal way to know that this is in fact the case.[2] Thus, while there may

[2]We are assuming that program-proving and pre and post conditions, while offering some degree of completeness and verification, break down with large systems that involve tens of millions of lines of code and hundreds of individuals, across multiple companies that are geographically separated.

not be a formal basis to support this assumption, if we value and strive for information consistency, it may be possible to at least attain the confidence that the "final" design is comprehensive and represents the specification. For this to occur, the database must be an indispensable component and security must be an integral feature, as we have indicated in Section 1.3 and will discuss in more detail in later chapters. In addition, our approach will propose a framework of analyses that augments the design or engineering process of an application. Like a civil engineer evaluating the design of a bridge, an IE can use analyses to evaluate the design of an application. Such analyses must be clear and easy to use, or else there is the potential for them to be ignored at the expense of information consistency and poor application design and development. Analyses improves a design by offering feedback to the IE on the semantic structure and interdependencies of an application. Through Part II of the book we intend to demonstrate that this results in an engineered application that has increased precision and improved accuracy, at least when compared against its specification.

2 What's Available to Support Information Engineering?

In order to support information engineering as defined to this point, we can focus on searching for an *approach* to application specification, design, development, and maintenance. The approach to the overall process must offer an end result that is: more complete, robust, and responsive; easier to understand and use; less prone to errors and misuse; and conducive to long-term maintenance and evolution, which contains, by some estimates, the bulk of the cost associated with software development [7, 22, 24]. We do not claim that this is an exhaustive list; rather it embodies the major features and characteristics that an approach or paradigm must support. The current choices that are available to support information engineering include conventional programming languages (such as `C`, `Pascal`, and `Fortran`) and traditional data models (such as relational, hierarchical, and network). All of these choices, either individually or in concert, are not suitable, for reasons discussed in Section 1.3, due to the impedance mismatch that exists. In addition, none of the choices have a tie in with a strong, language-independent design techniques such as modules or abstract data types (ADTs), which we believe is a minimal requirement for such a suitable approach to information engineering. If we focus on modules and ADTs, we are left to consider current programming languages such as `Modula-3` [4], `Ada` [2], `C++` [27], `Eiffel` [19], and so on, since they all support these concepts to varying degrees. In fact, this implies that we need to disassociate the approach from the programming language; the approach to design for supporting information engineering is the key, and given a solid design approach, its realization in a programming language can be viewed as an automatic step.

Consequently, ADTs appear to be an excellent place to continue our search, since they are a proven design approach with a long-standing history. ADTs were first proposed by Barbara Liskov for the `CLU` programming language in 1975 [16, 17]. An ADT is characterized by a set of operations (procedures and functions) that is often referred to as the *public interface*, that represents the behavior of a data type. The *private implementation* of the data type is hidden from the programmer or engineer who wishes to use the ADT. System-available ADTs have been extensively utilized in programming languages for many years. For example, in `Pascal`, when using the integer data type, the engineer is able to utilize all of the appropriate operations against integers (e.g., $+$, $-$, $*$, `div`, `mod`, etc.), without needing to know the implementation of integers in the underlying machine-dependent architecture. Designer-defined ADTs are more readily available today in languages such as `Ada`, `C++`, and `Eiffel`.

ADTs are a design technique that allow IEs to define their own data types. For example, the classic ADT is a `stack` that contains operations for `push`, `pop`, `top`, `initialize`, `is_empty`, and so on. These operations serve as the *public interface* for the IE, and typically include the type of the stack (e.g., integer, real, etc.) and the parameters and return types of the public operations. However, the *private implementation* of `stack`, that includes the implementation of all operations and the data representation (e.g., array or list, etc.) are hidden from the user. Through this division of behavior, ADTs achieve representation independence via its hidden private implementation, and abstraction via its visible public interface. Moreover, this allows implementation changes to be made (e.g., say from an array to a list) as long as these changes are transparent

to the public interface (i.e., the operations and their names, parameters, and return types cannot change).

ADT design is guided by a number of classic software engineering concepts, that are briefly reviewed for completeness:

- Separation of Concerns and Modularity: Any domain or application can be divided and decomposed into major building blocks and components. This decomposition allows the application requirements to be further defined and refined, while partitioning these requirements into a set of interacting components. Changes to the application are (hopefully) localized. In addition, team-oriented design and development can proceed with different team members concentrating on particular components.

- Abstraction and Representation Independence: These two concepts go hand in hand. Through abstraction, the details of an application's components can be hidden, providing a broad perspective on the design. This in turn allows changes to be made to the internal structure and function of each component, achieving representation independence since the external view of an ADT is not impacted.

- Incrementality/Anticipation of Change: The design process at all times is iterative or incremental. This is true whether a given set of ADTs represents an initial or final design for an application. There is an expectation that components will be changed, added, refined, etc., as needed to support evolving requirements.

- Cohesion/Coupling: An application is cohesive if each component does a single, well-defined task. Cohesion has a long history in computing. In the 'early days', the rule of thumb was that each procedure or function should be limited to one output page (approximately 60 lines). If so, then the resulting system was deemed to be cohesive. Coupling is used to signify the interactions components. Often considered the complement of cohesion, an application that minimizes coupling has components that require little or no interaction. When the application also exhibits high cohesion, the end result is a well-defined system with well-understood interactions between its components.

These terms and concepts will occur repeatly throughout the remainder of this book. Readers are referred to [7, 22, 24] for a thorough treatment of each concept.

3 What is the Object-Oriented Paradigm?

ADTs can be extended to include concepts related to object-oriented design, which are being supported today in many different object-oriented programming languages:
`Modula-3`,`C++`,`Eiffel`, `Smalltalk [8]`, `Object Pascal [28]`, etc.)
and database systems
`Ontos [21]`, `Gemstone [3]`, `O2 [6]`, `Vision [5]`, `ObjectStore [14]`, etc. This section examines the critical concepts of the paradigm through definitions and brief examples.

3.1 Object-Oriented Concepts and Terms

While there is no agreement as yet regarding terminology for the object-oriented paradigm, there are a number of core concepts that can be described:

- Object Type (OT): Used to model the features (information) and behavior (methods) for an application. Note that public interface and private implementation are as just discussed.

- Information: The private data of an OT. Information represents the different internal data components that define the OT and characterize all of its instances.

- Method: Contains the definition of the actions required for a particular operation against the private data of an OT.
- Encapsulation: The coupling of information and methods within an OT.
- Hiding: Controlling access to the information and methods of an OT.
- Inheritance: The controlled sharing of information/methods between related OTs of an application.
- Inheritance Hierarchy: All inheritance relationships between OTs that share a common parent (or grandparent) form a hierarchy with an identifiable root (ancestor).
- Instance: An occurrence of an OT, or the actual information.
- Message: An action (method call) that is initiated by an instance on itself or other instances.
- OT Library: All OTs and inheritance hierarchies for an application form a common library for use by tools and end-users.

Notice that the word information or data does not precede encapsulation or hiding in the above list. This is since in our view, it is not only data and information that is hidden. Rather, behavior and the implementation of methods/operations are also hidden from sight, and consequently, usage. Also note that inheritance as a concept is the main distinguishing factor between ADTs and the object-oriented approach. While it is impossible to clearly identify the roots of inheritance, in the database area, the classic article on aggregation and generalization as database abstractions by Smith and Smith [23] is a possibility.

Advanced concepts that are important in this book include signature, generic, overloading, and dispatching. Note that all of these concepts are not specific to the object-oriented approach, but are available in non-object-oriented programming languages. However, they do serve as important and reoccurring terminology:

- Signature: An OT's signature is the specification of its behavior, with an emphasis on its methods. For example, the signature of the previously described `stack` ADT includes the methods that comprise its public interface (`push`, `pop`, `top`, `initialize`, and `is_empty`), where each method's signature is characterized by its input parameters (and their types) and its output result.
- Generic: A type-parameterizable OT. For example, instead of having a stack that is bound to a specific data type (say, `integer`), the stack can require that the data type be provided as part of its initialization. Thus, the creation of a stack (e.g., `Stack(Real)`, `Stack(Char)`, etc.) binds the stack's methods to the appropriate types. Non-object-oriented languages such as Standard ml (`sml` [20]) and `Ada` support generics.
- Overloading: The ability to define two or more methods with the same name with different signatures. In all programming languages, operations like $+$, $-$, etc., are overloaded, since they can be used for integers, reals, sets, and so on. Other programming languages allow users to overload these and other operations in user-defined data types.
- Dispatching: The run-time or dynamic choice of the method to be called based on the type of the calling instance.

Note that Sections 3.3 and 3.4 will introduce and discuss the inheritance and dispatching concepts; the entire set of object-oriented concepts will be covered more completely in Chapter 2, using `C++` as an explanation vehicle.

3.2 Choosing Object Types

The first and most often question that is asked by newcomers to the object-oriented paradigm is often a variant of: *How are object types chosen?* Typical (lazy) answers to this question echo software engineering mantra (e.g., make your choices so that the end result is encapsulated with high cohesion and low coupling) or rely on that oldtime favorite "It comes with time and experience". While the first answer is marginally acceptable, a "better" answer should relate the following:

> Choosing object types/classes is not a first step in the design and development process, but rather must follow in a logical fashion from earlier efforts. In practice, the choice must be guided by a specification for an application, that contains the intent and requirements. The specification will make use of other software engineering techniques (e.g., DFDs, ERs, etc.), in an effort to define scope and breath of function, user interfaces, required user/system interactions, and so on. As the specification gains in content and complexity, the relevant object types begin to define themselves as a natural side-effect. This hopefully leads to a high-level object-oriented design. This in turn will be explored, refined, and evolved into a detailed design, which can then be transitioned to an implementation.

The moral is that it is unrealistic to 'jump' to an object-oriented design from only a basic understanding of an application. It would be just as unreasonable to make such a jump using another design and development technique.

Instead, one must acquire an understanding of what is appropriate to put into an object type or class. Three possibilities are illustrated below:

```
             Private Data   Public Interface

Employee     Name           Create_Employee()
OT/Class     Address        Give_Raise(Amount)
             SSN            Change_Address(New_Addr)
             ...            ...

ATM_Log      Acct_Name      Check_Database(Name)
OT/Class     PIN_Number     Verify_PIN(PIN)
             ...            Log_on_Actions(Name, PIN)
                            Reject()
                            ...

ATM_User     Action         Log_on_Steps()
OT/Class     Balance        Acct_WithD(Amt)
             WD_Amt         Check_Balance(Number)
             ...            Deposit_Check()
                            ...
```

The first OT has been designed from an information perspective, focusing on the idea that to track `Employee`s, standard data and operations are needed. The second OT, `ATM_Log`, embodies the functions that take place to log on an individual to an ATM session. Note that even with a functional view, information is needed to capture user input for verifying status. The third OT, `ATM_User`, represents a user interface by capturing the different interactions that are supported.[3] During this design process, there are a number of common design

[3] Any book on object-oriented concepts wouldn't be complete without an ATM example! Mercifully, this is the only one in this book!

flaws. First, a designer or IE places too much functionality in one OT. In this situation, the OT is often split into two or more OTs, but it is also possible that the functionality as it is split is absorbed into other, existing OTs. The latter leads to a second design flaw, an OT lacks functionality. In this situation, two more more OTs are often merged to result in an OT that exhibits a more cohesive behavior. Again, we emphasize that this has not been an exhaustive treatment of this subject. Rather, it is intended to provide an initial look at the different considerations that impact the choice of object types during design.

3.3 Inheritance: Motivation and Usage

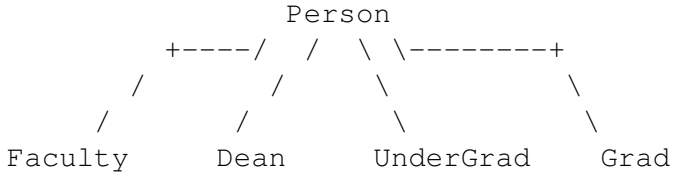
One of the key aspects that distinguishes the object-oriented paradigm from its ADT ancestor is the concept of *inheritance*. Instead of relying on a definition, an example is utilized to motivate inheritance. Consider the design for information in a university application, with four OTs defined:

```
Faculty      Dean        UnderGrad      Grad
  Name         Name        Name           Name
  SSN          SSN         SSN            SSN
  Rank         School      Dorm           Dorm
  Dept         Dept        Year           Program
  Yrs          Yrs         GPA            Degree
                                          GPA
```

For now, we focus only on information. To successfully utilize inheritance, an iterative process to identify commonalities (called generalization) and distinguish differences (called specialization) is undertaken. For the previous example, it is clear that two data components (`Name, SSN`) are common to all four OTs, and can be abstracted out (generalized) into the `Person` OT as given below:

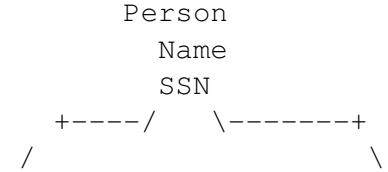
```
Person    Faculty:Person   Dean:Person   UnderGrad:Person   Grad:Person
  Name      Rank             School        Dorm               Dorm
  SSN       Dept             Dept          Year               Program
            Yrs              Yrs           GPA                Degree
                                                              GPA
```

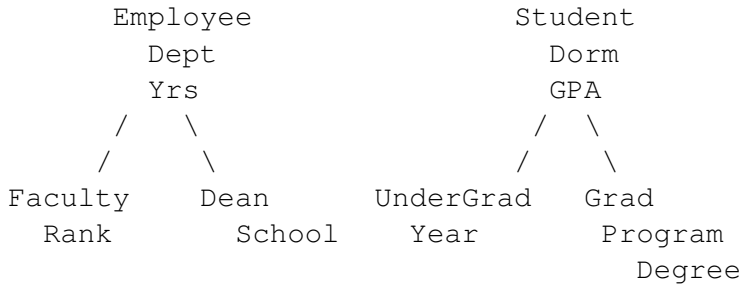
The original OTs (e.g., `Faculty:Person`) inherit private data (e.g., `Name` and `SSN`) from their ancestors (e.g., `Person`). This can be represented pictorially in an *inheritance hierarchy* as:



While this represents an improvement over the original characterization, it is by no means a 'final' or 'best' solution.

A 'better' solution to this problem would abstract out the commonalities of `Faculty` and `Dean`, as well as data shared by `UnderGrad` and `Grad`, leading to the following:





The better solution has utilized *single inheritance*, where there is at most one parent for each OT, to result in an *inheritance hierarchy or tree*. When viewed top down (starting from `Person`), each subsequent level in the hierarchy has been specialized, since the differences are captured. When viewed bottom up (starting from the leaves), generalization is represented, e.g., `Employee` contains the common characteristics of `Faculty` and `Dean`. Terminology such as parent, child, descendant, ancestor, and sibling has the same interpretation as when dealing with trees. In addition, the parent node in an inheritance association (e.g., `Person`) is referred to as a base or supertype, while its child (e.g., `Employee`) is a derived or subtype. Multiple inheritance, where an OT acquires behavior from two or more OTs, will be considered in later chapters.

In summary, inheritance can be defined as a relationship where the subtype acquires information and/or operations from a supertype. Inheritance is a transitive operation, allowing behavior to be passed from parent to child to grandchild, etc. Since the subtype is more refined than the supertype, it has been *specialized*. Alternatively, since the supertype contains common characteristics of all of its subtypes, it has been *generalized*. In practice, inheritance is utilized when two or more OTs are functionally and/or informationally related. That is, generalization and specialization decisions can be based on information (as in the previous example), function, or a combination of the two.

3.4 Illustration of Dispatching

One of the more important (some have argued, the **most** important) characteristics of the paradigm is support for dispatching. *Dispatching* allows a run-time or dynamic choice of the method that is to be called based on the class type of the invoking instance. As a concept, the effective use of dispatching is tightly bound with inheritance, and offers many benefits:

- versatility in the design and use of inheritance hierarchies;
- promotion of reuse and evolution of code, allowing hierarchies to be defined and evolved over time as needs and requirements change; and,
- development of code that is highly generic and easier to debug (and hence reuse/evolve).

But, as with all useful techniques, there is a caveat, namely, dispatching incurs a cost or overhead at both compile and runtime!

In Section 3.3, an inheritance hierarchy for a university application was designed, with a concentration on information. This basic design must be augmented with methods that initialize, store, manipulate, modify, and print the private data. Suppose that this example was augmented with the `Print_Info` method as follows:

```
Person::Print_Info() {Prints Name and SSN}

Employee::Print_Info() {Calls Person::Print_Info(); Prints Dept and Yrs; }

Faculty::Print_Info() {Calls Employee::Print_Info(); Prints Rank; }
```

```
Dean::Print_Info() {Calls Employee::Print_Info(); Prints School; }

Student::Print_Info() {Calls Person::Print_Info(); Prints Dorm and GPA; }

Grad::Print_Info() {Calls Employee::Print_Info(); Prints Program and Degree; }

UnderGrad::Print_Info() {Calls Employee::Print_Info(); Prints Year; }
```

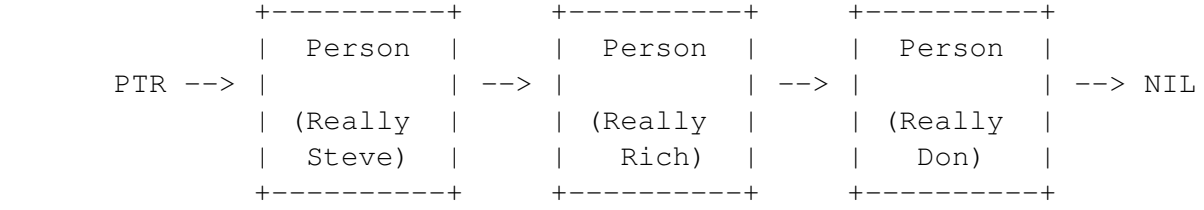
The double colon notation indicates which the OT on which a given method is defined. Notice the symmetry in these methods, where each OT in the inheritance hierarchy calls the `Print_Info` method of its immediate parent, followed by the printing of information that is specific to the OT. Given these methods, another OT can be utilized with this inheritance hierarchy to build a mixed list of instances of differing types that share a common ancestor. This can be achieved with the following pseudo-code, where the `LIST` OT is utilized to group different kinds of `Persons` (e.g., `Deans`, `Grads`, etc.) into a single named collection.

```
// Construct Faculty, Undergraduate, and Graduate Instances
FA1 = Faculty(Steve, 111, CSE, 7, AssocP);
UG1 = UnderGrad(Rich, 222, Towers, 3.5, Senior);
GR1 = Grad(Don, 333, OffCampus, 3.75, CSE, PhD);

PTR = POINTER TO CLASS FOR A LIST OF Persons;

// Note that Add_Person Inserts Instances at Front of LIST
PTR->LIST::Add_Person(GR1); // Place 'Don' Into LIST
PTR->LIST::Add_Person(UG1); // Place 'Rich' Into LIST
PTR->LIST::Add_Person(FA1); // Place 'Steve' Into LIST
```

The effect of the previous code is represented pictorially in the list of `Persons` pointed to by `PTR`.



Given this scenario, suppose that the the method given below has also been defined.

```
LIST::Print_A_Member()
  {
    Calls Print_Info(); // Defined on Person, Employee, Student, etc.
  }
```

What happens with the following function call?

```
PTR->Print_A_Member();
```

The call to `Print_A_Member()` invokes different `Print_Info` methods depending on the type of `Person` referenced by `PTR`. Specifically:

- If `Steve` is Being Referenced by PTR Then `Faculty::Print_Info` Called

- If `Rich` is Being Referenced by PTR Then `UnderGrad::Print_Info` Called
- If `Don` is Being Referenced by PTR Then `Grad::Print_Info` Called

This decision is made at runtime! The end result of such a capability is significant, and has a profound impact. First, there is no requirement to know in advance the number and different types of `Person`s that will be in the list referenced by `PTR`. Second, when there are additions to the inheritance hierarchy in the form of new OTs (e.g., `Staff:Employee`), the addition will allow `Staff` members to be placed in the list, and be manipulated as needed by the existing code, which remains unchanged! These and other benefits will be explored in later chapters.

4 Why Choose the Object-Oriented Paradigm?

While we have described many useful features of both ADTs and the object-oriented paradigm, we have still not enumerated a list of reasons for why the paradigm was chosen as the starting point for supporting information engineering. The main reasons behind our choice involve the abstraction and representation independence features of the paradigm, which are touted for the ability to support the following claims:

- Stresses Modularity: Achieved by the OT concept and encapsulation.
- Increases Productivity: While this is difficult to prove, it is a long-standing claim of the paradigm. Both inheritance and dispatching will have a strong impact on the attainability of this claim in practice.
- Controls Information Consistency: This is attained since hiding allows access to the private implementation to be managed.
- Promotes Software Reuse: IEs can reuse existing OTs for solving other problems (i.e., the stack is the classic example). In addition, through inheritance, IEs can define new OTs that acquire the characteristics of existing OTs (public interface) without violating the hidden implementation. Through dispatching, reuse is also supported. In the example of the previous section, when defining `Staff:Employee`, the `Print_Info` methods that already exist for `Employee` and `Person` are reused.
- Facilitates Software Evolution: The abstraction, encapsulation, hiding, and inheritance of the paradigm allows minor changes (to hidden or private implementations) to be transparently made, while major increases in functionality can be realized by extending the existing OT library through reuse. This was illustrated in the previous section with the example on dispatching. However, note that there are some major changes that require significant restructuring of the OT library in some applications. While evolution is facilitated, it is not as clearly and cleanly accomplished in these situations.

Testing cuts across all of these claims; done on an OT-by-OT basis (modularity); once tested, an OT can be used and reused (productivity); testing of changes that occur is limited to private implementation as long as public interface has not been changed (evolution). The end result of using the paradigm is supposed to be a clearer and easier conceptualization of the intended application. Further, the increased emphasis on design (in both time and effort) is offset by a reduction in implementation effort. However, these last two statements are difficult to verify in practice, since as a discipline, the object-oriented paradigm itself is still evolving. Our hope is that through the approach given in this textbook, readers will be convinced that these two statements and many of the above claims are indeed attainable.

5 What is Necessary and Sufficient for Information Engineering?

To this point, we have introduced many of the important concepts that form the basis for the approach for supporting information engineering. The major components of such an approach are the object-oriented capabilities that offer the many benefits and potential advantages as discussed in Sections 3 and 4. However, these capabilities must be extended with a number of characteristics:

1. The Need for Analyses: Incorporating engineering discipline and rigor into the design and development process, e.g., analyses by a civil engineer on the parameters and features of a design for a bridge.

2. Authorization and Sharability Issues: Controlling access to an application to both ensure confidentiality and to promote cooperation, e.g., the bidding process and communication of design modifications.

3. Enhanced Modeling Capabilities: Extending the object-oriented paradigm with a wide variety of design modeling constructs for more accurately capturing and representing the semantics of applications, e.g., various representations for the design of a bridge, including drawings, blueprints, scale models, etc.

4. Performance Modeling and Analyses: Considering the execution requirements for an application as an integral part of the overall design and development process, e.g., analyzing and evaluating the attainment of performance parameters for the design of a bridge.

5. Concurrent Engineering and Parallelism: Recognizing that team-oriented design and development is critical for large-scale applications. The existence of many parallel object-oriented programming languages also requires support for this capability at a design level, e.g., interactions between designers, engineers, contractors, suppliers, etc.

6. Integrated Design/Analyses Environment: Providing an environment that incorporates all of the aforementioned capabilities and characteristics. Such an environment must enhance and promote the application design and development process. When constructing a bridge, this is often performed on an ad-hoc basis by the prime contractor, and is an ideal candidate for upgrading and automation.

The reader will notice that we have already discussed many of these characteristics. The reminder of this section reviews the scope and content for them.

Analyses will be the vehicle through which our approach can supplement and extend the design process to include engineering rigor. Throughout the design process, IEs can invoke analyses against their designs to support a detailed and indepth evaluation of the design's content. We believe that such analyses are critical, since they allow IEs to compare and contrast an application's evolving design against its specification, which in turn promotes increased precision and accuracy. In later chapters, we will demonstrate the various levels of analyses, ranging from a microscopic inspection of a particular component of an application to high-level analyses, that yield a global view that intersects a large number of an application's components. For example, using analyses, an IE can determine which methods of the application write the same piece of information in an object type, since multiple writes on the same information has the potential to cause consistency problems. Alternatively, an IE can choose an object type and invoke an analysis technique that determines all other object types that interact with the chosen one. These interactions can be via inheritance, composition, relationships, and so on. In either example, if a problem is identified, corrections to the application's design can be made. When analyses are used in conjunction with incremental design, each design iteration should move closer and closer to the "final" design.

The second characteristic involves access issues, dealing with both authorization (who can use what when) and sharability (who needs to use the same what when). The "who" in either case might be an

object type, a method, an actual individual, or an application tool. The "what" refers to an OT, method, or a portion of the information in an instance. With respect to the object-oriented paradigm, the need for this characteristic is motivated by the fact that most object-oriented programming languages have a single public interface for each OT. Recall that the public interface contains all of the methods that must be made available to all of the possible users. Since this is the case, there are often one or two public methods that are part of the public interface and intended for use by only one user. However, since the method is public, it automatically becomes available to all users, i.e., access to public interface is total, selectivity is not allowed. It is our belief that there is a strong need for discretionary access, where different portions of the public interface are available to different users at different times. The solution utilized by our approach is to extend the object-oriented paradigm with user-role based security, where the responsibilities of an individual within the application determines his/her privileges. Finally, note that analyses are also critical for authorization and sharability, since they allow IEs to ask questions such as: Who uses a particular object type? Which components of an application does a given individual use? Which users write the same piece of information?

The third characteristic that will extend the paradigm is to provide an enhanced set of modeling choices and capabilities to support the design process of IEs. In most programming languages, there are libraries of routines that are available for specialized processing (e.g., I/O, string manipulation, graphics packages, etc.). Object-oriented programming languages are even more in sync with this philosophy, with extensive and complex class libraries available for use (and reuse). This library concept must also be available at the design level, with OT libraries that offer many different modeling choices. The advantage of such an approach is that information consistency within the library has been addressed and supported, thereby removing the burden from designers. We embrace such an approach to offer a wide variety of system-available modeling choices for IEs, and also provide the ability to support dynamic, designer-defined choices. These new modeling capabilities must also be augmented with a corresponding set of analyses, to provide a consistent basis for engineering discipline across all aspects of the application design and development process.

The fourth characteristic involves performance modeling and analyses, where the runtime or execution requirements for an application are specified. The role of performance occurs at many different levels within the design and development process. For example, queueing models are often used to determine gross system behavior as a means to evaluate design choices. Simulation can be used to obtain a finer-grained investigation of system performance, with time-complexity analyses employed to estimate detailed algorithms. In the most general case, performance focuses on system usage, and mandates that various parameters and limitations for an application be known at the very earliest stages. These include, but are not limited to: the speed and capacity of memory and/or disks; the response-time requirements of system components, functions, or interfaces; the maximum and/or minimum number of users; the tolerance (or lack thereof) to errors; and the potential impact of external features such as communications (via some type of network) or the physical environment. Overall, the main purpose of performance is to arrive at an understanding of the requirements and expected application behavior, which impacts on both structural and functional aspects of an application and its end-user needs and responsibilities. In addition, performance, like security, must be a first-class partner in the application development process. Lastly, to maintain consistency with the other characteristics, analyses must be available to support engineering design.

The fifth characteristic focuses on the expected usage of an application. Concurrent engineering for supporting collaboration and cooperative design are necessary from two perspectives. On the one hand, IEs that design systems and applications (see Section 1.2 again) must be able to work together to promote a successful development effort. From this perspective, the issues are:

* Who works on what portion of a design (at what times)?
* Who can see which portions of a design (at what times)?
* Who can modify which portions of a design (at what times)?

∗ Are localized design changes automatically propagated to all affected design components?

The first three questions involve time issues, while the last question addresses a need for maintaining consistency. On the other hand, it is just as likely that the application being designed requires concurrent engineering, e.g., a CAD/CAM application supports concurrent engineering to allow different engineers to cooperatively work on product design and development. In this situation, both the parallelization and distribution of the design for the application are key concerns. While there have been many proposals of concurrent and parallel object-oriented programming languages [NEED REFS], and there has been work on distribution issues for object-oriented DBSs [13], the majority of all of these efforts have emphasized implementation issues. Thus, for parallelism, there has been little work that addresses the following:

∗ What does parallelism mean for object-oriented design?

∗ How is an object-oriented design parallelized?

∗ How does concurrent engineering impact on parallelism?

These issues and others related to distribution must be addressed and resolved to adequately support the information engineering of complex applications.

The final characteristic captures and encapsulates object-oriented capabilities and the five earlier characteristics into an integrated design/analyses, prototyping, and development environment, as shown in Figure 1.2. Notationally, ISE, IDE, IDevE, are information engineers for specification, design, and development, respectively. Functionally, we propose and emphasize a language-independent environment (i.e., no

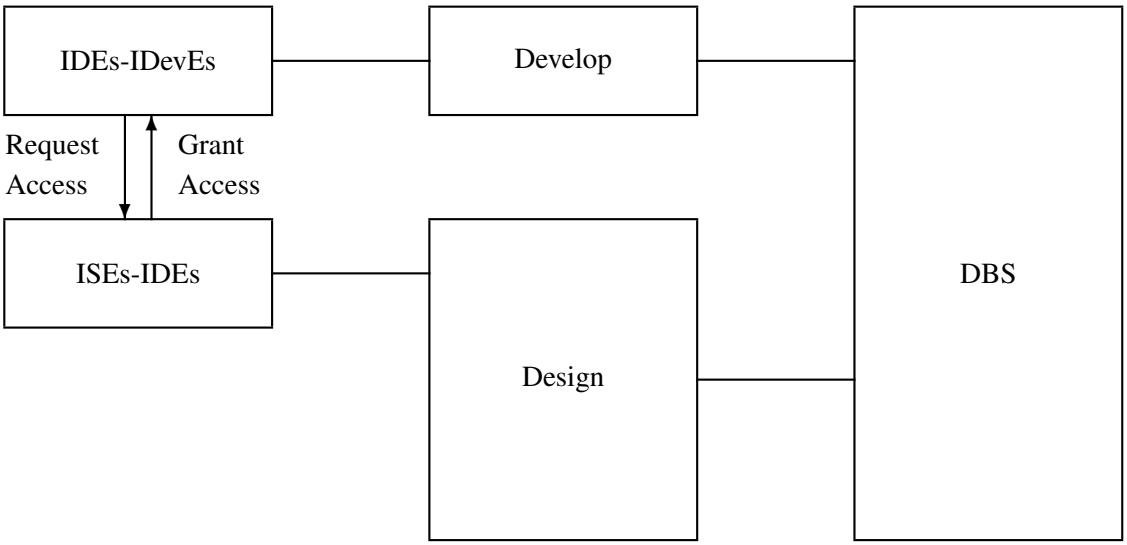


Figure 1.2: A Future Design, Prototyping, and Development Environment.

reliance on syntax and semantics of a particular object-oriented programming or database language) that is responsive to the needs and requirements of IEs by:

- Offering many choices, constructs, and alternatives when developing a design. Each choice has a fixed set of semantics (with respect to its behavior), and thus guarantees information consistency.
- Providing automatic feedback by alerting users when a design choice has the potential to cause problems and/or violate the information consistency of an application.

- Supporting a user-controlled portion of the environment for user-initiated design analyses, thereby promoting engineering discipline and rigor throughout the incremental and iterative design process.
- Upgrading the status of the usage to and access of the application by both IEs and end-users to an equal partner and integral aspect of the design and development process. Security for authorized users that also supports sharability is the means through which confidentiality issues can be addressed.
- Examining persistence, performance, concurrent engineering, and parallelism issues as part of the design process.
- Building a bridge between object-oriented design and subsequent implementation via the automatic generation of "code". This feature of the environment is strongly related to the first feature. That is, since each design choice has a fixed set of semantics, it is possible to automatically generate code for a significant percentage of the application, when given certain design-level information on the OTs that comprise an application. In our approach, this final code-generation step can occur in one or more programming and/or database language, providing a degree of flexibility or agility in the overall design and development process. Actually, the code generation should be in multiple, unified object-oriented programming/database languages. Through automatic code generation, a graceful transition to the implementation process is achieved.

Collectively, in Parts I and II of this book, all of these issues will be addressed in various levels of detail.

6 Looking Ahead and Further Readings

This introductory chapter has served as the basis to convey a philosophy of design and development for supporting information engineering. The remainder of Part I of the book examines traditional, and object-oriented programming and database techniques (Chapter 2), and investigates the specification process as a means for identifying the characteristics and functionalities of an application (Chapter 3). This material serves as the basis for the majority of the technical material in Part II on object-oriented design and analyses, which also will present an environment that is able to support, in part, our approach.

Lastly, the list below represents a set of readings that is intended to either coincide or follow the material of this chapter, with the purpose of reinforcing the concepts.

- *IEEE Trans. on Knowledge and Data Engr.*, Vol. 1, No. 1, March 1989, pages 1-32.
- Ghezzi, C., et al., *Fundamentals of Software Engineering*, Prentice-Hall, 1991.
- Budd, T., *An Introduction to Object-Oriented Programming*, Addison-Wesley, 1991, Chapter 1.

The first reading provides a set of forewords and one article to introduce, compare, and contrast data, information, and knowledge, a key motivational factor for this book. The second reading is an excellent software engineering textbook that starts with the idea that software engineering must embrace and evolve to be similar to other engineering disciplines, another key concept of our approach. Other software engineering textbooks are also appropriate [22, 24] and basically serve as the assumed background level or expertise of the reader. The third reading complements our design level approach by introducing and reviewing detailed issues related to object-oriented programming. Examples using `C++`, `Smalltalk`, `Objective-C`, and `ObjectPascal` are employed. Readings from Budd will be included throughout the remainder of our textbook.

Chapter 1 Exercises

1. The terms Data, Information, and Knowledge, conflict and overlap with one another in both definition and usage.

   a. Compare and contrast these three terms with their definitions and usage in software engineering, databases and artificial intelligence.

   b. Develop a set of your own definitions for these three terms.

   c. Do the three terms cover all of the possibilities? That is, are there concepts that precede data, follow knowledge, or fall somewhere in between the three?

2. The computing field is at a crossroads, and even after over 20 years since the first computing department at a university, there is still much debate over whether computing is science and/or engineering. Detail your own views on this issue.

3. One of the driving concepts in the book is that software engineering is an oxymoron, since there is little if any true "engineering" in the many different techniques and methodologies that comprise the discipline. Identify aspects of software engineering that you believe are definitely "engineering". Make sure that you argue your beliefs. If you do not believe that there is any true engineering, then your argument must reflect this assertion.

4. In Figure 1.1, Tomorrow's View of the interaction between a PLS and a DBS was given. While it's likely that the overlap will increase, the chances are that there will still be dedicated functions within each type of system. For a PLS and a DBS, propose two functions or responsibilities that you believe will always be separate from a combined hybrid system.

5. In Section 1.2, we reviewed issues related to secure access and management of information. Identify and describe at least two other examples where confidentiality and misuse of information can occur.

6. Section 1.2 contains a set of 11 questions that were utilized to elaborate on information engineering requirements for SDEs. Examine answers to this same set of questions for the design of a: bridge, microprocessor, casino blackjack machine, and application of your choice.

7. Information consistency was defined with respect to the usage, persistence, integrity and security, and validity (relevance) of information. Speculate on why information consistency is important and discuss how it is attained.

8. Open-ended design, as a concept, is similar to the long-standing premise that you can never find the last bug in a program, i.e., we can never know when we are totally done with a design. Do you agree with this concept? If so, why? If not, why not? Feel free to draw on your own experiences for answering this question.

9. Abstract data types (ADTs) are not a new concept, but are almost 20 years old. Object-oriented techniques as supported in languages like `Smalltalk`, `C++`, and `Objective C`, are just the most recent realization of ADTs. Compare and contrast the ADT/object-oriented concepts discussed in this chapter with modules in `Modula-2` and packages in `Ada`. Make sure that you address the following issues:

   a. Are encapsulation, hiding, inheritance, and dispatching supported?

   b. What, if any, are analogs to an OT library, generics, and overloading?

10. Choose a non-object-oriented programming language and discuss the degree to which the language either supports or falls short of the claims of the object-oriented paradigm (see Section 4 again).

11. An integrated design and analyses environment for supporting information engineering via the object-oriented paradigm has been proposed in Figure 1.2. Speculate on its functional characteristics and overall makeup.

12. The POSTGRES database system [26] provides support for complex objects via extensions to the relational data model and associated INGRES system. Such a system has attempted to retrofit object-oriented capabilities into an environment that is based on a model that definitely conflicts with object-oriented precepts and principles as discussed in this chapter. Comment on whether you believe such an approach has merit and is valid. Also, what are the inherent pitfalls of such an approach?

13. Develop and discuss an example of concurrent engineering and detail its role in advanced application development. How does concurrent engineering relate to and impact on other aspects of information engineering design, in particular, persistence via a database and its security.

References

[1] *The American Heritage Dictionary*, Dell Publishing Co., 1983.

[2] J.G.P. Barnes, *Programming in Ada plus Language Reference Manual*, third edition, Addison-Wesley, 1991.

[3] R. Bretl, et al., "The GemStone Data Management System", in *Object-Oriented Concepts, Databases and Applications,* W. Kim, and F. Lochovsky, (eds.), ACM Press, Addison-Wesley, 1989.

[4] L. Cardelli, et al., "Modula-3 Language Definition", *ACM SIGPLAN Notices*, Vol. 27, No. 8, Aug. 1992.

[5] M. Caruso and E. Sciore, "The Vision Object-Oriented Database Management System", in *Advances in Database Programming Languages*, F. Bancilhon and P Buneman (eds.), Addison-Wesley, 1990.

[6] O. Deux, et al., "The O2 System", *Comm. of the ACM*, Vol. 34, No. 10, Oct. 1991.

[7] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice-Hall, 1991.

[8] A. Goldberg, *Smalltalk-80: The Language*, Addison-Wesley, 1989.

[9] *Computing the Future: A Broader Agenda for Computer Science and Engineering*, J. Hartmanis and H. Lin (eds.), National Academy Press, 1992.

[10] S. Horowitz and T. Teitelbaum, "Relations and Attributes", *Proc. of the ACM SIGPLAN Software Engineering Sym. on Language Issues in Programming Environments*, July 1985.

[11] Forewords to first issue, *IEEE Trans. on Knowledge and Data Engr.*, Vol. 1, No. 1, March 1989, pages 1-16.

[12] W.Kim, et al., "Architecture of the Orion Next-Generation Database System" *IEEE Trans. on Knowledge and Data Engr.*, Vol. 2, No. 1, March 1990.

[13] W.Kim, et al., "A Distributed Object-Oriented Database System Supporting Shared and Private Databases" *IEEE Trans. on Information Systems*, Vol. 19, No. 1, Jan. 1991.

[14] C. Lamb, et al., "The ObjectStore Database System", *Comm. of the ACM*, Vol. 34, No. 10, Oct. 1991.

[15] M. Linton, "Relational Views of Programs", *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Sym. on Practical Software Development Environments*, May 1984.

[16] B. Liskov and S. Zilles "Specification Techniques for Data Abstraction", *IEEE Trans. on Software Engineering,* Vol. SE-1, March 1975.

[17] B. Liskov, et al., "Abstraction Mechanisms in CLU", *Comm. of the ACM*, Vol. 20, No. 8, Aug. 1977.

[18] J. McCarthy, "Merging CS and CE Disciplines is Not a Good Idea", *Computing Research News*, Vol. 5, No. 1, Jan. 1993.

[19] B. Meyer, *Eiffel: The Language*, Prentice-Hall, 1991.

[20] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML,* The MIT Press, 1990.

[21] "ONTOS Object Database Documentation", Release 2.1, Ontologic, Inc., Burlington, MA, June 1991.

[22] R. Pressman, *Software Engineering: A Practitioner's Approach*, third edition, McGraw-Hill, 1992.

[23] J. Smith and D. Smith, "Database Abstractions: Aggregation and Generalization", *ACM Trans. on Database Systems*, Vol. 2, No. 2, June 1977.

[24] I. Sommerville, *Software Engineering*, fourth edition, Addison-Wesley, 1992.

[25] M. Stonebraker, et al., "The Design and Implementation of Ingres", *ACM Trans. on Database Systems*, Vol. 1, No. 3, Sept. 1976.

[26] M. Stonebraker and L. Rowe, "The Design of Postgres", *Proc. of the 1986 ACM SIGMOD Intl. Conf. on Management of Data*, June 1986.

[27] B. Stroustrup, *The C++ Programming Language,* Addison-Wesley, 1986.

[28] L. Tesler, "Object Pascal Report", Apple Computer, Santa Clara, CA, 1985.

[29] *Webster's Ninth New Collegiate Dictionary*.

Information engineering , also known as Information technology engineering , information engineering methodology or data engineering, is a software engineering approach to designing and developing information systems.Â For faster navigation, this Iframe is preloading the Wikiwand page for Information engineering. Home. News. Information engineering is a family of data-oriented analysis and techniques used to design, develop, and maintain information systems which support strategic missions, decision processes, and daily operations of a company. It is often regarded as a data-oriented methodology rather than a process-oriented methodology. This paper provides a brief introduction to information engineering. nformation engineering domain [2]. â€¦ Figures - uploaded by Adebowale E. Shadare. Information engineering is the engineering discipline that deals with the generation, distribution, analysis, and use of information, data, and knowledge in systems. The field first became identifiable in the early 21st century. The components of information engineering include more theoretical fields such as machine learning, artificial intelligence, control theory, signal processing, and information theory, and more applied fields such as computer vision, natural language processing, bioinformatics View Information Engineering Research Papers on Academia.edu for free.Â Engineering Information: Conceptual Elements Related Information Management and Information Systems. The study addresses the challenges and transformations generated by access to the data and information management activity, considering the time to retrieve the correct information for decision making. The larger, the organization will be more. Gartner defines Information Engineering (IE) as " a methodology for developing an integrated information system based on the sharing of common data, with emphasis on decision support needs as well as transaction-processing (TP) requirements." It assumes logical data representations are relatively stable, as opposed to the frequently changing processes that use the data. Therefore, the logical data model, which reflects an organizationâ€™s rules and policies, should be the basis for systems development.