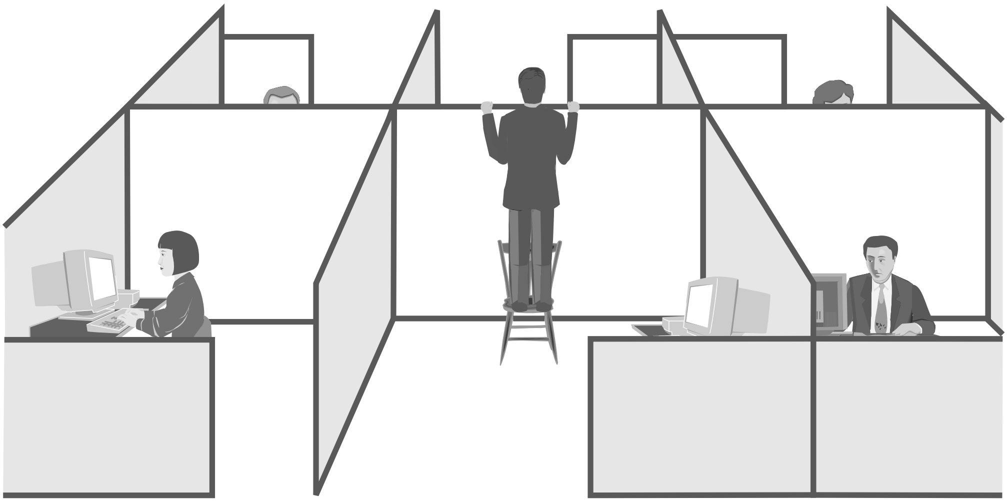
Preface



This book is directly aimed to eliminate exhausting amounts of frustration in getting to know and working with software objects in the most effective ways. It is concise and broad—and definitely *not* simplistic—specifically to strengthen each reader's object-oriented mentality and to mentally solidify individual pieces of information. This is because conciseness is more compatible with memory, and broadness is more compatible with understanding.

Very often, a book must be read once, just to get a general feeling for it; then, most or all of it must be read a second time to begin to thoroughly absorb the details. That's because each layer of the subject has been broken up into completely separate pieces of the book, which go from the most general to the most specific information for each aspect. As a result, the reader doesn't know much about the overall structure and, therefore, doesn't know how the details fit into it.

This book uses the strategy of hierarchic conveyance of information—explaining the most important components and how they relate to each other, then demonstrating how the next most important components fit with the structure that has been established, and continuing to build a solid mentality in that manner, including making recommendations for further reading for further details. With a mental structure established, the details can be taken in more individually, with the objective of directly understanding individual functionality. And the other recommended books effectively cover multiple views of the same topics; multiple views of any topic *intersect* in a strong overall feeling for it. But other books can be recommended for only Part I of this book, because Part II advances to places where no other books go.

This book is driven by the fact that *accuracy*—consistently aligned mentality—has fundamental and far-reaching benefits for beginners and veterans alike. Being accurate translates into not cutting important corners, which translates into eliminating holes at all levels of designs, causing them to flow *much* more smoothly. The end result is developers' *power* over the software. After all, the entire object-oriented concept is *based on* clarity. It's based on a flow of *thinking*. And it's the flow that provides something extra. This is a parallel to the fact that aligned molecules produce a flow of energy that provides a magnetic force.

The explanations of this book leverage both straightforward logic and significant new points of view to establish consistent orientation at all levels, eliminate bottlenecks in thinking and development, and create a *single feel* for the spectrum of object orientation. This is an example of the fact that consistency in any endeavor eliminates complication. This book was specifically written *across* concepts of object orientation, in order to establish *context* for any focus. It explains some concepts and uses some vocabulary as other explanations don't, allowing it to tie everything together as other explanations can't. Especially for a subject like this, having a clear mentality for all of its ramifications, all at the same time, is integral to real success. Without that, it's possible to make things *work*, but they're then far from optimal. Having a clear mentality frees developers to concentrate their efforts on the most effective solutions for each situation.

This book draws simple parallels between aspects of the entire development process. Its explanations make other explanations easier to understand, *explicitly* providing the cohesion and intuition that they don't. Also, it addresses explicitly points and concepts that are commonly perceived only vaguely. Further, it introduces comprehensive tools to best manage and work with object orientation; these actually further clarify the characteristics of software and its development. All of this is immediately very useful to every member of any software development team, at every level of responsibility. And the fact that it's fundamentally easier to understand and manage systems through these approaches will make them extremely valuable industrywide.

With a strong mentality, training requirements are much less of an impediment to choosing the best technology for the job at hand. The task is not about what exactly the team members (and potential team members) have done before. It's not about making the problem fit the solution. And it's not about just rolling the dice and doing what's trendy. It's about the practical ability to jump into a project and learn just the relevant details, at every level, very quickly; this is a parallel to the concept of a class structure and its extensions. More fundamentally, it applies to unobstructed mentality and directed checklists, working together to achieve optimal productivity. It's ultimately an extension of the principle that mental flexibility enables the best systems. Straightforward actions are just as helpful to developers as they are to users; further, straightforward mentality allows developers to continually and comprehensively relate to users—which enables the best systems.

Now, explaining it in one paragraph doesn't do it any kind of justice, but *iterating infusion* describes the fact that any system has multiple coexisting levels and that, repeatedly, separate but compatible technologies are brought together to create advancements. These can be baby-steps or leaps, with little more effort or even less effort. In more general terms, the same thing in a different context can take on much more power. And, actually, this phenomenon is at the heart of object-oriented software.

Organization of This Book

Iterating Infusion has a comprehensive introduction and five chapters in two parts, each feeding the next, building to the last. It is highly recommended that all be read, in order, by any audience. Skimming or skipping around is not nearly as effective. It's the entire book that demonstrates iterating infusion, a phenomenon that is independent of the subjects that are examined explicitly.

The first segment of the book, "Introduction", is crucial to the book as a whole. It's actually a set of introductions, one for each part of the book, all in one place. With this device, the course through the entire book is made immediately thoroughly familiar.

Part I, "Whole Consistency", contains the following:

- **Chapter One, Orientation: Comparisons Among Objects and Structures**, presents basic object-oriented concepts in the context of more traditional views. It addresses designing and programming properties and common language syntax—tools provided to significantly ease further study.

- **Chapter Two, Bi-design: Object-Oriented Designing Strategies**, is very much geared to a designing mind-set. It breaks down characteristics of object-oriented systems and discusses strategies for gaining control of the overall development effort.

- **Chapter Three, Untangled Web: The Evolution of an Enterprise-Level Design**, lays out a very common example of how a framework of devices and classes evolves to accommodate a specific need. It ties together the previous abstract points concretely.

Part II, "Derived Simplicity", consists of the following:

- **Chapter Four, x = Why: Interaction Algebra for Analyzing and Designing**, explains a specialized mathematically-based notation for describing object interactions. This highly structured technique helps to eliminate design holes and illuminate characteristics of object relationships, both general and specific.

- **Chapter Five, Live and Unscripted: Object Animation, a Clearer View of Automation**, establishes a revolutionarily simpler view of all software, especially object-oriented, and delineates a different *type* of software language—data oriented, as opposed to extended procedure oriented—that is derived from that view and fundamentally serves development.

Finally, the “Conclusion” element is a very brief wrap-up. It clearly demonstrates how much simpler and more advanced software development is with the understandings that the rest of the book provides.

Also, this book uses visual techniques that are specifically designed to best reinforce conveyance. First and foremost, it presents each diagram *before* the text that applies to it. This arrangement fosters mental focus, as opposed to trailing diagrams, which, ultimately, only tame scattered thoughts. Because of the common parallel, this technique is called “picture captioning”. Next, the book throws a “spotlight” on key points, in a bordered box with a different font, immediately following the paragraph in which the point appears. Last, it rearranges series of related information each into a list, immediately following the paragraph in which the series appears. Additionally, it employs all of these visual attributes in shades of gray, to contrast with the black text, for extra visual dimension.

A HELPFUL REMINDER

It should be kept in mind that many books, including the titles recommended by this one, have code examples that can be fundamentally difficult to follow, in at least three ways.

First, most of them don’t have any degree of explanation of the code until after it, even to explain the basic functionality of other code that the example *uses*. They unfortunately don’t employ the technique of “telegraphing”—that is, explaining the basic flow of the example, then showing the code, and then explaining it in detail. An effect of this is that interpreting the code can have a lot of gaps. In reading untelegraphed code, skipping to the explanation and referencing the code along the way is the quickest way to understanding the example.

Second, many complex examples present the code in fragments, between sets of explanation text, with very little visual assistance. These fragments are from both the same class and differing classes, again with very little visual differentiation. Even something as simple as separation lines between the text and the code, and a note-font class name header for each fragment, help to make all of the parts immediately distinctive. This has an effect of losing conveyance of the organization of the code—the whole point of object orientation. The only compensation for this is reviewing the example, mentally combining the fragments in the appropriate ways.

And third, some of the examples ultimately seem functionally pointless, specifically because they use hard-coded values in places where variables make more sense. They do this, of course, to make the examples shorter—not requiring database access—but they usually don’t mention it; an effect is that actual purpose is not conveyed. They could refer to variables that they explain come from an unseen database access, but they often don’t. In these cases, a mental substitution of variables from a database helps to establish purpose.

About the Author

GREG ANTHONY is a near-lifelong systems analyst who has been designing and programming software since he was 8 years of age, professionally since he was 12. In over 15 years, he has worked in all areas of development and systems management, often as a consultant, in environments from PC to mid-range to mainframe, and in industries including finance, insurance, retail, and transportation.

Throughout his career, he has also created utilities of all sizes to automate development tasks, especially code generators (fourth-generation tools), code analyzers (diagnostic tools), version-control facilities, and system software interface redesigns, enabling both extreme user friendliness and extreme efficiency—in both execution and development.

He is an alumnus of the Johns Hopkins University's Center for Talented Youth, the landmark organization for gifted children 8 to 18. He has compiled interlocking philosophies mostly through independent studying and experimentation. And his ability to explain things in plain language, and in many ways, has taken him from tutoring to training to dedicated writing.

Introduction

This is a comprehensive introduction to each part of the book, preceded by a very brief history, for complete context.

A Very Brief History

Computer software development has been occurring for decades. Everyone knows that the purpose of computer software is to help them to accomplish things. Software is *applied* to a variety of tasks, processes, and methods—for example, documentation (word processing), accounting, and picture manipulation—so each of these is called an **application**.

On first thought, the best way to create an application is to arrange all of it in *one* big group, but when an application has several major tasks, it's better to break up them into multiple units (**programs**), one for each major task. Further, it seems that the best way to arrange each program is consecutively, from beginning to end; this is known as **procedural** or **fall-through** code.

But *soft*ware is fundamentally changeable, as opposed to hardware, which is fundamentally *un*changeable, or firmware, which is hardware with switches (for logical options). And software has never occurred in completely consecutive steps; that began with the basic concept of **branching**—selecting the next step based on a condition while the program is running (executing). Over time, the more flexible software needed to be, the more complex branching became, and changing an application came to require *a lot* of searching through code to figure out execution paths—the actual order of the steps.

To *manage* branching, the concept of **structuring** software came about. Most succinctly put, this grouped the steps *between the branches*, creating a *logical organization*, with each branch *referencing* a group. Further, this created **modules**, *isolated* pieces of software, and even *categorized* them, meaning that different modules could accomplish the same types of things. It reduced searching significantly, but changing an application still required making changes in multiple pieces of code to accomplish a single functional change and figuring out how to improve one function without harming another.

To manage branching *better*, and especially to manage changes, the concept of organizing the groups into *functional* units became popularized, effectively extending modularization, isolation, and categorization. These units are commonly called **objects**, and the functional grouping is commonly called **object orientation**. This organization essentially helped to *centralize* code changes and make the pieces *more independent* of each other. With it, a functional change became much more self-contained (**encapsulated**) and safe.

Whole Consistency (Part I)

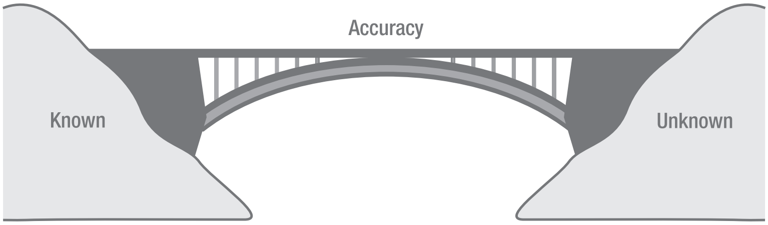
The principles of object orientation have made the processes of software development simpler. But, from its most introductory teaching, the principles themselves have commonly been made too complex. Further, this has led to the exponential complexity that comes with trying to have an off-the-shelf approach to every conceivable situation; so development is again becoming more and more of an effort, instead of less and less. This is because of the overhead of extensive conformity—and the fact that required closely related code modules effectively result in just structured software with more referencing. (This is also the fundamental flaw that many structured software veterans see, causing them to stay away from newer technologies.)

The vast benefits of object-oriented software require investments of managing and working with complex designs, which include many interdependent and dynamic components. Misunderstandings, large and small, about these complexities detract from the designs' effectiveness, blatantly and esoterically. And, compared with the earlier orientations, most of the techniques of object orientation are each only a *slightly* different approach to a task, with a different name; sometimes, the name is the *only* thing that is different. But a few things are *significantly* different, and the complication is that these are what the rest fall around. Over the years, as the popularity of object orientation has spread, designers and engineers have developed many pointed strategies for improving their effectiveness. But more comprehensive—more fundamentally effective—strategies tend to elude them, and far too many projects still fail, because knowing only technical devices is not enough.

There is a growing movement to simplify—to keep systems as simple as possible, as often as possible—to minimize developmental overhead. Much the way systems have historically needed to be overhauled, at a higher level, there is a growing movement to fundamentally overhaul the world of object-oriented software and its development. This higher level of overhaul becomes more necessary because of the open nature of the industry's evolution, specifically facilitated and intensified by the self-contained changeability of object orientation, which allows one group's changes to be plugged into several others'. Very effectively, however, this higher level of overhaul incorporates the newer technology of "hot swapping", because it must be driven by mental shifting—seeing existing, functional systems in new ways. This maximizes derived practical effectiveness. (It also allows all of those structured veterans to make the leap that they haven't yet.) And understanding how the spectrum of concepts fits together allows simplification without loss of power.

Orientation: Comparisons Among Objects and Structures (Chapter One)

Forget the fancy vocabulary. Forget the structure bashing. Forget the idea that object-oriented software is completely different from structured software. It *is* different thinking, but it really just requires a *solid* overview to clearly see how they are very much the same behind the scenes. And structured software veterans can leverage what they already understand from structures.



Further, there are established keywords and explanations of some aspects of object orientation that are misleading, so they unnecessarily complicate overall comprehension. For example, ambiguous meanings show a lack of accuracy: commonly in object orientation, “parent” and “child” are used to describe both object *definition* relationships and object collection relationships, and these relationships entail very different things. Most directly here, instead of the *leap* that is commonly required to get the feeling of object orientation, accuracy provides an easy bridge. This book delineates both the standard and more accurate vocabularies, so whenever the standard words are misleading, the more accurate words can simply be mentally substituted.

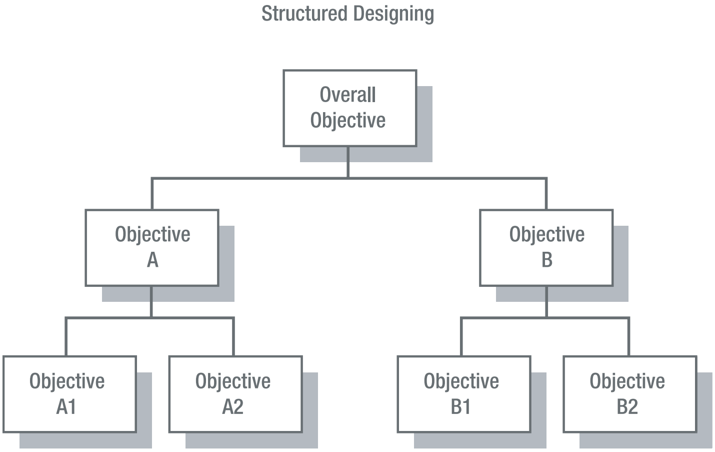
Sometimes, differing words for the same thing are reasonably driven by differing points of view—differing contexts. In fact, the history of software has had *many* instances of one entity being seen in multiple ways. Among many other benefits, being able to understand everything from a consistent point of view eliminates the frequent need for extra effort at figuring out context.

And two things should be kept in mind:

- *Procedure* orientation was the *pre*structured orientation.

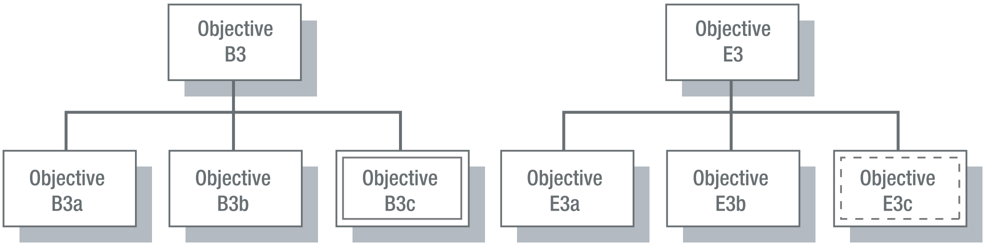
- The structured software development process has created a great deal of excellent software.

Related to Designing



The first thing that is needed in this overview is a comparable overview of the structured software development process. Ultimately, the structured process requires a system analysis that arrives at a design of **a hierarchic structure of objectives**, from the most general to the most specific. At all levels, this defines data items and what happens to them (processes). With each level of the hierarchy ordered chronologically, the system functions are clear. At that point, scenarios (also known as **use cases**) can be run through the structure, chaining the components in execution sequence, as a cross-check to make sure that nothing is missed. The structure also directly accommodates data flow diagrams (and process flow diagrams, which aren't really necessary when data flow diagrams are geared to low-enough levels of the system structure—but that's a later subject). It even includes the code-level objectives; structured programs are contiguous subsets of the overall system structure. Common functions are usually repeated and tailored to each particular usage.

Similarities and Differences



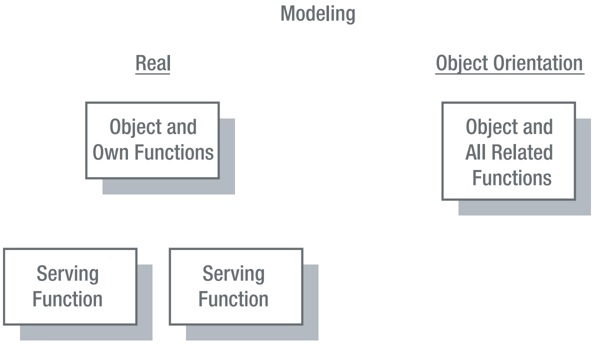
B3c and E3c differ.

The object-oriented software development process requires a system analysis that arrives at a design of **a network of *sets* of objectives**. This puts more focus on the functions than just how they fit into the system. The object-oriented process actually can continue from the point of the scenarios running through the structure. *Objects are defined by the similarities and differences between the execution scenarios*. This includes varying degrees of likely future scenarios, both common and system-specific. The combinations of similarities and differences define how code can be shared. A parallel to this can be found with conditional combinations—"and" and "or" conditions, sometimes with multiple sets of parentheses, in an "if" test—in their separation into progressing segments—with individual tests. Objects can then be further separated by whether shared segments are (very) closely related.

Of course, there are very different ways of looking at the object-oriented development process, especially as familiarity brings feeling for objects. Other views prove to be more direct, but this one can always serve as context for them. Universally, the most critical skill, in *any* orientation, is the ability to recognize patterns—commonalities, differentiations, and *dependencies*.

Taking a good look, it can be seen that *any application of an object-oriented network still requires the structured linking of objects*; in other words, the practical usage of object orientation still fundamentally requires an aspect of structured development. In many cases, no code, in any form, is written without an application in mind; there, at the very least, code can be created more independently than in pure structured development. This even allows *pieces* of systemwide functionality to be explicitly coded. Before this approach, the only way to handle pieces of functionality was with standard methods (protocols). Ultimately, object orientation is a very thorough way of approaching the traditional separation of shared code into utility programs.

The well-known idea of software objects is that they model objects that physically exist in the real world. Their data and processes are seen to be *characteristics*. But one reality of software objects is that they can also model objects that don't (yet) physically exist in the real world; these are conceptual objects. Looking at that more broadly, every built object that does physically exist was a conceptual object first; in other words, every physical object was a mental object first. And, often, there's no justification for building the physical object; but software is more flexible. This includes that a conceptual object can be shared with—in other words, implicitly duplicated for—other objects.

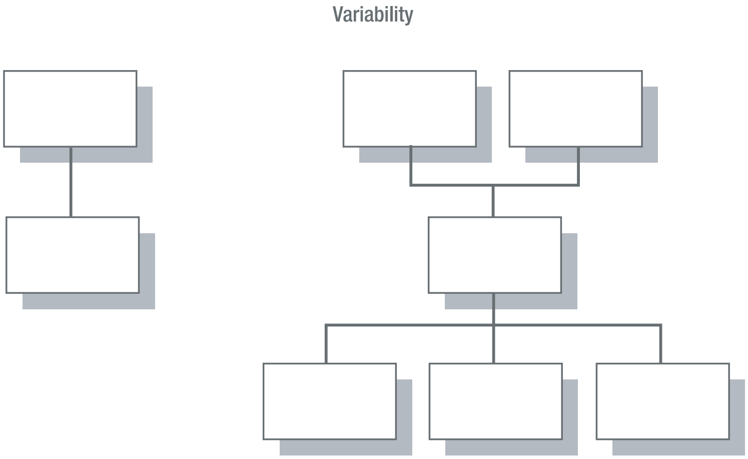


However, in an even more fundamental way, each object isn't really based on a real object; it's more based on functions that a real object *needs*. The significant practical difference between the two concepts is that interobject checks and balances are needed in the real world because of the factor of a lack of object integrity, but this factor doesn't exist in software. A very good example is that, in the real world, an employee can't be relied on to do his or her own payroll with complete integrity, but this is a perfect function to have in an employee object, simply because it serves the employee. This understanding is commonly utilized but not much mentioned. Commonly, a description of a particular class is that it "represents" a particular real object; here, it can be helpful to mentally substitute the word "serves".

Bi-design: Object-Oriented Designing Strategies (Chapter Two)

The inanimate components of any field of designing can have characteristics of being alive. The most effective designing requires feeling that phenomenon. It requires deeply understanding the components, individually and collectively, and balancing all of their needs at the same time; it requires orchestration. And it requires a *dedicated* thought process. As they are in many things, simple philosophies are the best guide through all levels of designing. Also, the biggest reason why there is a gap between cutting-edge (research-developed) designing techniques and everyday (business-practiced) ones is that the organization and length of common teaching techniques make it too difficult both to see the thinking that drives a comprehensive process and to understand how to apply it. This results in an inability to *manage* the process. What's needed is a comprehensive set of simple object-oriented designing philosophies and a dynamic overall strategy for applying them in various situations.

Interaction Mechanisms



Initial development requires creation of a network of classes *before* they can be combined to create an application, although third-party sets can be acquired and tailored for common functions. Combining sets to more easily create multiple applications requires areas of flexibility. The degree of flexibility that any part of the software must have has a direct impact on how complex its interaction mechanisms must be. Simply put, flexibility is served by a mechanism of variability. This is where objects (and polymorphism) contribute; they are used, in essence, as network variables—logically replacing hard-coded conditionals. This entails some factor of separation (**indirection**) between interacting methods, which is **loose coupling**, instead of **tight coupling**. The mechanism acts as a *translator*, typically between parts of the class's implementation or between its interface and its implementation.

A very simple example of indirection and loose coupling is a mathematical one. It's possible to programmatically convert a number from any base to any other base by converting to and from a constant base. For example, instead of converting directly from base 2 (binary) to base 16 (hexadecimal), converting from base 2 to base 10 (decimal), and then base 10 to base 16, yields the same result. And, with this configuration, any beginning and ending bases are possible with no further programming. (Because letters are used for digit values above 9, the highest practical base is 36—10 numerals + 26 letters.) This concept also relates to the properties of probabilities: the possible permutations—combinations considering sequence—of two factors are the possibilities of each, multiplied by the other; being able to deal with them separately is usually much less overall work. It's also why digital (representative individualized) processing has much more power than analog (quantitative overall) processing.

These loosely coupled parts are each a *type* of class (or part of a class); they each specialize in a particular type of role. This understanding brings object-oriented designing up another level. It's then fairly easy to see how individual parts of the same type can be swapped for each other, and how a team (an interdependent collection) of types of parts can be needed to build a whole logical function. While a usage of loose coupling is more difficult to comprehend, a usage of tight coupling is more difficult to change. Tight coupling means that parts are directly dependent on each other, which means that changes in one part are more likely to adversely affect other parts and thus require more changes. So, tight coupling (direct dependence) cascades the effects of changes.

It's very enlightening, here, to take a look at a bit of software history. When there was very little memory available for any one program, programs were very restricted in size; each was, therefore, a functional module. As memory availability grew, so did programs; few developers recognized the value of the interdependent pieces of code. The most popular thing to do was the easier thing, which didn't include the extra considerations of the ability to directly swap one piece of code for another; consequently, *the inherent modularity was lost.* It can easily be seen that those extra considerations at that time could have caused object orientation to become popular much earlier in software's history; it can easily be seen that the trends of software designing might actually have just gone in the wrong direction at that time.

Across all of the object-oriented systems that have ever existed, all of the countless interaction mechanisms have been of only a relatively few types; all of the interaction mechanisms of any particular type have common characteristics (components and behaviors). These types are commonly known as **design patterns**, and learning them makes designing simpler and smoother. Ultimately, they are standardized techniques for manipulating interaction variables. But it should be clearly understood that these are *design* patterns, not design*ing* patterns, which are part of what are commonly known as methodologies. A pattern of designing *needs* and ways to serve them defines a designing pattern (which includes an analyzing phase). There are many designing patterns, from many sources, public and private—and the public designing patterns must be tailored to best serve each (private) environment.

Software creation requires iterations of analyzing, then designing, and then programming (which is really the lowest level designing). The best software design creation requires thorough understanding of all of the levels and how to best manage them. To clarify how the various types of mechanisms fit together, it's very helpful to understand that interaction *types* need to be *identified* in the *analyzing* efforts, and interaction *mechanisms* need to be *applied* in the *designing* efforts. Ultimately, the best software design creation requires being able to *feel* the mechanisms.

Ultimately, design patterns are standardized techniques for manipulating interaction variables.

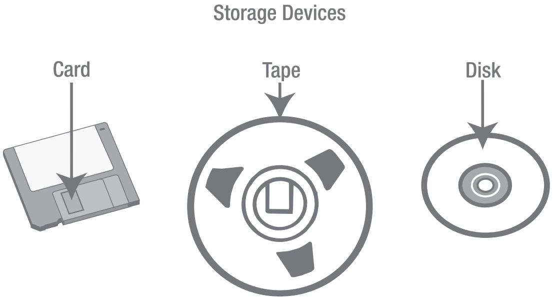
At a higher level, having a feel for combinations of all of these things, in combinations of circumstances, determines designing *strategies*.

Untangled Web: The Evolution of an Enterprise-Level Design (Chapter Three)

The open nature of the industry's evolution continually allows the better ideas to be built on—sometimes directly, sometimes only logically (through the lessons learned from them)—and to often gain popularity—which then feeds good ideas, and so on. This is based on ongoing occurrences of practical usage, adjustments for deficiencies, discussions and judgments, and comparisons of effectiveness. Overall, it fosters the best designs.

All software applications have the same basic characteristics. Most basically, to help people accomplish things, an application must interact with its users, manipulate information for them, and save relevant information for later usage. The common **3-tier architecture** design is of user presentation, core logic, and data storage. The separation of these most-basic functions is another example of serving flexibility.

Applications have always required ways to communicate with their users. The avenues for this, and the methods for managing them, have expanded and become much more effective over the years. The mouse has become very familiar, with the abilities that it provides to point and click, drag and drop, and scroll. Before these were only the abilities to type commands and fill in the blanks, with the keyboard.



Applications have also always required ways to store data for later usage. These avenues and their management have likewise, independently, expanded and become much more effective. Hardware for this has built from cards, to paper tape, to magnetic tape, to magnetic disk—logically, a tape roll turned on its side—to optical (laser-based) disk; these have increased storage density, and practical capacity, all along the way. Cards could accommodate only fixed-length records; beginning with tapes, variable-length records were possible. The only storage method possible until disks came about was sequentially accessed files. Disks enabled indexed (effectively randomly accessed) files and databases, which are combinations of indexed logical files. (They can be physical files, but databases are most efficiently managed as physical partitions of a single file.)

And, of course, applications have always been mostly thought of as what they do. At their heart, their processing is various calculations—comparisons, searches, extractions, duplications—and combinations of all of these. The basic functions have always been the same. But the ever-increasing speed of *hard*ware and the ever-increasing flexibility of techniques continue to make more and more applications practical.

Also independently, only relatively recently have the client/server and added internet concepts come about and become widespread.

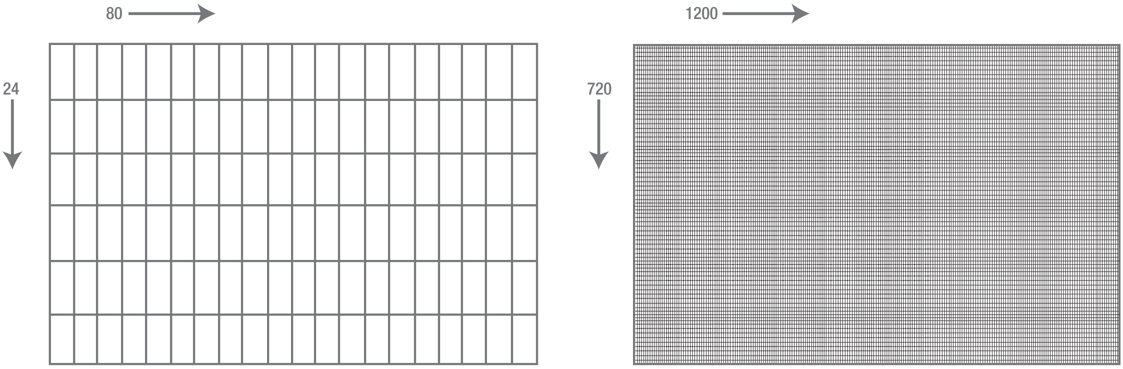
Early on, there was one user interface per computer; it was very similar to a typewriter, so the interaction was one line at a time. This interface could be shared by multiple users, taking turns. Eventually, the interface became a screen, but the interaction was still by single line. The idea of multiple interfaces per computer was made possible by the idea that the computer would give each interface a turn; so, many screens were *part of* one central computer.

OPERATING SYSTEMS

Computers have long separated more computer-dedicated operations and more user-dedicated applications with an **operating system** (**OS**). There have been and are many operating systems. An OS is actually just a set of programs—some of which the computer is always running. An OS has one main program, the **kernel**, and several extension programs, **system programs**, that the kernel runs only when needed.

The simplest explanation of how the kernel gets running is that, when the computer is turned on, it looks at a constant location on its startup disk—the **boot sector**—for the variable location of the OS kernel and starts it. Applications are started as requested by the user, immediately or by schedule, and effectively further extend the OS. Giving each interface a turn is a function of the OS.

Display Examples



Fixed Characters

Flexible Graphics

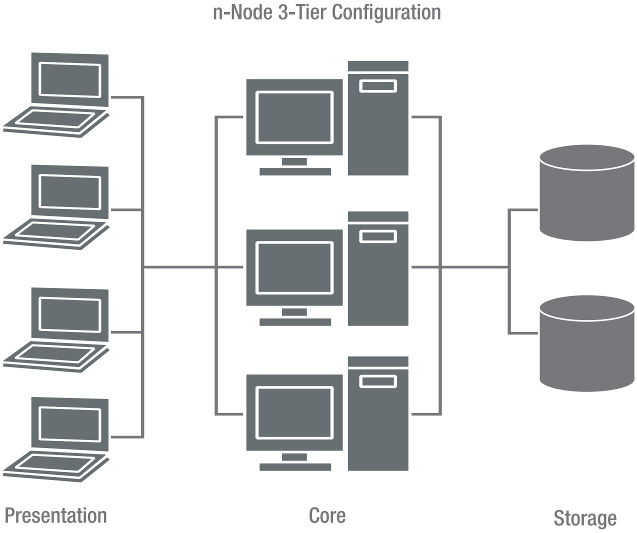
Also, user interfaces grew to include a degree of formatting. Each of the positions on each of the lines of the screen became identified by its coordinates, so many pieces of information could be addressed in one interface. Eventually, a **graphical user interface** (**GUI**) was made possible through the much finer coordinates of picture elements (**pixels**).

Early on, computers were *huge*—taking up to entire warehouses. Over time, multiple-user computers became *much* smaller, even as they became much more powerful. Again independently, the idea of one user interface per *very* small (personal) computer grew; over time, these computers went from having very little power to having more power than the warehouse-sized computers of decades earlier. Their growing capabilities spawned the idea of making the large computers even more powerful by using them in place of screens and shifting some of the overall processing to them. So, each of the small computers became a *client* of a large computer, which became a *server* of the application. Most directly, the client and the server are not the computers but corresponding *software* on the computers.

The client and the server are software.

Eventually came the idea of connecting servers, to form a *network*; this *distributed* processing and storage among many computers. Later came the idea of connecting networks, to form the *internet*. By comparison, this made an unconnected network an internal network, or int*ra*net. The internet has both public and private networks; a public subset of the internet, the **World Wide Web** (**WWW**), is commonly *referred to* as the internet or "the web". Then, another version of both mouse and keyboard client interfaces, the network browser, even gave the internet (and intranets) interaction capabilities that weren't available through any other configuration. Although networks can transmit any types of files, the web is commonly thought of in the context of *viewing* through a browser.

The 3-tier design can be applied on a single computer or a client/server configuration, using the client for the presentation tier and the server for the core (middle) tier and the storage tier. The storage tier can even have a separate server, to shift some heavy processing; this is a level of indirection. (In a context focused on the core/storage relationship, the software that manages the usage of this type of server has been called **middleware**.) A computer that is used as a server can actually have an application that has existed for a long time—a **legacy** system—and *might* still have users.



Further, there can be *multiple* storage servers. That configuration can even be used for multiple legacy systems, effectively combined by the core tier. Even further, the design can be applied to distributed processing, as multiple *core* servers. So, a single tier can have many instances; this is commonly called **n-tier**, but it's still the 3-tier design, just with *n* nodes. The term **enterprise software** refers to an application for shared data, typically among employees of a company; this can actually be applied to any shared configuration, but the term was created for the 3-tier design on the client/server configuration, because of its complexity.

Occurring at the same time as object orientation, and adding to its uses, all of these technologies are also becoming more organized—and more extensive. The newer ones have increasing infrastructure, fairly standardized, built by relatively few organizations. While this significantly intensifies each learning curve, at the same time, it allows designers and programmers to have more and more examples of (usually) very solid software to study and, to varying degrees, pattern after, because the infrastructures are built by experts—developers who are closest to the origin of the technologies or developers who have specific insights. So, this aspect is a compensation; it diminishes the learning curve.

Increasing infrastructure both intensifies and diminishes each learning curve.

The open nature of the industry defines the industry as a whole as the *collection* of experts. On the other hand, the occurrences of lack of simple philosophy throughout the industry cause this expertise to not be distributed as thoroughly as possible. Further, they cause varying degrees of confusion—which then feeds errors being built on errors. But understanding the various aspects of each design, and how the designs are related, cuts through both of these issues.

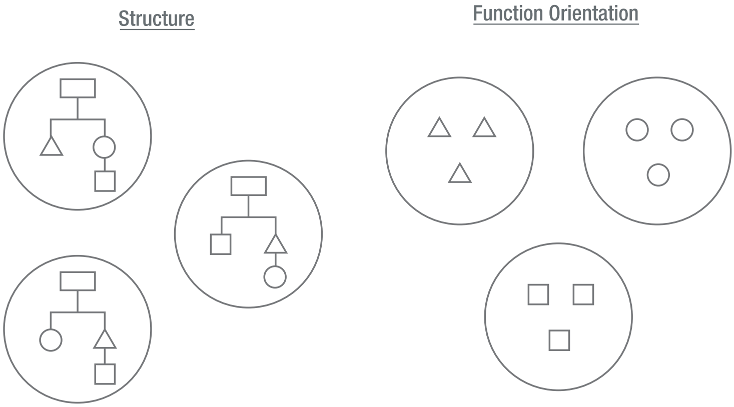
For complete context, for a fundamentally thorough demonstration of examples of designing and programming, and to serve simple philosophies, it's very important to understand how user and storage interfaces have been implemented in various hardware and software configurations—and the reasons behind the design decisions. And it's especially important to examine how the later approaches were built on the older ones, sometimes directly, sometimes only logically.

Then, all of this provides strong feeling for possibilities.

Derived Simplicity (Part II)

Structured software is very application specific. Object-oriented software is less application specific and very function specific. In fact, structured software could comparably be called **application oriented**. And, looking deeper, objects are actually *logical sets of functions*; object-oriented software could, more completely, be called "function-set oriented" or, more fundamentally, **function oriented**. (Calling the software function oriented is the subject of some debate, because objects have data outside of functions also, but these are separated into function sets by *functionality* and shared by the functions and separate executions of the same function.) And each complex logical function is still structured, in multiple code functions. For conceptual clarity (and fundamental benefits throughout the concepts), function orientation is the name that is generally used in this book.

Structured software could comparably be called "application oriented". And object-oriented software could, more fundamentally, be called "function oriented"; each function is still structured.



Function Organizations

Structure

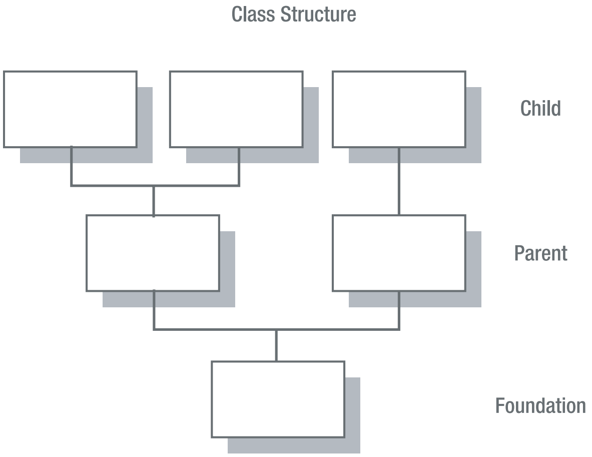
Function Orientation

Function-oriented thinking sees application-oriented thinking as fragmenting logical sets of functions and, additionally, unnecessarily duplicating many of those fragments. Without the duplication of fragments, each function—data and processing—is *owned* by a set; other sets must interact with that set to use the function. Developers must know what parameters each function requires and what *all* of its *external effects* are; but this has always been true of utilities. (And, actually, the best designs *separate* any side effect into its own function, allowing selective combination.) The most organized function-oriented approach is to have a database of the functions with "uses" and "is used by" references—as part of an integrated development environment (IDE). These added efforts allow each logical function to occur physically only once, so changes are centralized—and distributed by the computer; ongoing development is facilitated, so the added efforts are an investment in the future.

The main point of function orientation is easing of the ongoing development effort.

Functions *are* the heart of applications, so building *multiple* applications is more *organized* with a well-organized function set network (as is changing a single application). The key is that each set needs to be well defined, because poorly defined sets actually make changing them more complex. This means that each set should have a cohesive *purpose*, and a fairly limited one. A good guide is that a piece of code that benefits from a comment can actually be separated into its own function, or even its own function set, with a name that serves as the comment. A very straightforward example is that input and output for a particular record type should be in a dedicated set; these create what are known as **data objects**. It can be seen that when sets are too large, the structure of the functions dominates the network of the sets—so the system is actually application oriented.

Whereas application-oriented sets perform deep processing on a narrow spectrum of data, function-oriented sets perform shallow processing on a wide spectrum of data. In fact, to application-oriented veterans, function-oriented code can look unreasonably simplistic, so its jumping from module to module can seem pointless. But the limited purpose promotes function independence and, therefore, code swapability; this allows more possible recombinations of sets and, therefore, more possible applications, so that no application is overcommitted. An advantage of a whole logical function over just pieces of application-oriented code is that it is a complete unit; it ensures operational integrity. The ideal for building multiple applications is for there to be very little new design necessary for each new application; that would be like prefabricated development (with adjustable components). That's the whole idea: there's no *magic* in objects.



For perspective, it is very important to remember that both application-oriented and function-oriented systems are only logical views of a system. *The actual system occurs at execution, in a sequence, and that is essentially the same for both.* (This understanding is also very important for debugging and optimizing efforts.) Additional perspective requires some design-level vocabulary. It includes understanding that the word "object" is less-frequently appropriate in function-oriented development than the word "class". (This overusage is partly a product of overfocus on "object" orientation.) A class defines an object; it's a classification. So, mostly, a class is a function set in its definition form, and an object is a function set in its usage form. Further, in various ways, a class is thought of internally, and an object is thought of externally. Any object—real-world or software—can easily be *represented* by another object—a symbol or a piece of code—but that's external; that object is still *served* by the internal.

Both application-oriented and function-oriented systems are only logical views of a system. The actual system occurs at execution, in a sequence, and that is the same for both.

Now, a class structure is commonly referred to as a hierarchy, but it isn't really a hierarchy. In that representation, the higher levels wouldn't manage the lower levels; in fact, the reverse would be true. The clearest representation of a class structure is a literal *tree* structure (an upside-down hierarchy), which denotes that descendants are built on top of ancestors; descendants are more defined. Actually, the tree structure points out that they are really *a*scendants; this is a word that will be used from here on. A tree and a hierarchy (a near-literal *pyramid*, or a literal *root* structure) are mirror images, so they have a lot in common, and that causes confusion between them, but it's fundamentally important to understand the differences—and their effects. A child class is commonly referred to as a **subclass**, and a parent class is commonly referred to as a **superclass**, but these names, again, are compatible with a hierarchy. (Just to check the thinking, standard set theory can look at sets from two different points of view: abstract [from above] and concrete [from the side]. It's the abstract view, which serves theorization [digging to foundations], that says that a variable encompasses all of its constants. The concrete view, which serves application [building functionality], says that a superset has all of the characteristics of its subsets. A class structure is analyzed from the top and viewed from the side.) Each class structure can simply be called a **family**, and each class has a **lineage**. Other appropriate terminology refers to a parent as an **extended** class and a child as an **extension** class. Also appropriate, extending can be referred to as **specializing** or **growing**. The structure of a class and its ascendants is a **branch**; alternately, a descendant (ancestor) can be seen as a **platter** (or platform) for its ascendants. It's important to note that this is a static system, because each child can be in only one part of the family at a time. Further, it's a *doubly* static system, because each child can be in only one part of the family *ever*.

A very important point (which is very rarely made) is that, with one exception, *any one object is defined by more than one class*. The common explanation is as follows. An object is an **instance** of a class; the declaration of an object is an **instantiation**. Any class can be a parent class and have further definition in its child class, which **inherits** its parts (data items and functions); this includes that functions can be *re*defined. (The fact that, in the real world, a child is born of two parents can be supported also. The other parent is most appropriately identified as a step of the designing process.) But the accurate explanation of instantiation further accounts for the fact that each class inherits its parts; each object is an instance of its *lineage* (a class and all of its ancestors). In some environments, there is a most-basic generic class, which all other classes ascend from; any object of this class is the only type that has a lineage of just one class.

x = Why: Interaction Algebra for Analyzing and Designing (Chapter Four)

Beyond class network characteristics—from family structures to individual functions—the cooperative processes between classes in function-oriented systems can be very cumbersome and (therefore) difficult to communicate from one developer to another. So, comfort with a design among a team of developers (of all levels) spreads slowly—and discomfort (of varying degrees) lingers for the life of the system. This is a fundamental disabling factor in initial and continuing development. But it doesn't have to be.

It's frequently valuable to convey information with visual organization. Commonly, it's believed that text cannot be organized very visually and that the best way to present information visually is with diagrams. Specifically, for cooperative processes in function-oriented systems, the Unified Modeling Language (UML), especially its "structure" and "sequence" diagrams, is popular. Some problems with diagrams, however, are that they often take more space than text for the amount of information that they contain, and they are comparatively time-consuming to produce well.

Of course, text *can* be organized visually, in many ways. Further, when it's possible, the most efficient way to convey information visually is with mathematical notations. And an important part of the simplification movement is minimization of separate documentation effort.

An extremely clarifying view of class interaction mechanisms comes from a very structured, very concise technique for analyzing interactions and designing mechanisms, with a notation called **interaction algebra**. This concept grew out of the fact that there are several common characteristics between interaction mechanisms and standard algebra, including core manipulation concepts.

The first obvious commonality is the fact that, in both, flexibility is served by a mechanism of variability; in both, this adds a level of complexity. Further, the interaction algebra notation serves to identify the user of the mechanism, just as algebra isolates a variable. Interaction algebra can be seen as an inheritance (an extension) of algebra, and the properties of the notation show themselves to be so straightforward as to clearly apply to databases (data table interactions) as well.

Design Holes

The fact that interaction algebra is oriented to the user of any mechanism focuses attention on the *purpose* of the mechanism, which is an extremely important factor for analyzing and designing, especially as it applies to classes and their interactions. And a fundamentally important feature of interaction algebra is that, as with any structured approach, it eliminates design holes that free-form representations let slip by.

Interaction algebra is based on equations, just as is standard algebra. But it's geared to software entity interaction mechanisms; it has a specific format for its expressions. Further, it has two sets of syntax: one to represent classes and the other to represent database tables.

Live and Unscripted: Object Animation, a Clearer View of Automation (Chapter Five)

Even function orientation is a design that doesn't address the largest aspect of automation. Aside from the orientation to structures or functions, systems can be oriented to processes or data. In that larger context, function orientation, in its common form, is really a function-oriented procedure design. And, of course, common function-oriented languages are extended **procedure-oriented** languages. This is because each step of any process is commonly seen as procedural.

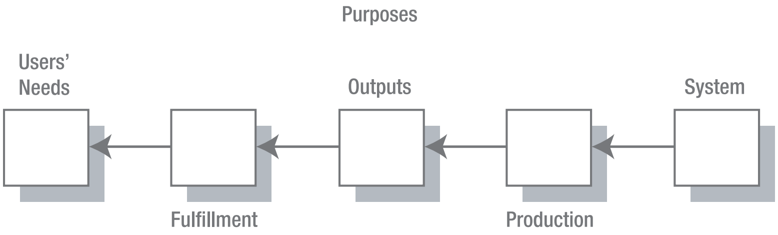
Inside of each function, code is mostly procedure oriented. The simple concept of a function set is geared to answering the "whats" of a system, ultimately thought of as a set of data. Digging deeper, the function set's role (responsibilities) in the system must be well understood; this is geared to answering the "hows" of the system, ultimately thought of as the functions that *support the data*. So, ultimately, the data is the priority of function orientation. But the code—the procedure-oriented code—becomes more geared to answering the hows than the whats, making the procedure the priority. This causes a conflict.

In software development, designing is more from the users' point of view, to get the computer to serve *what* the users need. With procedure orientation, programming has been more from the computer's point of view, to serve the computer also. Complete software development has always required thinking in both directions at once. This is a huge complication and, therefore, a huge bottleneck in the software development timeline. But it doesn't have to be.

Of course, function orientation is designed to ease the software development effort, but the problem is, in essence, that function orientation goes only halfway. And, generically, going halfway often causes complications and extra efforts. Extending the function-oriented design to include the **data-oriented** equivalent of functions, which continue to address *the whats and then the hows* of a system, completes the concept. (For now, it's simpler to continue to call the entities functions.) With a function-oriented *data* design, the flow of thought both outside and inside of sets is in the same direction.

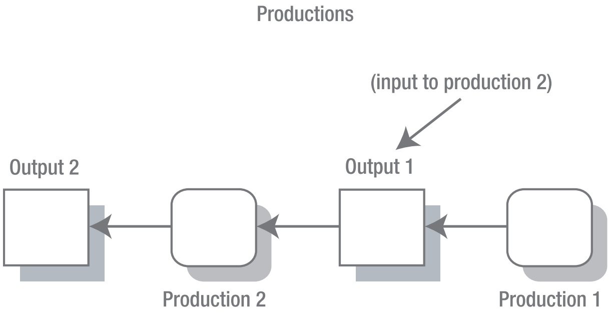
Data Orientation

Simply put, each system exists for a reason—more accurately, a set of reasons. This fact easily translates to the fact that any system is geared to whats—individually or in combination; that's the system as a whole and, with the complete concept of function orientation, at any level. Even each function set exists for a set of reasons.



Thinking this through, the reasons why any system really exists are to produce its outputs, which are whats. This isn't *all* of its outputs, because some of them are an intermediate form to facilitate further functions, but the purposes of any system are to produce its outputs. This reinforces the fact that data orientation is more important than procedure orientation. And the **product** is the focus of data orientation.

So, data orientation isn't about just *any* data, because a system has goals—to produce its products, and data orientation is about data that is the focus, or *subject*, of each step along the way. It could also be called "product orientation", which can be derived to "purpose orientation" or, further, to "goal orientation". Or it could be seen, probably controversially, but definitely ironically, as "*sub*ject orientation". But the most straightforward and comprehensive name is data orientation.



Of course, the system's inputs are important. Ultimately, a system produces all of its outputs from its inputs. More directly, the system produces some of its outputs from intermediate outputs. Really, system inputs are one form of intermediate outputs. This fact is made clear by the understanding that any system input was an output of another system (or a person). In data orientation, an input is seen as a **resource**.

The necessary derivative of focus on the products of a system is focus on the items that are necessary to assemble them—their **components**. For several years already, fourth-generation and CASE tools have commonly handled outputs in a data-oriented manner, with lists of components and their characteristics. It's extremely helpful to be able to extend that handling to account for value changes, but the limiting factor is that the value changes seldom occur in the order of the components of the output, so this aspect of these tools still requires coding and, therefore, (awkwardly) has still been procedure oriented. A data-oriented language enables this extension.

Value changes seldom occur in the order of the components of the output, so they have always been procedure oriented.

Effects on Development

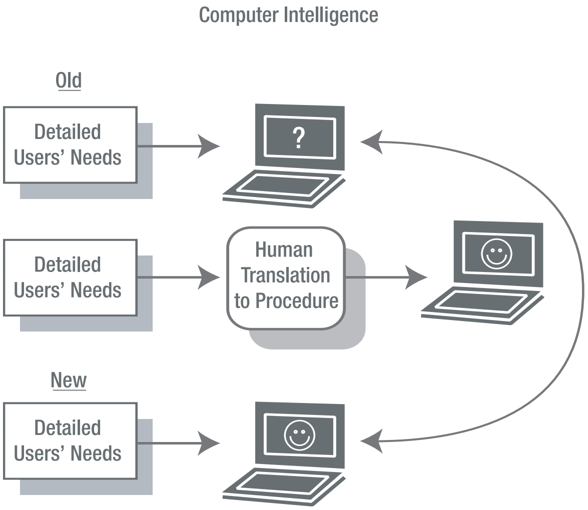
The most far-reaching effect of data orientation is the fact that it allows a single mind-set throughout the entire development process. It keeps focus on all of the possible components of the system, at any level; the conditions under which each component is created are secondary to that. This focus drives development, making all efforts clearer and bringing all pieces together with less effort.

Data orientation keeps focus on all of the possible components of the system, at any level. The conditions under which each component is created are secondary to that.

Effectively, the process of software development becomes much more like the process of *idea* development. *This effort is fundamentally more natural and intuitive.*

By the way, because testing is oriented to system purposes, data orientation even facilitates that. No longer is it necessary for developers to shift gears again, to change mental direction, to flip back to the other side of the development coin.

The simplification of the development process is completed with a data-oriented *programming* language, which creates an even more extensive philosophy. Ultimately, the computer then interacts completely from *any* user's point of view; software developers are users (of software development software), and programs are these users' specifications.



Data-oriented translators simply translate specifications that come from a different point of view. This philosophy elevates the concept of the intelligence of computers. It makes procedure-oriented computers seem dumb by comparison. The dumb computer is just a (processing) servant; the smart computer is a (data analyzing) partner and then a servant.

Whether for designing or for programming, the heart of a data-oriented language is the specification of how related pieces of data interact with each other, in the proper sequence. This task is most simply described as **data relationship management**, or **DRM**, most easily pronounced "dream". And a DRM language is made possible through a few observations about procedure-oriented languages, built on all of the observations of the earlier parts of this book.

With a strong understanding of function orientation, DRM languages are easy to adapt to. They have nearly all of the same components as function orientation, with added capabilities, so it's easy to feel the reorganization.