*917_Ch01_CMP_3 2/21/02 1:34 PM Page 1

CHAPTER 1

The Starting Point

In This Chapter:

- Finding the right starting point
- Contrasting the application-centric and architecture-centric views
- Practicalities and pitfalls of high reuse
- Patterns to enable reuse
- The metamorphic superpattern
- Using semantics to activate knowledge

The first and most critical decision we (or an enterprise) will make is in defining and embracing a starting point for our approach to product development. At the software project level, when organizing the user's requests, examining the available technology and converging on an approach, at some point the leader(s) must commit and begin the effort.

Many of us have broken open the technology boxes, jotted down a short list of user requests and *burst* the features from our desktops. If we did a great job, the user wanted even more. If we did an inadequate job, the user wanted rework to get their original requests fulfilled. Either way, we found ourselves sitting before the computer, thinking about a rework or rewrite, and consequently considering a new starting point.

The real question is: Where do we start? What is the *true* starting point, to eliminate or minimize the need for wholesale rewrite? Starting off on a deterministic, application-centric path makes us feel like we are in control. This might be the case until the user expands or changes the feature requirements and unravels our carefully crafted architecture.

Knowing Your Product—Software Defined

Our most appropriate starting point involves defining our understanding of "software." Everyone has a presupposition of what software is and what's required to develop it, and it's highly likely that we have different perspectives. Consider the following definitions:

- *Software* is simply textual script used to define structure and behavior to a computing machine. It is beholden to a published language definition (such as Visual Basic, C++, Perl, etc.) and requires interpretation within its language domain.

- *Program* is the compiled form of software. A language compiler will interpret the software, build machine-ready instructions, bundle it together with any required components, and write the results to a disk drive as an execution-ready program file. Just-in-Time (JIT) compilers may write directly to CPU memory and execute.

- *Application,* for the majority of developers, refers to a customized program. Software embedded in the program addresses specific user-defined entities and behaviors.

Application = Program?

For most software developers, the preceding definition of program encompasses a very broad domain. In typical developer's terms, application means "how we apply the software," or the physical code inside the program. They are one and the same. If the application is a custom program, then the database requirements, third-party components, and other supporting programs are part of a *solution.* A custom program pulls together and integrates other programs or components, rarely stands alone, and is highly and often directly dependent on deep details in other parts of the solution. A single change in a database definition, or a web page, or even an upgrade to an operating system parameter can initiate rebuild of the custom programs on the desktop or elsewhere.

Custom programs are often beholden to the tiniest user whim or system constraint.

Ideally, we want to service user requests without changing software or rebuilding the program. This is only possible if we redefine application to refer to user-required features, not program code. This definition serves to separate programs *physically* and *logically* from applications.

We'll want a program to address architecture and technical capabilities, while an application will address user-requested features. Our goal is to use the program as a multiple-application highway, but this requires us to place the entire realm of technology within the program on a leash. The next section explains how to embark on this journey.

Application <> Program!

Overcoming the Application-Centric View

We must separate our software from the volatile realm of user-driven application logic.

The *application-centric view* is what most of us understand and practice. It is *requirements driven,* meaning that nothing goes into the software that is not specifically requested by an end user. Anything else inside the software is viewed as necessary for support, administration or debugging, but not "part of the program" as defined by the end user. Therefore, these components could be expendable at a moment's notice. What I'll loosely call the *Common Development Product* depicted in Figure 1-1 is the natural outcome of the application-centric view.

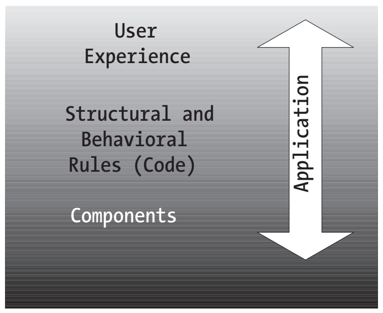

Figure 1-1. The Common Development Product weaves the application into the program so the two are hopelessly entangled.

The Common Development Model shown in Figure 1-2 is the standard methodology to deliver application-centric products. It includes deriving business justifications, specification and formulation of feature descriptions, followed by hard-coded feature representations and structures to deliver the requested behavior. As such, software is the universal constant in feature construction and deployment. Any flaw in a design requires software change and product rebuild. Any flaw or change in requirements is even more dramatic, often initiating redesign. To accommodate change, the end result is always a program rebuild and redelivery.

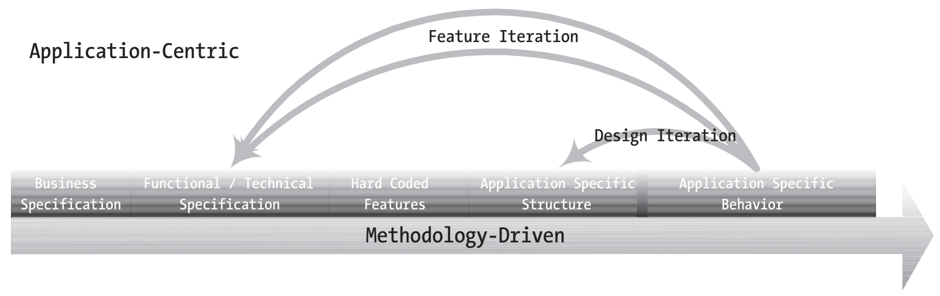

Figure 1-2. The Common Development Model for delivering application-centric programs is great for solution deployment but flawed for software development.

User requirements must justify every line of code.

Software developers at every level of expertise invoke and embrace the application-centric view. It's where we are most comfortable and sometimes most productive. However, application programs require constant care and feeding, and the more clever we get within this model, the more likely we will stay hooked into the application well into its maintenance cycle. The model is ideal for building applications, but not *software*. It is a highly flawed software development model and suffers from the following problems:

- Inability to deliver the smallest feature without a program rebuild.
- Necessity to maintain separate versions of similar software, one for quick maintenance of the deployed product, and one for ongoing enhancement.
- The laws of chaos impose a cruel reality: that an innocuous, irrelevant change in an obscure corner of the program will often have highly visible, even shocking affects on critical behaviors. We must regression-test *everything* prior to delivery.
- There is little opportunity for quick turnaround; the time-to-market for fixes, enhancements, and upgrades is methodology driven (e.g. analysis, design, implementation, etc.) and has fixed timeframes.
- Inability to share large portions of software "as is" across multiple projects/applications.
- We keep large portions of application functionality in our heads. It changes too often to successfully document it.
- Rapid application delivery is almost mythological. The faster we go, it seems, the more bugs we introduce.

The list could go on for many pages, but these practical issues often transform software development into arduous tedium. The primary reason we cannot gain the traction we need to eliminate the repeating problems in software development is because, in the application-centric approach, we cannot get true control over reusability. If we can reuse prior work, the story goes, we can solve hard problems once and reuse their solutions forever.

And these are only a few of my "favorite" things.

Desperately Seeking Reusability

If we can deliver applications without modifying core software, we will have fast, reliable deployment and happy users.

After completing one or more projects, we earnestly scour our recent efforts for reusable code. If object-oriented software is about anything, it's about solving a problem once and reusing it. *Reusability* is the practice of leveraging prior work for future success. We don't want to cut and paste software code (snippets or large blocks) from one project to the next, but use the *same* software, with no changes, for multiple projects. Thus, a bug fix or functional enhancement in a master code archive can automatically propagate to all projects.

The (Im)practicality of High Reuse

The term reusability *always elicits notions of* high *reuse of vast percentages of prior work rather than snippets or widgets.*

High reuse can be an elusive animal. We can often pull snippets or ideas from prior efforts, but rarely migrate whole classes or even subsystems without significant rework. Thus, we mostly experience reuse only within the context of our own applications or their spinoffs.

Here's the real dilemma: every software company boasts reuse as a differentiator for faster time to market, and every customer wants and expects it. However, while customers are happy (even demanding) to use work from our prior customers, they don't want *their* stuff reused for our future customers.

Software development efforts are very expensive, and no customer will let us give away their hard-won secrets for free. This creates a significant ethical dilemma, where we can't tell customers we'll start from scratch or we'll likely not get the contract. But if we use their business logic on the next round, probably with a competitor, we open ourselves up to ethical questions, perhaps even litigation.

Successful reuse requires a way to build software that leverages vast amounts of prior work without reusing application logic. In the application-centric model, this is impossible.

Many of us sincerely believe that high reuse is attainable, and some of us look for it under every rock, almost like hidden treasure. Amazingly, high reuse is right in front of us, if we *don't* look in the application-centric model as our initial starting point.

High reuse is technically and ethically impossible in the application-centric model.

Harsh Realities of Application-Centric Reuse

Many development teams take a quick assessment of the language, third-party widgets, and interfaces and then take off in a sprint for the finish line. We create hard-wired, application-specific modules with no deliberate plan for reuse. Looking back, we realize our mistake and rightly conclude that we picked the wrong starting point. We reset, believing that an application-centric *methodology* is needed, such as the one shown in Figure 1-2. More discipline creates a better outcome, right?

> **NOTE** *End users believe that undisciplined developers create havoc, but the native development environment breeds mavericks. Methodology and lockstep adherence may give users comfort, but it's an illusion.*

Methodologies abound, where companies codify their development practices into handy acronyms to help developers understand one thing: methodology is policy. Industry phenomena such as eXtreme Programming (XP) work to further gel a team into lockstep procedural compliance. Online collaboration in the form of Peer-To-Peer (P2P), Groove, and so on helps a team maintain consistency. However, if the team does not approach the problem from the correct starting point, the wheels still spin without traction.

Methodology Mania

Many companies expend enormous effort codifying and cataloguing their software and technology construction/deployment cycles, primarily to reduce the risk of project failure or stagnation. The objective is to congeal the human experiences of many project successes and failures so new projects can race toward success without fear of repeating the mistakes of the past.

Such companies expect every technologist to understand fully the methodology and the risks of deviating from it. Common prototypes include the basic stages of requirements gathering, design, development, deployment, and maintenance.

As for execution, two primary models exist: the *Waterfall*, where each stage of development must close before the next one commences, and the *Iterative*, where each stage can swing backwards into a prior stage. Waterfall methods have fallen out of favor as being unrealistic. Iterative methods require more architectural structure to avoid unraveling the software and the project effort.

NOTE *Perot Systems Corporation has codified their development methodology into an innovative acronym:* ***R****equirements,* ***A****nalysis,* ***D****esign,* ***D****evelopment,* ***I****mplementation,* ***O****peration (RADDIO).*

Process alone is not enough. If the methodology is flawed, or the approach is wrong, the outcome is still unacceptable.

If we tap the application-centric view as the only realm of understanding for project ideas and answers, we'll always pick another starting point within this realm. We may try giving more attention to *application detail* in the beginning of the project to yield better results at the end of the project. We may standardize our approach with detailed methodologies, still to no avail. We find ourselves changing software every time the end user speaks, and only see redesign and rewrite in our headlights. Our vision is often blurred by fatigue, fire-fighting, and support and maintenance concerns. We lack the time, funding, and support (perhaps experience) to think the problem through. We're simply victims of the following misconceptions:

Misconception #1: High reuse is generally attainable.
Reality: In an application-centric model, *high reuse is impossible.* Try and try again, only snippets remain.

Misconception #2: Application code should be reusable. All that software we built for *Client XYZ* and we can't use it somewhere else?
Reality: *Application-centric code is never reusable.* And, even if we could reuse it technically, would we risk the ethical questions?

Misconception #3: We can cut and paste software "as is" from project A to project B and keep up with changes in both places.
Reality: Don't kid yourself.

Misconception #4: Leaders, users, clients, and decision-makers share your enthusiasm about reuse.
Reality: They might share the enthusiasm, but doubt its reality so cannot actively fund it or support it.

Misconception #5: We can find answers for reusability questions within the application-centric model.
Reality: The only answer is that we're looking in the wrong place!

Application-centric development is a dream world, and envelops us like a security blanket, creating the illusion that nothing else exists. If we remain a prisoner to the computer's rules, protocols, and mathematical constants, and assume that our chosen software language is *enough,* we'll never achieve high reuse.

We must overcome these artificial constraints with higher abstractions under our control. If we use software to enable the abstractions, we will forever separate our software from the maelstrom of user requirements, and every line of code we write will be automatically reusable. Read on to discover how to achieve this goal.

Revealing the Illusion of Rapid Application Development

We think in terms of rapid development, but what the user wants is rapid delivery.

The terms *rapid* and *application* cannot coexist in the application-centric delivery context. Once we deliver an application, we'll have significant procedural walls before us, including regression testing and quality assurance, all of which can sink the redelivery into a quagmire. These processes protect us from disrupting the end user with a faulty delivery, but impede us from doing anything "rapid" in their eyes.

The application-centric model is tempting because we can use it to produce 80 percent of application functionality in 20 percent of the project cycle time (see Figure 1-3). This effect leads end users to expect quick delivery of the remainder. Unfortunately, we spend the remaining 80 percent of the cycle time on testing and integrating against problems created from moving too quickly. While this model apparently delivers faster, it's really just an illusion.

Rapid Application Development

Developers want ways to burst features from their desktops. All the user cares about is turnaround time—how long between the posting of a request and the arrival of the feature on *their* desktops?

The difference in these two concepts creates a disparity, sometimes animosity, in the expectations of the two groups. Developers might receive the request and turn it around rapidly, but the quality-assurance cycle protracts the actual delivery timeline. Of course, we could forego the quality assurance process altogether, and risk destabilizing the user with a buggy release (a.k.a. the "Kiss of Death").

TIP *The concept of Rapid Application Development is inadequate. We must think in terms of rapid* delivery, *encompassing the entire development and delivery lifecycle.*

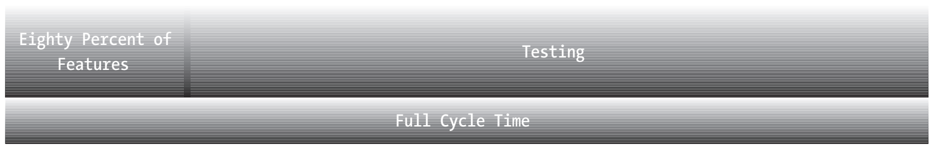

Figure 1-3. The application-centric model automatically protracts testing.

A prolonged project cycle soon bores our sharpest developers, and they find a way to escape, sometimes by leaving the company. Once leading developers leave, the workload only increases for the remaining team members. They must complete the construction and testing of the lingering 20 percent of features, and work slows further.

Many such software projects finish up with less than half of their original core development staff. People leaving the project vow never to repeat their mistakes.

Some companies have a cultural joke, that the only way to leave a project is to quit the company or go feet first!

NOTE *Developers don't like to compromise quality because we take pride in our work. Because users tend to make us ashamed of brittle implementations and prolonged project timelines, swift* and *accurate delivery is critical, and its absence is sometimes a personal issue.*

Because the application-centric model usually focuses on features/functionality and not delivery, the team saves the "final" integration and installation for the end. This too, can be quite painful and has many times been a solution's undoing. Teams who can recover from program installation gaffs breathe a sigh of relief on final delivery.

However, another trap is not considering the product's overall lifecycle. We may not realize our *final* delivery is actually the *first* production delivery, with many more to follow. We recoil in horror when we see the end user's first list of enhancements, upgrades, or critical fixes. This, too, can be a solution's undoing.

The application-centric view, and the Common Development Model illustrated in Figure 1-2, are fraught with the following major, merciless pitfalls, each threatening to destabilize the effort:

Software's so-called "final" delivery is actually the first in many redeliveries for the duration of the product's lifecycle.

Pitfall #1: Inability to meet changing user requirements without directly modifying (and risking the destabilization of) the software program(s)

Pitfall #2: Assuming that the "final" delivery is just that, when it is usually the first in many redeliveries (starting with the first change request).

Pitfall #3: Wiring user requirements into the software program, only to rework them when requirements change, morph, or even disappear.

Pitfall #4: Taking too long to finish, and risking change in user requirements before completing the originally defined functionality (a moving target).

Pitfall #5: Losing valuable team members because of boredom, increasing risk, and workload on the remaining team.

Pitfall #6: Encountering an upgrade, enhancement, or (gulp!) misunderstood feature that cannot be honored without significant rework, even redesign.

These pitfalls are visible and highly repeatable in development shops worldwide. Rather than ignoring them or attempting to eliminate them, let's embrace them, account for them, assume they will run alongside us, and never go away, *because they won't.*

The problem is not the developer, the methodology, the inability to deal with pitfalls, or even the technology. The problem is *systemic*; it is the application-centric view. The only way to escape this problem is to set aside the application-centric view *completely*, and forsake the common development model *outright*. We'll still use them to deploy features, *but not software*! However, we cannot toss them out without replacing them with another model—the architecture-centric view.

Embracing the Architecture-Centric View

The *architecture-centric view* depicted in Figure 1-4 addresses technical capabilities before application features. These include the following (to name a few):

- Database and network connectivity
- General information management
- Screen rendering and navigation
- Consistent error control and recovery
- Third-party product interfaces

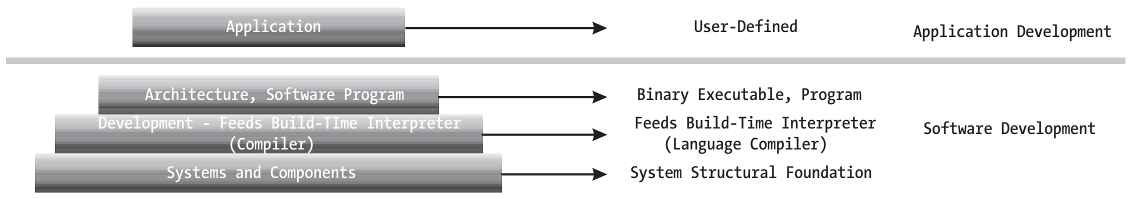
Application
User-Defined
Application Development
Architecture, Software Program
Binary Executable, Program
Development - Feeds Build-TIme Interpreter (Compiler)
Feeds Build-Time Interpreter (Language Compiler)
Software Development
Systems and Components
System Structural Foundation

Figure 1-4. The architecture-centric model prescribes that the software program (the binary executable) is oblivious to the user's requirements (the application).

However, these are just a fraction of possible capabilities. This view is based almost solely in technical frameworks and abstractions. Where typical methodologies include the user from the very beginning, this approach starts long before the user enters the picture, building a solid foundation that can withstand the relentless onslaught of changing user requirements before ever applying them.

This model is a broker, providing connectivity and consistent behavioral control as a foundation to one or more applications. It separates software development from application development. It is the most robust, consistent, scalable, flexible, repeatable, and redeliverable model.

Apart from supporting the user's features, what about actually testing and delivering the product? We'll need infrastructure capabilities for dynamic debugging and troubleshooting, reliable regression testing, seamless and automatic feature delivery, and the ability to address enhancements, fixes, and upgrades without so much as a hiccup. While users want the benefit of infrastructure, they believe it comes for free (after all, we're professionals, aren't we?). We fool ourselves when we believe the infrastructure really is for free (doesn't the .NET Framework cover all that stuff after all?).

Architecture-centric software physically separates the application (user features) from the program (compiled software).

This approach requires us to define software in terms of elemental, programmable capabilities and features in terms of dynamic collections of those capabilities. For example, say a customer tells me they'll require a desktop application, integrated to a database server with peer-to-peer communication. I know that these requirements are common across a wide majority of desktop programs. The architecture-centric approach would address these problems once (as capabilities) and allow all subsequent efforts to leverage them interchangeably. Conversely, the application-centric approach would simply wire these features into the program software wherever they happened to intersect a specified user requirement and would exclude them if the user never asked. Which approach will lead to a stronger, more reusable foundation?

NOTE *Design and construction should be the hard part; rebuilding and redeploying the program should be effortless. Enhancing and supporting application features should be almost fluid.*

The first and most important steps in driving application-centric structures and behaviors out of the software program include the following:

- Form capabilities using design patterns (discussed next) to radically accelerate reuse.
- Strive for metamorphism (discussed shortly) rather than stopping at polymorphism. Rather than shooting for rapid application delivery, go for *shockwave* application delivery.
- Define a simple, text-based protocol for feeding instructions into the program, including reserved lexical rules and mnemonics (such as symbols and keywords). This practice is discussed later in this chapter under "How Semantics Activate Knowledge."
- Design and implement the program itself in terms of elemental capabilities, each with programmable structure and behavior (see Chapters 2 and 3).
- Formulate power tools to establish and enable reusable capabilities, providing structural and behavioral consistency (see Chapter 4).
- Formulate frameworks around key capability sets, such as database handling, screen management, or inter-process/inter-application communication (see Chapters 5, 6, and 7).
- Formulate frameworks to accept external instructions (advanced metadata) and weave them dynamically into an application "effect" (see Chapter 4).
- Invoke separate development environments, one for the core framework (software development) and one for advanced metadata (application development).

Following these steps ultimately produces highly reusable software programs, where software code is reusable across disparate, unrelated applications. Doing so allows us to separate completely the volatile application-centric structures and behaviors into advanced metadata, with the software itself acting as an architecture-centric metadata broker. This promotes radically accelerated reuse coupled with rapid application *delivery*. Our end users will experience whiplash. Our deliveries will have a surrealistic fluidity.

To achieve these goals, we *must* separate the application structure and behavior from the core software program. Software will be an assembly of programmable capabilities. The application itself will appear only in metadata form, driving the software to weave capabilities into a fabric of features.

We tend to hard wire many user requests just to meet delivery deadlines. When we approach the next problem, the same technical issues rear their heads, but we have no real foundation for leveraging more than snippets of prior work. This is because those snippets are so closely bound to a prior custom application.

Therefore, to gain high reuse, we won't focus on the user's whims *at all*. We focus on harnessing the software language and interfaces to other systems, the physics as they affect performance and system interaction, the connection protocols, and the mechanics of seamless migration and deployment. We then build abstractions to expose this power in a controlled, focused form, rather than a raw, unleashed one.

The primary path toward building such a resilient framework is in understanding and embracing software patterns.

The principal objective is to understand, embrace, and harness the technologies, and enable their dynamic *assembly into features.*

Exploring Pattern-Centric Development

Once we master the language environment, productivity increases tenfold. However, once we master the discovery and implementation of patterns, productivity increases in leaps and bounds. This is because patterns promote *accelerated* reuse, so our productivity is no longer measured by keyboard speed alone.

DEFINITION Accelerated reuse *is the ability to reposit and leverage all prior work as a starting point for our next project, ever-increasing the repository's strength and reducing our turnaround time.*

The bald eagle's wings are perfectly suited to provide effortless flight in the strongest winds. Eagles have been noted gliding ever higher in the face of hurricane-force resistance. Could our programs have the aerodynamics of an eagle's wings, providing lift and navigation in the strongest maelstrom of user-initiated change?

DEFINITION *A* pattern *is a theme of consistently repeatable structural, behavioral, or relational forms. Patterns often tie together capabilities, and become fabric for building frameworks and components.*

Structural patterns *converge all objects into an apparently common form, while* behavioral patterns *emerge as frameworks.*

While a newer developer is still under the language environment's harnesses, and an experienced developer has taken the reins with brute force and high output, the pattern-centric developer has harnessed both the development language and the machine as a means to an end. He or she leverages patterns first (and deliberately) rather than allowing them to emerge as latent artifacts of application-specific implementations.

CROSS-REFERENCE *Gamma, et al. have built an industry-accepted lexicon for defining patterns and their rules in Design Patterns.*[1] *Developers with no exposure to these terms or concepts will have difficulty pinpointing and addressing patterns in their own code.*

Applying patterns, such as wrapping a few components, yields immediate benefits. Each time we bring an object into the pattern-based fold, we get an acceleration effect. If we apply patterns universally, we get a shockwave effect, not unlike an aircraft breaking the sound barrier.

DEFINITION *The transition from application-centric, brute-force output into accelerated reuse is something I loosely term* virtual mach. *It has a shockwave acceleration effect on everyone involved.*

Objects using both structural and behavioral patterns are able to unleash framework-based power. For example, one development shop chose to wrap several critical third-party interfaces with an Adapter pattern (the most common and pervasive structural pattern in software development). The shop mandated that all developers must use the adapters to invoke the third-party logic rather than invoking it arbitrarily. This practice immediately stabilized every misbehavior in those interfaces. The implementation of the adapter pattern was a step into the light, not a calculated risk.

CROSS-REFERENCE *Chapters 2 and 3 dive deeply into the patterns required to enable accelerated reuse.*

Every program has embedded creational, structural, and behavioral patterns in some form. Even without direct effort, patterns emerge on their own. Once we notice these *latent* patterns, it's usually too late to exploit them. However, even when we retrofit structural patterns, we catch wind in our sails, and see patterns popping up all over the place.

Another benefit emerges—structural patterns automatically enable interoperability because we expose their features to other components at a behavioral pattern level, not a special custom coding level. New and existing components that were once interoperable only through careful custom coding and painful integration, now become automatically interoperable because we've driven their integration toward structural similarities.

Any non-patterned structures will remain outside of the behavioral pattern model, requiring special code and maintenance, but bringing them into the fold will stabilize their implementation. We learn quickly that transforming an object into a structural pattern both stabilizes its implementation and enables it to be used in all behavioral patterns.

This model sets aside standard polymorphism (embracing and enforcing the uniqueness of every object). Rather, it promotes what I'll call *metamorphism*, which moves all objects toward manifesting their similarities. Polymorphism and metamorphism are discussed in the next section.

Over time, we will tire of banging out software, at whatever speed, for mere linear productivity.

When we experience the power of applying patterns, our skill in bursting high volumes of software diminishes in value and we place greater emphasis on leveraging prior work. Because high output alone is a linear model, it requires lots of time. However, high reuse is not a linear model. It is a radically accelerated, hyperbolic model that pushes our productivity into the stratosphere with little additional effort.

We can produce end-user features well within scheduled time frames, giving us breathing room to increase quality or feature strength.

With stable program-level capabilities in place, we can deploy features with radical speed. This foundation establishes a repeatable, predictable, and maintainable environment for optimizing software. Consider the process depicted in the sidebar, "Full Circle Emergence."

Full Circle Emergence

When we first produce application software, *latent patterns* emerge.

When we first acknowledge patterns, *understanding* emerges.

Upon first exploiting patterns, reusable *structure* emerges.

Upon first exploiting structural patterns, reusable *behavior* emerges.

When we combine structural and behavioral patterns, *frameworks* emerge.

When we combine frameworks, *applications* emerge.

Note the full circle, beginning with the patterns emerging from the application and ending with the application emerging from the patterns. In an architecture-centric model, *the application is along for the ride*, not embedded in the compiled software.

Using Metamorphism versus Polymorphism

If you've been in object-centric programming for any length of time, the term polymorphism has arisen, probably with subtly different definitions. Since polymorphism and inheritance are closely related, I'll provide a boiled-down working definition of each:

- *Inheritance* is the ability of a class to transparently acquire one or more characteristics of another class (subclass) such that the inheriting class can exploit (e.g., reuse) the structure and behavior of the subclass without re-coding it from scratch. Conversely, the inheriting class can extend the capabilities of the subclass.

- *Polymorphism* literally means multiple forms, where "poly" is *multiple* and "morph" is *form*. Polymorphism enables a class, primarily through inheritance, to transparently align its structure and behavior with its various subclasses. Thus, the class appears to have multiple forms because it can submit itself to services that were originally designed for the subclasses. The objective is to enable reuse through designing subclass-level services and converge them into a larger, polymorphic class.

Discussions on polymorphism follow a common template. Consider the objects cheese, milk, and soda. We identify each one through properties (weight, color, volume, taste, and so on) and examine their behaviors within this context. We can consume any of the three. However, we would measure cheese by weight and milk by volume. We would drink the milk or soda, but chew cheese. Milk and cheese carry all the properties of dairy products, while milk and soda carry all the properties of liquid refreshment.

These are all healthy and useful examples, but they have one flaw: they are all application-centric! Examine practically every use-case analysis book on the market, and all of them advocate, and charge us with one directive: Build application-centric objects into our software programs and allow them to inherit from each other. Before we're done, we'll have things like *DairyProduct.cls, Milk.cls, Cheese.cls, Soda.Cls, Refreshment.cls* and might bring them all to a *Picnic.cls*!

This practice will embed and directly enforce uniqueness into the software program. Practically every line of code touching these objects will be intimately tied to their structure and behavior. When we're done, we'll find lots of inheritance, probably stellar and creative polymorphism, and magnificent reuse *within the application*—but not a lot of reuse outside the application. Rather than standardize on common patterns, we've accommodated and even enforced application-centric uniqueness.

NOTE *A primary objective of pattern-based architectures is to thoroughly eliminate dependence upon uniqueness among objects, allowing them to play in the same behavioral space because of* their sameness.

After diving deeply into patterns, *superpatterns* emerge. Superpatterns are assemblies of pattern-based structures and behaviors that define another level of useful abstraction. One of the more dramatic superpatterns is *metamorphism*. Metamorphism is the run-time ability of a programmable object to completely change its structure and behavior.

Metamorphism allows an object to morph into a class definition that was unknown at design time.

For example, a given object may initially instantiate as *Milk*. It may enhance its status into a *Refreshment*, or metamorph into a *HotDog*, or perhaps a *Hamburger*, a *Horse*, a *Screwdriver*, or a *Mack Truck*. In every case of metamorphosis, it is still technically the same internal object reference, it's just been reprogrammed—rebooted if you will—with an entirely different identity and instruction set. And all this happens magically without changing a single line of program code! Metamorphism is a dramatic, even quantum leap in programmatic machine control. Its primary fuel is advanced *metadata*.

Metadata

Meta denotes both change and abstraction. We use symbols to abstract ourselves from expected volatility and software to address rapid and chaotic change.

Sadly, industry and product literature often define metadata solely within the static realm of data warehousing or information management: “Information about information.” But this is a limited definition.

Metadata itself is volatile information that exists outside of a program with the ability to influence the program’s structure *and* behavior, effectively changing the way the program operates.

Programs that subscribe to metadata are run-time programmable. But programs that are dependent on metadata as a sole source of fuel are the most powerful products in the world. Objects using metadata to dynamically define their internal structure and behavior are *metamorphic*.

NOTE *Clearly, the volatility of user requirements is the perfect domain for advanced metadata. If we can meet all user requirements for application structure and behavior through metadata, we minimize and eventually eliminate the need to change and rebuild programs to meet user requests.*

Metadata appears in two forms: *structural*, to define information and entities as building blocks; and *behavioral*, to describe processes that apply to information or entities (e.g., instructions). These forms allow metadata to provide high-octane fuel to a run-time interpreter. A metadata interpreter then interfaces and organizes structural building blocks and core behaviors into application-level features, enabling run-time programmability.

Apress Download

If you have not already downloaded this book’s supplemental projects, get a copy of them from the Apress web site at `http://www.apress.com` in the Downloads section. Unzip the file into the working directory of your choice. For consistency I will use the relative directory structure in the zip file’s extraction.

One top-level directory is VBNET, containing the .NET software, while another top-level directory is VB6/, containing the Visual Basic 6.0 versions of everything discussed in the book.

Under the top-level directory set is another directory called vUIMDemo/. Underneath this directory you'll find a Visual Basic .NET Solution file called Proto.sln and a program named vUIMDemo.exe. Follow along in the version you are comfortable with.

Follow these steps to watch metamorphism in action:

1. Double-click the program file (vUIMDemo.exe) and it will pop up a screen similar to Figure 1-5.

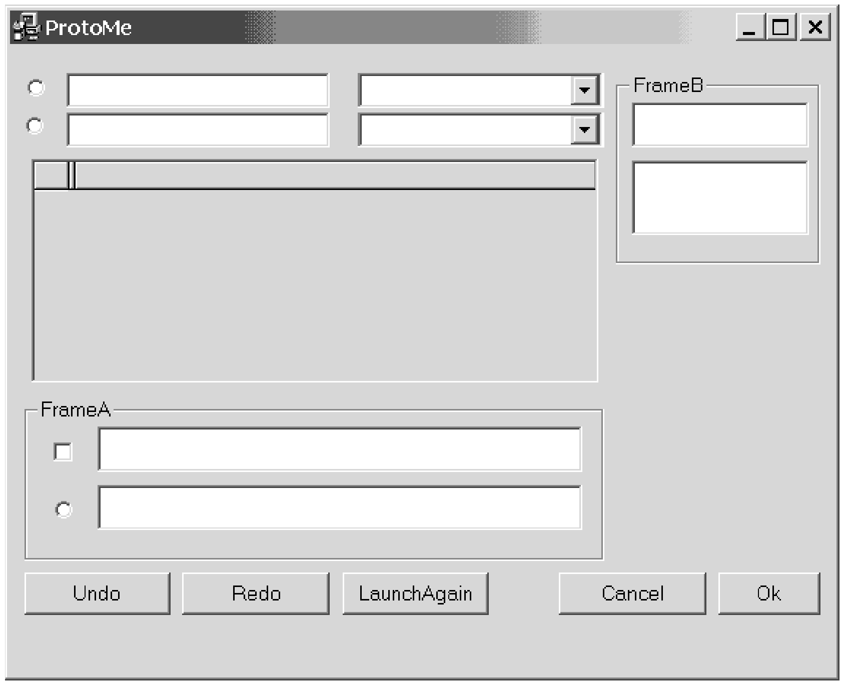

Figure 1-5. A display from the program file vUIMDemo.exe

2. Double-click the Proto.sln project to bring up Visual Basic, then open up the ProtoMe form inside it. You'll see the design-time rendering of the screen depicted in Figure 1-5.

3. Next, in the ProtoMe screen, delete the Tree View box above the Cancel/Ok buttons, and click/drag the Cancel button into the center of the open space, then save it (Ctrl+S).

4. Go back to the running vUIMDemo and click the LaunchAgain button at the bottom center of the screen. The vUIMDemo now renders your version of the screen.

5. Click the caption bar and drag it to the side, revealing both screens. The program, vUIMDemo, *is still running!* The screen changed without changing the program! We'll dive deeper into these capabilities in the Chapter 4 projects, with detailed discussion in Chapter 7.

You can repeat this exercise, adding to, reshaping, and changing the ProtoMe file. Each time you click Launch again, your work is instantly available. This is *metamorphism* at work, and we've just scratched the surface.

Pattern-based metamorphic architecture is devoid of any application-level knowledge. It does not recognize any internal distinction between applications for marketing, science, accounting, astrophysics, human resource management, and so on. The architecture simply executes, connects services, obeys external commands, and the application *happens*. This is a very different and abstract approach than the Common Development Product depicted in Figure 1-1, which embeds the application into software.

The metamorphic approach is an exciting and perhaps natural step for some developers, while a quantum leap for others.

If our software is to respond to advanced metadata, manufacturing the application "on-the-fly," we need a means to symbolically bridge the metadata to the underpinning technology.

Understanding the Language of Symbols

If we start with software code as textual script, we must understand what the script means. Special symbols understood by a compiler will ultimately bring our creation to life. It is what the compiler does with the symbols, not what we do with them, that determines their ultimate usefulness.

To grasp it all, we must commit to thinking in terms of symbolic representation rather than software code alone. Symbols can bundle and clarify enormous amounts of "noise" into useful knowledge. We understand symbols both automatically and relationally, but the machine only understands what we tell it. More importantly, the machine's symbols must always reduce to some form of mathematics.

Patterns in Mathematics

Galileo declared that mathematics is the only universal language, and this is an absolute in computer science. Math is pervasive in all computer systems and is the foundation for every activity, instruction, and event inside a programmable machine. Software is a *mnemonic,* or symbolic means to represent the details of the computer's math. Software represents instructions to the computer as to what mathematics to perform and in what sequence. These may be simple additions, complex multiplication, or just counting bytes as they move from one calculated memory address to another.

However, human thinking processes use math as a tool, not as a foundation. Humans have difficulty in programming machines to think because so much of thought is relational, not mathematical. In fact, mathematicians are persistent in finding and exploiting patterns in numbers and numeric relationships to better promote their understanding to peers and students.

Our goal is to abstract the software language's raw, elemental, structural definitions and instructions into higher and simpler symbols and semantics.

A Perfect 10?

Consider that the decimal (base 10) digits zero through nine (0-9), the familiar mental foundation for all decimal counting operations, is sometimes misused or misrepresented. A recent visit to a toy store revealed that most electronic counting games for small children teach them to count from 1 to 10. While this sounds simplistic, *ten is not a digit*! Ten is a combination of the digits 1 and 0.

However, a machine structure only understands the 0-9, and the position represented by *9* is the *tenth* element in the sequence. We may ignore this and get bitten when trying to access the element at position *10* of a zero-based array. We must understand *how the machine represents the structure*, not how humans understand it. This is the primary first step in mapping virtual structures into the human knowledge domain.

We must learn to think on the machine's terms. Avoiding this issue by twisting the structural and conceptual representations of objects and underpinning math will only increase our dependency on the language environment, the machine's rules, and the software code. Our goal is to master the machine and the programming language.

Our mistake is in resting on the software language alone, rather than using it to define and activate a higher, more dynamic but programmable, symbolic realm.

Every action, algorithm, and operation within a machine is a sequence of mathematical operations. If we can bundle sequences of actions at the appropriate elemental levels, we can treat these bundles as macro statements. When statements are strung together, they represent *methods*. If we can string them together at run time, we have a fully dynamic method. From this point, we can construct or reconstruct the method at will, and it becomes metamorphic. When we label it with a symbol, we can interchange it seamlessly across multiple metamorphic objects.

The mathematics inside a machine are too complex and dynamic to grasp without symbols, hence the need for software language.

Abstracting Knowledge

Albert Einstein proposed that a great chasm exists between the concrete and the abstract. He defined *concrete* as objects in the physical world such as wood, rocks, metal, and so on, and *abstract* as the *understanding* of wood, rocks, metal, and so on. This was Einstein's contention:

> *We have a habit of combining certain concepts and conceptual relations (propositions) so definitely with certain sense experiences that we do not become conscious of the gulf—logically unbridgeable—which separates the world of sensory experiences from the world of concepts and propositions.*[2]

Einstein asserted that the human mind creates the necessary bridges *automatically*, and we are unable to bridge them *logically*. This assertion has profound implications, because logical mechanisms are all we have inside the computing domain.

Einstein asserted that the key is symbolic verbal or written language as a common frame of reference. While human language effortlessly uses symbolic labels to represent tangible, concrete items, computer language must use symbols to represent abstract, virtual items. The difference between the human-understood symbols and their actual electronic representation creates yet another chasm, the *Cyber Gulf* (see Figure 1-6).

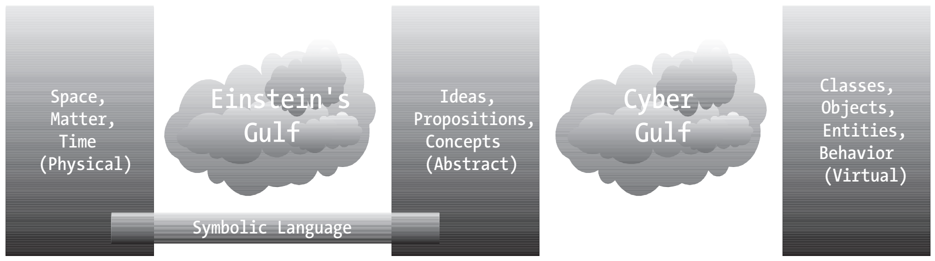

Figure 1-6. Understanding the Cyber Gulf

The machine has no means whatsoever to represent the physical world; it is completely and utterly removed from it. Nothing is concrete inside the machine's software processing domain. Computer representations of physical objects are virtual collections of mathematical abstractions, understood by humans only through similarly abstract symbols. This is a significant limitation for expressing highly complex human concepts such as humor, love, mercy, justice, and so on.

Another significant problem is in testing the computer's representation of these concepts, because at least two people must agree on the computer's conceptual representation. (We can't imagine two humans agreeing on an ultimate, mathematically accurate definition of a concept such as humor.) We rarely think of our work in such abstract terms, because programming is already so difficult and time-constrained that rising above it into abstract realms requires unavailable time and extraordinary effort.

NOTE *Just as humans automatically use symbolic labels for simplifying the physical world, we must learn how to instruct a machine to use symbols rather than hard-coded application instructions, and we must automatically think of programming in these terms.*

The difference in the two approaches is profound. The hard-coded approach only leads to rework, where the technology has mastered us. The abstract approach leads to reuse and repeatability, where we master the technology. It is, in fact, the long-sought path to freedom. Overcoming the technology itself is the ultimate strategy to escape application-centric gravity.

How does this apply? We must negotiate objects from within virtual space and find a means to simulate their equivalents in a near-physical electronic forum; for example, an employee entity is not a physical employee. Bridging two gulfs simultaneously requires an enormous amount of thought labor and mental continuity not required in actual physical construction. While a builder and a homeowner can point to a house and agree that it's a house, a developer and an end user can observe programmatic functionality and completely disagree on the same thing.

Now step back and examine the vast domain under our control, and the many bridges, highways, and other delivery avenues required to fulfill the user's requests. None of it is easy, and it is absolutely imperative that we gain ultimate, sovereign control of this domain.

Software that is wired directly into the user's churning, subjective chaos is constantly subject to change and even redesign.

Okay, okay, I hear you—enough concept. The reality is that as feature requests become more sophisticated and complex, with shrunken time-to-market and rapid change of features over time, we no longer have the luxury of building an application-centric program. The knowledge domain is already too wide, even for the simplest technical objective. We must embrace a more productive, reusable, and resilient deployment model in order to survive and be successful in the boiling technical marketplace. Only the architecture-centric model, with patterns of reuse that allow a malleable, metamorphic, and symbolic approach to application deployment, will enjoy repeatable success.

Symbols can aggregate huge amounts of processing logic, or allow interchange of complex details by reference rather than physical movement.

Dynamic versus Static Representation

Consider a class definition for *Rock* in application-centric mode. We must carefully define the specific interfaces for *Weight()*, *Size()*, or *Density()*. This creates a "wired" object with specific purpose and limited reuse. However, while we may believe this static definition provides visibility and control, at run time the machine understands and represents it mathematically, through complex, embedded machine logic. It is not wired at all, but *already* a virtual abstraction beyond our control.

Now let's configure a more flexible object, with a properties "collection" symbolically representing otherwise hard-wired references. We'll use *Item("Weight"), Item("Size"), Item("Density")*. From the machine's perspective, this representation is identical to a carefully crafted static object, with one primary structural difference affecting reuse: the interface is infinitely extensible without changing its definition. If the machine cannot discern the difference, why create hard-coded, artificially propped constructs (static class definitions) to support something the machine will bypass or even ignore?

This is an important distinction, because of what many developers believe about how a machine represents objects. We'll declare a class as a specific, application-bound definition and manipulate it as such because it feels like greater control exists. This is an illusion propped up by the software language. Business-object or science-object development books and discussions strongly encourage us to build specific objects such as Employee, Account, Invoice, or Indicator, Thermometer, and Gauge. Development languages and design tools directly support these suggestions.

We can define objects specifically or construct them in the abstract. The latter gives us freedom, and the machine doesn't know the difference.

In reality, these objects, their properties and methods are simply collections of smaller programmable parts and processes. They can be assembled with a hard-wired coding method, but should be assembled with a dynamic construction method; the two are functionally equivalent. The dynamic method requires abstraction, but it is far more resilient, flexible, reusable, and metamorphic than its hard-wired functional equivalent.

Why should a complex action require a specific software anchor? When we consider that every method, regardless of complexity, reduces to a series of mathematical functions, we simply aren't far from abstracting them already.

If each metamorphic object is influenced or managed by a run-time interpreter for execution of methods, no further reason exists to directly hard-wire a method into an object. If a "custom method" is simply an assembly of smaller, reusable processes, every object can dynamically construct methods that are "apparently custom" but, in fact, never existed before run time.

Pervasive structural and behavioral metamorphism provides practically limitless reusability to a majority of methods and properties for building any application, in any context. It supports the concept of *metamorphic class*. Such a class exists only at run time and manages all properties, methods, and inheritance as a function of reuse, not hard-wired application logic.

With dynamic properties, methods, and inheritance, all classes are defined at run time. Metamorphism (not just polymorphism) becomes the rule, not the exception.

By reducing our software-based classes to structural templates, the software can manufacture new, metamorphic classes with relative ease. The metamorphic class methods and properties are constructed from smaller, generalized, embedded structure classes. The best part is that we really need only one metamorphic framework, with all underpinning components focused on its support. We can then dynamically instantiate the framework into anything we want.

Summary of the Metamorphic Superpattern

Here is a short list of the primary principles we must understand and embrace in order to fully exploit metamorphism:

- All machine quantities reduce to math.
- Symbols can represent all machine mathematics, including complex structures and relationships.
- Semantics can describe complex behaviors through assembling elemental behaviors.
- Humans can use high-level metadata to organize symbols and semantics for custom application effects.
- Metadata interpreters can accept symbols and semantics as run-time instructions, translating them into internal metamorphic structures, behaviors, and relationships.

For some, metamorphism may seem too complex and uncontrollable. It is, in fact, a perfect means to guarantee consistent behavior, structural integrity, and deterministic outcome for every application we deploy. It also represents a foundation for capturing our knowledge and re-deploying it to accelerate future efforts and upgrade prior efforts.

We must forsake the idea that software is the maker, keeper, and foundation of structural and behavioral rules. We must now begin to view software only as a tool for exposing technical capabilities into abstract, programmable form. We must then view application-centric features as virtual abstractions: dynamic, run-time definitions that don't exist, and never existed, before the program's execution.

Symbols allow us to anchor and assemble structure and behavior, but the order of their assembly, plus parameters and modifiers, are in the domain of semantics.

Realizing How Semantics Activate Knowledge

Unless we want to institutionalize chaos, we cannot start by directly embedding the user's wishes into the software code, so we must therefore pick another starting point. We must build code that enables the representation of the user's wishes rather than directly representing them. While software as textual script contains keyword symbols recognized by the compiler, the context of the symbols is highly significant in representing *application knowledge*, which entails the embedded intentions of the end user as applied by the developer.

We can tag objects, properties, and methods with high-level symbols, but we need *semantics* to shape the symbols into active knowledge. A *Rock* simply "is" and will "do" nothing unless we apply a semantic command to combine the object with an action. *Throw Rock* should give us a raw effect, but we may need to modify the action, such as *Throw Rock Hard,* or even more vectored detail, such as *Throw Rock Hard at the Window.*

Of course, if the Rock's Item("Density") = "Low" *or its* Item("Weight") = "Light" *we might have an action result of "Bounce"!*

Predefined keywords clarify semantics, especially when combining or copying information from one symbol to another. For example, *Set Rock.Item("Texture") = "Smooth"* is different language than *If Rock.Item("Texture") = "Smooth."* The first performs assignment and the second tests a conditional relationship.

> **NOTE** Structured Query Language (SQL) *is a common semantic model used by most commercial database engines. In SQL, nouns (database tables, columns, or other entities) are instructed to behave according to verbs (insert, update, select, delete, and so on) with certain modifiers (search predicates, insertion, or update information, and so forth). The rules for syntactic construction are rigid enough to be accurate and efficient, but flexible enough to promote limitless reusability of the database engine for storage and retrieval logic.*

Think in terms of a development language compiler's syntax rules. Each rule is a semantic compiler cue to perform a certain task, build a particular construct, behave a certain way, etc. The compiler uses reserved keywords to affect its own programmable behavior. Nouns (objects) are the actors. Verbs (methods) are the actions, with modifiers (properties) to each action. Interpreted sentence structures can create dynamic objects, relationships, and activities that were unknown before program execution.

A language compiler provides the lexical and behavioral foundation for its own programmability. It is the ultimate programmable program, so examining its implementation provides insight to understanding how to put similar functionality into a dynamic application model.

Our objective is to boil down the plethora of object and information types into their simplest abstract equivalents.

Metadata instructions serve as pseudo-code, and cleanly separate the program's architecture from the application's structures and relationships. The software program containing the interpreter is compiled, deployed, and constantly reused thereafter. Run-time interpretation means the application logic can change at will, even by the minute, without affecting the program. We can build whole applications with textual metadata, then deploy them with a simple "file copy" to expose the metadata to the program. In fact, the application logic can change even while the executable is running. (See the vUIMDemo earlier in this chapter under "Metamorphism and Polymorphism" for an example of this.)

Shockwave Delivery

The vUIMDemo is a subsystem of a product (called *vMach*) that Virtual Machine Intelligence, Inc. regularly deploys on user desktops. Whenever we need to ship a new application upgrade to user screens, we bundle the metadata into a zip file and send it via email. The user unzips the attachment and it's installed. Can you imagine making changes to a Visual Basic screen in Visual Basic, then performing a simple file copy to deliver the goods? Or extending the database design without breaking existing code, with a model that "senses" and wraps itself around the changes rather than balking at them?

For users with a central file server, we send the changes to a central administrator, who in turn unzips them to the file server. The upgrade is instantly available to every desktop using the metadata. This transforms rapid application delivery into *shockwave delivery.*

We'll now use software code to build frameworks that manage actions between participating objects. The software will accept textual instructions as metadata, interpret them, and ultimately dispatch the activities between objects with no knowledge as to the nature of the activity itself. Whether system-specific or user-required, the software simply executes the action. The application *happens.*

Now we'll use metadata—not software rebuild—to meet user requests. We can jot down notes at their desktop and deliver features "on location."

The software program will interpret events, semantic commands or rules, and initiate activities. The program does not directly control the application behavior, only the semantic interpretation of metadata. It is therefore incumbent upon us to apply simplified interfacing capabilities to all embedded and dynamic entities, allowing the semantic interpreter to find and control objects quickly.

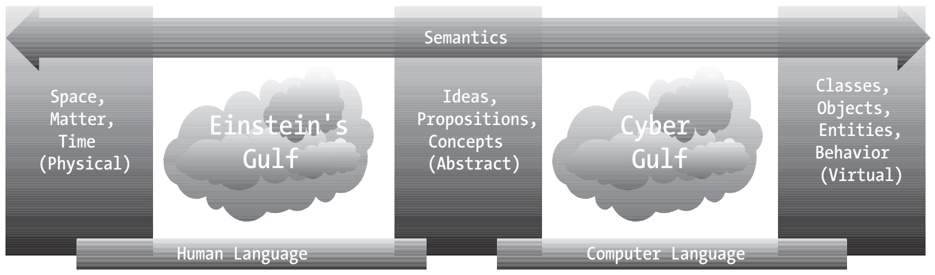
Semantics
Space, Matter, Time (Physical)
Einstein's Gulf
Ideas, Propositions, Concepts (Abstract)
Cyber Gulf
Classes, Objects, Entities, Behavior (Virtual)
Human Language
Computer Language

Figure 1-7. Virtual architecture symbolically spans the gulfs with semantics.

Like a symbolic skywalk (see Figure 1-7), a metamorphic architecture seamlessly spans the gulfs, from physical to virtual and all elements of understanding in between. It serves as an embedded language interpreter, actor, and observer, and puts a consistent and predictable leash on all behavior, regardless of processing domain or development language.

Developer in a Box

In an application-centric model, our "software development" activity consists of abstracting and arbitrating symbols across the gulfs as we communicate with the end users. This activity, when repeated a few times within and across projects, gets tedious and boring. What if we could embed enough intelligence into the software itself so this arbitration happens by proxy?

To contrast, we can define the symbols and build embedded software to enforce them, or we can build an arbitration mechanism that will act on our behalf just as we would. Once in place, we have effectively migrated our own development experience and expertise permanently into the software. Anyone else using it gains the immediate benefit of our accumulated experience, not just what a user asked for once upon a time.

We capture a developer's experience each time they touch reusable code. Like an Aladdin's Lamp, *each time we rub it, the magic begins again.*

This embeds our hard-won knowledge and skill into the architecture, rather than requiring us to repeat the same rote software development with each new application. This further accelerates reuse, so we can leverage our prior work, and others can too, and is rocket fuel for innovation because what we (and others) do is never lost again.

Development Projects

When executing projects using the architecture-centric approach depicted in Figure 1-8, all software is founded upon metamorphic, programmable capabilities. These form frameworks, driving the ability to dynamically generate complex features and thus, applications.

CROSS-REFERENCE *See Chapters 5, 6, and 7 for more detail on frameworks.*

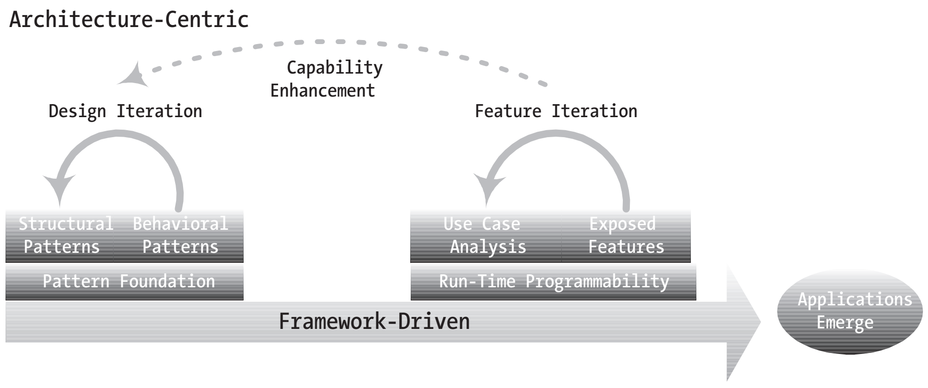

Figure 1-8. Architecture-centric project execution

Reviewing Figure 1-8, all feature changes occur within the metadata environment. When we cannot fulfill a feature with current capabilities, we identify the required capabilities and drive them back into the architecture as a capability enhancement. We are then able to fulfill the feature, but the capability remains for later reuse (by ourselves or others). This is the only true capability maturity model applicable for reusable software.

Capability (im)Maturity

Many companies build volumes of process engineering notes to track the development of "capabilities," mostly in an application-centric context. For example, a company might log that some of its developers rolled out a human resources application, a supply-chain solution, or a retail sales data mart. Invariably, the solutions are completely custom and not positioned for re-hosting to subsequent projects.

In each case, the company records the solution as part of a knowledge base, but it is in fact the accumulated experience of its employees. Many technologists balk at being pigeon-holed into repeating the same application endeavors, regardless of the potential success (and thus avoid re-assignment). Some employees are promoted out of direct implementation roles, while others leave the company.

The result is the same: the company isn't really gathering steam in developing or maturing capabilities. They are only creating a marginally viable portfolio that rapidly grows stale. The company, like many others, will only be as good as its last success.

Note that all capability enhancements are available to all features, not just the feature that initiated the change. We can deploy the same capability set, or binary program, for multiple applications. This is the only true high-reuse model, and as promised, enables reuse of every single line of compiled software.

As the architecture and capabilities stabilize over time, we drive fewer changes back into the program code. Architects have freedom to make the software program more powerful, faster, easier to install, or otherwise technically superior. Application developers using the program can depend on a reliable, reusable, and solid foundation for feature deployment. Additionally, because the software program is fueled with metadata, application-level deployment doesn't require a program rebuild, only a textual modification and file copy!

Architects can dive deep into technical issues, vicariously delivering their expertise and hard-won knowledge to the metadata interpreter.

In the vUIMDemo, our change was instantly available to the end user without changing the software. As quickly as we can change the metadata, we can change the structure and behavior of the application-level features.

Utilizing metamorphic objects, run-time programmability, and a stable, consistent application delivery, every known development methodology either stands itself on its head, or experiences stratospheric productivity.

Smooth Delivery Effect

Upon initial construction, the pattern-based and architecture-centric model doesn't deliver features right away. Developers and designers spend significant hours formulating and building structural and behavioral foundations for later feature deployment. When features appear, they arrive at a steady and sometimes dramatic pace from beginning to end. Because the end users experience a steady stream of smaller parts, the delivery highway for both features and binary software will be in solid order and smooth operation by the time of a "go-live" production delivery.

The initial time frame could be protracted because the team is trying to deliver features while also building capabilities, but the product will be consistently solid and the user won't see the brutal 80/20 effect of the application-centric deployment model.

In follow-on projects, users experience feature delivery in a relative fraction of the time required in the initial project. Capabilities are already present and stabilized, ready for reuse, enabling us to deliver features at *mach* speed compared to the application-centric model. Any new capabilities become automatically available to prior projects and applications.

The architecture-centric approach automatically supports incremental redelivery. Architects can focus on technical installation of the base program plus the mechanics of metadata transmission and make these processes rock-solid. This creates a repeatable delivery highway. Feature deployment, using metadata, becomes so rapid and regular it appears fluid.

If a project is like an airplane, iterative delivery is like scheduling multiple flights, with passengers (features) arriving regularly.

Ideally, this approach supports later maintenance activities. In fact, the initial delivery happens so early in the cycle it's often forgotten. Some architecture-centric solutions deliver hundreds of preliminary feature versions during iterative development. If the so-called final feature delivery is actually iterative delivery #289, what of it? By the time the application enters the maintenance cycle, its delivery mechanism is well-oiled and humming.

When first developing a metamorphic model, it does not quickly deliver in the front of the project cycle, and many development teams feel under pressure to deliver *something*. Thus it often falls on the sword of expedience, otherwise known as trading off the important (reuse and maintenance) for the urgent (quickie delivery).

However, this is a sensible delivery model for the following reasons:

Business: We can schedule, deploy, manage, and close the feature requirements in a timely manner. Once a feature is in front of the user for an extended period, it erodes their ability to arbitrarily change it later. Maintenance costs are lower because fewer programmers are necessary for upgrades and changes. Because of high productivity, team leaders can quickly prototype *working* proof-of-concept models rather than empty demonstrations. Rapid iteration creates manageable client satisfaction, streaming functionality into their hands rather than experiencing start/stop delivery.

Technology: As new technology arrives in the marketplace, the core program can assimilate it without breaking existing applications. New applications can directly leverage it while prior applications can access it at leisure. In fact, even if forced to replace the underpinning technology, including the development language, operating system, and components, we can avoid impacting the application. Additionally, business managers can negotiate with current clients to provide upgrades for additional revenue.

Quality: Fewer programmers touch the core software, releasing it less often than the metadata for application features. The core software release cycle becomes much more stable and manageable, occurring on boundaries under the control of the master architects. We then use metadata to meet user requirements, on a separate and more volatile release cycle. The users can get what they want without the inherent destabilization of a program rebuild.

Human resources: Master developers can oversee initial application design and have a footprint in multiple projects. Novice developers get productivity boosts using the architectural safety net provided in the core software. We can divide technical resources along flexible lines: those that work in core architecture, those that work in application-level logic, those that manage technical teams of both, etc. The high productivity of an architecture-centric model provides schedule flexibility and personal time for developers. It's also a lot of fun to work with, consistently achieves the "Wow" effect on delivery, and generally boosts the morale of any team regardless of the delivery context.

Implications to Development Teams

The architecture-centric model provides a platform for creatively challenging our developers and keeping users happy.

Many experienced developers believe that business and functional requirements are the primary variables from project to project. *Feature-point analysis,* which involves breaking the project deliverables into irreducible minimums, should yield a fairly predictable delivery schedule. When setting such schedules, we often use past experiences in order to predict scope of effort and time lines for upcoming projects.

For stable teams with little attrition, this method is ideal, in fact, perfect. For the average development shop with relatively high attrition, feature-point analysis is less predictable. The required knowledge base is unstable because developers take knowledge with them when they leave a project or company.

In an application-centric model, the skills of the developers, their relationship as a gelled team, and the complexity of the product all form convergent *variables*, creating a higher project risk. In fact, a developer's skills (not the user's requirements) are the most variable and volatile quantities from team to team.

As application complexities and feature-richness increase for competitive advantage, enormous pressure builds to assemble and keep a skilled team. It is, therefore, incumbent upon project and company leadership to keep the team's skills sharp, their energy focused, and their minds challenged.

The architecture-centric model captures knowledge, stabilizes teams, and abates attrition (generally), allowing feature-point analysis to gain more foothold and predictability. The model can satisfy the most demanding team's professional and career needs, because senior developers get the opportunity to gravitate toward technical issues and new developers find themselves in a rich mentoring environment as they deploy applications using the technology. With a stable workforce, innovation also has a stronger foothold.

We've seen enough of success and failure to know what works and what doesn't. Because software development, and especially innovation, is high-intensity thought labor, mentoring is effective for handing down experience and transferring knowledge, but only if we can keep our teams together. The architecture-centric model is an ideal mentoring foundation.

Innovation leverages prior effort, but only if the knowledge workers (or their experience) is available for reuse.

Implications on Revenue

Managers know they can make tons of money with custom development projects billed to a client by the hour. If enabling accelerated reuse shrinks delivery time, it also changes revenue equations: shorter projects mean less overall revenue if billed by the hour. For an artificially extended project, the end user might not know the difference if deliverables are on time and of high quality. They might smile and cut a check.

However, if the clients, or end users, realize they paid high dollars for custom craftsmanship when they could have paid much less, they will feel cheated, and rightly so. If a young, agile upstart-startup company masters reuse, it will drive a wedge into market share that a non-reuse company cannot remove. Even if the reuse-masters take the same amount of project time, they'll still have a higher quality product.

The conclusion is that the architecture-centric model gives us more options to increase quality, feature-richness, and keep the project timeline under our control rather than racing breathlessly to the finish line with a marginally functional product.

Companies using the architecture-centric model can deliver in a fraction of the timeframe required by the application-centric model. Does a quick-turnaround diminish the value of the deliverables? Hardly! If anything, it gives the company an extreme competitive advantage. Companies that bill by the hour will have to rethink their billing practices. They will have to assign value to their speed of deployment, increased product quality, and overall value of the application to its end users. Faster time to market is highly valuable, and savvy leaders can and will exploit this to competitive advantage.

Charge more for delivering higher quality and richer features in a more manageable time frame. Once delivered, keep the momentum and deliver more.

The true software development starting point exists well before the user pens the first application requirement. It exists before the developer breaks open the language development tools. We find it when we invoke pattern-based abstractions to meet user features, rather than wiring them into the software. When we begin to automatically think of programming in these terms, we've crossed the *virtual mach* boundary with a shockwave in our wake. Those who begin there will emerge with a firm technical, structural, and behavioral foundation for building all sorts of applications.

If we begin in the application realm, we've already lost. The application realm must rest on the foundation we prepare in software, and must be the result of core reusable software acting on metadata, not with application logic embedded in the software.

If the software development cycle is a continuum, the application realm actually marks the 80 percent point and moves toward closure at 100 percent. Beginning there forsakes the foundation itself. We will have missed four-fifths of the architectural maturity required to build highly reusable code.

Keeping the Momentum

Now that I've proposed a different approach, touting the possibilities of the architecture-centric view; the propulsion of accelerated reuse; the power of metamorphism; the positive traction it creates on our careers, our lives, and professional satisfaction; it's time to show you some of its finer details. I will strongly implore you to examine and perform the project exercises that you'll find throughout this book because each one builds on the last.

Project 1—Groundwork

Let's close out this chapter with a light example of some symbolic-tagging concepts. They will provide a lead-in to the following chapter, which introduces structural patterns. These are the primary project elements:

- Structural tools
- Factories
- Object/property manipulation
- Supporting code

The project language examples will be in Visual Basic .NET, but I've also included the Visual Basic 6.0 versions, (a minimum of Service Pack 3, preferably Service Pack 5). You won't need any other third-party products to support the examples. I've tested them under Windows NT 4.0 and 2000 Professional. The .NET, Visual Basic projects use the *.vbproj extension, while VB 6.0 uses the *.vbp extension.

*917_Ch01_CMP_3 2/21/02 1:34 PM Page 35

Additionally, in the interest of clarity all project examples in the book will use the shorthand version of class names (e.g., vTools.TurboCollection will appear as TurboCollection in the book content).

If you've not downloaded the book's project listings and examples from Apress, do so now and extract it into the working directory of your choice. Each project has its own subdirectory, and a separate directory called VB6/ contains the Visual Basic 6.0 equivalents.

ProjectOne includes a number of class modules and a *main* module used as a skeleton to browse and manipulate their activity. Enter the ProjectOne/ subdirectory and *double-click* the ProjectOne.sln file to load the project skeleton. Answers to inquiries are available at `info@virtualmach.com`.

TurboCollection

The first structure requiring attention is the TurboCollection. It appears as two separate class objects, the *TurboCollection.cls* and the *ParmNode.cls*. The TurboCollection is an *Iterator* pattern. It hides and manipulates a *list* of objects, in this case ParmNodes. The TurboCollection acts by containing and linking ParmNodes to each other, then browsing and manipulating the links. The ParmNode is a simple *Container* pattern. It provides a memory-based storage location for both simple and complex information. Together, these classes form structural glue to enable metamorphism, bind and automate all other structural and behavioral patterns.

The first exercise will add nodes (ParmNodes) to a TurboCollection to create a simple list. Note that each node is a self-contained, detachable/attachable container. The ParmNode is also an adapter of sorts, serving to wrap and anchor a structure of any kind. For the first example, we'll add and organize simple textual data and then move into objects.

Load ProjectOne and find the Public Sub Main() header (see Listing 1-1). You will see a number of data declarations at the top of the module. We'll enlist their aid as we move along. Supporting code also follows the Main() header.

From here, let's build a simple list of information points using a TurboCollection structure. Under *Public Sub Main()* find this code:

```
Dim tcTest as New TurboCollection()
Dim pWrk as ParmNode
```

This will instantiate one of the TurboCollection Iterators along with a ParmNode for workspace. Now, let's add a ParmNode container to the Iterator by invoking tcTest's Add:

```
tcTest.Add (New ParmNode())
```

Note that .NET will automatically wrap this statement with the appropriate parenthesis. This will allow the addition of a newly instantiated ParmNode to the Iterator's list. By default, the TurboCollection will add the node to the end, or *tail*. While this provides us with the power to add a pre-existing node, practically speaking, most lists are created on the fly, so this notation could get clunky and verbose. For agility, use the following shorthand:

```
tcTest.Add()
```

This assumes we want to add a ParmNode, so the TurboCollection will instantiate and add one for us. We can get access to the new ParmNode in one of two ways: on the TurboCollection or as a return value from the Add() method. Here's an example:

```
pWrk = tcTest.Add()
```

or

```
tcTest.Add()
pWrk = tcTest.Ref()   'Ref always contains the currently active node.
```

Next, let's add the following data to the ParmNode container. It has several generic properties, the *Item, Obj, ItemKey,* and *SortKey* among others. These properties are critical to node identification and manipulation.

Item: This property contains simple textual data associated with the node (text, numeric, etc).

Obj: This property contains a reference to an object (complex data) associated with the node.

ItemKey: This property contains a textual key to uniquely identify the node in the Iterator list.

SortKey: This property contains collating information associated with the node's other information, used to sort and order all the nodes in relation to each other.

Thus, the following property references are equivalent to set the Item, ItemKey, and SortKey properties, respectively. The first example shows how to set the values directly on the ParmNode:

```
pWrk.Item = "A1"
pWrk.ItemKey = "A10"
pWrk.SortKey = "A10"
```

While this second example shows how to set the values on the same ParmNode using the TurboCollection's current Ref:

```
tcTest.Ref.Item = "A1"
tcTest.Ref.ItemKey = "A10"
tcTest.Ref.SortKey = "A10"
```

Even this notation can get cumbersome when coding the construction of a list, so, since Item, ItemKey, and SortKey are the most-often used properties when building a list, another method on the TurboCollection facilitates this as a shortcut, the *AddItem*. Consider this example:

```
tcTest.AddItem ("A1", "A10", "A10")
```

This notation will perform the two following actions:

1. Create and add a ParmNode to the list.

2. Populate the Item, ItemKey, and SortKey values with the respective parameters shown.

Now if we want to add multiple nodes to the Iterator, we can invoke the following inline code:

```
tcTest.Clear()  'let's clear out the iterator first
tcTest.AddItem( "A1", "A10", "A10")
tcTest.AddItem ("A4", "A40", "A40")
tcTest.AddItem ("A2", "A20", "A20")
tcTest.AddItem ("A5", "A50", "A50")
tcTest.AddItem ("A6", "A60", "A60")
tcTest.AddItem ("A7", "A70", "A70")
tcTest.AddItem ("A3", "A30", "A30")
tcTest.AddItem ("A8", "A80", "A80")
```

Note that I have deliberately added them out of key sequence. To examine the TurboCollection list contents, let's perform this simple browse:

```
tcTest.MoveFirst()                        'move to the first
                                          ' Node in the list
Do
    Debug.WriteLine (tcTest.Ref.Item)     'display the Node's contents
tcTest.MoveNext()                         'move to the next node in the list
Loop While tcTest.More()                  'repeat until all nodes
                                          ' have been displayed
```

This loop will print the list nodes in exactly the order they were added. If we want to retrieve any member of the list, perform a *Find* using the Itemkey value, as shown here:

```
pWrk = tcTest.Find("A30")
Debug.Writeline( pWrk.Item)  'should be "A3"
```

or

```
Debug.WriteLine(tcTest.Find("A30").Item)
```

To reorganize information, simply browse the list once and set the SortKey to a unique collating value prior to sorting. We've already done this upon creation of the list, so let's invoke it. I can also invoke a *Dump*, as follows, when it's done to review the contents:

```
tcTest.SortList (strParms:="Descending")
tcTest.Dump()
```

This notation will execute a *descending* sort on the nodes in the list, then dump the result to Visual Basic's "Immediate Window" for review.

By default, the SortList will perform an ascending sort. Let's try that now:

```
tcTest.SortList()
tcTest.Dump()
```

On a sort, the TurboCollection does not move data or ParmNodes around in memory. The TurboCollection organizes information using the ParmNode's next/previous pointers, thus, when reorganizing, it simply reshuffles forward and backward references without moving the actual information. This results in blinding speed in data movement and manipulation. More on this in Chapter 4: Power Tools.

The *Find()* function, while fast on smaller lists, has a diminishing return on larger lists. To optimize this, the TurboCollection uses *binary trees* to reorganize information. A binary tree takes a sorted list and creates a navigable tree-structure supporting a binary search. Recall that a binary search will split a sorted list into two, select a targeted half for continuing the search, then repeat the split-and-find operation until it finds the target key. Statistically, a list with 10,000 elements will require a maximum of 13 compares to find any key, with an average of 11 compares per search. Now that's some serious search power!

Let's use the current list and build a tree as follows:

```
tcTest.TreeBuild()
tcTest.Dump()
```

The *TreeBuild* function will automatically sort the list, then perform the build. The tree structure is best suited for large, stable lists of information, usually those pulled from a database source. However, anywhere we need significant speed, the tree will provide it—and once constructed we can still add nodes to it.

These simple exercises served to introduce lists, the Iterator pattern, the container pattern, and how each helps the other to create a more powerful whole. However, the ParmNode has a broader scope of capability: that of containing and describing a data point with embedded metadata.

Property Manipulation

Within any class, adding functionality for the storage and retrieval of properties often causes *interface-bloat,* which changes or updates the class interface to expose new properties. Each time a user requests new features, developers often race back to their software and UML drawings and start drafting the new interface. The ideal solution should require changes only to the internal software, not the interfaces. How is this accomplished?

For example, let's say a *RocketEngine* class goes through several releases and the end user (an intergalactic shipper) requests some brand-new functionality in the form of *warp speed.* This entails the ability to jump into hyperdrive, warp space, and reduce time-to-market. The Intergalactic Space Administration has cleared the way for your engine design, so now you need to modify the onboard computers to run them.

In any rocket-engine design, new features like this could require extensive rework. Your mission is to perform the engine and system upgrades and install them without significantly changing any of the core navigation and control software. But how?

Let's say you have these three new properties:

Drive: Possible values include Twin-Ion (normal) and Warp (high-speed point-to-point). Until now, Drive has always been Twin-Ion.

Navigation: Possible values include Visual Flying Rules (VFR) and Gravitational (keying on location and gravity cues of passing planets and stars). Until now, VFR was more than sufficient. High-speed travel will require automated cues.

Inertial Dampening: The jump to hyperdrive can leave the passengers queasy and disoriented. Dampening must be automatic.

The *normal* course of action is to directly attack the RocketEngine's software interface with the following notations:

```
Public Property Drive(enum TwinIon, Warp:Default TwinIon)
Public Property Navigation(enum VFR, Grav: Default VFR)
Public Property InternalDamp(True, False Default False)
```

Unfortunately, declaring these three properties changes the RocketEngine's software interface *and* requires the RocketEngine's software consumers to wire specific functionality into their own code to take advantage of it. In short, this track will initiate the reconstruction and deployment of most system software on board the spacecraft. So much for plug-and-play.

Now let's build our instance of *RocketEngine*:

```
Dim pEngine as New RocketEngine()
```

After the necessary background initialization of the class (including interfacing to the actual engine hardware), we now have the following interface exposed to the control and navigation systems. It's a very simple interface, mapping a single property, called *Item*, and allows it to map values into their appropriate locations *behind* the RocketEngine's interface. The *KeyValue* will be the actual property name, while the *ItemValue* will be the actual value of the property.

```
pEngine.Item("Drive") = "TwinIon"
pEngine.Item("Navigation") = "VFR"
pEngine.Item("Inertia") = "Dampening=False"
```

*917_Ch01_CMP_3 2/21/02 1:34 PM Page 41

I've added one additional key value that will automatically set other key values behind the scenes. With such an interface, both the consumers and the developers have complete freedom to use it without fear of breakage or change. The developer can change literally anything within the RocketEngine core, exposing new features without breaking old ones. Here's an example:

```
pEngine.Item("Warp") = "True"
```

Now let's look at the internal implementations in Listing 1-1. The simple form of *Set Item* will accept a key and a value. It will attempt to change the key's associated value, if it exists, then call *ExamineItem* in order to determine if it should handle the "Warp" property key differently. For brevity, the action associated with "Warp" simply resets the Item values. In practice, it could kick off a whole host of background processing.

Listing 1-1. Item() and ExamineItem() Code

```
Public Property Item(Optional ByVal strK As String = "") As String
  Get
  On Error Resume Next
  Dim strVx As String
   strVx = Nothing                        'set to null in case Find() fails
   strVx = tcItem.Find(strK).Item         'do the find - if found will return value
   Item = strVx                           'return to calling proc
  End Get
  Set(ByVal Value As String)
    On Error Resume Next
     tcItem.Find(strK).Item = Value       'find and set the property
     If Err.Number <> 0 Then              'if not found, then need to add
         tcItem.AddItem(Value, strK, strK) 'else if ok to add then add it
     End If
  ExamineItem(strK, Value)                'and examine it further
   End Set
End Property
```

And now the definition of ExamineItem():

```
Private Sub ExamineItem(ByVal strK As String, ByVal strV As String)
   On Error Resume Next
    Select Case strK
        Case "Warp"                              'if "warp" is the key
         Item("Drive") = "Warp"                  'automatically set the required
         Item("Navigation") = "Gravitational"    'default configuration
         Item("Inertia") = "Dampening=True"
    End Select
 End Sub
```

This very simple strategy provides all consumers with a consistent property interface. In practice, the class initialization should internally build and expose the key-value pairs. The RocketEngine New()does this:

```
Public Sub New()
  Item("Drive") = "TwinIon"
  Item("Navigation") = "VFR"
  Item("Inertia") = "Dampening=False"
End Sub
```

While I've put some hard-wired values into the class definition, I've also cordoned them off into textual constants. Anything that can be reduced to text is a candidate to become structural, behavioral, or configuration metadata. Now that we have simple values under control, let's take a look at object manipulation.

Factories and Object Recycle

Within the scope of any software execution, especially object-modeled software, the need for memory and resource management is critical to maintaining control over the system and its performance. Memory leakage occurs in two primary ways: building instances of objects and not destroying them when appropriate, and destroying objects without the ability to fully reclaim all of the memory they acquired when first instantiated.

The first problem is fairly easy to rectify: simply exercise diligence and destroy objects after we're done with them. This usually requires discipline on the part of the individual developer, and sometimes presents a risk of runaway leakage.

The second problem is impossible to resolve completely. Each time our software instantiates an object, the system allocates memory for it along with additional memory to track the instance. When we destroy the object, the system does not necessarily throw away the tracking memory, only the original memory. Additionally, destroying the object automatically creates an object-sized hole in memory. The system will use this hole later to allocate an object of equal or lesser size. The result is memory fragmentation. Within high-speed systems, this effect can create a significant performance drag by literally throwing away useful memory and requiring its fresh reallocation.

These effects usually don't appear in slow-moving classes at the upper execution tier of a software program. They usually occur with the foundation classes, the ones that are the smallest, most used, and most often forgotten. The bad news is that a lack of attention on the part of our developers can create a runaway memory leak very quickly. The good news is we can track all of our memory and object allocations with a little forethought and the strategic use of the Factory pattern.

Within *main* the method *FactoryInit* calls *FactoryAdd* once for each of four classes already in ProjectOne. Each of these classes has a special interface to support factory generation. This interface allows any member of the class to be a "seed" object for multiple objects of its own kind. Thus FactoryAdd will simply add a single instance of the newly created object (Listing 1-2) and key it on the object's type name, also included in the call.

NOTE *Many languages do not support the ability to convert an object's type into a string value for textual searching, or their typename() functions are too brittle for compiled executables. This is why FactoryInit and FactoryAdd include the class names explicitly as string constants.*

Listing 1-2. Factory Logic Described

```
Private Sub FactoryInit()
  On Error Resume Next
  FactoryAdd ("StringAdapter", New StringAdapter())
  FactoryAdd ("TurboCollection", New TurboCollection())
  FactoryAdd ("ParmNode", New ParmNode())
  FactoryAdd ("Observer", New Observer())
  End Sub
```

Note in Listing 1-2 that for extensive class lists, the FactoryInit could invoke a TreeBuild at the end. This would sort and slice the class list into a binary tree for blazing run time access. Practically speaking, don't try to push all objects into a single factory. Most framework-level functions (such as databases, screens, and so on) warrant their own factories and sometimes are never co-related with the other frameworks or software subsystems. Use multiple factories to increase plug-and-play freedom of subsystems so that none of them are directly dependent on a master factory. I'll show in the next chapter how an *Abstract factory* can provide brokerage to multiple factories without direct dependency. Listing 1-3 shows how to add objects to this factory.

Listing 1-3. FactoryAdd

```
Public Sub FactoryAdd(strK As String, iObj As Object)
  On Error Resume Next
  Dim pWrk As ParmNode
  pWrk = tcFactory.Find(strK)                'try to find it
  If pWrk Is Nothing Then                    'If not already there - then
    tcFactory.AddItem( "", strK, strK)       'add a new Item with the proper keys
    tcFactory.Ref.obj = iObj                 'then set the ParmNode's Obj
reference
  Else
    pWrk.obj = iObj                          'otherwise, it's there,
                                             ' just add the reference
  End If
End Sub
```

Now that the tcFactory has "seed" objects with keys to each, let's build the following factory method to give us access to each type:

```
Public Function Factory(strK As String, iObj As Object) As Object
  On Error Resume Next
  Dim pWrk As ParmNode
  Factory = Nothing                                 'prep in case not found
  Factory = tcFactory.Find(strK).obj.Create(iObj)   'find the seed, create new one
                                                    ' from it, and return it.
End Function
```

Note how the first part of the function presets the Factory return value to Nothing. This is a best practice of proper error control (discussed later in Chapter 5). It will use *Find* on the tcFactory to retrieve the "seed," then call the seed's *Create*. Each member of the factory's list must support a Create function that returns an instance of its own kind.

What then, would be the most appropriate means to reclaim the object once it's no longer necessary? It seems a shame to build a factory function to manufacture them, but not use it to take them back. The *FactoryRelease* method allows a consumer to put the object back into the factory environment.

We will need to expand the roll of the tcFactory TurboCollection. Each node on tcFactory will now hold the seed object *and* another TurboCollection for storing the returns, now *available* objects. This will require the following simple change to the FactoryAdd method (highlighted):

```
Public Sub FactoryAdd(strK As String, iObj As Object)
  On Error Resume Next
  Dim pWrk As ParmNode
  pWrk = tcFactory.Find(strK)                  'find it
  If pWrk Is Nothing Then                      'if not found then
    tcFactory.AddItem( "", strK, strK)         'add it
    tcFactory.Ref.obj = iObj                   'and populate the seed
    tcFactory.Ref.Ref = New TurboCollection()  'add the "available" list here
  Else
    pWrk.obj = iObj                            'otherwise populate the seed
  End If
End Sub
```

We will then need the following new function, *FactoryRelease,* to receive the reclaimed objects:

```
Public Sub FactoryRelease(strK As String, iObj As Object)
  On Error Resume Next
  Dim tcAv As TurboCollection                  'allocate a TurboCollection
  Dim pWrk As ParmNode                         'and a placeholder
  pWrk = tcFactory.Find(strK)                  'find the class type with the key
  tcAv = pWrk.Ref                              'and get the "available" list
  tcAv.AddItem "", strK, strK                  'add the recycled object
  tcAv.Ref.obj = iObj                          'and set it to the list's obj
End Sub
```

Now we'll need to modify the Factory method, as follows, to take advantage of reusing the recycled objects before creating a new one (modification are in bold):

```
Public Function Factory(strK As String, iObj As Object) As Object
  On Error Resume Next
  Dim pWrk As ParmNode
  Dim tcAv As TurboCollection
  Set Factory = Nothing
  Set pWrk = tcFactory.Find(strK)
  Set tcAv = pWrk.Ref                         'get the available list
  If tcAv.DataPresent Then                    'If something Is on it?
     Set Factory = tcAv.Top.obj               'use the node on the top of the list
     tcAv.Remove tcAv.Top                     'then remove the top node
  Else                                        'otherwise it's empty
     Set Factory = tcFactory.Find(strK).obj.Create(iObj)
                                              'so create a new one
  End If
End Function
```

Now we'll just call to initialize the Factory (only once)

```
FactoryInit()
```

and invoke it to get a new instance of the Observer class.

```
Dim oWrk as Observer
oWrk = Factory("Observer")
```

Now put it back

```
FactoryRelease("Observer",oWrk)
```

and go get it again. It's the same object as before, only recycled.

```
oWrk = Factory("Observer")
```

The primary objective is to build a central clearinghouse for object creation and destruction rather than relying on the programmer's discipline alone. The alternative: having direct create/destroy functionality scattered throughout hundreds of thousands of lines of code, and encountering a pervasive memory leak we cannot pinpoint.

These constructs are more than strategies. They are patterns of plug-and-play capability that exist in every software program. By directly identifying and leveraging them, we enable them as building blocks for supporting all sorts of applications.