Software Development on a Leash

DAVID C. BIRMINGHAM
WITH VALERIE HAYNES PERRY

Apress™

Software Development on a Leash
Copyright © 2002 by David C. Birmingham with Valerie Haynes Perry

ISBN (pbk): 1-893115-91-7
Printed and bound in the United States of America 12345678910

Technical Reviewer: Tom Schraer
Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore, Karen Watterson, John Zukowski
Managing Editor: Grace Wong
Development Editor and Indexer: Valerie Haynes Perry
Copy Editor: Kim Goodfriend
Production Editor: Julianna Scott Fein
Compositor: Diana Van Winkle, Van Winkle Design
Cover Designer: Tom Debolski
Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010
and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.
Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 901 Grayson Street, Suite 204, Berkeley, CA 94710.
Phone 510-549-5930, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

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.

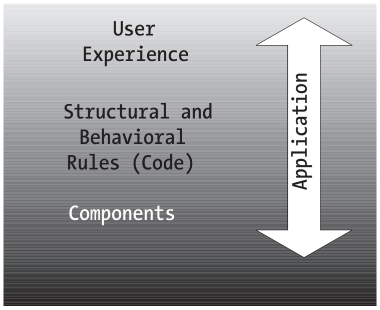

Figure 1-1. The Common Development Product weaves the application into the program so the two are hopelessly entangled.

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.

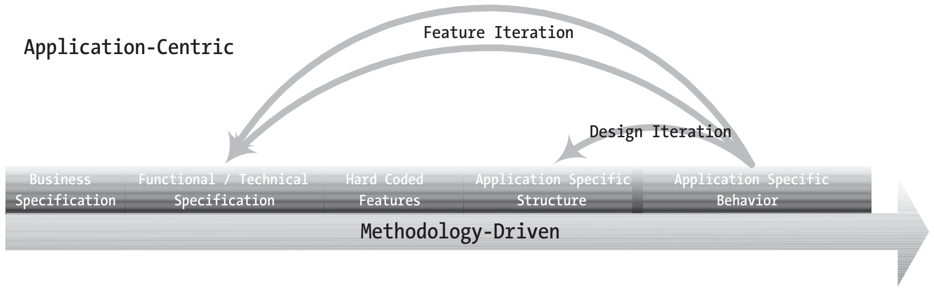

Figure 1-2. The Common Development Model for delivering application-centric programs is great for solution deployment but flawed for software development.

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).*

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:

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

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.*

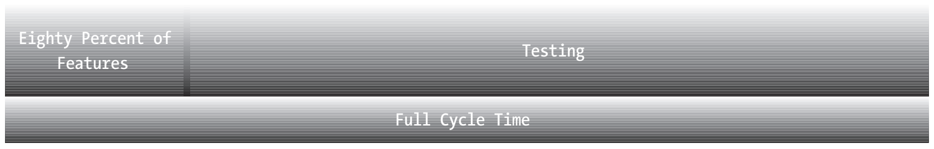

Figure 1-3. The application-centric model automatically protracts testing.

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

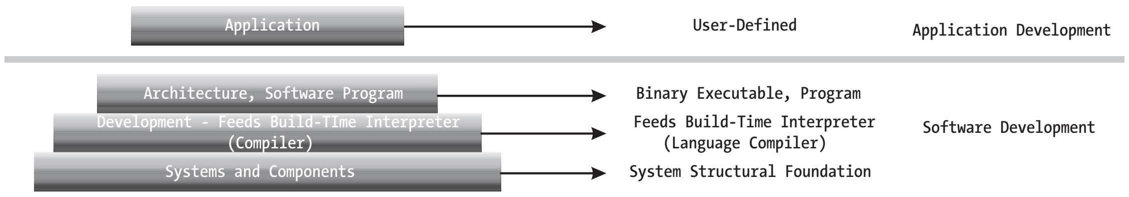
Application
User-Defined
Application Development
Architecture, Software Program
Binary Executable, Program
Development - Feeds Build-TIme Interpreter (Compiler)
Feeds Build-Time Interpreter (Language Compiler)
Software Development
Systems and Components
System Structural Foundation

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.

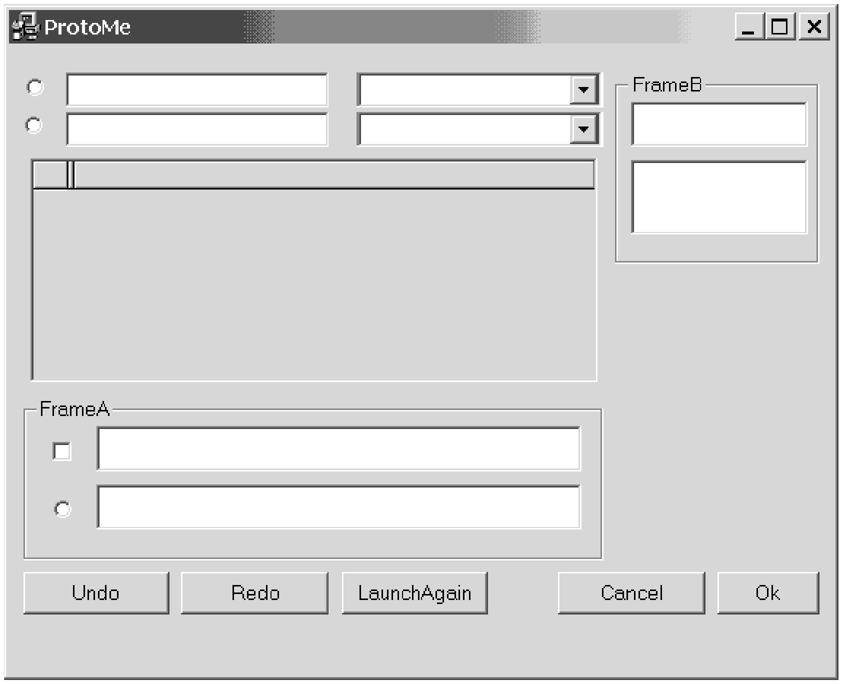

Figure 1-5. A display from the program file vUIMDemo.exe

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).

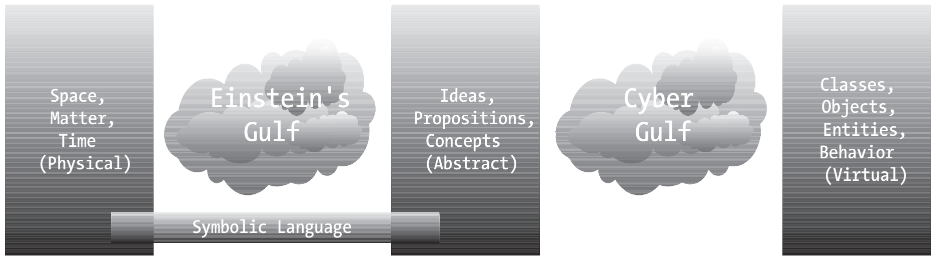
Space, Matter, Time (Physical)
Einstein's Gulf
Ideas, Propositions, Concepts (Abstract)
Cyber Gulf
Classes, Objects, Entities, Behavior (Virtual)
Symbolic Language

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.