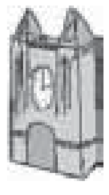
CODE CRAFT

THE PRACTICE OF WRITING EXCELLENT CODE

by Pete Goodliffe

NO STARCH PRESS

San Francisco



14

SOFTWARE ARCHITECTURE

Laying the Foundations of Software Design

In this chapter:

- What is software architecture?
- How is software architecture different from code design?
- The qualities of good architecture
- Overview of key architectural styles

Architecture is the art of how to waste space.
—Philip Johnson

Go into a city. Stand in the middle of it. Look around. Unless you've picked an unusual place, you will be surrounded by a large number of buildings of varying ages and styles of construction. Some fit into their surroundings sympathetically. Others look totally out of place. Some are aesthetically pleasing and seem well proportioned. Others are downright ugly. Some will still be there in 100 years' time. Many will not.

The architects who designed these buildings took a lot into consideration before they put pencil to paper. During the process of design, they worked carefully and methodically to ensure that the building was feasible to fabricate, and they balanced all the contending forces: user requirements, construction methods, maintainability, aesthetics, and so on.

Software is not made of bricks and mortar, but the same careful thought is required to ensure that a system meets similar sets of requirements. We have been erecting buildings far longer than we've been writing software, and it shows. We're still learning about what makes good software architecture.

In this little foray into the world of software architecture, we'll investigate some common architectural patterns and look at what software architecture really is, what it really isn't, and what it's used for.

UNDERGROUND MOVEMENT

I joined a project that had produced a large amount of undocumented software, erected without plan or purpose, with no architect to guide the construction process. Naturally, it had become an unsightly carbuncle. The time came when we needed to understand how it all *really* worked, and an architectural diagram of the system was drawn up. There were so many different components (many largely redundant), inappropriate interconnections, and different methods of communication that the diagram was an intense jumble of tightly woven lines in many interpretive colors—almost as if a spider had fallen into a few different cans of paint and then spun psychedelic webs across the office.

Then it struck me. We had all but drawn a map of the London Underground. Our system bore such a striking resemblance, it was uncanny—it was practically incomprehensible to an outsider, with many routes to achieve the same end, and the plan was still a gross simplification of reality. This was the kind of system that would vex a traveling salesman.

The lack of architectural vision had clearly made its mark on the software. It was hard to work with and hard to understand, with bits of functionality strewn across completely random modules. It had gotten to the point where the only useful thing you could do with it was throw it away.

In software construction, as in building construction, *the architecture really matters*.

What Is Software Architecture?

Is this just another term that stretches the *building* metaphor a little thinner (see "Do We Really *Build* Software?" on page 177)? Maybe so, but it is a genuinely useful concept. Software architecture is sometimes known as *high-level design*; regardless of the terms used, the meaning is the same. Architecture is a more evocative description of the concept.

Software Blueprints

As an architect prepares his blueprint for a building, the software architect prepares a blueprint for the software system. However, while a building's blueprint is a rigorously detailed plan with all the important features included, our software architecture is a top-level definition, an overview of the system that specifically avoids too much detail. It is macro, not micro.

In this high-level view, all implementation details are hidden; we just see the essential internal structure of the software and its fundamental behavioral characteristics. The architectural view does the following:

- Identifies the key software modules (or components, or libraries; at this point call them what you like—*blobs*)

- Identifies which components communicate with each other
- Helps to identify and determine the nature of all the important interfaces in the system, clarifying the correct *roles and responsibilities* of the various subsystems

This information allows us to reason about the system as a whole without having to understand how every individual part will work. The architecture provides a framework into which the later development fits. It shows how work can be split between teams and allows you to weigh different implementation strategies.

Not only does the architecture give a picture of how the system is composed, it also shows how it should be extended over time. In large teams, a program will develop more elegantly when there's a clear, unified vision of how the software should be adapted, of what should be put in each module, and of how modules connect.

KEY CONCEPT *The architecture is the single largest influence on the design and future growth of a software system. It is therefore* essential *to get it right in the early stages of development.*

As an up-front activity, the architecture is our first chance to map the *problem domain* (the Real World problem we are solving) to a *solution domain.* There isn't always a simple one-to-one mapping of objects and activities between the two, so the architecture shows how to think about one in terms of the other.

Exactly what needs to be addressed by the software architecture will differ from project to project. The target platform is not important at this stage; it may be possible to implement the architecture on a number of different machines using different languages and technologies. However:

- For certain projects, it may be important to specify particular hardware components, most likely for embedded designs.
- For a distributed system, the number of machines and processors and the split of work between them might be an architectural issue. Minimum and average system configurations should be considered.
- The architecture may also describe specific algorithms or data structures if they are fundamental to the overall design (although this is far less likely).

There is always a trade-off. The more information that is set in stone at the architectural level, the less room for maneuverability there is at a later design or implementation stage.

Points of View

In physical architecture, we use a number of different drawings or views of the same building: one for the physical structure, one for the wiring, one for the plumbing, and so on. Similarly, we develop different software views in the architectural process. Four views are commonly recognized:

The conceptual view
Sometimes called the *logical view,* this shows the major parts of the system and their interconnections.

The implementation view
This view is seen in terms of the real implementation modules, which may have to differ from the neat conceptual model.

The process view
Designed to show the dynamic structure in terms of tasks, processes, and communication, this view is best used when there's a high degree of concurrency involved.

The deployment view
Use this view to show the allocation of tasks to physical nodes, in a distributed system. For example, you may split functionality between a database server and a farm of web interface gateways.

You don't start with all of these. Particular views arise as development work progresses. The main result of the initial architectural phase is the *conceptual view*, and that's what we're concentrating on here.

FOR WHAT IT'S WORTH

Software architecture has wide-ranging implications—far beyond the initial structure of the code, right into the heart of the software factory. The architecture will be a lasting legacy, both in the technological and practical realms. Architecture affects how the code will grow and how teams of people will work together to extend it; software design affects workflow. With a three-tiered architecture, you'll end up with *three* teams of people working on the separate parts. There will probably be *three* sets of admin staff too, and *three* management reporting lines. Someone's early design decision will affect which desk you sit at.

Since the architecture determines how malleable the software is and how well the codebase can accommodate future requirements, it ultimately influences the commercial success of your company. A bad architecture is more than just inconvenient—it could cost you your livelihood. Serious stuff.

As programmers, it affects us most directly—it will affect how fun our work will be. No one wants to labor intensely to add a minuscule feature that would have taken two seconds with a correct initial design. At conception, check that the architecture supports what *you* think it should, not just what the architects believe.

Where and When Do You Do It?

The architecture is captured in a high-level document called something imaginative like the *architecture specification.* This specification explains the system's structure and shows how it fulfills the requirements, including important issues like the strategy to reach any performance requirements and how acceptable fault tolerance will be achieved.

KEY CONCEPT *Capture system architecture in a known place; a document accessible to everyone involved—programmers, maintainers, installers, managers (perhaps even customers).*

The architecture is the initial system design. It is therefore the *first* developmental step after the requirements have been agreed upon. It's important to generate a specification up front because it provides a first chance to review and validate the design decisions that will have the most significant impact on the project. It will expose weaknesses and potential problems. Reversing a bad decision this early on will save a lot of time, effort, and money. It's expensive to change the foundation of a system once a lot of code has been built upon it.

Architectural work is a form of design, but it is separate from the module design phase, and distinct from low-level code design, although it certainly overlaps somewhat. Later work on detailed design may feed changes back up to the system architecture. This is natural and healthy.

WHOSE JOB?

We've seen that software architecture affects *everyone* on the project—not just the programmers. In contrast, the architecture is determined by a far smaller group of people. What a responsibility.

The architecture designer is called a *software architect*. This is a grandiose title and, like *engineer*, somewhat contentious. "Real" architects must study, qualify, and reach levels of professional excellence to even be called architects. There are no such requirements in the software world.

Software architects are among the project initiators, working right at the beginning of the development cycle. As development ramps up, programmers will join the effort to implement this established architecture.

However, on smaller projects requiring less specialized architectural experience, the programmers themselves will devise the architecture. No big guns are drafted in. Be ready to contribute to architectural design.

What Is It Used For?

Architecture is the initial system design. But its uses stretch even further. We use the system architecture to:

Validate

The architecture is our first chance to validate what is going to be built. With it, we can mentally check that the system will meet all requirements. We can check that it really is feasible to build. We can ensure that the design is internally consistent and hangs together well with no special cases or gratuitous hacks. Nasty blemishes in the high-level design will only lead to more dangerous hacks at lower levels.

The architecture helps to ensure that there is no duplication of work, wasted effort, or redundancy. We use it to check that there are no gaps in the strategy, that we have included all the necessary pieces. We ensure that there will be no mismatches as separate sections are brought together.

Communicate

We use the architecture specification to communicate the design to all interested parties. These may be system designers, implementers, maintainers, testers, customers, or managers. It's the primary route to understand the system and is an important piece of documentation that should *always* be kept up to date as changes are made.

KEY CONCEPT *An architecture specification is an essential device to communicate the shape of your system. Ensure that you keep it in sync with the software.*

The architecture conveys the vision of your system, mapping the problem domain to the solution domain. It should neatly identify how future extensions fit in, helping to maintain the system's *conceptual integrity.* (Brooks 95) It implicitly provides a set of conventions and contains an element of style. For example, it's clear that you shouldn't introduce a new component with custom socket-based communication if the rest of the design uses a CORBA infrastructure.

The architecture provides a natural route into the next level of design without being too prescriptive.

Discriminate

We use the architecture to help us make decisions. For example, it identifies build versus buy decisions, determines whether a database is required, and clarifies the error-handling strategy. It will flag problem areas, areas of particular risk on the project, and help us plan to minimize this risk. Just as an architect's primary goal is to ensure his building stays up when it's built—under all expected conditions (and some unusual conditions too)—so should we ensure the resillience of our software structure. A little wind or extra load shouldn't topple the thing over.

We need this systemwide perspective to make the appropriate tradeoffs, ensuring that the design meets its required properties. These important issues are considered at the beginning rather than grafted in toward the end of development.

KEY CONCEPT *Make all software design decisions in the context of the architecture. Always check that you're working in line with the system vision and strategy. Don't create a little wart on the side that doesn't complement anything else.*

Of Components and Connections

Architecture mostly concerns itself with *components* and *connections.* It determines the number and type of each.

Components

Architecture captures information about each component, whatever *component* means in the architecture's context. It could be an object, a process, a library, a database, or a third party product. Each of the system's components is identified as a clear and logical unit. Each performs one task and does it well. No component includes a kitchen sink unless there's a specific kitchen-sink module.

While it won't dwell on component implementation issues, the architecture will describe all exposed facilities and perhaps the important externally visible interfaces. It defines the *visibility* of the component: what it can see and what it can't, and what can see it and what can't. Different architectural styles imply different visibility rules, as we'll see later.

ARCHITECTS VS. MARKETERS

An architecture is inadequate if it doesn't fulfill the product requirements for initial deployment or any future development; design quality is about more than just technical excellence. Technical issues must be addressed alongside product management and marketing considerations.

There is no point in developing a product that no one wants; it would obviously be a huge waste of time. But you can miss vital business opportunities by omitting marketing requirements from technical consideration. The marketing department identifies core business objectives including sales strategies (do you charge a one-off fee or employ a licensing/billing model?), the product's position in the marketplace (is it a high-end, feature-packed, high-cost product or a cheap, mass-produced item?), and the importance of a unique brand running through the system.

In some situations, visibly good architecture may be a *unique selling point* and may provide a strong competitive advantage. Other markets care less about the internal system structure, but an architecture that anticipates and handles future customer requirements is still essential to establish and maintain a strong market position.

Technical architects must work closely with the marketing decision makers to understand how new software will fit into the company's overall strategy and what the customer requires for a truly exceptional solution. The software architecture will address marketing issues such as usability, reliability, upgradeablity, and extensibility. Each of these has a real influence on the software design. Support for different charging methods alone may have a huge impact on the profitability of the project—the inclusion of rich logging support will pave the way for per-transaction billing, which may lead to increased product revenue. However, it may mandate the inclusion of additional security and fraud-prevention measures in the architectural planning.

Marketing requirements feed into the technical architecture. Technical considerations will also feed back to the marketing strategy. A truly great architecture is born when technical and strategic visions meet to create a product that stands out from its competitors.

Connections

The architecture identifies all the inter-component connections and describes the connection properties. A connection may be a simple function call or data flow through a pipe. It may be an event handler or a message passing through some OS or network mechanism. A connection can be *synchronous* (blocking the caller until the implementation has completed the request) or *asynchronous* (returning control to the caller immediately and arranging for a reply to be posted back at a later date). This is important, since it affects the flow of control around the system.

Some communication is indirect (and consequently quite subtle). For example, components can share certain resources and talk through them—rather like posting messages on a shared whiteboard. Examples of shared communication channels are: a subordinate component, a shared memory region, or something as basic as the contents of a file.

What Is Good Architecture?

The key to good architecture is *simplicity*. A few well-chosen modules and sensible communication paths are the aim. It also needs to be *comprehensible*, which often means visually represented. We all know that *a picture speaks a thousand words.*

KEY CONCEPT *Good system architecture is* simple. *It can be described in a single paragraph and summarized in one elegant diagram.*

In a well-designed system, there should be neither too few nor too many components. This criterion scales with the size of the problem. For a small program, the architecture may fit on (or even be done on) the back of an envelope, with just a few modules and some simple interconnections. A large system naturally requires more effort and more envelopes.

Too many fine-grained components lead to an architecture that is bewildering and hard to work with. It implies that the architect has gone into too much detail. *Too few* components means that each module is doing far too much work; this makes the structure unclear, hard to maintain, and hard to extend. The correct balance is somewhere between the two.

The architecture does not dictate the inner workings of each module—that's what module design is for. The goal is that each module should know very little about the other parts of the system. We aim for low coupling and high cohesion (see "Modularity" on page 247) at this level of design, as with all others.

KEY CONCEPT *Architecture* identifies *the key components of the system and how they interact. It doesn't define* how *they work.*

The architecture specification lists the design decisions made and makes it clear why this approach is being favored over any alternative strategies. It doesn't need to labor these other approaches, but should justify the chosen architecture and prove that some serious thought went into it. It must have correctly identified the primary goal of the system: For example, *extensibility* is a different game from *performance* and will lead to different architectural design decisions.

A good architecture leaves room for maneuverability; it allows you to change your mind. It may specify that we wrap third party components with abstract interfaces so we can swap one version out for another. It may suggest technologies that make it easy to select different implementations during deployment. As a project gains momentum, the correct implementation choices become clear—they aren't always obvious at first. A successful architecture is flexible, providing a mechanism for nimble design during these initial uncertainties. The architecture is the first pivot on which to balance contending forces; it will show how we trade one quality for another.

KEY CONCEPT *A good architecture leaves space for maneuverability, extension, and modification. But it isn't hopelessly general.*

The architecture must be clear and unambiguous. Preexisting, well-known architectural styles or well-known frameworks are best (see the next section for more on these). Architecture must be easy to understand and work with.

Like a good design, good architecture has a certain aesthetic appeal that makes it *feel* right.

Architectural Styles

Form ever follows function.
—Louis Henry Sullivan

Just as an immense gothic cathedral and a quaint Victorian chapel, or an imposing tower block and a 1970s public lavatory employ different architectural styles, there are a number of recognized software architectural styles that a system may be built upon. A style may be chosen for various reasons, good or bad—perhaps on sound technological grounds, or perhaps based on the architect's prior experience, perhaps even by what style is currently in fashion. Each architecture has different characteristics:

- Its resilience to changes in the data representation, algorithms, and required functionality
- Its method of module separation and connection
- Its comprehensibility
- Its accommodation of performance requirements
- Its consideration of component reusability

In practice, we might see a mixture of architectural styles in one system. Some data processing may progress through a pipe and filter process, while the rest of the system employs a component-based architecture.

KEY CONCEPT *Recognize the key architectural styles and appreciate their pros and cons. This will help you to sympathetically work with existing software and perform appropriate system design.*

The following sections describe some of the common architectural styles. And then compare them to pasta.

No Architecture

A system always has an architecture, but like my London Underground project, it may not have a *planned* architecture. Before long, this state of affairs becomes an albatross around the neck of your development team. The resulting software will be a mess.

Architecture as Pasta: *Spaghetti Ball*

Messy, uncontrollable, unmanagable morass of interwoven gloop.

Defining an architecture is essential if you want to build good software. Not planning an architecture is a surefire way to doom development before you've even started.

Layered Architecture

This is probably the most commonly used architectural style in conceptual views. It describes the system as a hierarchy of layers, with a building-block approach. It is a very simple model to comprehend; even a non-techie can quickly grasp what it's telling him.

Architecture as Pasta: *Lasagne*

Several distinct layers, arranged one on top of another.

Each component is represented by a single block in the stack. The positions in the stack indicate what lives where, how the components relate to each other, and which components can "see" which other components. Blocks may be placed alongside each other on the same level and can even become tall enough to span two layers.

A famous example of this is the OSI seven-layer reference model for network communication systems. (ISO 84) A more interesting example is the Goodliffe seven-layer trifle reference model shown in Figure 14-1.

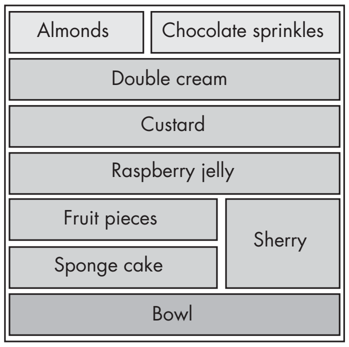


Figure 14-1: The Goodliffe seven-layer trifle reference model

At the lowest level of the stack, we find the hardware interface, if the system does indeed interact with physical devices. Otherwise, this level is reserved for the most basic service, perhaps the OS or a middleware technology like CORBA. The highest level will likely be occupied by the fancy interface that the user interacts with. As you rise further up the stack, you move further away from the hardware, happily insulated by the layers in between in the same way that the roof of a house doesn't have to worry about the magma at the earth's core.

At any point, you can brush out all the lower layers and slot in a new implementation of the layer below—the system will function as before. This is a key point: It means that you can run the same C++ code on any computing platform that supports your C++ environment. You can swap the hardware platform without touching your application code—relying on the OS layer (for example) to swallow the technical differences. Handy.

Higher levels use the public interfaces of the layer directly below. Whether they can use the public interfaces of the lower levels depends of your definition of layering. Sometimes the diagram is fiddled to represent this, like the sherry brick in the trifle stack. Whether or not components on the same layer can interconnect is also not rigidly defined. You certainly can't use anything from a higher level; if you break this edict, you no longer have a layered architecture, just a meaningless diagram drawn in stack form.

As you can see, most layer diagrams are informal. The relative size and position of boxes gives a clue as to importance of a component, and that is generally sufficient as an overview. Component connections are implicit, and the methods of communication irrelevant. (However, this can be a key architectural concern for the efficiency of the system—you won't send gigabytes of data down an RS232 serial port.)

Pipe and Filter Architecture

This architecture models the logical flow of data through the system. It is implemented as a string of sequential modules that each read some data, process it, and spit it out again. At the start of the chain is a data generator (maybe a user interface or perhaps some hardware harvesting logic). At the end is a data sink (perhaps the computer display or a log file). It's the old through-the-grapevine telephone game in digital form. The data flows down the pipe, encountering the various filters en route. The transformations are usually incremental; each filter does a single simple process and tends to have very little internal state.

Architecture as Pasta: *Cannelloni*

Good conduit for its contents, suits particular situations very well.

The pipe and filter architecture requires a well-defined data structure between each filter; it has the implicit overhead of repeatedly encoding the output data for transmission down the pipe and parsing it back again in each subsequent filter. For this reason, the data stream is usually very simple—just a plaintext format.

This architecture makes it easy to add functionality by just plugging a new filter into the pipeline. Its main downside is error handling. It is hard to determine where an error originated in the pipeline by the time a problem manifests itself at the sink. It's cumbersome to pass error codes down the chain toward the output stage; they need extra encoding and are hard to handle uniformly over several separate modules. The filters may use a separate error channel (e.g., `stderr`), but error messages can get mixed up all too easily.

Client/Server Architecture

A typically network-based architecture, the client/server model separates functionality into two key pieces: the *client* and the *server*. It differs from the older *mainframe* style of networked design in the division of work between each part; a mainframe "client" is a dumb terminal—little more than a means to capture and transmit keypresses, with some output display.

Architecture as Pasta: *Gemelli*

Two complementary strands, woven tightly together.

A SLAP IN THE INTERFACE

A key software construction principle is *modularity*, designing systems from replaceable components. This is almost a "LEGO brick" approach to construction. Done correctly, you should be able to take out a square, blue brick and replace it with a slightly fancier red one. If the bricks are the same size and shape and have the same kinds of connector, they will fit into the same hole and do the same job.

How do we implement this in software? We define *interfaces*; these are our connection points and component barriers. They define the size and shape of each component (as seen from the outside, at least) and determine what you have to do to provide a like-for-like replacement. Key types of interfaces are:

APIs
Application programming interfaces (APIs) are specified as collections of functions in a physically linked application. To replace a component that implements a particular API, you just reimplement all the functions and relink the code.

Class hierarchies
You can design an abstract "interface" class (in Java and C#, you'd actually define an `interface`). Then provide any number of concrete implementations that derive from it and implement that interface.

Component technologies
Technologies such as COM and CORBA allow your program to determine the correct implementation component at run time. Typically, interfaces are defined in an abstract *Interface Definition Language (IDL)*. The beauty of this approach is that components can be written in any language. It requires middleware or OS support.

Data formats
These formats can form a connection point in designs focused on the movement of data rather than the flow of control. You can replace any component in the data chain with an analog that interacts with the same data types.

As you can see, architecture—indeed, most of software design—is about crafting appropriate interfaces. Each of these interface techniques maps to a particular architectural style. Pick an interface mechanism that complements the architecture.

The clients of a client/server architecture are richer, more intelligent, and generally able to present data in an interactive, graphical manner. Here is a more detailed look at the role of the two elements:

Server
The server provides certain well-defined services to clients. It will generally be a powerful computer dedicated to providing specific functionality or to managing a resource (shared files, printers, a database, or pooled processing power).

The server waits for requests from clients and responds to them. It may be able to handle any number of simultaneous client connections or might be limited to certain usage patterns.

Client
The client consumes a server's services. It sends off requests and processes the results that are returned. Some clients are dedicated terminals which only fulfill one role; other clients serve many

functions (for example, a "client" application may run on a standard desktop PC that can also browse the web and view email).

There can be many different types of clients using one server, all performing the same set of requests but in different ways. One client might be web based, one might have a GUI interface, while another might provide command line access.

This client/server approach is sometimes known as a *two-tier* architecture, for obvious reasons. It's very common and is seen throughout the software development world. The means of communication between client and server varies—it's simplest to use standard network protocols, but you may also see use of remote procedure calls (RPC), remote SQL database queries, or even proprietary application-specific protocols.

There are various ways of splitting work between the two components. The main application logic (also known as *business logic*) may run on either the client or server, depending on how intelligent and specialized the client is supposed to be. As more application logic is pushed down to the client, the design becomes less flexible—separate clients have to reimplement similar features, negating the benefit of the central server. Clients are generally concerned with providing sensible human interfaces to the published server functionality.

We sometimes see an extension of this two-tier design, which introduces another layer (the *middle tier*). This component is explicitly designed to contain the business logic, separating it from both the client application (which is now most definitely only an interface) and the back-end data storage. This is a *three-tier* architecture.

A client/server approach is different from a *peer-to-peer* architecture, where no network node has more capability or importance than any other. Peer-to-peer architectures are harder to deploy but more tolerant of faults. The client/server design is crippled when the server is unavailable (through some software fault or routine maintenance): No client will be able to operate until the server comes back to life. For this reason, client/server installations generally require a designated administrator to keep all systems running smoothly.

Component-Based Architecture

This architecture decentralizes control and splits it into a number of separate collaborating *components* rather than a single monolithic structure. It is an object-oriented approach, but doesn't necessarily require implementation in an OO language. Each component's public interface is typically defined in an *Interface Definition Language (IDL)* and is separate from any implementation, although some component technologies (like .NET's built-in component support) can determine this from the implementation code itself.

Architecture as Pasta: *Conchiglie*

Separate little bits floating in some connecting goo.

Component-based design arrived with the lure of assembling applications quickly out of prefabricated components, supposedly enabling plug-and-play solutions. It's still up for debate how much of a success this has been. Not all components are designed for reuse (it's hard work), and it's not always easy to find a component that does what you want it to do. It's easiest for UIs, where popular frameworks and established marketplaces exist.

The core of a component-based architecture is a communication infrastructure, or *middleware*, which allows components to be plugged in, to broadcast their existence, and to advertise the services they provide. Components are used by looking up this information through a middleware mechanism, rather than by hardwiring a direct connection between two components. Common middleware platforms include CORBA, JavaBeans, and COM; each have different strengths and weaknesses.

A component[1] is essentially an implementation unit. It honors one (maybe more) specific published IDL interfaces. This interface is how clients of the component interact with it. There are no back doors. The client is concerned with dealing with an instance of that interface, rather than in how the component is implemented.

Each component is an individual, independent piece of code. Behind its interface, it implements some logic (perhaps business logic or user interface activity) and contains some data, which may just be local or may be published (say a filestore or database component). Components don't need to know much about one another. If they *are* tightly coupled, then the architecture is just an obfuscated monolithic system.

Component-based architectures can be deployed in a networked environment with components on different machines, but they can just as easily exist as a single machine installation. This may depend on the type of middleware in use.

Frameworks

Instead of developing a new architecture for a specific project, it may be appropriate to use an existing *application framework* and add development into that skeleton. A framework is an extensible library of code (usually a set of cooperating classes) that forms a reusable design solution for a particular problem domain. Most of the work in a framework has been done for you, with the remaining pieces following a fill-in-the-blanks approach. Different frameworks follow different architectural models; by using a framework, you commit to its particular style.

Architecture as Pasta: *Canned Ravioli*

Most of the work's already been done for you. Just heat and serve.

[1] We've already talked about components as modules, ephemeral implementation units. But this is a new definition for the word, quite specific to the world of component-based architecture. Sadly, the terms are overloaded with multiple meanings.

Frameworks differ from traditional libraries in the way they interact with your code. When using a library, you make explicit calls into the library components under your own thread of control. A framework turns this around; it is responsible for the structure and flow of control. It calls into your supplied code as and when necessary.

Sitting alongside off-the-shelf frameworks are architectural *design patterns*. While not an architectural style in their own right, patterns are small-scale architectural templates. They are micro-architectures for a few collaborating components, distilling a recurring structure of communication. Architectural patterns describe common component structures at the architectural design level, explaining how they fulfill the requirements of a given context. Patterns are a set of design best practices, described in the ubiquitous GoF book (Gamma et al. 94) and numerous subsequent publications (see "Design Patterns" on page 255).

In a Nutshell

The Roman architect Vitruvius made a timeless statement of what constitutes good architectural design: strength (*firmitas*), utility (*utilitas*), and beauty (*venustas*). (Vitruvius) This holds true for our software architectures. Without a well-defined, well-communicated architecture, a software project will lack a cohesive internal structure. It will become brittle, unstable, and ugly. Eventually, it will reach a breaking point.

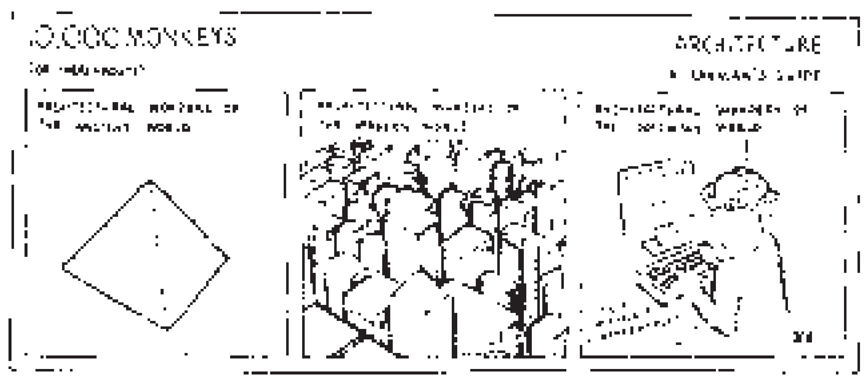
All this talk of pasta has made me hungry. I'm off to build a seven-layer reference trifle. . . .

Good programmers . . .

- Understand their software architecture and write new code within it
- Can apply the appropriate architecture to each design scenario
- Create simple architectures that are beautiful and elegant—they appreciate the aesthetics of software design
- Capture the system architecture in a live document that is continuously updated
- Relay problems with the structure back to the system architects in an attempt to improve the design

Bad programmers . . .

- Write code regardless of any overall architectural vision—resulting in unsympathetic blemishes and unintegrated components
- Fail to perform any high-level design before ploughing into code, ignoring any architectural alternatives
- Leave architectural information locked inaccessibly in people's heads or in a dangerously out-of-date specification
- Put up with inadequate architectures, adding more badly designed code rather than fixing the underlying problems—they can't be bothered to open a larger can of worms



See Also

Chapter 12: An Insecurity Complex
Security concerns must be addressed by a system architecture.

Chapter 13: Grand Designs
Code *design* is the subsequent level of code construction.

Chapter 15: Software Evolution or Software Revolution?
Architecture is the start of your software's life, but it is by no means the only thing that steers its development.

Chapter 22: Recipe for a Program
Where architectural design fits into the software development process.

Get Thinking

A detailed discussion of these questions can be found in the "Answers and Discussion" section on page 522.

Mull It Over

1. Define where *architecture* ends and *software design* begins.
2. In what ways can a bad architecture affect a system? Are there parts that wouldn't be affected by architectural flaws?
3. How easy is it to repair architectural deficiencies once they become apparent?
4. To what extent does architecture affect the following things?
    a. System configuration
    b. Logging
    c. Error handling
    d. Security

5. What experience or qualifications are required to be called a *software architect*?
6. Should sales strategy influence architecture? If so, how? If not, why?
7. How would you architect for *extensibility*? How would you architect for *performance*? How do these design goals affect the system, and how do they complement one another?

Getting Personal

1. How diverse is the range of architectural styles to which you are accustomed? What do you have the most experience with—how does it affect the software you write?
2. What personal experience do you have of architectures that succeeded or failed? What made them winning solutions or a hindrances?
3. Get every developer on your current project to draw a picture of the system architecture—individually (without talking to anyone) and without any reference to system documentation or the code. Compare the pictures. See what strikes you about each developer's efforts—aside from the relative artistic merit!
4. Do you have an architectural description that's commonly available for your current project? How up to date is it? Which kinds of view are you using? If you needed to explain the system to a newcomer or a potential customer, what would you really need to have documented?
5. How does your system's architecture compare to the architecture of your competitors in the marketplace? How has your architecture been defined to determine your project's success?