

# Software Architecture An Overview



Written By:

Rob Keefer  
Principle Consultant

Strategic Data Systems, Inc.

[www.sds-consulting.com](http://www.sds-consulting.com)

937-886-9405

Strategic Data Systems is a Dayton, Ohio-based software consultancy.

Copyright 2006 by Strategic Data Systems.

THIS STRATEGIC DATA SYSTEMS MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. STRATEGIC DATA SYSTEMS MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. STRATEGIC DATA SYSTEMS DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this document is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works for this document for external and commercial use should be addressed to the SDS Licensing Agent.

## Introduction

The phrase “Software Architecture” has become a nebulous term. At times it may refer to an over-arching framework and at others simply to system interoperability definitions. In this discussion, we refer to *Software Architecture* as the structure of the code that comprises a software system. In particular, we will focus on the code being written, though we will touch briefly on methods for integrating other systems.

To give a formal definition, software architecture is the structure(s) of a system comprising software components, the externally visible properties of those components, called architectural views, and the relationships among them. According to this definition, every software system has a software architecture. The quality of the architecture then becomes the focus of any study of Software Architecture.

Most complex systems have more than one architectural view. As we will see, an Application Service Provider (ASP) system may be built using a Layered Architecture. At the component level though, this system may be built using an object-oriented architecture. If the ASP also provides Web Services, it may also look like a Communicating Processes Architecture from a different perspective. So, for one application there may be many different views of the software architecture. It is good to understand this phenomenon and think about the different views of your system’s architecture as it evolves.

With the advent of Agile Software Development and Lean Methodologies, formal documentation of Software Architecture has taken a back seat to developing working software. This lack of documentation is often the right decision; however, there are two good reasons for making a system’s software architecture visible and understandable: communication and early design decisions.

An architectural vision can facilitate high-bandwidth communication. In home design, an architect may use terms like 'Ranch', 'Cape-Cod', or 'Two-Story', and clients instantly understand what he is talking about. The same is true for software design. When the developers of a system understand what a 'Layered' application is, the architect can save a lot of time describing the system to them. The team-members can use this jargon as a form of rich communication.

An architectural vision can also support early design decisions. Using the home design analogy, placing a bathroom on the second floor of a ranch style house would signal a problem with the architecture. In the same way, a developer talking about the UI of a Batch Process software architecture should cause concern. The architectural vision makes flaws in the design become quickly apparent.

Software architecture is influenced by everyone who has input into what the system does – management, marketing, end users, customers, etc. The architect must attempt to balance the concerns of everyone involved. Therefore, it is important to know and understand the requirements of a system as early in the development process as possible. To this end, architects must identify and actively engage these people to solicit their needs and expectations.

For example, the customer may be concerned with cost to the point of compromising usability. Managers are often concerned with immediate business needs, long-term business goals, and how to meet these objectives with the given team. It is the architect’s job to

work with customers, managers and users in ways that enable everyone involved to understand the issues and arrive at a mutually beneficial solution.

Even though all these people have input into the system, the architecture is mostly influenced by the practices and disciplines of the development team - primarily those of the Architect. Practices such as Test Driven Development, following Coding Standards, and using Design Patterns will greatly increase the quality of the software architecture.

The goal of the Architect is to keep the architecture optimized for current conditions, yet flexible enough to keep it optimized when those conditions change. No practice facilitates this optimization as well as Test Driven Development (TDD). When a team employs TDD, the team can be confident that the growing architecture is solid and optimized for current requirements. When a team does not have the ability to run tests that ensure the stability of the system, members are reluctant to make changes to the architecture. However, when a suite of tests is in place, a team can change anything in the system with confidence. A system can even be left in an evolutionary state, and not cause concern when all of the tests for the system pass.

## Architecture Patterns

An architect dedicated to the discipline of Test Driven Development will still find it useful to begin the system's development with one of five software architecture patterns. These patterns describe the initial skeleton upon which the system will be based. The five patterns are: Call-and-Return, Data-centered, Virtual Machine, Data-flow, and Independent Component.<sup>1</sup>

### Call-and-Return

Most systems built today use some form of a Call-and-Return architecture pattern. This pattern includes object-oriented architectures, remote procedure call (RPC) architectures, and tiered or layered architectures. The primary goal of these approaches is to easily accommodate changes in scale and feature function. For example, one can employ more than one processor using RPC to increase performance, or add new feature functionality using the layered approach.

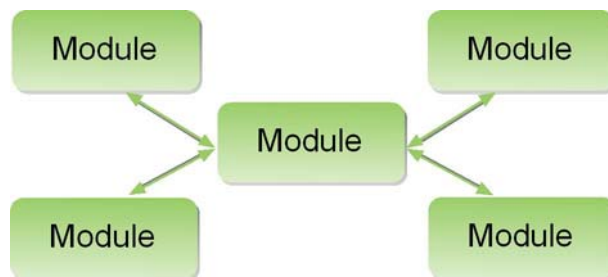


Figure 1: Call-and-Return Architecture

### Data-Centered Pattern

It is often difficult to assimilate data from multiple systems into one view. Data-centered patterns are useful when data integration is important (i.e. Data warehousing). In this architecture, clients and the data store are independent of each other. Thus, it is easy to add new clients, or change views of the data and make these changes available to all clients.

The primary purpose of a Data-centered Architecture pattern is to provide access to a heavily used data store. Clients of the data store are notified of changes in one of two ways: the client is directly notified of changes, or the client queries the data store for changes.

A data store that notifies clients of data changes is called an *active repository*. An active repository broadcasts changes to any client that has subscribed for updates. This model is also called a publish/subscribe model, since the repository acts as the publisher and the clients subscribe to their desired updates.

In a passive repository model, data is simply updated as requested. The clients must request any changes that have been made to the data store. This is the model that typical database applications follow.

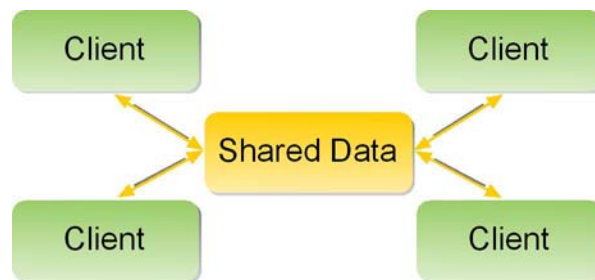


Figure 2: Data-Centered Architecture

### Virtual Machine Pattern

The primary purpose of a Virtual Machine (VM) Architecture pattern is to make a software system portable. The Java Runtime Environment and the Common Language Runtime (from Microsoft .NET) are good examples of virtual machines that can handle portability issues.

Another use for virtual machines is simulation. A VM can simulate a hardware platform or even a system that has not been built. Most developers who build software for cell phones, PDAs, and similar devices use a hardware simulator to test their system before loading the system onto the device. The primary drawback to running a program through a virtual machine is the performance cost resulting from the extra work required to run the program and the simulation code.

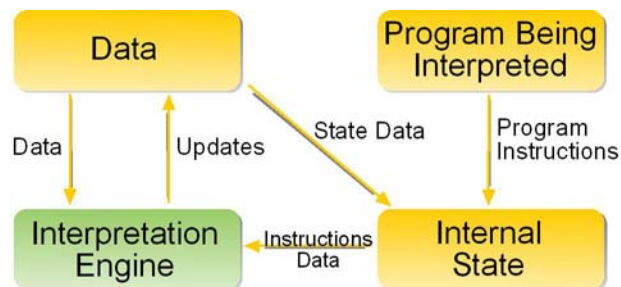


Figure 3: Virtual Machine Architecture

### Data-Flow Pattern

Data-flow Architecture patterns promote reuse of components and are built to accommodate change in feature function. A system built in this pattern will be composed of individual components that operate on data input in series and in succession. Input to the system flows through each component individually until the system is complete. Examples of this

architecture are a series of XSL transforms, and the classic pipe-and-filter mechanism made famous by the inventors of UNIX.

A great advantage of using data-flow architecture is its simplicity. There are no complex components to manage, and any combination of the components can produce a different system. However, due to the way a problem is dissected into individual components, the system is inherently single threaded, making it difficult to increase performance.



Figure 3: Data-flow Architecture

### Independent Component Pattern

Independent Component Architecture patterns are also known as Messaging Architectures. Each component in this architecture communicates with other components through messages. Thus, one component in a system does not have direct control of other components. The messages may be sent to a specific component or broadcast to all interested components through a publish/subscribe mechanism, but there is no guarantee when the components will process the received message.

Typically, event driven systems implement a publish/subscribe mechanism. Publisher components announce what data they are going to broadcast. Subscriber components then register with the publisher for the information they are interested in receiving. A message manager acts as a communication broker between the publisher and subscriber components.

The primary advantage of the Independent Component Architecture is that each component is completely decoupled from the others. This allows the components to run in parallel, facilitating higher scalability and performance.

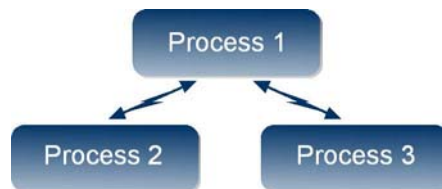


Figure 4: Component Architecture

## Architecture Principles

A good architecture is guided by one or two “architects” – people who assume responsibility for the architecture of the system. These people may also be called the “team leads”. A good architect will take the opportunity to mentor others in the group and have discussions with team members about the architecture and design of the system, but the decisions and ultimate responsibility remain with the architects.

The architect must fully understand the requirements of the system and develop a guiding plan that supports these requirements and accounts for change to as great an extent as possible. Every software system of any significance is subject to change at some point during its development. An experienced architect will be able to determine the priorities of the requirements and understand what needs to be accounted for in the early stages of a system’s development.

As part of the requirements gathering process, the architect should determine a set of metrics that will enable the customers and others interested in the software's success to know when the system is done, and answer the question "What does success look like?" This set of metrics should include measurable traits such as throughput, time for a page to load, etc., as well as more loosely defined qualities such as *information hiding* and *separation of concerns*.

The principle of information hiding generally refers to the practice of encapsulating details of a module, and exposing the module's functionality through a well defined set of interfaces. The concept applies equally well to software architecture, only on a grander scale. The layers or components of a software architecture can provide a set of services, and these services can be exposed through a set of interfaces. The architectural modules should include those that encapsulate idiosyncrasies of the computing infrastructure or third-party systems that are not a part of the system, yet are used by the system. This practice insulates the software from change. If a third-party component is changed, or an architectural component changes, the rest of the system is not affected as long as the interface remains constant.

The principle of separation of concerns generally refers to the practice of having a module do one thing well. In a software architecture, each layer or component is responsible for a set of tasks. In a large system, the architectural modules should reflect a separation of concerns that allows their respective development teams to work largely independently of each other.

It is often difficult to understand how these two principles are implemented in production software applications. Sometimes it is easier to notice when there is a problem rather than construct a feature correctly from scratch. Martin Fowler in his book [Refactoring](#)<sup>2</sup> develops the idea of coding smells. Code smells when it contains certain structures that suggest the possibility of refactoring. Similarly, problems in an architecture can be identified when certain structures are identified. Let us consider four architecture smells:

- Data Entity Virus

A common practice is to map the object model directly into data entities. That is, each class in the object model is transformed into a data entity. This results in a large number of fine-grained data entities and can cause a database to contain redundant data (i.e. become de-normalized), resulting in a detrimental effect on performance.

One approach to purging a system from this virus is to identify the parent-dependent object relationships in the object model and use the relationships as coarse-grained data entities. This will result in fewer entities, where each entity is composed of related data. At this point, normalizing the data would also improve the data integrity of the system.

Another approach to eliminate Data Entity Virus is to consolidate a set of related workflow interactions that involve one or more data entities into a single façade. This will provide a uniform coarse-grained service access layer.

With either approach, the entity must be transformed to a business object before it is used throughout the rest of the system. If the entities are simply used throughout the rest of the system, another architectural smell develops: Business Object Bloat.

- Business Object Bloat

Another blunder is to model each row in a database table as a business object. The result of this modeling is a large number of fine-grained objects, but business objects

are best designed as coarse-grained objects. Such a modeling also ties business objects to changes in the database, which violates the principle of separation of concerns.

A better approach is to develop a set of business objects that the business layer understands how to use. Then transform the business objects into data entities for the data access layer to use, and transform the data entities into business objects for the business layer to use.

If the business objects are pushed directly into the database, the Data Entity Virus develops, and if the data entities make their way to the client/presentation layer, Work-oholism develops.

- **Work-oholism**  
A system develops Work-oholism by exposing data entities to client or presentation tier objects. Work-oholism forces the client to work with numerous fine-grained entity objects. In a network based application this may increase network traffic.

Since clients often require more than one value from the business layer, a good practice for avoiding Work-oholism is to use a custom Client Object. A custom Client Object encapsulates the business data into a structure that is easily used by the client. This also reduces the number of calls to the business layer, thus reducing overhead.

- **Starving Data Entities**  
The last architectural smell we will consider here is called Starving Data Entities. When a data object is created with only a getter method implemented it is called a Starving Data Entity. If one of these objects is placed into a container object, and the container chooses to use reflection on the object to look for getter and setter methods it will not find any setter methods. This may cause the container to not work properly.

A better approach is to implement read-only access using a façade. The façade can then encapsulate the complexity of the interactions between the business objects participating in the workflow, and ensure the data integrity desired by not implementing setter methods.

Before we go on, let's review what we have covered so far. We considered five high level architecture patterns which can be used to classify any software system: Call-and Return, Data-Centered, Virtual Machine, Data-Flow, and Independent Component. We also discussed two fundamental principles that apply to all architectures: information hiding and separation of concerns. Finally, we looked at a few structures that hint at a flaw in a software architecture. All of this discussion has been abstract. Let us now look at a specific architecture and understand how it works in more depth.

## **Example Architecture**

An architecture should lend itself to being developed as a skeletal system containing the major communication paths throughout the system. At first the system will have very little functionality. This skeletal system will grow incrementally over time to fulfill the requirements of the system.

As a point of reference, consider a system built using the Call-and-Return architecture. This commonly used architecture may be built using object-oriented techniques or simple main-subroutine techniques. For purposes of this discussion, how the layers themselves are constructed is inconsequential as long as the interfaces are descriptive and well defined.

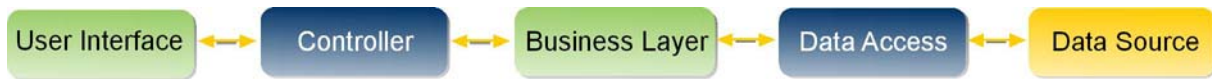


Figure 5: General Purpose Architecture

### User Interface (UI) Layer

The UI Layer should be as thin and lightweight as possible. The primary responsibility of this layer is to display the user interface and interact with the user. Data formats, table population, and lists should be created by other components of the architecture leaving this layer to simply present the required information.

For example, if an application uses a Grid Control in a MS-Windows application, it would be a good practice to build a DataSet object in the Controller Layer, and use the DataSet in the Grid Control. This method isolates the assembly of the data from the presentation of the data.

Another example of keeping the UI as thin as possible is date formatting. By allowing the Controller Layer to format the date, the UI can present the most usable form of the date, but also enable the system to store the date in a form that is most useful for the rest of the system.

By pushing the heavy processing down further into the architecture, the UI Layer stays as thin as possible, making it much easier to test. Unit tests for normal processing are much easier to write and maintain than tests for a UI. Thus, to ensure a stable system, the heavy processing should be done in lower layers of the architecture.

### Controller

The Controller Layer (or mapping layer) processes requests from the UI. The primary purpose of this layer often is to delegate processing to the Business Layer. However, as discussed above, the Controller can also perform the important task of isolating the presentation of the information from the manipulation of data.

Continuing with the DataSet example above, the UI Layer may use a DataSet to represent a table of data. Since DataSets are often difficult to manipulate, the lower levels of the architecture may prefer arrays or maps. In this case, the Controller can unwrap the DataSet from the UI and repackage the data for the rest of the system to use, and vice versa. When data is passed up to the UI, the Controller can wrap the data up into a DataSet.

### Business Layer

The Business Layer performs the most important processing of a system, and as such is considered the workhorse of the system. The Controller sends processing requests to the Business Layer, which contains the complex logic and data manipulation of the system. The Business Layer relies on a simple interface provided by the Data Access Layer to access the data sources used by the system.

### Data Access Layer

The Data Access Layer encapsulates all access to the data source. The Data Access Layer manages the connection with the data source to obtain and store data and is the only layer that actually changes the data. The Data Access Layer completely hides the data source implementation details from its clients by providing a simple interface. This interface does not change when the underlying data source implementation changes, thus enabling the Data Access Layer to use different storage schemes without affecting the business components.

The Data Access Layer may need to provide a Value Object Assembler, which works similarly to the Controller Layer between the UI and the Business Layer. The Value Object Assembler will use data objects retrieved from various data sources to construct a composite Value Object. The Value Object carries the data for the model to the Business Layer in a single method call. Since the model data can be complex, it is recommended that this value object be immutable. That is, the Business Layer should use the Value Object for presentation and processing, but should not make changes to it. The Data Access Layer should be the only layer that actually changes the data.

#### Data Source

The Data Source could be a persistent store like an RDBMS or XML database, a Web Service, or repository. It is easier to write unit tests for Data Access Layer components than for stored procedures. Thus, experience has shown that this layer should be as thin as possible. Techniques such as stored procedures and data caching should be considered when performance drops below required thresholds, but not necessarily used until it can be determined that the system performance is unsatisfactory without using them.

### **Final Note**

Creating a solid architecture for a software system can take a lot of work and discipline. A good architecture should be visible to everyone on the team through proper use of package names/namespaces, and directory structures. When new developers join a team, they should be able to look through the code and plainly see the architectural lines.

Programmers tend to forego the architecture in the presence of extreme deadlines and market pressures. However, by constructing and maintaining a robust architecture, it is much easier to maintain a product over the long run. When components work independently of one another, bugs are not tightly related and generally can be easily isolated.

### **References**

1. Bass, Len, et. al. Software Architecture in Practice. Addison Wesley. 1998.
2. Fowler, Martin. Refactoring. Addison Wesley. 2000.

## Appendix A

This appendix catalogs 11 different software architectures that can be used as generic skeletons for beginning the architectural task. These patterns have been adapted from Software Architecture in Practice, by Len Bass, et. al.<sup>1</sup> Following is the format for this appendix and a brief description of each heading:

<u>Architecture Name</u>	<u>Pattern-Group</u>
Purpose	
Motivation	
Topology	
Application	
Advantages	
Disadvantages	
Questions	

Architecture Name/Pattern-Group - the architecture's name and general pattern identify and categorize each architecture in the catalog.

Purpose - defines what software qualities the named architecture supports

Motivation - provides a brief description of the architecture

Topology – displays a figure of the architecture

Application – provides a further description of the architecture from an implementation point of view and an example of a system using that architecture

The advantages and disadvantages of the architecture follow this description. A few descriptive questions and related architecture patterns complete the catalog entry.

A list of questions (Appendix B) is provided to assist the architect in identifying an architecture or a set of architectures that would be useful in the design of a system. The question section identifies the question numbers from the list that may apply to a specific architecture.

## Data Repository

## Data-Centered

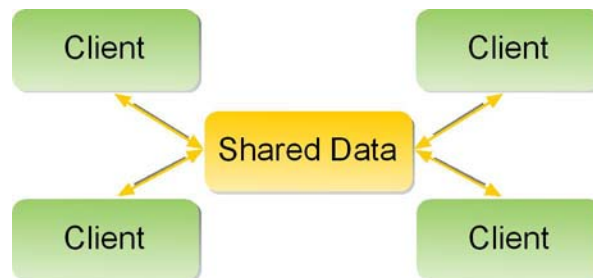
### Purpose

The Data Repository pattern supports a system that is scalable, modifiable, and whose purpose is focused primarily on large amounts of data.

### Motivation

The Data Repository pattern is useful for systems in which data access and update is shared by a number of individual clients. This pattern isolates the data store functionality from the data manipulation functionality. When the clients are built as independently executing processes, this pattern evolves into the client-server pattern.

### Topology



### Application

A client runs on an independent thread of control. The shared data store is a passive repository. Typical implementations of this pattern are database applications.

### Advantages

- Clients are relatively independent of each other.
- The data store is independent of the clients.
- New clients can be easily added.
- Provides an efficient way to share large amounts of data.
- No need to transmit data explicitly from one subsystem to another.

### Disadvantages

- Communication between clients may be slow.
- Subsystems must agree on the database structure.

### Questions

1, 3, 10, 22, 27, 29, 41, 43, 44, 49, 50

### Related Patterns

- Communicating Processes
- Blackboard

## Blackboard

## Data-Centered

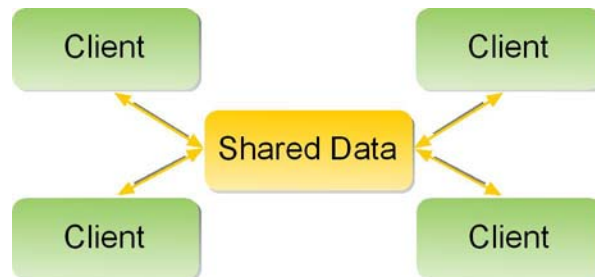
### Purpose

The Blackboard pattern supports a system that is scalable, modifiable, and whose purpose is focused primarily on large amounts of data.

### Motivation

Similar to the Data Repository, the Blackboard pattern is useful for systems in which data access and update is shared by a number of individual clients. The primary difference from the Data Repository pattern is that the Blackboard sends notification to subscribers when data of interest changes. This pattern isolates the data store functionality from the data manipulation functionality. When the clients are built as independently executing processes, this pattern evolves into the client-server pattern.

### Topology



### Application

A client runs on an independent thread of control. The shared data store is an active repository. In an active repository, clients subscribe to receive notification when data is updated. The repository then publishes notification to all subscribers when data of interest is updated.

### Advantages

- Clients are relatively independent of each other.
- The data store is independent of the clients.
- New clients can be easily added.

### Disadvantages

- Communication between clients may be slow.

### Questions

1, 3, 4, 10, 22, 27, 29, 41, 43, 44, 49, 50

### Related Patterns

- Event Systems
- Data Repository

## Batch Sequential

## Data Flow

### Purpose

The Batch Sequential pattern supports a system composed of reusable components, is easily modified, and may increase performance.

### Motivation

The Batch Sequential pattern is characterized by viewing the system as a series of transformations on successive pieces of input data. Data enters the system and flows through the components one at a time until they are assigned to some final destination (standard output or a data store). In the Batch Sequential pattern, each processing step (component) is an independent program. The assumption is that each step runs to completion before the next step starts. Each batch of data is transmitted as a whole between the steps.

### Topology



### Application

Each component maintains its own communication and control. Generally, each component is an individual program. The typical application for the Batch Sequential pattern is classical data processing. Another example may be a set of XSL transforms run in series.

### Advantages

- Quickly process large amounts of data
- Interchangeable components
- Jobs are processed in the background, no user monitoring

### Disadvantages

- Generally expensive hardware
- Interactive applications are difficult to create

### Questions

7, 9, 20, 26, 28, 30, 32, 37, 39, 40, 43, 44, 45, 47, 48, 49, 50

### Related Patterns

Pipe-and-Filter

### Purpose

The Pipe-and-Filter architecture supports component reuse and application modifiability.

### Motivation

The Pipe-and-Filter pattern is characterized by viewing the system as a series of transformations on successive pieces of input data. Data enters the system and flows through the components one at a time until they are assigned to some final destination (generally, standard output). Filters are stream transducers that incrementally transform data, use little contextual information, and retain no state information between instantiations. Pipes are stateless and simply exist to move data between filters.

### Topology



### Application

Both pipes and filters run until no more computations or transmissions are possible. Constraints on the pipe-and-filter pattern indicate the ways in which the pipes and filters can be joined. A pipe has a source end that can only be connected to a filter's output port and a sink end that can only be connected to a filter's input port. Any combination of filters connected by pipes can be packaged and appear to the external world as a filter. A common example of this architecture is the way in which various filters can be piped together in UNIX.

### Advantages

- No complex component interactions to manage
- Easily made parallel or distributed

### Disadvantages

- Interactive applications are difficult to create
- Performance is frequently poor

### Questions

7, 8, 9, 13, 15, 26, 28, 30, 37, 39, 40, 44, 47, 49, 50

### Related Patterns

Batch Sequential

## Interpreter

## Virtual Machine

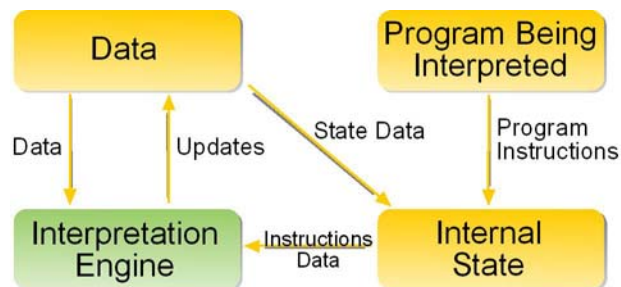
### Purpose

The Interpreter architecture supports system portability.

### Motivation

The Interpreter pattern allows a program to run on a machine without compiling the program into native code. This is typical of virtual machines in general, which simulate some functionality that is not native to the hardware on which it is implemented.

### Topology



### Application

An interpreter allows a program to be built on a machine that simulates the actual production machine. It can also simulate disaster modes that would be too complex, costly, or dangerous to test with a real system. Popular examples of this architecture are the Java Virtual Machine (Java) and the Common Language Runtime (MS .NET). This architecture allows the language to be platform independent.

### Advantages

- Able to interrupt a program at run time
- Able to query a program at run time
- Able to modify a program at run time

### Disadvantages

- Performance may be lower than native code

### Questions

- 5, 11, 16, 18, 28, 36, 38, 42, 44, 49, 50

### Related Patterns

- Rule-Based Systems

## Rule-Based System

## Virtual Machine

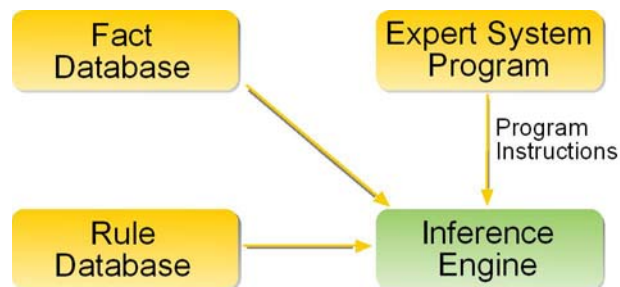
### Purpose

The Rule-Based System architecture supports system portability.

### Motivation

The Rule-Based System pattern is used to simulate a human decision making process. This is typical of virtual machines in general, which simulate systems that would be too complex to build as a real system.

### Topology



### Application

A Rule-Based System allows a program to simulate the human decision making process. After a thorough analysis of the systems requirements, a set of facts will be collected and placed into the Fact Database. The relationships between these facts are stored in the Rule Database. The Inference Engine uses these two databases in combination with the user data to infer a solution to a given input. An example of this architecture is an expert system, which given a set of characteristics, can be used to identify an animal's genus in a biological database.

### Advantages

- Able to interrupt a program at run time
- Able to query a program at run time
- Able to modify a program at run time

### Disadvantages

- Performance may be lower than native code

### Questions

17, 18, 36, 44, 49, 50

### Related Patterns

Interpreter

## Main and Subroutine

## Call and Return

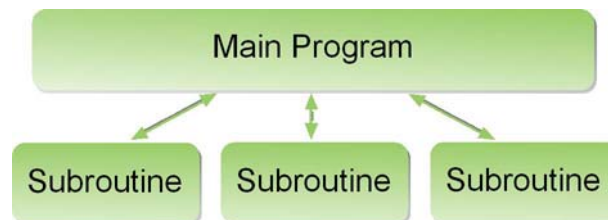
### Purpose

The Main and Subroutine architecture supports system modifiability, scalability, and performance.

### Motivation

The Main and Subroutine is the classic programming pattern. By separating functionality into modules (subroutines), the architecture separates concerns into smaller amounts of complexity, thereby managing the complexity more effectively.

### Topology



### Application

In the Main and Subroutine architecture, there is typically a single thread of control and each component in the hierarchy gets this control from its parent and passes it along to its children. Remote procedure call systems are Main and Subroutine systems that are decomposed into parts that live on computers connected via a network. The actual assignment of parts to processors is deferred until runtime.

### Advantages

- Easily modified
- Grow system functionality by adding more modules
- Simple to analyze control flow

### Disadvantages

- Parallel processing may be difficult
- May be difficult to distribute across machines (traditional)
- Exceptions to normal operation are awkward to handle

### Questions

3, 4, 12, 14, 22, 28, 30, 31, 33, 34, 35, 38, 41, 43, 44, 47, 49, 50

### Related Patterns

- Object Oriented
- Layered

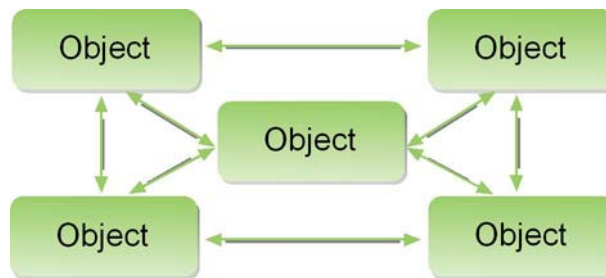
### Purpose

The Object Oriented architecture supports system modifiability, reuse, scalability, and performance.

### Motivation

The Object Oriented pattern emphasizes the bundling of data and the knowledge of how to manipulate and access that data. This bundle is an encapsulation that hides its internal secrets from its environment. Access to the object is allowed only through provided operations (or methods).

### Topology



### Application

The Object Oriented architecture is more conducive to multi-threaded applications, though traditional implementations are single threaded. The encapsulation promotes reuse and modifiability, principally because it promotes separation of concerns. Well organized Java, C#, and SmallTalk programs are examples of systems using this architecture pattern.

### Advantages

- Hierarchical sharing of definitions and code (inheritance)
- Ability to determine the semantics of an operation at runtime (polymorphism)
- Encapsulation
- Loose coupling
- Structure of system is relatively easy to understand

### Disadvantages

- More system overhead used to maintain objects
- Must explicitly reference the name and interface of other objects

### Questions

3, 4, 13, 14, 22, 31, 32, 33, 34, 35, 36, 38, 41, 44, 48, 49, 50

### Related Patterns

- Main and Subroutine
- Layered

## Layered

## Call and Return

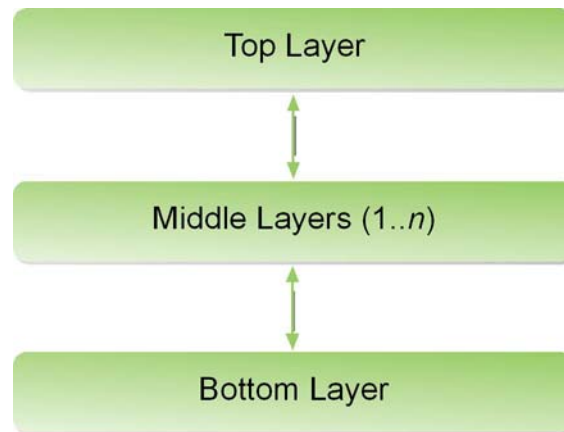
### Purpose

The Layered architecture supports system modifiability and portability.

### Motivation

Layered systems are composed of components that are assigned to layers, which control the inter-component interaction. In the pure version of this pattern, each level communicates only with its immediate neighbors.

### Topology



### Application

The lowest layer in the Layered architecture, provides some core functionality, such as hardware or an operating system kernel. Each successive layer is built on its predecessor, hiding the lower layer and providing some services that the upper layers make use of.

### Advantages

- Easily ported to different platforms
- Easily modified by substituting one layer for a new one

### Disadvantages

- Performance degradation due to the communication between layers
- Structuring systems may be difficult

### Questions

2, 13, 16, 19, 32, 33, 34, 35, 43, 44, 49, 50

### Related Patterns

- Main and Subroutine
- Communicating Processes
- Object Oriented

## Communicating Processes

## Independent Components

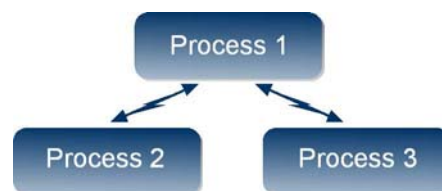
### Purpose

The Communicating Processes architecture supports system modifiability and scalability.

### Motivation

The Communicating Processes pattern consists of a number of independent processes or objects that communicate through messages. They send data to each other but typically do not directly control each other. Generally, the messages are passed through named participants.

### Topology



### Application

The Communicating Processes pattern is the classic multi-processing system. A good example of this pattern is the client-server model. A server exists to serve data to one or more clients, which are typically located across a network. The client originates a call to the server, which works, synchronously or asynchronously, to service the client's request. If the server works synchronously, it returns control to the client at the same time that it returns the requested data. If the server works asynchronously, it returns only data to the client, which maintains its own thread of control.

### Advantages

- Distribution is straightforward
- Easily implemented in parallel

### Disadvantages

- Need to know names of communicating processes/machines
- Communication across a network may be slow

### Questions

3, 8, 10, 13, 14, 27, 28, 30, 31, 32, 43, 44, 46, 49, 50

### Related Patterns

- Data Repository
- Layered
- Event Systems?

## Event Systems

## Independent Components

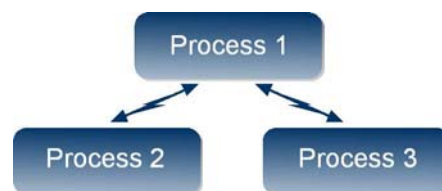
### Purpose

The Event Systems architecture supports system modifiability, scalability, and performance.

### Motivation

The Event Systems pattern consists of a number of independent processes or objects that communicate through messages. They send data to each other but typically do not directly control each other. Generally, the messages are passed among unnamed participants.

### Topology



### Application

The Event Systems pattern embodies control as part of the model. Individual components announce data that they wish to share (publish) with their environment. Other components may register an interest in this class of data (subscribe). If they do so, when the data appears, they are invoked and receive the data. Typical examples of this architecture are graphical user interfaces, which respond to mouse and keyboard events.

### Advantages

- Component implementation does not need to know the name of its subscribers.
- Components can run in parallel
- Control is decoupled from individual components
- Can process real-time events quickly

### Disadvantages

- Implementation is more complex
- Harder to test thoroughly
- Subsystems don't know if or when events will be handled.

### Questions

6, 13, 21, 26, 30, 33, 34, 35, 36, 37, 41, 43, 44, 46, 48, 49, 50

### Related Patterns

- Communicating Processes
- Blackboard

## Appendix B

Appendix B contains true and false questions as well as questions regarding the characteristics of the architecture. Both sets of questions serve only as a guide to assist in the choice of a system's architecture and do not necessarily point directly to a specific architecture. A graphical user interface may have an Event Systems architecture and be implemented using Object Oriented methodologies. It is for the architect to decide if the Event Systems architecture or the Object Oriented architecture provides an overall original skeleton for the system.

True and False questionnaire to guide an architect to an architecture.

1. T F Data is stored in one centralized location.
2. T F The system provides services to other programs.
3. T F Control is transferred from one physical machine to another.
4. T F There exists more than one independent thread of control.
5. T F The system simulates a disaster mode that would be too complex, costly, or dangerous to test.
6. T F The system has a graphical user interface.
7. T F The system operates in a batch mode.
8. T F Control/data interaction has an isomorphic shape.
9. T F The system produces a well defined easily identified output.
10. T F Data is transferred from one physical machine to another.
11. T F The system simulates a platform that has not been built.
12. T F The system is embedded in a physical device.
13. T F There are many system data types whose representation is likely to change.
14. T F The system runs on a multiprocessor platform.
15. T F The initial input data is the only input data.
16. T F The system is likely to be implemented on more than one hardware platform.
17. T F The system simulates a human decision making process.
18. T F The system simulates a process that would be too complex, costly or dangerous to build or test as a real system.
19. T F The system uses services of other programs.
20. T F New data is generated at discrete times.
21. T F The system functions in real-time.
22. T F The program and its data are stored and executed on the same machine.
23. T F The program is interactive.
24. T F Program follows common flow of operation (no exceptions).
25. T F System will use parallel processing.

Characteristics questionnaire to guide an architect to an architecture.

26. Processing Mode:	Batch	Interactive	Real Time		
27. Data Functionality:	Update	Query			
28. Interaction:	Command Driven	Static Screens	Graphical Interface		
29. Data Storage:	Passive	Active			
30. Processing:	Sequential	Parallel			
31. Process Distrib:	Local	Distributed			
32. Control:	Single Threaded	Multi-threaded			
33. Message Passing:	Intermodule	Interprocess	Intermachine		
34. Coupling:	High	Low			
35. Cohesion:	High	Low			
36. Exceptions:	Common	Uncommon			
37. Synchronicity:	Lock Step	Synchronous	Asynchronous		
38. Binding Time:	Compile Time	Run Time			
39. Data Continuity:	Continuous	Sporadic			
40. Data Volume:	High	Low			
41. Data Mode:	Passed	Shared			
42. Program Use:	Simulation	Production			
43. Data Storage:	Centralized	Distributed			
44. Data Direction:	Same	Opposite			
45. Connection Mech:	Shared	Remote Proc.	Data Stream		
46. Important Quality:	Perform	Portability	Reusability	Integrability	Modifiability
47. Data Flow:	One-way	Feedback	Multi-way	Event-Broadcasting	
48. Control Topology:	Linear	Acyclic	Hierarchical	Star	
49. System Compts:	Program	Object	Process	Filter	Manager
50. Important Quality:	Security	Function	Usability	Testability	