Archi-Patterns[1]:

A Process Pattern Language for Defining Architectures

Author: Gerard Meszaros
Object Systems Group
87 Connaught Drive NW,
Calgary, Alberta, Canada T2K 1V9
e-mail: gerard@osgcanada.com
Phone: 1-403-210-2967

Abstract

This pattern language is a first crack at capturing the key concepts and processes of defining an architecture for a computer system. It captures the experiences of the author based on well over ten years of building complex software systems in a variety of domains including fault-tolerant systems, telecommunications, real-time databases and distributed object computing.

1. Introduction:

In the spirit of the patterns movement, the patterns described here should be "old hat" to practicing software architects. They capture the kinds of decisions and thought processes that most architects have (in whole or in part) while they lay out the architecture of a system. As these patterns are drawn from a number of problem domains, many architects will recognize a subset of the patterns. Experience that crosses problem domains shows that all these patterns are useful to some extent in all types of systems.

This pattern language is, by necessity, very high level. The detailed, supporting patterns have not yet, for the most part, been documented. The author hopes that readers will take up the challenge and help "flesh out" this pattern language based on their experiences.

1.1 Notes to PLoP-97 Reviewers

This pattern language is a rather ambitious undertaking and will not likely be "done" for quite a long time. Given the finite time provided for review of each submission to PLoP-97, I would like to focus the writer's workshop discussions on the sections 2.6 Defining Architectural Interfaces, and 2.7 Component Implementation. Reviewers will likely find it useful to read or skim the higher level patterns which precede this section to better understand the context in which the patterns in the section to be discussed operate.

[1] The name of this pattern language is a pun on Architypes.

One area in which I hope to receive many comments is Related Patterns. I am sure that there are many patterns to which these patterns should refer. Some I am aware of but have not had the time to track down; others I may not even know about. Any concrete references you can provide will be greatly appreciated.

I would be more than happy to receive comments on the other patterns of the language in any form including marked-up manuscripts, e-mail, verbally, etc.

1.2 Notation and Organization

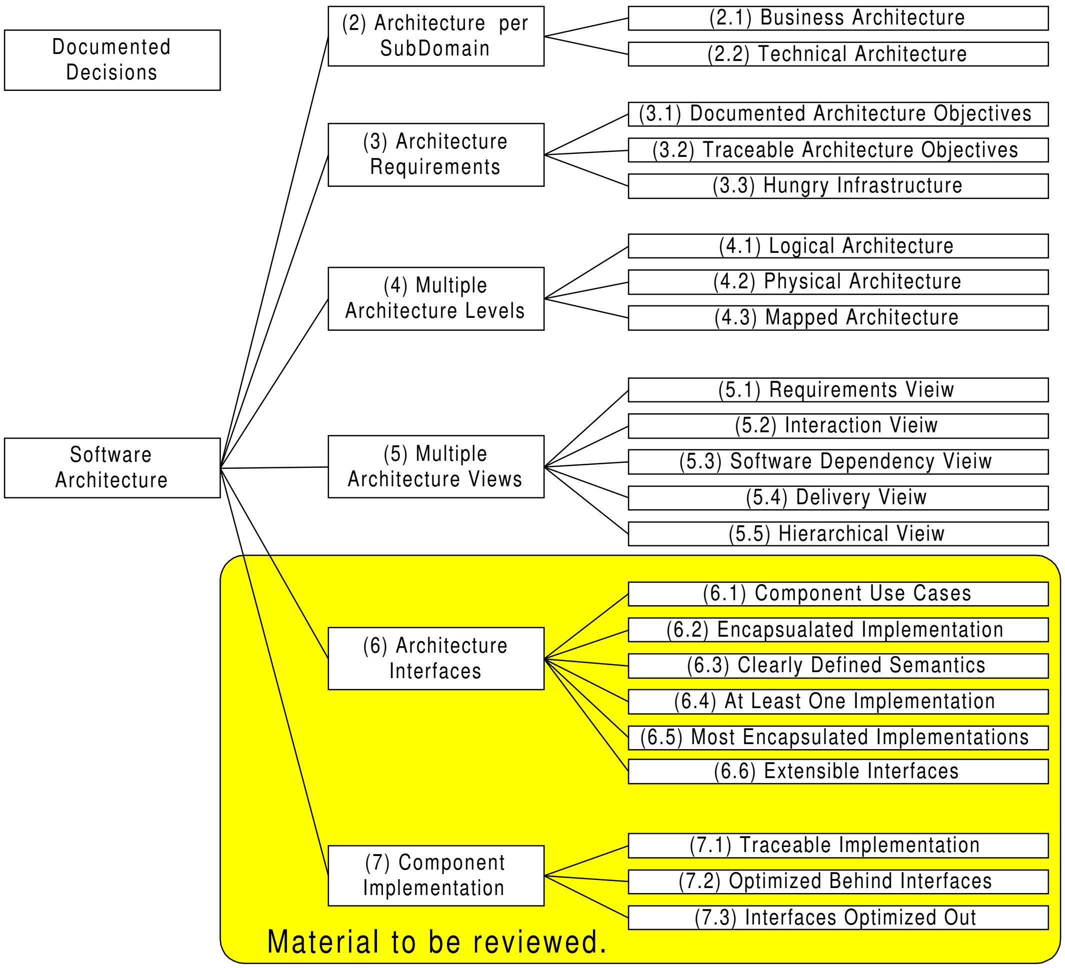
This pattern language uses a number of conventions suggested in Patterns for Writing Patterns [PWP97]. Specifically, it attempts to use EvocativePatternNames to communicate the essence of each pattern. To set them off from the surrounding text, the names are concatenated in the WikiWeb (or Smalltalk) style with all blanks removed.

Even though this is a process oriented pattern language, the patterns are named using NounPhraseNames which describe the end result of applying the pattern. To facilitate digestion of the large amount of information, this pattern language has an architecture which can be used to "zoom into" specific topics in more and more detail. This architecture is visible in the organization of the document sections

1.3 Pattern Language Summary

The following figure illustrates the structure of this pattern language as it is laid out in section 2.0. The numbers in parenthesis correspond to the heading number with the first digit omitted. (E.g. (3.1) can be found in section 2.3.1.)

All the patterns in this language are realizations (or specializations) of the patterns DocumentedDecisions. The pattern SoftwareArchitecture is the root of this pattern language. It suggests that a system should be partitioned into a number of interacting sub-components. ArchitecturePerSubDomain suggests partitioning the system and its architecture based on the different domains it touches. A first order decomposition results in two key sub-domains and their respective architectures, the TechnicalArchitecture and the BusinessArchitecture.



The pattern MultipleArchitectureLevels and its supporting patterns LogicalArchitecture, PhysicalArchitecture and MappedArchitecture deal with techniques for ensuring that the system distribution is handled in a way which facilitates changing of the distribution model.

The pattern MultipleArchitectureViews and its supporting patterns InteractionView, DeliveryView, HierarchyView and RequirementsView deal with ways of describing the architecture to audiences with different interests and needs.

The pattern ArchitectureInterfaces leads to a number of patterns which deal with defining the right interface for architecture-level components to facilitate subsequent “architectural substitution”. ClearlyDefinedSemantics states that all operations of the interfaces should have semantic descriptions in addition to syntactic descriptions. AtLeastOneImplementation states that the architect must be able to conceive of at least one reasonable implementation of every architectural concept introduced. MostImplementationsEncapsulated guides the architect when several possible interfaces must be evaluated and chosen amongst.

The pattern EncapsulatedImplementation leads to a number of patterns which deal with implementing the architecture. Traceable Implementation suggests that the implementation should be forward engineered from the architecture (making as few changes as are necessary) to ensure traceability. OptimizeBehindInterfaces suggests that optimizations should be done out of sight of the client. InterfacesOptimizedOut suggests that performance can be improved by ”flattening” the implementation.

1.4 Fundamental Principles

This section describes the fundamental principles behind this pattern language. Most of the patterns in this language specialize this pattern (or can be viewed as instances of this pattern. You choose your preferred metaphor.)

1.4.1 Documented Decisions

Context:

You are carrying out a task or activity whose results need to be understood by others who many need to continue it in the future.

Problem:

How do you help others understand what you have done and the rationale behind it?

Forces:

- Documenting decisions takes time and effort.
- Some decisions seem so obvious that it seems silly to spend the effort to document them.
- What is obvious to one person may be obtuse to another.
- Time is often in short supply. (We never have time to do it right, but we often have to make time to do it over.)

Solution:

Document all significant decisions. This decisions may pertain to the design of the architecture, assumptions about the deployment environment, details of the functionality requirements.

For our purposes here, a decision is significant if any of the following are true:

- It affects the implementation of the subsequent design.
- It is likely to be questioned at a review.
- It is likely to be reversed at some time in the future if the circumstances surrounding it change.
- It needs to be understood before making any changes in the areas it affects.

If in doubt, document the decision. The kinds of decisions which may need to be documented include:

- Designs chosen or rejected (at least the ones which appeared to be "good" on the surface)
- Requirements included or excluded
- Assumptions (about just about anything)

By documenting the decisions, it is easier for to question them, detect bad assumptions or misunderstandings, or to understand when the decision should be revisited.

Related Patterns:

Most of the patterns in this pattern language are specializations (i.e. examples or instances) of this pattern.

The "Decision Capture and Deferral Pattern Language" [Hopley95] discusses the principles involved in making conscious decisions about what decisions to make and which to defer until later.

2. Software Architecture

Alias: Application Architecture

Context:

You are defining a software system with complex requirements. It will probably be built by a multi-person team .You expect that the system will be modified over its lifetime or will be one member of a family of related systems.

Problem:

How do you organize your system to help manage the complexity?

Forces:

- An application / software system without a well-defined structure is hard to understand or maintain.
- Developers need guidelines for what parts of a software system are allowed to talk to another part. (rules about the parts and their interrelationships or dependencies …)
- A well partitioned application may be able to reuse much of the functionality from other applications that had similar requirements.
- A well partitioned application may be very easy to adapt to changing requirements (such as new kind of user interface).

- Without well defined system components and their interfaces, developers will introduce more interactions between system components than are necessary.
- “Divide and Conquer” is a good approach to managing complexity. This can be achieved through decomposition, abstraction and encapsulation.
- K.I.S.S. (Keep It Simple, Stupid)
- Make it as simple as possible, but no simpler. (Albert Einstein?)
- An arbitrary subdivision is worse than no subdivision.
- Defining an architecture for an application is difficult work that requires a lot of experience (and sometimes creativity) to do well.
- Reuse of an existing architecture may be made easier by frameworks. (& domain specific software architectures)

Solution:

Define an architecture for your application. You may be able to (re)use a pre-defined architecture if your requirements are similar to other applications, or you may need to define your own architecture to address specific needs.

Ensure that the architecture you choose or define addresses all current functional requirements as well as all the Change Cases [Change] of your application. (Or at least ensure that it can be easily extended to address them in the future.)

For the purpose of this pattern language, an architecture can be considered to be:

> A pattern (or blueprint) for organizing a set of components to solve a particular class of problem in a manner which satisfies a set of principles chosen to address the needs of the user and/or business.

The components of which we speak are the software entities which exist within the system. Depending on the particular view of the architecture, the components may be Classes, Packages, Subsystems, Executables and so on.

Related Patterns

To capture the architecture to everyone’s satisfaction, provide MultipleArchitectureViews for each of MultipleArchitectureLevels. Decompose the architecture into components using ComponentPerSubDomain.

Document the requirements of each component of the architecture using Architecture ComponentUseCases. These will later form the basis of the test suite to validate any implementation of the component. (description of the requirements of the architecture vs. description of the requirements of each component this architecture is made of/consists of …)

Architect Controls Product in [Cope95] suggests that an architect should be in charge of an entire family of systems while individual project managers are responsible for delivery each release.

Related Literature

Nearly everybody I know has a different understanding of the term 'architecture'. Here is just a sampling of definitions that appear in the literature:

- Perry; Wolf (92): "Architecture is a set of architectural (or, if you will, design) elements that have a particular form. We distinguish three different classes of architectural elements: *processing elements*; *data elements*; and *connecting elements*."
- Tichy; Habermann; Prechelt (93): "The architecture of a software system represents the *information* with the strongest leverage for software *development* and *evolution*."
- Gacek; Abd-Allah; Clark; Boehm (95): "A software system architecture comprises: A collection of software and system *components*, *connections*, and *constraints*. A collection of system stakeholders' *need statements*. A *rationale* which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements."
- Garlan; Perry (95): "The structure of the *components* of a program/system, their *interrelationships*, and *principles* and *guidelines* governing their design and evolution over time."
- Luckham; Kenney; Augustin; Vera; Bryan; Mann (95): "An architecture consists of a set of *special specifications* (called interfaces) of modules, a set of *connection rules* that define direct communication between the interfaces, and a set of *formal constraints* that define legal and/or illegal patterns of communication."
- Shaw; Garlan (96): "Abstractly, software architecture involves the description of *elements* from which systems are built, *interactions* among those elements, *patterns* that guide their composition, and *constraints* on these patterns. In general, a particular system is defined in terms of a collection of components and interactions among these components. Such a system may in turn be used as a (composite) element in a larger system domain. ... In addition to specifying the structure and topology of the system, the architecture shows the correspondence between the system requirements and elements of the constructed system, thereby providing some *rationale* for the design decisions. At the architectural level, relevant system-level issues typically include properties such as capacity, throughput, consistency, and component compatibility."

2.2 Architecture Per Sub-Domain

Context:

You have decided to define an architecture for your complex system.

Problem:

What is an appropriate way to subdivide your system?

Forces:

- "Divide and Conquer" is a good approach to managing complexity. (decomposition, abstraction & encapsulation … ?)
- A system with requirements from many domains will be necessarily complex.
- Each domain has its own terminology and concepts. (What is the meaning of domain here in this context?)
- A "Jack of all Trades" is most often a "master of none". Subdividing the system allows people to specialize and excel within their domain.
- Arbitrary subdivision may be worse than no subdivision (Documented Design Decisions …)

Solution:

Subdivide your system into a component per identifiable sub-domain with the appropriate interfaces between them. Document the requirements of each component using UseCases. Define an architecture for each component. Encapsulate complexity of each domain specific component behind a simple (domain related) interface that hides the complexity from the rest of the system.

A sub-domain is defined as being a collection of concepts (abstractions) that relate much more closely to each other than they do to concepts in the other sub-domains.

Related Patterns:

OrganizationFollowsArchitecture [Cope95] should be used to help reinforce the architecture by ensuring that organizational boundaries coincide with architectural ones. This prevents the two forces from working at cross-purposes.

Examples:

Examples of sub-domains of a typical business system architecture are BusinessArchitecture and TechnicalArchitecture. Examples of sub-domains of a typical application program might be UserInterface, Persistence Design, Process/Concurrency Control, Security, etc.

2.2.2 Business Architecture

Context

You are building a system which will need to inter-operate with other applications within your enterprise.

Problem

How can you make sure that all the applications in your enterprise can inter-operate properly?

Forces

- There are many entities in the typical business and relationships amongst them that can be quite complex.
- If each application defines its own entities, the applications will not share a common vocabulary.
- Even if they work from the same domain model but each application creates its own implementations of the entities, the applications will be hard pressed to inter-operate. (The so-called "application silo" problem.)
- Every business process (or the software application which implements it) may introduce additional associations between existing objects (what kind of relationships to what kind of objects?). Left unmanaged, this can create overly complex relationships which make persistence design (amongst others)(Do these unmanaged relationships only impact persistence design or is it just an example?) very difficult.
- Someone needs to be responsible for monitoring proposed new business objects (entities) for commonality with already defined ones. Once identified, it may be necessary to reconcile differences in attributes or naming, and to introduce new abstractions to capture the commonality where there are significant legitimate differences.
- Many developers look for differences rather than for similarities. This is then used to justify why they cannot reuse (with or without modification) the existing concepts.

Solution:

Define a Business Architecture based on the structure of the business. The business architecture defines the vocabulary of the business to ensure that all applications mean the same thing when they use a particular noun.

Assign an Architect (or ArchitectureTeam) to own it. (ArtifactOwner) Validate it using the BusinessUseCases which capture the BusinessProcesses. The Business Architecture will describe the BusinessObjects in your domain complete with all the operations (including attributes) they support and the associations they may have with other BusinessObjects

Resulting Context:

The Business Objects may become quite large due the varied requirements of the many applications which use them. You may find it necessary to use Business Object

Extensions[2] to add the additional behavior and attributes which are only required by some Business Processes.

Related Patterns:

ArtifactOwner [Cope95] states that any artifact which is considered valuable must have an owner who is assigned the responsibility for maintaining its integrity (otherwise it is sure to lose its value over time.)

Implementation:

When you get to the point of implementing your system, all your Business Objects should inherit their basic implementation from a single class. This class would be a good place to add the interface which allows all Business Objects to support Transactions and Persistence. The implementation of this interface would use the services of the Technical Infrastructure.

If you need to inter-operate with systems of other companies, you may need to jointly define an IndustryArchitecture which consists of the definitions of common concepts in the industry. The OMG has established a number of such groups for the purpose of standardizing on a set of common Business Objects across a particular industry.

2.2.3 Technical Architecture

Alias: Technical Infrastructure

Context:

You are developing a system with significant business and technical challenges.

Problem:

How do you make it possible for the application developer to focus on the business problem?

Forces:

- "Jack of all trades; master of none...": A person who studies computer science enough to understand how to implement transactional persistence and concurrency control likely doesn't have the time to become an expert in user interface design or business modeling.

[2] BusinessObjectExtensions: This is similar to ENVY's class extensions in that it involves providing optional attributes and behavior that can the associated with a Business Object at run-time and therefore it can be selectively included in only those systems using that Business Object that require it.

- Clearly separating the Computer Science domain from the Business Domain is hard, especially for computer scientists and engineers who like to get their hands dirty with the technology.

Solution:

As a key component of the system architecture, define a “technical infrastructure” component which encapsulates the complexities of the computing domain from the average application developer. Strive especially hard to encapsulate the PhysicalArchitecture (which is prone to frequent changes), the SecurityArchitecture (because if you understand it, we’ll have to shoot you) and the ConcurrencyArchitecture (because if you understand it, you will want to shoot yourself!) from the application.

Capture the requirements of the infrastructure as Infrastructure ComponentUseCases. The Infrastructure ComponentUseCases may be categorized into groups of related use cases collectively known as Infrastructure Services.

Define an Architecture for the TechnicalInfrastructure. This Technical Architecture can be used to integrate the various components of the Technical Infrastructure, including ensuring that various purchased components can be integrated into a single cohesive infrastructure.

Related Patterns:

The architecture of the Technical Infrastructure is also known as the Technical Architecture. Products based on the Common Object Request Broker Architecture (CORBA) are examples of a TI (may be captured my an examples section …).

2.3 Architecture Requirements

2.3.1 Documented Architecture Objectives

Synopsis: The objectives of the architecture should be documented and made available to all developers so they can ensure they don’t compromise any of them.

2.3.2 Traceable Architecture Objectives

Synopsis: The objectives of the architecture should be clearly traceable to the objectives of the business it supports. The objectives should not appear out of “thin air”. The set of objectives should be validated for completeness against the objectives of the business by asking “What else (services, flexibility) could the architecture provide to allow the business to be more efficient and competitive?”

2.3.3 Hungry Infrastructure

Context:

You are defining an architecture to be used for building a number of applications. You plan to implement part of this architecture as an Application Framework or Technical Infrastructure.

Problem:

How do you maximize the value provided by the architecture by providing the client application with a clean, usable and highly functional interface?

Forces:

- Every bit of functionality moved into the infrastructure reduces the amount of work required to build an application.
- Every bit of functionality added to the infrastructure reduces the complexity of the infrastructure and may make it harder to understand how to use it.

Solution:

Put yourself in the shoes of a developer of application to be built using the architecture. Capture all the things the architecture should do to make your application easy to construct. Ask yourself “How could I make it even easier to build this application?” Examine every step in a typical application programs asking whether this is something the application developer should have to deal with. If the work is “computer science” related, the odds are that it can and should be handled by the infrastructure. As a rule, only “application policy” and “business process” related decisions should be left in the hands of the application developer.

Move as much of the complexity as possible into the infrastructure. Capture these requirements as Component Use Cases of the infrastructure which describe what the application wants to achieve and scenarios which describe the circumstances in which the Use Cases are “invoked”.

2.4 Multiple Architecture Levels

Context:

You are building a system which will be deployed in a multi-computing environment (whether distributed or just having multiple processors) which is bound to change over time. There are many decisions to be made to get from a set of functional requirements (e.g. Use Cases), non-functional requirements (e.g. performance, constraints) and future requirements (e.g. Change Cases) to a working distributed system.

Problems:

How do you ensure that the decisions are made in an orderly fashion that results in a high quality system that is capable of being extended to address the future requirements at a later date?

Forces:

- Building a complex business system is hard; building a distributed one is even harder.
- First make it work, then make it efficient. In a performance oriented development culture, developers are prone to optimizing the design before they have even designed it.
- Deferring distribution may be scary if you are not confident in your ability to tune any performance issues quickly.
- An architecture based primarily on the physical environment (which changes very rapidly) is likely to become obsolete quickly. In the future, developers will need to explain that things in the architecture exist for "Hysterical reasons."

Solution:

Define a set of architecture models which address specific aspects of the system at defined levels of abstraction. The number and nature of these models may vary based on the type of system being built, however, for a typical distributed system, the bare minimum set of architecture models should include:

- The Logical Architecture
- The Physical Architecture
- The Mapped Architecture

Define the "logical" or "conceptual" architecture for the system without respect for the planned distribution boundaries. Let's call this the LogicalArchitecture. Separately, define the physical environment in which the system will be deployed. This is called the PhysicalArchitecture. Finally, define a mapping of the components of the LogicalArchitecture into the components of the PhysicalArchitecture. Do this as late as possible to ensure that the LogicalArchitecture is not corrupted by the PhysicalArchitecture.

Define all the interactions between components of the Logical Architecture. Define the mapping of the LogicalArchitecture into the PhysicalArchitecture only after the LogicalArchitecture has been validated as being complete and sufficient.

Optimize behind interfaces. (=> Related Pattern?)

Related Patterns:

The "Decision Capture and Deferral Pattern Language" [Hopley95] discusses the principles involved in making conscious decisions about which decisions to make and which to defer until later.

The patterns LogicalArchitecture, PhysicalArchitecture and MappedArchitecture elaborate on those principles as they apply to the process of distributed system design.

Rationale:

Distributing functionality onto processors before one has verified that the functionality actually works is a common mistake and often leads to systems that meet performance requirements but remain "buggy" throughout their lifetimes. With the levels of functionality and complexity seen in modern systems, the likelihood of getting it to work correctly without the extra step is fast approaching nil.

Taking a more disciplined, multi-step approach may feel like it is taking longer, but the result is typically more correct functionality and much less time spent chasing down hard-to-find problem.

Known Uses:

There are now more and more tools which support such "late distribution". These tools allow the Logical Architecture to be implemented in a programming language without concern for the distribution of the system. The resulting implementation is then "mapped" using the tools into the PhysicalArchitecture in a way which preserves the semantics of the implementation across multiple processors. Specific examples include IBM's VisualAge Distributed feature, CORBA based products such as ParcPlace-Digitalk's Distributed Smalltalk, and Forte. Some of these tools generate distributed applications in a "compile" step, while others introduce distribution at run-time.

2.4.2 Logical Architecture

Context:

You are building a system which will be deployed in a multi-computing environment which is bound to change over time. Note: in a non-distributed system, the LogicalArchitecture and the MappedArchitecture are one and the same.

Problems:

How do you ensure that the physical deployment model (the PhysicalArchitecture) does not unduly warp the architecture of the system.

Forces:

- Building a complex business system is hard; building a distributed one is even harder.
- Too much of a focus on efficiency before functionality can lead to systems with very efficient bugs.
- Deferring distribution may be scary if you are not confident in your ability to tune any performance issues quickly.
- Without a disciplined architecture-centric development process, it may take a physical (e.g. processor) boundary to force a component interface to be defined with any rigor.

- An architecture based on the physical environment (which changes very rapidly) is likely to become obsolete quickly. In the future, developers will need to explain that things in the architecture exist for "Hysterical reasons."

Solution:

Define the logical architecture without respect for the PhysicalArchitecture. Define components that make sense even if the system were not distributed at all. Define the interactions between the components of the Logical Architecture.

Related Patterns:

Define a mapping of the LogicalArchitecture into the PhysicalArchitecture only after the LogicalArchitecture has been validated as being complete and sufficient.

If necessary, optimize the implementation behind interfaces.

2.4.3 Physical Architecture

Context:

You are building a system which will need to execute in a multi-computing environment consisting of two or more processors or computers.

Problems:

How do you capture and describe on the physical environment in which the system will be deployed?

Forces:

- Ignoring the physical deployment of the system can result in systems with very poor performance.
- Systems which define only a physical model become obsolete as soon as the physical deployment changes.
- The physical computing environment contains many objects with often complex inter-connections. The details of the connections may often be changed and should have no impact on the applications which run in the environment.

Solution:

Define the physical architecture consisting of the important physical components and their interconnections. Model the interconnection of physical components in only as much detail as is necessary for the understanding and solving the current problem. Only those components which will have responsibility for application Use Cases should be included.

Later, you will decide how the logical architecture will be mapped into this physical architecture.

Example:

When dealing with a computer network from the perspective of an application, the Physical Architecture would consist of Servers (of various types such as Database Servers, Application Servers, Communication Servers, etc.) and the LAN which connects them. More detailed components such as disk drives, network cards, etc. are not of interest to the typical application in this domain.

When dealing with telephone calls in a telephone network, the Physical Architecture components of interest are Switching Centers and the connectors are Trunks (Payload carrying links) and Signaling Links. However, when dealing with network management of the same network, the Physical Architecture components may include Digital Cross Connects connected by Digital Carriers (such as T1s, DS512s and OC1s.) The increased level of detail is necessary for this domain but would get in the way when dealing with call processing.

Known Uses:

UML also refers to this as the Physical Architecture. [UML97]

2.4.4 Mapped Architecture

Alias: Deployment Architecture

Context:

You are building a system which will be deployed in a multi-computing environment. You have defined a Logical Architecture and have designed or been mandated a Physical Architecture.

Problems:

You need a representation of the system as it will be actually deployed in the network. (deployment vs. runtime/distribution…)

Forces:

- Making the distribution decision too early can result in your design being “married” to the Physical Architecture.
- Sooner or later, you will have to make the decisions regarding the distribution of your system so that you can implement it. Leaving it too late can leave significant risk factors unresolved until too late.
- The physical architecture and the deployment of the software into it are some of the highest “churn” areas of a software application.

Solution:

For each object in the LogicalArchitecture, decide in which network elements it will reside (using the patterns ObjectsLiveAtHome [OopsArch96] and HomelessObjectsLiveWithTheirBestFriends [OopsArch96]. Hide all distribution boundaries behind object interfaces. (Refer to the pattern Encapsulated Implementation)

Where an object needs to be easily accessible from several (or many) object spaces, replicate the object using HalfObjectPlusProtocol [HOPP95]. Where it is acceptable to use remote messaging to access an object, use RemoteProxy [Siemans96]. A hybrid approach is Caching Proxy[Siemans96]. These are all examples of OptimizedBehindInterfaces.

Related Patterns

UML refers to the deployed system architecture as the Deployment View. [UML97]

2.5 Multiple Architectural Views

Context:

You are capturing the architecture of a system or one of its components, for the purpose of communicating the architecture to others.

Problem:

How do you present the architecture of the system to audiences with potentially diverse needs?

How do you present different aspects of the system's architecture in a way appropriate to the audiences?

Forces:

- There are many potential audiences with different needs of the system's architecture.
- There are different aspects of a system's architecture that need to be presented in a different way/form …Providing a view for each audience could be very expensive indeed.
- Multiple views of the system loose their value if they do not describe the same architecture; that is, if they get out of sync.
- Keeping many views consistent is difficult
- The views must somehow be reconciled, at least when the system software is defined.

Solution:

Pick a small set of views of the system architecture which satisfy as many of the audiences as possible. Capture these views in a manner which facilitates keeping them consistent. This could be by using an Architecture Case Tool, or by building an ExecutableArchitectureModel[3].

The minimum recommended set of views includes:

- The RequirementsView, which describes the relationships between various pieces of the system requirements.
- The InteractionView, which describes the interactions between distinct components in a running instance of the system.
- The DeliveryView, which describes the relationships between the various fragments of code (code level or architecture level ???).
- The HierarchyView, which describes the commonality of behavior amongst various system components.

Together, this set of views should present a complete picture of the system without creating so many views as to be unmanageable. Each of these views should contain objects at a level of granularity appropriate to the level of model. As an example, at the Logical Architecture level, only logical architecture concepts should be included; objects related to implementing a distributed system would be omitted.

Of the four views, the Delivery View is the one used most in traditional software development processes. The Interaction View and Requirements View have become increasingly important as systems become more complex because such systems are hard to understand purely at the level of source code.. The HierarchyView is most common in Object Oriented programming languages which support inheritance, but can be quite useful even where the programming language does not support inheritance; it is the relationship of interfaces which should be modeled, not the relationships between programming classes.

2.5.2 Requirements View

Context:

You are gathering the requirements for a complex system. The requirements seem to be endless.

Problem:

How do you organize large amounts of requirements to make it possible to deal with them?

[3] ExecutableArchitectureModel: A software model (or implementation) of the architecture which exhibits the same characteristics as the full-scale architecture.

Forces:

- Organizing the requirements is hard work and may take a significant amount of time and effort.
- Dealing with the requirements without organizing them may not even be possible.
- The earlier you identify reuse opportunities, the more development effort you can save. Reuse of requirements can maximize your savings.

Indications:

You worry that it will be hard to comprehend all the requirements let alone make sure you deal with them all.

Solution:

Organize the requirements into a Requirements View of the architecture. Abstract out the details of the user interface to create simpler, less implementation oriented use Cases (ref: Essential Use Cases). Gather ConditionBasedScenarios [Cockburn96] with a common objective under a single GoalOrientedUseCase. [Cockburn96]

2.5.3 Interaction View

Context:

You are building a large complex system which will have many components which will have to interact with one another at run-time.

Problem:

How do you explain the dynamic aspects of the system behavior?

Forces:

- It may be hard to understand how a particular piece of software interacts with the rest of the system without having a higher level view.
- The DeliveryView does not describe the dynamic behavior of the system.

Solution:

Define an "interaction view" of your system architecture. This TypeModel describes the key high-level components that exist in the system at runtime and how they interact to fulfill the system's specification. An InteractionDiagram is a good way to describe the interactions between the components. At the highest level, the interactions would be between application components and components within the BusinessArchitecture and the TechnicalArchitecture.

Known Uses:

Many software development methods include notations for describing the interactions between components. Examples include:

- Booch: Object Interaction Diagrams and Sequence Diagrams
- UML: Collaboration Diagrams and Sequence Diagrams
- CCITT: Message Sequence Charts (Z.100)
- Jacobson/OOSE: …

2.5.4 Software Dependency View

Context:

You are building a large complex system which will have many bits of code much of which depends on one another.

Problem:

How do you manage the software (code) in your system such that you can reliably configure a system from it?

Forces:

Tracking the dependencies of every bit of software in your system may be too large a problem to keep up to date.

Building systems without tracking software dependencies will often result in failed system builds or random, unexplained errors in the resulting system.

Solution:

Organize the software of your system into collections of code which can be managed as a group. Bits of software with a common purpose and common dependencies should be packaged together. If necessary, define several levels of recursive packaging; only stop the recursion when the number of packages is manageable. Track the dependencies of each package using a Configuration Management Tool.

Examples:

Examples of packaging include Classes, Modules, Subsystems, Software Packages.

Known Uses:

Examples of systems which manage the delivery view of the system include Envy/Developer.

2.5.5 Delivery View

Context:

You are building a large complex system which will have many bits of code much of which depends on one another. You plan to do many system builds, either to support incremental development or to build different members of a product family.

Problem:

How do you ensure that there are no circular dependencies which could prevent assembly of the system? (E.g. compile or make/build errors.) How do you capture the rules for what software has to be included in a system configured from it?

Forces:

- There may be more than one delivered configuration of a system.
- It is hard to manage all the dependencies amongst the software components.

Without understanding the dependencies amongst the software components, building a system from them may not be a repeatable process.

Solution:

Capture the software dependencies within your system in a delivery view. Ideally, this is done proactively (i.e. before construction) but if omitted on the first pass, it will have to be done retroactively (i.e. retrofitted over top of an existing implementation.)

If you have already defined the implementation level software, capture and name the “clumps” or “packages” of software which should be treated as a unit for delivery purposes. Note all the dependencies of the software that a package contains and translate them into dependencies on the containing package(s). You now have a higher level representation of the software dependencies that can be more easily managed.

If you have yet to construct the implementation, you should define the packages proactively. Decide what parts of the software functionality will always be present and what parts will be optional. Which optional parts must be present together? Which ones are mutually exclusive? Which ones should depend on one another? Define packages to contain the functionalities you have thus partitioned and capture the dependencies between them.

Depending on the nature of your development environments, these software packages may or may not need to be translated into “executable packages”.

Examples:

UML has Packages in the Logical View as well as Components in a Component View. A Component describes the Executable Components (e.g. DLLs) while a Logical View

Package describes the source code used to generate it. The Deployment view shows how the components are mapped into the processors in the system (the PhysicalArchitecture). A Component is typically the result of having compiled and linked (often via a "make file") a set of classes, modules or packages.

OTI/IBM's Envy/Developer (and IBM's VisualAge Smalltalk and Java products which incorporate it) have a concept called "Application" (in Smalltalk versions) or "Package" (Java versions) which contain Classes (and Class Extensions in Smalltalk). These are used to provide the unit of software ownership and configuration. These packages are then assembled into "configurations".

2.5.6 Hierarchical View

Context:

You are building a complex system possibly using software technology capable of supporting polymorphism (and possibly inheritance.) You are documenting a SoftwareArchitecture.

Problem:

How do you identify and communicate reuse opportunities afforded by the KindOf relationships between entities in your architecture?

Forces:

- Knowing that two things behave in a similar way helps identify reuse opportunities through identification of common interfaces. This can lead to designing the KindOf hierarchies in the system.
- Discussing KindOf relationships too early can lead to premature focus on implementation reuse because developers will want to subclass a concept because it contains some useful implementation rather than to subtype something which has a useful interface definition.

Solution:

Define a HierarchyView of the system which describes SubType relationships between Types. Limit the discussions to true KindOf relationships as defined by the domain experts. Do not look for code reuse opportunities at this time as it will remove the necessary focus on understanding and analyzing the requirements. Note: Feel free to subtype several existing Types when defining a new Type. This does not need to imply that you will be using multiple inheritance; it just says that any implementation of your new (sub)Type must satisfy the interface definitions imposed by all the subtyped Types.

When it comes time to implement the system, this view may help identify opportunities for code which can be reused using your programming language's inheritance feature.

2.6 Defining Architectural Interfaces

This section of the pattern language deals with the definition of the interfaces that define the architecture.

2.6.1 Component Use Cases

Context:

You are defining the interfaces of one or more components of an architecture. The interfaces will provide access to the functionality provided by each component and define how the components will interact to provide the system functionality.

Problem:

How do you ensure that the architecture supports evolution of the system while providing the client with a clean, usable interface?

Forces:

- A good architecture supports easy evolution of the implementation. That is, it supports all the known change cases in addition to the required functionality.
- Coming up with an interface which support many possible implementations requires good abstraction skills and possibly several iterations of the architecture.
- An API which is complex is very hard to learn and use.
- Designing a minimalist API for a given set of functionality is difficult and takes time.
- Time is of the essence on most projects.

Solution:

Before starting to define the interface of the components (of the architecture), enumerate the requirements of the component(s) as a set of scenarios. From these scenarios, condense out the set of application objectives (or goals) they satisfy. Collect all the scenarios which serve to achieve a common goal or objective (from the applications' perspective) as a single use case with scenarios which describe the circumstances under which the Use Cases are invoked.

Assign each Use Case to a component of the architecture using standard OO design practices. Use the use cases to determine the necessary operations on the architecture component. Use the scenarios of the use case to help you ensure that the argument list of the operation is sufficient to handle all current and future requirements.

Resulting Context:

You should end up with a much smaller set of application-visible functionality to implement. But each bit of functionality will have a number of scenarios which must be implemented. This complexity can thus be hidden behind an interface who's breadth is based on the small set of use cases.

2.6.2 Encapsulated Implementation

Alias: Architecture Component Interface

Context:

You are defining the interfaces of one or more components of an architecture. You have defined a set of Component Use Cases for the component in question (e.g. the Technical Infrastructure or Application Framework.) The interfaces will provide access to the functionality provided by each component and define how the components will interact to provide the system functionality.

Problem:

How do you ensure that the system can be evolved while providing the client with a clean, usable interface?

Forces:

- A good architecture supports easy evolution of the implementation. That is, it supports all the known change cases in addition to the required functionality.
- Coming up with an interface which support many possible implementations requires good abstraction skills and possibly several iterations of the architecture.
- An API which is complex is very hard to learn and use.
- Designing a minimalist API to a given set of functionality is difficult and takes time.
- Time is of the essence on most projects

Solution:

Define an interface for each component of the architecture. Ensure that the interface hides the implementation and that it either supports or at least does not inhibit any of the ArchitectureObjectives or ChangeCases. Define the interfaces of each component in terms of operations with ClearlyDefinedSemantics. Define an acceptance test suite for the operations of the component based on the Architecture Component Use Cases they support.

Ensure that the implementation (of each operation) of each component complies with the ClearlyDefinedSemantics. When you have a choice of possible interfaces, chose the one which results in the MostImplementationsEncapsulated. Where the implementation needs to be optimized, ensure you OptimizeBehindInterfaces.

Known Uses:

The Catalysis methodology [Catalysis96] describes the use of Type Models to define the behavior of an interface.

In Analysis Patterns [Fowler96], Martin Fowler writes about the use of Type Models as a way of specifying the behavior of the implementation.

Implementation Notes:

Some Object Oriented Programming Languages (OOPLs), like Java, differentiate between the concept of Type (Interface or Protocol) and Class (or implementation.) In OOPLs which do not, a Type can be represented as a "fully abstract" class, while a concrete class is an implementation. Multiple Inheritance (when supported) can be used to link an implementation Class with all the Abstract Classes which define the Types (Interfaces or Protocols) it implements.

2.6.3 Clearly Defined Semantics

a.k.a. TypeModels Define Semantics

Context:

You are encapsulating one or more implementations behind an interface. You have defined the set of operations supported by the interface.

Problem:

How do you ensure that the interface is understandable and usable without examining one or more of the implementations?

Forces:

- An operation is only useful if you know how and when to use it.
- Descriptive operation names are important to facilitate understanding.
- Descriptive operation names are often not enough to convey the semantics.
- Describing semantics is hard without talking about implementation.
- A "reference implementation" is often taken as gospel by the implementers who dare not change it.

Solution:

Describe the interface using a TypeModel. Capture the conditions that are always true as Invariants. For each operation, use Pre-conditions and Post-conditions to describe the concepts that the operation manipulates and its effect on them. Stress that the concepts do not have to be implemented as described but merely have to behave as described. A "reference" test suite for the interface is a good way to capture the semantics. It can define the preconditions of each use case in terms of objects that exist (at least in concept) and which can be queried. The post-condition of each use case describes the resulting set of objects and their query-able states.

Known Uses:

The concept of Preconditions, Postconditions and Invariants are discussed in detail in [Meyer9?]. Though they are not cast in pattern form, they are generally accepted as a way of adding semantic information to an interface description. [Catalysis97] extends this work by proposing a grammar for specifying the semantics of the operations.

Type Models are defined in a number of recent works including:

- Catalysis [Catalysis96]
- Analysis Patterns [Fowler96]

A particularly amusing exposition on the subject is "Data or Behavior Driven" by Desmond DeSouza. [DeSouza96?]. It states that even for a relatively well understood concept such as a "Stack", the semantics are quite complicated. Operations such as Push can only be properly expressed based on concepts such as an ordered collection of stack entries, a stack entry counter, the "top of stack", etc. Some or all of the concepts may appear in a specific implementation, but it is only necessary that they *appear* to be implemented.

2.6.4 At Least One Implementation

Context:

You are encapsulating one or more implementations behind an interface. You have defined the set of operations supported by the interface.

Problem:

How do you ensure that the interface is implementable?

Forces:

- A well defined interface is lean, clean and elegant. But, an elegant interface is useless if it is not implementable. An architect who never thinks about implementation may find themselves defining interfaces which cannot be realized practically.
- An architect needs to maintain a complete system (or subsystem) view to provide the technical leadership which is their responsibility. Spending too much time on implementation issues can quickly use up all their time leaving none for architecting.

Solution:

The architect of the interface should ensure that they can come up with at least one implementation of the interface. It is enough to come up with one; there is no need to decide which of several possible implementations is best. (Leave that to the developer(s) of the implementation!)

Related Patterns:

ArchitectAlsoImplements [Cope95] ensures that the architect has current enough implementation skills so that (s)he can reliably envision at least one possible implementation for any architectural element.

2.6.5 Most Encapsulated Implementations

Context:

You have defined the components of your architecture. Now you are trying to define the interface on one of the components. You can think of several different interfaces for this component, each of which will satisfy the "use cases" of the component, but would like to choose a single one.

Problem:

How do you decide which of several possible interfaces is the best one?

Forces:

- There is no absolute "bests" in software design; "best" is often a matter of personal opinion.
- You may have as many proposed interface designs as you have people involved in creating the design.
- Choosing amongst designs without clear and precise guidelines is prone to turning into a personal or "religious" battle.

Solution:

Imagine many different implementations of the functionality to be provided behind the interface. Evaluate each candidate interface in terms of how many of the possible implementations could be encapsulated behind it. Pick the interface that could encapsulate the most, if not all, of the implementations.

If several candidates could hide different subsets of the implementations but none could hide all, it is likely that none of the candidates is abstract enough. In this case, you should look for a new interface which combines the best features of the interface candidates you already have. This is sort of like using "genetic algorithms" to create new interfaces by "mating" existing ones and the evaluating the rather random results. But this is how nature innovates, and it works!

2.6.6 Extensible Interfaces

Context

You are designing the signatures of the operations that make up the interface of a component.

Problem:

How do you ensure that the signatures do no lock you into a particular implementation and are easily extended to handle future requirements?

Forces:

- Signatures which have long lists of parameters are hard to remember.
- they are also less resilient to change.
- Using “whole objects” as parameters makes the signature more easily understood and extended, but requires the caller to build the objects before invoking the interface. In some systems, this can create considerable extra overhead.

Solution:

Design the operation interfaces such that the interface does not preclude passing new, unanticipated variants of parameters.

Techniques for doing this include passing “whole objects” rather than lists of parameters. (This makes it possible to pass additional information by extending an object and its constructors rather than visiting all “senders” of the operation.)

Optimize the creation of the “whole objects” by providing suitable factory methods or Prototype objects.

Related Patterns:

Whole Value from [Checks95] is an example of a pattern which makes interfaces more extensible by amalgamating several parameters into a single, potentially polymorphic, object.

Need to find the example from “Found Objects” by James Noble in PLOPD97 or the proceedings of PLoP-96.

2.7 Component Implementation

This sections discusses techniques to be used when realizing an Encapsulated Implementation.

2.7.1 Traceable Implementation

Context:

You are implementing a component. interface. You have defined the set of operations supported by the interface.

Problem:

How do you ensure that the implementation is easily understood and maintained?

Forces:

- A simple implementation is more easily understood than a complex one.
- A simple implementation may not meet performance requirements (such as memory consumption, processing cost, latency, etc.)
- Many developers feel a need to demonstrate their cleverness through “tricky” programming.
- Excessively clever programming may satisfy the programmer but is sure to drive anyone trying to understand the software mad.
- It is easier to come up with a design that solves a problem in a straightforward way than it is to try and explain a convoluted design in a way that anyone can understand.

Solution:

Build “traceability” into the implementation rather than trying to retrofit it onto an implementation that has poor “conceptual continuity” with the architecture it implements.

Start with the simplest, most obvious possible implementation. Assess its strengths and weaknesses and if they are acceptable, use it unchanged. If the weaknesses are unacceptable, address with as few changes as is necessary to avoid making the implementation difficult to understand.

Examples:

This is similar to the concept of “Structure Preserving Transformations” of which Christopher Alexander spoke in his keynote address at OOPSLA96. The structure of which we speak is the semantic behavior of the component as defined by the TypeModel. Our implementation may be changed to have different time/space/complexity tradeoffs as long as the functionality semantics are preserved.

2.7.2 Optimized Behind Interfaces

Context:

You are encapsulating one or more implementations behind an interface. You have defined the set of operations supported by the interface.

Problem:

How do you improve the performance of the system without sacrificing the "ilities" you worked so hard to build into the architecture?

Forces:

- A highly flexible implementation will handle a lot of different situations.
- When you don't need the flexibility, it would be good to not have to pay the run-time price (whether memory usage or processing time.)
- Different configurations of a system may have different optimization needs depending on whether the system is processor bound or memory bound.

Solution:

The beauty of a well defined interface is that you can substitute different implementations behind it. Take advantage of this by building an alternate implementation which is appropriately optimized. The optimized implementation can be bound to the interface at system configuration time if you know that the sacrificed flexibility will not be required in this configuration of the system. Otherwise, select the appropriate implementation at run-time, when you have enough information to know whether the flexibility will be required. You may be able to create several implementation optimized for different circumstances as alternatives to the "all-singing, all-dancing" flexible implementation. As long as you can identify the criteria for using each alternative at run-time, you can safely substitute them.

Examples:

An OrderedCollection is a container which keeps it elements in the same order as they were inserted. Examples of OrderedCollections include Stacks and FIFO Queues. The interface of OrderedCollection specifies the operations that it must support, and the semantics of those operations with respect to the contained elements. Any implementation which satisfies both the syntax and semantics of the operations can be used to satisfy the functional requirements, but different implementations may have different non-functional requirements such as response time (time required to insert or delete an element), capacity (number of elements), reliability (e.g. persistence), etc.

The choice of implementation is based on these non-functional requirements of the application. E.g. You can use an array-based implementation of OrderedCollection when you know the maximum size of the collection when it is created or the cost of extending the collection by copying to a larger array is acceptable in exchange for much faster traversal time. (This is because you do not expect extension to be done frequently.)

2.7.3 Interfaces Optimized Out

Context:

You have defined your system and implemented it. Upon measuring the real-time performance of the system, you find that it is too slow (that is, it uses more processing time than is allowed by the specification.)

Problem:

How do you improve the performance of the system so that it meets the spec?

Forces:

- A highly flexible implementation will handle a lot of different situations, but the interfaces required to support the flexibility add processing cost.
- When you don't need the flexibility, it would be good to not have to pay the run-time price (whether memory usage or processing time.)
- Removing lower level interfaces could reduce future reuse potential if the low level interface is not preserved.
- In-lining the implementation of an interface you use increases your coupling to that implementation and increases the amount of code which needs to be maintained (because now it exists behind the original interface and in the optimized implementation into which it was "in-lined".)

Solution:

One way to improve performance of a system is to remove unnecessary interfaces that add run-time overhead. This removes the cost of delegating through a number of layers of software, each of which adds value by providing a higher level interface. This can be achieved by combining two or more layers of software (which built upon each other) as long as yours was the only client of the removed interface. If that interface had other clients, you will either have to in-line the functionality into all other clients, or preserve the original interface for their use.

It is preferable to have tools do the in-lining so that you do not have to manually manage the multiple copies of the code which was in-lined.

Ensure you do this behind another interface so that your optimizations do not increase the coupling of applications to the implementation of the components they use. If at all possible, do not remove the "front-line" interface that hides the TechnicalInfrastructure from the application.

Related Patterns:

There are, of course, other ways to speed up a system. Examples include the patterns described in [AuerBeck96], [Meszaros96] and [Petriu97].

3. Concluding Remarks

I hope the reader has found these patterns useful and for the most part understandable. I would welcome comments directed to me by e-mail. I would also invite other architects to extend this pattern language with their own experiences by providing alternate patterns, more detailed (or lower level) patterns or by improving upon the exposition of these patterns.

3.1 Future Plans

In future versions of this language, I hope to include more introductory material as well as better cross-referencing between the patterns in this language as well as more references to related patterns in other pattern languages.

3.2 Acknowledgments

I would like to thank my PLoP97 shepherd Robert Hirschfeld who made extensive comments on the very rough initial drafts of the manuscript and who generously provided several pages of material for inclusion in the pattern language. I would also like to thank Brad Appleton who provided comments as well as useful references to related material and spent much time feeding material into his fax machine.

4. References

[AuerBeck96] Ken Auer, Kent Beck, "Lazy Optimization: Patterns for Efficient Smalltalk Programming" in PLoPD96.

[Catalysis96] Desmond D'Souza, Alan Wills. *CATALYSIS--Practical Rigor and Refinement*. URL: http:/www.iconcomp.com

[Change] Doug Bennett, A Technique for Evaluating Designs, OOPSLA'94 Tutorial

[Cockburn96] Alistair Cockburn, "Structuring Use-Cases with Goals"

[Cope95] Jim Coplien, "A Generative Process Pattern Language" in PLoPD95

[Fowler96] Martin Fowler, "Analysis Patterns - Reusable Object Models"

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2.

[Hopley96] Allan Hopley "Decision Capture and Deferral Pattern Language" in [PLoPD96]

[HOPP95] Gerard Meszaros, "Half-Object Plus Protocol" in PLoPD95

[Meszaros96] Gerard Meszaros, "Patterns for Improving the Capacity of Reactive Systems" in PLoPD96.

[Meyer9?] Bertrand Meyer, "Object Oriented Software Engineering???"

[OopsArch96] Gerard Meszaros et al, "Workshop Report - Patterns in System Architecture" in Addendum to the Proceedings of OOPSLA96

[Petriu] Dorina Petriu, Gurudas Somadder, "A Pattern Language For Improving the Capacity of Layered Client/Server Systems with Multi-Threaded Servers" in proceedings of EuroPlop97.

[PLoPD95] Jim Coplien, Doug Schmidt Eds. "Pattern Languages of Program Design" published by Addison-Wesley in 1995.

[PLoPD96] John Vlissides, Jim Coplien, Norm Kerth, Eds. "Pattern Languages of Program Design 2" published by Addison-Wesley in 1996.

[PLoPD97] ???? Eds. "Pattern Languages of Program Design 3" to be published by Addison-Wesley in 1997.

[PWP97] Gerard Meszaros and James Doble, "Patterns for Pattern Writing" in [PLoPD97]

[Shaw95] Mary Shaw, Patterns in Software Architecture. In [PLoPD95]

[Siemans96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Pattern-Oriented Software Architecture - A System of Patterns. John Wiley & Sons ISBN 0-471-95869-7

[UML97] Rumboochatory, "UML 1.0 Specification" URL: www.rational.com