3 Building Software with Patterns

Within this process, every individual act of building is a process in which space gets differentiated. It is not a process in which pre-formed parts are combined to create a whole: but a process of unfolding, like the evolution of an embryo, in which the whole precedes its parts, and actually gives birth to them, by splitting.

Christopher Alexander, The Timeless Way of Building

Many important aspects of building software systems with patterns are not addressed by today's pattern descriptions, such as how to integrate a pattern into a partially existing design. Yet we want to use these patterns in our architectures, designs, and implementations. Complementary to pattern-specific implementation guidelines we therefore need general guidelines that support a pattern-based software construction.

3.1 Introduction

Every well-described pattern provides information about its own implementation [GHJV95] [POSA1]. Many such patterns also provide information about their refinement and combination with other patterns. However, almost all these guidelines are pattern-specific. They do not address aspects of specifying whole software architectures with patterns. Examples of such aspects include the integration of patterns into a partial design, the order in which a number of given patterns should be applied, and the resolution of problems that cannot be solved by a single pattern in isolation.

Answering such questions is important for using patterns effectively. Otherwise we apply patterns, but the resulting design will likely expose unnecessary complexity. Even worse, the architecture under construction may not provide the properties we need. For example, many patterns introduce a level of indirection to resolve a design problem. Connecting several of such patterns in a row may, however, result in an inefficient indirection cascade. Instead we want to combine the patterns without introducing multiple levels of indirection, but in a way such that the essence of each pattern still remains. Section 1.4, *Roles, Not Classes* illustrates another example, where a specific combination of two Observer [POSA1] structures introduces more complexity than it resolves.

The above discussion reveals that the application of patterns is not a mechanical task. It needs some experience to compose them to large structures in a meaningful way. The reason for this is obvious. Large-scale software architectures introduce problems and forces just because of their sheer size and inherent complexity. These problems and forces are often independent of the design issues addressed by individual patterns. Consequently, pattern-specific implementations, as useful they are when specifying the details of a particular pattern, are insufficient for applying patterns in general—when building large-scale and complex real-word software systems.

A pattern language for software architecture is one approach for supporting the effective use of patterns (see Chapter 4, *Towards Pattern Languages*). Its constituent patterns would not only provide solutions to specific design problems. The language would also provide a process—embedded in the patterns—that helps applying them: when, how, and in which order. A good example for such a language is Kent Beck's 'Smalltalk Best Practise Patterns' [Bec97]. On the other hand, we neither have such a pattern language for software architecture at hand, nor is it likely that such a language could be provided in the near future. There are still many aspects of software architecture for which no patterns are documented, and almost every existing pattern must be re-written in order to fit into such a language. Today, most patterns for software architecture are organized in pattern catalogs and pattern systems [GHJV95] [POSA1] [PLoP94] [PLoP95] [PLoP96]. Unfortunately, these do not provide the support for using patterns that we need.

To use patterns of a pattern catalog or pattern system effectively in software development therefore calls for appropriate guidelines. These should enable us to act like an expert software architect who has designed systems with patterns for years. At least they should help us being aware of the problems that may arise when applying patterns, and the pitfalls into which we can stumble. In addition, they should integrate with the steps and activities defined by existing analysis and design methods. For these reasons, the kinds of guidelines we are looking for cannot be simple one- or two-line statements. We need more information. What problem in using patterns does a specific guideline address? When does this problem occur? And how do we apply patterns correctly in the presence of the problem?

Consequently, guidelines which support pattern-based software development are patterns by themselves: exposing a context, a problem that arises in that context, and a solution that resolves the problem. They therefore should be documented as patterns, using an appropriate pattern form.

In this chapter we present 11 patterns for pattern-based software development. Most of them were mined over the past years and reflect what we, and our colleagues world-wide, have learned when using patterns to design and implement commercial and industrial software systems.

The initial set of patterns was developed while we were building a system for factory automation [Bus98], by self-observation of our design activities. They were completed with insights documented in the available body of literature on software design, such as [Gab96] and [Fow97], and with experiences gained in the development of a hot (steel) rolling mill process automation framework [BGHS98]. Some patterns also build on Christopher Alexander's work [Ale79] [ANAK87]. Altogether they complement and complete the specific implementation guidelines that accompany well-described patterns for software architecture, for example the ones in [Cope92] [GHJV95] [POSA1] [POSA2] [PLoP94] [PLoP95] [PLoP96].

The patterns were also applied in the development of TAO, the ACE ORB, a configurable, CORBA compliant, high-performance, real-time Object Request Broker. We describe its architecture and design in *TAO: The St. Louis - Munich Experiment* [POSA4], the fourth volume of the *Pattern-Oriented Software Architecture* series.

For the description of our patterns for pattern-based software development we adopt Christopher Alexander's pattern form: *Name-Context-Problem-Solution-Example* and extend it with a *Consequences* section.

3.2 Patterns for Pattern-Based Software Development

Before we describe our patterns for pattern-based software development in full detail we briefly summarize their intents:

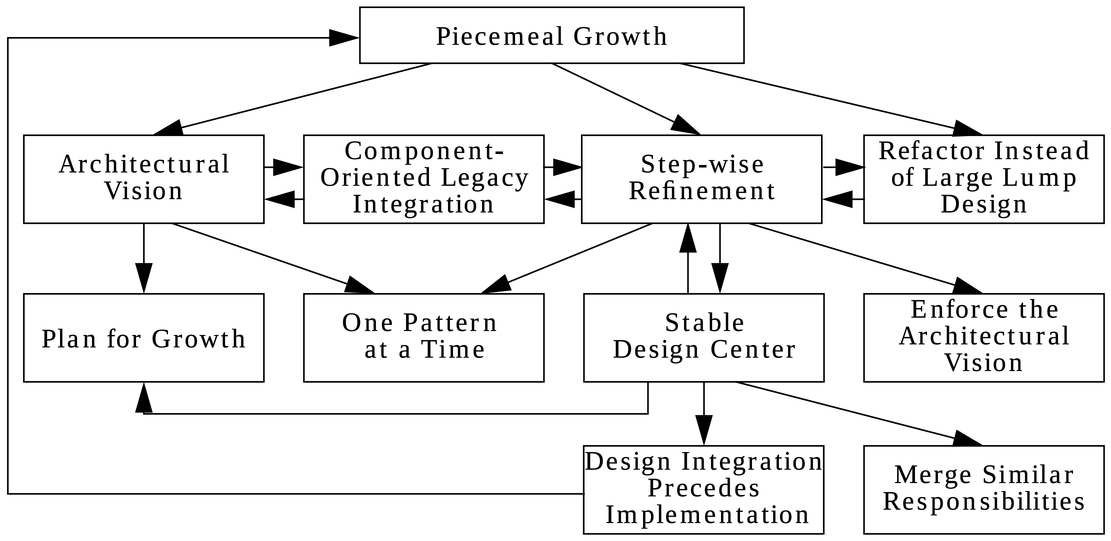
- *Piecemeal Growth* (43) outlines the overall process for constructing software systems with patterns. It is a jo-jo like process of top-down refinement and bottom-up refactoring, in which a software architecture grows and changes continuously until it is complete and consistent in all its parts.
- *Architectural Vision* (47) defines the first activity in the process of piecemeal growth: specifying the system's base-line architecture. Wherever useful, the pattern takes advantage of analysis and architectural patterns that help with the specification.
- *Step-wise Refinement* (53) defines the top-down design activities of the process of piecemeal growth. It describes how to resolve concrete design problems by detailing and extending the given

software architecture, if possible with help of analysis and design patterns.

- *Refactor Instead of Large Lump Design* (59) specifies the bottom-up activities of the process of piecemeal growth. It defines how to proceed if the optimal solution of a design problem does not integrate with the part of the existing design in which this solution must be contained.
- *Stable Design Center* (63) addresses how to specify designs that allow for extension and adaptation, but without the need to modify their key elements and abstractions. The pattern lists several concrete design patterns which help with this activity.
- *Plan for Growth* (67), outlines how a software architecture can be prepared for its own evolution over time. The pattern lists several architectural and design patterns that address this aspect.
- *Component-Oriented Legacy Integration* (71) defines how to take advantage of design patterns when integrating 3rd party or legacy components into an application.
- *Enforce the Architectural Vision* (75) supports the application of global design principles consistently in every part of a software architecture and at every level of detail.
- *One Pattern at a Time* (78) helps with combining several patterns which together should define the design of a specific system part.
- *Design Integration Precedes Implementation* (81) introduces a process for implementing a pattern in the context of a given software architecture.
- *Merge Similar Responsibilities* (85) shows how to combine patterns with existing design elements which partly provide similar or related responsibilities as the patterns' participants.

Our patterns for pattern-based software development strongly depend on each other. You start with the first pattern—*Piecemeal Growth* (43)—and will be directed to the patterns that apply thereafter. These complete the idea outlined in *Piecemeal Growth* (43) and also reference those patterns which are needed for their own completion. Every pattern thus makes most sense in the context of the patterns it precedes and completes. In other words, our patterns for pattern-based software development form a *pattern language*.

The diagram below briefly illustrates how the patterns in the language build upon each other—if a pattern uses another pattern, it points to it with an arrow.


Piecemeal Growth
Architectural Vision
Component-Oriented Legacy Integration
Step-wise Refinement
Refactor Instead of Large Lump Design
Plan for Growth
One Pattern at a Time
Stable Design Center
Enforce the Architectural Vision
Design Integration Precedes Implementation
Merge Similar Responsibilities

You may keep this overview in mind when reading the patterns. It allows you to recall the purpose and solution idea of a pattern you have not yet read, but which is referenced by the pattern you are reading. The diagram provides you with an overview of the language's structure.

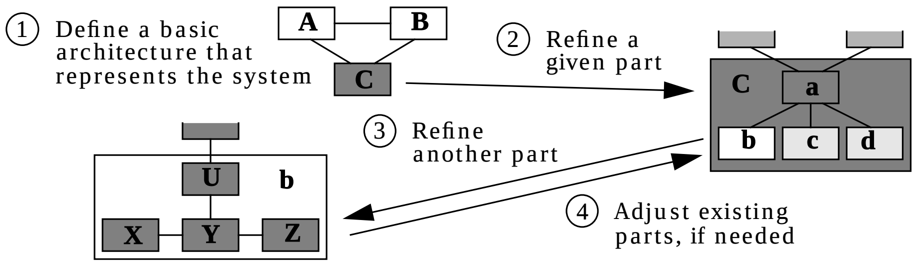
Piecemeal Growth

Context We want to design a new software system.

Problem *How can we ensure that the system is well-structured and organized*? In particular, that its architecture meets specific functional and non-functional requirements [POSA1], and that its architects, designers, and programmers feel habitable in it [Gab96]. Three *forces* are to be balanced:

- The software architecture should model the structural organization of the system's application domain in a way such that domain-specific tasks can be executed as required. In addition, it should support the system's adaptation, maintenance, and evolution.
- The solutions to particular design problems must meet these problems' own requirements, integrate with the higher-level parts of the design in which they are contained, and must allow for the optimal resolution of lower-level design problems.
- Assembling an application's architecture bottom-up will likely result in a system structure where many parts will be designed according to different design principles that do not match, such as strict information hiding and open implementations [POSA1]. On the other hand, when following a pure top-down approach, higher-level design structures will often be inappropriate for resolving lower-level design problems effectively, since these are mostly unknown until they arise.

Solution *Create the architecture of the system in a process of piecemeal growth* [Gab96]. It is a jo-jo like process of top-down refinement and bottom-up refactoring, in which a software architecture grows and changes continuously until it is complete and consistent in all its parts.


1 Define a basic architecture that represents the system
A
B
C
2 Refine a given part
C
a
b
c
d
3 Refine another part
U
b
X
Y
Z
4 Adjust existing parts, if needed

The process begins with specifying an *Architectural Vision* (47): what is the fundamental subsystem structure of the application and what are the general design principles that should govern its refinement. The subsystem structure provides the basis for the process of piecemeal growth. It defines the core components of the application domain and outlines the infrastructure components the system needs for its proper operation, for example, to interact with its users. The design principles, such as separation of concerns and separation of interface and implementation [POSA1], address the system's non-functional requirements, for example, extensibility and adaptability.

For every aspect of the *Architectural Vision* (47) that requires a more detailed specification, unfold its given design by *Step-wise Refinement* (53). Unfolding means extending and detailing the existing architecture with new components and relationships. These should always reflect the structural organization of the domain or infrastructure concept they are representing, but in accordance to the general design principles defined by the *Architectural Vision* (47). Every part that we add to the design then serves as a new 'micro-architecture' that can be unfold, until the whole architecture is complete.

It will likely happen, however, that the optimal decomposition of a given system aspect does not integrate with the part of the application's design in which its specification is to be contained. When this happens, recursively *Refactor Instead of Large Lump Design* (59), to adjust the existing architecture before continuing with its further refinement.

Consequences The process of piecemeal growth helps creating a coherent software architecture. Due to a balanced interplay between *Step-wise Refinement* (53) and *Refactor Instead of Large Lump Design* (59) we look onto every design aspect from three perspectives: the system view, the view of the aspect itself, and the view of lower-level issues that may impact its specification. At every level of abstraction the design of the system will correctly model the application domain. Domain-specific services the system offers to its users can be executed effectively. In addition, the architecture follows common design principles that ensure its non-functional requirements, from the coarse-grained structure down to the very details of its implementation. By this we explicitly consider the needs of the software engineers who must develop and maintain the system.

Example Suppose we want to design a warehouse management system[1]. Basically, such an application manages physical storage in a warehouse as well as the items stored, takes care for storing, fetching and commissioning items, organizes replenishment of production lines, controls the underlying process automation equipment, and keeps book of all its activities. In short, it organizes the whole warehouse operation.

Warehouse management systems are highly distributed and interactive. There are central servers for the database, the system's functional core, and the automation equipment control. Multiple clients, many of which are mobile, are connected to the servers. These clients often provide specialized user interfaces, such as for item receiving and commissioning, quality assurance, working-off item transport orders, and system administration. Finally, the system receives input from many sensors which listen for events to occur within the automation equipment. Each client of the system operates independently of other clients. Inputs are usually delivered concurrently, and must be coordinated and interpreted to trigger the 'right' operations in the system. Likewise, results of operations must be re-submitted to the users of the system and the warehouse basic automation.

According to the process of piecemeal growth the first step to perform is the definition of the system's *Architectural Vision* (47): what are its basic subsystems, their responsibilities and relationships, and, very importantly, their collaborations. A second aspect to consider is infrastructure, such as the mechanisms needed for inter-component communication in a distributed application. For the warehouse management system we will end up with subsystems that implement the warehouse management core functionality, the controlling and monitoring of the warehouse automation equipment, the warehouse's administration, the user interface, the database, and the location transparent inter-process communication.

Once the system's base-line architecture is defined we can refine the subsystems that it introduces, for example, the subsystem providing the warehouse management core functionality. By *Step-wise Refinement* (53) we specify and detail the subsystem's design: beginning

1. We will use the warehouse management system as a running example throughout all patterns in this chapter. By this we are able to show in detail how a concrete software architecture emerges by applying patterns in a process of piecemeal growth.

with the structure for representing physical storage in a warehouse, continuing with integrating the functionality that operates on this structure, and ending with preparing the design for extension and adaptation.

As it happens in real life, the design of this subsystem will not expose the desired quality right away. For example, if the original specifications for representing physical storage and the transport ways between them are not appropriately integrated with each other, it is hard to provide a straight-forward design and implementation of an item transport service. In such cases, we need to refactor existing design structures so that they become suitable for integrating the solutions to design problems that were unknown when specifying the original design.

This interplay between refinement and refactoring will continue until the whole subsystem design provides its required quality. Similarly, we then can specify other parts of the application, such as the user interface subsystem.

Step by step the software architecture of the system will grow until it is completely specified and defined. The architecture gets differentiated with every design step we take, but always according to the design principles pre-scribed by the larger structure whose parts we are refining and the needs that arise from lower-level design aspects.

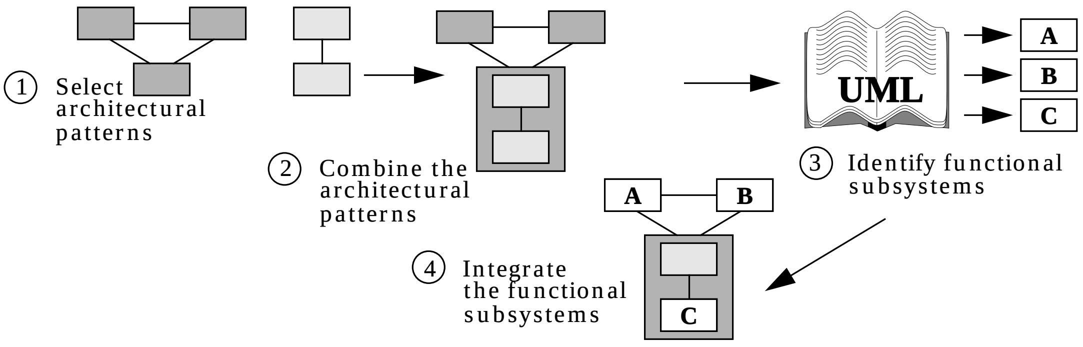
Architectural Vision

Context We are at the beginning of the process of *Piecemeal Growth* (43).

Problem *How can we define a base-line architecture* that captures an application's fundamental subsystem structure? Four *forces* arise:

- The base-line architecture must provide a global view onto the system: all relevant core components of the application domain must be present, and also the infrastructure components the system needs for its proper operation, such as for inter-component communication within a distributed computing environment. The relationships and collaborations of these components must allow for a correct and effective implementation of all services the system must offer to its users. On the other hand, the base-line architecture should be as simple as possible: essential design ideas must not be hidden within overly fine-grained design specifications. Furthermore, we cannot anticipate all future extensions to the system at the time it's base-line architecture is defined.
- The system's underlying design principles, such as separating user interface from service processing, must be clearly exposed, so that further refinement of the base-line architecture can follow these principles.
- The system may build upon 3rd party or legacy software.
- We want specify the base-line architecture with help of patterns.

Solution *Create an architectural vision*: a coarse-grained subsystem design which governs the specification of the application down to its implementation.


1 Select architectural patterns
2 Combine the architectural patterns
UML
A
B
C
3 Identify functional subsystems
A
B
C
4 Integrate the functional subsystems

First, specify the system's infrastructure: how does it interact with users, how do components cooperate, how its persistence managed, and how does application functionality integrates with this environment.

To define this infrastructure, identify all system-wide non-functional properties the application must expose, for example that it is distributed and interactive. According to the guidelines given in [POSA1], then select architectural patterns [POSA1] that address these properties—by providing corresponding system structures and their underlying design principles. If it is a requirement to follow a specific design philosophy for achieving a non-functional property, select an architectural pattern that implements this philosophy. For example, GUI builders often pre-scribe a particular way for connecting the user interface with the application's core functionality. For properties which are not addressed by the available architectural patterns, implement suitable structures by using a conventional design method. Or, use patterns that guide the process of specifying such structures, like the ones presented in [Coad95].

Consider that it is next to impossible to anticipate all extensions to the system customers might require over its life time. Therefore *Plan for Growth* (67) such that the system can evolve without a re-definition of its base-line architecture. Apply the selected architectural patterns, *One Pattern at a Time* (78), with descending order of importance of the system properties they address.

Second, identify the functional subsystems of the software under construction, including their relationships and collaborations. These should model the core structure of the system's application domain. Do not try to define a 'master-plan' that intends to capture every possible functionality the system might must expose in the future. Focus only on the requirements known by today. Since the infrastructure is prepared for growth, the specification of additional functionality can be deferred to the point in time it is requested.

If possible, use analysis patterns to support the specification of functional subsystems. Select these patterns according to the guidelines given in [POSA1]. Patterns for domains such as health care, accounting, and telecommunication, can be found in [Fow97], [PLoP94], [PLoP95], and [PLoP96]. If no analysis patterns are available for the system's application domain, use a conventional analysis and design method to define the functional subsystems. Or, use patterns that guide the process of identifying such subsystems [Coad95]. For 3rd party or legacy software that must be used, provide a *Component-Oriented Legacy Integration* (71). You are set if for every major functional responsibility which the application must provide a corresponding subsystem is defined, or if such a responsibility can be provided by several subsystems in collaboration.

Finally, integrate the functional subsystems with the system's infrastructure. Substitute its 'placeholders' for application functionality with concrete functional subsystems, and map their relationships onto the communication facilities the infrastructure defines.

The architectural vision represents a fundamental design decision that will govern every other design activity that follows. Thus, consider carefully which functional subsystems the application should provide and in which infrastructure they should live. Double-check the final specification with the system's requirements before calling the architectural vision done. Changes and adjustments to a system's base-line architecture at later points in time, when parts of its detailed design and implementation are already developed, will result in significant and cost intensive modification efforts.

Consequences We will receive the base-line architecture of the application under development: its relevant subsystems, their responsibilities and collaborations, and also the design principles that guide its further specification and refinement. Since we only focus on the application's decomposition into major functional and infrastructural subsystems, the base-line architecture is not overly detailed. However, the preparation of the base-line architecture for growth ensures the system's future evolution. The base-line architecture also adopts proven design concepts, due to the use of analysis and architectural patterns.

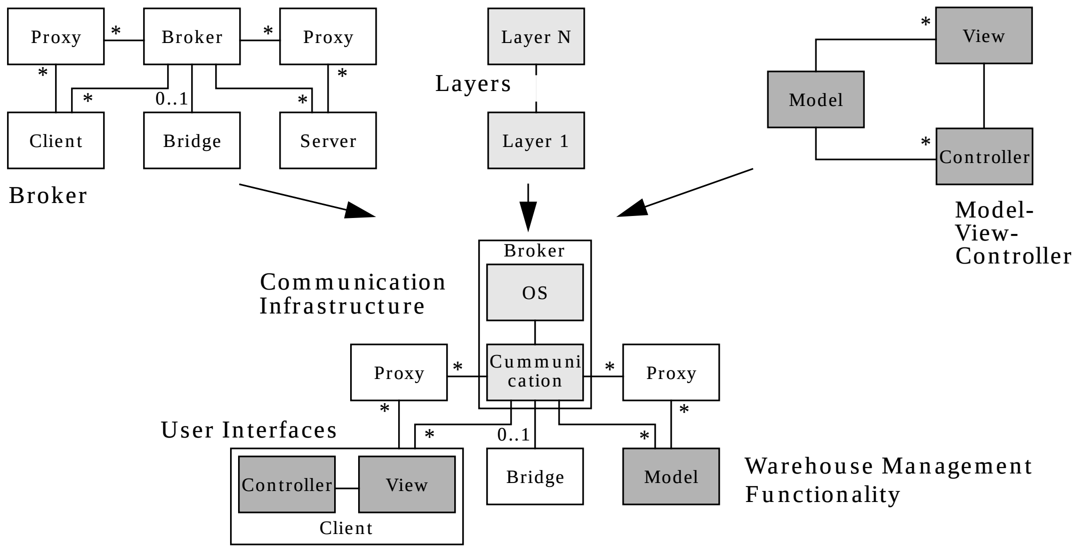
The architectural patterns also support the integration of functional with non-functional aspects: all services of the system can be effectively implemented, but under consideration of non-functional aspects that positively impact the system's development, maintenance, evolution, and adaptation [POSA1]. By applying the architectural patterns in their order of importance, it is ensured that design structures and design principles that help with implementing a more important non-functional system property will govern the system's base-line architecture more than design structures and design principles that address a less important property. The analysis and architectural patterns further help with communicating the architectural vision: what kind of subsystem structure is it, why is it exactly this structure, what are its underlying design ideas, and how is it built. We have created an architectural vision.

Example An analysis of the non-functional requirements for the warehouse management system reveals the following four factors that impact its base-line architecture:

- *Distribution*. The system is highly distributed across a computer network. Core functionality is located on a central server, with many clients accessing this functionality, from PCs to small hand-held terminals on forklifts.
- *Human-computer interaction*. Users communicate with the application using a huge variety of user interfaces. Examples include form-based interfaces, hand-held terminals with buttons to press, and menus and dialog boxes for event-driven interaction.
- *Platform independence*. The system must run on multiple hardware and operating systems. For example, we might want to use PCs with Windows NT for user interface clients and a UNIX server for the system's core functionality.
- *Integration*. Wherever useful we want to integrate 3rd party products, such as databases, or existing legacy software, such as for resource planning.

There are three architectural patterns that help with resolving above non-functional requirements. The Broker pattern [POSA1] supports distribution and integration, the Model-View-Controller pattern [POSA1] human-computer interaction, and the Layers pattern [POSA1] platform independence.

Let us assume, in the context of this example, that distribution is the most important system property. Thus, we apply Broker first. The resulting structure basically consists of clients and servers, and a broker for routing messages across process and machine boundaries. The Model-View-Controller pattern is then integrated into this structure, with the servers representing the different parts of the model, and the clients the view and controller components. The Layers pattern is applied in third place. It helps with splitting the broker component into two parts: the plain communication logic and a component which abstracts from low-level operating system features that implement basic communication mechanisms.



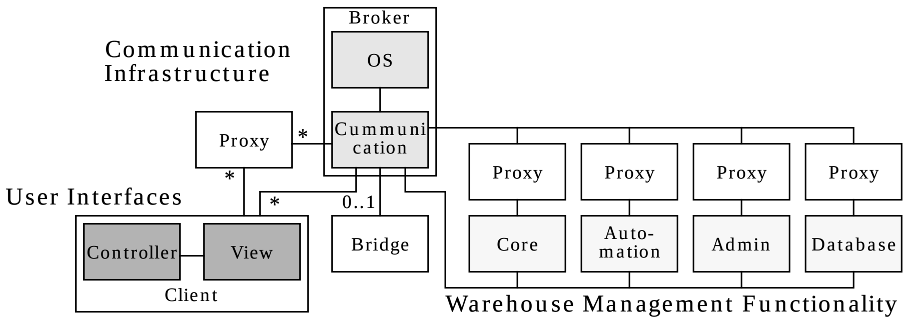
In terms of the warehouse management system this means that we develop user interfaces as thin clients, and the functional behavior of the system—the model component in terms of Model-View-Controller—as servers, all on top of a Broker-based communication infrastructure. Due to the Broker architecture we are able to distribute the system in a computer network. If we further use a ‘standard’ Broker implementation, such as a CORBA [OMG98] Object Request Broker, we can also integrate all 3rd party components that follow its communication model. Due to MVC’s separation of human-computer interaction from application functionality we are able to provide multiple user interfaces for controlling the system: forms, menus, dialog boxes, windows, panels, command lines—whatever is needed. Layers, finally, keeps the communication logic independent from operating system specifics, and thus supports the system’s portability to other hardware platforms.

The result of applying Broker, Model-View-Controller, and Layers is the basic component-communication and component-cooperation infrastructure for the warehouse management system. Broker defines how clients can access remote servers. Model-View-Controller specifies how user interface components interact with the functional core, and Layers how the general communication functionality exploits the corresponding mechanisms of the underlying operating system. The patterns further define the two general design principles for the system: separation of application from communication functionality and separation of the application functionality from the system's user interface.

For the functional decomposition of the warehouse management application we have no domain-specific patterns available. A 'conventional' analysis results in the following four subsystems:

- The *warehouse management core functionality* implements a representation of the warehouse's physical structure and all services that operate on this structure. Examples of such services include storing and fetching items.
- A subsystem for *controlling and monitoring the warehouse automation equipment* provides *sensors* which allow to acquire the actual state of the warehouse automation equipment, as well as *actors* for controlling its further behavior.
- The *warehouse administration* allows to configure the warehouse management system and to monitor its current state.
- The *database* is responsible for keeping and maintaining persistent data in the application.

We integrate the four functional subsystems into the system's infrastructure such that each becomes its own server component.



The architectural vision of the warehouse management system is complete.

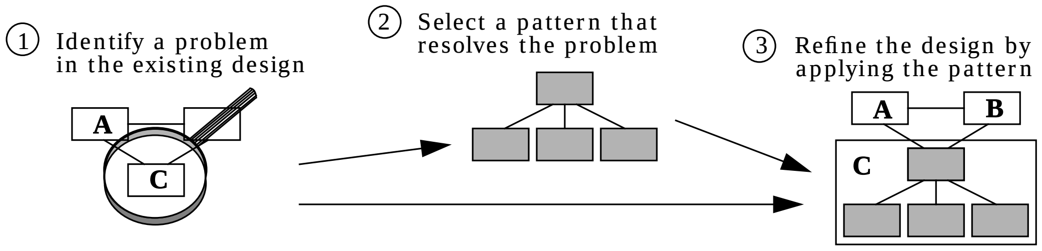
Step-wise Refinement

Context We are specifying a software architecture in a process of *Piecemeal Growth* (43). The *Architectural Vision* (47) of the system is defined.

Problem *How can we resolve a given design problem most effectively*? Four *forces* must be considered:

- The solution to the design problem must meet this problem's own requirements, but must also obey to the global design principles defined by the *Architectural Vision* (47).
- In order to avoid unnecessary system complexity, we do not want to resolve similar design problems completely differently.
- The system may build upon 3rd party or legacy software.
- We like to solve the problem with help of patterns.

Solution *Resolve the design problem by step-wise refinement*. Extend the software architecture with new components and relationships, and decompose existing design elements with finer-grained components and relationships, such that both the functional and non-functional aspects of the design problem are addressed.


1 Identify a problem in the existing design
A
C
2 Select a pattern that resolves the problem
3 Refine the design by applying the pattern
A
B
C

First, specify domain-specific components, their relationships, and corresponding collaborations that will address the functional aspects of the design problem at hand. Use analysis patterns for this specification, if possible, such as those described in [Fow97], [PLoP94], [PLoP95], and [PLoP96]. With help of design patterns then provide an infrastructure for integrating the domain-specific components, relationships and collaborations into the existing software architecture. The selected design patterns should not only address the non-functional aspects of the original design problem but should also obey to the design principles pre-scribed by the higher-level design that is refined. Select the analysis and design patterns according to the guidelines given in [POSA1]. If you are refining a part of the architecture that was specified by a pattern in a previous refinement step, you may also consult this pattern's implementation guidelines. These often reference further patterns which may also help in the current refinement step.

Split complex problems that cannot be resolved by a single pattern into smaller subproblems that can be solved by several patterns in combination. Always *Enforce the Architectural Vision* (75): ensure that solutions to specific design problems consistently follow the system's global design principles. Provide a *Component-Oriented Legacy Integration* (71) if 3rd party or legacy software must be used. If several patterns are selected to refine a specific part of the system, apply these patterns *One Pattern at a Time* (78), with descending order of importance of the aspects they address. When refining subsystems and large components, each of these should define a *Stable Design Center* (63).

If no patterns are available for resolving the design problem, create the solution by using an appropriate analysis and design method. Or, even better, use patterns that guide the analysis and solution of the problem, such as the patterns presented in [Coad95].

If the optimal solution of the design problem does not integrate with the design part you are detailing, *Refactor Instead of Large Lump Design* (59), to adjust the existing architecture before continuing with its further refinement

Recursively refine the components and relationships that were added to the design, if they give rise to new design problems. Stop the refinement when all design problems are resolved or if they can be implemented in a straight-forward manner without further decomposing existing design elements or extending the given software architecture.

Consequences We receive a pattern-based design that is coherent and complete in all its parts. The use of analysis patterns helps integrating the functional aspects of the design problems that arise during the refinement, and the use of design patterns their non-functional aspects. The resolution of design problems that share a common problem core with the same design principles avoids complex and patch-work-like design structures. Legacy or 3rd party software fits seamlessly into the existing design, since we provide a system-oriented integration of such artifacts rather than using them as they are. By applying the selected patterns in their order of importance, design structures that help with implementing a more important design problem will govern the system's architecture more than design structures that address a less important design problem. Designing subsystems as stable design centers supports us in specifying software architectures that allow for extension and adaptation, but without the need to modify their key elements and abstractions.

Example When specifying the subsystem that provides the functional core of our warehouse management system, a major aspect to consider is how to represent physical storage.

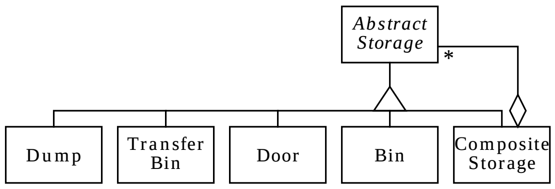
Unfortunately, there are no analysis patterns for the warehouse management domain available. To resolve the problem of storage representation we therefore follow an object-oriented analysis and design method.

Warehouses typically expose a hierarchical storage organization. They may consist, for example, of a certain number of aisles, with each aisle comprising two sides, each side a number of racks and each rack a number of bins. In addition, there is special-purpose storage, such as quality assurance, transfer bins, doors, and dump area. From a general perspective, each part of such a structure is a warehouse for itself. It is possible to store an item in an aisle, a side of an aisle, a rack, and a bin, or at a door. We can also fetch an item from each of these kinds of storage.

While this general schema underlies almost all warehouse structures, actual 'instances' vary significantly, for example in their number of aisles, racks per side, transfer bins, dump areas, and so forth.

The challenge here is to provide a design that captures this huge variety of possible storage organizations and configurations under a single 'warehouse' view. At the same time, the design should also be open for adaptation, modification, and extension. Furthermore, we do not want client components of our system to be dependent on this structure. Whether a storage is an aisle or a bin should be hidden from them.

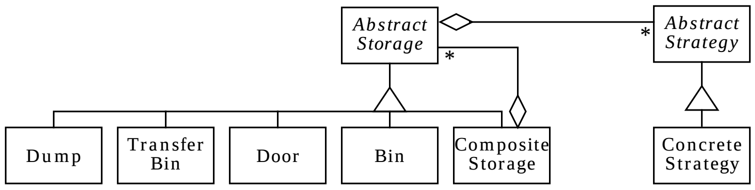
The Composite pattern [GHJV95] seems to fit well for resolving our problem. It provides an extensible design for representing arbitrary hierarchical object structures, while hiding the details of concrete structures from clients. We therefore model the physical warehouse structure as a hierarchy of composite classes: with a class `AbstractStorage` at its top, a class `CompositeStorage` for representing aisles, sides and other compound storages, and separate leaf classes for each 'atomic' storage such as bin, transfer bin, door, quality assurance, and dump area. Interfaces of operations that are shared by all kinds of storages are integrated into the `AbstractStorage` class, such as for storing and fetching an item.



However, even though all kinds of storage share a common interface, it is impossible to implement these methods in the corresponding classes of the hierarchy, as proposed by 'traditional' object-oriented design guidelines. Different kinds of storage that are represented by class `CompositeStorage`, such as aisles and racks—or even different instantiations of the same kind of storage—may differ in their concrete behavior with respect to these functions. For example, when storing items, there might be different bin selection strategies for different racks in the warehouse. There is only commonality in the interfaces and semantics of above functions, but not in their detailed behavior.

We therefore need to detail and extend the given design such that it supports the configuration of individual storage with individual behavior, for example with search strategies for free storage locations. However, we do not want to lose the common view onto the storage as a whole, which is provided by the current architecture. This is a case for the Strategy pattern [GHJV95].

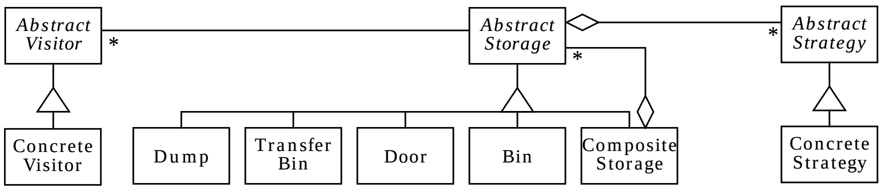
We introduce an `AbstractStrategy` class which provides common interfaces for all configurable aspects of a physical storage and connect it with the `AbstractStorage` class of our existing storage hierarchy. The methods of concrete storage classes are then refined to delegate the execution of all storage-specific behavior to a `ConcreteStrategy`, and the whole design is extended with the corresponding strategy hierarchy.



The refined design provides us much flexibility in configuring concrete warehouses—both with respect to physical storage structures as well as their configuration with concrete behavior.

In addition to storage-individual services, we must also implement services that operate on the whole warehouse structure, such as calculating free storage capacity. However, we like to integrate such services without giving up our current design.

Addressing this problem is the intent of the Visitor pattern [GHJV95]. Visitor provides a mechanism for 'hopping' over a given object structure in order to perform certain services. Thus, Visitor seems the be *the* candidate for refining our design.



We detail our current design as follows. A class `AbstractVisitor` provides the general interface for starting an operation on a given object. The class `AbstractStorage` is refined with a method for 'accepting' a Visitor for performing an operation on a storage structure. `ConcreteVisitor` classes then implement these operations. We have added even more flexibility to the design of our warehouse structure and the services that operate on it.

Further patterns apply for specifying other aspects of the subsystem, such as Flyweight [GHJV95] for providing properties for storages and Iterator [GHJV95] for supporting different traversal strategies for visitors.

Once the subsystem is completely defined we can turn our attention to other, not yet specified subsystems of the warehouse management system. Step by step, by applying patterns and governed by the design principles prescribed by the *Architectural Vision* (47), we refine the original base-line architecture, until its details are fully unfold.

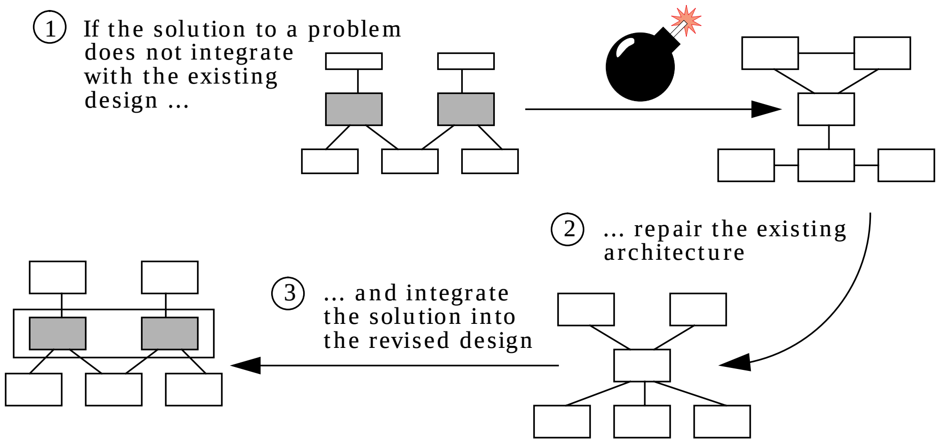
Refactor Instead of Large Lump Design

Context We are developing a software system in a *Piecemeal Growth* (43) fashion, being in the process of *Step-wise Refinement* (53).

Problem *How can we deal with design problems that arise later in the design process*, but which solutions don't integrate with the system's existing software architecture? For example, problems due to 'add-this-feature-immediately' requests by our customers. Or, when the programming language constraints the design solution space, due to features it does not offer. Resolving this conflict means to balance two *forces*:

- Aspects of the design problem that have an impact on already existing parts of the software architecture cannot be ignored. Rather the software architecture has to be specified under consideration of these aspects.
- The solution to the design problem at hand should integrate with the existing design. It should also be consistent to the global design principles defined by the *Architectural Vision* (47).

Solution *Refactor the existing architecture* [Opd92] [Fow97]. Do not constrain the optimal solution to the design problem at hand by inappropriate larger structures and global design principles.


1
If the solution to a problem does not integrate with the existing design ...
2
... repair the existing architecture
3
... and integrate the solution into the revised design

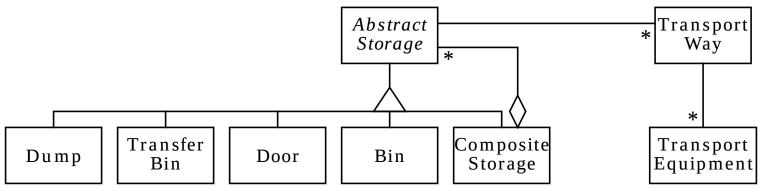
Identify all aspects of the solution to the design problem that impact the existing software architecture—specifically the part which will contain the solution to this problem. Adjust this part of the design according to the identified aspects. Similarly, revise the specification of all sub-parts of the changed design which are affected by the adjustment. We may need to modify these as well. Then embed the solution to the design problem into the revised software architecture.

In other words, we resolve—by *Step-wise Refinement* (53)—those higher-level design problems again which lead to the structure that is inappropriate for integrating the solution to the design problem at hand. However, this time we do not just take into account these higher-level problems' own requirements and the global design principles defined by the *Architectural Vision* (47). We also consider explicitly the needs of the lower-level design problem we are currently facing.

When adjusting a given structure, avoid violating even more design principles, if possible. Otherwise recursively repair the affected design structures—if necessary up to the *Architectural Vision* (47).

Consequences The software architecture under construction is not a product of large lump design [ANAK87]: a fixed construct where each part, once being specified, stays untouched forever. Rather the architecture changes and grows continuously all the time, and at all levels of granularity, to stay coherent and consistent. Thus, the aspect of refactoring is a vital principle of the process of *Piecemeal Growth* (43).

Example Suppose that the integration of transport ways and transport equipment into the structure for the warehouse management system has lead to the following design.



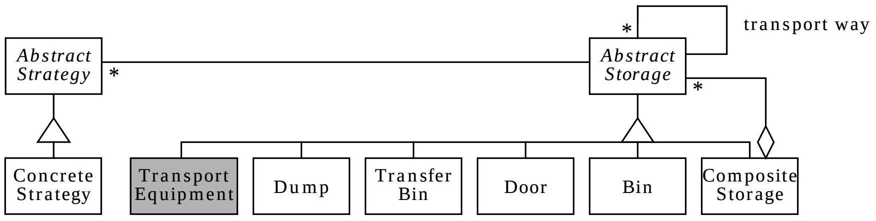
For each transport way between a pair of storage we introduce a separate `TransportWay` object that represents this relationship. If this object connects with a compound storage, such as an aisle, we assume that all lower-level storage is connected as well, in our example both sides within the aisle, and also its racks and bins. For each transport way we can further specify the transport equipment that can serve it. This structure allows us to initiate and perform item transports within the warehouse.

However, even though this design represents the physical world quite well, it does not seem to fit for our software model. The reason is simple. Organizing transports requires to involve three kinds of objects: storage, transport ways and transport equipment. Clients must distinguish between these objects explicitly; they cannot be treated uniformly. Neither they are organized in a single class hierarchy, nor do they provide uniform interfaces.

On the other hand, at least two of these classes share parts of their ‘semantics’. Transport equipment, such as forklifts, can store and release items, like a storage. The difference is that transport equipment is mobile, compared to a fixed location of a storage, and that it exposes pro-active behavior, compared to a storage’s re-active behavior.

As a result the item transport service implementation exposes some overhead to map between the two ‘worlds’, even though storage and transport equipment provide similar behavior from a general perspective. In other words, the existing design is not appropriate for a straight-forward integration of an item transport service.

However, there is a very simple and elegant solution for this conflict. Rather than providing separate class structures for storage and transport equipment, we integrate the equipment into the storage class hierarchy.



By this we can treat storage and transport equipment uniformly. The differences in their concrete behavior, such as mobile versus fixed and pro-active versus re-active, are factored out into strategies, which we have specified already (see *Step-wise Refinement (53)*).

Transport ways also degenerate to a single reference between storage. Attached to the base class of the class hierarchy we can form, in addition to arbitrary storage hierarchies, chains of storage. This means, that from a client perspective, an item transport can be seen as passing an item along a storage chain, with 'real' storage and 'transport' storage in alteration, until the item arrives at its final destination. This view allows a 'lean' approach to implementing item transport functionality.

We refactor our original design accordingly.

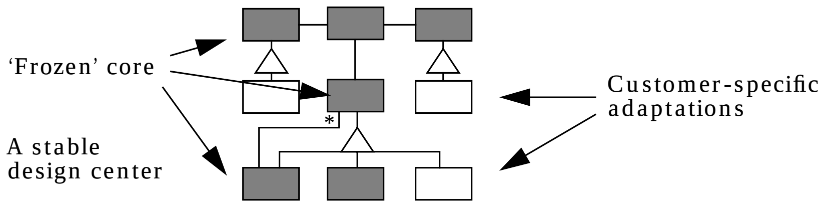
Stable Design Center

Context We are specifying a software architecture by *Step-wise Refinement* (53).

Problem *How can we ensure that a design is coherent and stable on the one hand while being open to adaptation and evolution on the other hand*? Software development organizations cannot afford building separate systems for every single customer—if these systems have the same purpose and share a common core. Too many resources would be bound: manpower, time, and money. Rather today's market forces require to continuously reduce development and maintenance costs of systems as well as their time-to-market [BGHS98]. However, systems that are sold to many customers must be adaptable to their specific needs. Almost no two sets of requirements to a particular application are identical. For example, every warehouse has its unique physical structure and its operation often follows company-specific strategies. Two *forces* further complicate this problem:

- The more that a system's design cannot be modified, the more stable it stays over its lifetime, but the less adaptable to customer-specific requirements it is. On the other hand, the more that a software architecture supports its evolution, the more likely a modification will break its coherency.
- Fundamental aspects of a software architecture should not be subject to change. Rather its structural and behavioral integrity should be preserved over the system's lifetime. Otherwise we may lose its *Architectural Vision* (47).

Solution *Create stable design centers* for every subsystem or component of the application that provides services which are needed in every system version or structures on which many services operate. Examples include the fundamental domain model or the system's communication infrastructure.



First, perform a Commonality/Variability Analysis [Cope98] to identify which aspects of such a subsystem or component stay invariant across different implementations and which aspects can vary. Aspects that commonly stay invariant include the subsystem's or component's general responsibility, its interface to clients, the communication protocols for using the interface, but also its basic structural decomposition and the principles of how these internal components cooperate. Aspects that typically vary are algorithms for processing particular operations, add-on behavior to specific services, and its user look-and-feel, if there is any.

Capture the invariant aspects of each analyzed subsystem or component in a so called *design center*, a micro-architecture which is intended to stay stable over the system's whole lifetime. Such parts of a software architecture are also referred to as *frozen spots* [Pree95]. To specify the design center, follow the guidelines defined by *Stepwise Refinement* (53).

Refine the design center with components and relationships that capture those aspects of the subsystem or component which—over the lifetime of the system—are likely to be modified or adapted to customer-specific requirements. Such parts of the design are also called *hot spots* [Pree95]. Integrate each variant aspect in a way that the resulting architecture supports the required kind of variability.

Several patterns help with this. For example, filtering existing service behavior, or adding new behavior to a service, is supported by the Decorator pattern [GHJV95]. In general, a decorator allows to execute additional operations before and/or after invoking an invariant service. Template Method and Strategy [GHJV95] help implementing multiple versions of a specific service. Both patterns define a so called template method [Pree95], which captures the invariant parts of a service and delegate the execution of variant behavior to hook methods [Pree95], which can be implemented in an application specific manner. The Acceptor-Connector pattern [POSA2] allows to vary the strategies for connection establishment between remote components in a distributed system, since it decouples connection establishment from service processing.

Avoid direct dependencies between client subsystems and components of the system part we are specifying, and the internal structure of the design center we have created. The Facade pattern [GHJV95], for example, supports a defined access to services of a design center without exposing its internal structure. Abstract Factory and Builder [GHJV95] free clients from details of how to instantiate the design center by introducing special components that implement the creation process. Service Configurator [POSA2] helps with the design center's initial configuration and subsequent run-time re-configurations. The pattern defines a component which allows to dynamically load and execute abstract factories and builders.

Consider that the design center will likely grow over time, due to evolving and also new requirements. Therefore *Plan for Growth* (67) such that extensions to the design center can be integrated without modifications of existing parts.

Consequences We get a design that is stable over the lifetime of the system. It has defined boundaries and responsibilities, and it is well-specified how it is to be used. Parts of the system that depend on the design center can rely on this stability. Yet the design center is open for its own evolution. Details can be adapted to specific requirements without the need to change the center's key elements or any other part of the software architecture. Future extensions can be attached with only few and local modifications to the existing structure of the design center.

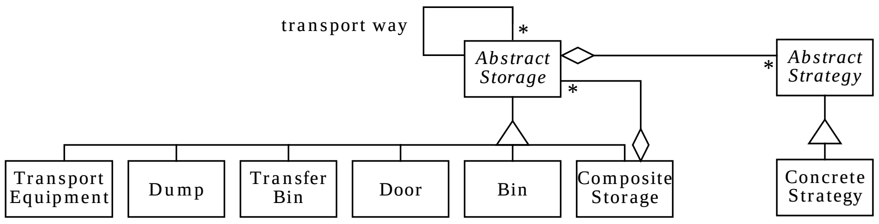
In other words, we have created a *product-line architecture* [BCK98]. Every running version of a system appears to be hand-crafted, since it reflects requirements that are specific to a particular customer, but in reality all versions share a common architecture. We are thus able to balance the main conflict of our original problem: meeting the requirements of both the customers of a system *and* the organization that develops it. If such a product-line architecture is completed with pre-defined implementations for all 'frozen' aspects, we end up with an (*application*) *framework* [POSA1].

If, however, we must change an aspect of a design center that is visible to its client components—and though minimizing this need we cannot completely exclude it for the whole lifetime of the system—this modification will be expensive. Such a change will affect almost all parts of the system that depend on that aspect. This is the major liability of creating stable design centers.

Example The design we developed for the warehouse management core functionality[2] forms a stable design center and which can be completed to an application framework. The Composite hierarchy [GHJV95] for representing storages provides—by the two classes `AbstractStorage` and `CompositeStorage`—a stable, but extensible, adaptable, and modifiable infrastructure for composing arbitrary physical storage configurations. Customer-specific storage implementations are completely encapsulated in the leaf classes of the structure.

The Strategy pattern [GHJV95] supports the configuration of storage objects with specific behavior. Class `AbstractStrategy` provides the infrastructure. Customer-specific behavior is implemented in the `ConcreteStrategy` classes.

Concrete configurations of this structure with customer-specific storage, storage behavior, and new 'general' services, do not break its design, neither its fundamental organization, nor its interfaces to clients. Customer-specific parts can also be added and changed independently.



With patterns like Abstract Factory [GHJV95] and Service Configurator [POSA2], we can further free clients from dependencies on how to actually configure a warehouse. They are totally shielded from the design center's internals.

2. Se the example sections of the patterns Step-wise Refinement (53) and Refactor Instead of Large Lump Design (59).

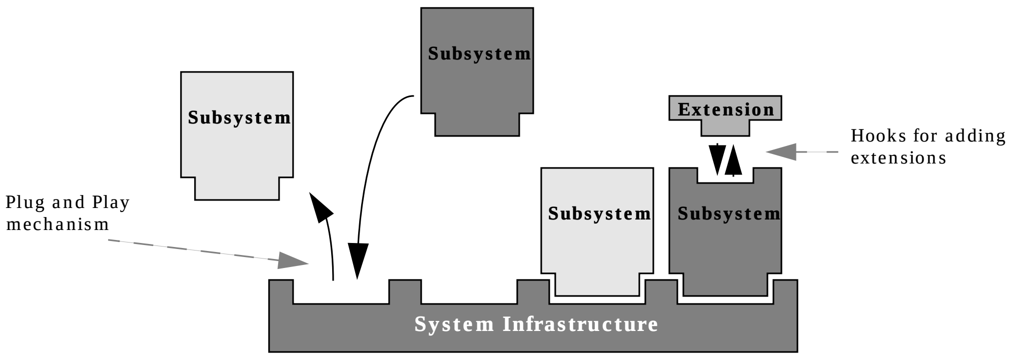
Plan for Growth

Context We are specifying a system's *Architectural Vision* (47) or one of its *Stable Design Centers* (63).

Problem *How can we prepare a software system for its evolution*, after the original development and customer-specific adaptation has finished? Two *forces* arise:

- The evolution of the software system should neither affect its base-line architecture nor its various design centers.
- A software system rarely completes: features are added or modified, bugs are fixed, performance is tuned, and environments are exchanged. However, it is next to impossible to foresee most changes and extensions which customers will require over the system's lifetime. It is also impossible to anticipate the evolution of its environment, such as hardware platforms and operating systems.

Solution *Plan for Growth*. Prepare the system's base-line architecture for adding, replacing and removing subsystems. Allow to extend subsystems and large components with new features, and support to change existing ones. However, protect the system's base-line architecture and the core structures of its design centers from any such modification: they should stay stable over the whole lifetime of the system. Similarly, shield clients of evolving subsystems or large components from changes, specifically if they do not depend on the evolving parts.



All architectural patterns address aspects of change and evolution. Broker and Microkernel [POSA1] provide so called 'plug and play' infrastructures: registration services allow to integrate new subsystems with the application, and also to replace and remove existing subsystems. Model-View-Controller [POSA1] supports to replace and change user interface subsystems without affecting the system's functional core. Pipes and Filters [POSA1] supports the replacement of any subsystem in a data driven processing pipeline. Layers [POSA1] enables the replacement of whole sets of functions which share a particular general responsibility, for example access services to the operating system. Presentation-Abstraction-Control [POSA1] allows to exchange all parts of a system which form a 'micro-application' of their own. Blackboard and Reflection [POSA1], finally, provide infrastructures that allow for changing almost any aspect of a system, for example, service-specific behavior, principles of subsystem interaction and collaboration, and the physical distribution of components. Use the architectural patterns when defining the system's *Architectural Vision* (47), to prepare its infrastructure for growth.

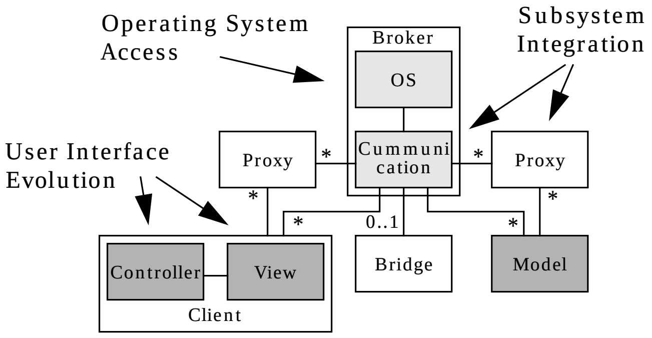
Many design patterns support adding services to *Stable Design Centers* (63) which cannot be foreseen by today. Visitor [GHJV95] provides an infrastructure that allows arbitrary and yet unknown services to traverse a given object structure, perform operations on this structure, accumulate data, and compute a result on the information and data acquired during the traversal. Extension Object [PLoP96] and Extension Interface [POSA2] define infrastructures that allow to implement service extensions to a design center as separate components. The extensions are accessible through the design center's original interface. It offers a special method to return the specific interface of an extension. Clients that do not use the new services are not affected by the design center's evolution.

The evolution of a design center's existing services is supported by all patterns that help adapting a system to customer-specific needs, as described in *Stable Design Centers* (63).

Consequences The system can grow. It can respond to any new or evolving requirement and also to changes in its environment. Yet the fundamental architecture of the system stays stable, due to the use of architectural and design patterns. They take care that evolution happens in a defined and controlled manner.

However, if for some reason the base-line architecture or a design center is affected by such a change, this will result in major cost and time effective, modifications to the whole application or the design center, respectively.

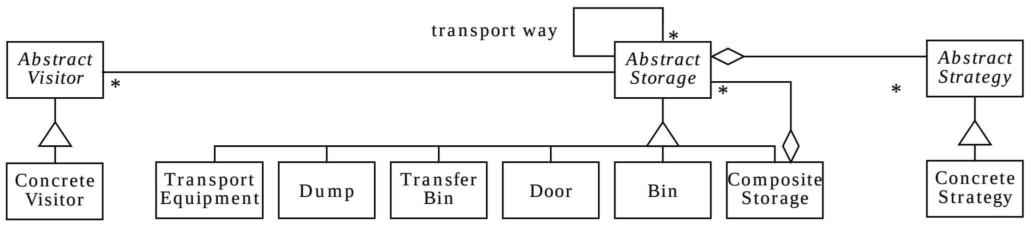
Example The base-line architecture of the warehouse management system is prepared for growth with respect to three aspects. The Broker pattern [POSA1] allows to integrate new subsystems that follow the component model defined by its concrete implementation, in case of the warehouse management system this is the CORBA [OMG98] model. Model-View-Controller [POSA1] prepares the system for user interface evolution. Layers [POSA1] keeps the whole system independent from the specifics of the underlying operating system platform.



When specifying the structure for representing physical storage in a warehouse, an important aspect we must consider is the integration of functionality which we do not know by today. On the other hand, anticipating customer requests from tomorrow is next to impossible. The problem we are facing is: can we define a structure that allows us to integrate new services on demand, without the need to change existing code? Fortunately, we do not need to modify anything in our current design in order to address this requirement. An interesting property of the Visitor pattern [GHJV95] we have applied to implement functionality that operates on the whole warehouse structure[3] is that we do not need to know the services in advance that we connect to the visited object structure—neither their name, nor

3. See the example section of *Step-wise Refinement* (53) for further details.

their semantics, interface and object structure traversal strategies. So we are set with this aspect.



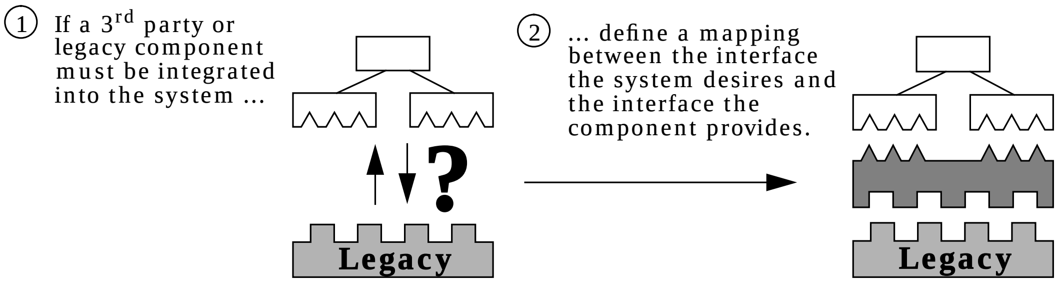
Component-Oriented Legacy Integration

Context We are specifying a system's *Architectural Vision* (47) or detailing its design by *Step-wise Refinement* (53).

Problem *How can 3rd party or legacy software be integrated into the application*? Four *forces* must be considered:

- The 3rd party or legacy artifact must integrate with the existing software architecture such that the system's domain-specific services can be executed as required and the global design principles defined by the *Architectural Vision* (47) are met.
- We cannot modify the legacy or 3rd party software. It has to be used as is. Unfortunately, interfaces of such artifacts often do not match with the view the application needs onto them.
- The 3rd party or legacy component may offer more services than needed, or less, or groups of services that are needed in different parts of the application.
- The implementation of the 3rd party or legacy software may follow different programming paradigms and may be implemented with different programming languages than the application.

Solution *Provide a component-oriented legacy integration*. Define the view the application needs onto the 3rd party or legacy artifact and map between this view and the actual interface the 3rd party or legacy software offers.


1
If a 3rd party or legacy component must be integrated into the system ...
?
Legacy
2
... define a mapping between the interface the system desires and the interface the component provides.
Legacy

First, specify the application's *desired* view onto the 3rd party or legacy component. Identify all services and collaboration protocols the system expects from that component in order to operate as required. Do not focus on whether or not the 3rd party or legacy artifact already provides these services and collaboration protocols. Rather let the specification be governed by the part of the existing design in which the 3rd party or legacy component must be integrated, and the global design principles defined by the *Architectural Vision* (47).

If the interface of the 3rd party or legacy artifact already matches with the interface the application desires, you are set. The component can be integrated as is. Otherwise map between the specified interface and the interface the 3rd party or legacy software provides. Several design patterns help with this.

The Adapter pattern [GHJV95] helps converting between the two interfaces. To the application, the adapter offers the services it needs, as specified in the first step. These, however, only delegate to the corresponding services of the 3rd party or legacy component that actually provides their implementations. The adapter also performs all bi-directional data transformations for parameters and return results as well as all conversions between programming paradigms and languages that are necessary to implement the mapping. If a service that the application requires can only be provided by several 3rd party or legacy services in combination, the adapter further implements their coordination. The adapter finally hides all services offered by the 3rd party or legacy component which the application does not need.

Provide a separate adapter for every different role the 3rd party or legacy component implements. For example, if such a component allows both to control and to monitor a telecommunication network or warehouse automation equipment, provide two adapters: one for the monitoring and one for the controlling services.

If the 3rd party or legacy software has a complex internal structure on which the application should not depend, the adapter is also a Facade [GHJV95] that shields the application from this structure. Service requests are transparently delegated to the corresponding subcomponent of the 3rd party or legacy artifact.

If the 3rd party or legacy component provides less services than required by the application, several patterns can apply. Decorator [GHJV95], which is often referred to as Wrapper, allows to attach additional features to the services the 3rd party component or legacy software provides. For example, if a legacy component for fault management only returns cryptic descriptions of the errors it detects, a decorator could add the corresponding human-readable descriptions of these errors before it returns to the client. Likewise, the decorator could remove features from these services, if the application does not need them.

Extension Interface [POSA2] and Extension Object [PLoP96], finally, support adding services to an existing component which it does not yet provide, but which are needed by the system. These services may not even be known by today. The missing or new services are implemented separately from the original component and can be linked dynamically to the system using the infrastructure the patterns provide for registering new service extensions. The new services are accessible through the original component's application-side interface. It offers a method that returns the interface of the service extension which then can be used by the application to activate the new services.

Integrate the mapping you have specified with the system's *Architectural Vision* (47) or the part of the software architecture you are detailing by *Step-wise Refinement* (53).

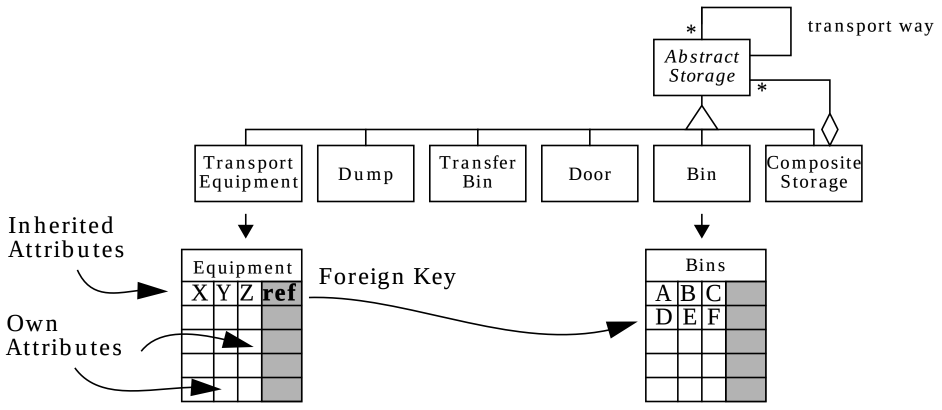
Consequences The 3rd party or legacy component is integrated into the application without being modified, but also without violating the perspective the application needs onto this component.

Example We must integrate one 3rd party component and one legacy component into our warehouse management system: the database and the warehouse automation. The database to be used is a commercial product. Unfortunately, it is a relational database—in contrast to the application, which design an implementation follows the object-oriented paradigm. The decision in favor for this product is purely based on business factors rather than on technological aspects. From the previous generation of warehouse management applications we have to reuse the subsystem that implements the controlling and monitoring functionality for the warehouse automation equipment. It is implemented in C rather than C++, the programming language we use to code the application. However, it allows full monitoring and control of the automation equipment and has been bug-free for years.

Integrating the controlling and monitoring subsystem is relatively straight forward. We introduce two adapters, one for controlling and one for monitoring the automation equipment. On the application side each adapter offers an object-oriented interface to the subsystem's functionality. Every service in this interface represents a service of equal semantics in the legacy subsystem. The adapters convert between the corresponding interfaces, thereby mapping from the object-oriented view in C++ that the application takes to the functional view in C that is taken by the legacy subsystem, and vice versa.

Much more complicated is the mapping between the object-oriented system and the relational database. Aspects we must consider are, for example, the mapping of objects, object relationships and inheritance hierarchies to tables. Our adapter ends up as a subsystem by itself, placed between the system and the database. We design it with help of the pattern language Object/Relational Access Layers [KCR99].

When mapping the objects of our class hierarchy for representing physical storage and transport equipment to tables, we follow the Foreign Key Aggregation pattern: if an object contains a data member that denotes another object, we store the referenced object in a separate table. In the table that contains the data for the original object we maintain a foreign key for the table and row that maintains the data of the referenced object. For mapping the class hierarchy we apply the One Inheritance Path One Table pattern: objects of different kinds of storage are stored in separate tables. These do not only keep the attributes of the concrete storage we maintain, however, but also the inherited attributes from class `AbstractStorage`. The interface of our adapter encapsulates all these issues, so that the application just can store and load an object.



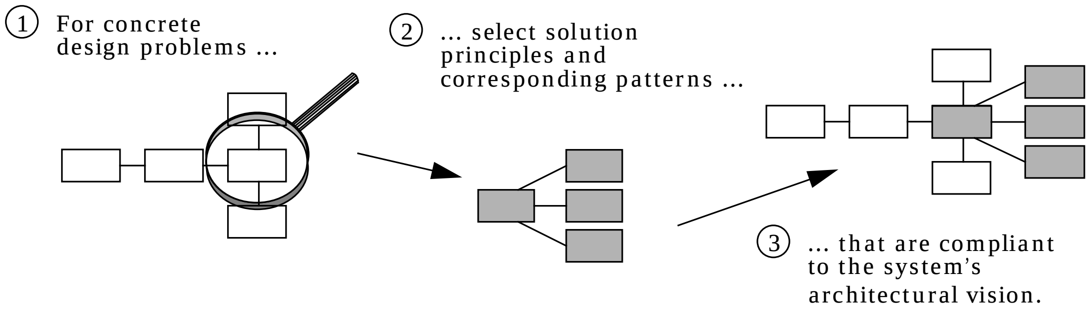
Enforce the Architectural Vision

Context A software architecture is detailed by *Step-wise Refinement* (53).

Problem *How can we support that a software architecture follows the design principles defined by its Architectural Vision* (47) *consistently?* If we do not support the application of the system's global design principles its architecture may become hard to understand and maintain: solutions to specific design problems may interpret the principles in multiple ways, or even follow completely different and contradictory design principles. The following *force* complicates this problem:

- Multiple groups may be responsible for developing the system. For example, each subsystem of the application could be designed and implemented by a different group, or person. The consistency of such independently developed designs must be ensured.

Solution *Enforce the Architectural Vision* (47). Ensure that the concrete design and coding principles selected for resolving specific design and programming problems are compliant with the global design principles defined by the system's architectural vision. For implementing the solutions to these problems select only design patterns and idioms which follow these specific principles.


1 For concrete design problems ...
2 ... select solution principles and corresponding patterns ...
3 ... that are compliant to the system's architectural vision.

For example, if separation of concerns is one of the global design principles and the design problem at hand requires to support the exchangeability of a component's implementation, the concrete design principle to apply is separation of interface and implementation. The design pattern that supports implementing this principle is Bridge [GHJV95]

Solve related design problems similarly—by using the same or related design principles and patterns. Implement their solutions according to the same coding principles and idioms. If such problems arise at the same level of abstraction, detail their solutions to the same level of granularity. Note, that this does *not* mean to use identical designs or implementations for resolving related problems. Each concrete design and implementation must meet the specific needs of the problem to be resolved. Using the same principles and patterns means to base solutions of related problems on the same core ideas. Their actual implementation will likely vary.

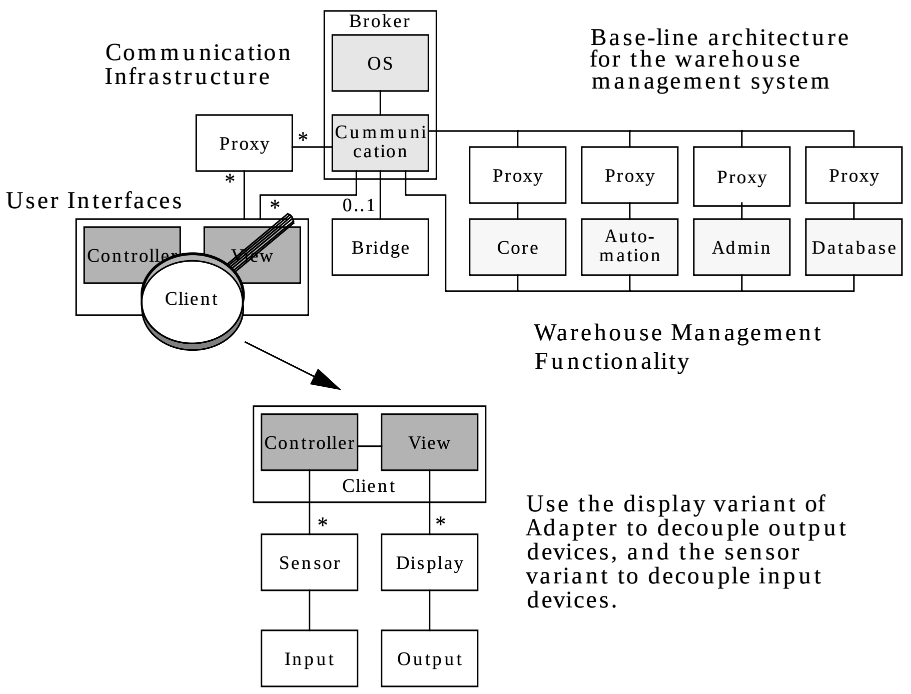
Let the software architect(s) of the system supervise and coordinate the activities of the design and development groups. This ensures that the groups interpret and 'refine' the global design principles consistently and that related problems resolved by different groups follow the same design principles, patterns, coding principles, and idioms. Apply the organizational patterns by Jim Coplien in [PLoP94] to define the appropriate project organization for this.

Sometimes it may be impossible, however, to resolve a problem with principles which are consistent to the architectural vision. For example, when its solution requires to address an additional force which does not need to be considered elsewhere in the system. Even a refactoring of the existing design, as proposed by the pattern *Step-wise Refinement* (53), may not help. In such cases, explicitly document the deviation from the 'standard' design and coding principles. By this the 'special purpose' design becomes visible. Developers who are maintaining the system at a later point in time thus will know what was done and why the design does not follow the general principles that apply for the rest of the system.

Consequences Parts of the architecture that deal with related system aspects expose similar finer-grained design structures and implementations. This makes a system easier to understand and change. Its architecture becomes balanced and coherent. There is a common vision behind the solutions to design problems of a similar kind. Exceptions from this rule are marked explicitly and a rational for each is given.

Example The global design principles selected for our warehouse management system[4] include that the application's functional core is to be separated from human-computer interaction. The base-line architecture reflects this principle by applying the Model-View-Controller pattern [POSA1].

To keep the system even more flexible we also want the user interface implementation to be independent of the concrete input and output devices used, such as mouse, scanners, buttons on hand-held terminals, displays, and printers. A close analysis reveals that the Adapter pattern [GHJV95], specifically its Sensor and Display variants, help with resolving this problem. In short, a sensor provides an unidirectional adaptation from a specific to a general interface. Likewise a display provides an unidirectional adaptation from a general to a specific interface.


Broker
OS
Cummuni cation
Communication Infrastructure
Base-line architecture for the warehouse management system
Proxy
*
*
0..1
User Interfaces
Controller
View
Client
Bridge
Proxy
Proxy
Proxy
Proxy
Core
Auto-mation
Admin
Database
Warehouse Management Functionality
Controller
View
Client
*
*
Sensor
Display
Input
Output
Use the display variant of Adapter to decouple output devices, and the sensor variant to decouple input devices.

4. See the example section of pattern *Architectural Vision* (47).

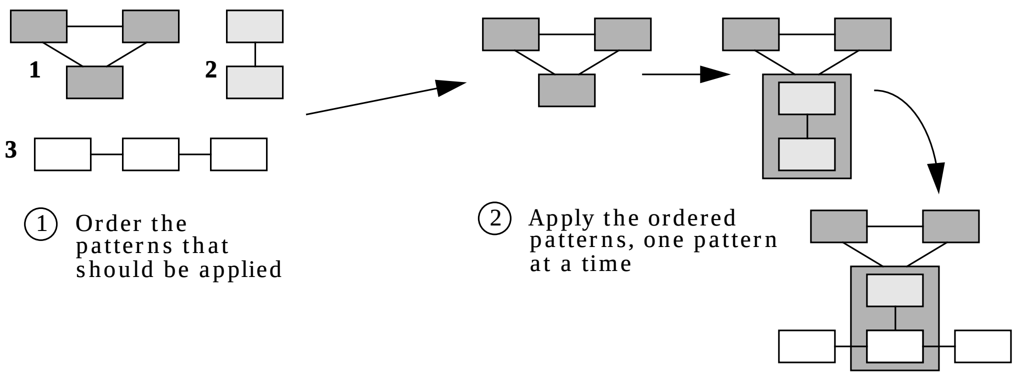
One Pattern at a Time

Context We are specifying an *Architectural Vision* (47) or—by *Step-wise Refinement* (53)—the details of a given software architecture. Several patterns were selected to help with this specification.

Problem *How can we combine a set of patterns* which together should specify a particular part of a software architecture? Three *forces* arise:

- The part of the architecture we are specifying may play multiple functional roles in the system. Each such role must be correctly defined, but some roles are more important than others. The same holds for the non-functional properties that must be met.
- The design should reflect the essence of the patterns that we apply.
- The more patterns we apply at the same time, the more aspects regarding their optimal implementation, combination, and integration arise. Dealing with many of such aspects may, however, may distract our attention from resolving the original problem.

Solution *Apply one pattern at a time* [Ale79]. Begin with the pattern which addresses the most important design aspect that must be considered. Incrementally refine and complete the initial design by applying the patterns that address the less important aspects. For all patterns that are applied, their *Design Integration Precedes Implementation* (81).

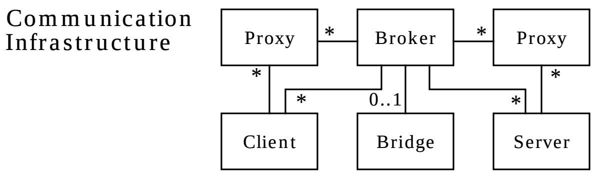

1
2
3
1 Order the patterns that should be applied
2 Apply the ordered patterns, one pattern at a time

When specifying a base-line architecture, infrastructural aspects are generally more important than functional aspects. Even the best functional design will not work well within an insufficiently specified infrastructure, such as for distributed object computing, database access, and human-computer interaction. When refining a given design, on the other hand, functional aspects are often more important than non-functional aspects, since components and relationships that do not provide the required roles and behavior cannot be used.

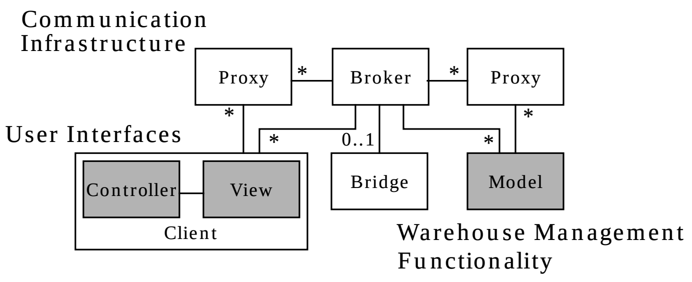
Some patterns already specify which other patterns help refining and completing them, as well as in which order and how the referenced patterns should be applied. Examples include the patterns in [POSA1] and [POSA2]. For instance, the Model-View-Controller pattern [POSA1] suggests to use the Observer pattern [GHJV95] for specifying the general relationships and collaborations between the model, the views, and the controllers in an MVC architecture. Follow such guidelines wherever a referenced pattern helps specifying an aspect that must also be considered in the software architecture under construction.

Consequences By applying one pattern at a time with descending order of importance, the resulting design will most likely provide the functional and non-functional properties that are required. Also, the design will reflect the essence of the patterns we apply. We can follow their implementation guidelines more easily, since aspects of other, not yet applied patterns, do not need to be considered. All patterns together form a coherent structure which effectively specifies the system part at hand.

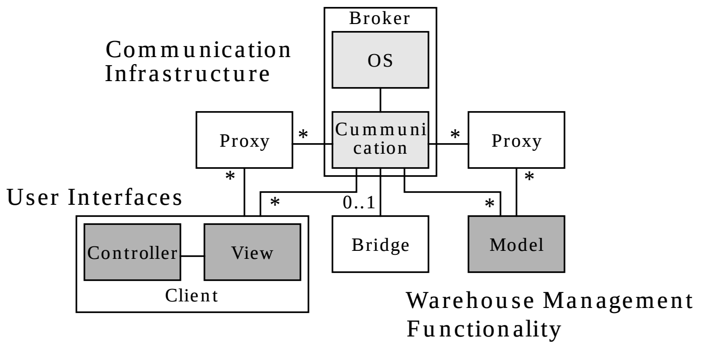
Example When specifying the *Architectural Vision* (47) for the warehouse management system we selected three patterns: Broker, Model-View-Controller, and Layers [POSA1]. Broker addresses the most important aspect of the system, namely how to provide a communication infrastructure for distributed systems. We thus applied Broker first.



Into this infrastructure we embedded Model-View-Controller. User interface clients are split into a view and controller part, the model of MVC is specified as a server.



Layers was applied thereafter to further separate high-level communication services from the low-level network programming interfaces.



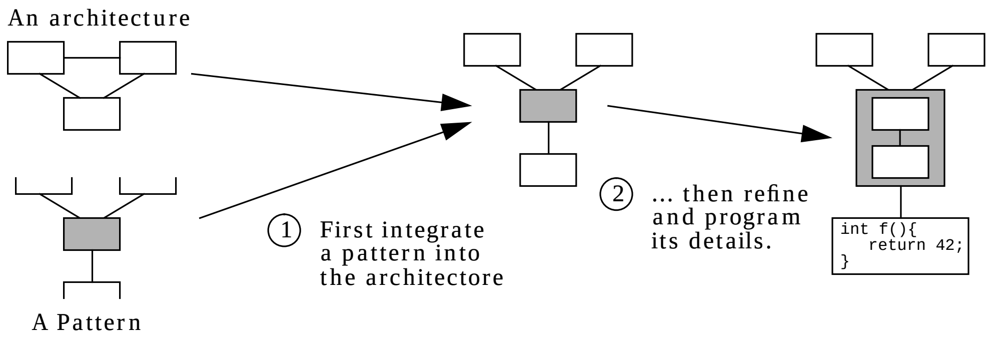
Design Integration Precedes Implementation

Context We are defining a specific part of a software architecture with patterns by applying *One Pattern at a Time* (78).

Problem *How can a pattern be implemented such that it optimally resolves a particular design problem in the context of a given software architecture?* Two *forces* arise:

- Applying a pattern means integrating new roles into a software architecture. Distinctions are made where no distinctions were before. Yet we want to preserve the general structure and behavior of the design part we are refining.
- Specifying the details of a pattern before integrating it with the existing software architecture may result in an inappropriate pattern implementation. You may create a design pearl, which is perfect and beautiful by itself, but which is not usable for the application under development. On the other hand, the essential structure and behavior of the pattern should not be violated.

Solution *Integrate the coarse-grained structure of the pattern into the existing software architecture before unfolding and coding its details.*


An architecture
A Pattern
1 First integrate a pattern into the architectore
2 ... then refine and program its details.
int f(){
   return 42;
}

First, tailor and adapt the pattern’s general structure such that it fits into the existing software architecture. To define *where* the pattern is to be integrated, identify which components of the software architecture must collaborate with which of the pattern’s participants. To specify *how* the pattern must be integrated, assign the roles which the pattern introduces to components and connect collaboration partners by appropriate relationships. ‘*Merge Similar Responsibilities*

(85), if components of the existing architecture must or should provide roles that the pattern introduces. Define new components only for those roles which cannot be integrated with existing components.

Then implement each pattern role and their collaborations to resolve the original design problem. Specify the required functional aspects and implement them according to the pattern's implementation guidelines.

Aspects that are too complex to be programmed directly should be further refined. For example, if a component plays multiple complex roles in the system, each role could be encapsulated in a separate subcomponent. Commonly used decomposition guidelines can be found in [Boo94], [Coad95], [Fow97] and [Kar95], and often also in the pattern's implementation guidelines. Follow the process of *Piecemeal Growth* (43) to unfold such aspects appropriately.

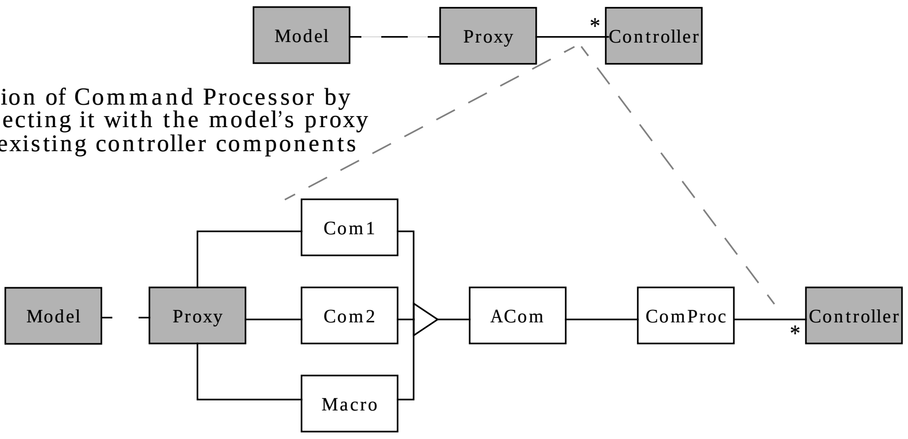
Aspects that can be coded in a straight forward fashion—according to the decomposition guidelines you are using—should not be further refined. For example, the `set` and `get` methods offered by a value component that maintains a single non-hierarchical data structure can often be implemented with just a few lines of code each. In general, an overly fine-grained decomposition will only result in a complex pattern implementation. The more that a component is split into subcomponents, for example, the harder it becomes to understand, implement and maintain. Furthermore, you may introduce performance penalties, due to additional communication and coordination between the subcomponents.

Consequences A pattern implementation is governed by the existing architecture with respect to its coarse-grained structure, and by the requirements of the design problem at hand concerning its detailed specification and concrete behavior. Yet the pattern's essential solution idea is still present in the software architecture, since the integration process always obeys to the implementation guidelines which the pattern defines.

Example In the warehouse management system we apply the Model-View-Controller pattern [POSA1] to separate human-computer interaction from the system's functional core. When refining the controller part of MVC, we specify concrete controllers for collecting user input and triggering application services, such as menu items and dialog boxes for initiating transports. To connect these controllers with the functional core, we refine the controller-model relationship with help of the Command Processor pattern [POSA1].

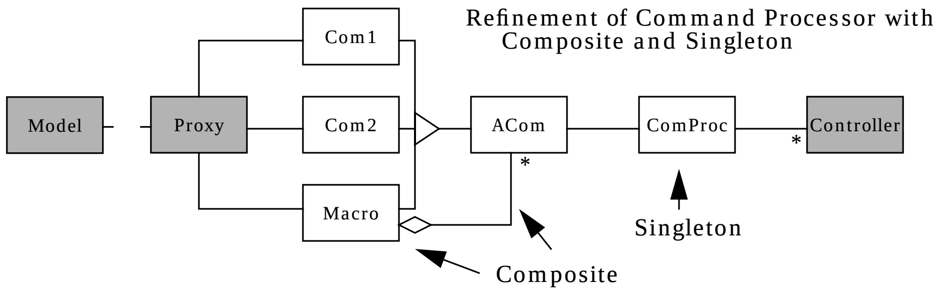
The implementation of the Command Processor pattern suggests us to specify controller components that receive user input, create so-called command components, and pass these command components to a command processor component for execution. However, we do not specify completely new controllers when implementing the Command Processor pattern. Rather we extend the controllers that were defined when applying Model-View-Controller with the two latter responsibilities from above.

New kinds of components are the command processor and the commands. Their detailed specification, on the other hand, is constraint by specific functional requirements. For example, when providing undo/redo and logging services, we must define appropriate interfaces for both the command components and the command processor.



Integration of Command Processor by connecting it with the model's proxy and existing controller components

After integrating the Command Processor pattern with the existing design we can proceed with specifying its details. Two patterns help with this. Composite [GHJV95] can be used for providing macro commands. Singleton [GHJV95] ensures that the command processor can be instantiated only once. These patterns as well as the relationships between the introduced components can—with one exception—be programmed straight forward and without further refinement. The exception is the relationship between the macro command and the abstract command components. Here we have to provide an infrastructure for maintaining the contents of a macro command. The Iterator pattern [GHJV95] helps with this refinement.



Another example for a more detailed and a straight forward implementation of a pattern is the use of Observer [GHJV95] when refining the model-view relationship in a Model-View-Controller structure. The model plays the role of the subject, the views the role of observers. The concrete implementation of this structure must be able to handle multiple views effectively. For maintaining these multiple views, we refine the model with a registry component. To notify views about changes to the model, we apply the Iterator pattern [GHJV95]: an iterator iterates over all registered views to let the change propagation mechanism call their `update` method.

For reasons of robustness, we need to notify the iterator about every unsubscription of a view from the model's view registry. Otherwise, the iterator might try to notify a view that does not exist. To resolve this problem, we can apply Observer a second time: with the view registry as subject and the iterator as observer. However, this time we can implement the pattern straight forward. An analysis reveals that at any given point in time not more than about 50 views are registered with the model. We thus only need one iterator to notify views about changes in the model. There is no need for multiple iterators to increase system performance by notifying several views simultaneously. Therefore, the view registry maintains a defined hard-coded reference to the singleton iterator and notifies it directly whenever a view registered or unregistered with the model.

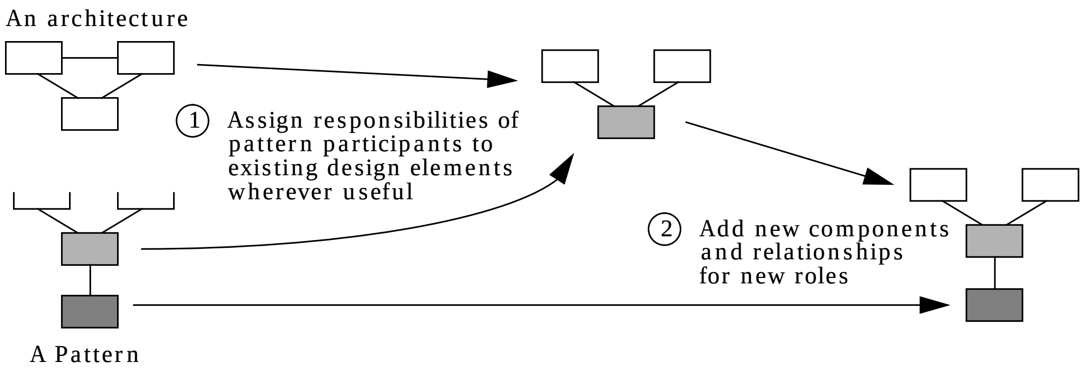
Merge Similar Responsibilities

Context *Design Integration Precedes Implementation* (81) of every pattern we apply.

Problem *How can we optimally integrate a pattern* into a partially existing software architecture? Five *forces* must be considered:

- The existing software architecture should govern the pattern's implementation.
- Components and relationships of the software architecture may already provide responsibilities that are defined by the pattern.
- Responsibilities of pattern participants may complement or complete responsibilities of existing components and relationships.
- Adding new components or relationships to the design likely increases the structural complexity of the software architecture.
- Assigning overly many and distinct responsibilities to a component or relationship will likely break the principle of separation of concerns. The resulting software architecture becomes hard to understand, change, and extend.

Solution *Assign responsibilities of pattern participants to elements of the existing software architecture* wherever it is useful and simplifies the system's structure and complexity.


An architecture
1 Assign responsibilities of pattern participants to existing design elements wherever useful
2 Add new components and relationships for new roles
A Pattern

If a design element of the structure into which we are integrating the pattern must, or should, play the role of a specific pattern participant, three situations are possible:

- The design element already provides all responsibilities of the pattern participant. In this case we are set. Otherwise we would implement the same responsibilities twice.
- The design element provides parts of the pattern participant's responsibilities. In this situation extend and complete the design element's specification with the missing parts. Otherwise we separate aspects that belong together.
- The design element does not provide any responsibility of the pattern participant. In this case, assign the responsibilities to the design element. Do not introduce a new component or relationship. Otherwise we increase the structural complexity of the system.

If responsibilities of a pattern participant and an existing design element complement each other, it may be useful to attach the pattern participant's responsibilities to the design element. However, this depends on the concrete context of the system.

On the one hand, combining roles can avoid communication overhead between otherwise strongly coupled components—and thus performance penalties. For example, when separating an interface from its possible implementations with the Bridge pattern [GHJV95] it may be useful to also attach authorization services to the interface, as defined by the Protection Proxy pattern [POSA1].

On the other hand, overloading a component with responsibilities may introduce inefficiency. If such a component plays distinct roles in several different contexts, it may become a performance bottleneck. For example, a peer component in a distributed system may play both the role of a client and a server. Thus, the peer has three responsibilities: processing services, initiating connections to remote servers if it plays the client role, and accepting connection requests from remote client components if it plays the server role. Keeping the three responsibilities separate from each other as suggested by the Acceptor-Connector pattern [POSA2], is more effective than combining them in a single component.

However, do not assign responsibilities of a pattern participant to an existing design element which—according to the pattern's implementation guidelines—must be implemented separately. Specify these as components or relationships of their own and integrate them with the existing software architecture. Otherwise we would break the principle of separation of concerns.

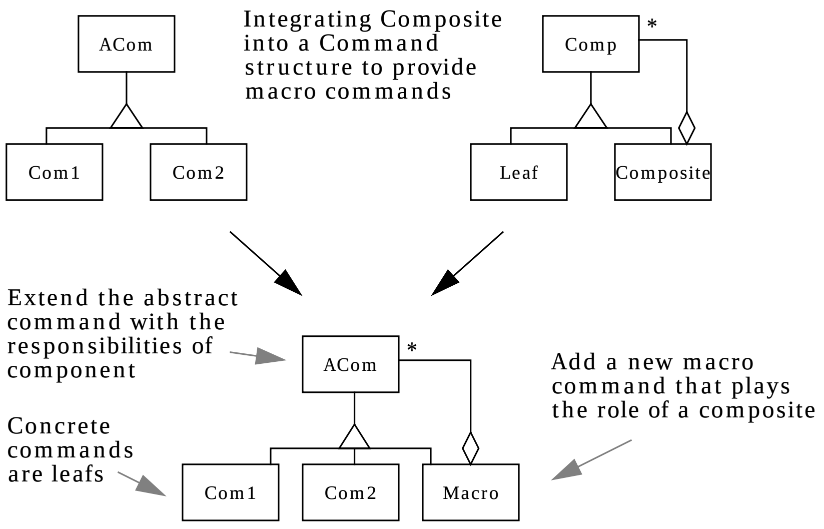
If a component plays multiple roles in the system which can be summarized by a single implementation, use the Extension Interface pattern [POSA2] to define role-specific views onto the component. For example, a component that maintains data which one component is allowed to modify, and all others only to read, could define two interfaces: one read/write interface and one read-only interface.

Consequences In general, assigning responsibilities of pattern participants to existing design elements helps reducing the structural complexity of a software architecture. It also ensures that the implementation of the pattern is tailored to the needs of the system, and not vice versa.

Example In the user interface subsystem of the warehouse management application we encapsulate service requests as objects by applying the Command pattern [GHJV95]. However, this supports the encapsulation of 'atomic' requests only. For providing macro commands, such as for initiating an item transport with an intermediate stop at the quality assurance, we need additional infrastructure. The Composite pattern [GHJV95] helps with solving this problem.

The responsibilities of the abstract command component in the Command pattern and those of the component component in the Composite pattern complement each other. Clients would not need to distinguish between macro commands and 'atomic' commands—they would just execute a command. Thus, we attach the responsibilities of the component component to the abstract command component. The concrete commands in the Command pattern already provide the roles of leafs in the Composite pattern. We do not need to do anything here.

In our design context, the composite component of the Composite pattern plays the role of a macro command. This responsibility, however, must be kept separate from atomic commands. Otherwise we lose flexibility in composing macro commands. We therefore implement a separate macro command component and integrate it into the design according to the guidelines of the Composite pattern.



Integrating Composite into a Command structure to provide macro commands
ACom
Com1
Com2
Comp
*
Leaf
Composite
Extend the abstract command with the responsibilities of component
Concrete commands are leafs
ACom
*
Com1
Com2
Macro
Add a new macro command that plays the role of a composite

Another example of merging similar responsibilities is the integration of transport equipment classes into the composite class hierarchy for representing physical storages. See *Refactor Instead of Large Lump Design* (59) for details.

3.3 Discussion

Now that we have presented the pattern language for building software systems with patterns in detail, we can step back to reflect upon its goals as well as the pre-conditions under which it works best.

The overall objective of the pattern language is, as we have motivated, to guide the use of patterns from pattern catalogs and pattern systems. From a technical perspective this means that we want to support developers in defining architectures which meet the needs of the systems they are building. In particular, that patterns help them to create the most effective solutions for the design problems at hand, under the constraints that must be considered. From a human perspective the intent of the pattern language for pattern-based software development is to enable developers to understand and control the process of building systems with patterns, to be creative when constructing these systems, and to feel habitable in their design and code.

The language works best under the following three conditions:

- There is an evolutionary development process in place that allows to adjust the design and implementation of the system whenever necessary; rather than a waterfall-like process, in which the system is constructed in a strictly top-down fashion.
- A software architect or a team of architects is in control and charge of the system's design.
- Specific patterns are selected according to the guidelines given in [POSA1].

In the following, we discuss the goals and pre-conditions in depth.

Technical and Human Aspects

A technical aspect—with respect to using patterns—is related either to a specific design goal, such as keeping a design flexible, or to the implementation of a given pattern. In the first case, the pattern language lists concrete kinds of patterns for software architecture that help with achieving the goal. For example, that the Broker architectural pattern [POSA1] defines a flexible infrastructure for distributed systems. In the latter the language provides appropriate implementation guidelines. For example, that a pattern should be integrated with the existing design first before it is implemented in full detail. Seven patterns of the pattern language address such technical aspects:

- *Architectural Vision* (47) tells us what kind of patterns help best with defining a system's basic structure and how these patterns are to be used.
- *Stable Design Center* (63) names specific patterns for software architecture which support us in keeping a design stable and flexible at the same time.
- *Plan for Growth* (67) lists several architectural and design patterns that enable a software architecture to grow and evolve over time.
- *Component-Oriented Legacy Integration* (71) references concrete design patterns which help integrating an existing software artifact into a new application, specifically its software architecture.
- *One Pattern at a Time* (78), *Design Integration Precedes Implementation* (81) and *Merge Similar Responsibilities* (85) complement pattern-specific implementation guidelines with aspects of their integration into an existing design structure.

Human issues deal, in general, with specific needs of developers when constructing software systems, for example, how they can be supported in tackling a design problem effectively, but without being swamped with overly many aspects that impact its solution.

Human aspects are very important for a software development project, more than they tend to be on a first look. The reason for this is simple. Software was built by humans in the past, it is built by humans today, and it still will be built by humans in the future—despite the numerous attempts to automate software construction. Thus, if we do not address the many human aspects that arise in a software development project, we neglect an important success factor.

The pattern language for pattern-based software development addresses several human aspects that are specifically related to using patterns for software architecture:

- *One Pattern at a Time* (78) supports developers in keeping their focus on the original design problem when solving it with patterns, rather than on issues of handling the details of their combination and integration to more complex structures.

- Keeping a software architecture understandable is supported by *Merge Similar Responsibilities* (85) and *Enforce the Architectural Vision* (75).

Most human issues in software development, however, are independent of the use of patterns. 'Applying Patterns' addresses these in seven of its patterns:

- *Piecemeal Growth* (43), *Step-wise Refinement* (53) and *Refactor Instead of Large Lump Design* (59) address, together, four general human aspects of software design. In particular, that humans usually cannot define the most optimal design in one pass, that they can handle only a limited number of design issues simultaneously, that they may forget to address important design aspects, and that unforeseen design problems may arise.
- *Architectural Vision* (47) addresses the aspect of communication: what are the key design ideas and concepts of the system.
- *Stable Design Center* (63) and *Plan for Growth* (67) address the fact that in the beginning of a development project humans, in this case the customers of the system, often have only vague ideas of what specific requirements the system must fulfil. During development, and even more while the system is in use, they tend to come up with 'add-this-feature-immediately' requests.
- *Component-Oriented Legacy Integration* (71) deals with another human factor: protecting existing investment into education and software artifacts. In the first place this sound like a business or economic factor. On the other hand, it has a strong human aspect too. The more people are skilled in using a certain technique, for example a relational database, the less they tend to be open for moving towards a new technique, such as using an object-oriented database instead. When specifying a software architecture by using techniques that do not match with what people know, such as object technology and relational databases, *Component-Oriented Legacy Integration* (71) helps to them to walk on safe ground.

Within our language both technical and human issues of constructing software systems with patterns are tightly interwoven. For example, *Architectural Vision* (47) addresses how to define a fundamental system structure *and* how to support communication of key design concepts. The goal of *Merge Similar Responsibilities* (85) is to avoid structural complexity *and* to help with understanding the system.

The tight interconnection between human and technical aspects is not accidental. Most activities in real-world software development expose both a technical and a human side. Only if we consider this inherent coupling explicitly we can achieve the original goal: constructing a software architecture that meets its functional and non-functional requirements, and in which architects, designers, and programmers feel habitable.

Integration With Methods and Processes

The main pre-condition for our pattern language for pattern-based software development is that we use an evolutionary software development process, in which we can adjust the given architecture whenever necessary. That our language does not work otherwise, for example when using a waterfall-like process model, is obvious. Neither *Refactor Instead of Large Lump Design* (59) for itself, nor its interplay with *Step-wise Refinement* (53) would be applicable. Both patterns, however, describe fundamental aspects of the process of piecemeal growth.

But how do the patterns of the language integrate with existing evolutionary process models, such as the fountain approach [HE95]? A close look onto such models reveals that they only define between which phases of software development we can jump back and forth. They do not specify in detail how this step should look like, and how bottom-up, top-down, refactoring and growth aspects are interwoven. This, however, is subject of our patterns for pattern-based software development. From a pragmatic view the languages defines how evolution happens. It does not re-define any concrete process step and the connections between them.

There is only one add-on that our pattern language for pattern-based software development suggests to existing process models. The language stresses the importance of specifying a base-line architecture for the system right at the beginning of its development, directly after the requirement analysis. This is manifested by the pattern *Architectural Vision* (47). The base-line architecture is defined even before the detailed specification of the domain model, which is usually the first development activity in most popular processes. With respect to integrating the pattern language with these processes, we can simply add an architecture specification phase right after the requirement analysis phase. After that, we proceed with following the original process, for example with specifying the domain specific parts. The patterns are then used to support the steps as defined by that process, and integrating the results into the *Architectural Vision* (47).

With this discussion another question arises. Does the pattern language for pattern-based software development define a new software development process or method? A process or method that may replace exiting approaches, such as Booch [Boo94], Coad/Yourdon [CY91], Object Modeling Technique [RBPEL91] or even the Unified Software Development Process that completes the UML [JBR98]. This is, however, not the case.

Instead, the pattern language complements existing approaches with respect to using patterns for software architecture. *One Pattern at a Time* (78), *Design Integration Precedes Implementation* (81), *Merge Similar Responsibilities* (85) and *Enforce the Architectural Vision* (75) define specific activities and principles for selecting, integrating, refining, and coding patterns for software architecture. *Architectural Vision* (47), *Step-wise Refinement* (53) and *Refactor Instead of Large Lump Design* (59) specify how to use patterns within general design activities that are defined by existing methods. The same holds for *Stable Design Center* (63), *Plan for Growth* (67), and *Component-Oriented Legacy Integration* (71) which define a general design goals, but under consideration of using patterns to achieve it. *Piecemeal Growth* (43), finally, details the idea of evolutionary development, as we discussed above.

In summary, our patterns for pattern-based software development help with using existing process and methods effectively when constructing systems with patterns.

The Need for a Software Architect

The second pre-condition for ‘Applying Patterns’ is that a software architect or a team of architects is in control and charge of the system’s design.

This requirement is reflected most obviously in *Architectural Vision* (47). An architectural vision usually cannot be defined by all developers of the system in a committee-like procedure. Rather it has to be created by a single person from the project team, or by a small group of key persons. All who are defining the architectural vision must be experienced developers. They must have an overview of the system as a whole, and its specific needs and requirements. They must define a fundamental design structure for the system and must communicate this structure to those who develop specific parts of it. And finally, they must integrate individually developed subsystems or parts to a coherent whole again. In other words, persons who develop an architectural vision need to be software architects.

Other patterns in our language also define activities that call for a software architect. For example, *Enforce the Architectural Vision* (75). If problems that are similar to each other arise in different parts of the system, and if these parts are implemented by different developers, the developers must agree on the same mechanisms and patterns to resolve the problems. It is the responsibility of a software architect to establish, supervise, and coach this coordination.

The pattern language for pattern-based software development is certainly useful even when there is no software architect in place. However, the larger that a software development team is, and the more that work is distributed among developers, the greater is the need for a software architect in order to use it successfully. How the 'architect role' can be defined and integrated into a project organization is beyond the scope of the pattern language for pattern-based software development. Jim Coplien's organizational patterns [PLoP94], however, address this aspect extensively.

Selecting Patterns

Several patterns of the pattern language require to selection of concrete analysis, architectural, design, and programming patterns that help with resolving the design problem at hand. However, we only reference the general pattern selection guidelines described in the first volume of the *Pattern-Oriented Software Architecture* [POSA1] series rather then specifying this process in detail. We thus want to give a short overview of the selection guidelines we have defined. They are building upon the pattern system concept we have developed [POSA1]. For details, please see their original specification [POSA1]. There are seven pattern selection guidelines:

1 *Specify the problem*. To be able to find a pattern that helps you solve a concrete problem, you must first specify the problem precisely: what is the general problem, and what are its forces?

2 *Select the pattern category* that corresponds to the design activity you are performing.

3 *Select the problem category* that corresponds to the general nature of the design problem.

4 *Compare the problem descriptions*. Each pattern in your selected problem category may address a particular aspect of your concrete problem, and either a single pattern or a combination of several can help to solve it.

5 *Compare benefits and liabilities*. This step investigates the consequences of applying the patterns selected so far. Pick the pattern that provides the benefits you need and whose liabilities are of least concern to you.

6 *Select the variant* that best resolves your design problem.

7 *Select an alternative problem category*. If there is no appropriate problem category, or if the selected problem category does not include patterns you can use, try to select a problem category that further generalizes your design problem.

Summary

Our pattern language for pattern-based software development helps us to use patterns for software architecture effectively for two reasons. First it considers both human and technical aspects of software development and using patterns. Second, it complements and completes existing software development processes and methods with pattern-specific steps and aspects. Thus, as concrete patterns for software architecture are a pragmatic approach to resolve concrete design problems, this pattern language provides a pragmatic approach for integrating the use of patterns with existing software engineering practise.

Credits

We like to thank Dirk Riehle for providing us with many insights and suggestions, which helped us shaping, improving, and polishing our pattern language for applying patterns. We also owe thanks to an anonymous reviewer of an earlier version of the pattern language who convinced us to use a running example from our industrial experience rather than the design of yet another MVC-based user interface.