6 Applying 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

Using patterns in a real-world project requires more than detailed guidelines for their implementation. Several issues are not addressed. For example, how to integrate a pattern into a partially existing design? Complementary to pattern-specific implementation guidelines we therefore need general guidelines for applying patterns.

6.1 Introduction

As we have learned, every well-described pattern provides information about its implementation [GHJV95] [BMRSS96]. Every pattern also provides information about its refinement and combination with other patterns. However, all these guidelines are pattern-specific. They do not support applying patterns in general, when building a real-word software system. But how to integrate a pattern into a partially existing design? What kinds of patterns should be applied in which order? How to solve problems that cannot be solved by a single pattern in isolation?

Answering such questions is important for being able to use patterns effectively. Otherwise we apply patterns, but the resulting design will likely expose unnecessary complexity. Even worth, the architecture under construction may not provide the properties we need. For example, many patterns introduce a level of indirection to solve 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.

The above example 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 independent of the design issues addressed by individual patterns.

A pattern language for software architecture is one approach for supporting the effective use of patterns (see Chapter 7, *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]. However, we neither have such a pattern language for software architecture available, nor is it likely that such a language could be provided in the near future. Today, most patterns for software architecture are organized in pattern catalogs and pattern systems [GHJV95] [BMRSS96] [PLoP94] [PLoP95] [PLoP96]; but these do not provide the support for using patterns that we need.

To apply patterns of a catalog or system effectively 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. For this reason, the kinds of guidelines we are looking for cannot be simple one- or two-line statements telling us what to do or not. We need a more structured form. What problem in applying 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 for applying patterns should be patterns by themselves: exposing a context, a problem that arises in that context, and a solution that resolves the problem. We can describe the patterns by using an appropriate pattern form.

In this chapter we present several *patterns for applying patterns*. They were mined over the past years and reflect what we, and our colleagues world-wide, have learned when using patterns to design and implement industrial software systems.

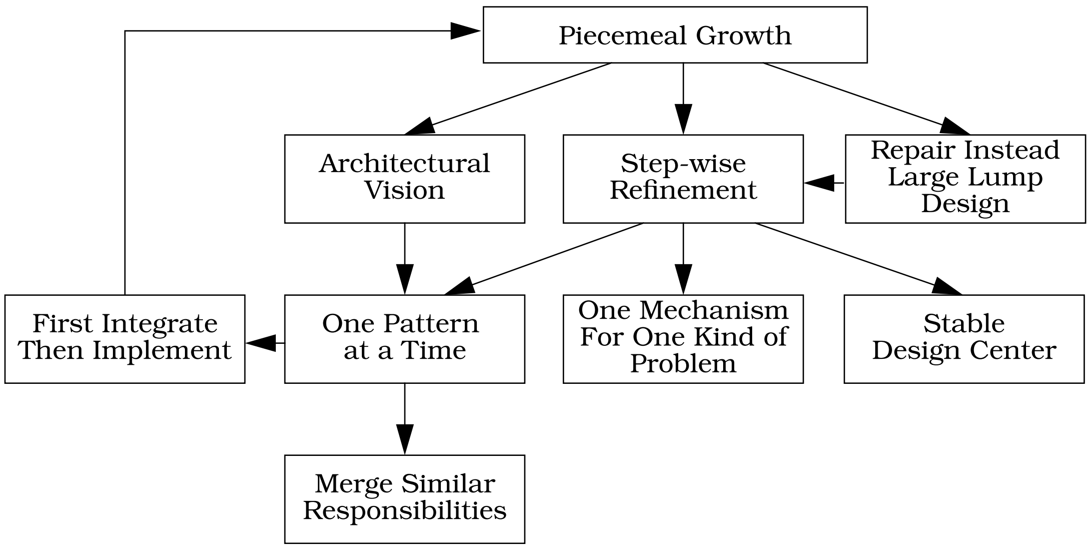
Some of our patterns for applying patterns build directly on Christopher Alexander's work [Ale79] [ANAK87]. Others emerged from the growing experience in using patterns for software development [Gab96]. Altogether they complement and complete the specific implementation guidelines that accompany well-described patterns for software architecture, for example the ones in [Cope92] [GHJV95] [BMRSS96] [PLoP94] [PLoP95] [PLoP96] and this book. For the description of our patterns for applying patterns we use Christopher Alexander's pattern form: *Name-Context-Problem-Solution-Example*.

6.2 A Pattern Language for Applying Patterns

Before describing our patterns for applying patterns in full detail we want to briefly characterize their intents, and also the global context in which they are to be seen. In total there are 9 patterns:

- *Piecemeal Growth* (61) outlines the overall process for constructing software systems with patterns.
- *Architectural Vision* (64) focuses on how to specify—with help of patterns—fundamental system structures and design principles that can serve as a starting point for the process of piecemeal growth.
- *Step-wise Refinement* (67) describes how to refine and complete a given design structure with patterns.
- *Repair Instead Large Lump Design* (69) specifies how to proceed if the optimal solution of a design problem does not integrate with the existing structure in which it must be contained.
- *Stable Design Center* (72) addresses how to specify structures that allow for extension and adaptation, but without the need to modify their key elements and abstractions.
- *One Mechanism for One Kind of Problem* (74) states how to solve similar and related design problems.
- *One Pattern at a Time* (76) defines how many patterns should be applied at once.
- *First Integrate Then Implement* (77) describes how to best codify a given pattern.
- *Merge Similar Responsibilities* (80) shows how to combine patterns with existing design structures whose participants partly provide similar or related responsibilities.

All 9 patterns strongly depend on each other. You start with the first pattern—*Piecemeal Growth* (61)—and will be directed to the patterns that apply thereafter. Every pattern makes most sense in the context of the patterns it precedes and completes. In other words, our patterns for applying patterns form a *pattern language*. The diagram below briefly illustrates the dependencies between the patterns.



The overall goal of 'Applying Patterns' is, as we have said, to support 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 'Applying Patterns' 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 [BMRSS96].

In Section 6.3, *Discussion* we will return to both the goals and preconditions of 'Applying Patterns' and discuss them in full detail. Before, we present the language itself.

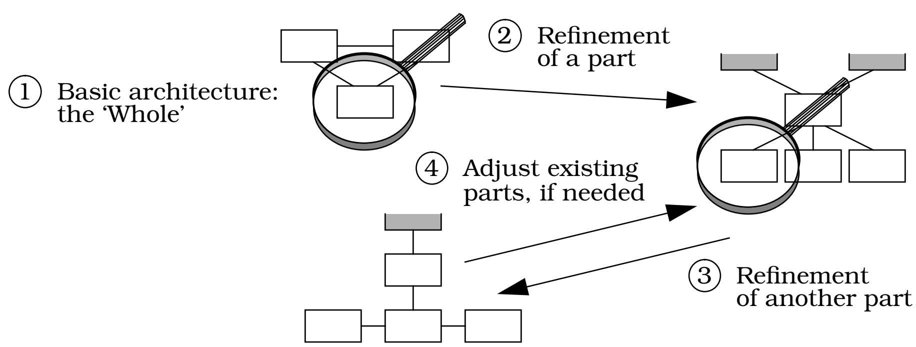
Piecemeal Growth

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, and that architects, designers, and programmers feel habitable in it [Gab96]. Three *forces* are to be balanced:

- The system should expose a defined global structure, and its design should obey to certain common principles. Both the structure and the design principles should be well-documented, so that developers can understand and follow the 'vision' of the system's architects.
- Every design problem should be appropriately solved. Dealing with each design problem in isolation, however, will result in an architecture that does not consider the needs of the system as a whole.
- Assembling the system's architecture bottom-up will likely result in a structure where many parts will be designed according to different principles that do not match. However, when following a strict top-down approach, larger structures will often be inappropriate for resolving lower-level design problems effectively.

Solution *Create the architecture of the system in a process of piecemeal growth* [Gab96]. It is a process of unfolding, in which the whole precedes the parts [Ale79]—and also governs their refinement and detailed specification. Unfolding means differentiating space: a given structure is extended with new elements, and existing elements are replaced by finer grained structures that fit into the larger structure.



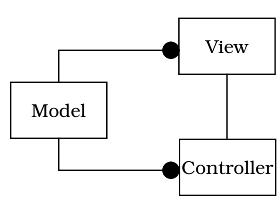
Adapted to software construction this means that we first define an *Architectural Vision* (64): what is the fundamental structure for the system and what are the general design principles for its refinement. The structure provides the basis for the process of piecemeal growth, the design principles impact the results of its subsequent steps. Furthermore, the *Architectural Vision* (64) helps with communicating the architecture of the system to developers.

Unfold this architectural skeleton by *Step-wise Refinement* (67)—abstraction level by abstraction level and separately for every design problem. By this we consider each problem's own needs, but governed by the *Architectural Vision* (64) and the larger structures in which the solutions are contained. Every part that we add to the architecture then serves as a new whole that can be unfold, until the architecture is complete.

It will likely happen, however, that the optimal resolution of a design problem does not integrate with the structure in which the solution is to be contained. When this happens, *Repair Instead Large Lump Design* (69), to adjust the existing architecture.

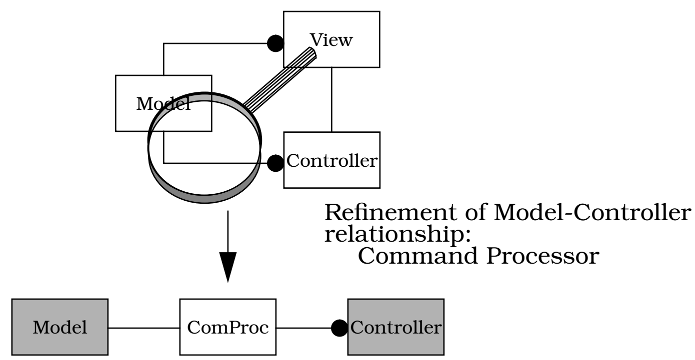
As a result, the process piecemeal growth helps creating a coherent software architecture. Due to a balanced interplay between *Step-wise Refinement* (67) and *Repair Instead Large Lump Design* (69) we look onto every design problem from three perspectives: the system view, the view of the problem itself, and the view of lower-level issues that may impact its resolution. At every level of abstraction the design of the system follows common principles, from its coarse-grained structure down to the very details of its implementation.

Example Suppose we are building an interactive software system. Its architectural vision can be defined by the Model-View-Controller pattern [BMRSS96]. Model-View-Controller introduces a structure where core functionality is separated from its control and the presentation of information.

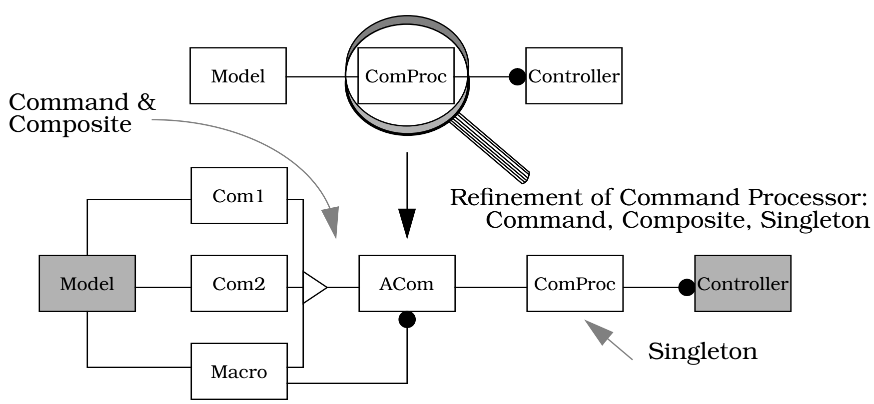


Architectural vision:
Model-View-Controller

By step-wise refinement we can unfold the details of this architectural skeleton. The general communication between components of the Model-View-Controller triad can be specified with help of the Publisher-Subscriber [BMRSS96] or Observer [GHJV95] pattern.The relationships between controller components and the model can be refined by applying the Command Processor pattern [BMRSS96], the relationships between view components and the model by using the View Handler pattern [BMRSS96]. The model itself can, for example, be structured according to the Layers pattern [BMRSS96].



We now could further refine the Command Processor structure. Singleton, Command, and Composite [GHJV95] may help with this. Other patterns apply for unfolding the details of other parts of the design.



Step by step the system will grow until its complete architecture is defined. Space gets differentiated with every pattern we apply, but always according to the principles pre-scribed by the larger structure whose parts we are refining.

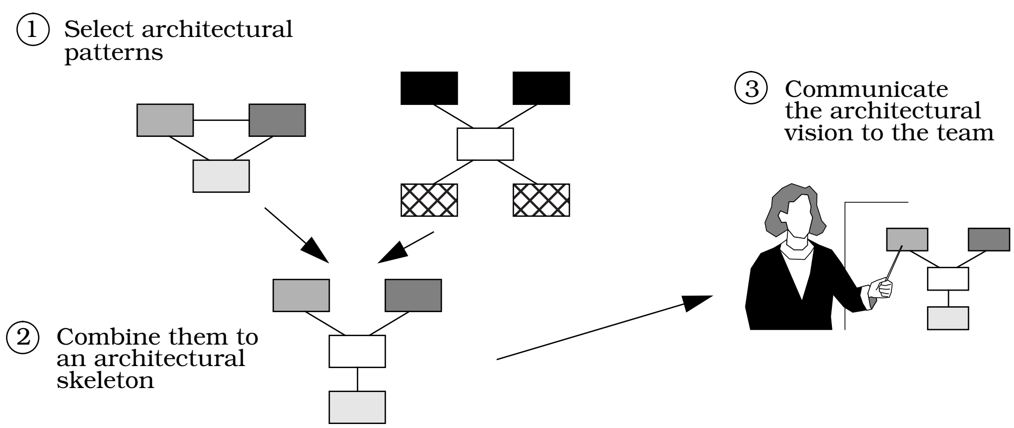
Architectural Vision

Context We are at the beginning of the process of *Piecemeal Growth* (61).

Problem *How can we define an architectural skeleton* that captures the system's fundamental structure and design principles? Five *forces* arise:

- The structure must provide a global view onto the system: all its relevant parts must be present, and every part must have defined responsibilities and relationships to other parts. The system's underlying design principles must be clearly exposed.
- The skeleton should be as simple as possible—so that essential design ideas s are not hidden within overly fine-grained structures.
- The architecture must be usable as a basis for the process of *Piecemeal Growth* (61): stable in its essence, but open for extension, refinement, and change.
- We must be able to communicate the structure—and its design principles—to avoid an architectural drift during its refinement and implementation.
- We want to base the architectural skeleton onto proven design concepts.

Solution *Create an architectural vision*: a fundamental design structure which governs the specification of the system down to its implementation.



To define the architectural vision, first collect all aspects that have an impact on the system's fundamental design. One kind of aspect are system-wide properties the application must expose, for example that it is distributed and interactive. Other factors that impact a software architecture are related to the use and integration of existing artifacts, such as legacy code and pre-fabricated components. For example, that it is possible to plug-in and access components that follow a specific component model.

If there are several of such properties and factors, bring them into an ordered sequence, with the most important property at its head and the least important property at its tail.

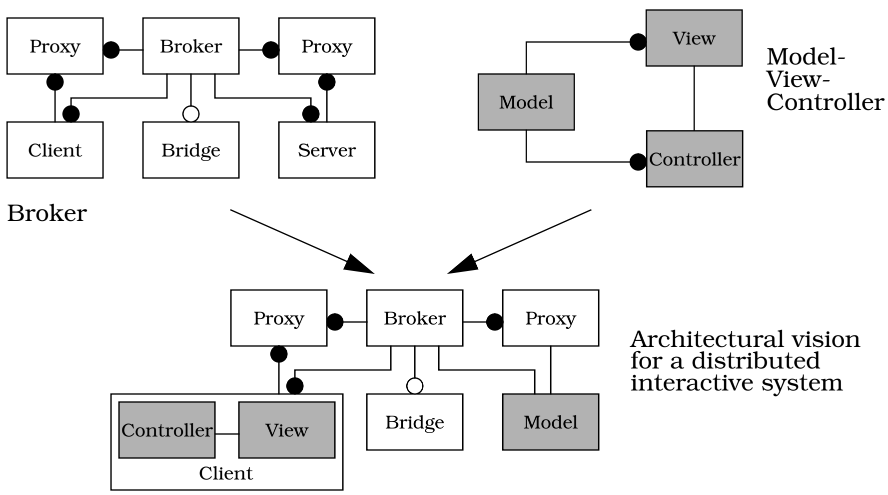
Select appropriate architectural patterns [BMRSS96] that address these properties and factors—by providing corresponding system structures and their underlying design principles. Apply the selected patterns, *One Pattern at a Time* (76), according to the order of importance of the aspects they address. By this, design structures and principles that help with implementing a more important aspect will govern the architecture of the system more than design structures and principles that expose a less important aspect.

For aspects 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 constructing such structures, like the ones presented in [Coad95].

As a result we will receive the fundamental architecture of the system: its subsystems, their responsibilities and collaborations, and also the design principles that guide its further specification and refinement.

Due to the use of patterns this architecture defines an ideal basis for the process of *Piecemeal Growth* (61). It is built with help of proven design concepts. It is stable in its overall structure, but open for extension and modification of local parts. And, the design is not overly detailed. We only focus on the system's decomposition into subsystems. The patterns that we used also help with communicating the architectural vision: what kind of structure is it, why is it exactly this structure, what are its underlying design ideas, and how is it built.

Example Suppose we want to develop a distributed and interactive system. There are, among others, two architectural patterns that help with this. The Broker pattern [BMRSS96] supports distribution, the Model-View-Controller pattern [BMRSS96] human-computer interaction. 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 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 two patterns also define the basic communication and cooperation mechanisms for components of the system. Broker defines how clients can access remote servers. Model-View-Controller specifies how user interface components interact with the functional core.

Both patterns also define the two general design principles for the system: separation of application from communication functionality and separation of the system's functional core from its user interface.

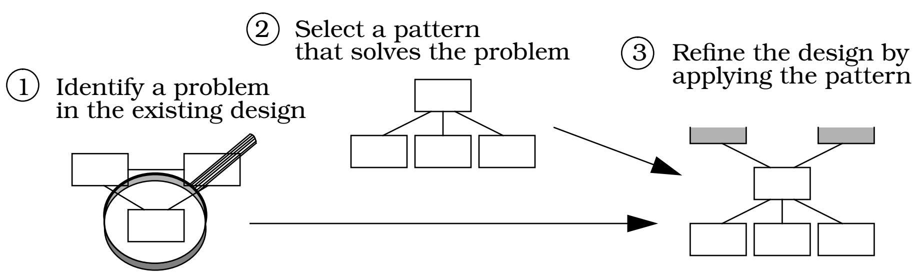
Step-wise Refinement

Context We are specifying a software architecture in a process of *Piecemeal Growth* (61).

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

- The solution must meet the problem's own needs, but also those of the whole system. It must further obey to the global design principles defined by the *Architectural Vision* (64).
- We do not want to solve similar design problems differently. Otherwise the architecture of the system would expose unnecessary complexity.
- We like to solve the problem with help of well-proven design concepts.

Solution *Resolve the design problem by step-wise refinement.* Extend the given architecture with new components and relationships, and detail existing ones. Select appropriate patterns that help with this refinement. These patterns should not just address the design problem at hand. Their properties should also match with the design principles and properties pre-scribed by the structure we are refining.



Split complex problems that cannot be solved by a single pattern into smaller subproblems that can be resolved by several patterns in combination. Always use *One Mechanism for One Kind of Problem* (74): a design problem that is similar to a problem that we tackled already should be solved similarly—with the same patterns and design principles. This helps with avoiding complex and patch-work-like design structures. Apply the selected patterns, *One Pattern at a Time* (76), to support their correct implementation.

When refining self-contained subsystems and large components, each of these should define a *Stable Design Center* (72). This supports us in specifying structures that allow for extension and adaptation, but without the need to modify their key elements and abstractions.

If no patterns are available for solving a specific 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].

Example As an example, take the implementation of an interactive version of the dice game 'Game of Greed' [Ree92]. Like many systems with human-computer interaction, its basic architecture can be defined with help of the Model-View-Controller pattern [BMRSS96].

To refine the model part of this triad, we can apply a domain-specific pattern that introduces a general structure for organizing dice games [Ree92]. The pattern specifies three kinds of participants. A game component represents the game, manages its rules, and validates throws. A game organizer component manages the current standings, the players, and the game's progress. Player components represent the players that are playing the game.

In the next step of refinement we specify the fundamental collaboration between the components of the 'dice game triad'. The analysis of the problem reveals that it is very similar to the problem of organizing the cooperation between the participants of the Model-View-Controller pattern, which we applied in a previous step. Thus we resolve the problem similarly, by applying the Publisher-Subscriber [BMRSS96] or Observer [GHJV95] pattern again.

The game component of our structure can be further refined by introducing components that represent the rules of the game, the dices, and a dice cup. The Objectifier pattern [PLoP94] helps with implementing the rules as objects. The Manager pattern [PLoP96] can be used to organize dice management: the dice cup is the manager, the dices are the managed subjects. Other patterns apply for refining the game organizer and player components.

Step by step we refine the original structure, until its details are fully unfold. However, the process of this refinement is always governed by the existing structure we are refining.

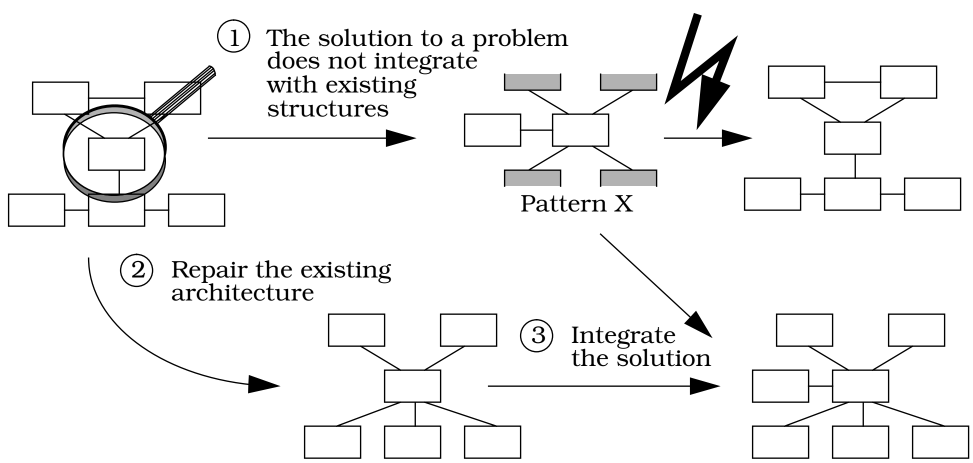
Repair Instead Large Lump Design

Context We are developing a software system in a *Piecemeal Growth* (61) fashion.

Problem *How can we deal with design problems that arise later in the design process*, but which solution impacts existing parts of the system's architecture? For example, problems due to 'add-this-feature-immediately' requests by our customers. Or, when prefabricated components we like to use do not integrate into the existing design. Resolving this conflict means to balance two *forces*:

- Aspects of the design problem that have an impact on the specification of already existing structures cannot be ignored. Rather the structures have to be specified under consideration of these aspects.
- The solution to the design problem should integrate with the existing design. It should also be consistent to the *Architectural Vision* (64) and global design principles.

Solution *'Repair' the existing architecture* [Gab96]. Do not constrain the optimal solution to the design problem at hand by inappropriate larger structures and global design principles.



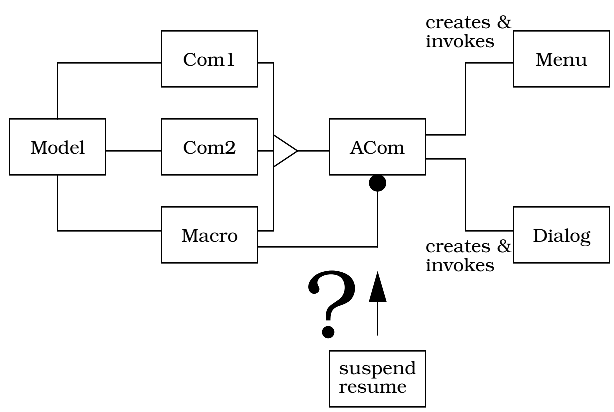
Identify all aspects of the solution to the design problem that impact the existing software architecture—specifically the part which contains the solution to this problem. Adjust this part of the design according to the identified aspects. Similarly, revise the specification of all sub-parts of this structure 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 structure.

In other words, we resolve—by *Step-wise Refinement* (67)—those 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 the needs of the system as a whole and the larger problems themselves, but also 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* (64).

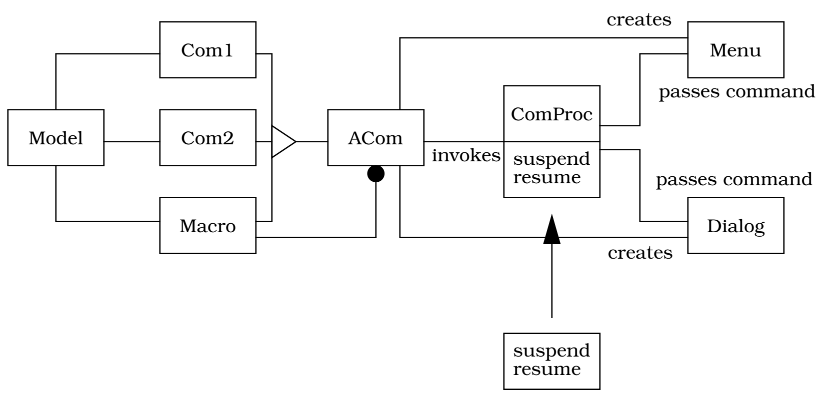
As a result, the software architecture under construction is not a product of large lump design [ASAIA75]: a fixed construct where each part, once being specified, stays untouched forever. Rather the architecture changes and grows continuously all the time, and at all levels of granularity, to stay coherent and consistent. Thus, the aspect of repair is a vital principle of the process of *Piecemeal Growth* (61).

Example Suppose we want to decouple the request for a service from the details of controlling its execution. In a first specification we apply the Command pattern [GHJV95] and encapsulate requests as objects. The user interface elements of our application, such as menu entries and dialog boxes, invoke the execution of commands.



After specifying this structure, an additional requirement comes up: suspending and resuming of command execution must be possible. The design as described above is insufficient to fulfill this requirement. We need access to all commands that are under execution—at every given point in time. The knowledge about which commands are active is, however, distributed among the invokers, rather than being centralized within a single component.

The Command Processor pattern [BMRSS96] provides the structure we need. Thus, we revise the existing design by inserting a command processor and reorganizing the relationships between invokers and commands. Invokers still create commands, but instead of triggering their execution, they pass them to the command processor. The command processor maintains the received commands and also starts and controls their execution. Functionality for suspending and resuming commands can now be implemented effectively [BMRSS96].



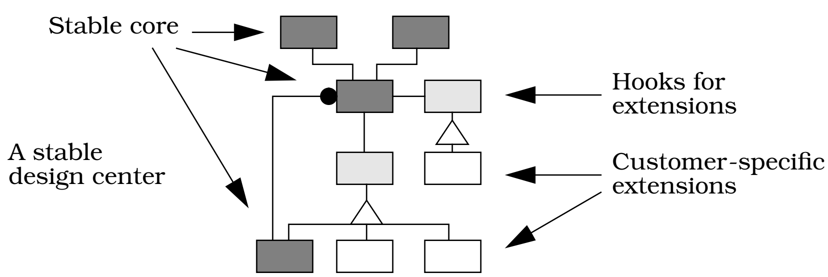
Stable Design Center

Context We are specifying a given subsystem or large component by *Step-wise Refinement* (67).

Problem *How can we ensure that the structure is coherent and stable on the one hand, but open to evolution and change on the other hand?* Three *forces* are associated with this problem:

- The more that a subsystem or component cannot be modified, the more stable and coherent it stays over the lifetime of the system. The less, however, we support the system's evolution.
- The more that a design structure is open for change, the more aspects of a system can be adapted to customer-specific requirements. On the other hand, the more likely a modification will break the coherency of the software architecture.
- Fundamental aspects of subsystems or large components should not be subject to change. Rather their structural and behavioral integrity should be preserved over the lifetime of the system. Otherwise we may endanger its general operability.

Solution *Create stable design centers* for important aspects of the system: core services that are needed in every version, and structures on which many services operate. Examples include the fundamental domain model or the system's communication infrastructure.



Capture these key aspects in a design structure which essential parts cannot be changed: its general responsibility, the 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. Consider that the design center must fit into every context of the system in which the captured aspects play a role.

Prepare those parts of the design center for evolution which—over the lifetime of the system—are likely to be modified or adapted to customer-specific requirements. Use appropriate design patterns for this, if possible. For example, to support filtering existing behavior, or adding new behavior, we can apply the Decorator pattern [GHJV95]. Or, to adapt a general behavior to a specific one, we can apply the Template Method pattern and the Strategy pattern [GHJV95].

Provide the design center with hooks that allow to attach future extensions that cannot be foreseen by today. Several patterns help with this, such as Visitor [GHJV95] and Facet [PLoP96].

Avoid direct dependencies between clients and the internal structure of the design center. The Facade pattern [GHJV95], for example, supports a defined access to services of a center, but without exposing its internal structure. Abstract Factory and Builder [GHJV95] free clients from details of how to instantiate the design center.

As a result we receive design that is stable over the lifetime of the system—from the perspective of its clients. 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 evolution. Details can be adapted to specific requirements without the need to change the center's key elements or any other part of the system. Future extensions can be attached with only few and local modifications to existing parts.

If, however, we must change an aspect of a design center that is visible to clients—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 Broker pattern [BMRSS96] defines a stable design center that provides an infrastructure for component communication and cooperation in a distributed system. Internal aspects may differ from implementation to implementation of a Broker architecture. The general structure and its interface to clients, however, is fixed and even standardized [OMG95a]. This allows client and server components, and even different Broker implementations, to cooperate with each other, using protocols that stay stable over the lifetime of the system.

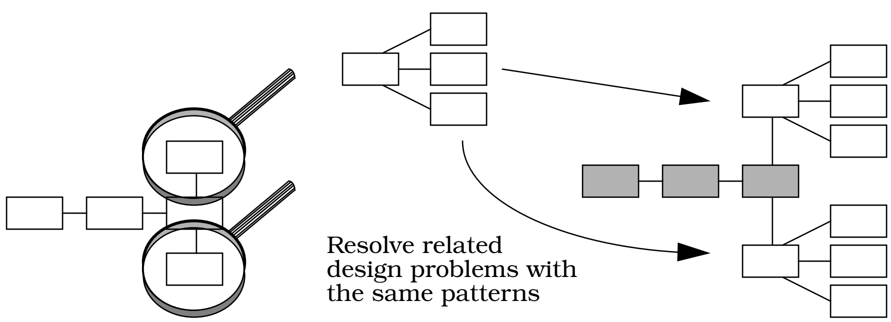
One Mechanism for One Kind of Problem

Context We are specifying and decomposing a software architecture by *Stepwise Refinement* (67).

Problem *How can we handle similar and closely related problems* that occur in different places of the system? An example is decoupling the specifics of concrete input and output devices from the general handling of input and output in an application. Two *forces* arise:

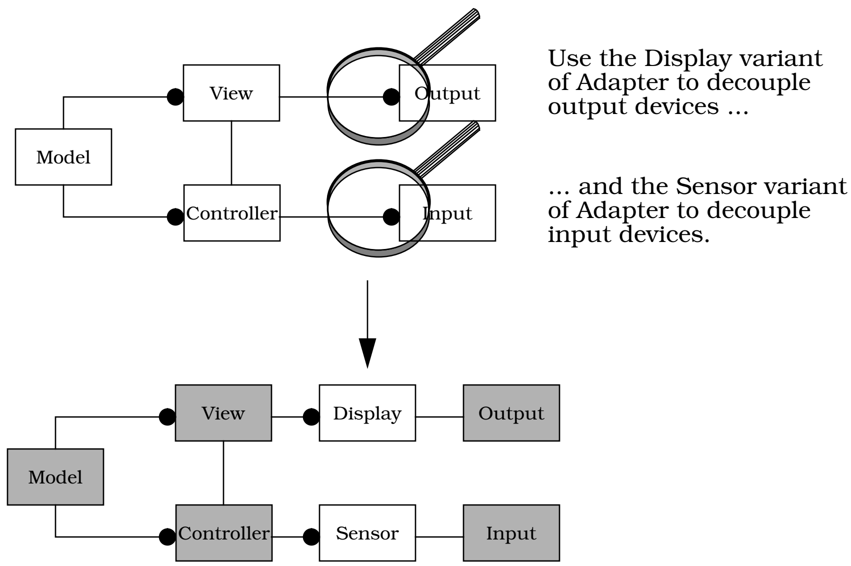
- Handling related design problems differently may result in a complex system structure. Though all problems share a common core, design and implementation of their solutions may vary significantly—even in their essence. As an effect, the architecture becomes hard to understand and maintain.
- The more general principles are used to solve related design problems, the less coherent a software architecture is.

Solution *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, specify their solutions to the same level of granularity.



As a result, 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.

Example When specifying a Model-View-Controller architecture [BMRSS96] you may want to decouple the general handling of input from specific details of the input devices used, such as mouse and light pen. The Sensor variant of the Adapter pattern [GHJV95] helps with this. In short, a sensor provides an unidirectional adaptation from a specific to a general interface. However, the system does not only receive input, it will also produce output. Thus, when refining the 'input-side' of a system, it is wise to also refine its 'output-side'. The Display variant of the Adapter pattern [GHJV95] provides a unidirectional adaptation from a general to a specific interface. It is the appropriate counterpart to the Sensor pattern applied previously.



One Pattern at a Time

Context We are specifying either an *Architectural Vision* (64) or— by *Step-wise Refinement* (67)—the details of a given software architecture. To resolve a specific design problem we selected several patterns.

Problem *How many patterns should be applied at once* to specify the solution to the design problem at hand? Two *forces* arise:

- The resulting 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, distract our attention from solving the original problem.

Solution *Apply one pattern at a time* [Ale79]. Begin with the pattern that addresses the most important sub-aspect of the problem, followed by the patterns that resolve less important aspects. *First Integrate Then Implement* (77) every pattern that is applied.

Step by step we develop the solution to the problem. By applying only one pattern at a time, the resulting design will most likely reflect the essence of the patterns we apply. We can follow their implementation guidelines more easily, and aspects of other, not yet applied patterns, do not need to be considered. The concrete implementations of all patterns together form a coherent structure which effectively solves the design problem at hand.

Example When implementing the Command Processor pattern [BMRSS96], several other patterns help with specifying its details, such as Command and Composite [GHJV95]. However, we do not apply both patterns at the same time. First we apply Command to encapsulate user request into objects. Composite is applied thereafter to support macro commands.

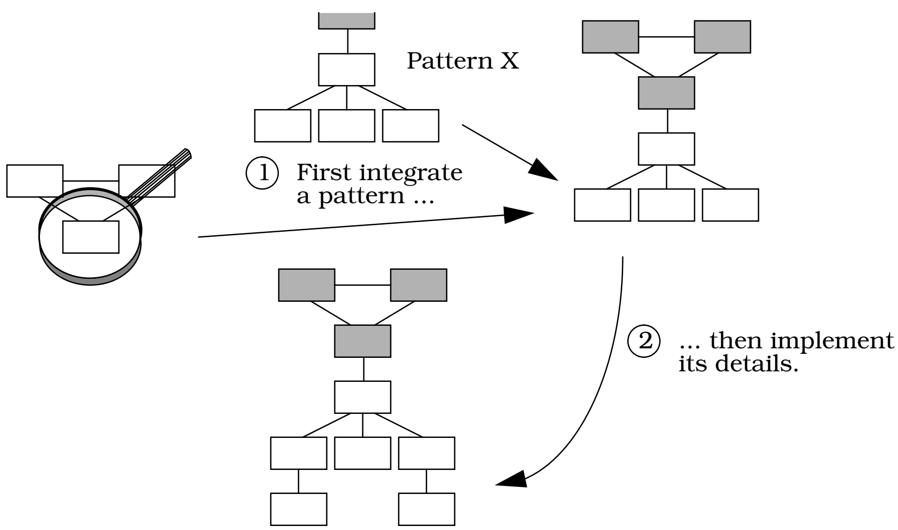
First Integrate Then Implement

Context We are defining a specific part of a software architecture with patterns by applying *One Pattern at a Time* (76).

Problem *How can a given pattern be best implemented*? Two *forces* must be considered:

- The implementation must be tailored to the needs of the design problem at hand, and also according to the needs of the whole system.
- 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.

Solution *Integrate the pattern into the existing architecture before implementing its details*. First, identify the pattern's clients. *Merge Similar Responsibilities* (80), if these provide roles that the pattern introduces. Otherwise, specify each pattern component according to the needs of its clients. Examples include input and output parameters, function names, data structures, and algorithms. Then, implement each pattern component according to its specification.



Parts of the pattern—components or relationships between them—that are too complex to be implemented straight forward, should be further refined. Follow the process of *Piecemeal Growth* (61) to unfold these details appropriately.

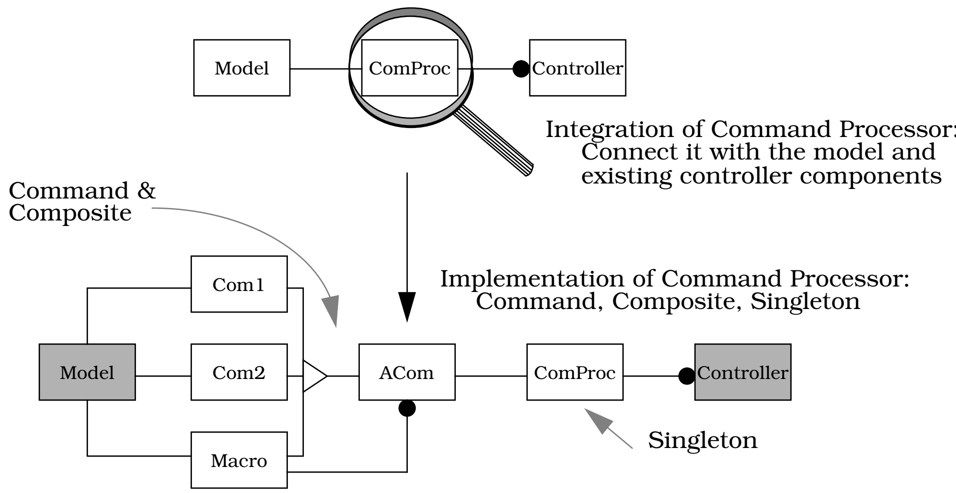
Parts of the pattern that can be implemented directly should not be further decomposed—even if you know patterns that help with this. It will only result in a complex pattern implementation. The more that a component is split into subcomponents, the harder that it becomes to understand, implement and maintain. Furthermore, an overly fine-grained decomposition may introduce performance penalties, due to additional communication between subcomponents.

Example Consider applying the Command Processor pattern [BMRSS96] to refine the relationships between the controllers and the model of a Model-View-Controller architecture [BMRSS96].

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

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

After integrating the Command Processor pattern with the existing design we can proceed with implementing its details. Several patterns help with this. Command [GHJV95] specifies how to encapsulate user request into objects. Composite [GHJV95] can be used for providing macro commands. Singleton [GHJV95] helps with ensuring that the command processor can be instantiated only once. The relationships between the pattern participants can be implemented straight forward and without further refinement, for example by using pointers or references.



An example for a 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 provide 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 and calls 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. To notify registered views about changes we only need one iterator. There is no need to prepare the view registry for handling multiple iterators. 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. We do not need to apply Iterator here, as in the first application of Observer.

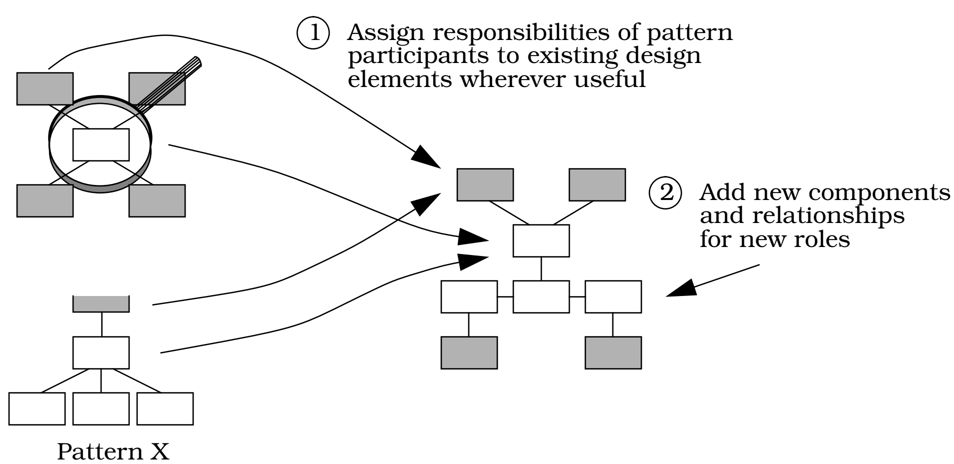
Merge Similar Responsibilities

Context We *First Integrate Then Implement* (77) a specific pattern.

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

- We must integrate the pattern such that the existing architecture governs its implementation.
- Elements of the existing architecture may already provide responsibilities that are defined by the pattern.
- Responsibilities of pattern participants may complement or complete responsibilities of existing design elements.
- Adding new components or relationships to the design likely increases the structural complexity of the system.
- Assigning overly many and distinct responsibilities to a design element will likely break the principle of separation of concerns. The resulting 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.



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 [BMRSS96]. On the other hand, overloading a component with overly many different responsibilities may introduce inefficiency. If the component plays distinct roles in several different contexts, it may become a performance bottleneck. Keeping the responsibilities separate from each other would be more effective.

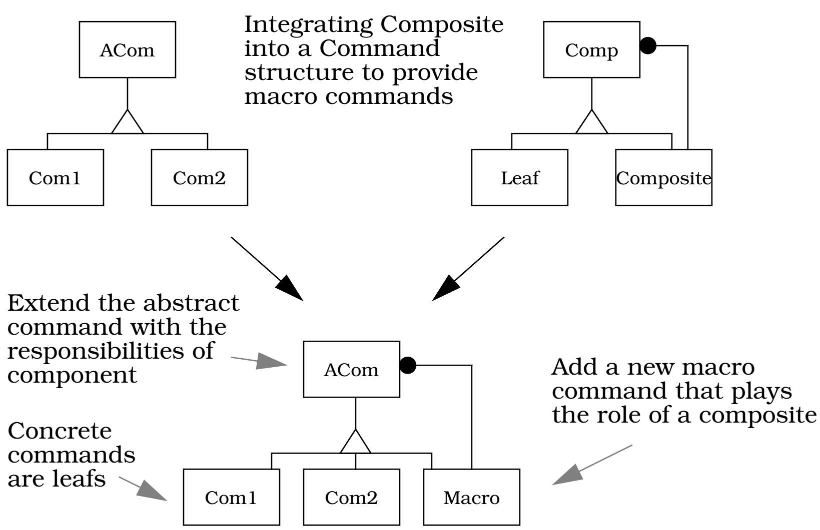
In general, assigning responsibilities of pattern participants to existing design elements helps reducing the structural complexity of a software architecture. It also ensures that the implementation of the pattern is tailored to the needs of the system, and not vice versa.

However, do not assign responsibilities of a pattern participant to an existing design element that it does not need to provide or which must be implemented separately. Specify these, according to the pattern's implementation guidelines, 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.

Example Suppose we are encapsulating service requests as objects by applying the Command pattern [GHJV95]. An extension to this would be a support for macro commands. The Composite pattern [GHJV95] helps with solving this problem. We refine our design as follows.

The responsibilities of the 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 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.



6.3 Discussion

In the introduction section we have said that 'Applying Patterns' addresses technical and human aspects of using patterns for software architecture. Now that we know the language in full length, we can discuss this statement in detail. We can also revisit the pre-conditions under which the language works best, and discuss their impact as well.

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 language lists concrete kinds of patterns that help with achieving the goal. For example, that the Broker architectural pattern [BMRSS96] 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. 5 patterns of 'Applying Patterns' address such technical aspects:

- *Architectural Vision* (64) 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* (72) names specific patterns for software architecture which support us in keeping a design stable and flexible at the same time.
- *One Pattern at a Time* (76), *First Integrate Then Implement* (77) and *Merge Similar Responsibilities* (80) 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 can developers be supported in tackling a design problem most 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.

'Applying Patterns' addresses several human aspects that are specifically related to using patterns for software architecture:

- *One Pattern at a Time* (76) 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* (80) and *One Mechanism for One Kind of Problem* (74).

Most human issues in software development, however, are independent of the use of patterns. 'Applying Patterns' addresses these in 4 of its patterns:

- *Piecemeal Growth* (61), *Step-wise Refinement* (67) and *Repair Instead Large Lump Design* (69) address, together, 4 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* (64) addresses the aspect of communication: what are the key design ideas and concepts of the system.

Within our language both technical and human issues of constructing software systems with patterns are tightly interwoven. For example, *Architectural Vision* (64) addresses how to define a subsystem structure *and* how to support communication of key design concepts. The purpose of *Merge Similar Responsibilities* (80) 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, as in 'Applying Patterns', 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 'Applying Patterns' 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 *Repair Instead Large Lump Design* (69) for itself, nor its interplay with *Step-wise Refinement* (67) would be applicable. Both patterns, however, describe fundamental aspects of the process of piecemeal growth.

But how does 'Applying Patterns' integrate into 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 and 'repair' aspects are interwoven. This, however, is subject of 'Applying Patterns'. 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 change or add-on that 'Applying Patterns' suggests to existing process models. The language stresses the importance of specifying an architectural skeleton for the system right at the beginning of its development, as manifested by the pattern *Architectural Vision* (64). This architectural skeleton is defined even before the specification of the domain model, which is usually the first activity in existing processes. With respect to integrating 'Applying Patterns' with these processes, we can simply add an architecture specification phase up-front. After that, we proceed with following the original process, for example with specifying the domain specific parts. 'Applying Patterns' is then used to support the steps as defined by that process.

With this discussion another question arises. Does 'Applying Patterns' 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 Modeling Language (UML) [RGB97]. This is, however, not the case.

Instead, 'Applying Patterns' complements existing approaches with respect to using patterns for software architecture. *One Pattern at a Time* (76), *First Integrate Then Implement* (77), *Merge Similar Responsibilities* (80) and *One Mechanism for One Kind of Problem* (74) define specific activities and principles for selecting, integrating, refining, and coding patterns for software architecture. *Architectural Vision* (64), *Step-wise Refinement* (67) and *Repair Instead Large Lump Design* (69) specify how to use patterns within general design activities that are defined by existing methods. The same holds for *Stable Design Center* (72), which defines a general design goal, but under consideration of using patterns to achieve it. *Piecemeal Growth* (61), finally, details the idea of evolutionary development, as we discussed above.

In summary, 'Applying Patterns' 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* (64). 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, *One Mechanism for One Kind of Prob-*

lem (74). 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 and supervise this coordination.

'Applying Patterns' 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 'Applying Patterns' successfully.

Summary

'Applying Patterns' 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, 'Applying Patterns' 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.