The essence of modern software engineering
OOP 2008
Software meets Business

Mo 2

January 21-25, 2008, Munich, Germany
ICM - International Congress Centre Munich

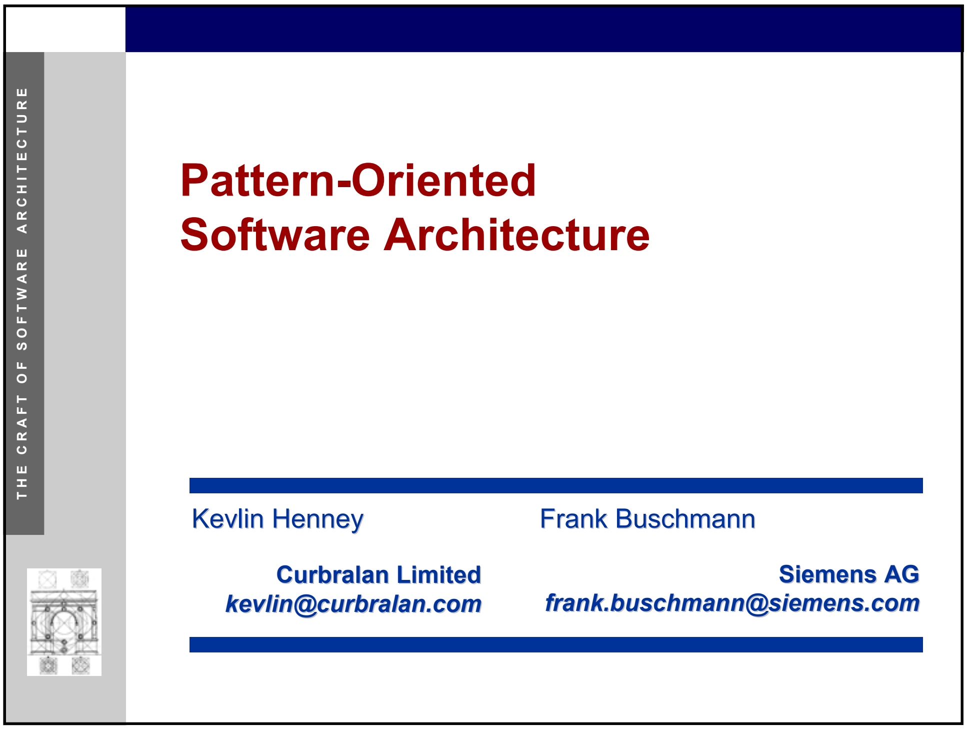
Pattern-Oriented Software Architecture

Frank Buschmann
Kevlin Henney

organized by

SIGS DATACOM
FACHINFORMATIONEN FÜR IT-PROFESSIONALS

SIGS DATACOM GmbH
Lindlaustraße 2c | D-53842 Troisdorf
Tel. +49 (0)2241.2341-100 | Fax +49 (0)2241.2341-199
info@sigs-datacom.de | www.sigs-datacom.de



THE CRAFT OF SOFTWARE ARCHITECTURE

Pattern-Oriented Software Architecture

Kevlin Henney

Curbralan Limited
kevlin@curbralan.com

Frank Buschmann

Siemens AG
frank.buschmann@siemens.com

PATTERN-ORIENTED SOFTWARE ARCHITECTURE

Agenda

- Introduction
- Stand-Alone Patterns
- Pattern Complements / Pattern Compounds
- Pattern Stories / Pattern Sequences
- Pattern Languages
- Outroduction
- References

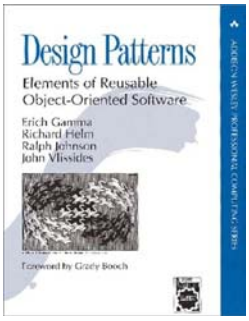
Where we are

- **Introduction**
- **Stand-Alone Patterns**
- **Pattern Complements / Pattern Compounds**
- **Pattern Stories / Pattern Sequences**
- **Pattern Languages**
- **Outroduction**
- **References**

On Designing with Patterns (1)

Patterns have become a popular tool in software design …

- They have changed the way software developers think about and practice software design
- They provide us with a software design vocabulary
- They help us to resolve recurring problems constructively and based on proven solutions
- They support us in understanding the architecture of a given software system
- …


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

But: applying patterns in software design is not necessarily designing with patterns!

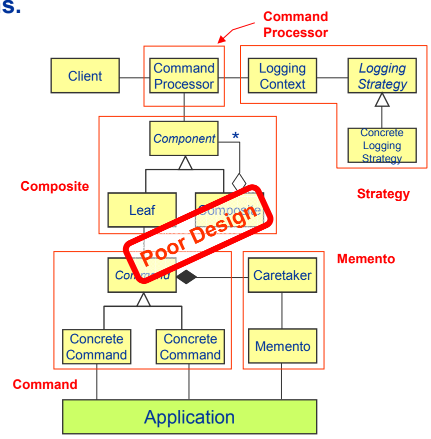
PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

On Designing with Patterns (2)

Patterns applied in isolation and as modular building blocks do not create sustainable designs.

Solutions

- Fail to consider the surrounding problems and their solutions.
- Have at best only local effects.
- Do not complement one another or mutually reinforce their beneficial properties.
- Create complex architectures that lack useful operational and developmental qualities.



PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

On Designing with Patterns (3)

There are two fundamental ways of integrating patterns:

- *Refinement*: One pattern refines the structure and behavior of another pattern to address a specific sub-problem or implementation detail.
- *Combination*: Two or more patterns arranged to form a larger structure that addresses a more complex problem.

There are also relationships regarding choice:

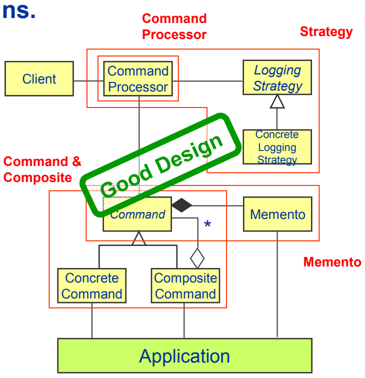
- *Alternatives*: Some patterns describe alternatives to one another. They address the same or a similar problem, but each pattern considers a slightly different set of forces. Thus, the patterns provide different solutions and have different consequences.
- *Cooperation*: Some patterns nicely complement one another, mutually reinforcing their structural and behavioral properties.

On Designing with Patterns (4)

Patterns applied by considering their relationships create balanced designs.

Solutions

- Consider their surrounding problem and solution space.
- Have systemic effects.
- Complement one another and mutually reinforce their beneficial properties.
- Create whole and balanced designs that expose the required operational and developmental qualities.



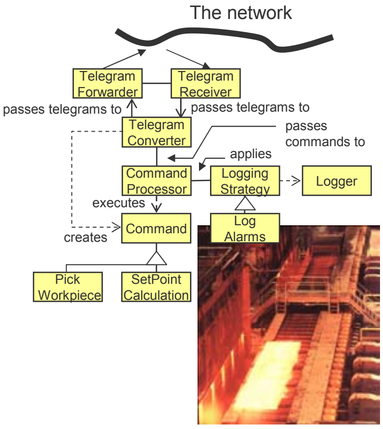
On Designing with Patterns (5)

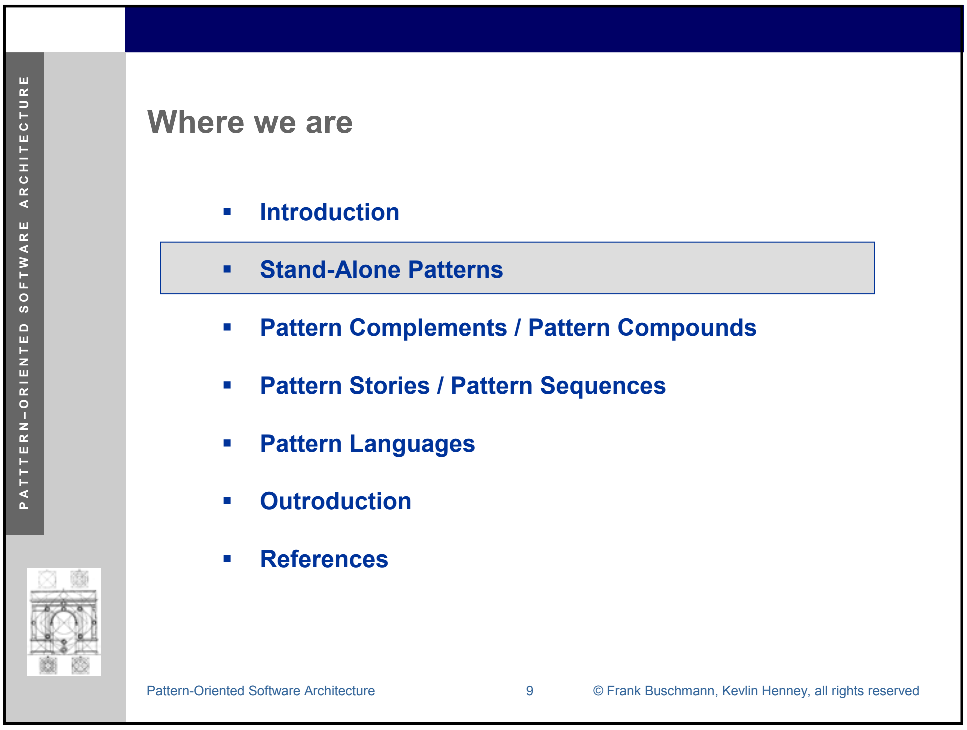
This tutorial is on designing with patterns to create,

with economy and elegance,

software designs and implementations

that work and meet their expected quality attributes!

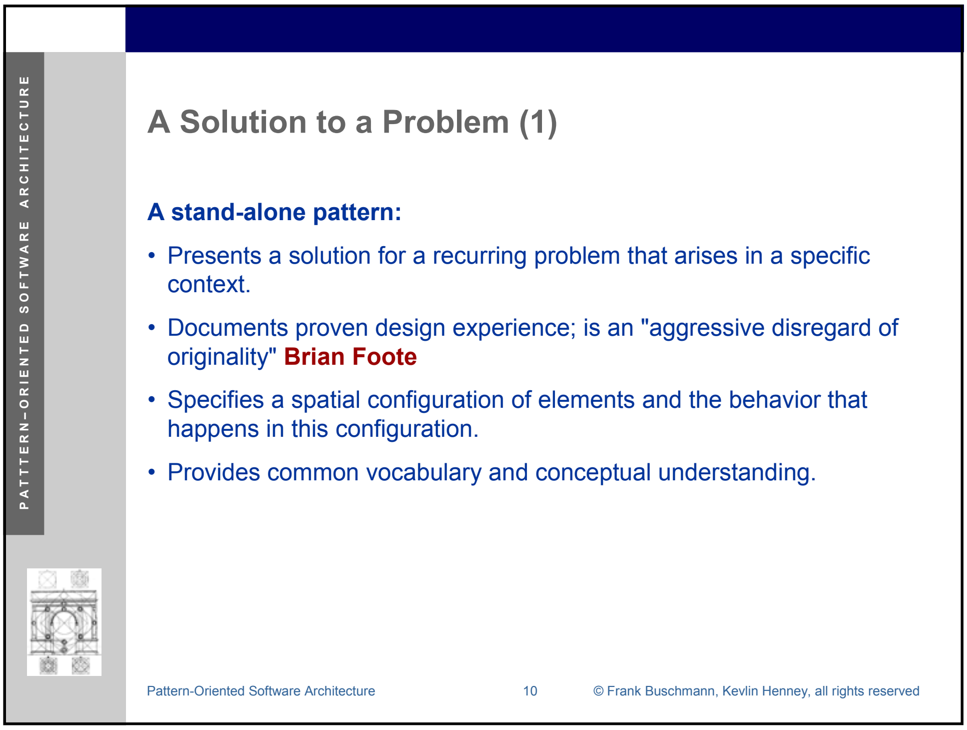




PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

Where we are

- Introduction
- Stand-Alone Patterns
- Pattern Complements / Pattern Compounds
- Pattern Stories / Pattern Sequences
- Pattern Languages
- Outroduction
- References



PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

A Solution to a Problem (1)

A stand-alone pattern:

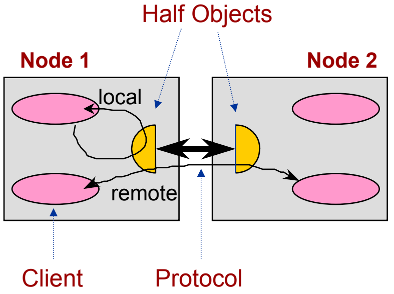
- Presents a solution for a recurring problem that arises in a specific context.
- Documents proven design experience; is an "aggressive disregard of originality" **Brian Foote**
- Specifies a spatial configuration of elements and the behavior that happens in this configuration.
- Provides common vocabulary and conceptual understanding.

Half-Object Plus Protocol at a Glance

Problem

How can we design objects in a distributed system:

- such that access to them and the execution of their services incurs minimal performance penalties?



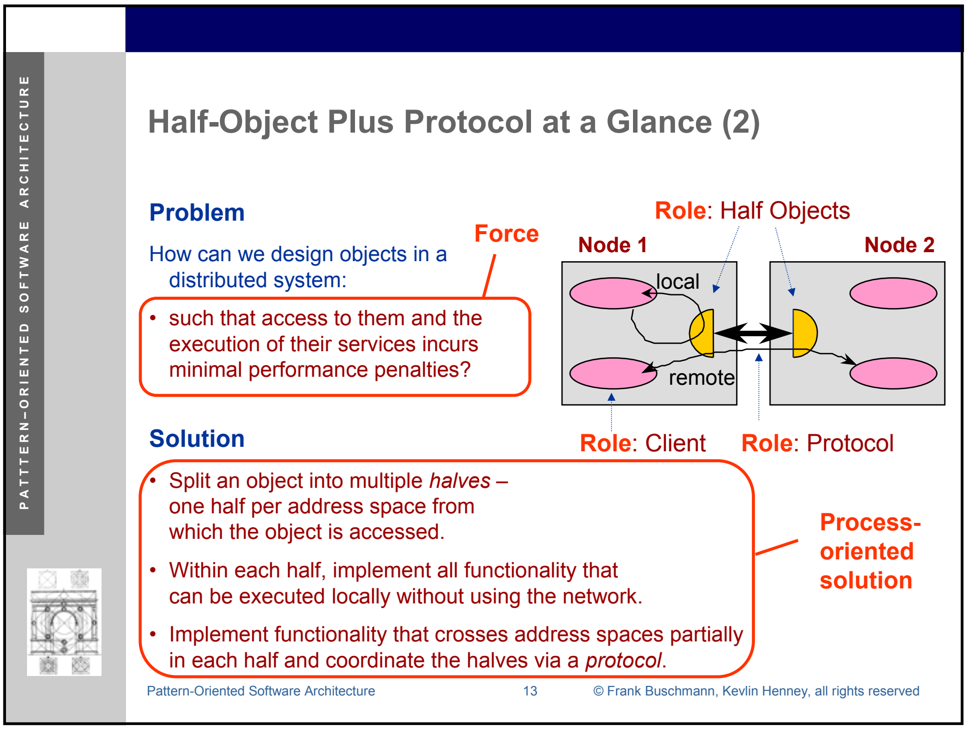
Solution

- Split an object into multiple *halves* – one half per address space from which the object is accessed.
- Within each half, fully implement all functionality that can be executed locally without using the network.
- Implement functionality that crosses address spaces partially in each half and coordinate the halves via a *protocol*.

A Solution to a Problem (2)

A stand-alone pattern:

- Is both a process and a thing, with the thing being created by the process.
- Addresses a set of forces that completes the general problem by describing requirements, constraints, and desired properties for the solution.
- Introduces a set of interacting roles that can be arranged in many different ways, not a fixed configuration of classes or components.
- Specifies a solution that is based on experience, judgment, and diligence – it cannot necessarily be constructed in a straightforward manner using a common engineering method.



PATTTERN-ORIENTED SOFTWARE ARCHITECTURE

Half-Object Plus Protocol at a Glance (2)

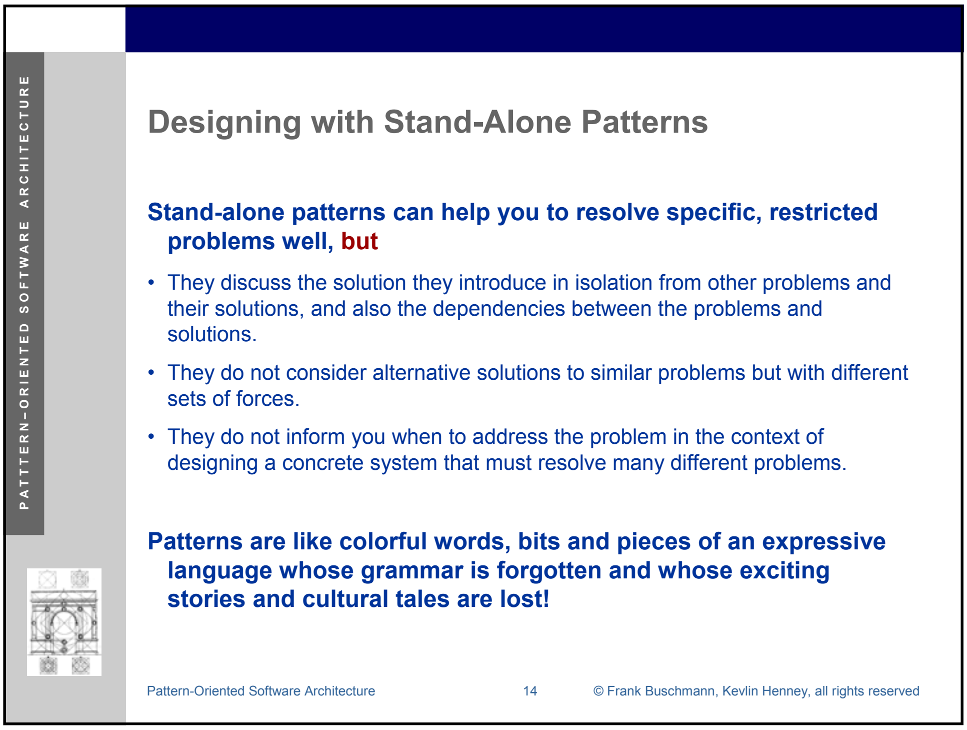
Problem

How can we design objects in a distributed system:

- such that access to them and the execution of their services incurs minimal performance penalties?

Solution

- Split an object into multiple *halves* – one half per address space from which the object is accessed.
- Within each half, implement all functionality that can be executed locally without using the network.
- Implement functionality that crosses address spaces partially in each half and coordinate the halves via a *protocol*.



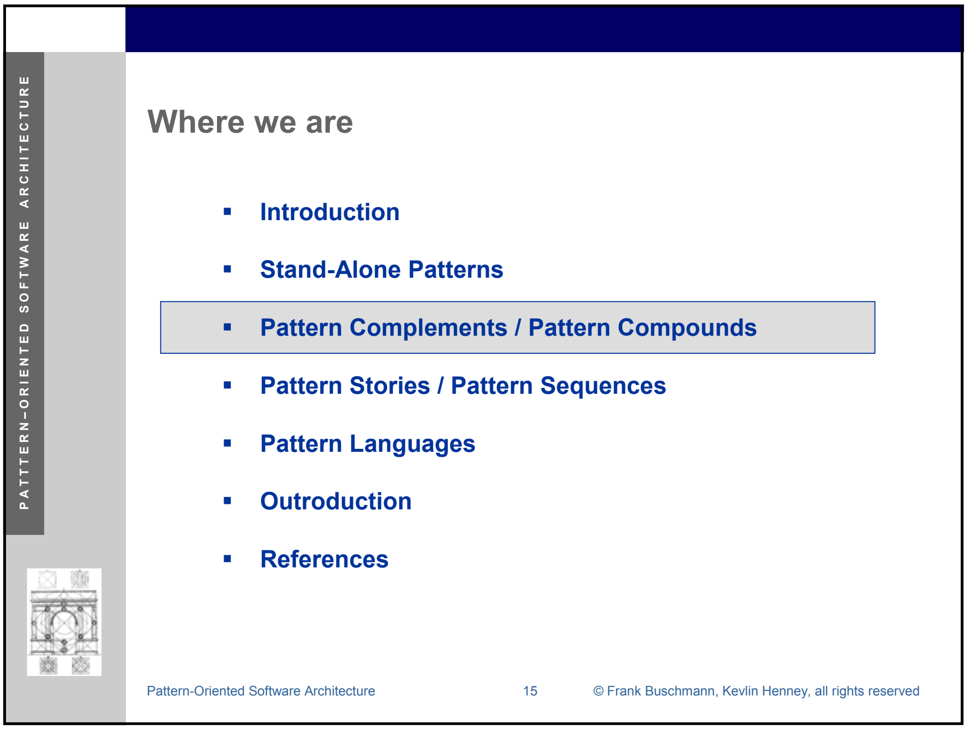
PATTTERN-ORIENTED SOFTWARE ARCHITECTURE

Designing with Stand-Alone Patterns

Stand-alone patterns can help you to resolve specific, restricted problems well, but

- They discuss the solution they introduce in isolation from other problems and their solutions, and also the dependencies between the problems and solutions.
- They do not consider alternative solutions to similar problems but with different sets of forces.
- They do not inform you when to address the problem in the context of designing a concrete system that must resolve many different problems.

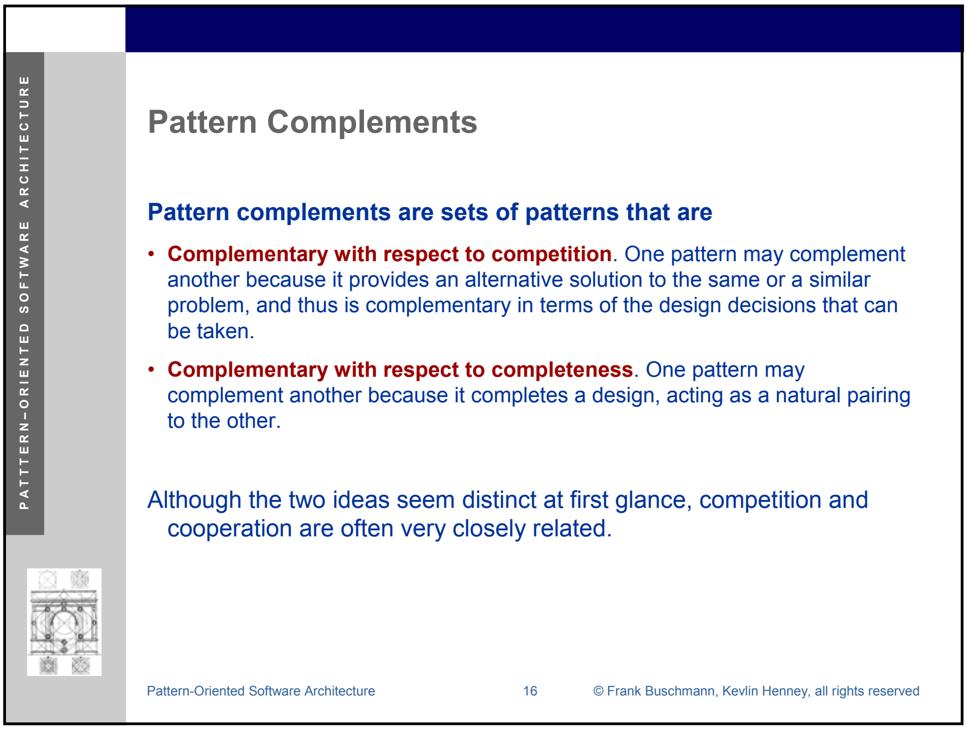
Patterns are like colorful words, bits and pieces of an expressive language whose grammar is forgotten and whose exciting stories and cultural tales are lost!



PATTTERN-ORIENTED SOFTWARE ARCHITECTURE

Where we are

- Introduction
- Stand-Alone Patterns
- Pattern Complements / Pattern Compounds
- Pattern Stories / Pattern Sequences
- Pattern Languages
- Outroduction
- References



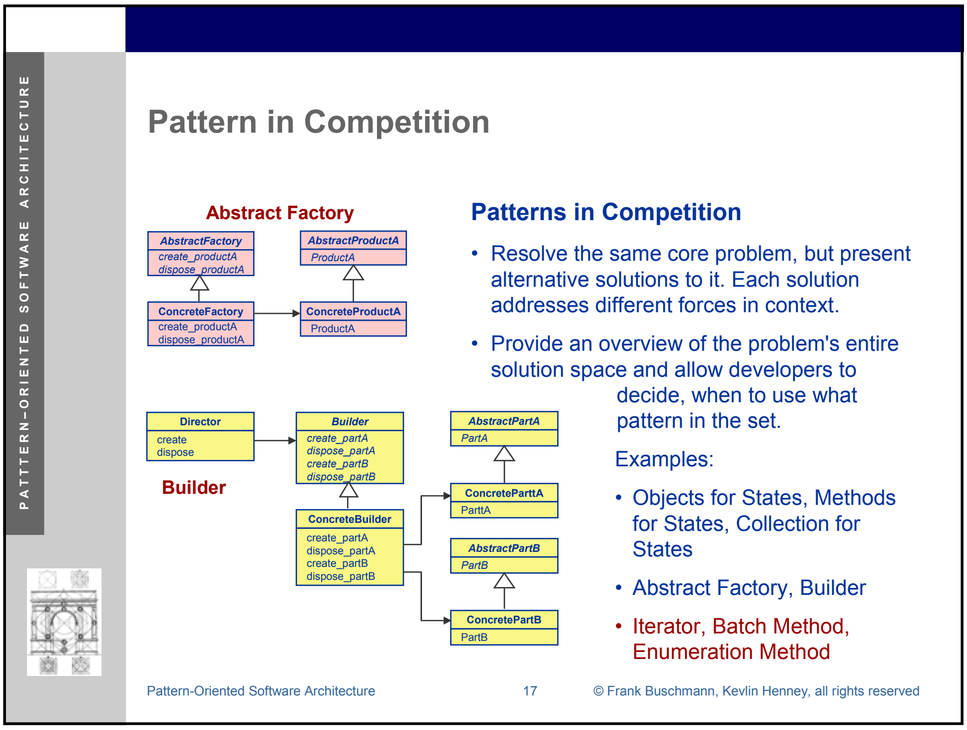
PATTTERN-ORIENTED SOFTWARE ARCHITECTURE

Pattern Complements

Pattern complements are sets of patterns that are

- **Complementary with respect to competition**. One pattern may complement another because it provides an alternative solution to the same or a similar problem, and thus is complementary in terms of the design decisions that can be taken.
- **Complementary with respect to completeness**. One pattern may complement another because it completes a design, acting as a natural pairing to the other.

Although the two ideas seem distinct at first glance, competition and cooperation are often very closely related.



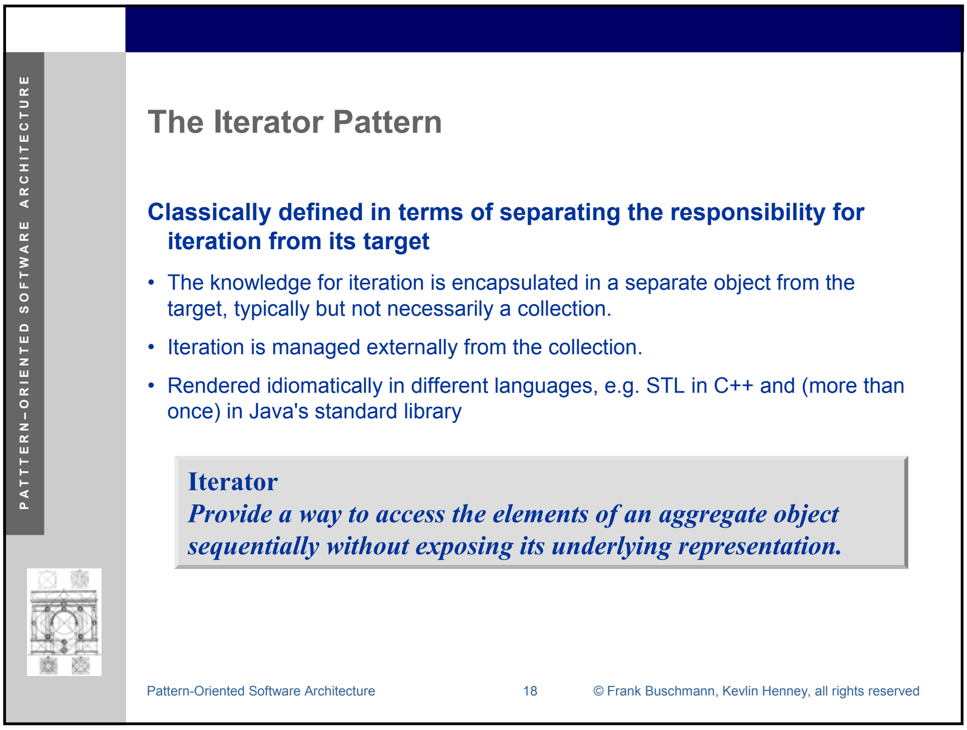
Pattern in Competition

Patterns in Competition

- Resolve the same core problem, but present alternative solutions to it. Each solution addresses different forces in context.
- Provide an overview of the problem's entire solution space and allow developers to decide, when to use what pattern in the set.

Examples:

- Objects for States, Methods for States, Collection for States
- Abstract Factory, Builder
- Iterator, Batch Method, Enumeration Method



The Iterator Pattern

Classically defined in terms of separating the responsibility for iteration from its target

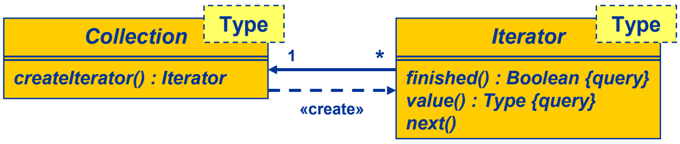
- The knowledge for iteration is encapsulated in a separate object from the target, typically but not necessarily a collection.
- Iteration is managed externally from the collection.
- Rendered idiomatically in different languages, e.g. STL in C++ and (more than once) in Java's standard library

> **Iterator**
> ***Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.***

Iterator Configuration Sketch

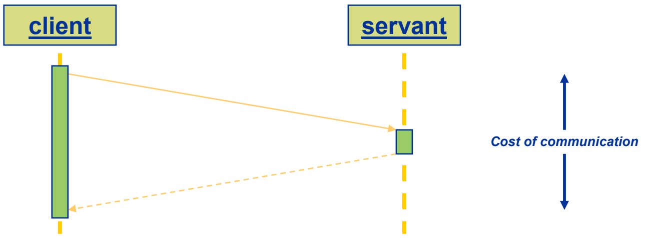
There are four essential, elementary operations associated with an Iterator's behavior

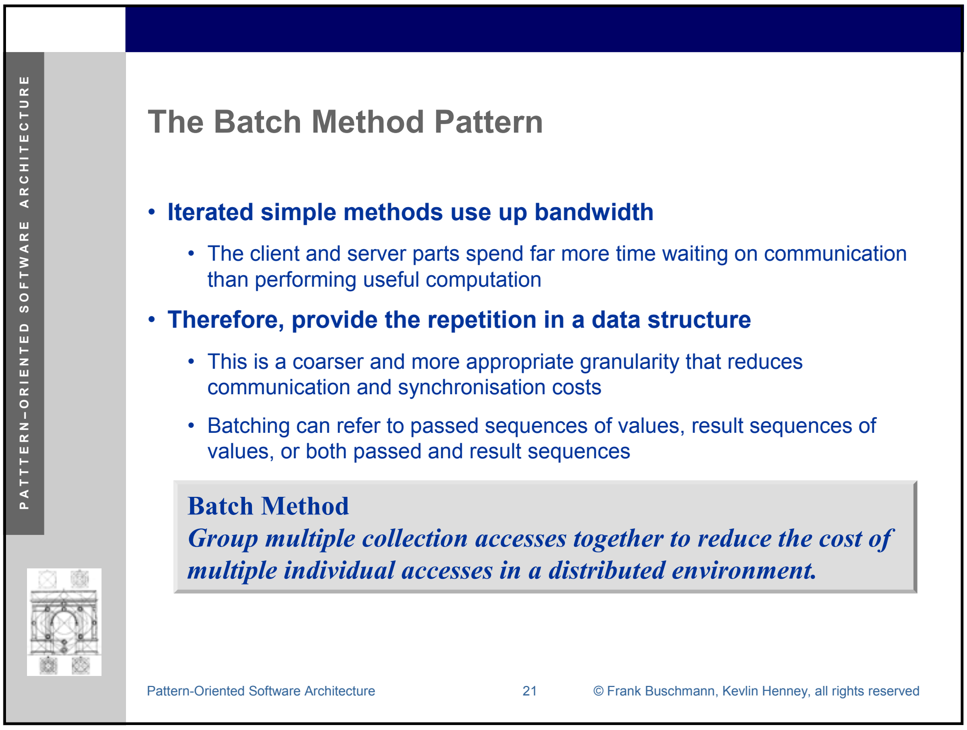
- Initializing an iteration, i.e. the initializer of a *for* loop
- Checking a completion condition, i.e. the continuation condition of a *for* loop
- Accessing a current target value, e.g. in the body of a *for* loop
- Moving to the next target value, i.e. the progression of a *for* loop



Distribution Issues

- **In a distributed environment concurrency is implicit**
  - But this is not the only force at work
- **Operation invocations are no longer trivial**
  - Communication can dominate computation
  - Partial failure is almost inevitable

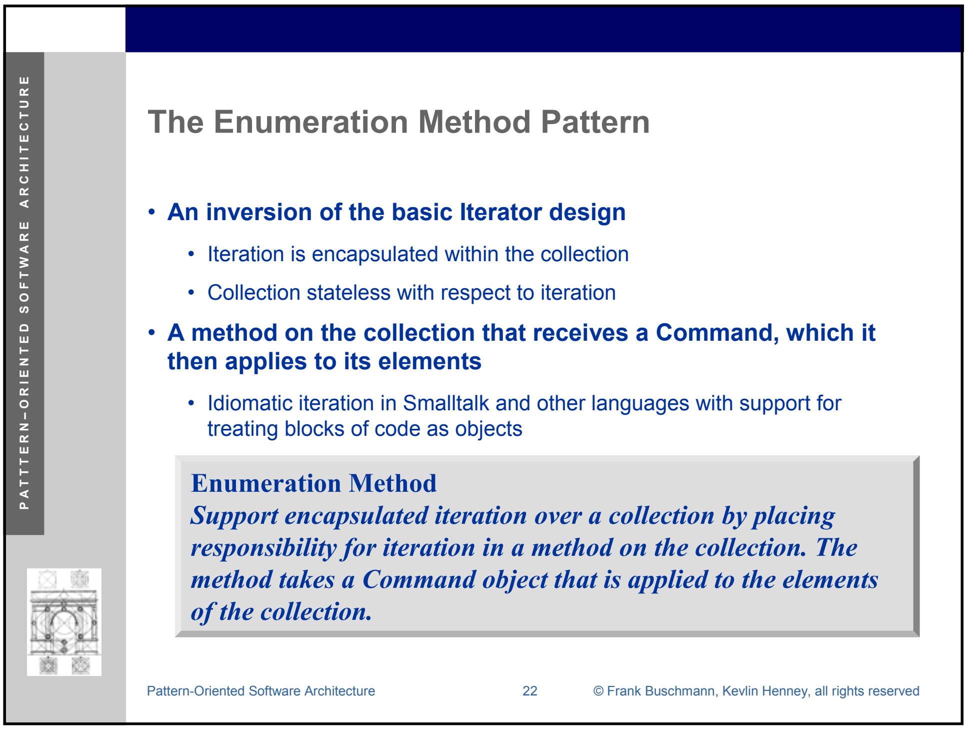




The Batch Method Pattern

- **Iterated simple methods use up bandwidth**
  - The client and server parts spend far more time waiting on communication than performing useful computation
- **Therefore, provide the repetition in a data structure**
  - This is a coarser and more appropriate granularity that reduces communication and synchronisation costs
  - Batching can refer to passed sequences of values, result sequences of values, or both passed and result sequences

> **Batch Method**
> ***Group multiple collection accesses together to reduce the cost of multiple individual accesses in a distributed environment.***



The Enumeration Method Pattern

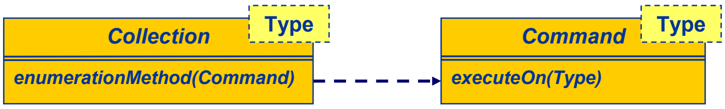
- **An inversion of the basic Iterator design**
  - Iteration is encapsulated within the collection
  - Collection stateless with respect to iteration
- **A method on the collection that receives a Command, which it then applies to its elements**
  - Idiomatic iteration in Smalltalk and other languages with support for treating blocks of code as objects

> **Enumeration Method**
> ***Support encapsulated iteration over a collection by placing responsibility for iteration in a method on the collection. The method takes a Command object that is applied to the elements of the collection.***

Enumeration Method Sketch

Has very different trade-offs when compared to the Iterator approach

- Client is not responsible for loop housekeeping details
- Synchronization can be provided at the level of the whole traversal rather than for each element access
- Sometimes known as the *Internal Iterator* pattern, but clearly has nothing in common with *(External) Iterator*



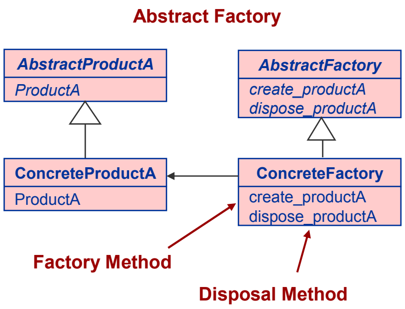
Loop administration is handled in the implementation of the *enumerationMethod*

Loop body is now provided as the implementation of the *executeOn* method

Pattern in Cooperation

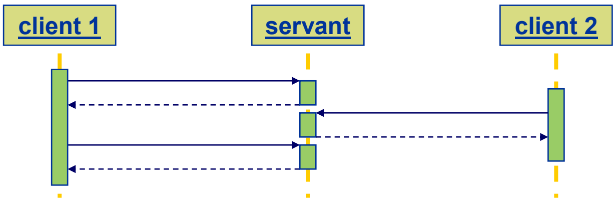
Patterns in Cooperation

- Naturally complement one another towards a more complete and balanced design.
- Mutually reinforce their quality properties.
- Examples:
  - Command and Command Processor
  - Abstract Factory, Factory Method, and Disposal Method
  - Iterator, Combined Method, and Batch Method



Concurrency Issues for Iterator

- **Synchronization is required to ensure consistent and coherent state change**
  - This cannot be handled adequately by the Iterator client
- **Property-style programming, i.e. using fine-grained getters and setters, is inappropriate**
  - There is a mismatch in granularity and coherence

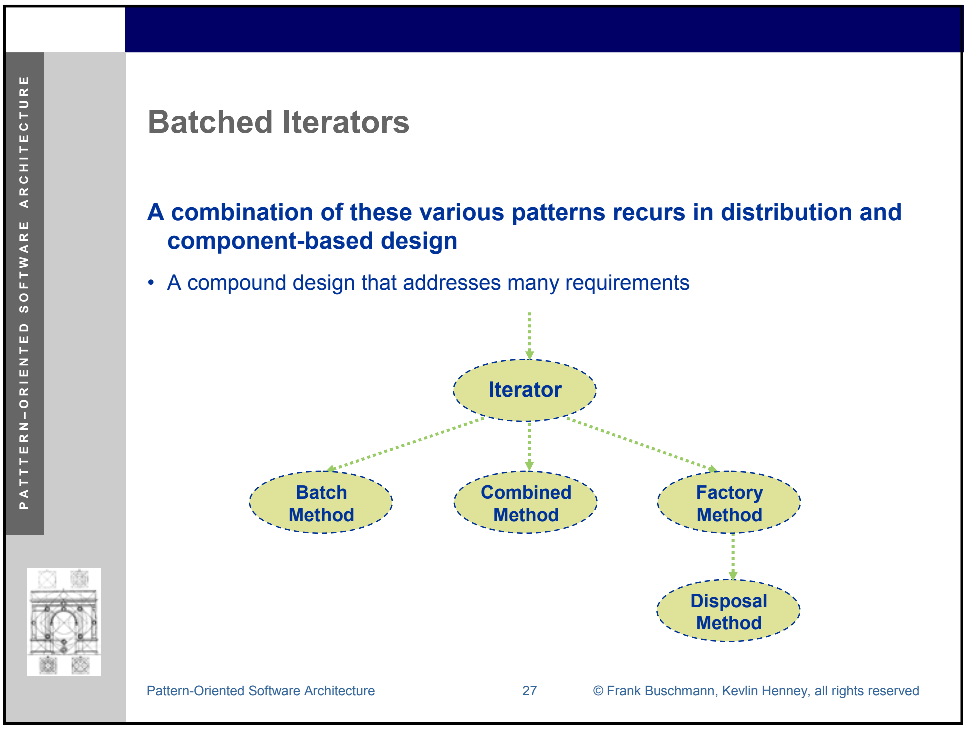


The Combined Method Pattern

Applying it gives slightly coarser granularity than the separated methods of an ordinary sequential Iterator

- Aligns and groups the unit of failure, synchronization and common use
- Makes method design more transactional in style
- Improves the encapsulation of collection use, isolating the client from unnecessary details of execution and failure

> **Combined Method**
> ***Combine methods that are commonly used together to guarantee correctness and improve efficiency in threaded and distributed environments.***

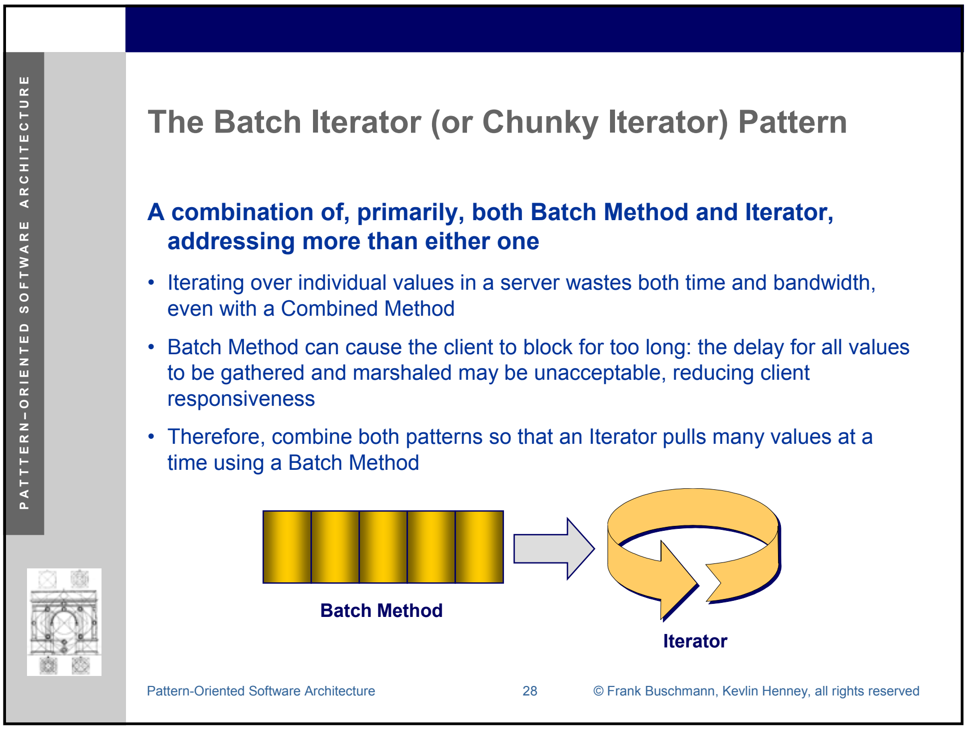


Batched Iterators

A combination of these various patterns recurs in distribution and component-based design

- A compound design that addresses many requirements

Pattern-Oriented Software Architecture 27 © Frank Buschmann, Kevlin Henney, all rights reserved

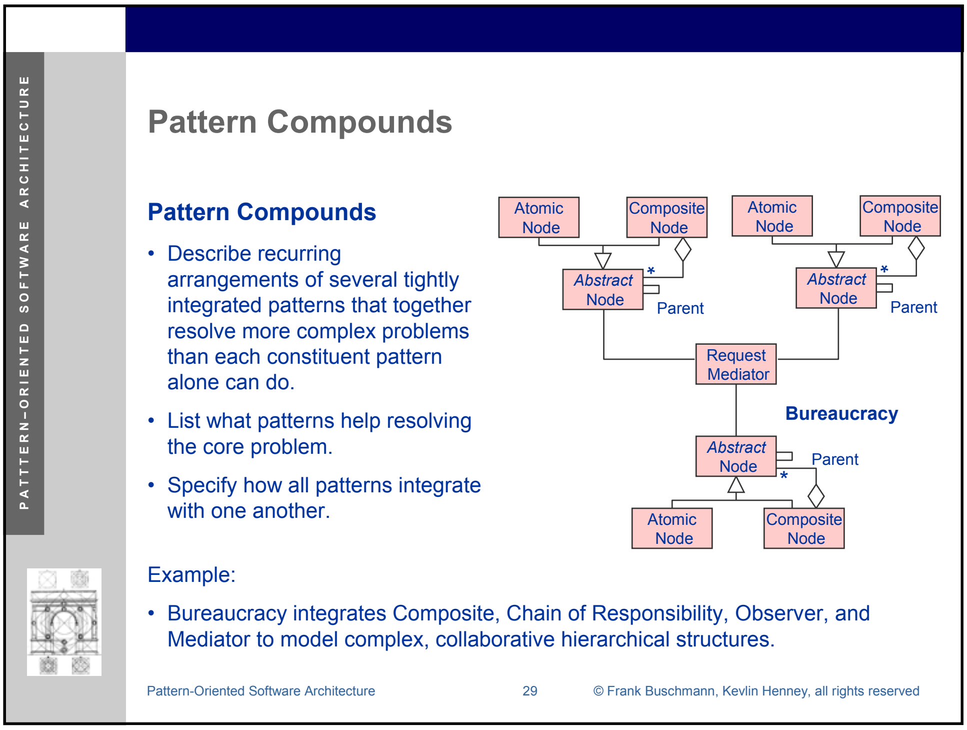


The Batch Iterator (or Chunky Iterator) Pattern

A combination of, primarily, both Batch Method and Iterator, addressing more than either one

- Iterating over individual values in a server wastes both time and bandwidth, even with a Combined Method
- Batch Method can cause the client to block for too long: the delay for all values to be gathered and marshaled may be unacceptable, reducing client responsiveness
- Therefore, combine both patterns so that an Iterator pulls many values at a time using a Batch Method

Pattern-Oriented Software Architecture 28 © Frank Buschmann, Kevlin Henney, all rights reserved



PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

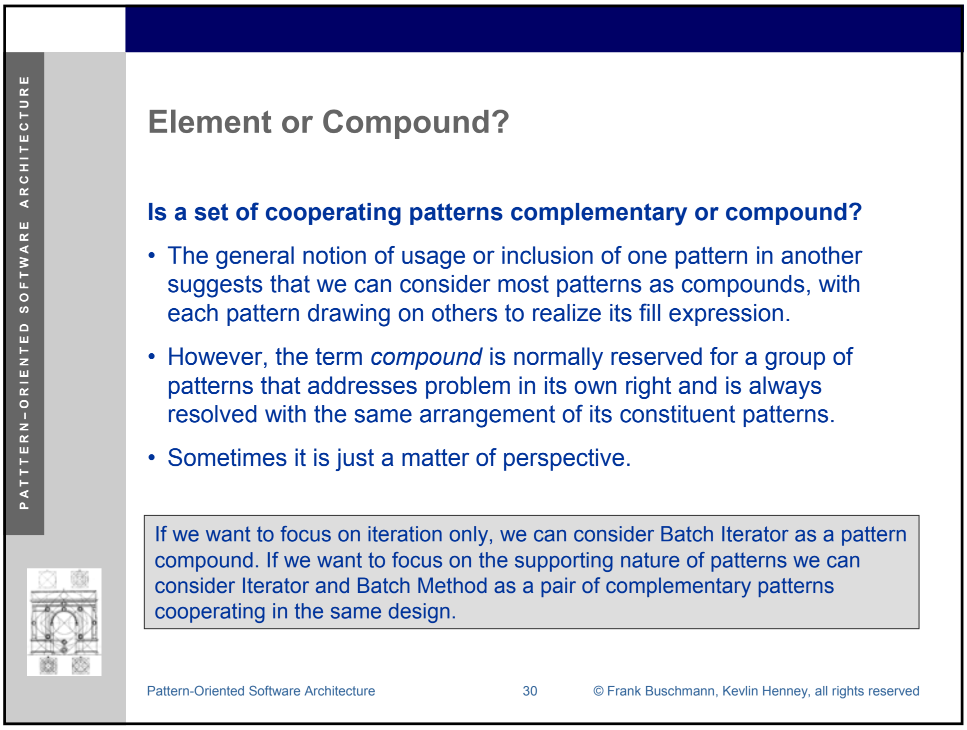
Pattern Compounds

Pattern Compounds

- Describe recurring arrangements of several tightly integrated patterns that together resolve more complex problems than each constituent pattern alone can do.
- List what patterns help resolving the core problem.
- Specify how all patterns integrate with one another.

Example:

- Bureaucracy integrates Composite, Chain of Responsibility, Observer, and Mediator to model complex, collaborative hierarchical structures.



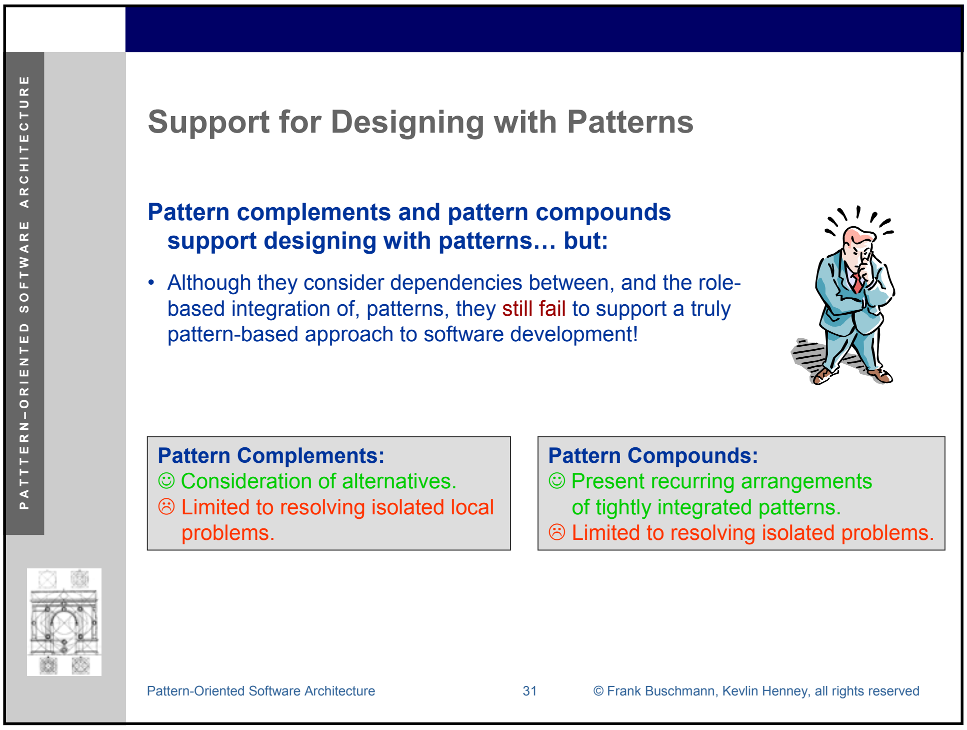
PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

Element or Compound?

Is a set of cooperating patterns complementary or compound?

- The general notion of usage or inclusion of one pattern in another suggests that we can consider most patterns as compounds, with each pattern drawing on others to realize its fill expression.
- However, the term *compound* is normally reserved for a group of patterns that addresses problem in its own right and is always resolved with the same arrangement of its constituent patterns.
- Sometimes it is just a matter of perspective.

> If we want to focus on iteration only, we can consider Batch Iterator as a pattern compound. If we want to focus on the supporting nature of patterns we can consider Iterator and Batch Method as a pair of complementary patterns cooperating in the same design.


PATTTERN-ORIENTED SOFTWARE ARCHITECTURE

Support for Designing with Patterns

Pattern complements and pattern compounds support designing with patterns… but:

- Although they consider dependencies between, and the role-based integration of, patterns, they still fail to support a truly pattern-based approach to software development!

Pattern Complements:
☺ Consideration of alternatives.
☹ Limited to resolving isolated local problems.

Pattern Compounds:
☺ Present recurring arrangements of tightly integrated patterns.
☹ Limited to resolving isolated problems.

PATTTERN-ORIENTED SOFTWARE ARCHITECTURE

Where we are

- **Introduction**
- **Stand-Alone Patterns**
- **Pattern Complements / Pattern Compounds**
- **Pattern Stories / Pattern Sequences**
- **Pattern Languages**
- **Outroduction**
- **References**

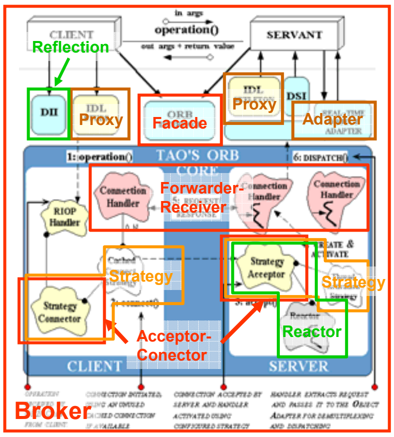
Pattern Stories

Pattern Stories

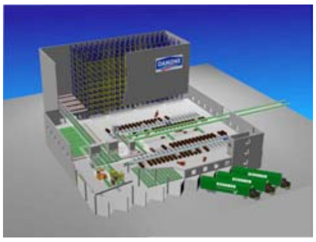
- Are like diaries that tells their readers how one specific software system, subsystem, or large component was developed with the help of patterns.

They discuss:

- What specific problems were to be resolved in what specific order.
- What patterns were considered and selected to resolve the problems.
- How the selected patterns were instantiated within the system's specific context and architecture.



A Short Story: Warehouse Management



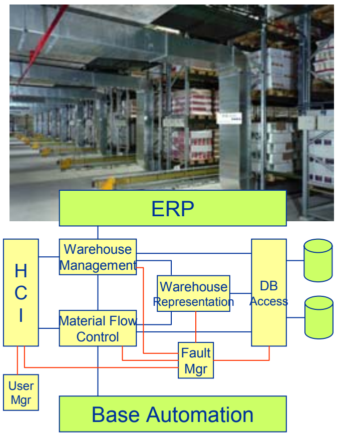
Warehouse Management Systems:

- Organize warehouse operation: storing, fetching, picking, replenishment, etc.
- Control and optimize the material flow within a warehouse.
- Control base automation.
- Cooperate with other applications, such as ERP systems and databases.

Key Challenges (1)

Building Warehouse Management Systems requires:

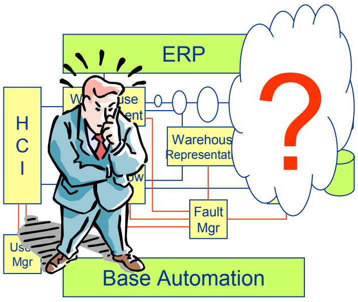
- defining an appropriate base-line architecture that specifies the system's functional and infrastructural subsystems, their relationships and key interactions.
- developing/selecting suitable component/communication middleware.
- designing and implementing the system's functionality with sufficient quality: performance, stability, scalability, extensibility, etc.
- providing an efficient connection of the system with the database.
- designing a user-friendly user interface.



Key Challenges (2)

Six key requirements affect the systems base-line architecture:

- *Modularity*: the system is developed by a large, globally distributed team
- *Distribution*: the system is highly distributed.
- *Human-computer interaction*: users interact with the system via different user interfaces.
- *Integration*: we want to integrate third-party products and legacy software.
- *Scalability*: the system must support small-scale warehouses as well as large-scale warehouses.
- *Performance*: the system must support high-performance and throughput.



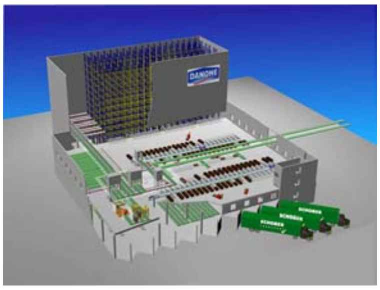
PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

Partitioning the Big Ball of Mud (1)

The basis for a sustainable base-line architecture is a clear separation and encapsulation of different system concerns.

Otherwise, the implementation of these concerns will likely be tangled rather than loosely coupled, which complicates their independent development, configuration, and deployment across a computer network.

How can we organize the system's functionality into coherent groups such that each group can be developed and modified independently?



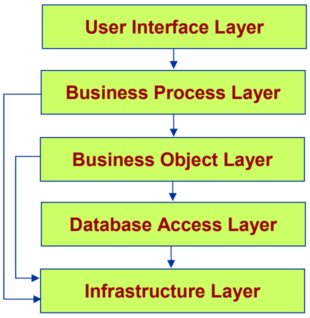
PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

Layers at a Glance

Problem

How can we partition the functionality in a system such that:

- Functionality of different kinds of abstraction and levels of granularity is decoupled as much as possible.
- Functionality at a particular level of abstraction or granularity can evolve at different times and rates without incurring rippling effects.



User Interface Layer

Business Process Layer

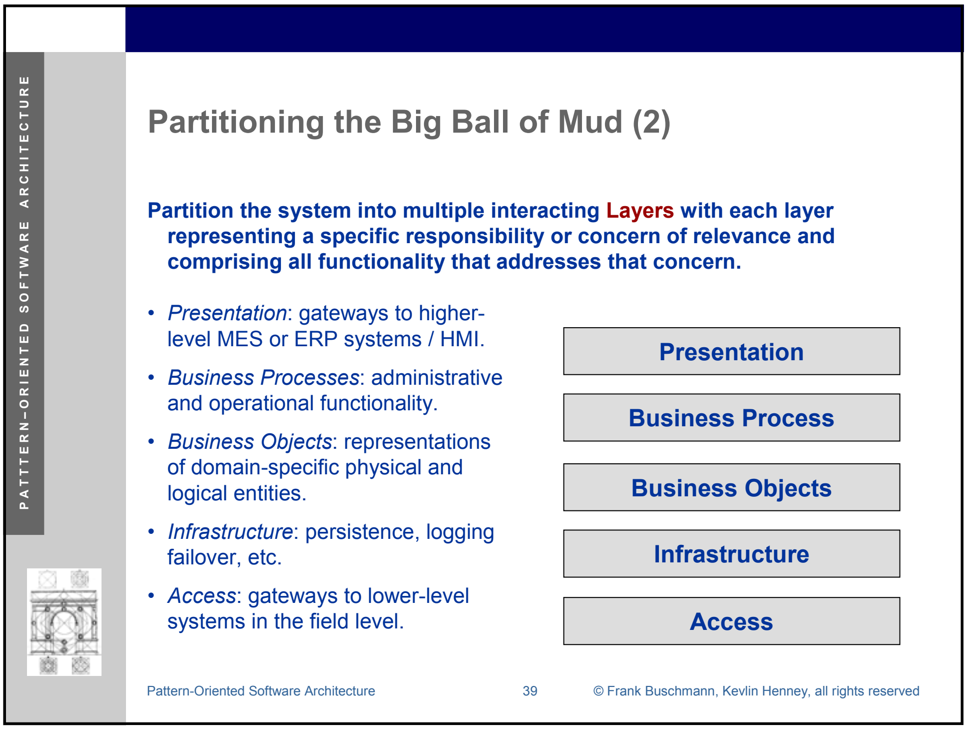
Business Object Layer

Database Access Layer

Infrastructure Layer

Solution

- Define one or more layers for the software under development with each layer having a distinct and specific responsibility, for instance, regarding abstraction, granularity, or rate of change.



PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

Partitioning the Big Ball of Mud (2)

Partition the system into multiple interacting Layers with each layer representing a specific responsibility or concern of relevance and comprising all functionality that addresses that concern.

- *Presentation*: gateways to higher-level MES or ERP systems / HMI.
- *Business Processes*: administrative and operational functionality.
- *Business Objects*: representations of domain-specific physical and logical entities.
- *Infrastructure*: persistence, logging failover, etc.
- *Access*: gateways to lower-level systems in the field level.

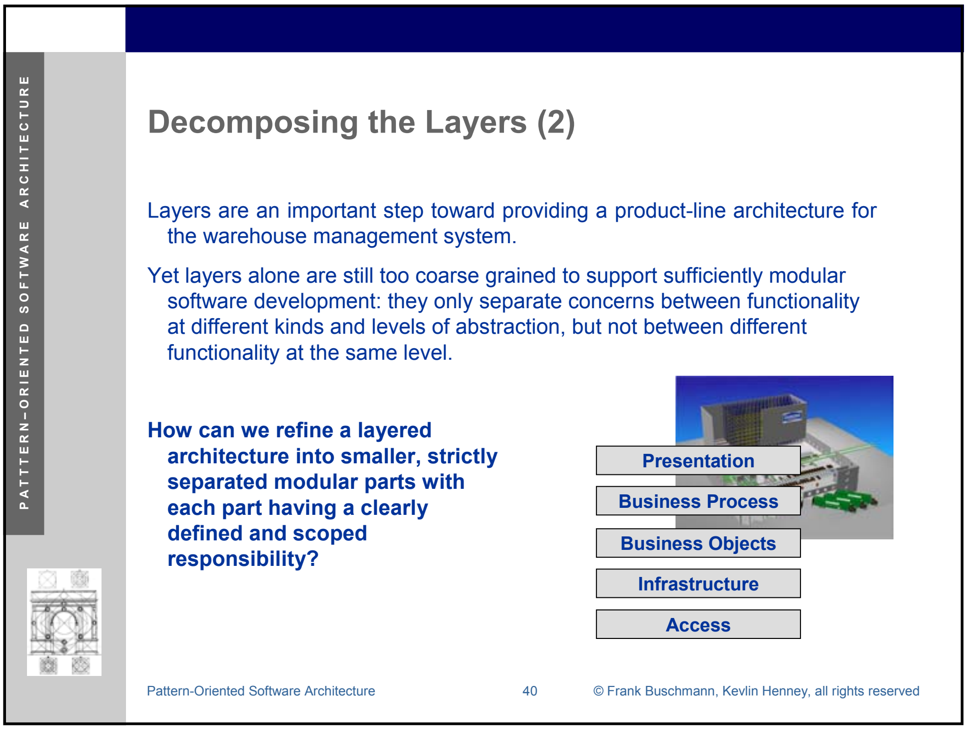
Presentation

Business Process

Business Objects

Infrastructure

Access



PATTERN–ORIENTED SOFTWARE ARCHITECTURE

Decomposing the Layers (2)

Layers are an important step toward providing a product-line architecture for the warehouse management system.

Yet layers alone are still too coarse grained to support sufficiently modular software development: they only separate concerns between functionality at different kinds and levels of abstraction, but not between different functionality at the same level.

How can we refine a layered architecture into smaller, strictly separated modular parts with each part having a clearly defined and scoped responsibility?

Presentation

Business Process

Business Objects

Infrastructure

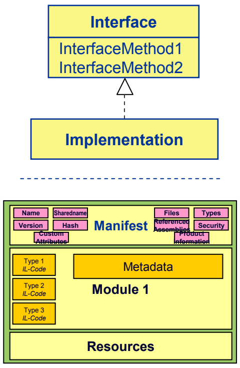
Access

Domain Object at a Glance

Problem

How can we partition the functionality in a system such that:

- Each self-contained unit of functionality is decoupled as much as possible from other self-contained units of functionality.
- Any unit of self-contained functionality can evolve at different times without incurring ripple effects.



Two forms of domain object

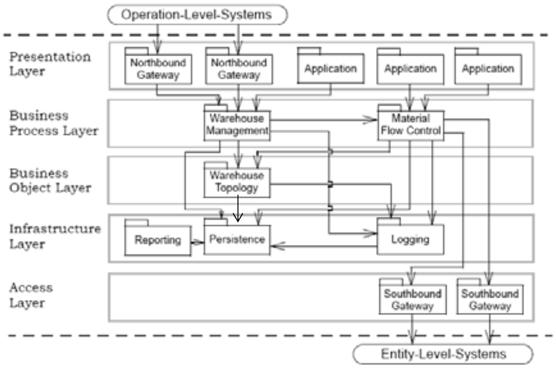
Solution

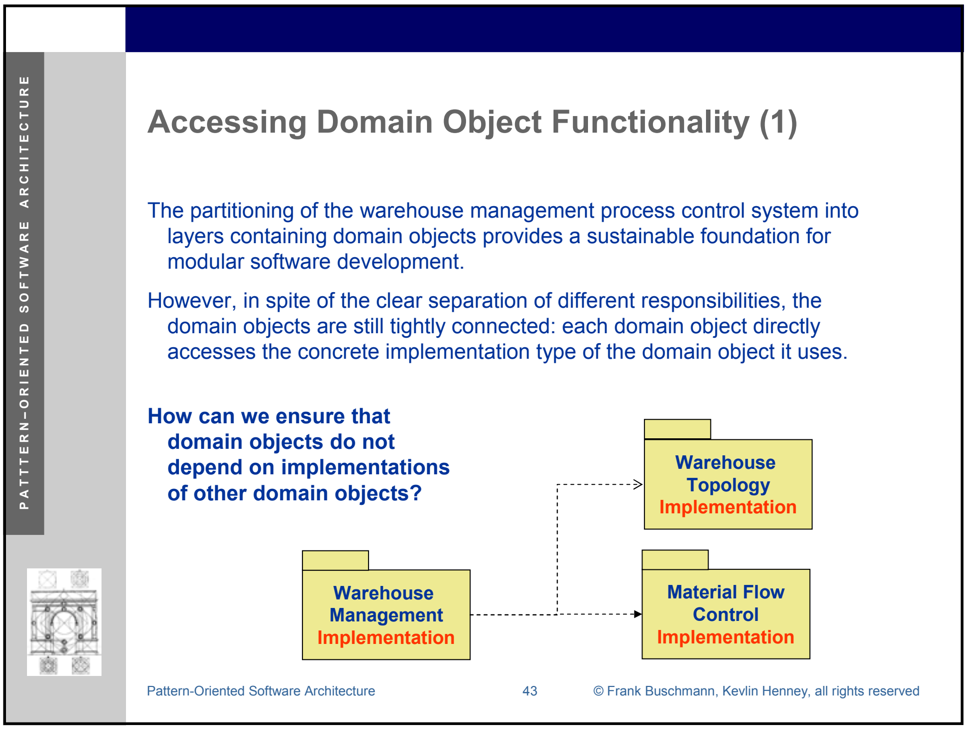
Encapsulate each self-contained functionality into a domain object

- An interface provides the accessible methods of the domain object.
- An implementation realizes the offered functionality of the domain object.

Decomposing the Layers (2)

Provide a Domain Object for each self-contained, coherent, and functionally related responsibility within a layered design.





PATTTERN-ORIENTED SOFTWARE ARCHITECTURE

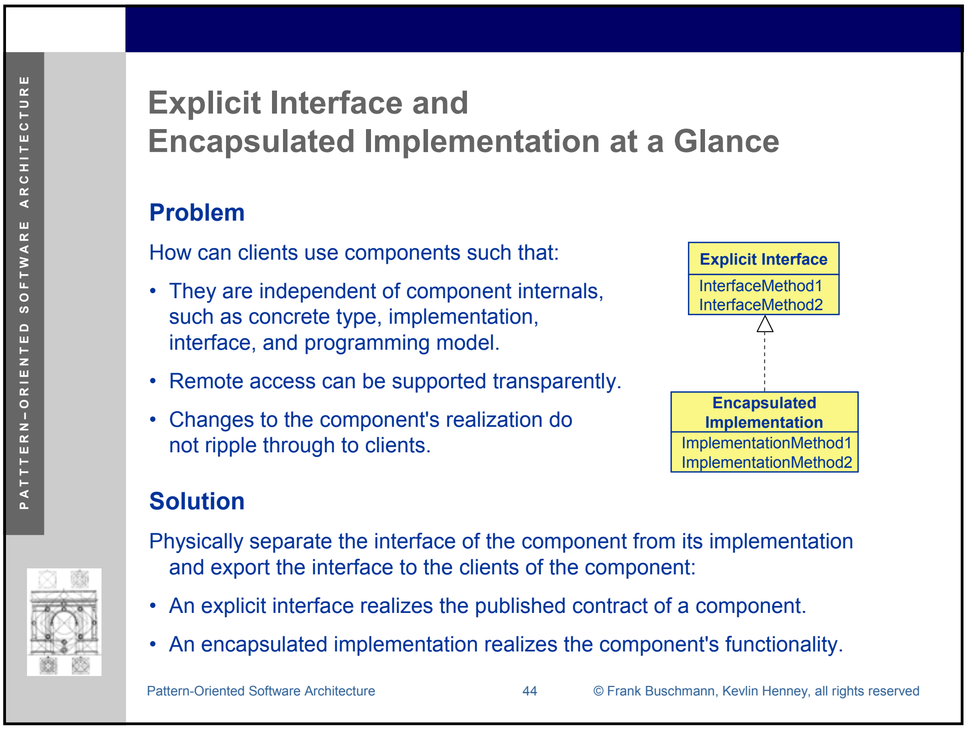
Accessing Domain Object Functionality (1)

The partitioning of the warehouse management process control system into layers containing domain objects provides a sustainable foundation for modular software development.

However, in spite of the clear separation of different responsibilities, the domain objects are still tightly connected: each domain object directly accesses the concrete implementation type of the domain object it uses.

How can we ensure that domain objects do not depend on implementations of other domain objects?

Warehouse Topology Implementation
Warehouse Management Implementation
Material Flow Control Implementation



PATTERN-ORIENTED SOFTWARE ARCHITECTURE

Explicit Interface and Encapsulated Implementation at a Glance

Problem

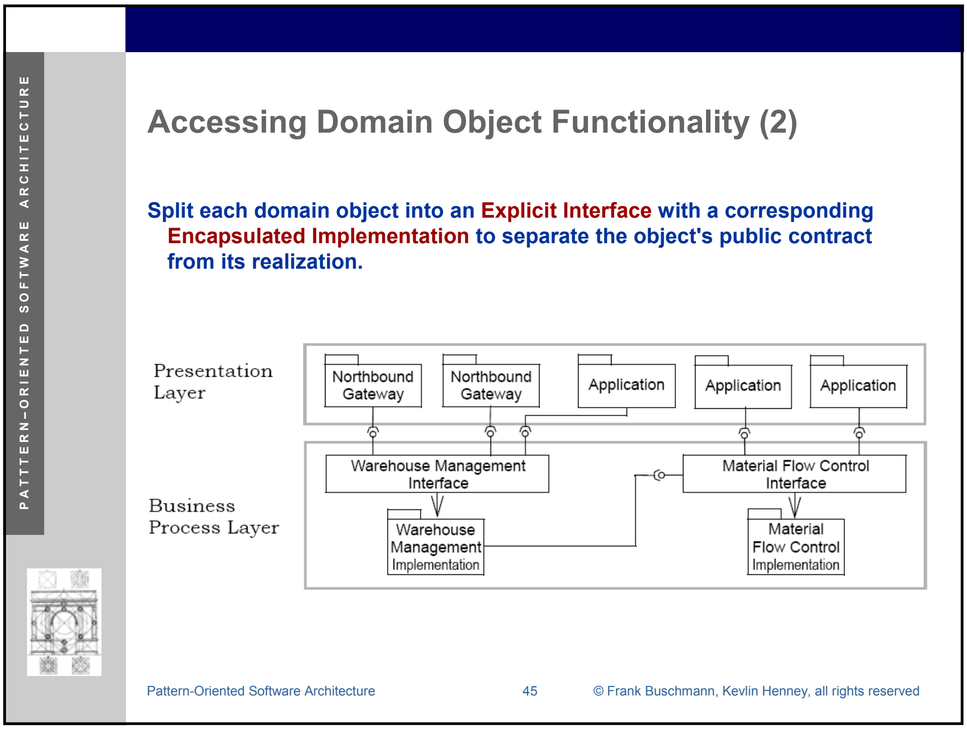
How can clients use components such that:

- They are independent of component internals, such as concrete type, implementation, interface, and programming model.
- Remote access can be supported transparently.
- Changes to the component's realization do not ripple through to clients.

Solution

Physically separate the interface of the component from its implementation and export the interface to the clients of the component:

- An explicit interface realizes the published contract of a component.
- An encapsulated implementation realizes the component's functionality.

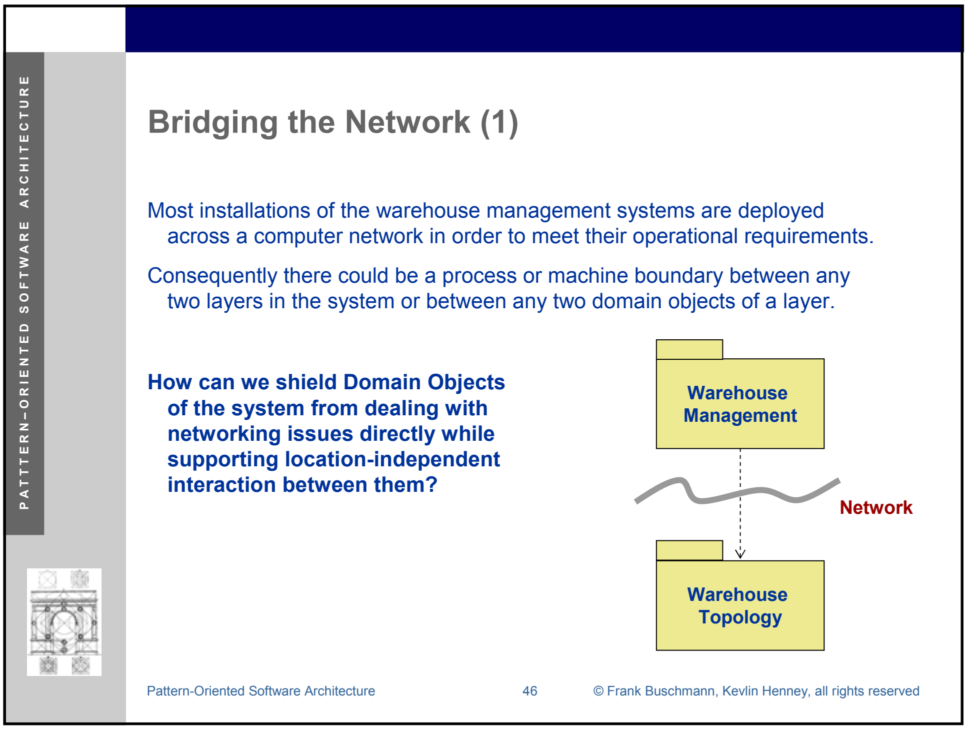


PATTTERN-ORIENTED SOFTWARE ARCHITECTURE

Accessing Domain Object Functionality (2)

Split each domain object into an Explicit Interface with a corresponding Encapsulated Implementation to separate the object's public contract from its realization.

Presentation Layer
Northbound Gateway
Northbound Gateway
Application
Application
Application
Business Process Layer
Warehouse Management Interface
Material Flow Control Interface
Warehouse Management Implementation
Material Flow Control Implementation



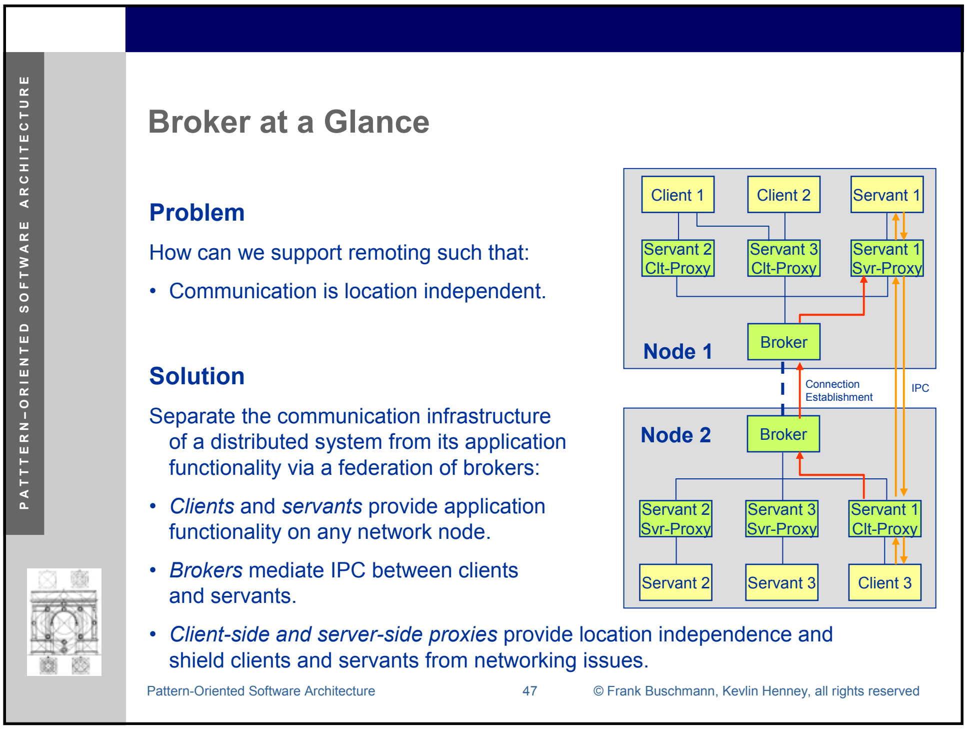
PATTTERN-ORIENTED SOFTWARE ARCHITECTURE

Bridging the Network (1)

Most installations of the warehouse management systems are deployed across a computer network in order to meet their operational requirements.

Consequently there could be a process or machine boundary between any two layers in the system or between any two domain objects of a layer.

How can we shield Domain Objects of the system from dealing with networking issues directly while supporting location-independent interaction between them?



PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

Broker at a Glance

Problem

How can we support remoting such that:

- Communication is location independent.

Solution

Separate the communication infrastructure of a distributed system from its application functionality via a federation of brokers:

- *Clients* and *servants* provide application functionality on any network node.
- *Brokers* mediate IPC between clients and servants.
- *Client-side and server-side proxies* provide location independence and shield clients and servants from networking issues.

Client 1 | Client 2 | Servant 1

Servant 2 Clt-Proxy | Servant 3 Clt-Proxy | Servant 1 Svr-Proxy

Node 1 | Broker

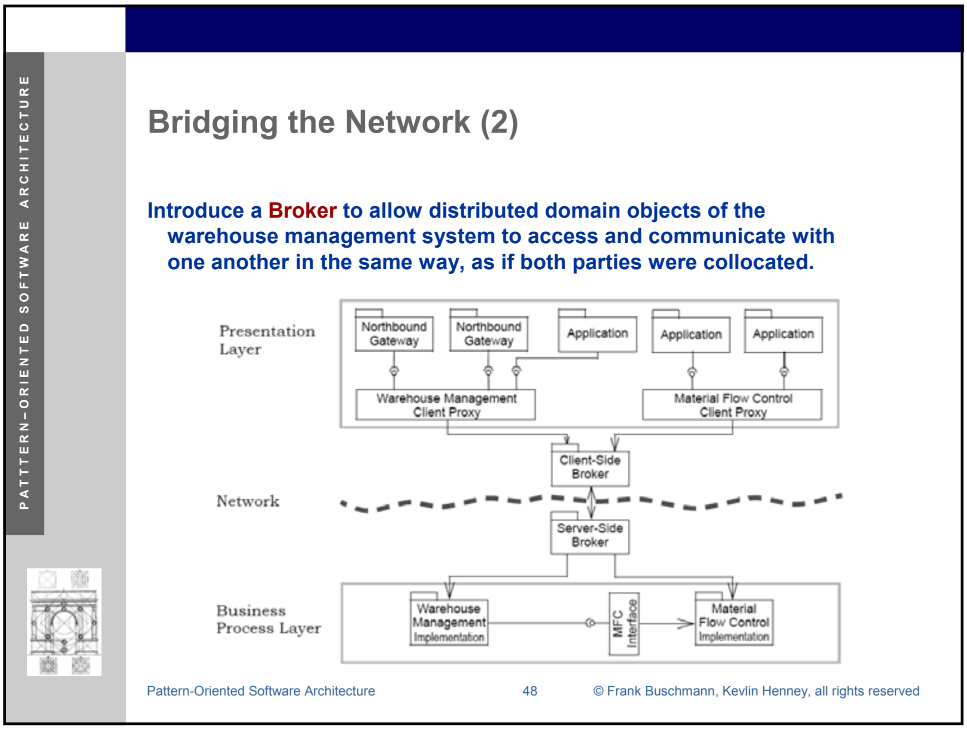
Connection Establishment | IPC

Node 2 | Broker

Servant 2 Svr-Proxy | Servant 3 Svr-Proxy | Servant 1 Clt-Proxy

Servant 2 | Servant 3 | Client 3

Pattern-Oriented Software Architecture 47 © Frank Buschmann, Kevlin Henney, all rights reserved



PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

Bridging the Network (2)

Introduce a Broker to allow distributed domain objects of the warehouse management system to access and communicate with one another in the same way, as if both parties were collocated.

Presentation Layer

Northbound Gateway | Northbound Gateway | Application | Application | Application

Warehouse Management Client Proxy | Material Flow Control Client Proxy

Client-Side Broker

Network

Server-Side Broker

Business Process Layer

Warehouse Management Implementation | MFC Interface | Material Flow Control Implementation

Pattern-Oriented Software Architecture 48 © Frank Buschmann, Kevlin Henney, all rights reserved

Separating User Interface (1)

Domain objects of the presentation layer neither implement any business logic nor maintain any business state—both responsibilities are assigned to other layers of the system that are accessed by the presentation layer.

Nevertheless, the information that is presented should be up-to-date, regardless of the number and location of client.

How can we ensure that the domain objects in the presentation layer always provide fresh and timely state information to their clients?

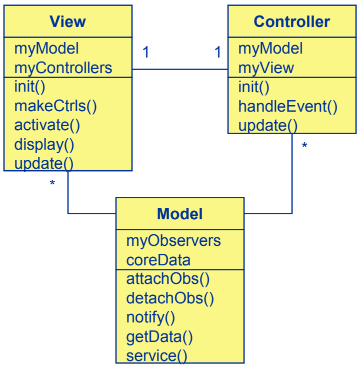
Warehouse Management System?

Model-View-Controller at a Glance

Problem

How can we develop user interfaces that:

- Support adaptation and change without affecting the application’s functional core.
- Display the current state of computation and respond to state changes immediately.



View
myModel
myControllers
init()
makeCtrls()
activate()
display()
update()

1

1

Controller
myModel
myView
init()
handleEvent()
update()

*

*

Model
myObservers
coreData
attachObs()
detachObs()
notify()
getData()
service()

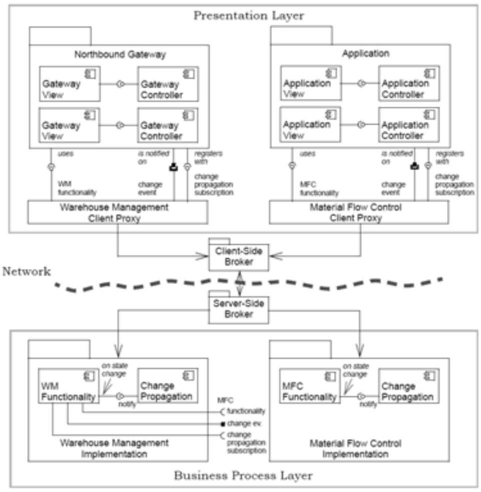
Solution

Divide an interactive application into three loosely coupled parts:

- A *model* object encapsulates an entity from the application's functional core.
- *Views* present data and information to the user.
- *Controllers* are associated with views and allow manipulation of the presented data and information.

Separating User Interface (2)

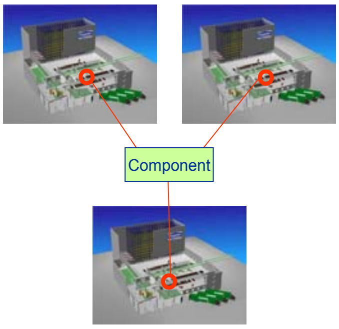
Use a Model-View-Controller arrangement to minimize the coupling between domain objects in the presentation layer and domain objects in the business layers, and ensure their efficient cooperation and mutual consistency.



Distributing Functionality (1)

While having a single and central implementation of each domain object is the simplest possible distribution model, it may, however, be insufficient for domain objects to meet their performance and scalability requirements.

Replication may also not help – especially in case of stateful domain objects.



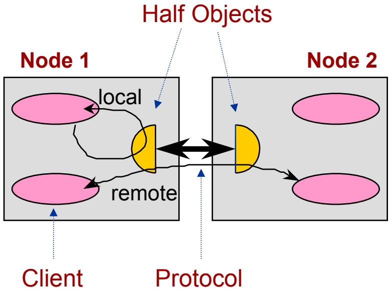
How can we provide efficient access to a domain object that maintains global state and whose clients reside in multiple address spaces?

Half-Object Plus Protocol at a Glance

Problem

How can we design objects in a distributed system:

- such that access to them and the execution of their services incur minimal performance penalties?

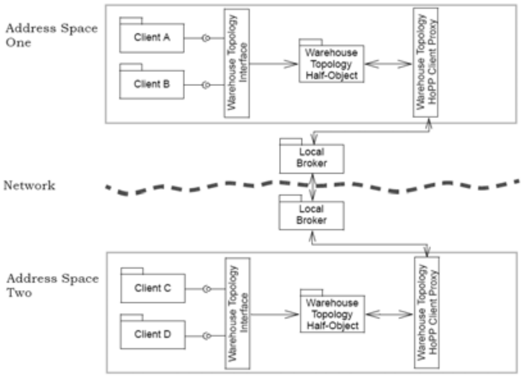


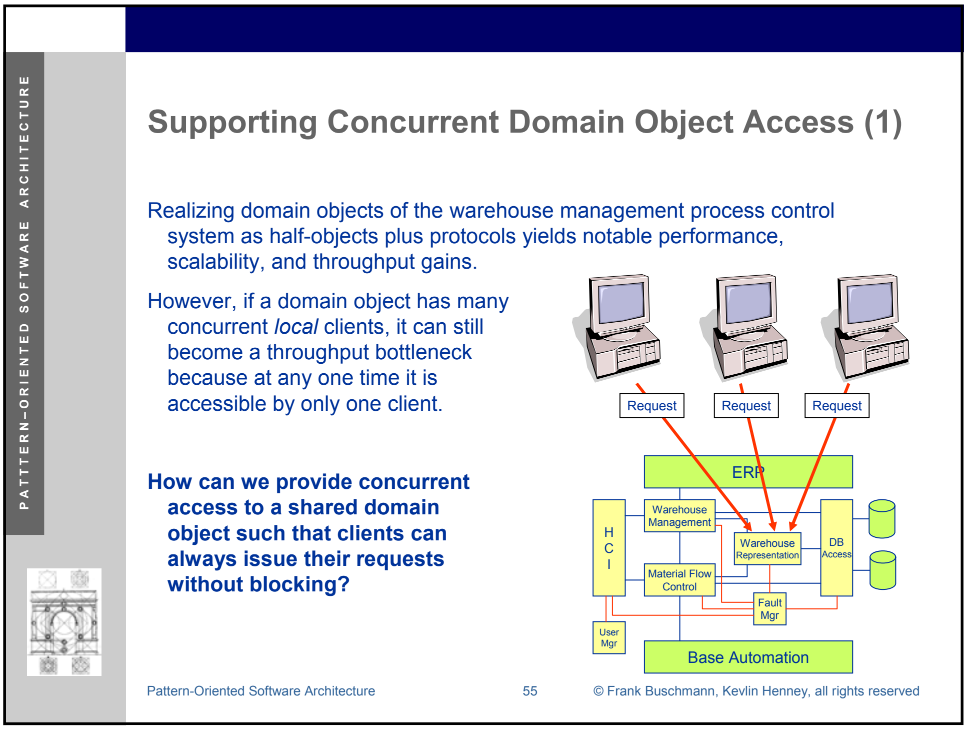
Solution

- Split an object into multiple *halves* – one half per address space from which the object is accessed.
- Within each half, fully implement all functionality that can be executed locally without using the network.
- Implement functionality that crosses address spaces partially in each half and coordinate the halves via a *protocol*.

Distributing Functionality (2)

Realize the domain object as a Half-Object plus Protocol that splits its functionality into a set of half-objects, with one half-object collocated in each client address space.


Address Space One
Client A
Client B
Warehouse Topology Interface
Warehouse Topology Half-Object
Warehouse Topology HoPP Client Proxy
Local Broker
Network
Local Broker
Address Space Two
Client C
Client D
Warehouse Topology Interface
Warehouse Topology Half-Object
Warehouse Topology HoPP Client Proxy



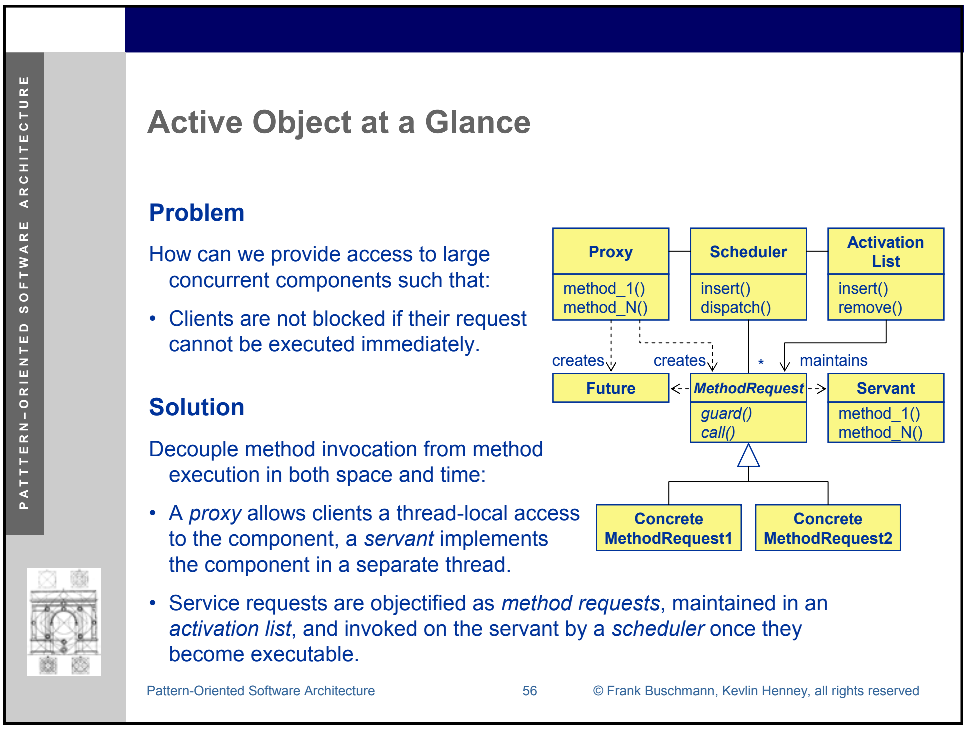
PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

Supporting Concurrent Domain Object Access (1)

Realizing domain objects of the warehouse management process control system as half-objects plus protocols yields notable performance, scalability, and throughput gains.

However, if a domain object has many concurrent *local* clients, it can still become a throughput bottleneck because at any one time it is accessible by only one client.

How can we provide concurrent access to a shared domain object such that clients can always issue their requests without blocking?



PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

Active Object at a Glance

Problem

How can we provide access to large concurrent components such that:

- Clients are not blocked if their request cannot be executed immediately.

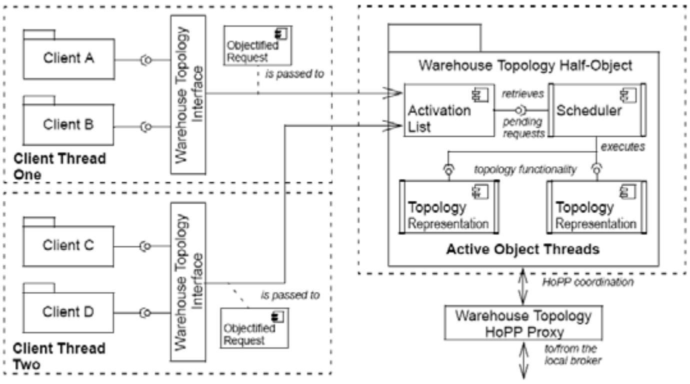
Solution

Decouple method invocation from method execution in both space and time:

- A *proxy* allows clients a thread-local access to the component, a *servant* implements the component in a separate thread.
- Service requests are objectified as *method requests*, maintained in an *activation list*, and invoked on the servant by a *scheduler* once they become executable.

Supporting Concurrent Domain Object Access (2)

Realize the domain object as an Active Object that separates request invocation from request execution in space and time.



Client A

Client B

Client Thread One

Warehouse Topology Interface

Objectified Request

is passed to

Client C

Client D

Client Thread Two

Warehouse Topology Interface

is passed to

Objectified Request

Warehouse Topology Half-Object

Activation List

retrieves

pending requests

Scheduler

executes

topology functionality

Topology Representation

Topology Representation

Active Object Threads

HoPP coordination

Warehouse Topology HoPP Proxy

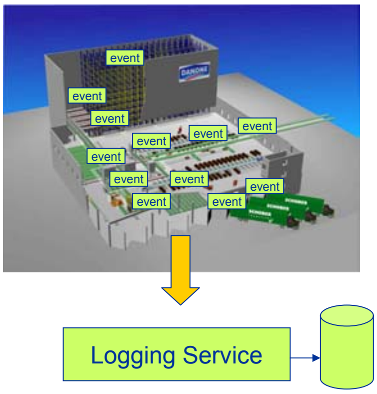
to/from the local broker

Achieving Scalable Concurrency (1)

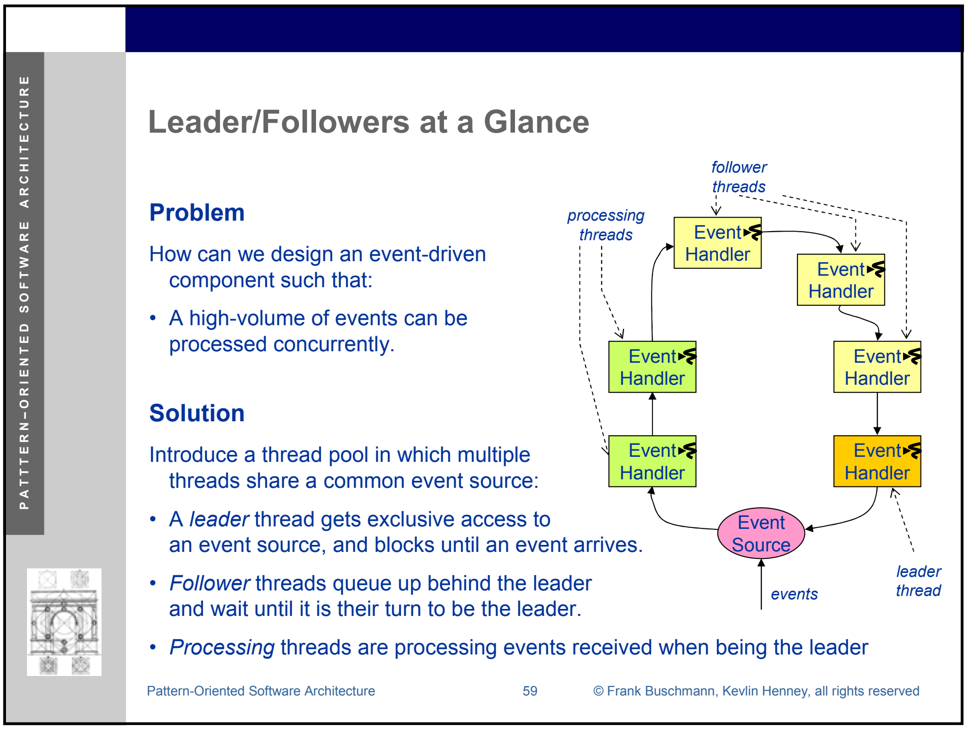
Various types of information that the domain objects of the warehouse management system can generate, for instance, errors and traces, are logged for later evaluation.

However, using the logging functionality should only have a minimal impact on the system's operational qualities.

How can we avoid having logging become a performance bottleneck, especially when a high number of logging records must be processed?



Logging Service



PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

Leader/Followers at a Glance

Problem

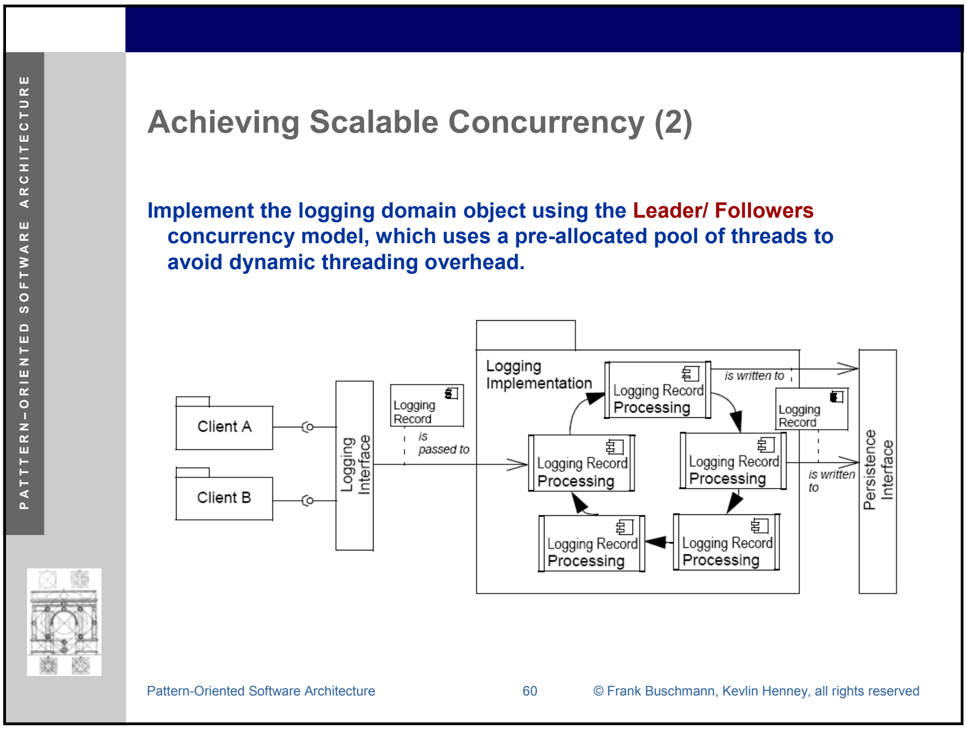
How can we design an event-driven component such that:

- A high-volume of events can be processed concurrently.

Solution

Introduce a thread pool in which multiple threads share a common event source:

- A *leader* thread gets exclusive access to an event source, and blocks until an event arrives.
- *Follower* threads queue up behind the leader and wait until it is their turn to be the leader.
- *Processing* threads are processing events received when being the leader



PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

Achieving Scalable Concurrency (2)

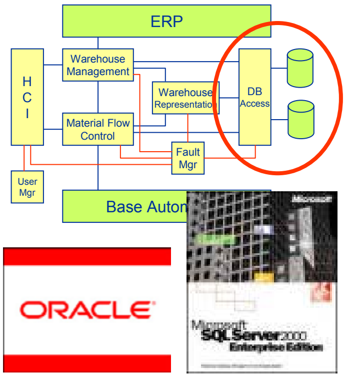
Implement the logging domain object using the Leader/ Followers concurrency model, which uses a pre-allocated pool of threads to avoid dynamic threading overhead.

Crossing the OR Divide (1)

All data created and maintained by the warehouse management process control system must be stored persistently, and changed transactionally, using a relational database.

However, domain objects should not become dependent on the relational paradigm.

How can we bridge the chasm between the object-oriented view—used within the warehouse management system—and the relational view—required by the database—without exposing each view to the other side?

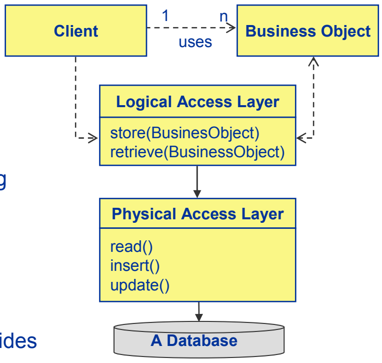


Database Access Layer at a Glance

Problem

How can we bridge the OR divide so that:

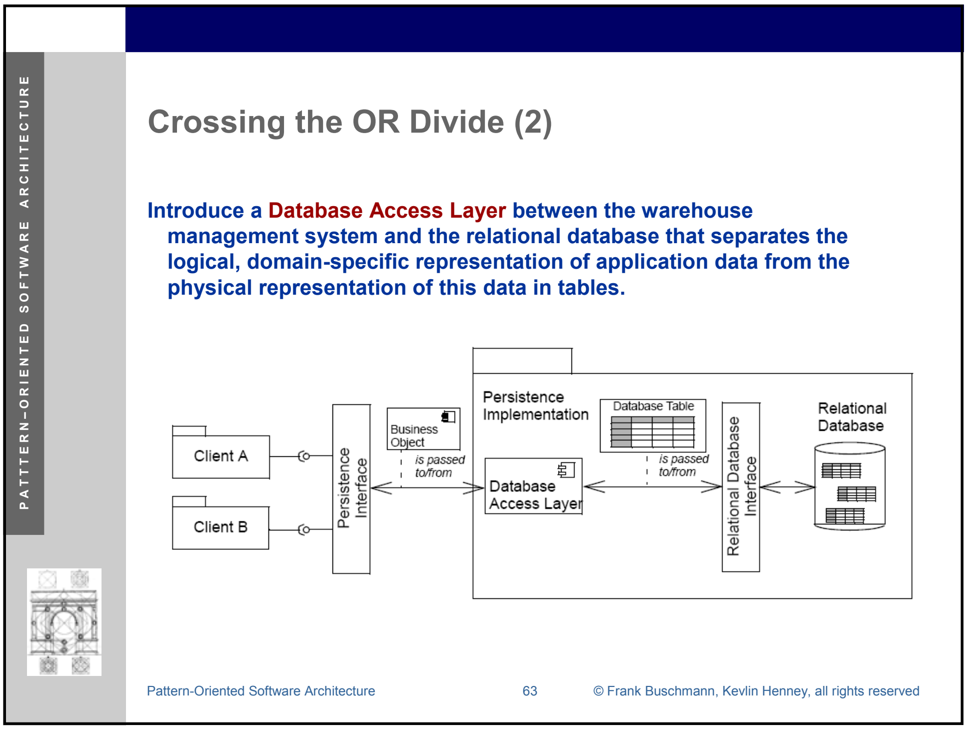
- Each side—application and database—can benefit from the most appropriate and familiar computational model without being dependent on the paradigm used by the other side.



Solution

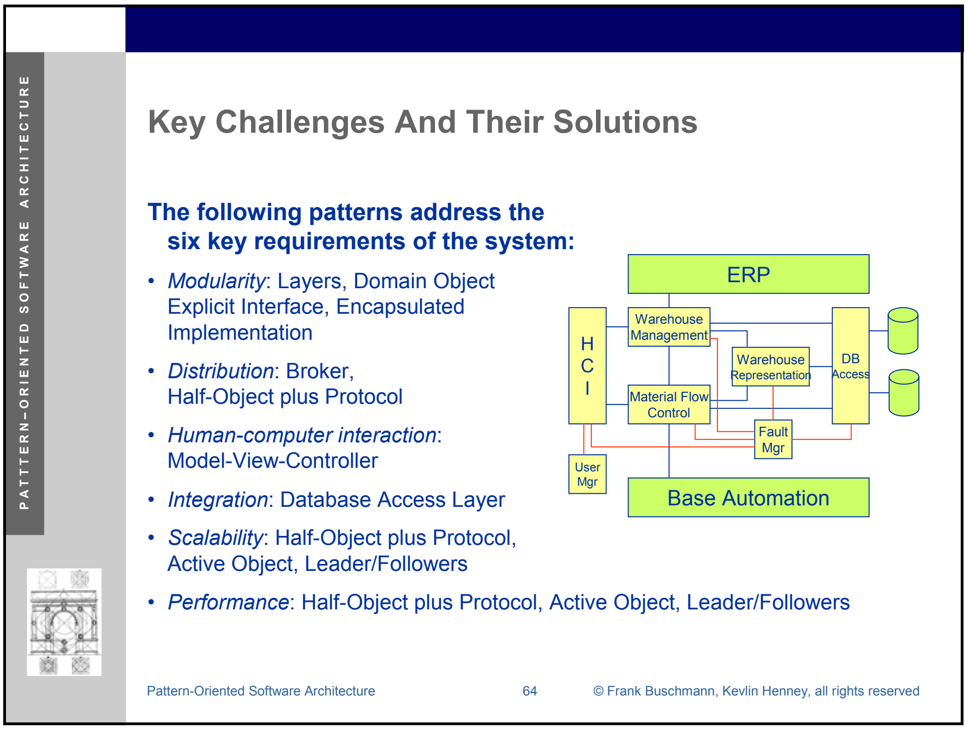
Introduce a database access layer that provides a bidirectional OR mapping:

- A *logical access layer* allows clients to store and retrieve application-level business objects, and provides caching and transactions.
- A *physical access layer* represents the interface to a concrete database and provides functionality for optimized database access.



Crossing the OR Divide (2)

Introduce a Database Access Layer between the warehouse management system and the relational database that separates the logical, domain-specific representation of application data from the physical representation of this data in tables.



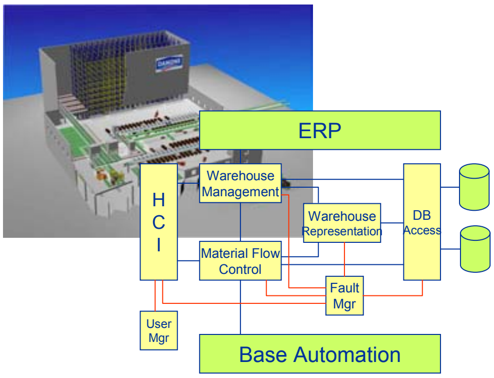
Key Challenges And Their Solutions

The following patterns address the six key requirements of the system:

- *Modularity*: Layers, Domain Object Explicit Interface, Encapsulated Implementation
- *Distribution*: Broker, Half-Object plus Protocol
- *Human-computer interaction*: Model-View-Controller
- *Integration*: Database Access Layer
- *Scalability*: Half-Object plus Protocol, Active Object, Leader/Followers
- *Performance*: Half-Object plus Protocol, Active Object, Leader/Followers

The End

And they happily lived ever after!



Pattern Sequences

Pattern Sequences

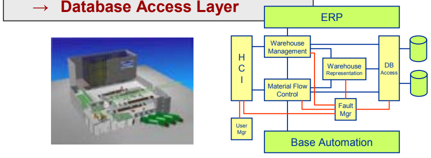
- Remove the story from a pattern story and describe how a type of software system, subsystem, or component can be developed with patterns using a particular solution approach or architectural style.

They discuss:

- What type of problems are to be resolved in what typical order.
- How the selected patterns are typically integrated with one another according to the envisioned style or approach.

Layers
- → **Domain Object**
- → **Explicit Interface**
- → **Encapsulated Implementation**
- → **Broker**
- → **Model-View-Controller**
- → **Half-Object plus Protocol**
- → **Active Object**
- → **Leader / Followers**
- → **Database Access Layer**

creates



Support for Designing With Patterns (1)

Pattern stories and pattern sequences support designing with patterns… but:

- Although they consider interdependencies and role-based integration of patterns, from a general and systems perspective they still fail to support fully a truly pattern-based software development!

Pattern Stories:
☺ Describe the pattern-based software development of a specific system (part).
☹ Describe one system instance only.

Pattern Sequences:
☺ Describe the pattern-based software development of a type of system (part).
☹ Cover one architectural style only.

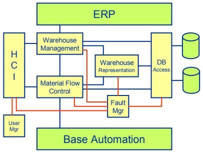
Support for Designing With Patterns (2)

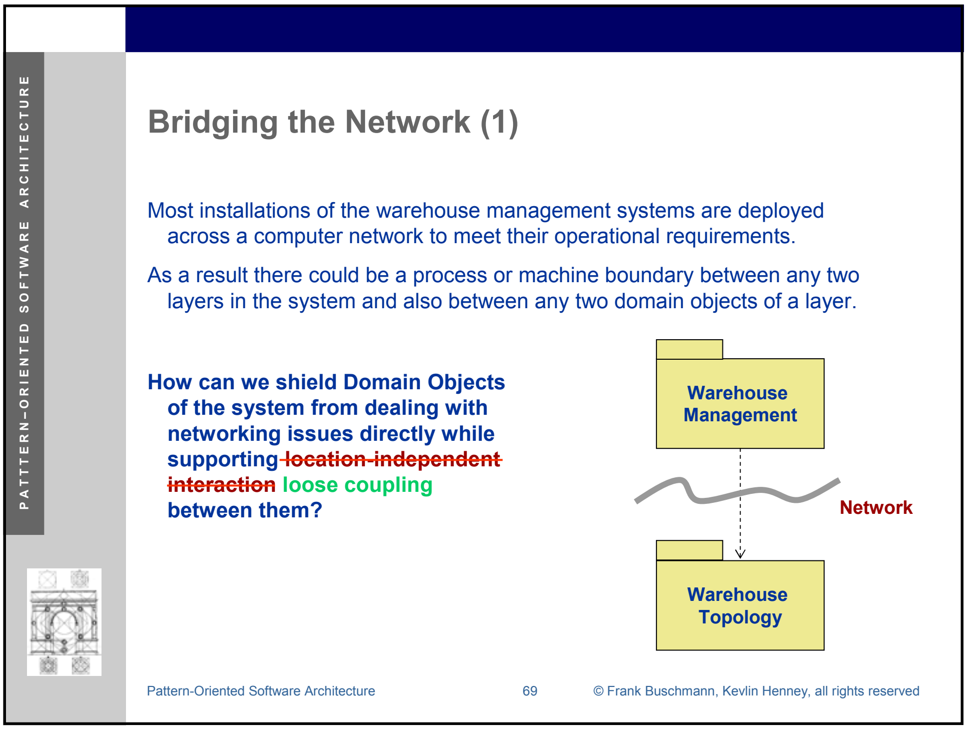
But: Different pattern sequences for the same type of system describe different, alternative architectural styles, each addressing different considerations and trade-offs!

Layers
→ Domain Object
→ Explicit Interface
→ Encapsulated Implementation
→ Broker
→ Model-View-Controller
→ Half-Object plus Protocol
→ Active Object
→ Leader / Followers

→ Database Access Layer

Layers
→ Domain Object
→ Explicit Interface
→ Encapsulated Implementation
→ Messaging
→ Presentation-Abstraction-Control
→ Replicated Component Group
→ Active Object
→ Half-Sync / Half-Async
→ Monitor Object
→ Database Access Layer



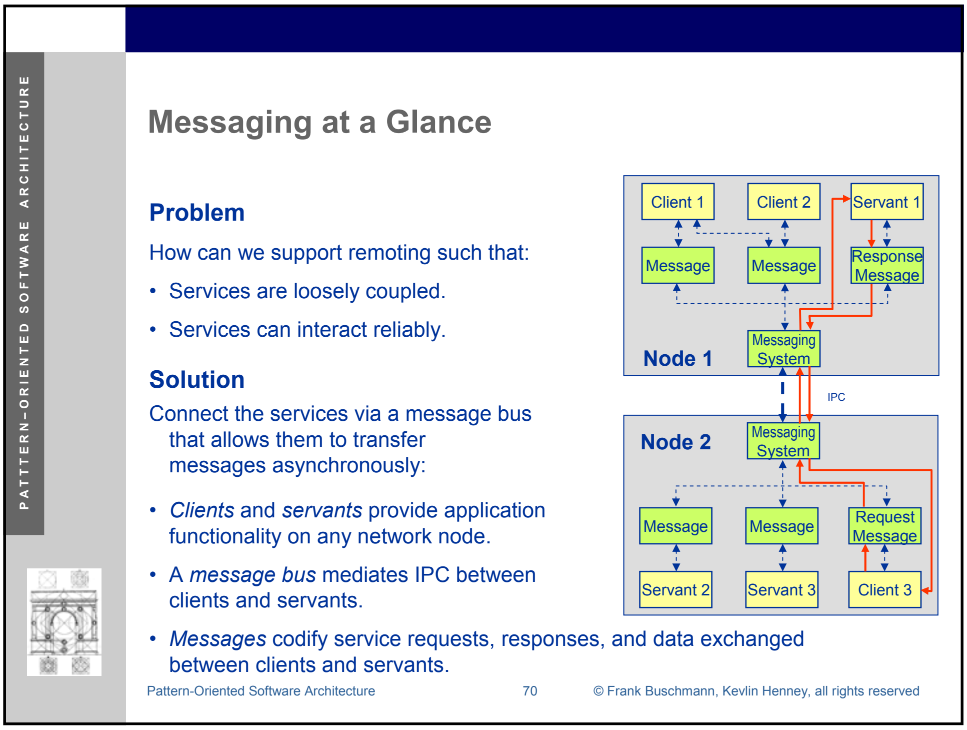

PATTERN-ORIENTED SOFTWARE ARCHITECTURE

Bridging the Network (1)

Most installations of the warehouse management systems are deployed across a computer network to meet their operational requirements.

As a result there could be a process or machine boundary between any two layers in the system and also between any two domain objects of a layer.

How can we shield Domain Objects of the system from dealing with networking issues directly while supporting ~~location-independent interaction~~ loose coupling between them?


PATTERN-ORIENTED SOFTWARE ARCHITECTURE

Messaging at a Glance

Problem

How can we support remoting such that:

- Services are loosely coupled.
- Services can interact reliably.

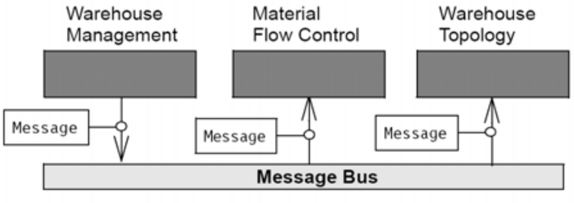
Solution

Connect the services via a message bus that allows them to transfer messages asynchronously:

- *Clients* and *servants* provide application functionality on any network node.
- A *message bus* mediates IPC between clients and servants.
- *Messages* codify service requests, responses, and data exchanged between clients and servants.

Bridging the Network (2)

Introduce a Messaging system to allow distributed domain objects of the warehouse management system to exchange messages codifying service requests and responses.



- Design the explicit interfaces of services so that they can receive request messages asynchronously.
- Realize encapsulated implementations of the services so that they dispatch messages on concrete functions and deliver response messages asynchronously to message senders.

Separating User Interface (1)

Domain objects of the presentation layer neither implement any business logic nor maintain any business state—both responsibilities are assigned to other layers of the system that are accessed by the presentation layer.

Nevertheless, the information that is presented should be up-to-date, regardless of the number and location of client.

How can we ensure that the domain objects in the presentation layer always provide fresh and timely state information to their clients and that each subsystem offers its own UI paradigm?

Presentation-Abstraction-Control at a Glance

Problem

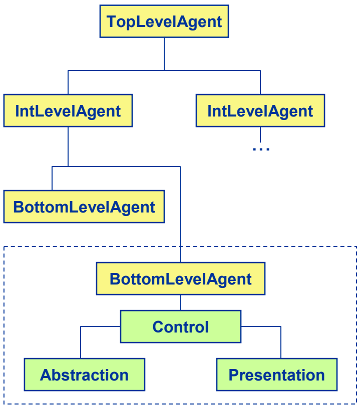
How can we design user interfaces that:

- Support adaptation and change without affecting the application's functional core.
- Support subsystem-specific UI paradigms.

Solution

Divide an interactive application into loosely coupled agents:

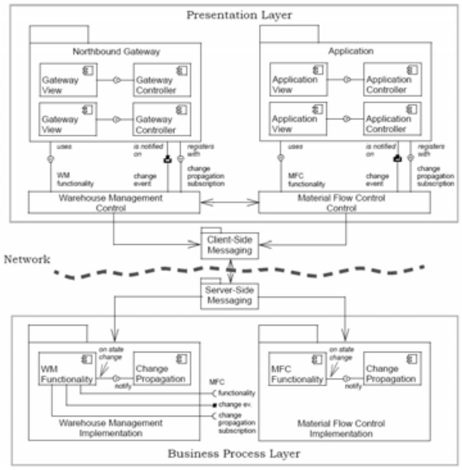
- Each *agent* is responsible for a providing a self-contained part of the system's functionality—including a specialized UI
- An agent consists of three parts: a *presentation* realizes the agent's UI; an *abstraction* its domain functionality; a *control* connects the presentation and abstraction, and allows communication with other agents.



Separating User Interface (2)

Use a Presentation-Abstraction Control design to:

- **minimize coupling between domain objects in the presentation layer and domain objects in the business layers,**
- **ensure their efficient cooperation and mutual consistency, and**
- **allow each self-contained subsystem to provide its own user interface.**

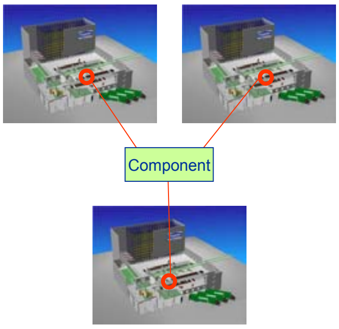


Distributing Functionality (1)

While having a single and central implementation of each domain object is the simplest possible distribution model, it may, however, be insufficient for domain objects to meet their ~~performance and scalability~~ availability requirements.

~~Replication may also not help – especially in case of stateful domain objects.~~

How can we provide an efficient yet highly available access to a domain object that maintains global state and whose clients reside in multiple address spaces?

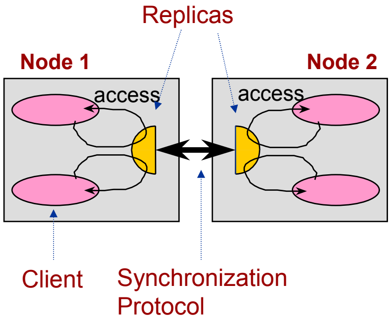


Replicated Component Group at a Glance

Problem

How can we design objects in a distributed system:

- such that they are highly available?

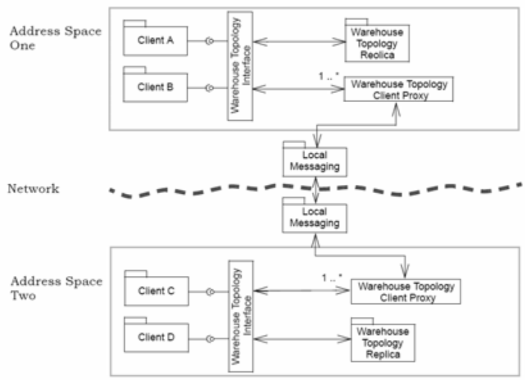


Solution

- Split a component into multiple *replicas* – one half per address space from which the object is accessed.
- Synchronize the state of the replicas via a *protocol*.

Distributing Functionality (2)

Realize the domain object as a Replicated Component Group that replicates its functionality onto multiple component instances, with one instance collocated in each client address space.

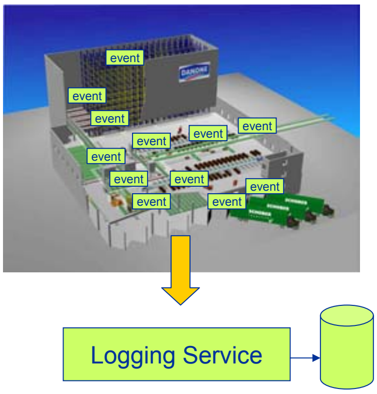

Address Space One
Client A
Client B
Warehouse Topology Interface
Warehouse Topology Replica
1 .. *
Warehouse Topology Client Proxy
Local Messaging
Network
Local Messaging
Address Space Two
Client C
Client D
Warehouse Topology Interface
1 .. *
Warehouse Topology Client Proxy
Warehouse Topology Replica

Achieving Scalable Concurrency (1)

Various types of information that the domain objects of the warehouse management system can generate, for instance, errors and traces, are logged for later evaluation.

However, using the logging functionality should only have a minimal impact on the system’s operational qualities.

How can we avoid having logging becomes a performance bottleneck~~, especially when a high number of logging records must be processed?~~

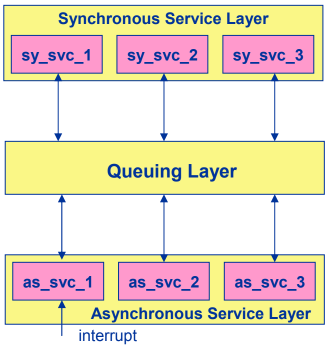


Half-Sync/Half-Async at a Glance

Problem

How can we design using asynchronous and synchronous processing such that:

- A reasonable volume of events can be processed concurrently.
- Asynchronous can be programmed with respect to performance.
- Synchronous services can offer a simple programming model.

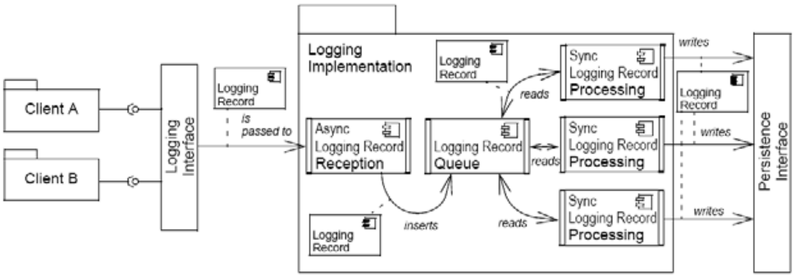


Solution

Separate synchronous and asynchronous services from another by dedicated layers and add a queuing layer between them to mediate communication between services.

Achieving Scalable Concurrency (2)

Implement the logging domain object using the Half-Sync/Half-Async concurrency model, which allows the service to receive logging records asynchronously but process them synchronously.

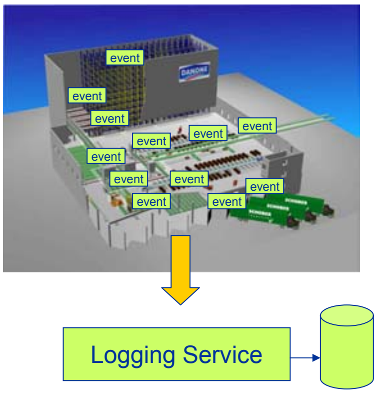


Implementing a Synchronized Request Queue (1)

At the heart of Half-Sync/Half-Async concurrency design is a logging record queue. To coordinate concurrent access to the queue, we must synchronize it.

- If 'simple' locking is used, threads can 'busy wait' when the queue is empty or full, which degrades performance.

How can the queue avoid race conditions or busy waiting when threads in different Half-Sync/Half-Async layers put and get logging records simultaneously?



Monitor Object at a glance

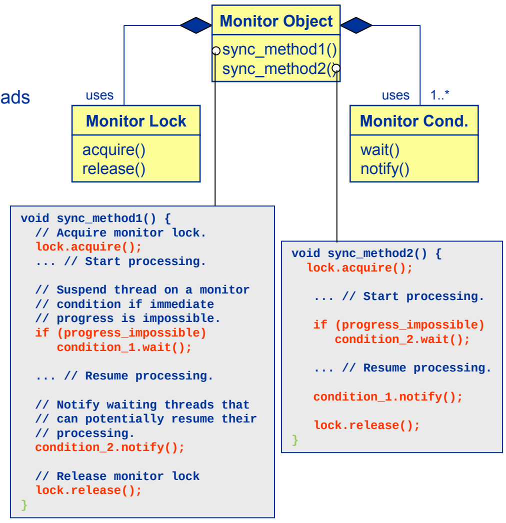
Problem

How can we coordinate multiple threads cooperatively?

Solution

Allow only one method to execute at any one time. Suspend the execution of methods that make no progress to allow other methods to execute.

- A *monitor object* is shared among multiple threads.
- *Synchronized methods* implement thread-safe functions.
- One or more *monitor conditions* allow the synchronized methods to schedule their execution using a *monitor lock*.

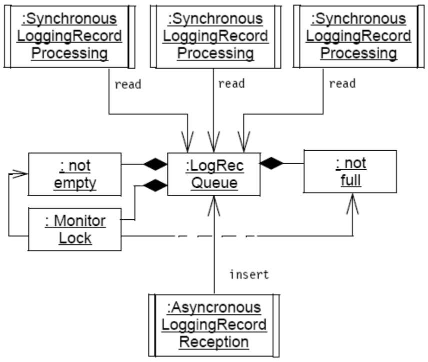


```
void sync_method1() {
  // Acquire monitor lock.
  lock.acquire();
  ... // Start processing.

  // Suspend thread on a monitor
  // condition if immediate
  // progress is impossible.
  if (progress_impossible)
     condition_1.wait();

  ... // Resume processing.

  // Notify waiting threads that
  // can potentially resume their
  // processing.
  condition_2.notify();

  // Release monitor lock
  lock.release();
}
```

```
void sync_method2() {
  lock.acquire();

   ... // Start processing.

   if (progress_impossible)
      condition_2.wait();

   ... // Resume processing.

   condition_1.notify();

   lock.release();
}
```

Achieving Scalable Concurrency (2)

Implement the queue as a **Monitor Object** to serialize concurrent method calls so that only one method runs at a time, allowing its methods to schedule their execution sequences cooperatively to prevent threads from busy waiting when the queue is full or empty.


:Synchronous LoggingRecord Processing
:Synchronous LoggingRecord Processing
:Synchronous LoggingRecord Processing
read
read
read
: not empty
:LogRec Queue
: not full
: Monitor Lock
insert
:Asyncronous LoggingRecord Reception

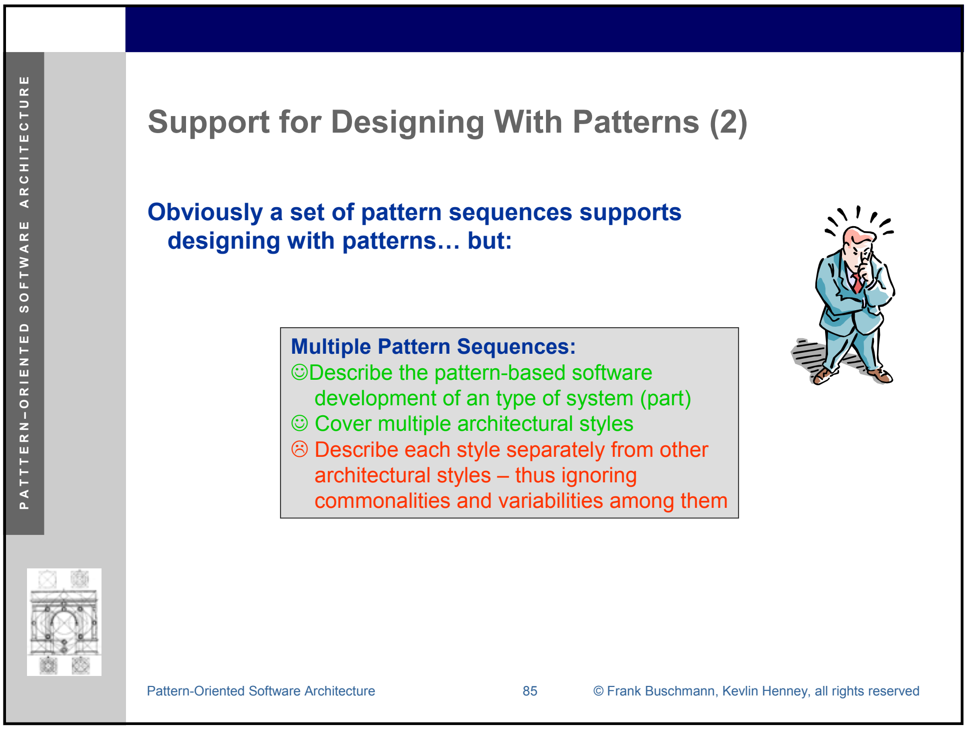
Support for Designing With Patterns (1)

The two pattern sequences address the following considerations and trade-offs – and thus, different architectural styles!

Layers
- → Domain Object
- → Explicit Interface
- → Encapsulated Implementation
- → Broker
- → Model-View-Controller
- → Half-Object plus Protocol
- → Active Object
- → Leader / Followers
- → Database Access Layer

Layers
- → Domain Object
- → Explicit Interface
- → Encapsulated Implementation
- → Messaging
- → Presentation-Abstraction-Control
- → Replicated Component Group
- → Active Object
- → Half-Sync / Half-Async
- → Monitor Object
- → Database Access Layer

Broker: Remote Method Invocation	Messaging: Message Exchange
MVC: One User Interface per System	PAC: One User Interface per Subsystem
HOPP: Federated State, High Availability	Replicated Object Group: Replicated State, Extreme Availability
Leader/Followers: Very High Volume Logging	Half-Sync / Half-Async & Monitor Object: High Volume Logging



Support for Designing With Patterns (2)

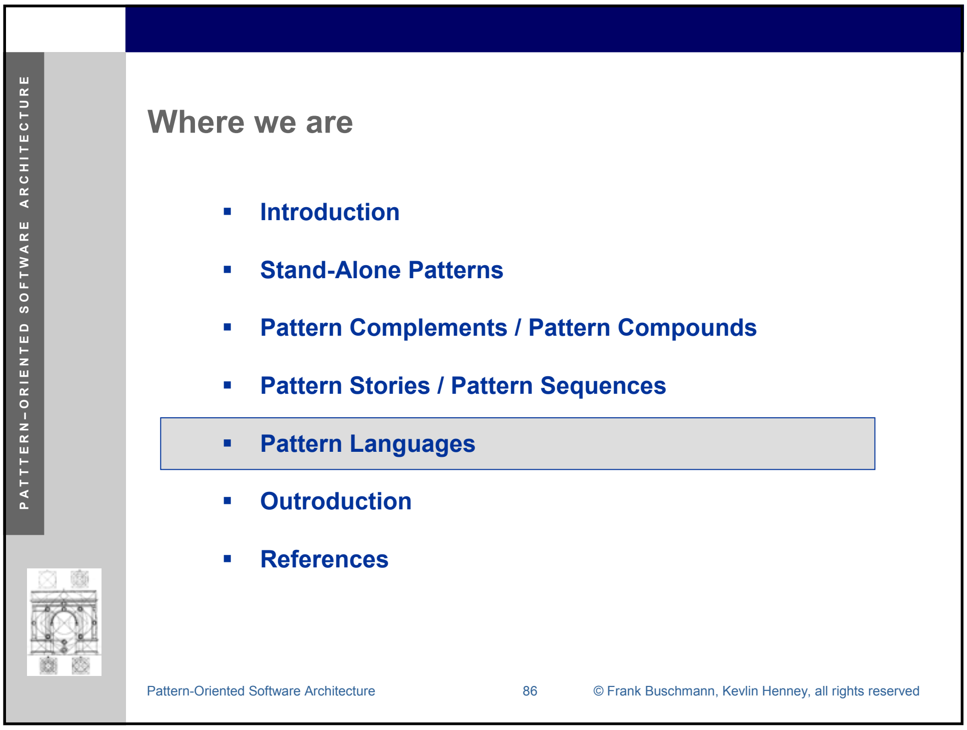
Obviously a set of pattern sequences supports designing with patterns… but:

Multiple Pattern Sequences:

☺Describe the pattern-based software development of an type of system (part)

☺ Cover multiple architectural styles

☹ Describe each style separately from other architectural styles – thus ignoring commonalities and variabilities among them



Where we are

- **Introduction**
- **Stand-Alone Patterns**
- **Pattern Complements / Pattern Compounds**
- **Pattern Stories / Pattern Sequences**
- **Pattern Languages**
- **Outroduction**
- **References**

Towards Pattern Languages (1)

Pattern sequences come close to the idea of designing with patterns:

- Their scope is a system type.
- They describe the what and the how in building an instance of this system type.
- But they lack genericity, supporting only one narrow interpretation of an architectural style.

Intuitive idea: what if we integrate several pattern sequences for designing a system for a specific domain with one another?

Towards Pattern Languages (2)

Pattern Languages

- Integrate multiple pattern sequences that describe how a type of software system, subsystem, or component can be developed systematically with patterns according to different feasible architectural styles.

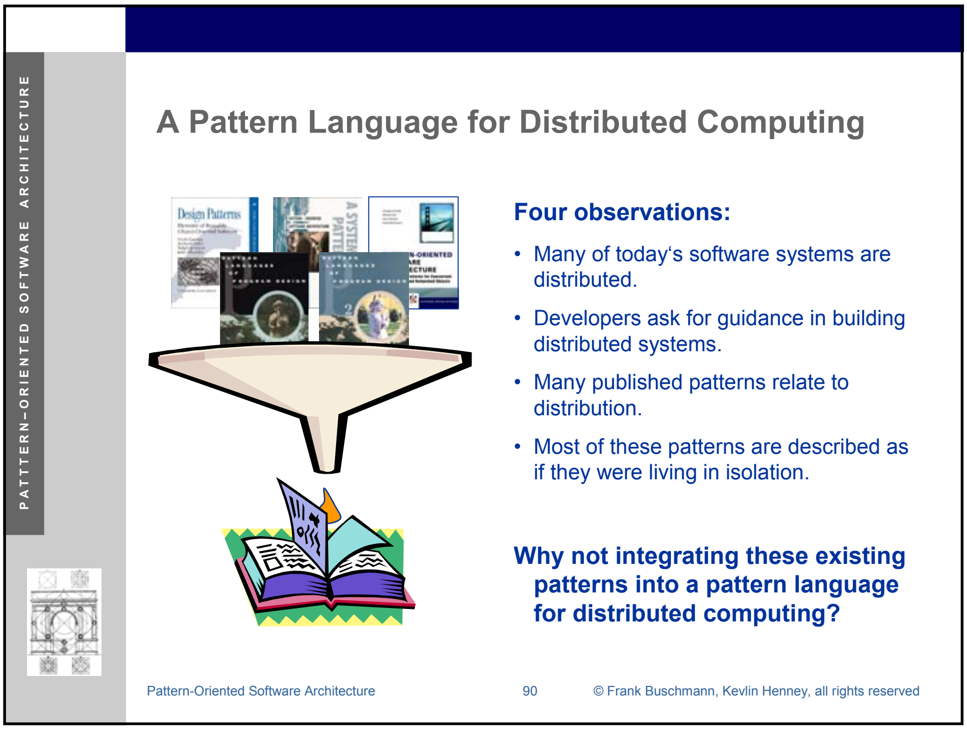
They discuss:

- What type of problems are to be resolved in what typical order.
- What alternative patterns help resolving the problems according to the envisioned architectural styles.
- How the selected patterns are typically integrated with one another according to the chosen architectural style.

Layers
→ Domain Object
→ Explicit Interface
→ Encapsulated Implementation
→ Broker | Messaging
→ Model-View-Controller | Presentation-Abstraction-Control
→ Half-Object plus Protocol | Replicated Component Group
→ Active Object
→ Leader / Followers | (Half-Sync/Half-Async → Monitor Object)
→ Database Access Layer

Pattern Languages

An Example



A Pattern Language for Distributed Computing

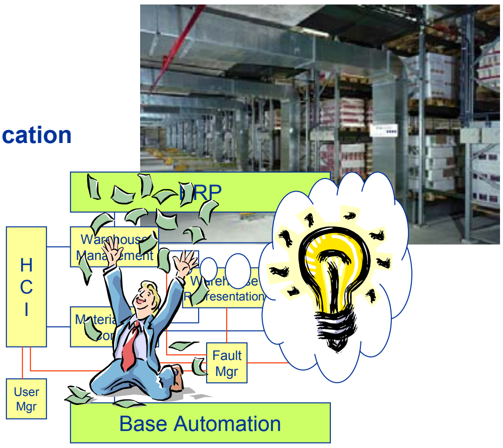
Four observations:

- Many of today‘s software systems are distributed.
- Developers ask for guidance in building distributed systems.
- Many published patterns relate to distribution.
- Most of these patterns are described as if they were living in isolation.

Why not integrating these existing patterns into a pattern language for distributed computing?

Intent and Scope

The prime intent of the Pattern Language for Distributed Computing is to serve as a guide through, and communication vehicle for, the best practices in major areas of distributed computing.



Audience

The language's main audience is:

- *Software architects*. They are supported in the design of new distributed middleware and applications, but also in improving and refactoring existing ones.

- *Application developers*. The language provides an overview of, and introduction to, the best practices and current state-of-the-art in distributed computing.

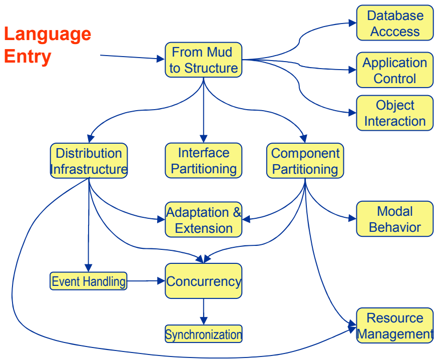
PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

Structure

The pattern language is partitioned into 13 problem areas…

… ranging from strategic infrastructural concerns to tactical aspects of component design and resource management:

- From Mud to Structure
- Distribution Infrastructure
- Application Control
- Interface Partitioning
- Component Partitioning
- Concurrency
- Synchronization
- Event Handling
- Adaptation & Extension
- Object Interaction
- Modal Behavior
- Resource Management
- Database Access



PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

Content

The pattern language integrates 114 patterns, and connects to about 150 patterns from other pattern languages:

Domain Model, Layers, Broker, Client-Dispatcher-Server, Pipes and Filters, Shared Repository, Blackboard, Model-View-Controller, Presentation-Abstraction-Control, Reflection, Microkernel, Domain Object, Explicit Interface, Extension Interface, Proxy, Facade, Object Group, Whole-Part, Composite, Master-Slave, Half-Object plus Protocol, Half-Sync/Half-Async, Leader/Followers, Active Object, Monitor Object, Guarded Suspension, Future, Thread-Safe Interface, Double-Checked Locking, Strategized Locking, Scoped Locking, Thread-Specific Storage, Immutable Value, Reactor, Proactor, Acceptor-Connector, Asynchronous Completion Token, Interceptor, Bridge, Visitor, Decorator, Template Method, Strategy, Wrapper Facade, Command Processor, View Handler, Forwarder-Receiver, Publisher-Subscriber, Adapter, Command, Chain of Responsibility, Mediator, Memento, Composite Message, Double Dispatch, (Objects for) State, Collections for State, Execute-Around Object, Combined Method, Enumeration Method, Batch Method, Null Object, Interpreter, Container, Component Configurator, Object Manager, Annotations, Coordinator, Lookup, Iterator, Pooling, Caching, Evictor, Activator, Leasing, Counting Handle, Explicitly Counted Object, Linked Handles, Flyweight, Eager Acquisition, Partial Acquisition, Lazy Acquisition, Lifecycle Callback, Database Access Layer, Data Mapper, Row Data Gateway, Table Data Gateway, Active Record, Abstract Factory, Builder, Mutable Companion, Factory Method, Disposal Method, Prototype, …

Concurrent Programming in Java Second Edition
SMALLTALK BEST PRACTICE PATTERNS
SERVER COMPONENT PATTERNS
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
Design Patterns
A SYSTEM OF PATTERNS

Presentation (1)

The presentation of the pattern language is structured into three levels:

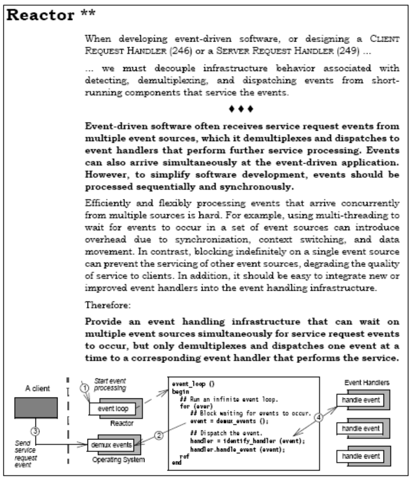
- **A general introduction outlines intent, scope, audience, structure and content of the language**
- **An introduction to each problem area presents**
  - the challenges arising in that problem area,
  - the original abstracts of all patterns that address these challenges,
  - diagrams that illustrate how the patterns are integrated into the language,
  - A brief discussion and comparison of the patterns.
- **The pattern descriptions in Alexandrian form**

WILEY SERIES IN SOFTWARE DESIGN PATTERNS
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
A Pattern Language for Distributed Computing
Volume 4
Frank Buschmann
Kevlin Henney
Douglas C. Schmidt

Presentation (2)

All patterns are described in Alexandrian form

- Stars rating the maturity of the pattern
- General context and "inbound" patterns – those patterns in whose realizations the pattern can be of use
- Problem and forces
- Solution description and visual sketch
- Discussion of consequences, core implementation hints, and "outbound" patterns that can help realizing the solution



Reactor **

When developing event-driven software, or designing a CLIENT REQUEST HANDLER (246) or a SERVER REQUEST HANDLER (249) ...

... we must decouple infrastructure behavior associated with detecting, demultiplexing, and dispatching events from short-running components that service the events.

♦ ♦ ♦

Event-driven software often receives service request events from multiple event sources, which it demultiplexes and dispatches to event handlers that perform further service processing. Events can also arrive simultaneously at the event-driven application. However, to simplify software development, events should be processed sequentially and synchronously.

Efficiently and flexibly processing events that arrive concurrently from multiple sources is hard. For example, using multi-threading to wait for events to occur in a set of event sources can introduce overhead due to synchronization, context switching, and data movement. In contrast, blocking indefinitely on a single event source can prevent the servicing of other event sources, degrading the quality of service to clients. In addition, it should be easy to integrate new or improved event handlers into the event handling infrastructure.

Therefore:

Provide an event handling infrastructure that can wait on multiple event sources simultaneously for service request events to occur, but only demultiplexes and dispatches one event at a time to a corresponding event handler that performs the service.

```
event_loop ()
begin
  ## Run an infinite event loop.
  for (ever)
     ## Block waiting for events to occur.
     event = demux_events ();

     ## Dispatch the event.
     handler = identify_handler (event);
     handler.handle_event (event);
  rof
end
```

Known Uses

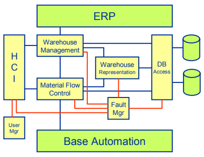
The language has informed the development of multiple real world systems:

- Communication and Component Middleware (CORBA, .NET, JEE)
- Network Management and Control Systems
- Warehouse Management
- Medical Imaging
- Real-Time Telecommunication
- Supervisory Control and Data Acquisition Systems
- ...

Warehouse Management Revisited

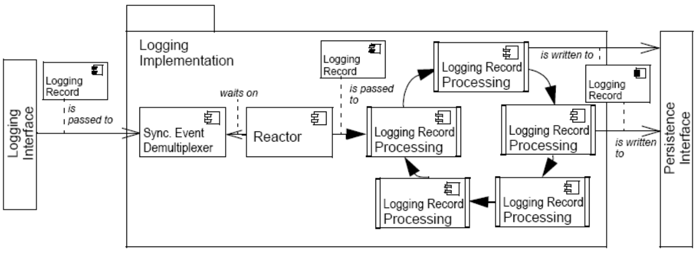
The pattern language for distributed computing suggests further patterns that can complete the architecture of the warehouse management system:

- **Reactor** to dispatch logging records that can arrive currently to multiple (concurrent) handlers that process the records
- **Component Configurator** to support (re-)configuration and deployment without degrading availability
- **Application Controller** to provide workflow support
- **... (see POSA4) ...**



Dispatching Logging Records Efficiently

Use a Reactor for receiving, demultiplexing, and dispatching logging records that can arrive concurrently from clients efficiently to handler threads in a Leader/Followers (or Half-Sync/Half-Async) design that process these records.



Reactor at a Glance

Problem

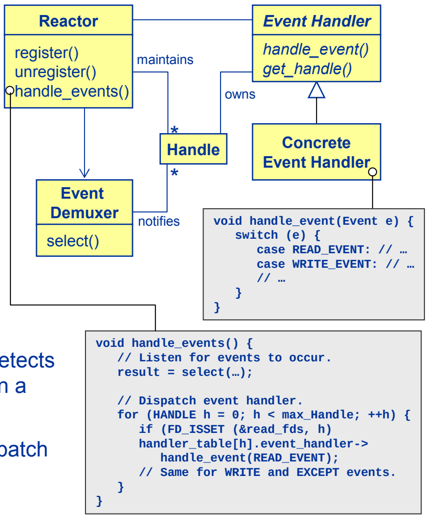
How can we provide an event handling infrastructure so that:

- Events can arrive concurrently from multiple clients?
- Event throughput is maximized?

Solution

Separate event demultiplexing and dispatching from event processing:

- A *Synchronous Event Demultiplexer* detects and demultiplexes events that occur on a set of event sources (*Handles*).
- A *Reactor* provides functionality to dispatch events to application services realized as *Event Handlers*.

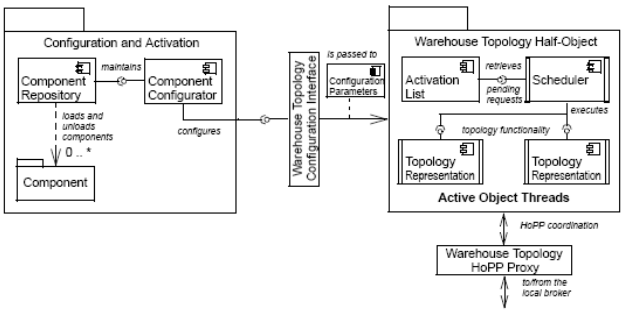


```
void handle_event(Event e) {
   switch (e) {
      case READ_EVENT: // …
      case WRITE_EVENT: // …
      // …
   }
}
```

```
void handle_events() {
   // Listen for events to occur.
   result = select(…);

   // Dispatch event handler.
   for (HANDLE h = 0; h < max_Handle; ++h) {
      if (FD_ISSET (&read_fds, h)
      handler_table[h].event_handler->
         handle_event(READ_EVENT);
      // Same for WRITE and EXCEPT events.
   }
}
```

Run-Time Configurability

Introduce a configuration and activation service realized as a Component Configurator to ensure run-time deployment, configuration, and exchange of components and services without degrading the availability of other system components.



Component Configurator at a Glance

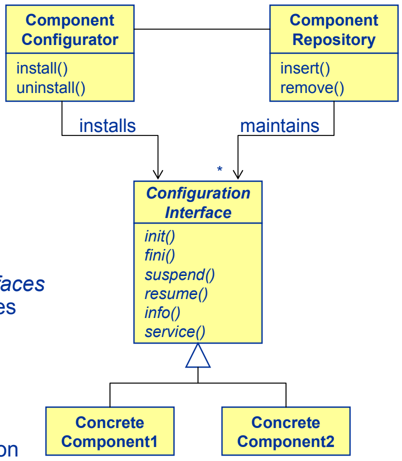
Problem

How can we support run-time (re-) deployment and (re-) configuration of a system?

Solution

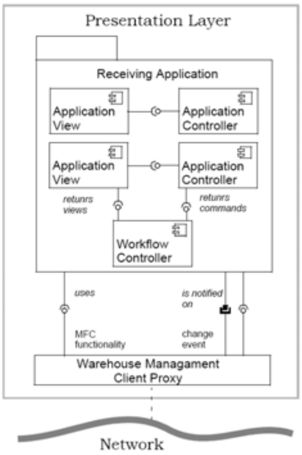
Provide an infrastructure for run-time lifecycle control of components.

- *Components* provide *Configuration Interfaces* separated from their operational interfaces
- A *Component Repository* maintains the component (to be) used in a specific system configuration.
- A *Component Configurator* controls and monitors the deployment and configuration of components.



Workflow Support

Introduce an Application Controller for workflow driven applications, such as Receiving and Shipping, that decides what UI view to display and what UI controllers and commands to offer in a given workflow state.

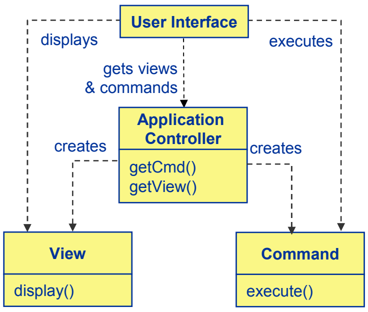


Application Controller at a Glance

Problem

How can we design workflow-oriented user interfaces so that:

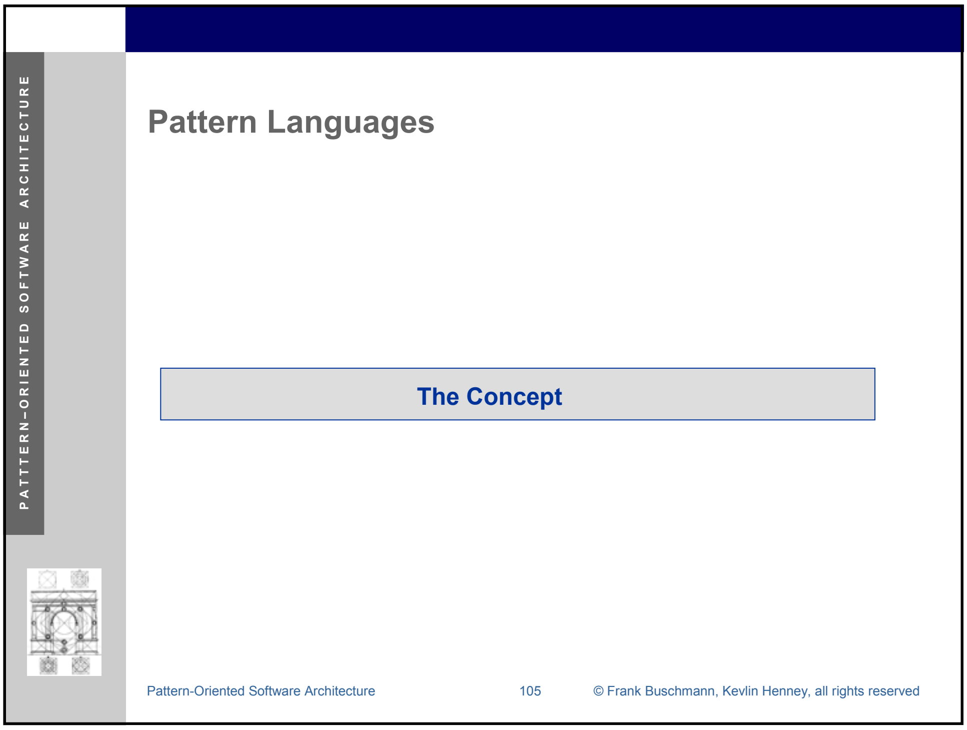
- Workflow information is not scattered across multiple UI controllers?
- Workflow logic is not coupled with presentation logic?



Solution

Introduce an *Application Controller* that captures the application workflow to provide the UI with information about:

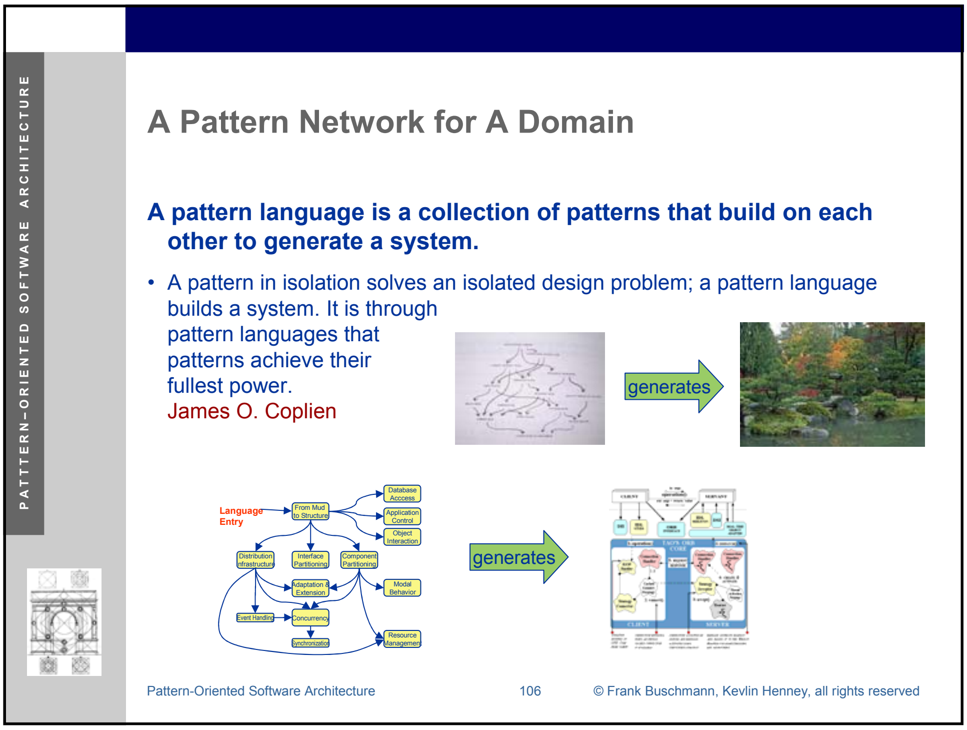
- What *Commands* can be executed in a specific workflow state in response to concrete request.
- What *View* to display in the UI.



PATTTERN-ORIENTED SOFTWARE ARCHITECTURE

Pattern Languages

The Concept



PATTTERN-ORIENTED SOFTWARE ARCHITECTURE

A Pattern Network for A Domain

A pattern language is a collection of patterns that build on each other to generate a system.

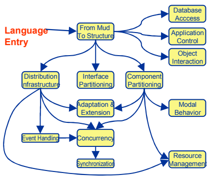
- A pattern in isolation solves an isolated design problem; a pattern language builds a system. It is through pattern languages that patterns achieve their fullest power.
  James O. Coplien

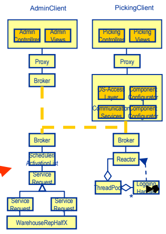
PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

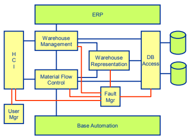
A Process and a Thing

A pattern language is both a process and a thing:

- How to design a system of a specific type or domain?
- What design space exists for this type of system?
- How to create a specific, feasible design option systematically?







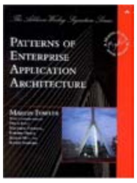
Many more...

PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

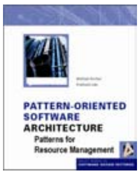
The Things

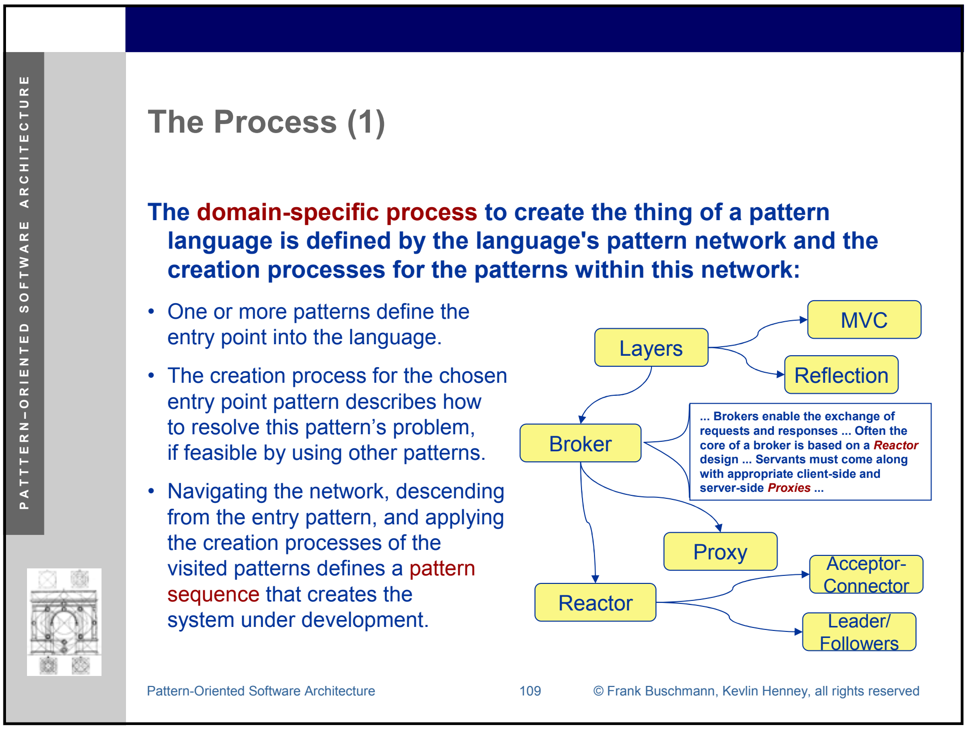
The thing of a pattern language can be any domain in software:

- A type or domain of software system, such as for Distributed Computing, Enterprise Applications, or Warehouse Management.
- A technical domain, such as Security, Resource Management, Messaging, Remoting, and Component Development.
- A good programming style, such as for C++ memory management and Java exception handling.



SERVER COMPONENT PATTERNS
Component Infrastructures Illustrated with EJB


PATTERN-ORIENTED SOFTWARE ARCHITECTURE
Patterns for Resource Management



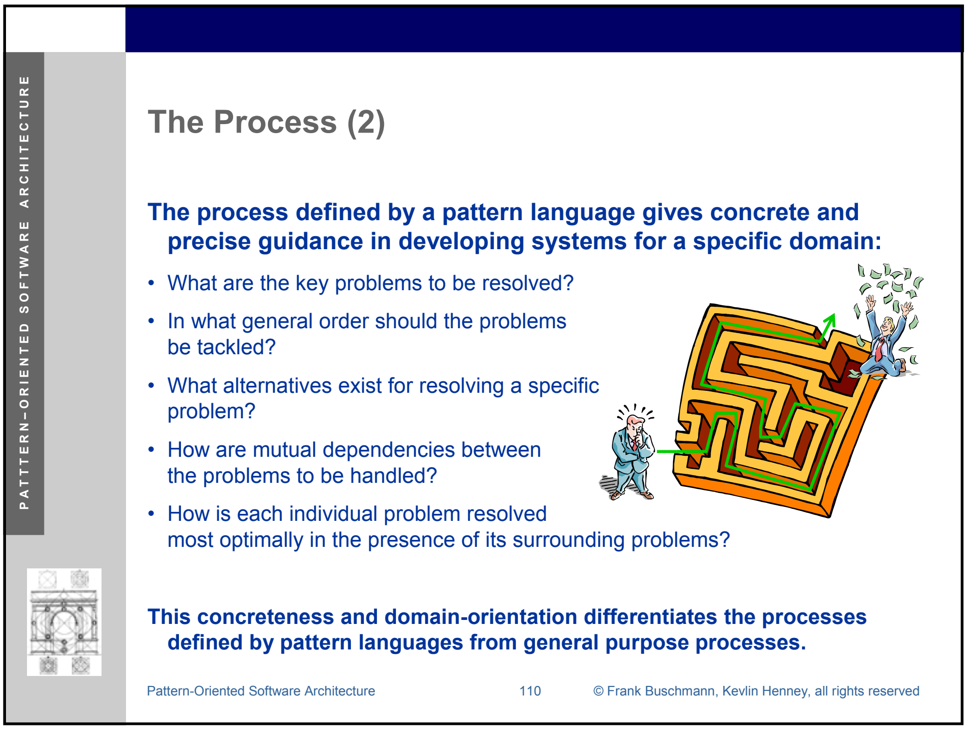
PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

The Process (1)

The domain-specific process to create the thing of a pattern language is defined by the language's pattern network and the creation processes for the patterns within this network:

- One or more patterns define the entry point into the language.
- The creation process for the chosen entry point pattern describes how to resolve this pattern’s problem, if feasible by using other patterns.
- Navigating the network, descending from the entry pattern, and applying the creation processes of the visited patterns defines a pattern sequence that creates the system under development.

Layers
MVC
Reflection
Broker
... Brokers enable the exchange of requests and responses ... Often the core of a broker is based on a *Reactor* design ... Servants must come along with appropriate client-side and server-side *Proxies* ...
Proxy
Reactor
Acceptor-Connector
Leader/Followers



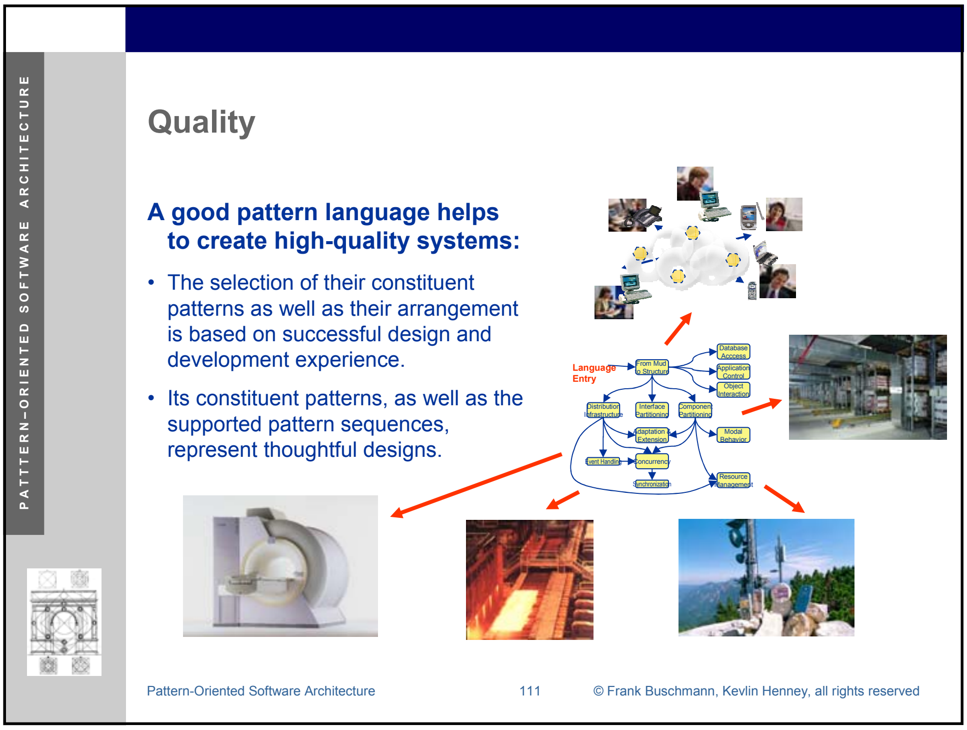
PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

The Process (2)

The process defined by a pattern language gives concrete and precise guidance in developing systems for a specific domain:

- What are the key problems to be resolved?
- In what general order should the problems be tackled?
- What alternatives exist for resolving a specific problem?
- How are mutual dependencies between the problems to be handled?
- How is each individual problem resolved most optimally in the presence of its surrounding problems?

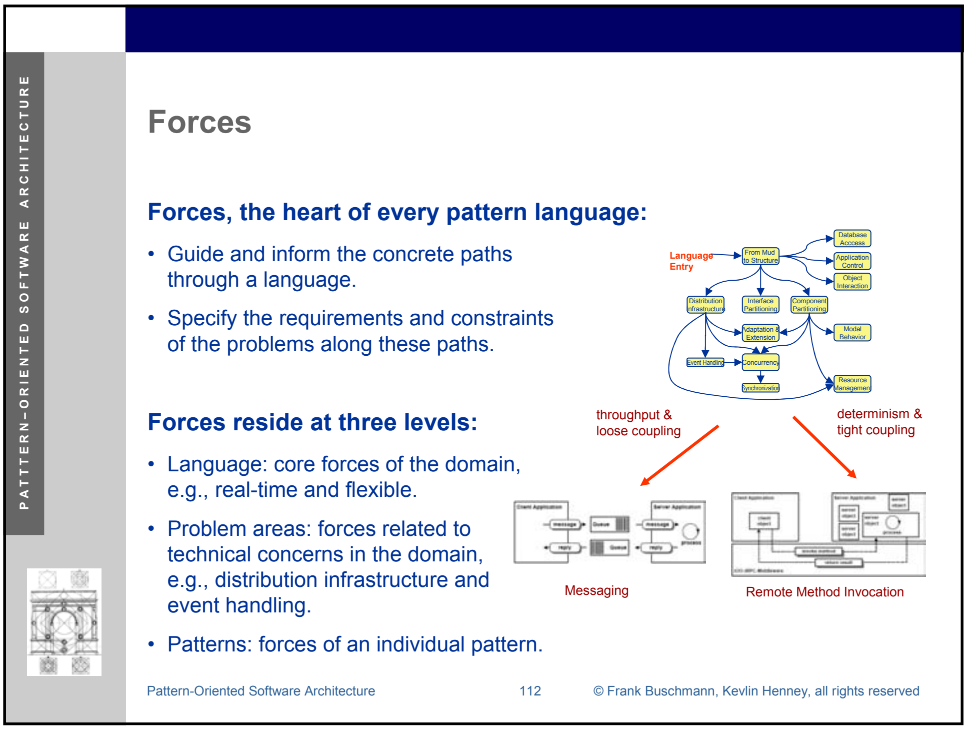
This concreteness and domain-orientation differentiates the processes defined by pattern languages from general purpose processes.


PATTERN-ORIENTED SOFTWARE ARCHITECTURE
Language Entry
From Mud to Structure
Database Acccess
Application Control
Object Interaction
Distribution Infrastructure
Interface Partitioning
Component Partitioning
Adaptation & Extension
Modal Behavior
Event Handling
Concurrency
Synchronization
Resource Management

Quality

A good pattern language helps to create high-quality systems:

- The selection of their constituent patterns as well as their arrangement is based on successful design and development experience.
- Its constituent patterns, as well as the supported pattern sequences, represent thoughtful designs.


PATTERN-ORIENTED SOFTWARE ARCHITECTURE
Language Entry
From Mud to Structure
Database Access
Application Control
Object Interaction
Distribution Infrastructure
Interface Partitioning
Component Partitioning
Adaptation & Extension
Modal Behavior
Event Handling
Concurrency
Synchronization
Resource Management
throughput & loose coupling
determinism & tight coupling
Messaging
Remote Method Invocation

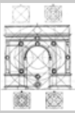
Forces

Forces, the heart of every pattern language:

- Guide and inform the concrete paths through a language.
- Specify the requirements and constraints of the problems along these paths.

Forces reside at three levels:

- Language: core forces of the domain, e.g., real-time and flexible.
- Problem areas: forces related to technical concerns in the domain, e.g., distribution infrastructure and event handling.
- Patterns: forces of an individual pattern.



Context

Pattern Contexts Define Topology and Architectural Style

- The context of a pattern in a language describes when in general the problem the pattern addresses can arise, and lists all patterns of the language in which implementation this problem can occur.
- The contexts of all patterns in a language define a pattern topology that includes all supported pattern sequences.
- The contexts of all patterns in a language define a set of architectural styles.

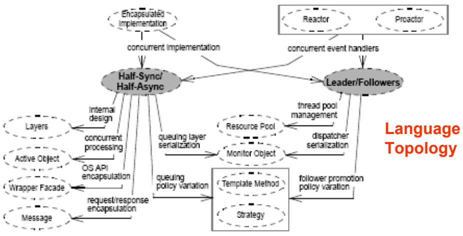
Language-specific Context

Half-Sync/Half-Async **

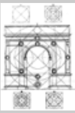
When developing a concurrent **Encapsulated Implementation** or a network server that employs a **Reactor** or **Proactor** event handling infrastructure ...

... we need to make performance efficient and scalable while also ensuring that any use of concurrency simplifies programming.

General Context



Language Topology



Grammar and Vocabulary

The patterns in a pattern language define its vocabulary, the supported pattern sequences its grammar.

Pattern Sequences

Broker → Layers → Reflection → Wrapper Facade → Reactor → Acceptor-Connector → Forwarder-Receiver → Strategy → Proxy → Facade → Adapter

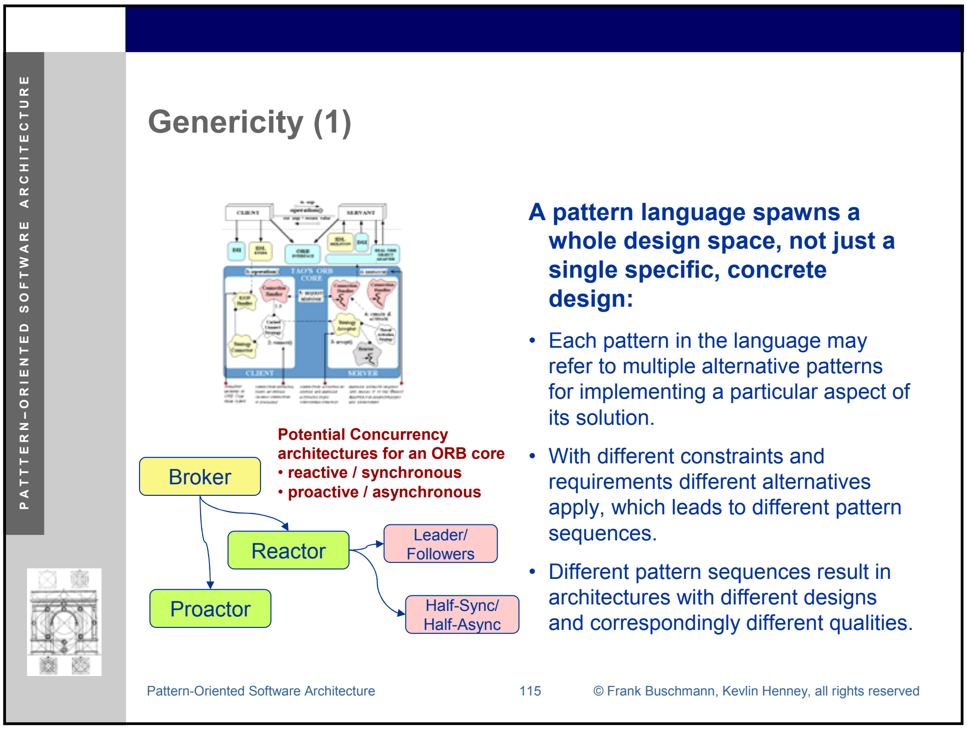
Broker → Layers → Reflection → Wrapper Facade → Proactor → Acceptor-Connector → Forwarder-Receiver → Template Method → Proxy → Facade → Adapter

Broker → Layers → Reflection → Wrapper Facade → Reactor → Acceptor-Connector → Leader/Followers → Forwarder-Receiver → Template Method → Proxy → Facade → Adapter

Broker → Layers → Reflection → ...

Pattern Language Grammar

Broker → Layers
→ Reflection
→ Wrapper Facade
→ (Reactor | Proactor)
→ Acceptor-Connector
→ (Leader/Followers | Half-Sync/Half-Async)
→ Forwarder-Receiver
→ (Strategy | Template Method)
→ Proxy → Facade → Adapter



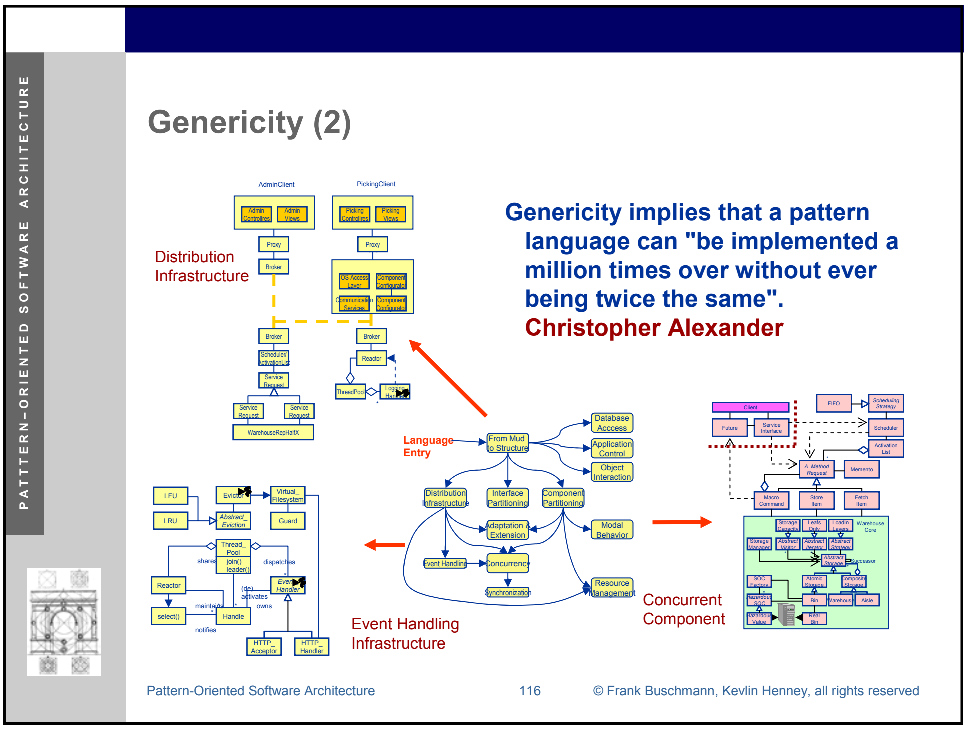
Genericity (1)

Potential Concurrency architectures for an ORB core
- reactive / synchronous
- proactive / asynchronous

Broker
Reactor
Proactor
Leader/ Followers
Half-Sync/ Half-Async
CLIENT
SERVANT
TAO'S ORB CORE
CLIENT
SERVER

A pattern language spawns a whole design space, not just a single specific, concrete design:

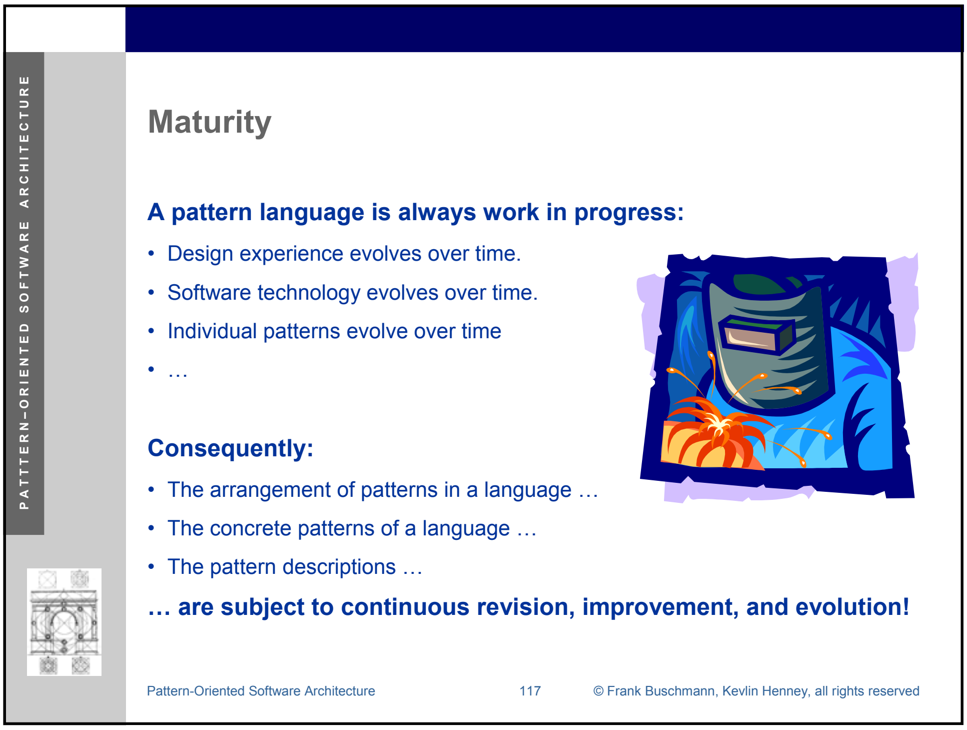
- Each pattern in the language may refer to multiple alternative patterns for implementing a particular aspect of its solution.
- With different constraints and requirements different alternatives apply, which leads to different pattern sequences.
- Different pattern sequences result in architectures with different designs and correspondingly different qualities.



Genericity (2)

Genericity implies that a pattern language can "be implemented a million times over without ever being twice the same".
Christopher Alexander

Distribution Infrastructure
AdminClient
PickingClient
Admin Controllers
Admin Views
Picking Controllers
Picking Views
Proxy
Broker
Reactor
ThreadPool
WarehouseRepHalfX
Language Entry
From Mud to Structure
Database Access
Application Control
Object Interaction
Distribution Infrastructure
Interface Partitioning
Component Partitioning
Adaptation & Extension
Modal Behavior
Event Handling
Concurrency
Synchronization
Resource Management
LFU
LRU
Evictor
Virtual_ Filesystem
Abstract_ Eviction
Guard
Thread_ Pool
Reactor
select()
Handle
Event Handler
HTTP_ Acceptor
HTTP_ Handler
Event Handling Infrastructure
Concurrent Component
Client
FIFO
Scheduling Strategy
Future
Service Interface
Scheduler
Activation List
Method Request
Memento



Maturity

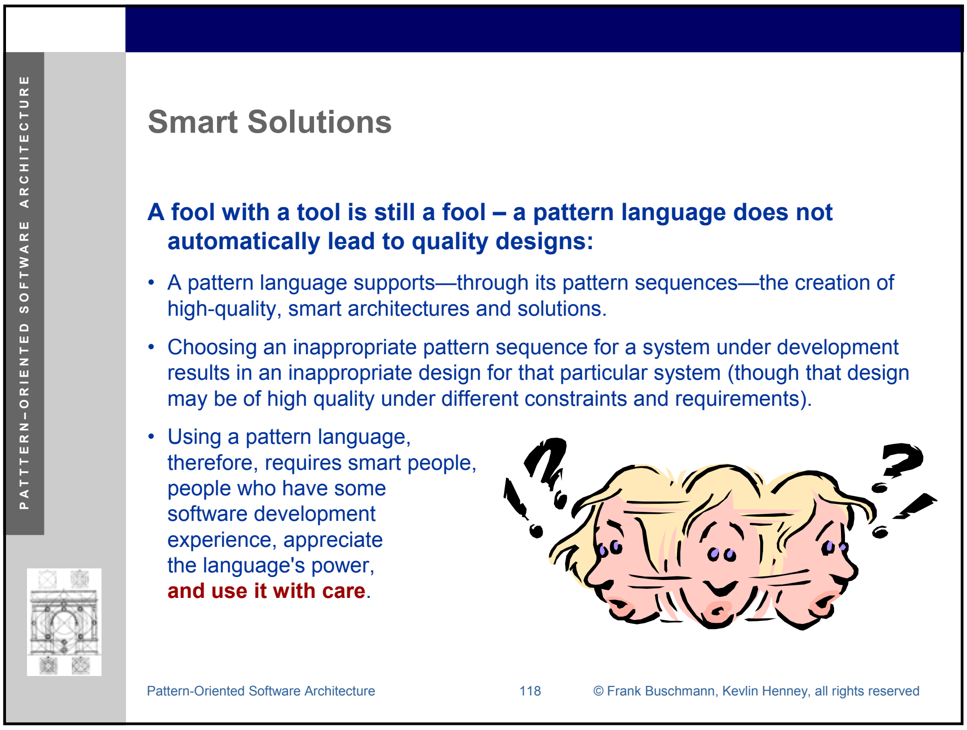
A pattern language is always work in progress:

- Design experience evolves over time.
- Software technology evolves over time.
- Individual patterns evolve over time
- …

Consequently:

- The arrangement of patterns in a language …
- The concrete patterns of a language …
- The pattern descriptions …

… are subject to continuous revision, improvement, and evolution!



Smart Solutions

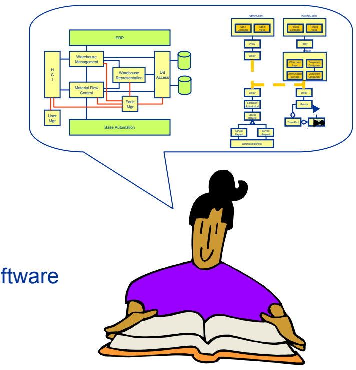
A fool with a tool is still a fool – a pattern language does not automatically lead to quality designs:

- A pattern language supports—through its pattern sequences—the creation of high-quality, smart architectures and solutions.
- Choosing an inappropriate pattern sequence for a system under development results in an inappropriate design for that particular system (though that design may be of high quality under different constraints and requirements).
- Using a pattern language, therefore, requires smart people, people who have some software development experience, appreciate the language's power, **and use it with care**.

Understanding

A pattern language initiates a dialog about a system and tells many software engineering success stories:

- It encourages its users to think first and then act, not vice versa.
- It forces its users to consider, evaluate, and weight alternative design options for addressing specific challenges.
- It guides its users through a design space, helping them to explore and understand this space.
- Each concrete pattern sequence tells a success story about how to design a software system under a specific set of forces (requirements/constraints).



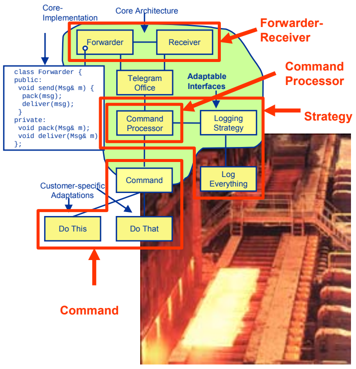
Application Area: Construction

The major application area for pattern languages is software construction.

Much of today's software is consciously designed with pattern languages.

- Telecommunication Management Networks
- Real-Time (Tele-) Communication Applications
- Hot Rolling Mill Process Automation
- Medical Imaging
- Power Distribution
- Warehouse Management
- Middleware
- Media and News Distribution

A Telegram Handling Framework



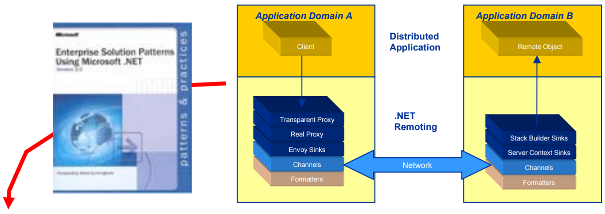
```
class Forwarder {
public:
 void send(Msg& m) {
  pack(msg);
  deliver(msg);
 }
private:
 void pack(Msg& m);
 void deliver(Msg& m)
};
```

Application Area: Understanding and Using

Pattern languages help understanding modern software products.

How are they designed, how do they work, why are they how they are?

- CORBA / CCM
- J(2)EE / EJB
- Swing
- .NET
- …

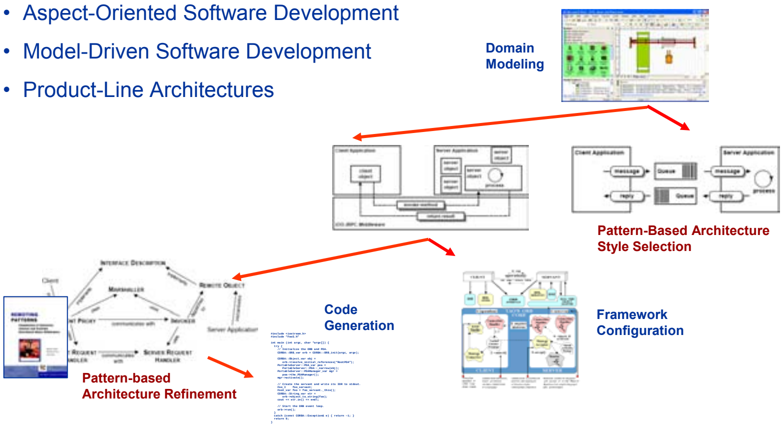


Broker	How can you structure a distributed system so that application developers don't have to concern themselves with the details of remote communication?
Implementing Broker with .Net Remoting	How do you Implement Broker in .Net?
Singleton	How do you make an instance of an object globally available and guarantee that only one instance of the class is created?
Implementing Singleton in .Net	How do you Implement Singleton in .Net

Application Area: Technology Enabling

Patterns support several other software technologies:

- Aspect-Oriented Software Development
- Model-Driven Software Development
- Product-Line Architectures



```
#include <iostream.h>
#include "FooS.h"

int main (int argc, char *argv[]) {
  try {
    // Initialize the ORB and POA.
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    CORBA::Object_var obj =
      orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa =
      PortableServer::POA::_narrow(obj);
    PortableServer::POAManager_var mgr =
      poa->the_POAManager();
    mgr->activate();

    // Create the servant and write its IOR to stdout.
    Foo_I    foo_servant;
    FooI_var foo = foo_servant._this();
    CORBA::String_var str =
      orb->object_to_string(foo);
    cout << str.in() << endl;

    // Start the ORB event loop.
    orb->run();
  }
  catch (const CORBA::Exception& e) { return -1; }
  return 0;
}
```

GOF is *not* a Pattern Language

The GOF patterns do not form a pattern language:

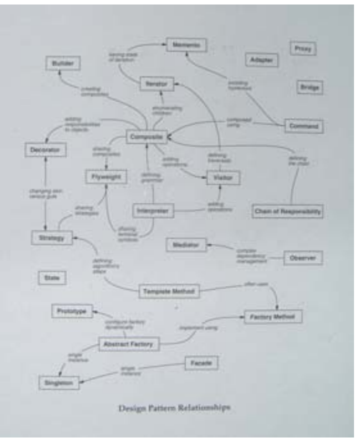
- A common misconception that the GOF never claimed.
- It is the map in the book that causes this misconception.

But a map is not the territory!

- It is largely not about *uses* relationships.
- The uses shown are often not useful.

The map is sometimes misleading:

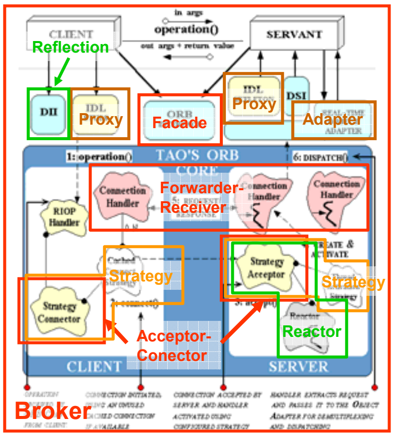
- What is the most important pattern?
- What are the most isolated patterns?



Where we are

- **Introduction**
- **Stand-Alone Patterns**
- **Pattern Complements / Pattern Compounds**
- **Pattern Stories / Pattern Sequences**
- **Pattern Languages**
- **Outroduction**
- **References**

Observations (1)



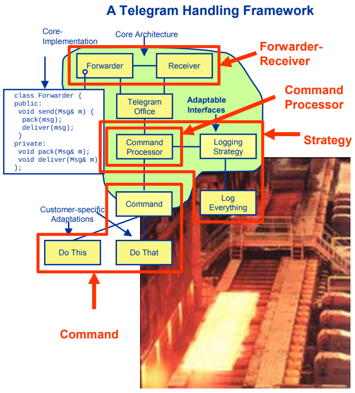
Quality architectures expose a high pattern density:

- Patterns that focus on problem domain understanding and broad architecture help in specifying the base-line architecture.
- Patterns further help in refining the base-line architecture.
- Patterns that are focused on the languages and technologies help in the implementation of a software architecture.

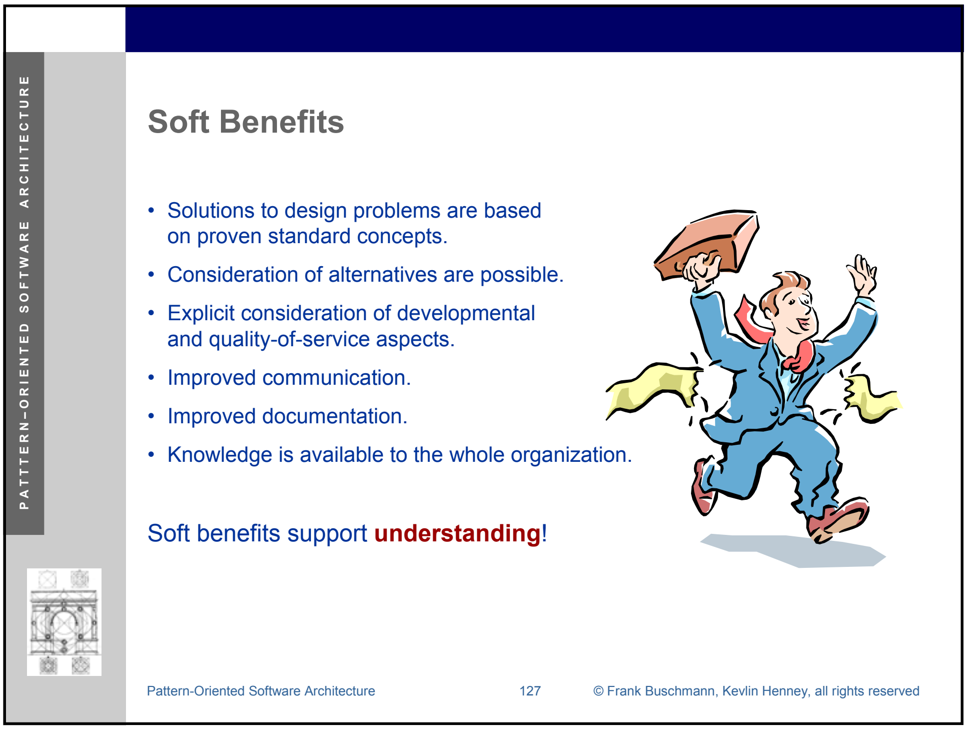
Observations (2)

Patterns support the development of frameworks:

- Patterns for structural decomposition and component cooperation support the creation of usable software systems and components with a stable and reasoned software design.
- Patterns for flexibility and configuration open a stable design for adaptation, extension, and evolution in a well-defined manner.
- **Frameworks = Patterns + Components**
  Ralph Johnson


A Telegram Handling Framework

```
class Forwarder {
public:
 void send(Msg& m) {
  pack(msg);
  deliver(msg);
 }
private:
 void pack(Msg& m);
 void deliver(Msg& m)
};
```



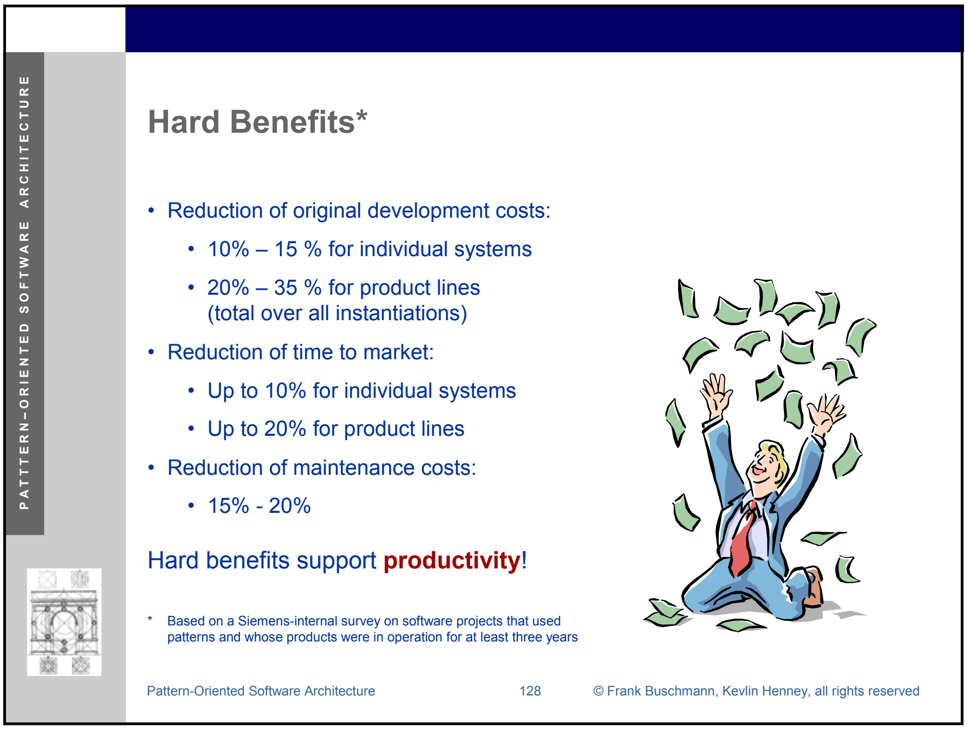
PATTTERN-ORIENTED SOFTWARE ARCHITECTURE

Soft Benefits

- Solutions to design problems are based on proven standard concepts.
- Consideration of alternatives are possible.
- Explicit consideration of developmental and quality-of-service aspects.
- Improved communication.
- Improved documentation.
- Knowledge is available to the whole organization.

Soft benefits support **understanding**!

Pattern-Oriented Software Architecture 127 © Frank Buschmann, Kevlin Henney, all rights reserved



PATTTERN-ORIENTED SOFTWARE ARCHITECTURE

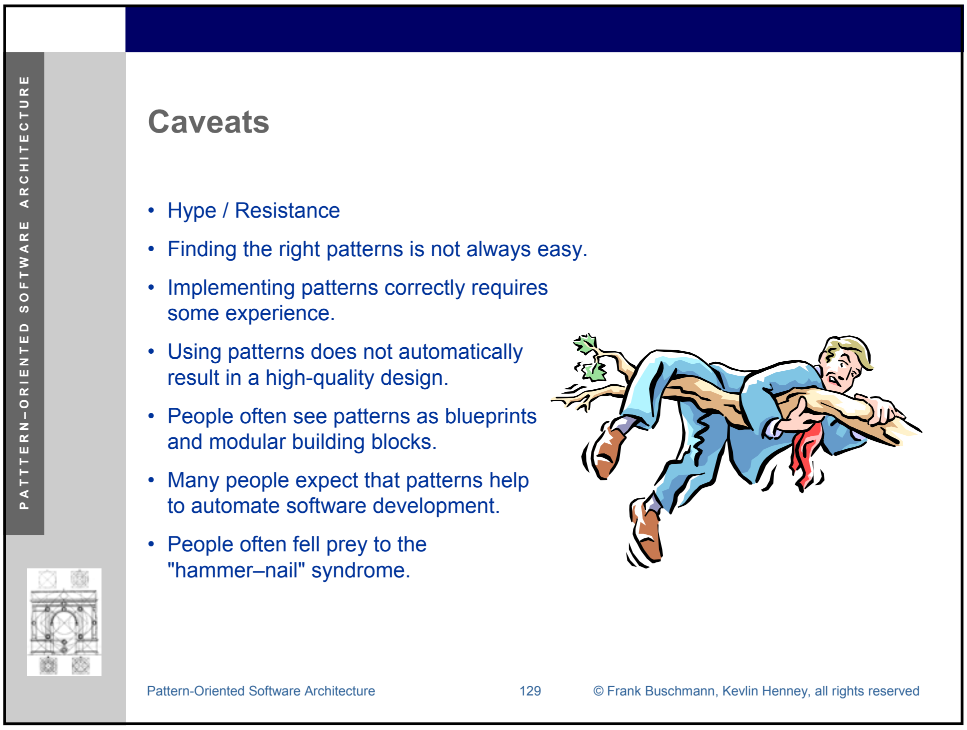
Hard Benefits*

- Reduction of original development costs:
  - 10% – 15 % for individual systems
  - 20% – 35 % for product lines (total over all instantiations)
- Reduction of time to market:
  - Up to 10% for individual systems
  - Up to 20% for product lines
- Reduction of maintenance costs:
  - 15% - 20%

Hard benefits support **productivity**!

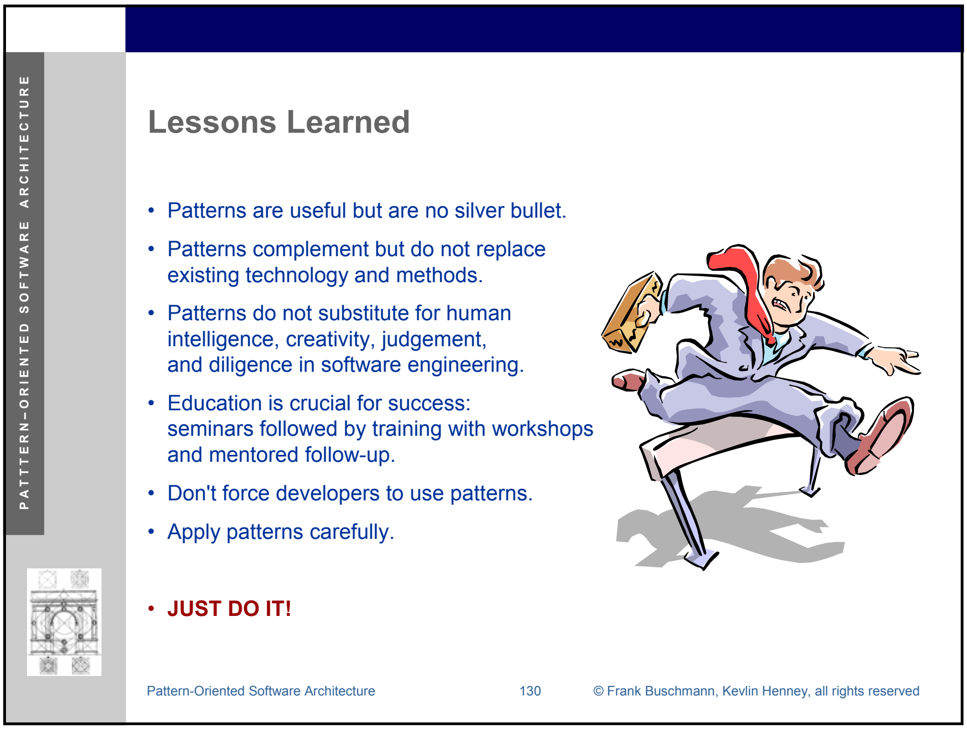
* Based on a Siemens-internal survey on software projects that used patterns and whose products were in operation for at least three years

Pattern-Oriented Software Architecture 128 © Frank Buschmann, Kevlin Henney, all rights reserved



Caveats

- Hype / Resistance
- Finding the right patterns is not always easy.
- Implementing patterns correctly requires some experience.
- Using patterns does not automatically result in a high-quality design.
- People often see patterns as blueprints and modular building blocks.
- Many people expect that patterns help to automate software development.
- People often fell prey to the "hammer–nail" syndrome.



Lessons Learned

- Patterns are useful but are no silver bullet.
- Patterns complement but do not replace existing technology and methods.
- Patterns do not substitute for human intelligence, creativity, judgement, and diligence in software engineering.
- Education is crucial for success: seminars followed by training with workshops and mentored follow-up.
- Don't force developers to use patterns.
- Apply patterns carefully.

- **JUST DO IT!**

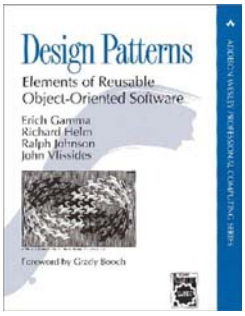
Where we are

- Introduction
- Stand-Alone Patterns
- Pattern Complements / Pattern Compounds
- Pattern Stories / Pattern Sequences
- Pattern Languages
- Outroduction
- **References**

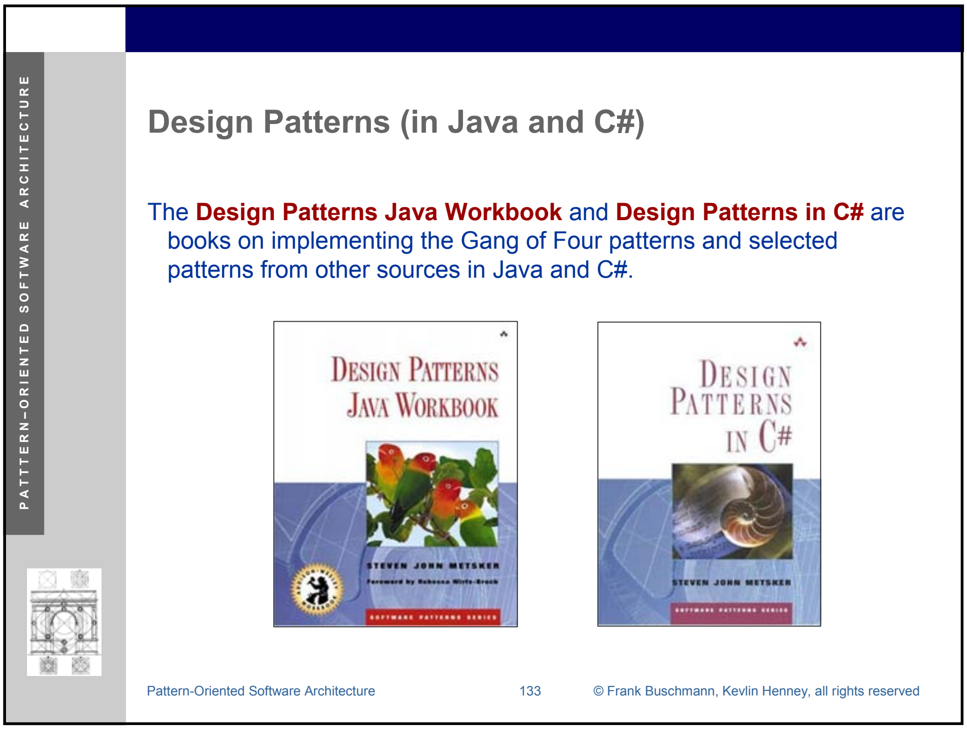
Design Patterns (Gang of Four)

The **Gang of Four** book is the first, and still the most popular pattern book. It contains 23 general purpose design patterns and idioms for:

- *Object creation*: Abstract Factory, Builder, Factory Method, Prototype, and Singleton
- *Structural Decomposition*: Composite and Interpreter
- *Organization of Work*: Command, Mediator, and Chain of Responsibility
- *Service Access*: Proxy, Facade, and Iterator
- *Extensibility*: Decorator and Visitor
- *Variation*: Bridge, Strategy, State, and Template Method
- *Adaptation*: Adapter
- *Resource Management*: Memento and Flyweight
- *Communication*: Observer

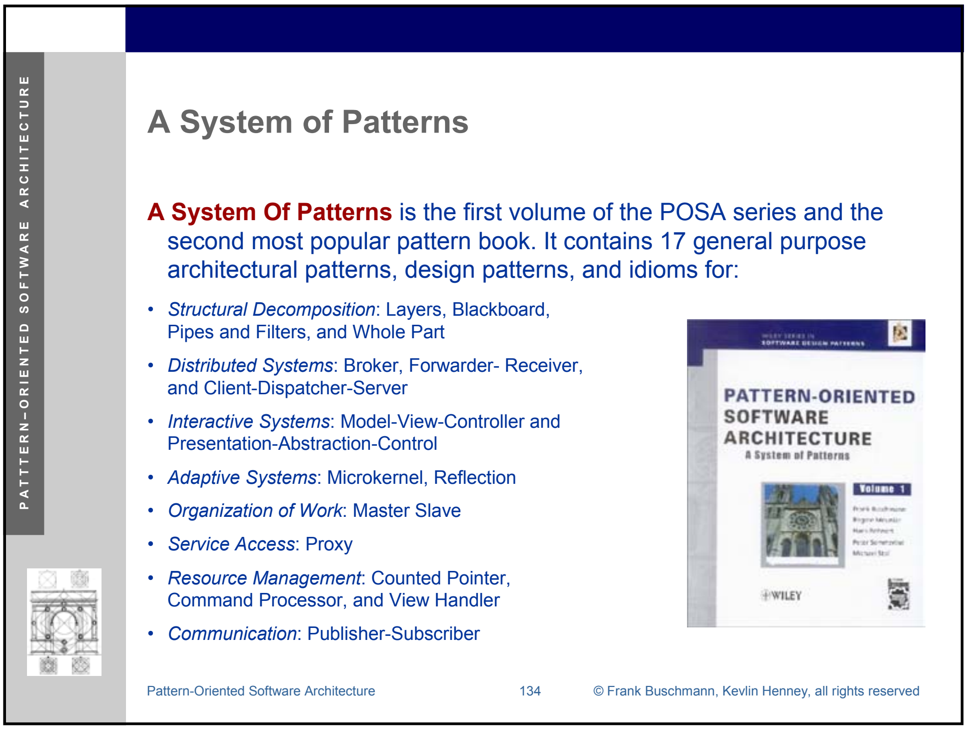


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



Design Patterns (in Java and C#)

The **Design Patterns Java Workbook** and **Design Patterns in C#** are books on implementing the Gang of Four patterns and selected patterns from other sources in Java and C#.



A System of Patterns

A System Of Patterns is the first volume of the POSA series and the second most popular pattern book. It contains 17 general purpose architectural patterns, design patterns, and idioms for:

- *Structural Decomposition*: Layers, Blackboard, Pipes and Filters, and Whole Part
- *Distributed Systems*: Broker, Forwarder- Receiver, and Client-Dispatcher-Server
- *Interactive Systems*: Model-View-Controller and Presentation-Abstraction-Control
- *Adaptive Systems*: Microkernel, Reflection
- *Organization of Work*: Master Slave
- *Service Access*: Proxy
- *Resource Management*: Counted Pointer, Command Processor, and View Handler
- *Communication*: Publisher-Subscriber

WILEY SERIES IN SOFTWARE DESIGN PATTERNS
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
A System of Patterns
Volume 1
WILEY

Patterns for Concurrent and Networked Objects

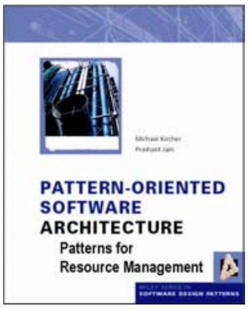
Patterns For Concurrent And Networked Objects is the second volume of the POSA series. It contains 17 architectural patterns, design patterns and idioms for concurrent, and networked systems:

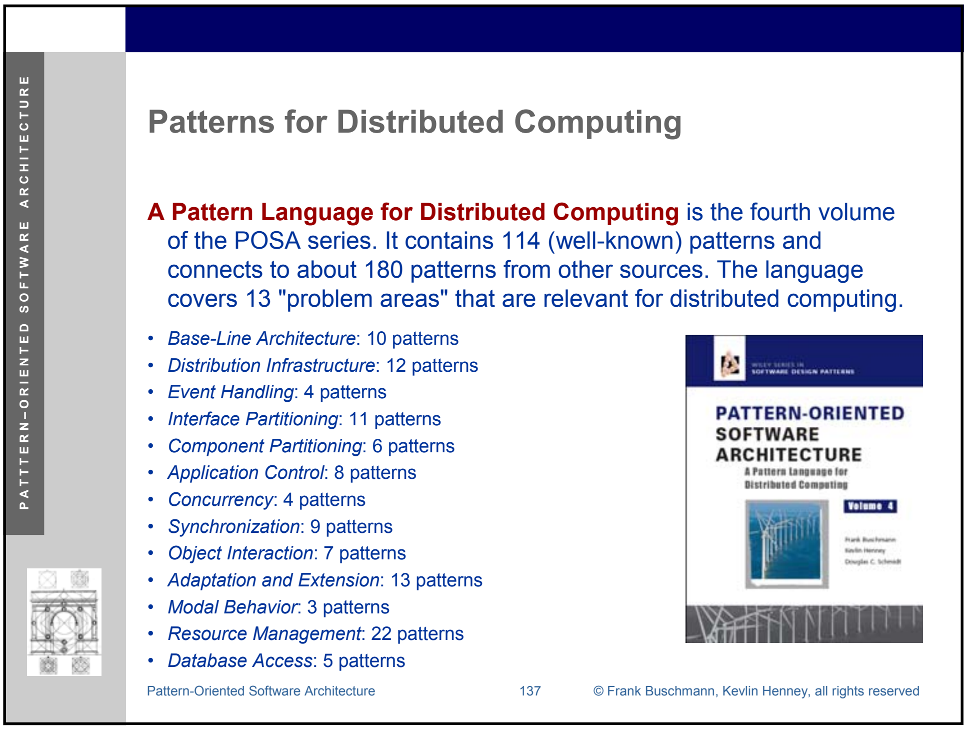
- *Service Access and Configuration*: Wrapper Facade, Component Configurator, Interceptor, and Extension Interface
- *Event Handling*: Reactor, Proactor, Asynchronous Completion Token, and Acceptor-Connector
- *Synchronization*: Scoped Locking, Double-Checked Locking, Strategized Locking, and Thread-Safe Interface
- *Concurrency*: Active Object, Monitor Object, Leader/Followers, Thread-Specific Storage, and Half-Sync/Half-Async

Patterns for Resource Management

Patterns For Resource Management is the third volume of the POSA series. It contains 10 patterns that address the lifecycle of resources: memory, threads, connections, and services:

- *Resource Acquisition*: Lookup, Lazy Acquisition, Eager Acquisition, and Partial Acquisition
- *Resource Lifecycle*: Caching, Pooling, Coordinator, and Resource Lifecycle Manager
- *Resource Release*: Leasing and Evictor


Michael Kircher
Prashant Jain
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
Patterns for Resource Management
SOFTWARE DESIGN PATTERNS



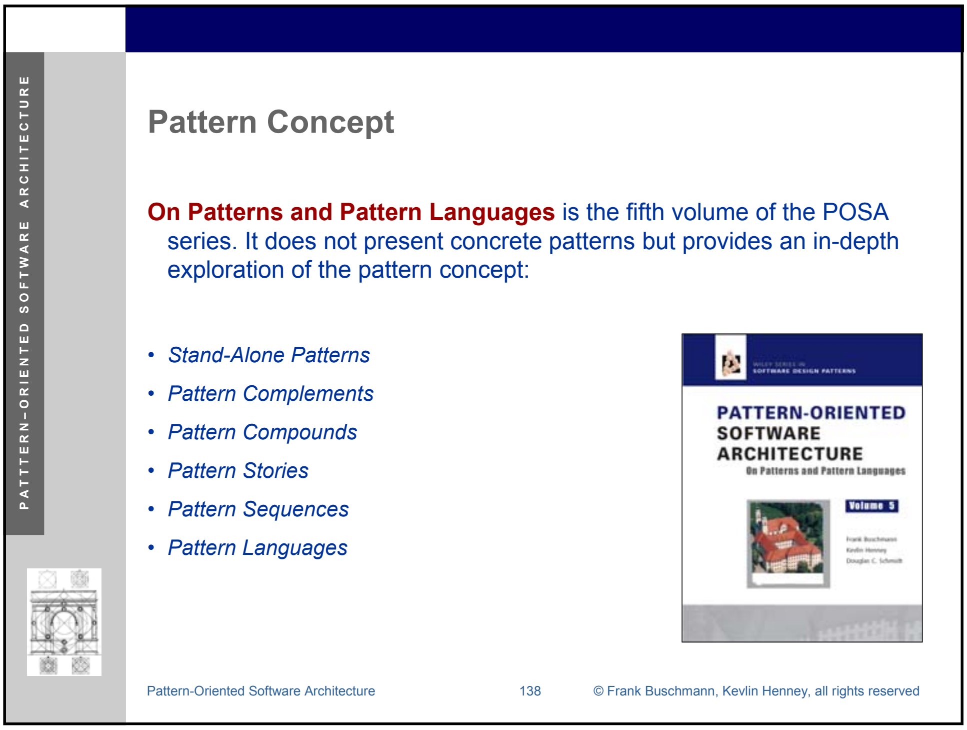
PATTTERN-ORIENTED SOFTWARE ARCHITECTURE

Patterns for Distributed Computing

A Pattern Language for Distributed Computing is the fourth volume of the POSA series. It contains 114 (well-known) patterns and connects to about 180 patterns from other sources. The language covers 13 "problem areas" that are relevant for distributed computing.

- *Base-Line Architecture*: 10 patterns
- *Distribution Infrastructure*: 12 patterns
- *Event Handling*: 4 patterns
- *Interface Partitioning*: 11 patterns
- *Component Partitioning*: 6 patterns
- *Application Control*: 8 patterns
- *Concurrency*: 4 patterns
- *Synchronization*: 9 patterns
- *Object Interaction*: 7 patterns
- *Adaptation and Extension*: 13 patterns
- *Modal Behavior*: 3 patterns
- *Resource Management*: 22 patterns
- *Database Access*: 5 patterns

WILEY SERIES IN SOFTWARE DESIGN PATTERNS
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
A Pattern Language for Distributed Computing
Volume 4
Frank Buschmann
Kevlin Henney
Douglas C. Schmidt



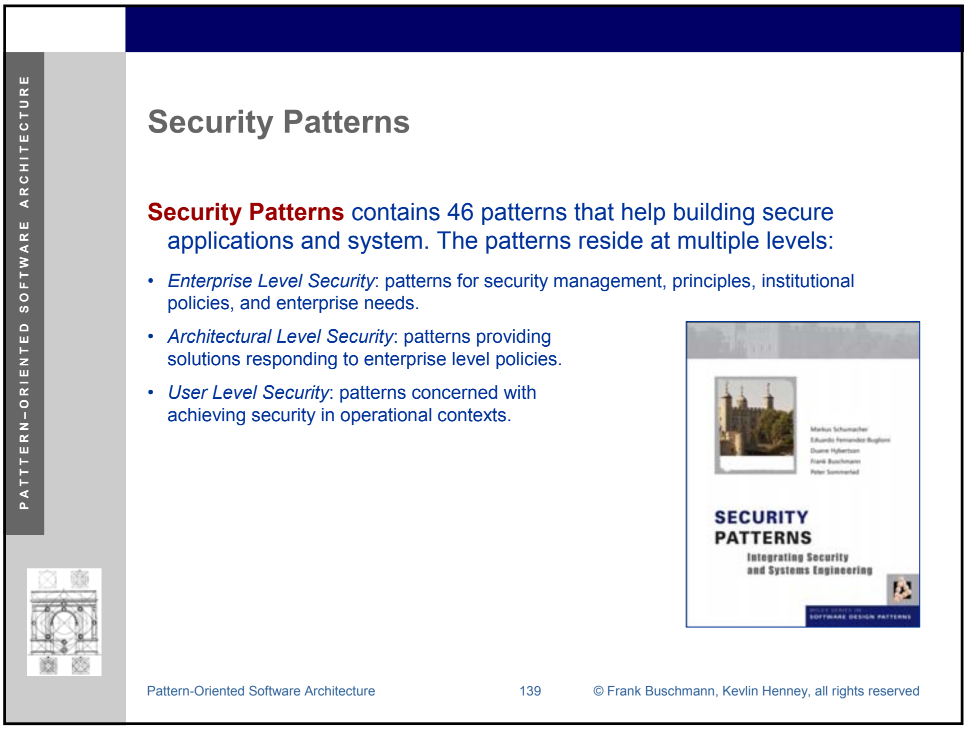
PATTTERN-ORIENTED SOFTWARE ARCHITECTURE

Pattern Concept

On Patterns and Pattern Languages is the fifth volume of the POSA series. It does not present concrete patterns but provides an in-depth exploration of the pattern concept:

- *Stand-Alone Patterns*
- *Pattern Complements*
- *Pattern Compounds*
- *Pattern Stories*
- *Pattern Sequences*
- *Pattern Languages*

WILEY SERIES IN SOFTWARE DESIGN PATTERNS
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
On Patterns and Pattern Languages
Volume 5
Frank Buschmann
Kevlin Henney
Douglas C. Schmidt

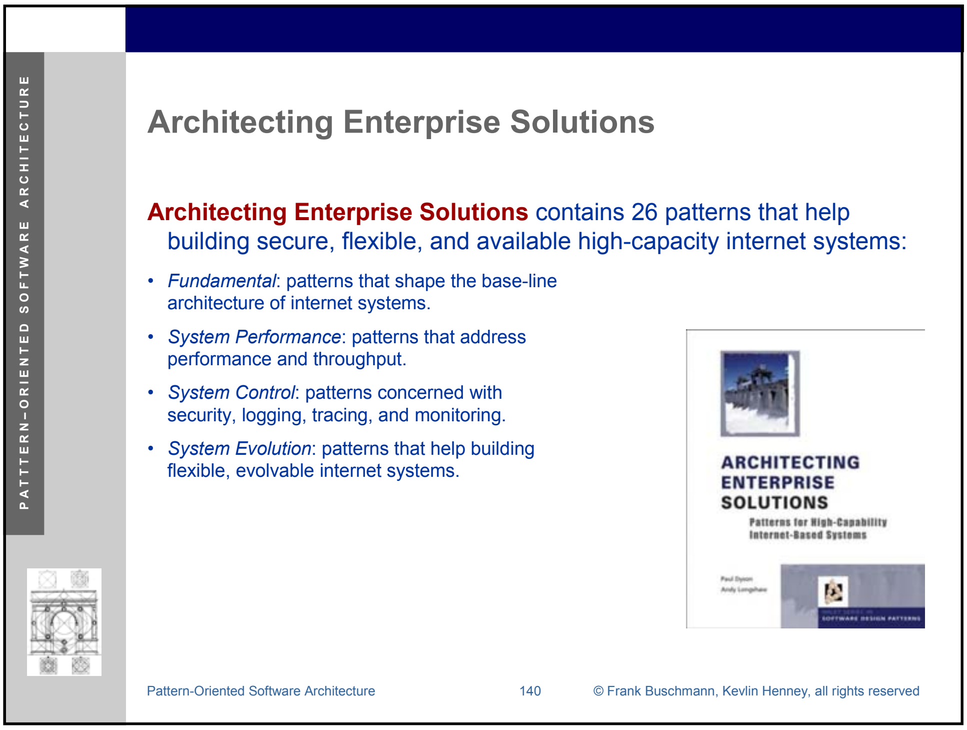


Security Patterns

Security Patterns contains 46 patterns that help building secure applications and system. The patterns reside at multiple levels:

- *Enterprise Level Security*: patterns for security management, principles, institutional policies, and enterprise needs.
- *Architectural Level Security*: patterns providing solutions responding to enterprise level policies.
- *User Level Security*: patterns concerned with achieving security in operational contexts.

Markus Schumacher
Eduardo Fernandez-Buglioni
Duane Hybertson
Frank Buschmann
Peter Sommerlad
SECURITY PATTERNS
Integrating Security and Systems Engineering
WILEY SERIES IN SOFTWARE DESIGN PATTERNS



Architecting Enterprise Solutions

Architecting Enterprise Solutions contains 26 patterns that help building secure, flexible, and available high-capacity internet systems:

- *Fundamental*: patterns that shape the base-line architecture of internet systems.
- *System Performance*: patterns that address performance and throughput.
- *System Control*: patterns concerned with security, logging, tracing, and monitoring.
- *System Evolution*: patterns that help building flexible, evolvable internet systems.

ARCHITECTING ENTERPRISE SOLUTIONS
Patterns for High-Capability Internet-Based Systems
Paul Dyson
Andy Longshaw
WILEY SERIES IN SOFTWARE DESIGN PATTERNS

Server Component Patterns

Server Component Patterns is a pattern language of 37 patterns that illustrates core design concepts for containers as well as fundamental design criteria for components.

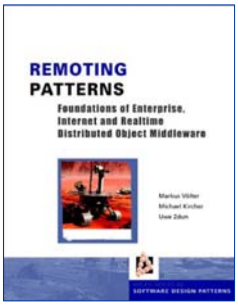
- *Core Infrastructure*: patterns that describe the types of components and their hosting environment.
- *Component Building Blocks*: patterns that help structuring a component.
- *Component Environment*: patterns that support accessing a component in a container and in application.
- *Component Deployment*: patterns that help deploying components.
- The book outlines how each pattern is implemented in EJB, CCM, and COM+.
- A separate part in the book describes the EJB implementation in full depth.

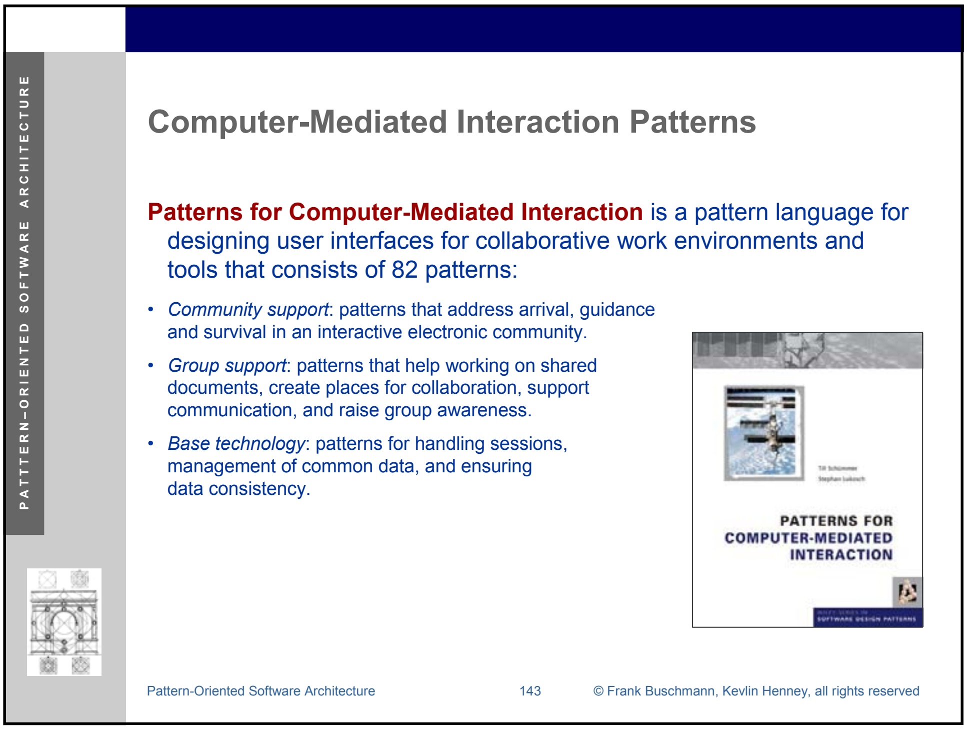
Markus Völter
Alexander Schmid
Eberhard Wolff
SERVER COMPONENT PATTERNS
Component Infrastructures Illustrated with EJB
SOFTWARE DESIGN PATTERNS

Remoting Patterns

Remoting Patterns is a pattern language for remoting that consists of 31 patterns:

- *Basic Remoting*: patterns that detail the Broker architecture underlying remoting infrastructures.
- *Identification*: patterns that help finding and accessing remote objects.
- *Lifecycle Management*: patterns that address the lifecycle of remote objects and support resource management.
- *Extension*: patterns that allow to add out-of-band and QoS functionality to remote objects.
- *Invocation Asynchrony*: patterns that support asynchronous access to remote objects.
- The book also includes technology projections of the language onto .NET, CORBA, and Web Services. It thus provides a vendor-independent view onto remoting.


REMOTING PATTERNS
Foundations of Enterprise, Internet and Realtime Distributed Object Middleware
Markus Völter
Michael Kircher
Uwe Zdun
SOFTWARE DESIGN PATTERNS

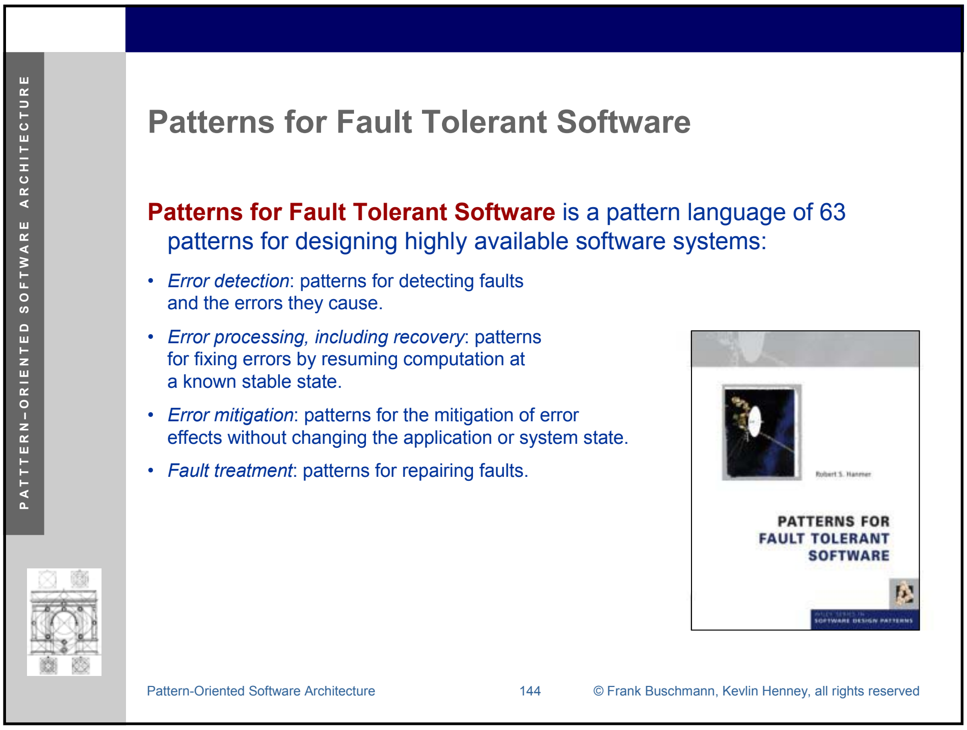


Computer-Mediated Interaction Patterns

Patterns for Computer-Mediated Interaction is a pattern language for designing user interfaces for collaborative work environments and tools that consists of 82 patterns:

- *Community support*: patterns that address arrival, guidance and survival in an interactive electronic community.
- *Group support*: patterns that help working on shared documents, create places for collaboration, support communication, and raise group awareness.
- *Base technology*: patterns for handling sessions, management of common data, and ensuring data consistency.

Till Schümmer
Stephan Lukosch
PATTERNS FOR COMPUTER-MEDIATED INTERACTION
WILEY SERIES IN SOFTWARE DESIGN PATTERNS



Patterns for Fault Tolerant Software

Patterns for Fault Tolerant Software is a pattern language of 63 patterns for designing highly available software systems:

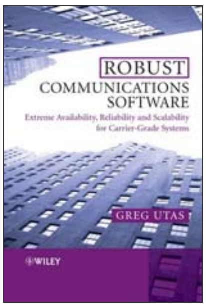
- *Error detection*: patterns for detecting faults and the errors they cause.
- *Error processing, including recovery*: patterns for fixing errors by resuming computation at a known stable state.
- *Error mitigation*: patterns for the mitigation of error effects without changing the application or system state.
- *Fault treatment*: patterns for repairing faults.

Robert S. Hanmer
PATTERNS FOR FAULT TOLERANT SOFTWARE
WILEY SERIES IN SOFTWARE DESIGN PATTERNS

Robust Communications Software

Robust Communications Software is a pattern collection for designing highly available, scalable, and reliable systems:

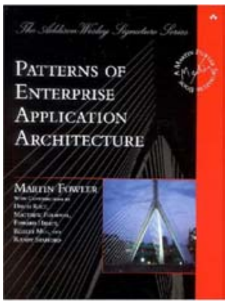
- *Object creation and access*
- *Thread scheduling.*
- *Distribution of work*
- *Fault protection*
- *Recovery*
- *Messaging*
- *Overload handling*
- *Failover*
- *Software installation*
- *System and software operability*
- *Debugging*
- *Capacity management*

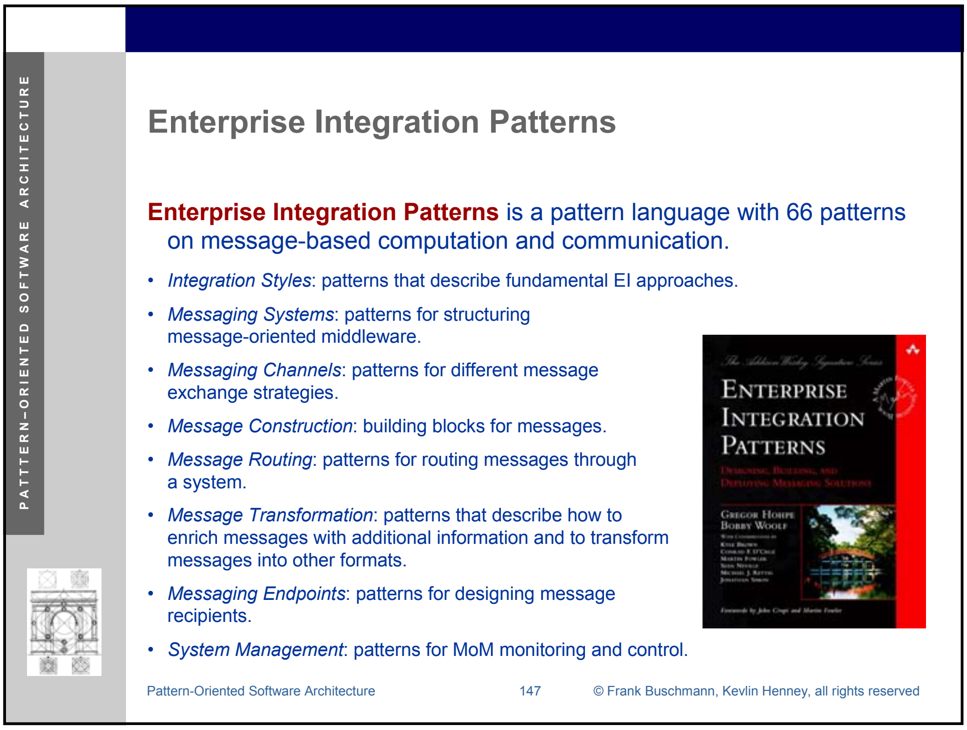

ROBUST
COMMUNICATIONS
SOFTWARE
Extreme Availability, Reliability and Scalability for Carrier-Grade Systems
GREG UTAS
WILEY

Patterns of Enterprise Application Architecture

Patterns of Enterprise Architecture is a pattern language with 51 patterns that illustrates how to design 3-tier enterprise business information applications.

- *Domain Logic*: patterns that help partitioning the application domain into tangible parts.
- *Data Source*: patterns that provide fundamental ways of designing an object-relational mapping.
- *OR-Behavioral*: patterns that help detailing an object-relational mapping.
- *Web Presentation*: patterns regarding the design of web-based UIs.
- Key strength of the book is its fine collection of patterns to map from an object-oriented application to a relational database. About 30 patterns in the language deal with this particular subject.

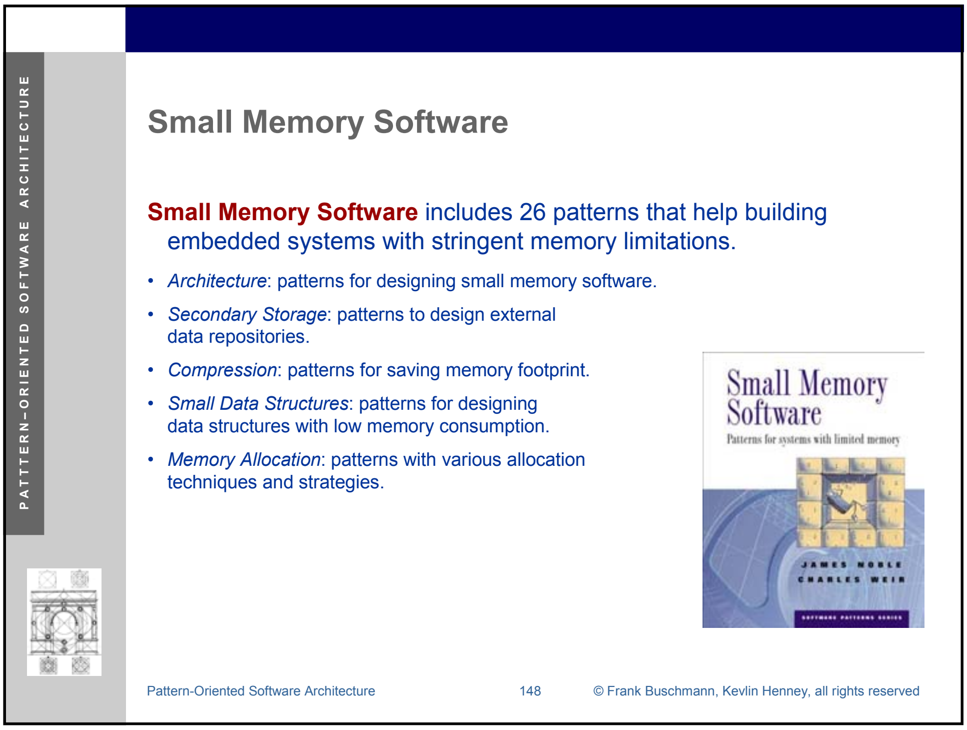




Enterprise Integration Patterns

Enterprise Integration Patterns is a pattern language with 66 patterns on message-based computation and communication.

- *Integration Styles*: patterns that describe fundamental EI approaches.
- *Messaging Systems*: patterns for structuring message-oriented middleware.
- *Messaging Channels*: patterns for different message exchange strategies.
- *Message Construction*: building blocks for messages.
- *Message Routing*: patterns for routing messages through a system.
- *Message Transformation*: patterns that describe how to enrich messages with additional information and to transform messages into other formats.
- *Messaging Endpoints*: patterns for designing message recipients.
- *System Management*: patterns for MoM monitoring and control.



Small Memory Software

Small Memory Software includes 26 patterns that help building embedded systems with stringent memory limitations.

- *Architecture*: patterns for designing small memory software.
- *Secondary Storage*: patterns to design external data repositories.
- *Compression*: patterns for saving memory footprint.
- *Small Data Structures*: patterns for designing data structures with low memory consumption.
- *Memory Allocation*: patterns with various allocation techniques and strategies.

PLoPD Series

PATTERN LANGUAGES OF PROGRAM DESIGN
PATTERN LANGUAGES OF PROGRAM DESIGN 5
PATTERN LANGUAGES OF PROGRAM DESIGN 4

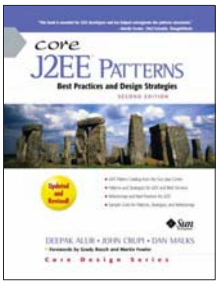
Pattern Languages Of Program Design vol. 1 – 5 include edited collections of patterns from the PLoP (Pattern Languages of Programming) conference series:

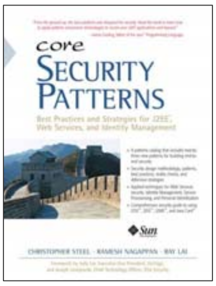
- The patterns in these volumes are not all of high quality, some are even questionable.
- Highlights are definitely the telecommunication analysis patterns (PLoPD1 & PLoPD2), as well as some organizational patterns (PLoPD1), patterns for accessing databases (PLoPD2, PLoPD3 & PLoPD4), as well as some general purpose patterns (in all volumes).

J2EE Patterns

Core J2EE Patterns and **Core Security Patterns** describes the patterns that help to build successful and secure J2EE applications.

- Many Core J2EE patterns also apply in Microsoft's .NET and MCF (formerly code-named Indigo) worlds!


core
J2EE PATTERNS
Best Practices and Design Strategies
DEEPAK ALUR • JOHN CRUPI • DAN MALKS

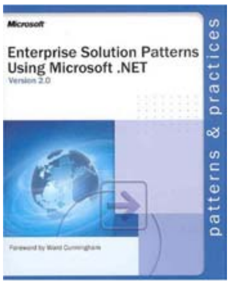


.NET Patterns

Enterprise Solution Patterns Using Microsoft .NET describes 18 patterns that help to build successful .NET applications.

- The patterns are not at all Microsoft specific, but describe how they are implemented in Microsoft .NET or should be implemented when building .NET-based enterprise systems.
- The patterns in this book address the same domain and as Patterns of Enterprise Application Architecture.

Cluster	Problem
Web Presentation	How do you create dynamic Web applications?
Deployment	How do you divide an application into layers and then deploy them onto a multi-tiered hardware infrastructure?
Distributed Systems	How do you communicate with objects that reside in different processes or different computers?
Performance and Reliability	How do you create a systems infrastructure that can meet critical operational requirements?


Microsoft
Enterprise Solution Patterns Using Microsoft .NET
Version 2.0
patterns & practices
Foreword by Ward Cunningham

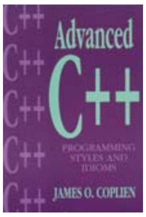
Programming Patterns

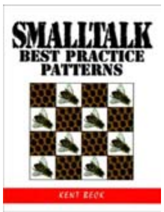
Advanced C++ Styles and Idioms presents useful patterns that help mastering the C++ language.

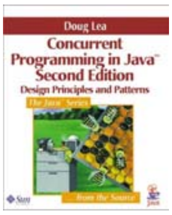
Smalltalk Best Practice Patterns presents more than 90 idioms for programming in Smalltalk. Yet many of these patterns apply to other languages as well, in particular C++ and Java.

Implementation Patterns presents 77 patterns for code-level detail, with a focus on Java.

Concurrent Programming in Java presents many patterns that help with implementing concurrent programs in Java. Many patterns apply in other languages as well, specifically in C++.



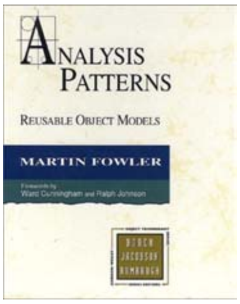



Doug Lea
Concurrent Programming in Java
Second Edition
Design Principles and Patterns
The Java Series
Sun
from the Source
Java

Analysis Patterns

Analysis Patterns includes a collection of patterns that describe the structure and workflow of systems in the health care and finance application domains.

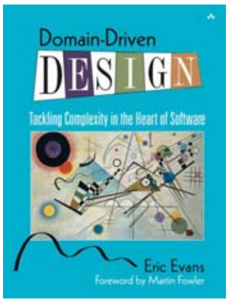
- Yet many of these patterns apply in other domains as well, for instance in most business information systems, as well as in a large number of systems that observe and measure values, and trigger actions in response to these observations and measurements, such as process control systems.

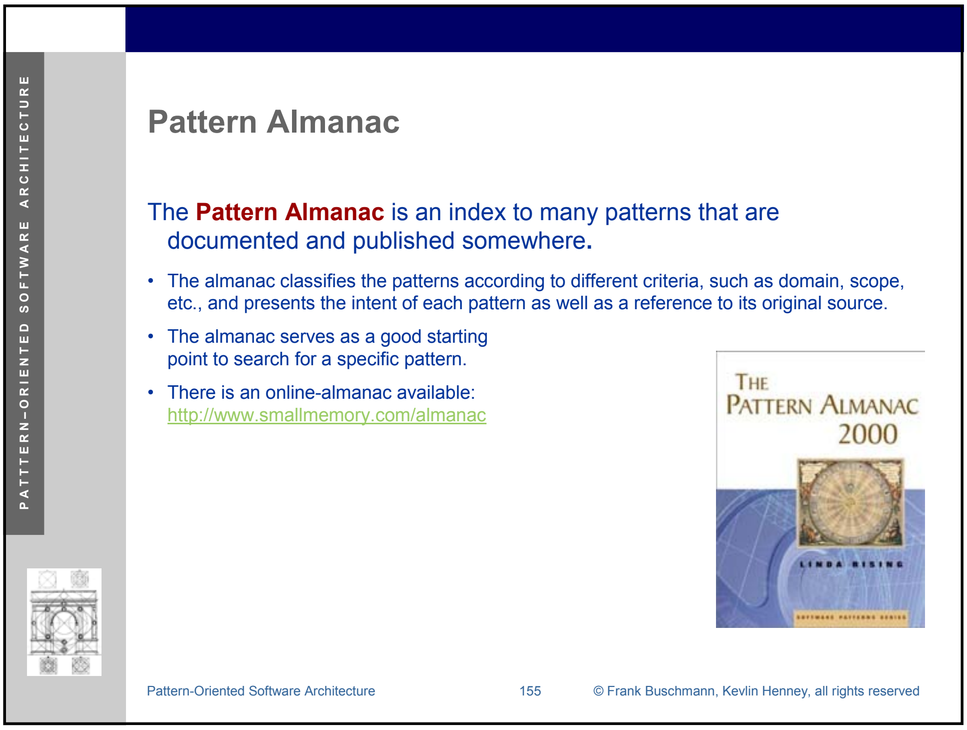


Domain-Driven Design

Domain-Driven Design is a useful pattern language that helps you to identify a proper domain model for an application and to transfer this model into a feasible component-based software architecture.

- Domain-Driven Design is more about development process rather than software technology or software architecture.
- The book helps you keeping the focus on the "business" case of a software system, and providing a partitioning of its functionality that is appropriate for that business.

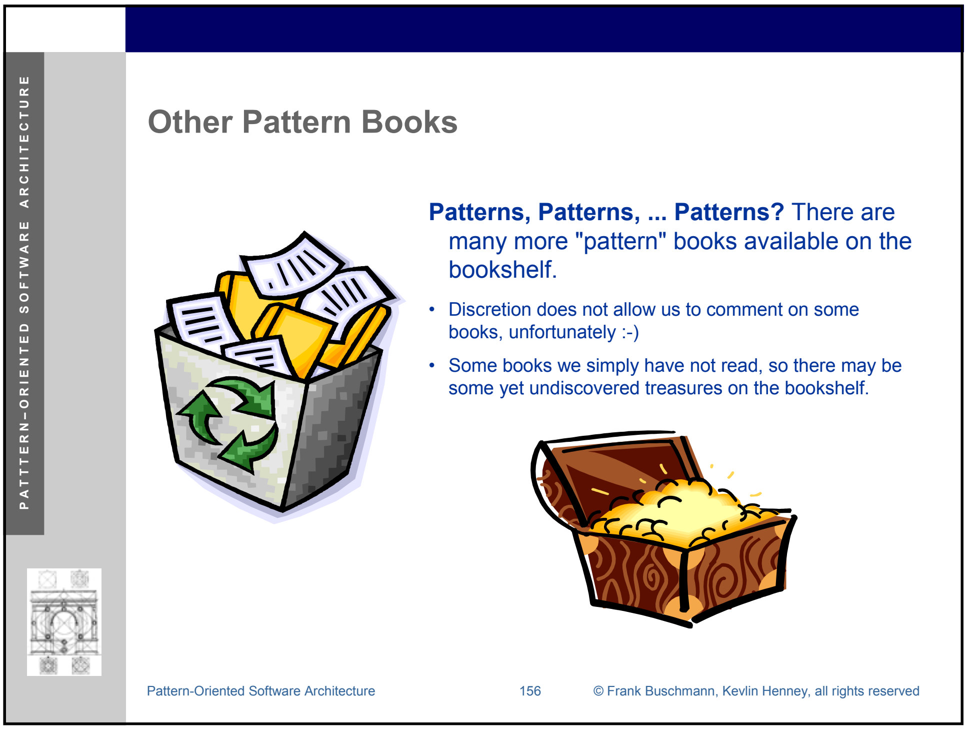

Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler



Pattern Almanac

The **Pattern Almanac** is an index to many patterns that are documented and published somewhere**.**

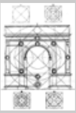
- The almanac classifies the patterns according to different criteria, such as domain, scope, etc., and presents the intent of each pattern as well as a reference to its original source.
- The almanac serves as a good starting point to search for a specific pattern.
- There is an online-almanac available: http://www.smallmemory.com/almanac



Other Pattern Books

Patterns, Patterns, ... Patterns? There are many more "pattern" books available on the bookshelf.

- Discretion does not allow us to comment on some books, unfortunately :-)
- Some books we simply have not read, so there may be some yet undiscovered treasures on the bookshelf.



References (1)

C. Alexander: The Timeless Way of Building, Oxford University Press, 1979

M. Fowler: UML Distilled, Addison-Wesley, 1997

R.P. Gabriel: Patterns of Software, Oxford University Press, 1996

E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

Various Eds. : Pattern Languages of Program Design, Vol. 1- 5, Addison-Wesley, 1995, 1996, 1997, 1999, 2005

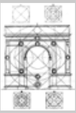
F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: Pattern-Oriented Software Architecture—A System of Patterns, John Wiley and Sons, 1996

D.C. Schmidt, M. Stal , H. Rohnert, F.Buschmann: Pattern-Oriented Software Architecture—Patterns for Concurrent and Networked Objects, John Wiley and Sons, 2000

M. Kircher, P. Jain: Pattern-Oriented Software Architecture—Patterns for Resource Management, John Wiley and Sons, 2004

F. Buschmann, K. Henney, D.C. Schmidt: Pattern-Oriented Software Architecture—A Pattern Language for Distributed Computing, John Wiley and Sons, 2007

F. Buschmann, K. Henney, D.C. Schmidt: Pattern-Oriented Software Architecture—On Patterns and Pattern Languages, John Wiley and Sons, 2007



References (2)

M. Fowler: Analysis Patterns, Addison-Wesley, 1997

E. Evans: Domain-Driven Design, Addison-Wesley, 2004

M. Fowler: Patterns of Enterprise Application Architecture, Addison-Wesley, 2003

G. Hohpe, B. Woolf: Enterprise Integration Patterns, Addison-Wesley, 2004

M. Völter, M. Kircher, U. Zdun: Remoting Patterns, John Wiley and Sons, 2004

J. Noble, C. Weir: Small Memory Software, Addison-Wesley, 2000

K. Beck: Smalltalk Best Practice Patterns, Prentice Hall, 1997

J.O. Coplien: Advanced C++ Styles and Idioms Addison-Wesley, 1992

D. Lea: Concurrent Programming in Java, Design Principles and Patterns, Addison-Wesley, 1999

M. Völter, A. Schmid, E. Wolff: Server Component Patterns — Component Infrastructures illustrated with EJB, John Wiley and Sons, 2002

D. Alur, J. Crupi, D. Malks: Core J2EE Patterns – Best Practices and Design Strategies, Prentice Hall, 2001

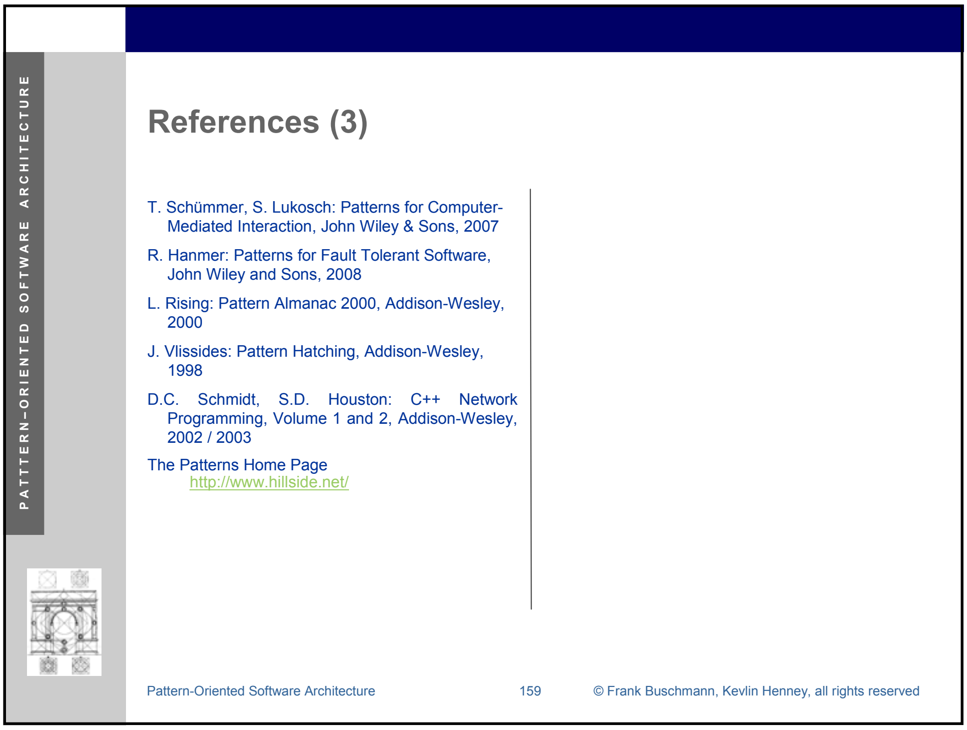
C. Steel, R. Nagappan, R. Lai: Core Security Patterns: Best Practices and Strategies for J2EE Web Services, and Identity Management, 2005

Microsoft Corporation: Enterprise Solution Patterns Using Microsoft .NET, Microsoft Press, 2003

S. Metsker: Design Patterns Java Workbook, Addison-Wesley, 2002

S. Metsker: Design Patterns C#, Addison-Wesley, 2004

G. Utas: Robust Communications Software – Extreme Availability, Reliability, and Scalability for Carrier Grade Systems, John Wiley and Sons, 2006



PATTTERN–ORIENTED SOFTWARE ARCHITECTURE

References (3)

T. Schümmer, S. Lukosch: Patterns for Computer-Mediated Interaction, John Wiley & Sons, 2007

R. Hanmer: Patterns for Fault Tolerant Software, John Wiley and Sons, 2008

L. Rising: Pattern Almanac 2000, Addison-Wesley, 2000

J. Vlissides: Pattern Hatching, Addison-Wesley, 1998

D.C. Schmidt, S.D. Houston: C++ Network Programming, Volume 1 and 2, Addison-Wesley, 2002 / 2003

The Patterns Home Page
http://www.hillside.net/