Michael Krisper, BSc

Finding the Right Design Pattern Using Binding Time Properties

MASTER'S THESIS

to achieve the university degree of
Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to
Graz University of Technology

Supervisor
Dipl.-Ing. Dr. Christian Kreiner

Institute of Technical Informatics

Graz, October 2016

This document is set in Palatino, compiled with pdfLaTeX2e and Biber. The used LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Graz, _____ _____

Datum Unterschrift

Abstract

In this thesis 120 design patterns from standard literature are described and analyzed for their respective binding times (Books: GoF, POSA1, POSA2, POSA3, POSA4). Binding time is an important aspect of design patterns to decide which one to choose for a specific purpose (to find the right pattern). Several binding time scenarios are presented which compare the different patterns on a temporal continuum from early static binding to late dynamic binding (as applicable). These scenarios give insights which pattern to apply for which purpose to achieve a flexible software design without over-engineering or inappropriately high complexity. For a better understanding the used design patterns are described and their binding time is specified in detail.

Thanks

I want to thank my supervisor Christian Kreiner for inspiring me and supporting me to write about the topic of Binding Time and Design Patterns. Also I want to thank my girlfriend Melanie who displayed a huge amount of patience during my turmoils of working on this thesis. Additionally I want to thank my family, my professors and my teachers.

Contents

List of Figures

1 Introduction

Design patterns play a huge role in the development of modern software applications. They give solution templates for commonly reoccurring problems. Understanding how and when to apply them, helps building more robust and flexible applications. This thesis is targeting software architects, designers and developers in the context of designing and developing a software application. The goal of this thesis is to analyze the binding times of the many different design patterns to better understand the consequences and to give a guide which pattern to apply for which binding time requirement. Applying design patterns without thinking about the binding time consequences could impose unnecessary complexity in the software design and therefore lead to performance or quality problems.

Some decisions are made early during design and implementation, others are deferred until runtime and decided dynamically during execution of an application. Although dynamic implementations give more freedom and flexibility later on, it often imposes another layer of abstraction which increases the complexity and takes longer to develop. If this flexibility is not needed it was a waste of resources. To avoid this it is important to know which binding times a pattern has, which flexibility it allows and what the effort and increase in complexity a pattern imposes.

Chapter 2.1: Design Patterns gives an introduction to design patterns as well as pattern forms which describe them. Chapter 2.2: Binding Time defines binding time and explains the basic terminology in the dimensions of binding time, participants and responsible role which is used throughout the whole work. Chapter 3: Finding the Right Pattern gives the main overview over some binding scenarios and the applicable patterns.
In the remaining Chapters 4-8 design patterns taken from five of the most important books in the design pattern standard literature are described and analyzed.

The following books are the basis for this work's analysis. All design patterns which are defined there are presented also in this work and are analyzed for their respective participants and binding times:

- **GOF**: Design Patterns - Elements of Reusable Object-Oriented Software (Gamma et al., 1995), *23 Patterns*, described in Chapter 4.
- **POSA1**: Pattern-Oriented Software Architecture Volume 1: A system of patterns (Buschmann, Meunier, et al., 1996), *15 Patterns*, described in Chapter 5.
- **POSA2**: Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects (Schmidt et al., 2000), *17 Patterns*, described in Chapter 6.
- **POSA3**: Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management (Kircher and Jain, 2004), *10 Patterns*, described in Chapter 7.
- **POSA4**: Pattern-Oriented Software Architecture Volume 4: Pattern Language for Distributed Computing (Buschmann, Henney, and Schmidt, 2007), *55 Patterns*, described in Chapter 8.

2 Background and Related Work

2.1 Design Patterns

A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context.

— (Gamma et al., 1995)

In this work 120 design patterns from standard literature are described and analyzed. For that purpose it makes sense to define what a design pattern is, and how it can be described.

Design Pattern A design pattern is a solution template to a commonly recurring problem in a given context. Patterns can be found by abstracting the core solution concepts from existing solutions. Mature patterns are already reinvented and applied in multiple software projects by different people and have proven helpful and useful. Such patterns are worthy describing and sharing, to give others the knowledge to also implement such good solutions.

2.1.1 Pattern Forms

Describing patterns has the purpose of knowledge transfer to other people in order to give them opportunity to also apply these approved and good solutions for their own problems. To better understand the applicability, the context, problem, and forces have to be described as well as the solution and consequences. To do that several pattern forms have emerged which all try to do good descriptions of pattern.

There are several different forms how to write a pattern. Here is a list of the most prominent ones in the software sector (C2, 2016):

- Compact Form (Minimal Form)
- Alexandrian Form
- GOF-Form
- Portland Form
- POSA1/2/3 Form
- POSA4 Form
- Beck Form
- Coplien Form (Canonical Form)

In the following sections these forms are described in detail.

Compact Form (Minimal Form)

Common ground for all patterns. This should be the minimal description a pattern has:

- Name
- Context
- Problem
- Solution
- Consequences

Alexandrian Form

> *For convenience and clarity, each pattern has the same format. First, there is a picture, which shows an archetypal example of that pattern. Second, after the picture, each pattern has an introductory paragraph, which sets the context for the pattern, by explaining how it helps to complete certain larger patterns. Then there are three diamonds to mark the beginning of the problem. After the diamonds there is a headline, in bold type. This headline gives the essence of the problem in one or two sentences. After the headline comes the body of the problem. This is the longest section. It describes the empirical background of the pattern, the evidence for its validity, the range of different ways the pattern can be manifested in a building, and so on. Then, again in bold type, like the headline, is the solution - the heart of the pattern - which describes the field of physical and social relationships which are required to solve the stated problem, in the stated context. This solution is always stated in the form of an instruction - so that you know exactly what you need to do, to build the pattern. Then, after the solution, there is a diagram, with labels to indicate its main components.*
> *After the diagram, another three diamonds, to show that the main body of the pattern is finished. And finally, after the diamonds there is a paragraph which ties the pattern to all those smaller patterns in the language, which are needed to complete this pattern, to embellish it, to fill it out.*
>
> — Christopher Alexander, (Alexander, 1979)

The alexandrian form was one of the first forms to write patterns. Although it was established for architectural patterns it can be applied to software development (with slight modifications). It typically is built up like this:

- Picture
- Prologue
- Three Diamonds (◇ ◇ ◇)
- **Problem**
- Discussion
- **Solution**
- Diagram
- Three Diamonds (◇ ◇ ◇)
- Epilogue

GoF Form

Established in the Gang-Of-Four-Book (Gamma et al., 1995).

- Name
- Classification
- Also Known As (optional)
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

Portland Form

Short and simplified narrative Form which is close to Alexandrian Form, but much shorter. Uses the word "**Therefore:**" as delimiter between Problem and Solution.

POSA1/2/3 Form

Established in the POSA1, POSA2, POSA3 Books (Buschmann, Meunier, et al., 1996; Schmidt et al., 2000; Kircher and Jain, 2004).

They distinguish between three category of Patterns: Architectural Patterns, Design Patterns, Idioms. Each layer is more detailed.

- Name
- Also Known As
- Example
- Context
- Problem
- Solution

- Structure
- Dynamics
- Implementation
- Example Resolved
- Specialization (only POSA3)
- Variants
- Known Uses
- Consequences
- See Also
- Credits (only POSA2, POSA3)

POSA4 Form

Established in the POSA4 Book (Buschmann, Henney, and Schmidt, 2007).

Written in a more narrative Style with text formatting. (more like the Alexandrian Form). Uses the Word "Therefore:" as delimiter between Problem and Solution.

- Pattern name and maturity level
- Inbound Patterns
- Context
- Border Line
- Problem statement
- Forces
- "Therefore:"
- Solution instruction
- Solution Sketch
- Solution structure and behavior
- Solution consequences
- Solution details and outbound patterns

Beck Form

Established in Smalltalk Best Practice Patterns by Kent Beck (Beck, 1996)

- Title
- Context
- Problem Question
- Forces
- Solution
- Resulting Context

Canonical Form (Coplien Form)

Established in the PLOPD Books (see (Coplien, 1995; Coplien and Vlissides, 1996)).

- Name
- Alias (optional)
- Problem
- Context
- Forces
- Solution
- Example (optional)
- Resulting Context
- Rationale (optional)
- Known Uses
- Related Patterns

2.1.2 Pattern Form used in this Work

Due to the fact that all patterns described here are already established in the books (GoF, POSA1, POSA2, POSA3, POSA4) and described there in detail, this work only gives a short description and a diagram to illustrate to the reader how we understand the pattern. After this short description, the section "Participants and Bindings" in each pattern describes the binding times of all contained participants in it.

- **Name**: The name or title of the pattern as it was given in the books.
- **Description**: A description of the pattern structure to give the reader an idea of its implementation and behavior.
- **Diagram**: A diagram showing the pattern structure with a kind of UML class diagrams (and often additional elements to depict dynamics or architectural aspects).
- **Participants and Bindings**: A list of the participants in the pattern and their respective dependencies and binding times.

Diagrams

The diagrams used in this work are based on UML class diagrams (Booch, Jacobson, and Rumbaugh, 1999) with the difference, that for simplicity some architectural elements are added to simplify and hide implementation details which are not relevant in general or unique to specific programming languages.

One basic assumption for the diagrams is a language which support object orientation, interfaces, inheritance, constructor, destructor and collections (arrays and dictionaries). Some patterns also need extra functionality somehow available (either via standard library, or external libraries) like some kind of synchronization (e.g. mutex, semaphore or condition variables) and threading (e.g. thread, timer). These technologies are available in many languages and therefore the patterns which need those are applicable in a broad range of programming languages.

2.2 Binding Time

Binding time is defined as "*the latest time during software lifecycle, when something flexible becomes decided and fixed*" (Kreiner, 2013). It depicts the temporal aspect of a binding and can generally be described using the time continuum of the program lifetime. Additionally the bound participant and the responsible role have to be defined. Figure 2.1 shows these dimensions used for analyzing binding time.

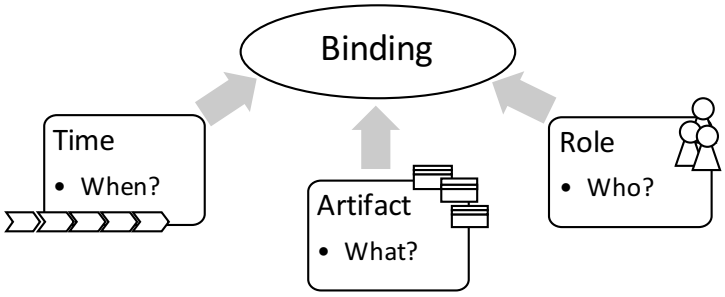

Figure 2.1: Binding

- **Binding-Time**: *When is it bound?* The phase of a program lifetime when the binding happens.
- **Participant**: *What is bound?* The software-artifact which is bound.
- **Role**: *Who binds it?* The role responsible for the binding decision.

There exist many binding-mechanisms which have different consequences for binding-time. These binding mechanism are defined in the programming language or standard library and are used implicitly while implementing a software application. Some Examples are: hardcoded method calls and type declarations, conditional compilation, object-orientation (encapsulation, inheritance, polymorphism, abstraction), applying design patterns, aspect oriented programming, using reflection, configuration, dynamic dispatch, dynamic loading, etc.

2.2.1 Phase

During the development of a software application the projects goes through different phases of completion. The easiest model is the distinction between compile-time and run-time. Another one is compile-time, link-time and start-up time (Linden, Schmid, and Rommes, 2007). More sophisticated models divide these into several fine-grained steps like requirements-analysis, design-time, compile-time, deployment, run-time (Capilla and Bosch, 2013; Fritsch, Lehn, and Strohm, 2002; Jaring and Bosch, 2002; Krueger, 2004). Each of these steps has its own form of binding.

The definition of a program timeline as in the following Section2.3 is convenient for analyzing software related binding times, but other domains could have different time lines. To give an example: Fritsch et al. defined another set of binding times for the domain of automotive embedded systems: Programming, Integration, Assembly, and Run Time (Fritsch, Lehn, and Strohm, 2002). While covering most the phases of the development of software application the program timeline does not incorporate a whole project plan or industrial software product lines. These span up new timelines which coexist in parallel and potentially overlap and synchronize sometimes (Myllymäki, 2001).

Additionally to the program lifetime also other lifelines exist in projects where binding may happen, e.g.:

- **Program Life-Cycle** (Krueger, 2004): Language Design, Platform Design, Source Reuse, Implementation/Generation, Pre-Compile, Compile, Package, Link & Load, Customization, Install, Start-up, Runtime, Dynamic Loading, Scripting, Unloading.
- **Software Development Life-Cycle**: Requirements Analysis, Architecture and Design, Implementation, Testing, Deployment, Maintenance
- **Software Product Lines**: Reuse, Configuration, Generation
- **Runtime Adoption**: Always On, Live-Updating System, Real-time Applications

2.2.2 Participant

An integral part of a binding is the participant. The participant is the actual artifact which is bound. In previous work this was directly called artifact (Krisper and Kreiner, 2016), but for design patterns it is more useful to talk about it as a participant. To analyze binding these participants have to be defined. Typical examples are a class, a object, a variable an specific instantiation of a type, or some behavior like a method, which could by dynamically dispatched at runtime. Some more examples:

- **Object**: A software-artifact which can be handled in the source code.
- **Instance**: A real instantiated object with an specific type.
- **Type**: A concrete description of properties and method-signatures of an object.
- **Interface**: An abstract description of properties and method-signatures
- **Method**: The definition and implementation of a function (optionally with parameters), bound to an object
- **Function**: The definition and implementation of a behavior or calculation in the source code
- **Property/Member**: A value with specific type which is part of an object.
- **Method-Call**: The process of executing a method with concrete parameter values from an outer context.
- **Interface-Implementor**: An object which implements an interface.
- **Wrapper-Class**: A class containing another object and using its properties and methods for its purpose.
- **Language features**: Syntactic and Semantic elements of a programming language.

2.2.3 Responsible Role

Every binding is decided by a mechanism which is somehow responsible for that binding. This mechanism is here defined as the responsible role. For example a software developer could decide an implementation detail during implementation, or a software architect decides which structure to use during the design phase. Even before that, the requirements are defined by the project leader and the customer. But decisions could also be made very late, if a software is configurable and flexible. Such late bindings are configured by the end-user when an application is already deployed and running (even after the actual development stopped). Obviously most software related binding decisions in detail are made by the software architect or some developer. The participants and stakeholders of a project all have their roles and could potentially decide binding.

- **Pattern-Designer**: Writes software design patterns. Decides the name and structure of the pattern.
- **Language-Designer**: Person or Company responsible for designing the syntactical and semantical definitions of a programming language.
- **Project-Manager**: Person who is coordinating a software project and is responsible for defining the requirements and resource limitations together with the customer.
- **Software-Architect**: Person who designs the software architecture and environment according to the requirements.
- **(Senior-)Developer**: Person who implements software and decides smaller but more frequent design decisions. Is lastly responsible for the whole source code and implementation.
- **Customer**: The project customer of a software project.
- **Configurator**: A person responsible for configuring a software product to the needs of a end-user.
- **Maintainer**: Person who maintains a software and keeps the environment up-and-running and up-to-date.
- **End-User**: A person which really has to work with a software-product in order to accomplish the daily work-load.

2.2.4 Binding Sequence

On an variation point a binding always goes through the following sequence:

1. **Undecided**: Binding has not happened yet. Variations are still open. Changes could still be made. Errors could still be mitigated.
2. **Binding**: Binding happens just now. The variation is bound, the decision is made.
3. **Fixed**: The binding has happened and the consequences are now realizing. If errors happened in the binding decision, these can only be recovered, resisted or rolled back (mitigation not possible anymore).

As already mentioned in the descriptions, this has a huge impact on error handling. Beforehand it is relatively easy to mitigate errors by simply deciding for the correct binding. Afterwards it is more difficult, because some languages even don't allow for a direct correction of bound variables. The corresponding method has to be reverted and called again, now with the correct parameters to decide for the correct binding.

2.3 Program Timeline

The actual time is the most important aspect of binding time. The program timeline is a sequentially increasing list of time phases during the development of a software application from the developer's view. It covers the whole development process, beginning with the requirements phase, going from architecture and design phase to the actual implementation, compilation, deployment and runtime. This temporal continuum is the basis for the analysis of design patterns in this thesis.

The program timeline can be seen in Figure 2.2 and the phases are described hereafter.

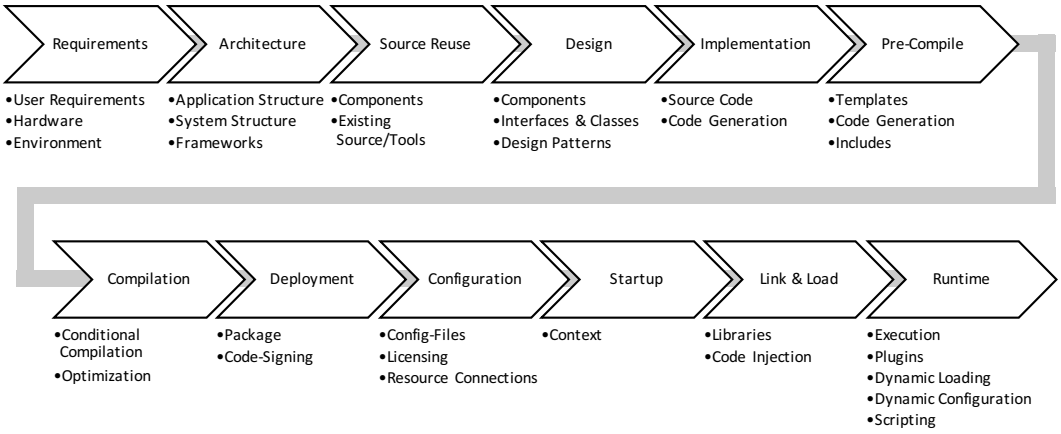

Figure 2.2: Program Timeline

In the following paragraphs these phases and their impact on the binding time are described.

Requirements The project phase where the requirements are defined. This happens before any line of code was written and sometimes even before the project even started. The customer, the stakeholders and the general environment set the requirements for the software application. This can be distinguished in functional and non-functional requirements as well as cus-

tomer requirements and internal requirements. The implications are huge because this is the basis for all further architectural thoughts and decisions, as well as all further work. The requirements influence many design decisions afterwards. This is also the phase which has the highest contact points to project management and customer relations. Typical outcome is a project description with a detailed plan of the requirements, quality of service contracts and also monetary and time restrictions.

Architecture One of the first phases in a software project is the Architecture Time wherein the architecture of the application is specified. Decision made in this phase are which basic technology is used, which platforms and which general structures the application will have. The decisions are mostly made by the software architect in coordination with the project manager and the senior developers. These decisions are the guidelines for every follow-up decision and have the highest impact in the whole software development process. Overview Diagrams, Component Diagrams, or Network Diagrams are the general outcomes of this phase as well as descriptions of the Quality Requirements and Development Guidelines.

Source Reuse In this phase it is decided which tools, components and libraries are used and which parts of the software are implemented new by the software-development team. The used software component often have implications for the design. This could mean that a framework demands a specific implementation, or that internal development components demand a specific structure. This has huge consequences for the design, and therefore has to be done before that. Typical outcomes for this phase is a description of used software components and libraries as well as the libraries themselves. Often they have to be acquired and this also happens during this phase.

Design In the design phase the internal structure of the software components is decided. Which domain objects exist, how

they are represented in the software. Which Design Patterns are applied. The general class and object structure is planned and decided by the senior-developers in junction with the other development team. The general outcomes for this phase are class diagrams, sequence-diagrams, flow-diagrams, or other detailed descriptions of the software structure.

Implementation The actual coding of the source code or creation of the resources. In this phase the software developers implement the functionality and class structure according to the requirements and class-definitions made in the previous phases. In this phase small detailed decisions are made with every line of code. How readable is the code? How flexible is the code? Does the code work at all? Should I take a array or a list for that list of object?

During implementation sometimes problems arise, that some design decisions are wrong, or even some requirements cannot be fullfilled with the used architecture or technology. This leads to feedback-loops where the architecture and design is revised, or even discussions with the customer to loosen up on the requirements or shift some features to later version of the software. Typical outcomes of the implementation phase are the source code, images, configuration files, media files.

Pre-Compile In this phase the source code it automatically generated or modified through text replacement mechanisms like Includes or Macros or generative code templates. During this phase many decisions made earlier are manifested in the sourcecode. Also decisions made in the coding templates manifest during this phase. Typical outcome are the same as during the implementation phase: source code, images, configuration files, media files.

A typical example for pre-compile processing is replacement of constant values in C++. Another would be the generation of Code via T4-Template in Visual Studio.

Compile-Time The compile-time is the phase where the source code and all resources are translated and bundled into an application. In this phase the compiler manifests many decisions which are made before. How big are the datatypes? Code optimizations? Generate machine code according to the calling conventions and memory layout and so on. This phase in itself is an own research area (compiler construction (Wirth, 1996)) and could be split up into several sub-phases: parsing, lexing, type inference, code optimization, code generation. Even after compilation there exist post-compile-tools which change the generated byte-code or assembler-code afterwards to accomplish some kind of aspect-oriented weaving. These sub-phases are on a lower abstraction level during compilation and will not be covered in detail here.

For interpreted languages the compilation-phase is completely shifted to the runtime of the interpreter (and repeated in a degenerated way for every new line of code).

The output of the compilation phase could typically be an executable application, a library, or intermediate byte code which could be interpreted and executed in an highly efficiently way in a runtime environment (as it is the case in .NET Languages and Java).

Deployment During deployment the compiled application package is packed, delivered and installed at the customer. This is typically done via installer or packages (installer files like msi/dmg, ios or android apps, or packages in linux package managers).

The deployment phase fixes decisions for the environment where the applications should be installed. Meta-information like the author, installation hints, or installation requirements are defined here.

Configuration After the application is deployed it can be configured to the users needs. This is done during the "configuration-time". Typically the definitions are stored in an configuration file (e.g. ini-file, or xml-settings) or a database and read on startup

or during runtime. The configuration influences some behavior of the application (e.g. restricted areas, or license checks, or database connections, ...).

Startup During startup the application gets environment information like system paths, time information, access-rights, executing user, or general platform values like cpu-count or is gpu-support available or not. Beside the environment variables, these decisions are made implicitly by the user just by executing the application on the target platform.

Link and Load Linking and loading the needed libraries at startup of an application. During this phase the system paths or local paths are searched for the needed libraries to load. During this phase external code is referenced and loaded into the application. This is a phase in its own right, because the libraries could change (version updates, DLL Hell (Nord, 2011)) and therefore bindings could be different.

Runtime During runtime the actual application logic is executed and interaction with other systems and the user happens. During runtime an application could e.g. run in multiple threads, spawn other processes, communicate over the network, wait for user interaction, do sophisticated calculations, and much more.

Depending on the implementation an application could even apply very late binding mechanisms like loading libraries dynamically into the memory at runtime or provide a kind of scripting system (e.g. with the Interpreter (47) Pattern) to allow the user write own applications. Also configuration files or database configuation could be read during runtime to change the behavior and look of the application. Depending on the programming language capabilities it is even possible to change its own code during runtime (interpreted languages often allow this by default, and compiled languages sometimes allow this via reflection or dynamic compilation to extend functions and class definitions).

2.4 Context and Motivating Scenarios

2.4.1 Software Architect Scenario

Consider a software architect planing the software design for an application. According to the project requirements and resource constraints he has to decide the overall structure and capabilities of an application. Concerns like flexibility, variability and embracing changing requirements have to be considered. Many things in an application depend upon the overall design and therefore this has to be done very carefully. Design patterns represent proven solution templates for a problem within a given context and therefore come in handy building a flexible and extensible model. They also help govern a healthy and understandable communication culture amongst the project team.

The architect has to balance two antagonizing forces here: The design should fulfill all requirements (functionality, quality, flexibility, ...) but this has to be accomplished with only limited resources (time, money, deadlines, staff, ...). This balancing act results in a design which could be very flexible (having late binding times) or rather rigid (with early binding times) or a reasonable mixture. Taken to the extreme, this leads to two problem scenarios:

Oversimplification While it is highly desirable to design a very flexible, extensible, future-proof architecture to cope with changes of requirements, it often cannot be done due to resource limitations (time, money). This leads to a cheaper design with early binding times, which is easier and faster to implement, but has its drawbacks in flexibility and extensibility.

Indecisive Generality While it is one of the recommended practices to defer decisions as long as possible (Kandt, 2003), this could lead to a situation called indecisive generality (Marquardt, 2005). Sometimes a decision cannot be made due to unclear requirements and therefore is either deferred or implemented

in an abstract way, so it can be changed easily later on. This unnecessary flexibility makes the source code more complicated, harder to implement and understand, which ultimately results in higher development and maintenance costs and higher risk of bugs in the source.

2.4.2 Software Developer Scenario

Consider a software developer implementing a specific function in a software project. While many of the interface and overall structure decisions are already made during the design phase, the internal mechanisms are implemented later on, and with it also many ad-hoc decisions made by the developer. While a software architect has to cope with external constraints, the developer has to solve internal requirements and quality issues, like functionality, runtime performance, code readability, testing, security, etc. Small decisions could have huge impact, like choosing a list, set, or dictionary as data structure; or deriving from an existing object or just wrap it in a decorator; or choosing which variant of a design pattern to implement. The coding style itself also greatly impacts the quality. Readability, code understanding, simplicity of changing and refactoring code highly depend upon this implementation decisions and therefore impact the flexibility of the software on an internal level. These decisions are manifested in the source code during implementation and therefore also have a binding time.

Imagine a task which consists of going through a list of objects and aggregating some values. This could be done with a simple for-loop, but also with the help of iterators, or a functional approach like map-reduce and deferred execution (e.g. LINQ in C#). All of these are just implementation details but have different consequences for the application affecting memory and performance. Using a deferred execution approach shifts the actual processing until the result is actually needed (later binding time), while a for-loop (with early binding time) could be easier to implement, debug and understand in some cases.

2.5 Related Work

This thesis is mainly based on the following books:

- **GOF**: Design Patterns - Elements of Reusable Object-Oriented Software (Gamma et al., 1995)
- **POSA1**: Pattern-Oriented Software Architecture Volume 1: A system of patterns (Buschmann, Meunier, et al., 1996)
- **POSA2**: Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects (Schmidt et al., 2000)
- **POSA3**: Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management (Kircher and Jain, 2004)
- **POSA4**: Pattern-Oriented Software Architecture Volume 4: Pattern Language for Distributed Computing (Buschmann, Henney, and Schmidt, 2007)

All design patterns described in these books are also in described in this thesis and analysed for their binding times.

Much of the definitions for binding time is taken from "Describing Binding Time in Software Design Patterns" by Krisper and Kreiner (Krisper and Kreiner, 2016) as well as "A binding time guide to creational patterns" by Kreiner (Kreiner, 2013).

Binding of variables and methods is in the core of programming languages. Different languages and programming paradigms have different binding mechanisms and these are well described and well known in the programming community, as described in (Burch, 2012) e.g. dynamic dispatch, late binding of methods, name binding, dynamic typing, deferred execution, etc.

Going beyond this technical level, Fritsch et al. proposed binding time as one of the several aspects for evaluating implementation mechanisms (Fritsch, Lehn, and Strohm, 2002). Myllymäki wrote about variability management and discussed the of binding and binding mechanisms as well as language capabilities in detail (Myllymäki, 2001). Regarding software product lines Svahnberg gave a taxonomy of variability where the binding time is one decision factor for introducing variants (Svahnberg, Gurp, and Bosch, 2005). Krueger also proposed a more general

taxonomy and graphical representation of software product lines and their variations (Krueger, 2004). In contrast to this broader and more general view, Capilla and Bosch give a detailed view on binding time and described different phases (Capilla and Bosch, 2013). They also go into the aspect of having multiple possible binding times available for creating variants.

In regards of design patterns Gamma et al. discuss the general scope of patterns (class level, object level), (Gamma et al., 1995) and sometimes discuss the governing forces and consequences in terms of binding time, but in a narrative and open form. Other relevant literature for design patterns are the "Pattern Languages of Program Design" (PLOPD) books (Coplien, 1995; Coplien and Vlissides, 1996; Martin, Riehle, and Buschmann, 1007; Foote, Harrison, and Rohnert, 1999; Manolescu, Voelter, and Noble, 2006).

3 Finding the Right Pattern

Many of the design patterns solve similar problems in approximately similar contexts and therefore can be compared in regards of binding time. This chapter gives an overview over the comparable scenarios and patterns and align them on the basis of binding time. The patterns are always listed in order from early binding going to late binding.

Many of the described patterns are not listed in this scenarios, and there are several reasons for that. Either they solve a unique problem which cannot be compared to others, or they give only a partial solution for the already listed patterns, or they are so general that their binding time could be anywhere depending on the implementation.

Binding Scenarios:

- Method Call Bindings (Dispatching Patterns)
- Object Creation Bindings (Creational Patterns)
- Behavior and Processing Bindings (Behavioral Patterns)
- Concurrency and Synchronisation Bindings (Concurrency Patterns)

3.1 Method Calls - Dispatching Patterns

The calling patterns have the purpose of finding, creating or dispatching the correct object or method to call.

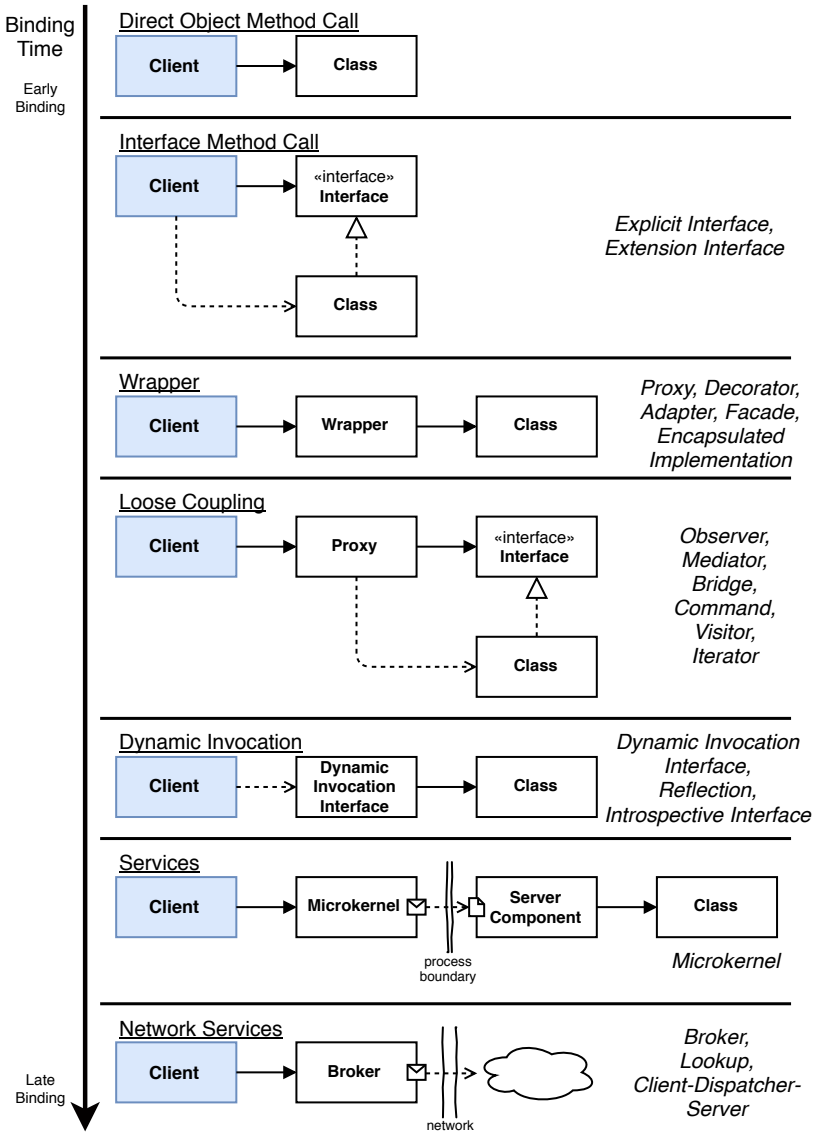

Figure 3.1: Binding Time - Method Calls

Object Method Call This is no pattern in itself, it is a simple direct call of a Method on an Object in the programming language. The client has to know the object type and the method at compile time and calls it at runtime.

Interface Method Call Here the call is made via an interface. The client does not know at compile time which object really is behind the call, just that the object implements the given interface.90

Wrapper The proxy concept takes the indirect dispatch one step further and wraps around the object to call. Still the client calls the interface, but the called object is still a wrapper around the object itself. The proxy could adapt the interface, could add or change functionality of the object, without the client knowing of the Proxy, the Object and what was changed.

Loose Coupling The mediator connects several colleague object via a defined interface and coordinates message between them. This allows for completely loosening compile time dependencies between them. It also allows for later registration of objects.

Dynamic Invocation For completely dynamic invocation some languages allow the calling of methods via a reflection system or a dynamic service provider. This allows for a complete duck typing system like interpreted languages like Python have.

Services Works like a mediator, with the difference that objects are outside of the process boundaries.

Network Services This is also a network based solution, with the aspect that brokers even could be chained behind each other. Therefore a request goes through a chain of brokers before reaching the server shifting the actual work even more out of sight of the client.

3.2 Creational Patterns

The creational scenario is about creating the needed object instance. It is also about resource acquisition and releasing.

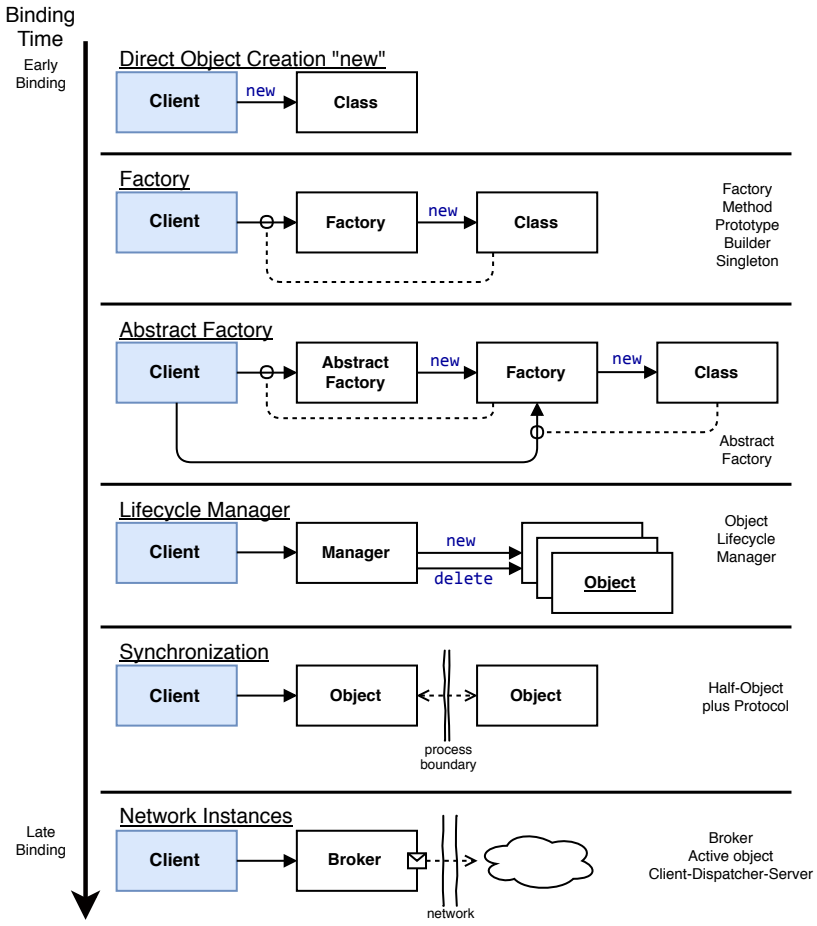

Figure 3.2: Creational Patterns Binding Times

Direct Object Creation "new" On this level the client directly creates its objects with the respective "new" command and direct constructor call. The client depends directly on the object and its constructor, and also has responsibility over the object.

Factory In the factory case, the client itself does not directly know the created object anymore. It just knows the interface and accesses the object through it. Also the client does not directly accesses the constructor, therefore the Factory can control instance count, reuse and the creation process.

Abstract Factory One abstraction level further is the concept of an abstract factory. Here the client requests a factory which itself creates the objects which the client needs. Again the knowledge which object is created lies in the factory itself, but now the client even has the possibility to choose between object groups which all implement the same interface.

Lifecycle Manager The lifecycle manager takes the concept of a factory but extends it even further to the whole life time of the object. The client does not has the responsibility to destroy the object or resource, but the lifecycle manager. This opens up interesting possiblities for caching resources, or pooling threads. Also in the case of counted pointer, it prevents for some implementation errors and makes the code more robust.

Synchronization Through Synchronization object instances can synchronize its data to each other without the client knowing. This allows for a distributed infrastructure where multiple processes share the same instances although they are in their own local memory address space.

Network Instances Instantiation of objects over the network complete strips away local dependencies. The objects are accessed remotely via a protocol and messaging and not called directly anymore. Through some mechanisms like sessions the clients can still be connected to the server side process in order to access the same objects.

3.3 Behavior and Processing Patterns

Patterns for delegating and splitting up work onto multiple objects, threads, or network nodes.

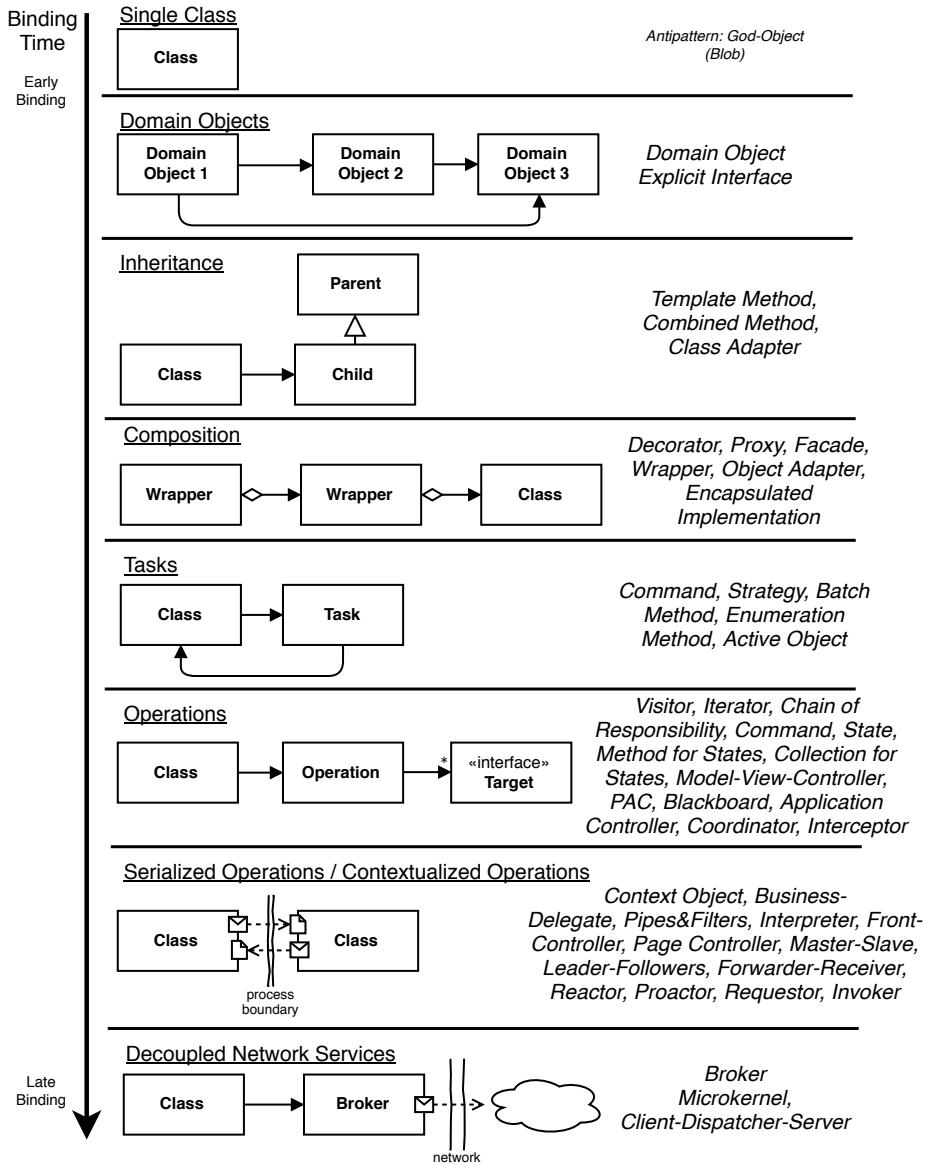

Figure 3.3: Behavior Patterns Binding Times

Single Class From behavior point of view this is a most basic and straight forward way to implement functionality. Just one class which includes every needed functionality. This is the so called *God-Class Antipattern* because it represents the most unflexible, highly coupled and most difficult to maintain approach of defining behavior. Suited only for small objects in very small domains.

Domain Objects To split up functionality into objects which have a specific purpose and responsiblity is a standard object oriented way of implementing functionality. Anyways, this approach has its downsides in that is still highly couples the objects together and does not allow much flexibility.

Inheritance This standard object oriented approach is the process of grouping similar objects together and implement the common functionality only once and inherit it to all the object which need this functionality. While this is a good method of avoiding duplication of functionality, depending on the language capabilities it sometimes is not possible to do it completely (multiple inheritance is often not supported). A huge inheritance-hierarchy makes an design hard to comprehend and therefore this approach could result in difficult to understand and maintain source code.

Composition As stated in GoF: "Favor object composition over class inheritance" (Gamma et al., 1995), a more flexible approach for splitting up behavior is the composition of objects. This approach wraps around objects and combines objects to achieve the needed functionality. Depending on the implementation this can be very flexible and changed at runtime.

Tasks Tasks are single methods or functions which are extracted from the objects to be executed on them. Although this goes in the direction of structured and functional way of programming, it is a good way of separating the behavior of an

object and its data structures. The tasks can be applied to the objects on demand.

Operations Operations are loose coupled tasks which implement against an interface instead of the object. In this way the Operations are applyable to a much more broader range of objects. This adheres to the principle of "Programming to an Interface, not an Implementation" (Gamma et al., 1995).

Serialized Operations / Contextualized Operations The next step of decoupling the Operations is to give them their own context in order to run them independently or completely serialize the data and chain together independent functions. This completely shuts of the object oriented dependencies and depends just on the common data format to operate correctly.

Decoupled Network Services The most decoupled behavior binding one could achieve are network services, where functionality is even not on the same machine anymore but somewhere in the network in a decoupled registered service structure which allows for service lookup and execution. Here the data also has to be serialized, but the client even doesn't know where the data goes and how it is processed. It just uses the service.

3.4 Concurrency and Synchronization Patterns

Patterns for mutual exclusion and concurrency of multiple objects and threads.

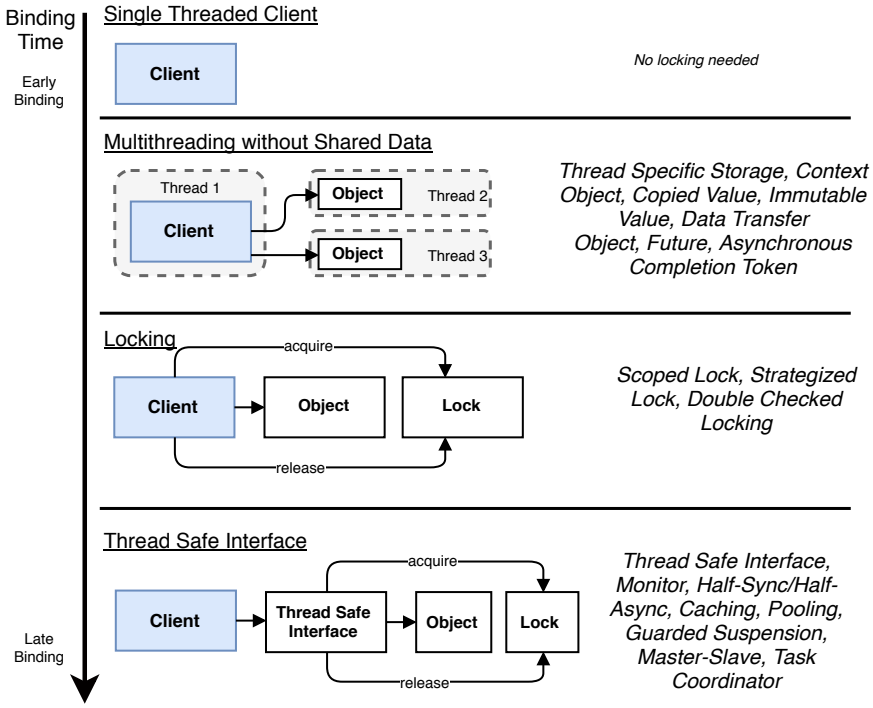

Figure 3.4: Concurrency Patterns Binding Times

Single Threaded Client A single threaded client uses only one thread for all processing and resources and therefore does not need explicit locking mechanism. The locking overhead even would slow down the whole client. So if it is not needed - don't use it.

Multithreading without Shared Data From functional programming comes the principle that functions should not have side effects and that there is not shared data. This can be applied

to multithreading environments to avoid the need of locking mechanisms explicitly. There are patterns to avoid shared data, or to copy and transfer data for every thread in order to avoid simultaneous access to objects.

Locking If sharing data is not avoidable, the classic Locking Mechanisms like Mutex, Semaphores or Condition Variables have to be used to synchronize simultaneous access. There are some design patterns which makes it easy to use these locking mechanisms.

Thread Safe Interface The king class of locking are simply wrappers around the data or resources which support a thread safe interface so that the client does not have to lock the data by itself. Also resources and asynchronous calls can be guarded and coordinated by such wrappers to make it easy to use.

4 Gang of Four Design Patterns

In this chapter 23 design patterns from the book "Design Patterns - Elements of Reusable Object-Oriented Software" (Gamma et al., 1995) are analyzed:

- **Creational Patterns**: Factory Method (45), Abstract Factory (36), Builder (39), Prototype (52), Singleton (54)
- **Structural Patterns**: Adapter (37), Bridge (38), Composite (42), Decorator (43), Facade (44), Proxy (53)
- **Behavioral Patterns**: Interpreter (47), Template Method (57), Chain of Responsibility (40), Command (41), Iterator (48), Mediator (49), Memento (50), Flyweight (46), Observer (51), State (55), Strategy (56), Visitor (58)

4.1 Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

— (Gamma et al., 1995)

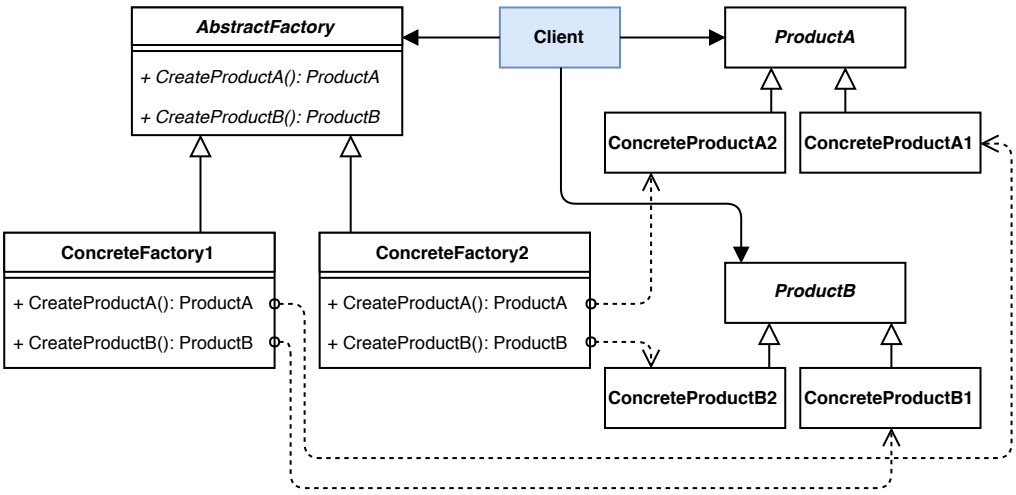

Figure 4.1: Abstract Factory

Participants and Bindings:

- **AbstractFactory**: Specifies the interface for the ConcreteFactory at *compile time*. This has to be done by another Factory which could implement this directly in the source-code at *compile-time* or dependent on a configuration file at *configuration-time* or by flags during execution at *run-time*.
- **AbstractProduct**: Defines the interface for the Product at *compile time*.
- **ConcreteFactory**: Decides which concrete product group to take (*binding time depends on variant*).
- **ConcreteProduct**: Defines the concrete product at *compile time*.
- **Client**: Depends on the AbstractFactory and AbstractProduct at *compile time*. Uses the ConcreteFactory and ConcreteProduct at *runtime*.

4.2 Adapter

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

— (Gamma et al., 1995)

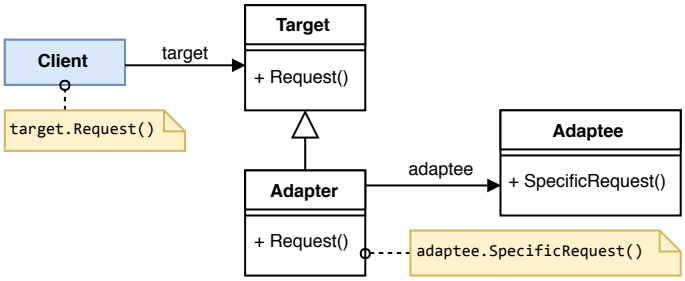

Figure 4.2: Adapter

Participants and Bindings:

- **Client**: Dependent on Target Class at *compile time*. Calls the request method at *runtime*.
- **Target**: Defines interface at *compile time*.
- **Adapter**: Depends on Adaptee at *compile time*. Calls the SpecificRequest at *runtime*.
- **Adaptee**: SpecificRequest gets called at *runtime*.

4.3 Bridge

Decouple an abstraction from its implementation so that the two can vary independently.

— (Gamma et al., 1995)

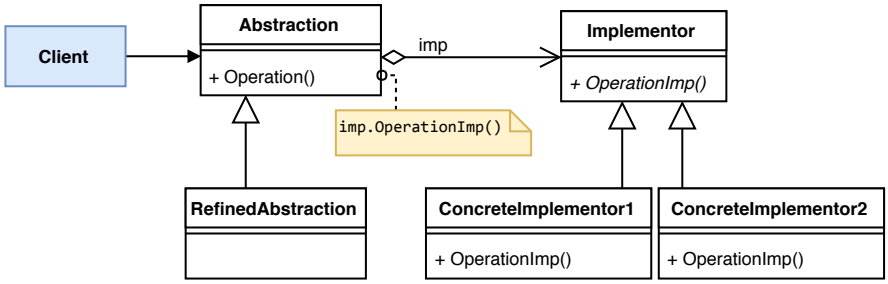

Figure 4.3: Bridge

Participants and Bindings:

- **Client**: Depends on Abstraction-Interface at *compile time*. Calls the Operation-Method at *runtime*.
- **Abstraction**: Depends on Implementor-Interface at *compile time*. Calls the OperationImp-Method at *runtime*.
- **Implementor**: Defines the interface at *compile time*.
- **ConcreteImplementor**: Depends on Implementor interface at *compile-time*. Gets called at *runtime*.

4.4 Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

— (Gamma et al., 1995)

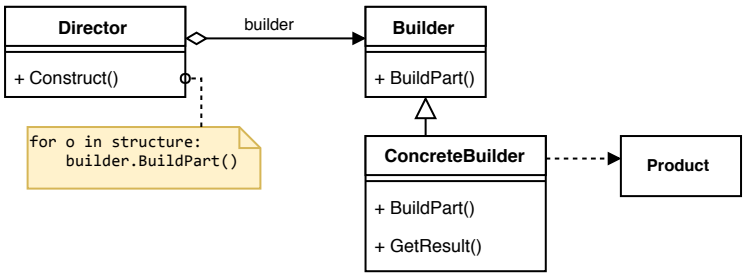

Figure 4.4: Builder

Participants and Bindings:

- **Director**: Depends on Builder Interface at *compile time*. Calls BuildPart() at *runtime*.
- **Builder**: Specifies Interface at *compile time*.
- **ConcreteBuilder**: Depends on Builder Interface and on Product at *compile time*. Creates a Product at *runtime*.
- **Product**: Gets created at *runtime*.

4.5 Chain of Responsibility

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

— (Gamma et al., 1995)

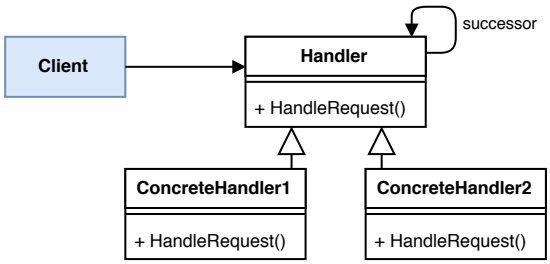

Figure 4.5: Chain Of Responsibility

Participants and Bindings:

- **Client**: Depends on Handler-Interface at *compile time*. Executes a request at *runtime*.
- **Handler**: Specifies an Interface at *compile time*.
- **ConcreteHandler**: Depends on Handler Interface at *compile time*. Handle the request or forwards it to next Handler at *runtime*.

4.6 Command

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

— (Gamma et al., 1995)

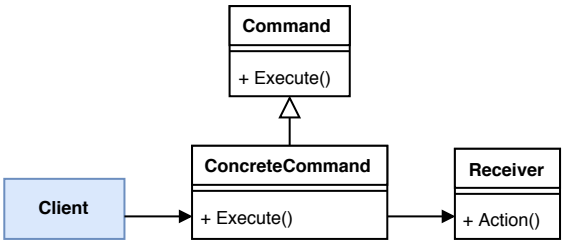

Figure 4.6: Command

Participants and Bindings:

- **Client**: Depends on ConcreteCommand and Command Interface at *compile time*. Creates a ConcreteCommand and sets its receiver at *runtime*.
- **Command**: Specifies the Interface for Commands at *compile time*.
- **ConcreteCommand**: Implements the Command-Interface at *compile time*. Depends on the Receiver at *compile time*. Implements the command at *compile time*. Executes the behavior on the receiver at *runtime*.
- **Receiver**: Implements the actions to execute some behavior at *compile time*. Is created by the client at *runtime*.

4.7 Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

— (Gamma et al., 1995)

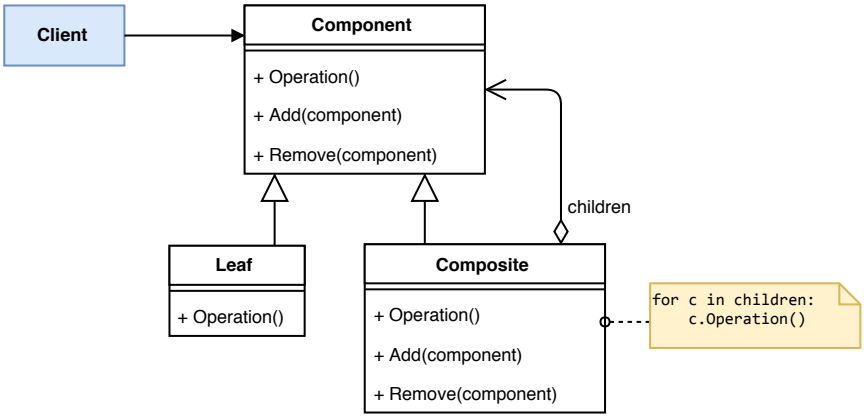

Figure 4.7: Composite

Participants and Bindings:

- **Client**: Depends on the component interface at *compile time*. Adds or Removes children at *runtime*. Calls the operation at *runtime*.
- **Component**: Defines the interface at *compile time*.
- **Leaf**: Depends on the component interface at *compile time*. The operation is called at *runtime*.
- **Composite**: Depends on the component interface at *compile time*. Children are added and removed at *runtime*. The operation is called at *runtime*.

4.8 Decorator

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

— (Gamma et al., 1995)

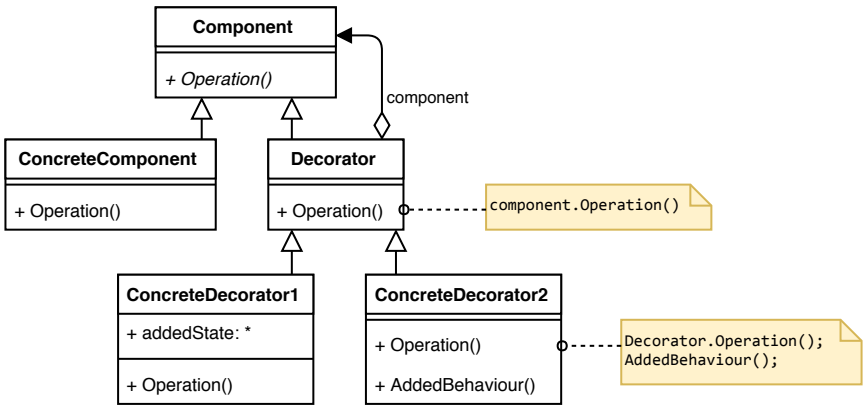

Figure 4.8: Decorator

Participants and Bindings:

- **Component**: Specifies the Interface for the Components at *compile time*.
- **ConcreteComponent**: Depends on Component Interface at *compile time*. Gets decorated at *runtime*.
- **Decorator**: Depends on the Component-Interface and specifies the Interface for the Decorators at *compile time*. Calls the component's operation at *runtime*.
- **ConcreteDecorator**: Depends on the Decorator Base Class at *compile time*. Calls the base-class operation at *runtime* and adds own behavior.

4.9 Facade

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

— (Gamma et al., 1995)

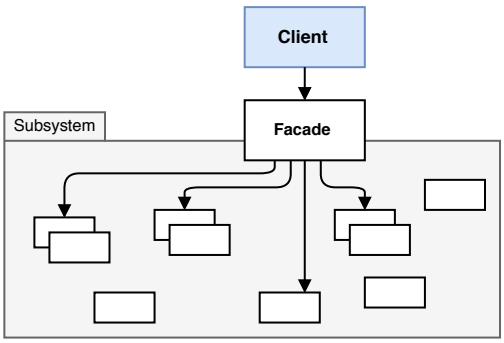

Figure 4.9: Facade

Participants and Bindings:

- **Client**: Depends on the Facade at compile time. Uses the Facade's Methods at runtime.
- **Facade**: Depends on the subsystem to call some methods at *compile time*. Specifies an Interface for the caller at *compile time*. Gets called and reroutes the calls to the specific implementor objects at *runtime*.
- **Subsystem**: Provides classes and methods for the facade at *compile time*. Gets used at *runtime*.

4.10 Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

— (Gamma et al., 1995)

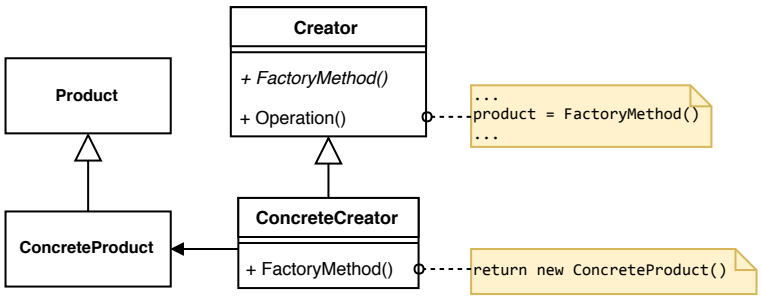

Figure 4.10: Factory Method

Participants and Bindings:

- **Creator**: Defines interface for the Factory-Methods at *compile time*. Calls the actual implementation at *runtime*.
- **ConcreteCreator**: Depends on the Creator and the ConcreteProduct at *compile time*. Implements the Creator Interface at *compile time*. Creates a ConcreteProduct at *runtime*.
- **Product**: Defines an interface for the products at *compile time*.
- **ConcreteProduct**: Depends on the Product and implements the Product-Interface at *compile time*.

4.11 Flyweight

Use sharing to support large numbers of fine-grained objects efficiently.
— (Gamma et al., 1995)

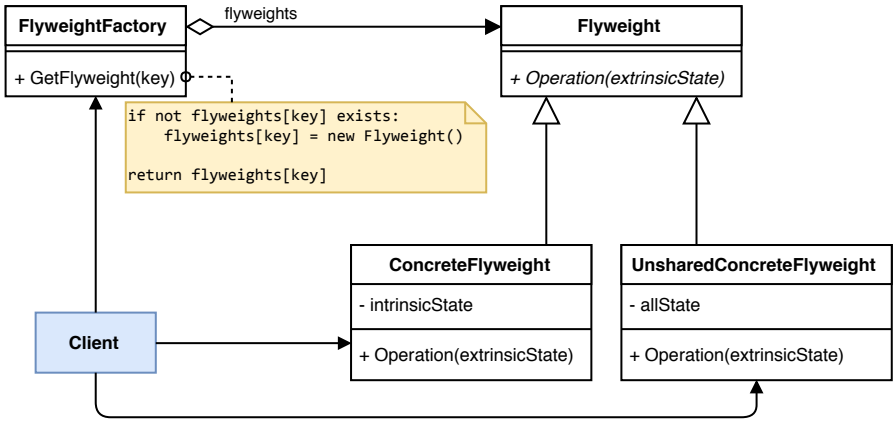

Figure 4.11: Flyweight

Participants and Bindings:

- **Client**: Depends on the FlyweightFactory and Flyweight Interface at *compile time*. Calls the ConcreteFlyweights-Operation at *runtime*.
- **FlyweightFactory**: Depends on the Flyweight-Interface and ConcreteFlyweight and UnsharedConcreteFlyweight at *compile time*. Creates and stores Flyweight Instances at *runtime*.
- **Flyweight**: Specifies the Interface for Flyweights at *compile time*.
- **ConcreteFlyweight**: Implements the Flyweight-Interface. Gets called by the client at *runtime*.
- **UnsharedConcreteFlyweight**: Implements the Flyweight-Interface. Gets called by the client at *runtime*.

4.12 Interpreter

Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
— (Gamma et al., 1995)

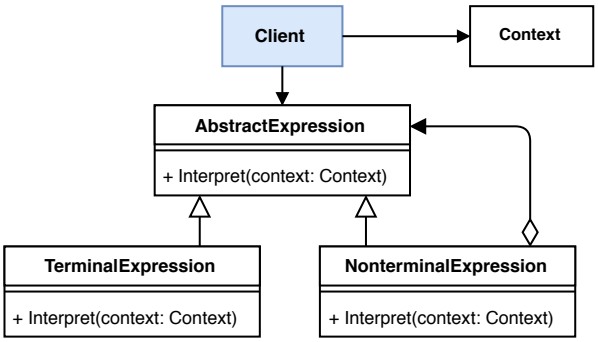

Figure 4.12: Interpreter

Participants and Bindings:

- **Client**: Dependend on the Context and the AbstractExpression interface at *compile time*. Calls the Interpret-Function with a instantiated context at *runtime*.
- **AbstractExpression**: Specifies an interface for the Expressions at *compile time*.
- **TerminalExpression**: Implements the interpreter interface at *compile time*. Is called at *runtime*.
- **NonterminalExpression**: Implements the interpreter interface at *compile time*. Is called at *runtime*.
- **Context**: Is created and used by the client at *runtime*.

4.13 Iterator

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

— (Gamma et al., 1995)

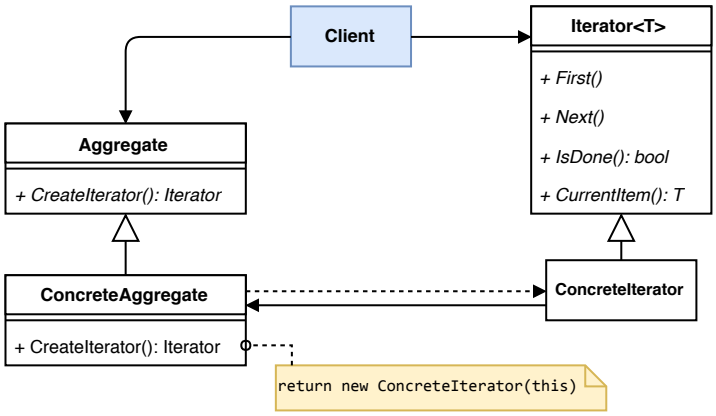

Figure 4.13: Iterator

Participants and Bindings:

- **Iterator**: Specifies the Interface for the iterators at *compile time*.
- **Aggregate**: Specifies the interface to create an iterator at *compile time*.
- **ConcreteAggregate**: Implements the Aggregate-Interface at *compile time* to create an Iterator-Instance. Creates a specific ConcreteIterator instance at *runtime*.
- **ConcreteIterator**: Implements the Iterator-Interface for a specific object type at *compile time*. Is used by the client at *runtime* to iterate through the objects.
- **Client**: Depends on the Iterator and the Aggregate at *compile time*. Uses them at *runtime*.

4.14 Mediator

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

— (Gamma et al., 1995)

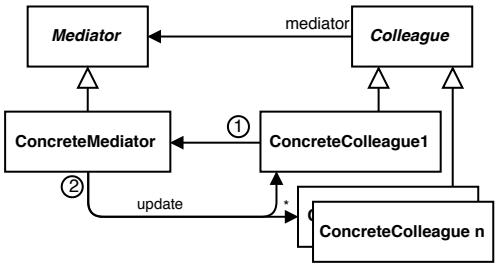

Figure 4.14: Mediator

Participants and Bindings:

- **Mediator**: Specifies the interface for the Mediator at *compile time*.
- **Colleague**: Specifies the interface for the Colleagues at *compile time*.
- **ConcreteMediator**: Implements the Mediator Interface at compile time. Depends on the Colleague-Interface at compile time. Spreads out requests from a Colleague to the other Colleagues at runtime. Depending on the detail of implementation also depends on the ConcreteColleagues at compile time when calling their non-inherited methods at runtime.
- **ConcreteColleague**: Implements the Colleague Interface at compile time. Dependent on the Mediator Interface at compile time. Calls the ConcreteMediator at runtime to spread out the updates to the other Colleagues.

4.15 Memento

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

— (Gamma et al., 1995)

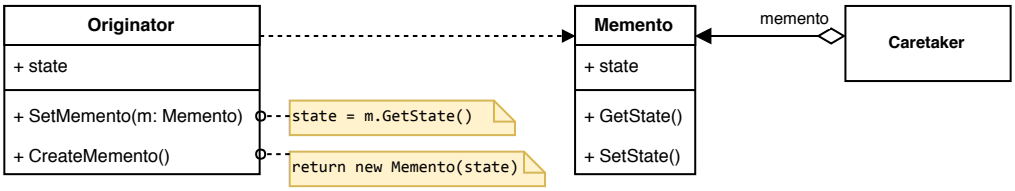

Figure 4.15: Memento

Participants and Bindings:

- **Originator**: An object defined at *compile time*, which manifests its state in form of a Memento at *runtime*.
- **Memento**: Data object which represents the Originator's state and can be used by the Originator at *runtime* to save or restore it's inner state. Defined at *compile time*, created and used at *runtime*.
- **Caretaker**: Depends on the Originator-Interface at *compile time* to Get or Set Memento Objects to it at *runtime*.

4.16 Observer

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
— (Gamma et al., 1995)

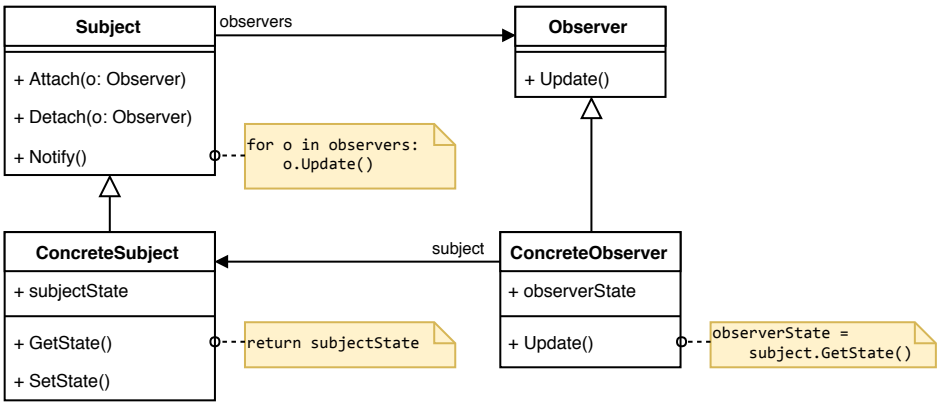

Figure 4.16: Observer

Participants and Bindings:

- **Subject**: Specifies an interface for observable objects and is dependend on the observer-interface at *compile time*. Holds a list of observers at *runtime*.
- **ConcreteSubject**: Implements the Subject Interface at *compile time*. Notifies the observers and manages its state at *runtime*.
- **Observer**: Specifies an interface at *compile time* for notifying observers.
- **ConcreteObserver**: Implements the Observer-Interface and depends on the ConcreteSubject at compile time. On Update it gets the state of the concrete subject at *runtime*.

4.17 Prototype

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

— (Gamma et al., 1995)

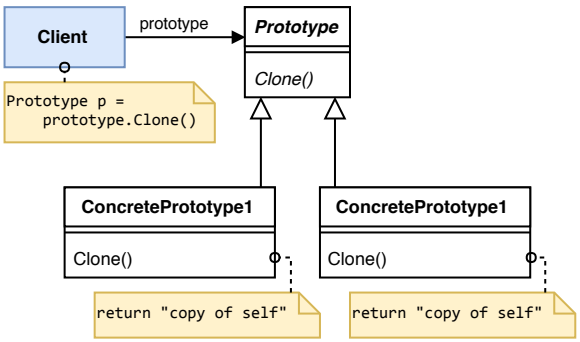

Figure 4.17: Prototype

Participants and Bindings:

- **Client**: Depends on the Prototype Interface at *compile time*. Calls the Clone()-Method at *runtime*.
- **Prototype**: Specifies the Interface at *compile time*.
- **ConcretePrototype**: Implement the Prototype interface at *compile time*. Gets called at *runtime*.

4.18 Proxy

Provide a surrogate or placeholder for another object to control access to it.
— (Gamma et al., 1995)

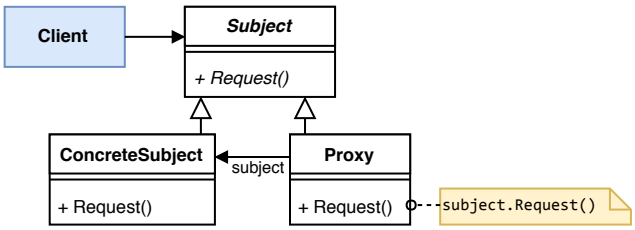

Figure 4.18: Proxy

Participants and Bindings:

- **Client**: Depends on the Subject-Interface at *compile time*. Calls the actual Request at *runtime*.
- **Subject**: Specifies the Interface for the Subjects at *compile time*.
- **ConcreteSubject**: Implement the Subject-Interface at *compile time*. Gets called by the Proxy at *runtime*.
- **Proxy**: Implements the Subject-Interface at *compile time*. Depends on the ConcreteSubject at *compile time*. Reroutes the requests to ConcreteSubject at *runtime*.

4.19 Singleton

Ensure a class only has one instance, and provide a global point of access to it.
— (Gamma et al., 1995)

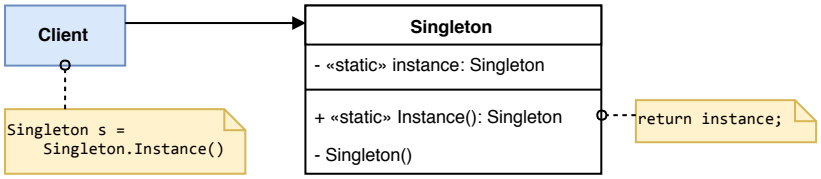

Figure 4.19: Singleton

Participants and Bindings:

- **Client**: Depends on the Singleton at *compile time*. Calls the static instance of the singleton at *runtime*.
- **Singleton**: Manages the single instance of the Singleton at *runtime*.

4.20 State

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

— (Gamma et al., 1995)

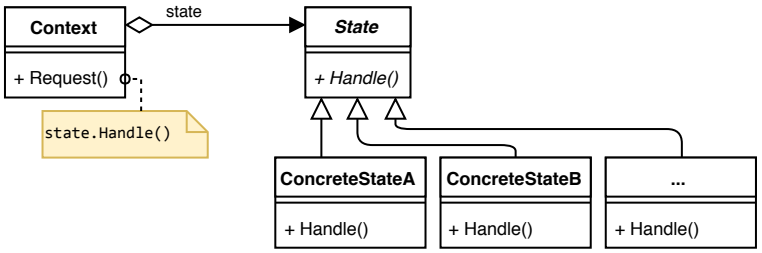

Figure 4.20: State

Participants and Bindings:

- **Context**: Depends on the State Interface at *compile time*. Manages and uses the states at *runtime*.
- **State**: Specifies the interface for all States at *compile time*.
- **ConcreteState**: Implements the State Interface at *compile time*. Handles the Requests at *runtime*

4.21 Strategy

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

— (Gamma et al., 1995)

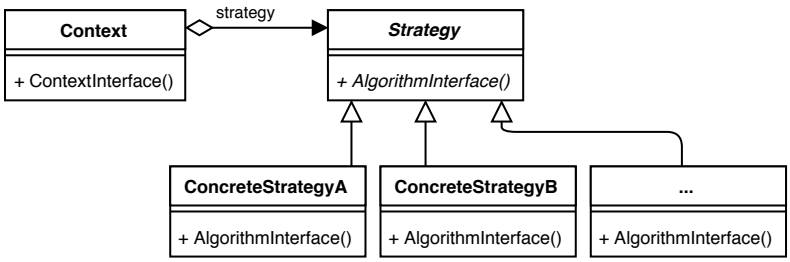

Figure 4.21: Strategy

Participants and Bindings:

- **Strategy**: Specifies the Interface for the Strategy at *compile time*.
- **ConcreteStrategy**: Implements the Strategy interface at *compile time*. Gets called by the Context at *runtime*
- **Context**: Depends on the Strategy Interface at *compile time*. Calls the Methods at *runtime*.

4.22 Template Method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

— (Gamma et al., 1995)

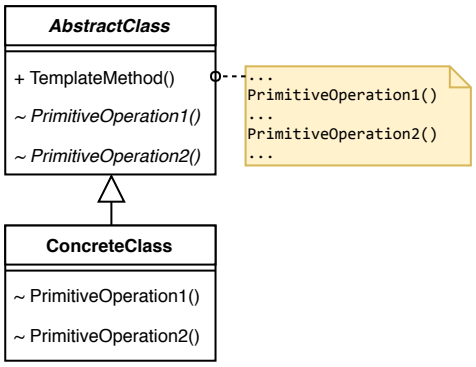

Figure 4.22: Template Method

Participants and Bindings:

- **AbstractClass**: Defines the Interface for the Templates at *compile time*. Calls the overloaded methods at *runtime*.
- **ConcreteClass**: Implements the AbstractClass Methods at *compile time*. Gets called at *runtime*.

4.23 Visitor

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

— (Gamma et al., 1995)

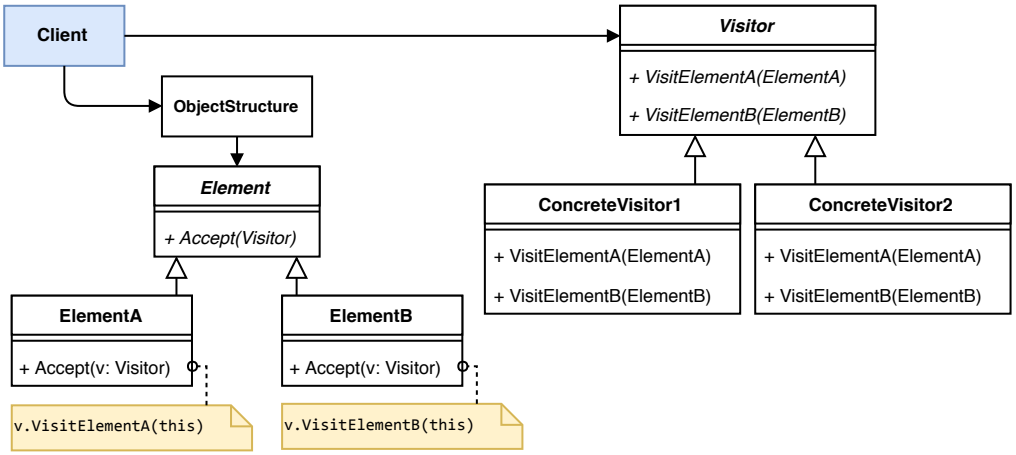

Figure 4.23: Visitor

Participants and Bindings:

- **Visitor**: Specifies the interface for the visitors at *compile time*.
- **ConcreteVisitor**: Implements the Visitor interface at *compile time*. gets called at *runtime*.
- **ObjectStructure**: An arbitrary collection containing objects defined at *compile time* and used at *runtime*.
- **Element**: Specifies the interface at *compile time* for elements which are visitable by the visitors.
- **ConcreteElement**: Implements the Element Interface at *compile time* and calls a concrete method of the visitor interface at *runtime*.
- **Client**: Depends on Visitor and Element Interface and on the ObjectStructure at *compile time*. Creates a concrete visitor and applies it to the elements at *runtime*.

5 POSA 1 Design Patterns

In this chapter 15 design patterns from the book "Pattern-Oriented Software Architecture Volume 1: A system of patterns" (Buschmann, Meunier, et al., 1996) are analyzed:

- **Architectural Patterns**:
  - *From Mud to Structure*: Layers (60), Pipes and Filters (61), Blackboard (62)
  - *Distributed Systems*: Broker (63)
  - *Interactive Systems*: Model-View-Controller (65), Presentation-Abstraction-Control (66)
  - *Adaptable Systems*: Microkernel (68), Reflection (69)
- **Design Patterns**:
  - *Structural Decomposition*: Whole-Part → Composite (42)
  - *Organization of Work*: Master-Slave (70)
  - *Access Control*: Proxy (53)
  - *Management*: Command Processor (71), View Handler (72)
  - *Communication*: Forwarder-Receiver (73), Client-Dispatcher-Server (74), Publisher-Subscriber → Observer (51)
- **Idioms**: Counted Pointer (75)

5.1 Layers

Define one or more layers for the software under development, with each layer having a distinct and specific responsibility.

— (Buschmann, Meunier, et al., 1996)

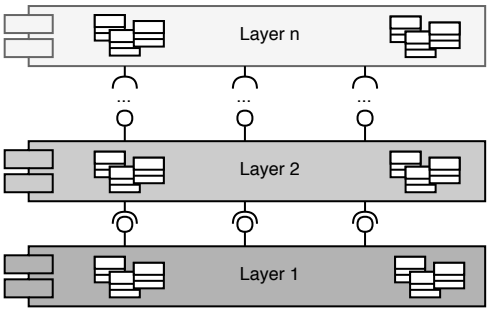

Figure 5.1: Layers

Participants and Bindings: The decision how many layers and which responsibilities the layers have, is bound at *architecture time*. Which objects and the internal structure of the layers is bound at *design time*. Every layer depends on the underlying interfaces and defines interfaces for the next layer at *compile time*. Internally every layer also depends on its contained objects at *compile time*. *At runtime* the layers are actually coupled together by some creation mechanism.

5.2 Pipes and Filters

Divide the application's task into several self-contained data processing steps and connect these steps to a data processing pipeline via intermediate data buffers.

— (Buschmann, Meunier, et al., 1996)

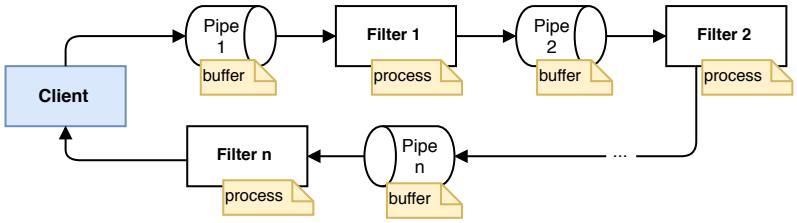

Figure 5.2: Pipes and Filters

Participants and Bindings:

- **Client**: Dependent on the common data format at *runtime*.
- **Pipe**: Dependent on the common data format at *runtime*.
- **Filter**: Dependent on the common data format at *runtime*.

5.3 Blackboard

Use heuristic computation to resolve the task via multiple smaller components with deterministic solution algorithms that gradually improve an intermediate solution hypothesis.

— (Buschmann, Meunier, et al., 1996)

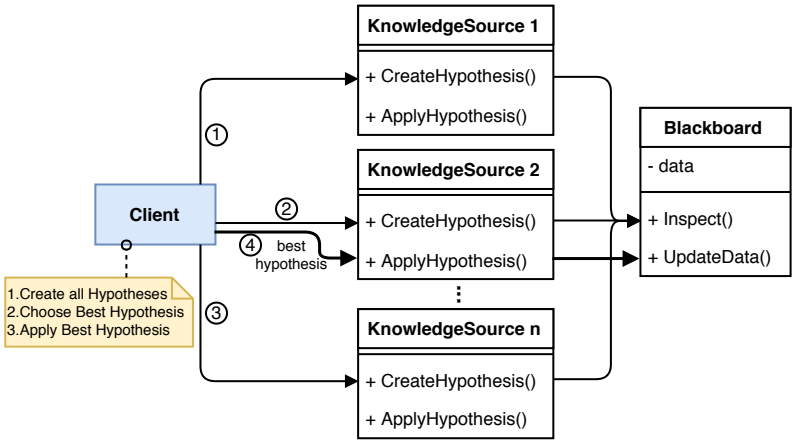

Figure 5.3: Blackboard

Participants and Bindings:

- **Client**: Depends on the KnowledgeSources at *compile time*. Calls the KnowledgeSources at *runtime*, chooses and applies the best solution at *runtime*.
- **KnowledgeSource**: Depends on the Blackboard-Interface and Data structure at *compile time*. Uses the Blackboard for inspection at *runtime*.
- **Blackboard**: Specifies the Data-Structure and Blackboard-Interface at *compile time*. Gets inspected and updated by the KnowledgeSources at *runtime*.

5.4 Broker

Use a federation of brokers to separate and encapsulate the details of the communication infrastructure in a distributed system from its application functionality. Define a component-based programming model so that clients can invoke methods on remote services as if they were local.

— (Buschmann, Meunier, et al., 1996)

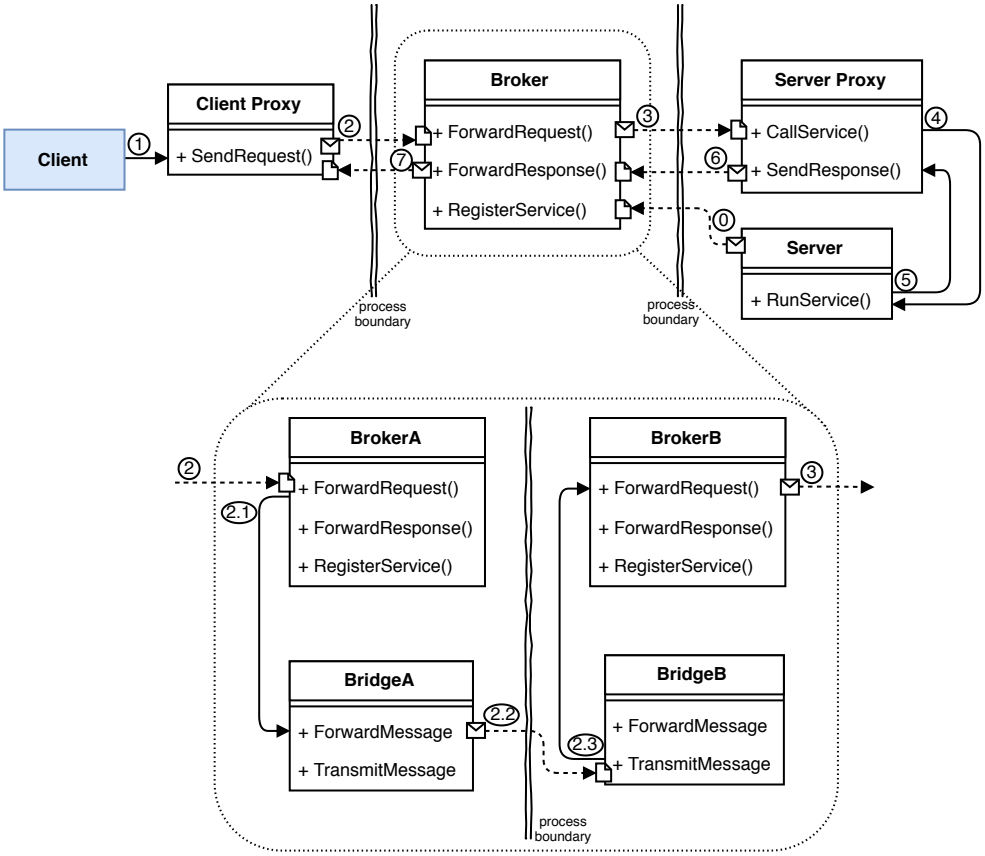

Figure 5.4: Broker

Participants and Bindings:

- **Client**: Depends on the Client-Proxy at *compile time*. Uses the Client-Proxy at *runtime*.

- **Client-Proxy**: Depends only on the common data serialization format between Client-Proxy and the Broker at *runtime*.
- **Broker**: Depends on the Bridge-Interfaces at *compile time*. Depends on the common data serialization format between Client-Proxy and Broker at *runtime*. Uses the Bridges at *runtime* to communicate to other Brokers. Uses the Server-Proxy at *runtime*.
- **Bridge**: Converts the request data serialization format of one Broker to the format of another Broker at *runtime* (ensures compatibility). Is used by the Broker and uses another Bridge at *runtime*.
- **Server-Proxy**: Depends on the Server at *compile time*. Convert the common data serialization format of Broker to Server-Proxy at *runtime* into a form which is appropriate for the Server to use.
- **Server**: Registers at the Broker at *runtime*. Gets called by the Server-Proxy at *runtime*.

5.5 Model-View-Controller

Divide the interactive application into three decoupled parts: processing, input, and output. Ensure the consistency of the three parts with the help of a change propagation mechanism.

— (Buschmann, Meunier, et al., 1996)

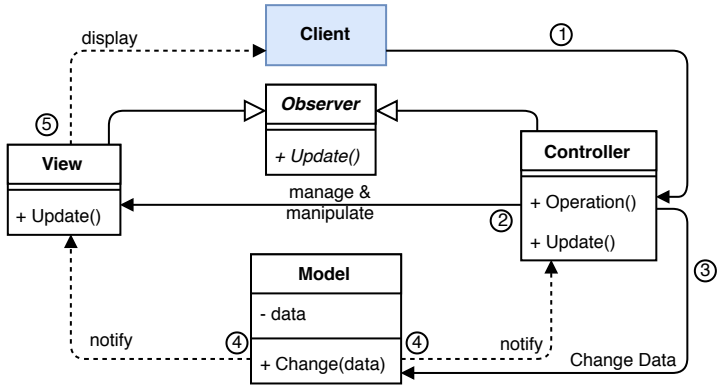

Figure 5.5: Model-View-Controller

Participants and Bindings:

- **Client**: Dependent on the Controller Interface at *compile time*. Initiates some action on the Controller at *runtime*.
- **Controller**: Implements the Observer-Interface at *compile time*. Depends on the Model Interface and the View-Interface at *compile time*. Updates the model at *runtime*. Manages and manipulates the Views at *runtime*.
- **Model**: Depends upon the Observer-Interface at *compile time*. Gets updated at *runtime* and also notifies its observers at *runtime*. Manages its Observers at *runtime*.
- **View**: Implements the Observer-Interface at *compile time*. Gets notified for changes on the underlying model at *runtime*. Is managed and manipulated by the Controller at *runtime*.
- **Observer**: Specifies the Interface for the Observers at *compile time*.

5.6 Presentation-Abstraction-Control

Structure the interactive application as a hierarchy of decoupled agents: one top-level agent, several intermediate-level agents, and many bottom-level agents. Each agent is responsible for a specific functionality of the application and provides a specialized user interface for it.

— (Buschmann, Meunier, et al., 1996)

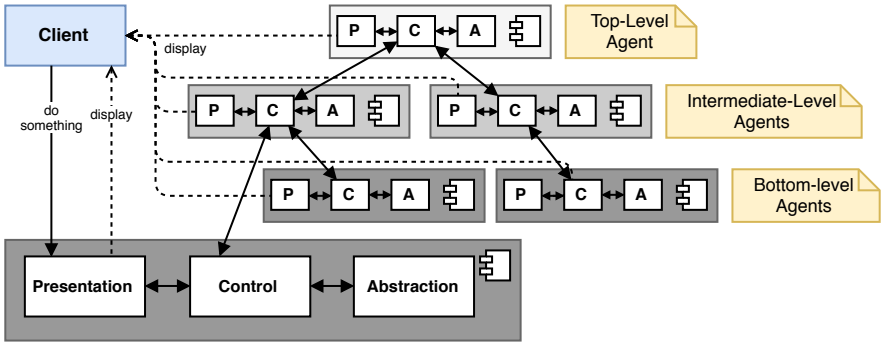

Figure 5.6: Presentation-Abstraction-Control

Participants and Bindings:

- **Client**: Depends on the Interface to the Presentation-Parts of the PAC-Agents at *compile time*. Uses the Presentation-Part to do something (invoke a signal, method, or event) at *runtime*. Gets updated by de Presentation-Parts at *runtime*.
- **Presentation**: Depends on the Control-Interface and specifies the Interface and display-logic to the Client at *compile time*. Receives calls from the Client, gets changed by the Control-Part and displays the changes back at the Client at *runtime*.
- **Abstraction**: Depends on the Controller-part of the PAC-Agent at *compile time*. Manages the data abstraction and manifestation in a PAC-Agent at *runtime*.
- **Control**: Depends on the Presentation and Abstraction interfaces in a PAC-Agent at *compile time*. Also depends on the other Control-Parts of the other PAC-Agents at *compile time*. Manages the control and data flow between Presenta-

tion and Abstraction part and also other Control-parts at *runtime*.

- **Bottom-Level Agents**: Manage the lowest-level display items like e.g. textboxes, labels, buttons at *runtime*.
- **Intermediate-Level Agents**: Manage intermediate container items like e.g. Panels, Groupboxes, Layout-Managers at *runtime*. Delegates detailed drawing tasks to the bottom-level agents at *runtime*.
- **Top-Level Agent**: Manages the top level states in an application like e.g. the application window or other top level modal containers at *runtime*. Delegates layout and drawing tasks to the intermediate-level agents at *runtime*.

5.7 Microkernel

Applies to software systems that must be able to adapt to changing system requirements It separates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration

— (Buschmann, Meunier, et al., 1996)

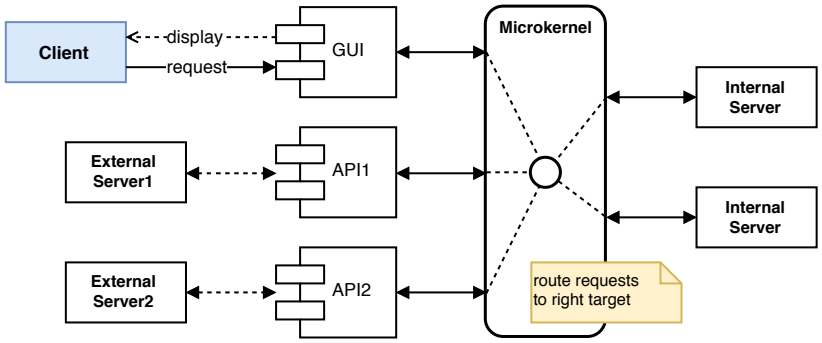

Figure 5.7: Microkernel

Participants and Bindings:

- **Client**: Depends on the GUI Interface at *compile time*. Requests the GUI at runtime and gets Display back at *runtime*.
- **GUI/API**: Depend on the Microkernel Interface and the external Access Protocoll (e.g. via a Webservice, or GUI Events) at *compile time*. Communicate externally at runtime and route Events to the Microkernel at *runtime*. Get updated by the Microkernel at *runtime*.
- **Microkernel**: Depends on the Internal Server Interfaces at *compile time*. Manages the registered internal Servers at *runtime*. Route the Events to the correct server at *runtime*.
- **Internal Server**: Depends on the Microkernel Interface at *compile time*. Registers at the Microkernel at *runtime*. Gets called by the Microkernel at *runtime*.
- **External Server**: Depend on the API at *compile time*. Call the API's functions at *runtime*.

5.8 Reflection

Objectify information about properties and aspects of the application's structure into metaobjects. Separate the metaobjects from the core application logic via a two-layer architecture: a meta level and a base level.

— (Buschmann, Meunier, et al., 1996)

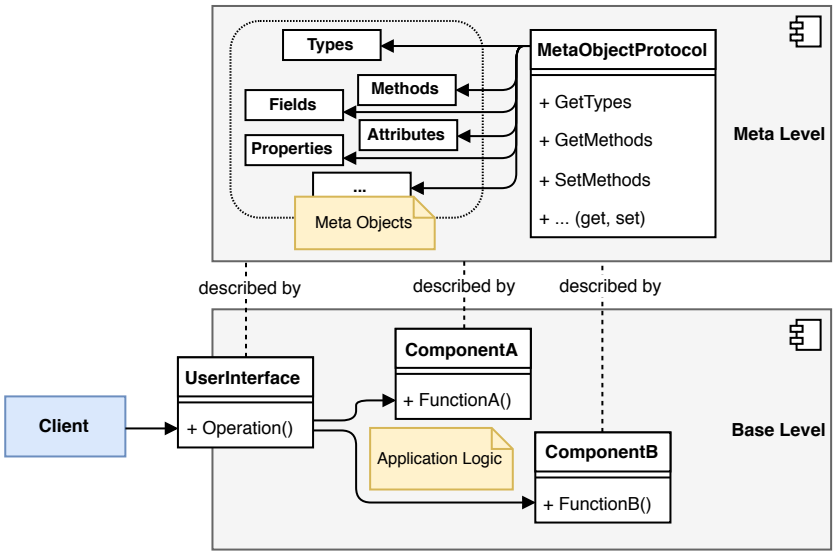

Figure 5.8: Reflection

Participants and Bindings:

- **Client**: Depends on the Base-Level Objects at *compile time*. Uses the User-Interface at *runtime*.
- **Application Logic**: Actual implementation of the application logic, components and objects at *compile time* which are called at *runtime*.
- **MetaObjectProtocol**: Depends on the Meta Objects at *compile time*. Depends on the Base Level Objects at *compile time* (static) at *startup time* (once) or *runtime* (dynamic late binding). Creates and manages the MetaObjects at *runtime*.
- **MetaObjects**: Depend on each other at *compile time*. Are managed by the MetaObjectProtocol at *runtime*.

5.9 Master-Slave

Meet the performance, fault-tolerance, or accuracy requirements of the component via a "divide and conquer" strategy. Splits its services into independent subtasks that can be executed in parallel, and combine the partial results returned by these subtasks to provide the service's final results.

— (Buschmann, Meunier, et al., 1996)

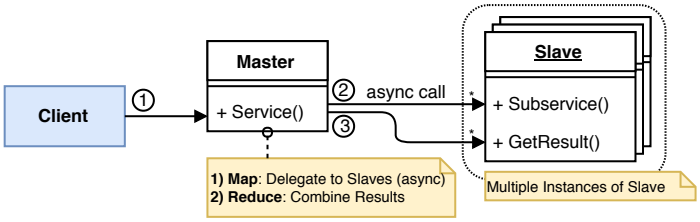

Figure 5.9: Master-Slave

Participants and Bindings:

- **Client**: Dependent on the Master at *compile time*. Calls the Service-Method at *runtime*.
- **Master**: Dependent on the Slave-Interface at *compile time*. Asynchronously calls the slaves at *runtime* and combines their results at *runtime*.
- **Slave**: Gets called by the Master at *runtime*.

5.10 Command Processor

Introduce a command processor to execute requests to the application. The command processor acts on behalf of the clients and within the constraints of the application.

— (Buschmann, Meunier, et al., 1996)

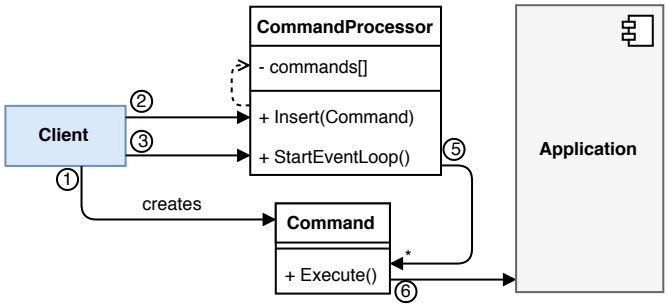

Figure 5.10: Command Processor

Participants and Bindings:

- **Client**: Depends on the Commands and the CommandProcessor at *compile time*. Starts the CommandProcessor at *runtime*. Creates Commands and handles them over to the CommandProcessor at *runtime*.
- **CommandProcessor**: Depends upon the Command-Interface at *compile time*. Manages the commands at *runtime* (thread-safe). Executes the commands at *runtime*.
- **Command**: Depends on the Application-Interface at *compile time*. Is created by the Client at *runtime*. Calls the implementation at *runtime*.
- **Application**: Gets called by the commands at *runtime*.

5.11 View Handler

Helps to manage all views that a software system provides A view handler component allows clients to open, manipulate and dispose of views. It also coordinates dependencies between views and organizes their update.

— (Buschmann, Meunier, et al., 1996)

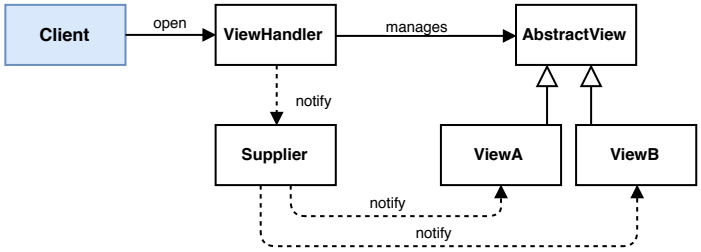

Figure 5.11: View Handler

Participants and Bindings:

- **Client**: Dependent on the ViewHandler at *compile time*. Calls the ViewHandler at *runtime*.
- **ViewHandler**: Dependent on the AbstractView Interface at *compile time*. Manages the Views at *runtime*. Notifies the registered Suppliers at *runtime*.
- **AbstractView**: Specifies the Interface for views at *compile time*.
- **Supplier**: Implements some kind of notification-mechanism at *compile-time*. Delegates changes to the registered View-Objects at *runtime*. Registers at the ViewHandler to get notified for changes at *runtime*.
- **View**: Implements the AbstractView-Interface at *compile time*. Gets created and managed by the ViewHandler at *runtime*. Gets notified for changes by the supplier at *runtime*.

5.12 Forwarder-Receiver

Provides transparent inter-process communication for software systems with a peer-to-peer interaction model. It introduces forwarders and receivers to decouple peers from the underlying communication mechanisms.

— (Buschmann, Meunier, et al., 1996)

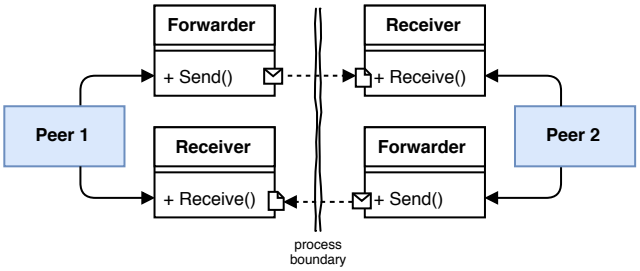

Figure 5.12: Forwarder-Receiver

Participants and Bindings:

- **Peers**: Depends on the Forwarder- and Receiver-Object at *compile time*. Call Send and Receive methods at *runtime*.
- **Forwarder**: Depends on the common data exchange format for marshalling at *runtime*.
- **Receiver**: Depends on the common data exchange format for unmarshalling at *runtime*.

5.13 Client-Dispatcher-Server

Introduces an intermediate layer between clients and servers, the dispatcher component. It provides location transparency by means of a name service, and hides the details of the establishment of the communication connection between clients and servers.

— (Buschmann, Meunier, et al., 1996)

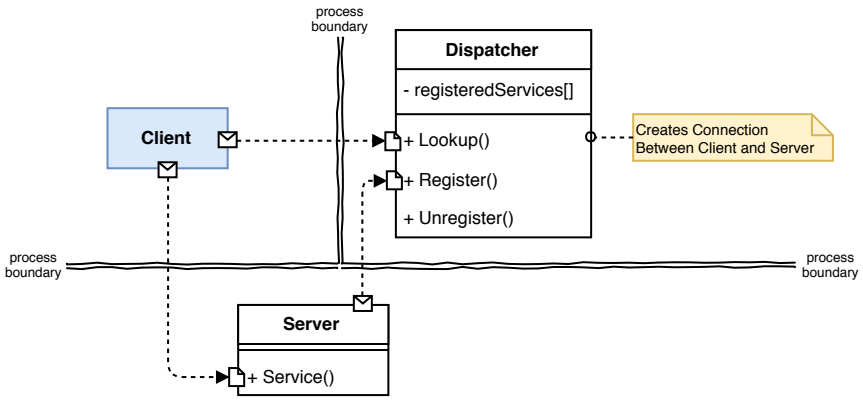

Figure 5.13: Client-Dispatcher-Server

Participants and Bindings:

- **Client**: Dependent on the common data format to the Dispatcher for looking up services at *runtime*. Looks up the a service and calls it at *runtime*. Dependent on the common data format between the Client and the Server at *runtime*.
- **Dispatcher**: Manages the registered Services at *runtime*. Gets requested by the Client and the Server at *runtime*.
- **Server**: Dependent on the common data format for registering the Services at the Dispatcher at *runtime*.

5.14 Counted Pointer

Makes memory management of dynamically-allocated shared objects in C++ easier. It introduces a reference counter to a body class that is updated by handle objects. Clients access body class objects only through handles via the overloaded operator "->".

— (Buschmann, Meunier, et al., 1996)

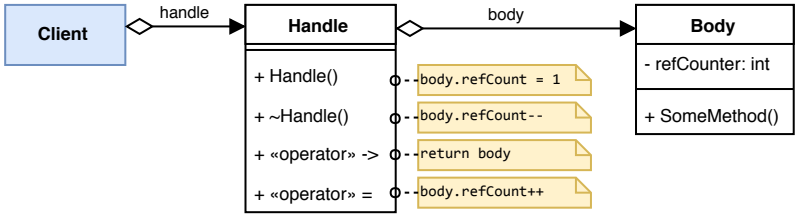

Figure 5.14: Counted Pointer

Participants and Bindings:

- **Client**: Dependent on Handle-Interface and Body-Object at *compile time*. Creates Handle and Body at *runtime* and uses it at *runtime*.
- **Handle**: Dependent on the Body-Class at *compile time*. Gets created at *runtime* and also creates a Body-Object at *runtime*. Manages the reference count of the corresponding Body-Object at *runtime*.
- **Body**: Gets created by the handle at *runtime*. Gets destroyed when last reference gets destroyed at *runtime*.

5.15 Duplicate Patterns

In POSA1 some patterns are duplicates from the GoF-Book. Here is a listing of those patterns:

- **Publisher-Subscriber** → see GOF Observer (51)
- **Whole-Part** → see GOF Composite (42)
- **Proxy** → see GOF Proxy (53)

6 POSA 2 Design Patterns

In this chapter 17 design patterns from the book "Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects" (Schmidt et al., 2000) are analyzed:

- **Service Access and Configuration Patterns**: Wrapper Facade (78), Component Configurator (79), Interceptor (80), Extension Interface (81)
- **Event Handling Patterns**: Reactor (82), Proactor (83), Asynchronous Completion Token (84), Acceptor-Connector (85)
- **Synchronization Patterns**: Scoped Locking (86), Strategized Locking (87), Thread-Safe Interface (88), Double-Checked Locking (89)
- **Concurrency Patterns**: Active Object (90), Monitor Object (91), Half-Sync/Half-Async (92), Leader/Followers (93), Thread-Specific Storage (94)

6.1 Wrapper Facade

Avoid accessing low-level function-based APIs directly. Instead, wrap each related group of functions and data within such an API in a separate, cohesive wrapper facade class.

— (Schmidt et al., 2000)

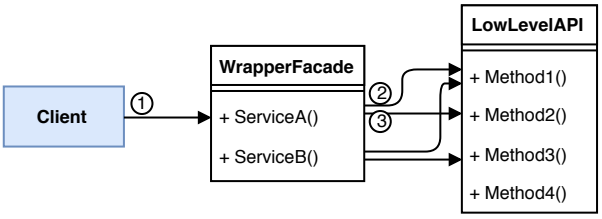

Figure 6.1: Wrapper Facade

Participants and Bindings:

- **Client**: Depends on the WrapperFacade at *compile time*. Calls the Service-methods at *runtime*.
- **Wrapper Facade**: Depends on the LowLevelAPI at *compile time*. Calls its methods at *runtime*.
- **Low Level API**: Specifies an interface and implements the functionality at *compile time*. Gets called by the WrapperFacade at *runtime*.

6.2 Component Configurator

Decouple component interfaces from their implementations and provide a mechanism to (re)configure components in an application dynamically without having to shut down and restart it.

— (Schmidt et al., 2000)

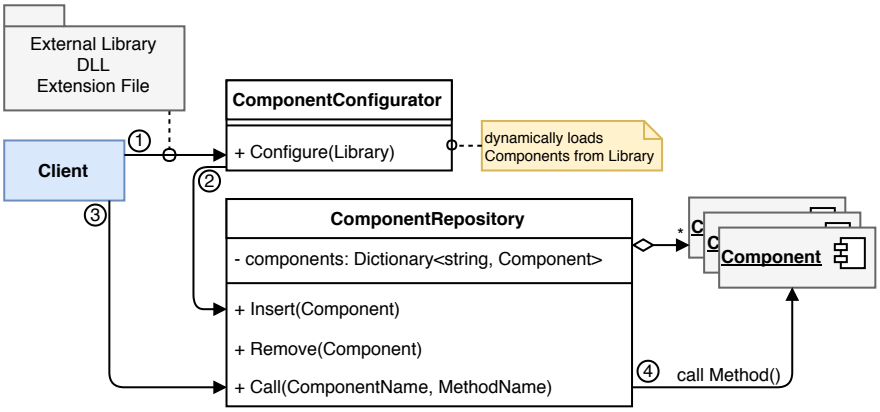

Figure 6.2: Component Configurator

Participants and Bindings:

- **Client**: Depends on the Component Configurator Interface and the Component Repository Interface at *compile time*. Calls the Configure method on the Configurator with an external Library to load at *runtime*. Uses the Component Repository to call Methods on the dynamically loaded Components at *runtime*.
- **Library**: An external component library which is loaded at *runtime*.
- **Component Configurator**: Depends on the dynamic loading facilities of the programming language at *compile time* to load an external library into the Component Registry at *runtime*. Depends on the Component Registry Interface at *compile time*. Loads an external library and stores the Components and Services in the Component Registry at *runtime*.

- **Component Repository**: Manages the loaded Components and Services at *runtime*. Calls the Methods on the Components at *runtime*.
- **Component**: Is defined in the external library (within its *own lifecycle*, but at runtime when the client wants to load the library) and loaded dynamically at *runtime*.

6.3 Interceptor

Allow users to tailor a software framework by registering out-of-band service extensions via predefined callback interfaces, known as "interceptors", then let the framework trigger these extensions automatically when specific events occur.

— (Schmidt et al., 2000)

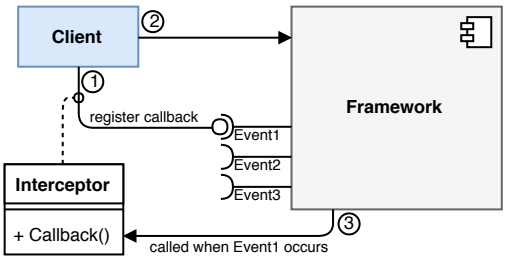

Figure 6.3: Interceptor

Participants and Bindings:

- **Client**: Depends on the Framework at *compile time*. Uses it at *runtime*.
- **Framework**: Specifies Events to be intercepted by Interceptors at *compile time*. Stores the registered Interceptors at *runtime* and notifies the Interceptors at *runtime* when specific events occur.
- **Interceptor**: Register at the Framework for specific Events at *runtime*. Gets called at *runtime* when these Events occur.

6.4 Extension Interface

Let clients access a component only via specialized extension interface, and introduce on such interface for each role that the component provides. Introduce new extension interfaces whenever the component evolves to include new functionality or updated signatures within existing extension interfaces.

— (Schmidt et al., 2000)

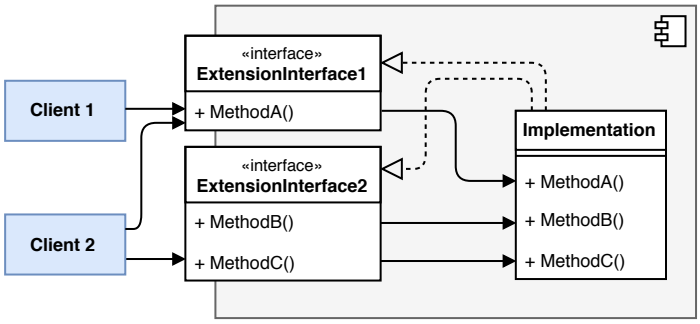

Figure 6.4: Extension Interface

Participants and Bindings:

- **Clients**: The clients are dependent on the used interfaces at *compile time* and call the defined Methods at *runtime*.
- **Interfaces**: Specifies the interfaces for the individual roles at *compile time*.
- **Implementation**: Implements all the interfaces representing the different roles or usage scenarios at *compile time*. Gets called by the Clients at *runtime* (but only via the interfaces).

6.5 Reactor

Provide an event handling infrastructure that waits 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.

— (Schmidt et al., 2000)

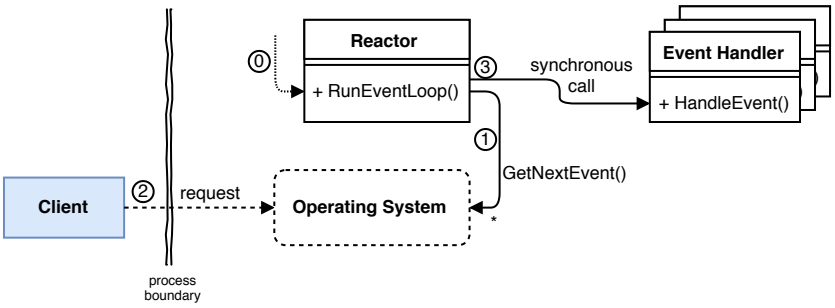

Figure 6.5: Reactor

Participants and Bindings:

- **Client**: Makes requests to the operating system at *runtime*. Depends on the communication method at *runtime*.
- **Reactor**: Dependent on the Operating System and the Event Handlers at *compile time*. Runs the Event Loop and Routes events to the correct Event Handler at *runtime*.
- **Operating System**: Dependent on the communication method at *runtime*. Returns incoming events one after another to the Reactor at *runtime*.
- **Event Handler**: Gets called by the Reactor at *runtime*. Handles the Event at *runtime*.

6.6 Proactor

Split an application's functionality into asynchronous operations that perform activities on event sources and completion handlers that use the results of asynchronous operations to implement application service logic. let the operating system execute the asynchronous operations, but execute the completion handlers in the application's thread of control.

— (Schmidt et al., 2000)

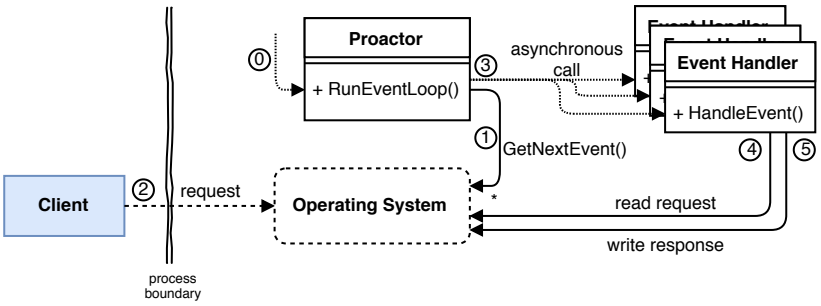

Figure 6.6: Proactor

Participants and Bindings:

- **Client**: Dependent on the communication method at runtime.
- **Proactor**: Depends on the Operating System and the Event Handler Classes at *compile time*. Runs the asynchronous Event Loop at *runtime*.
- **Operating System**: Dependent on the communication method at *runtime*. Returns incoming events one after another to the Proactor at *runtime*. Gets interacted with by the asynchronous Event Handlers at *runtime*.
- **Event Handler**: Get called asynchronously by the Proactor at *runtime*. Gets the request, processes it and writes the response to the operating system at *runtime*.

6.7 Asynchronous Completion Token

Along with each call that a client issues on an synchronous operation, transmit an synchronous completion token that contains the minimum amount of information needed to identify how the client should process the operation's response.

— (Schmidt et al., 2000)

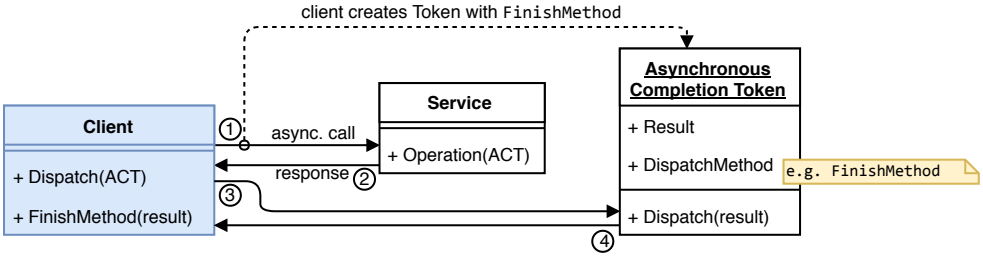

Figure 6.7: Asynchronous Completion Token

Participants and Bindings:

- **Client**: Dependent on Asynchronous Completion Token Object and Service Interface at *compile time*.
- **Service**: Dependent on Asynchronous Completion Token at *compile time*. Sets the result at runtime at returns it back to the client at *runtime*.
- **Asynchronous Completion Token**: Created by the Client and Manipulated by the Service at *runtime*.

6.8 Acceptor-Connector

Decouple the connection and initialization of peer event handlers in a networked system from the processing that these peers subsequently perform.
— (Schmidt et al., 2000)

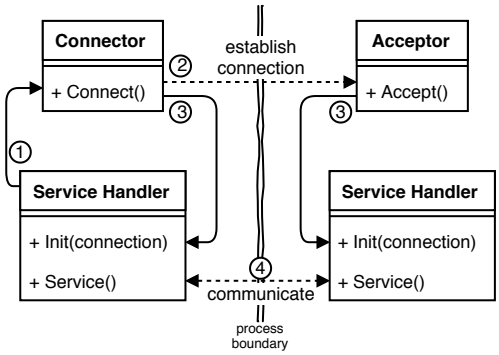

Figure 6.8: Acceptor-Connector

Participants and Bindings:

- **Connector**: Dependent on the Service Handler Interface at *compile-time*. Dependent on the communication method to the Acceptor at *runtime*.
- **Acceptor**: Dependent on the Service Handler Interface at *compile-time*. Dependent on the communication method from the Connector at *runtime*.
- **Service Handler**: Dependent on the Connector-Interface at *compile time*. Either initiates the communication or gets connected at *runtime*. Dependent on the communication format between Service Handlers at *runtime*.

6.9 Scoped Locking

Scope the critical section - if this has not already been done - and acquire the lock automatically when control enters the scope. Similarly, automate the release of the lock when control leaves the scope via any exit path.

— (Schmidt et al., 2000)

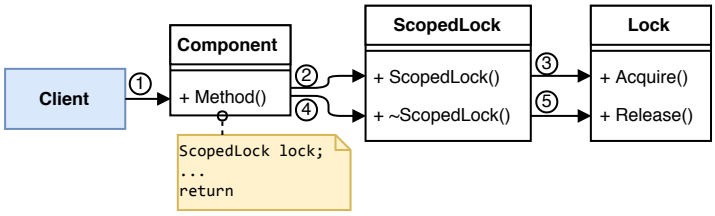

Figure 6.9: Scoped Locking

Participants and Bindings:

- **Client**: Depends on the Component at *compile time*. Calls the method of the Component at *runtime*.
- **Component**: Depends on the ScopedLock at *compile time*. Creates a local instance of the ScopedLock which acquires a Lock at *runtime*. After the callstack is unwinded, the destructor of the ScopedLock releases the Lock again at *runtime*.
- **ScopedLock**: Depends on the Lock at *compile time*. On construction it acquires the Lock, and on desctruction it releases the Lock at *runtime*.
- **Lock**: Depends on the threadsafe locking mechanism of the operating system at *runtime*. Gets acquired and release by the ScopedLock at *runtime*.

6.10 Strategized Locking

Define locks in terms of "pluggable" types, with each type objectifying a particular synchronization strategy. Provide all types with a common interface, so that a component can use all lock types uniformly without being dependent on their implementation.

— (Schmidt et al., 2000)

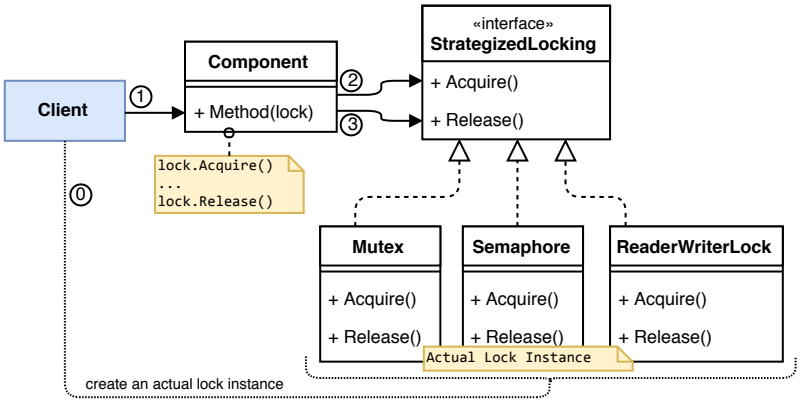

Figure 6.10: Strategized Locking

Participants and Bindings:

- **Client**: Depends on the StrategizedLocking interface, the actual used Lock Class and the Component at *compile time*. Creates the actual Lock instance at *runtime* and configures the Component with the actual Lock instance at *runtime*. Calls the Method of the Component at *runtime*.
- **Component**: Depends on the StrategizedLocking interface at *compile time*. Calls the Acquire and Release Methods at *runtime*.
- **StrategizedLocking**: Specifies the common Interface for Lock-Objects at *compile time*.
- **Actual Lock Instance (Mutex, Semaphore, ...)**: Implements the StrategizedLocking Interface at *compile time*. Is created by the Client at *runtime* and set to the Component at *runtime*. Is used by the Component at *runtime* to acquire and release a lock.

6.11 Thread-Safe Interface

Split a component's methods into publicly accessible interface methods and corresponding private implementation methods. An interface method acquires a lock, calls its corresponding implementation method, and released the lock. An implementation method assumes the necessary lock is held, does its work, and only invokes other implementation methods.

— (Schmidt et al., 2000)

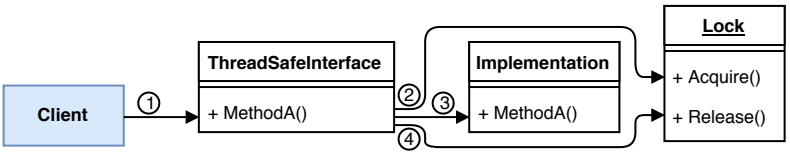

Figure 6.11: Thread-Safe Interface

Participants and Bindings:

- **Client**: Dependent on the ThreadSafeInterface-Interface at *compile time*. Calls the Methods of it at *runtime*.
- **ThreadSafeInterface**: Implements thread-safe version of the Implementation Methods at *compile time*. Acquires the Lock, calls the methods, and releases the Lock at *runtime*.
- **Implementation**: Implements the Methods at *compile time*. Gets called at *runtime* with the assumption that all necessary locks are held.
- **Lock**: Is acquired by the ThreadSafeInterface at *runtime*. Assures that only one object can hold it at *runtime*.

6.12 Double-Checked Locking

Provide the shared objects with a "hint" as to whether execution of a particular critical section is necessary. Check this hint before and after acquiring the lock that guards this critical section.

— (Schmidt et al., 2000)

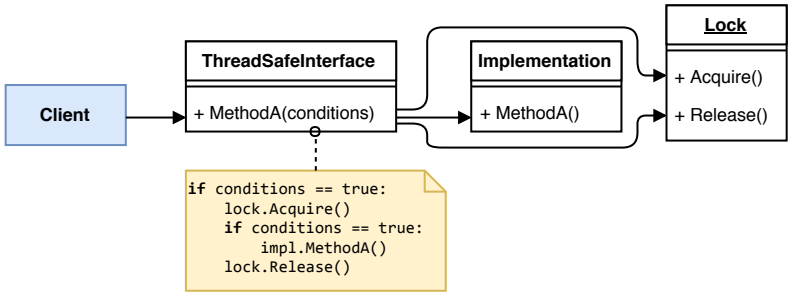

Figure 6.12: Double-Checked Locking

Participants and Bindings:

- **Client**: Dependent on the ThreadSafeInterface-Interface at *compile time*. Calls the Methods of it at *runtime*.
- **ThreadSafeInterface**: Implements thread-safe version of the Implementation Methods at *compile time*. Checks the prerequisites (condition), acquires the Lock, checks again, calls the methods, and releases the Lock at *runtime*.
- **Implementation**: Implements the Methods at *compile time*. Gets called at *runtime* with the assumption that all necessary locks are held.
- **Lock**: Is acquired by the ThreadSafeInterface at *runtime*. Assures that only one object can hold it at *runtime*.

6.13 Active Object

Define the units of concurrency to be service requests on components, and run service requests on a component in a different thread from the requesting client thread. Enable the client and component to interact asynchronously to produce and consume service results.

— (Schmidt et al., 2000)

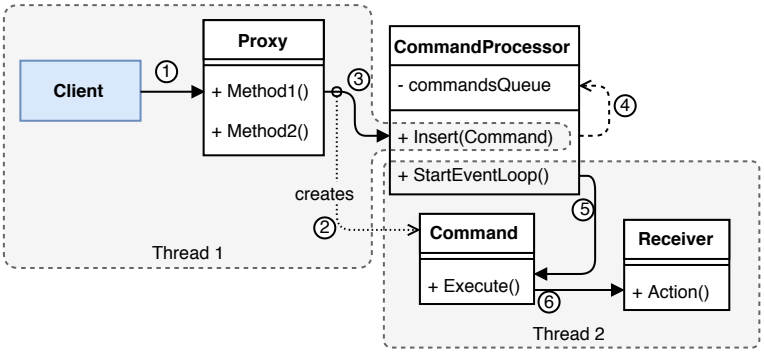

Figure 6.13: Active Object

Participants and Bindings:

- **Client**: Depends on the Proxy at *compile time*. Calls the methods of the Proxy at *runtime*.
- **Proxy**: Depends on the Command and the CommandProcessor at *compile time*. Creates Command objects at *runtime* and gives it to the CommandProcessor at *runtime*.
- **CommandProcessor**: Depends upon the Command-Interface at *compile time*. Manages the commands at *runtime* (thread-safe). Executes the commands at *runtime*.
- **Command**: Depends on the implementation interface at *compile time*. Is created by the Proxy in a different thread than it is actually executed at *runtime*. Calls the implementation at *runtime*.
- **Receiver**: Gets called by the commands at *runtime*.

6.14 Monitor Object

Execute a shared object in each of its client threads, and let it self-coordinate a serialized, yet interleaved, execution sequence. Access the shared object only through synchronized methods that allow execution of only one method at a time.

— (Schmidt et al., 2000)

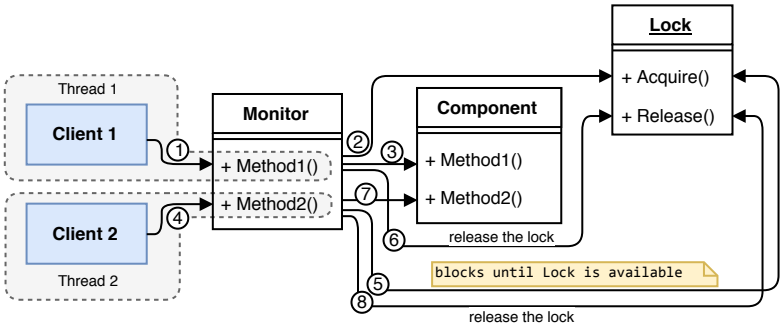

Figure 6.14: Monitor Object

Participants and Bindings:

- **Clients**: Depend on the Monitor-Interface at compile time. Call the Methods at runtime.
- **Monitor**: Depends on the Lock and the Component at compile time. Ensures at runtime that only one Method call per Type (or depending on the implementation per Object Instance) is possible at runtime.
- **Component**: Gets called by the Monitor at runtime.
- **Lock**: Depends on the operating systems methods to implement a threadsafe lock at compile time. Gets Acquired and Released by the Monitor at runtime.

6.15 Half-Sync/Half-Async

Decompose the services of concurrent software into two separated layers - synchronous and asynchronous - and add a queuing layer to mediate communication between them.

— (Schmidt et al., 2000)

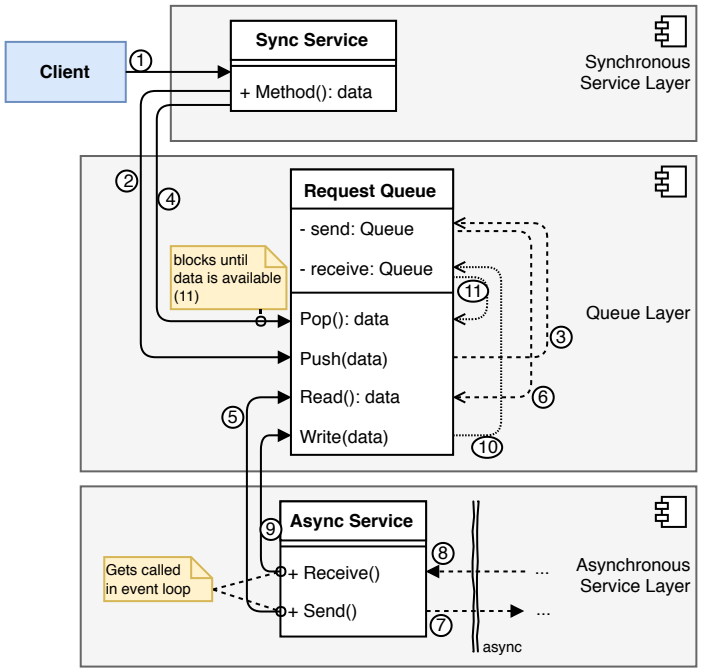

Figure 6.15: Half-Sync/Half-Async

Participants and Bindings:

- **Client**: Depends on the Synchronous Service Layer (especially the SyncService) at *compile time*. Calls the Method at *runtime* and requires it to be run synchronously.
- **Sync Service**: Depends on the Queue Layer (especially the Request Queue) at *compile time*. Adds requests at *runtime* and waits until a response is available at *runtime*.
- **Request Queue**: Implements a waiting queue at *compile time*. If the queue is empty the Pop-Method blocks until an element is available at *runtime*.

- **Async Service**: Depends on the Queue Layer (especially the Request Queue) at *compile time*. Depends on the communication method and message format at *compile time*. Issues asynchronous requests to the corresponding resource at *runtime* and adds the Response to the Request Queue as soon as it is available at *runtime*.

6.16 Leader/Followers

Use a pre-allocated pool of threads to coordinate the detection, demultiplexing, dispatching, and processing of events. In this pool only one thread at a time - the leader - may wait for an event on a set of shared event sources. When an event arrives, the leader promotes another thread in the pool to become the new leader and then processes the event concurrently.

— (Schmidt et al., 2000)

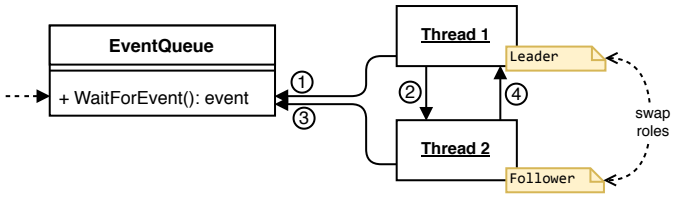

Figure 6.16: Leader/Followers

Participants and Bindings:

- **EventQueue**: Depend on the event method at *compile time*. Waits for incoming events at *runtime*.
- **Thread**: Depends on the EventQueue at *compile time*. If it is the current Leader is calls the WaitForEvents Method at *runtime* and then sets the next Follower as Leader. If it is a Follower is just waits until it is a Leader at *runtime*.

6.17 Thread-Specific Storage

Introduce a common access point for the environmentally bound object, but maintain its physical object instances in storage that is local to each thread.
— (Schmidt et al., 2000)

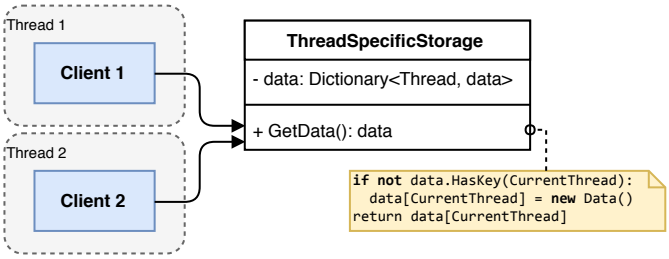

Figure 6.17: Thread-Specific Storage

Participants and Bindings:

- **Client**: Depends on the ThreadSpecificStorage at *compile time*. Gets its data as it would be a local variable at *runtime*.
- **Thread Specific Storage**: Ensures that every Client Thread5 can only access its own instance of the data at *runtime*.

7 POSA 3 Design Patterns

In this chapter 10 design patterns from the book "Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management" (Kircher and Jain, 2004) are analyzed:

- **Resource Acquisition**: Lookup (96), Lazy Acquisition (97), Eager Acquisition (98), Partial Acquisition (99)
- **Resource Lifecycle**: Caching (100), Pooling (101), Coordinator (102), Resource Lifecycle Manager (102)
- **Resource Release**: Leasing (103), Evictor (104)

7.1 Lookup

Provide a lookup service that allows services in a distribute system to register their references when they become available, and deregister their references when they become unavailable.

— (Kircher and Jain, 2004)

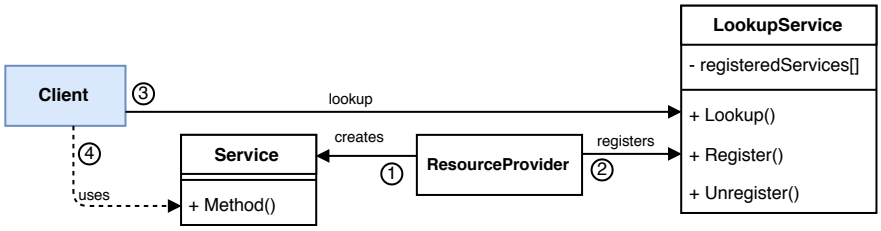

Figure 7.1: Lookup

Participants and Bindings:

- **Client**: Dependent on the LookupService at *compile time*. Looks for services at *runtime* and calls these services at *runtime*.
- **LookupService**: Manages the registration of Services at *runtime*. Looks up the registered Services at *runtime*.
- **ResourceProvider**: Dependen on the LookupService and the Service-Class at *compile time*. Creates and registers Services at the LookupService at *runtime*. Maintains access to resources used by the services at *runtime*.
- **Service**: Gets created by the ResourceProvider at *runtime*, gets called by the Client at *runtime*.

7.2 Lazy Acquisition

Acquire resources at the latest possible point in time. The resource is not acquired until it is actually about to be used. At the point at which a resource user is about to use a resource, it is acquired and returned to the resource user.
— (Kircher and Jain, 2004)

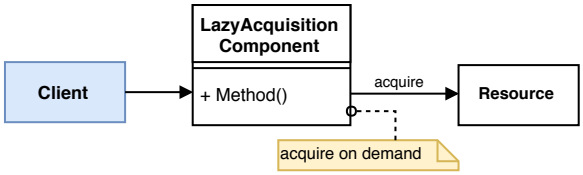

Figure 7.2: Lazy Acquisition

Participants and Bindings:

- **Client**: Depends on the Lazy Acquisition Component at *compile time*. Calls its Method at *runtime*.
- **Lazy Acquisition Component**: Depends on the Resource Interface at *compile time*. Gets called by the Client at *runtime* and acquires the Resource at the latest possible moment at *runtime*.
- **Resource**: Gets acquired by the Lazy Acquisition Component at *runtime*.

7.3 Eager Acquisition

The Eager Acquisition pattern describes how run-time acquisition of resources can be made predictable and fast by eagerly acquiring and initializing resources before their actual use

— (Kircher and Jain, 2004)

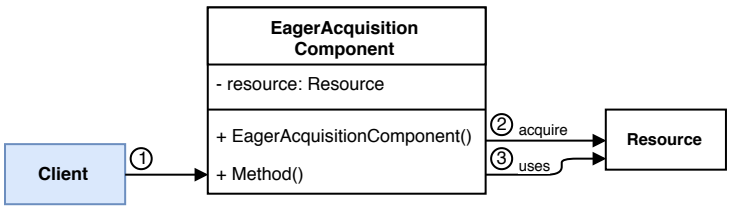

Figure 7.3: Eager Acquisition

Participants and Bindings:

- **Client**: Depends on the Eager Acquisition Component at *compile time*. Calls its Method at *runtime*.
- **Eager Acquisition Component**: Depends on the Resource Interface at *compile time*. Acquires the Resource at the earliest possible point (constructor) at *runtime* and stores it for later. Gets called by the Client at *runtime* and uses the Resource at *runtime*.
- **Resource**: Gets acquired early by the Eager Acquisition Component at *runtime*.

7.4 Partial Acquisition

Split the acquisition of each resource into multiple stages. In each stage, acquire only a part of the resource, so that its acquisition gradually completes over time, in accordance with overall application quality of service needs.

— (Kircher and Jain, 2004)

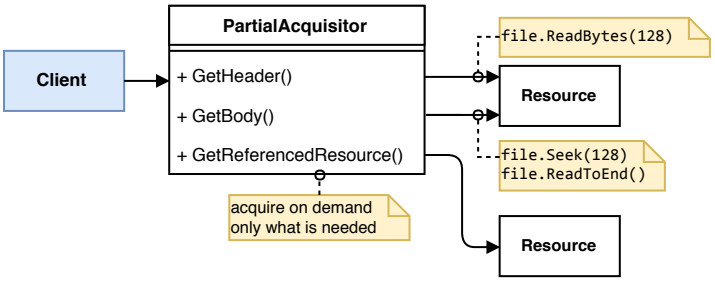

Figure 7.4: Partial Acquisition

Participants and Bindings:

- **Client**: Depends on the Partial Acquisitor at *compile time*. Calls its Methods at *runtime*.
- **PartialAcquisitor**: Depends on the Resource-Interface at *compile time*. Acquires it at *runtime*, but only as much as needed. Acquires additional Resource as needed at *runtime*.
- **Resource**: Gets acquired by the PartialAcquisitor at *runtime*.

7.5 Caching

Rather than destroying a resource after use, store it in an in-memory cache. When the resource is needed again, fetch it from the cache and return it, instead of creating it anew.

— (Kircher and Jain, 2004)

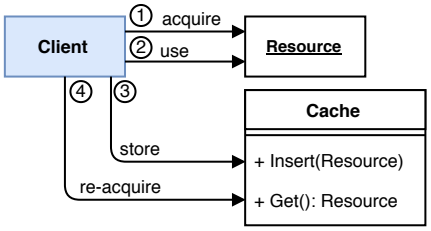

Figure 7.5: Caching

Participants and Bindings:

- **Client**: Depends on the Resource and the Resource Cache at compile time. Acquires and uses the Resource at runtime. Stores it in the Cache for later use at runtime. Re-acquires the Resource if necessary from the Cache at runtime.
- **Resource**: Gets acquired and used by the Client at runtime and stored in the Cache at runtime.
- **Cache**: Stores already acquired Resources for later use at runtime.

7.6 Pooling

Keep a certain number of resources available in an in-memory resource pool. Rather than repeatedly creating resources from scratch, retrieve the resources from the pool quickly and predictably. When the application no longer needs a resource, it must be returned to the pool so it becomes available for subsequent acquisition.

— (Kircher and Jain, 2004)

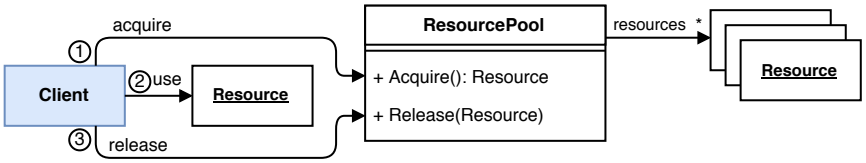

Figure 7.6: Pooling

Participants and Bindings:

- **Client**: Depends on the ResourcePool and the Resource at *compile time*. Acquires and Releases Resources at the ResourcePool at *runtime*. Uses the Resource at *runtime*.
- **ResourcePool**: Manages the Resources at *runtime*.
- **Resource**: Is managed by the ResourcePool and gets acquired and released by the Clients at *runtime*.

7.7 Coordinator

Introduce a coordinator that supervises the execution and completion of a task by all participants. The coordinator ensures that either all contributing participants complete successfully or, in the event of even a single participating task failing, it appears that the entire task did not execute at all.

— (Kircher and Jain, 2004)

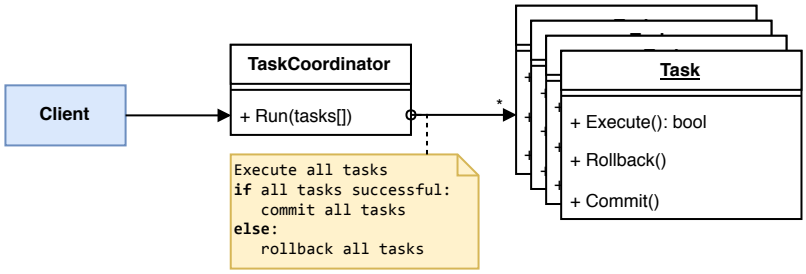

Figure 7.7: Task Coordinator

Participants and Bindings:

- **Client**: Depends on the TaskCoordinator at *compile time*. Calls the TaskCoordinator at *runtime*.
- **Task Coordinator**: Depends on the Task-Interface at *compile time*. Executes the Tasks at *runtime* and depending on the success either commits or rolls back all Tasks at *runtime*.
- **Task**: Implements a Task which can be Executed, Commited or Rolled Back at *compile time*. Gets called by the Task Coordinator at *runtime*.

7.8 Resource Lifecycle Manager

See 8.44 Object Manager (on page 151).

7.9 Leasing

Have the provider create a lease for each resource held by clients. Include a time duration in the lease that specifies how long a client can use the resource. After the time duration expires, release the reference to the resource in the client and the resource in the provider.

— (Kircher and Jain, 2004)

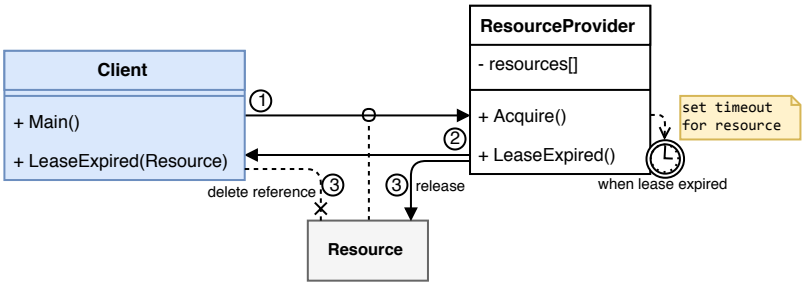

Figure 7.8: Leasing

Participants and Bindings:

- **Client**: Depends on the ResourceProvider Interface and the Resource Interface at *compile time*. Acquires the Resource at the ResourceProvider at *runtime*. Uses the Resource at *runtime*. Release the reference at latest when the Lease expired at *runtime*.
- **Resource Provider**: Depends on the Timer-Interface and the Resource-Interface at *compile time*. Manages the resources at *runtime*, sets up timer for each Resource and after the lease expires, informs all clients and releases the resource at *runtime*.
- **Resource**: Gets acquired at *runtime*, and released after an expiration timeout at *runtime*.

7.10 Evictor

Introduce an evictor to monitor the use of resources and control their lifetime. Resources that are not accessed after a specific period of time are removed to free up space for other resources.

— (Kircher and Jain, 2004)

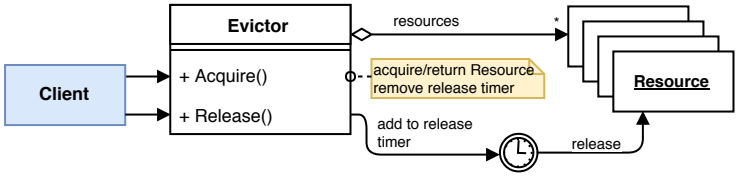

Figure 7.9: Evictor

Participants and Bindings:

- **Client**: Depends on the Evictor Interface and the Resource Interface at *compile time*. Calls the Evictor's methods to acquire and release resources at *runtime*. Uses the Resources at *runtime*.
- **Evictor**: Depends on the Timer-Interface and the Resource-Objects at *compile time*. Acquires Resources on demand as requested by the Client and holds its references even after Release for a specific amount of time to be immediately reused at *runtime*. After expiration it releases the Resource to free up memory at *runtime*.
- **Resources**: Get acquired and released by the Evictor at *runtime*. Get used by the Client at *runtime*.

8 POSA 4 Design Patterns

In this chapter 55 design patterns from the book "Pattern-Oriented Software Architecture Volume 4: Pattern Language for Distributed Computing" (Buschmann, Henney, and Schmidt, 2007) are analyzed:

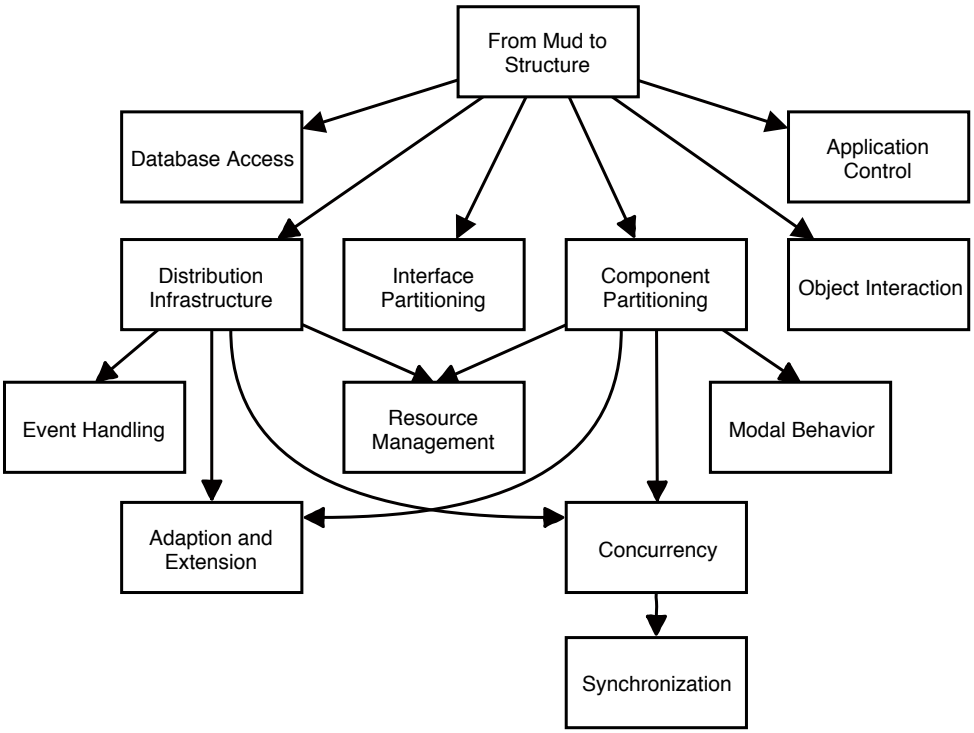

Figure 8.1: Overview

- **From Mud to Structure**: Domain Model (108), Layers (60), Model-View-Controller (65), Presentation-Abstraction-Control (66), Microkernel (68), Reflection (69), Pipes and Filters (61), Shared Repository (109), Blackboard (62), Domain Object (110)

- **Distribution Infrastructure**: Messaging (111), Message Channel (113), Message Endpoint (114), Message Translator (115), Message Router (116), Broker (63), Client Proxy (117), Requestor (118), Invoker (119), Client Request Handler (120), Server Request Handler (121), Publisher-Subscriber → Observer (51)
- **Event Handling**: Reactor (82), Proactor (83), Acceptor-Connector (85), Asynchronous Completion Token (84)
- **Interface Partitioning**: Explicit Interface (122), Extension Interface (81), Introspective Interface (123), Dynamic Invocation Interface (124), Proxy (53), Business Delegate (125), Facade (44), Combined Method (126), Iterator (48), Enumeration Method (127), Batch Method (128)
- **Component Partitioning**: Encapsulated Implementation → Proxy (53), Whole-Part → Composite (42), Composite (42), Master-Slave (70), Half-Object plus Protocol (129), Replicated Component Group (130)
- **Application Control**: Page Controller (131), Front Controller (132), Application Controller (133), Command Processor (71), Template View (134), Transform View (135), Firewall Proxy (136), Authorization (137)
- **Concurrency**: Half-Sync/Half-Async (92), Leader/Followers (93), Active Object (90), Monitor Object (91)
- **Synchronization**: Guarded Suspension (138), Future (139), Thread-Safe Interface (88), Double-Checked Locking (89), Strategized Locking (87), Scoped Locking (86), Thread-Specific Storage (94), Copied Value (140), Immutable Value (141)
- **Object Interaction**: Observer (51), Double Dispatch (142), Mediator (49), Memento (50), Context Object (143), Data Transfer Object (144), Command (41), Message (112)
- **Adaptation and Extension**: Bridge (38), Object Adapter → Adapter (37), Interceptor (80), Chain of Responsibility (40), Interpreter (47), Visitor (58), Decorator (43), Template Method (57), Strategy (56), Null Object (146), Wrapper Facade (78), Execute-Around Object (145), Declarative Component Configuration (147)
- **Object Behavior**: Objects for States → State (55), Methods

for States (148), Collections for States (149)

- **Resource Management**: Object Manager (151), Container (150), Component Configurator (79), Lookup (96), Virtual Proxy (152), Lifecycle Callback (153), Task Coordinator → Coordinator (102), Resource Pool → Pooling (101), Resource Cache → Caching (100)), Lazy Acquisition (97), Eager Acquisition (98), Partial Acquisition (99), Activator (154), Evictor (104), Leasing (103), Automated Garbage Collection (155), Counting Handle → Counted Pointer (75), Abstract Factory (36), Builder (39), Factory Method (45), Disposal Method (156)
- **Database Access**: Database Access Layer (157), Data Mapper (158), Row Data Gateway (159), Table Data Gateway (160), Active Record (161)

8.1 Domain Model

Create a model that defines and scopes a system's business responsibilities and their variations: model elements are abstractions meaningful in the application domain, while their roles and interactions reflect the domain workflow.

— (Buschmann, Henney, and Schmidt, 2007)

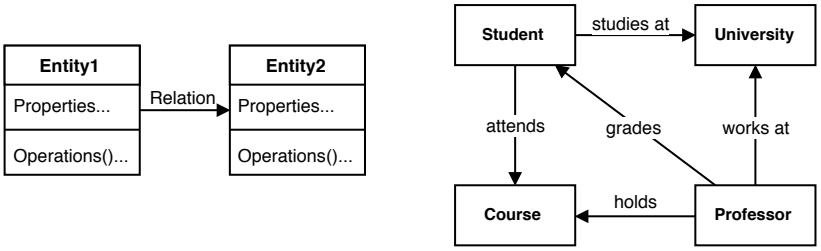

Figure 8.2: Domain Model

Participants and Bindings: The Domain Model is on a highly abstract conceptual level and must not directly correspond to the actual technical implementation. Here the objects and their relations get bound at *requirements analysis time*, *architecture time*, or at *design time*. These decisions are some of the earliest one can take.

8.2 Shared Repository

Maintain all data in a central repository shared by all functional components of the data-driven application and let the availability, quality, and state of that data trigger and coordinate the control flow of the application logic.

— (Buschmann, Henney, and Schmidt, 2007)

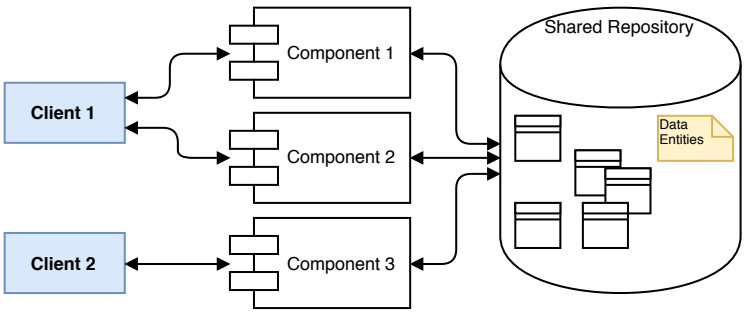

Figure 8.3: Shared Repository

Participants and Bindings:

- **Client:** Multiple Clients may use multiple functional components at *runtime* in parallel and may manipulate even the same objects.
- **Component:** The Components depend upon the shared repository for data storage and manipulation at *runtime*. They get called by the Clients at *runtime*.
- **Shared Repository:** Stores the Data Entities for all different components at runtime. Manages access and synchronization amongst the data at *runtime*.

8.3 Domain Object

Encapsulate each distinct functionality of an application in a self-contained building-block - a domain object.

— (Buschmann, Henney, and Schmidt, 2007)

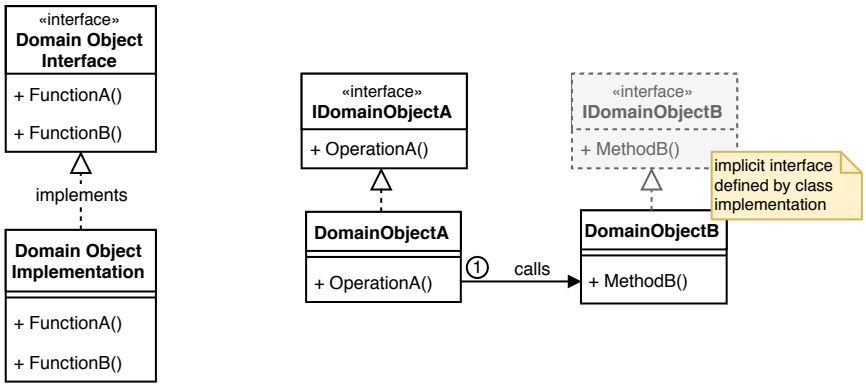

Figure 8.4: Domain Object

Participants and Bindings:

- **Domain Object Implementation**: Implements a Domain Object Interface at *compile time*. May depend on other Domain Objects at *compile time*. May call other Domain Objects at *runtime*.
- **Domain Object Interface**: Specifies the interface for a Domain Object at *compile time*. If a Domain Object has no interface, its method signatures implicitly defined a Domain Object Interface at *compile time*.

8.4 Messaging

Connect the services via a message bus that allows them to transfer data messages asynchronously. Encode the messages so that senders and receivers can communicate reliably without having to now all the data type information statically.

— (Buschmann, Henney, and Schmidt, 2007)

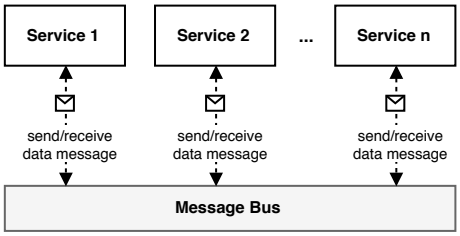

Figure 8.5: Messaging

Participants and Bindings:

- **Service**: Dependent on the common message data format and the communication method at *runtime*.
- **Message Bus**: Dependent on the common message data format and the communication method at *runtime*.
- **Data Message**: Specifies the common message data format at *compile time*.

8.5 Message

Encapsulate method requests and data structures to be sent across the network into messages: byte streams that include a header specifying the type of information being transmitted, its origin, destination, size, and other structural information, and a payload that contains the actual information.
— (Buschmann, Henney, and Schmidt, 2007)

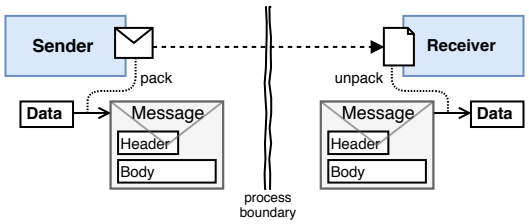

Figure 8.6: Message

Participants and Bindings:

- **Sender**: Depends on the communication method and the common message format at *compile time*. Packs Data into Messages and sends them to the receiver at *runtime*.
- **Message**: Specifies the common message format at *compile time*. Gets created at *runtime*.
- **Receiver**: Depends on the communication method and the common message format at *compile time*. Unpacks Messages into Data at *runtime*.

8.6 Message Channel

Connect the collaborating clients and services using a message channel that allows them to exchange messages.

— (Buschmann, Henney, and Schmidt, 2007)

Figure 8.7: Message Channel

Participants and Bindings:

- **Client**: Dependent on the common messaging data format to send the data at *runtime*. Dependent on the communication form of the message channel at *compile time* (if statically bound) or *runtime* (with a dynamic approach).
- **Message Channel**: Dependent on the common messaging data format (at least the header or basic byte stream) at *runtime*. Gets established by some lookup-mechanism at *runtime*.
- **Receiver**: Dependent on the common messaging data format to read the data at *runtime*. Dependent on the communication form of the message channel at *compile time* (if statically bound) or *runtime* (with a dynamic approach).

8.7 Message Endpoint

Connect the clients and services of an application ot the messaging infrastructure using specialized message endpoints that allow clients and services to exchange messages.

— (Buschmann, Henney, and Schmidt, 2007)

Figure 8.8: Message Endpoint

Participants and Bindings:

- **Client / Receiver**: Dependent on the Message Endpoint at *compile time*. Writes or Reads Data onto or from it at *runtime*.
- **Message Endpoint**: Dependent on the common message data format and the communication type to the middleware at *runtime*.
- **Messaging Middleware**: Routes Messages from one endpoint to another at *runtime*. Maybe dependent on the common message data format (headers, encoding, ...) at *runtime*, but not necessarily.

8.8 Message Translator

Introduce message translators between clients and services of an application that convert messages from one format into another.

— (Buschmann, Henney, and Schmidt, 2007)

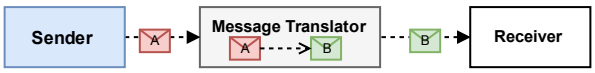

Figure 8.9: Message Translator

Participants and Bindings:

- **Sender**: Dependent on the message format A and the general communication method at *runtime*.
- **Message Translator**: Dependent on the message format A and message format B and the general communication method at *runtime*.
- **Receiver**: Dependent on the message format B and the general communication method at *runtime*.

8.9 Message Router

Provide message routers that consume messages from one message channel and reinsert them into different message channels, depending on a set of conditions.

— (Buschmann, Henney, and Schmidt, 2007)

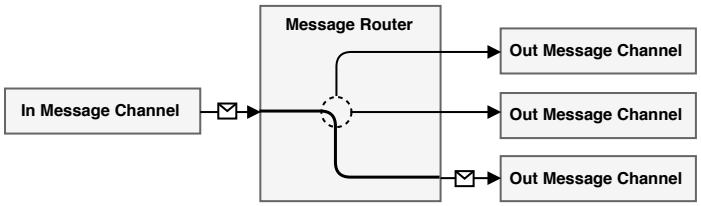

Figure 8.10: Message Router

Participants and Bindings:

- **In Message Channel**: Dependent on the common message format and the general communication method at *runtime*.
- **Message Router**: Dependent on the common message format and the general communication method at *runtime*.
- **Out Message Channel**: Dependent on the common message format and the general communication method at *runtime*.

8.10 Client Proxy

Provide a client proxy in the client's address space that is a surrogate for the remote component. The proxy provides the same interface as the remote component, and maps client invocations to the specific message format and protocol used send these invocations across the network.

— (Buschmann, Henney, and Schmidt, 2007)

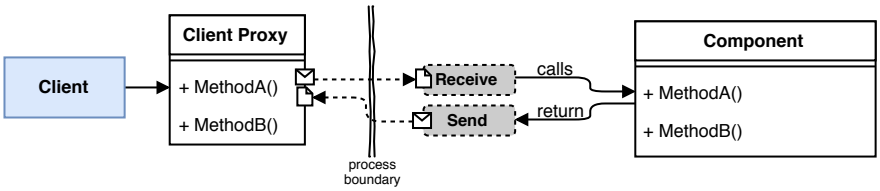

Figure 8.11: Client Proxy

Participants and Bindings:

- **Client**: Dependent on the Client Proxy at *compile time*. Calls the Methods at *runtime*.
- **Client Proxy**: Dependent on the common message format and communication method on the client side at *runtime*.
- **Receive/Send**: Dependent on the common message format and communication method on the server side at *runtime*.
- **Component**: Gets called at *runtime*.

8.11 Requestor

Create a requestor that encapsulates the creation, handling and sending of request messages to remote components.

— (Buschmann, Henney, and Schmidt, 2007)

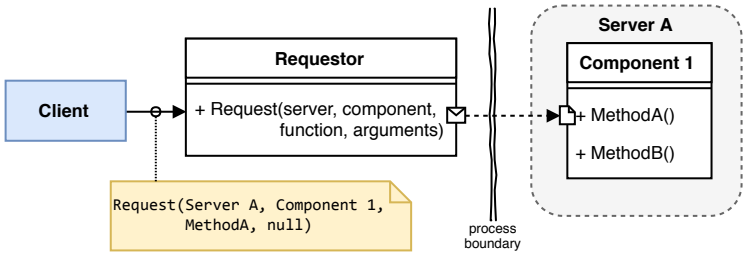

Figure 8.12: Requestor

Participants and Bindings:

- **Client**: Dependent on the Requestor-Class at *compile time*. Calls the Requestor with specific arguments at *runtime*.
- **Requestor**: Dependent on the communication method to the servers at *runtime*. Gets called by the client at *runtime*.
- **Server/Component**: Gets requests from the Requestor at *runtime*.

8.12 Invoker

Create an invoker that encapsulates the reception and dispatch of request messages from remote clients in a specific method of a component implementation.

— (Buschmann, Henney, and Schmidt, 2007)

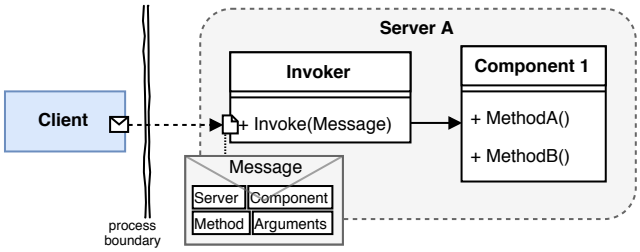

Figure 8.13: Invoker

Participants and Bindings:

- **Client**: Dependent on the communication method and the common message format at *runtime*.
- **Invoker**: Dependent on the Component-Class at *compile time*. Dependent on the communication method and the common message format at *runtime*.
- **Message**: Specifies the common message format at *runtime*.
- **Server**: Dependent on the communication method at *runtime*.
- **Component**: Gets called by the Invoker at *runtime*.

8.13 Client Request Handler

Provide a specialized client request handler that encapsulates and performs all IPC tasks on behalf of client component that send requests to and receive replies from the network.

— (Buschmann, Henney, and Schmidt, 2007)

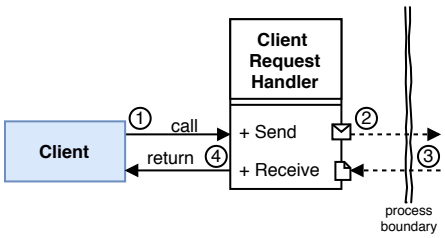

Figure 8.14: Client Request Handler

Participants and Bindings:

- **Client**: Dependent on the Client Request Handler at *compile time*. Calls the Send Method at *runtime*.
- **Client Request Handler**: Dependent on the communication method and the common message format at *runtime*.

8.14 Server Request Handler

Provide a speicalized server request handler that encapsulates and performs all IPC tasks on behalf of remote components that receive requests from and send replies to the network.

— (Buschmann, Henney, and Schmidt, 2007)

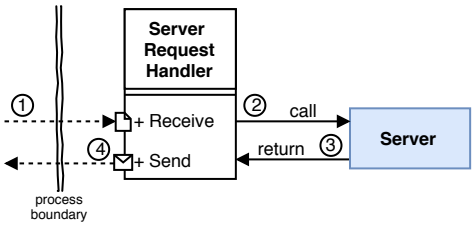

Figure 8.15: Server Request Handler

Participants and Bindings:

- **Server Request Handler**: Dependent on the server at *compile time*. Dependent on the communication method and the common message format. Uses the Server at *runtime*.
- **Server**: Gets called by the Server Request Handler at *runtime*.

8.15 Explicit Interface

Separate the declared interface of a component from its implementation. Export the interface to the clients of the component, but keep its implementation private and location-transparent to the client.

— (Buschmann, Henney, and Schmidt, 2007)

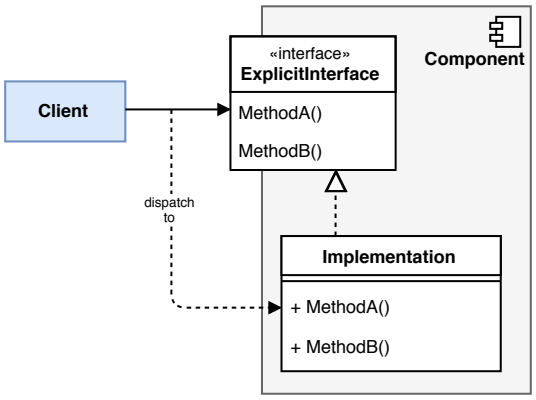

Figure 8.16: Explicit Interface

Participants and Bindings:

- **Client**: Dependend on the Interface at *compile time*. Executes the actual implementation at *runtime*.
- **Interface**: Specifies the Interface at *compile time*.
- **Implementation**: Implements the Interface at *compile time*. Gets called at *runtime*.

8.16 Introspective Interface

Introduce a special introspective interface for the component that allows clients to access information about its mechanism and structure. Keep the introspective interface separate from the component's operational interfaces.

— (Buschmann, Henney, and Schmidt, 2007)

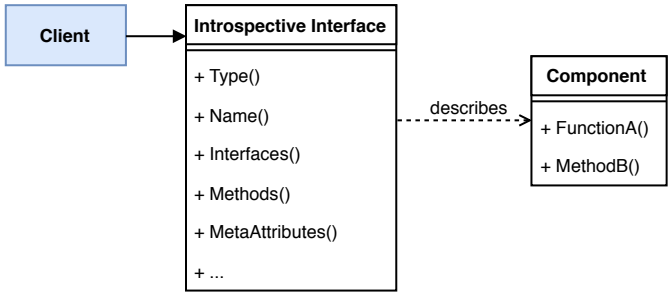

Figure 8.17: Introspective Interface

Participants and Bindings:

- **Client**: Dependent on the IntrospectiveInterface-Interface at *compile time*. Calls its methods at *runtime*. May call methods on the Component at *runtime*.
- **Introspective Interface**: Specifices Methods for providing meta-data for the Component at *compile time*. May construct the MetaData at *pre-compile-time*, *compile time*, or *runtime* dependent on the implementation (static vs. dynamic).
- **Component**: An actual implementation of a domain object at *compile time*. Gets created and called at *runtime*. Is described by the Introspective Interface at *runtime*.

8.17 Dynamic Invocation Interface

Introduce an invocation interface for the component that allows clients to compose calls on the component dynamically. Methods are identified at runtime by strings, and arguments are passed as generally typed collections. Keep the dynamic invocation interface separate from the components operational interfaces.

— (Buschmann, Henney, and Schmidt, 2007)

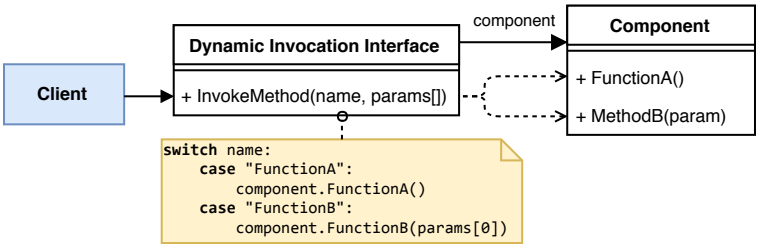

Figure 8.18: Dynamic Invocation Interface

Participants and Bindings:

- **Client**: Dependent on the Dynamic Invocation Interface at *compile time*. Calls InvokeMethod with specific parameters at *runtime*.
- **Dynamic Invocation Interface**: Dependent on the Component at *compile time*. Invokes the Method as requested by the Client on the Component at *runtime*.
- **Component**: Is used by the Dynamic Invocation Interface at *runtime*.

8.18 Business Delegate

Introduce a business delegate for each remote component that can be created, used, and disposed of like a collocated component, and whose interface is identical to that of the component it represents. Let the business delegate perform all networking tasks transparently for clients using the component.
— (Buschmann, Henney, and Schmidt, 2007)

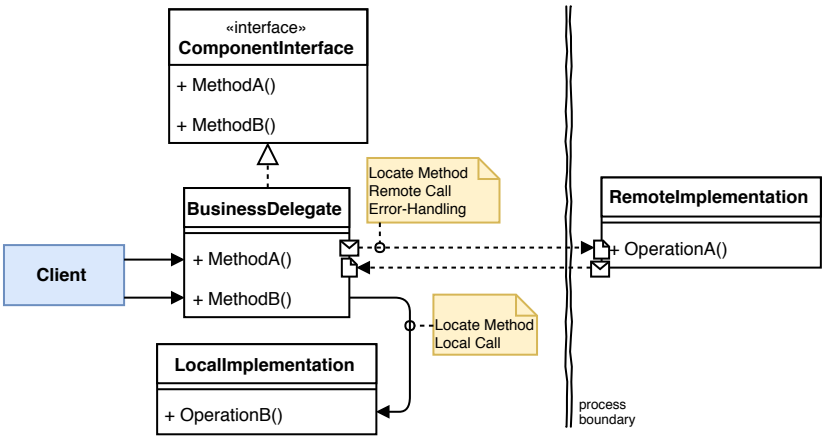

Figure 8.19: Business Delegate

Participants and Bindings:

- **Client**: Depends on the BusinessDelegate-Interface at *compile time*. Calls the methods of the Business Delegate at *runtime*.
- **ComponentInterface**: Specifies the interface for the business component at *compile time*.
- **Business Delegate**: Implements the Component Interface at *compile-time*. Calls the Implementations at *runtime*. Depends on the communication format to the Remote Implementations at *runtime*. If local or remote Implementation is used, is decided at *runtime*.
- **Remote Implementation**: Gets called at *runtime*. Depends on the communication format at *runtime*.
- **Local Implementation**: Gets called at *runtime*.

8.19 Combined Method

Combine methods that must be, or commonly are, executed together on a component into a single method.

— (Buschmann, Henney, and Schmidt, 2007)

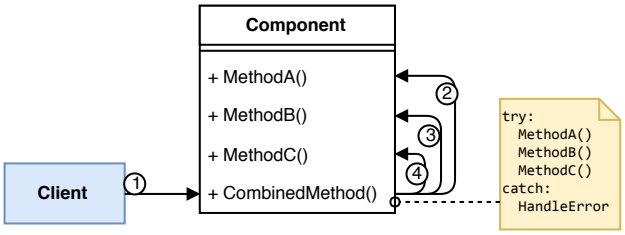

Figure 8.20: Combined Method

Participants and Bindings:

- **Client**: Depends on the Component at *compile time*. Calls the Combined Method at *runtime*.
- **Component**: Implements the CombinedMethod at *compile time*. Calls its own Methods as needed at *runtime*.

8.20 Enumeration Method

Bring the iteration inside the aggregate and encapsulate it in a single enumeration method that is responsible for complete traversal. Pass the task of the loop - the action to be executed on each element of the aggregate - as an argument to the enumeration method, and apply it to each element in turn.
— (Buschmann, Henney, and Schmidt, 2007)

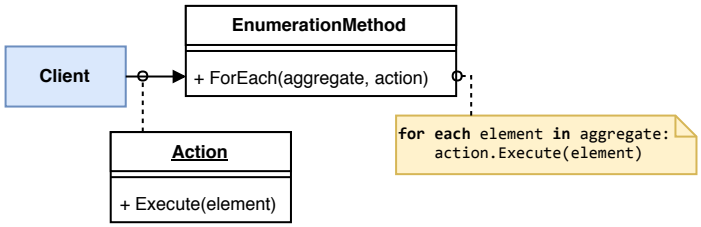

Figure 8.21: Enumeration Method

Participants and Bindings:

- **Client**: Dependent on the Action-Interface and the EnumerationMethod-Interface at compile time. Creates the Action and the EnumerationMethod at runtime and calls the ForEach-Method with the aggregate at runtime.
- **Enumeration Method**: Dependent on the Aggregate-Interface to loop over it at *compile time*. Calls the Action at *runtime*.
- **Aggregate**: Specifies the the Aggregate-Interface at *compile time* and gets used by the EnumerationMethod to iterate over it at *runtime*.
- **Action**: Dependent on the Element-Interface at *compile time*. Gets called by the EnumerationMethod for each element in the aggregate at *runtime*.
- **Element**: Specifies an Element-Object in the Aggregate at *compile time*. Is used by the action at *runtime*.

8.21 Batch Method

Define a single batch method that performs the action on the aggregate repeatedly. The method is declared to take all the arguments for each execution of the action, for example via an array or a collection, and to return results by similar means.

— (Buschmann, Henney, and Schmidt, 2007)

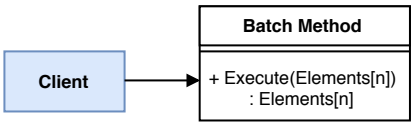

Figure 8.22: Batch Method

Participants and Bindings:

- **Client**: Dependent on the BatchMethod at *compile time*. Creates the Elements and calls the Batch Method in batches of n Elements at *runtime*.
- **Batch Method**: Gets called by the Client at *runtime*.

8.22 Half-Object plus Protocol

Divide the objects into multiple "half-objects", one for each address space in which they is are used. Each half object implements the functionality and data required by the clients that reside in the respective address space. A protocol between the half objects helps to coordinate their activities and keep their state consistent.

— (Buschmann, Henney, and Schmidt, 2007)

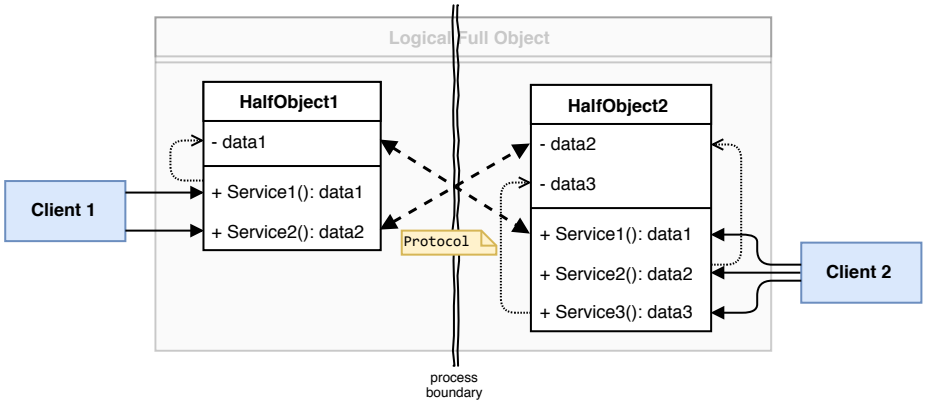

Figure 8.23: Half-Object plus Protocol

Participants and Bindings:

- **Clients**: Depends on the Half-Objects Interfaces at *compile time*.
- **Logical Full Object**: The domain object which is split up into Half-Objects during *design time*, implemented at *compile time* and synchronized over the client domains at *runtime*.
- **Half-Object**: The part of an object which is needed for a specific domain. Specified during design time, implemented at *compile time*. Depends on the Protocol at *compile time* and uses it to synchronize with its other parts if needed at *runtime*.
- **Protocol**: The communication method and message data format specified at *compile time*. Is used by the Half-Objects to synchronize at *runtime*.

8.23 Replicated Component Group

Provide a group of component implementations instead of a single implementation, and replicate these implementations across different network nodes. Forward client requests on the component interface to all implementation instances, and wait until on of the instances returns a result.
— (Buschmann, Henney, and Schmidt, 2007)

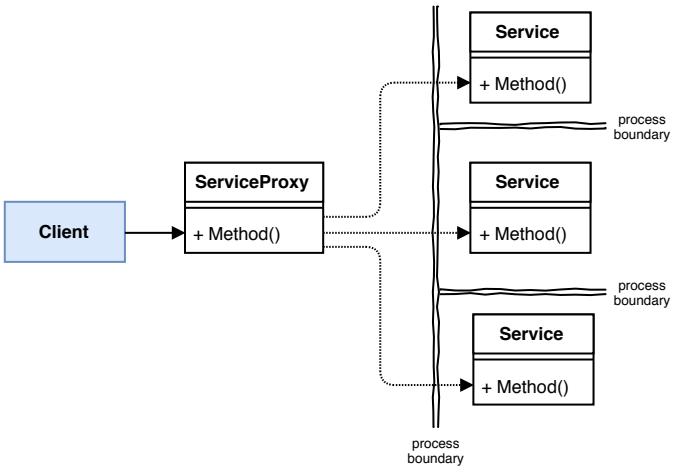

Figure 8.24: Replicated Component Group

Participants and Bindings:

- **Client**: Depends on the Service Proxy at *compile time*. Calls the Method at *runtime*.
- **Service Proxy**: Depends on the communication method and common message format at *compile time*. Depends on the Service-Instances at *runtime*.
- **Service**: Implements the Service-Method at *compile time*. Depends on the communication method and the common message format at *compile time*. Gets called by the Service-Proxy at *runtime*.

8.24 Page Controller

For each form offered by an applications's user interface, introduce a page controller to control the execution of all requests issued from that form.
— (Buschmann, Henney, and Schmidt, 2007)

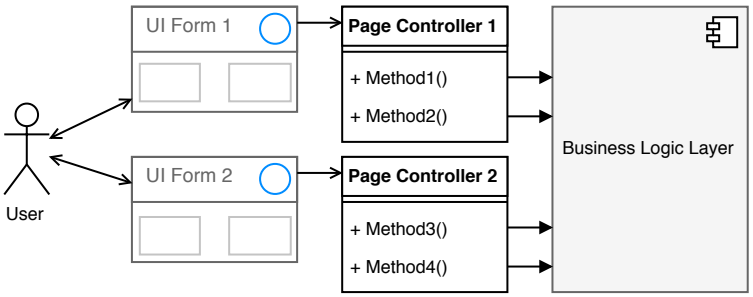

Figure 8.25: Page Controller

Participants and Bindings:

- **User**: Uses the UI Forms at *runtime*.
- **UI Forms**: Dependent on the Page Controller at *compile time*. Interacts with the User at *runtime*. Delegates the Requests to its corresponding Page Controller and gets controlled by it at *runtime*.
- **Page Controller**: Dependent on the UI Form Interface and the Business Logic Interface at *compile time*. Gets requests by the UI Form at *runtime*. Redirects those requests to the Business Logic Layer at *runtime*. Controlls the behaviour of the UI Forms at *runtime*.
- **Business Logic Layer**: Implements the Business Logic for the Application at *compile time*. Gets used by the Page Controller at *runtime*.

8.25 Front Controller

Introduce a front controller that publishes the application's functionality and transforms client service request into specific requests that can be invoked on the application's components.

— (Buschmann, Henney, and Schmidt, 2007)

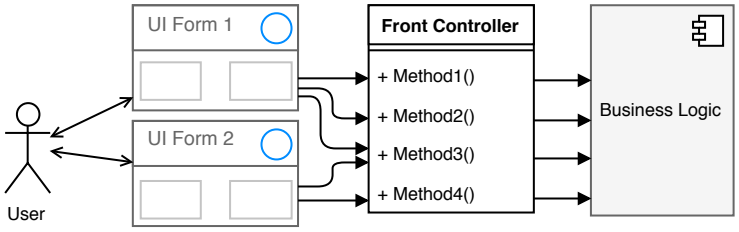

Figure 8.26: Front Controller

Participants and Bindings:

- **User**: Uses the UI Forms at *runtime*.
- **UI Forms**: Dependent on the Front Controller at *compile time*. Interacts with the User at *runtime*. Delegates the Requests to the Front Controller and gets controlled by it at *runtime*.
- **Front Controller**: Dependent on the UI Form Interface and the Business Logic Interface at *compile time*. Gets requests by the UI Form at *runtime*. Redirects those requests to the Business Logic Layer at *runtime*. Controlls the behaviour of the UI Forms at *runtime*.
- **Business Logic Layer**: Implements the Business Logic for the Application at *compile time*. Gets used by the Front Controller at *runtime*.

8.26 Application Controller

Encapsulate the application's workflow within a separate application controller. User-interface controllers use the application controller to determine the appropriate actions to invoke on application logic, as well as the correct view to display after the action has been executed.

— (Buschmann, Henney, and Schmidt, 2007)

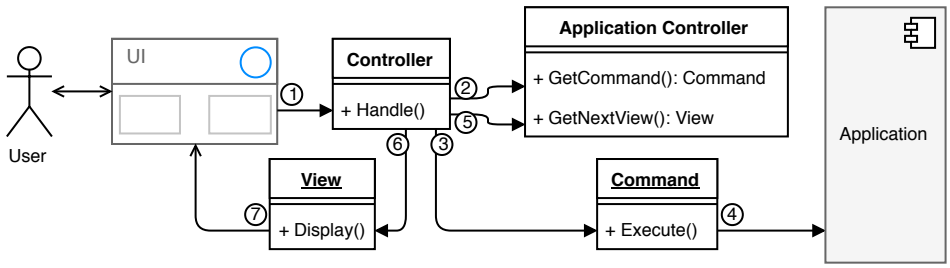

Figure 8.27: Application Controller

Participants and Bindings:

- **User**: Interacts with the UI at *runtime*.
- **UI**: Depends on the Controller Interface and the View Interface at *compile time*. Gets acted on by the User at *runtime* and requests the corresponding Controller to Handle the Event at *runtime*.
- **Controller**: Depends on the Application Controller, the Command-Interface and the View-Interface at *compile time*. Gets called by the UI to handle an interaction event at *runtime*. Calls the Application Controller to get the next command and executes the Command at *runtime*. Calls the Application Controller to get the next View and display's the View at *runtime*.
- **ApplicationController**: Dependent on the Commands and Views at *compile time*. Controls the application's workflow at *runtime* by returning the appropriate commands and view depending on the current state of the Application at *runtime*.
- **Command**: Depends on the Application at *compile time*. Gets created by the Application Controller and called by the

Controller at *runtime*. Acts on the Application at *runtime*.
- **Application**: Gets acted on by the Command at *runtime*.
- **View**: Depends on the UI Interface at *compile time*. Gets created by the Application Controller and displayed by the Controller at *runtime*.

8.27 Template View

Introduce a template view that predefines the view's structure and which contains placeholders for dynamic application data.

— (Buschmann, Henney, and Schmidt, 2007)

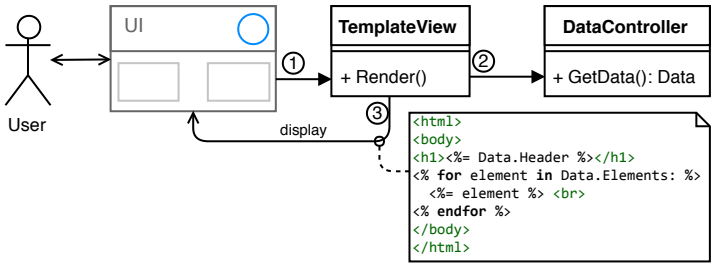

Figure 8.28: Template View

Participants and Bindings:

- **User**: Interacts with the UI at *runtime*.
- **UI**: Depends on the TemplateView Interface at *compile time*. Gets acted on by the User at *runtime* and routes the event to the TemplateView *runtime*.
- **Template View**: Depends on the DataController Interface at *compile time*. Gets the Data from the DataController at *runtime*. Runs a Text-Replacement Engine to render the view for the UI with the Data at *runtime*.
- **DataController**: Delivers the Data to the TemplateView at *runtime*.

8.28 Transform View

Introduce a transform view that walks the structure of the data received from the application, recognizes the data to display, and transforms it into a specific output format.

— (Buschmann, Henney, and Schmidt, 2007)

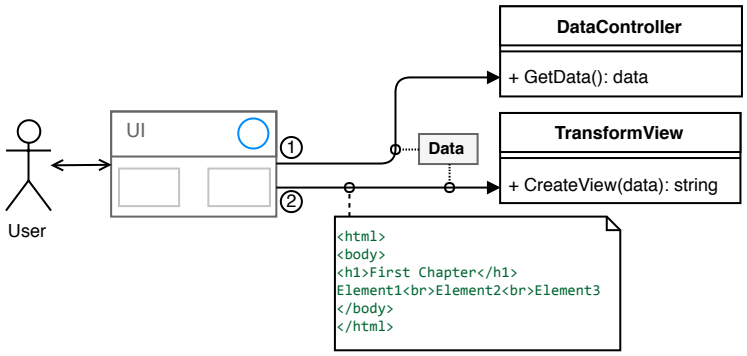

Figure 8.29: Transform View

Participants and Bindings:

- **User**: Interacts with the UI at *runtime*.
- **UI**: Depends on the DataController and the TransformView at *compile time*. Gets acted on by the User at *runtime*. Gets the data from the DataController and Transforms it into the output format via the TransformView at *runtime*.
- **TransformView**: Depends on the common data format at *compile time*. Transforms given data to the wished output format at *runtime*.
- **DataController**: Returns the Data needed for the view at *runtime*.

8.29 Firewall Proxy

Introduce a firewall proxy for the publicly accessible functionality of the application. This proxy enforces security policies on each client request to protect the components that implement this functionality from cyber attacks.
— (Buschmann, Henney, and Schmidt, 2007)

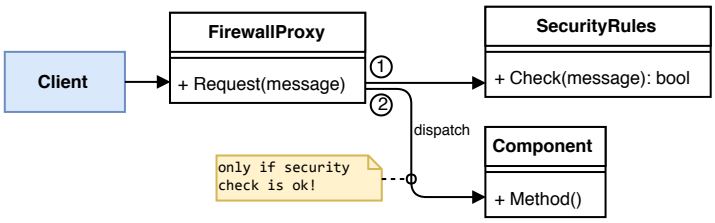

Figure 8.30: Firewall Proxy

Participants and Bindings:

- **Client**: Depends on the FirewallProxy interface at *compile time*. Calls the Request method at *runtime*.
- **Firewall Proxy**: Depends on the SecurityRules interface and the Component interface at *compile time*. Calls the SecurityRules to check the message and calls Components Methods at *runtime*.
- **Security Rules**: Checks the security of a message at *runtime*.
- **Component**: Implement some functionality at *compile time*. Gets called at *runtime* if the security is checked and ok.

8.30 Authorization

Assign access rights to each client that can send service requests to the security-sensitive subsystem and check these rights before executing any request on the subsystem.

— (Buschmann, Henney, and Schmidt, 2007)

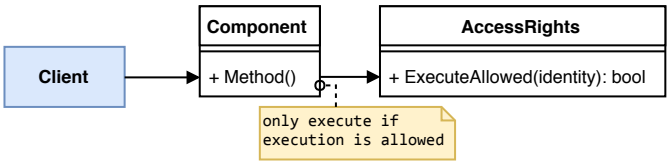

Figure 8.31: Authorization

Participants and Bindings:

- **Client**: Depends on the Component Interface at *compile time*. Calls the Method at *runtime*.
- **Component**: Depends on the AccessRights Interface at *compile time*. When called, checks the AccessRights first and if allowed it executes the Method at *runtime*.
- **AccessRights**: Checks the access rights of an identity at *runtime*.

8.31 Guarded Suspension

Instead of aborting the method, suspend its client thread so that other client threads can access the shared component safely and change the state of the methods' guard condition. If this state changes, resume the suspended thread so that the thread can try to continue the execution of the interrupted method.
— (Buschmann, Henney, and Schmidt, 2007)

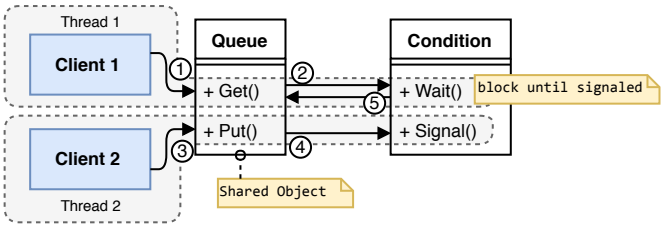

Figure 8.32: Guarded Suspension

Participants and Bindings:

- **Clients**: Depending on the Shared Object Interface at compile time. Calls the methods of the Shared Object at runtime. If the call cannot be answered immediately the Thread waits until the call may be answered.
- **Queue**: An example of an shared object. Depends on the Condition-Interface at *compile time*. If the queue is empty it calls the Wait method to suspend the calling thread until the Signal Method on the Condition is called at *runtime*. Calls the Signal-Method on the Condition when data is put into the queue at *runtime*.
- **Condition**: Calling the Wait()-Method suspend the calling Thread until the Signal()-Method is called by another Thread at *runtime*. With a call to Signal, it wakes up all waiting Threads to continue with their work at *runtime*.

8.32 Future

Immediately return a "virtual" data abject - called a future - to the client when it invokes a service. This future keeps track of the state of the service's concurrent computation and only provides a value to clients when the computation is complete.

— (Buschmann, Henney, and Schmidt, 2007)

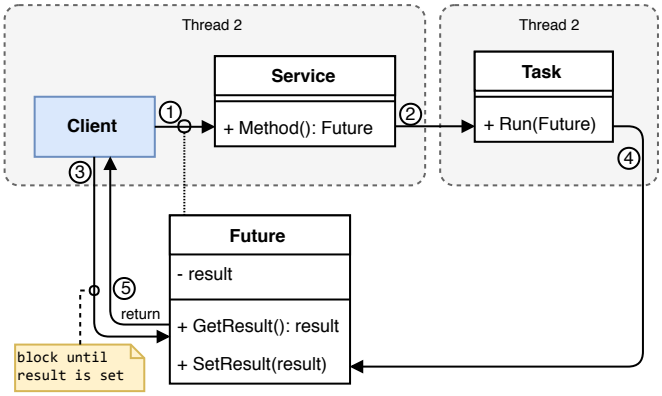

Figure 8.33: Future

Participants and Bindings:

- **Client**: Depends on the Future-Interface and the Service-Interface at *compile time*. Calls the Service and gets a Future at *runtime*. After that, gets the Result of the Future at *runtime*.
- **Service**: Creates a Future-Object at *runtime*. Creates a Task which runs in parallel in a separate Thread at *runtime* and sets the Future-Object. Returns the Future object to the caller which relates to the result of that Task at *runtime*.
- **Task**: Depends on the Future-Interface at *compile time*. Does some operation in parallel in an own thread at *runtime* and after completion sets the result on the Future at *runtime*.
- **Future**: Depends on the result-object at *compile time*. Is created by the Service and handled over to the Task and the Client at *runtime*. Waits until the Result has been set by the Task and returns it to the Client afterwards at *runtime*.

8.33 Copied Value

Define a value object type whose instances are copyable. When a value is used in communication with another thread, ensure that the value is copied.
— (Buschmann, Henney, and Schmidt, 2007)

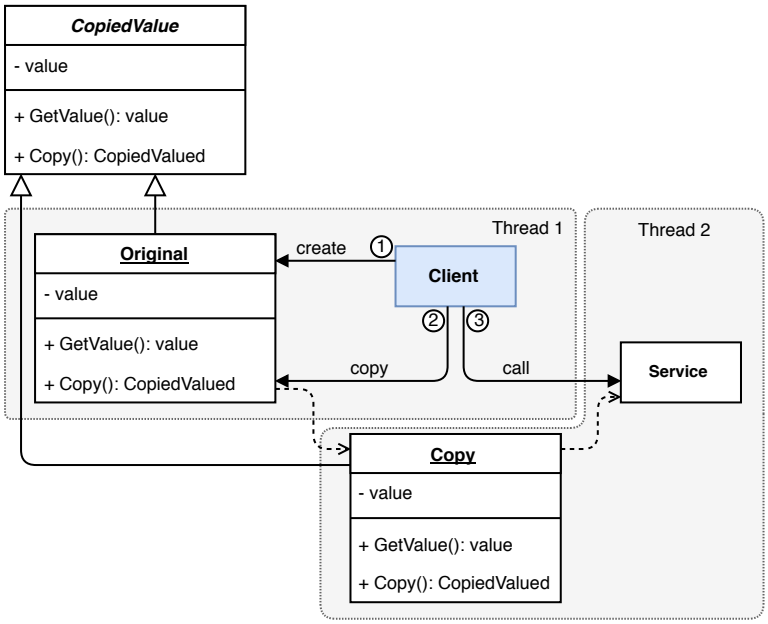

Figure 8.34: Copied Value

Participants and Bindings:

- **Client**: Depends on the CopiedValue-Interface at *compile time*. Creates and manages the Original at *runtime*. Copies the Original and calls the service with the Copy at *runtime*.
- **CopiedValue**: Implements a copied value object *compile time*.
- **Original**: The Original value created by the Client at *runtime*. Gets copied at *runtime*.
- **Copy**: The copy of the Original value to be handled over to the Service at *runtime*.
- **Service**: Dependent on the CopiedValue at *compile time*. Gets called with the Copy-Object at *runtime*.

8.34 Immutable Value

Define a value object type whose instances are immutable. The internal state of a value object is set at construction and no subsequent modifications are allowed.

— (Buschmann, Henney, and Schmidt, 2007)

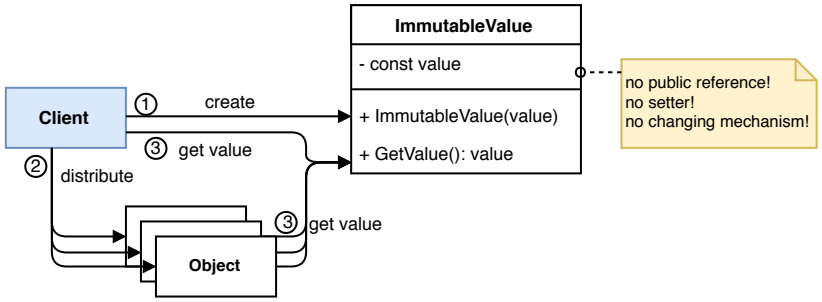

Figure 8.35: Immutable Value

Participants and Bindings:

- **Client**: Depends on the ImmutableValue and the corresponding Value-Object at *compile time*. Creates the ImmutableValue object and uses it at *runtime*.
- **ImmutableValue**: Depends on the Type of the underlying value object at *compile time*. Gets created and accessed at *runtime*.
- **Objects**: Depends on the ImmutableValue object and the underlying Value at *compile time*. Accesses it at *runtime*. Get shared references of the ImmutableValue at *runtime*, which all refer to the same ImmutableValue instance with the same underlying value object.

8.35 Double Dispatch

Pass the caller object to the receiver object as an extra argument. Within the receiver, call back the caller object to run caller-class dependent logic, passing the receiver as an additional argument, so that the caller can behave appropriately.

— (Buschmann, Henney, and Schmidt, 2007)

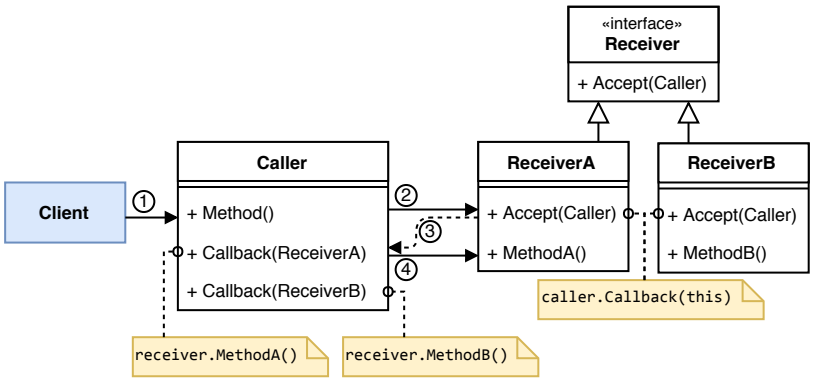

Figure 8.36: Double Dispatch

Participants and Bindings:

- **Client**: Dependent on the Caller at *compile time*. Calls the Method at *runtime*.
- **Caller**: Dependent on the Receiver-Interface and the Receiver-Implementations at *compile time*. Calls the Accept-Method on the Interface at *runtime*. The corresponding Callback method is called by the Receiver at *runtime* and calls the respective methods on the Receiver-Object at *runtime*.
- **Receiver-Interface**: Specifies the Interface for all Receivers at *compile time*.
- **Receiver**: Implements the Receiver-Interface at *compile time*. Depends on the Caller Object at *compile time*. Calls the respective overloaded Callback on the Caller Object at *runtime*.

8.36 Context Object

Represent the information and services in an object that encapsulates the required context. Provide this object to the operations, component, and layers that need the context.

— (Buschmann, Henney, and Schmidt, 2007)

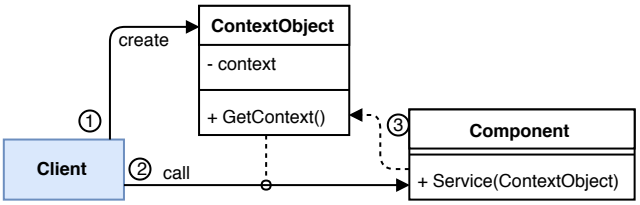

Figure 8.37: Context Object

Participants and Bindings:

- **Client**: Depends on the ContextObject and the Component at *compile time*. Creates the ContextObject at *runtime* and handles it over to the Component at *runtime* by calling the Service-Method.
- **ContextObject**: Defines a container for the context at *compile time*. Gets created by the Client at *runtime*.
- **Context**: Placeholder for the Context of the Service-Method Call. Is used by the Component at *runtime*.
- **Component**: Depends on the ContextObject at *compile time*. Uses it when the Service-Method is called at *runtime*.

8.37 Data Transfer Object

Bundle all data items that might be needed into a single data transfer object used for querying or updating attributes together.

— (Buschmann, Henney, and Schmidt, 2007)

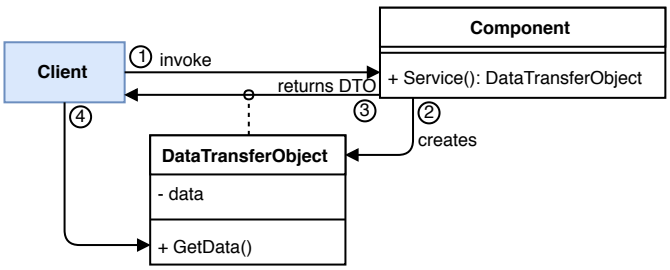

Figure 8.38: Data Transfer Object

Participants and Bindings:

- **Client**: Depends on the Component and the Data Transfer Object at *compile time*. Calls the Component at *runtime* and uses the Data contained in the returned Data Transfer Object at *runtime*.
- **Data Transfer Object**: Container for the Data specified at *compile time*. Is created by the Component at *runtime* and returned to the Caller of the Service-Method.s
- **Data**: Placeholder for the actual data returned by the Service-Method.
- **Component**: Depends on the Data Transfer Object at *compile time*. Creates it and sets the Data at *runtime*.

8.38 Execute-Around Object

Provide a helper class whose constructor implements the pre-sequence action and whose destructor the post-sequence action. Define an object of this class on the stack before the sequence of statements, and provide its constructor with the necessary arguments to perform the pre- and post-sequence actions.
— (Buschmann, Henney, and Schmidt, 2007)

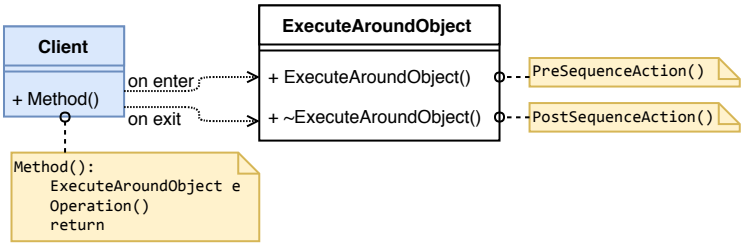

Figure 8.39: Execute-Around Object

Participants and Bindings:

- **Client**: Depends on the Execute-Around Object at *compile time* and uses it at *runtime*. Calls the Method at *runtime*.
- **Execute-Around Object**: Implements the PreSequenceAction and PostSequenceAction at *compile time*. Gets repeatedly created and destroyed on the stack at *runtime*.
- **Pre-Sequence-Action**: The Action which is executed at creation of the ExecuteAroundObject. Is executed at *runtime* before the actual operation of the client.
- **Post-Sequence-Action**: The action which is executed at the destruction of the ExecuteAroundObject. Is called at *runtime* either manually or at stack-unwinding after the method exits (but more important: after the operation was executed).

8.39 Null Object

Provide something for nothing: a class that conforms to the interface required of the object reference, implementing all of its methods to do nothing, or to return suitable default values. use an instance of this class, a so-called "null object", when the object reference would otherwise have been null.

— (Buschmann, Henney, and Schmidt, 2007)

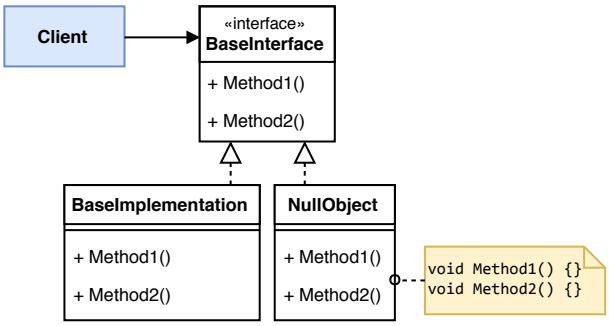

Figure 8.40: Null Object

Participants and Bindings:

- **Client**: Depends on the BaseInterface at *compile time*. Uses its implementations at *runtime*.
- **BaseInterface**: Specifies a base interface at *compile time*.
- **BaseImplementation**: Implements the BaseInterface at *compile time*. Gets used by the Client at *runtime*.
- **Null Object**: Implements the BaseInterface with empty methods at *compile time*. Gets used by the Client at *runtime*, without him recognizing that this is just a placeholder.

8.40 Declarative Component Configuration

Specify a separate declarative component configuration for each component that indicates to the runtime environment the system resources and services it needs to execute correctly, as well as how it will use these resources and services.

— (Buschmann, Henney, and Schmidt, 2007)

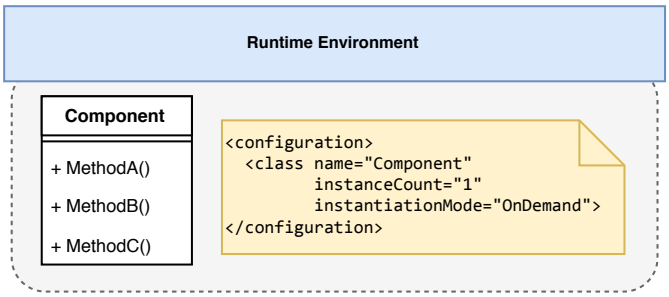

Figure 8.41: Declarative Component Configuration

Participants and Bindings:

- **Runtime Environment**: Reads the configuration and manages the described Component with the respective behavior at *runtime*.
- **Component**: Implements some functionality at *compile time*. Gets created and managed at *runtime* by the Runtime Environment.
- **Configuration**: Describes some behavioral aspects of the component for the Runtime Environment at *pre-compile-time*, *compile time*, *configuration time*, *startup time* or even *runtime* (depending on the implementation).

8.41 Methods for States

Implement state-dependent behavior as internal methods of the object, and use data structures to reference the methods that represent the behavior of a specific state.

— (Buschmann, Henney, and Schmidt, 2007)

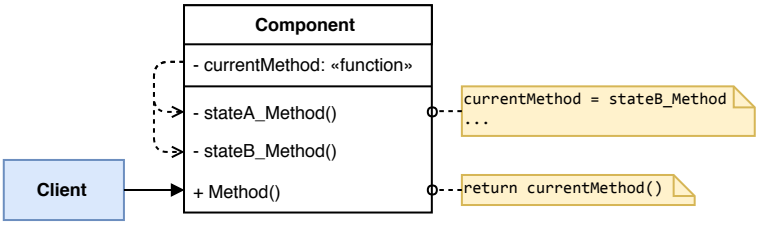

Figure 8.42: Methods for States

Participants and Bindings:

- **Client**: Depends on the Component at *compile time*. Calls the Method at *runtime*.
- **Component**: Implements the state methods at *compile time*. Gets called and changes its internal state at *runtime*.
- **State Methods**: Implement the different states inside the Component at *compile time*. Get called and switch at *runtime*.

8.42 Collections for States

Within the client, represent each state of interest by a separate collection that refers to all objects in that state.

— (Buschmann, Henney, and Schmidt, 2007)

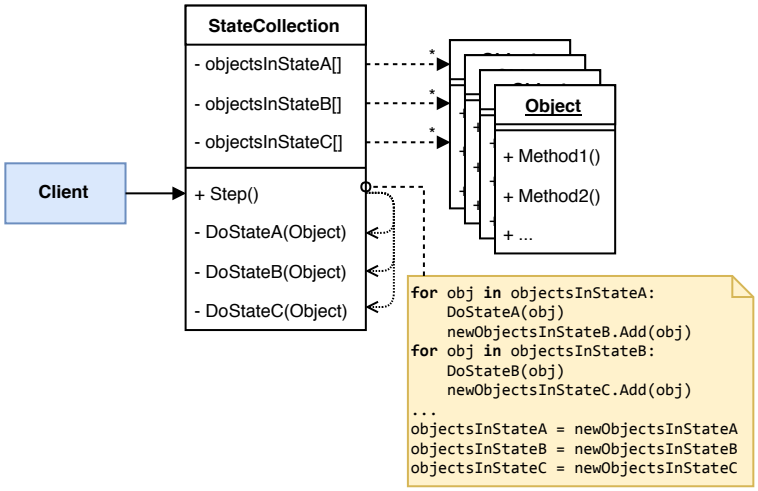

Figure 8.43: Collections for States

Participants and Bindings:

- **Client**: Depends on the StateCollection at *compile time*. Calls the Step Method at *runtime*.
- **StateCollection**: Depends on the Objects at *compile time*. Calls the Methods on the objects at *runtime* (depending in which state collection they are). Moves the objects between the Object-Lists to change their state.
- **Objects**: Are stored in the object-Lists at *runtime*. Get called at *runtime*.
- **Object-Lists**: Store the Objects which are in the respective State at *runtime*.
- **State-Methods**: Depends on the Object-Interface at *compile time*. Act on the Objects which are in the respective State at *runtime*.

8.43 Container

Define a container to provide the execution environment for a component that supports the necessary technical infrastructure to integrate components into application-specific usage scenarios, and on specific system platforms, without tightly coupling the components with the applications or platforms.

— (Buschmann, Henney, and Schmidt, 2007)

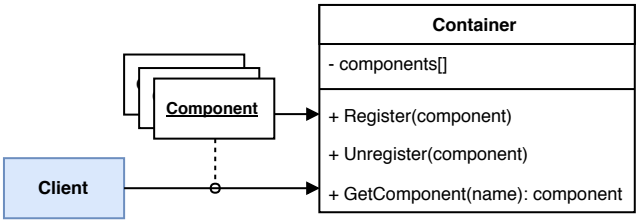

Figure 8.44: Container

Participants and Bindings:

- **Client**: Depends on the Container and the Components at *compile time*. Gets the needed Components from the Container at *runtime*. Uses the Components at *runtime*.
- **Container**: Stores a list of components at *runtime*. Gets requested by the Client at *runtime*.
- **Component**: Implements some functionality at *compile time*. Gets created and registered at the Container at *runtime*. Gets used by the Client at *runtime*.

8.44 Object Manager

Separate object usage from object lifecycle and access control. Introduce a separate object manager whose responsibility is to manage and maintain a set of objects.

— (Buschmann, Henney, and Schmidt, 2007)

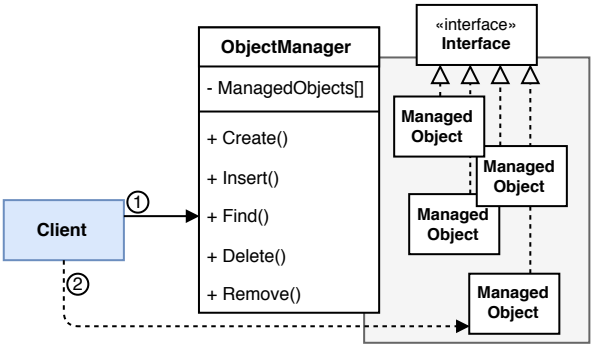

Figure 8.45: Object Manager

Participants and Bindings:

- **Client**: Depends on the ObjectManager at *compile time*. Depends on the object interfaces at *compile time*. Uses the ObjectManager to manage the objects at *runtime*, uses the objects at *runtime*.
- **Object Manager**: Manages a list of objects at *runtime*.
- **Managed Object**: Implements the interface at *compile time*. Gets created and destroyed by the ObjectManager at *runtime*. Gets used by the Client at *runtime*.

8.45 Virtual Proxy

Introduce a proxy for an object that does not currently exist in memory. The proxy may be able to handle simple requests, such as a query of the intended target objects' identifying key, but when more complete object behavior is needed, the actual target objects is created and initialized as needed.

— (Buschmann, Henney, and Schmidt, 2007)

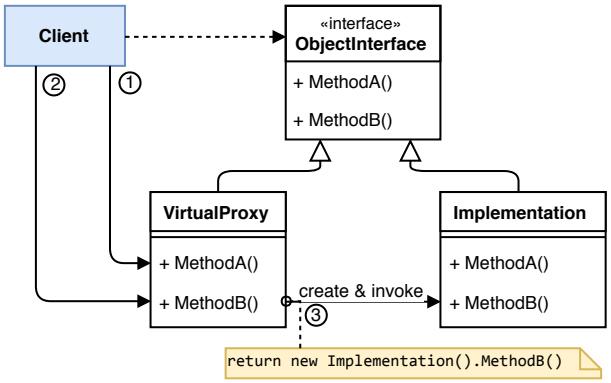

Figure 8.46: Virtual Proxy

Participants and Bindings:

- **Client**: Depends on the Object Interface at *compile time*. Uses the Virtual Proxy at *runtime* (and also indirectly the Implementation at *runtime*).
- **Virtual Proxy**: Implements the Object Interface at compile time. Depends on the Implementation at *compile time*. Handles some calls at *runtime* by itself, but delegates expensive calls to the Implementation at *runtime*.
- **Object Interface**: Specifies the interface for the Implementation and Virtual Proxy at *compile time*.
- **Implementation**: Implements the Object Interface at *compile time*. Gets created and called by the Virtual Proxy on demand at *runtime*.

8.46 Lifecycle Callback

Define key lifecycle events as callbacks in an interface that is supported by framework objects. The framework uses the callbacks to control the objects's lifecycle explicitly.

— (Buschmann, Henney, and Schmidt, 2007)

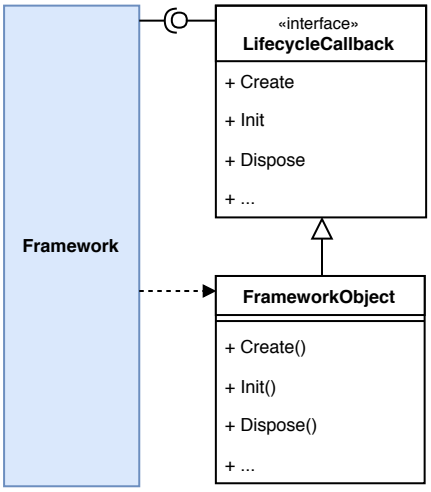

Figure 8.47: Lifecycle Callback

Participants and Bindings:

- **Framework**: Depends on the LifecycleCallback-Interface at *compile time*. Calls the respective methods on the FrameworkObjects at *runtime*.
- **LifecycleCallback**: Specifies the needed interface in the Framework for the FrameworkObjects at *compile time*.
- **FrameworkObject**: Implements the LifycycleCallback-Interface at *compile time*. Gets used by the Framework at *runtime*.

8.47 Activator

Minimize resource consumption by activating services on demand and deactivating services when they are no longer accessed by clients. Use proxies to decouple client access transparently from service behavior and lifecycle management.

— (Buschmann, Henney, and Schmidt, 2007)

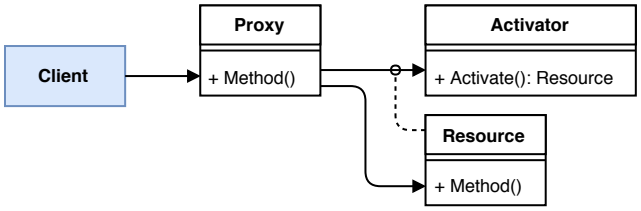

Figure 8.48: Activator

Participants and Bindings:

- **Client**: Depends on the Proxy-Interface at *compile time*. Calls the method at *runtime*.
- **Proxy**: Implements the Proxy-Interface at *compile time*. Depends on the Activator-Interface and the Resource at *compile time*. Calls the Activator at *runtime* and uses the Resource at *runtime*.
- **Activator**: Depends on the Resource at *compile time*. Gets called by the Proxy at *runtime* and returns a Resource at *runtime*.
- **Resource**: Get activated at *runtime* by the Activator.

8.48 Automated Garbage Collection

Define a garbage collector that identifies which objects are no longer referenced by live objects in the application,m and reclaims their memory. The garbage collector performs the identification and reclamation automatically and transparently.

— (Buschmann, Henney, and Schmidt, 2007)

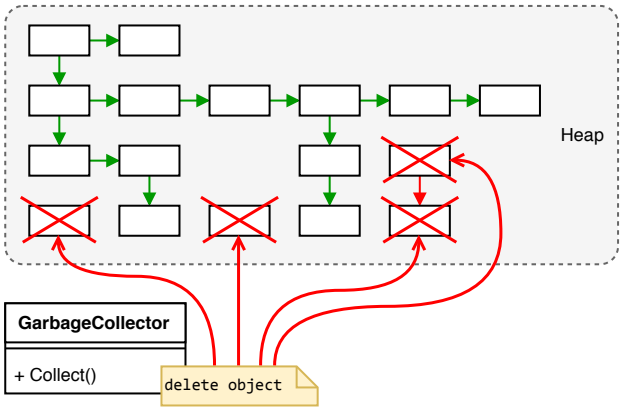

Figure 8.49: Automated Garbage Collection

Participants and Bindings: The **Garbage Collector** depends on the structure of the Heap and the object reference method at *compile time*. Periodically checks the object reference graph or reference count on the heap and deletes all objects which are not referenced at *runtime*.

8.49 Disposal Method

Encapsulate the concrete details of objects disposal withing a dedicated method, instead of letting clients delete or discard objects themselves.

— (Buschmann, Henney, and Schmidt, 2007)

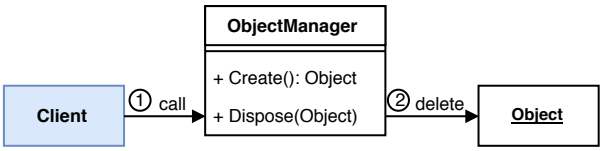

Figure 8.50: Disposal Method

Participants and Bindings:

- **Client**: Depends on the ObjectManager at *compile time*. Calls the Dispose-Method at *runtime*.
- **ObjectManager**: Specifies the Interface for managing Objects (especially disposing object) at *compile time*. Upon calling the Dispose method it deletes the given object at *runtime*.

8.50 Database Access Layer

Introduce a separate database access layer between the application and the relational database that provides a stable object-oriented data-access interface for application use, backed by an implementation that is database-centric.

— (Buschmann, Henney, and Schmidt, 2007)

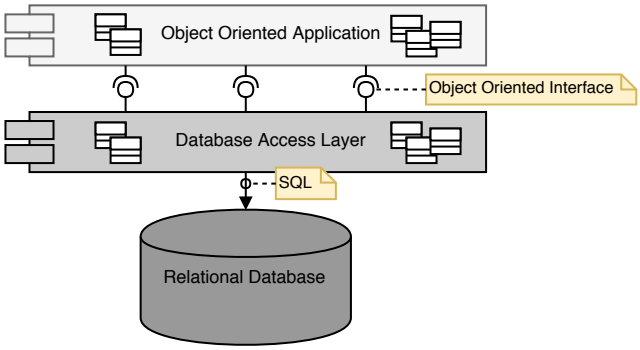

Figure 8.51: Database Access Layer

Participants and Bindings:

- **Object Oriented Application**: Depends on the Interfaces provided by the Database Access Layer at *compile time*. Manipulates objects in an object oriented way at *runtime*.
- **Database Access Layer**: Specifies an object oriented Interface for the Application at *compile time*. Implements accessing the Relational Database by translating and reflecting requests to SQL at *compile time*, although dynamic approaches (like hibernate, or LINQ to SQL) could defer the creation of the SQL query to *runtime*. Executes those requests at *runtime*.
- **Relational Database**: Represents the data as relational tables in normal form which can be manipulated with SQL queries at *runtime*. The database has its *own development lifecycle* and therefore can be established at any time of the application lifecycle, but the latest binding time is actually at *runtime* when an SQL query is applied to it.

8.51 Data Mapper

Introduce a data mapper for each type of persistent application objects whose responsibility is to transfer data from the object to the relational database, and vice-versa.

— (Buschmann, Henney, and Schmidt, 2007)

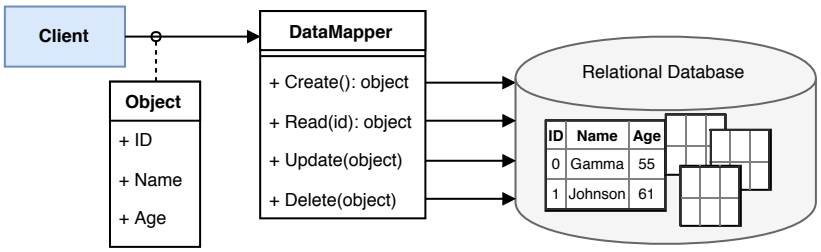

Figure 8.52: Data Mapper

Participants and Bindings:

- **Client**: Depends on the Object and the DataMapper at *compile time*. Manipulates the Object at *runtime* and handles it over to the DataMapper to persist the changes at *runtime*.
- **Object**: Specifies an object oriented way to access the information of an object at *compile time*. Gets manipulated at *runtime* and persisted by the DataMapper into the database at *runtime*.
- **Data Mapper**: Depends on the Object and communication form to the Relational Database (e.g. SQL) at *compile time*. Also depends on the data structure inside the Relational Database at *compile time* (but this could be deferred to *runtime* by a dynamic approach like hibernate or LINQ to SQL). Gets called by the Client at *runtime* and translates this calls to SQL queries which are applied to the Database at *runtime*.
- **Relational Database**: Represents the data as tables in normal form which can be manipulated with SQL queries at *runtime*. The database has its *own development lifecycle* and therefore can be established at any time of the application lifecycle, but the latest binding time is actually at *runtime* when an SQL query is applied to it.

8.52 Row Data Gateway

Wrap the data structures and their database access code within row data gateways whose internal structure looks exactly like a database record, but which offer a representation-independent data access interface to clients.

— (Buschmann, Henney, and Schmidt, 2007)

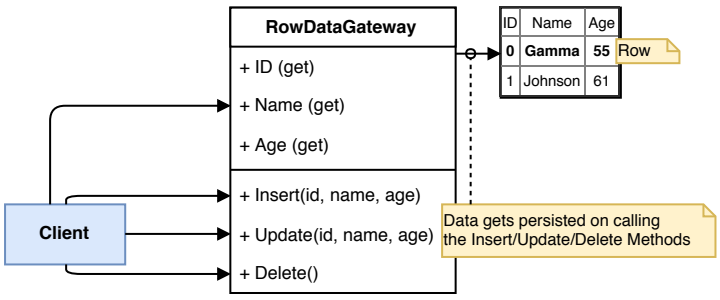

Figure 8.53: Row Data Gateway

Participants and Bindings:

- **Client**: Depends on the RowDataGateway Interface at *compile time*. Calls the Methods at runtime to manipulate data rows at *runtime*.
- **RowDataGateway**: Depends on the Database communication method and the data structure (data table) inside the database at *compile time* (*or later* when using dynamic approaches like configurations as in hibernate or LINQ to SQL).

8.53 Table Data Gateway

Wrap the database access code for a specific database table within a specialized table data gateway, and provide it with an interface that allows applications to work on domain-specific data collections.

— (Buschmann, Henney, and Schmidt, 2007)

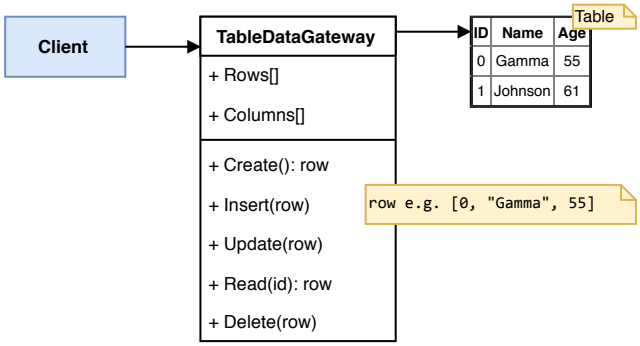

Figure 8.54: Table Data Gateway

Participants and Bindings:

- **Client**: Depends on the TableDataGateway Interface at *compile time*. Calls the Methods at runtime to manipulate data rows at *runtime*.
- **TableDataGateway**: Depends on the Database communication method and the data structure (data table) inside the database at *compile time* (*or later* when using dynamic approaches like configurations as in hibernate or LINQ to SQL).

8.54 Active Record

Encapsulate the data, the corresponding database access code, and the data-centered domain behavior in active record objects that offer a domain-specific interface to clients.

— (Buschmann, Henney, and Schmidt, 2007)

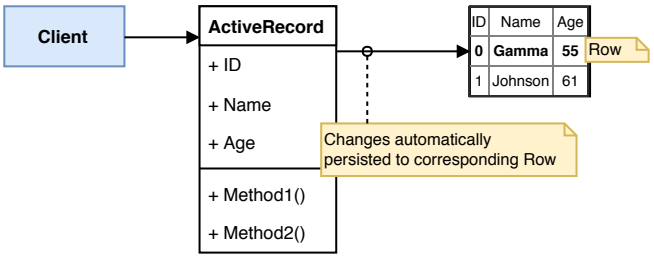

Figure 8.55: Active Record

Participants and Bindings:

- **Client**: Depends on the Active Record Interface at *compile time*. Manipulates the object directly at *runtime*.
- **Active Record**: Depends on the Database communication method and the Table-Structure at *compile time* (or later with dynamic approaches like hibernate or LINQ to SQL). Gets changed by the Client at *runtime* and afterwards persists these changes automatically into the Database also at *runtime*.

8.55 Duplicate Patterns

In POSA4 some already described patterns where also depicted. Here is a list of the duplicate patterns and where to find them in the previous books:

- **Layers** → see POSA1 Layers (60)
- **Command Processor** → see POSA1 Command Processor (71)
- **Component Configurator** → see POSA2 Component Configurator (79)
- **Model-View-Controller** → see POSA1 Model-View-Controller (65)
- **Presentation-Abstraction-Control** → see POSA1 Presentation-Abstraction-Control (66)
- **Microkernel** → see POSA1 Microkernel (68)
- **Reflection** → see POSA1 Reflection (69)
- **Pipes and Filters** → see POSA1 Pipes and Filters (61)
- **Blackboard** → see POSA1 Blackboard (62)
- **Publisher-Subscriber** → see GOF Observer (51)
- **Broker** → see POSA1 Broker (63)
- **Reactor** → see POSA2 Reactor (82)
- **Proactor** → see POSA2 Proactor (83)
- **Acceptor-Connector** → see POSA2 Acceptor-Connector (85)
- **Asynchronous Completion Token** → see POSA2 Asynchronous Completion Token (84)
- **Extension Interface** → see POSA2 Extension Interface (81)
- **Proxy** → see GOF Proxy (53)
- **Facade** → see GOF Facade (44)
- **Iterator** → see GOF Iterator (48)
- **Encapsulated Implementation** → see GOF Proxy (53)
- **Whole-Part** → see GOF Composite (42)
- **Composite** → see GOF Composite (42)
- **Master-Slave** → see POSA2 Master-Slave (70)
- **Half-Sync/Half-Async** → see POSA2 Half-Sync/Half-Async (92).
- **Leader/Followers** → see POSA2 Leader/Followers (93)
- **Active Object** → see POSA2 Active Object (90)
- **Monitor Object** → see POSA2 Monitor Object (91)
- **Thread-Safe Interface** → see POSA2 Thread-Safe Inter-

face (88)
- **Double-Checked Locking** → see POSA2 Double-Checked Locking (89)
- **Strategized Locking** → see POSA2 Strategized Locking (87)
- **Scoped Locking** → see POSA2 Scoped Locking (86)
- **Thread-Specific Storage** → see POSA2 Thread-Specific Storage (94)
- **Observer** → see GOF Observer (51)
- **Mediator** → see GOF Mediator (49)
- **Command** → see GOF Command (41)
- **Memento** → see GOF Memento (50)
- **Bridge** → see GOF Bridge (38)
- **Object Adapter** → see GOF Adapter (37)
- **Chain of Responsibility** → see GOF Chain of Responsibility (40)
- **Interpreter** → see GOF Interpreter (47)
- **Interceptor** → see POSA2 Interceptor (80)
- **Visitor** → see GOF Visitor (58)
- **Decorator** → see GOF Decorator (43)
- **Template Method** → see GOF Template Method (57)
- **Strategy** → see GOF Strategy (56)
- **Lookup** → see POSA3 Lookup (96)
- **Task Coordinator** → see POSA3 Coordinator (102)
- **Resource Pool** → see POSA3 Pooling (101)
- **Resource Cache** → see POSA3 Caching (100)
- **Lazy Acquisition** → see POSA3 Lazy Acquisition (97)
- **Eager Acquisition** → see POSA3 Eager Acquisition (98)
- **Partial Acquisition** → see POSA3 Partial Acquisition (99)
- **Evictor** → see POSA3 Evictor (104)
- **Leasing** → see POSA3 Leasing (103)
- **Counting Handle** → see POSA1 Counted Pointer (75)
- **Abstract Factory** → see GOF Abstract Factory (36)
- **Builder** → see GOF Builder (39)
- **Factory Method** → see GOF Factory Method (45)
- **Wrapper Facade** → see POSA2 Wrapper Facade (78)
- **Objects for States** → see GOF State (55)

9 Conclusion

In this thesis, the importance of binding time as an additional supportive criterion for selecting the right design pattern was demonstrated. If the context and requirements are quite similar for multiple patterns, binding time considerations can help choosing the best fitting pattern for a given problem, as is shown in the binding time scenarios in chapter 3 Finding the Right Pattern. The program timeline (see chapter 2.3) defines the phases of time for the whole lifetime of a program. Binding may happen in any of these phases from early architectural decisions until late dynamic bindings and configurations at runtime. The later the binding in this timeline happens, the more flexible an application is, but this flexibility comes with the price of complexity and higher costs which is often not desirable.

Binding time considerations can help out of this dilemma, by making clear statements which bindings are needed at which phase of the program timeline. In such a way, the design patterns which support the requirements and introduce the least complexity, can be selected amongst other competing alternatives.

Bibliography

Alexander, Christopher (1979). *The Timeless Way of Building*, p. 552. ISBN: 0195024028. DOI: 10.1080/00918360802623131 (cit. on p. 5).

Beck, Kent (1996). *SmallTalk Best Practice Patterns*. New Jersey: Prentice Hall, p. 240. ISBN: 978-0134769042 (cit. on p. 8).

Booch, Grady, Ivar Jacobson, and James Rumbaugh (1999). *The Unified Modeling Language Reference Manual*, p. 550. ISBN: 020130998X (cit. on p. 9).

Burch, Carl (2012). *Binding time*. (Visited on 04/22/2016) (cit. on p. 22).

Buschmann, Frank, Kevlin Henney, and Douglas Schmidt (2007). *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*, p. 639. ISBN: 9780470059029 (cit. on pp. 2, 7, 22, 105, 108–161).

Buschmann, Frank, Regine Meunier, et al. (1996). "Pattern-Oriented Software Architecture Volume 1: A system of patterns." In: *John Wiley&Sons* 1, p. 476. ISSN: 0007-1250. DOI: 10.1192/bjp.108.452.101 (cit. on pp. 2, 6, 22, 59–63, 65, 66, 68–75).

C2 (2016). *Pattern Forms*. (Visited on 09/25/2016) (cit. on p. 4).

Capilla, Rafael and Jan Bosch (2013). "Binding Time and Evolution." In: *Systems and Software Variability Management*. Berlin, Heidelberg: Springer Berlin Heidelberg. Chap. 4, pp. 57–73. ISBN: 9783642365836. DOI: 10.1007/978-3-642-36583-6_4 (cit. on pp. 11, 23).

Coplien, James O. (1995). *Pattern Languages of Program Design*. Pattern Languages of Program Design. Addison-Wesley, p. 576. ISBN: 978-0201607345 (cit. on pp. 8, 23).

Coplien, James O. and John Vlissides (1996). *Pattern Languages of Program Design 2*. Pattern Languages of Program Design.

Addison-Wesley, p. 624. ISBN: 978-0201895278 (cit. on pp. 8, 23).

Foote, Brian, Neil Harrison, and Hans Rohnert (1999). *Pattern Languages of Program Design 4*. Pattern Languages of Program Design. Addison-Wesley, p. 784. ISBN: 978-0201433043 (cit. on p. 23).

Fritsch, Claudia, Andreas Lehn, and Thomas Strohm (2002). "Evaluating Variability Implementation Mechanisms." In: *Proceedings of International Workshop on Product Line Engineering*, pp. 59–64 (cit. on pp. 11, 22).

Gamma, Erich et al. (1995). *Design Patterns - Elements of Reusable Object-Oriented Software*. 1st ed. Boston, USA: Addison-Wesley, p. 395. ISBN: 978-0201633610 (cit. on pp. 2, 3, 6, 22, 23, 31, 32, 35–58).

Jaring, Michel and Jan Bosch (2002). "Representing Variability in Software Product Lines: A Case Study." In: *Software Product Lines (LNCS)*. LNCS. Vol. 2379. Berlin Heidelberg: Springer-Verlag, pp. 15–36. ISBN: 3540439854. DOI: 10.1007/3-540-45652-X_2 (cit. on p. 11).

Kandt, Kirk (2003). "Software Design Principles and Practices." In: *Proceedings of the ICSE'03*, pp. 1–9 (cit. on p. 20).

Kircher, Michael and Prashant Jain (2004). *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. Vol. 3. ISBN: 0470845252 (cit. on pp. 2, 6, 22, 95–104).

Kreiner, Christian (2013). "A binding time guide to creational patterns." In: *Proceedings of the 18th European Conference on Pattern Languages of Program - EuroPLoP '13*. New York, New York, USA: ACM Press, pp. 1–10. ISBN: 9781450334655. DOI: 10.1145/2739011.2739025 (cit. on pp. 10, 22).

Krisper, Michael and Christian Kreiner (2016). "Describing Binding Time in Software Design Patterns." In: *EuroPLoP '16* (cit. on pp. 12, 22).

Krueger, Charles W (2004). "Towards a Taxonomy for Software Product Lines." In: *Software Product-Family Engineering (LNCS)*. Ed. by Frank van der Linden. Vol. 3014. Berlin Heidelberg: Springer Verlag, pp. 323–331. ISBN: 978-3-540-24667-1. DOI: 10.1007/978-3-540-24667-1_25 (cit. on pp. 11, 23).

Linden, Frank van der, Klaus Schmid, and Eelco Rommes (2007). *Software Product Lines in Action*. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-540-71436-1. DOI: 10.1007/978-3-540-71437-8 (cit. on p. 11).

Manolescu, Dragos, Markus Voelter, and James Noble (2006). *Pattern Languages of Program Design 5*. Pattern Languages of Program Design. Addison-Wesley Professional, p. 624. ISBN: 978-0321321947 (cit. on p. 23).

Marquardt, Klaus (2005). "Indecisive generality." In: *Proceedings of the EuroPLoP'05*, p. 16. ISBN: 978-387940805-4 (cit. on p. 20).

Martin, Robert C., Dirk Riehle, and Frank Buschmann (1007). *Pattern Languages of Program Design 3*. Pattern Languages of Program Design. Addison-Wesley Professional, p. 656. ISBN: 978-0201310115 (cit. on p. 23).

Myllymäki, Tommi (2001). *Variability Management in Software Product Lines*. Tech. rep. Tampere: Tampere University of Technology, Software Systems Laboratory, ARCHIMEDES, p. 49 (cit. on pp. 11, 22).

Nord, Joe (2011). "History of DLL Hell and why it will repeat itself" (cit. on p. 19).

Schmidt, Douglas et al. (2000). *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Vol. 2, pp. 1–482. ISBN: 0471606952. DOI: 10.1080/13581650120105534 (cit. on pp. 2, 6, 22, 77–94).

Svahnberg, Mikael, Jilles van Gurp, and Jan Bosch (2005). "A taxonomy of variability realization techniques." In: *Software: Practice and Experience* 35.8, pp. 705–754. ISSN: 0038-0644. DOI: 10.1002/spe.652 (cit. on p. 22).

Wirth, Niklaus (1996). *Compiler Construction*. Lecture Notes in Computer Science. Addison Wesley, p. 176. ISBN: 978-0201403534. DOI: 10.1007/11688839. arXiv: arXiv:1011.1669v3 (cit. on p. 18).