FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Transformation patterns for a reactive application

Bruno Miguel Mendonça Maia

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Hugo Sereno Ferreira, Assistant Professor

July 25, 2014

Transformation patterns for a reactive application

Bruno Miguel Mendonça Maia

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: João Cardoso, Ph.D.

External Examiner: Ricardo Machado, Ph.D.

Supervisor: Hugo Sereno Ferreira, Ph.D.

July 25, 2014

Abstract

Today's Internet is a prime example of a large-scale, highly engineered, yet highly complex software system. In the recent past, a typical large system would have tens of server nodes, its response times would be measured in the order of seconds, maintenance downtime would last for hours per month and data volume would be measured in the order of GBs. Nowadays a large system has thousands of server nodes, users expect sub-second response time, there should be no maintenance downtime and the system should be available at all times, and we're entering the age of *Big Data* with data volumes reaching the order of TBs and going into PBs and are too large and complex to manipulate with standard tools.

Such characteristics suggest new types of architectures that are able to cope with an increasing demand in both quantity and performance. But if building such systems from the ground is a demanding task, what to say about *evolving* legacy systems towards the same requirements?

This dissertation assumes that the concept of reactive applications, that are event-driven, scalable, resilient and responsive, is a good solution for the type of systems aforementioned and, as such, we explore the process of incrementally *refactoring* an existing system following a non-reactive architecture into one that does through a set of transformation patterns.

We then proceed to validate the patterns by providing known uses in the industry and finally, analyze the case study, Blip, and pinpoint parts of the system where we can apply each transformation pattern.

Resumo

Atualmente, a Internet é um bom exemplo de sistemas de *software* de grande escala, altamente sofisticados e no entanto, extremamente complexos. No passado recente, um típico grande sistema seria composto por dezenas de servidores, o seu tempo de resposta seria medido em segundos, o tempo de manutenção duraria horas cada mês e o volume de data processado seria medido em GBs. Hoje em dia, um grande sistema é composto por centenas de servidores, os utilizadores esperam um tempo de resposta na ordem dos milissegundos, deve estar disponível permanentemente e sem tempos de manutenção, e o volume de data processado está a chegar aos TBs e a caminhar para os PBs sendo demasiado grande e complexa para manipular com ferramentas convencionais.

Estas características sugerem novos tipos de arquiteturas que consigam lidar com a demanda crescente quer em quantidade, quer em desempenho. Mas se desenvolver sistemas desta vertente de raiz é uma tarefa exigente, que dizer sobre *evoluir* sistemas legados em direção aos mesmos requisitos?

Esta dissertação assume que o conceito de sistemas reativos, que são guiados por eventos, escaláveis, resistentes e responsivos, é uma boa solução para o tipo de sistemas mencionados previamente e, como tal, explora o conceito de *refatorizar* incrementalmente um sistema existente baseado numa arquitetura não-reativa para uma que seja através de um conjunto de padrões de transformação.

Procedemos então para validar os padrões fornecendo utilizações conhecidas na industria e finalmente, analisamos o caso de estudo, Blip, e identificamos partes do sistema onde possamos aplicar cada padrão de transformação.

Acknowledgements

I'd like to thank Hugo for all the guidance and philosophical talks, and João for all the insights and eye openers.
But most of all, I'd like this to be in memory of my dear friend, Eduardo Jesus, who I'm sure would have done a 10 times better job.

Bruno Maia

"If all you have is a hammer, everything looks like a nail"

Abraham Maslow

Contents

1 Introduction **1**
1.1 Context and Framing . . . 1
1.2 Motivation . . . 2
1.3 Dissertation scope and Case Study . . . 2
1.4 Objectives and Contributions . . . 2
1.5 Dissertation Structure . . . 3

2 Background **5**
2.1 Reactive Systems . . . 5
2.1.1 Characteristics . . . 6
2.1.2 Conclusion . . . 10
2.2 Patterns . . . 10
2.3 Refactoring . . . 12
2.4 Refactoring to Patterns . . . 13
2.5 Epistemology of patterns and refactorings . . . 14
2.6 Case study . . . 16
2.7 Summary . . . 17

3 State of the art **19**
3.1 Reactive Paradigm . . . 19
3.1.1 Primitives . . . 20
3.1.2 Abstractions . . . 24
3.1.3 Immutability . . . 24
3.1.4 Blocking vs Non-blocking . . . 25
3.2 Supporting technologies . . . 25
3.2.1 Reactive Extensions . . . 25
3.2.2 RxJava . . . 26
3.2.3 Finagle . . . 26
3.2.4 Akka . . . 26
3.3 Summary . . . 27

4 Problem definition **29**
4.1 Overview . . . 29
4.2 Epistemological stance . . . 31
4.3 Main thesis . . . 31
4.4 Approach . . . 33

5 Patterns 35
5.1 Anatomy 35
5.2 Forces 36
5.3 Tranformations 37
5.3.1 Synchronous to Asynchronous processing 39
5.3.2 Blocking I/O to Non-blocking I/O 42
5.3.3 Blocking Servlet to Asynchronous Servlet 44
5.3.4 Parallelize and aggregate 46
5.3.5 Add Circuit Breaker 49
5.3.6 Pull to Push Model 52
5.3.7 Monolithic to Microservice based Architecture 56
5.3.8 Add event streams 59
5.3.9 Minimize Shared State 63
5.3.10 Add Supervision 64
5.3.11 Retry on Failure 67
5.4 Summary 70

6 Case study 71
6.1 System Description 71
6.1.1 Cougar 71
6.1.2 Mantis 73
6.1.3 Wall 73
6.2 Main challenges 73
6.3 Pattern application 74
6.3.1 Ongoing transformations 76
6.4 Summary 77

7 Conclusions and Future Work 79
7.1 Summary 79
7.2 Conclusion and Main Contributions 80
7.3 Future Work 80

References 83

A Refactoring catalogs 89

List of Figures

2.1 Reactive systems properties dependencies . . . 6
2.2 Amdahl's law on theoretical maximum speedup using parallel computing . . 8
2.3 Number of Betfair's unique visitors from the US . . . 17

3.1 Comparison between Iterable and Observable syntax . . . 22
3.2 Example of several underlying implementations of Observables . . . 23
3.3 Composition flow diagram . . . 24

4.1 Example of linear vs non-linear scalability . . . 30
4.2 Approach for creating the catalog of patterns . . . 33

5.1 Relation between the different forces . . . 36
5.2 Relation between the different patterns . . . 37
5.3 Sequential execution of tasks . . . 47
5.4 Parallelized execution of tasks . . . 47
5.5 Diamond execution of tasks . . . 48
5.6 Circuit breaker diagram . . . 50
5.7 Circuit breaker state diagram . . . 51
5.8 Circuit breaker control sequence example . . . 51
5.9 Pull model diagram . . . 53
5.10 Push model diagram . . . 54
5.11 Sequence diagram comparison of a chat application using different models . 55
5.12 Hybrid Push model diagram . . . 56
5.13 Comparison between a monolithic and a microservice architecture . . . 58
5.14 Marble diagram of the mouse events composition . . . 61
5.15 Example of a Future to Observable conversion . . . 62
5.16 Flow diagram of possible components response . . . 66
5.17 Component hierarchy with supervision . . . 66
5.18 Example of a faulty communication with a retry strategy . . . 69

6.1 Backend system underlying architecture based on Cougar . . . 72
6.2 Frontend system based on Mantis . . . 73
6.3 Service displacement with monolithic architecture at Betfair . . . 76
6.4 Service displacement with microservice architecture at Betfair . . . 77

List of Sources

3.1 Example of creating and handling a Future in Scala 20
3.2 Example of creating and handling a Future by using a Promise in Scala . . . 21
3.3 Example of creating and composing of Observables in Scala 22

5.1 Syncronous composition of tasks 41
5.2 Asynchronous sequential composition of tasks using Futures 41
5.3 Examples of Synchronous and asynchronous approaches for file reading . . 44
5.4 Example of the parallelization and aggregation of tasks 48
5.5 Composition of mouse events using Observables 61
5.6 Example of a autocomplete search input implementation using Observables 62

List of Tables

3.1 Duality between the Synchronous and Asynchronous models 19

4.1 Example of system requirements in the past and now. 29

5.1 Classification of the Transformation patterns' main influence within the reactive traits. 70

6.1 Technologies' characteristics valuation for Betfair 74

A.1 Catalog of *refactorings*. 89
A.2 Catalog of *Refactoring to patterns*. 90

Abbreviations

AJAX	Asynchronous JavaScript and XML
AIO	Asynchronous Input/Output
API	Application programming interface
CB	Circuit Breaker
CPU	Central processing unit
DSL	Domain specific language
FY	Fiscal Year
IDE	Integrated development environment
HTTP	Hypertext Transfer Protocol
IO	Input/Output
TB	TeraByte
JDBC	Java Database Connectivity
PB	PetaByte
REST	Representational State Transfer
RPC	Remote procedure call

Chapter 1

Introduction

1.1 CONTEXT AND FRAMING

Today's Internet is a prime example of a large-scale, highly engineered, yet highly complex system. It is characterized by an enormous degree of heterogeneity any way one looks and continues to undergo significant changes over time [WG02]. Where in the past, large systems would have tens of servers, seconds of response time, hours of offline maintenance and gigabytes of data, today's systems are deployed on everything from mobile devices to cloud-based clusters running thousands of multicore processors. Users expect near-instantaneous — in the order of millisecond or even microsecond — response times and 100% uptime while data needs are expanding into the petabytes.

Initially the domain of innovative internet-driven companies like Google [Goo98] or Twitter [Twi06], these application characteristics are now surfacing in a wide set of industries. Finance and telecommunication were the first to adopt new practices to satisfy the new requirements, and others have followed [Arm97].

New requirements demand new technologies. Where previous solutions have emphasized managed servers and containers, recent ones make use of the *Pull* model — where clients continuously poll the server (wasting resources), and servers would be *busy-waiting* — would not enter an idle state even in the absence of activity,scaling was achieved through buying larger servers and concurrent processing via multi-threading.

But now a *"new"* architecture *trend* has evolved to let developers conceptualize and build applications that satisfy today's demands. We call these reactive systems. This architecture allows developers to build systems that are *event-driven* (adopting the *Push* model), scalable, resilient and responsive: delivering highly responsive user experiences with a real-time feel, backed by a scalable and resilient application stack, ready to be deployed on multicore and cloud computing architectures.

While the underlying concepts of this architecture date more than one decade, only now, with the shift in system's requirements, it is being popularized. And if, systems that need to scale must adapt.

1.2 MOTIVATION

Companies scale, products reach more audience, services expand; and with change comes new sets of problems to solve. Systems whose core architecture was not planned with scaling in mind — either by under-engineering, miscalculated scope, or legacy reasons — will have an hard time expanding horizontally and/or vertically to meet their new system requirements. For this imminent transformation, there are two alternatives: either it is done gradually through a set of transformations or it is made from the ground up, resulting in big expenses and total system *reengineering*.

While, ultimately, the choice to rebuild from the ground-up or to refactor is system dependent and has several factors associated to it, in the latter choice, the direction of the migration of the architecture could be supported by a catalog of refactoring patterns that when applied to a non-reactive architecture, gradually transforms it into a reactive architecture.

1.3 DISSERTATION SCOPE AND CASE STUDY

We assume that the reactive paradigm is a good solution for buidling scalable, fault-tolererant systems and thus, our work will focus on the transformation of a non-reactive system A, to a reactive system B, through a set of transformations. Given the lack of literature on the subject, our focus will be divided into two sets of concepts that aim at answering the following questions: "*how do we structure our transformations?*" and "*what concepts to tackle in our transformations?*" by exploring patterns and refactoring (chapter 2), and the reactive paradigm (chapter 3) respectively.

A context for this kind of transformation would prove to be too generic, since systems come in all forms and formats. Instead, we will focus in the case study of a large distributed system, due to its size, complexity, heterogeneity and distribution: Betfair (an online betting exchange company further explored in Section 2.6). Blip, the company where this dissertation will take place, integrates itself with Betfair, and as such this system will be our primary focus point. Here, we will apply our transformations to subparts of their system and provide theoretical solutions.

1.4 OBJECTIVES AND CONTRIBUTIONS

The proposed project, inserted in the context of the our master's dissertation, is the design of a catalog of refactoring patterns to support the transform the Case Study's architecture (the system will be described in detail, in section 2.6) into a reactive one, which:

(a) is composed by an set of refactoring patterns that transform a non-reactive system, like the Case Study's system, to a reactive system. These patterns will be focus on the transformation of parts of the system;

(b) when applied to a system, should improve the relative performance over the old architecture. The improvement could be externally measured by: *(i)* latency[1], *(ii)* maximum number of connections the system can handle simultaneously, and *(iii)* CPU usage;

(c) and finally, provides a viable method, cost and complexity wise, to migrate from a non-reactive system. After the "transformation", the system should exhibit 4 properties: *(i)* be event-driven, *(ii)* scalable, *(iii)* resilient, and *(iv)* responsive.

While this dissertation is scoped for a specific context, we believe the benefits of this set of transformations would likely remain valid when applied to a system with similar architecture, and would likely yield similar results.

Our contribution is not just based on the development of patterns, as many may have already be identified and formalized, but on the development of **transformation patterns**.

1.5 DISSERTATION STRUCTURE

The remaining of this dissertation is organized into six chapters, with the following overall structure:

Chapter 2: Background. This chapter reviews the most important background concepts necessary to understand the context of the problem in this dissertation and an overview of the case study (2.6).

Chapter 3: State of the art. This one provides and extended view on issues relevant for the dissertation and a state of the art overview mostly focused on reactive paradigm.

Chapter 4: Problem. The fourth chapter states the dissertation problem, our main thesis (4.3) and the approach (4.4).

Chapter 5: Patterns. The fifth chapter presents the developed transformation patterns, along with their forces and relationships.

Chapter 6: Case Study. Here we provide an insight into the case study infastructures, their systems (6.1), and main challenges (6.2). It also relates the developed patterns with their system (6.3).

Chapter 7: Conclusions. The seventh and final chapter presents the conclusions.

[1] Time elapsed between the request and the reply

Chapter 2

Background

This chapter describes the most important background concepts necessary to understand the context of the problem in this dissertation. Considering the lack of literature on the topic of *refactoring to a reactive system*, a more theoretical set of contents will be explored in this chapter that aims at answering the question of "*how do we structure our transformations?*". Section 2.1 exposes and explains the necessary essential characteristics a system is required to have in order to be classified as *"Reactive"*. Sections 2.2 and 2.3 describe the concepts of Software Patterns and Refactoring. Section 2.6 presents an overview of case study where this dissertation will focus.

2.1 REACTIVE SYSTEMS

First of all, let's begin by describing what a Reactive System is. Merriam-Webster defines *reactive* as *"readily responsive to a stimulus"* [Dic14], i.e. its components are *"active"* and are always ready to receive events. The Reactive Manifesto [Man14] definition captures the essence of reactive systems in a way, that they:

- **react to events.** The event-driven nature enables the following qualities.
- **react to load.** Focus on scalability by avoiding contention on shared resources.
- **react to failure.** Are resilient with the ability to recover at all levels.
- **react to users.** Honor response times guarantees regardless of load.

Consequently, a reactive system's underling architecture is composed in such a way that it possesses 4 core characteristics: **Event-driven**, **Scalable**, **Resilient** and **Responsive**.

Being event-driven is a technical property that enables the properties of being scalable and resilient and all 3 properties together enable systems that are *responsive*, as is shown in Figure 2.1.

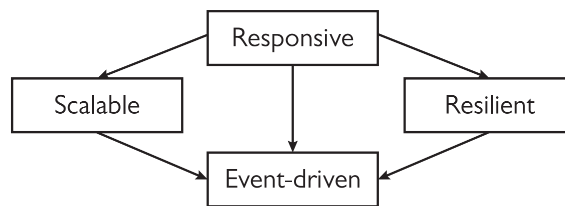


Figure 2.1: Reactive systems properties dependencies

Typical applications that benefit from a reactive design include user interfaces, required to coordinate the various requests (from the keyboard, the mouse and other devices) with information coming from the application (enabling and disabling input components and so on). Such systems generally decompose into independent parallel components cooperating to solve a given task, and exhibit a high degree of concurrency [GR92].

2.1.1 Characteristics

In the next sections we explore which properties a **Reactive System** is thus required to have to be classified as such.

2.1.1.1 Event-driven

A system based on asynchronous communication implements a loosely coupled design, better than one purely based on synchronous method calls [TS02]. The sender and recipient can be implemented without regards to the details of how the events are propagated, allowing the interfaces to focus on the content of the communication. This leads to an implementation which is easier to extend, evolve and maintain, offers more flexibility and reduces maintenance cost [Kuh14].

Since the recipient of asynchronous communication can remain dormant until an event occurs or a message is received, an event-driven approach can make efficient use of existing resources, allowing large numbers of recipients to share a single hardware thread. A non-blocking application that is under heavy load can thus have *lower latency* and *higher throughput* than a traditional, non-reactive, system based on blocking synchronization and communication primitives. The result is a better usage of available resources and lower operational costs [Kuh14].

In an event-driven scenario, the component interactions are performed at the expense of production and consumption of *events* — discrete pieces of information describing facts — through asynchronous and non-blocking communications. The system then relies much more on *push* rather then *pull* or *poll* methods [AFZ97], i.e. it's the *producer* that communicates with the *consumer* when it is available and not the other way around (which wastes resources by having them continually poll).

- *Asynchronous* communication through events — also called message passing — results in the system being highly concurrent by design and that it can take advantage

of multi core hardware without any change. Multi cores within a CPU are able to process message events, leading to a dramatic increase in opportunities for parallelization [Keg04].

- *Non-blocking* means the ability to make continuous progress in order to for the system to be responsive at all times, even under failure and burst scenarios. For this all resources needed for a response — for example CPU, memory and network — must not be monopolized. As such it can enable both lower latency, higher throughput and better scalability [Keg04].

Traditional, non-reactive, server-side architectures rely on shared mutable state and blocking operations on a single thread. Both contribute to the difficulties encountered when scaling such a system to meet changing demands. Sharing mutable state requires synchronization, which introduces accidental complexity and non-determinism, making the program code hard to understand and maintain. Putting a thread to sleep by blocking uses up a finite resource and incurs a high wake-up cost.

The decoupling of event generation and processing allows the runtime platform to take care of the synchronization details and how events are dispatched across threads, while the programming abstraction is raised to the level of business work flow and frees the programmer from dealing with low-level primitives. They also enable loose coupling between components and subsystems. This level of indirection is, as we will see, one of the prerequisites for scalability and resilience.

When applications are stressed by requirements for high performance and large scalability it is difficult to predict where bottlenecks will arise. Therefore it is important that the entire system is asynchronous and non-blocking.

A reactive system must be reactive from *top to bottom* for it to be effective. This principle is well illustrated by the Amdahl's Law that states:

$$S(n) = \frac{1}{B + \frac{1}{n}(1 - B)}$$

where $S(n)$ is the theoretical speedup factor, $n \in \mathbb{N}$ the number of threads) and $B \in [0, 1]$ is the fraction of the algorithm that is strictly serial.

Therefore, the speedup of a program using multiple threads in parallel computing is limited by the sequential fraction of the program; and even in the unlikeness of a program whose 95% of the code can be parallelized, the maximum theoretical speedup would be $20x$ and never higher, no matter how many threads are used [Gus88], as shown in figure 2.2.

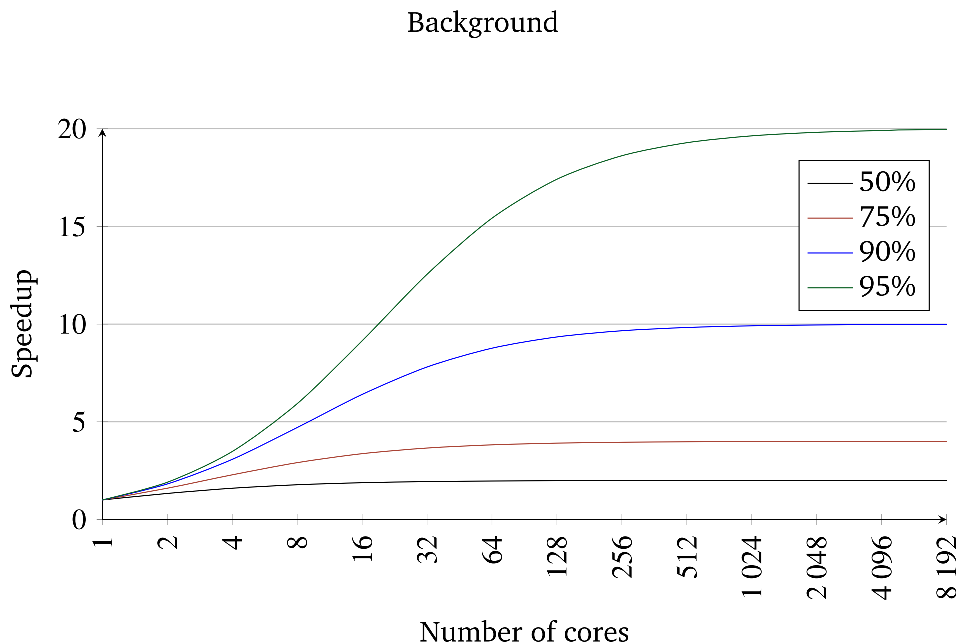


Figure 2.2: Amdahl's law on theoretical maximum speedup using parallel computing at different percentage levels of parallelization of a program

2.1.1.2 Scalable

Weinstock defines *Scalability* as the ability to handle increased workload by repeatedly applying a cost-effective strategy for extending a system's capacity [WG06]. A scalable system is able to be expanded according to its usage. This can be achieved by adding elasticity to the application, or in other words, the option of being able to scale out or in (add or remove nodes) on demand. In addition, the architecture makes it easy to scale up or down (deploying on a node with more or fewer CPUs) without redesigning or rewriting the application. Elasticity makes it possible to minimize the cost of operating applications in a cloud computing environment, allowing you to profit from its pay-for-what-you-use model.

Scalability also helps to manage risk: providing too little hardware to keep up with user load leads to dissatisfaction and loss of customers, while having too much hardware idling for no good reason results in unnecessary expense. A scalable solution also mitigates the risk of ending up with an application that is unable to make use of new hardware becoming available: we will see processors with hundreds, if not thousands of hardware threads within the next decade, and utilizing their potential requires the application to be scalable at a very fine-grained level.

An event-driven system based on asynchronous event messaging provides the foundation for scalability. The loose coupling and location independence between components makes it possible to:

- *scale vertically — or up:* make use of parallelism in multicore systems.
- *scale horizontally — or out:* make use of multiple server nodes.

Adding more instances of a component increases the system's capacity to process events. In terms of implementation, there is no difference between scaling up by utilizing multiple cores or scaling out by utilizing more nodes in a datacenter or cluster. The topology of the application becomes a deployment decision, which is expressed through configuration and/or adaptive runtime algorithms responding to application usage. This is what we call *location transparency* [TS02].

2.1.1.3 Resilient

The term *resilience* comes borrowed from the social-ecological field and Walker et al. define it as the capacity of a system to absorb disturbance and reorganize, while undergoing change so as to still retain essentially the same function, structure, identity, and feedbacks [WHCK04]. In a reactive system, concurrency is ubiquitous, therefore failure (such as software failure, hardware failure or connection failures) must be treated as a first class construct. Resilience cannot be an afterthought, it has to be part of the design right from the beginning, with means imposed to react and manage failure leading to highly tolerant systems able to heal and repair themselves in runtime. A system can be considered resilient if it can recover quickly from failure.

The loose coupling characteristic presented in the event-driven model provides for fully isolated components in which failures can be contained within their context, encapsulated as events and sent off to other components — there are components whose function within the system is to supervise the failure of other components — that can inspect the error and decide on how to act at runtime.

This approach creates a system where business logic remains clean, separated from the handling of the unexpected, where failure is modeled explicitly in order to be compartmentalized, observed, managed and configured in a declarative way, and where the system can heal itself and recover automatically. It works best if the compartments are structured in a hierarchical fashion, much like a large corporation where a problem is escalated upwards until a level is reached which has the power to deal with it. This approach is purely event-driven, based upon reactive components and asynchronous events and therefore location transparent [Man].

2.1.1.4 Responsive

A *responsive* system is able to quickly respond or react appropriately. This last property is the most perceived by the end-user, and it is dependent on all the other 3 previous properties. A system is responsive if it provides near to real-time interaction with its users even under load and in the presence of failures. One example is Google Docs[1], which enables users to edit documents collaboratively, in real-time — allowing them to see each other's edits and comments live, as they are made.

[1] www.drive.google.com

Systems that respond to events need to do so in a timely manner, even in the presence of failure. If an application does not respond within an applicable time constraint — otherwise known as latency — then it is effectively unavailable and therefore cannot be considered resilient.

Responsive systems can be built on a event-driven, scalable, and resilient architecture. However, this does not automatically lead to an responsive system. The architecture must also account for system design, back pressure, etc.

2.1.2 Conclusion

Reactive systems are nothing less complex systems. They are event-driven, scalable, resilient and responsive. Together with the fact that concurrency is ubiquitous among this systems, all evidence suggest we should embrace it. However, parallelism is elusive. It is in its most primal form, non-deterministic and there is an extra effort to develop a concurrent application that functions properly, let alone one that runs efficiently. Therefore, strategies must be applied that tackle this problem. This solution, emerges from the modeling of several known problems, and aims to solve it not only for a specific context, but for a recurring one.

2.2 PATTERNS

The early concept of *pattern* was defined as: a solution to a problem in context by Christopher Alexander [AIS77] — a civil architect — in 1977. He extended this notion beyond the triplet of *(problem, forces, solution)* to that of a pattern language. Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice [AIS77]. Riehle and Zullighoven [RZ96], give a definition of the term *pattern* as:

> *A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts.*

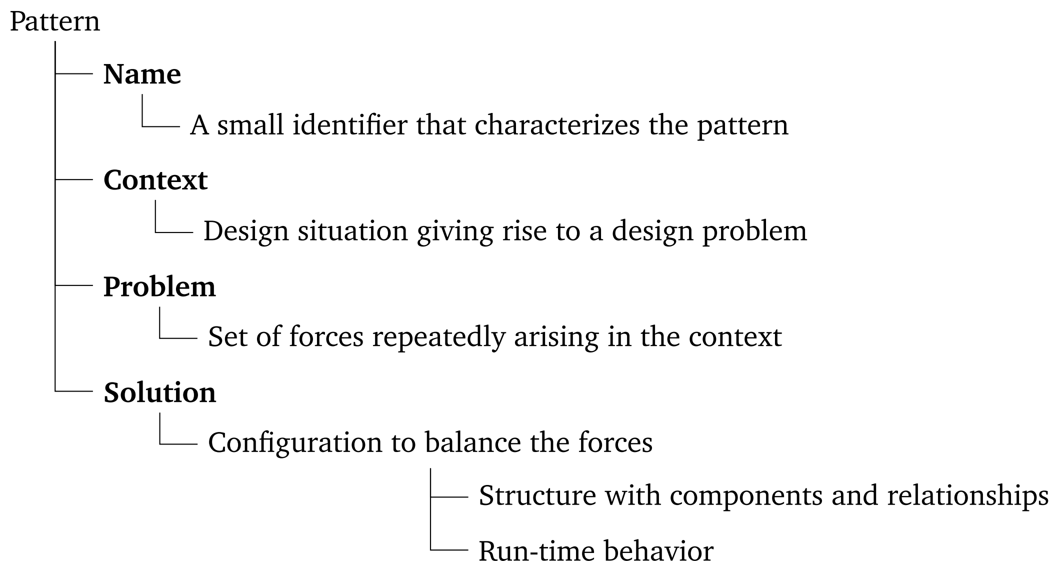
Using the pattern form, the description of the solution tries to capture the essential insights which it embodies, so that others may learn from it and make use of it in similar situations. Later, adapting Alexander for software, Gamma *et al.* [GVHJ95] (also known as *"Gang of Four"* or *"GoF"*) and Buschmann *et al.* [BMR+96] published the first books on software *design* and *architectural* patterns. These two types of patterns, along with a third called *idioms* can be defined as follows:

- **Architectural patterns** express fundamental structural organization schemes for software systems, decomposing them into *subsystems*, along with their responsibilities and interrelations.

- **Design patterns** are medium-scale tactical patterns, specified in terms of interactions between elements of object-oriented design, such as *classes*, *relations* and *objects*, providing generic, prescriptive templates to be instantiated in concrete situations. They do not influence overall system structure, but instead define micro architectures of subsystems and components.
- **Idioms** (sometimes also called coding patterns) are low-level patterns that describe how to implement particular aspects of components or relationships using the features of a specific programming language.

However, software patterns touch a broader range of subjects than the previously mentioned object oriented design as we have seen in Elements of Reusable Object-Oriented Software [GVHJ95]. We can also see their applicability in Enterprise Integration Patterns [HW04], Pattern oriented Software Architecture [BMR+96, SSRB00, KJ04, BHS07b, BHS07c], Analysis Patterns [Fow, Fow96] and Adaptive Objective models [DYBJ04, Fer10].

A pattern provides a solution schema rather than a fully-specified artifact or blueprint. We should be able to reuse the solution in many implementations, but so that its essence is still retained. A pattern is a mental building block. After applying a pattern, an architecture should include a particular structure that provides for the roles specified by the pattern, but adjusted and tailored to the specific needs of the problem at hand. No two implementations of a given pattern are likely to be the same. Although there exists some variations from different authors, the anatomy of a pattern [BMR+96] can be defined as:



However, there are patterns that are not formally described and can still be considered as such, e.g. Functional Patterns [Has09].

The pattern should exhibit traits in such a way that: *(a)* describes the form of **recurrent** solutions and their contexts in the application, *(b)* **explains the reason** for this

fitness in terms of force, *(c)* **predicts** that in another context of a similar kind, the pattern will help to solve the problem, *(d)* is **instructive** (not prescriptive) for practical design action, *(e)* is a **three part rule** consisting of `IF X` (context) `THEN Z` (solution) `BECAUSE of Y` (problem as a network of interrelated forces). Pattern languages have had a profound impact in the way software engineers build and manage software today. In the case of design patterns, the essential catalog, primordial is known as *Gang of four Design Patterns* [GVHJ95].

However, a pattern alone is most useful when building systems from the ground-up, when still designing an architecture. Therefore, what to take of fully structured systems that wish to take advantage of these?

2.3 REFACTORING

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the solution yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you *refactor* you are improving the design of the code after it has been written.

Fowler et al. [FBB+] claims that a *refactor* is not the cure for all software ills as it is no *"silver bullet"*. However, they can, and should, be used for several purposes, such as:

- **Improving the Design of Software.** Poorly designed code usually takes more code to do the same things. Thus an important aspect of improving design is to eliminate duplicate code. Eliminating code duplication ensures that the code states everything once and only once, which is the essence of good design.
- **Making software easier to understand.** Code is likely to be changed in time, often by others. *Refactoring* makes software more structured, which, in turn, better communicates its purpose and is consequently easier to understand by third parties.
- **Helping to find Bugs.** More understandable and structured programs makes more robust (with less bugs) code. Robust in the sense that is less likely to have bugs in it. Thus, *refactoring* code makes it easy to find and eliminate bugs.
- **Helping to program faster.** A good design is essential to maintaining speed in software development. Thus, *refactoring* helps develop software more rapidly.

A common concern with *refactoring* is the effect it has on the performance of a program. To make the software easier to understand, you often make changes that will cause the program to run more slowly. Refactoring certainly will make software go more slowly, but it also makes the software more amenable to performance tuning. The secret to fast software, in all but hard real-time contexts, is to write tunable software first and then to tune it for sufficient speed [FBB+].

Fowler et al. created a catalog of widely accepted *refactorings* (seen in table A.1 in Appendix A) that share a common structure and can be labeled as the *refactor's* anatomy:

- **Name** — Name of the refactoring. Important to define a vocabulary of *refactorings*.
- **Motivation** — Short description of the situation in which one needs the *refactoring* and a summary of what the *refactoring* does.
- **Mechanics** — Concise, step-by-step description of how to carry out the refactoring.
- **Example** — Simple use of the refactoring to illustrate how it works.

The decision of *when to refactor* code is just as important as knowing *how to refactor* code. Fowler does not mean to give precise criteria when a *refactoring* is overdue, but instead enlists common roots of *"evil"* [FBB+], code smells, such as:

- duplicated code
- long methods
- large classes
- long parameter list
- divergent changes
- shotgun surgery
- feature envy
- data clumps
- primitive obsession
- parallel inheritance hierarchies
- lazy classes
- speculative generality
- temporary fields
- message chains
- middle man
- inappropriate intimacies
- alternative classes with different interfaces
- incomplete library classes
- data classes
- refused bequests

Although it is possible to *refactor* manually, tool support is considered crucial. The usability of a tool is affected by several characteristics and can be grouped by: automation, reliability, configurability, coverage, scalability (of the refactoring tool) and language independence [MT04].

Also, *refactoring* is an important activity in the software development process, used in Software *reengineering* which we can divide into 2 fields: *(i)* Agile software development and *(ii)* Framework-based or product line software development [MT04].

2.4 REFACTORING TO PATTERNS

Kerievsky [Ker05] created the concept of *Refactoring to Patterns*. Refactoring to Pattern is similar to low-level Refactorings [FBB+] as both help to improve designs by reducing or removing duplication, simplify what is complicated and improve the code's capacity to communicate its intention.

However, *Refactoring to Patterns* is meant to bridge the gap between patterns and refactoring. This relation between the two was already noted by Buschmann et al. [BMR+96] and Fowler et al. [FBB+], while Kerievsky [Ker05] created a catalog of *Refactoring to Patterns*, that can be seen in table A.2 of appendix A.

Furthermore, these *Refactoring to patterns* can have 2 directions: *to* and *away*. The direction a pattern-directed refactoring takes is often influenced by the nature of the pattern:

- Moving *to* is implementing a refactoring in which the result is the pattern as a whole.

- Moving *away* when something simpler is discovered.

In addition, once one applies a refactoring *to* pattern, one must evaluate whether the design has indeed improved. If it has not, one needs to backtrack or refactoring *away* from a pattern.

In similarity, both *Refactoring* and *Refactoring to patterns* share the same anatomy (name, motivation, mechanics and example). The difference is that, with *Refactorings* we're transforming code in order to maximize some attribute (modularization, flexibility, readability or performance) without changing functionality, whereas in the latter option, they're transforming code to maximize the application of patterns, which indirectly may lead to the improvement of several metrics. Additionally the latter does not present a formalized structure and may be difficult to create automatic transformations (e.g. via IDE) and should be seen as more of a "*guide*" to converting the code *A* in a certain specific format to code *B* defined as a pattern by applying the transformation *C*.

2.5 EPISTEMOLOGY OF PATTERNS AND REFACTORINGS

We've explored in the previous sections the core definitions of patterns and *refactorings* but one cannot make use of these without first going into the scientific background that proves (or tries to disprove) their truthfulness.

Whether or not **patterns** can be called scientific has been the topic of discussion among *practicioners* and researchers. They argue that, although a good pattern contains "'nothing new", it captures existing knowledge and that pattern mining is a valid scientific endeavor, respectively. However, the perception of the validity of a pattern is wide open to interpretation and directly related to the individual asserting it. It is therefore necessary to be able to distinguish between "correct" and "incorrect" patterns. This means evaluating their content empirically and making sure its structure is logically sound. Buschmann [BHS07a] states that patterns are not "created artificially just to be patterns" and that "good patterns are those solutions that are both recurrent and proven", in other words, the notion that a pattern is natural rather than artificial. There is however the notion of pattern as theories [KP09], that a pattern is not the same as its manifesting artifacts, and that abstract entities can only manifest in real instances.

Looking at patterns as specific theories means we do not have to invent new standards to test and justify them. Instead we can rely on tried and tested methods. From an empirical perspective, all knowledge must be based on experience. Therefore, different approach can be taken when mining for a pattern: *(a) inductive* — the process of inferring from a number of observed positive cases the properties of new cases (generalization and extrapolation), *(b) deductive* — if we apply a pattern in the right context but it does not solve the present problems it is considered falsified, *(c) abductive* — to look for a pattern in a phenomenon and suggest a hypotheses. Consequently, the extent to which we can test a pattern depends on its empirical content. The confidence we can put into a pattern does not only rely on the number of cases it was based on, but also on how much a pattern claims with its force resolution. Thus, the more constrained is the context claimed by the pattern, the lower the empirical content; in contrast the more general the context, the higher the empirical content. While in terms of the claimed solution, the more specific, the higher the empirical content.

Going back to the notion of validity, patterns are not tried and true *per se*; just like theories, they have to be subjected to empirical tests. Since it is not a simple hypothesis (*if* context *then* solution) but a set of hypothesis that explain the forces which cause fitness between context and solution, where each of them can be falsified empirically. When testing for this, one cannot do this in isolation; the pattern must be tested as a whole, which includes a lot of implicit tests that justify the "Rule of three". This rule informally suggests that should be at least three known uses. A singular solution is just for design, two occurrences might be random, whereas the third suggests a solution a "pattern". It can make a pattern significant but not necessarily plausible or true, as for any statistical method. However, considering a limited set of objects it is not even likely that a specific pattern re-occurs two times by change. So, actually three recurrences is the minimum number of cases necessary to suggest the validity of an induced pattern.

As for **Refactorings**, they are often seen as formalized ways for cleaning up code. In a way, the base context for a *refactoring* is exactly that, but it goes further by providing techniques for cleaning up code in a more efficient, controlled manner which induces less *bugs*. The main focus of a *refactoring* is to make software easier — for humans — to understand and modify and, to change little to nothing in the observable behavior. Because of that, *refactoring* is often tightly associated with testing because if we're transforming a piece of code into another "supposedly better" piece of code, because we need insurance that the base logic remains unaltered,or in other words, we need to make sure that the software still carries the same function as it did before. Consequently, when developing, time is often divided into distinct and exclusive activities: adding function and refactoring. The first should only focus on adding new functionalities, and its progress can be measured by adding tests and getting them to work. The latter must make a point of not adding function; only restructuring code while maintaining the behavior. There is always this implication of preserving the "behavior", but a precise definition of behavior

is rarely provided, or may be too inefficient to be checked in practice. This leads to the problem of intuitive definition of observational behavior equivalence; stating that "for the same input, we should obtain exactly the same output", does not always suffice [Gor03]. For instance: in *real-time systems*, an essential behavior is the timing of execution with attention to the order sequenced operations, in *embedded systems*, memory constraints and power consumption should also be taken into account; and for *safety critical systems*, there is a concrete notion of safety that needs to be preserved. Therefore, in an ideal world, refactoring should also be able to preserve these properties as well.

When, instead of trying to *refactor to patterns*, we're transforming our architecture from our initial code to code that has been extensively studied in every aspect and its wide accepted as a valid pattern by the community, we can therefore with great significance assume that within a specific context, this specific pattern will be a good fit. We do this because we do not want to *"reinvent the wheel"* every single time.

2.6 CASE STUDY

Betfair claims to be the *"world's leading online betting exchange"* and, became popular as it enables punters[2] to choose their own odds and bet against each other, even after an event has started. In addition to the exchange service, *Betfair* has an established games portfolio: *Betfair* Games, *Betfair* Poker, *Betfair* Casino, *Betfair* Mobile among others.

As one of the pioneers of the world's online betting exchange, Betfair's website today operates at very large scale, with a website and architecture that must support multiple web and mobile applications, vast numbers of users and a large volume of data and operations. From a business perspective, Betfair's operations (in 2013) include [Cru14]:

- 4 million funded user accounts.
- 140 locations worldwide.
- 30,000 bets placed per minute.
- 120 requests per second.
- 347 million euros of customer's money on the books at any given time.
- 2.6 billion euros of mobile business in FY12, representing 111% growth in mobile year on year.
- 500 GB of logging data per month.
- 30 thousand unique US visitors on average per month in the last year (figure 2.3).

On a recent data management technology evaluation [Cru14], the top priorities for *Betfair* included:

[2] A person who gambles, places a bet, or makes a risky investment

- **Performance** — especially deterministic performance on virtual machines.
- **Strong consistency** — database centric.
- **Scaling** — both current and future.
- **Schema Flexibility** — database centric.
- **Multi-tenancy** — when required.
- **Simplicity** — which *Betfair* saw as fundamental to reliability, continuous delivery and deterministic performance.

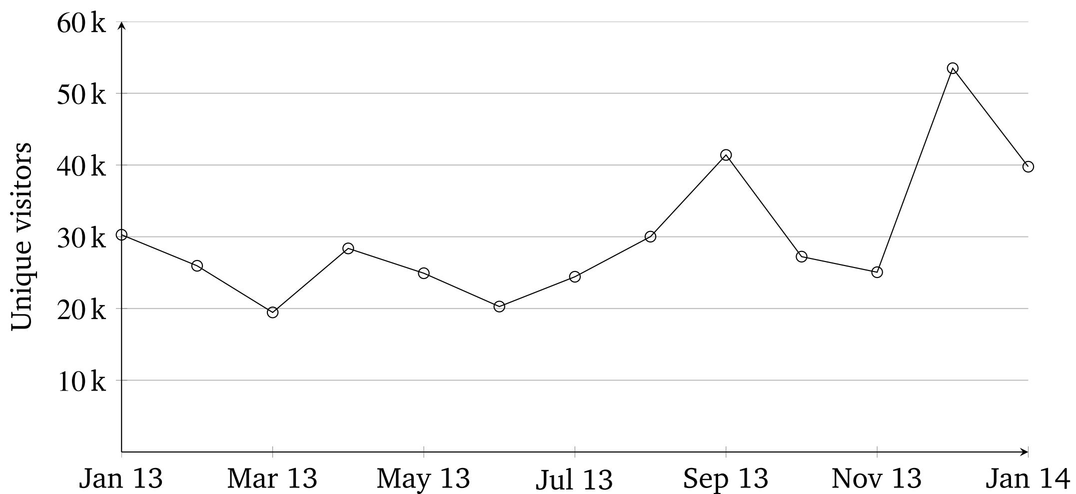


Figure 2.3: Number of Betfair's unique visitors from the US per month from January 2013 to January 2014 , according to Compete [Com14]

The *Betfair's* system is a large distributed system, due to its size, complexity, heterogeneity and distribution. As aforementioned, a subpart of this system, Blip, will be our case study for the context of this dissertation. The system integrates itself with Betfair's, relies heavily in JVM, and runs on pure Spring[3].

Finally, at the present time, Blip has been experiencing scaling issues. Their scaling rate is not linear which hinders aspects they'd like to have, such as: cost reductions, bigger adaptability and affordable scalability. Our work will, therefore, provide a study of the advantages of this transformation and pinpoint parts of Blip's system where it they can be applied.

2.7 SUMMARY

In this chapter we introduced and described what is a **reactive system**: that it is event-driven, scalable, resilient and scalable. We have reviewed the concepts of a **pattern**: a solution in a sufficiently generic manner as to be applicable in a wide variety of contexts,

[3]Java MVC framework at www.spring.io

and a **refactoring**: a transformation that does not alter the external behavior of the solution yet improves its internal structure. We then make the bridge between patterns and refactorings with the concept of **refactoring to patterns**. Lastly, we introduced the case study: Blip that operates under Betfair, a online betting company.

Chapter 3

State of the art

This chapter describes the State of the art work relevant to the context of the problem. Given the stated goal, we'll focus on the state of the art work of the endpoint system of our transformation, and for that we'll explore the Reactive paradigm (section 3.1) and support technologies that assist the development of reactive systems (section 3.2).

3.1 REACTIVE PARADIGM

Reactive programming is a programming paradigm oriented around data flows and the propagation of change, and focuses on behaviors rather than on data [ABBC06]. It makes it possible to express static or dynamic data flows with ease in the programming languages used, and that the underlying execution model will automatically propagate changes through the data flow.

Table 3.1: Duality between the Synchronous and Asynchronous models expressed as Scala types

	Single result	Many results
Pull/Synchronous	`Try[T]`	`Iterable[T]`
Push/Asynchronous	`Future[T]`	`Observable[T]`

In this paradigm we embrace that concurrency is ubiquitous in our programs and it must be dealt with as a first class problem. Consequently, in an environment predominated by asynchronous methods, abstractions are defined that deal with them in a more *composable*, error-tolerant manner. As shown in table 3.1, in the single value return field, we move from the synchronous computation to the asynchronous by using Futures (Section 3.1.1.1) and in the multiple return value return field we do so by using Observables (seciton 3.1.1.3). These allows us to follow a *Push* Model opposed to the *Pull — "Ask for"* — Model [AFZ97], as we saw in Section 2.1.1.1.

These abstractions allow us to follow an *event-driven* model; that is where *Reactive Programming* shows its true value with asynchronous event handling.

3.1.1 Primitives

In this Section we'll explore primitives that facilitate the creation of reactive systems using the Scala syntax.

3.1.1.1 Futures

A Future [Ell09] is an object holding a value which may become available at some point. This value is usually the result of some other computation:

- If the computation has not yet completed, we say that the Future is not completed.
- If the computation has completed with a value or with an exception, we say that the Future is completed.

Completion can take one of two forms:

- When a Future is completed with a value, we say that the future was successfully completed with that value.
- When a Future is completed with an exception thrown by the computation, we say that the Future was failed with that exception.

A Future has an important property that it may only be assigned once. Once a Future object is given a value or an exception, it becomes in effect immutable — it can never be overwritten.

```scala
val f: Future[String] = Future {
  getDataFromNetwork()
}

f onComplete {
  case Success(data) => println(data)
  case Failure(t) => println("An error has occured: " + t.getMessage)
}
```

Source 3.1: Example of creating and handling a Future in Scala

As exemplified in source 3.1, we can easily wrap a blocking computation in a Future and register a callback by using the `onComplete` method, which takes a callback function of type `Try[T]`[1] `=> U`. The callback is applied to the value of type `Success[T]` if the future completes successfully, or to a value of type `Failure[T]` otherwise.

Furthermore, Futures are FUNCTORS: contains a method specified by the generic algorithm. In this case and derived from functional programming in general, the method we

[1] A abstraction specifically designed to either hold a value or some throwable object

are referring about is `map`, which allow us call it on a instance of a Future and get another Future as result, thus promoting composition.

Another fact is that it's underlying implementation may vary from language to language, and its method signature may be different but the concepts remain the same. With Futures, we explicitly state that the operation is time consuming in the return type.

3.1.1.2 Promises

We have considered future objects created by asynchronous computations started using the future method. However, futures can also be created using Promises. While futures are defined as a type of read-only placeholder object created for a result which does not yet exist, a promise can be thought of as a writable, single-assignment container, which completes a future. That is, a promise can be used to successfully complete a future with a value (by *"completing"* the promise) using the `success` method. Conversely, a promise can also be used to complete a future with an exception, by failing the promise, using the `failure` method.

```
val p = Promise[T]
val f = p.future

val producer = Future {
  val r = produceSomething()
  p.success(r)
  continueDoingSomethingUnrelated()
}

val consumer = Future {
  startDoingSomething()
  f.onSuccess {
    case r => doSomethingWithResult()
  }
}
```

Source 3.2: Example of creating and handling a Future by using a Promise in Scala

In source 3.2, we begin two asynchronous computations. The first one does some computation, resulting in value *r*, which is then used to complete the future *f* by fulfilling the promise. The second does some computation, and then reads the result *r* of the completed future *f*. It is worth noting that the *consumer* can obtain the result before the *producer* computation is finished with the *continueDoingSomethingUnrelated()* method.

3.1.1.3 Observables

An *Observable* is an asynchronous data stream abstraction that lets other object (subscribers) subscribe to it. Then, the subscriber reacts whenever the *Observable* object emits events. This abstraction facilitates concurrent operations by non blocking while waiting

for the *Observable* to emit events, but, instead creates an *Subscriber* that awaits ready to *react* to upcoming events at whatever future time the *Observable* does so, be it synchronously or asynchronously.

Observables make the change to the asynchronous world as a dual of *Iterable*, just as Futures do (for single values) as a dual of *Try*, and while Futures are straightforward to use for a single level of asynchronous execution, they start to add non-trivial complexity when they are nested. This change is made in a way that the syntax between the duals is very similar as we can see in figure 3.1, and that methods like `map`[2] and `filter`[3] can be applied to any one of them.

```
//Iterable[String]
getDataFromLocalMemory()
  .drop(10)
  .take(5)
  .map(_ + "transformed")
  .foreach(s => println("next" + s))
```

(a) Blocking code using Iterables

```
//Observable[String]
getDataFromNework()
  .drop(10)
  .take(5)
  .map(_ + "transformed")
  .subscribe(s => println("next" + s))
```

(b) Non-blocking code using Observables

Figure 3.1: Comparison between Iterable and Observable syntax usage in Scala.

More important even, *Observables* are *composable*. They allow chaining operations and composition from other *Observers*, as shown in source 3.3. *Composability* is a corner-stone of functional and reactive programming as we will see in section 3.1.2.1.

```
val ticks: Observable[Long] = Observable.interval( 1 seconds )
val evens: Observable[Long] = ticks.filter( s => s%2==0 )
val textity: Observable[String] = evens.map( s => "Hello from "+s )

val subscriber = textify.subscrible( text => println(text))

subscriber.unsubscribe()
```

Source 3.3: Example of creating and composing of Observables in Scala

The Observable type adds two missing semantics to the Gang of Four's OBSERVER pattern [GVHJ95], to match those that are available in the *Iterable* type:

- the ability for the producer to signal to the consumer that there is no more data available (a `foreach` loop on an *Iterable* completes and returns normally in such a case; an Observable calls its observer's `onCompleted()` method)
- the ability for the producer to signal to the consumer that an error has occurred (an *Iterable* throws an exception if an error takes place during iteration; an Observable calls its observer's `onError()` method)

[2] An operation that associates each element of a given set with one or more elements of a second set
[3] An operation that removes all elements of a set that do not meet a specified condition

One benefit of Observables is that it is abstracted from the source of concurrency. It is not opinionated on how the backing implementation works. For example (figure 3.2), an observable could: **(a)** use the calling thread to synchronously execute and respond; **(b)** use a thread-pool to do the work asynchronously and callback with that thread; **(c)** use multiple threads, each thread calling `onNext(T)` when the value is ready; **(d)** do work asynchronously on a actor (or multiple actors); **(e)** do network access asynchronously using *NIO* [4] and perform callback on a *Event Loop*; **(f)** do work asynchronously and perform callback via a single or multi-threaded *Event Loop*.

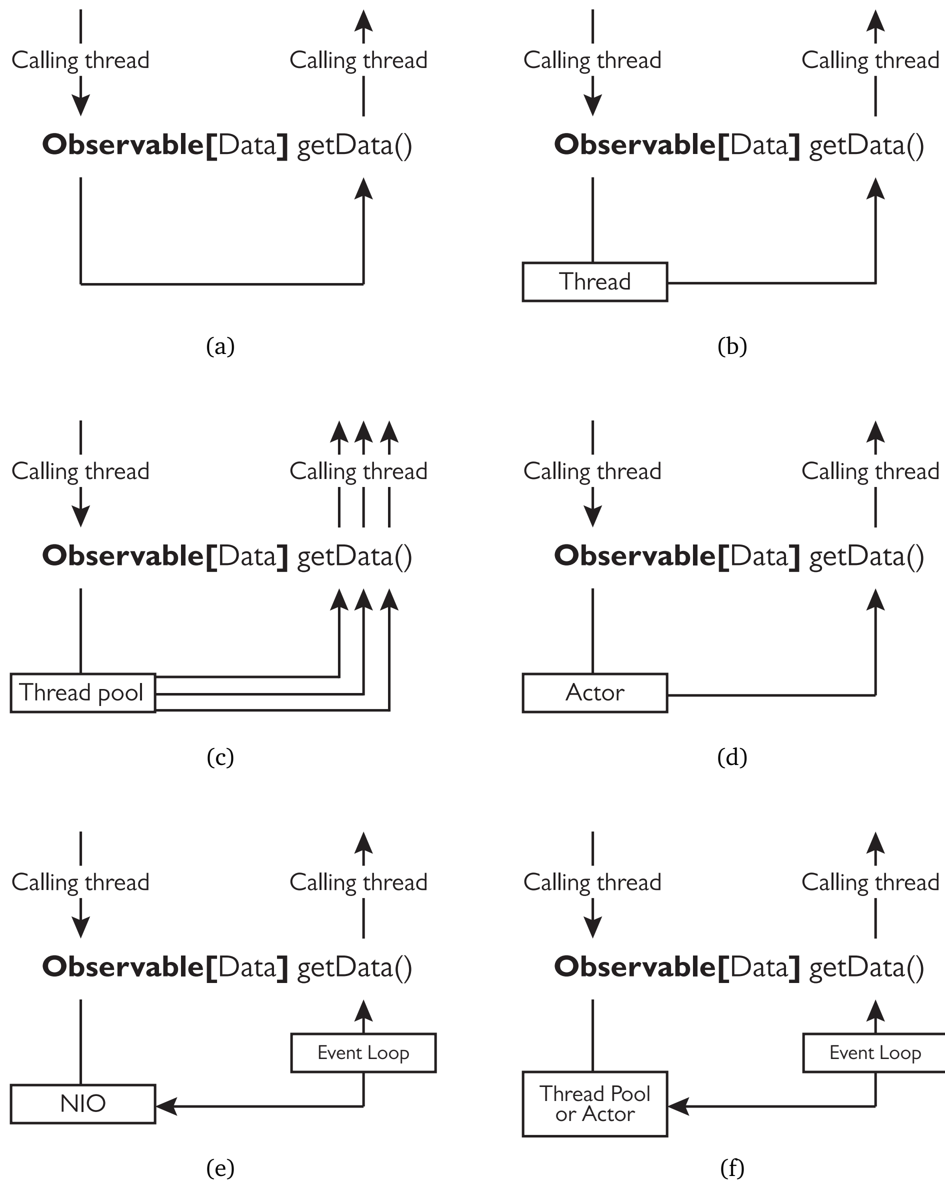


Figure 3.2: Example of several underlying implementations of Observables

[4]New I/O is a collection of Java APIs that offer features for intensive I/O operations

3.1.2 Abstractions

In this Section we will explore abstract concepts of the reactive paradigm.

3.1.2.1 Composability

"Composition" is the principle of putting together simpler things to make more complex things, then putting these together to make even more complex things, and so on. This "building block" principle is crucial for making even moderately complicated constructions; without it, the complexity quickly becomes unmanageable [FY97].

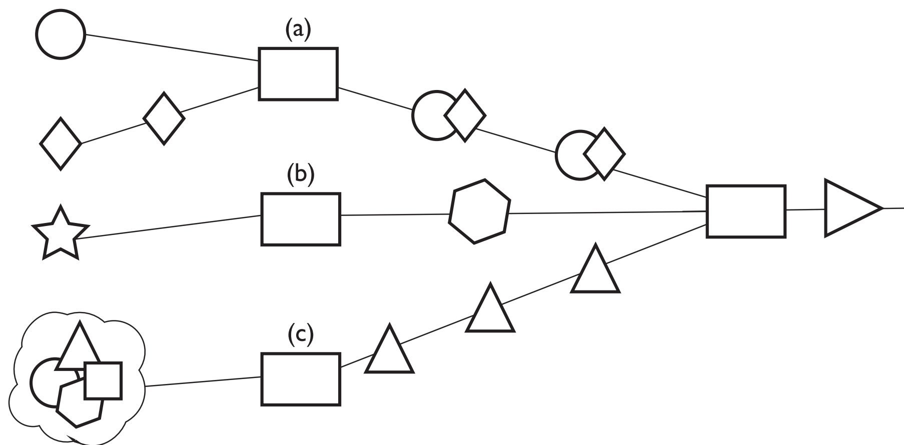


Figure 3.3: Composition flow diagram

Since Futures, Promises, and Observables are treated like first-class objects, we can compose simpler ones to form more complex ones. Figure 3.3 displays a composition flow where we can see examples of **(a) Compose** — we are joining two simpler *object* into a more complex one; **(b) Map** — we are transforming an *object* into a different one; and **(c) Filter** — we are filtering all *objects* that do not match a specified criteria. In the end, we obtain a more complex *object* which encloses all of the previous simpler ones.

Elliot and Hudak [EH97] present a good example of *Composition* with a functional reactive animation, where they compose complex animations (transformations, translations, scalings) of several objects composed by simpler ones and animate them over time. Next, they expand on the ease of extending it by other external factors, by instead of having the animations perform a static routine, adding some dynamic behavior, e.g. following the mouse cursor while performing other tasks.

3.1.3 Immutability

While immutability is not a requirement for reactive programming, avoiding the use of global mutable data allows for code that is easier to reason with, test and code, and that *scales* better right off the box [GP12]. By *scaling better*, we mean: inherent parallelization, executing different parts of the code in different threads or machines without worries of misbehavior or dependencies for example. Sharing mutable state requires synchronization, which introduces incidental complexity and non-determinism, making the program

code hard to understand and maintain. Putting a thread to sleep by blocking uses up a finite resource and incurs a high wake-up cost.

Functional programming takes advantage of this *referential transparency* [WH00] and, for example, in Scala several immutable collections can be made to work in parallel out of the box (taking advantage of all the cores of the host system), just by adding `par` to its invocation.

3.1.4 Blocking vs Non-blocking

We now continue on the idea that a reactive system must be reactive from *top to bottom* for it to be effective. The system should be composed entirely of non-blocking methods, or risk itself to become a weak link — a bottleneck [5] — wherever blocking methods are used.

Take the following example, trying to read a big file from disk. If we follow an blocking approach, we will request a chunk of the file and all operations on that execution context will come to an halt until the chunk is returned. I/O operations are expensive, and if we keep doing this as a common approach in a program it will certainly become a bottleneck. On the other hand, in a non-blocking approach, we would request the chunk and this would happen on another execution context, and we are free to keep executing other pieces of code here. When the chunk is returned, the original requester is signaled.

3.2 SUPPORTING TECHNOLOGIES

The following libraries represent the state of the art in the field on reactive programming frameworks. Most of them use abstractions that have been presented in the previous sections, with a few exceptions.

3.2.1 Reactive Extensions

Originally created by Erik Meijer, the Reactive Extensions[6] (Rx) is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators. While initially developed for .NET (Rx.NET), it was adapted for other languages such as Javascript (RxJS), C and C++ (RxCpp), Ruby (Rx.rb) and Python (RxPy).

It supports abstractions described on the previous chapters to help the development of reactive systems such as Observables to represent asynchronous data streams, query asynchronous data streams using LINQ operators (operators implemented by the Observable to create, convert, combine, and perform more auxiliary functions) and parameterize the concurrency in the asynchronous data streams using Schedulers (which manage the execution context of Observables) [LB11].

[5] The performance or capacity of an entire system is limited by a single or limited number of components
[6] `www.rx.codeplex.com`

3.2.2 RxJava

RxJava[7] is a JVM based implementation of Microsoft Reactive Extensions, with language bindings for Java, Scala, Clojure, and Groovy. The library composes asynchronous and event-based programs by using observable sequences.

It extends the Observer pattern to support sequences of data/events, and adds operators that allow you to compose sequences together declaratively while abstracting away aspects like low-level threading, synchronization, thread-safety, concurrent data structures, and non-blocking I/O.

It was developed by Netflix[8], an highly distributed system for video streaming, for their scaling needs.

3.2.3 Finagle

Finagle[9] is an extensible RPC system for the JVM, used to construct high-concurrency servers. Finagle implements uniform client and server APIs for several protocols, and is designed for high performance and concurrency. Most of Finagle's code is protocol agnostic, simplifying the implementation of new protocols. Finagle is written in Scala, but provides both Scala and Java idiomatic APIs.

Finagle is mainly based on Futures contrarily to Rx and its Observers. Additionally, the library is packaged with an extensive number of methods for managing a HTTP Server from Load balancers, to transport layers and Rate limiting to name a few.

It was developed by Twitter[10] to help with their user load problems and scaling issues resulting in Fail whale[11] pages.

3.2.4 Akka

Akka[12] is a toolkit and runtime framework for building highly concurrent, distributed, and fault tolerant event-driven applications on the JVM. It stands out from the previous frameworks and while it, internally, uses a great part of already mentioned abstractions, it explores a model not described in the previous chapters, the *Actor Model*.

The actor model surged in 1973, initially for research in artificial intelligence by Hewitt et al. [HBS73]. Nearly two decades later, it had its first commercial use in the telecommunications platform using Erlang (a pure functional programming language whose concurrency model is based on actors) as a success in terms of robustness and resilience [AVWW96]. In 2009, Akka was created.

The actor model represents objects and their interactions, resembling human organizations and built open the laws of physics. Each actor has an unique identity, a behavior (a

[7] www.github.com/Netflix/RxJava
[8] www.netflix.com
[9] www.twitter.github.io/finagle/
[10] www.twitter.com
[11] Twitter's characteristic page served when its overloaded
[12] www.akka.io

well defined task it performs) and it only interacts with other actors using asynchronous message passing [HBS73].

In general, actors give you the benefit of offering a high level of abstraction for achieving concurrency and parallelism. Furthermore, the Akka actors library adds these additional benefits:

- Lightweight, event-driven processes. The documentation states that there can be approximately 2.7 million actors per gigabyte of RAM.
- Fault tolerance. Akka actors can be used to create "self-healing systems".
- Location transparency. Akka actors can span multiple JVMs and servers; they are designed to work in a distributed environment using pure message passing.

For fault-tolerance it adopts the "Let it crash" model which the *telecom* industry has used with great success to build applications that self-heal and systems that never stop. It achieves that by using actor supervision by other actors. Since actors are organized in a hierarchical fashion (actors spawn actors and become their "parent"), more volatile tasks can be pushed lower down the chain and on failure, the parent can then handle its children failure.

Actors also provide the abstraction for transparent distribution (a program working locally is ready to work on a network of nodes) and the basis for truly scalable and fault-tolerant applications.

3.3 SUMMARY

In this chapter we presented the state of the art of the reactive paradigm along with it's primitives, abstractions and frameworks. For primitives we explored **Futures**: an object holding a value which may become available at some point in time. This value is usually the result of some other computation. It promotes task asynchronism and composition, **Promises**: a writable, single-assignment container object, which completes a future and **Observables**: An event stream obtained by combining the Iterator and the Observer patterns. It allows for asynchronous, ordered processing of a sequence of events with a publisher and subscriber interface. The abstract concepts defined were composability and immutability and finally the frameworks presented that support the reactive paradigm, such as the **Reactive extensions** which are asynchronous and non-blocking. While they do scale up to use multiple cores on a machine, none of the current implementations scale outward across nodes. They also do not provide any native mechanisms for fault tolerance, and it is up to developers that use them to write their own constructs to handle failure that may occur. And **Akka**, which is asynchronous, non-blocking and support message passing since it is based on the Actor model. It scales up to use multiple cores on a machine. Moreover, it's implementation scales outward across nodes and provides supervision mechanisms as well in support of fault tolerance.

Chapter 4

Problem definition

This chapter gives an overview of the state-of-the-art and pinpoints the problem which we'll focus on in section 4.1. It later presents the epistemological stance on the matter and formally describes the main thesis in sections 4.2 and 4.3 respectively. Finally, it describes the approach and steps taken in the process in section 4.4.

4.1 OVERVIEW

Reactive systems have become more and more needed over the years. The reason for that is mostly driven by a change in requirements as shown in Figure 4.1. In the past, a typical large system would have tens of server nodes, its response times would be measured in the order of seconds, maintenance downtime would last for hours per month and data volume would be measured in the order of GBs. Nowadays a large system has thousands of server nodes, users expect sub-second response time — in the order of milliseconds — there should be no maintenance downtime and the system should be available at all times, and the data volume has reached the order of TBs and going into PBs.

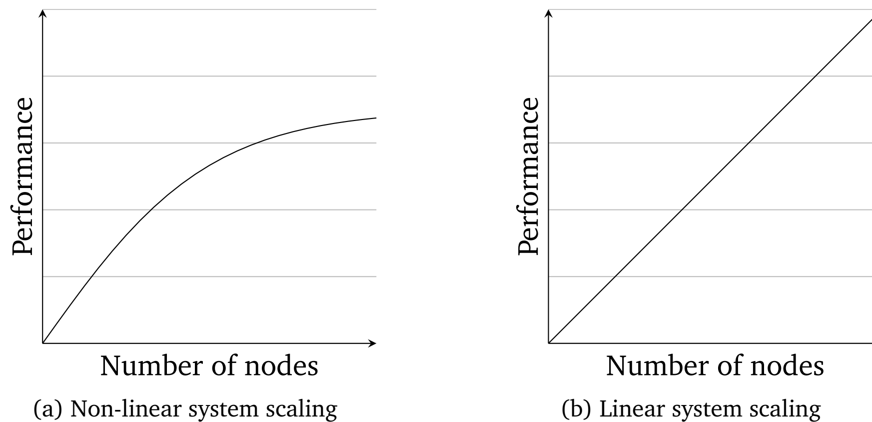
Table 4.1: Example of system requirements in the past and now.

	10 years ago	Now
Server nodes	10's	1000's
Response times	seconds	milliseconds
Maintenance downtimes	hours	none
Data volume	GBs	TBs $\rightarrow$ PBs

Traditional, non-reactive systems, when subjected and exposed to a larger working environment, expose the following flaws:

- **Difficulty scaling linearly.** The underlying mechanisms — that deal with concurrency — of a non-reactive architecture do not allow for linear scalability. While for

a low number of nodes, adding an extra one tends to double the performance, the same doesn't hold for when the node count is already high and we often obtain a curve similar to the one in figure 4.1a. However, our goal, is that no matter how many nodes are being used by the system, adding one more should struggle to linearly influences performance and obtain a curve similar to the one in figure 4.1b.



(a) Non-linear system scaling

(b) Linear system scaling

Figure 4.1: Example of linear vs non-linear scalability

- **One thread per request model.** They often use the one thread per request, which means that each new requests that arrives, a new thread is spawned to handle it, thus simulating concurrent processing. However, this approach causes a lot of overhead and memory usage, which ultimately leads to less capability to multiple concurrent requests [Keg04].

- **Pull instead of push model.** In the Pull model, the server must be interrupted continuously to deal with pull requests and can easily become a scalability bottleneck with large client populations [AFZ97]. In this model, it is the client who is in control of the data flow even though the business logic is on the host. Contrarily, in the Pull model, clients monitor the broadcast and receive the items they require, while those are broadcasted by the host when adequate.

- **Shared mutable state.** Traditional, non-reactive, server-side architectures rely on shared mutable state and blocking operations on a single thread. Sharing mutable state requires synchronization, which introduces accidental complexity and non-determinism, making the program code hard to understand and maintain. Putting a thread to sleep by blocking uses up a finite resource and incurs a high wake-up cost.

Such shortcomings demand new types of architectures that are able to cope with an increasing demand in both quantity and performance. Reactive systems as seen in section 2.1, suggest to do just that. But, if building a such a system from the ground-up is

a demanding task, what to say about evolving non-reactive systems with the aforementioned problems to a reactive architecture?

The solution must be described in a sufficiently generic manner as to be applicable in a wide variety of contexts within the non-reactive architecture scope. The state of the art on these transformations, or *refactorings*, whose endpoint system is a reactive architecture is limited and near non-existent. Therefore our main focus on this dissertation is the contribution to the completion of such.

The transformation itself as a whole, cannot be direct from a system $A \rightarrow$ system B. It has to be a series of iterative and incremental *refactorings*, such as: $A \rightarrow A' \rightarrow A'' \rightarrow \ldots \rightarrow B$, that do not break the semantics of the underlying system and gradually push a non-reactive systems to the boundaries of a reactive one.

4.2 EPISTEMOLOGICAL STANCE

In order to understand the way software engineers build and maintain complex and evolving software systems, researchers need to focus beyond the tools and methodologies; they need to delve into the social and their surrounding cognitive processes vis-a-vis individuals, teams, and organizations. In this sense, research in software engineering is regarded as inherently coupled with human activity, where the value of generated knowledge is directly linked to the methods by which it was obtained. Because the application of reductionism to assess the practice of software engineering, particularly in field research, is very complex (if not unsuitable), we suggest that the presented research is to be aligned with a pragmatist view of truth, valuing acquired practical knowledge. In other words, we choose to use whatever methods seemed more appropriate to prove — or at least improve our knowledge about — the questions here raised. Formally, a systematic scientific approach based on this epistemological stance requires the use of mixed methods [Fer10], among which are *(a)* (Quasi-)Experiments, used to primarily assess exploratory or very confined questions, and are suitable for an academic environment and *(b)* industrial Case-Studies, as both a conduit to harvest practical requirements, as to provide a tight feedback and real-world application over the conducted investigation.

4.3 MAIN THESIS

It is our belief that the presented problems could be efficiently mitigated with a migration to a reactive architecture. Notwithstanding the characteristics presented in section 2.1, we do not consider part of this dissertation to prove the validity of a system based on a reactive architecture. Instead, we assume that: (i) the context of this dissertation should be able tackle betfair's current system (section 2.6), (ii) there are problems the current, non-reactive, architecture cannot mitigate, and (iii) the system we're leaning *towards*, provides mechanisms that mitigate with a higher degree of efficiency the problems described.

Should one agree with the aforementioned premises, our fundamental main thesis may be stated as:

Given two systems A and B, where A is non-reactive and B is reactive, what are the set of transformations $(T_1, T_2, \ldots, T_n)$ *that transform A into B while simultaneously preserving the original key functionalities?*

Consequently, the main goal of this dissertation is the creation of a catalog of *Refactoring to patterns* that focus a reactive system as an endpoint. More specifically, we assume the following hypothesis:

A non-reactive system upon applying a series of Refactoring to patterns to a reactive system should, gradually, begin to benefit of greater scalability, better adaptability and better performance. This characteristics should be quantified in terms of: (i) better resource usage, (ii) linearly scaling, (iii) lower latency, and (iv) number of concurrent requests handled in simultaneous. Such an architecture should present a better overall system performance than the previous one, and present the properties: be event-driven, scalable, resilient and responsive.

This statement uses terms whose meaning may not be consensual, and therefore lead to questions that deserve further discussion:

1. **What should be understood by *Refactoring to patterns to a reactive system*?**
   Based on Gamma et al [GVHJ95] and Buschmann [BMR+96], the refactorings shall be structure in the same way as presented in Section 2.3.

2. **What should be understood by *better resource usage*?**
   The concept of better resource usage is linked to external metric that can be extracted from the system, such as: *cpu* usage and *RAM* consumption.

3. **What should be understood by *linear scaling*?**
   Controversially to a non-reactive architecture, the addition of more machines or the improvement of the current ones, should linearly improve the performance of the system.

4. **What should be understood by *lower latency*?**
   The time elapsed between the request by part of the *end-user* and the reply given by the system until it reaches the latter.

5. **What should be understood by *overall system performance*?**
   The measurement of the systems performance, by its external metrics, should increase positively after the transformation.

Finally, although the dissertation context is scoped, it is our belief that the catalog could likely be applied to system of similar characteristics and would likely yield similar results.

4.4 APPROACH

The approach for this problem starts by analyzing in detail the reactive frameworks seen in the state of the art chapter 3 and the their underlying concepts and technologies. This will be the foundation for the target system our transformation will aim towards.

The process to develop the catalog will happen by steps as shown in Figure 4.2 and is as follows:

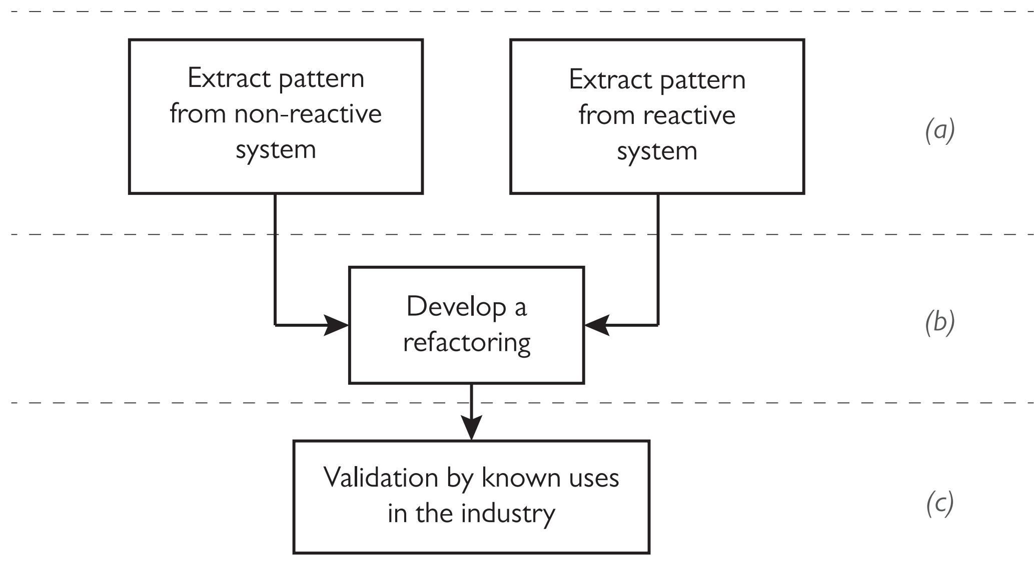


Figure 4.2: Approach for creating the catalog of patterns

(a) **Identifying and extracting patterns from reactive and non-reactive systems**
In the first phase we will analyse non-reactive systems such as Blip along with others and reactive systems in order to identify patters. We are looking for pattern duality (patterns that are the opposite of each other in a specific context) which we can use to create our transformation patterns. These patterns can be ones already formalized in the literature (e.g. the SINGLETON or OBSERVER [GVHJ95]) or patterns that we will identify ourselves. This process follows an inductive approach for detecting patterns as described in section 2.5.

(b) **Develop a refactoring that transforms a non-reactive pattern into its reactive dual**
At this point, we should have a clear idea on the starting point and the ending point of the transformation pattern. If the patterns are in fact duals, we will then create a Pattern in the form of *"Pattern A to Pattern B"*. However in the event that the starting point is non-existent and we only have an relevant ending point, the transformation will not be based on the duality of two patterns. Instead, the pattern will take the form of *"Add Pattern B"*.

(c) **Validation by known uses in the industry**

A pattern is not tried and true *per se* and thus after the development of the transformation, we must now provide evidence to support their truthfulness. Different approaches were considered for this purpose:

1. Informal proofs that guarantee that a set of functional tests pass post transformation;
2. Empiric analysis showing that when the set of transformations is applied to a set of (statistically significant) of systems, they exhibit the same external behavior.

The first approach would require a set of systems with such properties and the case study (section 6 did not complied with these requirements). For the second approach, we formulated two possibilities: (a) Apply the transformations to non-reactive systems, while obtaining metrics of the system prior and after the transformation. This approach also incited some questions. How many systems would be required for a sufficient sample? As seen by the rule of three (section 2.5), it informally suggests that the minimum would be 3 systems. (b) Provide industry known uses cases of such transformation in order to support the existence of a pattern since it is observed throughout several different systems.

Due to the time constrains for the completion of this dissertation, we only provided known uses observed in the industry for each transformation while the remaining should be added future work.

Chapter 5

Patterns

In this chapter we describe the anatomy of a pattern in section 5.1, followed by an enumeration of the forces relevant for the context of our patterns in section 5.2, and finally the detailed transformation patterns and their respective relationships in section 5.3.

5.1 ANATOMY

The Pattern anatomy can be structured as we see in the following sections:

1. **Summary.** An introductory paragraph, which sets the intent of the pattern.

2. **Context.** The scenario in which the problem occurs and where this pattern can be applied.

3. **Problem.** Describes the problem a pattern manifests in a scoped context.

4. **Forces.** Each pattern has a set of forces, things that should be weighted in order to achieve a good solution.

5. **Solution.** Describes the actions necessary to solve the stated problem along with the resolution of the forces.

6. **Example.** A situation where the pattern has been applied, with a possible graphical representation, snippets of code or a hypothetical example.

7. **Implementation notes.** Some additional concerns on effectively applying the pattern and dealing with lower-level issues, such as performance and memory consumption.

8. **Known Uses.** A pattern is a pattern because there is empirical evidence for its validity. This section points to systems where the pattern has been previously observed.

5.2 FORCES

There are a set of forces that are relevant for the context of our patterns, and each of our patterns will be weighed with them in mind. The relationship between the following forces observed in the patterns (each pattern only weights in the relevant forces for its context **prior** to the application of the transformation while the force resolution is implicit in the solution **after** the said application) is presented in figure 5.1:

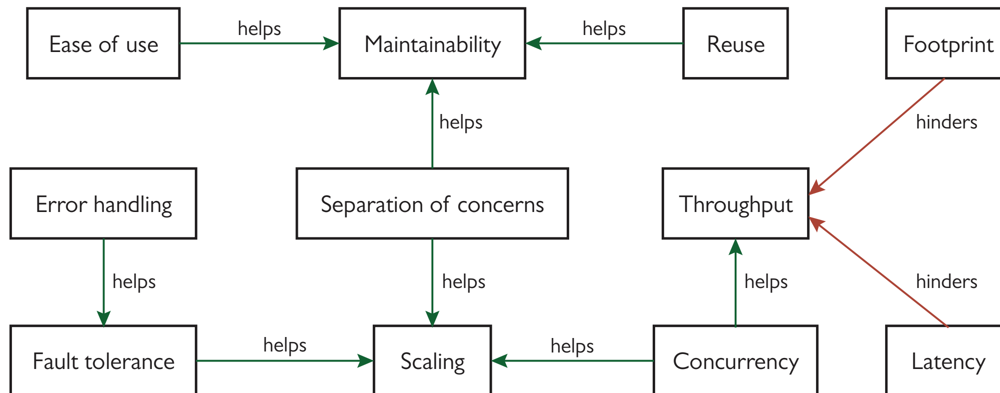


Figure 5.1: Relation between the different forces

- **Ease of use**. The difficulty of using and understanding an abstraction or a technology. It also includes the change in mindset that may have to occur when transitioning from to new, more powerful abstractions that aim to produce the same results.
- **Maintainability**. The amount of effort required to maintenance the code over a period of time, including refactoring, addition of new features and bug fixes.
- **Reuse**. The possibility to reusing existing artifacts, or knowledge, to build or synthesize new solutions, or to apply existing solutions to different artifacts.
- **Separation of concerns**. This is a general design force the establishes the fact that a particular functionality of a systems should be the concern of a different component.
- **Error handling**. The difficulty to handle errors while using an abstraction. If they are composable and reusable throughout our code.
- **Fault tolerance**. If an error happens, the system should try its best to recover without human intervention before giving up and informing the user mitigating the results of inevitable programming bugs, hardware failures, et al., while isolating them from the rest of the system.
- **Scaling**. The ability to scale vertically and/or horizontally using the abstraction.

- **Throughput**. The amount of requests per second a system can withstand simultaneously.
- **Concurrency**. The abstraction's relationship with concurrency problems, if it accepts the underlying conditions and provides methods that mitigate them. It is manly relevant due to performance.
- **Footprint**. The amount of required memory and resources needed for the abstraction.
- **Latency**. The amount of time occurred in communications. This time is measured from start to end and therefore is affected by intermediary delays.

5.3 TRANFORMATIONS

This section presents the 11 transformations patterns for a reactive application. These are structured in: *Summary*, *Context*, *Problem*, *Forces*, *Solution*, *Implementation notes* and *Known uses*; as described in the previous — "Anatomy of a pattern" — section.

The relationship between the following patterns is presented in figure 5.2:

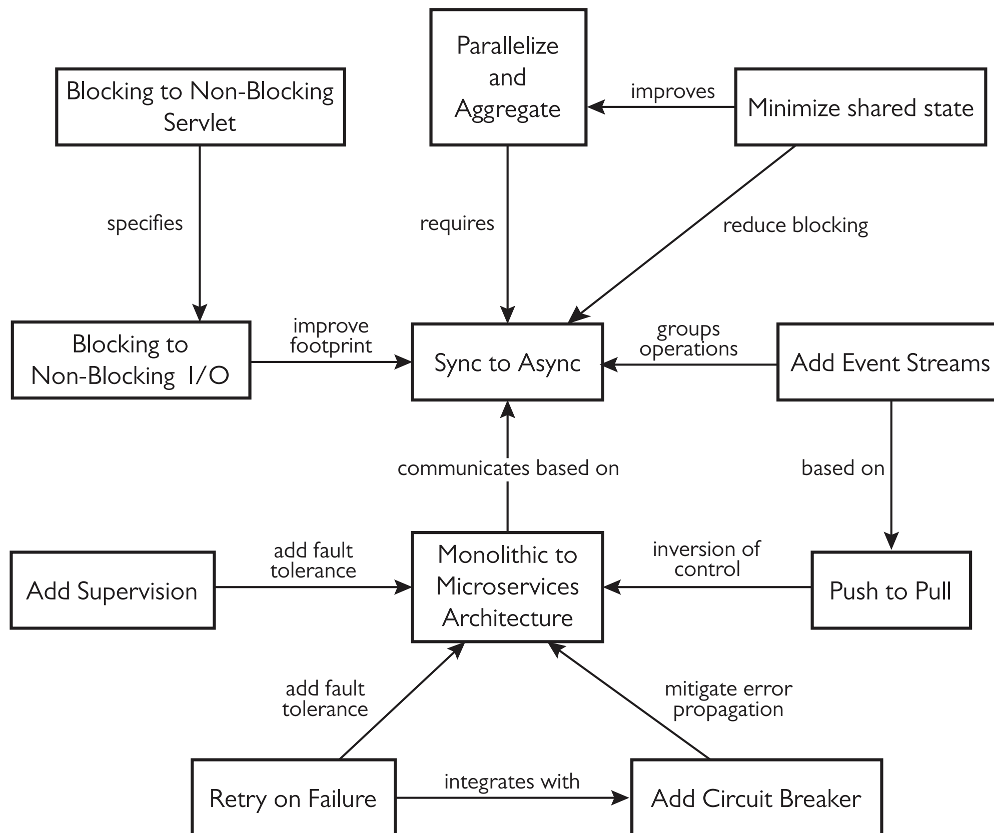


Figure 5.2: Relation between the different patterns

1. **Synchronous to Asynchronous processing (5.3.1).** The purpose of this pattern is migrate from a synchronous to a asynchronous application. It provides better alternatives to handle concurrency.

2. **Blocking I/O to Non-blocking I/O (5.3.2).** The purpose of this pattern is to migrate from blocking I/O application to non-blocking I/O application.

3. **Blocking Servlet to Asynchronous Servlet (5.3.2).** The scope of this pattern is entirely restricted to the java programming language and its objective is to migrate from a blocking servlet to a asynchronous servlet.

4. **Parallelize and aggregate (5.3.4).** This pattern illustrates the advantages of parallelizing tasks and aggregating them posteriorly while taking advantage of the lack of dependency among them.

5. **Add Circuit Breaker (5.3.5).** This pattern focuses on handling transient faults when dealing with remote services and mitigate the propagation of errors by containing them. This pattern can improve the stability and resiliency of an application.

6. **Pull to Push Model (5.3.6).** This pattern scope aims to describe beneficial aspects where an migration to a push model from the traditional pull model.

7. **Monolithic to Microservice based Architecture (5.3.7).** The application of this pattern is related to the system's architecture and it's goal is to migrate an monolithic architecture, where the application is built as a single unit, to a microservice architecture, where the application is built as a composition of small services.

8. **Add Event Streams (5.3.8).** This pattern focus on grouping correlated data or events in a single stream while inverting the flow of control to a push model.

9. **Minimize Shared State (5.3.9).** This pattern targets the minimization of shared state within a application, mitigating the need for synchronization.

10. **Add supervision (5.3.10).** This pattern focuses on giving entities the responsibility of monitoring the life cycle of their subordinates, thus improving the application's fault tolerance.

11. **Retry on failure (5.3.11).** This pattern focuses on handling anticipated, temporary failures when it attempts to connect to a service or resource by transparently retrying an operation that has previously failed in the expectation that the cause of the failure is transient.

5.3.1 Synchronous to Asynchronous processing

The purpose of this pattern is migrate from synchronous code to asynchronous code processing. It provides better alternatives to handle concurrency.

CONTEXT

This pattern targets, especially, applications that need to perform several tasks simultaneously.

Web servers are perfect examples of applications screaming for an asynchronous approach. Their purpose is to receive requests and respond accordingly. These requests arrive at a nondeterministic time and volume and need to be processed as fast as possible to reply to the client. This processing must be done asynchronously and concurrently in order to process large amounts of clients since sequential execution would prove entirely impractical.

PROBLEM

Sequential execution of functions is well-supported by all popular programming languages out of the box, where any thread performing an synchronous computation will block and wait for the result, however when it comes to asynchronous execution usually needs some extra thought and library support.

Languages that lack this feature are usually stuck emulating it by dealing with low-level threads when implementing their functions to achieve concurrency and asynchronism. Dealing with concurrency at such a low level requires more effort on the developer, usually induces sources of errors (e.g. race conditions, critical regions, deadlockings) and results in increased resource usage.

Furthermore, in order to achieve a higher level of concurrency in this thread per request model, a high number of threads will have to be used. Since threads have a significant footprint associated with them, this will effectively drain the server resources and hinder its performance.

Another important aspect of reactive application is resilience, when we are coding in a synchronous fashion we can handle failure by using *try and catch* which catches exceptions propagated through the stack but in the asynchronous world the context is lost and this method is not longer valid, and when the the exceptions do undoubtedly rise, its handling proves to be more complex.

FORCES

(−) Concurrency. Synchronous programming is not the best suited model for dealing with concurrency as the execution will start and block the current thread while waiting for the result.

(−) Throughput. While a thread waits for the expensive execution to return its result, the OS can exchange active threads to promote concurrency, but this has overhead costs and hinders throughput due to thread context switching and cache invalidation.

(−) Latency. Thread blocking on execution and the lower throughput due to the OS exchanging active threads and consequently cache invalidation leads to poorer latency.

(+) Maintainability. Synchronous programming and its sequential execution model provides an easy to reason with concept that in turn increase maintainability. Furthermore, error handling in sequential execution is easier to tackle.

(+) Ease of use. The synchronous sequential model and its typical imperative programming style provides a familiar thinking model that results in ease of use.

SOLUTION

The solution relies on the use of non-blocking, asynchronous primitives that enable us to do asynchronous processing with ease. Several alternatives allow us to achieve this purpose but we will focus on the more popular, as observed in our state of the art review.

Futures as seen in section 3.1.1.1, are encapsulated asynchronous computations that hold a value and which may become available at some point in time. When created they return immediately and the actual computation is done in a different execution context allowing the caller to continue normal sequential execution. The advantage to this approach is that futures are composable and callbacks can be attached to them on different events such as `onCompletion` or `onFailure`. Failure can be handled by the latter, and on composed futures the error will also be propagated providing a reliable method to handle failure.

To be able to execute concurrently, Futures run in a different execution context than the caller. Commonly these consist of a threadpool[1] but the underlying implementation of execution is free to be modified at will. As such, we can think of futures as tasks and a threadpool, as their executor. This enables the creation of asynchronous code and enables us to better make use of the host's resources. However, an important downside to tackle is code debugging. Since execution is not sequential anymore, conventional methods for debugging prove useless and different approaches must be taken when debugging asynchronous code.

EXAMPLE

Suppose we have 3 tasks that we want to perform sequentially and each of them depends on the previous one. In the end, after all have finished we just want to do something with result. In this case, we have tasks A, B and C and our goal is to reply with the result of task C that depends on results of task B which depends on the result of task A.

[1] A collection of threads on which work items can be scheduled. The threads in the thread pool are typically managed by a class library (or operating system) rather than by the application.

An synchronous sequential implementation can be observed in source 5.1.

```
//def replyA()         : replyA
//def replyB(replyA a) : replyB
//def replyC(replyB b) : replyC

val a : ReplyA = taskA()
val b : ReplyB = taskB(a)
val c : ReplyC = taskC(b)
respondWith(c)
```

Source 5.1: Syncronous composition of tasks

In the example above, the caller would initiate task *A* and block, waiting for the result; it would then start task B with the result from *A*, blocking until its completion; and the same for task *C* using the result from task *B*. Finally it would reply with the result from task *C*. It is important to note that the caller would be blocked in during these tasks unable to perform any other computations.

As we move towards an asynchronous implementation we would transform our task functions so they return a Future instead of the result directly. In source 5.2, we see the resulting transformation where the caller would schedule all the tasks and the attach the reply to be executed asynchronously. The result is an application which is not blocking during the execution of these tasks and is free to perform other computations.

```
//def replyA()         : Future[ReplyA]
//def replyB(ReplyA a) : Future[ReplyB]
//def replyC(ReplyB b) : Future[ReplyC]

//monodic implementation
a flatMap b flatMap c map respondWith

//or with a for comprehension
for {
  a <- taskA()
  b <- taskB(a)
  c <- taskC(b)
} yield respondWith(c)
```

Source 5.2: Asynchronous sequential composition of tasks using Futures

IMPLEMENTATION NOTES

As mentioned above, the future code is run by a executor. Whatever the selected executor, the signaling of the completion of a task can be achieved in multiple ways.

This can be done by means of callbacks, which, in essence means that on the completion of that task the defined callback will be invoked and the thread will be released.

Alternatively, this can be achieved by an event loop which continuously queries the executor and dispatches the callback of the completed futures.

KNOWN USES

Desktop applications often can be given the luxury of performing long, time-consuming synchronous tasks but even now, that is changing and they try to harness all potential of the underlying host's hardware (e.g. multi-core CPUs) by parallelizing tasks and making them asynchronous.

Web servers built synchronously (e.g. pure PHP) have difficulties scaling without having the increase the resources exponentially whereas those built with asynchronously in mind (e.g. Akka) scale much better.

Another type of application is a database driver for example. Whereas in the past, synchronous communication with the database was acceptable, in today's world of concurrent programming, we are transitioning to asynchronous access to the database (the operations at the database level may still remain blocking in nature but the communication as seen by the application is asynchronous), e.g. MongoDB [Mon14] and PostgreSQL/MySQL [Git14b] drivers. This transition proves hard because of legacy reasons but ideally, this is the model we want to end up.

5.3.2 Blocking I/O to Non-blocking I/O

The purpose of this pattern is to migrate from blocking I/O code to non-blocking I/O code.

CONTEXT

The input/output models can be divided into two groups: basic I/O and advanced I/O. Basic I/O is synchronous and blocking. An I/O operation is synchronous if the thread initializing the I/O operation cannot switch to other operation until the I/O operation is finished. Whereas a asynchronous I/O operation — which needs the support of the underlying operating system — returns immediately and the thread initializing the operation can perform other operations, being later called back when the operation finishes. Even though synchronous solutions may be more performant in numerous cases, given the scope of our target applications, we are sure to be gaining from this transformation. Since we typically have multiple tasks to dispatch at any point in time, which are I/O bound instead of CPU bound, a non-blocking I/O would be the wiser choice.

PROBLEM

A function or method is blocking if it does not return until its execution either successfully finishes or encounters an error. This means that the thread allocated to the request must be held during that wait along with all its resources: kernel thread, stack memory, wasting the system resources to hold these resources while waiting. Furthermore, I/O calls are

extremely costly in terms of CPU cycles. So, if our intention is to process other operations concurrently with the I/O operation we will have to allocate another thread to wait on it along with all the disadvantages it is associated with.

We can easily argue that this approach is costly and does not scale well and would be impractical with a large number of operations in simultaneous.

FORCES

(−) Concurrency. Executing blocking I/O calls will block the current thread until it returns, leaving the concurrency handling to the OS's scheduler which adds overhead.

(−) Footprint. Maximizing the amount of performed I/O tasks while performing Blocking I/O takes its toll on resources. Blocked threads will still hold resources which are not being used.

(+) Maintainability. An sequential execution model is easier to reason with, debug and handle exceptions.

SOLUTION

We will present three ways to make advanced I/O. The first approach is to construct a loop to keep trying an I/O option while catching I/O errors until it succeeds. This approach is called polling and it wastes CPU time.

The second approach is I/O multiplexing uses `select()`-like system functions and is supported by most operating systems. The descriptors of connections can be registered on a selector, which calls `select()` to check if there is any I/O event for each of the descriptors. So a thread initializing an I/O operation can delegate the job to a selector and switch to other job. Note that `select()` is blocking until one of the registered descriptors has an event or timeout happens.

The third approach is asynchronous I/O (AIO), which is both asynchronous and non-blocking and requires the support of the underlying operating system.

This approach allows us to ask for the resource (which will be loaded in the background) and immediately resume so we can continue processing other tasks. The result is a lower footprint, specially in memory from the saved blocked threads, and better scalability.

EXAMPLE

We can see the blocking and non-blocking approaches for I/O in source 5.3. The blocking and synchronous code snippet will open, and block while reading a file, process it in `parseFile` and only when this returns, will the `otherTask` take place. On the other hand, the non-blocking and asynchronous will open and read the file, process it with `parseFile` when it is finished reading while the `otherTask` executes concurrently.

```
//Synchronous approach
val content : String = readFIle(path)
parseFile(content)
otherTask() //only executes after parsing the file

//Asynchronous approach
val content : Future[String] = readFileAsync(path)
content.onComplete( content => parseFile(content))
otherTask() //executes immediately regardless of the above
```

Source 5.3: Examples of Synchronous and asynchronous approaches for file reading

IMPLEMENTATION NOTES

Programming languages have different for support for this. Non-blocking I/O can be supported by the underlying Operating system or not. In the latter, there will have to be a 1-to-1 correlation between pending asynchronous I/O operations and threads (or at least managed threads) waiting for those I/O operations to complete. In the former, the OS will take the responsibility of handling the I/O event and notify us when it is completed. In both cases, after the completion notification, the respective callback is invoked.

KNOWN USES

Netty [Net13c] is an asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers and clients. It is a very popular stack and it's used by great number of even more popular applications [Git14a] such as Akka, elasticSearch, Couchbase, etc.

5.3.3 Blocking Servlet to Asynchronous Servlet

The scope of this pattern is entirely restricted to the Java programming language and its objective is to migrate from a blocking servlet to a asynchronous servlet.

CONTEXT

A servlet is a Java programming language class used to extend the capabilities of a server. Although servlets can respond to any types of requests, they are commonly used to extend the applications hosted by web servers, so they can be thought of as Java applets that run on servers instead of in web browsers.

Many web applications need to wait at some stage of a HTTP request, for example:

- Waiting for a resource to be available before processing the request (e.g. thread, JDBC Connection)
- Waiting for an application event in an AJAX Comet application (e.g., chat message, price change)

- Waiting for a response from a remote service (e.g., RESTful or SOAP call to a web service).

PROBLEM

The Servlet API *(<=2.5)*, only supports the synchronous call style, and any time consuming call will result in thread blocking. Unfortunately this means that the thread allocated to the request must be held during the duration of the call along with all its resources: kernel thread, stack memory and often pooled buffers, character converters, etc. And it is wasteful of system resources to hold these resources while waiting.

FORCES

(−) Concurrency. The blocking servlet will execute blocking I/O calls that block the current thread until it returns, leaving the concurrency handling to the OS's scheduler which adds overhead.

(−) Footprint. Thread blocking holds back resources while simply waiting for the execution to complete.

(−) Scalability. I/O bound applications suffer in terms of scalability as I/O becomes a bottleneck.

(+) Maintainability. Once again, the execute and wait for the reply, as well as sequential execution lead to a simple model to maintain. Servlets also divide the business logic layer from the the implementation layer, giving the developer an higher abstracting to deal with while hiding the underlying implementation.

SOLUTION

Throughout the years, the servlet API has been evolving and adding support for more capabilities. The servlet API *(>3)* introduced a change that allows a request to be dispatched multiple times to a servlet. If the servlet does not have the resources required on a dispatch, then the request is suspended (or put into asynchronous mode), so that the servlet may return from the dispatch without a response being sent. When the waited-for resources become available, the request is re-dispatched to the servlet, with a new thread, and a response is computed.

EXAMPLE

Consider a web application that accesses a remote web service (e.g., SOAP service or RESTful service). Typically a remote web service can take hundreds of milliseconds to produce a response — eBay's RESTful web service frequently takes 350ms to respond with a list of auctions matching a given keyword — while only a few *10s* of milliseconds of CPU time are needed to locally process a request and generate a response.

To handle 1000 requests per second, which each perform a *200ms* web service call, a web application would need $\frac{1000*(200+20)}{1000} = 220$ threads and *110MB* of stack memory. It would also be vulnerable to thread starvation[2] if bursts occurred or the web service became slower.

If handled asynchronously, the web application would not need to hold a thread while waiting for web service response. Even if the asynchronous mechanism cost *10ms* (which it does not), then this webapp would need $\frac{1000*(20+10)}{1000} = 30$ threads and *15MB* of stack memory. This is a 86% reduction in the resources required and *95MB* more memory would be available for the application.

Furthermore, if multiple web services request are required, the asynchronous approach allows these to be made in parallel rather than serially, without allocating additional threads.

IMPLEMENTATION NOTES

Sometimes the update to a servlet API version which supports non-blocking I/O, namely *(>3)*, is impossible (e.g. for legacy reasons of the system). In this cases, we can achieve this feature by using vendor implementations; for example, Jetty 6 offers asynchronous support throught the form of *continuations*.

KNOWN USES

Betfair made the transition from blocking to asynchronous servlets. They started out by using an API version under 3.1, later started using Jetty's continuations and now use the 3.1 API.

5.3.4 Parallelize and aggregate

This pattern illustrates the advantages of parallelizing tasks and aggregating them posteriorly while taking advantage of the lack of dependency among them.

CONTEXT

In many cases there is one possibility for latency reduction which immediately presents itself. If for the completion of a request several other services must be involved, then the overall result will be obtained quicker if the other services can perform their functions in parallel. This requires that no dependencies exist between the tasks at hand.

PROBLEM

While performing a task which is composed of several sub-tasks, these can have dependencies on one another or be completely independent. Often, we deal we both cases. Tasks

[2] The process is perpetually denied necessary resources

which we cannot start because it requires data that we will only have access after another task terminates. And tasks whose input data is not dependent from the completion of other tasks or do not require input at all.

A problem arises when the execution of these tasks, independently of their type described above (with or without dependencies), in performed in a sequential manner as shown in Figure 5.3.

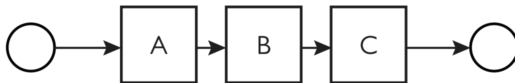


Figure 5.3: Sequential execution of tasks

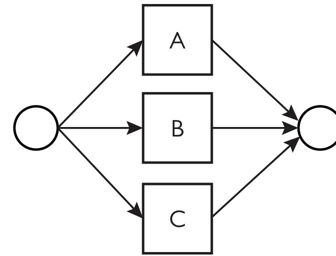


Figure 5.4: Parallelized execution of tasks

We can observe that the total latency for this sequence of tasks is equal to the sum of the latency of A, B and C, plus any inner processing delays that occur at each node.

FORCES

(−) Concurrency. The way the pattern executes tasks sequentially prevents optimizations such as parallelization.

(−) Throughput. Due to the sequentially of the tasks being performed, we do not take full advantage of the available resources and thus hinder throughput.

(+) Maintainability. Once again, the execute and wait for the reply, as well as sequential execution lead to a simple model to maintain.

(+) Ease of use. When performing a set of tasks sequentially, we fire one and wait for the result before firing the next one. This is a predictable, deterministic model, where exception handling can be achieved by the familiar *"Try and Catch"*.

SOLUTION

The solution targets the tasks that have inter-dependencies among themselves. The tasks whose execution have no dependencies can be initiated independently and in parallel instead of sequentially and later posteriorly aggregated back together as depicted in Figure 5.4.

While in Figure 5.3 the total latency will depend on the sum of the three individual latencies i.e. $totalLatency = latency(A) + latency(B) + latency(C)$, in Figure 5.4, the total latency is equal to the maximum latency of either of the subtasks i.e. $totalLatency = max(Latency(A), Latency(B), Latency(C))$.

Coordinating this type of tasks described above can be achieved naively by using temporary variables that keep track of the tasks that haven't completed yet, and using that to execute the aggregation method when the value drops to zero. However, we are best suited using abstractions that support asynchronous methods and offer functions that allow us to do just this. Primitives like the Future (Section 3.1.1.1), allow us to compose the tasks in complex scenarios and better handle failure.

EXAMPLE

Consider the following example composed of dependent and independent tasks. The execution flow consists on obtaining data from one external service; making use of that to obtain data from two other external services; and finally perform some computation on the retrieved data and returning it as illustrated in Figure 5.5.

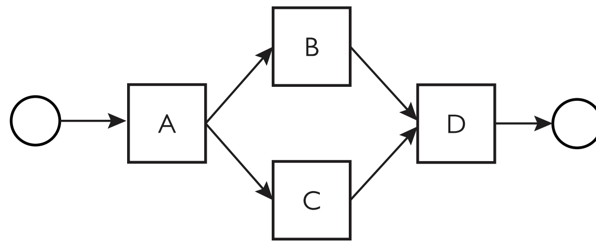


Figure 5.5: Diamond execution of tasks

Both tasks *B* and *C* depend on task *A* but neither *B* depends on *C*, nor *C* on *B*, and finally *D* depends both on *B* and *C*. This is we call a diamond execution, as we can parallelize intermediary tasks but ultimately we need them aggregated at the end. The idea here is to reduce the overall latency of the main task.

The implementation would take the following form using Scala's syntax as seen in source 5.4.

```
//def taskA()          : Future[ReplyA]
//def taskB(ReplyA a)  : Future[ReplyB]
//def taskC(ReplyA a)  : Future[ReplyC]

val replyA = taskA()
val taskBwithA = replyA flatMap taskB
val taskCwithA = replyA flatMap taskC

for {
  replyB <- taskBwithA
  replyC <- taskCwithA
} yield processData(replyB,replyC)
```

Source 5.4: Example of the parallelization and aggregation of tasks

KNOWN USES

The task of paralleling and aggregating has become very natural in scalable applications. It's point is to take maximum advantage of our resources given the task limitations (in this case, the inter-dependency of tasks that are targeted for parallelization). In fact, this pattern has become so common that in Akka, for example, we can extend an actor with the *"Aggregator"* trait to simplify and ease the complexity of these kinds of tasks [Akk14].

5.3.5 Add Circuit Breaker

This pattern focuses on handling transient faults when dealing with remote services and mitigate the propagation of errors by containing them. This pattern can improve the stability and resiliency of an application.

CONTEXT

Nowadays, large systems are not monolithic, they are built using other services where each one of them has a well defined part to play. Moreover, most of the times, they must also resort to external services to satisfy their needs.

Therefore, for the completion of a single request by part of the user, several requests to multiple internal services must be invoked until all the data required to process the request is obtained.

PROBLEM

When users are momentarily overwhelming a service, then its response latency will rise and eventually it will start failing. The users will receive their responses with more delay, which in turn increases their own latency until they get close to their own limits. In order to stop this effect from propagating across the whole chain of user–service relationships, the users need to shield themselves from the overwhelmed service during such time periods.

FORCES

(−) Fault tolerance. Without a circuit breaker, we propagate errors between different services and we have to deal with cascading failures, thus diminishing the application's fault tolerance.

(−) Latency. Sending requests to a service that is overwhelmed or simply is down, stacks up latency for a reply to arrive. Moreover, if we try this, along with retry strategies the latencies become even higher.

(−) Throughput. The accumulation of requests on a overwhelmed service (e.g. network congestion) diminishes its throughput and therefore affects the all the other services that make use of it and their respective throughputs.

SOLUTION

The way to solve this is well known in electrical engineering: install a circuit breaker [Nyg07] as seen in Figure 5.6.

The solution relies on monitoring the time a response takes to come back when involving another service. If the time consistently rises above the allowed threshold which this user has factored into its own latency budget for this particular service call, then the circuit breaker trips and from then on requests will take a different route of processing which either fails fast or gives degraded service just as in the case of overflowing the bounded queue in front of the service. The same should also happen if the service replies with failures repeatedly, because then it is not worth the effort to send requests at all.

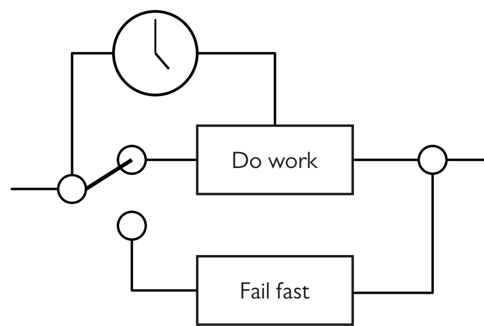


Figure 5.6: Circuit breaker diagram

This does not only benefit the user by isolating it from the faulty service, it also has the effect of reducing the load on the struggling service, giving it some time to recover and empty its queues, and to protect the downstream of components from being hammered with requests they, currently, can not comply with them in any case.

When the service has had some time to recover, the circuit breaker should snap back into a half-closed state in which some requests are sent in order to try out whether the service is back in shape. If not, then it can trip immediately again, otherwise it closes automatically and resumes normal operations. The possible states and the their transitions can be seen in figure 5.7.

EXAMPLE

In figure 5.8 we can see an interaction example between a client, a circuit breaker and a supplier. The communication between the the client and the supplier is is subjected to a middle man, the Circuit Breaker (*CB*). In the example we can see that the first request hits the *CB*, which in turn hits the supplier and the responses come back in a timely fashion meaning the supplier is operating correctly. However, on the second request we observe that the communication between the the *CB* and the supplier results in a timeout. The *CB* assuming this failure may in fact be transient, propagates the timeout reply back to client.

In the client's point of view, all he performed was a request and after a period of time, an timeout exception was replied with. In the third request sent, the interaction between the *CB* and the supplier results in another timeout, this time the *CB* assumes that the error may not be transient and thus *trips*, effectively opening the circuit, that propagates the timeout reply again. In the final request of the example, the circuit is open therefore the request fails fast, and the client gets an immediate "supplier not available" reply. Thus mitigating the timeout await time for the client, and minimizing the request load for the supplier so we can become functional faster. Further requests will continue to fail fast between the client (allowing it gracefully degrade the service if possible) and the supplier until the circuit goes to the half-open provisory state and later back to the close state again as we can see in figure 5.7.

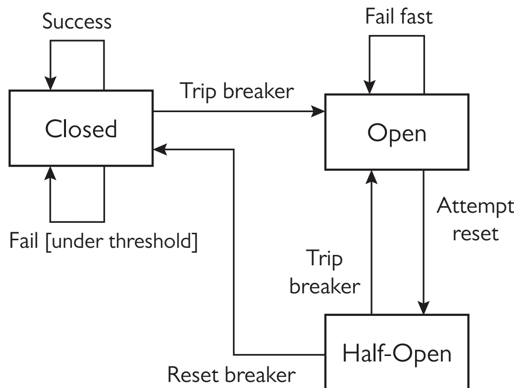


Figure 5.7: Circuit breaker state diagram

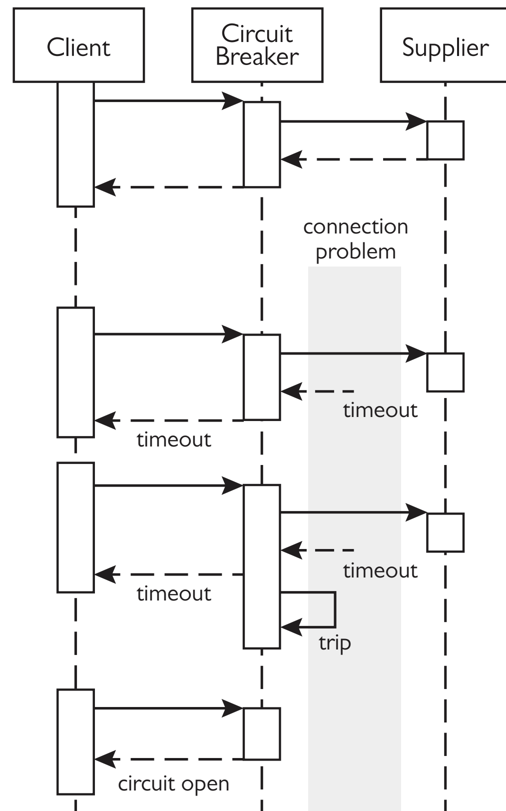


Figure 5.8: Circuit breaker control sequence example

IMPLEMENTATION NOTES

The Circuit Breaker Design Pattern should be implemented asynchronously. The reason is to offload the logic to detect failures from the actual request. This requires Circuit Breaker to use a persistent storage layer, e.g. a network cache such as Memcached or Redis, or

local cache (disk or memory based) to record the availability of what is, to the application, an external service.

Therefore, the circuit Breaker records the state of the external service on a given interval and before the external service is used from the application, the storage layer is queried to retrieve the current state.

KNOWN USES

Web applications are now commonly composed by several remote services. This communication cannot always be assumed as reliable, and as such we much create methods to mitigate failures. The use of a circuit breaker is now being observed more and more when faced with this situations. For example, Netflix —a highly scalable provider of on-demand internet streaming media— employs the circuit breaker pattern together with fallbacks as graceful degradation of service, failing silently, and failing fast in this order [Net11].

5.3.6 Pull to Push Model

This pattern scope aims to describe beneficial contexts where an migration to a push model from the traditional pull model.

CONTEXT

In software engineering, the pull model and the push model designate two well-known approaches for exchanging data between two distant entities.

Dynamic client applications often require updated data from the server to show the user the most actual information.

Refreshing the whole page would be a costly and slow solution therefore, presently, this is achieved by AJAX calls at regular time intervals, known as Time to Refresh (TTR), to the server and it occurs blindly regardless of whether the state of the application has changed.

PROBLEM

The polling model (figure 5.9) problem is that in order to achieve high data accuracy and data freshness, the pulling frequency has to be set equally high. This, in turn, induces high network traffic and unnecessary high load on both client and server. This increase in load is more noticeable in the server as it receives the requests from all the clients, while in the client perspective, we only have to considerate the load associated with their own requests. This way the solution needs more server and bandwidth which makes the solution more expensive. The application also wastes some time querying for the completion of the event, thereby directly impacting the responsiveness to the user. Mechanism such as adaptive TTR allow the server to modify the frequency on which the clients pulls for

data. This provides better results than a static TTR approach, however, it will never reach complete data accuracy, and it will still create unnecessary traffic.

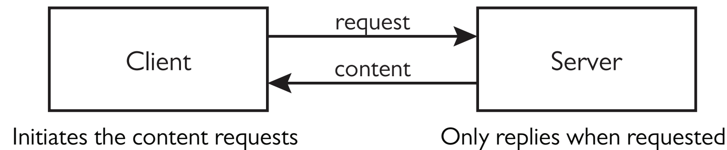


Figure 5.9: Pull model diagram

FORCES

(−) Footprint. In a pull model, the clients naively request new information from the server, resulting in more requests per second and ultimately a higher footprint. In a push model, it is the server's responsibility to notify the clients, which leads to less requests received and the possibility of optimization.

(−) Scaling. The characteristically high footprint derived from Pull models, consumes our host's resources very fast and deteriorates the scaling capacity of the system.

(−) Separation of concerns. The pull model puts the responsibility of asking for new information at the client side. But the client has no idea when new data is available and therefore must naively question the server.

(−) Latency. It puts the client of in charge of requesting data, this can result in a high load of requests sent to the server thus congestioning the network and therefore increasing the latency.

(−) Throughput. The throughput of the system is wasted on the pull model. Many requests sent to the server (e.g. requesting updates of data) may be unnecessary. This results in a high number of "useless" processed requests that hinders the actual "needed" throughput.

(+) Ease of use. For tasks that only need to be executed at well defined intervals, the pull model provides an easier to use interface. Plus, all the browsers support this approach.

SOLUTION

The solution relies on delegating the responsibility of initiating the exchange of information to the server instead of the client (Figure 5.10). When the client initiates the communication with the server, it creates a connection typically using a websocket to allow for bidirectional communication. This allows the server to send information to the clients without them, explicitly requesting it.

There are several advantages at play while using this approach. First, we avoid having the client periodically having to query the server for new information. Instead, the server

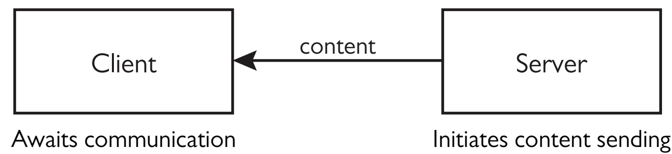


Figure 5.10: Push model diagram

knows when new information is available for the clients and only then will the communication take place. Moreover, the server can then optimize this process by batching replies to maximize throughput and footprint.

Secondly, the client becomes more lightweight with the removal of its former responsibilities.

And finally, this means less *"periodic"* requests sent to the server from the clients, resulting in better resource usage and better scaling possibilities.

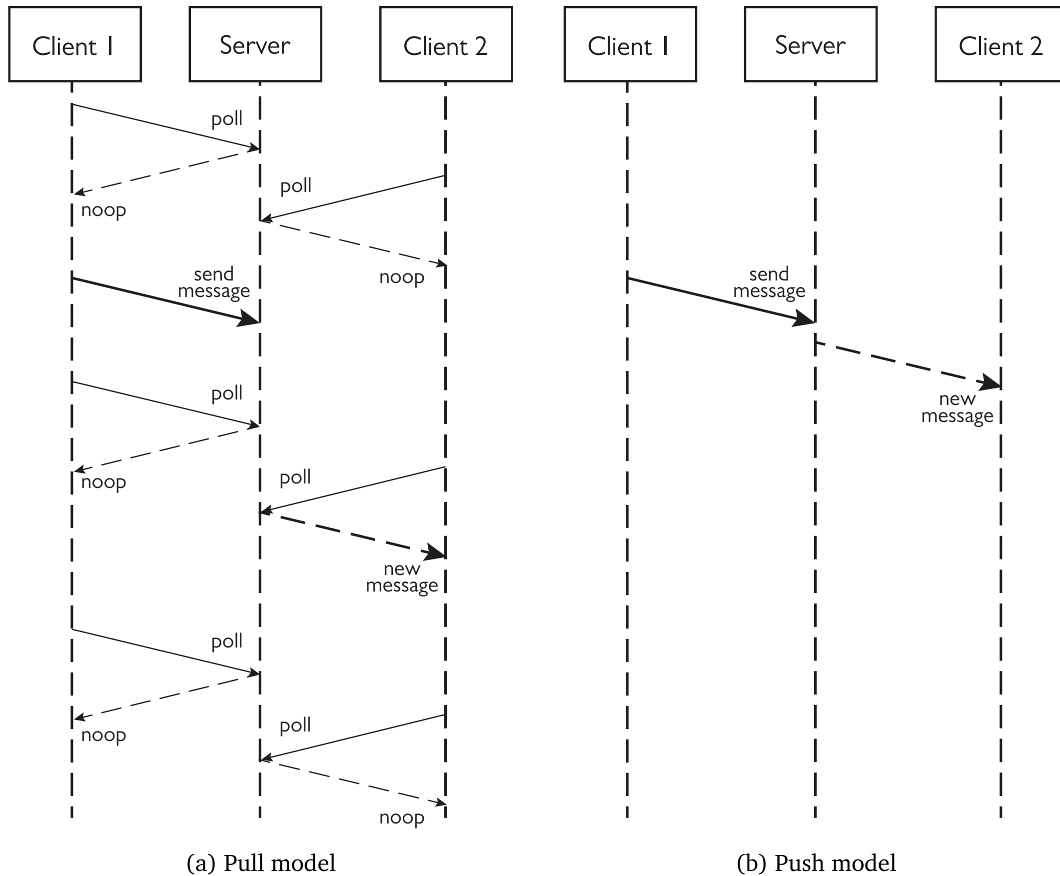
EXAMPLE

Take a typical chat application for example. Several clients connect to one room serviced by the server. After the initial connections are established, they can send messages to other member on the same room. We will compare interaction using two different models as seen in figure 5.11. Figure 5.11a employs a Pull model where the clients continuously and periodically poll the server for new content, resulting in multiple superfluous requests (requests which the server replies with *"no new content"*) and possible delays between the sending of a message and its respective acknowledgment by other clients in the room (If the TTR is high, and the message is sent shortly after the a poll, its acknowledgment will only take place on the next poll).

On the other hand, figure 5.11b employs a Push model where the clients just listens for new notifications per part of the server. This reduces the number of periodic superfluous requests, reducing the load on the server and minimizes the delay between message acknowledgments (when the server receives an new message it can immediately notify all the other clients in the room).

IMPLEMENTATION NOTES

The example described above tackles the problem by migrating to the pull model in its more pure form. However, there are times, mostly due to technical limitations (lack of support of pushing either in the client side e.g. websockets, or in the server side e.g. languages whose architecture cannot withstand high concurrency at low performance cost) where we cannot achieve this. Instead we can degrade our approach gracefully (AJAX



(a) Pull model

(b) Push model

Figure 5.11: Sequence diagram comparison of a chat application using different models

long polling) instead of remaining with our pull model. Here, we will explore two common problems.

The first is the lack of support for pure pushing of data. This can be mitigated by a method called long-polling which is a mixture of pure server push and client pull. After a subscription to a channel, the connection between the client and the server is kept open, for a defined period of time. If no event occurs on the server side, a timeout occurs and the server asks the client to reconnect asynchronously. If an event occurs, the server sends the data to the client and the client reconnects.

The second problem arises when the origin of the events occurs on an external service to which we have limited control over or even none. Since we cannot force the external entity to feed us events whenever they occur, we cannot pursue an pure push solution in our system's architecture. However, we can perform an hybrid approach as seen in Figure 5.12. The server polls the remote service (e.g. a service which we have no control over) for data periodically, and the events are propagated between the server and the client like a regular push model. This solution does not offer the same advantages as its pure approach described previously but it manages to mitigate the usage of resources used

in the client and the server interaction.

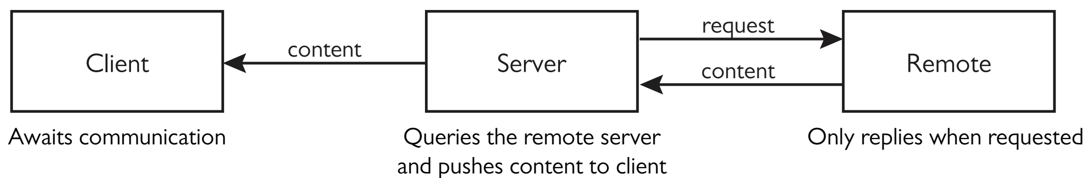


Figure 5.12: Hybrid Push model diagram

KNOWN USES

The push model is being accepted in the industry as the best accepted model for real-time data communication. The range of applicability is vast and it includes real-time social feeds, multiplayer games, collaborative editing and coding, and multimedia chat, among others. To achieve maximum client compatibility, both Facebook with its chat system [Fac08], and Google Docs with its live edition refer from using websockets and instead use the long polling technique to achieve the push model.

5.3.7 Monolithic to Microservice based Architecture

The application of this pattern is related to the system's architecture and it's goal is to migrate an monolithic architecture, where the application is built as a single unit, to a microservice based architecture, where the application is built as a suite of services.

CONTEXT

A monolithic server is a natural way to approach building such a system. All the logic for handling a request runs in a single process, allowing you to use the basic features of your language to divide up the application into classes, functions, and namespaces. With some care, you can run and test the application on a developer's laptop, and use a deployment pipeline to ensure that changes are properly tested and deployed into production. One can horizontally scale the monolith by running many instances behind a load-balancer.

PROBLEM

Monolithic applications can be successful, but increasingly people are feeling frustrations with them — especially as more applications are being deployed to the cloud. Change cycles are tied together — a change made to a small part of the application, requires the entire monolith to be rebuilt and deployed. Over time it's often hard to keep a good modular structure, making it harder to keep changes that ought to only affect one module within that module. Scaling requires scaling of the entire application rather than parts of

it that require greater resource. In fact there are some known patterns that monolithic applications disregard such as *"separation of concerns"*, *"single responsibility principle"*, *"interface segregation"* for example. These hinder the maintainability, difficult the evolution of the system and make them harder to deploy.

FORCES

(−) Maintainability. In a monolithic architecture it's difficult to individually keep up with different service maintainability, as well as their scaling and deployment. This is mainly due to the fact that if we alter something in service A we're also affecting service B.

(−) Separation of concerns. A monolithic architecture is composed by several services that encapsulate different functionalities.

(−) Reuse. Reuse of subparts of the system is difficult as they are tightly coupled.

(−) Scaling. Each service can be scaled individually on demand without affecting the others.

(+) Latency. The communication between components is tight and many times they are even located in the same host, thus minimizing the latency cost.

(−) Fault tolerance. Errors and exceptions are not compartmentalized and might cascade onto other components.

(+) Ease of use. It is easier to create an monolithic application as it involves much less architectural overhead, especially for systems that are growing organically.

SOLUTION

The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API, resulting in segregation of responsibilities [Fow14].

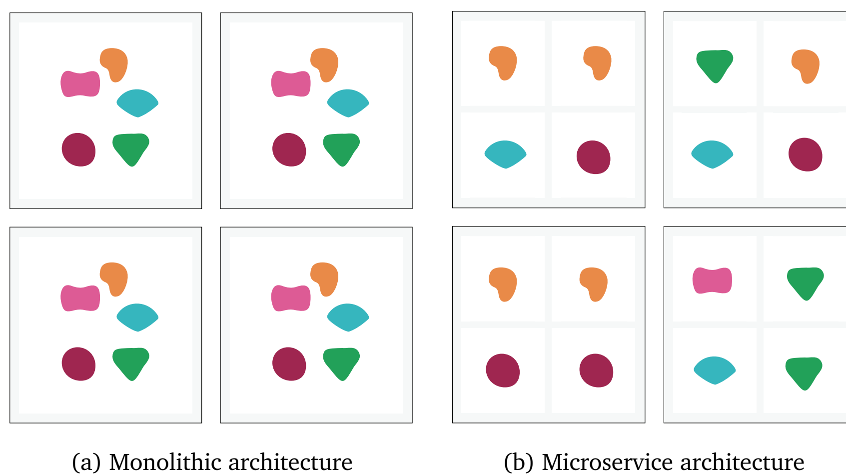
These services are built around business capabilities and are independently deployable and scalable by fully automated deployment machinery. There is a bare minimum of centralized management of these services and a firm module boundary, even allowing for different services to be written in different programming languages. They can also be managed by different teams.

Each microservice should encapsulate only one, well defined, functionality. This along the high degree of separation of concerns avoids the creation of more services for one specific task and promotes an architecture where services make use of other already built services. Moreover, this approach results in a loose coupling between the services where the communication, as above mentioned, may take it's toll but the forces are balanced out by the increase in fault tolerance this approach provides by compartmentalizating

errors and exceptions, thus preventing the problem of cascading failures and allowing the management of problems in isolation.

EXAMPLE

A monolithic application puts all its functionality into a single process and scales by replicate the monolith on multiple services (Figure 5.13a). On the other hand, an microservices architecture puts each element of functionality into a separate service and scales by distributing these services across the servers, replicating as needed (Figure 5.13b).



(a) Monolithic architecture (b) Microservice architecture

Figure 5.13: Comparison between a monolithic and a microservice architecture

Figure 5.13 illustrates these differences in deployment when scaling an application, where each colored shape represents an service and each container a server.

IMPLEMENTATION NOTES

When talking about components we run into the difficult definition of what makes a component. Our definition is that a component is a unit of software that is independently replaceable and upgradeable.

It is important to distinguish that microservice architectures will use libraries, but their primary way of compartmentalizating their own software is by breaking down into services. We define libraries as components that are linked into a program and called using in-memory function calls, while services are out-of-process components who communicate with a mechanism such as a web service request, or remote procedure call. Nonetheless, the biggest issue in changing a monolith into microservices lies in changing the communication pattern. A naive conversion from in-memory method calls to RPC leads to chatty communications which don't perform well. Instead you need to replace the fine-grained communication with a coarser-grained approach.

In an application that consists of a multiple libraries in a single process, a change to any single component results in having to redeploy the entire application. But if that application is decomposed into multiple services, you can expect many single service changes to only require that service to be redeployed.

KNOWN USES

Soundcloud infrastructure was based in a monolithic Ruby application. Although due to scalability issues they now refactored to a microservice based architecture using Scala and Clojure [Sou14]. Moreover, Netflix also refactored their backend API to adopt a more microservice oriented approach [Net13b].

5.3.8 Add event streams

This pattern focus on grouping correlated data or events in a single stream while inverting the flow of control to a push model.

CONTEXT

Being event-driven, as the name suggests results in interacting with multiple sources of asynchronous events that have to be reasoned with in order to do something productive. We have to perform management of adding and removing event listeners, registering callbacks, and composing events to serve a specific purpose. Furthermore, these events can arrive at any point in time and we have no guarantees of the volume being pushed to us during that fixed period of time, which in turn, adds even more complexity to the topic.

PROBLEM

When dealing with asynchronous and event-based programming, we may find ourselves dealing with I/O operations and computationally expensive tasks that might take a long time to complete and potentially block other active threads. Composing sequences of events that should take place one after another, becomes a complex task in these situations often leading to situations commonly labeled as “Callback hell”. Moreover, handling exceptions, cancellation, and synchronization is difficult and error-prone.

Not to mention, we often have to deal with data producers who emit values faster than the consumers can keep up with. The result is consumer side buffers to try to keep up. What we see here is the lack of a mechanism for back pressuring.

FORCES

(–) Concurrency. We encounter difficulties when dealing with events that may be emitted at any point in time and we want to process them sequentially to achieve a certain degree of determinism.

(−) Footprint. Dealing with events in this manner, by explicitly handling the addition and removal of listeners if often error prone, leading to memory leaks. Moreover, when composing events, into higher ones, one often has to make use temporary variables to track states.

(−) Maintainability. Events typically need to be composed before they become useful. Applications that require a great degree of event composition are subjected to a complex task to simply compose them and furthermore, to maintain the code afterwards.

(−) Reuse. Since event composition is arduous, reusing previous implementations is not as frequent.

(−) Error handling. It is hard to decouple the implementation from the error handling management.

(+) Ease of use. Dealing with events this way, although arduous, requires no extra library and no shift in thinking mentality.

SOLUTION

The solution consists on making use an abstraction entitled Observable as we saw in section 3.1.1.3. It combines the observer and iterator pattern and allows us to perform operations (such as mapping, combining and filtering) on a stream, the same way as we were working with a typical collection. We can therefore state that, Observables are composable and they enable us to group events into streams and then process them asynchronously in sequential fashion, allowing for propagation and handling of errors down the stream thus improve the application's maintainability.

In most use cases, the application's footprint decreases, as memory leaks are minimized while managed by the stream instead of the programmer. Furthermore, due to the opinionated underlying implementation, changes can be made especially to have a grainer control of how the concurrency is handled under the hood, without breaking the abstraction's user level semantics. They also provide back pressure capabilities to synchronize producers and consumers when required.

EXAMPLE

The following example shows the ease of composability of events using Rx Observables. Capturing mouse drags using traditional approaches is a simple yet a complex task. Making use of Observables we can write it like we see in source 5.5.

Here, we create an observable that emits events when we're dragging the mouse. Each time we observe an "mouse down" event we want to transform — map — that event into a sequence of mouse moves until the we obtain an "mouse up" event, and lastly we can extract the mouse coordinates.

```
val mouseDrags =
control.mouseDowns
   .map{ mouseDown =>
     control.mouseMoves.takeUntil(control.mouseUps)
   }
   .concatAll()
     .map(evt => (evt.x, evt.y))

mouseDrags.subscribe(
  case (x,y) => println(s"X:$x Y: $y"), // onNext event
  println("Error: "+_), // onError event
  println("Completed") // onCompleted event
)
```

Source 5.5: Composition of mouse events using Observables

The process can be better understood by taking a look at a marble diagram[3] of the events as we can see in Figure 5.14.

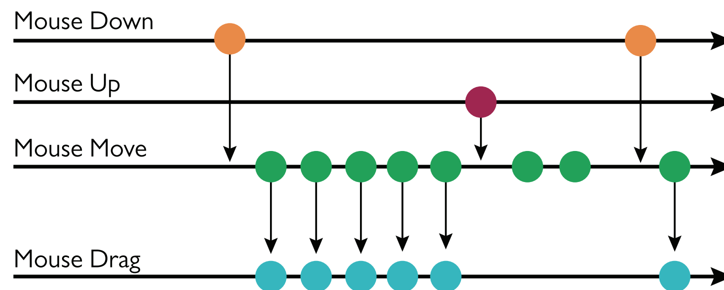


Figure 5.14: Marble diagram of the mouse events composition

Similarly, the same mindset can be applied to a scenario involving asynchronous method calls together with event composition. In source 5.6, we are implementing a autocomplete search input using Observables.

We use `throttle` to handle the key presses debouncing[4], next we map into a asynchronous AJAX call and along with `retry`, which means it can fail up to 3 times before the error is propagated. We then apply `takeUntil` to avoid race conditions due to the asynchronous nondeterministic nature of the AJAX calls (if we type *"a"* and then *"b"*, we want to make sure if the *"a"* reply arrives posteriorly, we do not override the *"ab"* reply). Finally, we put it all together and update the search results every time we emit a new value.

[3]A graphical representation of asynchronous streams and it's manipulation and composition. The X axis represents time. Each horizontal line on the Y axis represents a subscription to a source sequence. The vertical arrows associated with each marble represent the usage of a event emitted by a source being used by another.

[4]Execute the original event once per "bunch" of calls.

```
val searchResultsSets =
  keyPresses
    .throttle(250)
    .map{ key =>
       getJSON("/searchResults?q="+input.value)
          .retry(3)
          .takeUntil(keyPresses)
    }
    .concatAll()

searchResultsSets.forEach(
  resultSet => updateSeachResults(resultSet),
  error => showMessage("the server appears to be down")
)
```

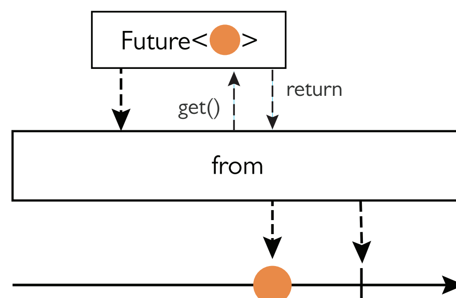
Source 5.6: Example of a autocomplete search input implementation using Observables

IMPLEMENTATION NOTES

An Observable can be made to integrate with any other asynchronous operation, encapsulating it (as seen in source and resulting marble diagram in Figure 5.15). This is a great advantage for graceful migration from one operation stack to another, we can simply facade the interactions at first and slowly migrate all our code to make full use of the abstraction.

```
val future: Future = doAsyncWork()
val obs = Obserservable.from(future)
```

(a) Conversion from future to Observable



(b) Marble diagram of the conversion

Figure 5.15: Example of a Future to Observable conversion using RxScala.

KNOWN USES

Netflix completely revamped their client side event handling approach and the revamped their API in the backend by using the Reactive extensions [Net13a]. This change to RxJava enabled Netflix developers to leverage server-side conconcurrency without the typical thread-safety and synchronization concerns.

5.3.9 Minimize Shared State

This pattern targets the minimization of shared state within a application, mitigating the need for synchronization.

CONTEXT

Modern hardware does not advance any more primarily by increasing the computing power of a single sequential execution core, physical limits have started impeding our progress on this front around the year 2006 so our processors host more and more cores instead. In order to benefit from this kind of growth we must distribute computation even within a single machine.

PROBLEM

Using a traditional approach with shared state concurrency based on mutual exclusion by way of locks the cost of coordination between CPU cores becomes very significant. That is to say, to mutate shared state concurrently we will have to resort to some synchronization mechanism in order to achieve sequential consistency. There is an obvious drawback to such approach: the portions that require synchronization cannot be executed in parallel, meaning they effectively run single-threaded even if they execute on different threads, since only one can be active at any given moment. This effect on runtime is described by Amdahl's law as discussed in section 2.1.1.1.

Furthermore, shared state concurrency is hard to get right and error prone: developers either deploy synchronization mechanism (e.g. as locks or semaphores) excessively or in fault leading to poor performance and possibly deadlocks. This, of course, introduces incidental complexity and non-determinism, making the program code hard to understand and maintain.

FORCES

(−) Scaling. The amount of synchronization required to concurrently access the shared state hinders the application's scaling capabilities.

(−) Maintainability. Synchronization not only hinders the application performance but also adds incidental complexity and non-determinism that make the code hard to reason with and maintain.

(+) Ease of use. Requires no shift in mentality as using synchronization is easier to understand.

SOLUTION

The conclusion we arrive is that synchronization fundamentally limits the scalability. Thus, the optimum solution would be to share nothing, meaning no synchronization is necessary.

Therefore, if we build our system on fully isolated compartments which are executed independently we can abstain ourselves from performing synchronizations. Within the system we can also make use of immutable data structures that are thread-safe and do not require synchronization. In practice we need to exchange requests and responses, which requires some form of synchronization as well, but the cost of that is very low in comparison by using event or message passing. In fact, on commodity hardware it is possible to exchange several hundred million events per second between CPU cores.

EXAMPLE

A good example where a core factor in the abstraction is the minimization of shared mutable state is Actor systems. Erlang and Akka actors, for example, communicate between themselves exclusively using message passing and their state is only accessible internally. Received messages by an actor are stored in a queue and processed synchronously and similarly as a method call would return a value, the actor responds back to its sender though message passing as well.

KNOWN USES

The minimization of shared state is essential for distributed systems that need to scale. Facebook was having difficulties with its chat system, specifically in online-idle-offline states of users. The solution achieved was by using an Erlang based backend service [Fac08]. Coursera's infrastructures were built using PHP, but their user base scaled this resulted in poor performance and difficulties scaling. They solved this by moving their infrastructure to a reactive platform using Scala [Typ14] which along with other benefits, promoted the minimization of shared state.

5.3.10 Add Supervision

This pattern focuses on giving entities the responsibility of monitoring the lifecycle of their subordinates, thus improving the application's fault tolerance.

CONTEXT

At this point, our system should follow an microsystem architecture instead of a monolithic one (5.3.7), meaning services make use of other services in a way that completely isolates them from each other concerning failures in order to avoid those failures from cascading across the entire system. Nevertheless, failure is ubiquitous and since we cannot prevent it we must be prepared to handle it. Traditional systems using synchronous methods, the mechanism for handling failure is provided by exceptions. This approach is unfeasible since everything is asynchronous; the processing will happen outside the call stack of the user and therefore exception cannot reach it.

On the other hand, the usage and deployment of these services in such a scale that they are capable of handling user load without failing and still remaining responsive is, ultimately, our goal. Thus, the question asked is: *How many services A and services B do I need to handle the current user load?*

PROBLEM

One problem at hand is how to handle failure of a service. While reasonable exchanges between user and the machine (either valid replies or validation errors) are well described by the system on how to handle them. Failures on the other hand are characterized by the inability of the machine to produce a response. Thus, the question at hand is: at what part/level of the system should we deal with this failure? Should he restart himself? The current node cannot be set in charge of this because he has no knowledge of the scope of the failure, that may have put him in an invalid state and having him restore himself may lead to contaminating the rest of the system.

Another problem is scale of deployment of the services. We could deploy hundreds of services to guarantee that we are always able to respond to any significant user load but this approach is not viable from a business point of view, where the goal is to maximize profit. This approach would lead to many of the deployed services just idling on a normal day-to-day user load.

FORCES

(−) Fault tolerance. Failure handling has to be performed manually or at the service level which may lead to unwelcome outcomes like invalid states.

(−) Maintainability. The scaling and deployment of nodes must be performed manually.

(−) Separation of concerns. Apart from the business logic the service processes, it is also responsible for restoring itself back to a valid state in case of a failure.

SOLUTION

The solution for either our problem of handling failure and dynamic deployment is delegation — to add supervision to a service [Kuh14]. In the first case, what this means, is that normal requests and responses (including negative ones such as validation errors) flow separately from failures: while the former are exchanged between the user and the service, the latter travel from the service to its supervisor as we can see in figure 5.16. The supervisor is therefore responsible for keeping the service alive and running.

In the second case, the supervisor is responsible for managing its services and guarantee that the number of deployed instances is high enough to handle the workload, and low enough that resources are just being wasted.

The supervisor service will then have the responsibility of deploying and undeploying, and maintaining his child services functional in the event of failures. If the supervisor

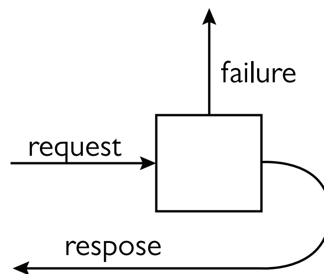


Figure 5.16: Flow diagram of possible components response

himself and the level it is situated at, is not high enough to handle the failure, this is escalated even higher.

In fact, ownership means commitment. That is to say, if a certain piece of the problem is owned by a module, then that means that this module needs to solve that and provide the corresponding functionality because no other module will do it in its stead.

We can then state that our system will be organized hierarchically where each node is responsible for all its subordinate child nodes.

EXAMPLE

We've seen that supervision can be applied at the architectural level but in fact it can even be implemented at the application level as well. Take for example an search application as seen in figure 5.17, where the search module has low-latency modules subordinates where each one can be on different geographical regions to provide low latency to all people across the globe, and each one of this modules has its own fuzzy index which enables it to quickly find the information we're looking for.

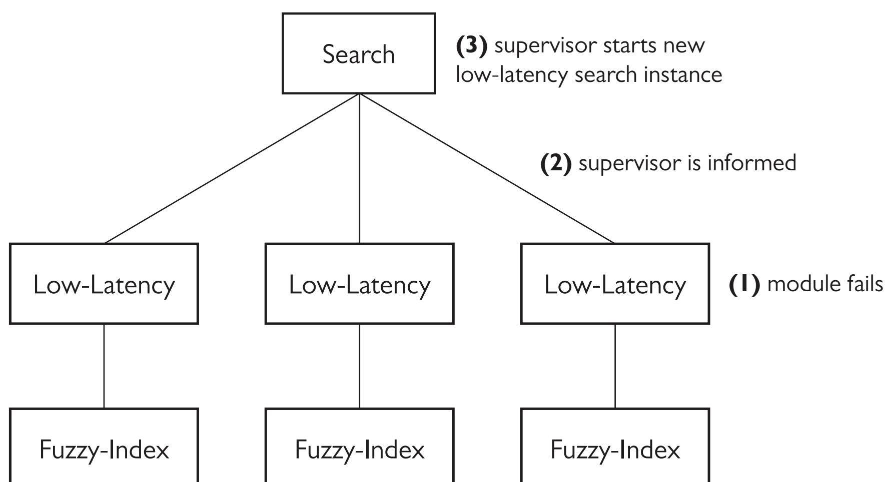


Figure 5.17: Component hierarchy with supervision

The figure depicts a sequence of events: *(1)* one of the Low-latency subordinate modules fails, *(2)* a failure message is propagated to its supervisor, in this case the search module, *(3)* the search module, as supervisor, decides that the best option for this case is to start a new low-latency search instance and terminate the old, faulty one.

IMPLEMENTATION NOTES

Sometimes supervision implementation is not possible either by hardware or software restrictions. However, one can mitigate this by using metrics and monitoring on the existing modules of the system.

KNOWN USES

Akka is a good example of Supervision at the application level. The actors in a system are organized in a hierarchy and each one supervises their subordinates [Akk13]. At the system level, Netflix uses a tool called "Chaos Monkey" for supervising and auto-scaling for service groupings [Net12a].

5.3.11 Retry on Failure

This pattern focuses on handling anticipated, temporary failures when it attempts to connect to a service by transparently retrying an operation that has previously failed in the expectation that the cause of the failure is transient.

CONTEXT

An application that communicates with elements running in a remote network must be sensitive to the transient faults that can occur in this environment. Such faults include the momentary loss of network connectivity to components and services, the temporary unavailability of a service, or timeouts that arise when a service is busy.

PROBLEM

These faults are typically self-correcting, and if the action that triggered a fault is repeated after a suitable delay it is likely to be successful. For example, a database service that is processing a large number of concurrent requests may implement a throttling strategy that temporarily rejects any further requests until its workload has eased. An application attempting to access the database may fail to connect, but if it tries again after a suitable delay it may succeed.

FORCES

(−) Fault tolerance. The act of considering that a requests has failed after a single requests was not successful, hinders our fault tolerance as transient failures are common.

(−) Reuse. Some communications with external remote sources that are unreliable will require us to retry requests until we succeed. The lack of a retry on failure strategy hinders the code reuse for cases like this.

(−) Ease of use. When we are faced with the need to retry some operation; this has to be implemented at the application level each time instead of a consistent interface for this purpose.

(−) Error handling. The error handling lacks graceful degradation; if the request fails, there are no other alternatives to try to correct this failure.

SOLUTION

Transient faults are not uncommon and an application should be designed to handle them elegantly and transparently, minimizing the effects that such faults might have on the business tasks that the application is performing.

If an application detects a failure when sending a request, it can handle the failure depending on the outcome, that is:

- If the fault indicates that the failure is not transient or unlikely to be successful if repeated (e.g. invalid authentication credentials), the application should abort the operation and report a suitable exception;
- if the fault is unusual or rare, it may be caused by freak circumstances (e.g. corrupted network packet) and retrying the failing requests immediately will probably be result in a successful request;
- if the fault is caused by network congestion or if the service is experiencing heavy workload, the network of service may require a short period of time until the issues are rectified or the workload processed. The application should wait for a suitable period of time before retrying the request.

For the more common transient failures, the period between retries should be chosen so as to spread requests from multiple instances of the application as evenly as possible. This can reduce the chance of a busy service continuing to be overloaded. If many instances of an application are continually bombarding a service with retry requests, it may take the service longer to recover.

If the request still fails, the application can wait for a further period and make another attempt. If necessary, this process can be repeated with increasing delays between retry attempts until some maximum number of requests have been attempted and failed. The delay time can be increased incrementally, or a timing strategy such as exponential back-off can be used, depending on the nature of the failure and the likelihood that it will be corrected during this time, until we finally cease and inform the user that the service is unavailable temporarily.

EXAMPLE

The application should wrap all attempts to access a remote service in code that implements a retry policy matching one of the strategies listed above. Requests sent to different services can be subject to different policies, and some vendors provide libraries that encapsulate this approach. These libraries typically implement policies that are parameterized, and the application developer can specify values for items such as the number of retries and the time between retry attempts.

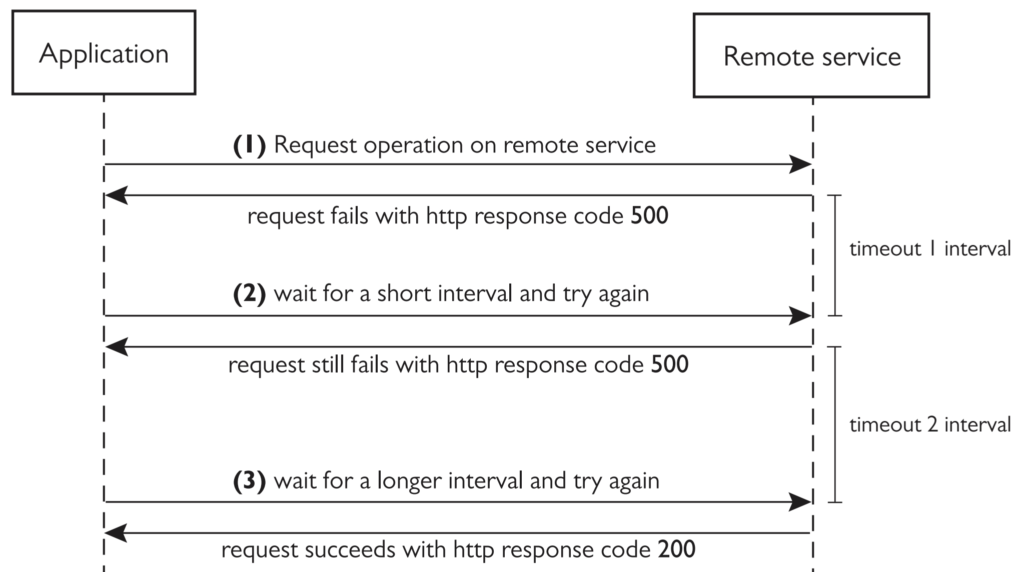


Figure 5.18: Example of a faulty communication with a retry strategy

In figure 5.18 we see the interaction between the application and the remote service. It first send the request only to have it fail, it then waits a specified in the policy interval for the first failure and retries again, though it still results in a failure. Finally, the application waits the timeout interval for the second retry and sends the request that succeeds.

Another scenario could have occurred where the final request also resulted in failure. Depending on the number of retries specified in the policy, the application could stop retrying and present an error message.

IMPLEMENTATION NOTES

The retry policy should be tuned to match the business requirements of the application and the nature of the failure. It may be better for some non-critical operations to fail fast rather than retry several times and impact the throughput of the application. Furthermore, a highly aggressive retry policy with minimal delay between attempts, and a large number of retries, could further degrade a busy service that is running close to or at capacity. This retry policy could also affect the responsiveness of the application if it is continually attempting to perform a failing operation rather than doing useful work. If a

request still fails after a significant number of retries, it may be better for the application to prevent further requests going to the same resource for a period and simply report a failure immediately. This can be achieved by adding a Circuit Breaker pattern (5.3.5).

KNOWN USES

In conjunction with its circuit breaker implementation, Netflix also employs several retry on failures policies to achieve fault tolerance [Net12b]. We can also observe this strategy in Gmail [Goo98] even at the client side e.g. sending an email or connecting to the chat system.

5.4 SUMMARY

In this chapter, we presented our created transformation patterns for a reactive application. These were preceded by an analysis of its internal structure on how they are organized, followed by a summary of the forces observed in each one and their respective relations among themselves. The transformations in themselves were also interrelated to form a ecosystem. All in all, the patterns targeted different levels of the application (*low-level*, *high-level*, *architectural-level*) and could be divided into two different categories: the ones focused of a duality transformation where we provide support for the transition between a non-reactive pattern A to a reactive pattern B, and the ones that aim to add a specific concept as it is non-existent prior to the transformation. Finally in table 5.1, we can see main influence of each pattern in the reactive traits scope. It is important to note that the responsive trait is left out because, ultimately, it is the trait we want to achieve and requires the remaining traits for that purpose.

Table 5.1: Classification of the Transformation patterns' main influence within the reactive traits.

	Event-driven	Scalable	Resilient
Synchronous to Asynchronous processing	√		
Blocking I/O to Non-blocking I/O		√	
Blocking Servlet to Asynchronous Servlet		√	
Parallelize and aggregate		√	
Add Circuit Breaker			√
Pull to Push Model	√		
Monolithic to Microservice based Architecture		√	
Add Event Streams	√		
Minimize Shared State		√	
Add supervision			√
Retry on failure			√

Chapter 6

Case study

In this chapter, we present a overall system description of some parts of the Betfair infrastructure along with their main frameworks in section 6.1, followed by the main challenges presented by this architecture in section 6.2, and finally the relation between each of the transformation patterns in the observed infrastructures in section 6.3.

6.1 SYSTEM DESCRIPTION

Betfair consists of tens of services that can be evolved and scaled independently. That is, all functionality is broken down into separate codebases, deployed on separate hardware/hypervisor, and exposed via well-defined APIs (generally through BSIDL—Betfair Service Interface Definition Language— contracts). As of today, the consumers of these services are different client applications. These are built in proprietary Betfair frameworks:

- **Cougar** — backend service framework.
- **Mantis** — frontend system.
- **Wall** — microservice systems.

Of course there is also an database front but most of the services at Blip, do not interact directly with the betfair databases. That is to say, the interaction is made using the REST protocol instead of SQL queries — there is a Data Access Layer [CSKH12].

6.1.1 Cougar

Cougar — figure 6.1 — is a Betfair application for bootstraing mock servers and generating skeletons. It processes a BSIDL which is a XML based DSL for describing versioned service interfaces allowing the description of the methods, request/response parameters, exceptions as well as events and all documentation associated with these entities.

The generated sources contain interfaces compatible with the service, configuration files to bind everything together in Spring and empty skeletons of classes implementing the interfaces that compose the service with synchronous or asynchronous variations.

Since the core of the Spring Framework is based in the Inversion of Control principle [Bur12], all these resources are bootstrapped into Spring, which in turn, bootstraps Jetty (a servlet engine and HTTP server) [Per04] to handle the HTTP connection and communication part or generates a *.war* file that can be deployed in a normal application server that suuports servlets (e.g. tomcat).

The resulting application is an FACADE [GVHJ95] for a specific service, which in turn may or may not use internal Facades to access other services, that can be used by client applications.

Its execution flow is as follows:

- The client application requests an operation of the service;
- The request and response (which may come in several supported formats) is unmarshalled;
- The execution resolver matches the operation key with the correct executable containing the business logic;
- Depending on the implementation used, the operation runs synchronously or asynchronously.

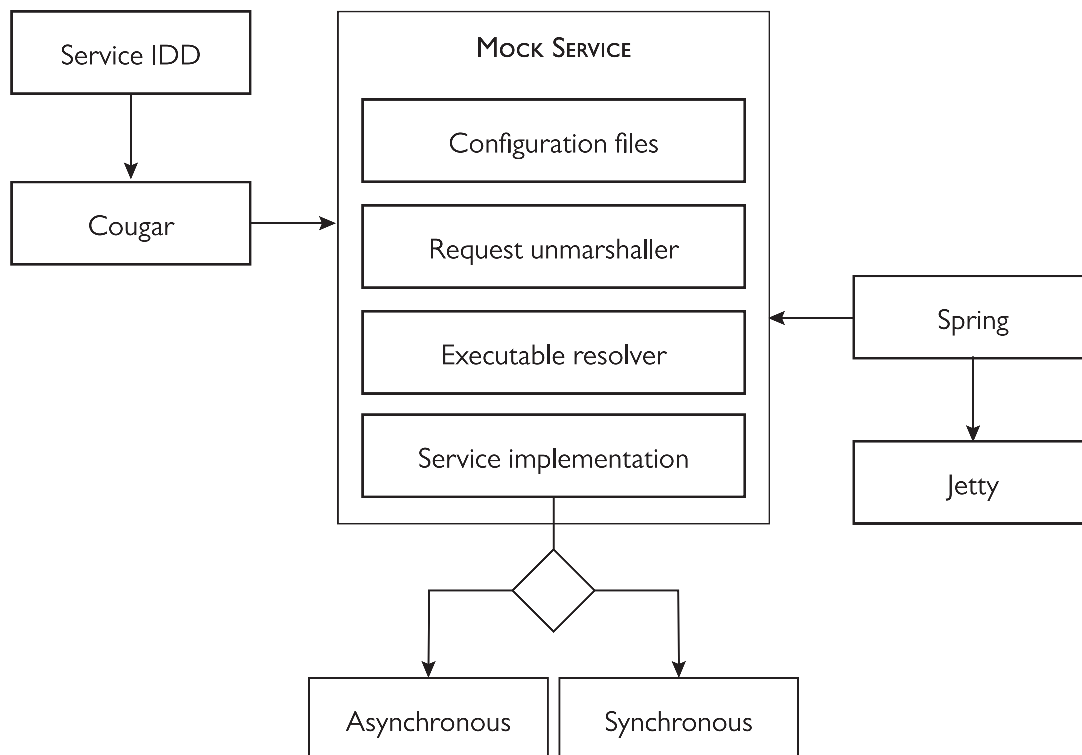


Figure 6.1: Backend system underlying architecture based on Cougar

6.1.2 Mantis

Mantis —figure 6.2— is a Betfair application for frontend development built upon Spring. It provides (1) integration with Betfair's infrastructure (auth, logging, measuring, inter connectivity with other components), (2) declarative page composition, (3) parallel component execution (module configuration framework), (4) task framework (promises-like implementation) and (5) bundling out-of-the-box (by using wro4j[1]).

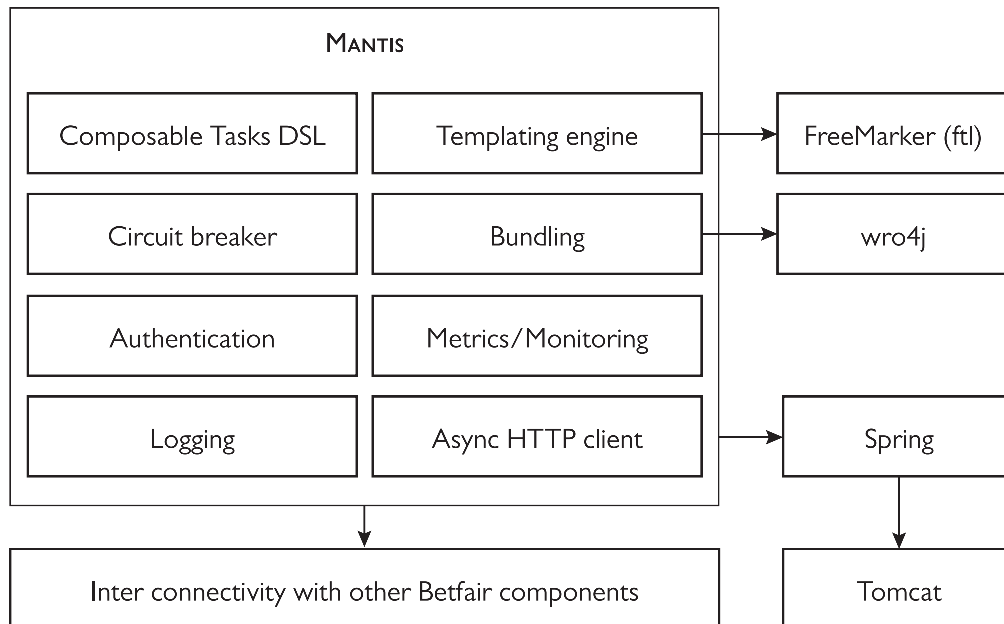


Figure 6.2: Frontend system based on Mantis

6.1.3 Wall

The Wall is a set of microservices being developed at this moment. Each one of these microservices is entitled a *"Brick"* and it is being built in Scala, making use of Twitter's Finagle and Apache's Thrift [2]. This system is the result from a migration of existing monolithic services to a more microservice based architecture as seen in the *"Monolithic to microservice based Architecture"* pattern (section 5.3.7).

6.2 MAIN CHALLENGES

Concerning the proprietary Betfair frameworks: Mantis and Cougar. Mantis concern is to produce modular (but monolithic) websites and Cougar interest is to build BSIDL enabled services. For example we may have separate client application (e.g. SSW, SBW, MES, MSS, ...) that talk to separate platform services (e.g. ERO, FACET,...) which in turn talk to separate data services (Oracle, MQs, ...).

[1] Web resource optimizer — www.code.google.com/p/wro4j
[2] www.thrift.apache.org

Today, sharing code between these client applications is difficult and different parts of the application can't be scaled in isolation.

In light of these difficulties, it was decided on the development of the Wall and to decide its underlying technology implementation a study was made weighing several forces. These forces and its importance to Blip is shown in in table 6.1, where the values range from 0 to 10 — from "not important" to "important".

Table 6.1: Technologies' characteristics valuation for Betfair

Item	Description	Weight
Async IO	Asynchronous operations support	10
Scalability	Ability of scale	10
Push	Websocket support for the push model	2
Ramp-up speed	Existing skills on team or ease of learning	7
Velocity	$\text{Time}_{\text{Think}} + \text{Time}_{\text{Implement}}$	10
Maintenance	Maintenance cost	10
Throughput	Out-of-the-box RPS	5
Community	How healthy and helpful is the community?	8
Ecosystem	Volume and quality of existing packets	8
Enterprise-ability	Does it fit the enterprise requirements?	8
Sponsorship	Who is backing the solution	5
Cost of ownership	Is it free?	3
Testability	Ease of testing	9

Several frameworks (based on the JVM: Vertx[3], Finagle, Play[4], Spray[5]; and based on Javascript: Express[6] and Restify[7]) were evaluated with these forces in mind for the development of the wall, and the *winner* was Express, followed by Finagle in second place. However, due to other unknown aspects, Expressed was dismissed and the development of the Wall started using Finagle.

6.3 PATTERN APPLICATION

Some of the transformation pattern can already be found implemented in the Betfair infrastructure, while some of them are accounted for partially and others not implemented at all. Thus, we can pinpoint each of the patterns in such a way:

[3] www.vertx.io
[4] www.playframework.com
[5] www.spray.io/
[6] www.expressjs.com
[7] www.mcavage.me/node-restify

- **Synchronous to Asynchronous processing**
  Both Mantis and cougar already have the underlying architectural basis for implementing asynchronous and synchronous tasks. It is up to the programmer to develop the asynchronous over the synchronous solution.

- **Blocking I/O to Non-blocking I/O**
  Although the most recent versions of Mantis and Cougar support asynchronous programming, non-blocking I/O is not supported, and it is one of the things that could be improved upon. On the other hand, the services built within the Wall will support it resorting to Netty.

- **Blocking Servlet to Asynchronous Servlet**
  Cougar operates using servlets and it uses a servlet API version which already supports AIO.

- **Parallelize and aggregate**
  Mantis has a task composition DSL that is promise-like and allows us to parallelize and aggregate tasks.

- **Add Circuit Breaker**
  The mantis core already has an circuit breaker implementation integrated that can be customized for different services.

- **Pull to Push Model**
  At the time being no service makes use of the push model, and it is in fact, an physical infrastructure limitation. However, in the future, the push model could be employed and it would bring great advantages when used along with systems as *"live scores"* which at the moment are implemented by using the pull model.

- **Monolithic to microservice based Architecture**
  In production, the architecture is all based on monolithic systems. However, this particular transformation to microservices architecture is currently taking place using Scala, Finagle and Thrift as its core technologies. Thus, we explore this transformation in greater detail in the subsection 6.3.1.

- **Add Event Streams**
  No event stream technologies are employed. The majority of the business logic would not take much advantage of event streams backend wise, except some caching strategies. This cache is populated by using the pull model but could be better modeled by using a stream. We could also employ event streams more directed into the frontend. This would result in a complete revamp on how events are handled in the client application with all the advantages we have already seen.

- **Minimize Shared State**
  The minimization of shared state is promoted when coding, but it is not enforced. That is to say, that the underlying technologies allow the use of both methodologies but the explore of a piece of code is completely on the hands of the developer. Furthermore, synchronization is avoided whenever possible by using futures and threadpools.

- **Add supervision**
  No supervision mechanisms are employed. Systems are contained by using monitoring and metrics. Supervision could be added to the Wall and their bricks for increased fault tolerance.

- **Retry on failure**
  No retry mechanisms or policies are implemented in the code base. That is to say, if a requests fails, the method that made the request also fails. There is however an exception, where the system has to communicate with a unreliable third party system and employs an specific retry strategy using a message queue system. The ideal would be an implementation of a retry policy can could be used for every request and be customized for different types of requests.

This exercise aims to prove how the analyzed system could take advantage of such transformations but an actual implementation at a enterprise level. Such process in a large enterprise is an lengthy process and associated with the reduced access to the system, the actual implementation was not concluded in this dissertation time frame and will continue as future work.

6.3.1 Ongoing transformations

The development of the Wall, i.e. the transformation from a *"Monolithic architecture to a microservice based architecture"* is taking place at the time being and focuses on tackling the main challenges found in the system as we saw in section 6.2.

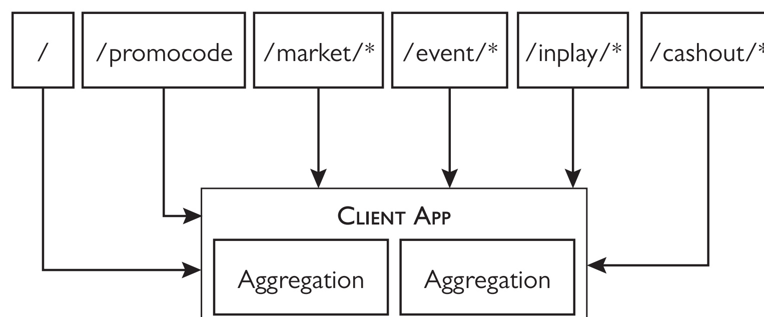


Figure 6.3: Service displacement with monolithic architecture at Betfair

The monolithic architecture we are migrating from is shown in figure 6.3, where we see several markets interact with the same monolithic application that performs data aggregation to reply, and has the problems we have already seen.

The microservice architecture we are now migrating towards can be seen in figure 6.4, where we no longer have a monolithic application but instead several small services, and will solve two problems mentioned before:

- the sharing of code between channels
  - by having separate components reusable between services (e.g. different JARs, NPM packages);
  - each service is free to adapt the component to suit its needs.
- scaling services in isolation
  - if a certain service hits a traffic peak, it can be scaled without scaling the neighboring services.

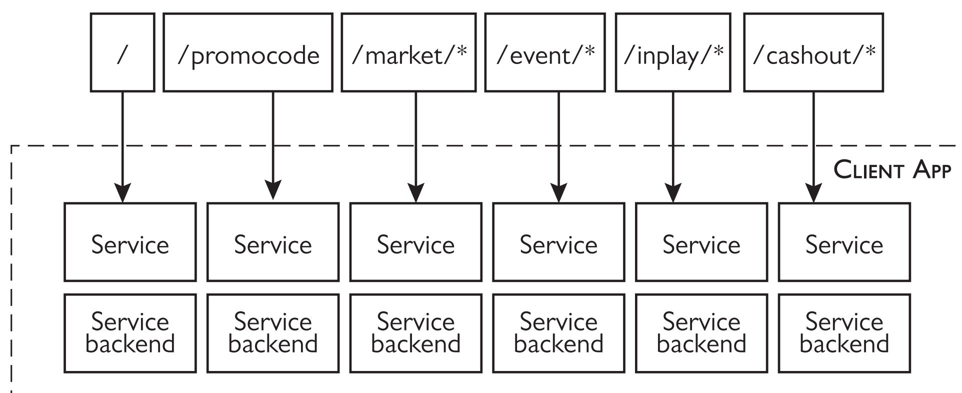


Figure 6.4: Service displacement with microservice architecture at Betfair

6.4 SUMMARY

In this chapter we described the overall infrastructure of some parts of Betfair system and more specifically, the underlying architecture of the frameworks that Blip makes use of: Cougar, Mantis and — new being-developed — Wall. Next, we presented the current main challenges for Blip and saw their scaling difficulties and thus the creation of the Wall, which we presented further details. Finally, we pinpointed parts of Blip system where we could apply each transformation pattern.

Chapter 7

Conclusions and Future Work

In this section we will present a quick summary of what was discussed in this dissertation (section 7.1), next we make a summary of our main contributions (section 7.2) and finally we'll discuss possible future work (section 7.3).

7.1 SUMMARY

In this dissertation, we started by performing introduction to the theme of transformation patterns for a reactive application, along with the context, motivation, scope and objectives associated.

Next, we took a look at the background of Reactive systems —what traits does a system have to have, to be classified as reactive? be event-driven, scalable, resilient and responsive— and the state of art of refactorings and patterns along with some epistemological questions. And, we finish this section by introducing the case study where this dissertation took place: Blip — that works for Betfair.

We proceeded to explore the current state of the art of reactive systems, describing in detail abstraction concepts in reactive programming and supporting technologies that allow us to this. Here, we saw several characteristics a system must possess and shortcomings it should avoid. This resulted in the our problem section.

As we defined the problem associated with the current state of art, we formalized our main thesis and sketched our approach to the subject.

We then present the reader with the developed transformation patterns, following a strict format along with their respective forces and inter-relationships.

And finally, we relate the transformation patterns to the infrastructure observed in the case study by presenting its architecture and pinpoints parts of the system where each pattern could be applied.

7.2 CONCLUSION AND MAIN CONTRIBUTIONS

The reactive paradigm covers a broad range of concepts, a context scoping was of the utmost importance. There had to be a compromise between the level of depth of information in each concept versus the spectrum of diversity of information covered in the patterns. And, within this balance, we also had to consider the level of detail of a pattern versus the level of abstraction of the pattern to maximize its range of applicability to a more generic context, as it directly affects its level of empirical content. Given that our main thesis question was stated as:

> *Given two systems A and B, where A is non-reactive and B is reactive, what are the set of transformations* $(T_1, T_2, ..., T_n)$ *that transform A into B while simultaneously preserving the original key functionalities?*

We can infer the following conclusions:

- the state of the art of reactive applications is constantly evolving at the time being and provides a good approach for building scalable, distributed, fault-tolerant systems.
- there is a substantial difference between non-reactive and reactive systems in terms of characteristics which associated with the cost of rebuilding a system from the ground-up points to a necessity for an easier transitioning process.
- the developed catalog provides support for this transition, along with its 11 transformations, and it was applied to the Betfair system to study its evolution to a reactive system.

7.3 FUTURE WORK

As future work, there are several aspects that could be improved upon, namely:

- Further explore the currently evolving state of art on primitives, abstractions and frameworks of reactive systems.
- Expanding the pattern language, with pattern such as, but not limited to:
  - **Throttle pattern**. Control the consumption of resources used by an instance of an application, an individual tenant, or an entire service;
  - **Event sourcing pattern**. Use an append-only store to record the full series of events that describe actions taken on data in a domain, rather than storing just the current state, so that the store can be used to materialize the domain objects;
  - **Sharding pattern**: Divide a data store into a set of horizontal partitions or shards.

- Provide stronger forms of validation (as the ones considered unfeasible due to time restrains in section 4.4) to offer more substantial proof of the truthfulness of a transformation.
  - Obtain metrics from non-reactive systems, comparing the before and after states of the application with the developed transformation patterns applied;
  - Continue the study of the Betfair system throughout the completion of the transformation process.

References

[ABBC06] Roberto M. Amadio, Gérard Boudol, Frédéric Boussinot, and Ilaria Castellani. Reactive Concurrent Programming Revisited. *Electron. Notes Theor. Comput. Sci.*, 162:49—-60, 2006. Cited on page 19.

[AFZ97] S Acharya, M Franklin, and S Zdonik. Balancing push and pull for data broadcast. *ACM SIGMOD Rec.*, 1997. Cited on pages 6, 19, and 30.

[AIS77] C Alexander, S Ishikawa, and M Silverstein. A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series). 1977. Cited on page 10.

[Akk13] Akka. Supervision and Monitoring — Akka Documentation. Available from http://doc.akka.io/docs/akka/snapshot/general/supervision.html [Accessed at 23/06/14]. Cited on page 67.

[Akk14] Akka. Aggregator Pattern — Akka Documentation. Available from http://doc.akka.io/docs/akka/2.3.0/contrib/aggregator.html [Accessed at 23/06/14]. Cited on page 49.

[Arm97] J Armstrong. The development of Erlang. *ACM SIGPLAN Not.*, 1997. Cited on page 1.

[AVWW96] J Armstrong, R Virding, C Wikstr, and M Williams. Concurrent programming in ERLANG. 1996. Cited on page 26.

[BHS07a] F Buschmann, K Henney, and D Schimdt. Pattern-oriented Software Architecture: On Patterns and Pattern Language. 2007. Cited on page 14.

[BHS07b] F Buschmann, K Henney, and DC Schmidt. Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing (v. 4). 2007. Cited on page 11.

[BHS07c] Frank Buschmann, Kevlin Henney, and Douglas C Schmidt. *Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages*. Wiley, Chichester, UK, 2007. Cited on page 11.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. 1996. Cited on pages 10, 11, 14, and 32.

[Bur12] K Burns. Inversion of Control. *Begin. Wind. 8 Appl. Dev. XAML . . .*, 2012. Cited on page 72.

[Com14] Compete. Compete - Betfair. Available from https://siteanalytics.compete.com/betfair.com/ [Accessed at 06/02/14]. Cited on page 17.

[Cru14] Crunchbase. Betfair Bets on Couchbase NoSQL. Available from http://www.couchbase.com/customer-stories/betfair-bets-couchbase-nosql [Accessed at 31/01/14]. Cited on page 16.

[CSKH12] A Chatterjee, CR Shah, M Kishore, and JP Hsu. Data access layer. *US Pat. 8,260,757*, 2012. Cited on page 71.

[Dic14] Webster Dictionary. Webster Dictionary. Available from http://www.merriam-webster.com/dictionary/ [Accessed at 01/02/14]. Cited on page 5.

[DYBJ04] A Dantas, JW Yoder, P Borba, and RE Johnson. Using Aspects to Make Adaptive Object-Models Adaptable. *RAM-SE*, 2004. Cited on page 11.

[EH97] C Elliott and P Hudak. Functional reactive animation. *ACM SIGPLAN Not.*, 1997. Cited on page 24.

[Ell09] Conal M. Elliott. Push-pull functional reactive programming. In *Proc. 2nd ACM SIGPLAN Symp. Haskell - Haskell '09*, page 25, New York, New York, USA, September 2009. ACM Press. Cited on page 20.

[Fac08] Facebook. Facebook Techblog: Facebook Chat. Available from https://www.facebook.com/note.php?note_id=14218138919. Cited on pages 56 and 64.

[FBB$^+$] M Fowler, K Beck, J Brant, W Opdyke, and D Roberts. Refactoring: improving the design of existing code. 1999. *ISBN 0-201-48567-2*. Cited on pages 12, 13, and 14.

[Fer10] Hugo Sereno Ferreira. *Adaptive Object-Modeling Patterns , Tools and Applications*. PhD thesis, University of Porto, 2010. Cited on pages 11 and 31.

[Fow] Martin Fowler. Analysis Patterns. Available from http://martinfowler.com/articles.html#id314249. Cited on page 11.

[Fow96] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Menlo Park, CA, 1996. Cited on page 11.

[Fow14] Martin Fowler. Microservices. Available from http://martinfowler.com/articles/microservices.html [Accessed at 23/06/14]. Cited on page 57.

[FY97] B Foote and J Yoder. Big ball of mud. *Pattern Lang. Progr. Des.*, 1997. Cited on page 24.

[Git14a] Github. Netty related projects. Available from https://github.com/netty/netty/wiki/Related-projects. Cited on page 44.

[Git14b] Github. postgresql-async & mysql-async - async, Netty based, database drivers for MySQL and PostgreSQL written in Scala. Available from https://github.com/mauricio/postgresql-async [Accessed at 22/06/14]. Cited on page 42.

[Goo98] Google. Google. Available from www.google.com [Accessed at 11/2/20]. Cited on pages 1 and 70.

[Gor03] Tom Mens; Serge Demeyer; Bart Du Bois; Hans Stenten; Pieter Van Gorp. Refactoring: Current Research and Future Trends. *Electron. Notes Theor. Comput. Sci.*, 82(3), 2003. Cited on page 16.

[GP12] CS Gordon and MJ Parkinson. Uniqueness and reference immutability for safe parallelism. *Proc. . . .* , 2012. Cited on page 24.

[GR92] ER Gansner and JH Reppy. A foundation for user interface construction. *Lang. Dev. User . . .* , 1992. Cited on page 6.

[Gus88] JL Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 1988. Cited on page 7.

[GVHJ95] E Gamma, J Vlissides, R Helm, and R Johnson. Design patterns: Elements of reusable object-oriented software. *Read. Addison-Wesley*, 1995. Cited on pages 10, 11, 12, 22, 32, 33, and 72.

[Has09] Haskell. Typeclassopedia. Available from http://www.haskell.org/haskellwiki/Typeclassopedia. Cited on page 11.

[HBS73] C Hewitt, P Bishop, and R Steiger. A universal modular actor formalism for artificial intelligence. *Proc. 3rd Int. Jt. . . .* , 1973. Cited on pages 26 and 27.

[HW04] G Hohpe and B Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. 2004. Cited on page 11.

[Keg04] Kegel. The C10K problem. Available from http://www.kegel.com/c10k.html [Accessed at 11/02/14]. Cited on pages 7 and 30.

[Ker05] J Kerievsky. *Refactoring to patterns*. 2005. Cited on pages 13 and 14.

[KJ04] M Kircher and P Jain. Pattern-oriented software architecture volume 3: patterns for resource management. 2004. Cited on page 11.

[KP09] Christian Kohls and Stefanie Panke. Is that true...?: thoughts on the epistemology of patterns. In *Proc. 16th Conf. Pattern Lang. Programs*, PLoP '09, pages 1–14. ACM, 2009. Cited on page 14.

[Kuh14] Roland Kuhn. *Reactive Design Patterns*. Manning, 2014. Cited on pages 6 and 65.

[LB11] J Liberty and P Betts. *Programming Reactive Extensions and LINQ*. 2011. Cited on page 25.

[Man] SR Mantena. Transparency in Distributed Systems. *crystal.uta.edu*. Cited on page 9.

[Man14] Reactive Manifesto. The Reactive Manifesto. Available from http://www.reactivemanifesto.org/ [Accessed at 31/01/14]. Cited on page 5.

[Mon14] MongoDB. MongoDB Asynchronous Java Driver - MongoDB Asynchronous Java Driver. Available from http://www.allanbank.com/mongodb-async-driver/index.html [Accessed at 22/06/14]. Cited on page 42.

[MT04] T Mens and T Tourwé. A survey of software refactoring. *Softw. Eng. IEEE Trans. . . .* , 2004. Cited on page 13.

[Net11] Netflix. The Netflix Tech Blog: Making the Netflix API More Resilient. Available from http://techblog.netflix.com/2011/12/making-netflix-api-more-resilient.html [Accessed at 23/06/14]. Cited on page 52.

[Net12a] Netflix. The Netflix Tech Blog: Chaos Monkey Released Into The Wild. Available from http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html [Accessed at 23/06/14]. Cited on page 67.

[Net12b] Netflix. The Netflix Tech Blog: Fault Tolerance in a High Volume, Distributed System. Available from http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html [Accessed at 23/06/14]. Cited on page 70.

[Net13a] Netflix. The Netflix Tech Blog: Functional Reactive in the Netflix API with RxJava. Available from http://techblog.netflix.com/2013/02/rxjava-netflix-api.html [Accessed at 23/06/14]. Cited on page 62.

[Net13b] Netflix. The Netflix Tech Blog: Optimizing the Netflix API. Available from http://techblog.netflix.com/2013/01/optimizing-netflix-api.html [Accessed at 23/06/14]. Cited on page 59.

[Net13c] Netty. Netty. Available from http://netty.io/. Cited on page 44.

[Nyg07] M Nygard. Release It!: Design and Deploy Production-Ready Software. 2007. Cited on page 50.

[Per04] B Perry. *Java Servlet & JSP Cookbook*. 2004. Cited on page 72.

[RZ96] D Riehle and H Züllighoven. Understanding and using patterns in software development. *TAPOS*, 1996. Cited on page 10.

[Sou14] Soundcloud. Backstage Blog - Building Products at SoundCloud —Part I: Dealing with the Monolith - SoundCloud Developers. Available from http://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith [Accessed at 17/06/14]. Cited on page 59.

[SSRB00] Douglas C Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2000. Cited on page 11.

[TS02] AS Tanenbaum and M Van Steen. *Distributed systems*. 2002. Cited on pages 6 and 9.

[Twi06] Twitter. Twitter. Available from www.twtter.com [Accessed at 11/2/13]. Cited on page 1.

[Typ14] Typesafe. Coursera Case Study: Accessible Online Education with Typesafe. Available from http://typesafe.com/blog/making-online-education-accessible-with-typesafe [Accessed at 23/06/14]. Cited on page 64.

[WG02] W Willinger and R Govindan. Scaling phenomena in the Internet: Critically examining criticality. *Proc. ...*, 2002. Cited on page 1.

[WG06] CB Weinstock and JB Goodenough. On system scalability. 2006. Cited on page 8.

[WH00] Z Wan and P Hudak. Functional reactive programming from first principles. *ACM SIGPLAN Not.*, 2000. Cited on page 25.

[WHCK04] B Walker, CS Holling, SR Carpenter, and A Kinzig. Resilience, adaptability and transformability in social–ecological systems. *Ecol. Soc.*, 2004. Cited on page 9.

REFERENCES

Appendix A

Refactoring catalogs

Table A.1: Catalog of *refactorings*.

Category	**Refactorings**
Composing Methods	Extract Method, Inline Method, Inline Temp, Replace Temp with Query, Introduce Explaining Variable, Split Temporary Variable, Remove Assignments to Parameters, Replace Method with Method Object, Substitute Algorithm
Moving Features Between Objects	Move Method, Move Field, Extract Class, Inline Class, Hide Delegate, Remove Middle Man, Introduce Foreign Method, Introduce Local Extension
Organizing Data	Self Encapsulate Field, Replace Data Value with Object, Change Value to Reference, Change Reference to Value, Replace Array with Object, Duplicate Observed Data, Change Unidirectional Association to Bidirectional, Change Bidirectional Association to Unidirectional, Replace Magic Number with Symbolic Constant, Encapsulate Field, Encapsulate Collection, Replace Record with Data Class, Replace Type Code with Class, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Subclass with Fields
Simplifying Conditional Expressions	Decompose Conditional, Consolidate Conditional Expression, Consolidate Duplicate Conditional Fragments, Remove Control Flag, Replace Nested Conditional with Guard Clauses, Replace Conditional with Polymorphism, Introduce Null Object, Introduce Assertion
Making Method Calls Simpler	Rename Method, Add Parameter, Remove Parameter, Separate Query from Modifier, Parameterize Method, Replace Parameter with Explicit Methods, Preserve Whole Object, Replace Parameter with Method, Introduce Parameter Object, Remove Setting Method, Hide Method, Replace Constructor with Factory Method, Encapsulate Downcast, Replace Error Code with Exception, Replace Exception with Test
Dealing with Generalization	Pull Up Field, Pull Up Method, Pull Up Constructor Body, Push Down Method, Push Down Field, Extract Subclass, Extract Superclass, Extract Interface, Collapse Hierarchy, Form Template Method, Replace Inheritance with Delegation, Replace Delegation with Inheritance
Big Refactorings	Tease Apart Inheritance, Convert Procedural Design to Objects, Separate Domain from Presentation, Extract Hierarchy

Table A.2: Catalog of *Refactoring to patterns*.

Refactorings
Chain constructor, Compose method, Encapsulate classes with factory, Encapsulate composite with builder, Extract adapter, Extract composite, Extract parameter, Form template method, Inline singleton, Introduce null object, Introduce polymorphic creation with factory method, Limit instantiations with singleton, Move accumulation to collecting parameter, Move accumulation to visitor, Move creation knowledge to factory, Move embellishment to decorator, Replace conditional dispatcher with command, Replace conditional logic with strategy, Replace constructors with creation methods, Replace hard coded notifications with observer, Replace implicit language with interpreter, Replace implicit tree with composite, replace one/many distinctions with composite, Replace state altering conditionals with state, Replace type code with class, Unify interfaces, Unify interfaces with adapter