Service Cutter

A Structured Way to Service Decomposition

Michael Gysel & Lukas Kölbener

BACHELOR THESIS

University of Applied Sciences of Eastern Switzerland (HSR FHO)

Department of Computer Science

in Rapperswil

Advised by Prof. Dr. Olaf Zimmermann

Co-Examined Internally by Prof. Dr. Andreas Rinkel

Co-Examined Externally by Dr. Gerald Reif

Supported by Zühlke Engineering AG

December 2015 (v1.0.1)

Declaration

We hereby declare

- that this bachelor thesis and the work presented in it is our own, original work.
- that all sources we consulted and cited are clearly attributed. We have acknowledged all main sources of help.
- that no copyright secured material has been used by unfair means.

Rapperswil, December 18, 2015

Lukas Kölbener Michael Gysel

Contents

Abstract

Decomposing a software system into smaller parts has been an important challenge in the software industry for many years. With the rise of distributed systems, it has become even more important to split a system into loosely coupled and highly cohesive parts. The architectural style Service Oriented Architecture (SOA) and the currently trending microservices tackle many challenges of such systems, but remain vague on how to decompose a system into services.

We propose a structured approach to service decomposition by providing a comprehensive catalog of 16 coupling criteria. We abstracted them from existing literature, the experience of our industry partner and our thesis advisor.

These coupling criteria are the basis of the Service Cutter tool, a prototype that extracts coupling information out of well-established software engineering artifacts such as domain models and use cases. Using this information, the Service Cutter suggests service cuts to assist an architect's decomposition decisions.

We developed a scoring system that transforms the coupling data into an undirected, weighted graph. On this graph, we employ two graph clustering algorithms from the literature to find densely connected clusters as service candidates. This approach ensures that the Service Cutter produces service cuts that minimize coupling between services while promoting high cohesion within a service.

In our tests, we successfully decomposed two sample applications. Most scenarios resulted in applicable service cuts while others were inadequate. These results suggest that our structured and automated way to assist service decomposition decisions is a promising approach. The thesis lays the foundation for further research in this area.

1. Management Summary

Context

A major challenge of writing software has always been to keep created source code maintainable. Early in the history of software development, modules have been used to structure code in manageable pieces and to make it reusable. With the rise of distributed systems, engineers started to implement services communicating with each other over a network. Coupling between such services has gained relevance as aspects like consistency or release cycles have become more challenging.

Several methodologies exist to guide a software architect when he or she designs services. "Service Oriented Architecture" is especially common in enterprise environments, microservices became popular in recent years. Leaving technical differences aside, all approaches share a common challenge: How can a big collection of data and functionality be decomposed into smaller pieces while retaining high cohesion and low coupling.

A Structured Approach to Service Decomposition

We found no extensive description of architecturally significant requirements in existing literature of distributed systems that optimize loose coupling and high cohesion in service decomposition. We therefore compiled a catalog of 16 coupling criteria that aims to form a comprehensive but not conclusive collection based on literature and the input of our industry partner and our advisor.

These coupling criteria help a software architect to structure architecturally significant requirements influencing the service decomposition decisions. Figure 1.1 outlines the criteria catalog structured in viewpoints (rows) and criteria types (columns).

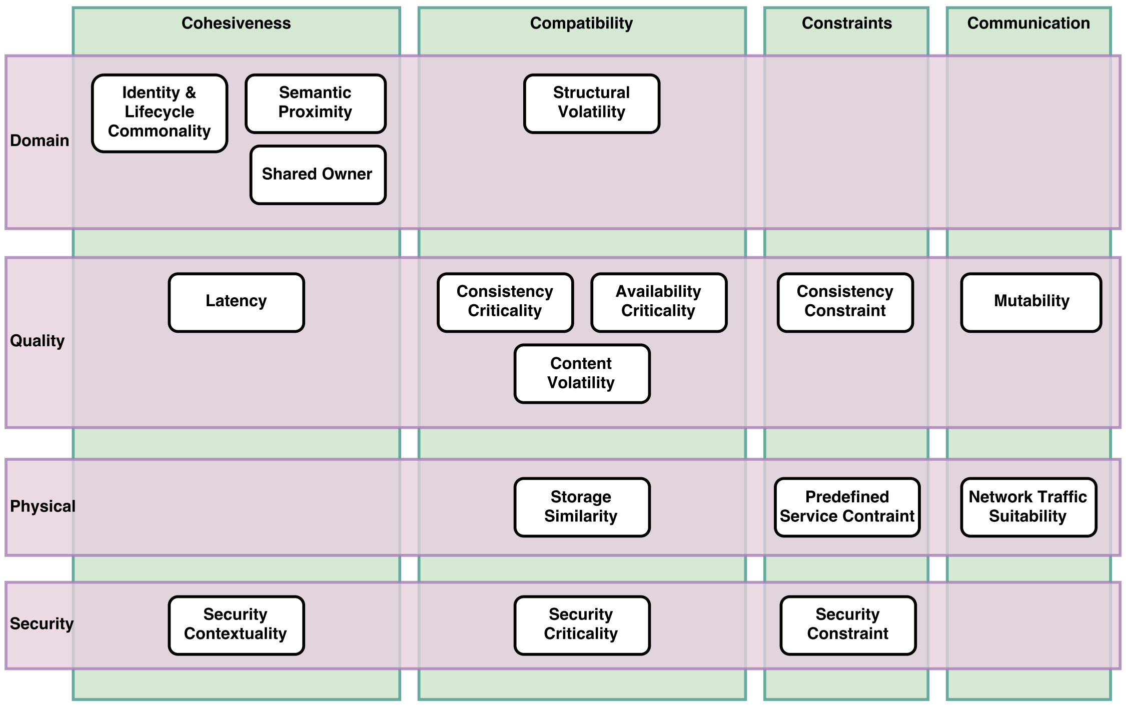


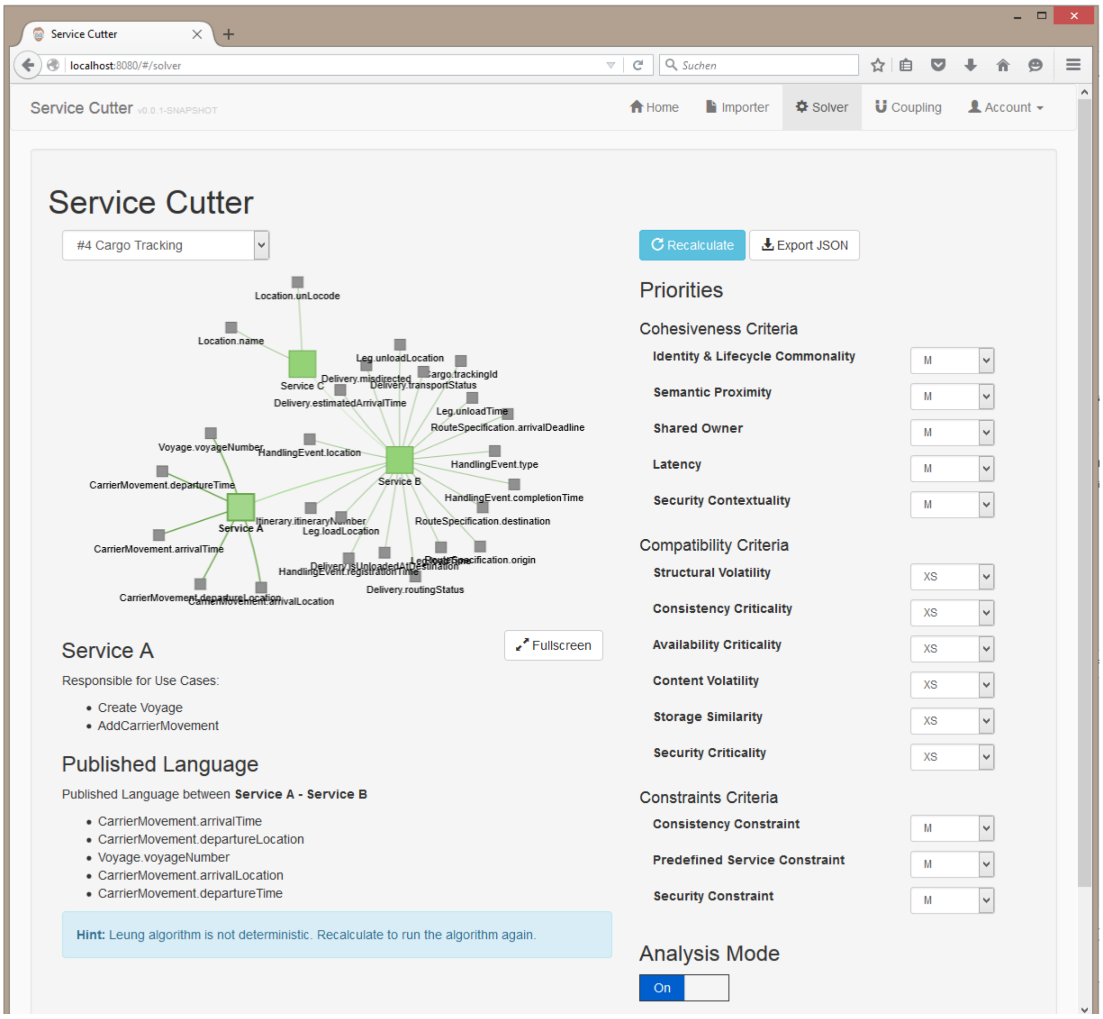
Figure 1.1: Coupling Criteria Catalog

Service Cutter

Complementary to the catalog we described an approach to process coupling criteria in a software to optimize loose coupling between services and high cohesion within services. We prototypically implemented the Service Cutter, shown in Figure 1.2, to verify this approach.

The Service Cutter analyzes a user's system based on its nanoentities. Nanoentities are elements used by a service to provide business capabilities. They are defined either as data fields, operations or artifacts. The system is decomposed into services by defining a certain number of services and assigning all nanoentities to exactly one service.

A user can specify his system by means of well established software artifacts such as the entity-relationship model or use cases. Based on these specifications, the coupling between nanoentities is quantified with a score for each coupling criterion.


Service Cutter
localhost:8080/#/solver
Service Cutter v0.0.1-SNAPSHOT
Home
Importer
Solver
Coupling
Account
Location.unLocode
Location.name
Service C
Delivery.misdirected
Delivery.transportStatus
Cargo.trackingId
Leg.unloadLocation
Delivery.estimatedArrivalTime
Leg.unloadTime
RouteSpecification.arrivalDeadline
Voyage.voyageNumber
HandlingEvent.location
HandlingEvent.type
Service B
CarrierMovement.departureTime
HandlingEvent.completionTime
Itinerary.itineraryNumber
Leg.loadLocation
RouteSpecification.destination
Service A
CarrierMovement.arrivalTime
RouteSpecification.origin
HandlingEvent.registrationTime
Delivery.isUnloadedAtDestination
Delivery.routingStatus
CarrierMovement.departureLocation
CarrierMovement.arrivalLocation
Fullscreen
Recalculate
Export JSON

Service Cutter

#4 Cargo Tracking

Priorities

Cohesiveness Criteria

Identity & Lifecycle Commonality	M
Semantic Proximity	M
Shared Owner	M
Latency	M
Security Contextuality	M

Compatibility Criteria

Structural Volatility	XS
Consistency Criticality	XS
Availability Criticality	XS
Content Volatility	XS
Storage Similarity	XS
Security Criticality	XS

Constraints Criteria

Consistency Constraint	M
Predefined Service Constraint	M
Security Constraint	M

Analysis Mode

On

Service A

Responsible for Use Cases:

- Create Voyage
- AddCarrierMovement

Published Language

Published Language between **Service A - Service B**

- CarrierMovement.arrivalTime
- CarrierMovement.departureLocation
- Voyage.voyageNumber
- CarrierMovement.arrivalLocation
- CarrierMovement.departureTime

Hint: Leung algorithm is not deterministic. Recalculate to run the algorithm again.

Figure 1.2: Screenshot Service Cutter

The exact importance of coupling is highly dependent on the context of a software system. Consistency for example is significantly divergent in a banking environment compared to an online social network. To reflect this, we rate the coupling criteria scores using priorities.

Decomposition by Graph Clustering

All scores are collected and utilized to construct a weighted, undirected graph. The nodes represent nanoentities and the weighted edges embody the strength of the coupling between two nanoentities.

Once the graph is constructed, a graph clustering algorithm calculates clusters cutting as few edges as possible. A cluster represents a candidate service. Edges connecting nodes of two clusters represent coupling between the services. This process produces candidate service cuts with high cohesion and low coupling. Figure 1.3 illustrates a graph created to analyze a sample application.

We utilized two complementary graph clustering algorithms. Girvan-Newman takes the desired number of clusters as parameter and is especially suitable for scenarios where a monolithic system is sequentially decomposed into services. The "Epidemic Label Propagation" algorithm by Leung computes the number of clusters by itself and therefore suggests a suitable number of services to the user.

We performed tests based on an imaginary "Trading System", heavily inspired by real banking software, and the sample application "Cargo Tracking" as introduced by Eric Evans in his book on Domain Driven Design. The Leung algorithm provided expected or applicable service cuts for both systems while Girvan-Newman only met our expectations for the Trading System.

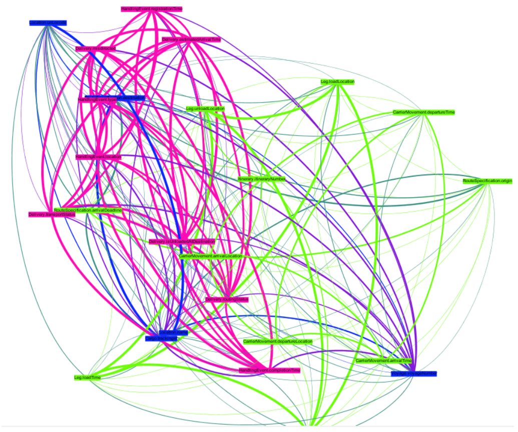


Figure 1.3: A graph created for the Cargo Tracking application. The colors represent the detected clusters.

Conclusion

In our thesis, we structured the architecturally significant requirements for service decomposition into the coupling criteria catalog. The test results suggest that these criteria are quantifiable and can be optimized leveraging algorithms and software.

The Service Cutter structures and assists the decision making process for new or already existing systems. We suggest that future projects either focus on tool development to integrate the Service Cutter into existing software development processes or invest in further research scoring and algorithms.

2. Introduction

This chapter introduces the project's goals, scope and context. The original project definition, signed at the beginning of project, is documented in Appendix E.

2.1 Hypothesis

D. L. Parnas published a paper titled *On the Criteria To Be Used in Decomposing Systems into Modules*[28] in 1972. Since then, decomposition of software systems has become an important area in the field of software engineering. As systems grew more complex, software engineers started to distribute modules over computer networks and hence called them services. Architectural styles like Software Oriented Architecture (SOA) have been introduced to tackle many challenges of such distributed systems.

Nevertheless, even with microservices, the latest incarnation of service orientation, decomposition is more described as an art than a structured discipline. C. Richardson writes in his popular introduction to microservices on InfoQ:

> *Deciding how to partition a system into a set of services is very much an art but there are number of strategies that can help. One approach is to partition services by verb or use case.*[31]

We consider the described strategies as suitable approaches to service decomposition. However, we assume that service decomposition can be approached in more structured way. This leads us to our first hypothesis:

> *The driving forces for service decomposition of a software system can be assembled in a comprehensive criteria catalog.*

To validate this first hypothesis, we created a comprehensive but not conclusive catalog of coupling criteria. Taking this structured approach to service decomposition a step further, we formulated a second hypothesis:

> *Based on the criteria catalog, a system's specification artifacts can be processed in a software to optimize loose coupling between services and high cohesion within services in a structured and automated way.*

To validate this second hypothesis, we developed a prototype based on the criteria catalog. This tool, hence called the "Service Cutter", analyzes a system's specification and suggests candidate service cuts in order to optimize loose coupling between services and high cohesion within services. A system's specification contains an entity-relation model, use cases, and further artifacts as illustrated in Figure 2.1.

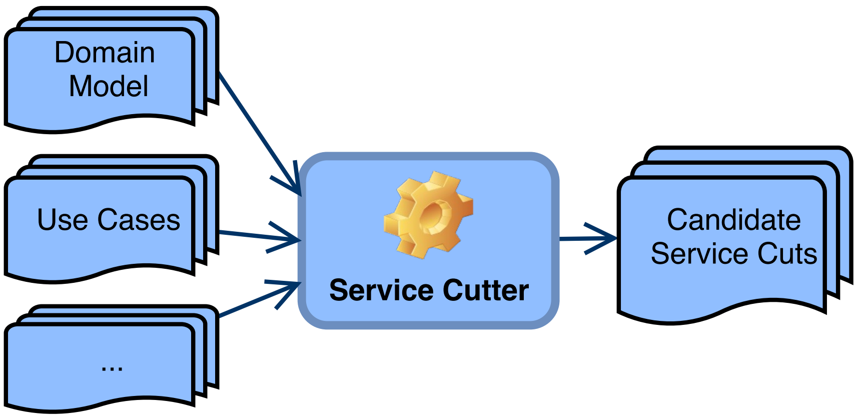


Figure 2.1: Input and output of the Service Cutter.

The Service Cutter's goal is to assist and advise a software architect or developer in his design decisions regarding service decomposition. The architect needs to assess the candidate service cuts and compare them with his expectations. The Service Cutter's mission is accomplished, if the architect's expectations are verified or unexpected but reasonable candidate cuts challenge his preoccupations.

2.2 Project Scope

This section describes the scope and boundaries of this thesis. It first defines some of the relevant terms used throughout this document.

A *system* refers to a software application whose architecture needs to be decomposed.

A *service* can be seen as a module providing a remote Application Programming Interface (API) to communicate with other services. The term is explained in more detail in Chapter 3.

Service decomposition refers to splitting a system's functionality and data into services. While we focus on service decomposition, most of the concepts are also true for non-distributed systems where a software internally is decomposed into modules.

Before a system can be decomposed, its functional and non-functional requirement need to be analyzed and specified in an entity-relationship model, use cases, and other artifacts. Based on these specifications the system can be decomposed into services. These are later implemented and connected using intra service communication. Figure 2.2 illustrates this process.

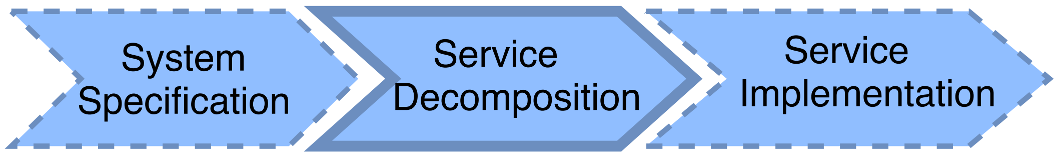


Figure 2.2: The thesis in the context of system development.

Our thesis focuses solely on service decomposition. Consequently, the following areas are not in scope:

- Requirements engineering and system specification need to be done before a system can be analyzed with the Service Cutter.
- Intra service communication is not in scope of this thesis. S. Newman documents in his book *Building Microservices*[26] multiple popular ways for intra service communication like Remote Procedure Calls (RPC), RESTful HTTP services or asynchronous event-based collaboration. Decomposition only defines *what* is communicated but now *how*.
- Composing multiple services into workflows or business processes using notations like Business Process Model and Notation (BPMN) is not in scope.
- Service decomposition tries to minimize coupling between services. Tactics like caches or Command Query Responsibility Segregation (CQRS), which try to lower the consequences of coupling introduced by decomposition, are not analyzed.

2.3 Context and Influences

The ideas and concepts presented are influenced by the work of many others. We reused and embodied existing concepts to our structured way of service decomposition.

2.3.1 Service Oriented Architecture

It was during a course titled *Advanced Distributed Systems Design using SOA & DDD* by Udi Dahan where our initial idea to assist service decomposition with an automated approach emerged. Dahan is the founder of NServiceBus[65], the most popular service bus for .NET and a well known Service Oriented Architecture (SOA) and Domain-Driven Design (DDD) expert. The approach to tackle service decomposition from the 4+1 View Model[20] is inspired by him. Approaching decomposition on the basis of data fields, or the later in the document introduced nanoentites, is motivated by his course.

Further SOA influences are provided by our supervisor throughout the project and during his course *Application Architecture* at Hochschule für Technik Rapperswil (HSR).

2.3.2 Microservices

In recent years, microservices substituted SOA as the trending architectural style, but can be seen as a new incarnation of the service oriented approach. Valuable concepts like service definitions or decomposition criteria are inspired by leading evangelists in this area such as Martin Fowler, Sam Newman, and Chris Richardson.

2.3.3 Domain-Driven Design

Nevertheless, decomposition is not solely a problem in distributed systems. Eric Evans introduced in his book *Domain-Driven Design: Tackling Complexity in the Heart of Software*[12] a collection of patterns to handle decomposition complexity. Especially the patterns *Aggregate*, *Entity*, *Published Language* and *Bounded Context* are integrated in our approach and serve as input or output of the Service Cutter.

2.4 Market Overview

We were not able to find projects that try to structure and automatically assist service decomposition. Nevertheless, there are several methodologies and decomposition tools tackling some of the relevant challenges.

2.4.1 Software Methodologies

The already introduced approaches SOA, microservices, and DDD discuss some decomposition criteria but do not provide a comprehensive criteria catalog.

Other approaches tackle related problems but are focused on different layers of software development. Object-Oriented Analysis and Design (OOAD) focuses more on abstractions like classes and object instances. We integrated OOAD artifacts like the Entity-Relationship-Model (ERM) as part of the system specification given to the Service Cutter as input. Business Process Management (BPM) lays an abstraction layer above services, focusing on business processes and therefore the usage of services rather than their identification and specification. A detailed analysis on the correlation of OOAD and BPM with service orientation was published by IBM[41]. Service oriented modeling approaches like Service-Oriented Modeling and Architecture (SOMA)[2] target similar questions as this thesis but do not provide detailed descriptions of decomposition approaches. SOMA suggests decomposition by use cases which resembles our decomposition criterion *Semantic Proximity* introduced in Chapter 4.

2.4.2 Decomposition Tools

Kenny Bastani suggests a graph based analysis to decompose monolithic software into microservices[4]. In his decomposition approach, he focuses on dependencies from user stories to RESTfull HTTP resources. He uses Neo4J GraphGist[64] to visualize the dependencies but does not run any automated analysis on the graph.

The barrio eclipse plugin[9] analyzes dependencies based on Java source code. It suggests a candidate package structure by leveraging the Girvan-Newman clustering algorithm. The tool has been published as part of a student's project of the Massey University, New Zealand.

After introducing our hypothesis's and the broader context of service decomposition, the next Chapter analyzes the definition of a service and service decomposition principles in more detail.

3. Domain Analysis

This chapter analyzes the concepts of services and service decomposition in more detail and concludes with a questionnaire that helps to assess decompositions.

3.1 Service Definition

Service is one of the most used terms in the field of software architecture and has been defined differently in many papers, books, and blog posts in numerous ways and various contexts. This section consolidates multiple definitions and defines service for this thesis.

3.1.1 Different Views on Services

The "4+1 View Model of Software Architecture" by Philippe Kruchten[20] describes software architecture using the views *Logical View*, *Physical View*, *Development View*, and *Process View* which are illustrated by *Scenarios*. During our research we discovered that the difficulty to clearly define the word service lies in the fact that different definitions focus on contrasting views. For this thesis we use multiple definitions for the term service depending if we write about the *logical* or the *physical* view of a service.

Logical Service

A service is the technical authority for a specific business capability
— Udi Dahan[8]

This definition focuses more on the logical or scenario view of a service than its technical representation. He further defines that all data and operations required to provide a business capability are owned by one and only one service.

Udi Dahan implies that a service is not restricted to a specific application, process, technology or layer. In fact, it contains required layers itself, including databases, logic, and User Interface (UI) code.

A logical service is autonomous and composed from many processes, webservices or databases, but keeps a clear boundary and interface against the outer world. Communication with other parts of the system only happens on a well defined interface on a common communication channel.

Bounded Context

Another concept describing logical services is the bounded context as defined in the Domain-Driven Design[11]:

> *A description of a boundary (typically a subsystem, or the work of a particular team) within which a particular model is defined and applicable.*

A model only used within one bounded context is defined and visible only in that context. Accordingly, a model used in multiple services needs to have a globally shared definition, defined as *Published Language* in the context of DDD[11]:

> *The translation between the models of two bounded contexts requires a common Language.*

The process of service decomposition as done by the Service Cutter automatically defines the published language of the system.

Physical Service

Martin Fowler describes a service as following:

> *A service will be used remotely through some remote interface, either synchronous or asynchronous.*[16]

This definition by Martin Fowler is based on the physical structure and is close to what recently has been advertised as a *microservice*:

> *In short, 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.*[13]

A process providing a remote API might provide business logic, pure technical functionality or a data store. A service commonly includes at least a data store, wrapped by a RESTful HTTP API. Physical services might be congruent with logical services but very often more complex cases split logical services in multiple physical services.

Should the Service Cutter Produce Logical or Physical Service Candidates?

The Service Cutter incorporates logical and physical aspects in the decomposition process but focuses more on the former.

Not all reasons to create physical services can be analyzed in a structured way. The Service Cutter cannot decide if a service using a database runs in a single process or connects to its database over a remote interface. Similarly, an operations team might decide to run different logical services on the same machine or even in the same process to save resources and simplify deployment. These decisions are often reasoned by operational aspects rather than the system's characteristics.

Nevertheless, some physical aspects can be analyzed. As an example, the Service Cutter is able to receive information about the storage requirements for the system's data. It suggests that data with very high storage requires an own service because a different database technology is necessary.

We define that the Service Cutter focuses on logical services while incorporating physical aspects whenever possible.

3.1.2 Nanoentities, Building Blocks for Services

Sam Newman writes in his book *Building Microservices*[26, p. 34]:

> *When you start to think about the bounded contexts that exist in your organization, you should be thinking not in terms of data that is shared, but about the capabilities those contexts provide the rest of the domain.*

In order to provide capabilities, a service needs resources. We identified three types of resources which are the building blocks of services:

Data A service has ownership over some of the system's data. It is the only instance responsible for changes on that data and optionally informs other services about changes. The data is often, but not necessarily, stored in a database. Data which is published to other services belongs to the published language of the system.

Operations A service has ownership over business rules and calculation logic. These operations are often, but not necessarily, based on the data the service owns.

Artifacts A service has ownership over artifacts. An artifact is a collection of data or operation results transformed into a specific format. An example is a business report which has been built using operations and data.

In order to enable a structured approach to service decomposition, we generalize these resources with the concept of a *nanoentity*. Examples for possible nanoentities are illustrated in Figure 3.1.

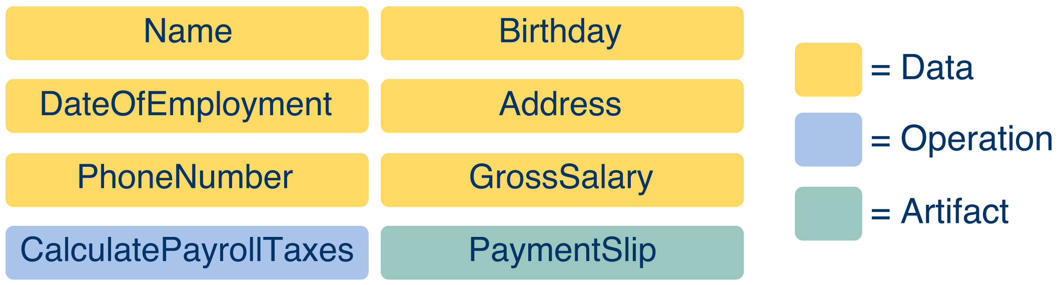


Figure 3.1: Nanoentities related to an employee.

A service must contain at least two type of nanoentities to be considered a logical service.

- Something only providing CRUD[1] functions on data is considered a database.
- Something only providing operations is considered a function.
- Something only providing artifacts is considered a resource or a database.

Service decomposition is the act of defining a number of services and assigning all nanoentities to the responsible service. The driving forces for decomposition are discussed in more detail in the next section.

3.2 Service Decomposition

Well experienced software architects decompose systems by reason of driving forces to ensure a maintainable, robust and consistent system with business relevance and good performance. This section describes the forces mostly considered by architects.

Decomposition has been a main discipline for programmers since early in the history of our industry. David L. Parnas published a paper entitled “On the Criteria To Be Used in Decomposing Systems into Modules” in 1972[28]. Shortly after, the terms *coupling* and *cohesion* as software design metrics appeared as part of the *Structured Design* technique[35]:

Coupling *A measure of how closely connected two routines or modules are.*
In software design, a measure of the interdependence among modules in a computer program.[36]

Cohesion *The manner and degree to which the tasks performed by a single software module are related to one another.*
In software design, a measure of the strength of association of the elements within a module.[36]

[1]Create, Read, Update, and Delete

Software architects commonly started to use these metrics to define that good architectures have high cohesion within and low coupling between its parts. Robert Martin later described a general principle to achieve loose coupling and high cohesion:

Single Responsibility Principle *Gather together the things that change for the same reasons. Separate those things that change for different reasons.*[23]

Starting from these principles, we analyzed different types of coupling and cohesion and created the decomposition model described in the next chapter.

4. Decomposition Model

The decomposition model describes the quality attributes of good service decomposition solutions and the criteria leading to such. This chapter starts with an overview over all defined coupling criteria and concludes with the definition of a good decomposition solution.

A coupling criterion describes an architecturally significant requirement why two nanoentities should or should not be owned by the same service. These criteria define the semantic model on which the Service Cutter is built on.

The coupling criteria are a product of literature research and a workshop assembling the collective software architect experience of our thesis advisor, our industry partner, and us. We transformed the resulting ideas into the following structured catalog.

4.1 Catalog Overview

We arranged the coupling criteria in a grid as shown in Figure 4.1.

The grid columns represent the following partitions:

Cohesiveness - Criteria describing the cohesiveness of nanoentities and therefore why they should belong to the same service.

Compatibility - Criteria describing the divergent characteristics of nanoentities. The service should not contain nanoentities with different characteristics. Examples for incompatible characteristics are *High*, *Eventually*, and *Weak* for the criterion *Consistency Criticality.*

Constraints - Criteria describing constraints which enforce that groups of nanoentities must be distributed amongst different services or form a service by itself.

Communication - Criteria describing which nanoentities are suitable to be used as part of the published language shared between services.

The rows are inspired by the “4+1 View Model of Software Architecture” by Kruchten[20]. Domain is an enhancement of the *Logical View*, quality resembles the *Process View* and physical matches the *Physical View* but also includes predefined service constraints. Security is included in the *Development View* by Kruchten. We decided to promote it to a separate layer as it was a prominent requirement in our workshop and other aspects of the *Development View* were not relevant for our application.

Domain Criteria describing nanoentities from a business domain perspective.

Quality Criteria describing the quality requirements of a nanoentity directly or related to a use case. Non-functional requirements are predominantly represented in this row.

Physical Criteria describing the physical or technological aspects of nanoentities.

Security Criteria describing nanoentities from a security perspective.

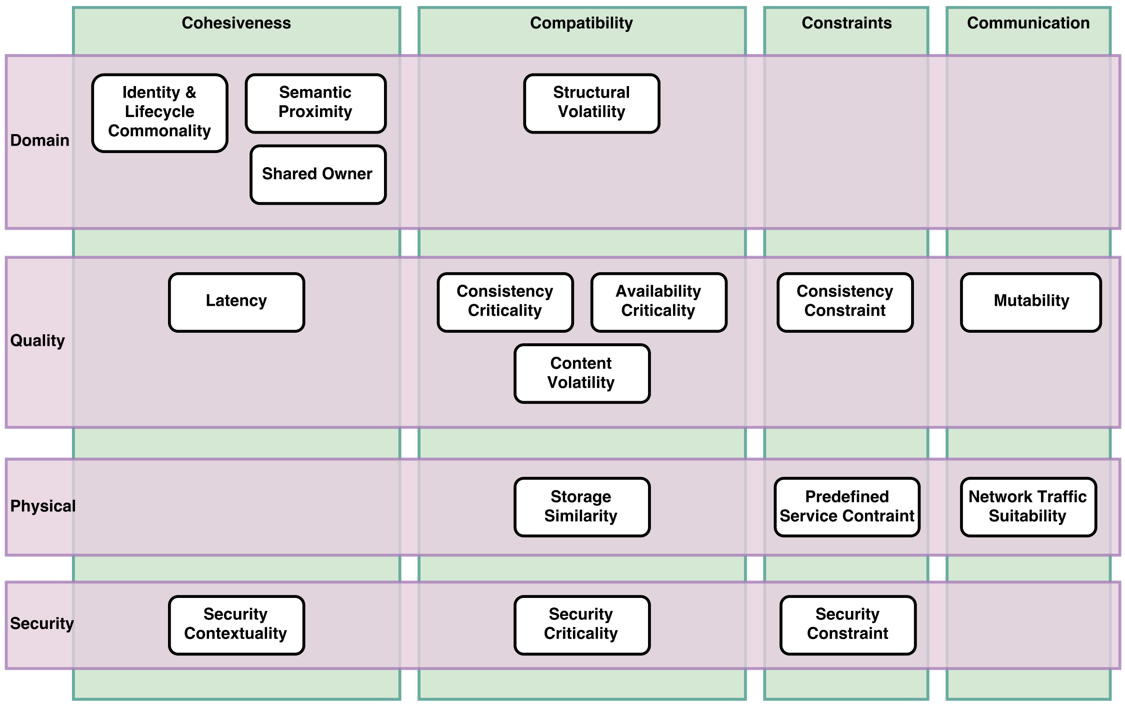


Figure 4.1: Coupling Criteria Catalog

4.2 Coupling Criteria Cards

We specified all coupling criteria listed in the catalog as “CC cards” like the following:

CC-1 Identity & Lifecycle Commonality

Description	Nanoentities that belong to the same identity and therefore share a common lifecycle.
User Representation	- Entity-Relationship Models - Domain-Driven Design Entities.
Literature	Entity definition in Domain-Driven Design: *Some objects are not defined primarily by their attributes. They represent a thread of identity that runs through time and often across distinct representations.*[12]
Type	Cohesiveness
Perspective	Domain
Characteristics	n/a

Cards share the following information:

Description explains the coupling criteria in more detail.

User Representation lists concepts or artifacts familiar to the architect that can be used to feed the criteria information into the Service Cutter.

Literature references the coupling criteria to descriptions in existing literature.

Type Cohesiveness, Compatibility, Constraint or Communication.

Perspective Domain, Quality, Infrastructure or Security.

Characteristics can be applied to a nanoentity and are defined for criteria of type “compatibility”.

CC-2 Semantic Proximity

Description	Two nanoentities are semantically proximate when they have a semantic connection given by the business domain. The strongest indicator for semantic proximity is coherent access on nanoentities within the same use case.
User Representation	- Coherent access or updates on nanoentities in use cases. - Aggregation or association relationships in an entity-relationship model.
Literature	Chris Richardson on microservice decomposition: *Deciding how to partition a system into a set of services is very much an art but there are number of strategies that can help. One approach is to partition services by verb or use case.*[30] Single Responsibility Principle by Robert Martin: *Gather together the things that change for the same reasons. Separate those things that change for different reasons.*[23]
Type	Cohesiveness
Perspective	Domain
Characteristics	n/a

CC-3 Shared Owner	
Description	The same person, role or department is responsible for a group of nanoentities. Service decomposition should try to keep entities with the same responsible owner together while not mixing entities with different responsible instances in one service.
User Representation	User defined persons, roles or departments with each containing a group of nanoentities. A nanoentity can only be associated once.
Literature	Conway's law: *Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.*[5] Single Responsibility Principle by Robert Martin: *Gather together the things that change for the same reasons. Separate those things that change for different reasons. [...] However, as you think about this principle, remember that the reasons for change are people. It is people who request changes. And you don't want to confuse those people, or yourself, by mixing together the code that many different people care about for different reasons.*[23]
Type	Cohesiveness
Perspective	Domain
Characteristics	n/a

CC-4 Structural Volatility	
Description	How often change requests need to be implemented affecting a nanoentity's structure.
User Representation	Classification of nanoentities in characteristics.
Literature	David Parnas on modular programming: *We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others.*[28]
Type	Compatibility
Perspective	Domain
Characteristics	Often, Normal *(default)*, Rarely

CC-5 Latency

Description	Groups of nanoentities with high performance requirements for a specific user request. These nanoentities should be modelled in the same service to avoid remote calls.
User Representation	Use cases with latency requirements. All nanoentities read or written by the same use case belong a group. A nanoentity can belong to multiple groups.
Literature	Design guidelines for application performance by Microsoft: *Minimize round trips to reduce call latency. For example, batch calls together and design coarse-grained services that allow you to perform a single logical operation by using a single round trip.*[24]
Type	Cohesiveness
Perspective	Quality
Characteristics	n/a

CC-6 Consistency Criticality

Description	Some data such as financial records loses its value in case of inconsistencies while other data is more tolerant to inconsistencies.
User Representation	Classification of nanoentities in characteristics.
Literature	Werner Vogels on consistency requirements: ***Strong consistency****: After the update completes, any subsequent access (by A, B, or C) will return the updated value.* ***Weak consistency****: The system does not guarantee that subsequent accesses will return the updated value. A number of conditions need to be met before the value will be returned. The period between the update and the moment when it is guaranteed that any observer will always see the updated value is dubbed the inconsistency window.* ***Eventual consistency****: The storage system guarantees that if no new updates are made to the object eventually all accesses will return the last updated value.*[37]
Type	Compatibility
Perspective	Quality
Characteristics	High consistency *(default)*, eventual consistency, weak consistency

CC-7 Availability Criticality

Description	Nanoentities have varying availability constraints. Some are critical while others can be unavailable for some time. As providing high availability comes at a cost, nanoentities classified with different characteristics should not be composed in the same service.
User Representation	Classification of nanoentities in characteristics.
Literature	Eoin Woods on availability and resilience: *Getting your availability characteristics wrong can be very expensive. However, increased online availability comes at a cost, whether in terms of more hardware, increased software sophistication, or redundancy in your communications network.*[33]
Type	Compatibility
Perspective	Quality
Characteristics	Critical, Normal *(default)*, Low

CC-8 Content Volatility

Description	A nanoentity can be classified by its volatility which defines how frequent it is updated. Highly volatile and more stable nanoentities should be composed in different services.
User Representation	- Volatility can be calculated from use case definitions if they are equipped with a frequency information. - Nanoentities can be classified by data types to determine the volatility: Master Data (regularly), Reference Data (rarely), Transaction Data (often) and Inventory Data (often) to determine the volatility.
Type	Compatibility
Perspective	Quality
Characteristics	Often, Regularly *(default)*, Rarely

CC-9 Consistency Constraint	
Description	A group of nanoentities that have a dependent state and therefore need to be kept consistent to each other.
User Representation	An aggregate as defined in domain-driven design.
Literature	Aggregate as defined in domain-driven design by Eric Evans: *A cluster of associated objects that are treated as a unit for the purpose of data changes. External references are restricted to one member of the aggregate, designated as the root. A set of consistency rules applies within the aggregate's boundaries.*[12] Udi Dahan on service decomposition: *If modifying the value of one attribute involves changing the value of another, then those two attributes should fall under the responsibility of the same service.*[3]
Type	Constraint
Perspective	Quality
Characteristics	n/a

A *Consistency Constraint* differs from the *Consistency Criticality* coupling criteria in such a way that the constraint groups a set of nanoentities ensuring their atomic processing.

For example, a payment and the account balance should be linked using a *Consistency Constraint.* The Service Cutter therefore guarantees that those fields are in the same service. The *Consistency Criticality* criterion is of type compatibility and only separates divergent characteristics.

CC-10 Mutability

Description	Immutable information is much simpler to manage in a distributed system than mutable objects. Immutable nanoentities are therefore good candidates for the published language shared between two services. Service decomposition should be done in a way that favors sharing immutable nanoentities over mutable ones.
User Representation	Classification of each nanoentity in mutable or immutable.
Literature	Udi Dahan on finding service boundaries: *When you find something immutable, that is an indication that there is some loose coupling between the two sides passing immutable data.*[6]
Type	Communication
Perspective	Quality
Characteristics	n/a

CC-11 Storage Similarity

Description	Storage that is required to persist all instances of a nanoentity.
User Representation	The user classifies nanoentities into the given characteristics. The classification is system specific. In one system a nanoentity classified as *huge* might need 1MB, but in another 1GB storage per instance.
Type	Compatibility
Perspective	Infrastructure
Characteristics	Huge, Normal *(default)*, Tiny

CC-12 Predefined Service Constraint

Description	There might be the following reasons why some nanoentities forcefully need to be modeled in the same service: - Technological optimizations - Legacy systems
User Representation	User defined service with each containing a group of nanoentities. Each nanoentity can be associated only once.
Type	Constraint
Perspective	Infrastructure
Characteristics	n/a

CC-13 Network Traffic Similarity

Description	Service decomposition has a significant impact on network traffic, depending on which nanoentities are shared between services and how often. Small and less frequently accessed nanoentities are better suited to be shared between services.
User Representation	Use cases define the frequency of access on nanoentities. The size of each entity can be determined by CC-11: *Storage Similarity*
Type	Communication
Perspective	Infrastructure
Characteristics	n/a

CC-14 Security Contextuality

Description	A security role is allowed to see or process a group of nanoentities. Mixing security contexts in one service complicates authentication and authorization implementations.
User Representation	User defined security roles with each containing a group of nanoentities. A nanoentity can be associated to multiple groups.
Type	Cohesiveness
Perspective	Security
Characteristics	n/a

CC-15 Security Criticality

Description	Criticality of an nanoentity in case of data loss or a privacy violation. Represents the reputational or financial damage when the information is disclosed to unauthorized parties. As high security criticality comes at a cost, nanoentities classified with different characteristics should not be composed in the same service.
User Representation	Classification of nanoentities in characteristics
Type	Compatibility
Perspective	Security
Characteristics	Critical, Internal *(default)*, Public

CC-16 Security Constraint

Description	Groups of nanoentities are semantically related but must not reside in the same service in order to satisfy information security requirements. This restriction can be established by an external party such as a certification authority or an internal design team.
User Representation	Demilitarized zones or other groups of nanoentities that should be composed to different services.
Type	Constraint
Perspective	Security
Characteristics	n/a

4.3 Decomposition Questionnaire

The following questionnaire is based on the coupling criteria and aims to fulfill the principles introduced in Chapter 3. In a good decomposition solution, the answer to all those questions should be *yes*.

1. Does the service cut comply all constraint criteria?
2. Does the service cut combine as few nanoentities with diverging characteristics into one service as possible?
3. Does each service depends on as few nanoentities of other services as possible? A use case should cross as few service boundaries as possible.
4. Are the nanoentities that are part of a published language between services suitable for intra service communication?
5. Is the coupling between services similar? It is not the size of services that requires homogeneity within the system but the amount of published language between services.
6. Are there not too many services? This is called the nanoservice antipattern[32].
7. Are there not too few services? This is a monolithic architecture.

These questions form a checklist to validate a service cut suggested by the Service Cutter. This approach is outlined in Section 9.2.8.

After presenting the decomposition model, the next chapter defines the requirements for the prototypically implementation of the Service Cutter.

5. Service Cutter Requirements

This chapter describes the (non-)functional requirements of the Service Cutter and covers the characteristics of its target users.

The requirements in this chapter are not prioritized. As part of the sprint planning meetings, these requirements are transformed to tasks and prioritized. The description in this chapter is meant to provide a high level overview and establish a common sense between all stakeholders of this project.

5.1 Personas

The personas are inspired by a series of discussions in meetings and workshops with our stakeholders.

5.1.1 Junior Jedi-Master

Junior has been a fast learning and aspiring developer since he graduated from university with a master's degree in information technology eight years ago. In his new job, Junior finds himself in the role of an architect for a new and promising product that enjoys the support of well known investors. In consequence of his experience in distributed systems, he has been assigned with the task to decompose the system's business model into logical services of which each will be split into multiple separately deployable microservices.

Junior strongly believes in automation and using every tool available to support and complete his work. For his current project he plans to try the Service Cutter as a foundation and verification of his architectural decisions.

5.1.2 Walter Wisenheimer

Walter is an architect with many years of experience in the industry and has built numerous systems already. Walter has seen many automation concepts and tools failing their goals. In Walter's view a natural and obvious outcome – an architect's world is too complex to model and automate in a system or algorithm.

After a longer discussion with a very motivated junior developer who recently joined his company, Walter starts to see the benefits of the well structured format the Service Cutter organizes architecturally relevant information. Using the tool to structure and document his systems characteristics might be of benefit as currently a lot of his precious knowledge remains tacit[42].

Walter decided to try the Service Cutter to structure the information for the current project he is working on.

5.1.3 Stan Student

Stan studies Computer Science and as part of his class in service oriented software architecture he is supposed to design a set of services for the Cargo Tracking[49] domain. The Service Cutter guides him through the important decisions by asking a set of questions and presents him a set of possible service cuts.

Being overwhelmed of all the data requested by the Service Cutter, he would like to configure the tool to only focus on the data he got provided in his exercises.

Stan then discusses the advantages and disadvantages of the presented options with his fellow students. He furthermore asks his Professor about the to him unknown criteria the Service Cutter requested and why that information might have an impact on service decomposition.

5.1.4 Tom Tutor

Tom wants to introduce his students to the software architecture craft. He uses the Service Cutter to visualize the different ways of distributing data into services during his lectures. By changing the calculation parameters, he can demonstrate that software architecture mostly depends on the context of the requirement. The same problem might have different solutions in varying circumstances.

5.1.5 Eddie Enterprise

Eddie is an enterprise architect employed by a large software consulting company. He usually works for a couple of months on a project. His customers expect from him that he influences important decisions even beyond his assignment. To achieve this, he decided together with the local project team that every decision they take has to be documented using a structured approach. He would like to utilize the Service Cutter's approach to ensure that all important aspects of coupling and cohesion are considered as part of the service decomposition.

5.2 Functional Requirements

The Service Cutter faces the functional requirements presented in this section.

5.2.1 Coupling Criteria

The criteria of type *Cohesiveness*, *Compatibility* and *Constraints* must be supported. The Service Cutter needs to rate for each criterion which nanoentities need be placed in one service and which need to be separated over multiple services. Within this rating, every coupling criterion is equally respected.

Criteria of type *Communication* do not describe the need to merge or separate nanoentities but characterize nanoentities suitable for inter service communication and are handled with a lower priority than other criteria.

5.2.2 User Representations

To achieve better usability, the user does not need to know the exact definitions of the coupling criteria or their internal structure. He can use well known software engineering artifacts called *user representations* to describe his system, from which the relevant nanoentities and coupling criteria data will be extracted by the Service Cutter. At least the following User Representation must be supported by the system:

Entity Relationship Diagram containing entities and their relations to each other. Each entity contains a list of nanoentities building the basis for a systems analysis.

Use case containing at least information about all nanoentities read or written in a particular case.

Categorization of the nanoentities in different characteristics per compatibility coupling criterion.

5.2.3 Priorities

As every coupling criteria is respected equally in the rating process, the user needs an additional way to prioritize criteria according to its system characteristics. For an application processing financial data, security might be more important than network traffic. For an application that needs to support high volumes of data, volatility and resilience might be a primary focus.

The system should allow to change priorities for all supported coupling criteria while providing reasonable defaults.

5.2.4 Candidate Service Cuts

Based on the input provided by user representations and the defined criteria priorities, the Service Cutter produces a set of candidate service cuts for the analyzed system. The definition of a good service cut is elaborated in Section 4.3

A candidate service cut can be exported in a machine-readable format.

5.2.5 Visualize Published Language

If relevant user input (e.g. use case definitions) is available, the Service Cutter is able to tell which service depends on which nanoentities of other services. These dependencies need to be visualized. All nanoentities shared between different services make up the published language of the system and need to be visualized as such.

Identify Hard Architectural Decisions

The rating of two solutions might be similar so that is not clear which variant is the better one according to the user's priorities. Such cases indicate difficult design decisions the architect has to take. Presenting these cases and the impact they have on each coupling criteria, the Service Cutter provides great support for the architect in identifying and taking architectural decisions.

5.3 Non-Functional Requirements

The following non-functional requirements should be satisfied by the Service Cutter.

5.3.1 Usability

A software architect should be able to use the software without any training. All controls are clearly named and, where appropriate, documented using an inline user manual including representative samples.

The user gets well introduced to the important concepts of the Service Cutter. These concepts include:

1. Nanoentities
2. Coupling criteria, their meaning and decomposition impact
3. User representations and their impact on coupling criteria
4. Coupling criteria priorities

Often used configuration like coupling criteria priorities should be shown directly to the user to encourage its usage. More advanced configuration like the values of characteristics should be accessible for the user but do not need to be shown directly in a standard workflow.

5.3.2 Simplicity

A simple system analysis can be achieved with not more than 5 clicks. All steps are provided with useful defaults that can be changed.

5.3.3 Performance

All regular user interactions should not take more than one second.

Calculations of service cuts should meet the following conditions assuming a data set of 2000 nanoentities.

- Calculations that are used once per day should take less than 10 minutes.
- Calculations that are used once per minute should take less than 5 seconds.

5.3.4 Monitoring, Logging, Deployment, Availability

The scope of this thesis is to prototypically implement the Service Cutter. Operational aspects only need to be covered on a basic level:

- Log files should be written using SLF4J[74].
- Deployment should be provided with Docker[51] containers.

5.3.5 Fault Tolerance

As the prototype does not need to handle every possible use case or unexpected input, a common error handling needs to be built into the application and ensures that operations continue even in case of unexpected input or state.

5.3.6 Maintainability

In case of a success the prototype might be the basis for further development or other thesis projects. It should be built in clearly separated modules (or even remote services) to ensure good maintainability. At least the following modules need to be clearly separated:

1. Internal representation of coupling criteria
2. Data of a user's system (nanoentities, use case definition etc.)
3. Decomposition algorithm (Solver)

4. User interface

If one or multiple of the modules are implemented as physical services, the remote API needs to be implemented as RESTful HTTP interface.

The application should leverage existing open source frameworks or libraries wherever possible to ensure a minimal maintenance effort.

5.3.7 State-of-the-Art Technology

To support further development of the Service Cutter the prototype needs to be implemented in state-of-the-art technology. Our industry partner has suggested the following technology stack:

1. Java Spring[70] as a base framework.
2. JHipster[60] with Spring Boot[68] for project setup.
3. AngularJS[43] and Bootstrap[47] for the user interface.
4. Docker[51] for container and deployment configuration and handling.

5.3.8 License

All involved parties decided to release the Service Cutter under the terms of the Apache 2.0 open source license.

After defining all (non-)functional requirements, the next Chapter outlines design and implementation steps which were taken to satisfy these requirements.

6. Service Cutter Design and Implementation

To satisfy the requirements defined in Chapter 5, the Service Cutter was built. This chapter documents design and implementation aspects.

6.1 Overview

This overview introduces the Service Cutter process and its most important concepts.

6.1.1 Service Cutter Concepts

Figure 6.1 illustrates the important concepts the Service Cutter is based on.

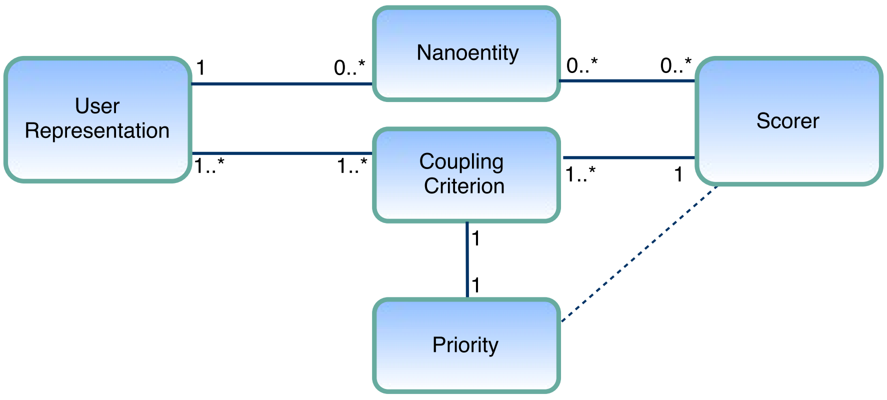


Figure 6.1: Service Cutter Concepts

The illustration does not show technical representation of the concepts but rather their logical relations:

- A user representation might contain a list of nanoentities and contains data relevant for one or multiple coupling criteria.
- A coupling criterion analyzes one or multiple user representations considering the given priority. Each coupling criterion has a scorer assigned.
- A scorer calculates coupling between nanoentities and is used for one or multiple coupling criteria.
- A nanoentity is imported by a user representation. Its coupling to other nanoentities is scored by scorers containing relevant information.

6.1.2 Service Cutter Layers

The Service Cutter is split into two layers shown in Figure 6.2. The *Engine* contains all coupling criteria information, stores a user's system specification, and calculates candidate service cuts. The Engine provides a RESTful HTTP web service for all relevant interactions. Based on this API, the web based *Editor* provides a graphical user interface.

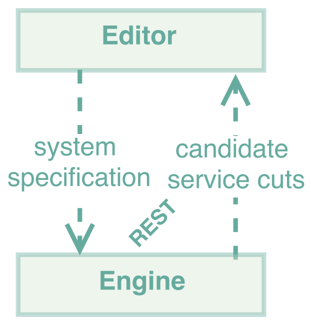


Figure 6.2: Service Cutter Layers

6.1.3 Service Cutter Process

The Service Cutter's decomposition calculation is based on nanoentities. In a first step, nanoentities are inserted into the Service Cutter and define a system to be analyzed. The user then specifies his system with user representation in a second step. During the third step, the decomposition process provides candidate service cuts showing how the nanoentities could be split into services. The user finally analyzes the cuts and compares them with his own expectations. The process is illustrated in Figure 6.3.

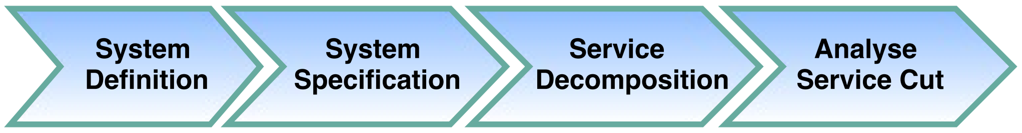


Figure 6.3: Service Cutter Process

Step 1: System Definition

In this step all nanoentities of the system to be analyzed are loaded in the Service Cutter. For the prototype, the only way to define a system's nanoentities is by uploading an ERM. An entity–relationship model consists of data fields, entities and relationships. The data fields are imported as nanoentities and define the system to be analyzed.

Step 2: System Specification

The second step adds coupling information to the system's nanoentities by uploading user representations. By uploading the ERM, the first coupling information have already been added. The entities define the *Identity & Lifecycle Commonality* criterion data and relationships of type aggregation influence the *Semantic Proximity* criterion.

The Service Cutter considers entities connected with an inheritance or a composition relation to have one identity and lifecycle. The Engine internally combines entities connected by these relations to one group of nanoentities. In order to determine which entities can be combined, we perform a topological sort[27] on the ERM relations.

The ERM is just one of many user representations listed in Section 6.2 to specify a system.

Step 3: Service Decomposition

Once the user has defined and specified his system, he optionally sets coupling criteria priorities matching his system requirements and then starts the decomposition process. The Engine groups nanoentities in a way that a good decomposition solution as described in Section 4.3 is found.

The Editor then presents the candidate service cuts to the user.

Step 4: Analyze Service Cuts

Once candidate service cuts have been presented, the user needs to interpret them and compare them with his own expectations.

If use cases have been imported as user representations, the Service Cutter defines which candidate service is responsible for which use case. A use case is assigned to the service owning most of the relevant nanoentities of that use case.

Because use cases are assigned to services, the published language between services can be calculated. The published language contains all nanoentities a service needs from other services to process his use cases. Use case responsibilities and the published language are presented to the user assuming they have been loaded into the Service Cutter.

The candidate service cuts can be exported as a JavaScript Object Notation (JSON) file as defined by the JSON Schema listed in Appendix D.2.

After introducing a broad overview, the next sections describe the introduced concepts in more detail.

6.2 User Representations

The Service Cutter provides a data import based on the user representations.

A user representation is a concept familiar to the architect that can be used to feed the criteria information into the Service Cutter. The following user representations are supported:

- An **ERM** consists of data fields, entities and relationships. The data fields are imported as nanoentitites, the entities and relationships of type composition or inheritance as *Identity & Lifecycle Commonality* and relationships of type aggregation as *Semantic Proximity*.
- **Use cases** primarily describe the *Semantic Proximity* of a group of nanoentities. They can also be marked as latency critical which then results in *Latency*.
- **Shared owner group** represents a person, role or department that is responsible for a group of nanoentities, defining *Shared Owner*.
- An **aggregate** is a DDD pattern defining a group of nanoentities that require consistency to each other which results in *Consistency Constraint*.
- The **entity** is a DDD pattern defining a common lifecycle and identify for a group of nanoentities. This defines *Identity & Lifecycle Commonality*.
- A **predefined service** represents a service that already exists and therefore is harder or impossible to change. It defines the identically named criterion *Predefined Service*.
- **Separated security zones** represent groups of nanoentities that should not be combined into a common service for security reasons. They are defining the *Security Constraints*.

- A **security access group** represents a group of nanoentities that share a common security context, defining the *Security Contextuality*.
- **Compatibilities** can be used to import all coupling criteria of type *Compatibility*.

Figure 6.4 lists all user representations and indicates which coupling criteria they are linked to.

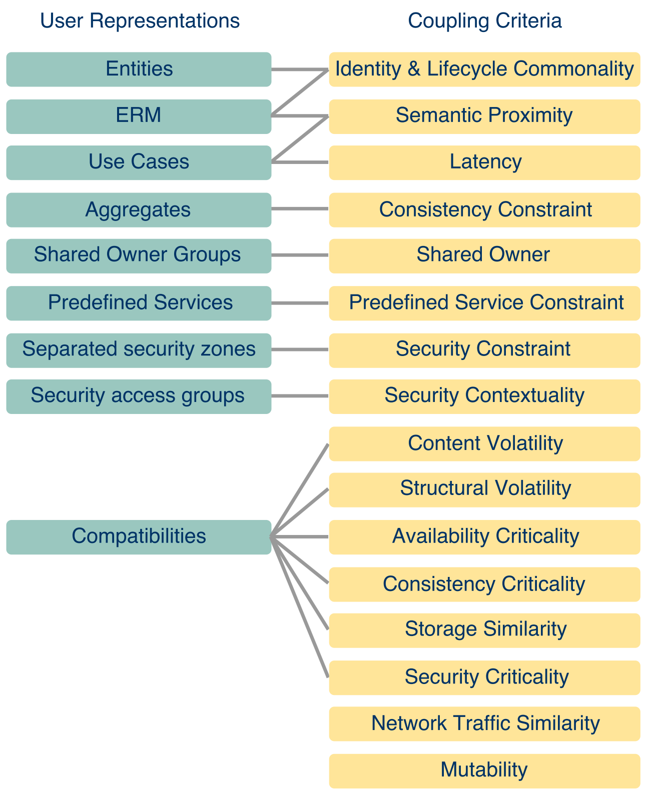


Figure 6.4: User representations and the related coupling criteria

The technical format including JSON schema files of all user representations is documented in Section 6.6.4 or in more detail on the GitHub wiki[67].

6.3 Decomposition by Graph Clustering

Service decomposition describes the task of grouping nanoentities into services. Achieving a good solution according to the defined coupling criteria as described in Section 4.3 requires a non trivial algorithm. We approach this problem by using a weighted undirected graph and clustering algorithms as described in this section. Other approaches have been evaluated but could not provide satisfying results as documented in Appendix A.

To create a weighted undirected graph, every nanoentity in the model is represented by a node. Edges define a relationship between two nanoentities. The weight on each edge shows how *close* or *cohesive* two nanoentities are. The higher the weight, the more likely they should belong to the same service. Figure 6.5 shows an abstract example of such a graph.

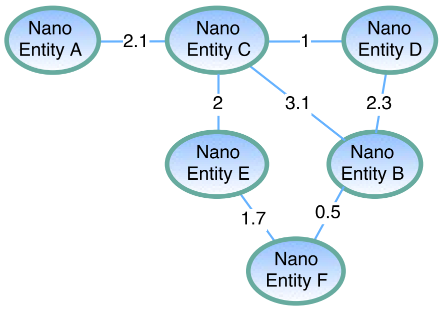


Figure 6.5: Example of a Weighted Graph

Figure 6.6 outlines a solution sketch for this approach. User representations like an ERM or use cases are imported. The *Importer* component extracts the nanoentities and the coupling meta data and stores this information in a database.

The *Solver* then creates all nodes from the nanoentities and builds weighted edges between them according to the stored coupling criteria instances. This task is complex for the following reasons:

1. Coupling criteria are not homogeneous as described in Section 4.1. Cohesiveness criteria describe why nanoentities should belong to the same service while compatibility criteria ask for separation of nanoentities. Constraints criteria might require both. Criteria of type communication need to be processed differently as they define which nanoentities are suitable to be transmitted across services. The information from all types needs to be represented with a single unit to define a number used as a weight of an edge.
2. Using a single number with only one unit of measurement bears the risk of unintended change of the relative importance between the coupling criteria. It is important to carefully assure that each coupling criterion uses the same range of numbers to allow comparison and prioritization of each criterion.
3. To allow the user to define the specific requirements of his system, priorities per coupling criteria can optionally be defined to influence the weights of each coupling criteria instance.

The *Clustering Algorithm* then analyzes the graph and creates clusters of nanoentities so that as few edges as possible need to be cut.

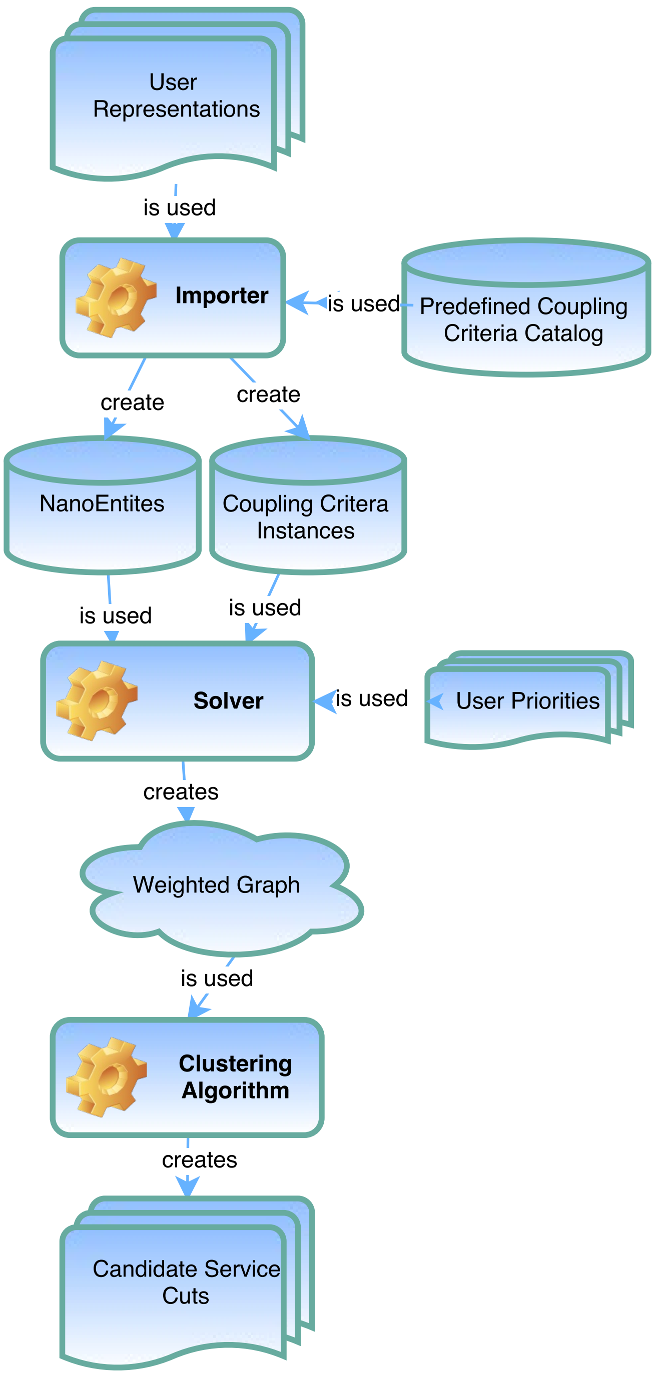


Figure 6.6: Solution Sketch for a Weighted Graph with a Clustering Algorithm

6.4 Clustering Algorithms

A clustering algorithm is required to split the undirected, weighted graph into groups of nodes having as few connections between the clusters as possible.

A full comparison of the evaluated algorithms is attached in Appendix C.

The remaining three algorithm candidates are Girvan-Newman, MCL, and Leung. Leung is provided by the GraphStream project. Girvan-Newman and MCL are implemented as Gephi plugins. Gephi is a desktop platform with a well developed UI to explore and visualize complex graphs and network systems. The platform provides a toolkit to use its functionality without a UI. The source code of the algorithms can be extracted from the plugins as Java Archive (JAR) files using the UnpackNBM tool[75].

Table 6.1 documents the detailed evaluation of the three candidate algorithms.

Table 6.1: Evaluation of cluster algorithms

	MCL	**Leung**	**Girvan-Newman**
Author	Stijn van Dongen[10]	U. N. Raghavan[29], Ian X.Y. Leung[22]	M. E. J. Newman, M. Girvan[25]
Year	2000	2007/2009	2003
Name	Markov Cluster Algorithm	Epidemic Label Propagation	Girvan–Newman
Approach	Random walks	Labels spread to their neighbors	Edge betweenness, based on shortest-paths
Performance *N: Nodes* *E: Edges*	$O(Nk^2)$ *k: pruning constant* [10, p.126]	$O(EN)$ [22, p.3]	$O(N^3)$[25, p.14]
Java Implementation	Plugin of Gephi[57]	GraphStream project[58]	Plugin of Gephi[57]
Test result	Bugs	Good	Good
Deterministic	No	No	Yes
Number of clusters parameter	No	No	Yes

The Markov Cluster (MCL) algorithm is theoretically suitable to calculate the clusters. However, we found the Java implementation not to be mature enough as the output contains overlapping or missing nodes so that the *distinct clusters* requirement is not met. Leung and Girvan-Newman are used in the Service Cutter and both provide good results as documented in Appendix B.

6.4.1 Girvan-Newman

M. E. J. Newman and M. Girvan[25] proposed to use a divisive approach to graph clustering. This approach repeatedly finds the least similar connected pair of nodes and removes the edges between them. The least similar connected pair is defined by *edge betweennees* which can be calculated using different approaches as described by Girvan-Newman. The Gephi implementation used in the Service Cutter calculates edge betweenness as the number of shortest paths of any pair of nodes running through an edge. Edges with the highest edge betweenness are removed first. This process divides the graph into smaller and smaller components and can be stopped at any time to select the components at this time to be the graph clusters.

In the ordered graph in Figure 6.7 Girvan-Newman would remove the edges in ascending order. The graph is split into 4 clusters after 2 iterations.

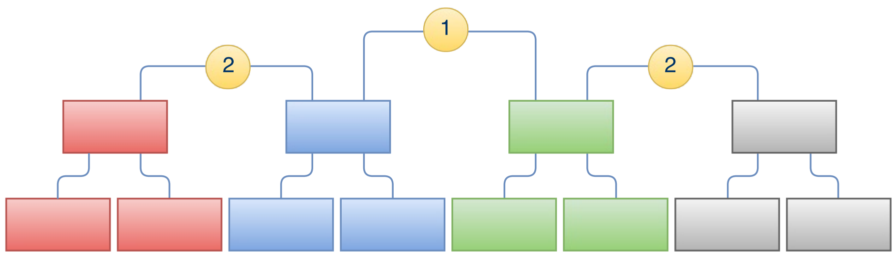


Figure 6.7: Girvan-Newman forms clusters by removing edges iteratively

Girvan-Newman receives the number of clusters as a parameter and stops as soon as the desired number has been reached. One iteration might remove multiple edges with the maximum edge betweenness. One iteration can therefore produce numerous new clusters so that the requested number of clusters cannot be provided. In this case the gephi implementation looks for a result having the least difference in number of clusters, The Service Cutter displays a warning that the requested number of services could not be provided.

As this algorithm does not include random elements, the proposed clusters are always identical.

6.4.2 Leung

In 2009 Leung *et al*[22] refined the "Epidemic Label Propagation" algorithm as proposed by Raghavan *et al*[29] in 2007.

Raghvan described the algorithm as follows:

> *Each node in the network chooses to join the community to which the maximum number of its neighbors belong to, with ties broken uniformly randomly. We initialize every node with unique labels and let the labels propagate through the network. As the labels propagate, densely connected groups of nodes quickly reach a consensus on a unique label.*[29, p. 4].

Leung does not specify an order in which nodes are to be processed. Therefore, the labels may spread differently in repeated runs. We witnessed varying clusters while testing the Service Cutter as document in Appendix B.

The algorithm occasionally forms so called "monster" families as described by Leung *et al* resulting in a drop of cohesion in a cluster.

> *[. . .] certain communities do not form strong enough links to prevent a foreign "epidemic" to sweep through.*[22, p. 5]

Leung *et al* therefore refined the algorithm by adding a "Hop Attenuation Factor" δ to prevent labels from spreading across two clusters. He explains:

> *The value δ governs how far a particular label can spread as a function of the geodesic distance from its origin. This additional parameter adds in extra uncertainties to the algorithm but may encourage a stronger local community to form before a large cluster start to dominate.*[22, p. 5]

These parameters decelerates the propagation of a cluster from its origin and therefore encourages a stronger local cluster to form before a large cluster starts to dominate.

We observed monster clusters especially in the DDDSample documented in Appendix B.2 and were able to reduce their likelihood by increasing the hop attenuation factor to a value of $\delta \approx 0.55$ compared to a default of $\delta = 0.1$ as provided by the GraphStream implementation. We decided to keep the value of $\delta = 0.55$ as the default of the Service Cutter.

6.4.3 Discussion

Two main differences between Leung and Girvan-Newman algorithms are determinism and the number of clusters parameter. This section discusses the advantages and disadvantages of each.

Determinism

Results of a deterministic algorithm like Girvan-Newman can be reproduced by running the algorithm again with the same input data. The influence of different input data, scoring values and priorities can therefore be analyzed as the algorithm itself does not contain a random element.

A non-deterministic algorithm like Leung complicates analysis since changes in results cannot clearly be identified as consequence of changes in input or randomness. Furthermore results always need to be safely persisted and reloaded since they cannot be reliably reproduced.

Nevertheless, our industry partner accurately pointed out that an element of randomness is not only a disadvantage. Running multiple algorithm cycles presents different solutions and outlines where the difficult architectural decisions reside.

Number of Clusters Parameter

Providing the number of clusters as a parameter to the algorithm has the advantage of analyzing the service decomposition with any possible number of services. This feature can be used to better understand the structure and coupling between parts of the analyzed system by running an analysis with a number of services that is not optimal. Requesting a high number of service provides indications of how services internally might be further decomposed into modules.

Furthermore, the parameter allows to help in the process of emerging from a monolithic architecture to service orientation as described in Section 7.1.

Nevertheless, algorithms requesting the number of services as a parameter hand over the responsibility to answer this critical question to the user instead of answering it itself. Our industry partner pointed out that architects are often prejudiced on the number of services their system should be composed of. Having the Service Cutter providing not only the content of each service but also the number of services challenges the user to reassess his ideas against the candidate service cuts provided.

After analyzing and evaluating different algorithms, the next section documents the scoring process used to define the weights between on edges between nodes.

After documenting the algorithm evaluation, the next section describes the scoring process used to define the weights of the edges in the graph.

6.5 Scoring

The scoring process rates the relation of two nanoentities with a score. A higher positive score states that these entities should be modeled in the same service while a negative score asks for a separation of the nanoentities into different services.

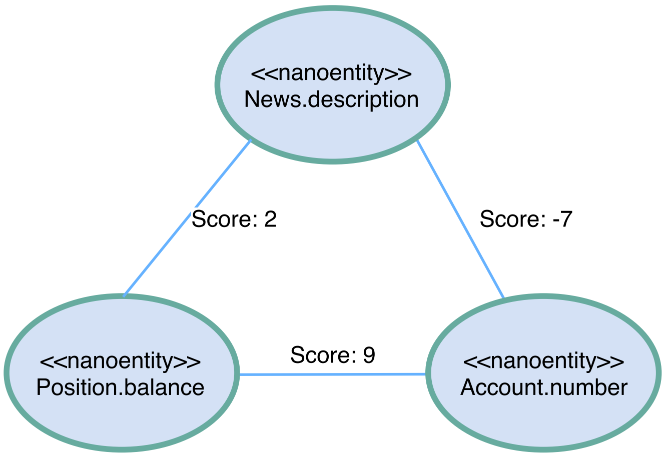


Figure 6.8: Simple Scoring Example

Figure 6.8 shows three nanoentities from the Trading System (see Appendix B.1) and their potential scores to each other. The relation *Position.balance* to *Account.number* has a high score as these nanoentities are often accessed in the same use cases and therefore get a high score from the *Semantic Proximity* criterion. *News.description* to *Account.number* has a negative score. These nanoentities do not have common use cases and do not belong to the same entity, so the positive scores from *cohesiveness* criteria are low or zero. Different characteristics like *low* availability requirements for *News.description* and *critical* availability requirements for *Account.number* lead to a negative score.

In this simple example, the Service Cutter would most probably suggest two services, one with *Account.number* and *Position.balance* and another one with the *News.description*.

6.5.1 Single Dimensionality

A big challenge with the graph based approach is that weights between nodes only have one dimension with a single unit of measurement. The edges can describe how close or cohesive two nanoentities are. This suits well to all coupling criteria of type cohesiveness as they describe reasons to combine nanoentities in the same service. Coupling criteria describing separation or incompatibility, where separation of nanoentities is asked by constraints or different nanoentity characteristics, cannot accurately be described using the weight on the edge.

Our approach to solve this problem is to introduce negative scores as shown in the example above. A negative score implies in the view of a coupling criterion that two fields should be separated into different services. If the negative score exceeds the positive score given by cohesiveness criteria, the edge between two nanoentity nodes is removed.

This approach implies two disadvantages:

1. By removing edges with a score lower than zero, information about the need for separation is lost. The scores of -120 and -1 are processed equally.
2. Positive and negative scoring criteria must be balanced out. Too many negatively scoring criteria or high prioritization of such criteria leads to elimination of the positive data and therefore of all information available.

Nevertheless, the practical assessment documented in Appendix B proves effectiveness of this approach. Firstly, nanoentities without an edge are unlikely to be placed into the same service by the algorithm. Secondly, systems are described using an ERM and use cases so that cohesiveness criteria are already provided with information. Negative scoring criteria only produce scores if nanoentities are described with characteristics not equal to the default characteristics. We furthermore defined the default priorities for each criterion, as described later in this section, in a way that positive scoring criteria are prioritized higher. Section 9.4.1 outlines the need for a better concept the handle separations.

6.5.2 Scoring Process

To determine the final score between two nanoentities, three steps are required. First every coupling criterion scores the relations it has information about. After that, priorities for each criterion are applied. Finally, the prioritized scores of all criteria are summed up to a final score.

Step 1: Score by Coupling Criterion

Every coupling criterion calculates a score between -10 and 10 for each nanoentity relation AB.

$$criteria[i].score_{AB} = -10...10$$

A score of 10 implies that "From the view of coupling criterion i, the nanoentities A and B should definitely reside in the same service." A score of -10 implies that "From the view of coupling criterion i, the nanoentities A and B should definitely **not** reside in the same service."

The range from -10 to 10 was chosen to be able to compare scores of different coupling criterion with each other. A fixed range guarantees that not one criterion receives more importance than another one due to its scoring logic. Other ranges like -1.00 to 1.00 would have worked as well, but in our opinion integer numbers are easier to understand for users than rational numbers.

It is important to notice that not every criterion has to use the full possible range. Coupling criteria of type *cohesiveness* tend to score only positive values, as they describe why nanoentities should be composed in the same service. Criteria of type *compatibility* or *constraints* on the other hand tend to score negative, as they describe reasons to split nanoentities into different services.

Step 2: Prioritized Coupling Criteria

In the second step, the coupling criteria are prioritized relatively to each other. Table 6.2 outlines all possible priorities.

Table 6.2: Coupling Criterion Priorities

Representation	Value
IGNORE	0
XS	0.5
S	1
M	3
L	5
XL	8
XXL	13

The priorities are represented by t-shirt sizes each defining a priority value. By recommendation of our industry partner and inspired by agile estimation scales[1], we chose a progressive and nearly exponential value sequence.

Having very high values, allows analyzing a system in the Service Cutter with focus on one or two coupling criteria only, as giving a criterion the priority XXL quickly makes it more important than many of the other criteria together. With progressive values we obtain this ability while still keeping the number of priorities small in order to make it simpler to map each criterion to a priority.

The priority of each criterion is multiplied with all scores given by that criterion.

$$criteria[i].prioritizedScore_{AB} = criteria[i].score_{AB} * criteria[i].priorityValue$$

It is important to distinguish between a criterion score (step 1) and a prioritized score (step 2). A criterion score is a statement on the relation of two nanoentities per coupling criterion where each coupling criterion has the same importance (-10 to 10). The prioritized score multiplies this statement by the importance of the criterion in the analyzed system. Priorities are very context sensitive. A real-time entertainment system for example will have for example very different priorities than a system handling financial transactions.

[1]Alex Yakyma wrote an insightful paper on why progressive estimations are efficient for teams[38].

Step 3: Final Relation Score

The final relation score between two nanoentities is the sum of all prioritized scores per coupling criterion:

$$finalScore_{AB} = \sum_{i=1}^{n} criteria[i].score_{AB} * criteria[i].priorityValue_{AB}$$

The undirected, weighted graph is then built with each node representing a nanoentity. Nanoentity relations with a positive score are connected by a weighted edge:

$$edgeWeight_{AB} = max(0, finalScore_{AB})$$

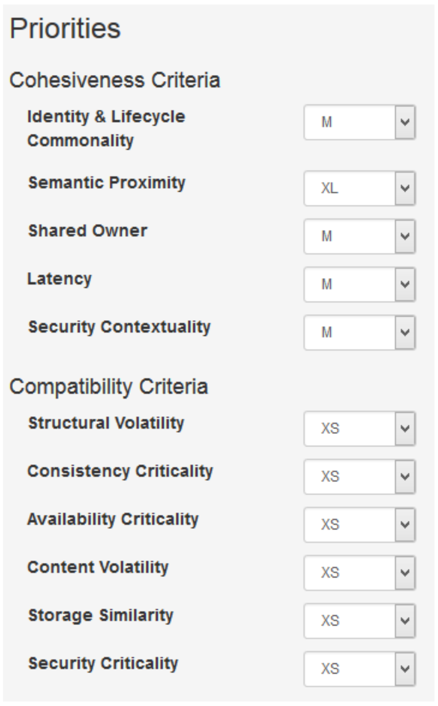
6.5.3 Default Priorities

By reasons of the single dimensionality problem described in Section 6.5.1 we defined the default priorities for coupling criteria in a way that negatively scoring criteria are prioritized lower than positively scoring criteria. This does not apply to constraints as these commonly play an important role in the decomposition process.

We defined the following defaults for the coupling criteria types.

- Cohesiveness: M
- Compatibility: XS
- Constraints: M

Figure 6.9 demonstrates how a user can change the priorities in the Service Cutter.


Priorities

Cohesiveness Criteria

Identity & Lifecycle Commonality	M
Semantic Proximity	XL
Shared Owner	M
Latency	M
Security Contextuality	M

Compatibility Criteria

Structural Volatility	XS
Consistency Criticality	XS
Availability Criticality	XS
Content Volatility	XS
Storage Similarity	XS
Security Criticality	XS

Figure 6.9: Screenshot of default priorities in the Service Cutter.

6.5.4 Scoring Logic

This section documents how each coupling criterion calculates its criterion score from -10 to 10.

Figure 6.10 shows the CriterionScorer interface used to calculate criterion scores.

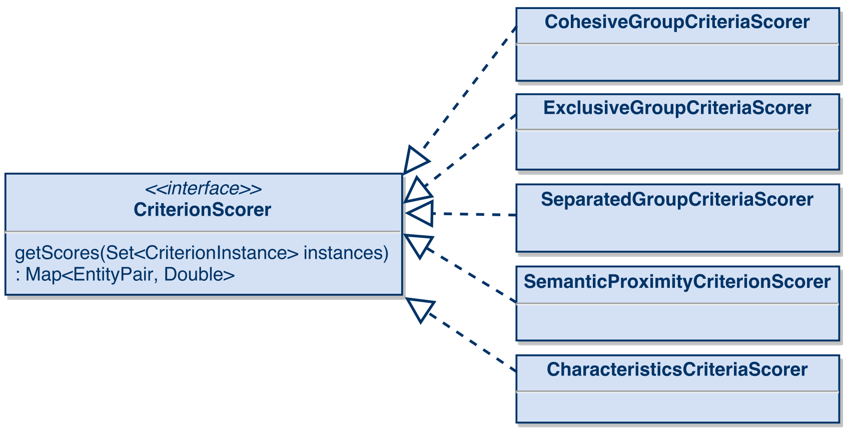

<<interface>>
CriterionScorer
getScores(Set<CriterionInstance> instances) : Map<EntityPair, Double>
CohesiveGroupCriteriaScorer
ExclusiveGroupCriteriaScorer
SeparatedGroupCriteriaScorer
SemanticProximityCriterionScorer
CharacteristicsCriteriaScorer

Figure 6.10: CriterionScorer interface and its implementations.

Table 6.3 outlines which implementation is used for which coupling criterion.

Table 6.3: Coupling Criteria and their scorer implementation.

Coupling Criteria	Scorer Implementation
Shared Owner Consistency Constraint Identity & Lifecycle Commonality Latency Security Contextuality	CohesiveGroupCriteriaScorer
Predefined Service Constraint	ExclusiveGroupCriteriaScorer
Security Constraint	SeparatedGroupCriteriaScorer
Semantic Proximity	SemanticProximityCriterionScorer
Content Volatility Consistency Criticality Availability Criticality Structural Volatility Storage Similarity Security Criticality	CharacteristicsCriteriaScorer
Mutability Network Traffic Suitability	No Implementation

CohesiveGroupCriteriaScorer

The criteria using the CohesiveGroupCriteriaScorer define one or more groups of nanoentities that for some reason should be kept together in one service.

The scorer sets the maximum score of 10 for every relation between nanoentities inside a group.

ExclusiveGroupCriteriaScorer

A ExclusiveGroupCriteriaScorer is used for the *Predefined Service Constraint* criterion. The scorer implements the same logic as a CohesiveGroupCriteriaScorer, but additionally sets a penalty of -10 from a nanoentity inside a group to all other nanoentities which do not share the same group.

SeparatedGroupCriteriaScorer

A SeparationCriteriaScorer is used for the criterion *Security Constraint*. The criterion defines two or more groups of nanoentities which should clearly be separated into different services for security reasons.

The implementation sets a negative score of -10 from each nanoentity to all other nanoentities defined in other groups than the own one.

SemanticProximityCriterionScorer

The SemanticProximityCriterionScorer is a more complex scoring implementation. It considers use cases and aggregations in a ERM. To calculate the score, the scorer uses an intermediate score summing up occurrences of the following conditions for the nanoentities A and B:

READ_ACCESS A and B are both read in the same use case.

WRITE_ACCESS A and B are both written in the same use case.

MIXED_ACCESS One of A or B is read and the other one written in the same use case.

AGGREGATED_ENTITY A belongs to an entity which has an aggregation to B's entity in UML diagram.

For each occurrence a number is added to the intermediate score of the AB relation. Currently the following numbers are implemented:

SCORE_WRITE_ACCESS: 10
SCORE_READ_ACCESS: 3
SCORE_MIXED_ACCESS: 3
SCORE_AGGREGATION: 1

More important than the actual numbers are the relative differences to each other which define the importance of the occurrences. Literature for *Semantic Proximity* listed in Chapter 4 suggests that one of the important coupling criteria is, which nanoentities are changed by the same use cases. We therefore set the default values above in a way that emphasizes common write occurrences the most.

To reduce the intermediate score to the criterion score, the following rules are applied:

- The top 10% of all relations with the highest intermediate score receive a criterion score of 10.
- The lowest intermediate score of the top 10% defines the reference intermediate score. The other 90% are calculated relatively to that reference and receive a criterion score between 0 and 10.

The calculation for the lower 90% works as following:

$$\frac{referenceIntermediateScore}{x} = 10 \tag{6.1}$$

$$x = \frac{referenceIntermediateScore}{10} \tag{6.2}$$

$$criterionScore_{AB} = \frac{intermediateScore_{AB}}{X} \tag{6.3}$$

Both the numbers counted for access or aggregation occurrences and the reduction to the criterion score have been experimentally evaluated. The algorithm documented here has proven to produce reasonable results for the example systems but might require further investigations for other systems.

CharacteristicsCriteriaScorer

Each coupling criterion of type compatibility defines multiple characteristics. The idea of these criteria is to create services with as homogeneous nanoentities as possible. The scorer therefore sets negative scores for a relation between two nanoentities having different characteristics.

As an example, the criterion *Structural Volatility* describes how often change requests need to be implemented that affect a nanoentity. A nanoentity can have the characteristic *often, normal* or *rarely.* Each characteristic has a number between 0 and 10 assigned:

Often: 10
Normal: 4 *(default)*
Rarely: 0

Two nanoentities with different characteristics receive a negative score with the difference between the numbers. As an example, a nanoentity A with characteristic *often* and a nanoentity B with characteristic *normal* receive a criterion score of -6.

It is important that the highest and lowest number have a difference of 10 to fully use the range a criterion has to rate a relation.
In this case there is an intermediate value of 4 for the *normal* characteristic. In our experience the often changing nanoentities are the most interesting and have the highest impact on service decomposition. With using the number 4 instead of 5 for *normal* we put an emphasis on *often* as the differences to these characteristics become bigger.

To improve usability, we introduced default characteristics so that a user does not need to define characteristics of all criteria for all nanoentities.

Table 6.4 outlines characteristics of all criteria with their values and defaults.

Table 6.4: Criteria characteristics with values and defaults.

Criterion	**Characteristics**
Structural Volatility	Often 10 Normal 4 *(default)* Rarely 0
Consistency Criticality	High 10 *(default)* Eventually 4 Weak 0
Availability Criticality	Critical 10 Normal 4 *(default)* Low 0
Content Volatility	Often 10 Regularly 5 *(default)* Rarely 0
Storage Similarity	Huge 10 Normal 3 *(default)* Tiny 0
Security Criticality	Critical 10 Internal 3 *(default)* Public 0

After covering the algorithm evaluation and scoring process, the next section documents the implementation and design of the prototype.

6.6 Prototype

This section introduces the prototypical implementation of the Service Cutter.

6.6.1 Design

The Service Cutter is divided into two processes:

1. The *Editor* is a web application and handles all user interactions.
2. The *Engine* runs in a separate Java Virtual Machine (VM) and offers the capability to upload nanoentities and coupling information. This data is then used to calculate candidate service cuts.

We decided to split the Service Cutter into a web application and a separate Engine so that the Engine can be reused in a different context without the overhead of a fully fledged UI.

Figure 6.11 introduces the important components and packages of the Service Cutter.

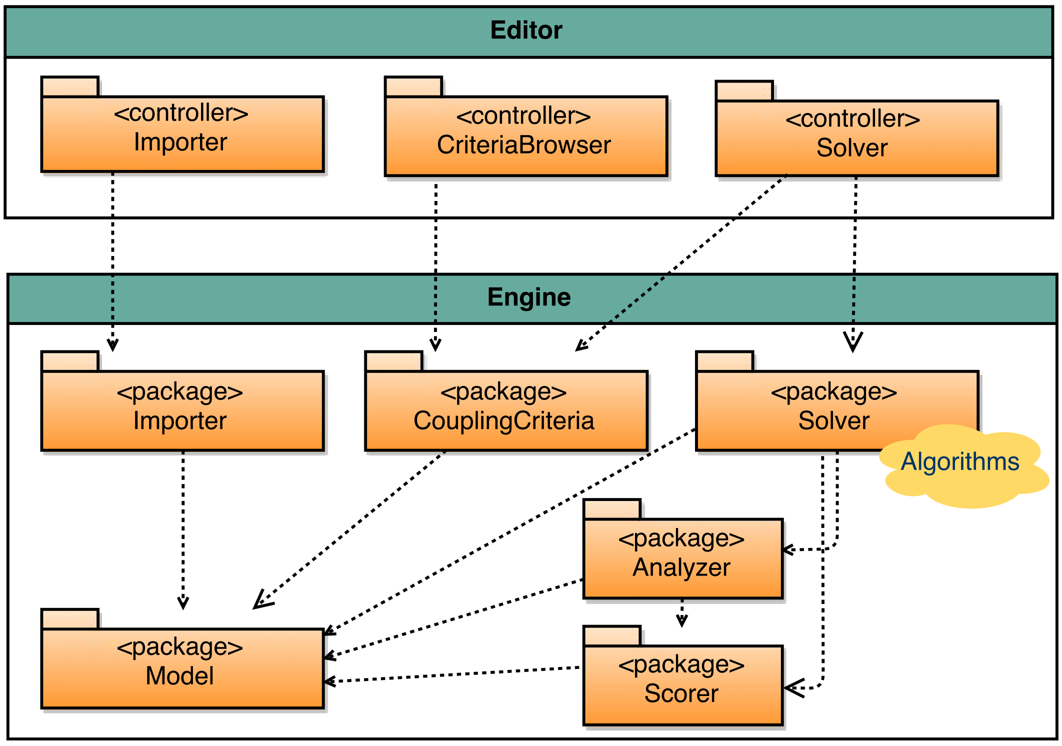


Figure 6.11: Components and packages of the Service Cutter.

The *Editor* consists of three components:

Importer is used to upload nanoentities and user representations into the Engine.

CriteriaBrowser is used to display all implemented coupling criteria and their description.

Solver is used to parametrize the decomposition calculations and to visualize candidate service cuts.

The Engine offers a RESTful HTTP web service for all Editor components. It contains the following packages:

Importer contains the importer endpoint and maps user representations to the Engine internal model.

CouplingCriteria contains an endpoint for the coupling criteria data to be displayed in the CriteriaBrowser.

Solver runs the scorers to create the graph, starts the algorithm, processes the result with the analyzer, and exposes the candidate service cuts on a web service.

Analyzer analyzes the candidate service cuts to determine use case responsibilities and published language between the candidate services.

Scorer calculates the scores between nanoentities based on the user's system specification and coupling criteria.

Model contains the coupling criteria catalog, all nanoentities and coupling data. The sub package repository encapsulates all interactions with the database.

The next section outlines the model that is used to persist user systems and the coupling catalog. This data is owned by the Engine but imported and utilized by the Editor component.

6.6.2 Model

Figure 6.12 describes the model used to persist a system with its nanoentities and the coupling catalog. All classes are persisted to a relational database using Java Persistence API (JPA) annotations and Hibernate[59].

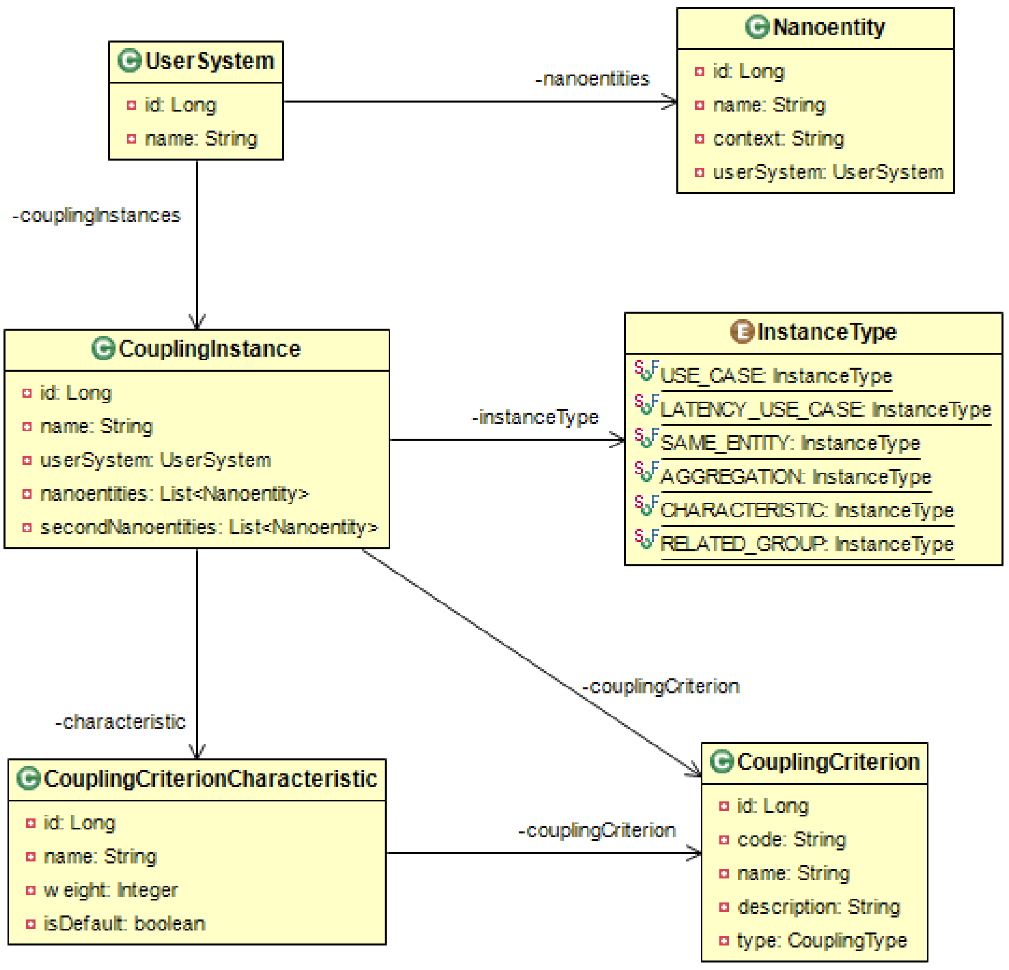


Figure 6.12: Model of a user system and the coupling catalog.

- For each imported system, a *UserSystem* and a set of *Nanoentity* and *CouplingInstance* objects are created.
- The imported user representations are transformed into objects of the class *CouplingInstance*. The assigned *InstanceType* is used to keep track of the originating user representation.
- A coupling instance is linked to a *CouplingCriterionCharacteristic* for a coupling criterion of type compatibility. Otherwise this relation is null.
- *CouplingCriterionCharacteristic* and *CouplingCriterion* only change in case of amendments to the coupling criteria catalog.

6.6.3 Technology

We followed the recommendation of our industry partner and used Spring Boot[68] and JHipster[60] as the underlying frameworks.

The main reasons were:

- Our industry partner uses them successfully.
- Spring Boot is considered state-of-the-art for Spring based applications nowadays.
- JHipster is based on established technologies that are partially already familiar to us. Furthermore, it provides a code generator and samples. This can noticeable speed up development especially in the prototyping area.
- We were able to implement a technological proof of concept in a few days and did not face major obstacles.

The Service Cutter is implemented as a three tier application.

The first tier is the web *browser* of the user. The application is based on AngularJS[43] and uses a template based on Bootstrap[47]. A RESTful HTTP interface provides access to the Editor tier. Graph visualizations are implemented using the vis.js library[76].

The *Editor* component is a web application based on the JHipster[60] framework. It provides the user the possibility to import system information using a JSON upload and feeds this information into the Engine. All security and user interface aspects are handled in this layer.

The *Engine* is the isolated component that holds the logic to calculate candidate service cuts. It is based on Spring technologies and provides RESTful HTTP interfaces. The Engine does not provide any security measures and it is therefore necessary to restrict access to the Engine. We recommend to achieve this using a Docker[51] internal network as described in Section 6.6.5.

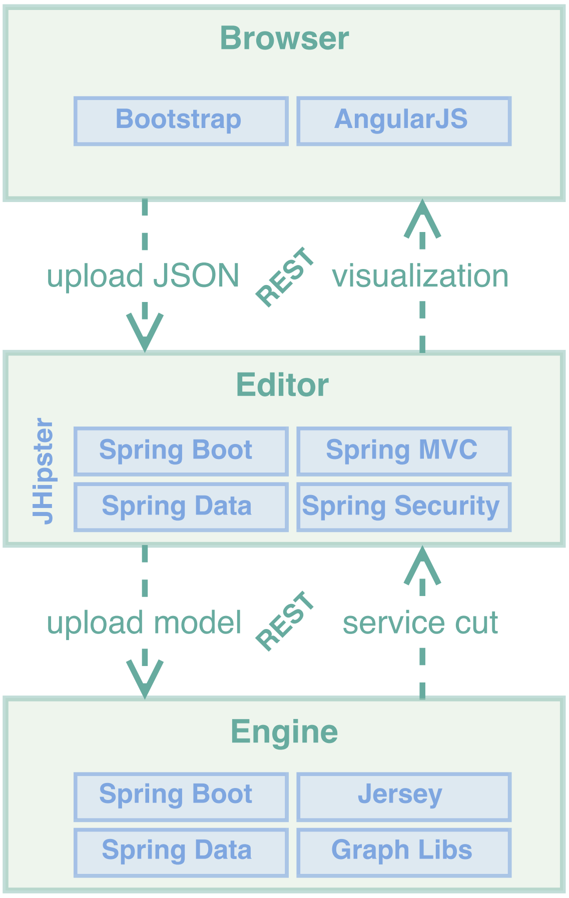


Figure 6.13: Technology Overview

6.6.4 RESTful HTTP Interfaces

This section documents all RESTful HTTP interfaces provided by the Engine. The Editor is built on these interfaces, but they could also be used by another user interface.

The required JSON formats are specified by JSON Schemas[61]. The schemas can be found in the *JSON_Schemas* folder. An example for a JSON Schema is listed in D.2, defining the export format of candidate service cuts.

Table 6.5 outlines all provided RESTful HTTP interfaces and the required JSON specifications:

Table 6.5: RESTful HTTP Interfaces

Title	**Path**	**Method**	**Request Schema**	**Response Schema**
importERM	/engine/import/	POST	*1_erm*	*4_importResult*
import Nanoentities	/engine/import/{systemId}/ nanoentities/	POST	*3_nanoentities*	*5_userSystem*
importUser Representations	/engine/import/{systemId}/ userrepresentations/	POST	*2_userReps*	*4_importResult*
getSystems	/engine/systems/	GET	-	*5_userSystem*
getSystem	/engine/systems/{systemId}/	GET	-	*5_userSystem*
createSystem	/engine/systems/	POST	`"name"`	*5_userSystem*
solveSystem	/engine/solver/{systemId}	POST	*6_solverConfig*	*7_solverResult*

These JSON files can either be uploaded through the Editor or directly, for example by using the Linux bash:

```
curl -i -H "Content-Type: application/json" -X POST
http://localhost:8090/engine/import/{systemId}/userrepresentations
-d booking_2_user_representations.json
```

While we leverage the HTTP protocol and its verbs, the interfaces are modeled as operations rather than resources. To comply with level 1 and 2 in the Richardson Maturity Model[15] for RESTful HTTP, the interfaces could be refactored to represent resources rather than operations.

6.6.5 Deployment

Technically the Service Cutter can be operated on a single server providing a relational database and a Tomcat web server. However, we developed a more sophisticated infrastructure aligned with modern standards which is presented as follows.

- Every components runs in an isolated Docker[51] container.
- Docker Compose[52] is used to connect the Docker containers.
- The Engine and the Editor operate in an embedded Tomcat[45] server as provided by Sprint Boot web.
- The Engine and the Editor both require a relational database. As Hibernate[59] and Liquibase[63] are used, all database related code is vendor agnostic. However we only tested the Service Cutter on Postgres 9.4[66].

Docker provides a layer of abstraction between the application and the underlying operating system. It uses resource isolation features of the Linux kernel to avoid the overhead of starting and maintaining full virtual machines. We implemented our deployment using Docker as it provides us with the ability to install the application components in isolated containers on a single machine with only a few command line statements.

The Docker Compose definition is attached in Appendix D.1.

6.6.6 Information Security

The web application is secured using an authentication and authorization implementation based on Spring Security[71]. Any other internal components such as the database or web services are hidden behind the server's firewall and therefore do not need any special security measures.

The uploaded data models are shared amongst all registered users.

After covering the important design and implementation aspects, the next chapter discusses the provided results.

7. Discussion

This chapter discusses the outcome of the thesis. It starts with usage scenarios and benefits of the Service Cutter and concludes with a requirements assessment.

7.1 Usage Scenarios

We identified two usage scenarios suitable for the Service Cutter.

Monolith First

The most likely scenario with existing software is the transition from a monolith to a service oriented architecture. Martin Fowler recommends to use this approach for every project and to not start a project with services[14]. The Service Cutter is able to identify candidate service cuts with a given number of services. With the algorithm set to Girvan-Newman and the number of services to 2, the Service Cutter suggests a first service to be extracted from the monolith. by iteratively increasing the number of services the user can plan his process towards a service oriented architecture.

Greenfield Scenario

The other scenario is the greenfield scenario when the system is not yet developed but partially specified and designed. The requested user representations can be used as check-list during requirements engineering and design processes. Specification artifacts are then loaded into the Service Cutter and candidate service cuts can be the basis for a discussion between architects. Once agreed on service cuts, the assigned use cases and published language help to implement the services and their interfaces of each other.

7.2 Benefits

The Service Cutter offers the following benefits:

- By requesting different user representations, an architect is challenged to analyze which user representations and characteristics are relevant in his system. He might use the user representations as a checklist for requirement engineering.
- The user representations and coupling criteria can further be used to educate junior architects or students on the driving forces of service decomposition.
- The Service Cutter provides candidate service cuts based on the defined user representations. With these candidate service cuts the architect's expectations of the number of services and their definition is either verified or challenged.
- The greenfield scenario as well as an iterative approach from moving from a monolith to service orientation are supported by the Service Cutter.
- Use cases are assigned to their responsible service. The published language between services is displayed in order to assist the development of services and their interfaces to each other.
- By storing the candidate service cuts, architectural decisions can be persisted and documented (not yet implemented).

7.3 Requirements Assessment

This section assesses the developed solution based on the defined requirements. The two sample system as described in Appendix B as well as the implemented Service Cutter itself serve as the test scenario.

All requirements are rated with a rating from $1-3$.

1 The requirement is fully satisfied.

2 The requirement is partially satisfied.

3 The requirement is not satisfied.

7.3.1 Functional Requirements

Table 7.1 assesses the provided solution against the defined functional requirements described in Section 5.2.

Table 7.1: Assessment of functional requirements.

Requirement	**Rating**	**Assessment**
Coupling Criteria	1	All coupling criteria have been implemented in the Service Cutter.
User Representations	1	All required user representations are supported by the importer of the Service Cutter.
Priorities	1	Priorities are built into the scoring process.
Candidate Service Cuts	1	The Service Cutter visualizes a candidate service cut using a chart. The candidate services can be exported in a JSON format documented in Appendix D.2
Published Language	1	The published language is visualized when selecting a service in the visualization.
Hard Architectural Decisions	2	This feature is not explicitly implemented but partially given by non-deterministic algorithms as discussed in Section 6.4.3

7.3.2 Non-Functional Requirements

Table 7.2 assesses the provided solution against the defined non-functional requirements described in Section 5.3.

Table 7.2: Assessment of non-functional requirements.

Requirement	**Rating**	**Assessment**
Usability	2	We reviewed the user interface within the project team but did not conduct usability tests.
Simplicity	1	A simple analysis can be performed with 5 mouse clicks.
Performance	2	The sample systems can be decomposed in not more than two seconds. Extensive performance tests have not been conducted due to time budget constraints as decided with our stakeholders. Section 9.2.9 lists this as a future activity.
Logging, Deployment	1	Logging is based on SLF4J and a deployment based on Docker is implemented.
Fault Tolerance	1	All errors are handled and occurring errors are logged. The Service Cutter has remained robust even if exceptions or errors occurred. The user receives detailed validation feedback if a problem occurs while uploading JSON files.
Maintainability	2	The implementation is based on two services (Editor and Engine) both built with suitable application layers. Communication between the layers is implemented with RESTful HTTP communication. Open source tools are used wherever possible. We rate this requirement as only partially satisfied as the RESTful HTTP interfaces do not meet the Richardson maturity level requirements.
State-of-the-Art Technology	1	The application is based on the technology stack suggested by our industry partner.
License	2	The source code has been released under the Apache 2.0 license. The Girvan-Newman algorithm implementation however is published under the GPL license. This dependency should be replaced when the Service Cutter is used in a commercial environment. Section 9.2.10 lists this task.

After discussing the Service Cutter's usage scenarios, benefits, and assessing the defined requirement, the next chapter documents our conclusion.

8. Conclusion

This chapter evaluates the outcome of the thesis with its hypothesis and closes with a summary and outlook.

8.1 Hypothesis Evaluation

At the beginning of this project we formulated two hypothesizes. We were able to produce results validating both.

> *The driving forces for service decomposition of a software system can be assembled in a comprehensive criteria catalog.*

We successfully compiled a catalog of 16 coupling criteria that aims to form a comprehensive but not conclusive collection.

The catalog helps a software architect to structure driving forces for service decomposition. The developed criteria may provide a basis for a common language amongst architects.

> *Based on the criteria catalog, a system's specification artifacts can be processed in a software to optimize loose coupling between services and high cohesion within services in a structured and automated way.*

In the Service Cutter assessment documented in Appendix B, we tested two sample applications with the algorithms Girvan-Newman and Leung. Girvan-Newman provided expected and therefore satisfying results in only one of the two example systems. Leung on the other hand did not only provide expected service cuts for both systems but surprised us with suggestions that were unexpected but definitely reasonable.

8.2 Summary and Outlook

The hypothesis could be validated by producing the coupling criteria catalog and the Service Cutter prototype. Most of the requirements were successfully implemented. The goals of the initial project definition document in Appendix E could be reached.

The Service Cutter as of now is not a production grade architectural tool but a proof that our concept of structured and automated decomposition optimization generally works and is worth further investigations.

Without more sophisticated means to define or import a systems specification the effort to specify a system is likely to high for an average user. We trust that the great value to its users will become apparent when further efforts are put in the following aspects:

1. The **Service Cutter Prototype** should be enhanced to a production ready tool with graphical user interfaces for defining, editing, and storing a user's system specification and candidate service cuts.
2. The Service Cutter should be integrated into a **toolkit chain**. The input could automatically be generated from other tools or diagrams and the output used for code or API generation.
3. **Graph clustering algorithms** should further be analyzed and optimized. Possible alternative approaches as documented in Appendix A could furthermore solve some of the conceptual challenges of the scoring process.
4. The **scoring process** for the different type of criteria should further be analyzed and tested. More sophisticated solutions for conceptual challenges like the single dimensionality problem documented in Section 6.5.1 could improve the accuracy and meaningfulness of results.

The next chapter describes the proposed improvements in more detail.

9. Future Work

This chapter introduces possible enhancements of the Service Cutter.

9.1 Algorithms and Approach

This section illustrates a series of possible improvements to the algorithms.

9.1.1 Additional Leung Layer

The non-deterministic nature of the Leung algorithm causes the calculation result to be unstable. This has advantages and disadvantages as outlined in Section 6.4.3. As an architect, I would expect the Service Cutter to assist me with non-deterministic algorithms in a way that the possible service cuts are automatically calculated, compared and rated. This additional layer would allow me to select the best candidate service cut and help furthermore help to identify hard architectural decisions.

9.1.2 MCL adapter

As outlined in Section 6.4, the MCL algorithm could be used in the Service Cutter as well.

The reference implementation of the MCL algorithm is provided as a C based command line tool. With some effort this algorithm could be integrated into the Service Cutter using Java Native Interface (JNI) or an integration based on text files.

An assessment of the MCL algorithm may produce better results compared to the other algorithms.

9.1.3 Alternative Algorithms and Optimizations of Existing Algorithms

Our evaluation of suitable graph clustering algorithms was limited to the ones having a stable Java implementation. Further research may prove that other algorithms are capable of calculating candidate service cuts even better according to the defined criteria in Section 4.3.

Furthermore, existing algorithms may be improved with optimizations. For instance, Lancichinetti and Fortunato[21] valuated a set of improved versions of the Girvan-Newman algorithm as well as a collection of alternative algorithms.

We assume that optimizations of the Service Cutter's algorithm can be achieved when the selection is not limited to existing Java implementations.

9.2 Service Cutter Improvements

A set of enhancements of the Service Cutter as a tool may increase its value to the users.

9.2.1 Traceability of User Representations

It is important that the calculated data is presented to the user in a way for him to understand why the candidate service cuts have been selected. By visualizing the user's input on candidate service cuts, the user gets a better understanding of how his input and definitions affect the suggested decomposition.

Along with the user representations, the scores could be visualized per coupling criterion so that the user understands the impact of priority changes and input enhancements. However, this feature should be disabled by default a novice user.

9.2.2 Adjust Suggested Solutions

The architect is able to adjust the solution by moving a nanoentity from one to another service. He instantly sees the impact of his adjustment as the rating of each coupling criterion is visualized.

9.2.3 Configurations for Advanced Users

For a normal usage, the Service Cutter provides reasonable defaults which have been tested with example systems. Advanced users should be enabled to change or enhance existing characteristics of compatibility criteria.

9.2.4 Configuration through a Questionnaire

The Service Cutter could guide the architect through a questionnaire in order to apply different presets of characterization defaults or priorities.

9.2.5 Editor for User Representations

To improve usability and reduce the effort needed to define the input for the Service Cutter, a sophisticated user interface to create and edit user representations should be built. The user interface should simplify input creation as much as possible. One example to achieve this is to the ability to define characteristics on entities which are then applied to all nanoentities of the entity.

9.2.6 Iterative Enhancements

Projects are often implemented in subsequent stages. The selected service cut of the first iteration influences the design decision of the second iteration. The Service Cutter therefore should allow the user to load a previously calculated service cut into the new model.

One solution would be to persist candidate service cuts and the used priorities.

9.2.7 Candidate Cuts Comparison

To compare two different candidate service cuts, a feature should be added to visually illustrate differences between two solutions.

A history or back and forward buttons would be other features to help compare different solutions. Changes to parameters could be undone and earlier calculated results could be revisited.

9.2.8 Candidate Cuts Grading

We suggested a questionnaire to assess candidate service cuts in Section 4.3. This distinction could automatically be performed by the Service Cutter and visualized using a indicator similar to a traffic light.

9.2.9 Performance Tuning

The initially defined performance requirements were descoped to allow more functional enhancements. We assume that the Service Cutter should be able to support the required volumes as outlined in Appendix D.3. The scorer implementation as well as the algorithms need to be verified with a large enough data set to confirm the scalability of the Service Cutter.

9.2.10 Licenses of Libraries

The implementation of the Girvan-Newman algorithm[56] is licensed under the GPLv3. It has to be analyzed whether the Service Cutter can be used in a commercial context. Otherwise the implementation of the Girvan-Newman algorithm has to be replaced with an alternative implementation.

9.3 Toolchain Integration

Providing interfaces to existing tools lowers the cost of using the Service Cutter considerably. We therefore recommend to develop integrations with popular software development tools.

9.3.1 Adapters for the Input Format

Writing the required input is a significant effort. The model containing nanoentities and their relations could automatically be parsed from different sources. Adapters could be written for an Object-Relational Mapping (ORM) configuration, a database schema, or Unified Modeling Language (UML) diagrams provided by tools like the Enterprise Architect[54].

9.3.2 Use Solution as a Basis for Working Software

As the Service Cutter has sophisticated information about the usage of nanoentities in use cases, it is able to generate the APIs used to communicate between services.

Depending on the communication layer used to communicate between services, these APIs look differently. In a messaging based system, the Service Cutter could generate a set of messages or events needed to communicate between services. When services interact by RESTful HTTP interfaces, the Service Cutter could generate Swagger[73] API definitions for the resources which need to be part of the published language.

9.4 Scoring

The developed scoring process works well for our test scenarios. However other systems might require further enhancements.

9.4.1 Better Handling for Separations

As introduced as *single dimensionality* in Section 6.5.1, we mapped coupling of type compatibility and constraints to negative scores. Once all coupling criteria have been processed, we remove all edges with a negative total score. This approach retains information about the coupling in a system. Finding a solution for the single dimensionality problem would lead to more accurate candidate service cuts.

9.4.2 Implement Criteria of Type Communication

As part of this project we only implemented 14 out of 16 identified coupling criteria. The following two criteria are solely described as part of the decomposition model in Chapter 4.

- *Mutability* defines whether a nanoentity is mutable or immutable.
- *Network Traffic Suitability* illustrates the network traffic implications when this nanoentity is exposed to a remote interface.

Both criteria of type *communication* do not describe which nanoentities should or should not be modeled in the same service, but which nanoentities are more suitable for being exposed as published language and therefore used in intra service communication. For immutable data, consistency is not an issue and it is therefore simpler to handle and more suitable for published language then mutable nanoentities. Similarly, nanoentities that are frequently accessed and need high storage resources are less suitable for being exposed on the network.

If a nanoentity needs to be exposed is defined by use cases and which service is responsible for which use case. The communication criteria require that the use case accessing unsuitable nanoentities are owned by the same services as the nanoentities.

9.4.3 Calculate Content Volatility from Use Cases

The current implementation requires a characterization of nanoentities by their content volatility. If use cases would provide information of how frequently they're executed, the content volatility of nanoentities could be calculated out of use cases. This would reduce the user's effort to specify his system.

9.5 Conceptual Refinements

Besides the technical enhancements, we also collected a set of conceptual improvements.

9.5.1 Logical and Physical Services

As outlined in Section 3.1, services can be analyzed from different views like the logical or physical view. Udi Dahan describes the confusion of these views as one of the common pitfalls in implementing SOA[7].

In Section4.1 we categorized the coupling criteria into the views *Domain*, *Quality*, *Physical*, and *Security*. These views should be further analyzed and integrated in the Service Cutter to improve an architect's understanding of different views on his system and the definition of a service. For example, service decomposition could start with candidate bounded contexts by only considering domain criteria. These bounded contexts would further be split into physical services by taking the other views into consideration.

9.5.2 Caching

Caching is data redundancy that is used to reduce the cost of coupling between services. By the means of use cases, the Service Cutter has knowledge of which nanoentities need to be shared between services and, if the use case contains frequency information, even how often it needs to be shared. The Service Cutter could therefore suggest to the architect where caches would be appropriate.

9.5.3 Document Architectural Decisions

> *Architectural decisions capture key design issues and the rationale behind chosen solutions.*[39]

Documenting architectural decisions is a significant documentation artifact of every long-term software project. In order to retrace architectural decisions taken with help of the Service Cutter, important solutions enhanced with discussion notes could be saved persistently. These notes could be captures as free text, Y-Templates[40] or other architectural decision templates[1].

9.5.4 Relationships between Coupling Criteria

A relationship between coupling criteria like the following might have significant impact on service decomposition:

Persisted nanoentities with huge storage requirements should not be placed in the same service as nanoentities with high consistency requirements as they are handled using different database technologies.

Such a relationship should be incorporated in the Service Cutter's decomposition algorithm. Another way of integration would be to present a warning to the user whenever a critical combination of characteristics appears in a candidate service cut.

[1]O. Zimmermann *et al* compiled a comparison of seven publicly available decision templates[42, p. 3]

A. Decomposition Approach Evaluation

During early feasibility tests with graph clustering, we encountered significant problems as documented in the next section. As consequence of these results, we conducted a feasibility assessment with a professor of mathematics on the graph based approach. Out of the feasibility assessment, two additional approaches on how the Service Cutter can solve the decomposition problem were defined and are discussed in this chapter. Nevertheless, the conclusion section states how challenges in these approaches and further research on clustering graphs led us back to follow the graph based approach.

A.1 Graph Clustering Problems

At first, the clustering algorithm evaluated documented in Appendix C did not contain the Leung algorithm as this has been found later during the project. The two candidate algorithms were MCL and Girvan-Newman. We did a feasibility test using a small booking sample containing three entities:

Customer Entity containing address, accountNr, creditCardNr, and name.
Article Entity containing articleName, price, and serial.
Booking Entity containing totalPrice, paymentDate, bookingDate, and bookingState.

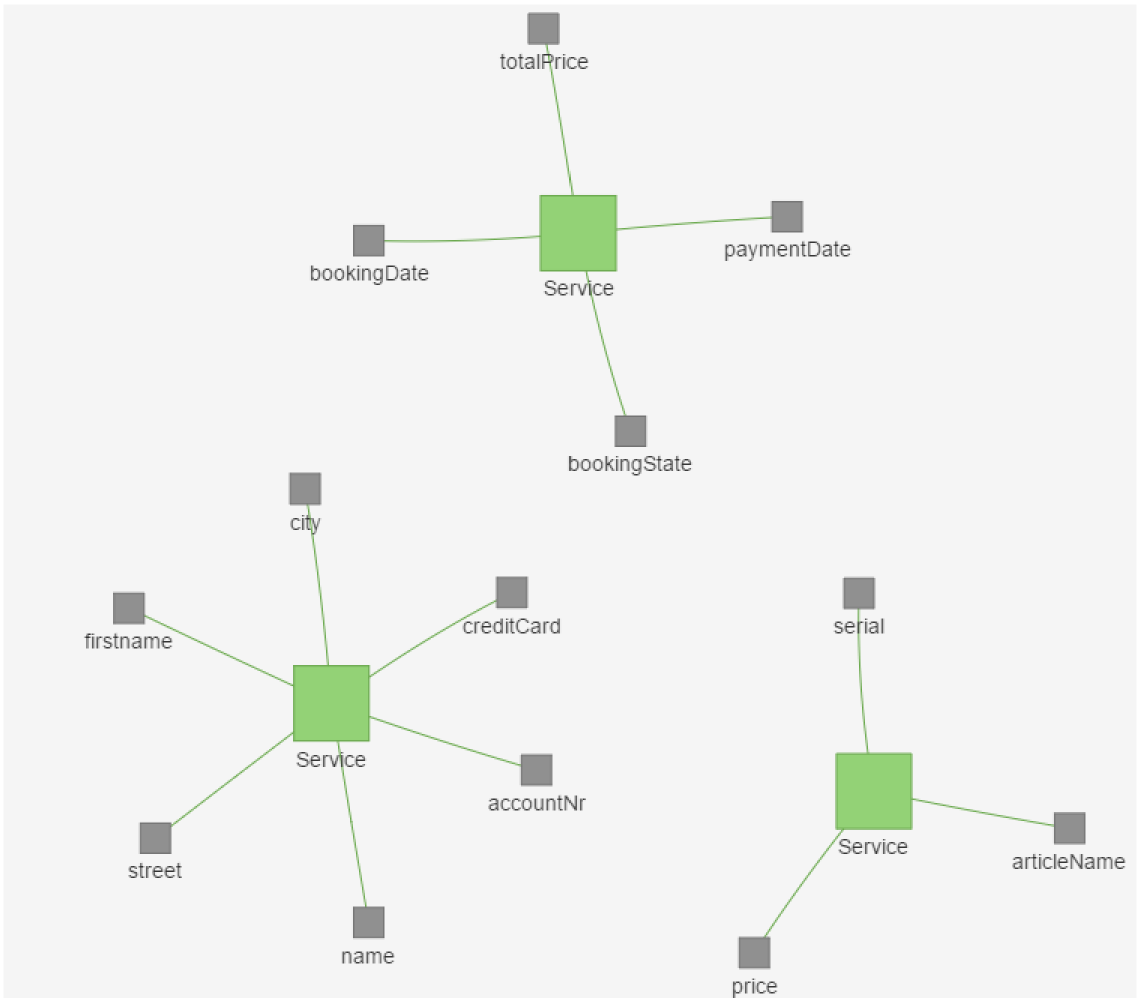


Figure A.1: Expected output for the booking example.

To keep the sample simple, we only added information for the *Lifecycle & Identity Commonality* criterion, so that the output is expected to show exactly the entity borders as shown in Figure A.1.

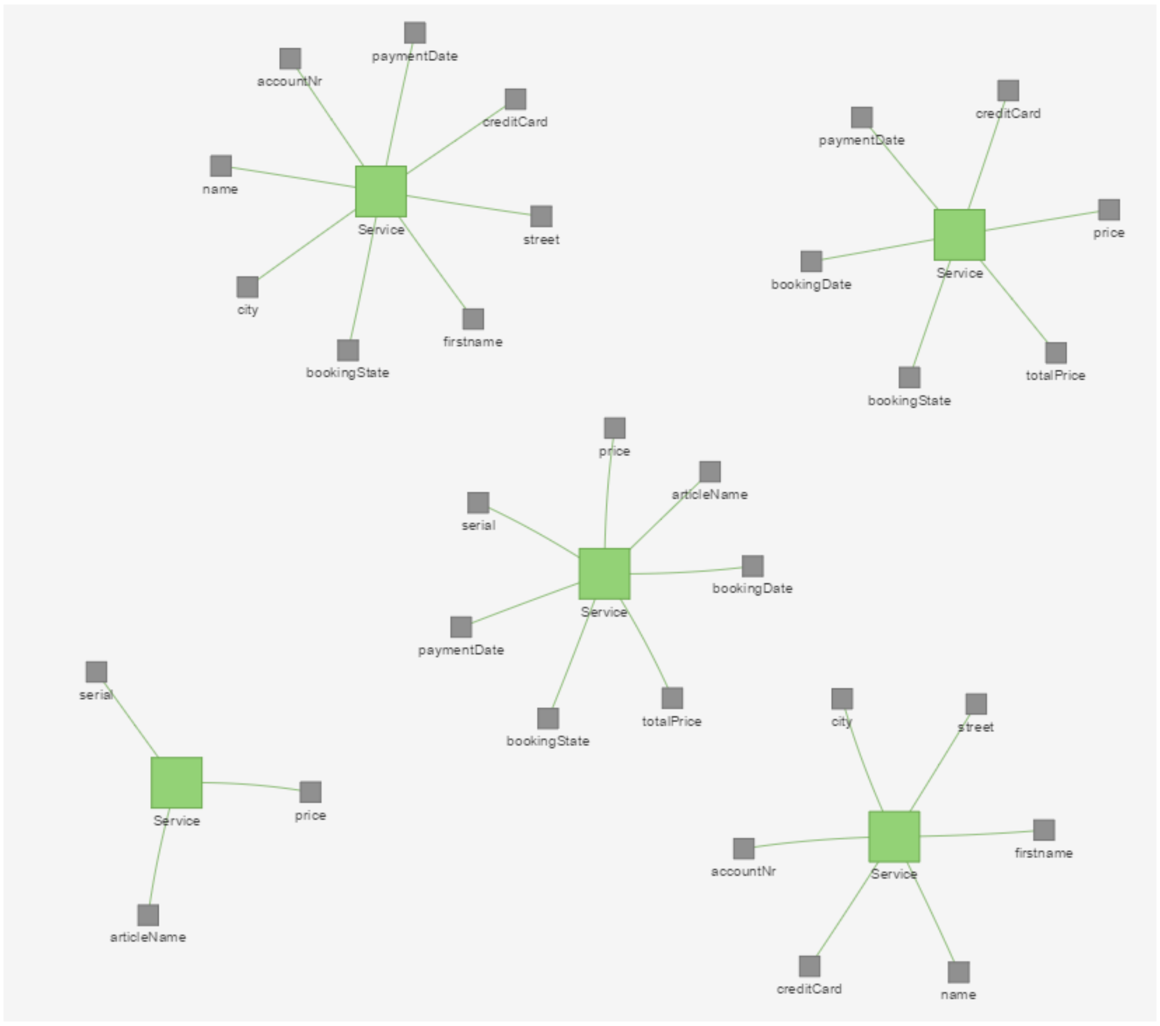


Figure A.2: Booking sample with the MCL algorithm.

The MCL algorithm's result for this example is shown in Figure A.2. This does not match the expectations as nanoentities are attached to multiple services. This violates the *distinct clusters* requirement. A nanoentity should be assigned to one and only one service.

The distinct clusters requirement is satisfied by the original MCL algorithm written in C. We therefore assume that this is an implementation problem of the Gephi plugin[57]. A solution to this problem would be to write a Java wrapper for the C implementation as described in Section 9.1.2.

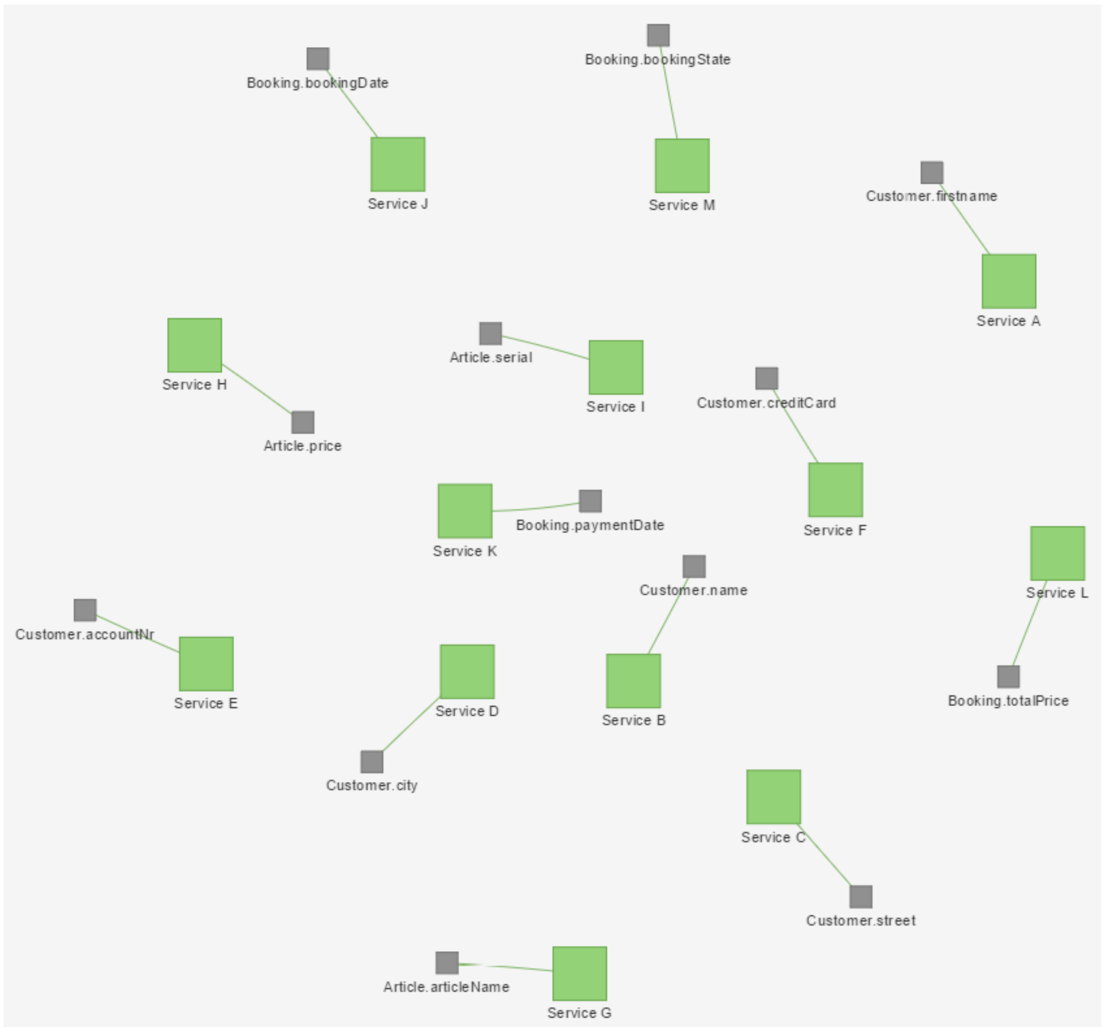


Figure A.3: Booking example with the Girvan-Newman algorithm.

Figure A.3 shows the unsatisfying result provided by Girvan-Newman. By reasons of these unexpected results, we consulted a professor of mathematics which led to the alternative approaches described in the next Sections.

A.2 Approach #2: Rating of Possible Service Cuts

The idea this approach introduces is to create a set of all possible service cuts and rate the cuts isolated per coupling criteria. The approach is illustrated in Figure A.4.

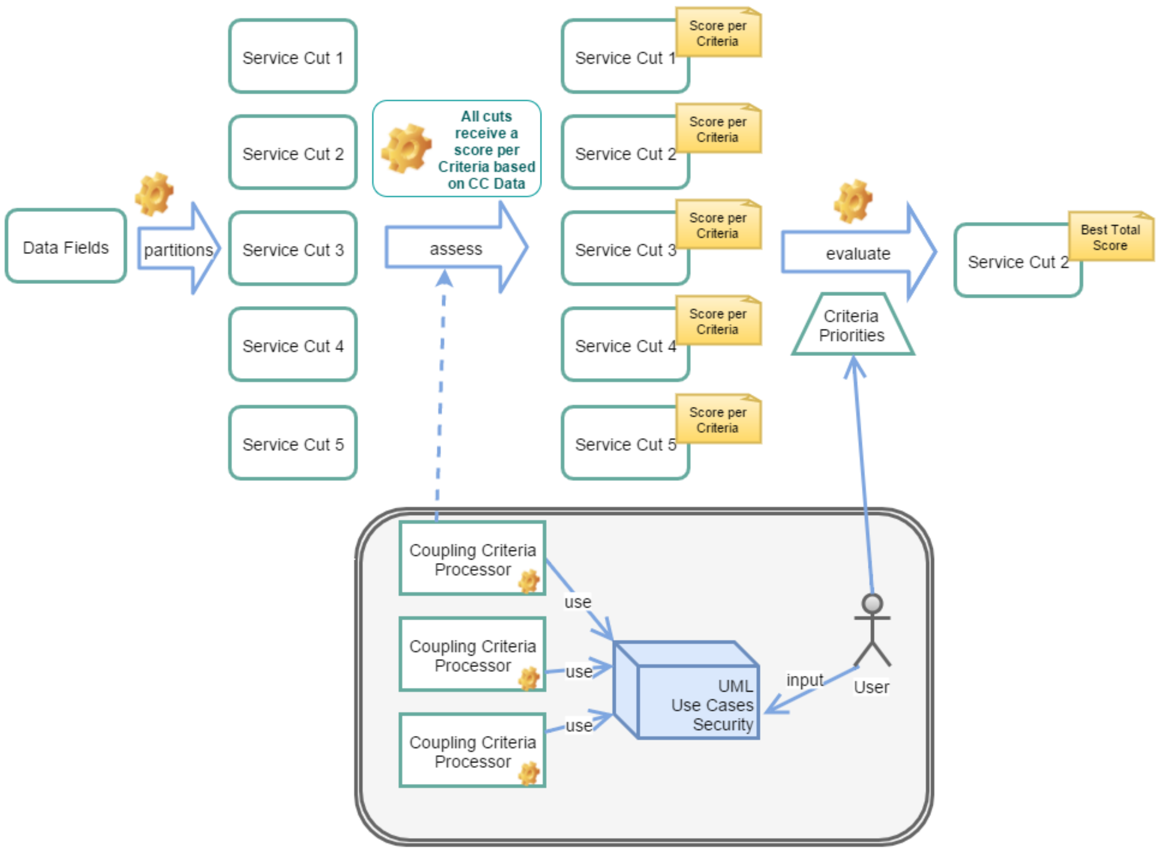


Figure A.4: Approach #2: Service Cut Rating

This approach is processed in three steps:

Partitioning Based on the nanoentities, a set of all possible candidate service cuts is calculated. This includes every theoretically possible service cut for any number of services. For practical usage, this step needs to be optimized.

Assessment For all coupling criteria a scorer assesses all service cuts with a score describing how well the criteria's requirements are met. The score is a number between 0 and 10, while 10 implies that all requirements are perfectly satisfied.

Evaluation The user optionally defines priorities how important each criteria is for his system. The priorities are defined with approximately exponential numbers like the Fibonacci sequence. These priorities are applied on the service cut scores. The resulting best candidate cut is then presented to the user.

A.2.1 Discussion

An advantage of this approach is that each relevant step is clearly separated and can thus be analyzed, debugged and visualized better than in the graph based approach. The assessment and score calculation is done separately for every cut and for every coupling criteria. Each criteria scorer scores candidate cuts with a uniform scoring range. As candidate service cuts do not need to be constructed but only rated, the single dimensionality problem described in Section 6.5.1 does not apply.

The weak spot is the partitioning process. Theoretically every possible set of services where each nanoentity is contained in one and only one service is a candidate cut. In mathematics this is described as the *partition of a set*[17] problem. The Bell number B_n defines the amount of possible partitions:

$$B_{n+1} = \sum_{j=0}^{n} \binom{n}{j} B_j$$

For the Service Cutter, n is the number of nanoentities. The number of possible service cuts for $n = 20$ nanoentities is $51'724'158'235'372$[1].

The Bell number includes cuts for $1 - n$ number of services. In the context of a software system only certain numbers of services are realistic. The *Stirling numbers of the second kind* calculate the Bell number for a given number of sets k:

$$\left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^n$$

For $n = 20$ nanoentities and $k = 4$ services the equation results in $45'232'115'901$ possible cuts. For $k = 6$ the result is $4'306'078'895'384$.

During a discussion with our industry partner and supervisor, we decided that the Service Cutter should be able to process system models with up to 2000 nanoentities. We therefore concluded in the same meeting that this approach is not attainable without a heuristic attempt of finding a small set of relevant candidate cuts.

A possible heuristic approach is to take into account one or a few coupling criteria information about the system to find service candidates. A simple example would be to only analyze cuts where nanoentities of the same entity are not split across services so that only entities and not its nanoentities need to be considered.

[1] We do not print the number for the required 2000 nanoentities for lack of space in this document.

As we tried to find a heuristic approach to calculate candidate cuts, we discovered a new idea for the clustering algorithm described in the next section.

A.3 Approach #3: Greedy Service Construction

While analyzing the decomposition problem we realized that finding a good number of services is one of the key challenges. Cohesiveness criteria are mainly satisfied by consolidating nanoentities in one service while compatibility criteria request exactly the opposite, namely the separation of nanoentities. Figure A.5 illustrates this dilemma.

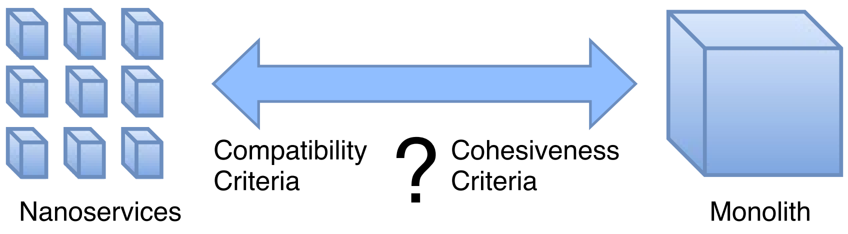


Figure A.5: Finding a good number of services is a key challenge of service decomposition.

To simplify the problem, we assume the number of services is given by the user and does not need to be computed. Very often an architect has a reasonable assumption about the number of services suitable for his system.

Given the number of services, the services can be imagined as empty boxes that need to be filled with nanoentities. The boxes are filled by construction and optimization steps. In every construction step, one nanoentity is assigned to a box by means of a greedy algorithm.

Construction Step One nanoentity is taken from an unordered list of all unassigned nanoentities and put in the best suitable service. The suitability is calculated by the criterion scores between the selected nanoentity and the nanoentities already assigned to a service.

Such greedy algorithms often end in a local maxima and not in the global maxima that represents the optimal solution. In every step the assignment decision is only based on the already assigned nanoentities. Assignments by former steps cannot be changed. To improve this behavior, the algorithm should process optimization steps between or after construction.

Optimization Step One service with already assigned nanoentities is randomly chosen. Within the service, the score from each nanoentity to all its neighbors is calculated to determine the least suitable nanoentity in a service. This nanoentity is taken out of the service and put back on the list of unassigned nanoentities.

The algorithm starts by assigning the first nanoentity to a random service and then alternates between construction and optimization stages. It finishes either after a given time, by the user stopping the optimization or by detecting that no further optimization is possible. This is detected when nanoentities taken from services in an optimization step are put back to their original service during the construction step. To finish the algorithm all nanoentities must be assigned to a service.

We have not proved that this algorithm solves the desired problem. The approach focuses only on cohesion within services but does not take coupling between services into consideration. Whether these two requirements build a duality, meaning that optimizing one automatically optimizes the other, is to be analyzed. However, we did not continue to investigate in this approach due to new findings on graph clustering described in the next section.

A.3.1 New Findings on Graph Clustering

Parallel to finding new approaches we continued to investigate the graph clustering.

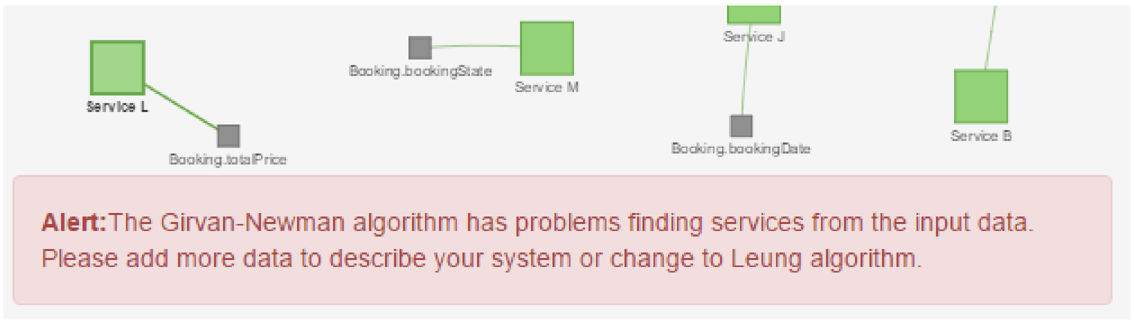
By analyzing the Girvan-Newman algorithm documented in Section 6.4.1, we were able to identify the problem encountered with the booking sample. As the sample only provides *Lifecycle & Identity Commonality* data, every pair of nanoentities in the graph was either directly or not connected at all. The calculated edge betweenness, which Girvan-Newman is based on, is therefore equal for all edges in the graph as every shortest path only passes one edge. The Gephi[55] implementation of Girvan-Newman then consequently removes all edges in the first iteration leaving every nanoentity isolated which then results in one service per nanoentity. Adding only one more information like nanoentity characteristics or use cases solved this problem.

Through further research on the topic we found the algorithm defined by Leung implemented in the GraphStream[58] project and integrated it into the Service Cutter. First tests using the booking sample provided the expected results.

A.4 Conclusion

While all approaches possibly lead to the desired results, the new findings on the Girvan-Newman and Leung algorithms promised the best results with adequate effort. Rating possible service cuts or heuristic construction of services would both require greater effort. Together with our stakeholders we decided that the effort is not worth taking, considering that the graph clustering provides reasonable results.

We furthermore implemented a warning in the Service Cutter should a user provide input leading to the faulty behavior of Girvan-Newman as shown in Figure A.6



Alert: The Girvan-Newman algorithm has problems finding services from the input data. Please add more data to describe your system or change to Leung algorithm.

Figure A.6: Service Cutter shows a warning for faulty Girvan-Newman results.

B. Service Cutter Assessment

We used two example systems to test the Service Cutter. After defining the input used to describe the systems, we discussed and documented our expectations on how service decomposition of the systems would make sense from our experience. In order to rate the candidate service cuts provided by the service cutter we defined three categories of cuts:

Excellent Service Cut The cut is not the way we expected it but we find reasons why the cut makes sense from an architect's perspective. It is therefore improving our own view of the analyzed system.

Expected Service Cut The cut is exactly the way we expected it.

Unreasonable Service Cut The cut is not as we expected it and we do not find reasons why the cut would provide any benefits to a systems architecture.

To assess the Service Cutter's output, the following language is used:

- An *excellent* output contains zero unreasonable service cuts and at least one excellent service cut.
- An *good* output contains zero unreasonable service cuts.
- An *acceptable* output contains at most one unreasonable service cut.
- A *bad* output contains two or more unreasonable service cuts.

We evaluated the service cuts of the following test systems:

1. For the fictional Trading System we engineered requirements based on our experience with similar software.
2. For the Cargo Tracking System we extracted the requirements from the existing application.

We therefore tested the Service Cutter once with forward- and once with reverse-engineering. The following sections present our test results in detail.

B.1 Trading System

We developed the Trading System as a fictional example based on personal experience with similar systems. Its goal is to include various different coupling aspects in a rather small model.

The Trading System is an application one might find in a typical Swiss private bank offering its customers the ability to manage their stocks portfolio.

- The main focus is to buy or sell *stocks* at a specific price (*Order.triggerPrice*) using an *order*.
- *Prices* are frequently imported from a market data provider and upon import of a price, all orders are checked for orders that can be triggered.
- When an order is executed, an instruction is sent to the market to purchase or sell the stocks. The *PaymentInfo* contains all necessary information to do so.
- *News* are imported from an external provider and are linked to a specific stock. They provide valuable, contextual information when using the system. However traders and customers can easily fall back to any online source should this information not be available.
- *Recommendations* are suggested to the user of the system based on his existing portfolio.

Figure B.1 shows a domain model of the Trading System.

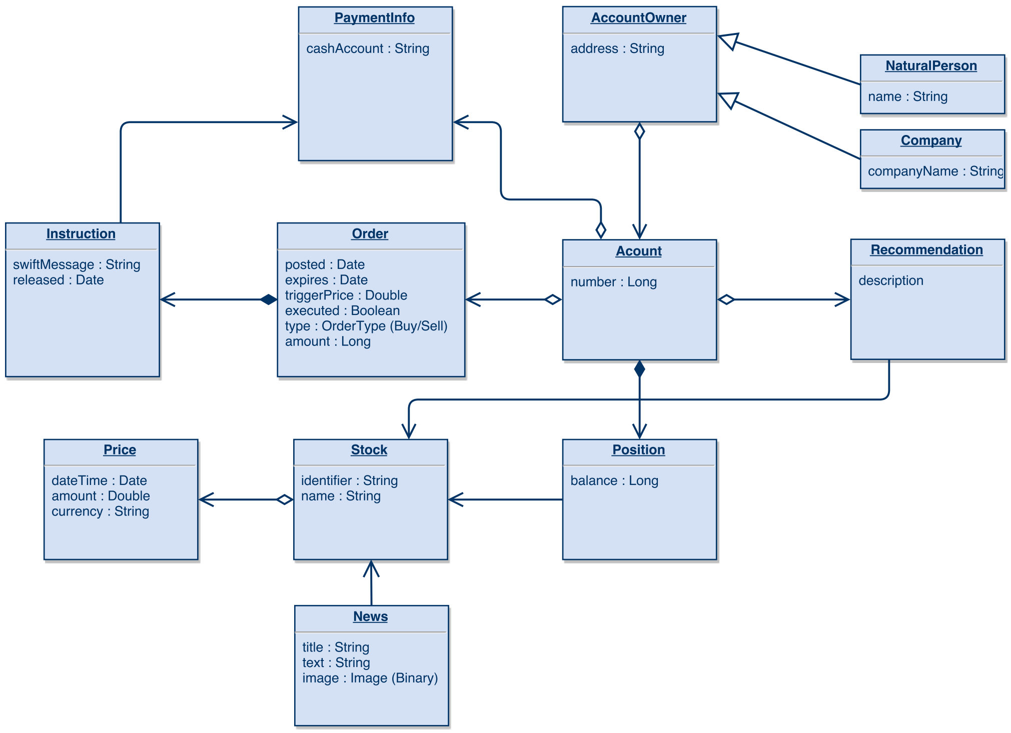


Figure B.1: Trading System Class Diagram

The following ten use cases describe the supported functionality of the application.

1. Post Order
   - Nanoentities written: Order.posted, Order.expires, Order.triggerPrice, Order.executed, Order.type, Order.amount
   - Nanoentities read: Account.number, Stock.identifier, Stock.stockName
2. Instruct Order
   - Nanoentities written: Instruction.instructedTime, Order.executed, Position.balance
   - Nanoentities read: PaymentInfo.cashAccount
3. Import Price and Check for Due Orders (Technical)
   - Nanoentities written: Price.dateTime, Price.price, Price.currency
   - Nanoentitiess read: Order.triggerPrice, Stock.identifier
4. Read News
   - Nanoentities written: -
   - Nanoentities read: Stock.identifier, News.title, News.text, News.image

5. Import News (Technical)
   - Nanoentities written: News.title, News.text, News.image
   - Nanoentities read: Stock.identifier
6. View Recommendations
   - Nanoentities written: -
   - Nanoentities read: Account.number, Recommendation.description, Stock.identifier, Stock.stockName
7. Suggest Recommendations (Technical)
   - Nanoentities written: Recommendation.description
   - Nanoentities read: Account.number, Stock.identifier, Stock.stockName, Position.balance
8. Create Account
   - Nanoentities written: Account.number
   - Nanoentities read: AccountOwner.address, NaturalPerson.name, Company.companyName
9. Create Account Owner
   - Nanoentities written: AccountOwner.address, NaturalPerson.name, Company.companyName
   - Nanoentities read: -
10. View Portfolio
    - Nanoentities written: -
    - Nanoentities read: Account.number, Position.balance, Stock.identifier, Stock.stockName, Order.triggerPrice, Order.amount, Order.posted, Order.expires, Order.executed, Order.type

In addition to the use cases, we defined the following characteristics.

Security Criticality

- **Critical**: AccountOwner.address, NaturalPerson.name, Company.companyName
- **Public**: Stock.identifier,Stock.stockName, Price.dateTime, Price.price, Price.currency, News.title, News.text, News.image

Content Volatility

- **Often**: Price.dateTime, Price.price, Price.currency
- **Rarely**: AccountOwner.address, NaturalPerson.name, Company.companyName, Account.number

Consistency Criticality

- **Eventually**: Price.dateTime, Price.price, Price.currency

Storage Similarity

- **Huge**: News.image

Structural Volatility

- **Often**: Recommendation.description

Availability Criticality

- **Low**: News.title, News.text, News.image, Recommendation.description

Furthermore, all default characteristics as documented in Section 4.2 are taken into account.

B.1.1 Expected Service Cuts

From our experience in software architecture we expect the Service Cutter to decompose the Trading System into the services presented in Figure B.2.

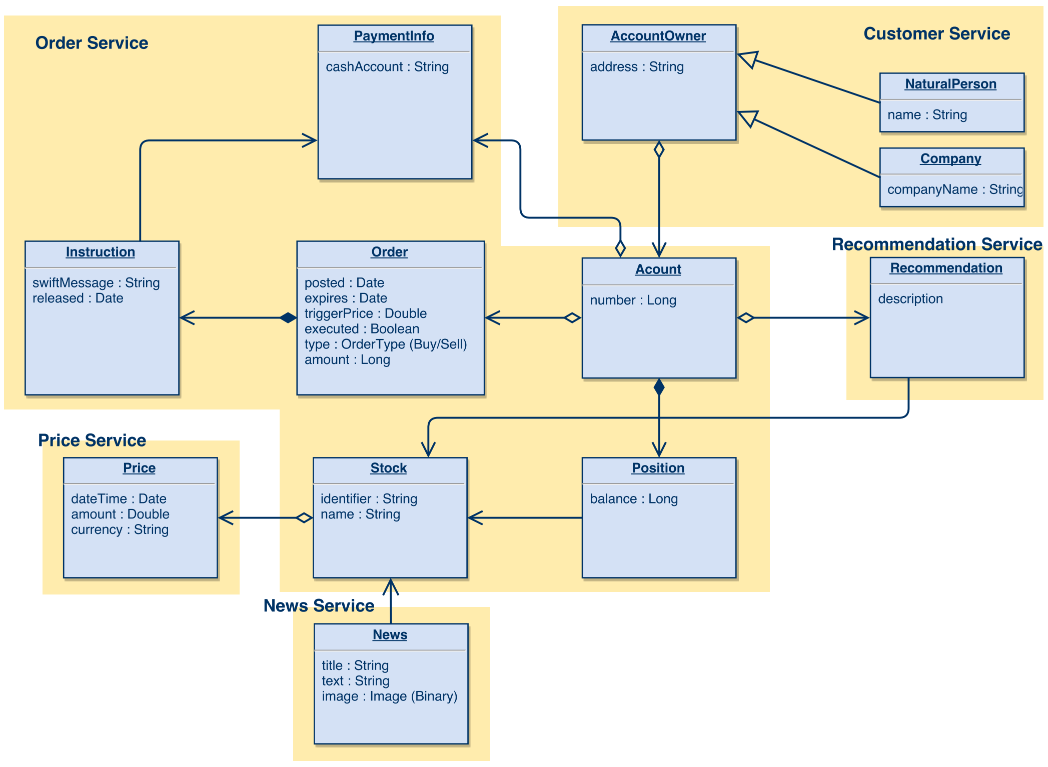


Figure B.2: Trading System expected service cuts.

The following reasons led us to this decision.:

- The service *Order* encapsulates many use cases and contains several entities that need to be processed with high consistency (Order, Position).
- The high content volatility of the entity price led to the isolation of this part into a separate service *Price*.
- News are not part of the core operations and therefore require lower availability and security criticality. News images furthermore require a significant amount of storage. Therefore, we would isolate this into a separate *News* service.
- The recommendation algorithm will be changed frequently. A dedicated *Recommendation* service allows independent deployment of updated versions. Moreover recommendations are, like the news, not part of the core operations and require lower availability.
- Security restrictions requires all Personally Identifiable Information (PII) to be separated from other data. Extracting it into a *Customer* service allows the architect to protect this data with additional measures.

B.1.2 Girvan-Newman Algorithm Assessment

Figure B.3 is the suggested cut as calculated by the Service Cutter with the Girvan-Newman algorithm.

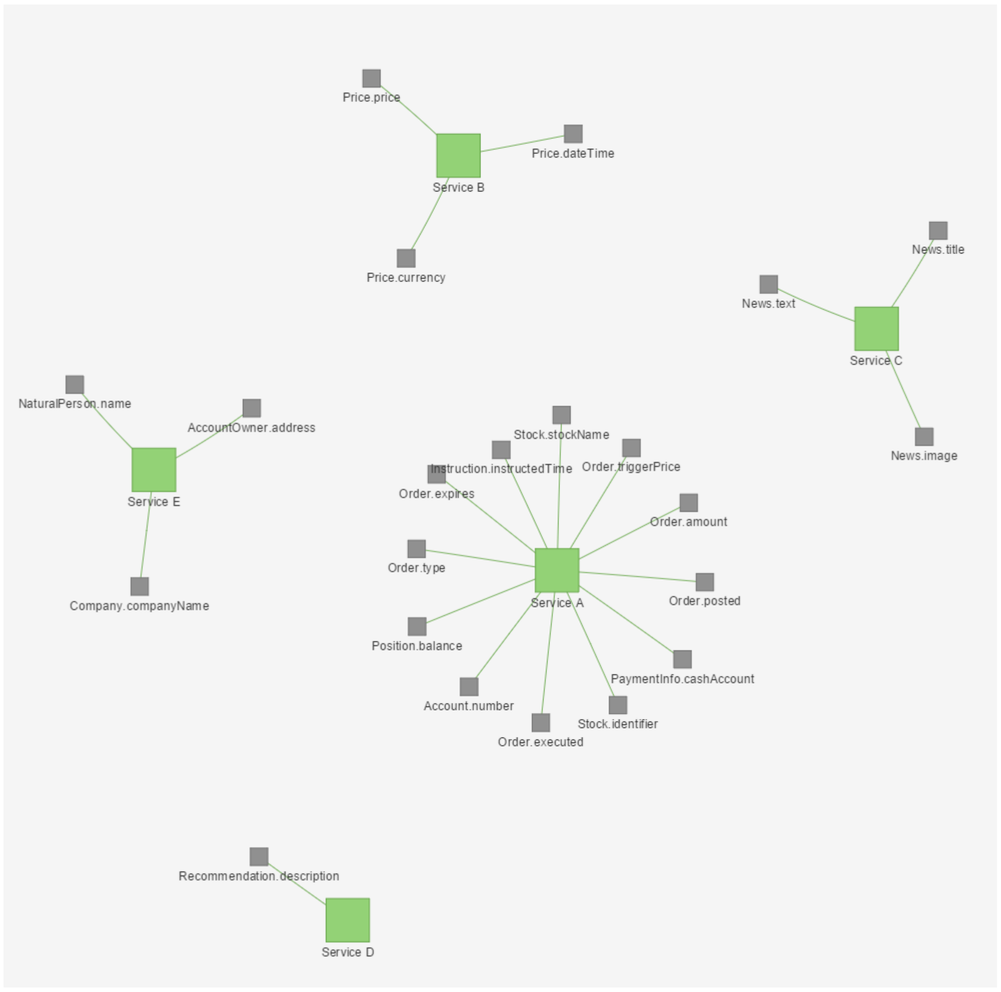


Figure B.3: Trading System actual service cuts.

The parameters that produce a cut as seen in B.3 are the following:

- Coupling criteria priorities: Defaults as defined in 6.5.3
- Number of services: 5

The presented candidate service cuts match exactly the expected services and are therefore considered a good result.

Priorities Sensitivity

The candidate service cuts are very stable when changing the priorities. Changing the two relevant cohesiveness parameters *Identity & Lifecycle Commonality* and *Semantic Proximity* to any combination between *XS* and *L* only affects the result when both set to

L. With these priorities the Service Cutter suggests an own service for *Stock.stockName* and *Stock.identifier* instead of *Recommendation.description*.

An own service for stocks is reasonable as these nanoentities can be categorized as master data and not transaction data like most of the other nanoentities in the order service. Requesting 6 number of services with these priorities creates cuts for both the recommendation and the stock service.

Changing the priority of compatibility or constraint criteria to any value between *XS* and *L* does not affect the resulting cuts. However, the possibility to request a small number of services gets lost as too many relations between nanoentites are cut if criteria scoring negative values receive a high priority.

Number of Services

As Girvan-Newman receives a number of services parameter we can analyze how a monolithic architecture would be split up step by step. The resulting services by each parameter are listed in Table B.1.

Table B.1: Girvan-Newman cuts of trading system with different number of services.

Number of services	**Services**
1	Not supported
2	Customer Service extracted (only supported with *Semantic Proximity* L)
3	Price Service extracted
4	News Service extracted
5	Recommendation Service extracted
6	Not supported
7	Payment Service and Instruction Service extracted
8	Stock Service extracted
9	Stock Service split in *Stock.name* and *Stock.identifier*

Reasonable cuts are presented for up to 8 different services. The results produced by Girvan-Newman with the Trading System example are considered *good*.

B.1.3 Leung Algorithm Assessment

Leung produces varying suggested cuts as it is not a deterministic algorithm. The algorithm is therefore harder to analyze, as we can't determine if changes in results emerge from changes in priorities or by coincidence.

Generally speaking, the results are similar to those by Girvan-Newman. With the default priorities, the result matches the one of Girvan-Newman shown in Figure B.3 in approximately 50%. In the cases it is not matching it suggests fewer service by merging two services together. This is due to the accidentally created monster clusters described in Section 6.4.2.

Priorities Sensitivity

As already mentioned we cannot assess the priority sensitivity with Leung. Changing criteria priorities to values from *XS* to *L* has in some cases produced the following results differing from the one shown in Figure B.3.

- An extra service for PaymentInfo.
- An extra service for Stock.
- An extra service for Account.
- A service combining Account, PaymentInfo and Position.
- A service combining News and Stock.

Except of the last cut we find reasonable justifications for all cases. In fact, combining PaymentInfo, Account and Position into one service seems to be a considerable suggestion we have not thought of before and is therefore considered an *excellent* solution. These entities are closely related to an account while the other entities in the order service are more focused on the trading itself.

Number of Services

The number of services suggested by Leung vary between 2 and 7 services. There is a tendency towards a higher number of services when cohesiveness criteria are prioritized high and compatibility, and constraints criteria are prioritized low. The same applies vice versa.

With only small changes to the priorities Leung often suggests 4 or 5 services which matches with our expectation.

B.1.4 Conclusion

In conclusion we can say that the Service Cutter did suggest the Trading System service cuts as we expected. For both algorithms results are considered *good* with only one or two exceptions where the result was considered *acceptable* as described above. These are very satisfying results for a first test of the Service Cutters scoring and algorithms.

The tests furthermore show the advantages and disadvantages of a deterministic algorithm. Girvan-Newman produced very stable results and nicely shows the path from a monolith to a more service oriented architecture. Leung on the other hand is harder to analyze but provided us more reasonable candidate service cuts we have not considered before. Leung also suggests a reasonable number of services while this consideration completely lies in the responsibility of the user when using Girvan-Newman.

B.2 Cargo Tracking System - Domain-Driven Design Sample

The Cargo Tracking System is a well known software project created to illustrate the concepts and patterns described in the DDD book by E. Evans[12]. The DDDSample is hosted on Github[49] and a short screencast on YouTube[50] outlines its functionality.

The Cargo Tracking System provides a domain with a suitable complexity and, unlike the Trading System, comes with an already implemented and well reasoned architecture. With reverse engineering, we extracted the domain model, use cases and some characteristics from the code. The Cargo Tracking implements the following functionalities:

- The main focus is to transport a *Cargo* from *Location* A to *Location* B. *Cargos* are created with a *TrackingId* and specified with a *RouteSpecification*. Once created, one of multiple suitable *Itineraries* is assigned.
- The system calculates suitable *Itineraries* for a *Cargo* from existing *Voyages* each containing a list of *CarrierMovements*.
- Once a *Cargo* is routed, *HandlingEvents* track the progress of each *Cargo's Itinerary*. A *HandlingEvent* contains information about the event and references a *Cargo* on a specific *Voyage* and occurs in a particular *Location*.
- The *Delivery* of a *Cargo* informs about its state, estimated arrival time and contains information whether the *Cargo* is on track or not.

The extracted domain model is not a one-to-one copy of the domain classes in the code. The domain classes contain some calculated and therefore redundant information which have been merged into single nanoentities in the domain model shown in Figure B.4.

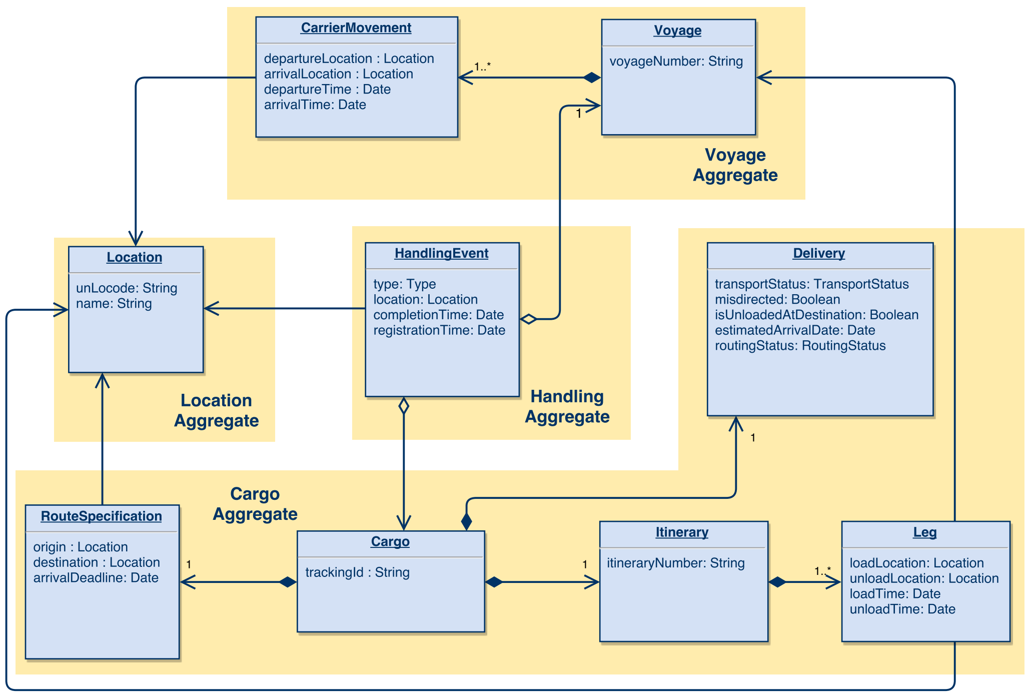


Figure B.4: DDD Sample with Aggregates.

Figure B.4 additionally outlines the package and aggregate structure provided by the DDDSample which is a first indication about service decomposition.

The following use cases describe the supported functionality of the application.

1. ViewTracking
   - Nanoentities written: -
   - Nanoentities read: Cargo.trackingId, HandlingEvent.type, HandlingEvent.location, HandlingEvent.completionTime, Delivery.transportStatus, Delivery.estimatedArrivalTime, Delivery.misdirected, Voyage.voyageNumber, RouteSpecification.destination, Stock.stockName
2. ViewCargos
   - Nanoentities written: -
   - Nanoentities read: Cargo.trackingId, RouteSpecification.destination, RouteSpecification.arrivalDeadline, Delivery.routingStatus, Itinerary.itineraryNumber
3. BookCargo

- Nanoentities written: Cargo.trackingId, RouteSpecification.origin, RouteSpecification.arrivalDeadline, RouteSpecification.destination
- Nanoentities read: Location.unLocode

4. ChangeCargoDestination
   - Nanoentities written: RouteSpecification.destination
   - Nanoentities read: Cargo.trackingId, RouteSpecification.destination
5. RouteCargo
   - Nanoentities written: Itinerary.itineraryNumber, Leg.loadLocation, Leg.unloadLocation, Leg.loadTime, Leg.unloadTime
   - Nanoentities read: Cargo.trackingId, RouteSpecification.destination, RouteSpecification.origin, RouteSpecification.arrivalDeadline, Location.unLocode, Voyage.voyageNumber, CarrierMovement.departureLocation, CarrierMovement.arrivalLocation, CarrierMovement.departureTime, CarrierMovement.arrivalTime
6. Create Location
   - Nanoentities written: Location.unLocode, Location.name
   - Nanoentities read: -
7. Create Voyage
   - Nanoentities written: Voyage.voyageNumber
   - Nanoentities read: -
8. Add CarrierMovement
   - Nanoentities written: CarrierMovement.departureLocation, CarrierMovement.arrivalLocation, CarrierMovement.departureTime, CarrierMovement.arrivalTime
   - Nanoentities read: Voyage.voyageNumber
9. Handle Cargo Event
   - Nanoentities written: HandlingEvent.type, HandlingEvent.completionTime, HandlingEvent.registrationTime, HandlingEvent.location, Delivery.transportStatus, Delivery.misdirected, Delivery.estimatedArrivalTime, Delivery.isUnloadedAtDestination, Delivery.routingStatus
   - Nanoentities read: Voyage.voyageNumber, Cargo.trackingId

In addition to the use cases, we assumed the following characteristics.

Content Volatility

- **Often**: HandlingEvent.type, HandlingEvent.completionTime, HandlingEvent.registrationTime, HandlingEvent.location, Delivery.transportStatus
- **Rarely**: Location.unLocode, Location.name

Structural Volatility

- **Rarely**: Location.unLocode, Location.name

Furthermore all default characteristics as documented in Section 4.2 are taken into account.

The DDDSample provides multiple interfaces depending on the user's role. The main interface provides tracking information, the administration panel offers means to create cargos and plan their itinerary and an additional interface is used to inform about handling events. Out of these distinction by the UI we defined the roles listed in Table B.2 each containing a group of nanoentities sharing an owner.

Table B.2: Shared owners in the DDDSample.

Role	**Owned nanoentities**
CargoPlanner	Cargo.trackingId RouteSpecification.origin RouteSpecification.destination RouteSpecification.arrivalDeadline Itinerary.itineraryNumber Leg.loadLocation Leg.unloadLocation Leg.loadTime Leg.unloadTime Delivery.estimatedArrivalTime Delivery.routingStatus
CargoTracker	HandlingEvent.type HandlingEvent.completionTime HandlingEvent.registrationTime HandlingEvent.location Delivery.transportStatus Delivery.misdirected Delivery.isUnloadedAtDestination
VoyageManager	Voyage.voyageNumber CarrierMovement.departureLocation CarrierMovement.arrivalLocation CarrierMovement.departureTime CarrierMovement.arrivalTime
Admin	Location.name, Location.unLocode

B.2.1 Expected Service Cuts

From our experience in software architecture we expect the Service Cutter to decompose the Cargo Tracking System into the services presented in Figure B.5.

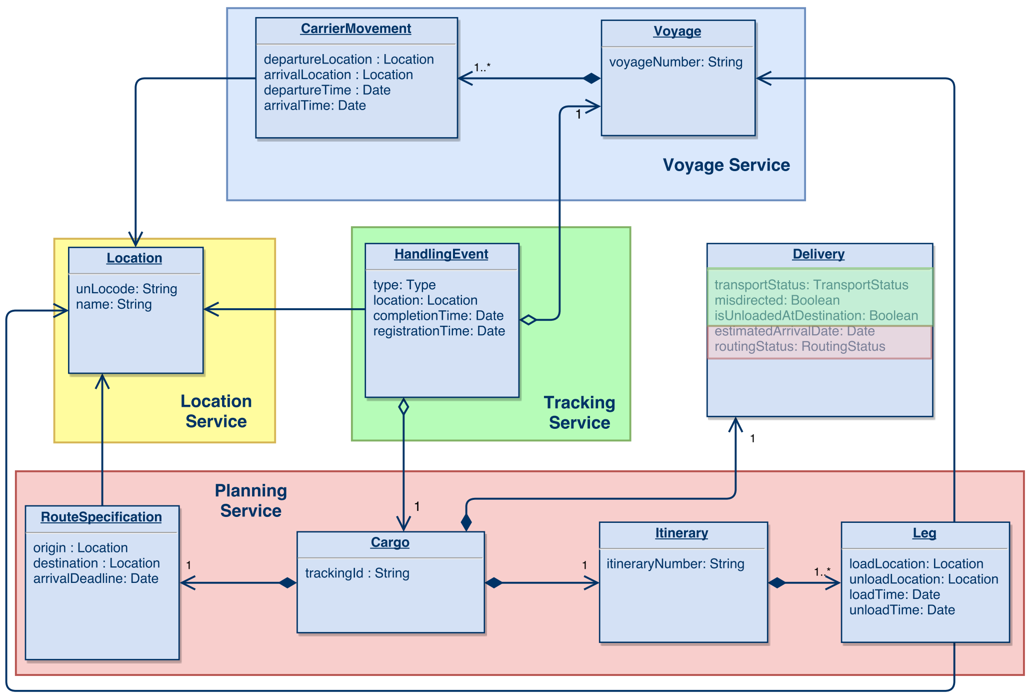


Figure B.5: DDD Sample with expected services.

As there are not many compatibility criteria defined, the main reasons for our decomposition solution are responsibilities and semantic proximity by use cases:

- The *Voyage Service* contains all nanoentities regarding actual voyages and their movements, regardless of any cargo.
- The *Location Service* is separated as a consequence of the low data and structural volatility of these nanoentities. Locations are used and referred from almost all entities but very rarely written. This service could be categorized as a master data service.
- The *Planning Service* handles all nanoentities regarding cargos and their itinerary.
- The *Tracking Service* is responsible to track the actual events of a cargo. The aggregates defined in the DDDSample assign the delivery to the Planning Service. In our opinion the nanoentities *transportStatus*, *misdirected* and *isUnloadedAtLocation*

are better assigned to the Tracking Service as they are defined as a consequence of handling events.

B.2.2 Girvan-Newman Algorithm Assessment

For the Cargo Tracking System, we adjusted the default priorities to the following values. These priorities provided the best results and match the characteristics of the Cargo Tracking System.

- Content Volatility: *S* instead of *XS*
- Structural Volatility: *S* instead of *XS*
- Shared Owner: *L* instead of *M*

The resulting candidate service cuts by the Girvan-Newman algorithm for 4 services are shown in Figure B.6.

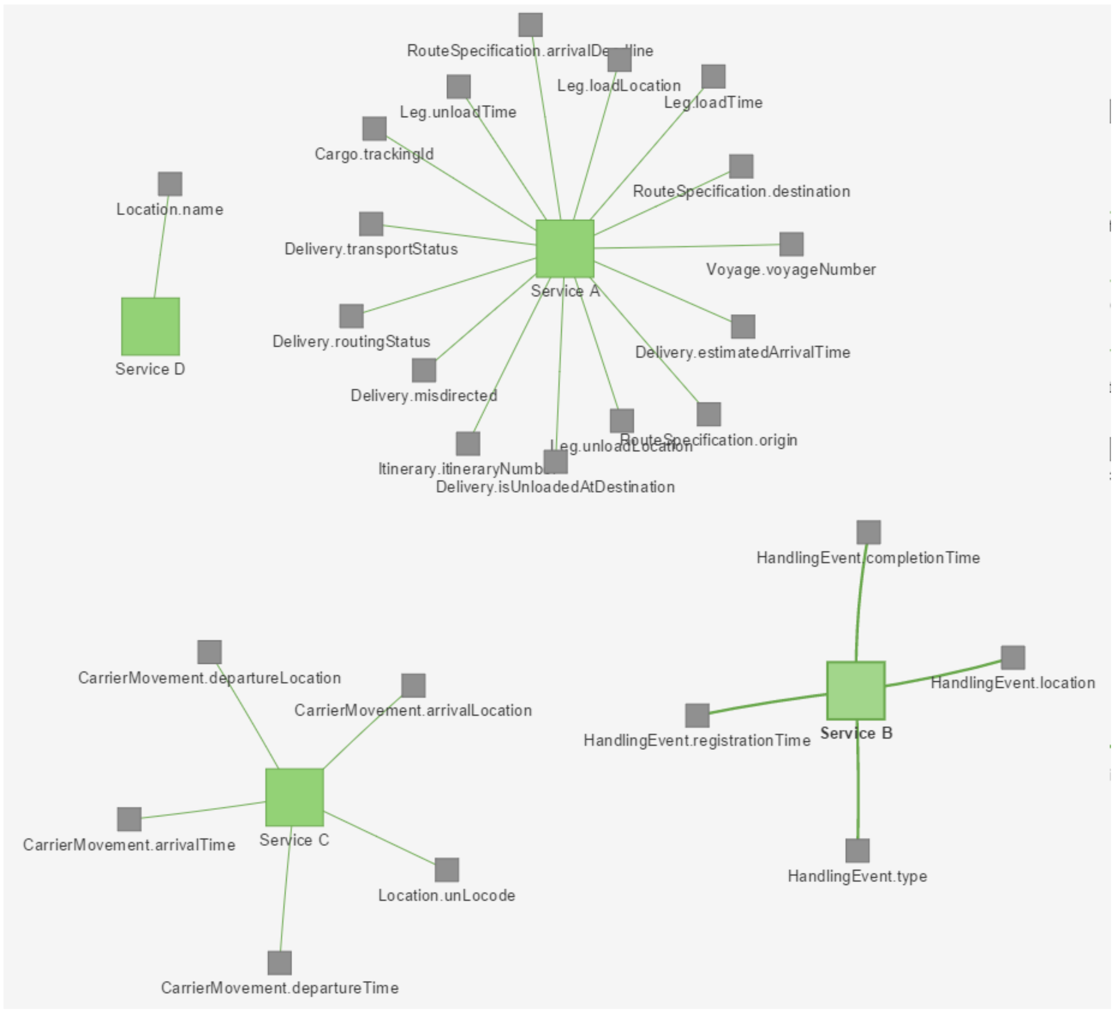

RouteSpecification.arrivalDeadline
Leg.loadLocation
Leg.loadTime
Leg.unloadTime
Cargo.trackingId
RouteSpecification.destination
Location.name
Delivery.transportStatus
Voyage.voyageNumber
Service A
Delivery.routingStatus
Delivery.estimatedArrivalTime
Service D
Delivery.misdirected
Leg.unloadLocation
RouteSpecification.origin
Itinerary.itineraryNumber
Delivery.isUnloadedAtDestination
HandlingEvent.completionTime
CarrierMovement.departureLocation
CarrierMovement.arrivalLocation
HandlingEvent.location
HandlingEvent.registrationTime
Service B
CarrierMovement.arrivalTime
Service C
Location.unLocode
HandlingEvent.type
CarrierMovement.departureTime

Figure B.6: Cargo Tracking System service cuts by Girvan-Newman.

Surprisingly, the location has been split to Service D and Service C. Service C contains the carrierMovement nanoentities but misses the voyageNumber which is closely related to the carrierMovement through responsibilities and use cases. Service B represents the handling aggregate as defined by the DDDSample but does not contain delivery nanoentites as expected by us.

None of the service contains the nanoentities as expected by us and only Service B can be categorized as reasonable service cut. We rate this a *bad* result.

The split of location might be due to the fact that in most use cases only Location.unLocode is used and not Location.name. We changed the priority for *Semantic Proximity* from *M* to *S* which resulted in the candidate service cuts shown in Figure B.7

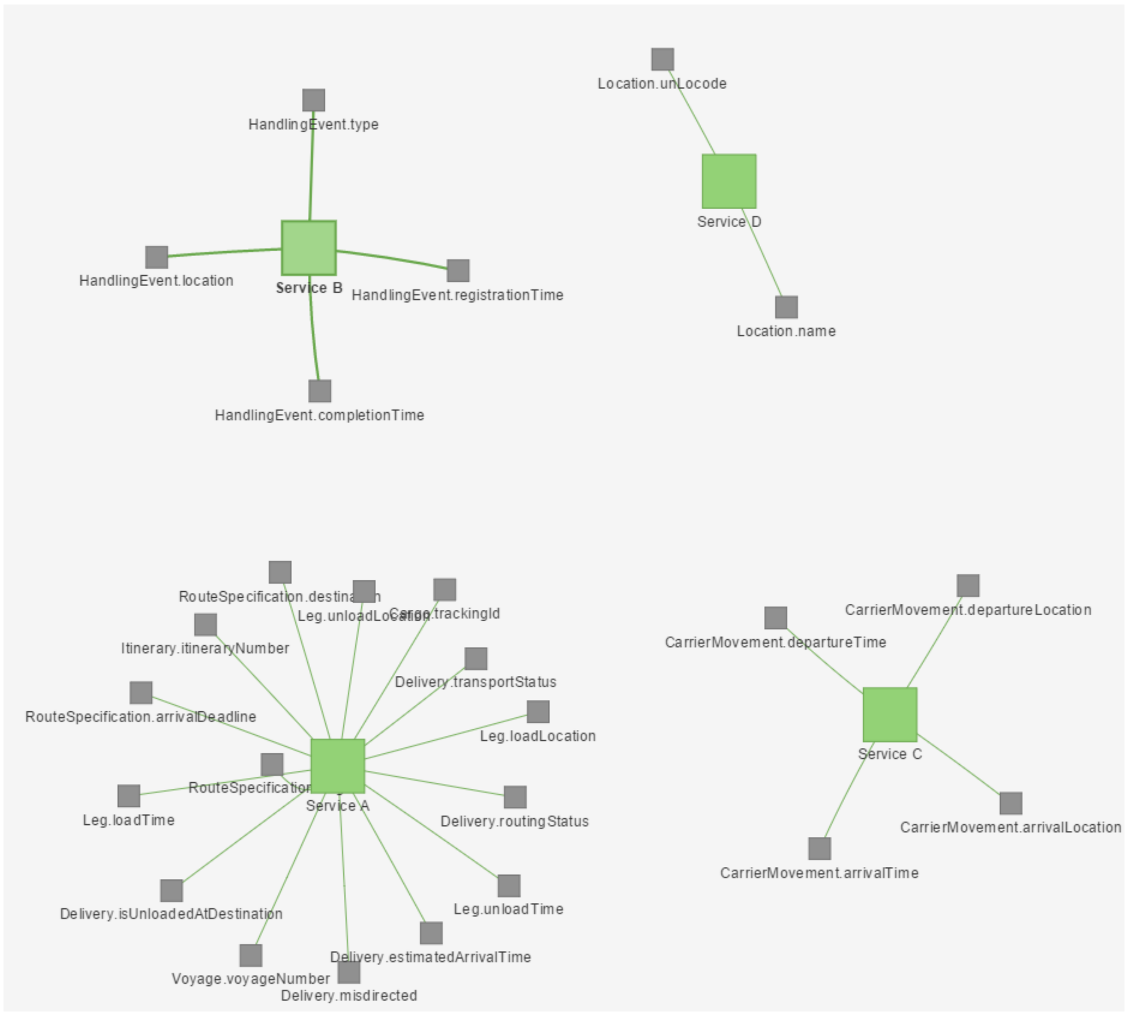

HandlingEvent.type
HandlingEvent.location
Service B
HandlingEvent.registrationTime
HandlingEvent.completionTime
Location.unLocode
Service D
Location.name
RouteSpecification.destination
Leg.unloadLocation
Cargo.trackingId
Itinerary.itineraryNumber
Delivery.transportStatus
RouteSpecification.arrivalDeadline
Leg.loadLocation
RouteSpecification
Leg.loadTime
Service A
Delivery.routingStatus
Leg.unloadTime
Delivery.isUnloadedAtDestination
Delivery.estimatedArrivalTime
Voyage.voyageNumber
Delivery.misdirected
CarrierMovement.departureLocation
CarrierMovement.departureTime
Service C
CarrierMovement.arrivalLocation
CarrierMovement.arrivalTime

Figure B.7: Semantic Proximity with priority S instead of M.

Location has now its own service as expected. This improves the result a little, but still not enough to consider it *acceptable*.

Priorities Sensitivity

Changing priorities of the relevant coupling criteria to values between *XS* and *L* results in minor changes. The following alternations have been produced:

- Delivery.transportStatus is assigned to the service containing handling events if *Identity &Lifecycle Commonality* is set to *XS*. This is closer to our expectation of a tracking service.

- Increasing the priority of *Semantic Proximity* to *L* produces an unreasonable service containing Cargo, Voyage and RouteSpecification nanoentities.

We were not able to produce *acceptable* or *good* results for the Cargo Tracking System with Girvan-Newman. The sensitivity of the results on priority changes seems acceptable.

B.2.3 Leung Algorithm Assessment

Like Girvan-Newman, the Leung algorithm only suggests a location service if *Semantic Proximity* is set to *S* instead of *M*. The candidate service cuts are shown in Figure B.8.

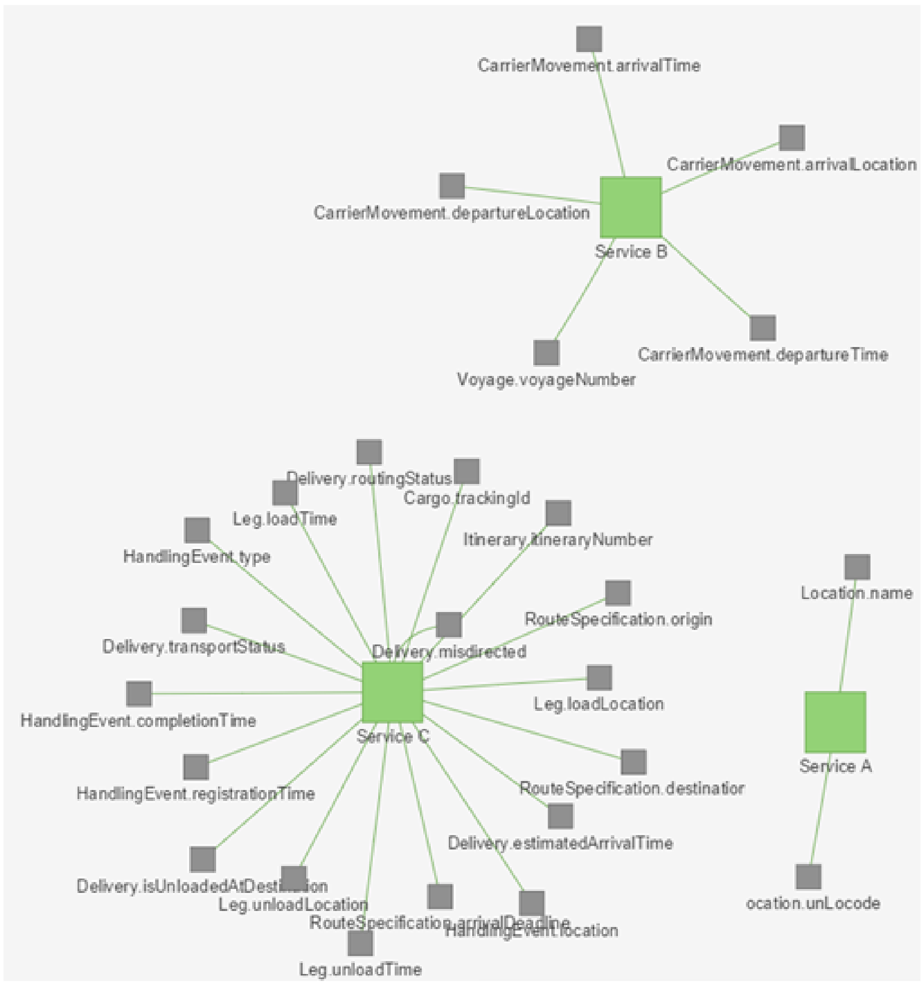

CarrierMovement.arrivalTime
CarrierMovement.arrivalLocation
CarrierMovement.departureLocation
Service B
CarrierMovement.departureTime
Voyage.voyageNumber
Delivery.routingStatus
Cargo.trackingId
Leg.loadTime
Itinerary.itineraryNumber
HandlingEvent.type
Location.name
RouteSpecification.origin
Delivery.transportStatus
Delivery.misdirected
Leg.loadLocation
HandlingEvent.completionTime
Service C
Service A
HandlingEvent.registrationTime
RouteSpecification.destination
Delivery.estimatedArrivalTime
Delivery.isUnloadedAtDestination
ocation.unLocode
Leg.unloadLocation
RouteSpecification.arrivalDeadline
HandlingEvent.location
Leg.unloadTime

Figure B.8: Cargo Tracking System service cuts by Leung.

A location and a voyage service have been extracted while the planning and tracking service are merged together. A different run with the same priorities is shown in Figure B.9.

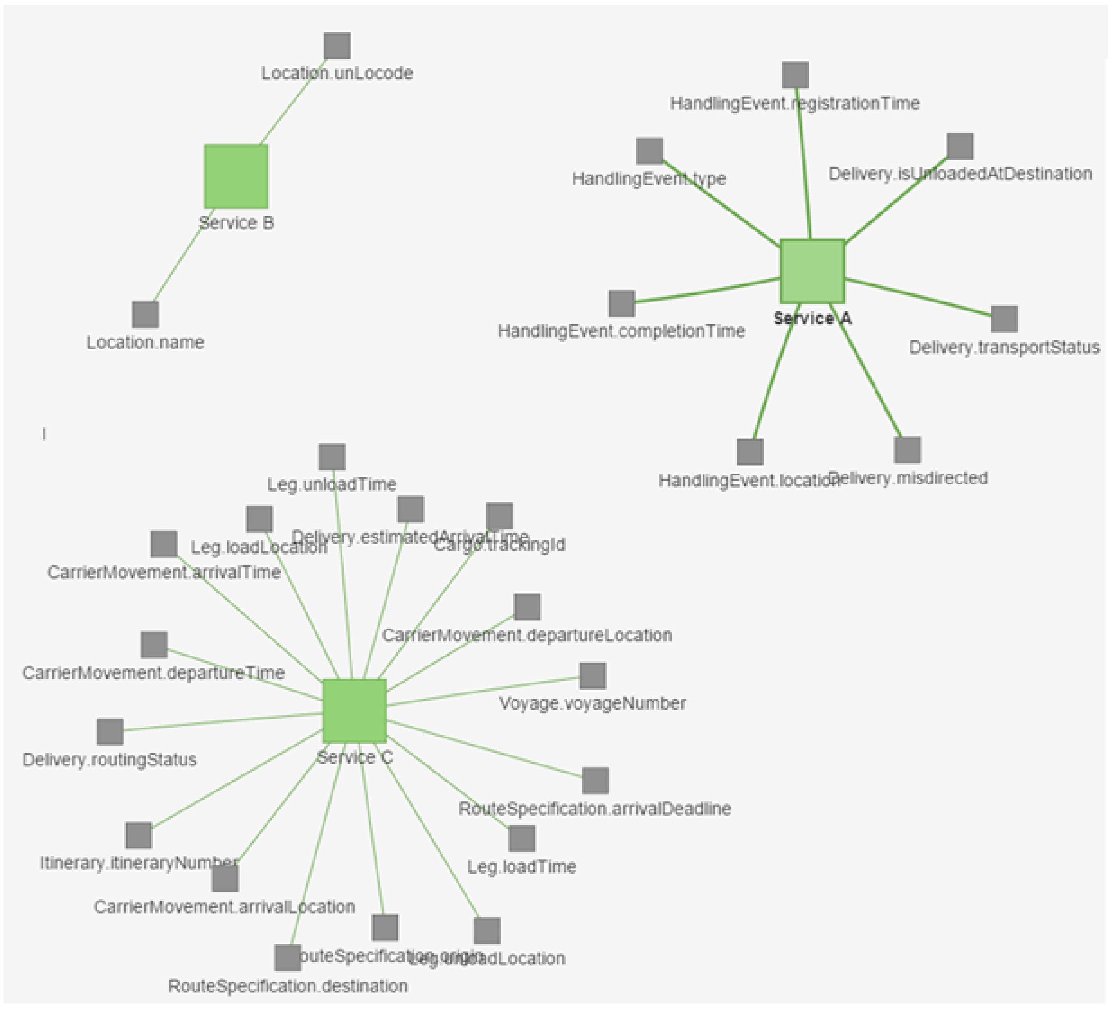

Location.unLocode
Service B
Location.name
HandlingEvent.registrationTime
HandlingEvent.type
Delivery.isUnloadedAtDestination
HandlingEvent.completionTime
Service A
Delivery.transportStatus
HandlingEvent.location
Delivery.misdirected
Leg.unloadTime
Leg.loadLocation
Delivery.estimatedArrivalTime
Cargo.trackingId
CarrierMovement.arrivalTime
CarrierMovement.departureLocation
CarrierMovement.departureTime
Voyage.voyageNumber
Delivery.routingStatus
Service C
RouteSpecification.arrivalDeadline
Leg.loadTime
Itinerary.itineraryNumber
CarrierMovement.arrivalLocation
RouteSpecification.origin
Leg.unloadLocation
RouteSpecification.destination

Figure B.9: Cargo Tracking System service cuts by Leung.

This time Leung extracted the tracking service and kept voyage and planning services together. Noteworthy is the fact that Leung splits the delivery entity in two different services the same way we expected it.

Other runs resulted in similar cuts whereas often only the location service was extracted. The cuts done by Leung meet our expectations precisely but provides less services than expected. We assume this results from the label propagation problem described in Section 6.4.2.

The results provided by Leung can be classified as *acceptable*.

B.3 Conclusion

We conclude that, depending on the specified system, one or both algorithms produce *acceptable* or *good* candidate service cuts that help an architect. However, the architect always has to verify the result before taking any decision. These results suggest that the Service Cutter approach can be used to assist service decomposition decisions. Further improvements on the scoring system or algorithms presented in Chapter 9 may improve the presented result.

C. Graph Clustering Evaluation

This Appendix documents the requirements and evaluation of clustering algorithms.

C.1 Requirements

The requirements listed in Table C.1 should be met by the algorithm and its implementation:

Table C.1: Algorithm Requirements

Name	Description	Priority
Distinct Clusters	Every nanoentity is contained once and only once in a cluster.	High
Minimal Coupling	The total weight of the edges connecting clusters should be minimal.	High
Implementation	A free implementation of the algorithm should be available either in Java or another language easily callable from the Java Virtual Machine (JVM).	High
Number of Clusters	The number of clusters should be defined by a parameter.	Medium
Performance	The algorithm should not take longer than 2 minutes on an average computer to cluster 2000 nodes.	Medium
Simplicity	It should be possible to understand the mechanism and parameters of the algorithm within a day assuming the mathematical background of an average HSR student.	Low
Edge Cases	Cases in which it is unclear which cut is best should be visualized in form of a hint or multiple solution suggestions.	Low
License	The implementation license needs to be compatible with the Apache 2 license to be used in the Service Cutter. GPL should be avoided.	High

C.2 Algorithms Assessment

The algorithms listed in Table C.2 were found by consulting clustering algorithm comparisons published by the Computer Science Review[34] and the Physical Review E[21]. Further Research using Google's search engine and the community driven question and answer platform Stackoverflow[72] were used to find applicable implementations of the algorithms.

Table C.2: Algorithm Evaluation

Name	**Description**	**Implementation**	**Assessment**
MCL - Markov Cluster Algorithm[10]	A clustering algorithm working on weighted undirected graphs. MCL is based on Random Walks with Markov Chains.	Implementations of MCL are available in R and in Java as a plugin of the Gephi[55] platform.	Positive
HCS - Highly Connected Subgraphs[19]	A clustering algorithm working on unweighted undirected graphs. The CLICK clustering algorithm enhances HCS for weighted edges.	Implementations only available in R.	No Java implementation
Girvan–Newman[25]	A clustering algorithm working on weighted undirected graphs based on Edge-Betweenness.	Java implementations exist as part of the Jung[62] framework (only unweighted graphs) and as a plugin[56] of the Gephi[55] platform.	Positive
K-means[18]	A clustering algorithm working with vectors in an n-dimensional space.	Multiple implementations, for example as part of the Spark[44] framework, are available.	No simple way to transform the problem from a graph to vector based representation found.
Apiacoa[46]	A clustering algorithm working on unweighted undirected graphs. This algorithm is based on maximal modularity clustering.	Apiacoa.org provides an implementation of the algorithm in Java.	No support for weighted edges
Epidemic Label Propagation	A clustering algorithm working on weighted (un-)directed graphs. This algorithm was suggested by Raghavan[29] and refined by Leung[22].	GraphStream[58] provides an implementation of the algorithm in Java as part of their `gs-algo` package.	Positive

The three algorithms positively assessed are MCL, Girvan-Newman and Epidemic Label Propagation. These algorithms haven been tested and further evaluated as described in Section 6.4

D. Implementation Details

This appendix documents miscellaneous implementation details.

D.1 Docker Compose

A YAML file is used to configure and start all Docker containers required for the Service Cutter. The file shown in Listing D.1 is delivered as `docker-compose.yml` in the source code of the Service Cutter.

In a productive environment the default passwords have to be changed.

Listing D.1: Docker Compose definition for the Service Cutter.

```
db:
  image: postgres:9.4
  restart: always
  expose:
    - 5432
  environment:
    POSTGRES_USER: "editor"
    POSTGRES_PASSWORD: "N0A8KHWWHMjRBTrc8UjL"
editor:
  image: services-toolkit/editor
  ports:
   - "40001:8080"
  links:
    - db
    - engine
  environment:
    POSTGRES_USER: "editor"
    POSTGRES_PASSWORD: "N0A8KHWWHMjRBTrc8UjL"
    ENGINE_HOST: "engine"
    ENGINE_PORT: "8080"
engine:
  image: services-toolkit/engine
  expose:
   - 8080
  ports:
   - "40000:8080"
```

D.2 JSON Schema Export

Listing D.2 lists the JSON Schema that specifies the export format for candidate service cuts.

Listing D.2: JSON Schema for candidate services export.

```
{
    "$schema":"http://json-schema.org/draft-03/schema#",
    "type":"object",
    "additionalProperties": false,
    "properties":{
        "services":{
            "type":"array",
            "items":{
                "type":"object",
                "additionalProperties": false,
                "properties":{
                    "nanoentities":{
                        "type":"array",
                        "items":{
                            "type":"string"
                        }
                    },
                    "name":{
                        "type":"string"
                    },
                    "id":{
                        "type":"string"
                    }
                }
            }
        },
        "relations":{
            "type":"array",
            "items":{
                "type":"object",
                "additionalProperties": false,
                "properties":{
                    "serviceA":{
                        "type":"string"
                    },
                    "serviceB":{
                        "type":"string"
                    },
                    "score":{
                        "type":"integer"
                    },
                    "sharedEntities":{
                        "type":"array",
                        "items":{
                            "type":"string"
                        }
                    }
                }
            }
        },
        "useCaseResponsibility":{
            "type":"object"
        }
    }
  }
```

D.3 Performance

This section theoretically discusses whether the Service Cutter scales to support large data volumes.

The theoretical number of edges may cause performance problems. The maximum number of edges in a graph can be described using a formula: A graph of n nodes, where every node is connected to all other nodes, has e edges.

$$e = \frac{n(n-1)}{2}$$

The number of edges grows almost quadratically with up to 1'999'000 edges for 2000 nodes.

Our implementation however will unlikely have anything close to the theoretical number of edges as:

- Coupling of type *cohesiveness* only adds edges where nanoservices are in a relationship with each other.
- Coupling of type *compatibility* only adds a negative score (penalty) to existing relationships.
- Coupling of type *constraint* only removes existing edges.

The number of edges can therefore only cause problems when a cohesiveness coupling is specified that includes a large number of nodes. An example of such a coupling is for a use case including a very large number of nanoentities which is a very unlikely case. We therefore conclude that the number of edges is probably not an issue.

E. Project Definition

The following project definition is an excerpt of the original project definition which was signed at the beginning of the thesis project.

E.1 Context

The idea of splitting a monolithic landscape into smaller and manageable pieces is not new. Having been promoted e.g. in the Service Oriented Architecture (SOA) and its predecessors (OOAD, CBSE), it is being discussed under the banner of microservices nowadays. One challenge remains the same: How, and considering which criteria, do you split data and functions into manageable and maintainable pieces? The microservices community suggests Domain-Driven Design (DDD) to identify service boundaries.

With microservices, loose coupling and the single responsibility principle have received even more importance. Unnecessary coupling between microservices results in performance loss, development overhead and complex system landscapes that are hard to test and maintain. Engineers and architects are used to draw borders between components, but are generally doing this “as it feels right”. Coming from an object oriented programming, very often logical entities such as classes are bundled to packages or microservices. Our assumption is that there is a better, more sophisticated way to model data and functions in order to achieve loose coupling and domain decomposition.

E.2 Goals and Deliverables

The goal of the thesis project it to conceptualize and prototypically implement a system that allows a software architect to model data and logic and enhance it with a series of characterizations such as volatility, security, volumes, consistency and others. This data can then be used to suggest a set of services (or bounded contexts in DDD terminology) to the architect. We do not aim for full automation; the architect may or may not make use of these suggestions.

We identify four separate tasks to implement the described idea:

1. Research after which criteria data can be grouped into services to achieve loose coupling.
2. Definition of a format to model components, data and its characterizations.
3. Creation or evaluation of an algorithm to find one or more “optimized” suggestions.
4. Implementation of a system taking the defined format as an input to process it into one or multiple suggestions.

Every task can be done in a very simple or a more advanced way. The implementation for example, could be done as a simple command line tool and later on enlarged with a web user interface and a graph based database for data processing. The complexity of the tasks increase considerably with every new criterion taken into account.

Critical success factors for this bachelor thesis are the maturity and practical applicability of the developed concepts and their implementation, as well as their generality and extensibility. Algorithm and support system shall be applied to several sample applications (both public ones such as Pet Store and DDD Sample Application) so that effectiveness and efficiency of the developed approach and its value for the architect can be demonstrated.

Glossary

.NET .NET Framework is a software framework developed by Microsoft. 8

API Application Programming Interface. 7, 32, 34, 65, 69

BPM Business Process Management. 9

BPMN Business Process Model and Notation. 7

C C is a general-purpose computer programming language. 66

coupling criterion A architecturally significant requirement that impacts service decomposition. 1

CQRS Command Query Responsibility Segregation. 7

DDD Domain-Driven Design. 8, 9, 11, 36, 90

Entity A group of nanoentities which are part of a bigger concept sharing an identity and a common lifecycle.[12]. 29

ERM Entity-Relationship-Model. 9, 35, 36, 39, 45, 50

HSR Hochschule für Technik Rapperswil. 8, 101

InfoQ InfoQ is a online knowledge base on various programming topics. 5

JAR Java Archive. 40

JNI Java Native Interface. 66

JPA Java Persistence API. 54

JSON JavaScript Object Notation. 36, 37, 57, 58, 62

JVM Java Virtual Machine. 101

MCL Markov Cluster. 40

OOAD Object-Oriented Analysis and Design. 9

ORM Object-Relational Mapping. 69

PII Personally Identifiable Information. 86

RPC Remote Procedure Calls. 7

SOA Service Oriented Architecture. 8, 9, 70

SOMA Service-Oriented Modeling and Architecture. 9

system A system refers to a software application whose architecture needs to be decomposed.. 2

UI User Interface. 10, 40, 53, 93

UML Unified Modeling Language. 69

VM Virtual Machine. 52

YAML YAML is a recursive acronym and stands for “YAML Ain’t Markup Language”. YAML is a human-readable data serialization format. 104

References

Literature

[2] A. Arsanjani. “Service-oriented modeling and architecture”. In: *IBM developer works* (2004), pp. 1–15 (cit. on p. 9).

[3] A. Avram. *InfoQ - Udi Dahan on Defining Service Boundaries*. URL: http://www.infoq.com/news/2015/02/service-boundaries-healthcare (cit. on p. 22).

[4] K. Bastani. *Using Graph Analysis to Decompose Monoliths into Microservices with Neo4j*. URL: http://www.kennybastani.com/2015/05/graph-analysis-microservice-neo4j.html (visited on 2015-09-24) (cit. on p. 9).

[5] M. E. Conway. “How do committees invent”. In: *Datamation* 14.4 (1968), pp. 28–31 (cit. on p. 19).

[6] U. Dahan. *Finding Service Boundaries – illustrated in healthcare (NDC conference video: 1:05 - 1:07*. URL: https://vimeo.com/113515335 (visited on 2015-11-29) (cit. on p. 23).

[7] U. Dahan. *Logical and Physical Architecture*. URL: http://udidahan.com/2010/11/08/logical-and-physical-architecture/ (cit. on p. 70).

[8] U. Dahan. *The known unknowns of SOA*. URL: http://udidahan.com/2010/11/15/the-known-unknowns-of-soa/ (visited on 2015-10-22) (cit. on p. 10).

[9] J. Dietrich et al. “Cluster analysis of Java dependency graphs”. In: *Proceedings of the 4th ACM symposium on Software visualization*. ACM. 2008, pp. 91–94 (cit. on p. 9).

[10] S. van Dongen. “Graph Clustering by Flow Simulation”. PhD thesis. University of Utrecht, 2000. URL: http://micans.org/mcl/ (visited on 2015-12-01) (cit. on pp. 40, 103).

[11] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. Dog Ear Publishing, 2014 (cit. on p. 11).

[12] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Pearson Education, 2003 (cit. on pp. 8, 17, 22, 90, 109).

[13] M. Fowler. *Microservices.* URL: http://martinfowler.com/articles/microservices.html (visited on 2015-10-22) (cit. on p. 11).

[14] M. Fowler. “MonolithFirst”. In: (2010). URL: http://martinfowler.com/bliki/MonolithFirst.html (visited on 2015-12-17) (cit. on p. 60).

[15] M. Fowler. “Richardson Maturity Model: steps toward the glory of REST”. In: (2010). URL: http://martinfowler.com/articles/richardsonMaturityModel.html (visited on 2015-12-17) (cit. on p. 58).

[16] M. Fowler. *Service Definition by Martin Fowler in his article about Inversion of Control.* URL: http://www.martinfowler.com/articles/injection.html (visited on 2015-10-22) (cit. on p. 11).

[17] P. R. Halmos. *Naive set theory.* Springer Science & Business Media, 1960 (cit. on p. 77).

[18] J. A. Hartigan and M. A. Wong. “Algorithm AS 136: A k-means clustering algorithm”. In: *Applied statistics* (1979) (cit. on p. 103).

[19] E. Hartuv and R. Shamir. “A clustering algorithm based on graph connectivity”. In: *Information processing letters* 76.4 (2000) (cit. on p. 103).

[20] P. B. Kruchten. “The 4+ 1 view model of architecture”. In: *Software, IEEE* 12.6 (1995), pp. 42–50 (cit. on pp. 8, 10, 16).

[21] A. Lancichinetti and S. Fortunato. “Community detection algorithms: a comparative analysis”. In: *Phys. Rev. E 80* (2009). arXiv: 0908.1062v2 (cit. on pp. 67, 102).

[22] I. X. Y. Leung et al. “Towards real-time community detection in large networks”. In: *Phys. Rev. E 79* (2009). arXiv: 0808.2633 (cit. on pp. 40–42, 103).

[23] R .C. Martin. *Agile software development: principles, patterns, and practices.* Prentice Hall PTR, 2003 (cit. on pp. 14, 18, 19).

[24] J. D. Meier et al. *Design Guidelines for Application Performance.* URL: https://msdn.microsoft.com/en-us/library/ff647801.aspx (visited on 2015-11-29) (cit. on p. 20).

[25] M. E. Newman and M. Girvan. “Finding and evaluating community structure in networks”. In: *Phys. Rev. E 69* (2004). arXiv: cond-mat/0308217 (cit. on pp. 40, 41, 103).

[26] S. Newman. *Building Microservices.* " O’Reilly Media, Inc.", 2015 (cit. on pp. 7, 12).

[27] C. Pang et al. “Topological sorts on DAGs”. In: *Information Processing Letters* 115.2 (2015), pp. 298–301 (cit. on p. 35).

[28] D. L. Parnas. “On the criteria to be used in decomposing systems into modules”. In: *Communications of the ACM* 15.12 (1972), pp. 1053–1058 (cit. on pp. 5, 13, 19).

[29] U. N. Raghavan, R. Albert, and S. Kumara. “Near linear time algorithm to detect community structures in large-scale networks”. In: *Phys. Rev. E 76* (2007). arXiv: 0709.2938 (cit. on pp. 40–42, 103).

[30] C. Richardson. *Microservices: Decomposing Applications for Deployability and Scalability.* URL: http://www.infoq.com/articles/microservices-intro (visited on 2015-11-29) (cit. on p. 18).

[31] C. Richardson. “Microservices: Decomposing applications for deployability and scalability”. In: (2014) (cit. on p. 5).

[32] A. Rotem-Gal-Oz. *Services, Microservices, Nanoservices.* URL: http://arnon.me/2014/03/services-microservices-nanoservices/ (visited on 2015-10-28) (cit. on p. 26).

[33] N. Rozanski and E. Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives.* Pearson Education, 2011 (cit. on p. 21).

[34] S. E. Schaeffer. “Graph clustering”. In: *Computer Science Review* 1.1 (2007), pp. 27–64 (cit. on p. 102).

[35] W.P. Stevens, G.J. Myers, and L.L. Constantine. “Structured design”. In: *IBM Systems Journal* 13.2 (1974), pp. 115–139 (cit. on p. 13).

[36] “Systems and software engineering – Vocabulary, ©2011 IEEE”. In: *IEEE Std. 24765-2010* (2010) (cit. on p. 13).

[37] W. Vogels. *Eventually Consistent - Revisited.* URL: http://www.allthingsdistributed.com/2008/12/eventually_consistent.html (visited on 2015-11-29) (cit. on p. 20).

[38] A. Yakyma. *Why Progressive Estimation Scale Is So Efficient For Teams.* URL: http://www.yakyma.com/2012/05/why-progressive-estimation-scale-is-so.html (visited on 2015-12-01) (cit. on p. 46).

[39] O. Zimmermann. “Architectural decisions as reusable design assets”. In: *IEEE software* 1 (2011), pp. 64–69 (cit. on p. 71).

[40] O. Zimmermann. “Making Architectural Knowledge Sustainable, the Y-approach”. In: *IEEE Software talk from SATURN.* 2012. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=22132 (visited on 2015-11-25) (cit. on p. 71).

[41] O. Zimmermann, P. Krogdahl, and C. Gee. “Elements of service-oriented analysis and design”. In: *IBM developerworks* (2004) (cit. on p. 9).

[42] O. Zimmermann et al. “Architectural Decision Guidance Across Projects”. In: *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on.* IEEE. 2015, pp. 85–94 (cit. on pp. 28, 71).

Online Sources

[43] *AngularJS. A Java Script MVC framework.* URL: https://angularjs.org/ (visited on 2015-11-18) (cit. on pp. 32, 57).

[44] *Apache Spark. Lightning-fast cluster computing.* URL: http://spark.apache.org (visited on 2015-09-23) (cit. on p. 103).

[45] *Apache Tomcat Webserver.* URL: http://tomcat.apache.org/ (visited on 2015-12-16) (cit. on p. 58).

[46] *Apiacoa. Graph clustering and graph visualization.* URL: http://apiacoa.org/research/software/graph/index.en.html (visited on 2015-09-23) (cit. on p. 103).

[47] *Bootstrap. A front end framework based on HTML, CSS and Java Script.* URL: http://getbootstrap.com/ (visited on 2015-11-18) (cit. on pp. 32, 57).

[49] *DDDSample on Github.* URL: https://github.com/citerus/dddsample-core (visited on 2015-12-06) (cit. on pp. 28, 90).

[50] *DDDSample Screencast on Youtube.* URL: https://www.youtube.com/watch?v=eA8xgdtqqs8 (visited on 2015-12-06) (cit. on p. 90).

[51] *Docker. Docker is an open platform for building, shipping and running distributed applications.* URL: https://www.docker.com/ (visited on 2015-10-19) (cit. on pp. 31, 32, 57, 58).

[52] *Docker Compose. Docker Compose is a tool to define and run multi-container applications.* URL: https://docs.docker.com/compose/ (visited on 2015-11-19) (cit. on p. 58).

[54] *Enterprise Architect. A comprehensive UML analysis and design tool.* URL: http://www.sparxsystems.eu/start/home/ (visited on 2015-12-11) (cit. on p. 69).

[55] *Gephi. The Open Graph Viz Platform.* URL: http://gephi.github.io (visited on 2015-09-23) (cit. on pp. 79, 103).

[56] *Gephi Marketplace: Girvan Newman Clustering.* URL: https://marketplace.gephi.org/plugin/girvan-newman-clustering/ (visited on 2015-12-02) (cit. on pp. 69, 103).

[57] *Gephi Marketplace: Markov Cluster Algorithm (MCL).* URL: https://marketplace.gephi.org/plugin/markov-cluster-algorithm-mcl/ (visited on 2015-12-02) (cit. on pp. 40, 74).

[58] *Graphstream Project: Epidemic Label Propagation Algorithm.* URL: http://graphstream-project.org/ (visited on 2015-12-02) (cit. on pp. 40, 79, 103).

[59] *Hibernate ORM. Domain model persistence for relational databases.* URL: https://hibernate.org/ (visited on 2015-11-19) (cit. on pp. 54, 58).

[60] *JHipster. A framework to implement Java based web applications based on Spring and AngularJS.* URL: http://jhipster.github.io/ (visited on 2015-11-18) (cit. on pp. 32, 56, 57).

[61] *JSON Schema.* URL: http://json-schema.org/ (visited on 2015-12-16) (cit. on p. 57).

[62] *Jung. Java Universal Network/Graph Framework.* URL: http://jung.sourceforge.net (visited on 2015-09-23) (cit. on p. 103).

[63] *Liquibase. A database-independent library to manage and apply database schema changes using XML files.* URL: http://www.liquibase.org/ (visited on 2015-11-19) (cit. on p. 58).

[64] *Neo4j Graph Gists are an easy way to create and share your documents containing example graph models and use-cases.* URL: http://neo4j.com/developer/graphgist/ (visited on 2015-12-13) (cit. on p. 9).

[65] *NServiceBus. A service bus for .NET.* URL: http://particular.net/nservicebus (visited on 2015-12-12) (cit. on p. 8).

[66] *PostgreSQL. An open-source object-relational database management system.* URL: http://www.postgresql.org/ (visited on 2015-11-19) (cit. on p. 58).

[67] *Service Cutter Wiki.* URL: https://github.com/ServiceCutter/ServiceCutter/wiki (visited on 2015-12-16) (cit. on p. 37).

[74] *Simple Logging Facade for Java (SLF4J).* URL: http://www.slf4j.org/ (visited on 2015-04-12) (cit. on p. 31).

[68] *Spring Boot is a framework to build Spring applications.* URL: http://projects.spring.io/spring-boot/ (visited on 2015-10-12) (cit. on pp. 32, 56).

[70] *Spring Framework.* URL: https://spring.io/ (visited on 2015-12-07) (cit. on p. 32).

[71] *Spring Security Project.* URL: http://projects.spring.io/spring-security/ (visited on 2015-12-16) (cit. on p. 59).

[72] *Stackoverflow - Graph Clustering Library in Java.* URL: http://stackoverflow.com/questions/32627479/graph-clustering-library-in-java/32629606#32629606 (visited on 2015-09-23) (cit. on p. 102).

[73] *Swagger.io - Swagger is a specification and complete web application framework implementation for describing, producing, consuming, and visualizing RESTful web APIs.* URL: http://swagger.io/ (visited on 2015-10-23) (cit. on p. 69).

[75] *UnpackNBM used for Gephi Toolkit Plugins.* URL: https://github.com/mbastian/UnpackNBM (visited on 2015-12-07) (cit. on p. 40).

[76] *Vis.js. A browser based visualization library.* URL: http://visjs.org/ (visited on 2015-12-18) (cit. on p. 57).