FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Designing a business platform using microservices.

Rajendra Kharbuja

FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Designing a business platform using microservices.

Entwerfen einer Business-Plattform mit microservices.

Author:	Rajendra Kharbuja
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Manoj Mahabaleshwar
Submission Date:	

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, Rajendra Kharbuja

Acknowledgments

Abstract

Contents

1 Context

1.1 Monolith Architecture Style

A Mononlith Architecture Style is the one in which an application is deployed as a single artifact. The architecture inside the application can be modular and clean. In order to clarify, the figure 1.1 shows architecture of an Online-Store application. The application has clear separation of components such as Catalog, Order and Service as well as respective models such as Product, Order etc. Despite of that, all the units of the application are deployed in tomcat as a single war file.[Ric14a][Ric14c]

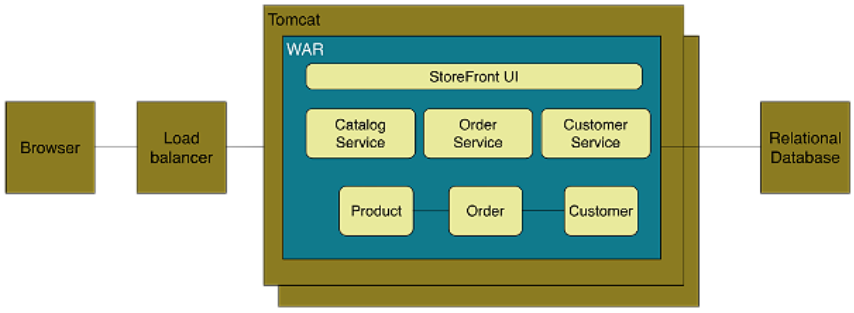

Figure 1.1: Monolith Example from [Ric14a]

1.1.1 Types of Monolith Architecture Style

According to [Ann14], a monolith can be of several types depending upon the viewpoint, as shown below:

1. Module Monolith: If all the code to realize an application share the same codebase and need to be compiled together to create a single artifact for the whole application then the architecture is Module Monolith Architecture. An example is show in figure 1.2. The application on the left has all the code in the same codebase in the form of packages and classes without clear definition of modules and get compiled to a single artifact. However, the application on the right is

developed by a number of modular codebase, each has separate codebase and can be compiled to different artifact. The modules uses the produced artifacts which is different than the earlier case where the code referenced each other directly.

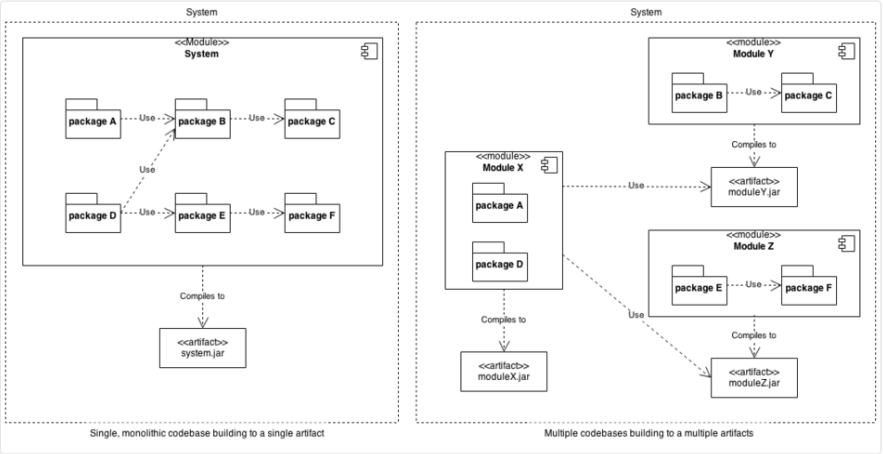

Figure 1.2: Module Monolith Example from [Ann14]

2. Allocation Monolith: An Allocation Monolith is created when all code is deployed to all the servers as a single version. This means that all the components running on the servers have the same versions at any time. The figure 1.3 gives an example of allocation monolith. The system on the left have same version of artifact for all the components on all the servers. It does not make any differenct whether or not the system has single codebase and artifact. However, the system on the right as shown in the figure is realized with multiple version of the artifacts in different servers at any time.

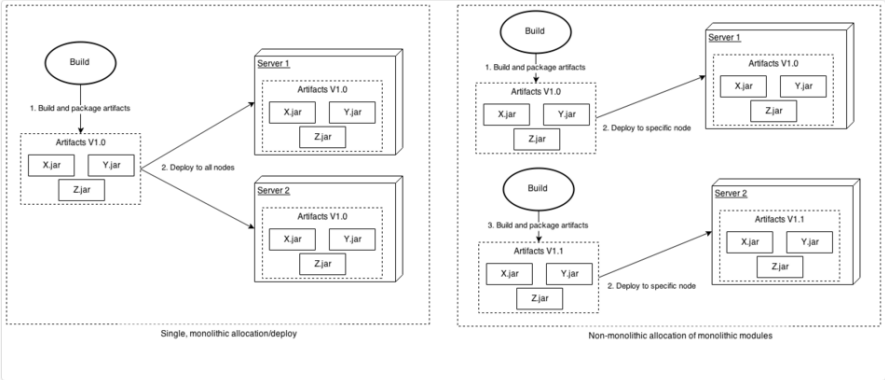

Figure 1.3: Allocation Monolith Example from [Ann14]

3. Runtime Monolith: In Runtime Monolith, the whole application is run under a single process. The left system in the figure 1.4 shows an example of runtime monolith where a single server process is responsible for whole application. Whereas the system on the left has allocated multiple server process to run distinct set of component artifacts of the application.

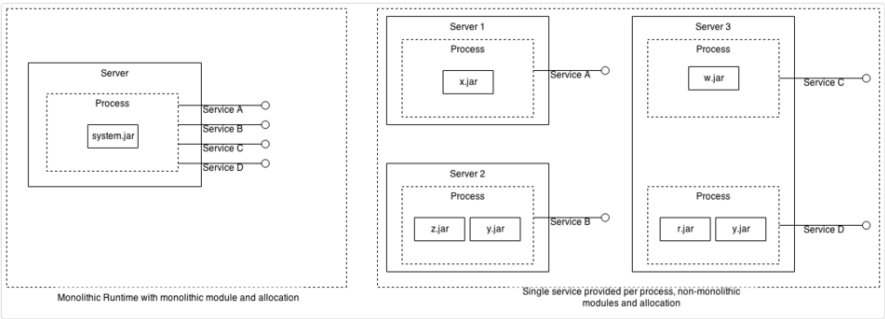

Figure 1.4: Runtime Monolith Example from [Ann14]

1.1.2 Advantages of Monolith Architecture Style

The Monolith architecture is appropriate for small application and has following benifits:[Ric14c][FL14][Gup15][Abr14]

- It is easy to develop a monolith application since various development tools including IDEs are created around the single application concept. Nevertheless,

it is also easy to test the application by creating appropriate environment on the developer's machine.

- The deployment can be simply achieved by moving the single artifact for the application to an appropriate directory in the server.
- The scaling can be clearly and easily done by replicating the application horizontally across multiple servers behind a load balancer as shown in figure 1.1
- The different teams are working on the same codebase so sharing the functionality can be easier.

1.1.3 Disadvantages of Monolith Architecture Style

As the requirement grows with time, alongside as application becomes huge and the size of team increases, the monolith architecture faces many problems. Most of the advantages of monolith architecture for small application will not be valid anymore. The challenges of monolith architecture for such agile and huge context are as given below:[NS14][New15][Abr14][Ric14a][Ric14c][Gup15]

- Limited Agility: As the whole application has single codebase, even changing a small feature to release it in production takes time. Firstly, the small change can also trigger changes to other dependent code because in huge monolith application it is very difficult to manage modularity especially when all the team members are working on the same codebase. Secondly, to deploy a small change in production, the whole application has to be deployed. Thus continuous delivery gets slower in case of monolith application. This will be more problematic when multiple changes have to be released on a daily basis. The slow pace and frequency of release will highly affect agility.

- Decrease in Productivity: It is difficult to understand the application especially for a new developer because of the size. Although it also depends upon the structure of the codebase, it will still be difficult to grasp the significance of the code when there is no hard modular boundary.Additionally, the developer can be intimidated due to need to see the whole application at once from outwards to inwards direction. Secondly, the development environment can be slow to load the whole application and at the same time the deployment will also be slow. So, in overal it will slow down the speed of understandability, execution and testing.

- Difficult Team Structure: The division of team as well as assigning tasks to the team can be tricky. Most common ways to partition teams in monolith are by technology and by geography. However, each one cannot be used in all the situations. In any case, the communication among the teams can be difficult and slow. Additionally, it is not easy to assign vertical ownership to a team from particular feature from development to relase. If something goes wrong in the deployment, there is always a confusion who should find the problem, either operations team or the last person to commit. The approprate team structure and ownership are very important for agility.

- Longterm Commitment to Technology stack: The technology to use is chosen before the development phase by analysing the requirements and the maturity of current technology at that time. All the teams in the architecture need to follow the same techonology stack. However, if the requirement changes then there can be situation when the features can be best soloved by different sets of technology. Additionally, not all the features in the application are same so cannot be treated accordingly in terms of technology as well. Nevertheless, the technology advances rapidly. So, the solution thought at the time of planning can be outdated and there can be a better solution available. In monolith application, it is very difficult to migrate to new technology stack and it can be rather painfull process.

- Limited Scalability: The scalability of monolith application can be done in either of two ways. The first way is to replicate the application along many servers and dividing the incoming request using a load balancer in front of the servers. Another approach is using the identical copies of the application in multiple servers as in previous case but partitioning the database access instead of user request. Both of these scaling approaches improves the capacity and availability of the application. However, the individual requirement regarding scaling for each component can be different but cannot be fulfilled with this approach. Also, the complexity of the monolith application remains the because we are replicating the whole application. Additionally, if there is a problem in a component the same problem can affect all the servers running the copies of the application and does not improve resilency.[Mac14][NS14]

1.2 Decomposition of Application

The section 1.1.3 specified various disadvantages related to monolith architecture style. The book [FA15] provides a way to solve most of the discussed problems such as agility, scalability, productivity etc. It provides three dimensions of scalability as shown in figure 1.5 which can be applied alone or simultaneously depending upon the situation and desired goals.

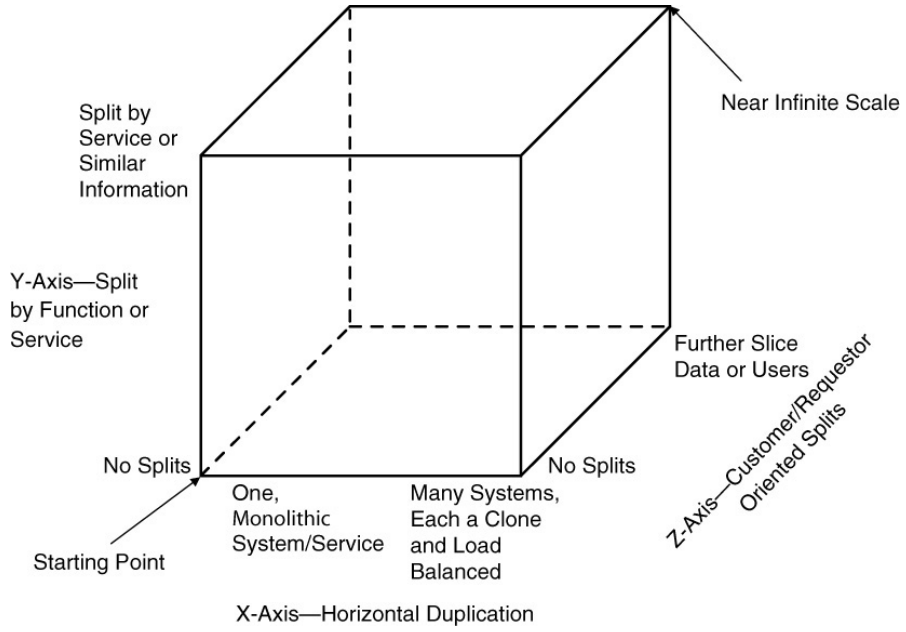

Figure 1.5: Scale Cube from [FA15]

The scaling along each dimensions are described below. [FA15][Mac14][Ric14a]

1. X-axis Scaling: It is done by cloning the application and data along multiple servers. A pool of requests are applied into a load balancer and the requests are deligated to any of the servers. Each of the server has the full capability of the application and full access to all the data required so in this respect it does not make any difference which server fulfills the request. Rather, it is about how many requests are fulfilled at any time. It is easy to scale along X-axis as the number of requests increases. The solution is as simple as to add additional clones. However, with this type of scaling, it is not scale with the increase in data. Moreover, it also does not scale when there are large variation in the frequency of any type of requests or there dominant requests types because all the requests

are handled in an unbiased way and allocated to servers in the same way.

2. Z-axis Scaling: The scaling is done by spliting the request based on certain critera or information regarding the requestor or customer affected by the request. It is different than X-axis scaling in the way that the servers are responsible for different kinds of requests. Normally, the servers have same copy of the application but some can have additional functionalies depending upon the requests expected. The Z-axis scaling helps in fault isolation and transaction scalability. Using this scaling, certain group of customers can be given added functionality or a new functionality can be tested to a small group and thus minimizing the risk.

3. Y-axis Scaling: The scaling along this dimension means the spliting of the application responsibility. The separation can be done either by data, by the actions performed on the data or by combination of both. The respective ways can be referred to as resoure-oriented or service-oriented splits. While the x-axis or z-axis split were rather duplication of work along servers, the y-axis is more about specialization of work along servers. The major advantage of this scaling is that each request is scaled and handled differently according to its necessity. As the logic along with the data to be worked on are separated, developers can focus and work on small section at a time. This will increase productivity as well as agility. Additionally, a fault on a component is isolated and can be handled gracefully without affecting rest of the application. However, scaling along Y-axis can be costly compared to scaling along other dimensions.

1.3 Microservice Architecture Style

The section 1.1.3 listed various disadvantages of following monolithic architecture style especially when the application grows in size rapidly. To counteract those issues, the section 1.2 proposed a way of using scale-cube to decompose the application into individual features, each feature being scaled individually. The same approach is utilized by Microservice Architecture Style, in which an application is decomposed into various individual components, each component runs in a separate process and can be deployed as well as scaled individually.
There are several definitions given by several pioneers and early adapters of the style.

Definition 1: [Ric14b]

"It is the way to functionally decompose an application into a set of collaborating services, each with a set of narrow, related functions, developed and deployed independently, with its own database."

Definition 2: [Woo14]

"It is a style of software architecture that involves delivering systems as a set of very small, granular, independent collaborating services."

Definition 3: [Coc15]

"Microservice is a loosely coupled Service-Oriented Architecture with bounded contexts."

Definition 4: [FL14][RTS15]

"Microservices are Service-Oriented Architecture done right."

Definition 5: [FL14]

" Microservice architecture 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. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."

The authors have their own way of interpretation of microservices but at the same time agree upon some basic concepts regarding the architecture. However, each definition can be used to understand different aspect of the microservices. A distinct set of keywords can be identified which represents different aspects pointed by the

authors and various concepts they agree. Moreover, the table lists important keywords. These keywords provide us some hint regarding various topics of discussion related to microservices, which are listed in columns. Finally, understanding these keywords can be one of the approaches to understand microservices architecture and answer various questions.

#	keywords	size	Quality of good microservice	communication	process to create microservices
1	Collaborating Services			✓	
2	Communicating with lightweight mechanism like http			✓	
3	Loosely coupled, related functions		✓	✓	
4	Developed and deployed independently				✓
5	Own database		✓		✓
6	Different database technologies				✓
7	Service Oriented Architecture		✓		✓
8	Bounded Context	✓	✓		✓
9	Build around Business Capabilities	✓	✓		✓
10	Different Programming Languages				✓

Table 1.1: Keywords extracted from various definitions of Microservice

2 Granularity

2.1 Introduction

The granularity of a service is often ambiguous and has different interpretation. In simple term, it refers to the size of the service. However, the size itself can be vague. It can neither be defined as a single quantitative value nor it can be defined in terms of single dependent criterion. It is difficult to define granularity in terms of number because the concepts defining granularity are vague and subjective in nature. If we choose an activity supported by the service to determine its granularity then we cannot have one fixed value instead a hierarchical list of answers; where an activity can either refer to a simple state change, any action performed by an actor or a complete business process. [Linnp], [Hae+np]
Although, the interest upon the granularity of a component or service for the business users only depends upon their business value, there is no doubt that the granularity affects the architecture of a system. The honest granularity of a service should reflect upon both business perspective and should also consider the impact upon the overall architecture.
If we consider other units of software application, we come from object oriented to component based and then to service oriented development. Such a transistion has been considered with the increase in size of the individual unit. The increase in size is contributed by the interpretation or the choice of the abstraction used. For example, in case of object oriented paradigm, the abstraction is chosen to represent close impression of real world objects, each unit representing fine grained abstraction with some attributes and functionalities.
Such abstraction is a good approach towards development simplicity and understanding, however it is not sufficient when high order business goals have to be implemented. It indicates the necessity of coarser-grained units than units of object oriented paradigm. Moreover, component based development introduced the concept of business components which target the business problems and are coarser grained. The services provide access to application where each application is composed of various component services. [Linnp]

2.2 Related Work

A number of researches have been done in the area of service and component granularity. The focus of the researches includes various areas such as definition, effect, its dependent criteria and various measurement metrics.
According to [Pad04], the granularity of a component is inversely related to various non functional qualities such as customization and maintainaibility.
According to Hazen and Sims [HS00], component granularity is rather recursive as a component can be composed of various fine-grained components. They further classify recursion as of discrete or continuous nature but then prefer on discrete recursion. Thus, component granularity can be divide into three discrete layers: system, business and distributed where the composition happens from right to left order and granularity decreases in the opposite order. It is also useful to relate granularity with reusability. Coarse-grained components have high reuse efficiency because they are well focused to a specific business functionality. Whereas fine-grained components have high reusability since they focus on small functionality and thus can be used by other coarse-grained components to accomplish higher level business capabilities. [Mil+01], [WXZ05]
Additionally, Sims [Sim05] gave some insights to measure granularity. The granularity can be measured either a) by the number of components called from an operation on a service interface, b) by the number of components calling the service interface or c) by the number of database tables updated.
Service Oriented Architecture Framework (SOAF [EAK06] also gives some way to measure the granularity in terms of quantitative value. The value can be derived from a) the number of components that are invoked by the service interface and b) the number of changes of states in resources like the database tables updates influenced by the service. Again, the granularity can affect reusability and the network communication to accomplish any task. Coarse grained components have more volume of data exchanged where as fine grained components have high frequency of round trips to complete the full business functionality. It is recommended to balance them which can depend upon various factors such as abstraction level, change expectation rate, service complexity, desired coupling and cohesion. Furthermore, the granularity can be at different levels within an application. High-level business process can be realized as a coarse-grained services and each coarse-grained component can be composed of various fine-grained services.
The idea of multi-layered granularity is further supported by Security Trading case study [EAK06] and International Financial and Brokerage Services IFBS case study [Nav08]. The design of pilot application for Security Trading consists of various services across various levels such as process, application, shared data and infrastructures and containing services with granularity decreasing from left to the right in the order they

are listed. The application for IFBS is refactored from large number of fine-grained service to multi-layered granular services such as process services, business services, composite services or orchestration services, utility services etc.
A research performed by Steghuis [Ste06] talks about various aspects of granularity such as functionality, flexibility, reusability, complexity, context-independence, Performance, Genericity and sourcing. Some differenciating aspects in comparision to previous researches are context-independence, genericity and sourcing. Context-independence means independence and self sufficiency in terms of data as well as logic. It promotes loose coupling and insists lack of knowledge of state of surrounding and inessential maintainance of own state. Genericity is the quality of making generic services which can be used in different situations such as messaging service. Finally, sourcing is the process of identifying cohesive group of tasks which can be outsourced by considering the coupling associated.

2.3 Basic Principles to Service Granularity

In addition to quantitative perspective on granularity, it can be helpful to approach the granularity qualitatively. The points below are some basic principls derived from various scientific and research papers, which attempt to define the qualitative properties of granularity. Hopefully, these principles will be helpful to come with the service of right granularity.

1. The correct granularity of a component or a service is dependent upon the time. The various supporting technologies that evolve during time can also be an important factor to define the level of vertical decomposition. For eg: with the improvement in virtualization, containerization as well as platform as a service technologies, it is fast and easy for deployment automation which supports multiple fine-grained services creation.[HS00]

2. A good candidate for a service should be independent upon the implementation but should depend upon the understandability of domain experts.

   [Hae+np; HS00]

3. A service should be an autonomous reusable component and should support various cohesion such as functional (group similar functions), temporal(change in the service should not affect other services), run-time(allocate similar runtime environment for similar jobs; eg. provide same address space for jobs of similar computing intensity) and actor (a component should provide service to similar users). [Hae+np], [HS00]

4. A service should not support huge number of operations. If it happens, it will affect high number of customers on any change and there will be no unified view on the functionality. Furthermore, if the interface of the service is small, it will be easy to maintain and understand.[Hae+np], [RS07]

5. A service should provide transaction integrity and compensation. The activities supported by a service should be within the scope of one transaction. Additionally, the compensation should be provided when the transaction fails. If each operation provided by the service map to one transaction, then it will improve availability and fault-recovery. [Hae+np], [Foo05] [BKM07]

6. The notion of right granularity is more important than that of fine or coarse. It depends upon the usage condition and moreover is about balancing various qualities such as reusability, network usage, completeness of context etc. [Hae+np], [WV04]

7. The level of abstraction of the services should reflect the real world business activities. Doing so will help to map business requirements and technical capabilities. [RS07]

8. If there are ordering dependencies between the operations, it will be easy to implement and test if the dependent operations are all combined into a single service. [BKM07]

9. There can be two better approaches for breaking down an abstraction. One way is to separate redundant data or data with different semanting meaning. The other approach is to divide services with limited control and scope. For example: A Customer Enrollment service which deals with registration of new customers and assignment of unique customer ID can be divided into two independent fine-grained services: Customer Registration and ID Generation, each service will have limited scope and separate context of Customer.[RS07]

10. If there are functionalities provided by a service which are more likely to change than other functionalities. It is better to separate the functionality into a fine-grained service so that any further change on the functionality will affect only limited number of consumers. [BKM07]

2.4 Dimensions of Granularity

As already mentioned in 2.1, it is not easy to define granularity of a service quantitatively. However, it can be made easier to visualize granularity if we can project it

along various dimensions, where each dimension is a qualifying attribute responsible for illustrating size of a service. Eventhough the dimensions discussed in this section will not give the precise quantity to identify granularity, it will definitely give the hint to locate the service in granularity space. It will be possible to compare the granularity of two distinct services. Moreover, it will be interesting and beneficial to know how these dimensions relate to each other to define the size.

2.4.1 Dimension by Interface

One way to define granularity is by the perception of the service interface as made by its consumer. The various properties of a service interface responsible to define its size are listed below.

1. Functionality: It qualifies the amount of functionality offered by the service. The functionality can be either default functionality, which means some basic group of logic or operation provided in every case. Or the functionality can be parameterized and depending upon some values, it can be optionally provided. Depending upon the functionality volume, the service can be either fine-grained or course-grained than other service. Considering functionality criterion, A service offering basic CRUD functionality is fine-grained than a service which is offers some accumulated data using orchestration. [Hae+np]

2. Data: It refers to the amount of data handled or exchanged by the service. The data granularity can be of two types. The first one is input data granularity, which is the amount of data consumed or needed by the service in order to accomplish its tasks. And the other one is ouput data granularity, which is the amount of data returned by the service to its consumer. Depending upon the size and quantity of business object/objects consumed or returned by the service interface, it can be coarse or fine grained. Additionally, if the business object consumed is composed of other objects rather than primitive types, then it is coarser-grained. For example: the endpoint "PUT customers/C1234" is coarse-grained than the endpoint "PUT customers/C1234/Addresses/default" because of the size of data object expected by the service interface. [Hae+np]

3. Business Value: According to [RKKnp], each service is associated with an intention or business goal and follows some strategy to achiee that goal. The extent or magnitude of the intention can be perceived as a metric to define granularity. A service can be either atomic or aggregate of other services which depends

upon the level of composition directly influenced by the extent of target business goal. An atomic service will have lower granularity than an aggregate in terms of business value. For example, sellProduct is coase-grained than acceptPayment, which is again coarse-grained than validateAccountNumber. [Hae+np]

2.4.2 Dimension by Interface Realization

The section 2.4.1 provides the aspect of granularity with respect to the perception of customer to interface. However, there can be different opinion regarding the same properties, when it is viewed regarding the imlementation. This section takes the same aspects of granularity and try to analyze them when the services are implemented.

1. Functionality: In section 2.4.1, it was mentioned that an orchestration service has higher granularity than its constituent services with regard to default functionality. If the realization effort is focused, it may only include compositional and/or compensation logic because the individual tasks are acomplshed by the constituent service interfaces. Thus, in terms of the effort in realizing the service it is fine-grained than from the view point of interface by consumer. [Hae+np]

2. Data: In some cases, services may utilize standard message format. For example: financial services may use SWIFT for exchanging finanicial messages. These messages are extensible and are coarse grained in itself. However, all the data accepted by the service along with the message may not be required and used in order to fulfil the business goal of the service. In that sense, the service is coarse-grained from the viewpoint of consumer but is fine-grained in realization point of view. [Hae+np]

3. Business Value: It can make a huge impact when analyzing business value of service if the realization of the service is not considered well. For example: if we consider data management interface which supports storage, retrieval and transaction of data, it can be considered as fine-grained because it will not directly impact the business goals. However, since other services are very dependent in its performance, it becomes necessary to analyse the complexity associated with the implementation, reliablity and change the infrastructure if needed. These comes with additional costs, which should well be analyzed. From the realization point of view, the data service is coarse-grained than its view by consumer. [Hae+np]

Principle: A high Functionality granularity does not necessarily mean high business value granularity
It may seem to have direct proportional relationship between functionality and business value granularity. The business value of a service reflects business goals however the functionality refers to the amount of work done by the service. A service to show customer history can be considered to have high functionality granularity because of the involved time period and database query. However, it is of very low business value to the enterprise. [Hae+np]

2.4.3 R^3 Dimension

Keen [Kee91] and later Weill and Broadbent [WB98] introduced a separate group of criteria to measure granularity of service. The granularity was evaluated in terms of two dimensions as shown in the Figure 2.1

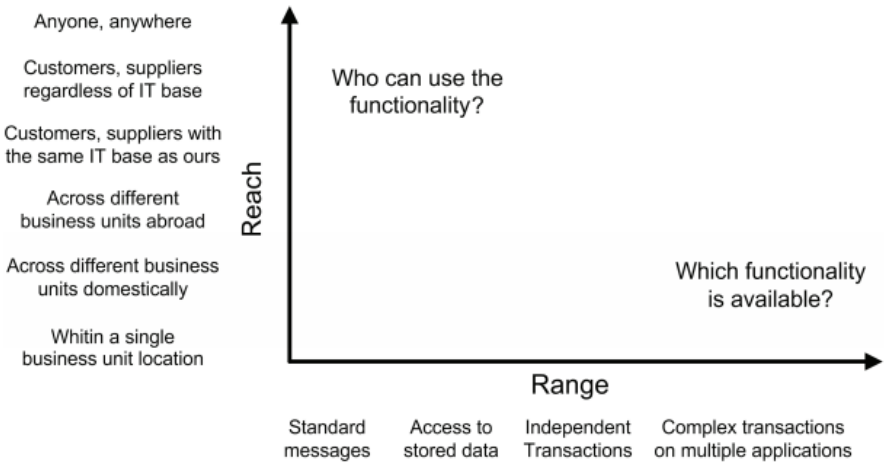

Figure 2.1: Reach and Range model from [Kee91; WB98]

Reach: It provides various levels to answer the question "Who can use the functionality? ". It provides the extent of consumers, who can access the functionality provide by the service. It can be a customer, the customers within an organization, supplier etc. as given by the Figure 2.1.
Range: It gives answer to "Which functionality is available? ". It shows the extent to which the information can be accessed from or shared with the service. The levels of

information accessed are analyzed depending upon the kind of business activities accessible from the service. It can be a simple data access, a transaction, message transfer etc as shown in the Figure 2.1 The 'Range' measures the amount of data exchanged in terms of the levels of business activities important for the organization. One example for such levels of activities with varying level of 'Range' is shown in Figure 2.2.

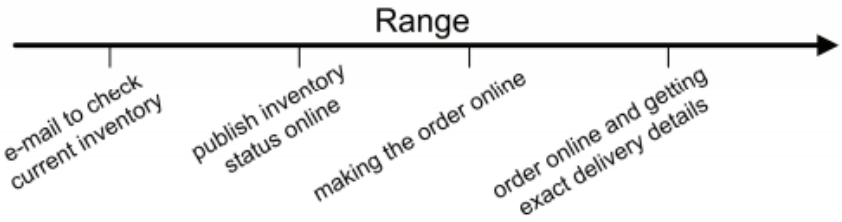

Figure 2.2: Example to show varying Range from [Kee91; WB98]

In the figure 2.2, the level of granularity increases as the functionality moves from 'accessing e-mail message' to 'publishing status online' and then to 'creating order'. It is due to the change in the amount of data access involved in each kind of functionality. Thus, the 'Range' directly depends upon the range of data access.
As the service grows, alongside 'Reach' and 'Range' also peaks up, which means the extent of consumers as well as the kind of functionality increase. This will add complexity in the service. The solution proposed by [Coc01] is to divide the architecture into services. However, only 'Reach' and 'Range' will not be enough to define the service. It will be equally important to determine scope of the individual services. The functionality of the service then should be define in two distinct dimensions 'which kind of functionality' and 'how much functionality'. This leads to another dimension of service as described below. [Kee91; WB98; RS07]
Realm: It tries to create a boundary around the scope of the functionality provided by the service and thus clarifies the ambiguity created by 'Range'. If we take the same example as given by Figure **??**, the range alone defining the kind of functionaly such as creating online order does not explicitly clarifies about what kind of order is under consideration. The order can be customer order or sales order. The specification of 'Realm' defining what kind of order plays role here. So, we can have two different services each with same 'Reach' and 'Range' however different 'Realm' for customer order and Sales order. [Kee91; WB98; RS07]

The consideration of all aspects of a service including 'Reach', 'Range' and 'Realm' give us a model to define granularity of a service and is called R^3 model. The volume in the R^3 space for a service gives its granularity. A coarse-grained service has higher R^3

volume then fine-grained service. The figures **??** and 2.4 show such volume-granularity analogy given by R^3 model. [Kee91; WB98; RS07]

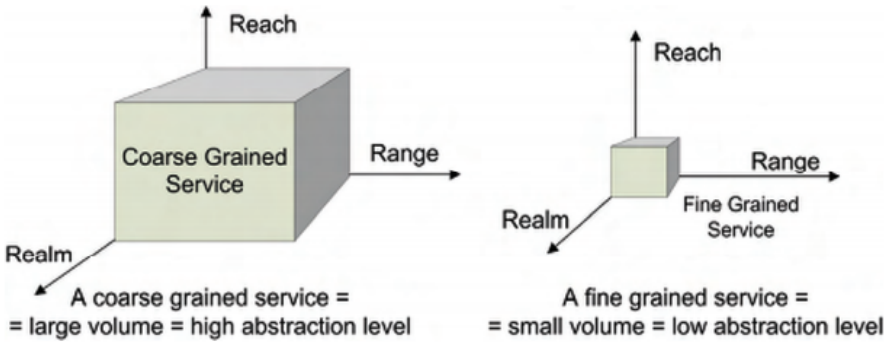

Figure 2.3: R^3 Volume-Granularity Analogy to show direct dependence of granularity and volume [RS07]

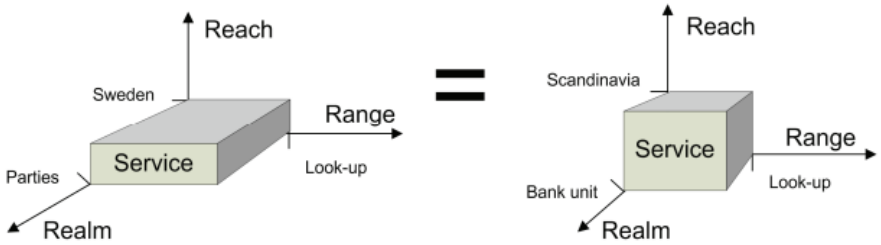

Figure 2.4: R^3 Volume-Granularity Analogy to show same granularity with different dimension along axes [RS07]

2.4.4 Retrospective

The sections 2.4.1 and 2.4.2 classified granularity of a service along three different directions: data, functionality and business value. Moreover, the interpretation from the viewpoint of interface by consumer and the viewpoint of producer for realization were also made. Again, the section 2.4.3 divides the aspect of granularity along Reach, Range and Realm space, the granularity given by the volume in the R^3 space. Despite of good explanation regarding each aspect, the classification along data, functionality and business value do not provide discrete metrics to define granularity in terms

of quantitatively. However, the given explanations and criteria are well rounded to compare two different services regarding granularity. These can be effectively used to classify if a service is fine-grained than other service. The R^3 model in section 2.4.3 additionally provides various levels of granularity along each axis. This makes it easy to at least measure the granularity and provides flexibility to compare the granularity between various services.
A case study [RS07] using data, functionality and business criteria to evaluate the impact of granularity on the complexity in the architecture is held. The study was done at KBC Bank Insurance Group of central Europe. Along the study, the impact of each kind of granularity is verified. The important results fromt the study are listed below.

1. A coarse-grained service in terms of 'Input Data' provides better transactional support, little communication overhead and also helps in scalability. However, there is a chance of data getting out-of-date quickly.

2. A coarse-grained service in terms of 'Output Data' also provides good communication efficiency and supports reusability.

3. A fine-grained service along 'Default Functionality' has high reusability and stability but low reuse efficiency.

4. A coarse-grained service due to more optional functionalities has high reuse efficiency, high reusability and also high stability but the implementation or realization is complicated.

5. A coarse-grained service along 'Business Value' has high consumer satisfaction value and emphasize the architecturally crucial points fulfilling business goals.

Similarly, a study was carried out in Handelsbanken in sweden, which has been working with Service-Oriented architecture. The purpose of the study was to analyze the impact of R^3 model on the architecture. It is observed that there are different categories of functionalities essential for an organization. Some functionality can have high Reach and Range with single functionality and other may have complex functionality with low Reach and Range. The categorization and realization of such functionalities should be accessed individually. It is not necessary to have same level of granularity across all services or there is no one right volume for all services. However, if there are many services with low volume, then we will need many services to accomplish a high level functionality. Additionally, it will increase interdependency among services and network complexity. Finally, the choice of granularity is completely dependent upon the IT-infrastructure of the company. The IT infrastructure decides which level of granularity it can support and how many of them. [RS07]

Principle: IT-infrastructure of an organization affects granularity.
The right value of granularity for an organization is highly influenced by its IT infrastructure. The organization should be capable of handling the complexities such as communication, runtime, infrastructure etc if they choose low granularity. [RS07]

Additionally, it is observed that a single service can be divided into number of granular services by dividing across either Reach, Range or Realm. The basic idea is to make the volume as low as possible as long as it can be supported by IT-infrastructure. Similarly, each dimension is not completely independent from other. For example: If the reach of a service has to be increased from domestic to global in an organization then the realm of the functionality has to be decreased in order to make the volume as low as possible, keep the development and runtime complexities in check. [RS07]

Principle: Keep the volume low if possible.
If supported by IT-infrastructure, it is recommended to keep the volume of service as low as possible, which can be achieved by managing the values of Reach, Range and Realm. It will help to decrease development and maintenance overload. [RS07]

%comment from Andrea

3 Quality of Service

3.1 Introduction

In addition to allign with the business requirements, an important goal of software engineering is to provide high quality. The quality assessment in the context of service oriented product becomes more crucial as the complexity of the system is getting higher by time. [ZL09; GL11; Nem+14] Quality models have been devised over time to evaluate quality of a software. A quality model is defined by quality attributes and quality metrics. Quality attributes are the expected properties of software artifacts defined by the chosen quality model. And, quality metrics give the techniques to measure the quality attributes. [Man+np] The software quality attributes can again be categorized into two types: internal and external attributes. [Man+np; BMB96] The internal quality attributes are the design time attributes which should be fulfilled during the design of the software. Some of the external quality attributes are loose coupling, cohesion, granularity, autonomy etc.[Ros+11; SSPnp; EM14] On the other hand, the external quality attributes are the traits of the software artifacts produced at the end of Software Development Life Cycle Some of them are reusability, maintainability etc. [EM14; Man+np; FL07; Feu13] For that reason, the external quality attributes can only be measure after the end of development. However, it has been evident that internal quality attributes have huge impact upon the value of external quality attributes and thus can be used to predict them. [HS90; Bri+np; AL03; Shi+08]The evaluation of both internal and external quality attributes are valuable in order to produce high quality software.[Man+np; PRF07; Per+07]

3.2 Quality Attributes

As already mentioned in section 3.1, the internal and external qualities determine the overall value of the service composed. There are different researches and studies which have been performed to find the features affecting the service qualities. Based on the published research papers, a comprehensive table 3.1 has been created. The table provides a minimum list of quality attributes which have been considerd in various research papers.

#	Attribute	[SSPnp]	[Xianp]	[AZR11]	[Shi+08]	[Ma+09]	[FL07]
1	Coupling	✓	✓	✓	✓	✓	✓
2	Cohesion	✓	X	✓	✓	✓	✓
3	Autonomy	✓	X	X	X	X	X
4	Granularity	✓	✓	✓	✓	✓	✓
5	Reusability	✓	X	X	✓	X	✓
6	Abstraction	✓	X	X	X	X	X
7	Complexity	X	X	X	✓	X	X

Table 3.1: Quality Attributes

The table 3.1 gives a picture of the studies done so far around service quality attributes. It can be deduced that most of the papers focus on coupling and granularity of the service and only few of them focus on other attributes such as complexity and autonomy.

Coupling refers to the dependency and interaction among services. The interaction becomes inevitable when a service requires a functionality provided by another service in order to accomplish its own goals. Similarly, Cohesion of any system is the extent of appropriate arrangement of elements to perform the functionalities. It affects the degree of understandability and sensibility of its interface for its consumers. Whereas, the complexity attribute provides the way to evaluate the difficulty in understanding and reasoning the context of the service or component.[EM14]
The reusability attribute for a service measures the degree to which it can be used by multiple consumer services and can undergo multiple composition to accoplish various high order functionalities. [FL07]
Autonomy is a broad term which refers to the ability of a service for self-containment, self-controlling, self-governance. Autonomy defines the boundary and context of the service. [MLS07] Finally, granularity is described in chapter 2 in detail. The basic signifance of granularity is the diversity and size of the functionalities offered by the service. [EM14]

3.3 Quality Metrics

There have been many studies made to define metrics for quality components.A major portion of the research have been done for object oriented and component based development. These metrics are refined to be used in service oriented systems.[Xianp; SSPnp] Firslty, various papers related to quality metrics of service oriented systems are collected. The papers which conducted their findings or proposition based on quality

amount of scientific studies and have produced convincing result are only selected. Additionally, the evaluation presented in the research papers were only done using the case studies of their own and not real life scenarios. Furthermore, they have used diversity of equations to define the quality metrics. On the basis of case studies made in the papers only, a fair comparision and choice of only one way to evaluate the quality metrics cannot be made. This is further added by the fact that the few papers cover most of the quality metrics, most of them focus on only few quality metrics such as coupling but not all of them focus on all the quality metrics. Nevertheless, they do not discuss about the relationship between all quality metrics. [EM14] This creates difficulty to map all the quality metrics into a common calculation family.
With such situation, this section will focus on facilitating the understanding of the metrics. Despite the case that there is no idea of threshold value of metrics discussed to define the optimal quality level, the discussed method can still be used to evaualte the quality along various metrics and compare the various design artifacts. This will definitely help to choose the optimum design based on the criteria. Also, there is complexity in understanding and confusion to follow a single proposed metrics procudure. But, the existing procedures can be broken down further to the simple understandable terms. By doing so, the basic terms which are the driving factors for the quality attributes and at the same time followed by most of the procedures as defined in the papers, can be identified.
The remaining part of this section will attempt to collaborate the metrics definitions proposed in various papers, analyze them to identify their conceptual base of measurement in simplest form.

3.3.1 Context and Notations

Before looking into the definitions of metrics, it will help understanding, to know related terms, assumptions and their respective notations.

- the business domain is realized with various processes defined as $P = \{p_1, p_2...p_s\}$
- a set of services realizing the application is defined as $S = \{s_1, s_2...s_s\}$; s_s is the total number of services in the application
- for any service $s \epsilon S$, the set of operations provided by s is given by $O(s) = \{o_1, o_2, ...o_o\}$ and $|O(s)|$ = O
- if an operation $o \epsilon O(s)$ has a set of input and output messages given by M(o). Then the set of messages of all operations of the service is given by $M(s) = \cup_{o \epsilon O(s)} M(o)$

- the consumers of the service $s \epsilon S$ is given by $S_{consumer}(s) = \{S_{c1}, S_2, ...Sn_c\}$; n_c gives the number of consumer services
- the producers for the service or the number of services to which the given service $s \epsilon S$ is dependent upon is given by $S_{producer}(s) = \{S_{c1}, S_2, ...Sn_p\}$; n_p gives the number of producer services

3.3.2 Coupling Metrics

[SSPnp] defines following metrics to determine coupling

$$SOCI(s) = number\_of\_operations\_invoked$$
$$= |\{o_i \epsilon s_i : \exists_{o \epsilon s} calls(o, o_i) \wedge s \neq s_i\}|$$

$$ISCI(s) = number\_of\_services\_invoked$$
$$= |\{s_i : \exists_{o \epsilon s}, \exists_{o_i \epsilon s_i} calls(o, o_i) \wedge s \neq s_i\}|$$

$$SMCI(s) = size\_of\_message\_required\_from\_other\_services$$
$$= | \cup M(o_i) : (o_i \epsilon s_i) \vee (\exists_{o \epsilon s}, \exists_{o_i \epsilon s_i} calls(o, o_i) \wedge s \neq s_i)|$$

where,

- SOCI is Service Operation Coupling Index
- ISCI is Inter Service Coupling Index
- SMCI is Service Message Coupling Index
- $call(o, o_i)$ represents the call made from service 'o' to service 'o_i'

[Xianp] defines

$$Coupling(S_i) = p \sum_{j=1}^{n_s} (-log(P_L(j)))$$
$$= \frac{1}{n} \sum_{j=1}^{n_s} (-log(P_L(j)))$$

where,

- n_s is the total number of services connected
- 'n' is the total number of services in the entire application
- $p = \frac{1}{n}$ gives the probability of a service participating in any connection

[Kaz+11] defines

$$Coupling = \frac{dependency\_on\_business\_entities\_of\_other\_services}{number\_of\_operations\_and\_dependencies}$$

$$= \frac{\sum_{i \epsilon D_s} \sum_{k \epsilon O(s)} CCO(i,k)}{|O(s)|.K}$$

where,

- $D_s = \{D_1, D_2, ...D_k\}$ is set of dependencies the service has on other services
- $K = |D_s|$
- CCO is Conceptual Coupling betweeen service operations and obtained from BE X EBP (CRUD) matrix table constructed with business entities and business operations, where BE is business entity and EBP any logical process defined in an operation.

[Shi+08] defines

$$coupling(s) = \frac{number\_of\_services\_connected}{number\_of\_services}$$

$$= \frac{n_c + n_p}{n_s}$$

where,

- n_c is number of consumer services
- n_p number of dependent services
- n_s total number of services

[AZR11] defines

$$coupling = \frac{number\_of\_invocation}{number\_of\_services}$$

$$= \frac{\sum_{i=1}^{|O(s)|}(S_{i,sync} + S_{i,async})}{|S|}$$

where,

- $S_{i,sync}$ is the synchronous invocation in the operation O_i
- $S_{i,async}$ is the asynchronous invocation in the operation O_i

#	Papers	Metrics Definition
1	[SSPnp]	SOCI : number of operation of other services invoked by the given service ISCI : number of services invoked by a given service SMCI : total number of messages from information model required by the operations
2	[Xianp]	Coupling is evaluated as information entropy of a complex service component where entropy is calculated using various data such as total number of atomic components connected, total number of links to atomic components and total number of atomic components
3	[Kaz+11]	The coupling is measured by using the dependency of a service operations with operations of other services. Additionally, the dependency is considered to have different weight value for each kind of operation such as Create, Read, Update, Delete (CRUD) and based on the type of business entity involved. The weight is referred from the CRUD matrix constructed in the process.
4	[Shi+08]	given by the average number of directly connected services
5	[AZR11]	defines coupling as the average number of synchronous and asynchronous invocations in all the operations of the service

Table 3.2: Coupling Metrics

The table 3.2 shows simpler form of metrics proposed for coupling. Although, the table shows different way of evaluating coupling, they somehow agree to the basic metrics to calculate coupling. It can be deduced that the metrics to evaluate coupling

uses basic metrics such as number of operations, number of provider services and number of messages.

3.3.3 Cohesion Metrics

[SSPnp] defines following metric for cohesion.

$$SFCI(s) = \frac{number\_of\_operations\_using\_same\_message}{number\_of\_operations}$$

$$= \frac{max(\mu(m))}{|O(s)|}$$

where,

- SFCI is Service Functional Cohesion Index
- the number of operations using a message 'm' is $\mu(m)$ such that $m \epsilon M(s)$ and |O(s)| > 0
  The service is considered highly cohesive if all the operations use one common message. This implies that a higly cohesive service operates on a small set of key business entities as guided by the messages and are relavant to the service and all the operations operate on the same set of key business entities. [SSPnp]

[Shi+08] defines

$$cohesion = \frac{n_s}{|M(s)|}$$

[PRF07] defines

$$SIDC = \frac{number\_of\_operations\_sharing\_same\_parameter}{total\_number\_of\_parameters\_in\_service}$$

$$= \frac{n_{op}}{n_{pt}}$$

$$SIUC = \frac{sum\_of\_number\_of\_operations\_used\_by\_each\_consumer}{product\_of\_total\_no\_of\_consumers\_and\_operations}$$

$$= \frac{n_{oc}}{n_c * |O(s)|}$$

$$SSUC = \frac{sum\_of\_number\_of\_sequentials\_operations\_accessed\_by\_each\_consumer}{product\_of\_total\_no\_of\_consumers\_and\_operations}$$

$$= \frac{n_{so}}{n_c * |O(s)|}$$

where,

- SIDC is Service Interface Data Cohesion
- SIUC is Service Interface Usage Cohesion
- SSUC is Service Sequential Usage Cohesion
- n_{op} is the number of operations in the service which share the same parameter types
- n_{pt} is the total number of distinct parameter types in the service
- n_{oc} is the total number of operations in the service used by consumers
- n_{so} is the total number of sequentially accessed operations by the clients of the service

[AZR11] defines

$$cohesion = \frac{max(\mu(OFG, ODG))}{|O(s)|}$$

where,

- OFG and ODG are Operation Gunctionality Granularity and Operation Data Granularity respectively as calculated in section 3.3.4
- $\mu(OFG, ODG)$ gives the number of operations with specific value of OFG and ODG

The table 3.3 gives simplified view for the cohesion metrics. The papers presented agree on some basic metrics such as similarity of the operation usage behavior, commanility in the messages consumed by operations and similarity in the size of operations.

3.3.4 Granularity Metrics

[SSPnp] defines

$$SCG = number\_of\_operations = |O(s)|$$
$$SDG = size\_of\_messages = |M(s)|$$

where,

#	Papers	Metrics Definition
1	[SSPnp]	defines SFCI which measures the fraction of operations using similar messages out of total number of operations in the service operations of the service
2	[PRF07]	SIDC : defines cohesiveness as the fraction of operations based on commonality of the messages they operate on SIUC : defines the degree of consumption pattern of the service operations which is based on the similarity of consumers of the operations SIUC : defines the cohesion based on the sequential consumption behavior of more than one operations of a service by other services
3	[Shi+08]	cohesion is given by the inverse of average number of consumed messages by a service
4	[AZR11]	cohesion is defined as the consensus among the operations of the service regarding the functionality and data granularity which represents the type of parameters and the operations importance as calculated in the section 3.3.4

Table 3.3: Cohesion Metrics

- SCG is Service Capability Granularity
- SDG is Service Data Granularity

[Shi+08] defines

$$granularity = \frac{number\_of\_operations}{size\_of\_messages} = \frac{|O(s)|^2}{|M(s)|^2}$$

[AZR11] defines

$$ODG = fraction\_of\_total\_weight\_of\_input\_and\_output\_parameters$$
$$= \left(\frac{\sum_{i=1}^{n_i o} W_{pi}}{\sum_{i=1}^{n_i s} W_{pi}} + \frac{\sum_{j=1}^{n_j o} W_{pj}}{\sum_{j=1}^{n_j s} W_{pj}} \right)$$

$$OFG = complexity\_weightage\_of\_operation = \frac{W_i(o)}{\sum_{i=1}^{|O(S)|} W_i(o)}$$

$$granularity = sum\_of\_product\_of\_data\_and\_functionality\_granularity$$

$$= \sum_{i=1}^{|O(S)|} ODG(i) * OFG(i)$$

where,

- ODG is Operation Data Granularity
- OFG is Operation Functionality Granularity
- W_{pi} is the weight of input parameter
- W_{pj} is the weight of output parameter
- n_io is the number of input parameters in an operation
- n_jo is the number of output parameters in an operation
- n_is is the total number of input parameters in the service
- n_js is the total number of output parameters in the service
- $W_i(o)$ is the weight of an operation of the service
- $|O(S)|$ is the number operations in the service

The table 3.4 presents various view of granularity metrics based on different research papers. The chapter 2 discuss in detail regarding the topic. Based on the table 3.4, the granularity is evaluated using some basic metrics such as the number and type of the parameters, number of operations and number of messages consumed.

3.3.5 Complexity Metrics

[ZL09] defines

$$RIS = \frac{IS(s_i)}{|S|}$$

RCS

$$complexity = \frac{coupling\_of\_service}{number\_of\_services} = \frac{CS(s_i)}{|S|}$$

where,

#	Papers	Metrics Definition
1	[SSPnp]	the service capability granularity is given by the number of operations in a service and the data granularity by the number of messages consumed by the operations of the service
2	[AZR11]	ODG : evaluated in terms of number of input and output parameters and their type such as simple, user-defined and complex OFG : defined as the level of logic provided by the operations of the services SOG : defined by the sum of product of data granularity and functionality granularity for all operations in the service
3	[Shi+08]	evaluated as the ratio of squared number of operations of the service to the squared number of messages consumed by the service

Table 3.4: Granularity Metrics

- $CS(s_i)$ gives coupling value of a service
- $|S|$ is the number of services to realize the application
- $IS(s_i)$ gives importance weight of a service in an application

The complexity is rather given by relative coupling than by coupling on its own. A low value of RCS indicates that the coupling is lower than the count of services where as RCS with value 1 indicates that the coupling is equal to the number of services. This represents high amount of complexity.
Similarly, a high value of RIS indicates that a lot of services are dependent upon the service and the service of critical value for them. This increases complexity as any changes or problem in the service affects a large number of services to a high extent.
[AZR11] defines

$$Complexity(s) = \frac{Service\_Granularity}{number\_of\_services}$$

$$= \frac{\sum_{i=1}^{|O(s)|}(SG(i))^2}{|S|}$$

where,

- |S| is the number of services to realize the application
- SG(i) gives the granularity of ith service
  calculated as described in section 3.3.4

refer to the document for effect of RCS and RIS on complixity

#	Papers	Metrics Definition
1	[ZL09]	RCS : complexity given by the degree of coupling for a service and evaluated as the fraction of its coupling to the total number of services RIS : measured as the fraction of total dependency weight of consumers upon the service to the total number of services
2	[AZR11]	the complexity is calculated using the granunarity of service operations

Table 3.5: Complexity Metrics

The table 3.5 provides the way to interprete complexity metrics. The complexity is highly dependent upong coupling and functionality granularity of the service.

3.3.6 Autonomy Metrics

[Ros+11] defines

$$Self-Containment(SLC) = sum\_of\_CRUD\_coefficients\_for\_each\_Business\_entity$$
$$= \frac{1}{h_2 - l_2 + 1} \sum_{i=l_2}^{h_2} \sum_{sr \epsilon SR} (BE_{i,sr} XV_{sr})$$

$$Dependency(DEP) = dependency\_of\_service\_on\_other\_service\_business\_entities$$
$$= \frac{\sum_{j=L1_i}^{h1_i} \sum_{k=1}^{BE} V_{sr_{jk}} - \sum_{j=L1_i}^{h1_i} \sum_{k=L2_i}^{j2_i} V_{sr_{jk}}}{nc}$$

$$\text{autonomy} = \begin{cases} SLC - DEP & \text{if SLC > DEP} \\ 0 & \text{otherwise} \end{cases}$$

where,

- nc is the number of relations with other services
- $BE_{i,sr} = 1$ if the service performs action sr on the ith BE and sr represents any CRUD operation and V_{sr} is the coefficient depending upon the type of action
- $V_{sr_{jk}}$ is the corresponding value of the action in jkth element of CRUD matrix, it gives the weight of corresponding business capability affecting a business entity
- $l1_i, h1_i, l2_i, h2_i$ are bounding indices in CRUD matrix of ith service

#	Papers	Metrics Definition
1	[Ros+11]	SLC : defined as the degree of control of a service upon its operations to act on its Business entities only DEP : given by the degree of coupling of the given service with other services autonomy is given by the difference of SLC and DEP if SLC > DEP else it is taken as 0

Table 3.6: Autonomy Metrics

The table 3.6 shows a way to interprete autonomy. It is calculated by the difference SLC-DEP when SLC is greater than DEP. In other cases it is taken as zero. So, autonomy increases as the operations of the services have full control upon its business entities but decreases if the service is dependent upon other services.

3.3.7 Reusability Metrics

[SSPnp] defines

$$\begin{aligned} Reusability &= number\_of\_existing\_consumers \\ &= |S_{consumers}| \end{aligned}$$

[Shi+08] defines

$$Reusability = \frac{Cohesion - granularity + Consumability - coupling}{2}$$

The table 3.7 shows the reusability metrics evaluation. Reusability depends upon coupling, cohesion and granularity. It decreases as coupling and granularity increases.

#	Papers	Metrics Definition
1	[SSPnp]	SRI defines reusability as the number of existing consumers of the service
2	[Shi+08]	evaluated from coupling, cohesion, granularity and consumability of a service where consumability is the chance of the service being discovered and depends upon the fraction of operations in the service

Table 3.7: Reusability Metrics

3.4 Basic Quality Metrics

The section 3.3 presents various metrics to evaluate quality attributes based upon different papers. Additionally, the section analyzed the metrics and tried to derive them in simplest form possible. Eventually, the tables demonstrating the simplest definition of the metrics shows that the different quality attributes are measured on the basis of some basic metrics. Based on the section 3.3 and based on the papers, this section will attempt to derive basic metrics.

#	Metrics	Coupling	Cohesion	Granularity	Complexity	Autonomy	Reusability
1	number of service operations invoked by the service	+			+	-	-
2	number of operation using similar messages		+				+
3	number of operation used by same consumer		+				+
4	number of operation using similar parameters		+				+
5	number of operation with similar scope or capability		+				+
6	scope of operation			+	+		-
7	number of operations			+	+		-
8	number of parameters in operation			+	+		-
9	type of parameters in operation			+	+		-
10	number and size of messages used by operations	+		+	+	-	-
11	type of messages used by operations		+				+
12	number of consumer services	+			+	-	+
13	number of producer services	+			+	-	-
14	type of operation and business entity invoked by the service	+			+	-	-
15	number of consumers accessing same operation		+				+
16	number of consumer with similar operation usage sequence		+				+
17	dependency degree or imporance of service operation to other service				+		
18	degree of control of operation to its business entities					+	

Table 3.8: Basic Quality Metrics

Moreover, the effect of the basic metrics upon various quality attributes are also evaluated. Here, the meaning of the various symbols to demonstrate the affect are as given below:

- + the basic metric affect the quality attribute proportionlly
- - the basic metric inversely affect the quality attribute
- there is no evidence found regarding the relationship from the papers

3.5 Principles defined by Quality Attributes

The section 3.1 has already mentioned that there are two distinct kind of quality attributes: external and internal. The external quality attributes cannot be evaluated during design however can be predicted using the internal quality attributes. This kind of relationship can be used to design service with good quality. Moreover, it is important to identify how each internal attributes affect the external attributes. The understanding of such relationship will be helpful to identify the combined effect of

the quality attributes and eventually to identify the service with the appropriate level of quality as per the requirement. The remaining part of the section lists relationship existing in various quality attributes as well as the desired value of the quality attributes.

1. When a service has large number of operations, it means it has large number of consumers. It will highly affect maintainability because a small change in the operations of the service will be propagated to large number of consumers. But again, if the service is too fine granular, there will be high network coupling. [FL07; Xianp][BKM07]

2. Low coupling improves understandability, reusability and scalability where as high coupling decreases maintainability. [Kaz+11][Erl05][Jos07]

3. A good strategy for grouping operations to realize a service is to combine the ones those are used together. This indicates that most of the operations are used by same client. It highly improves maintainability by limiting the number of affected consumer in the event of any change in the service. [Xianp]

4. A high cohesive quality attribute defines a good service. The service is easy to understand, test, change and is more stable. These properties highly support maintainability. enumerate[np01]

5. A service with operations those are cohesive and have low coupling which make the service reusable. [WYF03][FL07][Ma+09]

6. Services must be selected in a way so that they focus on a single business functionality. This highly follows the concept of low coupling. [PRF07][SSPnp]

7. Maintainability is divided into four distinct concepts: analyzability, changeability, stability and testability.[np01] A highly cohesive and low-coupled design should be easy to analyze and test. Moreover, such a system will be stable and easy to change.[PRF07]

8. The complexity of a service is determined by granularity. A coarse-grained service has higher complexity. However, as the size of the service decreases, the complexity of the over system governance also increases. [AZR11]

9. The complexity depends upon coupling. The complexity of a service is defined in terms of the number of dependencies of a service, number of operations as well as number and size of messages used by the service operations.[AZR11][SSPnp][Lee+01]

10. The selection of an appropriate service has to deal with multi-objective optimization problem. The quality attributes are not independent in all aspects. Depending upon goals of the architecture, tradeoffs have to be made for mutually exclusive attributes. For example, when choosing between coarse-grained and fine-grained service, various factors such as governance, high network roundtrip etc. also should be considered. [JSM08]

11. Business entity convergence of a service, which is the degree of control over specific business entities, is an important quality for the selection of service. For example: It is be better to create a single service to create and update an order. In that way change and control over the business entity is localized to a single service.[Ma+09]

12. Increasing the granularity decreases cohesion but also decreases the amount of message flow across the network. This is because, as the boundary of the service increases, more communication is handled within the service boundary. However, this can be true again only if highly related operations are merged to build the service. This suggests for the optimum granularity by handling cohesion and coupling in an appropriate level.[Ma+09][BKM07]

13. As the scope of the functionality provided by the service increases, the reusability decreases. [FL07]

14. If the service has high interface granularity, the operations operates on coarse-grained messages. This can affect the service performance. On the other hand, if the service has too fine-grained interface, then there will be a lot of messages required for the same task, which then can again affect the overall performance. [BKM07]

3.6 Relationship among Quality Attributes

In order to determine the appropriate level of quality for a service, it is also important to know the relationship between the quality attributes. This knowledge will helpfull to decide tradeoffs among them in the situation when it is not possible to achieve best of all. Based on the section 3.2 and section 3.5, the identified relationship among the quality attributes are shown in the table 3.9.

Here, the meaning of the various symbols showing nature of relationship are as given below:

#	Quality Attributes	Coupling	Cohesion	Granularity
1	Coupling		-	+
2	Cohesion	-		
3	Granularity	+		
4	Complexity	+		+
5	Reusability	-	+	-
6	Autonomy	-		
7	Maintainability	-	+	-

Table 3.9: Relationship among quality attributes

- + the quality attribute on the column affects the quality attribute on the corresponding row positively
- - the quality attribute on the column affects the quality attribute on the corresponding row inversely
- there is no enough evidence found regarding the relationship from the papers or these are same attributes

4 Service Candidate

4.1 Introduction

In the previous chapters, granularity and quality of a microservice was focussed. The qualitative as well as quantitative aspects of granularity were described. Additionally, various quality metrics were discussed to measure different quality attributes of a microservce. Moreover, a set of principle guidelines and basic metrics to qualify the microservices were also listed.
In addition to that, it is also equally important to agree on the process to identify the service candidates. In this chapter and following chapters, the ways to recognize microservices will be discussed. There are three basic approaches to identify service candidates: Top-down, bottom-up and meet-in-the-middle.
The top-down approach defines the process on the basis of the business model. At first, various models are designed to capture the overall system and architecture. Using the overall design, services are identified upfront in the analysis phase.
The next approach is bottom-up, which base its process on the existing architecture and existing application. The existing application is studied to identify the cohesion and consistency of the features provided by various components of the system. This information is used to aggregate the features and identify services in order to overcome the problems in the existing system.
Finally, the meet-in-the-middle approach is a hybrid approach of both top-down and bottom-up approaches. In this approach, the complete analysis of system as a whole is not performed upfront as the case of top-down approach. Where as, a set of priority areas are identified and used to analyse their business model resulting in a set of services. The services thus achieved are further analysed critically using bottom-up approach to identify problems. The problems are handled in next iterations where the same process as before is followed. The approach is continued for other areas of the system in the order of their priority.[RS07][Ars04]

4.2 Related Work

There are various efforts made regarding the identification of service candidates. Among them, most follows top-down approach. Service Oriented Development Architecture (SODA) uses service driven modeling approach to design and implement services.[Nig05] Service Oriented Analysis and Design (SOAD) on the other hand uses existing modeling process and notation to desing services. [Ars05] Service Oriented Unified Process(SOUP) is highly influenced by Rational Unified Process and extreme programming techniques.[Mit05]
A bottom-up approach Feature Oriented Domain Analysis is also proposed to analysis the existing system features for identifying services.[Kan+90]
Among the few meet-in-the-middle approaches is Feature Analysis for Service-Oriented Reengineering which uses business modeling and FODA. [Che+05] Similarly, Service Oriented Modeling Architecture (SOMA) emphasise on using goal-oriented meet-in-the-middle approach.[Ars04]
Again, there are a majority of papers using use case modeling to identify services. The papers [Mil+07] [How+09] defines Service Responsibility and Interaction Design Method (SRI-DM) to utilize use case to identify the service with cohesive set of functionalities. Similarly, the paper [Far08] identifies various levels of services based on the abstraction level of the use cases.

5 Selection By Use Case

5.1 Use case

A use case is an efficient as well as a fancy way of capturing requirements. Another technique of eliciting requirement is by features specification. However, the feature specification technique limits itself to answer only "what the system is intended to do". On the other hand, use case goes further and specifies "what the system does for any specific kind of user". In this way, it gives a way to specify and most importantly validate the expectation and concerns of stakeholders at the very early phase of software development. Undoubtedly for the same reason, it is not just a tool for requirement specification but an important software engineering technique which guides the software engineering cycle. Using use cases, the software can be developed to focus on the concerns that are valueable to the stakeholders and test accordingly.[NJ04]
It can be valueable to see the ways it has been defined.

Definition 1: [Jac87]

"A use case is a sequence of actions performed by the system to yield an observable result of value to a particular user."

Definition 2: [RJB99]

"A use case is a description of a set of sequences of actions, including variants, that a system performs that yield an observable result of value to an actor."

5.2 Use case Refactoring

According to [Jac87] it is not always intuitive to modularize use cases directly. As per the definitions provided in 5.1, a use case consists of a number of ordered functions together to accomplish a certain goal. [Jac87] defines these cohesive functionalites

distributed across usecases as clusters. And the process of identifying the clusters scattered around the use cases is the process of identifying services. [NJ04] and [Jac03] describes use case possibly consisting of various cross-cutting concerns. The papers use the term 'use case module' to define the cohesive set of tasks. The figure 5.1 demonstrate the cross cutting functionalities needed by a use case in order to accomplish its goal. For example, the use case 'does A' has to perform separate functionalities on different domain entities X and Y. Similarly, the use case 'does B' needs to perform distinct functions on entities X, Y and Z.

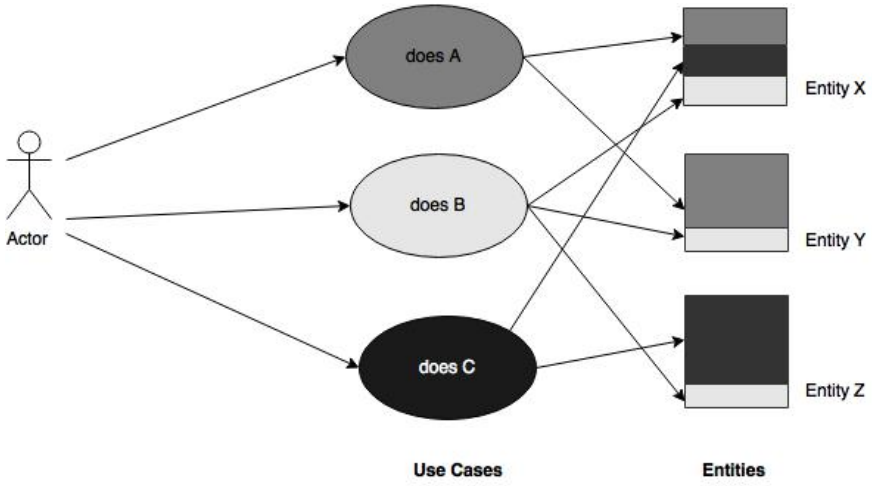

Figure 5.1: Use cases with cross-cutting concerns [NJ04]

This situation prompts for the analysis of the use cases for cross cutting tasks and refactoring them in order to map the cohesive functionalities and use cases. Refactoring helps to achieve the right level of abstraction and granularity by improving functionality cohesion as well as elimination of redundancy and finally promoting the reusablity. [DK07]
Furthermore, the refactoring is assisted by various relationships in use case model to represent dependencies between use cases, which are include, generalization and extend. [NJ04]

5.3 Process for Use Case Refactoring

In order to refactor the use cases and finally map use cases to the service, the papers [KY06], [YK06] and [DK07] provide a comprehensive method. It is a type of meet-in-the-middle approach where use case models are first created and then refactored to create new set of use cases to accomplish high cohesion in functionality and loose coupling. This will ultimately create use cases supporting modularity and autonomy. [Far08] There are three distinct steps for service identification using service refactoring.

1. **TaskTree Generation**
   In this step, the initial use case model created during domain analysis is used to create task trees for each distinct use case. The task trees provide sequence of individual tasks required to accomplish in order to achieve the goal of the use case.

2. **Use Case Refactoring**
   In this step, the task tree generated in the previous step is analysed. As already mentioned in the section 5.2, that the initial use case consists of various cross cutting functionalities which runs through large number of business entities. In order to minimize that, refactoring is performed. The detail rules are provided in section 5.4.

3. **Service Identification**
   The use cases achieved after refactoring have correct level of abstractions and granularity representing cohesive business functionality. The unit use cases thus achieved appreciate reuse of common functionality. [DK07] Furthermore, the approach considers the concerns of stakeholders in terms of cohesive business functionalities to be represented by use cases.[Far08] Additionally, the process also clarifies the dependencies between various use cases. Thus, the final use cases obtained can be directly maped to individual services.

5.4 Rules for Use Case Refactoring

The context is first defined before distinct rules are presented.

Context 1

If U represents use case model of the application, t_i represents any task of U and T being the set of tasks of U. Then, $\forall t_i \in T$, t_i exists in the post-refactoring model U'. The refactoring of a use case model preserves the set of tasks.

Context 2

A refactoring rule R is defined as a 3-tuple (Parameters, Preconditions, Postconditions). Parameters are the entities involved in the refactoring, precondition defines the condition which must be satisfied by the usecase u in order for R to be applied in u. Postcondition defines the state of U after R is applied.
The various refactoring rules are as listed below.

5.4.1 Decomposition Refactoring

When the usecase is complex and composed of various functionally independent tasks, the tasks can be ejected out of the task tree of the usecase and represented as a new use case. The table 5.1 shows the decomposition rule in detail.

Parameters	u: a use case to be decomposed t: represents task tree of u t': a subtask tree of t
Preconditions	t' is functionally independent of u
Postconditions	1. new use case u' containing task tree t' is generated 2. a dependency is created between u and u'

Table 5.1: Decomposition Rule

5.4.2 Equivalence Refactoring

If the two use cases share their tasks in the task tree, we can conclude that they are equivalent and redundant in the use case model. The table 5.2 shows the rule for the refactoring.

Parameters	u_1 , u_2: two distinct use cases
Preconditions	the task trees of u_1 and u_2 have same behavior
Postconditions	1. u_2 is replaced by u_1 2. all the relationship of u_2 are fulfilled by u_1 3. u_2 has no relationship with any other use cases

Table 5.2: Equivalence Rule

5.4.3 Composition Refactoring

When there are two or more small-grained use cases such that they have related tasks, the use cases can be represented by a composite unit use case.

Parameters	u_1, u_2: the fine grained use cases t_1, t_2: the task trees of u_1 and u_2 respectively
Precondition	t_1 and t_2 are functionally related.
Postconditions	1. u_1 and u_2 are merged to a new unit use case u 2. u has new task tree given by $t_1 \cup t_2$ 3. the dependencies of u_1 and u_2 are handled by u 4. u_1 and u_2 are deleted along with their task trees t_1 and t_2

Table 5.3: Composition Rule

5.4.4 Generalization Refactoring

When multiple use cases share some volume of dependent set of tasks in their task trees, it can be implied that the common tasks set can be represented by a new use case. The table 5.4 provides the specific of the rule.

Parameters	u_1 , u_2: two distinct use cases t_1 , t_2: task trees of u_1 and u_2 respectively
Precondition	t_1 and t_2 share a common set of task $t = \{x_1, x_2...x_n\}$
Postconditions	1. a new use case u is created using task tree $t = \{x_1, x_2...x_n\}$ 2. relationship between u with u_1 and between u with u_2 is created 3. the task tree $t = \{x_1, x_2...x_n\}$ is removed from task tree of both u_1 and u_2 4. the common relationship of both u_1 and u_2 are handled by u and removed from them

Table 5.4: Generalization Rule

5.4.5 Merge Refactoring

When a use case is just specific for another use case and the use case is only the consumer for it, the two use cases can be merged into one.

Parameters	u, u′: the use cases r defines the dependency of u with u′ t, t′: the task trees of u and u′ respectively
Precondition	there is no dependency of other usecases with u′ except u
Postconditions	1. u′ is merged to u 2. u has new task tree given by $t \cup t'$ 3. r is removed

Table 5.5: Merge Rule

5.4.6 Deletion Refactoring

When a use case is defined but no relationship can be agreed with other use cases or actors then it can be referred that the use case represent redundant set of tasks which has already been defined by other use cases.

Parameters	u: a distinct use case
Precondition	the use case u has no relationship with any other usecases and actors
Postconditions	the use case u is deleted

Table 5.6: Deletion Rule

5.5 Example Scenario

In this section, a case study will be taken as an example. The case study will be modeled using use case. Finally, the use case will be refactored using the rules given by **??**.

Case Study

The case study is about an international hotel named 'XYZ' which has branches in various locations. In each location, it offers rooms with varying facilities as well as prices. In order to make it easy for customers to find room irrespective of the location of customer, it is planning to offer an online room booking application. Using

this application, any registered customer can search for a room according to his/her requirement in any location. When he/she is satisfied, can book the room online as well. The customer will be sent notification by email regarding booking. At present, the payment will be accepted in person, only after the customer gets into the respective branch. Finally, if the customer wants to cancel the room, he/she can do online anytime. The customer will also be notified by e-mail to confirm the cancelation of booking. It is an initial case study so only priority cases and conditions are considered here.

The remaining part of this section presents the steps to indentify the services using use case refactoring following the process defined in section 5.3 and rules presented in the section 5.4

Step 1:

The initial analysis of the case study will produce use case model as given by figure 5.2.

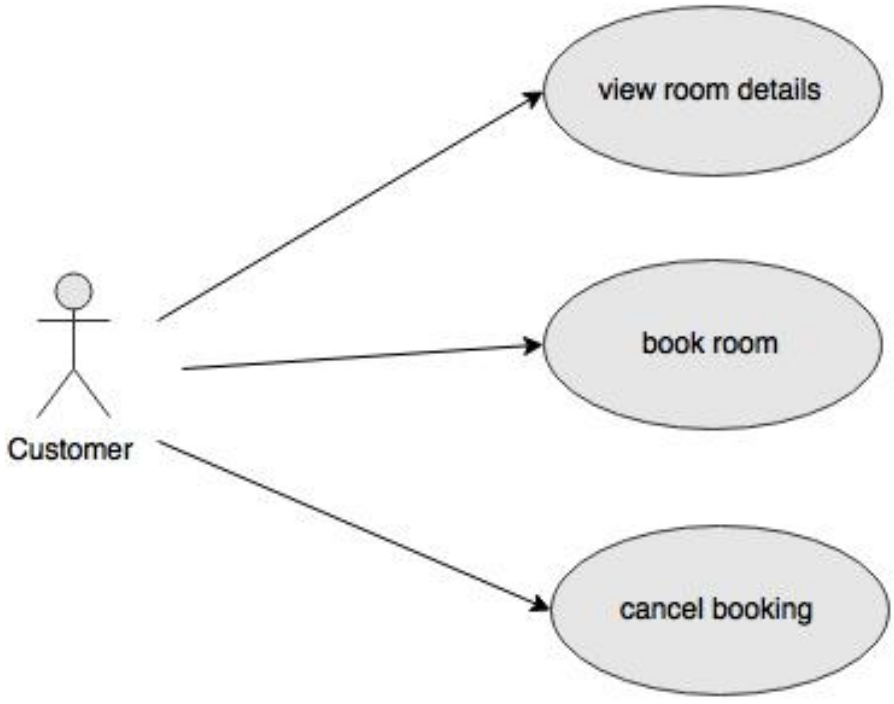

Figure 5.2: Initial Use Case Model for Online Room Booking Application

Step 2:

For each initial use case, task trees are generated which are needed to accomplish the desired functionality of respective use cases.

Use Case	Task Tree
book room	customer enters access credentials system validates the credentials customer enters location and time details for booking system fetch the details of empty rooms according to the data provided system displays the details in muliple pages customer choose the room and submits for booking system generates a booking number System updates the room system sends notification to the customer regarding booking
cancel booking	customer enters access credentials system validates the credentials customer enters the booking number system validates the booking number customer cancels the booking system updates the room system sends notification to the customer regarding cancelation
view room details	customer enters access credentials system validate the credentials customer enter location and time system fetch the details of empty rooms according to the data provided system display the details in multiple pages

Table 5.7: Task Trees for Initial Use Cases

Step 3:

The initial task trees created for each use case at Step 2 is analysed. There are quite a few set of tasks which are functionally independent of the use case goal and also few tasks which are common in more use use cases. The following table 5.8 lists those tasks from the tasks trees 5.7 which are either functionally independent from their corresponding use cases or common in more use cases.

Tasks Type	Tasks
Independent Tasks	system validates the credentials system fetch the details of empty rooms according to the data provided system generates a booking number system validates the booking number system updates the room system sends notification to the customer
Common Tasks	system validates the credentials system fetch the details of empty rooms according to the data provided system updates the room system sends notification to the customer

Table 5.8: Common and Independent Tasks

Now, for independent tasks the docomposition rule given by 5.1 can be applied and for common tasks, the generalization rule given by 5.4 can be applied. The use cases obtained after applying these rules is shown by the figure 5.3

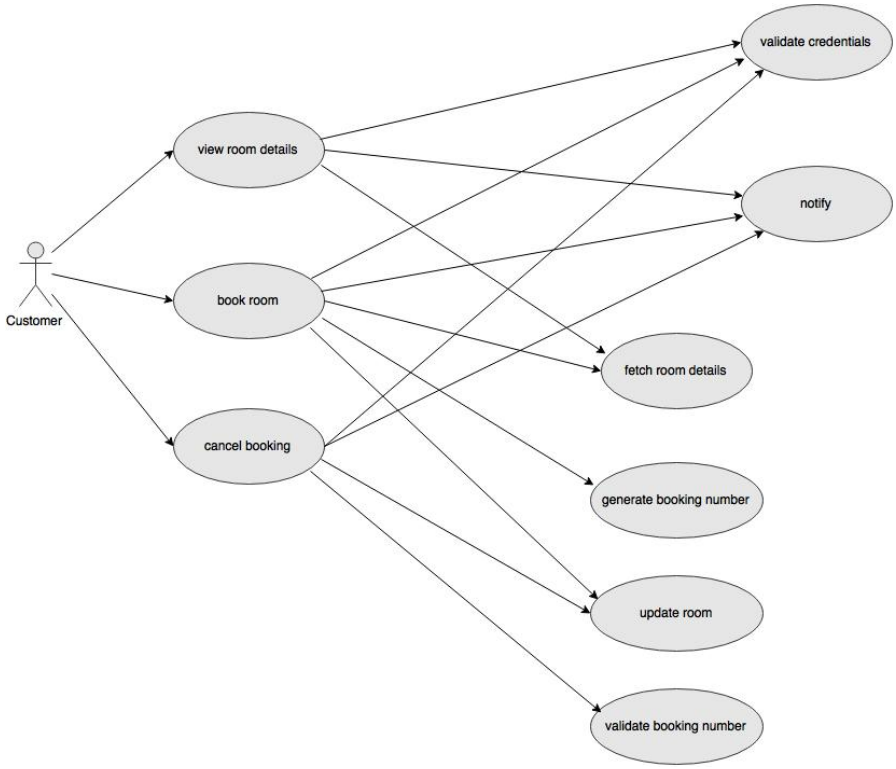

Figure 5.3: Use Case Model after applying Decomposition and Generalization rules

Step 4:

The use case model 5.3 obtained in Step 3 is analysed again for further refactoring. It can seen that the use cases 'fetch room details' and 'update room' are fine grained and related to same business model 'Room'. Similarly, the use cases 'generate booking number' and 'validate booking number' are related to business entity 'Booking Number'. The composition rule given by 5.3 can be applied to these use cases, which will then result in use case model given by figure 5.4

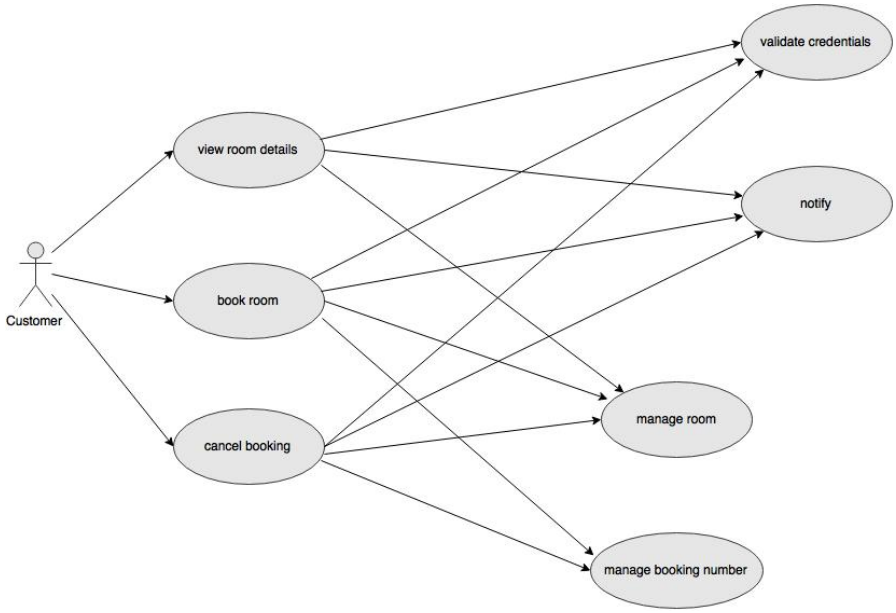

Figure 5.4: Use Case Model after applying Composition rule

Step 5:

Finally, the use case obtained in step 4 is used to identify the service candidates. The final use cases obtained in Step 4 have appropriate level of granularity and cohesive functionalities to be identified as individual services. Most importantly, the refactoring has now separated the cross cutting concerns in terms of various reusable fine grained use cases as shown by the figure. If we compare the the final use case model 5.5 with respect to figure 5.1, then it can be implied that the functionalities operating on business entities as well as the business logic serving the candidate concerns are separated and represented by individual use cases. So, the each use case can be mapped to individual services. The services are as listed in the bottom of the figure 5.5.

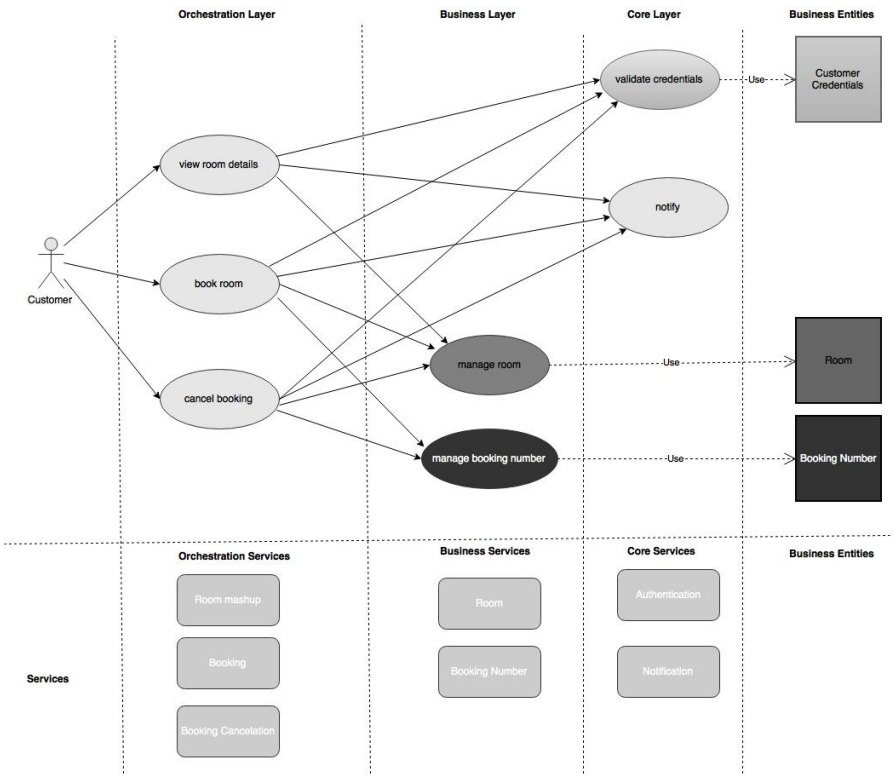

Figure 5.5: Use Case Model for identification of service candidates

Service Layers

The services or use cases obtained from the refactoring creates various levels of abstractions. The levels of abstractions represents different level of functionality. The services at the bottom layer are core services which serve many higher level services. These services are not dependent on any other services. 'Notification' and 'Authentication' are core services obtained from refactoring. The layer above the core layer is business layer. The business service can either have entity services, which control some related business entities or can have task services which provides some specific business logic to higher level services. 'Room' and 'Booking Number' are the entity services obtained from refactoring. Finally, the top layer is mashup layer which contains high level business logic and collaborates with business services as well as core services. 'Room mashup' , 'Booking' and 'Booking Cancelation' are orchestration layer services obtained. The individual services as well as their respective layers are given in figure 5.5. [Far08][Emi+np][Zim+05]

6 Domain Driven Design

6.1 Introduction

Understanding the problem space can be a very useful way to design software. The common way to understand the problem and communicate them is by using various design models. The design models are the abstraction as well as representation of the problem space. However, when the abstractions are created, there are chances that important concepts or data are ignored or misread. This will highly impact the quality of software produced. The software thus developed may not reflect the real world situation or problem entirely.
Domain Driven Design provides a way to represent the real world problem space so that all the important concepts and data from real world remains intact in the model. The domain model thus captured respects the differences as well as the agreement in the concepts across various parts of the problem space. Moreover, it also provides a way to divide the problem space into manageable independent partitions and makes it easy for developers as well as stakeholders to focus on the area of concern as well as be more agile. Finally, the domain models act as the understandable and common view of the business for both domain experts as well as the developers. This will make sure that the software developed using domain driven design will comply with business need.[Eva03][Ver13]

6.2 Process to Domain Driven Design

There are three basic parts to implement domain driven design:

1. Ubiquitous Language
2. Strategical Design
3. Tactical Design

As the focus is to describe the process of identifying microservice candidates, only the first two parts are relevant and will be focused in the later part of this chapter.

6.2.1 Ubiquitous Language

Ubiquitous Language is a common language agreed among domain experts and developers in a team. It is important to have a common understanding about the concepts of a business which is being developed and ubiquitous language is the way to assure that. Domain Experts understand the domain in terms of their own jargon and concept. It is difficult for a developer to understand them. Usually, developers translates those jargons into the terms they understand easily during desing and implementation. However along the way of translation, major domain concepts can get lost and the immediate value of the resulting solution might decrease tremendously. In order to prevent creation of such low valued solution, there should be a meet-in-the-middle approach to define common vocabularies and concepts understood by all domain experts as well as the developers. These common vocabularies and concepts make the core of the ubiquitous language.[Eva03][Ver13]
Along with the common vocabulary and concepts, domain models provide backbone to create ubiquitous language. The models represents not only artifacts but also functionalities, rules and strategies. It is a way to express the common understanding in a visual form providing an easy tool to comprehend.[Eva03][Fow06] According to [Fow03], domain model should not be confused with data model which represents the business in datacentric view but rather each domain object should contain data as well as logic closely related to the data contained. Domain models are conceptual models rather than software artifacts but can be effectively visualized using UML.[SR01]

Figure 6.1: Process to define Ubiquitous Language [Eva03]

The figure 6.1 visualizes the process of discovering the ubiquitous language in any domain. It is not a discretely timed phenomenon but a continous procedure with various phases occuring in a cycle throughout the development.
On arrival of any new term, concept or any confusion, the contextual meaning of those terms are made clear before adding to the domain vocabulary. Next, the vocabularies and concepts are used to create various domain models following UML. The domain models and various vocabularies for the common laguage among domain experts and developers within the team. It is very important to use only domain vocabularies for communication and understanding which will not only help to reflect the hidden domain concepts in the implementation but also create opportunities to create new vocabularies or refine existing ones in the event of confusion and disagreement.[Eva03]
Thus, the common vocabulary and domain models form the core of the ubiquitous

language and the only way of coming up with the better one is by applying it in communication extensively. Ubiquitous language is not just a collection or documentation of terms but is the approach of communication within a domain.

6.2.2 Strategical Design

When applying model-driven approach to an entire enterprise, the domain models get too large and complicated. It becomes difficult to analyse and understand all at once. Furthermore, it gets worse as the system gets bigger. The strategical design provides a way to divide the entire domain models into small,manageable and interoperable parts which can work together with low dependency in order to reflect the functionalities of the entire domain. The goal is to divide the system into modular parts which can be easily integrated. Additionally, the all-cohesive unified domain models of the entire enterprise cannot reflect differences in contextual vocabularies and concepts.[Fow14][Eva03][Ver13]
The strategical design specifies two major steps which are crucial in identifying microservices.

Step 1: Divide problem domain into subdomains

The domain represents the problem being solved by the software. The domain can be divided into various sub-domains based on the organizational structure of the enterprise, each sub-domain responsible for certain area of the problem. The [Eng+np] provides a comprehensive set of steps to identify sub-domains.

1. **Identify core business functionalities and map them into domain:**
   A core business functionality represents the direct business capability which holds high importance and should be provided by the enterprise in order for it to succeed. For example, for any general e-commerce enterprise, the core focus will on the high amount orders being received from the customers and maintain the inventory to fulfil the orders. So, for those kind of e-commerce enterprises, 'Order Management' and 'Inventory Management' will be the some of the core domains.

2. **Identify generic and supporting subdomains:**
   The business capabilities which are viable for the success of business but not represent the specialization of the enterprise falls into supporting subdomains.

The supporting subdomain does not require the enterprise to excel in these areas. Additionally, if the functionalities are not specific to the business but in a way support the business functionalities, then these are covered by generic subdomains. For example: for an e-commerce enterprise, 'Payment Management' can be a supporting subdomain whereas 'Reporting' and 'Authentication' can be generic subdomains.[Ver13]

3. **Divide existing subdomains having multiple independent strategies:**
   The subdomains found from earlier steps are analyzed to check if there are mutually independent strategies to handle the same functionality. The subdomain can then be further divided into multiple subdomains along the dimension of strategies. For example: the 'Payment Managment' subdomain discovered in earlier step can have different way of handling online payment depending upon the provider such as bank or paypal etc. In that case 'Payment Management' can be further divided into 'Bank Payment Management' and 'Paypal Payment Management'.

Step 2: Indentify bounded contexts

As stated earlier of this section, the attempt to use a single set of domain models for the entire enterprise adds complexity for understanding and analysis. There is another important reason supporting the issue which is called unification of the domain models. Unification represents internal consistency of the terms and concepts described by the domain models. It is not always possible to have consistent meaning of terms without any contradictory rules throughout the enterprise. The identification of the logically consistent and inconsistent domain model creates a boundary around domain model. This boundary will bind all the terms which share consistent logic from those which are different from them so that there is clear understanding of the concept inside the boundary without any confusion. The shared context inside the boundary is defined as the bounded context.

Definition 1:

" A Bounded Context delimits the applicability of a particular model so that team members hava a clear and shared understanding of what has to be consistent and how it relates to other contexts. A bounded context has specific functionality and explicitly defines boundary around team organization, code bases and database schemas."[Eva03]

Definition 2:

" A Bounded Context is an explicit boundary within which a domain model exists. Inside the boundary all terms and phrases of the Ubiquitous Language have specific

meaning, and the model reflects the Language with exactness." [Ver13]
Definition 3: " A Bounded Context is a specific responsibility enforced by explicit boundaries." [Mik12]
Thus, the idea of bounded context provide the applicability of the ubiquitous language inside its boundary performing a specific responsibility. Outside the bounded contexts, teams have different ubiquitous language with different terms, concepts, meanings and functionalities. Apart from identifying independent and specific responsibilities inside a subdomain in order to identify bounded contexts, there are some general rules which can be considered as well.

1. **Assign individual bounded context to the subdomains identified in Step 1**
   Each subdomains have distinct functionalities and complete set of independent domain models to accomplish the functionalities. Thus, one to one mapping of subdomain and bounded context is possible ideally whereas in real cases, it may not be always possible which can be tackled using the rules discussed in further steps. For example: Order Management and Inventory Management can be two distinct bounded context each with own consistent ubiquitous language. [Fow14][Gor13]

2. **Identify polysemes**
   Polysemes are the terms which have different meaning in separate contexts. If the example of the generic subdomain 'Authentication' is taken. It can have an model 'Account' however the same model can refer to multiple concept such as social account identified by e-mail or registered account identified by unique username. The handling of these two separate concepts is also different which clearly suggests two separate context for each within 'Authentication' subdomain. Additionally, there can be polysemes in two separate domains as well. For example: there is also 'Account' model to identify bank account in the subdomain 'Bank Payment Management'. The identification of polysemes is vital to create clear boundary around which the concept and handling of such polysemes are consistent. [Fow14]

3. **Identify independent generic functionality supporting the subdomain**
   There can be a necessity of independent technical functionality required to accomplish the business functionality of the subdomain only. Since it is only required by the particular subdomain, the functionality cannot be nominated as a generic subdomain. In such a case, the independent technical functionality can be realized as a separate bounded context. For example, in the subdomain 'Inventory

Management', the functionality of full-text search can be assigned a separate bounded context as it is independent.[Gor13]

6.3 Microservices and Bounded Context

Analyzing various definitions of microservices provided in section 1.3, a list of important features with respect to specific category such as granularity or quality was compiled in the table 1.1. In addition to the definitions of bounded context provided in section 6.2.2, the following definition of Domain Driven Design can be helpful to find analogy between a microservice and a bounded context.

Definition: Domain Driven Design

"Domain Driven Design is about explaining what your company does in isolated parts, in performing these specific parts and you can have a dedicated team to do this stuff, and you can have different tools for different teams." [Rig15]

Using table 1.1 and concepts extracted from various definitions of bounded context as well as Domain Driven Design provided, the table 6.1 is generated.

#	Expected features of a microservice	Fulfilled by Bounded Context
1	Loosely coupled, related functions	✓
2	Developed and deployed independently	✓
3	Own database	✓
4	Different database technologies	✓
5	Build around Business Capabilities	✓
6	Different Programming Languages	✓

Table 6.1: Analogy of Microservice and Bounded Context

The table 6.1 lists various features expected by a microservices. Again, it shows that the tabulated features are fulfilled by a bounded context. It can be implied that a microservice is conceptually analogous to a bounded context. The section 6.2 provided the detailed process to discover bounded contexts within a large domain, which ultimately also provides guidelines to determine microservices using domain driven design principles.
Furthermore, there can be found enough evidence regarding the analogy as well as

practical implementation of the concept. The table lists various articles which agree on the analogy and in most cases, have utilized the concepts to create microservices in real world.

#	Articles	Agreement on the Concept	Utilized Bounded Context to create Microservices
1	[Mau15]	✓	✓
2	[Hug14]	✓	
3	[FL14]	✓	
4	[Sok15]	✓	
5	[Day+15]	✓	✓
6	[Rig15]	✓	
7	[Bea15]	✓	✓
8	[KJP15]	✓	✓
9	[Vie+15]	✓	✓
10	[BHJ15]	✓	✓

Table 6.2: Application of Bounded Context to create Microservices

6.4 Example Scenario

In this section, a case study will be discussed. The case study will be used to design the system using microservices architecture following the domain driven design given supported by the section 6.2.

Case Study

The case study is for the same hotel introduced in the section 5.5. The hotel 'XYZ' is thinking of upgrading its current system and adding new business usecases in order to tune its competitive edge in the market.
Previously, the hotel only supported booking of the rooms, payment by cash, cancelation of booking and finally viewing of the room details.
The hotel has planned to provide various package offerings to the either general customers or specific group of customers satisfying certain profile. Firstly, a hotel staff creates a package with name, description, valid time period, applicable type of room and location and discount associated with the package. Furthermore, the package will also contain certain constraints such as maximum number of offerings and maximum number of offerings per customer. The offering is represented by coupon and has

unique identity. Once a package is created, the package has to be activated either by the hotel staff or by the activation time trigger. Only, the activated packages are visible to the customers. The customers can apply for the active packages. The customer profile is validated against the expected profile for the package and then a new coupon is sent to the customer.
The room booking workflow has no changes from the old system where a customer can view list of rooms with various information and send room booking request to the system. The system will then send confirmation with unique booking code to the customer.
Finally, the payment can be done by debit card or by paypal. During the payment, the customer can specify a valid coupon. The payment system will then redeem the coupon so that the respective discount represented by the coupon is deducted from the overal price.
Additionally, a detailed report is planned to be made available for customers showing various transactions with the hotel for the room bookings along with the information of the rooms till date. Similarly, hotel staff has various types of reports such as profit statement and room booking status.
The steps listed in 6.2 can be used to design the system defined by the case study 6.4.
Step 1:
With reference to the steps shown by the figure 6.1, the first step would be to find out new domain vocabularies, understand the meaning, agree on them and use them as the common vocabularies. The table 6.3 lists some important domain terms taken from the case study 6.4 and clarifies the meaning for each. It is interesting to notice that the term 'Profile' has two different meaning depending upon the context of package and customer and it is a polyseme.

Domain Terms	Definition
Package	a discount offering to certain group of customers
Profile	the expected characteristics of customers to be elligible to apply a package
Discount	the reduction value offered in the total price for hotel service
Constraint	defines the limitation of creating coupons for any package
Coupon	represents the authorized assignment of a valid package to a valid customer, which can be used later
Customer Profile	the current characteristics of a customer
Room Booking	assignment of a room to a customer for certain period
Booking Code	a unique code representing the valid room booking
Redeem Coupon	request to use the coupon whereby the associated discount value is deducted from the total price
Price	the total amount for the hotel service

Table 6.3: Domain Keywords

Step 2:
Next, the subdomains are identified following the step 1 6.2.2 of strategical design. The

core domains, supporting domains and generic domains are identified and listed in the table.

Core Domains	1 Package 2 Booking 3 Checkout
Supporting Domains	1 Payment 1.1 Paypal 1.2 CardPayment 2 User Management 2.1 Customer Management 2.2 Staff Management 3 Room
Generic Domains	1 Reporting 2 Authentication

Table 6.4: Subdomains

The supporting domain "User Management" has different stragegy for Customer and Staff so can be further divided into "Customer Management" and "Staff Management" subdomains. For the same reason, the "Payment" subdomain can also be further divided into "Paypal" and "CardPayment".

Step 3:

The next action is to identify bounded context following the process described in step 6.2.2. The process is implemented in each subdomains. In order to accomplish this, domain models can be constructed for each subdomain. Along the process, the various ambguities in the domain models and clear understanding of the concept of the models can be clarified. This will highly help to achieve consistent ubiquitious language with the unification of domain models as well as shared vocabulary inside each subdomain. The boundary around the uniform and consistent domain represents the bounded contexts. Additionally, the rules given in the section 6.2.2 can be applied when appropriate to identify bounded contexts. The following section will provide the analysis of major subdomains to identify the bounded contexts within.

Customer Management

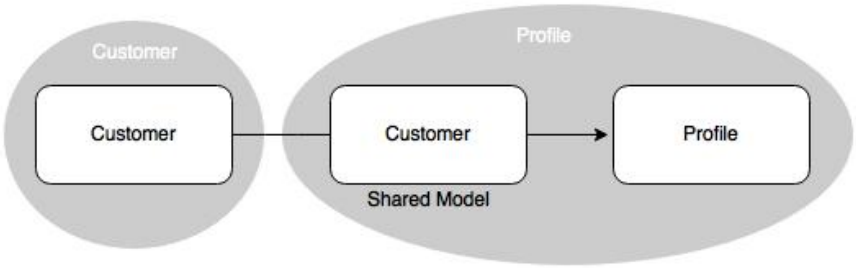

Figure 6.2: Domain Model for Customer Management

The management of customer includes two distinct broad task which are managing the customer itself and management of attributes of each customer which contribute to creating profile of customer at any point of time. Although, these two functionalities are related to customer, are actually loosely coupled and have different performance requirement. For eg: the change in the decision of adding attribute to a customer profile does not change the way customer is managed and also the rate at which customer profile is updated will definitely have high value than the rate customer is managed. Additionally, not all the attributes from customer is relevant to customer profile which makes 'Customer' a polyseme and a shared model between the two bounded contexts 'Customer' and 'Profile'.

Booking

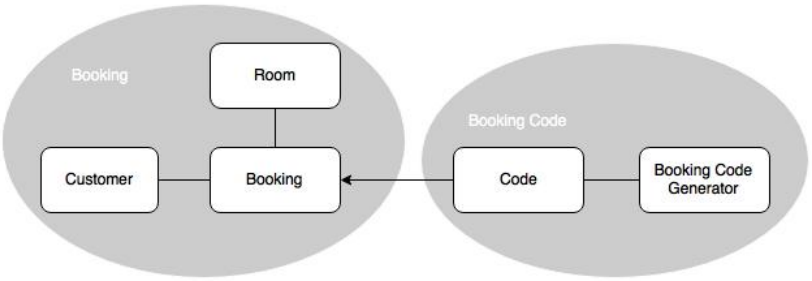

Figure 6.3: Domain Model for Booking

The figure 6.3 shows the domain models of "Booking" subdomain. All the domain models and terms inside the subdomain are consistent and clear. However, the models 'Room', 'Customer', 'Booking' and 'Code' are closely related to booking functionality whereas the functionality of generating the booking code can be considered independent of booking. This gives two loosely coupled bounded contexts: "Booking" and "Booking Code". It is also interesting to notice that the models 'Customer' and 'Room' are polysemes with regard to other 'Customer' and 'Room' bounded contexts.

Checkout

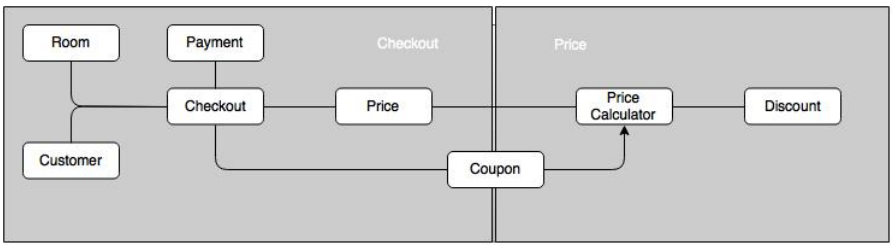

Figure 6.4: Domain Model for Checkout

The overall billing price calculation during checkout, which includes validation of coupon and redeem of the coupon onto the total price, is independent of the orchestration logic which occurs during checkout. The subdomain can be realized into two different bounded contexts 'Checkout' and 'Price'. Additionally, the domain models

'Customer, 'Discount', 'Room', 'coupon' and 'price' are polysemes for these bounded context with respect to other bounded contexts.

Package Management

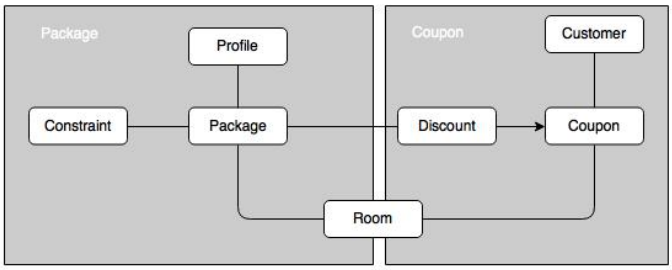

Figure 6.5: Domain Model for Package Management

The logic of package management can be considered independent of the way coupon is generated and validated. The packages are created by staff memeber whereas a coupon is created upon receiving a request from customer. Additionally, these are very specific and independent responsibilities, eligible of having own bounded contexts. The only data shared by the bounded contexts are about the information regarding eligible rooms and discount values. It should be made clear at this point that the 'Profile' domain model here is very different than 'Profile' domain model from 'Profile' boundend context of 'Customer Management' subdomain. In here, 'Profile' model defines the attributes of generic customer eligible for a specific package, however the 'Profile' domain model in 'Profile' bounded context provides the updated characteristics of any individual customer at the moment. Similarly, the models such as 'Customer' and 'Room' are polysemes.
The same process can be applied to identify bounded contexts for rest of the subdomains.

7 Architecture at Hybris

7.1 Overview

SAP YaaS provides a variety of business services to support as well as enhance the products offered as SAP hybris front office such as hybris Commerce, hybris Marketing, hybris Billing etc. Using these offered services, developers can create their own business services focussed on their customer requirements.
The figure 7.1 provides the overview of YaaS. YaaS provides various business processes as a service (bPaaS) essential to develop applications and services thus filling up the gap between SaaS and HCP. For that purpose, it consumes the application services (aPaaS) provided by HCP.

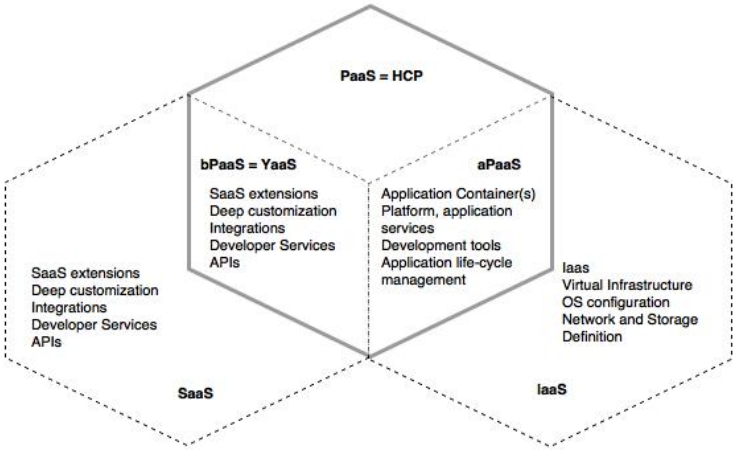

Figure 7.1: YaaS and HCP [Hir15]

7.2 Vision

The vision of YaaS can be clarified with the following statement.

"A cloud platform that allows everyone to easily develop, extend and sell services and applications."
[Stu15]

The vision can be broadly categorized into following objectives.

1. **Cloud First**
   The different parts of the application need to be scaled independently.
2. **Autonomy**
   The development teams should be able to develop their modules independent of other teams and able to freely choose the technology that fits the job.
3. **Retain Speed**
   The new features should be able to be released as fast as possible.
4. **Community**
   It should be possible for the components to be shared across internal and external developers.

The definition of microservices 1.1 as well as the characteristics of microservices [ref] signifies clearly that microservices architecture can be a good fit for YaaS architecture.

7.3 YaaS Architecture Principles

The Agile Manifesto [Bec+11] provides various principles to develop a software in a better way. It focus on fast response to the requirement changes with frequent continous delivery of software artifacts with close collaboration of customer and self-organizing teams.
The Reactive Manifesto [Bon+14] lists various qualities of a reactive system which includes responsiveness, resilience, elasticity and asynchronous message passing. Loose coupling is highly focused.
Furthermore, the twelve factors from Heroku [Wig12] provides methodology for minimizing time and cost to develop software applications as services. It emphasizes on scalability of applications, explicit declaration as well as isolation of dependencies among components, multiple continuous deployments from a single version controlled codebase with separate pipelines for build, release and run.
Finally, the microservices architecture provides techiques of developing an application

as a collection of autonomous small sized services focused on single responsibility. [section 1.3] It focus on independent deployment capability of individual microservices and suggest to use lightweight mechanisms such as http for communication among services. Following the architecture offers various advantages not limited to individual independent scalability of each microservice, resilience by isolating failure in a component and technology heterogeneity among various development teams.[New15] Following the principles mentioned above, a list of principles are compiled to be used as a guidelines for creating microservices. [Stu15] The list of principles are listed below.

1. **Self-Sufficient Teams**
   The teams have independence and freedom for any decision related to the design and development of their components. This freedom is balanced by the responsibility for the team to handle the complete lifecycle of their components including smooth running as well as performance in production and troubleshooting in case of any problems.

2. **Open Technology landscape**
   The team have freedom to choose any technology that they believe fits the requirement. They are completely responsible for the quality of their product. This gives teams, ownership as well as satisfaction for their products.

3. **Release early, release often**
   The agile manifesto and twelve factors from Heroku also focus on continuous delivery of product to the clients. This will decrease time of feedback and ultimately fast response to the feedback resulting in high customer satisfaction. The teams are responsible to create build and delivery pipeline for all available environments.

4. **Responsibility**
   The teams are the only responsible groups to work directly with the customers on behalf of their products. They need to work on the feedback provided by the customers. It will increase the quality of the products and relationship with customers. All the responsibilities including scaling, maintaining, supporting and improving products are handled by respective teams.

5. **APIs first**
   The API is a contract between service and consumers. The decision regarding design and development is very important as it can be one of the greatest assets if good or else can be a huge liability if done bad. [Blo16] The articles [Blo16] and [Bla08] list various characteristics of good API including simplicity, extensibility, maintainability, completeness, small and focussing on single

functionality. Furthermore, a good approach to develop API is to first design iteratively before implementation in order to understand the requirement clearly. Another important aspect of a good API among many is complete and updated documentation.

6. **Predictable and easy-to-use UI**
   The user interfaces should be simple, consistent across the system and also consistent to various user friendly patterns.[Sol12] The articles [Mar13] and [Por16] specify additional principles to be considered when designing user interfaces. A few of them includes providing clarity with regard to purpose, smart organization and respecting the expectation as well as requirements of customers.

7. **Small and Simple Services** A service should be small and focused on cohesive functionalities. The concept closely relates to the single responsibility principle. [Mar16] A good approach is to explicitly create boundaries around business capabilities.[New15]

8. **Scalability of technology**
   The choice of technologies should be cloud friendly such that the products can scale cost-efficiently and without delay. It is influenced by the elasticity principle provided by the reactive manifesto.[Bon+14]

9. **Design for failure** The service should be responsive in the time of failure. It can be possible by containment and isolation of failure within each component. Similarly, the recovery should be handled gracefully without affecting the overal availability of the entire system. [Bon+14]

10. **Independent Services**
    The services should be autonomous. Each service should be able to be deployed independently. The services should be loosely coupled, they could be changed independently of eachother. The concept is highly enforced by exposing functionalities via APIs and using lightweight network calls as only way of communication among services. [New15]

11. **Understand Your System** It is crucial to have a good understanding of problem domain in order to create a good design. The concept of domain driven design strongly motivates this approach to indentify individual autonomous components, their boundaries and the communication patterns among them. [New15] Furthermore, it is also important to understand the expectations of consumers regarding performance and then to realize them accordingly. It is possible only by installing necessary operational capabilities such as continuous delivery, monitoring, scaling and resilience.

7.4 Interviews

With the intension to anticipate the overall belief, culture and practice followed by hybris to develop YaaS, a number of interviews were conducted with a subset of key personnels who are directly involved in YaaS. The list of interviewees with their corresponsing roles is shown in the table 7.1. In order to preserve anonymity, the real names are replaced with forged ones.

Names	Roles
John Doe (Rene)	Product Manager
Ivan Horvat (Sebastian)	Senior Developer
Jane Doe (Andrea)	Product Manager
Mario Rossi (Tomasz)	Product Manager
Nanashi No Gombe (Victor)	Product Manager
Hans Meier (Michael)	Architect
Otto Normalverbraucher (Klaus)	Product Manager
Jan Kowalski (Krzysztof)	Senior Developer

Table 7.1: Interviewee List

7.4.1 Hypothesis

During the process of solving the concerned research questions, various topics of high value have been discovered. The topics are:

1. Granularity
2. Quality Attributes
3. Process to design microservices

Similarly, based on research findings, a list of hypothesis has been made with respect to each topic.

Hypothesis 1:

The correct size of microservices is determined by:

1. **Single Responsibility Principle**

2. Autonomy, and

3. Infrastructure Capability

According S.Newman, a good microservices should be small and focus on accomplishing one thing well. Additionally, it could be independently deployed and updated. This strongly suggests the requirement of autonomy. [New15] Futhermore, M. Stine also agree that the size of a microservices should be determined by single responsibility principle. [Sti14] Also, the principle listed in 2.4.4 indicate that the advancement in the current technology and culture of the organization also defines size of microservices. [RS07]

Hypothesis 2:

The domain driven design is the optimum approach to design microservices.

The concept of domain driven design to design microservices is promoted by S. Newman and M. Fowler. [New15] [FL14] Similary, there can be found evendence of domain driven design being used in various projects to design microservices. The list of projects is shown in the table 6.2.

7.4.2 Interview Compilation

For each topic listed in section 7.4.1, a list of questionaires is prepared. The questions were then asked to the interviewees. In the remaining part of this section, an attempt is made to compile the response from the interviews on the questionaires for each topic.
1. Granularity

Question 1.1

"Do you consider size of microservices when desigining microservice architecture?"

Response 1.1	John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	**Score**
Important	1	1	1	0	1	1	0	0	**5**
Secondary, Not primary	0	0	0	1	0	0	1	1	**3**
Not Important	0	0	0	0	0	0	0	0	**0**

Question 1.2

"How do you measure size of a microservice?"

Response 1.2		John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	**Score**
Qualitatively		1	1	1	1	1	1	1	1	**8**
a.	Functionality	1	1	1	1	1	1	1	1	**8**
b.	Business value	1	1	1	1	1	1	1	1	**8**
Quantitatively		0	0	0	0	0	0	0	0	**0**

Question 1.3

"What kind of size do you try to achieve for a microservice?"

Response 1.3		John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	**Score**
as small as possible in terms of functionality		1	1	1	1	1	1	1	1	**8**
influenced by other attributes										
a.	Business Value	1	1	1	1	1	1	1	1	**8**
b.	Single Responsibility	1	1	1	1	1	1	1	1	**8**
c.	Loose Coupling	0	1	0	0	1	1	0	1	**4**
d.	Cohesion	0	1	0	0	0	0	1	1	**3**
e.	Autonomy	0	1	1	0	0	1	0	0	**3**
f.	Maintainability	0	1	0	0	0	1	1	0	**3**
g.	Reusability	0	1	0	0	1	1	1	0	**4**
h.	Scalability	1	1	0	1	1	1	1	0	**6**
i.	Network Complexity	1	0	0	0	1	1	0	0	**3**
Quantitatively		0	0	0	0	0	0	0	0	**0**

Question 1.4

"Suppose that you do not have the agile culture like continuous integration and delivery automation and also cloud infrastructure. How will it affect your decision regarding microservices and their size?"

Response 1.4	John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	Score
No microservices at all, start with Monolith, extract one microservice at a time as automation matures	1	1	0	1	1	1	1	1	**7**
Start with bigger sized microservices with high business value, low coupling and single Responsibility	0	0	0	1	0	0	1	1	**3**

Question 1.5

"Are you facing any complexities because of microservices architecture and the size you have chosen?"

Response 1.5	John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	Score
communication overhead in network	0	1	0	0	0	0	1	0	**2**
complexity in failure handling	0	1	0	0	0	0	0	0	**1**
Monitoring is difficult	0	1	0	0	0	0	1	0	**1**
Operational support is costly	0	1	1	0	0	0	0	0	**2**
difficult to trace a request	0	1	1	0	0	0	0	0	**2**
updates can be difficult due to dependencies among microservices	0	0	1	0	0	0	0	0	**1**

2. Quality Attributes

Question 2.1

"Do you consider any quality attributes when selecting microservices?"

Response 2.1	John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	Score
Loose Coupling	1	1	1	1	1	1	1	1	**8**
Cohesion	1	1	1	1	1	1	0	1	**7**
Autonomy	1	1	0	0	0	1	1	0	**4**
Scalability	1	0	0	1	1	0	1	1	**5**
Complexity	1	0	0	0	1	0	1	0	**3**
Reusability	0	1	0	0	0	0	1	0	**2**

Question 2.2

"Do you consider any metrics for evaluating the quality attributes?"

Response 2.2	John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	**Score**
No	1	1	1	1	1	1	1	1	**8**
Yes	0	0	0	0	0	0	0	0	**0**

Question 2.3

"Do you think the table of basic metrics can be helpful?"

Response 2.2	John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	**Score**
No	0	0	0	0	0	0	0	0	**0**
Yes	1	0	1	0	1	0	1	0	**4**

3. Process to design microservices

Question 3.1

"Do you follow specific set of consistent procedures across the teams to discover microservices?"

Response 3.1	John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	**Score**
No	1	1	1	1	1	1	1	1	**8**
Yes	0	0	0	0	0	0	0	0	**0**

Question 3.2

"Can you define the process you follow to come up with microservices?"

Response 3.1	John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	**Score**
By brain-storming to divide a usecase into finer functionalities considering business value, cohesion, loose coupling [Single Responsibility], autonomy, reusability and scalability.	1	0	1	1	0	0	1	1	**5**
Do you consider bounded context as well?									
Yes	1	0	1	0	0	0	1	0	**3**
By identifying bounded context using domain-driven-design Understand the problem domain interacting with domain experts and studying the interaction among domain models and flow of data.	0	1	0	0	1	1	0	0	**3**

7.4.3 Interview Reflection on Hypothesis

The data gathered from the interviews which are compiled in the section 7.4.2 can be a good source to analyse the hypothesis listed on the section 7.4.1.

Hypothesis 1

The response from question 7.4.2 has helped to point out some important constraints which play significant role in industry while choosing appropriate size of microservice. It is also interesting to notice how the terms in answer closely relate to the hypothesis 7.4.1. The figure attempts to clarify the relationship among the terms with the hypothesis.

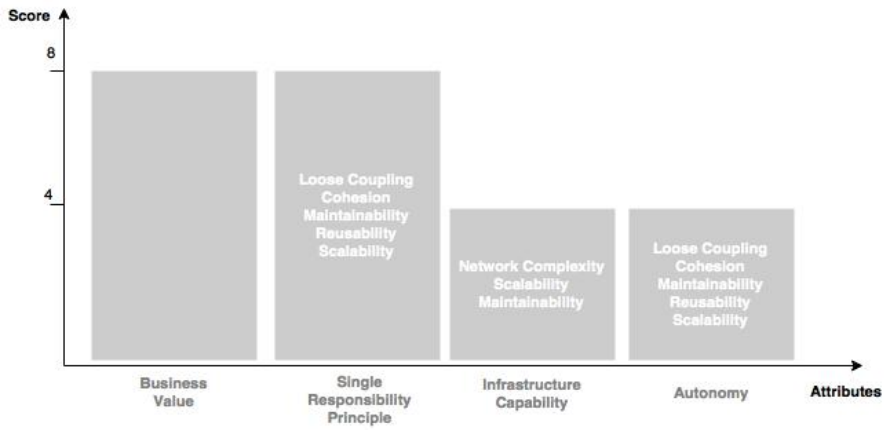

Figure 7.2: Attributes grouping from interview response

The interview unfolded various quality attributes which determine the correct size of a microservice. Moreover, these quality attributes can be classified into three major groups which are:

1. Single Responsibility Principle
2. Automony
3. Infrastructure Capability

Additionally, the response from the question 7.4.2 highly suggest the significance of availability of proper infrastructure to decide about size of microservices and the microservices architecture. This highly moves in the direction to support the hypothesis. Finally, there appears one additional constraint to affect the size of microservices, not covered by hypothesis but mentioned by all interviewees, which is the business value provided by the microservices.

Hypothesis 2

From the response to question 7.4.2, it can be implied that the organization has no consistent set of specific guidelines which are agreed and pratised across all teams. However, the various reactions to the question 7.4.2 appear to fall with two class of process. The response from 7.4.2 can be represented as in figure 7.3 to make it easy to perceive.

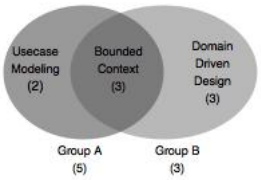

Figure 7.3: Process variations to design microservices

There is one group of reactions which follow a process to functionally divide a usecase into small functions based upon various factors such as single responsiblity and the requirements of performance. This strongly resemble to the usecase modeling technique as described in the chapter 5. Moreever, within the same group, a partition are also using the concept of bounded context to ensure autonomy, which also suggests towards the direction of domain driven design as mentioned in the chapter 6. Finally, the rest of the interviewees agree on domain driven design to be the process to design microservices. So, the response is of mixed nature, both usecase modeling and domain driven design are used in desigining microservices. However, it should also be stated that domain driven design are used on its own or along with usecase modeling to design autonomous microservices.

7.5 Interview Summary

Granularity of a microservice is considered as an important aspect when designing microservices architecture. The first impression in major cases is to make the size of microservice as small as possible however there are also various attributes and concepts which affects the decision regarding the size of microservices. The major aspects are single responsibility principle, autonomy and infrastructure capability.
The single responsibility gives impression to assume the size of a microservice as small as possible so that the service has minimum cohesive functionality and less number of consumers affected. Whereas, in order to make the microservice autonomous, the service should have full control over its resources and less dependencies upon other services for accomplishing its core logic. This suggests that the size of the service should be big enough to cover the transactional boundary upon its core logic and have full control on its business resources.
Moreover, the small the size of the service is, the number of required microservices increases to accomplish the functionalities of the application. This increases network complexity. Additionally, the logical complexity as well as maintainability will increase. Undoubtedly, the independent scalability of individual service will also increase. However, this increases the operational complexity for maintaining and deploying the microservices, due to the number of services and number of environment to be maintained for each services.
The research findings as well as interview response leads to the idea that the question regarding "size" of a microservices has no straight answer. The answer itself is a multiple objective optimization problem whose answer depend upon the level of abstraction of the problem domain achieved, self-governance requirement, self-containment requirement and the honest capability of the organization to handle the operational complexity.
Finally, there is an additional deciding factor called "Business Value". The chosen size of the microservice should give business and competitive advantage to the organization and closely lean towards the organizational goals. A decision for a group of cohesive functionalities to be assigned as a single microservice means that a dedicated resources in terms of development, scaling, maintainance, deployment and cloud resources have to be assigned. A rational decision would be to relate the expected business value of the microservice against the expected cost value required by it.
The teams do not follow any quantitative metrics for measuring various quality attributes such as granularity, coupling and cohesion. According to the reactions from interviews, the basic metrics to evaluate the quality attributes visualized in the table 3.8 can be helpful to approach the design of microservices architecture in an efficient way.
There is no single consistent process followed across all the teams. Each team has

its own steps to come up with microservices. However, the steps and decision are highly influenced by a set of YaaS standard principles 7.3. The reactions from the interviewees are already visualized in 7.3. The process followed by a portion of teams closely relate to usecase modeling approach, where a usecase is divided into several smaller abstractions guided by cohesion, reusability and single responsibility principle. Within this group, a major number of teams also use the concept of bounded concept in order to realize autonomous boundary around the service. Finally, the remaining portion of the teams closely follow domain driven design, where a problem domain is divided into sub-domains and in each sub-domain, various bounded contexts are discovered, which is then mapped into microservices.
The domain driven design can be considerd as the most suitable approach to design microservices. Following this approach, not only the problem domain is functionally divided into cohesive group of functionalities but also leads to a natural boundary around the cohesive functionalities with respect the real world, where concept as well as differences are well understood and the ownership of business resources and core logic are well preserved.

Acronyms

API Application Programming Interface.

CRUD create, read, update, delete.

DEP Dependency.

HCP Hana Cloud Platform.

IDE Integrated Development Environment.

IFBS International Financial and Brokerage Services.

ISCI Inter Service Coupling Index.

ODC Operation Data Granularity.

ODG Operation Data Granularity.

OFG Operation Functionality Granularity.

RCS Relative Coupling of Services.

RIS Relative Importance of Services.

SCG Service Capability Granularity.

SDG Service Data Granularity.

SDLC Software Development Life Cycle.

SFCI Service Functional Cohesion Index.

SIDC Service Interface Data Cohesion.

SIUC Service Interface Usage Cohesion.

SIUC Service Sequential Usage Cohesion.

SLC Self Containment.

SMCI Service Message Coupling Index.

SOAF Service Oriented Architecture Framework.

SOCI Service Operational Coupling Index.

SOG Service Operations Granularity.

SRI Service Reuse Index.

SWIFT Society for Worldwide Interbank Financial Telecommunication.

UML Unified Modeling Language.

YaaS Hybris as a Service.

List of Figures

List of Tables

Bibliography

[Abr14] S. Abram. *Microservices*. Oct. 2014. URL: http://www.javacodegeeks.com/2014/10/microservices.html.

[AL03] M. Alshayeb and W. Li. *An Empirical Validation of Object-Oriented Metrics in Two Different Iterative Software Processes*. Tech. rep. IEEE Computer Society, 2003.

[Ann14] R. Annett. *What is a Monolith?* Nov. 2014.

[Ars04] A. Arsanjani. *Service-oriented modeling and architecture How to identify, specify, and realize services for your SOA*. Tech. rep. IBM Software Group, 2004.

[Ars05] A. Arsanjani. *How to identify, specify, and realize services for your SOA*. Tech. rep. IBM, 2005.

[AZR11] S. Alahmari, E. Zaluska, and D. C. D. Roure. *A Metrics Framework for Evaluating SOA Service Granularity*. Tech. rep. School of Electronics and Computer Science University Southampton, 2011.

[Bea15] J. Beard. *State Machines as a Service: An SCXML Microservices Platform for the Internet of Things*. Tech. rep. McGill University, 2015.

[Bec+11] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. *Manifesto for Agile Software Development*. 2011.

[BHJ15] A. Balalaie, A. Heydarnoori, and P. Jamshidi. *Migrating to Cloud-Native Architectures Using Microservices: An Experience Report*. Tech. rep. Sharif University of Technology and Imperial College London, 2015.

[BKM07] P. Bianco, R. Kotermanski, and P. F. Merson. *Evaluating a Service-Oriented Architecture*. Tech. rep. Carnegie Mellon University, 2007.

[Bla08] J. Blanchette. *The Little Manual of API Design*. June 2008.

[Blo16] J. Bloch. *How to Design a Good API and Why it Matters*. Jan. 2016.

[BMB96] L. C. Briand, S. Morasca, and V. R. Basili. *Property-Based Software Engineering Measurement*. Tech. rep. IEEE Computer Society, 1996.

[Bon+14] J. Bonér, D. Farley, R. Kuhn, and M. Thompson. *The Reactive Manifesto*. Sept. 2014.

[Bri+np] L. C. Briand, J. Daly, V. Porter, and J. Wüst. *A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems*. Tech. rep. Fraunhofer IESE, np.

[Che+05] F. Chen, S. Li, H. Yang, C.-H. Wang, and W. C.-C. Chu. *Feature Analysis for Service-Oriented Reengineering*. Tech. rep. De Montfort University, National Chiao Tung University, and TungHai University, 2005.

[Coc01] A. Cockburn. *WRITING EFFECTIVE USE CASES*. Tech. rep. Addison-Wesley, 2001.

[Coc15] A. Cockcroft. *State of the Art in Mircroservices*. Feb. 2015. URL: `http://www.slideshare.net/adriancockcroft/microxchg-microservices`.

[Day+15] S. Daya, N. V. Duy, K. Eati, C. M. Ferreira, D. Glozic, V. Gucer, M. Gupta, S. Joshi, V. Lampking, M. Martins, S. Narain, and R. Vennam. *Microservices from Theory to Practice Creating Applications in IBM Bluemix Using the Microservices Approach*. IBM, Aug. 2015.

[DK07] K.-G. Doh and Y. Kim. *The Service Modeling Process Based on Use Case Refactoring*. Tech. rep. Hanyang University, 2007.

[EAK06] A. Erradi, S. Anand, and N. Kulkarni. *SOAF: An Architectural Framework for Service Definition and Realization*. Tech. rep. University of New South Wales, Infosys Technologies Ltd, 2006.

[EM14] A. A. M. Elhag and R. Mohamad. *Metrics for Evaluating the Quality of Service-Oriented Design*. Tech. rep. Universiti Teknologi Malaysia, 2014.

[Emi+np] C. Emig, K. Langer, K. Krutz, S. Link, C. Momm, and S. Abeck. *The SOA's Layers*. Tech. rep. Universität Karlsruhe, np.

[Eng+np] G. Engels, A. Hess, B. Humm, O. Juwig, M. Lohmann, J. Richter, M. Voß, and J. Willkomm. *A METHOD FOR ENGINEERING A TRUE SERVICE-ORIENTED ARCHITECTURE*. Tech. rep. Software Design and Management Research, np.

[Erl05] T. Erl. *Service-Oriented Architecture Concepts, Technology, and Design*. Prentice Hall Professional Technical Reference, 2005.

[Eva03] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

[FA15] M. T. Fisher and M. L. Abbott. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise, Second Edition*. Addison-Wesley Professional, 2015.

[Far08] N. Fareghzadeh. *Service Identification Approach to SOA Development*. Tech. rep. World Academy of Science, Engineering and Technology, 2008.

[Feu13] G. Feuerlicht. *Evaluation of Quality of Design for Document-Centric Software Services*. Tech. rep. University of Economics and University of Technology, 2013.

[FL07] G. Feuerlicht and J. Lozina. *Understanding Service Reusability*. Tech. rep. University of Technology, 2007.

[FL14] M. Fowler and J. Lewis. *Microservices*. Mar. 2014. URL: `http://martinfowler.com/articles/microservices.html`.

[Foo05] D. Foody. *Getting web service granularity right*. 2005.

[Fow03] M. Fowler. *AnemicDomainModel*. Nov. 2003.

[Fow06] M. Fowler. *UbiquitousLanguage*. Oct. 2006. URL: `http://martinfowler.com/bliki/UbiquitousLanguage.html`.

[Fow14] M. Fowler. *BoundedContext*. Jan. 2014.

[GL11] A. Goeb and K. Lochmann. *A software quality model for SOA*. Tech. rep. Technische Universität München and SAP Research, 2011.

[Gor13] L. Gorodinski. *Sub-domains and Bounded Contexts in Domain-Driven Design (DDD)*. Sept. 2013. URL: `http://gorodinski.com/blog/2013/04/29/sub-domains-and-bounded-contexts-in-domain-driven-design-ddd/`.

[Gup15] A. Gupta. *Microservices, Monoliths, and NoOps*. Mar. 2015.

[Hae+np] R. Haesen, M. Snoeck, W. Lemahieu, and S. Poelmans. *On the Definition of Service Granularity and Its Architectural Impact*. Tech. rep. Katholieke Universiteit Leuven, np.

[Hir15] R. Hirsch. *YaaS: Hybris' next generation cloud architecture surfaces at SAP's internal developer conferences*. Mar. 2015. URL: `http://scn.sap.com/community/cloud/blog/2015/03/03/yaas-hybris-next-generation-cloud-architecture-surfaces-at-sap-s-internal-developer-conferences`.

[How+09] Y. Howard, N. Abbas, D. E. Millard, H. C. Davis, L. Gilbert, G. B. Wills, and R. J. Walters. *Pragmatic web service design:An agile approachwith the service responsibility and interaction designmethod*. Tech. rep. University of Southampton, 2009.

[HS00] P. Herzum and O. Sims. *Business Components Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*. John Wiley Sons, 2000.

[HS90] S. Henry and C. Selig. *Predicting Source=Code Complexity at the Design Stage*. Tech. rep. Virginia Polytechnic lnstirure, 1990.

[Hug14] G. Hughson. *Microservices and Data Architecture – Who Owns What Data?* June 2014. URL: `https://genehughson.wordpress.com/2014/06/20/microservices-and-data-architecture-who-owns-what-data/`.

[Jac03] I. Jacobson. *Use Cases and Aspects - Working Seamlessly Together*. Tech. rep. Rational Software Corporation, 2003.

[Jac87] I. Jacobson. *Object Oriented Development in an Industrial Environment*. Tech. rep. Royal Institute of Technology, 1987.

[Jos07] N. M. Josuttis. *SOA in Practice The Art of Distributed System Design*. O'Reilly Media, 2007.

[JSM08] P. Jamshidi, M. Sharifi, and S. Mansour. *To Establish Enterprise Service Model from Enterprise Business Model*. Tech. rep. Amirkabir University of Technology, Iran University of Science, and Technology, 2008.

[Kan+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature Oriented Domain Analysis (FODA)*. Tech. rep. Carnegie-Mellon University, 1990.

[Kaz+11] A. Kazemi, A. N. Azizkandi, A. Rostampour, H. Haghighi, P. Jamshidi, and F. Shams. *Measuring the Conceptual Coupling of Services Using Latent Semantic Indexing*. Tech. rep. Automated Software Engineering Research Group, 2011.

[Kee91] P. G. Keen. *Shaping The Future of Business Design Through Information Technology*. Harvard Business School Press, 1991.

[KJP15] A. Krylovskiy, M. Jahn, and E. Patti. *Designing a Smart City Internet of Things Platform with Microservice Architecture*. Tech. rep. Fraunhofer FIT and Politecnico di Torino, 2015.

[KY06] Y. Kim and H. Yun. *An Approach to Modeling Service-Oriented Development Process*. Tech. rep. Sookmyung Women's University, 2006.

[Lee+01] J. K. Lee, S. J. Jung, S. D. Kim, W. H. Jang, and D. H. Ham. *Component Identification Method with Coupling and Cohesion*. Tech. rep. Soongsil University and Software Quality Evaluation Center, 2001.

[Linnp] D. Linthicum. *Service Oriented Architecture (SOA)*. np.

[Ma+09] Q. Ma, N. Zhou, Y. Zhu, and H. Wang1. *Evaluating Service Identification with Design Metrics on Business Process Decomposition*. Tech. rep. IBM China Research Laboratory and IBM T.J. Watson Research Center, 2009.

[Mac14] L. MacVittie. *The Art of Scale: Microservices, The Scale Cube and Load Balancing*. Nov. 2014.

[Man+np] M. Mancioppi, M. Perepletchikov, C. Ryan, W.-J. van den Heuvel, and M. P. Papazoglou. *Towards a Quality Model for Choreography*. Tech. rep. European Research Institute in Services Science, Tilburg University, np.

[Mar13] S. Martin. *Effective Visual Communication for Graphical User Interfaces*. Jan. 2013.

[Mar16] R. C. Martin. *The Single Responsibility Principle*. Jan. 2016. URL: `http://programmer.97things.oreilly.com/wiki/index.php/The_Single_Responsibility_Principle`.

[Mau15] T. Mauro. *Adopting Microservices at Netflix: Lessons for Architectural Design*. Feb. 2015.

[Mik12] Mike. *DDD - The Bounded Context Explained*. Apr. 2012.

[Mil+01] H. Mili, A. Mili, S. Yacoub, and E. Addy. *Reuse-based software engineering: techniques, organization, and controls*. Wiley-Interscience New York, 2001.

[Mil+07] D. E. Millard, H. C. Davis, Y. Howard, L. Gilbert, R. J. Walters, N. Abbas, and G. B. Wills. *The Service Responsibility and Interaction Design Method: Using an Agile approach for Web Service Design*. Tech. rep. University of Southampton, 2007.

[Mit05] K. Mittal. *Service Oriented Unified Process(SOUP)*. Tech. rep. IBM, 2005.

[MLS07] Y.-F. Ma, H. X. Li, and P. Sun. *A Lightweight Agent Fabric for Service Autonomy*. Tech. rep. IBM China Research Lab and Bei Hang University, 2007.

[Nav08] V. D. Naveen Kulkarni. *The Role of Service Granularity in A Successful SOA Realization – A Case Study*. Tech. rep. SETLabs, Infosys Technologies Ltd, 2008.

[Nem+14] H. Nematzadeh, H. Motameni, R. Mohamad, and Z. Nematzadeh. *QoS Measurement of Workflow-Based Web Service Compositions Using Colored Petri Net*. Tech. rep. Islamic Azad University Sari Branch and Universiti Teknologi Malaysia, 2014.

[New15] S. Newman. *Building Microservices*. O'Reilly Media, 2015.

[Nig05] S. Nigam. *Service Oriented Development of Applications(SODA) in Sybase Workspace*. Tech. rep. Sybase Inc, 2005.

[NJ04] P.-W. Ng and I. Jacobson. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley Professional, 2004.

[np01] np. *ISO/IEC 9126-1 Software Engineering Product Quality – Quality Model*. Tech. rep. International Standards Organization, 2001.

[NS14] D. Namiot and M. Sneps-Sneppe. *On Micro-services Architecture*. Tech. rep. Open Information Technologies Lab, Lomonosov Moscow State University, 2014.

[Pad04] F. Z. Padmal Vitharana Hemant Jain. *Strategy-Based Design of Reusable Business Components*. Tech. rep. IEEE, 2004.

[Per+07] M. Perepletchikov, C. Ryan, K. Frampton, and Z. Tari. *Coupling Metrics for Predicting Maintainability in Service-Oriented Designs*. Tech. rep. RMIT University, 2007.

[Por16] J. Porter. *Principles of User Interface Design*. Jan. 2016.

[PRF07] M. Perepletchikov, C. Ryan, and K. Frampton. *Cohesion Metrics for Predicting Maintainability of Service-Oriented Software*. Tech. rep. RMIT University, 2007.

[Ric14a] C. Richardson. *Microservices: Decomposing Applications for Deployability and Scalability*. May 2014.

[Ric14b] C. Richardson. *Pattern: Microservices Architecture*. 2014.

[Ric14c] C. Richardson. *Pattern: Monolithic Architecture*. 2014. URL: `http://microservices.io/patterns/monolithic.html`.

[Rig15] J. Riggins. *Building Microservices, Toiling With the Monolith and What it Takes to Get it Right*. Aug. 2015. URL: `http://thenewstack.io/building-microservices-toiling-with-the-monolith-and-what-it-takes-to-get-it-right/`.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Professional, 1999.

[RKKnp] C. Rolland, R. S. Kaabi, and N. Kraiem. *On ISOA: Intentional Services Oriented Architecture*. Tech. rep. Université Paris, np.

[Ros+11] A. Rostampour, A. Kazemi, F. Shams, P. Jamshidi, and A. Azizkandi. *Measures of Structural Complexity and Service Autonomy*. Tech. rep. Shahid Beheshti University GC, 2011.

[RS07] P. Reldin and P. Sundling. *Explaining SOA Service Granularity– How IT-strategy shapes services*. Tech. rep. Linköping University, 2007.

[RTS15] G. Radchenko, O. Taipale, and D. Savchenko. *Microservices validation: Mjolnirr platformcase study*. Tech. rep. Lappeenranta University of Technology, 2015.

[Shi+08] B. Shim, S. Choue, S. Kim, and S. Park. *A Design Quality Model for Service-Oriented Architecture*. Tech. rep. Sogang University, 2008.

[Sim05] O. Sims. *Developing the architectural framework for SOA - part 2-service granularity and dependency management. CBDI Forum Journal*. Tech. rep. CBDI, 2005.

[Sok15] B. Sokhan. *Domain Driven Design for Services Architecture*. Aug. 2015.

[Sol12] K. Sollenberger. *10 User Interface Design Fundamentals*. Aug. 2012. URL: `http://blog.teamtreehouse.com/10-user-interface-design-fundamentals`.

[SR01] K. Scott and D. Rosenberg. *Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example*. Addison-Wesley Professional, 2001.

[SSPnp] R. Sindhgatta, B. Sengupta, and K. Ponnalagu. *Measuring the Quality of Service Oriented Design*. Tech. rep. IBM India Research Laboratory, np.

[Ste06] C. Steghuis. *Service Granularity in SOA Projects: A Trade-off Analysis*. Tech. rep. University of Twente, 2006.

[Sti14] M. Stine. *microservices are solid*. June 2014. URL: `http://www.mattstine.com/2014/06/30/microservices-are-solid/`.

[Stu15] A. Stubbe. *Hybris-as-a-Service: A Microservices Architecture in Action*. May 2015.

[Ver13] V. Vernon. *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013.

[Vie+15] N. Viennot, M. Lecuyer, J. Bell, R. Geambasu, and J. Nieh. *Synapse: A Microservices Architecture for Heterogeneous-Database Web Applications*. Tech. rep. Columbia University, 2015.

[WB98] P. Weill and M. Broadbent. *Leveraging The New Infrastructure: How Market Leaders Capitalize on Information Technology*. Harvard Business School Press, 1998.

[Wig12] A. Wiggins. *THE TWELVE-FACTOR APP*. Jan. 2012. URL: `http://12factor.net/`.

[Woo14] B. Wootton. *Microservices - Not A Free Lunch!* Apr. 2014. URL: `http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html`.

[WV04] L. Wilkes and R. Veryard. "Service-Oriented Architecture: Considerations for Agile Systems." In: *np* (2004).

[WXZ05] Z. Wang, X. Xu, and D. Zhan. *A Survey of Business Component Identification Methods and Related Techniques*. Tech. rep. International Journal of Information Technology, 2005.

[WYF03] H. Washizakia, H. Yamamoto, and Y. Fukazawa. *A metrics suite for measuring reusability of software components*. Tech. rep. Waseda University, 2003.

[Xianp] W. Xiao-jun. *Metrics for Evaluating Coupling and Service Granularity in Service Oriented Architecture*. Tech. rep. Nanjing University of Posts and Telecommunications, np.

[YK06] H. Yun and Y. Kim. *Service Modeling in Service-Oriented Engineering*. Tech. rep. Sookmyung Women's University, 2006.

[Zim+05] O. Zimmermann, V. Doubrovski, J. Grundler, and K. Hogg. *Service-Oriented Architecture and Business Process Choreography in an Order Management Scenario: Rationale, Concepts, Lessons Learned*. Tech. rep. IBM Software Group and IBM Global Services, 2005.

[ZL09] Q. Zhang and X. Li. *Complexity Metrics for Service-Oriented Systems*. Tech. rep. Hefei University of Technology, 2009.