Chapter 3
Developing Application Architecture

The previous chapters introduce the notion of EA and architecture management (Chap. 1) as well as the business architecture (Chap. 2), which deals with providing mechanisms for describing the business. This chapter will have a look at the application architecture. We basically deal with concepts for describing applications, which information about applications are quite relevant, and how applications belong to business capabilities.

Learning Objectives

After completing this chapter, you will be able to . . .

- . . . explain the relationships between business and application architecture
- . . . create an application landscape
- . . . classify and document interactions between applications
- . . . explain the notion of data architecture

The learning objectives are given above. After finishing this chapter, you should be capable of explaining the relationship between business architecture on one side and application architecture on the other. We will basically focus on the relationships between applications and business capabilities. Section 3.2 will further present a method for deriving an ideal application landscape from the business capability map. You will also learn how to create an application landscape for a company. This will be drawing a map showing all the applications of an organisation, including their relationships. These relationships are not anonymous but can be of certain types. We will use data flows and services for describing interactions between applications. You will be capable of identifying them and also classifying interactions according to those types. Section 3.4 introduces the notion of *data architecture* as being part of an EA. It will base on the business object model

J. Jung, B. Fraunholz, *Masterclass Enterprise Architecture Management*,
https://doi.org/10.1007/978-3-030-78495-9_3

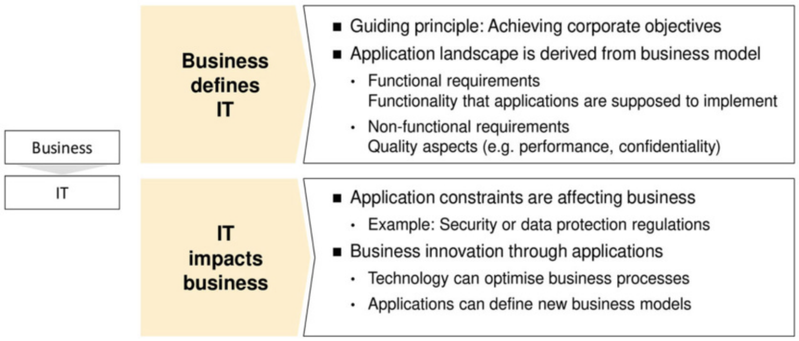

Business
IT
Business defines IT
Guiding principle: Achieving corporate objectives
Application landscape is derived from business model
Functional requirements
Functionality that applications are supposed to implement
Non-functional requirements
Quality aspects (e.g. performance, confidentiality)
IT impacts business
Application constraints are affecting business
Example: Security or data protection regulations
Business innovation through applications
Technology can optimise business processes
Applications can define new business models

Fig. 3.1 Business-driven IT

introduced in the previous chapter (Sects. 2.3 and 3.4 cover more implementation specific data models).

3.1 Application Architecture

Let us reiterate, what EAM is all about as depicted in Fig. 3.1. As emphasised in Chap. 1, we want to derive our application landscape from business needs on a holistic basis. We want to understand the business needs so that we can define the IT landscape—more specific: the applications—required for supporting our business. The basic direction so far was from top understanding the business to bottom we define the IT. Nevertheless, we should always keep in mind that software systems and innovations in IT can also influence the business. For example, by introducing a process-based system, we can automate our business processes. There will be less manual work which will impact the work people are supposed to do.

We will see in Chap. 4 that any kind of change in the application landscape will have an impact on the business side. The direction from top to bottom for us will be the driver. The requirements from the top will drive how IT needs to change or needs to behave. But at the same time, we are always aware of the fact that IT will have an impact on the business side.

The basic artefact discussed in this section is the *application architecture* as defined by Definition 3.1. Sometimes it is also referred to as *application layer* as already introduced in Sect. 1.4 while describing the three-layered structure for EA. It encompasses all software applications of an organisation together with their relationships—interfaces between applications and relationship to the business context. We will not discuss applications as a means on their own, but keep their value for business in mind.

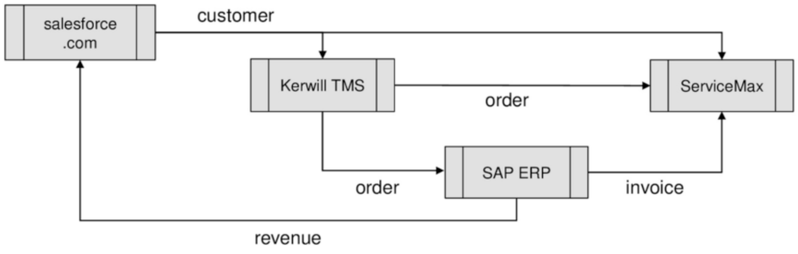


Fig. 3.2 Example application landscape with data flow

Definition 3.1 (Application Architecture) The application architecture (also referred to as application layer) encompasses all software applications of an organisation together with:

- relationship to the business architecture
- interfaces between applications

Chapter 1 already introduced typical EA maps as for example, process maps for showing business processes. There is the so-called application landscape, which is a map showing the application architecture, including all the relationships, and including certain data that is required for managing applications. We will now use these maps for explaining application landscapes by example.

The example in Fig. 3.2 shows an application landscape consisting of four systems exchanging data. Data is represented by business objects assigned to lines between applications. The arrow head at the end of each line indicates the direction of the data flow. It is the same kind of business objects we introduced in Sect. 2.3 as part of the business architecture. There are business objects **customer**, **order**, **invoice** and **revenue**.[1] Business objects have special meaning and relevance for business people, but can also be used for describing information flows between applications. In this case, the customer relationship management system called *salesforce.com* provides customer data, represented by the business object customer, to a transport management system and a customer service system.

I am sure you also remember the map in Fig. 3.3 as it is one of the examples from Sect. 1.3. It shows applications assigned to individual organisational units (i.e. application is used by an organisational unit). We have three organisational units. One is representing the global headquarters and we have subsidiaries in the two regions Europe and Asia-Pacific. The map can be used for getting an overview on applications used in various organisational units. They might share common software systems but there might also be clear redundancies. If subsidiaries are allowed to procure their own IT, this usually leads to huge amount of individual

[1]Customer, order and invoice have already been introduced before. *Revenue* represents the income of a company.

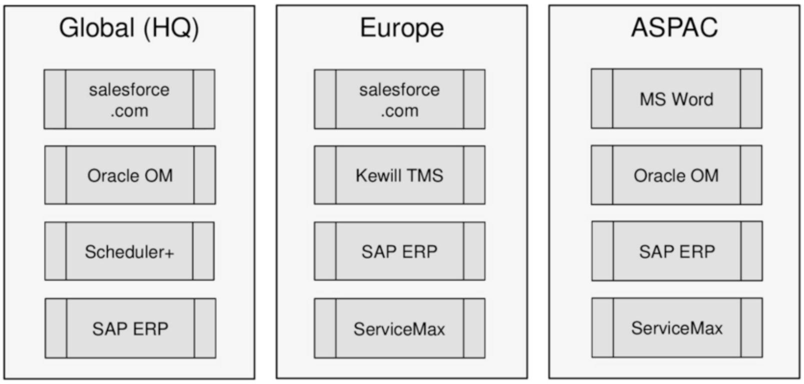


Fig. 3.3 Application landscape—Example context

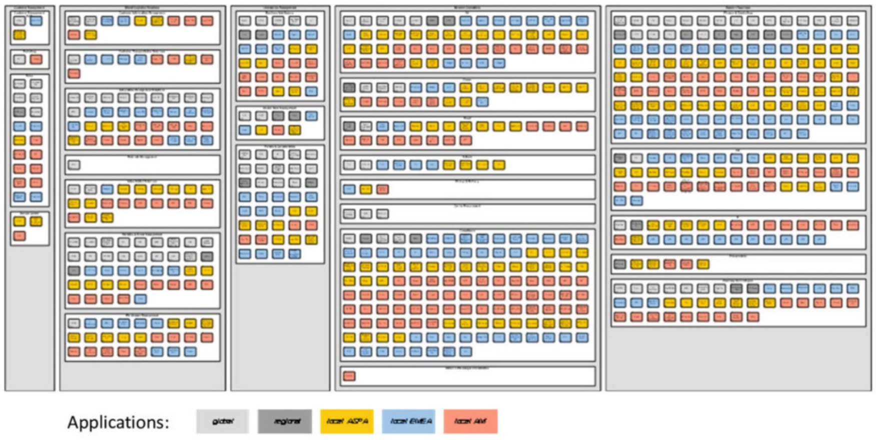


Fig. 3.4 Example application landscape

software.[2] The map in Fig. 3.3 can be created easily—manually or by an EA tool. Applications located within the box representing an organisational unit then this organisational unit is using this application.

We should not forget that real-world application landscapes tend to be large[3] and complex.[4] Even though most of the examples in this textbook are rather small, they are only focusing on showing you the concepts and how to use the concepts. They

[2]We should not forget the vessel in Fig. 1.6 on page 11. Having no strict Governance on IT can result in a plethora of (redundant) systems and, therefore, increase IT budgets.

[3]Corporate application landscapes can have a couple of hundred or more than thousand software applications.

[4]Each application is potentially connected to several others, resulting in thousand of interfaces.

are far away from the size of real-world maps and this is why we are using the map in Fig. 3.4 again.

It is showing applications (represented by the small coloured rectangles) mapped against business capabilities(grey and rectangular white areas as the background). You can also see the strengths of having structured maps. You can already get an overview on the functional distribution of applications. There are capabilities with many applications while others are only supported by a single one. If you have an additional colour coding, as provided here, you can even show some more information. The example here is showing whether it is a legacy application, a global application, specific for a region, or if it is only needed for certain products.

And in fact, this is an application landscape that was the result of a project. And this application landscape, you could see in each office of each of the top management people. They were using it as an overview for having an indication about the complex IT they are managing. During that time, the poster was showing the current situation, hence, it was an *as-is* application architecture. The same kind of visualisation can also be applied for showing how the application architecture should look like in the future. This is then called *to-be*. In the following section, we will introduce a method that will help with defining an ideal application landscape. It does not involve the as-is architecture but aims at deriving applications from business capabilities. Therefore, the relationship between capabilities and applications will be shown, following the direction *business drives IT*.

3.2 Deriving the Application Landscape from Capabilities

The focus of the notion of EA within this textbook is on business-IT alignment. We want to understand how to implement an IT landscape that will support the business adequately. The concept of business capabilities has been introduced as one of the key concepts on how to describe—and understand—the business part of the company (cf. Sect. 2.2). We will now further elaborate on how an *ideal application landscape* should look like.

This ideal landscape will be derived from the business capabilities. The method presented here will not incorporate any existing applications but derive the applications from the business capability map only. This might look a bit unrealistic as companies usually have an application landscape in place. However, the method here aims at . . .

- . . . creating an ideal state that can also be used as the long-term to-be architecture.
- . . . demonstrating how decisions on applications can be made just based on business needs.

The simple example in Fig. 3.5 illustrates the example of what the method is aiming at. The map shows four business capabilities we already used in previous sections. It contains **Market development** with all the marketing activities, **Logistics operations** for planning transports and moving parcels, **Supplier management**

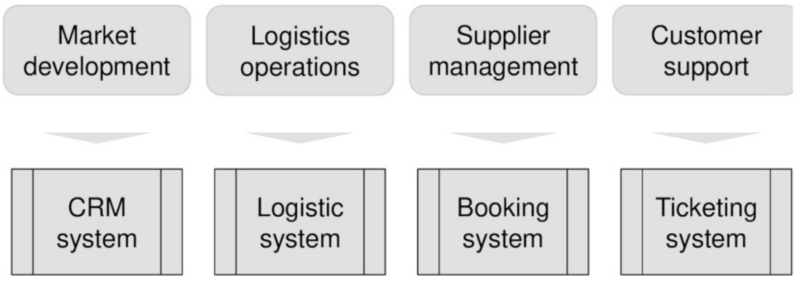


Fig. 3.5 Example capabilities and applications

for monitoring and controlling external partners, and the **Customer support** capability. This is a common map for showing the links between capabilities and aplications, showing the capabilities on top and supporting applications directly below each of them. A **CRM system** supporting marketing development, our **Logistics system** supporting logistics management, **Booking system** for supplier management, and a **Ticketing system** for customer support.

The map looks very straight forward as we have a dedicated system for each business capability. There are no redundancies or overlaps and each application has a distinct set of functionality.[5] Furthermore, each capability is supported by an application so that we are not experiencing any gaps in the application landscape. This example illustrated what we would expect from a method for deriving an ideal application landscape from business capabilities.

There are two drivers for an ideal application landscape for supporting business capabilities:

1. each automated capability is supported by a corresponding software application
2. avoid redundant applications for individual capabilities

The first one ensures that we are *effectively* supporting the business with applications. We do not miss any automation potential and provide quality systems for the company. The second driver enforces **efficiency** for application support. We are reducing run cost by avoiding redundant systems. We want to support making a conscious decision on how a minimal application landscape should look like—minimal with respect to minimising cost—but still making sure that all the automated capabilities are supported by IT.

[5]Needless to say, that such a map is rather the exception as real architectures are much larger and contain a lot of issues.

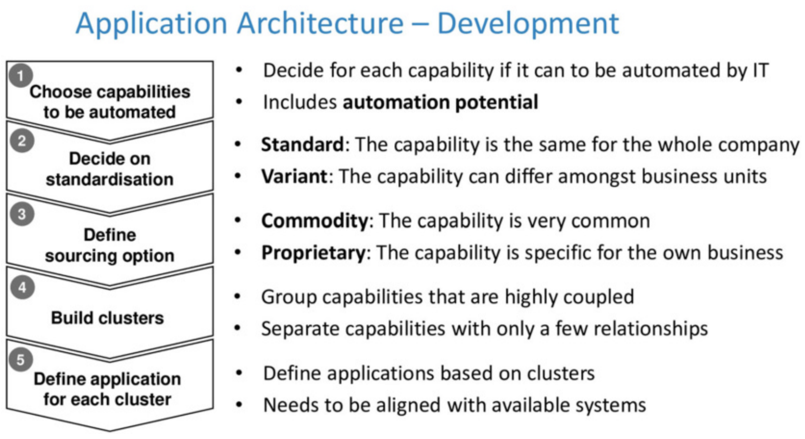

Application Architecture – Development
1 Choose capabilities to be automated
2 Decide on standardisation
3 Define sourcing option
4 Build clusters
5 Define application for each cluster
• Decide for each capability if it can to be automated by IT
• Includes **automation potential**
• **Standard**: The capability is the same for the whole company
• **Variant**: The capability can differ amongst business units
• **Commodity**: The capability is very common
• **Proprietary**: The capability is specific for the own business
• Group capabilities that are highly coupled
• Separate capabilities with only a few relationships
• Define applications based on clusters
• Needs to be aligned with available systems

Fig. 3.6 Developing an application architecture (overview)

3.2.1 Method Overview

The method presented here is not a standard methodology from a textbook. It has been developed in the context of an international logistics corporation. The company has a large IT department and can also develop software for core processes. An overview on the method is shown in Fig. 3.6 and will be explained here. Subsequent sections then have a spot on the details of some of the activities.

It starts with a complete business capability map of the organisation. The first step is to determine all capabilities that will have to be supported by IT applications (step 1 in Fig. 3.6). There is no general rule for such a decision as each organisation is different. This decision needs to be made together by business and IT people and has to be aligned with corporate strategy. Application support does not mean that the capability is fully automated but only that applications are required—among other resources.

First step: Assess for each capability if it needs application support or not.

After we made this decision on automation for each and every capability the next step will address standardisation (i.e. standard software for the whole company) as shown in step 2 in Fig. 3.6. It will be decided upon the answers to the following questions:

- Is the capability the same all over the company?
- Is the capability the same, independent of the corresponding country, organisational units or product?
- Is the capability always associated with the same business objects as input and output?

It is basically about making the decision whether we should prefer to have only one standard system. Benefits of a standard system as opposed to providing various redundant systems for the same capability are as follows:

- Reduced run cost (e.g. hosting, support, incident handling) as only one system needs to be supported
- Standardised training material and user trainings throughout the organisation
- New requirements only need to be incorporated into one software application instead of several ones
- Single source for all information related to the capability instead of data being spread over several applications

However, a standard system might not be sufficient for supporting the whole corporation. There might be legal requirements or physical restriction that require a differing application support for a single capability. This can be illustrated by some examples:

- Each country has legal requirements of its own for customs processing. This can not be provided by a single system globally but requires individual systems in each country.
- Automation of mail sorting differs between letters and parcels. Different software will be required for each.
- Payroll management has to follow local legislation so that country-specific systems will be required.
- Industry-specific compliance rules can impose special rules for different products.
- Different business partners may request for different systems for e-business integration (different formats and protocols)

An example for the last bullet point is a logistics company. They are doing bookings with external transport service providers. There might be two different systems necessary; one for airfreight bookings for interacting with the airlines and another system being responsible for making bookings with ocean lines The capability *Booking management* is the same but there are some differences. Especially the standards for interacting with airlines are different from the ones interacting with ocean freight carriers.

Second step: Go for a standard system unless there are relevant business reasons.

The next step (step 3 in Fig. 3.6) will consist of a make-or-buy decision. This step is only valid for organisations that have skills and capacity for developing a system on their own. If this is not the case, then it can be skipped. However, large corporations usually have their own development teams or can manage the development of custom-built systems together with external software development partners.

Buying a standard software system is usually a more cost-efficient option compared to a complete new development. Let us assume that most companies are trying to reduce IT-related cost, so that buying is supposed to be the preferred option—

even with own development capacities. A CRM system is, for example, a standard software system and there are many products available on the market, fulfilling requirements of different kinds of companies. Sales and marketing activities are very much standardised in the way how they are performed all over the industries. Consequently, there might not be a strong need for having a completely different customer relationship management systems. We are going for the standard by introducing a commodity system. You know the cost for it, and you know what the system can provide and also what it is not providing.

However, there are some capabilities that have an outstanding business relevance: business capabilities that make a difference between us and the competition, different industries or companies. There might be no system available, or we want to have a better system for this capability because this is a differentiator for us from the competition.

Developing a system on your own is more expensive than buying a commodity. You need to have software development skills and competencies. You need to manage the requirements. You have to manage risk during development because the development project might fail or being out of scope or not achieving the objectives as expected. In order to make a make-or-buy decision, you need to find a balance between business relevance on one side (make) as well as cost and risk on the other side (buy). If you need more flexibility or if you want to be different, developing your own system will accrue more cost and will also be some risk. But we will then have the benefit of a unique system that helps us with differentiating from your competition.

Third step: Software applications should be acquired on the market unless there is a clear business reason for having a unique system.

At the current stage we made decisions on which capabilities to support with IT. For those to be supported we decide whether we are going for a corporate standard system or allow for local variations. We might need different systems for a single capability in certain cases. We also made a decision on either buying or making the system. After making those decisions we should check if there are clusters of similar decisions (step 4 in Fig. 3.6). There should be capabilities that belong together because we made similar decisions or because they are very similar from a functional/data perspective.

One example might be marketing, sales and customer service capabilities. Even though they are three distinct capabilities, they are still closely related because marketing is generating all the contracts that are required for sales. Sales can be tightly coupled with customer service as they both interact with customers. They share a common business object: *customer*. And while sales is focusing on generating new orders, customer service is dealing with supporting the customer after or while buying a product. Both are working on the same kind of data, the customer accounts, they both need the same kind of information.[6] In today's world many CRM systems combine marketing, sales and customer service functionality.

The same usually holds true for other capabilities. Transport management, logistics operations and booking are closely related to each other because they are working with the same kind of business objects. We can make decisions on capabilities that relate to each other that build a group that cannot be easily separated from a business perspective. Each groups of related capabilities will build a cluster.

Fourth step: Build clusters of capabilities based on:

- same decisions made during previous steps
- related functions
- shared business objects

Decisions on potential software applications for each cluster will be made during step 5:

1. Each cluster with standard capabilities will be supported by a single application.
2. Each cluster allowing for local variants can have several applications of the same type.

The resulting application landscape can consist of concrete applications (e.g. *salesforce.com*) or application types instead of specific product names (e.g. *Customer Relationship Management*). An example landscape is shown in Fig. 3.10 on page 89. Before discussing it, we will have a look at three of the activities here. Let us start with step number two, the decision on the standardisation of applications.

3.2.2 Standardisation

Driven by the idea of cost reduction for the application landscape, we aim at reducing the number of systems and reducing the complexity of the whole landscape. The driving factor is in getting rid of systems for many IT organisations. One option is having one standard system for the whole corporation. This is not easy to achieve as software applications are directly connected with business processes and, therefore, are used for generating value. Please, remember the complexity of the IT landscape after a merger, having a company that has been merged by buying several other companies. This might lead to the fact that we have redundant CRM systems or systems for payroll management.

One indication for a standard system might be if this is a system that requires collaboration throughout all organisational units. Whenever those indications are given, then the recommendation for the application would be: implementing one central application. You can imagine the saving potential if we decide for a stan-

[6]The decisions should be made on business objects. We refer to data as business objects as it represents business data on a high level. We will further elaborate on this in Sect. 3.4.

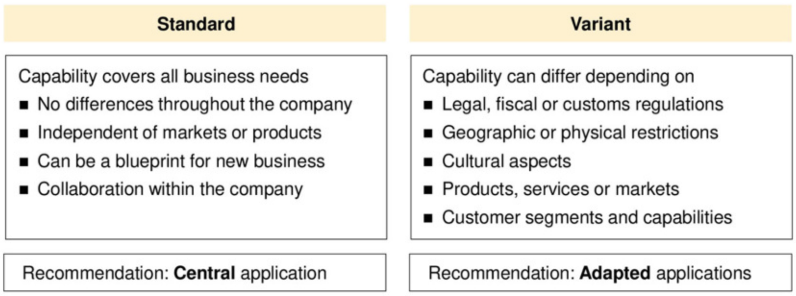

Standard
Capability covers all business needs
■ No differences throughout the company
■ Independent of markets or products
■ Can be a blueprint for new business
■ Collaboration within the company
Recommendation: **Central** application
Variant
Capability can differ depending on
■ Legal, fiscal or customs regulations
■ Geographic or physical restrictions
■ Cultural aspects
■ Products, services or markets
■ Customer segments and capabilities
Recommendation: **Adapted** applications

Fig. 3.7 Application architecture: Standardisation

dardised CRM system. Unless there is a relevant business reason, an organisation should introduce a system that can be used throughout the whole organisation—especially for those capabilities that . . .

- do not have any differences throughout the company
- are independent of any markets or products
- serve as a blueprint for a new business

Let us take a parcel logistics company as an example. It has a delivery network already in Europe and now wants to establish business in further countries and regions—for example in the Middle East or South-east Asia. Having a standard capability and a related delivery application can be the blueprint for establishing new businesses in these and further regions.

Some criteria that help with making the decision for standard or variants is given in Fig. 3.7. As you will see, there is no black-and-white. Criteria from both sides may apply at the same time. The enterprise architect then needs to make a judgement together with business experts. They need to evaluate which criterion is stronger than others.

We should not forget that there are certain reasons for also allowing variances in local applications. If there are **legal** requirements, *fiscal* requirements, or especially customs regulations that require different systems, then you cannot go for a global standard system. Example international trade and logistics: It is not possible to have one global standard system because the requirements on customs processes are different in each country. You are not even allowed to use a system that has not been certified by the country's customs organisation. Consequently, you cannot introduce a single system, but need to have a system for each country. Such systems are required for submitting customs declarations electronically and support clearance processes with customs organisations.

There might be other restrictions, like **geographic** or **physical** limitations. Further on, cultural differences need to be considered. Keep in mind that *applications systems* also includes user interfaces or customer portals. A global web portal might be adequate for one country, but might not respect cultural aspects of another country. Cultural differences will always influence user interfaces of software systems but also inter-organisational business processes (e.g. processes across a supply chain of several companies).

Furthermore, different **products**, **services**, or serving different **markets** require specific application support. In a logistics company, the software used for logistics management for mail delivery is different from the one required for parcel delivery, which also is much different from a logistics system that is used for air freight and ocean freight containers. Each capability is the same, like for example *Logistics management*, but due to the differences between the various products, various systems might be needed. The same goes with markets. Imagine you want to go for e-business. E-business requires not only that you are having electronic workflows, but also that you have electronic channels for interacting with your partners.

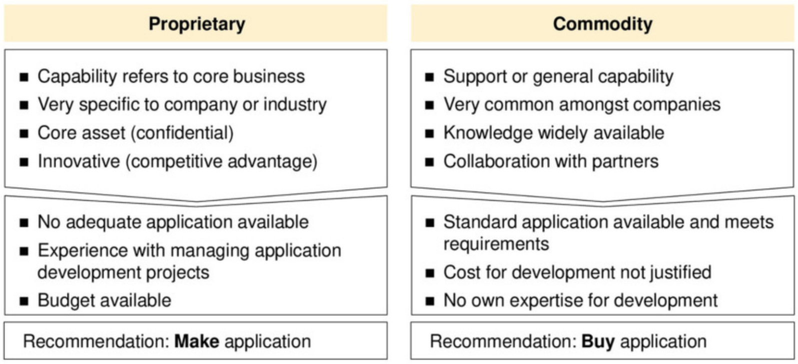
Simple capabilities (e.g. *Order handling*) already have differences between **customer segments**. For example a logistics company providing digital services for creating the shipment labels need different channels depending on private or large corporate customers. The result of such a service is a file containing the shipping label that can be attached to a parcel or container. A private customer only having a parcel once in a while definitely needs a web portal with an HTML user interface. It can be used for entering the shipment data and then get the label as a PDF or any graphic file for printing.

The use case looks much different for large corporate customers, big companies sending hundreds or thousands of parcels every day. They cannot have dedicated employees that enter data into a web portal and then print PDFs. They need a webservice that can be integrated with the customers' IT. It will support sending data for all parcels and then providing the labels as PDF electronically. The capability is the same for both customer segments: *Shipping label provision*. Due to differing levels of technical systems of the customers, the capability cannot be supported by a single label creation system.

Nevertheless, these criteria should be evaluated very carefully. Variances will result in redundant systems and, therefore, lead to higher run budgets and support effort. But an inadequate standard application can hamper the business. Of course, we are looking at this decision from a theory perspective. Having criteria makes it look simple. It will get very complicated when being in a corporate environment and talking to people because they feel very special. They always find a way for telling that they need a different system as the others because they have specific requirements. Therefore, the default should be marking each capability as standard so that it can be supported by a single application. Only if people provide evidence for a variant, then the recommendation will go for variants.

3.2.3 Make-or-Buy

The third step in the methodology, after standardisation, is considering make-or-buy. What is the best option for introducing the application? Is it better to develop an application on our own (because of the business relevance of the capability)? Or



Proprietary

- Capability refers to core business
- Very specific to company or industry
- Core asset (confidential)
- Innovative (competitive advantage)

- No adequate application available
- Experience with managing application development projects
- Budget available

Recommendation: **Make** application

Commodity

- Support or general capability
- Very common amongst companies
- Knowledge widely available
- Collaboration with partners

- Standard application available and meets requirements
- Cost for development not justified
- No own expertise for development

Recommendation: **Buy** application

Fig. 3.8 Application architecture: Sourcing

should we just buy a cheaper standard system (in order to minimize expenditures and risk)? The default is *buy*, because it is cheaper. But many business stakeholders will have a saying in this one for pushing for developing a system on our own.

Let us have a look at the criteria summarised in Fig. 3.8. The capability should be considered as proprietary, if it refers to the **core business** (i.e. it is a core capability). Core business means that it is the heart or the core of our value chain. These capabilities directly contribute to revenue generation. Other capabilities (e.g. guiding and supporting) only indirectly contribute to value creation. In a large logistics company, this might be the system for making the route planning—how to transport parcels and containers. The capability is industry specific and the system helps with optimising business performance.

Requirements that are very specific to a company (or an industry) can justify a proprietary application. Some super sorting algorithm for parcel distribution, or for planning our logistics network, can be a **core asset**. They might be unique for the company and significantly improve business performance. A company does not want others to know about it or share it with them.

Developing something very **innovative** (based on artificial intelligence) for the core business (e.g. routing optimisation for network) will provide a competitive advantage. It is not recommended to buy a standard system as it will be available to everyone. A company might also consider developing it internally and not with external development partners as the knowledge is strictly confidential.

I talked to a major airline a while ago and they are thinking of using artificial intelligence (AI) for optimising their network. It is supposed to optimise their schedule and the routing of the fleet. It provides an advantage for being better than others and can optimise business efficiency. From a pure business perspective, this is a core asset and very special to the company. Such a system is not available on the market and would provide a competitive advantage.

The method emphasises a pure business point of view by just looking at the capability and its business relevance. At the same time, one also has to consider the market of existing applications. If no such application is **available** on the market then you need to develop it on your own.

A company also needs to have skills and **experience** with managing application development. It needs to have the implementation workforce. It needs to have project managers being experienced with IT and software development projects. And of course, it needs to have the **budget**. If this is given, then the recommendation would be make the application, and don't try to buy an existing one.

On the other hand, there are valid reasons for procuring an existing software. Especially for a very common commodity. Most of the support commodities (like human resources, finance, IT, customer relationship management) are very similar in many companies, even across industries.

Implementing a new application would compete with existing ones and most probably not be that innovative. Criteria supporting such an option are listed on the right-hand side of Fig. 3.8. We first need to check if the capability common amongst companies. Customer Relationship Management is a well understood function and there are mature solutions available on the market. They are also supported by textbooks on marketing, sales and customer service. There is plenty of research in academia and practice that constantly improves the discipline. Furthermore, the same concepts are applicable for any kind of company or industry. If it is common knowledge provided within this capability, or even if it requires collaboration with partners, then rather think of deciding for an application to be bought on the market.

An application can only be procured if it is available on the market (which holds true for most supporting capabilities) and if it meets requirements. If the effort for development is not justified (because expenditures exceed the benefit by far) or the expertise in development is not available, then rather go for buying an existing application.

Honestly speaking, the world is never black and white. There will never be hard facts for making the decision for either make or buy. It will always be about checking and weighing the criteria, one against each other, prioritising the criteria, and then make that decision. Some criteria are stronger than others. If no system is available on the market, then you don't have any chance. You need to develop it, or if it is too expensive and does not justify the benefit, then there might even be the decision we are not going for an application at all. Developing an application providing a smaller benefit compared to implementation cost, does not make sense.

If you don't have software development expertise you should still consider buying an application—even for core capabilities that are very specific to your company. There are some start-up companies developing software for niche markets that tailor their system to specific customer needs. Such applications are either already configurable or the small company wants to increase their customer base. It will then also spend effort on adding additional functionality that can later be sold to further clients. Instead of buying it, having a partner that can customise an existing software with respect to your needs, can be a good choice. Even though it relates to the core business, and even though it should be a differentiator, you can let the partner develop the application.

Those criteria here should be seen as a rule of thumb that you can keep in mind. They are not hard facts. You always need to have the big picture. And then at the end, make a decision based on all relevant facts.

3.2.4 Clustering

The last step was dealing with building groups of capabilities or building clusters of capabilities. A capability map can be very large by having specialised capabilities and a very deep decomposition level.[7] It is not reasonable to have a software application for each individual capability. Consequently, capabilities need to be grouped so that we reduce the number of applications by only having one per cluster.

Clustering can follow some simple rules:

1. Only automated capabilities are relevant for clustering
2. A top-level capability (or to be more precise: all its automated sub-capabilities) can already be a cluster.
3. A top-level capability cannot mix standard/variant or proprietary/commodity. In this case it needs to be split into two or more clusters for the respective decisions.
4. Top-level capabilities can be clustered if they are similar.
5. Any capability can be a cluster if all its sub-capabilities are similar.
6. Capabilities are similar if …
   - … they have similar functions
   - … work with the same business objects

Clustering is based on similarity which covers the decisions for standard vs. variant and proprietary vs. commodity. You cannot mix them in a cluster as we would then have a conflict for the application. Similarity can also be decided by functionality (which should be the case for a decomposition of one capability) or common business objects. Also the complexity of the application landscape should be considered during clustering. One should, for example, reduce the number of interfaces between applications. Splitting common functionality or business objects across several applications will lead to interfaces for data exchange or service requests (cf. Sect. 3.3).

Applications can be derived from clusters as shown in Fig. 3.9. It does not necessarily mean that you need to have a certain product in mind. You can at least

[7]Imagine having a capability map with ten top-level capabilities that are decomposed by two more levels. If each (sub-)capability is in average decomposed into 5 sub-capabilities the we will have around 310 capabilities (10 on level one, 50 on level two and 250 on level three).

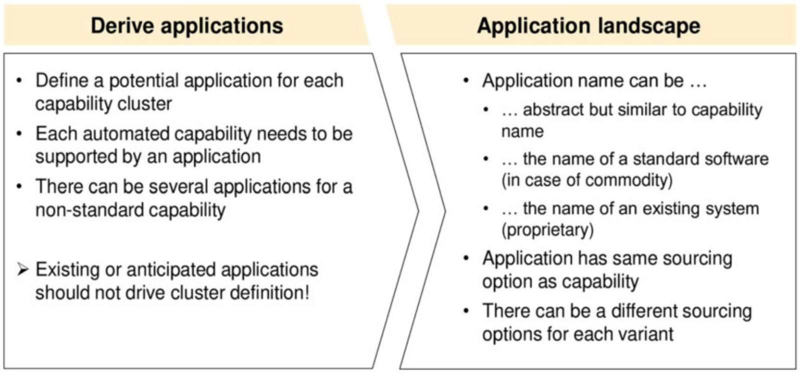

Derive applications
• Define a potential application for each capability cluster
• Each automated capability needs to be supported by an application
• There can be several applications for a non-standard capability
➢ Existing or anticipated applications should not drive cluster definition!
Application landscape
• Application name can be …
• … abstract but similar to capability name
• … the name of a standard software (in case of commodity)
• … the name of an existing system (proprietary)
• Application has same sourcing option as capability
• There can be a different sourcing options for each variant

Fig. 3.9 Application architecture: Derive applications

have an abstract name or an application type. It can even have the same name as the capabilities. For customer service, you might decide for example:

1. *salesforce.com* (concrete application)
2. *Customer Service System* (derived from capability name)
3. *Ticketing system* (class of application)

Large companies already have an application landscape in place. If those systems are supposed to be reused, then you can already put it there as a representative. For each application, you need to make sure that they have the same sourcing option as their capability.

If the capability is standard, then you can have one standard application throughout the company. If the sourcing option was *buy* based on the capability, then you can procure the application. In case of the variants, you might have different sourcing options for the same capability. Let's reconsider the example with the customs processing. Some countries have established customs systems available that you can procure on the market. If no such system is available for other countries, you can develop it on your own, even considering selling it later on. But this is a different story.

3.2.5 Resulting Application Landscape

A simple example for showing the result after applying the method is shown in Fig. 3.10. We are not seeing the decisions here, but the result of this method. Those capabilities are on a very high level, we have different systems for market development, which includes marketing and sales, we made the decision we want to have one integrated **CRM system** supporting all our marketing activities, and then also directly drive sales.

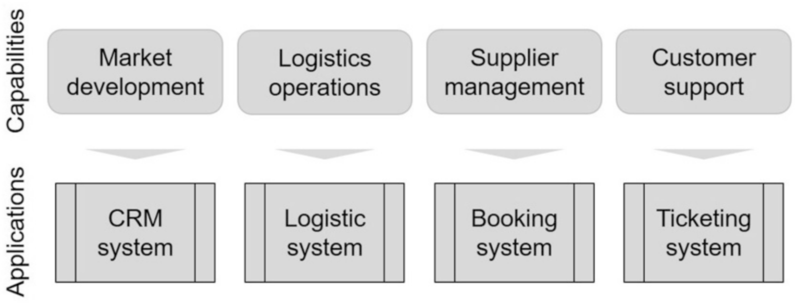


Fig. 3.10 Application architecture: Result

For logistics operations we made the decision we want to have one integrated **Logistics system**. It can be different from the CRM system and operates on different data. We want a different system for the supplier management because we are having systems that are integrated with the systems of our partners. We are not only managing our bookings, but also use the **Booking system** for sending booking data towards our partners, like airlines and ocean freight companies.

The last one is a **Ticketing system** for our customer support. Most probably, for the capability market development we decided on going for a global standard, because we also want to have one repository with all the customer data so that we can use it throughout all our marketing departments globally. Marketing and sales are not that much different across industries, so we can buy an existing CRM system. We did not make the decision on which one yet, but we made the decision we want to have a standard system.

It is different for logistics operations because here, even though systems are available, we can implement better systems that are more suited for our products and our customer segments. It is a differentiator from other competitors. This system will be developed on our own, so it will be a proprietary system and also a global standard system. We are going for an existing booking system because we have several partners and we don't want to implement integration with other partners. We want to have existing systems that already enable us sending booking data to partners. And we can also allow for variance here because there is one system for airlines and another one for ocean lines. The differentiator here are our products. We have airfreight and ocean freight products. They are very different with respect to partner integration. Hence, for the one capability, we need different booking systems. But we can buy them on the market as this functionality is publicly available.

Having those systems is the starting point for developing our application landscape. At the current stage, we used the business capabilities and derived some kind of the best application landscape we can imagine from the business capabilities. Do you think this is enough for talking about applications? Will just having the symbols on the map help us optimising the application landscape? If you say yes, it is enough, then you are too fast. Just consider being the CIO that now needs to make the decision about which products to buy or which systems to reuse and what to do with them. Which kind of information might be missing here?

3.3 Application Details

Yes, you are right. We need more information about the properties of applications as well as their relationships to each other. Applications are usually very complex entities and can be described in various ways. The following two sections will provide an overview on . . .

- . . . relationships between applications
- . . . properties relevant for managing applications

3.3.1 Relationships

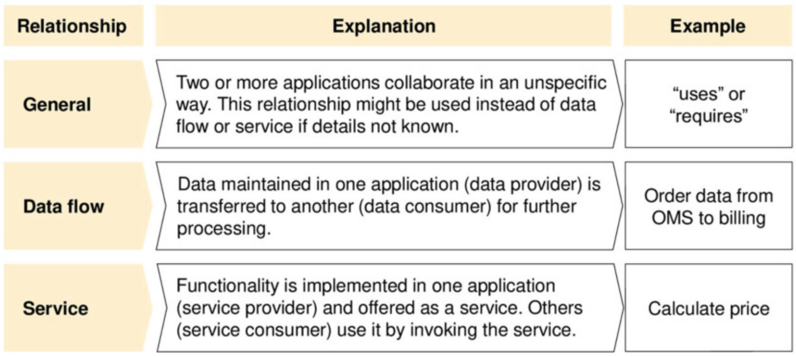
While remembering the examples for application landscapes, we know that applications are not only working on their own. Usually, a whole process is supported by several applications and those applications need to work together. An overview on typical relationships is given in Fig. 3.11. One of the previous examples specifies the exchange of business objects between applications.[8] Salesforce.com providers information on customer accounts to the transport management system and also to the ticketing system in Fig. 3.2 on page 75.

Application landscapes are not only about showing a system we own but also the collaboration between applications. In the past years, there was also some hype on service oriented architectures which is now superseded by the notion of micro services. We will not stress this buzzword here.[9] However, the notion of providing services is still a dominant mechanism in today's application landscapes. It encompasses applications *providing* services and others *using* them.

There are some examples on the very right-hand side of Fig. 3.11. The order data being maintained in the order management system will be required for billing, so it will be sent to a billing system. The same can happen with the customer accounts, as already discussed. Any business object can be subject to **data flow** between applications.

[8]We will transfer this principle to data objects as well. In the remainder of this section, *data flow* and *exchange of business objects* will be used synonymously.

[9]The service-oriented paradigm is heavily recognised in software development as it fosters reuse of application functionality. There is a popular book about Service-Oriented Architecture (SOA) by Erl [1]. There are several books available on microservice-based architecture, like for example [2].



Relationship	Explanation	Example
General	Two or more applications collaborate in an unspecific way. This relationship might be used instead of data flow or service if details not known.	"uses" or "requires"
Data flow	Data maintained in one application (data provider) is transferred to another (data consumer) for further processing.	Order data from OMS to billing
Service	Functionality is implemented in one application (service provider) and offered as a service. Others (service consumer) use it by invoking the service.	Calculate price

Fig. 3.11 Application relationships (overview)

A **service** represents some self-contained unit of functionality implemented by or within one application. This service can be provided so that other applications can use it. It works similar to a function call in any programming language. A service has a name as well as input (data provided during service invocation) and output parameters (result returned after finishing execution). For example, we can have an electronic product catalogue that helps with calculating prices for a product. The calculation can be based on the amount of ordered products, additional services or further product properties. This kind of service might be required for creating the offer to the customer and later for processing the order. Finally, it will be used by the finance system for generating the invoice. The rules for calculating the price are always the same and it makes sense having a single software system for this service. This functionality will be reused by others. A service can be implemented for exchanging data (i.e. data flow) but also implement functionality for reuse.

Especially during an early stage, while getting an overview on the application landscape, people only know *that* one system works with the other. They are aware of the relationship but do not have enough knowledge for classifying it as a service or data flow. Details are missing very often. But it is important to document the relationship so that we are aware of the connection. In this case, we got the concept of a **general** relationship between applications.

Figure 3.12 presents an application landscape showing data flows between applications. It is very similar to previous examples in this textbook. It represents a customer portal that can be used by our customers for entering order details. The customer order is then transferred to the order management system (data flow **customer order** between the two applications). Order data is then also sent to the billing system. It is a simple description, but still very powerful. It is used in many organisations for showing how applications are working with each other. Integrating e-commerce processes and companies along a supply chain is based on well defined data flows.

This is one of the most important map when describing the application landscape. Yes, this example is simple. But let us always keep in mind, maps of real world

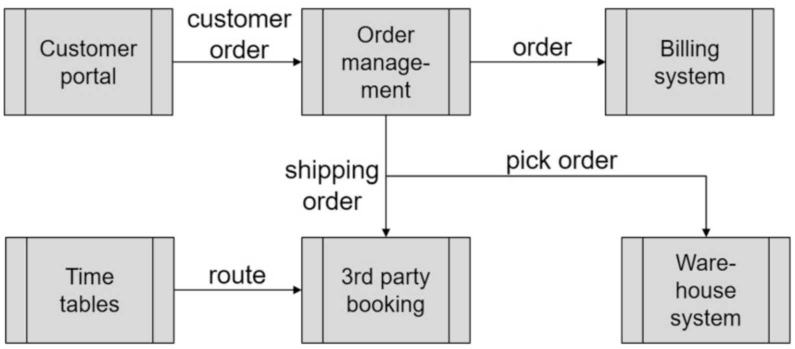


Fig. 3.12 Example: Data flow

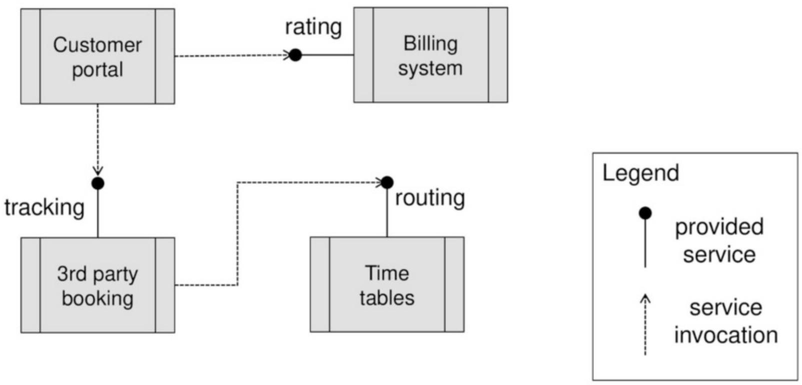


Fig. 3.13 Service-oriented relationships

companies are much larger and complex. They are not only having six systems, but a couple of hundred, or even more than a thousand software applications. Understanding how they relate to each other can only be done with a good overview map. The map can also be considered as *best practise* visualisation for showing how applications work with each other.

The notion of service provisioning might sound a little bit abstract. We want to explain it with a quick example in Fig. 3.13. The application landscape has a **Billing system** which provides the **rating** service for other applications. This service is supposed to calculate the rate (i.e. the price) of a product. Other applications, like the customer portal, can use the rating functionality by just invoking the service.

Imagine the customer entering the new order in the customer portal. This portal shows all relevant information for the product and also automatically calculates the price by just using the rating service of the billing system. The functionality for price calculations is only implemented in the billing system and not in the customer portal. The rating service can also be used by other systems, like the order management system or the customer relationship management system. The benefit of having those services is economies of scale: You implement them once and then they can be reused by other applications.

There are two more services

1. **tracking** service in the **3rd party booking** system
2. **routing** service in the **Time tables** system

Monitoring logistics flows requires an overview on the status of each shipment. Status information is generated whenever a certain activity[10] has been performed on a shipment or it has been handed over between logistics partners. A logistics company provides those *track events* by itself but also receives them from their logistics partners.

Parcel logistics companies need to receive tracking information, for example, from the airline. Did the plane with the parcel (or the container with all my parcels) already arrive at the destination? Did the shipment pass customs? Are there any delays? Instead of implementing tracking functionality in each system, we can implement it once in a single system (in this case the third-party booking system). It can then be reused by the customer portal so that the customer can track the status of the booking. The service can also be used by logistics management systems or order management systems (not shown in Fig. 3.13).

The **routing** service is implemented in the **Time tables** application. This service calculates the optimal routing for a new shipment based on available capacity and cost. Routing is one of the core capabilities of a logistics company. It aims at minimising the cost for shipment by evaluating different options for moving a shipment from A to B. Those options can compare direct flights with multi-stops or rates from different airlines. The service is only used by the 3rd party booking system in this example but can also be offered to further applications. It can support network optimisation, order management and order execution.

3.3.2 Properties

Let us now get a little bit more specific about what kind of information is needed for assessing the application landscape. It will be required for the following two scenarios:

- *as-is analysis* is about understanding the current application landscape. We need data that will help with identifying and documenting current issues. This will be the starting point for optimising the application landscape.[11]

[10]Example activities are: export processing finalised, customs clearance done, shipment sent to destination country, shipment on hold.

[11]Transforming the application landscape will be further discussed in Sect. 5.1.

- *to-be planning* will provide a map describing the future state of the application landscape. It is supposed to be *better* than the as-is, hence, we need criteria for comparing as-is with to-be.

There are two kinds of information that are relevant for the analysis of application landscape as provided in Fig. 3.14. First of all, we need to know the cost associated with existing (or planned future) application systems. This will cover on-time and periodic fixed cost for implementation and provisioning. Maintenance generates periodic variable costs for bug fixing and system updates based on new requirements. Costs are the key concern for most IT organisations as discussed in Sect. 1.1 (visualised by Fig. 1.6). Especially when facing an application landscape that has been growing in an uncontrolled way, people need to consider how to reduce the number of applications (and complexity). After identifying redundant software applications, the IT department needs to make a decision on how to consolidate them. Is it a good idea to keep the cheapest one and decommission the other ones?

Most probably not as you need to consider application quality during such a decision. This can be addressed by the following questions:

- What is the technical implementation platform of an application? Out-dated implementation environments can result in increased maintenance cost in the future. It is recommended to switch off applications on legacy infrastructure (e.g. mainframe computer) instead of modern server systems. This is referred to as **technical fitness**.
- How well does the application fit into your overall architecture? The implementation of applications in an organisation should follow the same principles (**architectural fit**). The structure of the application landscape should also fit to business processes.
- Can it easily exchange data with other applications or do we need a dedicated data translation system? This is a common symptom for an architectural mismatch. Value-add along end-to-end business processes requires a seamless flow of data. Translations can hamper this flow by information loss or inconsistencies during data exchange.

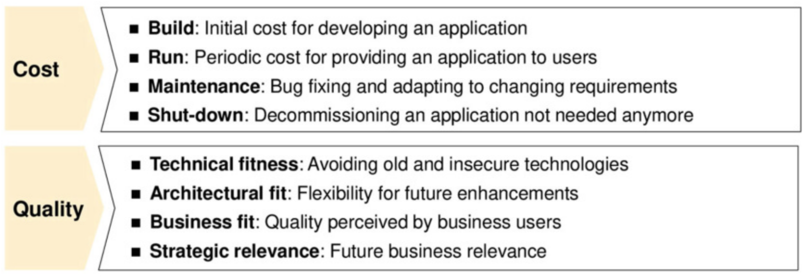

Cost
▪ **Build**: Initial cost for developing an application
▪ **Run**: Periodic cost for providing an application to users
▪ **Maintenance**: Bug fixing and adapting to changing requirements
▪ **Shut-down**: Decommissioning an application not needed anymore
Quality
▪ **Technical fitness**: Avoiding old and insecure technologies
▪ **Architectural fit**: Flexibility for future enhancements
▪ **Business fit**: Quality perceived by business users
▪ **Strategic relevance**: Future business relevance

Fig. 3.14 Application properties

- How well is an application supporting the business? All business capabilities should be fully supported by software applications. Applications should help with performing business processes in an optimal way (**business fit**).
- How well is the application contributing to the corporate strategy? The prioritisation of software applications also needs to consider their **strategic relevance**. However, the strategic relevance of an application is usually determined by the capabilities it supports.[12] The strategic relevance does not only consider immediate effects but also investments into future business opportunities.

The properties from Fig. 3.14 can be taken into consideration when assessing existing applications and a complete application architecture. They help with making decisions based on hard facts like reducing cost or increasing quality. Beside the complexity reduction mentioned above, they can also substantiate a decision on an investment—including the replacement of an existing application.

Example 3.1 (CRM Replacement) We have a CRM system, but users are complaining about it. It is buggy and makes the work of customer service very cumbersome. It is also quite old and does not serve all the business requirements. The sales and marketing heads ask for a replacement but you have to provide the information for a making conscious decision.

Your first investigation starts with determining cost:

- What are the costs associated with the existing CRM system? (provisioning and maintenance)
- How much do we need to pay for implementing the new system? (either make or buy)
- What will be the run costs of the new system compared to the old one?
- Will we have less maintenance cost with the new one, compared to the old system?
- How much do we need to pay for shutting down the old system?

These are only a few example questions for a pure cost-related comparison between *keeping the existing system* or *introducing a new CRM application*. However, this is not only a cost-centric decision. You also need to consider quality attributes:

- Technical fitness of the new system should be better than the old one in order to solve one of the biggest pain points.
- The new CRM system needs to fit to the corporate application landscape.

(continued)

[12] Applications—as any resource—do not have a business relevance. Their relevance is decided by the business they are supporting as only value-add provides result in a corporate environment.

Example 3.1 (continued)

- Business support should be improved due to less frictions caused by insufficient IT.

The quality attributes can be applied for assessing one CRM system and also the comparison between several options. For example: Is salesforce.com better suited for us because it is a cloud solution? Would Oracle (or another customer relationship management system) be a better option as it will be hosted by us? And we also need to have technical support for the new application. At the same time, does salesforce.com really support our business processes better than Oracle?

All these information will influence your decision on changes on your application landscape. Therefore, it information needs to be collected and maintained for each and every application. Maintaining a repository with application data will be further discussed in Sect. 5.2 when describing the role of the enterprise architect.

Figure 3.15 provides some more detail on quality attributes of applications. Technical fitness is a view on the architecture and implementation of an individual software system:

- Is it an old system based on mainframes, or is it implemented using modern technologies?
- Or did we use modern web frameworks or user interface frameworks?
- Does it follow a holistic architecture?

The better the maintainability of a system, the higher the **technical fitness** will be. If the application only has a few bug reports and only requires a very few technical changes, then the technical fitness might be higher. This is something you can even measure for existing applications. You can check bug reports or incident

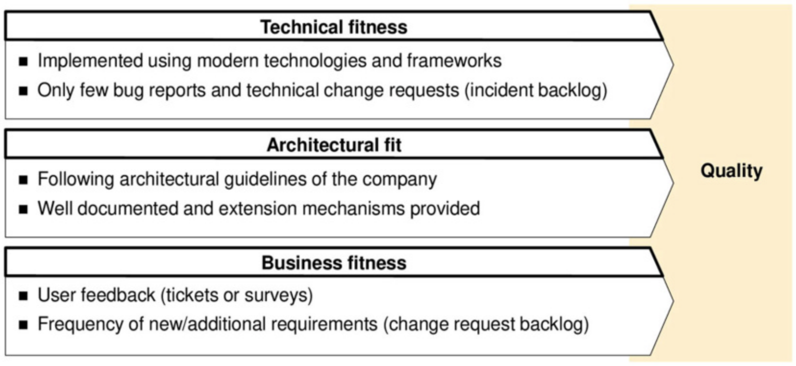

Technical fitness
Implemented using modern technologies and frameworks
Only few bug reports and technical change requests (incident backlog)
Architectural fit
Following architectural guidelines of the company
Well documented and extension mechanisms provided
Business fitness
User feedback (tickets or surveys)
Frequency of new/additional requirements (change request backlog)
Quality

Fig. 3.15 Quality aspects

logs for applications. The number of incidents negatively affects the technical fitness (more incidents usually implies less technical fitness).

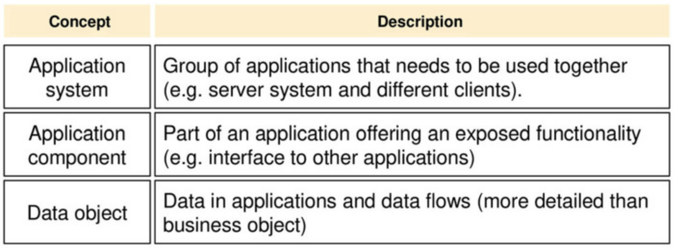
The **architectural fit** does not only look at the system (i.e. its technical implementation) but its integration with other applications. Does the application follow architectural guidelines that have been provided by the companies, or does it follow the rules provided by EA? If yes, it fits our architectural vision. At the same time, we also need to make sure that the software and its interfaces are well documented. Documentation will be required later on for any kind of maintenance activity (extension of integration). The better the documentation, and also the architectural quality, the better the architectural fit can be assessed.

Business fitness measures how well the application supports the business. Sometimes this is not easy to assess because different people can assess the quality differently. Some people might prefer a Windows-based user interface while others still prefer the old mainframe style with keyboard short-cuts. These users can be more efficient with entering data in the old mainframe-style than with mixed mouse and keyboard interactions. However, such an assessment can be subjective. There are some criteria that can help with measuring the business fitness of a system.

- *Survey*: First of all, we might conduct a survey concerning business fitness. We ask people how well the application supports their activities. A survey does not only address individuals but can be performed with many people.
- *Number of tickets*: The more business users are complaining about a software system, the worse the technical fitness might be. Furthermore, if the business fit is high, then we should expect that people are not complaining or opening less tickets with respect to a certain application.
- *Change frequency*: The more changes are required by business stakeholders for a specific application, the less it supports current requirements. Otherwise, they would not request the requirements.

In summary, a huge backlog of change requests is an indication (or a measure) for assessing the business fit. If the application is properly supporting the business, users would not open many tickets, and they would not have many change requests. The other way around, if you have a lot of tickets and change requests, then you should investigate further. The number of tickets and change requests can be a measure for poor business fit. However, this assumption can be wrong. Many change request might also be the result of the popularity of an application. Users might like it and, therefore, extend it with further functionality.

That's it so far about the core concept and the core information we need about application landscapes or concepts that we need to describe application landscapes. Most important, we have the application itself. We have relationships between applications, which might be data flows or services being used or just general relationships without further specification. We also need data about applications. Collecting data is quite tedious but required for doing analysis to make decisions on optimising application landscapes.



Concept	Description
Application system	Group of applications that needs to be used together (e.g. server system and different clients).
Application component	Part of an application offering an exposed functionality (e.g. interface to other applications)
Data object	Data in applications and data flows (more detailed than business object)

Fig. 3.16 Additional application concepts

3.3.3 Further Application Concepts

There is not much more to add to application landscape, except a few additional concepts, shown in Fig. 3.16. We will briefly present them here so that you are aware of them. Some EAM tools may require them. They will not be required in the remainder of this textbook.

Some tools provide concepts for **application systems**, which is a group of applications that relate to each other. SAP is, for example, such an application system, consisting of several applications. It encompasses an Enterprise Resource Planning (ERP) module, Transportation Management (TM), Customer Relationship Management (CRM) and some more. They build a whole software system working together.

There is another abstraction called the **application component**. An application can be further devided into several modules or components. This provides a more fine-grained description of functionalities implemented by such an application. A logistics management application can, for example, consist of a module for planning the bookings, for transmitting bookings to partners, and also for monitoring the transports. The application consists of three components. Even having interfaces to other applications might justify a dedicated component.

We will now have a look at the topic of data objects. When talking about the application architecture, people also relate it very often to data objects. And in fact, our examples for the application landscape already indicated that we have data flows between applications. And for describing those, we need the notion of a **data object**.

3.4 Data Architecture

I would like to remember the levelling scheme of EA as introduced in Sect. 1.4. Figure 1.20 shows the three-layered structure for EA, consisting of

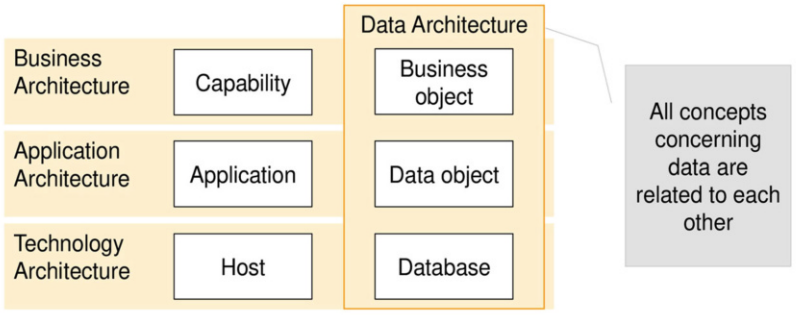


Fig. 3.17 Data architecture as cross-cutting view

- *Business architecture*: introducing business processes, capabilities, business objects and further business-related concepts.
- *Application architecture*: consisting of applications and their relationships.
- *Technical architecture*: will not be covered in this textbook.

We did not talk that much about data but only business objects. It was already stated that business objects can be a starting point for deriving data models for applications and software development. Some frameworks use the notion of the *data architecture*. We will not change the three-layered approach here by adding another layer. We will rather explain, how the notion of data fits into the current model.

Whenever we see, read or hear the term data architecture, we should keep in mind it refers to data on various level of detail in our enterprise as shown in Fig. 3.17. In this sense, the business object relates to the data architecture. It is the business view on the data represented by business objects. On application architecture, we also have a view on data, which is represented by data objects. Data objects represent data that is stored in applications and exchanged between applications. And going further downwards on the levels, we have the database as a representation of business objects in the technology layer.

We will not follow the notion of data architecture that much in the course of this textbook. We just introduced it here for showing the term also exists. Sometimes people interpret it differently. In our case, if we think of data and data architecture, it is a separate dimension over the layers. We already had concepts for representing data on the level of abstraction for each level. Namely, business object as the business architecture representation of data. Data objects are a more detailed view in the application architecture. The next level is considering the implementation of the technology. This comprises installing concrete application system that use databases making data persistent.

These three concepts represent the same kind of information, it is all about data. The difference is in the level of abstraction (Fig. 3.18). While the business objects are on a high level (business level), we only think of the most important terms that are relevant for the business. They are reflected by a name and can have a description in natural language (like a glossary). They can also already contain properties

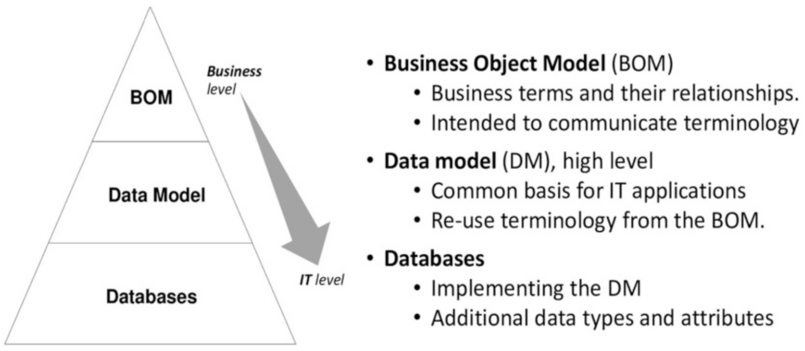

BOM
Data Model
Databases
Business level
IT level
• Business Object Model (BOM)
• Business terms and their relationships.
• Intended to communicate terminology
• Data model (DM), high level
• Common basis for IT applications
• Re-use terminology from the BOM.
• Databases
• Implementing the DM
• Additional data types and attributes

Fig. 3.18 Levels of abstraction of data

defining concrete information contained in that object. A business object model (BOM) shows relationships between them, and is usually driven by the terminology used by the business.

The business object model can be the starting point for an application-specific data model. A data model is a very common tool during software development, helping the software engineers, designing and implementing the software system. Ideally, the data model is not something completely new, but should refer to the business object model by reusing terminology from the business object model. It will define further data objects and provide more details on attributes and relationships between data objects.

If business people are talking about bookings, then there's no need for having an abstract name in the data model, something like XYZ5. Even if it is nicer for the programmers, it is not really helpful if business stakeholders do not recognise business concepts. The data model should link back to the business object model by reusing the terminology introduced by the business objects. There will be more data types as well. Defining the data model for the application, we need to introduce new data that is required for implementing the system. Implementation details should not be relevant on a business level.

A concrete database (technology architecture) being implemented in our corporate environment will contain specific information about primary keys, foreign keys, tables for implementing end to end relationships—very specific details that are required for database design. All those details are not required on the business level. Going from the business object model via the data model towards the database will result in more technical details on data objects required for system implementation

3.5 Further Reading

For further reading, there are three titles that I would like to recommend. There are two books available that deal with a language called ArchiMate. The first one by Lankhorst and co-authors, is now available in the fourth edition (the first edition being published already more than 10 years ago). It is very motivating and describes the basics of the ArchiMate modelling language, which is quite popular for describing EA. It offers a lot of concepts for modelling applications and application landscapes. Whoever needs to have an overview and then more details, [3] would be a recommended reading.

There's also some kind of textbook for teaching how to use ArchiMate, also published by a Dutch guy called Gerben Wierda. He has a very detailed book about how to use ArchiMate, how to describe certain architectures using the ArchiMate language and tools [4]. This book provides many examples on how to describe an application architecture and its relationships to the business architecture.

The third one is a blog entry by a German consultant called Stefan Tilkov, who's very popular in the software engineering community, and also in the architecture community. He provided some thoughts on something called a *canonical data model*. Starting from the business object model, we can think of having a standard data model for the whole enterprise. He provides a very critical discussion and comes to the conclusion that we should not aim at having a detailed enterprise-wide data model. If you're interested in this one, there's a discussion on this in the blog provided by Stefan Tilkov [5].

3.6 Summary

That's it for this chapter. It introduced application architecture concepts for describing the application landscape. It basically covered software applications together with their relationships and business context. We also introduced a method for deriving an ideal application landscape from the business capability map. More concepts have been introduced for describing information about software applications and their interfaces (data flow and services). They are required for describing and analysing the as-is application architecture. Analysis will be subject of the next chapter.

We should have an overview on what we covered so far in this book (Fig. 3.19). We started with the introduction into EA and EAM, providing the motivation why we are doing it in Chap. 1. This was followed by two sections describing special architectures. Chapter 2 covers concepts for describing the business architecture with the two core concepts, business capability and business object. Chapter 3 added the perspective on application architecture with the core concepts application and data object. We learned how to derive an ideal application landscape from our business capabilities. Additional information was introduced in order to further describe software applications. The subsequent Chap. 4 called *analysing EA* will address the following topics:

- How can we describe and analyse EA?
- What is the business support matrix and how can it help with improving our EA?
- Which typical problems can be found in an application landscape?

Introduction to Enterprise Architecture
Understanding Business Architecture
Developing Application Architecture
Analysing Enterprise Architecture
Managing Enterprise Architecture
Applying Frameworks

Fig. 3.19 Following next: Analysing enterprise architecture

3.7 Exercises

Exercise 3.1 (Application Interfaces) Please, describe in your own words the difference between *data flow* and *service* relationships between applications.

Exercise 3.2 (Commodity Applications) You are an enterprise architect in a pharmaceutical company. The marketing and sales head wants to replace the legacy Customer Relationship Management (CRM) system by a new one. He wants the IT department to develop a proprietary system so that industry-specific knowledge can be incorporated as well as modern data analytics functionality.

What would you recommend? Should they develop a new system or procure an established one from the market? Please, explain your answer thoroughly!

Exercise 3.3 (Standard Applications) You are an enterprise architect in a international banking corporation having subsidiaries in Australia, New Zealand, United Kingdom, Malaysia and Hong Kong. Your Chief Operations Officer aims at automating compliance and security check by introducing intelligent data analytics systems. As those compliance rules look very complicated, he wants to introduce an individual system for each country organisation.

What would you recommend? Should they introduce a global system or individual ones for each country? Please, explain your answer thoroughly!

Exercise 3.4 (Target Application Landscape) You are the Chief Information Officer (CIO) of the Deakin University and are tasked to create a target application landscape for the university.

You should address the task in the following way:

1. Use the capability map as a starting point
2. Identify applications for supporting these capabilities
3. Try to eliminate redundant applications