3
Understanding the Domain

Moving from other roles, such as a software developer role, to being a software architect requires one to expand and broaden their sphere of knowledge to include a deep understanding of their domain. Effectively designing a solution for a problem space requires knowledge about the domain and the requirements for the software.

This chapter begins by describing the foundation of understanding a domain, which includes general business knowledge and a keen understanding of your organization's business. We will go over the important concepts of **domain-driven design** (**DDD**), and how it can help your software project team handle complexity and model a solution around real-world concepts. The chapter also details the different types of software requirements and the techniques to elicit them from key stakeholders.

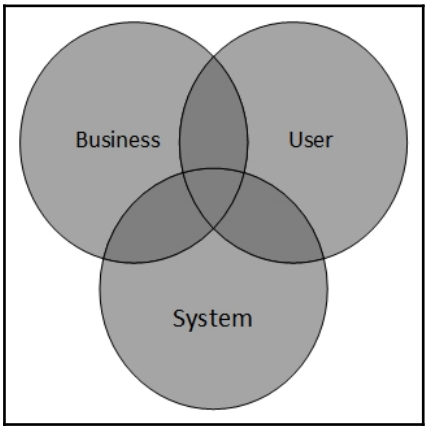
In this chapter, we will cover the following topics:

- Developing business acumen
- Domain-driven design
- Requirements engineering
- Requirements elicitation

Developing business acumen

While being a software architect requires in-depth technical knowledge, to be successful in the role also requires a thorough understanding of your organization's business. In order to design an appropriate architecture, you need to have knowledge of the business problems you are trying to solve and the business opportunities your organization is seeking to exploit. A technically advanced software system is of no use if it does not meet its goals.

When designing a software architecture, in order to ensure that the solution is an appropriate one, you must consider the goals of the **business**, the **users**, and the **software system**:



Each of these focus areas has its own goals, which can significantly overlap and impact each other. For example, a business goal of a specific, aggressive time to market could mean that there is not enough time for things such as proper requirements analysis or quality assurance testing, which could significantly impact user goals. When goals are in conflict with each other, it requires you to find an acceptable balance among them when designing your software architecture.

Familiarity with general business topics

Having a competent and practical understanding of business and its terminology will be useful to you in becoming a complete architect. While your passion may be for technology, as a software architect you will benefit from business knowledge more than other roles, such as a software developer. Software architects need to interact with a variety of stakeholders, and understanding the language of business will ensure that you have a common understanding with them.

Ultimately, you are designing a software architecture that will fulfill business goals, and your understanding of business will guide you in that task. Having general business knowledge of topics such as finance, operations, management, and marketing will help you to understand the value your software is supposed to bring to an organization. Business decisions will be made based on things like **return on investment** (**ROI**) calculations for the software project and cost-effectiveness analysis of different approaches. A good grasp of these concepts will help you to add value to such discussions.

One way to gain this knowledge is through formal education. If that is not feasible, there are other ways to get at least a rudimentary understanding of business topics. There are online classes available, some of which you can attend for free. Another option is to obtain and read one or more books on the relevant topics.

Understanding your organization's business

Once you have some general business knowledge, you will want to gain a good understanding of your organization's business. It is a crucial aspect of being a successful software architect and separates a good architect, who only has technical knowledge, from a great one.

A good starting point is to gain an understanding of your organization's products and services, and the value they provide to their customers. How does your organization make money? If you are the software architect for a specific product, pay particular attention to that product. Invest the time to understand the various business processes of your organization.

You should learn about the market that your organization operates in and its trends. It is prudent to become familiar with your organization's competitors. You should seek out answers to questions such as:

- What do your competitors do differently?
- What do they do that is similar?
- What are the strengths and weaknesses of your competitors?

Most importantly, spend the time to understand your organization's customers. The software products you help design are for your customers, and they are perhaps the most important aspect of your organization's business. What does their business do? How do they use your products and services? Why did they choose your products and services over those of a competitor?

Once you become familiar with your organization's business, the market in which it operates, its products/services, and its customers, you are on the path to fully understanding your organization's domain.

Domain-driven design

Understanding the domain of your software application is part of what is necessary to discover the appropriate architecture for any solutions you need to develop. The domain is the subject and body of knowledge on which the software will be applied.

The term **domain-driven design** (**DDD**) was coined by Eric Evans in his book, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. DDD is an approach to developing software that aims to make the software better by focusing on the domain. It has a set of concepts and patterns that have successfully been used to solve problems and build great software.

DDD is particularly useful for large software applications that have complex and sizable models. DDD helps you to solve complex problems. Even if you decide not to follow DDD fully while architecting your applications, some DDD concepts may be helpful to you. It is beneficial to become familiar with these concepts as some of them are referenced in other areas of this book, and in the event that you encounter them in your work.

Encourages and improves communication

One of the benefits of DDD is the fact that it encourages and improves communication. Communication among all team members is encouraged. In particular, DDD stresses the importance of interacting with domain experts.

A **domain expert**, or **subject matter expert** (**SME**), is someone who possesses expertise about, and is an authority in, a particular area. Understanding the domain of your software application is highly beneficial and domain experts will help the entire team gain this understanding.

In addition to encouraging communication with domain experts, DDD improves communication among all team members and stakeholders by introducing the concept of a ubiquitous language.

What is a ubiquitous language?

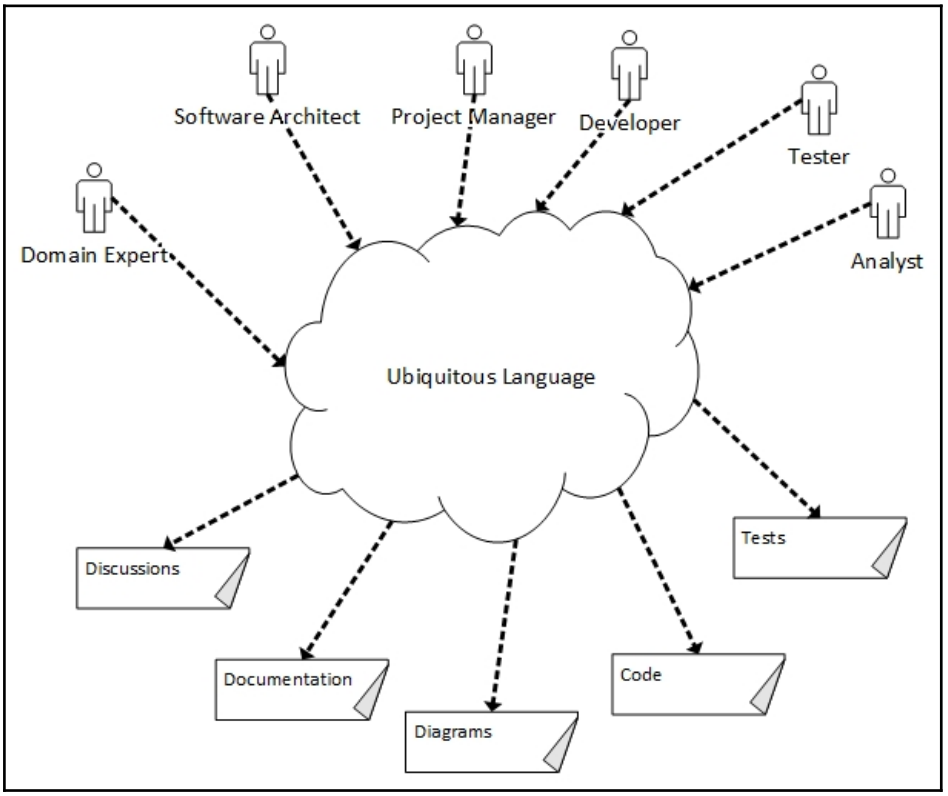
The development team may not have a strong understanding of the domain, and may not be familiar with terms and concepts used by stakeholders, including the domain experts. They may use their own language when discussing the functionality and discuss the domain in terms of their technical design. The stakeholders, including the domain experts, will use their own jargon when discussing their domain, and may not have a good understanding of technical terms. Because different people may use different language to describe the same concepts in a particular domain, it can take longer to communicate ideas, and it can lead to misunderstandings.

In *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Eric Evans described this problem:

> *"The terminology of day-to-day discussions is disconnected from the terminology embedded in the code (ultimately the most important product of a software project). And even the same person uses different language in speech and in writing, so that the most incisive expressions of the domain often emerge in a transient form that is never captured in the code or even in the writing.*
>
> *Translation blunts communication and makes knowledge crunching anemic.*
>
> *Yet none of these dialects can be a common language because none serves all needs."*

Some team members may become familiar with the domain terminology and act as *translators* for the rest of the team, but they can become bottlenecks.

In order to mitigate these types of risks, Eric Evans created the concept of a **ubiquitous language**. It is a common language among all team members and stakeholders based on the domain model:



Developing a ubiquitous language can take time, and can evolve and grow as the team's understanding of the domain changes. Domain experts should use their understanding of the domain to point out terms that don't correctly express an idea, and everyone can look for inconsistencies and ambiguities in an effort to improve the ubiquitous language.

Although it takes effort, once you have a ubiquitous language, it simplifies communication and leads to a greater understanding among everyone involved with the project. No translation will be needed because everyone has agreed on and understands the various terms. The important thing is to use it consistently and throughout the project. The ubiquitous language should be used during **discussions** and in all of the project artifacts such as **documentation**, **diagrams**, **code**, and **tests**.

Entities, value objects, and aggregates

Some of the basic building blocks of DDD are entities, value objects, and aggregates. When modeling them, the ubiquitous language should be used.

Entities

Entities are objects that are defined by their identity and not their attributes. They are mutable because the values of their attributes can change without changing their identity. If two objects have the same values for their attributes, other than their unique identifier, they are not considered equal.

For example, if you had two `Person` objects with the same first and last name values for those corresponding attributes, they are still two different objects because they have different identities. This also means that a value for an attribute such as last name can be changed on a `Person` object and it still represents the same person.

Value objects

Unlike entities, value objects are objects that describe some characteristic or attribute, but have no concept of identity. They are defined by the values of their attributes and are immutable. If two objects have the same values assigned to their properties, they can be considered equal.

For example, if two objects that represent points on a graph using Cartesian coordinates have the same x and y values, they can be considered equal and would be modeled as a value object.

Aggregates and root entities

Aggregates are groupings of entities and value objects that are treated as a single unit. A boundary is defined that groups them together. Without aggregates, complicated domain models can become unwieldy to manage, as the many entities and their dependencies grow large in number. Retrieving and saving an entity and all of its dependent objects can become difficult and error-prone.

An example of an aggregate is an order object that contains an address object and a collection of line item objects. The address object and the collection of line item objects are all separate objects, but they are treated as a single unit for data retrieval and changes.

Separating the domain into subdomains

One practice of DDD is to separate the domain model into multiple subdomains. While a domain is the entire problem space that the software solution is being developed for, a subdomain is a partitioned piece of the overall domain. This is particularly useful for large domains, where it is not feasible to have one large and unwieldy domain model.

By focusing on one subdomain at a time, it reduces complexity and makes the overall work more digestible. Rather than attempting to address too many issues at once, dividing your domain into subdomains provides more of a *divide and conquer* approach.

For example, in a student information system, you may have subdomains for contact management, admissions, financial aid, student accounts, and academics, among others.

One or more of the subdomains may be designated as a **core domain**, which is typically the part of the domain that is fundamental to the organization. If there is a part of the domain that differentiates the organization from competitors, it is probably one of the core domains. Core domains are the reason that the software is worth writing, rather than buying existing software off the shelf or outsourcing the work.

The domain experts on the project can help with identifying the core domains, as well as the division of domains into subdomains.

What are bounded contexts?

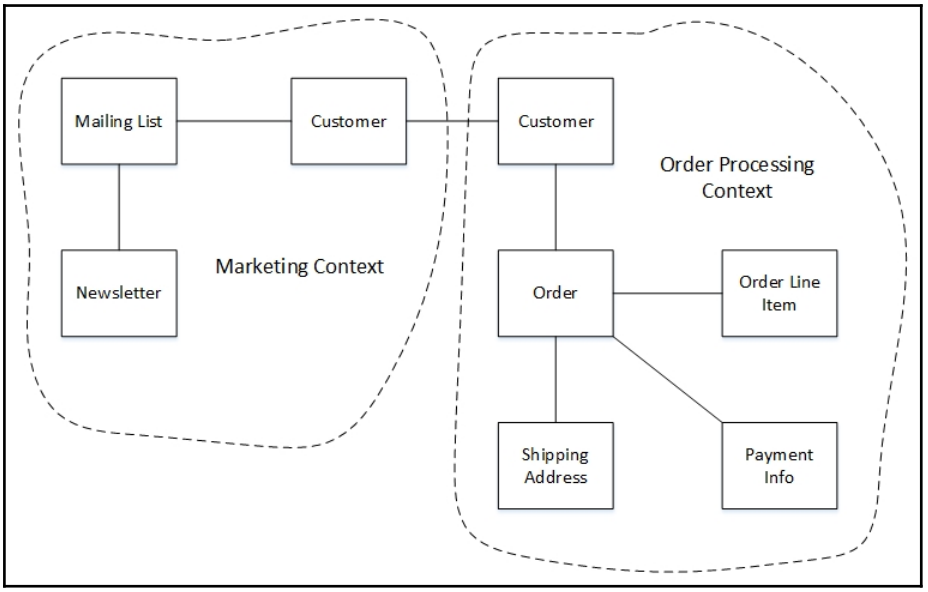
A domain model is a conceptual model based on the domain and includes both behaviors and data. It represents a part of the overall solution that fulfills the goals of the business. **Bounded contexts** are a pattern in DDD that represent partitions in the domain model. Similar to subdomains, which are partitions in the *domain*, bounded contexts are partitions in the *domain model*. As is the case with subdomains, creating partitions and boundaries reduces the overall complexity.

A bounded context may map to a single subdomain, but keep in mind that is not always the case. The domain model for a subdomain may require multiple bounded contexts for the overall solution of that subdomain.

For example, if we were creating a software system for a business that sells clothing online, we might allow customers to sign up for a newsletter that contains deals and discounts. Another part of the application would allow customers to place orders and provide payment information.

With these two pieces of functionality, some concepts are shared, while some are not. If different development teams, or different developers on a single team, are working on these two sets of functionality, it is not clear what overlap, if any, exists. If there is overlap, what should or should not be shared between these two pieces of functionality? This is where the concept of bounded contexts is applicable. A domain model applies to a particular context, so we can define the various contexts to clear up some of the ambiguities that exist.

In this example, we could create one bounded context for marketing (**Marketing Context**), and one for order processing (**Order Processing Context**). Each bounded context may have entities that are unique to itself. For example, the **Order Processing Context** has the concept of an order line item, whereas the **Contact Management Context** does not. However, both bounded contexts have the concept of a **Customer**. Is **Customer** referring to the same concept in both bounded contexts? By separating them out, we can begin to answer this question:



In the context of marketing, all that may be required for a **Customer** entity is an identity (unique identifier), first name, last name, and email address. However, in the context of placing an order, the **Customer** entity would require additional information, such as a shipping address and payment information.

You could create one **Customer** entity, but using it for different contexts adds complexity and can lead to inconsistencies. Validation that requires payment information only applies in the **Order Processing Context**, and not the **Marketing Context**. The behavior required for a **Customer** in the **Order Processing Context** should not prevent a **Customer** from being created in the **Marketing Context**, where only the first name, last name, and email address are required.

We will discuss the **single responsibility principle** (**SRP**) later in this book, but the basic idea is that each class should be responsible for a single aspect of the functionality. The **Customer** entity is still small now, but you can begin to see how it could grow quickly. If it were to be used in multiple contexts, it would attempt to fulfill too many disparate responsibilities and break the SRP.

The context for each model should be clearly defined and there should be an explicit boundary between bounded contexts. They are created so that everyone on the team, or across multiple teams, can have the same understanding of what belongs in each context. While the example used is a simplistic one, a large domain model will have many entities and contexts, and it usually isn't immediately clear what is unique or common across different contexts, and how each context should interact with each other.

DDD and the concept of bounded contexts work well with microservices, which we will be discussing in further detail later in this book. Now that we have a better understanding of DDD concepts, let's go into detail about requirements. Working with domain experts and other stakeholders, we need to have an understanding of the requirements prior to design.

Requirements engineering

In order to model your domain and design an appropriate architecture, you will need to know the requirements for the software you are building. **Requirements engineering** involves establishing the functionality that is required by the stakeholders, along with the constraints under which the software must be developed and operate. It encompasses all of the tasks involved in eliciting, analyzing, documenting, validating, and maintaining the requirements of the software system. As a software architect, you will be participating in these tasks so it is helpful to be familiar with them.

Types of software requirements

There are different types of software requirements, and software architects should be knowledgeable about them. The main types of software requirements include:

- Business requirements
- Functional requirements
- Non-functional requirements
- Constraints

Let's take a closer look at each of these types.

Business requirements

Business requirements represent the high-level business goals of the organization building the software. This type of requirement defines the business problems that the software will solve or the business opportunities that will be addressed by the software.

Business requirements may include requirements that come from the market. An organization may want to ensure that they are not excluding some functionality that a competitor is providing. They may also want to differentiate themselves from a competitor by providing functionality that a competitor is not, or provide the same functionality but in some improved form (for example, faster response times). As a result, business requirements often influence the quality attributes of a software system.

Functional requirements

Functional requirements describe the functionality of the software., In other words, what the software system must do. They detail the capabilities of the software system in terms of behavior. The functionality and capabilities described by the functional requirements enable stakeholders to perform their tasks.

Functional requirements include the interaction of the software with its environment. They consist of the inputs, outputs, services, and external interfaces that should be included with the software.

Keep in mind that requirements can come from a variety of sources, such as the following:

- **Organizational requirements**: Requirements that are based on organizational policies and procedures
- **Legislative requirements**: Non-functional requirements that detail any laws and regulations that the software must follow
- **Ethical requirements**: Any requirements for the ethical operation of the software, such as concerns related to privacy or safety
- **Delivery requirements**: Requirements that are related to the delivery and deployment of the software
- **Standards requirements**: Requirements for any standards that must be followed for the development of the software or how the software must operate

- **External requirements**: Requirements that originate externally, such as requirements from external systems that must integrate with the software system being designed

Non-functional requirements

Non-functional requirements are conditions that must be met in order for the solution to be effective, or constraints that must be taken into consideration. Business analysts and other stakeholders tend to do a good job at capturing functional requirements, but they don't always focus as much on non-functional requirements. However, non-functional requirements are an important part of requirements engineering. The success of the project is dependent on the non-functional requirements and whether or not they are met.

When a software architecture is designed, the software architect must ensure that the non-functional requirements can be satisfied. Non-functional requirements can have a significant impact on the design of the architecture. For that reason, they are of great importance to software architects. Software architects need to play an active role in eliciting non-functional requirements from stakeholders and ensuring that they are captured.

Quality attributes are a subset of non-functional requirements and include the *ilities*, such as maintainability, usability, testability, and interoperability. We'll go into more detail on some of the different quality attributes in the next chapter.

Constraints

Constraints are some type of restriction on the solution and may be technical or non-technical in nature. Some constraints on a project might be captured and classified as a functional or non-functional requirement, or they might be explicitly categorized as a constraint. Either way, the important thing is that they are decisions that have already been made and must be honored. Typically, a constraint cannot be changed, and the software architect does not have any control over it. However, if you have reasons why you believe a constraint should be changed or removed, there may be situations in which you could provide your input.

Constraints can pertain to a number of aspects of a software project. The following are some examples of constraints:

- An organization might have an existing agreement with a particular vendor or has already purchased a certain technology or tool that you will be required to use
- There may be a law or regulation that the software must follow

- There may be a particular deadline for a milestone or the final delivery of the software that cannot be altered
- Management might dictate that a certain number of resources be assigned to the project, or that the project must utilize outsourced resources
- If the development team already exists and they are skilled in a particular programming language, the organization may require that programming language be used

Constraints should be considered while designing the solution, just like other types of requirements.

The importance of requirements engineering

The importance of requirements analysis cannot be overstated. Proper requirements analysis is crucial for a successful project since it affects all of the subsequent phases. If it isn't done properly, additional work will be required, resulting in time and cost overruns.

The later in the life cycle such problems are encountered, the more it will cost and the longer it will take to correct those mistakes. When a problem with requirements is discovered later in the life cycle, the deliverables that have already been produced in subsequent phases, such as design and development, may require refactoring. In *Code Complete (Second Edition)*, Steve McConnell explains that *the principle is to find an error as close as possible to the time at which it was introduced. The longer the defect stays in the software food chain, the more damage it causes further down the chain*.

Some of the many benefits of proper requirements analysis include:

- Reduced rework
- Fewer unnecessary features
- Lower enhancement costs
- Faster development
- Reduced development costs
- Better communication
- More accurate system testing estimates
- Higher customer satisfaction levels

It is imperative that management understands the importance and benefits of proper requirements engineering. If they do not, an attempt must be made to communicate that to them so that proper time can be scheduled for requirements engineering.

Software requirements must be measurable and testable

When defining software requirements, they should be complete in that all of them are defined, and consistent in that they are clear and do not contradict each other. Each software requirement should be unambiguous, measurable, and testable. Testing should be considered when requirements are written. Requirements need to be specific enough that they can be verified.

Business analysts and other stakeholders who are defining the requirements must write them in a way so that they are measurable and testable. As a software architect, if you see requirements that do not satisfy these conditions, you need to point them out so that they can be modified.

If a requirement is to be considered measurable, it should provide specific values or limits. In order for a requirement to be testable, there must be a practical, cost-effective way to determine whether the requirement has been satisfied. It must be possible to write a test case that can verify whether or not the requirement has been met.

For example, consider a requirement that states that *the web page must load in a timely manner*. What exactly does that mean? Stakeholders and the development team may have a different understanding of what will satisfy such a requirement. It should be written with a specific limit in mind, such as *the web page must load within two seconds*.

A common understanding and mutually agreed upon expectations need to be set with stakeholders so that there are no surprises when the final product is delivered.

Software requirements that affect architecture

As a software architect who is designing an architecture that will satisfy the requirements, complete and validated requirements are crucial to your job. Requirements, particularly the quality attributes, can greatly affect the architectural design.

However, the degree to which a particular requirement has an effect on the architecture varies. Some do not have any effect, while others have a profound one. You must be able to recognize the requirements that may affect decisions you are making architecturally.

Many times, the requirements that affect a software architecture's design are quality attributes. Therefore, you should pay particular attention to those. Be aware though that it is common for the definition of quality attributes to be lacking when defining requirements. Stakeholders may focus on functional requirements, and may not define quality attributes, or if they do, they may do so in a way that is not measurable and testable.

Software architects may need to make an extra effort to understand the quality attributes that are important to the stakeholders, and the values expected to make them testable to get the quality attributes defined and documented. In the next section, let's examine ways in which you might elicit requirements, including quality attributes, from stakeholders.

Requirements elicitation

Perhaps you have heard of the terms *known knowns*, *known unknowns*, and *unknown unknowns*. They are used to describe that which we know about, that which we are aware of but do not know about, and that which we are not even considering because we do not know about them.

Ideally, the requirements and business domain of a software project are well understood. However, the development team may not have such an understanding from the onset of the project. Even for those who do have the knowledge, such as some of the stakeholders, they may not know exactly what they want from the software.

As a result, you will be dealing with both knowns and unknowns. Part of requirements engineering involves gaining as much knowledge as possible regarding the requirements of the system we want to build. We seek to eliminate the *unknown unknowns* and consider as many of the requirements as possible when designing the software.

The start of that process is to elicit requirements from stakeholders, which is known as **requirements gathering** or **requirements elicitation**. Requirements gathering seems to imply simply collecting requirements that are easy to discover, although it typically involves much more than that. Often, it is necessary to *elicit* the requirements from stakeholders because not all of them are at the forefront of the thoughts of stakeholders. It is more of a proactive, and not a reactive, process.

As Andrew Hunt and David Thomas point out in *The Pragmatic Programmer*:

> *"Requirements rarely lie on the surface. Normally, they are buried deep beneath layers of assumptions, misconceptions, and politics."*

Techniques to elicit requirements

Obtaining information from stakeholders takes effort, but there are proven techniques that can assist you to draw them out. Each technique has its own advantages and disadvantages, so select the ones that are most likely to work given your situation. Don't forget that you can use more than one of these techniques. Using multiple techniques in conjunction with each other may yield the best results.

Interviews

One way to elicit requirements is to conduct **interviews** with stakeholders. Interviews for this purpose can either be conducted formally or informally. Each interview session should either be with a single person or a small group. If it is with more than one person, you don't want to have too many people in the session or you risk not getting the maximum amount of information from each individual stakeholder.

One or more people can ask questions, and at least one person should be designated to take notes. Ask open-ended questions to spur discussion and get information, and closed-ended questions can be used to confirm facts.

As with all of the techniques, the success of interviews depends on the knowledge of the interviewee, as well as their willingness to participate. It is good to interview different types of stakeholders in order to get different perspectives. You need to take into consideration their knowledge and experience when reviewing the results of the interviews. Interviews are not always a good way to reach consensus because not all of the stakeholders may be present, but they could be effective to obtain information.

Requirements workshops

Requirements workshops are one of the most common and effective elicitation methods. They are used to collect and prioritize requirements. A group of relevant stakeholders is invited to attend a session in which they will provide their feedback. An inevitable result of having such discussions will be a higher level of clarity on how the software should work, and what it is required to do.

A clear agenda should be set for each requirements workshop. The scope varies, but you might want to keep each session restricted to a certain part of the business process or software application. Someone should be designated as the facilitator who can run the meeting, and a different person should take notes.

The duration of a requirements workshop can vary and is dependent on the scope. It can last anywhere from an hour to several days. The length of the workshop should be appropriate for its scope.

You can acquire quite a bit of information from a requirements workshop. Just be sure to have an ideal number of people participate. If there are too many, the process could be slowed down, and some people may not have an opportunity to share their thoughts. On the other hand, if there are not enough attendees, then you may not gather enough information.

Sometimes it can be difficult to get all of the stakeholders in one place at the same time. If you can't arrange that, you could consider running multiple workshops on the same topic.

Brainstorming

Brainstorming sessions involve getting thoughts from a group spontaneously and documenting those thoughts. It can be a fun, productive way to get requirements for a system. If you are going to conduct such a session, make sure you invite the relevant stakeholders. If there are many stakeholders, you may want to consider holding multiple sessions and keeping the attendance of each one to between five and ten people.

When inviting the stakeholders, make sure you have a variety. Different types of stakeholder will have different perspectives and may provide ideas that others would not have considered.

The brainstorming session should have clear goals, and each session should not be too broad. You may conduct a brainstorming session to get requirements for a specific piece of functionality within the software system.

Try to hold the brainstorming session in a relaxed, comfortable environment so that the participants feel comfortable sharing their ideas. Someone should be designated as the facilitator. The facilitator may need to encourage participation, especially at the beginning of the session, since some participants may hold back their thoughts. As a software architect who wants the group to generate ideas, you can take it upon yourself to come up with the first idea to encourage others.

Criticism of ideas shouldn't be tolerated, as you don't want to discourage anyone from participating further. While there may not be any bad ideas, sometimes there will be thoughts that are not relevant to the goals of the meeting. If a discussion does get off topic, the facilitator should limit the discussion and steer it in another direction.

Either the facilitator or another person should take notes, preferably on a whiteboard so that everyone can see the ideas that have previously been given. For remote meetings where everyone is not in a room together, someone should share their screen so that the ideas are visible.

There should be a time limit so that everyone is aware when the session will end. If there is a clear ending where no more new ideas are being generated, the session could be called off early.

Observation

Observation is a technique where someone studies a stakeholder in their work environment, performing tasks related to the software project. It is particularly useful when you are attempting to understand a current process. It is effective because the observer may notice things that aren't mentioned through other elicitation techniques. Stakeholders may forget certain requirements or may not even be aware that what they are doing is a requirement that needs to be documented. By observing the actual work performed, you can sometimes gather important information.

This technique can either be performed in a passive way or in an active way, depending on what is agreed upon and what would be most effective. If the observer is passive, then he or she makes every attempt not to be disruptive. The observer does not ask many questions, nor do they interrupt the tasks that the stakeholder is performing. If the observer is active, then they can have an ongoing dialog with the stakeholder and ask questions while they are performing their tasks.

There are disadvantages to this technique. It can be time-consuming to observe someone performing his or her daily work. The person who is being observed may find it disruptive to their work, so you may only get limited time to perform the observation.

Even if a lot of time is spent observing, you may not witness all of the possible scenarios, and being aware of scenarios that do not happen as frequently is still important for the requirements of the software.

Although you shouldn't use this technique as the only one, it can be useful as a supplement to other techniques because it may draw out requirements that would not be revealed using other elicitation techniques.

Focus groups

Focus groups can be organized to elicit requirements. This technique is more formal than brainstorming and involves inviting a group of participants to provide feedback. This technique is commonly used for public applications that will have external users. In that case, the invited participants are users or outside experts who are external to the organization.

A moderator runs the session. The selected moderator is often skilled at running focus groups and is hired specifically to perform that role. The moderator asks the questions and encourages the participation of all the participants. Moderators should remain neutral during the session.

The questions asked during a focus group are typically open-ended and promote discussion. Responses in a focus group are typically spoken, as opposed to written. In that type of setting, things like nonverbal communication and group interaction can be observed. Focus group participants can foster new ideas from each other. This technique can be faster than conducting interviews individually.

Despite the advantages, there are some disadvantages to this technique. Focus groups run the risk of individuals following the crowd as they hear the feedback from others in the group. Some people are hesitant about sharing their ideas in a group setting, and the moderator may need to be paid to conduct the focus group.

Surveys

Surveys can be created and given to stakeholders to get information. Surveys should have a clear purpose. Rather than create a large survey that covers many topics, it may be more effective to create multiple surveys, each covering a portion of the business processes or software application. Some people will be averse to filling out extremely long surveys.

The questions in the survey should be well thought out, clear, and concise. Although surveys can have open-ended questions, typically the questions in a survey are closed-ended ones. This makes it easier for participants to provide answers, and, more importantly, the answers will be easier to analyze. If you do want to ask open-ended questions in a survey, keep in mind that it will require more effort to analyze the answers.

Document analysis

Document analysis utilizes existing documentation to obtain information and requirements. The documentation may cover the relevant business processes or existing software systems. If there is an existing system in use, it can serve as a starting point for the requirements of the new system. The documentation may come in the form of technical documentation, user manuals, contracts, statements of work, emails, training materials, and anything else that may be of use.

The documentation may even be **commercial off-the-shelf** (**COTS**) software package manuals. There may be existing software that provides part or all of the functionality you are seeking to implement, and by analyzing that documentation you can get the requirements for your software system.

Analyzing documents is particularly useful if stakeholders are not available for other requirements elicitation techniques.

Prototyping

Prototyping is a requirements elicitation technique that involves building a prototype that stakeholders can use to some degree, or at least see. Some people are more visually-oriented than others, and having a prototype can trigger ideas regarding requirements.

The disadvantage of prototyping is that it can take time to build a prototype. However, with most modern technologies a prototype can be built quickly. There is also the option of simply creating visual diagrams of the software, rather than a prototype. For web applications, this involves creating wireframes, which are visual representations of web pages that let a person see the layout and structure of web pages.

The scope of a prototype can be as broad or narrow as you want it to be. While it can demonstrate an entire application, it could be focused on a specific piece of functionality. Prototyping can be useful in conjunction with other techniques so that you can validate requirements and uncover things that had not already been discussed.

Prototyping can also be taken to a different level in which a working version of the software is produced. In a situation where the direction and purpose of the software have not been fully evaluated yet, perhaps because the stakeholders don't know where to begin, or they have many ideas but cannot agree among themselves, an initial prototype can be developed. If you are using an agile methodology, a few initial iterations can take place, each ending with a working version of the software that can be shared with the stakeholders.

Once they have something concrete to look at and use, it may inspire them with new ideas and approaches. Everyone will be able to see what works, and just as important, what does not work. If done properly, as refactoring occurs and further iterations take place, the requirements will become more apparent as the software takes shape.

Reverse engineering

Reverse engineering is a method in which existing code is analyzed to determine the requirements. It is similar to the document analysis technique in that it assumes that there are existing artifacts to analyze. This is not always the case when designing a new software system. It also requires access to the source code, and someone with the technical skill to analyze the code and extract requirements from it.

It is a time-consuming technique but might be used as a last resort if other techniques are not possible. For example, if stakeholders are not available to you, or the ones who are available to you are not knowledgeable, many of the other techniques may not be viable. If there is also a lack of documentation, then document analysis may also not be possible.

This method is not just a final course of action when others are not possible though. When appropriately used, it can be a powerful technique. Stakeholders may have limited perspectives, and may not think of everything that the software is required to do. If there is an existing system, looking at the code is a way to determine exactly what needs to happen.

Get access to the proper stakeholders

Even armed with techniques to elicit requirements, it can be difficult to get them from the proper stakeholders. You may find yourself in situations where certain stakeholders are not made available to you. You may also find yourself in a situation where certain stakeholders, for various reasons, are not being helpful or do not want to participate in the project.

Due to the importance of requirements analysis, you must make the effort to get access to the relevant stakeholders. This may involve speaking with management to get the proper access. Although this may be easier if you work for the same organization, many stakeholders will be external to the organization. The success of the project may depend on it, so you may need to escalate this need to your own management or to the management of the stakeholder's organization.

Summary

Being an effective software architect means understanding the domain of the software you are building. Gaining knowledge of general business topics and a deep understanding of the organization's business is the foundation for becoming an expert on the problem space for which you will be designing a solution.

DDD is a proven approach to modeling a domain. Creating a ubiquitous language that will simplify and facilitate communication between everyone involved in the software project, and working with domain experts, will facilitate learning a particular domain.

Other practices, such as separating your domain into subdomains and creating bounded contexts in your domain model, will minimize complexity and allow you and your team to firmly grasp even complex domains.

One of the keys to building software successfully is proper requirements engineering, including knowing how to effectively elicit requirements from stakeholders. Knowing the requirements for the software is crucial to designing an appropriate solution.

In the next chapter, we will further explore one of the most important types of software requirements: quality attributes. Building quality software requires the software architect to know and understand the details of the quality attributes that are important to the stakeholders.