INFORMATICA
COMMUNICATIE
ACADEMIE

reader

software guidebook

year:	2015 - 2016
program:	informatics
owe:	develop a web application project
authors:	Maria Boes, Lars Tijsma, Theo Theunissen, Sander Leer, Pim Haenen & Robert Holwerda
version:	1.0

HAN

about

this reader contains part 5 of (Brown, 2015).

November 25, 2015
Lars Tijsma

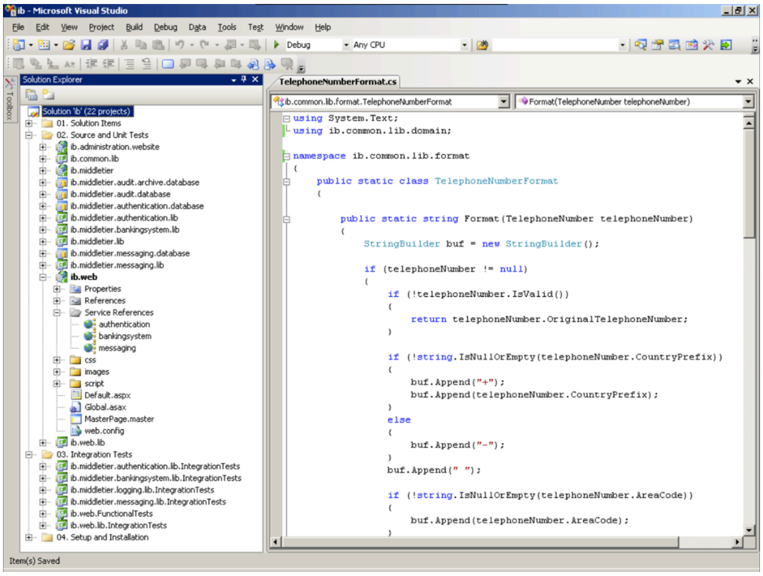
V Documenting software

This part of the book is about that essential topic we love to hate - writing documentation!

48. The code doesn't tell the whole story

We all know that writing good code is important and refactoring forces us to think about making methods smaller, more reusable and self-documenting. Some people say that comments are bad and that self-commenting code is what we should strive for. However you do it, everybody *should* strive for good code that's easy to read, understand and maintain. But the code doesn't tell the whole story.

Let's imagine that you've started work on a new software project that's already underway. The major building blocks are in place and some of the functionality has already been delivered. You start up your development machine, download the code from the source code control system and load it up into your development environment. What do you do next and how do you start being productive?


ib - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Test Window Help
Debug
Any CPU
Solution Explorer
Solution 'ib' (22 projects)
01. Solution Items
02. Source and Unit Tests
ib.administration.website
ib.common.lib
ib.middletier
ib.middletier.audit.archive.database
ib.middletier.audit.database
ib.middletier.authentication.database
ib.middletier.authentication.lib
ib.middletier.bankingsystem.lib
ib.middletier.lib
ib.middletier.messaging.database
ib.middletier.messaging.lib
ib.web
Properties
References
Service References
authentication
bankingsystem
messaging
css
images
script
Default.aspx
Global.asax
MasterPage.master
web.config
ib.web.lib
03. Integration Tests
ib.middletier.authentication.lib.IntegrationTests
ib.middletier.bankingsystem.lib.IntegrationTests
ib.middletier.logging.lib.IntegrationTests
ib.middletier.messaging.lib.IntegrationTests
ib.web.FunctionalTests
ib.web.lib.IntegrationTests
04. Setup and Installation
Item(s) Saved
TelephoneNumberFormat.cs
ib.common.lib.format.TelephoneNumberFormat
Format(TelephoneNumber telephoneNumber)

```
using System.Text;
using ib.common.lib.domain;

namespace ib.common.lib.format
{
    public static class TelephoneNumberFormat
    {

        public static string Format(TelephoneNumber telephoneNumber)
        {
            StringBuilder buf = new StringBuilder();

            if (telephoneNumber != null)
            {
                if (!telephoneNumber.IsValid())
                {
                    return telephoneNumber.OriginalTelephoneNumber;
                }

                if (!string.IsNullOrEmpty(telephoneNumber.CountryPrefix))
                {
                    buf.Append("+");
                    buf.Append(telephoneNumber.CountryPrefix);
                }
                else
                {
                    buf.Append("-");
                }
                buf.Append(" ");

                if (!string.IsNullOrEmpty(telephoneNumber.AreaCode))
                {
                    buf.Append(telephoneNumber.AreaCode);
                }
```

Where do you start?

If nobody has the time to walk you through the codebase, you can start to make your own assumptions based upon the limited knowledge you have about the project, the business domain, your expectations of how the team builds software and your knowledge of the technologies in use.

For example, you might be able to determine something about the overall architecture of the software system through how the codebase has been broken up into sub-projects, directories, packages, namespaces, etc. Perhaps there are some naming conventions in use. Even from the previous static screenshot of Microsoft Visual Studio, we can determine a number of characteristics about the software, which in this case is an (anonymised) Internet banking system.

- The system has been written in C# on the Microsoft .NET platform.
- The overall .NET solution has been broken down into a number of Visual Studio projects and there's a .NET web application called "ib.web", which you'd expect since this is an Internet banking system ("ib" = "Internet Banking").
- The system appears to be made up of a number of architectural tiers. There's "ib.web" and "ib.middletier", but I don't know if these are physical or logical tiers.
- There looks to be a naming convention for projects. For example, "ib.middletier.authentication.lib", "ib.middletier.messaging.lib" and "ib.middletier.bankingsystem.lib" are class libraries that seem to relate to the middle-tier. Are these simply a logical grouping for classes or something more significant such as higher level components and services?
- With some knowledge of the technology, I can see a "Service References" folder lurking underneath the "ib.web" project. These are Windows Communication Foundation (WCF) service references that, in the case of this example, are essentially web service clients. The naming of them seems to correspond to the class libraries within the middle-tier, so I think we actually have a distributed system with a middle-tier that exposes a number of well-defined services.

The code doesn't portray the intent of the design

A further deep-dive through the code will help to prove your initial assumptions right or wrong, but it's also likely to leave you with a whole host of questions. Perhaps you understand what the system *does* at a high level, but you don't understand things like:

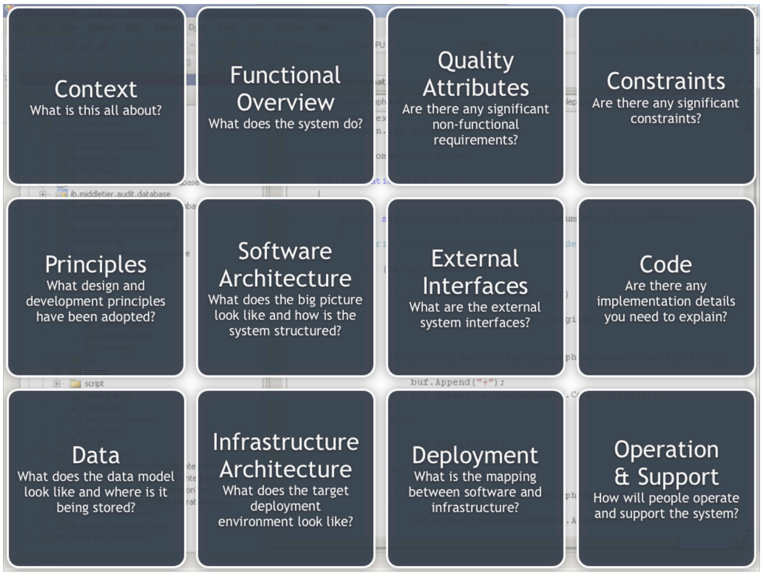
- How the software system fits into the existing system landscape.
- Why the technologies in use were chosen.
- The overall structure of the software system.
- Where the various components are deployed at runtime and how they communicate.

- How the web-tier “knows” where to find the middle-tier.
- What approach to logging/configuration/error handling/etc has been adopted and whether it is consistent across the codebase.
- Whether any common patterns and principles are in use across the codebase.
- How and where to add new functionality.
- How security has been implemented across the stack.
- How scalability is achieved.
- How the interfaces with other systems work.
- etc

I’ve been asked to review and work on systems where there has been no documentation. You can certainly gauge the answers to most of these questions from the code but it can be hard work. Reading the code will get you so far but you’ll probably need to ask questions to the rest of the team at some point. And if you don’t ask the right questions, you won’t get the right answers - you don’t know what you don’t know.

Supplementary information

With any software system, there’s another layer of information sitting above the code that provides answers to these types of questions and more.


Context
What is this all about?
Functional Overview
What does the system do?
Quality Attributes
Are there any significant non-functional requirements?
Constraints
Are there any significant constraints?
Principles
What design and development principles have been adopted?
Software Architecture
What does the big picture look like and how is the system structured?
External Interfaces
What are the external system interfaces?
Code
Are there any implementation details you need to explain?
Data
What does the data model look like and where is it being stored?
Infrastructure Architecture
What does the target deployment environment look like?
Deployment
What is the mapping between software and infrastructure?
Operation & Support
How will people operate and support the system?

There's an additional layer of information above the code

This type of information is complementary to the code and should be captured somewhere, for example in lightweight supplementary documentation to describe what the code itself doesn't. The code tells *a* story, but it doesn't tell the whole story.

49. Software documentation as a guidebook

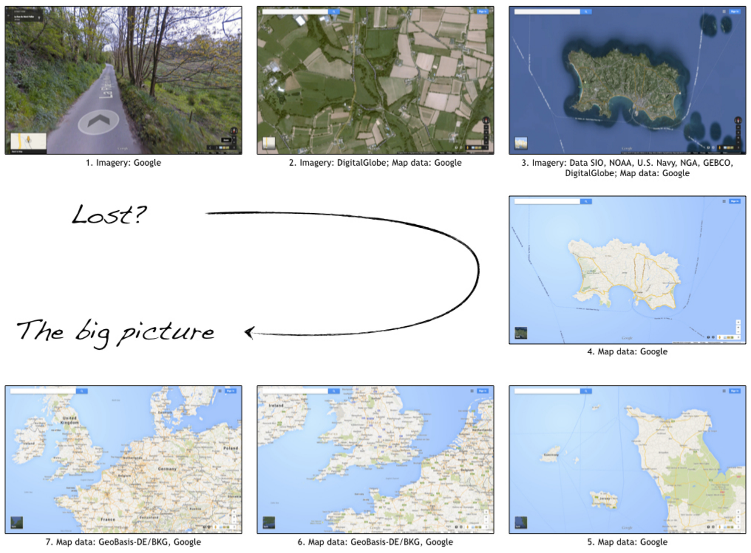
“Working software over comprehensive documentation” is what the Manifesto for Agile Software Development says and it’s incredible to see how many software teams have interpreted those five words as “don’t write *any* documentation”. The underlying principle here is that real working software is much more valuable to end-users than a stack of comprehensive documentation but many teams use this line in the agile manifesto as an excuse to not write any documentation at all. Unfortunately the code doesn’t tell the whole story and not having a source of supplementary information about a complex software system can slow a team down as they struggle to navigate the codebase.

I’m also a firm believer that many software teams have a duty to deliver some supplementary documentation along with the codebase, especially those that are building the software under an outsourcing and/or offshoring contract. I’ve seen IT consulting organisations deliver highly complex software systems to their customers without a single piece of supporting documentation, often because the team doesn’t *have* any documentation. If the original software developers leave the consulting organisation, will the new team be able to understand what the software is all about, how it’s been built and how to enhance it in a way that is sympathetic to the original architecture? And what about the poor customer? Is it right that they should *only* be delivered a working codebase?

The problem is that when software teams think about documentation, they usually think of huge Microsoft Word documents based upon a software architecture document template from the 1990’s that includes sections where they need to draw Unified Modeling Language (UML) class diagrams for every use case that their software supports. Few people enjoy reading this type of document, let alone writing it! A different approach is needed. We should think about supplementary documentation as an ever-changing travel guidebook rather than a comprehensive static piece of history. But what goes into a such a guidebook?

1. Maps

Let’s imagine that I teleported you away from where you are now and dropped you in a quiet, leafy country lane somewhere in the world (picture 1). Where are you and how do you figure out the answer to this question? You could shout for help, but this will only work if there are other people in the vicinity. Or you could simply start walking until you recognised something or encountered some civilisation, who you could then ask for help. As geeks though, we would probably fire up the maps application on our smartphone and use the GPS to pinpoint our location (picture 2).


1. Imagery: Google
2. Imagery: DigitalGlobe; Map data: Google
3. Imagery: Data SIO, NOAA, U.S. Navy, NGA, GEBCO, DigitalGlobe; Map data: Google
Lost?
4. Map data: Google
The big picture
7. Map data: GeoBasis-DE/BKG, Google
6. Map data: GeoBasis-DE/BKG, Google
5. Map data: Google

From the detail to the big picture

The problem with picture 2 is that although it may show our location, we're a little too "zoomed in" to potentially make sense of it. If we zoom out further, eventually we'll get to see that I teleported you to a country lane in Jersey (picture 3).

The next issue is that the satellite imagery is showing a lot of detail, which makes it hard to see where we are relative to some of the significant features of the island, such as the major roads and places. To counter this, we can remove the satellite imagery (picture 4). Although not as detailed, this abstraction allows us to see some of the major structural elements of the island along with some of the place names, which were perviously getting obscured by the detail. With our simplified view of the island, we can zoom out further until we get to a big picture showing exactly where Jersey is in Europe (pictures 5, 6 and 7). All of these images show the same location from different levels of abstraction, each of which can help you to answer different questions.

If I were to open up the codebase of a complex software system and highlight a random line of code, exploring is fun but it would take a while for you to understand where you were and how the code fitted into the software system as a whole. Most integrated development environments have a way to navigate the code by namespace, package or folder but often the physical structure of the codebase is different to the logical structure. For example, you may have many classes that make up a single component, and many of those components may make up a single deployable unit.

Diagrams can act as maps to help people navigate a complex codebase and this is one of the most important parts of supplementary software documentation. Ideally there should be a small number of simple diagrams, each showing a different part of the software system or level of abstraction. My C4 approach is how *I* summarise the static structure of a software system but there are others including the use of UML.

2. Sights

If you ever visit Jersey, and you should because it's beautiful, you'll probably want a map. There are visitor maps available at the ports and these present a simplified view of what Jersey looks like. Essentially the visitor maps are detailed sketches of the island and, rather than showing every single building, they show an abstract view. Although Jersey is small, once unfolded, these maps can look daunting if you've not visited before, so what you ideally need is a list of the major points of interest and sights to see. This is one of the main reasons that people take a travel guidebook on holiday with them. Regardless of whether it's physical or virtual (e.g. an e-book on your smartphone), the guidebook will undoubtedly list out the top sights that you should make a visit to.

A codebase is no different. Although we *could* spend a long time diagramming and describing every single piece of code, there's really little value in doing that. What we really need is something that lists out the points of interest so that we can focus our energy on understanding the major elements of the software without getting bogged down in all of the detail. Many web applications, for example, are actually fairly boring and rather than understanding how each of the 200+ pages work, I'd rather see the points of interest. These may include things like the patterns that are used to implement web pages and data access strategies along with how security and scalability are handled.

3. History and culture

If you do ever visit Jersey, and you really should because it *is* beautiful, you may see some things that look out of kilter with their surroundings. For example, we have a lovely granite stone castle on the south coast of the island called Elizabeth Castle that was built in the

16th century. As you walk around admiring the architecture, eventually you'll reach the top where it looks like somebody has dumped a large concrete cylinder, which is not in keeping with the intricate granite stonework generally seen elsewhere around the castle. As you explore further, you'll see signs explaining that the castle was refortified during the German occupation in the second world war. Here, the history helps explain why the castle is the way that it is.

Again, a codebase is no different and some knowledge of the history, culture and rationale can go a long way in helping you understand why a software system has been designed in the way it was. This is particularly useful for people who are new to an existing team.

4. Practical information

The final thing that travel guidebooks tend to include is practical information. You know, all the useful bits and pieces about currency, electricity supplies, immigration, local laws, local customs, how to get around, etc.

If we think about a software system, the practical information might include where the source code can be found, how to build it, how to deploy it, the principles that the team follow, etc. It's all of the stuff that can help the development team do their job effectively.

Keep it short, keep it simple

Exploring is great fun but ultimately it takes time, which we often don't have. Since the code doesn't tell the whole story, *some* supplementary documentation can be very useful, especially if you're handing over the software to somebody else or people are leaving and joining the team on a regular basis. My advice is to think about this supplementary documentation as a guidebook, which should give people enough information to get started and help them accelerate the exploration process. Do resist the temptation to go into too much technical detail though because the technical people that will understand that level of detail will know how to find it in the codebase anyway. As with everything, there's a happy mid-point somewhere.

The following headings describe what you might want to include in a software guidebook:

1. Context
2. Functional Overview
3. Quality Attributes

4. Constraints
5. Principles
6. Software Architecture
7. External Interfaces
8. Code
9. Data
10. Infrastructure Architecture
11. Deployment
12. Operation and Support
13. Decision Log

Beware of the "views"

Many typical software architecture document templates aren't actually too bad as a starting point for supplementary documentation, but often the names of the various sections confuse people. If you glance over the list of section headings that I've just presented, you might be wondering where the typical software architecture "views" are.

If you've not seen these before, there are a number of different ways to look at a software system. Examples include IEEE 1471, ISO/IEC/IEEE 42010, Philippe Kruchten's 4+1 model, etc. What they have in common is that they all provide different "views" onto a software system to describe different aspects of it. For example, there's often a "logical view", a "physical view", a "development view" and so on.

The big problem I've found with many of these approaches is that it starts to get confusing very quickly if people aren't versed in the terminology used. For example, I've heard people argue about what the difference between a "conceptual view" and a "logical view" is. And let's not even start asking questions about whether technology is permitted in a logical view! Perspective is important too. If I'm a software developer, is the "development view" about the code, or is that the "implementation view"? But what about the "physical view"? I mean, code is the physical output, right? But then "physical view" means something different to infrastructure architects. But what if the target deployment environment is virtual rather than physical?

My advice is, however you write documentation, just be clear on what it is you're trying to communicate and name the section accordingly. One option to resolve the terminology issue is to ensure that everybody on the team can point to a clear definition of what the various architectural views are. Software Systems Architecture by Eoin Woods and Nick Rozanski comes highly recommended in this regard. Another approach is to simply rename the sections to remove any ambiguity.

Product vs project documentation

As a final note, the style of documentation that I'm referring to here is related to the *product* being built rather than the *project* that is creating/changing the product. A number of organisations I've worked with have software systems approaching twenty years old and, although they have varying amounts of *project-level* documentation, there's often nothing that tells the story of how the product works and how it's evolved. Often these organisations have a single product (software system) and every major change is managed as a separate project. This results in a huge amount of change over the course of twenty years and a considerable amount of project documentation to digest in order to understand the current state of the software. New joiners in such environments are often expected to simply read the code and fill in the blanks by tracking down documentation produced by various project teams, which is time-consuming to say the least!

I recommend that software teams create a single software guidebook for every software system that they build. This doesn't mean that teams shouldn't create project-level documentation, but there should be a single place where somebody can find information about how the product works and how it's evolved over time. Once a single software guidebook is in place, every project/change-stream/timebox to change that system is exactly that - a small delta. A single software guidebook per product makes it much easier to understand the current state and provides a great starting point for future exploration.

50. Context

A context section should be one of the first sections of the software guidebook and simply used to set the scene for the remainder of the document.

Intent

A context section should answer the following types of questions:

- What is this software project/product/system all about?
- What is it that's being built?
- How does it fit into the existing environment? (e.g. systems, business processes, etc)
- Who is using it? (users, roles, actors, personas, etc)

Structure

The context section doesn't need to be long; a page or two is sufficient and a context diagram is a great way to tell most of the story.

Motivation

I've seen software architecture documents that don't start by setting the scene and, 30 pages in, you're still none the wiser as to why the software exists and where it fits into the existing IT environment. A context section doesn't take long to create but can be immensely useful, especially for those outside of the team.

Audience

Technical and non-technical people, inside and outside of the immediate software development team.

Required

Yes, all software guidebooks should include an initial context section to set the scene.

51. Functional Overview

Even though the purpose of a software guidebook isn't to explain what the software does in detail, it can be useful to expand on the context and summarise what the major functions of the software are.

Intent

This section allows you to summarise what the key functions of the system are. It also allows you to make an explicit link between the functional aspects of the system (use cases, user stories, etc) and, if they are significant to the architecture, to explain why. A functional overview should answer the following types of questions:

- Is it clear what the system actually does?
- Is it clear which features, functions, use cases, user stories, etc are significant to the architecture and why?
- Is it clear who the important users are (roles, actors, personas, etc) and how the system caters for their needs?
- It is clear that the above has been used to shape and define the architecture?

Alternatively, if your software automates a business process or workflow, a functional view should answer questions like the following:

- Is it clear what the system does from a process perspective?
- What are the major processes and flows of information through the system?

Structure

By all means refer to existing documentation if it's available; and by this I mean functional specifications, use case documents or even lists of user stories. However, it's often useful to summarise the business domain and the functionality provided by the system. Again, diagrams can help, and you could use a UML use case diagram or a collection of simple wireframes showing the important parts of the user interface. Either way, remember that the purpose of this section is to provide an *overview*.

Alternatively, if your software automates a business process or workflow, you could use a flow chart or UML activity diagram to show the smaller steps within the process and how they fit together. This is particularly useful to highlight aspects such as parallelism, concurrency, where processes fork or join, etc.

Motivation

This doesn't necessarily need to be a long section, with diagrams being used to provide an overview. Where a context section summarises how the software fits into the existing environment, this section describes what the system actually does. Again, this is about providing a summary and setting the scene rather than comprehensively describing every user/system interaction.

Audience

Technical and non-technical people, inside and outside of the immediate software development team.

Required

Yes, all software guidebooks should include a *summary* of the functionality provided by the software.

52. Quality Attributes

With the functional overview section summarising the functionality, it's also worth including a separate section to summarise the quality attributes/non-functional requirements.

Intent

This section is about summarising the key quality attributes and should answer the following types of questions:

- Is there a clear understanding of the quality attributes that the architecture must satisfy?
- Are the quality attributes SMART (specific, measurable, achievable, relevant and timely)?
- Have quality attributes that are usually taken for granted been explicitly marked as out of scope if they are not needed? (e.g. "user interface elements will only be presented in English" to indicate that multi-language support is not explicitly catered for)
- Are any of the quality attributes unrealistic? (e.g. true 24x7 availability is typically very costly to implement *inside* many organisations)

In addition, if any of the quality attributes are deemed as "architecturally significant" and therefore influence the architecture, why not make a note of them so that you can refer back to them later in the document.

Structure

Simply listing out each of the quality attributes is a good starting point. Examples include:

- Performance (e.g. latency and throughput)
- Scalability (e.g. data and traffic volumes)
- Availability (e.g. uptime, downtime, scheduled maintenance, 24x7, 99.9%, etc)
- Security (e.g. authentication, authorisation, data confidentiality, etc)

- Extensibility
- Flexibility
- Auditing
- Monitoring and management
- Reliability
- Failover/disaster recovery targets (e.g. manual vs automatic, how long will this take?)
- Business continuity
- Interoperability
- Legal, compliance and regulatory requirements (e.g. data protection act)
- Internationalisation (i18n) and localisation (L10n)
- Accessibility
- Usability
- …

Each quality attribute should be precise, leaving no interpretation to the reader. Examples where this isn't the case include:

- "the request must be serviced quickly"
- "there should be no overhead"
- "as fast as possible"
- "as small as possible"
- "as many customers as possible"
- …

Motivation

If you've been a good software architecture citizen and have proactively considered the quality attributes, why not write them down too? Typically, quality attributes are not given to you on a plate and an amount of exploration and refinement is usually needed to come up with a list of them. Put simply, writing down the quality attributes removes any ambiguity both now and during maintenance/enhancement work in the future.

Audience

Since quality attributes are mostly technical in nature, this section is really targeted at technical people in the software development team.

Required

Yes, all software guidebooks should include a summary of the quality attributes/non-functional requirements as they usually shape the resulting software architecture in some way.

53. Constraints

Software lives within the context of the real-world, and the real-world has constraints. This section allows you to state these constraints so it's clear that you are working within them and obvious how they affect your architecture decisions.

Intent

Constraints are typically imposed upon you but they aren't necessarily "bad", as reducing the number of available options often makes your job designing software easier. This section allows you to explicitly summarise the constraints that you're working within and the decisions that have already been made for you.

Structure

As with the quality attributes, simply listing the known constraints and briefly summarising them will work. Example constraints include:

- Time, budget and resources.
- Approved technology lists and technology constraints.
- Target deployment platform.
- Existing systems and integration standards.
- Local standards (e.g. development, coding, etc).
- Public standards (e.g. HTTP, SOAP, XML, XML Schema, WSDL, etc).
- Standard protocols.
- Standard message formats.
- Size of the software development team.
- Skill profile of the software development team.
- Nature of the software being built (e.g. tactical or strategic).
- Political constraints.
- Use of internal intellectual property.
- etc

If constraints do have an impact, it's worth summarising them (e.g. what they are, why they are being imposed and who is imposing them) and stating how they are significant to your architecture.

Motivation

Constraints have the power to massively influence the architecture, particularly if they limit the technology that can be used to build the solution. Documenting them prevents you having to answer questions in the future about why you've seemingly made some odd decisions.

Audience

The audience for this section includes everybody involved with the software development process, since some constraints are technical and some aren't.

Required

Yes, all software guidebooks should include a summary of the constraints as they usually shape the resulting software architecture in some way. It's worth making these constraints explicit at all times, even in environments that have a very well known set of constraints (e.g. "all of our software is ASP.NET against a SQL Server database") because constraints have a habit of changing over time.

54. Principles

The principles section allows you to summarise those principles that have been used (or you are using) to design and build the software.

Intent

The purpose of this section is to simply make it explicit which principles you are following. These could have been explicitly asked for by a stakeholder or they could be principles that *you* (i.e. the software development team) want to adopt and follow.

Structure

If you have an existing set of software development principles (e.g. on a development wiki), by all means simply reference it. Otherwise, list out the principles that you are following and accompany each with a short explanation or link to further information. Example principles include:

- Architectural layering strategy.
- No business logic in views.
- No database access in views.
- Use of interfaces.
- Always use an ORM.
- Dependency injection.
- The Hollywood principle (don't call us, we'll call you).
- High cohesion, low coupling.
- Follow SOLID (Single responsibility principle, Open/closed principle, Liskov substitution principle, Interface segregation principle, Dependency inversion principle).
- DRY (don't repeat yourself).
- Ensure all components are stateless (e.g. to ease scaling).
- Prefer a rich domain model.
- Prefer an anaemic domain model.

- Always prefer stored procedures.
- Never use stored procedures.
- Don’t reinvent the wheel.
- Common approaches for error handling, logging, etc.
- Buy rather than build.
- etc

Motivation

The motivation for writing down the list of principles is to make them explicit so that everybody involved with the software development understands what they are. Why? Put simply, principles help to introduce consistency into a codebase by ensuring that common problems are approached in the same way.

Audience

The audience for this section is predominantly the technical people in the software development team.

Required

Yes, all software guidebooks should include a summary of the principles that have been or are being used to develop the software.

55. Software Architecture

The software architecture section is your "big picture" view and allows you to present the structure of the software. Traditional software architecture documents typically refer to this as a "conceptual view" or "logical view", and there is often confusion about whether such views should refer to implementation details such as technology choices.

Intent

The purpose of this section is to summarise the software architecture of your software system so that the following questions can be answered:

- What does the "big picture" look like?
- Is there are clear structure?
- Is it clear how the system works from the "30,000 foot view"?
- Does it show the major containers and technology choices?
- Does it show the major components and their interactions?
- What are the key internal interfaces? (e.g. a web service between your web and business tiers)

Structure

I use the container and component diagrams as the main focus for this section, accompanied by a short narrative explaining what the diagram is showing plus a summary of each container/component.

Sometimes UML sequence or collaboration diagrams showing component interactions can be a useful way to illustrate how the software satisfies the major use cases/user stories/etc. Only do this if it adds value though and resist the temptation to describe how *every* use case/user story works!

Motivation

The motivation for writing this section is that it provides the maps that people can use to get an overview of the software and help developers navigate the codebase.

Audience

The audience for this section is predominantly the technical people in the software development team.

Required

Yes, all software guidebooks should include a software architecture section because it's essential that the overall software structure is well understood by everybody on the development team.

56. External Interfaces

Interfaces, particularly those that are external to your software system, are one of the riskiest parts of any software system so it's very useful to summarise what the interfaces are and how they work.

Intent

The purpose of this section is to answer the following types of questions:

- What are the key external interfaces?
  - e.g. between your system and other systems (whether they are internal or external to your environment)
  - e.g. any APIs that you are exposing for consumption
  - e.g. any files that your are exporting from your system
- Has each interface been thought about from a technical perspective?
  - What is the technical definition of the interface?
  - If messaging is being used, which queues (point-to-point) and topics (pub-sub) are components using to communicate?
  - What format are the messages (e.g. plain text or XML defined by a DT-D/Schema)?
  - Are they synchronous or asynchronous?
  - Are asynchronous messaging links guaranteed?
  - Are subscribers durable where necessary?
  - Can messages be received out of order and is this a problem?
  - Are interfaces idempotent?
  - Is the interface always available or do you, for example, need to cache data locally?
  - How is performance/scalability/security/etc catered for?
- Has each interface been thought about from a non-technical perspective?
  - Who has ownership of the interface?
  - How often does the interface change and how is versioning handled?
  - Are there any service-level agreements in place?

Structure

I tend to simply list out the interfaces (in the form "From X to Y") along with a short narrative that describes the characteristics of the interface. To put the interfaces in context, I may include a simplified version of the container or component diagrams that emphasise the interfaces.

Motivation

The motivation for writing this section is to ensure that the interfaces have been considered and are understood because they're typically risky and easy to ignore. If interface details haven't been captured, this section can then act as a checklist and be a source of work items for the team to undertake.

Audience

The audience for this section is predominantly the technical people in the software development team.

Required

No, I only include this section if I'm building something that has one or more complex interfaces. For example, I wouldn't include it for a standard "web server -> database" style of software system, but I would include this section if that web application needed to communicate with an external system where it was consuming information via an API.

57. Code

Although other sections of the software guidebook describe the overall architecture of the software, often you'll want to present lower level details to explain how things work. This is what the code section is for. Some software architecture documentation templates call this the "implementation view" or the "development view".

Intent

The purpose of the code section is to describe the implementation details for parts of the software system that are important, complex, significant, etc. For example, I've written about the following for software projects that I've been involved in:

- Generating/rendering HTML: a short description of an in-house framework that was created for generating HTML, including the major classes and concepts.
- Data binding: our approach to updating business objects as the result of HTTP POST requests.
- Multi-page data collection: a short description of an in-house framework we used for building forms that spanned multiple web pages.
- Web MVC: an example usage of the web MVC framework that was being used.
- Security: our approach to using Windows Identity Foundation (WIF) for authentication and authorisation.
- Domain model: an overview of the important parts of the domain model.
- Component framework: a short description of the framework that we built to allow components to be reconfigured at runtime.
- Configuration: a short description of the standard component configuration mechanism in use across the codebase.
- Architectural layering: an overview of the layering strategy and the patterns in use to implement it.
- Exceptions and logging: a summary of our approach to exception handling and logging across the various architectural layers.
- Patterns and principles: an explanation of how patterns and principles are implemented.
- etc

Structure

Keep it simple, with a short section for each element that you want to describe and include diagrams if they help the reader. For example, a high-level UML class and/or sequence diagram can be useful to help explain how a bespoke in-house framework works. Resist the temptation to include all of the detail though, and don't feel that your diagrams need to show everything. I prefer to spend a few minutes sketching out a high-level UML class diagram that shows selected (important) attributes and methods rather than using the complex diagrams that can be generated automatically from your codebase with UML tools or IDE plugins. Keeping any diagrams at a high-level of detail means that they're less volatile and remain up to date for longer because they can tolerate small changes to the code and yet remain valid.

Motivation

The motivation for writing this section is to ensure that everybody understands how the important/significant/complex parts of the software system work so that they can maintain, enhance and extend them in a consistent and coherent manner. This section also helps new members of the team get up to speed quickly.

Audience

The audience for this section is predominantly the technical people in the software development team.

Required

No, but I usually include this section for anything other than a trivial software system.

58. Data

The data associated with a software system is usually not the primary point of focus yet it's arguably more important than the software itself, so often it's useful to document something about it.

Intent

The purpose of the data section is to record anything that is important from a data perspective, answering the following types of questions:

- What does the data model look like?
- Where is data stored?
- Who owns the data?
- How much storage space is needed for the data? (e.g. especially if you're dealing with "big data")
- What are the archiving and back-up strategies?
- Are there any regulatory requirements for the long term archival of business data?
- Likewise for log files and audit trails?
- Are flat files being used for storage? If so, what format is being used?

Structure

Keep it simple, with a short section for each element that you want to describe and include domain models or entity relationship diagrams if they help the reader. As with my advice for including class diagrams in the code section, keep any diagrams at a high level of abstraction rather than including every field and property. If people need this type of information, they can find it in the code or database (for example).

Motivation

The motivation for writing this section is that the data in most software systems tends to outlive the software. This section can help anybody that needs to maintain and support the data on an ongoing basis, plus anybody that needs to extract reports or undertake business intelligence activities on the data. This section can also serve as a starting point for when the software system is inevitably rewritten in the future.

Audience

The audience for this section is predominantly the technical people in the software development team along with others that may help deploy, support and operate the software system.

Required

No, but I usually include this section for anything other than a trivial software system.

59. Infrastructure Architecture

While most of the software guidebook is focussed on the software itself, we do also need to consider the infrastructure because software architecture is about software *and* infrastructure.

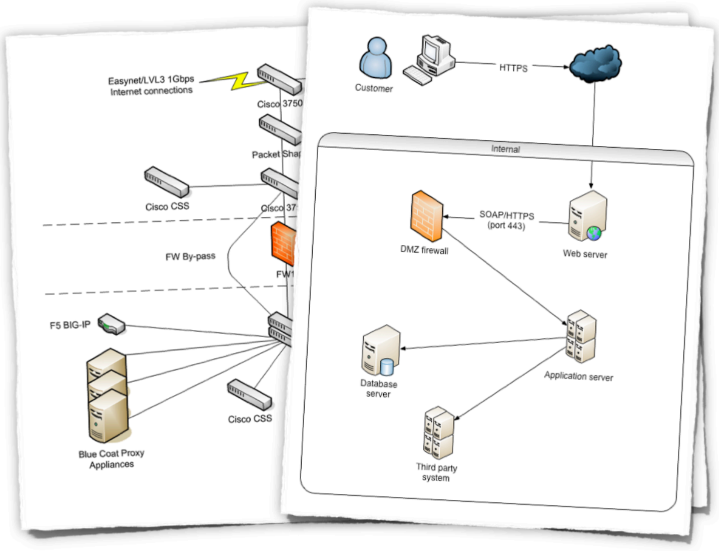
Intent

This section is used to describe the physical/virtual hardware and networks on which the software will be deployed. Although, as a software architect, you may not be involved in designing the infrastructure, you do need to understand that it's sufficient to enable you to satisfy your goals. The purpose of this section is to answer the following types of questions:

- Is there a clear physical architecture?
- What hardware (virtual or physical) does this include across all tiers?
- Does it cater for redundancy, failover and disaster recovery if applicable?
- Is it clear how the chosen hardware components have been sized and selected?
- If multiple servers and sites are used, what are the network links between them?
- Who is responsible for support and maintenance of the infrastructure?
- Are there central teams to look after common infrastructure (e.g. databases, message buses, application servers, networks, routers, switches, load balancers, reverse proxies, internet connections, etc)?
- Who owns the resources?
- Are there sufficient environments for development, testing, acceptance, pre-production, production, etc?

Structure

The main focus for this section is usually an infrastructure/network diagram showing the various hardware/network components and how they fit together, with a short narrative to accompany the diagram.



Example infrastructure diagrams, typically created in Microsoft Visio

If I'm working in a large organisation, there are usually infrastructure architects who look after the infrastructure architecture and create these diagrams for me. Sometimes this isn't the case though and I will draw them myself.

Motivation

The motivation for writing this section is to force me (the software architect) to step outside of my comfort zone and think about the infrastructure architecture. If I don't understand it, there's a chance that the software architecture I'm creating won't work or that the existing infrastructure won't support what I'm trying to do.

Audience

The audience for this section is predominantly the technical people in the software development team along with others that may help deploy, support and operate the software system.

Required

Yes, an infrastructure architecture section should be included in all software guidebooks because it illustrates that the infrastructure is understood and has been considered.

60. Deployment

The deployment section is simply the mapping between the software and the infrastructure.

Intent

This section is used to describe the mapping between the software (e.g. containers) and the infrastructure. Sometimes this will be a simple one-to-one mapping (e.g. deploy a web application to a single web server) and at other times it will be more complex (e.g. deploy a web application across a number of servers in a server farm). This section answers the following types of questions:

- How and where is the software installed and configured?
- Is it clear how the software will be deployed across the infrastructure elements described in the infrastructure architecture section? (e.g. one-to-one mapping, multiple containers per server, etc)
- If this is still to be decided, what are the options and have they been documented?
- Is it understood how memory and CPU will be partitioned between the processes running on a single piece of infrastructure?
- Are any containers and/or components running in an active-active, active-passive, hot-standby, cold-standby, etc formation?
- Has the deployment and rollback strategy been defined?
- What happens in the event of a software or infrastructure failure?
- Is it clear how data is replicated across sites?

Structure

There are a few ways to structure this section:

1. Tables: simple textual tables that show the mapping between software containers and/or components with the infrastructure they will be deployed on.

2. Diagrams: UML deployment diagrams or modified versions of the diagrams from the infrastructure architecture section showing where software will be running.

In both cases, I may use colour coding the designate the runtime status of software and infrastructure (e.g. active, passive, hot-standby, warm-standby, cold-standby, etc).

Motivation

The motivation for writing this section is to ensure that I understand how the software is going to work once it gets out of the development environment and also to document the often complex deployment of enterprise software systems.

This section can provide a useful overview, even for those teams that have adopted continuous delivery and have all of their deployment scripted using tools such as Puppet or Chef.

Audience

The audience for this section is predominantly the technical people in the software development team along with others that may help deploy, support and operate the software system.

Required

Yes, a deployment section should be included in all software guidebooks because it can help to solve the often mysterious question of where the software will be, or has been, deployed.

61. Operation and Support

The operations and support section allows you to describe how people will run, monitor and manage your software.

Intent

Most systems will be subject to support and operational requirements, particularly around how they are monitored, managed and administered. Including a dedicated section in the software guidebook lets you be explicit about how your software will or does support those requirements. This section should address the following types of questions:

- Is it clear how the software provides the ability for operation/support teams to monitor and manage the system?
- How is this achieved across all tiers of the architecture?
- How can operational staff diagnose problems?
- Where are errors and information logged? (e.g. log files, Windows Event Log, SMNP, JMX, WMI, custom diagnostics, etc)
- Do configuration changes require a restart?
- Are there any manual housekeeping tasks that need to be performed on a regular basis?
- Does old data need to be periodically archived?

Structure

This section is usually fairly narrative in nature, with a heading for each related set of information (e.g. monitoring, diagnostics, configuration, etc).

Motivation

I've undertaken audits of existing software systems in the past and we've had to spend time hunting for basic information such as log file locations. Times change and team members move on, so recording this information can help prevent those situations in the future where nobody understands how to operate the software.

Audience

The audience for this section is predominantly the technical people in the software development team along with others that may help deploy, support and operate the software system.

Required

Yes, an operations and support section should be included in all software guidebooks, unless you like throwing software into a black hole and hoping for the best!

62. Decision Log

The final thing you might consider including in a software guidebook is a log of the decisions that have been made during the development of the software system.

Intent

The purpose of this section is to simply record the major decisions that have been made, including both the technology choices (e.g. products, frameworks, etc) and the overall architecture (e.g. the structure of the software, architectural style, decomposition, patterns, etc). For example:

- Why did you choose technology or framework “X” over “Y” and “Z”?
- How did you do this? Product evaluation or proof of concept?
- Were you forced into making a decision about “X” based upon corporate policy or enterprise architecture strategies?
- Why did you choose the selected software architecture? What other options did you consider?
- How do you know that the solution satisfies the major non-functional requirements?
- etc

Structure

Again, keep it simple, with a short paragraph describing each decision that you want to record. Do refer to other resources such as proof of concepts, performance testing results or product evaluations if you have them.

Motivation

The motivation for recording the significant decisions is that this section can act as a point of reference in the future. All decisions are made given a specific context and usually have trade-offs. There is usually never a perfect solution to a given problem. Articulating the decision making process after the event is often complex, particularly if you're explaining the decision to people who are joining the team or you're in an environment where the context changes on a regular basis.

Although "nobody ever gets fired for buying IBM", perhaps writing down the fact that corporate policy forced you into using IBM WebSphere over Apache Tomcat will save you some tricky conversations in the future.

Audience

The audience for this section is predominantly the technical people in the software development team along with others that may help deploy, support and operate the software system.

Required

No, but I usually include this section if we (the team) spend more than a few minutes thinking about something significant such as a technology choice or an architectural style. If in doubt, spend a couple of minutes writing it down, especially if you work for a consulting organisation who is building a software system under an outsourcing agreement for a customer.

63. Questions

1. We should all strive for self-documenting code, but does this tell the whole story? If not, what is missing?
2. Do you document your software systems? If so, why? If not, why not?
3. If you have lots of project-level documentation but very little product-level documentation, how do new joiners to your team understand your software system(s)? What could make their job easier?
4. What would you consider to be a minimum level of supplementary software documentation within your own environment?
5. Where do you store your supplementary documentation? (e.g. source control system, network file share, SharePoint, wiki, etc). Is this the best solution given your intended audience?

bibliography

Brown, S. (2015). *Software Architecture for Developers.* Victoria, British Columbia: Leanpub.