LIVING DOCUMENTATION

A low-effort approach of Documentation, always up-to-date, inspired by Domain-Driven Design

Cyrille Martraire

Living Documentation by design, with Domain-Driven Design

Accelerate Delivery Through Continuous Investment on Knowledge

Cyrille Martraire

This book is for sale at http://leanpub.com/livingdocumentation

This version was published on 2019-06-14

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Cyrille Martraire by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Living Documentation: what if documentation was as fun as coding?
https://leanpub.com/livingdocumentation #LivingDocumentation

The suggested hashtag for this book is #LivingDocumentation.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#LivingDocumentation

Contents

Note to reviewers

Many thanks for reading this first version of my book!

Don't hesitate to share your feedback, even if on one single part or section.

I especially need feedback on:

- **What is interesting and needs to be elaborated in more details**
- What is not or less interesting
- **What you strongly disagree with, or you think is plain wrong**
- Suggestions of any additional examples
- Suggestions of any additional illustrations
- More generally, overall feedback on the content more than on form

Also if you have already put in practice some of the ideas of the book and want to be quoted about it, don't hesitate to tell me about it.

I know the current book has a lot of :

- Poorly written sentences (English is not my native language, and the text has not be edited yet)
- Typos (not fully spell-checked yet)
- Low-quality images or too large images

Please send all feedback or other comment through the Leanpub feedback form:

Leanpub Email the author[1]

(or at my personal email if you manage to get it :)

Thanks!

– Cyrille

[1] https://leanpub.com/livingdocumentation/email_author/new

A simple question

Are you happy with the documentation you create?

 yes / no

When you read documentation material, are you suspicious that it is probably a bit obsolete?

 yes / no

When you use external components, do you wish their documentation was better?

 yes / no

Do you believe that the time you spend doing documentation is time that could be better spent?

 yes / no

Preface

I never planed to write a book on living documentation. I didn't even have in mind that there was a topic under this name that was worth a book.

Long ago I had a grandiose dream of creating tools that could understand the design decisions we make when coding. I spent a lot of free time over several years trying to come up with a framework for that, only to find out it's very hard to make such a framework suitable for everyone. However I tried the ideas on every occasion, whenever it was helpful in the projects I was working in.

In 2013 I was speaking at Oredev on Refactoring Specifications. At the end of the talk I mentioned some of the ideas I'd been trying over time, and I had been surprised at the enthusiastic feedback I had around the living documentation ideas. This is when I recognized there was a need for better ways to do documentation. I've done this talk again since then and again the feedback was about the documentation thing and how to improve it, how to make it realtime, automated and without manual effort.

By the way, the word Living Documentation was introduced on the book Specifications by Example by Gojko Adzic, as one of the many benefits of Specifications by Example. Living Documentation is a good name for an idea which is not limited to specifications.

There was a topic, and I had many ideas to share about it. I wrote down a list of all these things I had tried, plus other stuff I had learnt around the topic. More ideas came from other people, people I know and people I only know from Twitter. As all that was growing I decided to make it into a book. Instead of offering a framework ready for use, I believe a book will be more useful to help you create quick and custom solutions to make your own Living Documentation.

About this book

> “Make very good documentation, without spending time outside of making a better software”

The book “Specification by Example” has introduced the idea of a “Living Documentation”, where an example of behavior used for documentation is promoted into an automated test. Whenever the test fails, it signals the documentation is no longer in sync with the code so you can just fix that quickly.

This has shown that it is possible to have useful documentation that doesn’t suffer the fate of getting obsolete once written. But we can go much further.

This book expands on this idea of a Living Documentation, a documentation that evolves at the same pace than the code, for many aspects of a project, from the business goals to the business domain knowledge, architecture and design, processes and deployment.

This book is kept short, with illustrations and concrete examples. You will learn how to start investing into documentation that is always up to date, at a minimal extra cost thanks to well-crafted artifacts and a reasonable amount of automation.

You don’t necessarily have to chose between Working Software and Extensive Documentation!

Acknowledgements

The ideas in this book originate from people I respect a lot. Dan North, Chris Matts, Liz Kheogh derived the practice called BDD, which is one of the best example of a Living Documentation at work. Eric Evans in his book Domain-Driven Design proposed many ideas that in turn inspired BDD. Gojko Adzic proposed the name "Living Documentation" in his book Specification by Example. This book elaborates on these ideas and generalizes them to other areas of a software project.

DDD has emphasized how the thinking evolves during the life of a project, and proposed to unify domain model and code. Similarly, this book suggests to unify project artifacts and documentation.

The patterns movement and its authors, starting with Ward Cunningham and Kent Beck, made it increasingly obvious that it is possible to do a better documentation by referring to patterns, already published or to author through PLoP conferences.

Pragmatic Programmers, Martin Fowler, Ade Oshyneye, Andreas Ruping, Simon Brown and many other authors distilled nugets of wisdom on how to do better documentation, in a better way. Rinat Abdulin first wrote on Living Diagrams, he coined the word as far as I know. Thanks to you all guys!

Eric Evans, thanks for all the discussions with you, usually not on this book, and for your advices.

I would also like to thank Brian Marick for sharing to me his own work on Visible Workings. As encouragements matter, discussions with Vaughn Vernon and Sandro Mancuso on writing a book did help me, so thanks guys!

Some discussions are more important than others, when they generate new ideas, lead to better understanding, or when they are just exciting. Thanks to George Dinwiddie, Paul Rayner, Jeremie Chassaing, Arnauld Loyer and Romeu Moura for all the exciting discussions and for sharing your own stories and experiments.

Through the writing of this book I've been looking for ideas and feedbacks as much as I could, and in particular during open space sessions at software development conferences. Maxime Saglan gave me the first encouraging feedback, along with Franziska Sauerwein, so thanks Franzi and Max! I want to thank all the participants of the sessions I ran on Living Documentation in these conferences and unconferences, for example in Agile France, Socrates Germany, Socrates France, Codefreeze Finland, and during the Meetup Software Craftsmanship Paris round tables and several Jams of Code at Arolla in the evening.

I've been giving talks at conference for some time now, but always around practices already widely accepted in our industry. With more novel content like Living Documentation I also had to test the acceptance from various audiences, and I thank the first conferences who took the risk to select the topic: NCrafts in Paris, Domain-Driven Design eXchange in London, Bdx.io in Bordeaux and ITAKE Bucharest for hosting the first versions of the talk or workshop. It is very helpful to have great feedback to spend more effort into the book.

I am very lucky at Arolla to have a community of passionate colleagues; thanks you all for your contributions and for being my very first audience, in particular Fabien Maury, Romeu Moura, Arnauld Loyer, Yvan Vu and Somkiane Vongnoukoun. Somkiane suggested to add stories to make the text "less boring" and it was one of the best ideas to improve the book.

Thanks to the coachs of the Craftsmanship center at SGCIB for all the lunch discussions and ideas, and their enthusiasm to get better in how we do software. In particular Gilles Philippart, mentioned several times in this book for his ideas, and Bruno Boucard, Thomas Pierrain. I must also thank Clémo Charnay and Alexandre Pavillon for early supporting some of the ideas as experiments in the SGCIB commodity trading department Information System, and Bruno Dupuis and James Kouthon for their help making it become real. Many of the ideas in this book have been tried in the previous companies I worked with: the Commodity department at SGCIB, the Asset Arena teams at Sungard Asset Management, all the folks at Swapstream and our colleagues at CME, and others.

Thanks to Café Loustic and all the great baristas there. This was the perfect place as an author, where I've written many chapters, usually powered by an Ethiopean single origin coffee from Cafènation.

Lastly, I want to thank my wife Yunshan who's always been supporting and encouraging throughout the book. You also made the book a more pleasant experience thanks to your cute little pictures! Chérie, your support was key, and I want to support your own projects the same way you did with this book.

How to read this book?

This book is on the topic of Living Documentation, and it is organized as a network of related patterns. Each pattern stands on its own, and can be read independently. However to fully understand and implement a pattern, there is usually the need to have a look at other related patterns, by reading their thumbnail at a minimum.

I'd like to make this book a *Duplex Book*, a book format suggested by Martin Fowler: The first part of the book is kept short and focuses on a narrative that is meant to be read cover-to-cover. In this form of book, the first part goes through all the content without diving too much into the details, while the rest of the book is the complete list of detailed patterns descriptions. You can read of course this second part upfront, or you may also keep it as a reference to go to whenever needed.

Unfortunately a Duplex book is hard to do at first try, and the book you are reading at the moment is not one yet. Feel free to skim, dig one area, and read it in any order, though I know readers who enjoyed reading it cover to cover.

Part 1 Reconsidering Documentation

A tale from the land of Living Documentation

Why this feature?

Imagine a software project to develop a new application as part of a bigger information system in your company. You are a developer in this project.

You have a task to add a new kind of discount to recent loyal customers. You meet Franck, from the marketing team, and Lisa, a professional tester. Together you start talking about the new feature, ask questions, and ask for concrete examples. At some point, Lisa asks "Why this feature?" Franck explains that the rationale is to reward recent loyal customers in order to increase the customer retention, in a Gamification approach, and suggests a link on Wikipedia about that topic. Lisa takes some notes, just notes of the main points and main scenarios.

All this goes quickly because everyone is around the table, so communication is easy. Also the concrete examples make it easier to understand and clarify what was unclear. Once it's all clear, everyone gets back to their desk. Lisa writes down the most important scenarios and sends them to everyone. It's Lisa doing it because last time it was Franck, and you do turns. Now you can start coding from that.

You remember your previous work experience where it was not like that. Teams were talking to each other through hard-to-read documents full of ambiguities. You smile. You quickly turn the first scenario into an automated acceptance test, watch it fail, and you start writing code to make it pass to Green.

You have the nice feeling to spend your valuable time on what matters and nothing else.

Tomorrow you won't need this sketch anymore

In the afternoon a pair of colleagues Georges and Esther ask the team about a design decision to make. You meet around the whiteboard and quickly evaluate each option while sketching. Not much of UML, some custom boxes and arrows, as long as everybody understands it right now. A few minutes later a solution is chosen: we'll use two different topics in the messaging system because we need "full isolation between the incoming orders and the shipment requests". That's the rationale for this decision.

Esther takes a picture of the whiteboard with her phone just in case someone would erase the whiteboard during the day. But she knows that in half a day it will be implemented, and she can

then safely delete the picture stored in her phone. One hour later, when she commits the creation of the new messaging topic, she takes care to add the rationale “isolation between incoming orders and shipment requests” in the commit comment.

The next day, Dragos, who was away yesterday, notices the new code and wonders why it’s like that. He does ‘git blame’ on the line and immediately gets the answer.

Sorry, we don’t have marketing documents!

The week after, a new marketing manager, Michelle, replaces Franck. Michelle is deep into customer retention, more than Franck. She wants to know what’s already implemented in the application in the area of customer retention, so she asks for the corresponding marketing document, and she is surprised to learn there is none.

“It’s not serious!”, she first says. But you quickly show her the website with all the acceptance tests that is produced during each build. There’s a search area on top so she can enter “customer retention”. She clicks submit, and what a pleasant surprise! The scenarios about the special discount for recent loyal customers appears in the result list! Michelle smiles. She didn’t even have to browse a marketing document to find what she wanted:

```
In order to increase customer retention
As a marketing person
I want to offer a discount to recent loyal customers

Scenario: 10$ off on next purchase for recent loyal customer
...

Scenario: Recent loyal customers have bought 3 times in the last week
...
```

“Could we do the same discount for purchases in euro?” she asks. “I’m not sure the code manages currencies well, but let’s just try” you reply. In your IDE, you change the currency in the acceptance test, and you run the tests again. They fail, so you know there is something to do to support that. Michelle has her answer within minutes. She begins to think that your team has something special compared to her former work environments.

You keep using this word, but this is not what it means

The next day Michelle has another question: what is the difference between a ‘purchase’ and an ‘order’?

Usually she would just ask the developers to look in the code and explain the difference. However this team has anticipated that and the website of the project displays a glossary. “Is this glossary

up-to-date?" she asks. "Yes, it's updated during every build, automatically from the code," you reply. She's surprised. Why doesn't everybody do that? "You need to have your code closely in line with the business domain for that, " you say, while you're tempted to elaborate on the Ubiquitous Language of DDD.

Looking at the glossary she discovers a confusion that nobody has spotted before in the naming, and she suggests to fix the glossary with the correct name. But this is not the way it works here. You want to fix the name first and foremost in the code. So you rename the class and run the build again, and voila, the glossary is now fixed as well. Everybody is happy, and you just learnt something new about the business of e-commerce.

Show me the big picture, and you'll see what's wrong there

Now you'd like to remove a toxic dependency between two modules, but you're not so familiar with the full codebase, so you ask for a dependency diagram to Esther, who has the most knowledge of that in the team. But even her does not remember every dependency. "I'll generate a diagram of the dependencies from the code. It's something I've long wanted to do. This will take me a few hours, but then it'll be done forever", she says.

Esther already knows a few open-source libraries to easily extract the dependencies from a class or a package, so it's rather quick for her to wire that to Graphviz, the magical diagram generator that does the layout automatically. A few hours later, her little tool generates the diagram of dependencies. You get what you wanted and you're happy. She then spends one extra half-hour to integrate this tool into the build.

But the funny thing is that when Esther first looks at the generated diagram she immediately notices something intriguing: "I didn't know there was this dependency between these two modules, it should not be there". By comparing her mental view of the system with the generated view of the actual system, it was easy to spot the design weakness.

In the next project iteration, the design weakness is fixed, and in the next build, the dependency diagram is automatically updated. It is now a cleaner diagram, and it shows visually.

The future of Living Documentation is now

This tale is not about the future. It is already there, right now, and it has been there for years already. This "future has already happened, it's just not very evenly distributed" yet, to quote William Gibson.

The tools are there. The techniques are there. People have been doing all that for ages, and it's just not mainstream yet. It's a pity because these are powerful ideas for software development teams.

In the next chapters, we'll go through all these approaches, and many other, and you'll learn how to implement them in your projects.

The problem with traditional documentation

> Documentation is the castor oil of programming —managers think it is good for programmers, and programmers hate it!
>
> Gerald Weinberg in *Psychology of Computer Programming*

Documentation is such a boring topic. I don't know about you, but in my work experience so far documentation has mostly been a great source of frustration.

When I'm trying to consume documentation, the one I need is always missing. When it's there is often obsolete and misleading, so I can't even trust it.

When I'm trying to create documentation for other people, then it's a boring task and I'd prefer to be coding instead.

But it does not have to be this way.

There has been a number of times when I've seen, used, heard about better ways to deal with documentation. I've tried a lot of them. I've collected a number of stories, that you'll find in this book.

There's a better way, if we adopt a new mindset about documentation. With this mindset and the techniques that go with it, we can make, indeed, documentation as fun as coding.

Documentation is not cool

What comes to mind when you hear the word *documentation*?

- It's boring.
- It's about writing lots of text.
- It's about trying to use Microsoft Word without losing your sanity with picture placement.
- As a developer I love dynamic, executable stuff that exhibits motion and behavior. In contrast, documentation is like a dead plant, it's static and dry.
- It's supposed to be helpful but it's often misleading.

Documentation is a boring chore. I'd prefer be writing code instead of doing documentation!

Oh no... I'd better be coding!

There's something wrong with Documentation. It takes a lot of time to write and to maintain, is obsolete quickly, is incomplete at best, and is just not fun. Documentation is a fantastic source of frustration.

So documentation sucks. Big time. And I'm sorry to bring you on this journey on such a crappy topic.

The flaws of documentation

> "Like cheap wine, paper documentation ages rapidly and leaves you with a bad headache."
> - @gojkoadzic on Twitter

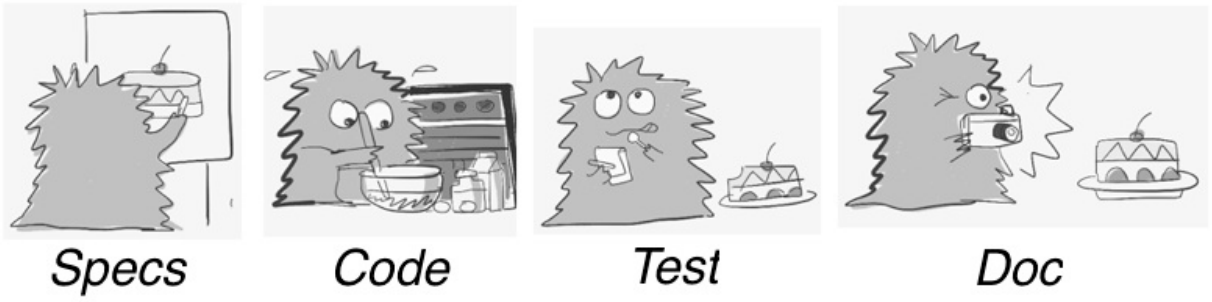
Traditional documentation suffers from many flaws and several common anti-patterns.

An anti-pattern is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive. From Wikipedia[2]

Some of the most frequent flaws and anti-patterns of documentation are described below. Do you recognize some of them in your own projects?

Separate Activities

Even in software development projects which claim to be agile, deciding what to build, doing the coding, testing and preparing documentation are too often **Separate Activities**.



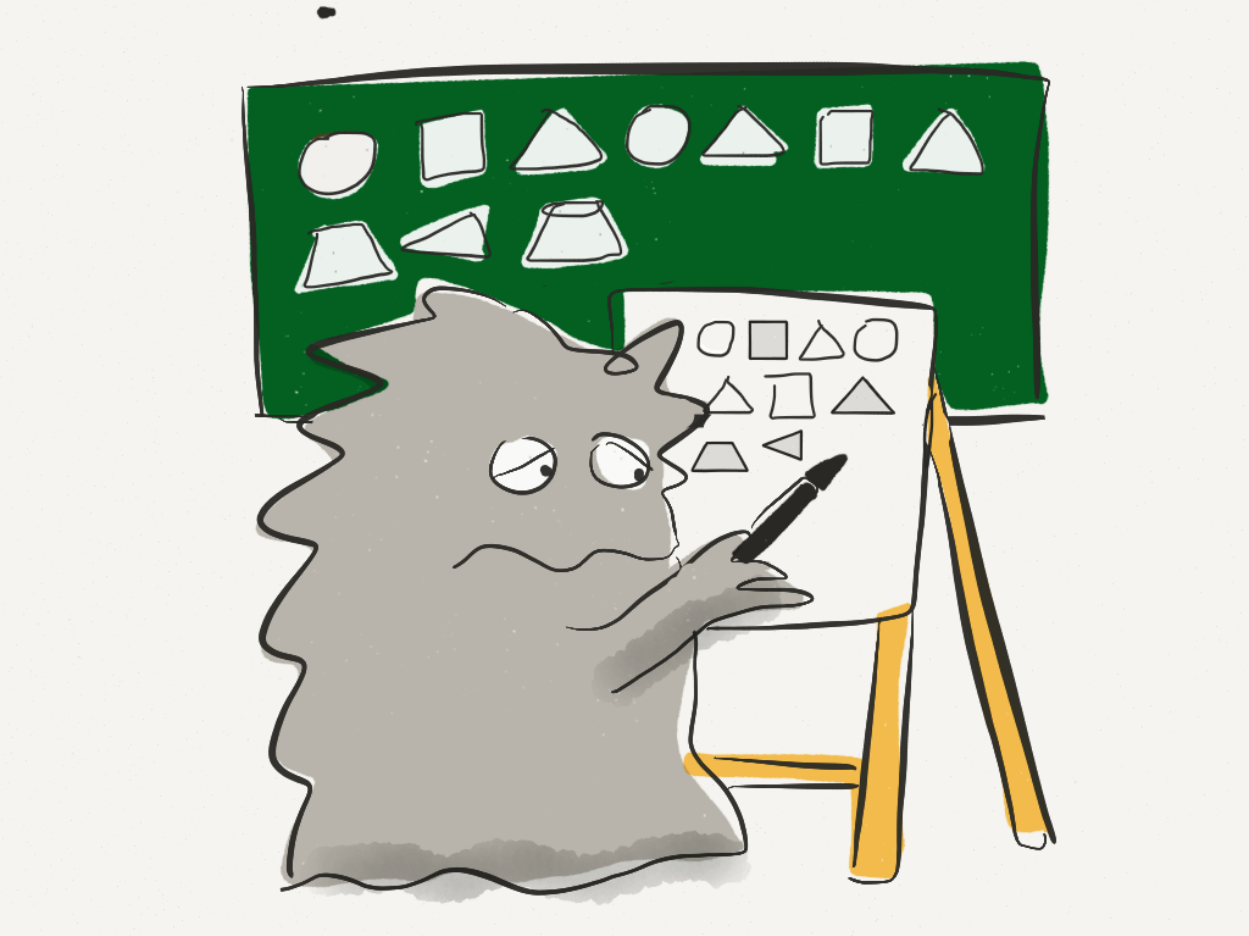
Separate Activities

Separate activities induce a lot of waste and lost opportunities. Basically the *same* knowledge is manipulated during each activity, but in different forms and in different artifacts, probably with some amount of duplication. And this "same" knowledge can evolve during the process itself, which may cause inconsistencies.

Manual Transcription

When comes the time to do documentation, members of the team select some elements of knowledge of what has been done and perform a **Manual Transcription** into a format suitable for the expected audience. Basically, it's about taking the knowledge of what has just been done in the code to write it in another document.

[2] http://en.wikipedia.org/wiki/Anti-pattern

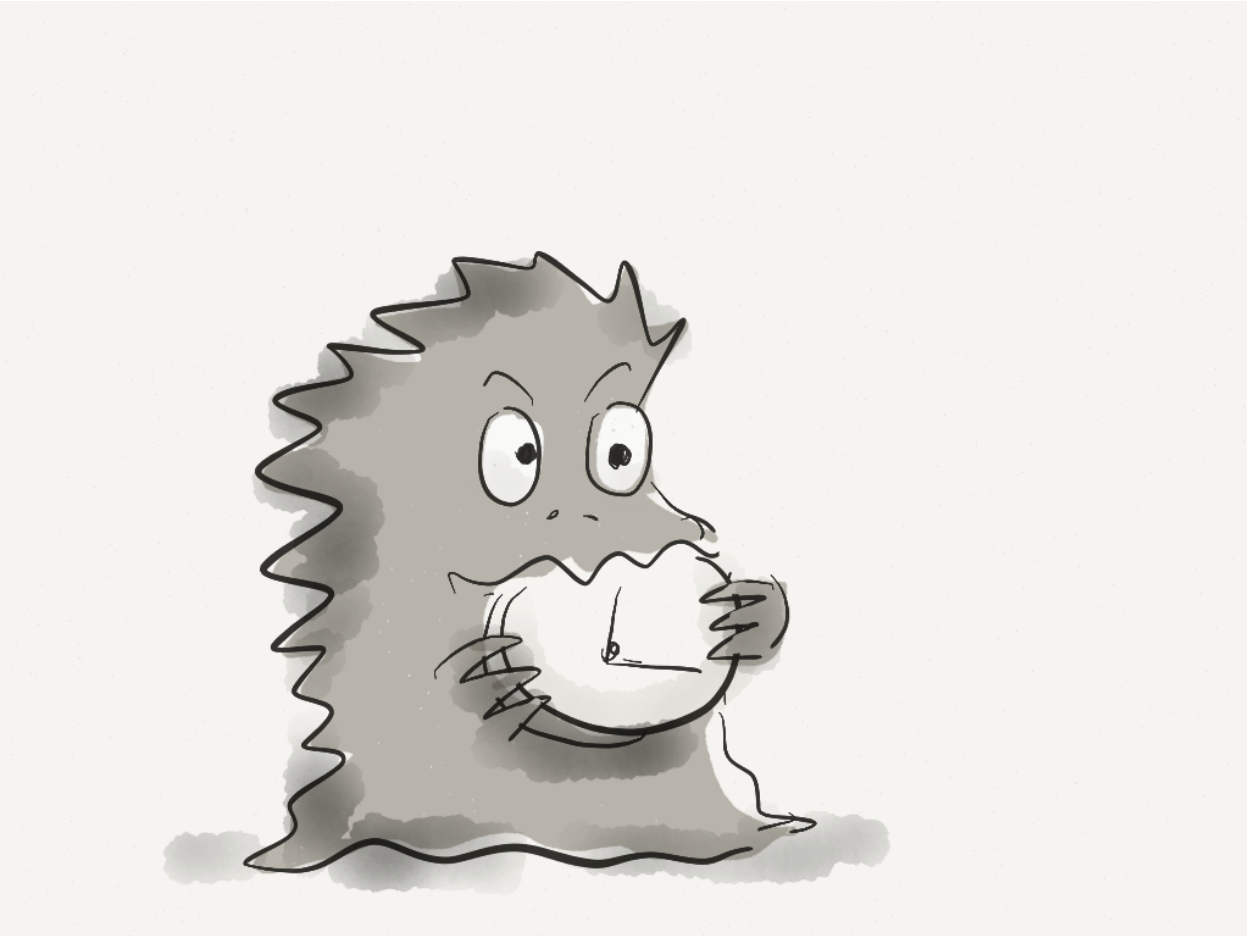


Manual Translation

Redundant Knowledge

This transcription leads to **Redundant Knowledge**: there is the original source of truth, usually the code, and all the copies that duplicate this knowledge in various forms. Unfortunately, when one artifact changes, for example the code, it is hard to remember to update the other documents. As a result the documentation quickly becomes obsolete, and you end up with an incomplete documentation that you cannot trust. How useful is that documentation?

Boring Time Sink



Documentation is often a time sink

Managers want documentation for the users and to cope with the turnover in the team, so they ask for documentation. However developers hate writing documentation. It is not fun compared to writing code or compared to automating a task. Dead text that get obsolete quickly and that does not execute is not particularly exciting to write for a developer. When developers are working on documentation, they'd prefer to be working on the real working software instead.

However when they want to reuse third-party software, they often wish it had more documentation available.

Technical writers like to do documentation and are paid for that. However they usually need developers to have access to the technical knowledge, and then they're still doing manual transcription of knowledge.

Brain Dump

A brain dump is not necessarily useful as documentation

Because writing documentation is not fun and is done "because we have to", it is often done arbitrarily, without much thinking. The result is a random **brain dump** of what the writer had in mind at the time of writing. The problem is that there is no reason for this random brain dump to be any helpful for someone.

Polished Diagrams

This anti-pattern is common with people who like to use CASE tools. These tools are not meant for sketching. Instead they encourage the creation of polished and large diagrams, with various layouts and validation against a modeling referential etc. All this takes a lot of time. Even with all the auto-magical layout features of these tools, it still takes too much time to create even a simple diagram.

Notation Obsession

It is now increasingly obvious that UML is not really popular, however in the decade since 1999 it was the *Universal* notation for everything software, despite not being suited for all situations. This means that no other notation has been popularized during this time. This also means that many people did use UML to document stuff, even when it was not well-suited for that. When all you know is UML, everything looks like one of its collection of standard diagrams, even when it's not.

No Notation

In fact, the opposite of notation obsession was rather popular. Even with the dominant UML, many simply ignored it, drawing diagrams with custom notations that nobody understands the same way, or mixing random concerns like build dependencies, data flow and deployment concerns together in a happy mess.

Information Graveyard

Enterprise knowledge management solutions are the places where knowledge goes to die:

- Enterprise wiki, SharePoint
- Large office documentation systems
- Shared folders
- Ticketing systems and wikis with poor search capabilities

These approaches to documentations too often fail either because it's too hard to find the right information, or because it's too much work to keep the information up-to-date, or both. It's a form of *Write-Only documentation*, or *Write-Once documentation*.

On a recent Twitter exchange with James R. Holmes, Tim Ottinger asked:

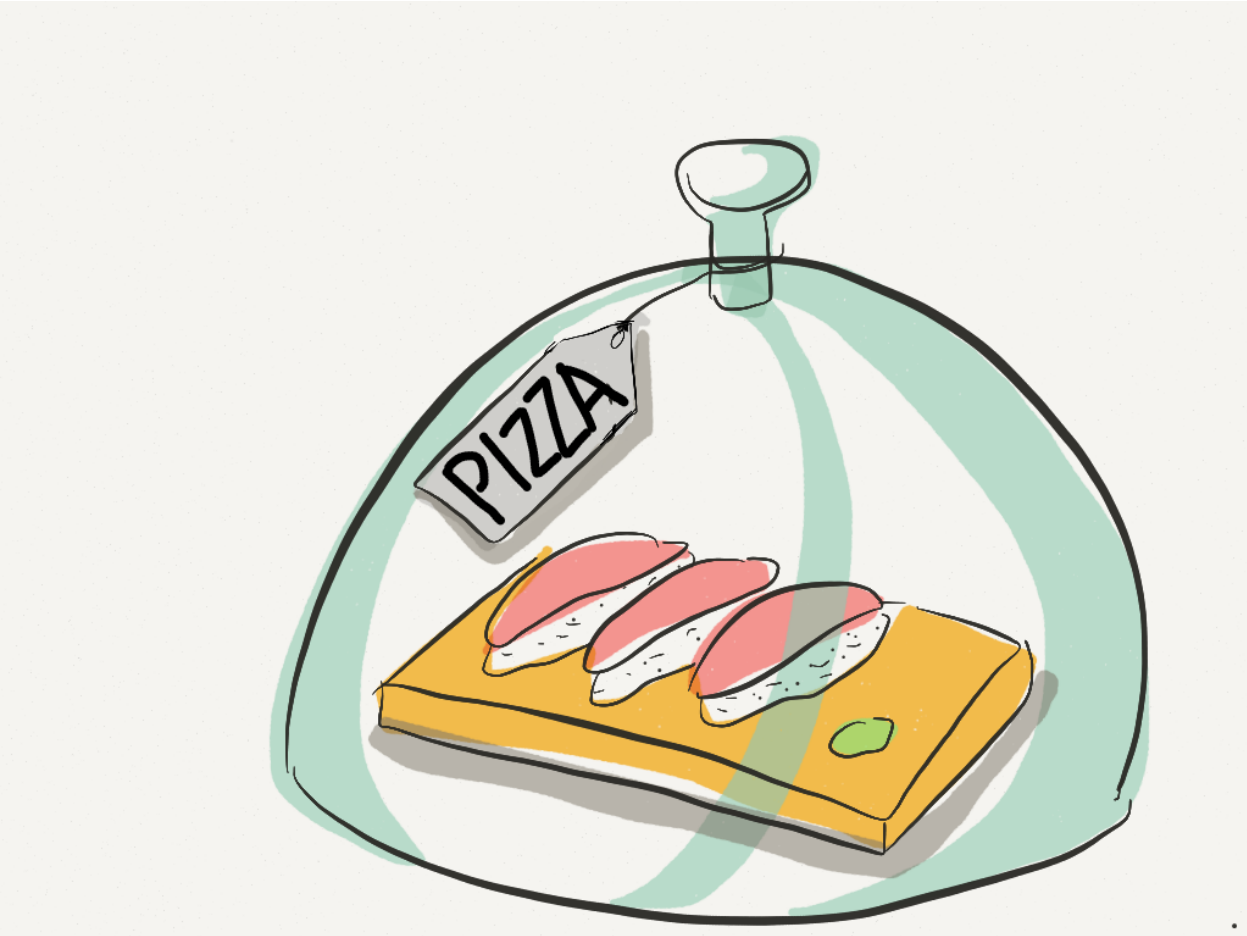
> Product category: "Document Graveyard" – are all document management & wiki & SharePoint & team spaces doomed?

Holmes replied:

> Our standard joke is that "It's on the intranet" leads to the response, "Did you just tell me to go screw myself?"

It's not because it's documented that it's useful.

Misleading Help



Documentation can be toxic when misleading

Whenever documentation is not strictly kept up-to-date, it becomes misleading. It pretends to help, but it is wrong. As a result, it may still be interesting to read it, but there's an additional cognitive load trying to find out what's still right Vs. what's become wrong by now.

There's always something more important right now

Good documentation needs a lot of time to be written and even more so to be maintained. When you are under time pressure, documentation tasks are often skipped or done quickly and badly.

The agile manifesto and documentation

In 2001 the Agile Manifesto was written. It says:

> Seventeen anarchists agree: We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Follows a list of preferences, expressed as "we value the things on the left and on the right, but we value the things on the left more". Here are these 4 preferences:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

The statement "Working software over comprehensive documentation" is frequently misunderstood. Many people believe that it disregards documentation completely. In fact the Agile Manifesto does not say "don't do documentation". It's only a matter of preference. In the words of the authors of the manifesto: "We embrace documentation, but not to waste reams of paper in never-maintained and rarely-used tomes." Martin Fowler and Jim Highsmith[3]

Still, with agile approaches becoming mainstream in larger corporations, the misunderstanding is still there and many people neglect documentation.

However I've noticed recently that the lack of documentation is a big source of frustration for our customers and colleagues,and this frustration is getting bigger. I was also surprised to see some great appetite for the topic of documentation after I first mentioned living documentation at the Oredev conference in Sweden.

It's time for Documentation 2.0

Traditional documentation is flawed, but now we know better. Since the end of the 90's practices like Clean Code, Test-Driven Development (TDD), Behavior-Driven Development (BDD), Domain-Driven Design (DDD) and Continuous Delivery have become increasingly popular. All these practices have changed the way we think about delivering software.

With TDD the tests are first considered as specifications. With DDD, we identify the code and the modeling of the business domain, breaking with the tradition of models kept separately from the code. One consequence is that we expect from the code to tell the whole story about the domain. BDD borrowed the idea of the business language and made it more literal, with tool support. Finally Continuous Delivery is showing that an idea that looked ridiculous a few years ago (delivering several times a day in a non-event fashion) is actually possible and even desirable if we decide to follow the recommended practices. <examples>

Another interesting thing happening is the effect of time: old ideas like literate programing or HyperCard did not become mainstream, but just like FP languages they are increasingly influential,

[3] http://agilemanifesto.org/history.html

and more recent programming languages like F# or Clojure bring some of the old ideas to the foreground.

All that background means that now at last we can expect an approach to documentation that is useful and always up-to-date, at a low cost. And fun to create.

We acknowledge all the problems of the traditional approach to documentation, yet we also acknowledge that there is a need to be fulfilled. This book explores and offers guidances on other approaches to meet the needs in more efficients ways.

But first let's explore what documentation really is.

It’s all about knowledge

It’a all about knowledge. Software development is all about knowledge, and decision-making based on it, which in turn becomes additional knowledge. The given problem, the decision that was made, the reason why it was made that way, and the facts that led to this decision, and the considered alternative are all knowledge.

You may not think about it that way, but each instruction typed in a programming language is a decision. There are big and small decisions, but it’s just decisions taken. In software development, there is no expensive construction phase following a design phase: the construction is so cheap (running the compiler) that there’s only an expensive, sometime everlasting, design phase.

This design activity can last for a long time. It can last long enough to forget about previous decisions made, and their context. It can last long enough for people to leave, with their knowledge, and for new people to join, with missing knowledge. Knowledge is central to a design activity like software development.

This design activity is also most of the time, and for many good reasons, a teamwork, with more than one person involved. Working together means taking decisions together or taking decisions based on someone else’s knowledge.

Something unique with software development is that the design involves not only people but also machines. Computers are part of the picture, and many of the decisions taken are simply told to the computer to execute. It’s usually done through documents that are called “source code”. Using a formal language like a programming language, we pass knowledge and decisions to the computer in a form it can understand.

Having the computer understand the source code is not the hard part though. Even unexperienced developers usually manage to succeed at that. The hardest part is for other people to understand what has been done, in order to do a better and faster work.

The larger the ambition, the more documentation becomes necessary to enable a cumulative process of knowledge management that scales beyond what fits in our heads. When our brains and memories are not enough, we need assistance from technologies like writing, printing, and software to help remember and organize larger sets of knowledge.

Knowledge origination

Where does knowledge come from?

Knowledge primarily comes from **conversations**. We develop a lot of knowledge through conversations with other people. This happen during collective work like pair programming, or during meetings, or at the coffee machine, on the phone, or via a company chat or emails.

Examples: BDD specification workshops, 3 amigos, concrete examples

However as software developers we have conversations with machines too, which we call **experiments**. We tell something to the machine in the form of code in some programming language, and the machine runs it and tells us something in return: the test fails or goes green, the UI reacts as expected, or the result is not what we wanted, in which case we'll learn something new.

Examples: TDD, Emerging Design, Lean Startup experiments

Knowledge also comes from observation of the context. In a company you learn a lot by just being there, listening to other people conversations, behavior and emotions.

Examples: Domain Immersion, Obsession Walls, Information Radiators, Lean Startup "Get out of the building"

> ## Conversations, Experiments, Context
>
> Knowledge comes from conversations with people and experiments with machines in an observable context.

How does that knowledge evolve?

Some knowledge is stable in years, whereas other is living all the time, over months or even over hours.

This means that whatever we do about documentation has to consider the cost of maintenance, and to make it as close to zero as possible. For stable knowledge everything's simple, and traditional methods of documentation work. But in most cases the knowledge is changing frequently enough that writing text and updating it on every change is just not an option.

The effect of acceleration in the software industry means that we want to be in a position to evolve the software so quickly that it's obviously impossible to spend time writing pages and pages of documentation. And yet we want all the benefits of documentation.

The reasons why knowledge is necessary

When creating software, we go through a lot of questions, decisions and adjustments as we learn:

- What problem are we trying to solve?
- Oh no, what problem are we really trying to solve? (got it wrong the previous time)
- Trade and Execution are not synonyms, we made the confusion so far
- We tried this new DB and it didn't match our needs for these 3 reasons
- We decided to decouple the shopping cart and the payment because we noticed that the changes to one had nothing to do with changes to the other, please don't couple them again
- We found out that this feature is useless, we'll delete the code next month (note: we'll forget and this code will become a mystery from now on)
- We found out late that we had to comply to this official process for our change management and our delivery process (we didn't know before)

On existing software, when missing the knowledge developed before, we end up:

- Redoing what's already done, because we don't know it's there
- Putting a feature in an unrelated component, for lack of knowing where it should be, making it bloated, while the code about the feature is now fragmented across various components

If only we had the knowledge available to answer everyday questions like the ones listed below!

- Where shall I fix that issue safely?
- Where shall I add this enhancement?
- Where would the original authors add this enhancement?
- I don't know if I can delete this line of code that looks useless
- I'd like to change a signature but I don't know where the impacts will be
- I need to reverse engineer the code just to understand a bit how its works
- The Business Analysts keep on asking us to tell them the business rules by checking directly the code
- A customer asks for a feature but we don't even know if it' already supported of if it requires new developments
- We have the feeling that the way we evolve the code is the best possible, but we lack a complete understanding of how it works

- We always have to look everywhere in the code to find where's the part that deals with a particular feature

The cost of lack of knowledge mainly manifests itself in the form of:

- **Waste of time** (time that could be better invested in improving something else)
- **Sub-optimal decisions** (decisions that could have been more relevant, i.e. cheaper in the long term)

These two expenses compound for the worst over time: the time spent on finding the missing knowledge is time not spent on taking better decisions. In turn, sub-optimal decisions will compound to make our life progressively more miserable, until we have no choice but to decide that the software is no longer maintainable, and start again.

It sounds like a good idea to be able to access the knowledge that is useful to perform the development tasks.

And now for a pedantic word: Stigmergy

Michael Feather recently shared a link to a fantastic article online[4] by Ted Lewis who introduces the concept of stygmergy in relation to our work in software as a team:

> Stigmergy is a mechanism of indirect coordination, through the environment, between agents or actions. The principle is that the trace left in the environment by an action stimulates the performance of a next action, by the same or a different agent. Stigmergy on Wikipedia[5]
>
> The French entomologist Pierre-Paul Grassé described a mechanism of insect coordination he called "stigmergy" —work performed by an actor stimulates subsequent work by the same or other actors. That is, the state of a building, code base, highway, or other physical construction determines what needs to be done next without central planning or autocratic rule. The actors —insects or programmers- know what to do next, based on what has already been done. This intuitive urge to extend the work of others becomes the organizing principle of modern software development.
>
> Ants use a special type of chemical marker —pheromones— to highlight the results of their activity. (…)

Similarly, programmers manufacture their own markers through emails, Github issues and all kinds of documentation that augments the code itself. As Ted concludes:

[4] http://ubiquity.acm.org/blog/why-cant-programmers-be-more-like-ants-or-a-lesson-in-stigmergy
[5] https://en.wikipedia.org/wiki/Stigmergy

The essence of modern software development is stigmergic intellect and markers embedded within the code base. Markers make stigmergy more efficient, by more reliably focusing a programmer's attention on the most relevant aspects of the work that needs to be done.

Software Programming as Theory Building and Passing

Long ago, in 1985, Peter Naur's [famous paper *Programming As Theory Building *](http://www.dc.uba.ar/materias/p *perfectly revealed the truth about programming as a collective endeavor: it's not so much about telling the computer what to do, but its more about sharing with other developers the *theory of the world* (think "mental model") that has been patiently elaborated by learning, experiment, conversations and deep reflections.

In the words of Peter Naur:

> Programming properly should be regarded as an activity by which the programmers form or achieve a certain kind of insight, a theory, of the matters at hand. This suggestion is in contrast to what appears to be a more common notion, that programming should be regarded as a production of a program and certain other texts.

The problem is that most of the theory is *tacit*. The code only represents the tip of iceberg. It's more a consequence of the theory in the mind of the developers than a representation of the theory itself.

In Peter Naur's view, this theory encompasses three main areas of knowledge, the first being the **mapping** between code and the world it represents:

> 1/ The programmer having the theory of the program can explain how the solution relates to the affairs of the world that it helps to handle.

The second is about the **rationale** of the program:

> 2/ The programmer having the theory of the program can explain why each part of the program is what it is, in other words is able to support the actual program text with a justification of some sort.

And the third is about the **potential** of extension or evolution of the program:

> 3/ The programmer having the theory of the program is able to respond constructively to any demand for a modification of the program so as to support the affairs of the world in a new manner.

Over time we've learnt a number of techniques to help passing theories between people in a durable way. Clean Code and Eric Evans' Domain-Driven Design encourage to find ways of expressing more of the theory in your head literally into the code. For example DDD's Ubiquitous Language bridges the gap between the language of the world and the language of the code, helping solve the *mapping* problem. I hope future programming languages will recognize the need to represent not only the behavior of the code but also the bigger mental model of the programmers, of which the code is a consequence.

Patterns and patterns languages also come to mind, as literal attempts to package nuggets of theories. The more patterns we know, the more we can encode the tacit theory, making it explicit and transferable to a wider extent. Patterns embody in the description of their forces the key elements of the *rationale* in choosing them, and they sometime hint at how extension should happen, i.e. hinting at the *potential* of the program: for example a Strategy pattern is meant to be extended by adding new strategies.

But as we progress in the codification of our understanding, we also tackle more ambitious challenges, so our frustration remains the same. I believe his sentence from 1985 will still hold in the next decades:

> For a new programmer to come to possess an existing theory of a program it is insufficient that he or she has the opportunity to become familiar with the program text and other documentation.

We'll never completely solve that knowledge transfer problem, but we can accept it as a fact and learn to live with it. The theory as a mental model in programmers' head can never be fully shared if you weren't part of the thought process that led to build it.

> The conclusion seems inescapable that at least with certain kinds of large programs, the continued adaption, modification, and correction of errors in them, is essentially dependent on a certain kind of knowledge possessed by a group of programmers who are closely and continuously connected with them.

It's worth nothing that permanent teams who regularly work collectively don't suffer that much from this issue of theory-passing.

Documentation is about transferring knowledge

The word "documentation" often brings a lot of connotations to mind: written documents, MS Word or Powerpoint documents, documents based on company templates, printed documents, big heavy and boring text on a website or on a wiki, etc. However all these connotations anchor us to practices of the past, and they exclude a lot of newer and more efficient practices.

For the purpose of this book, we'll adopt a much broader definition of documentation:

> Documentation is about transferring valuable knowledge. Transferring valuable knowledge to other people now. Transferring valuable knowledge to people in the future.

There's a logistic aspect to it. It's about transferring knowledge in space between people, and to transfer it over time, which we call persistence or storage. Overall, our definition of documentation looks like shipment and warehousing of goods, where the goods are knowledge.

Transferring knowledge between people is actually transferring knowledge between one brain to other brains.

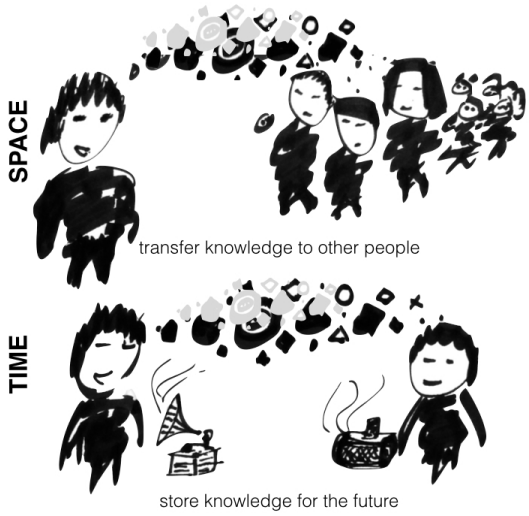
From one brain to other brains, it's a matter of **transmission**, or diffusion, for example to reach a larger audience.

From brains now to brains later, it's about persisting the knowledge and it's a matter of **memory**.

> The development tenure half-life is 3.1 years, whereas the code half-life is 13 years Rob Smallshire in his blog[a]
>
> [a] http://sixty-north.com/blog/predictive-models-of-development-teams-and-the-systems-they-build

From the brain of a technical person to the brains of non technical people, it's a matter of **making the knowledge accessible**. Another case of making knowledge accessible is to make it efficiently **searchable**.

And there are other situations like putting knowledge into a specific format of document for **compliance** reasons, because you just have to.



Documentation is about transferring and storing knowledge

Choosing the right media

Under the definition of documentation as a transfer of valuable knowledge, documentation can take many forms: written documents, face-to-face conversations, code, activity on social tools, or nothing at all when it's not necessary.

With this definition of documentation we can express some important principles.

- Knowledge that is of interest for a **long period of time** deserves to be documented
- Knowledge that is of interest to a **large number of people** deserves to be documented
- Knowledge that is **valuable** or **critical** may also need to be documented

And on the other hand, you probably don't need to care about documentation of knowledge that isn't in any of these cases. Spending time or effort on it would be just waste.

The *value* of the considered knowledge matters. There's no need to make the effort to transfer knowledge that's not valuable enough for enough people over a long-enough period of time. If a piece of knowledge is already well-known or is only useful for one person, or if it's only of interest till the end of the day, then there's probably no need to transfer or store it.

> Default is Don't
>
> There is no point in doing any specific effort documenting knowledge unless there's a compelling reason to do it, otherwise it's waste. Don't feel bad about it.

Specific vs. Generic Knowledge

There is knowledge that is specific to your company, your particular system or your business domain, and there is knowledge that is generic and shared with many other people in many other companies in the industry.

Generic Knowledge

Knowledge about programming languages, developers tools, software patterns and practices belongs to the Generic knowledge category. Examples include: DDD, patterns, CI, using Puppet, Git tutorial etc.

Knowledge about mature sectors of the business industries is also generic knowledge. Even in very competitive areas like Pricing in finance or Supply Chain Optimization in e-commerce, most of the knowledge is public and available in industry-standard books, and only a small part of the business is specific and confidential for a while.

For example each business domain has its essential reading lists, with books often referred to as "The Bible of the field": Options, Futures, and Other Derivatives (9th Edition) by John C Hull, Logistics and Supply Chain Management (4th Edition) by Martin Christopher etc.

The good news is that generic knowledge is already documented in the industry literature. There are books, blog posts, conference talks that describe it quite well. There are standard vocabularies to talk about it. There are trainings available to learn it faster with knowledgable people.

Learn generic knowledge

You also learn generic knowledge by doing your job of course, but mostly by reading books and attending trainings and conferences. This only takes a few hours, and you know beforehand what you're going to learn, how long it will take and how much it will cost. It's almost as easy to learn generic knowledge as going to the store to buy food.

Generic knowledge is basically a solved problem. This knowledge is ready-made, ready to be reused by everyone. When you use it, you just have to link to an authoritative source and you're done documenting. This is as simple as putting an Internet link or a bibliographic reference.

Focus on Specific Knowledge

Specific knowledge is the one your company and team has that is not (yet) shared with other peers in the same industry. This knowledge is more expensive to learn, it takes practicing, making mistakes

and failures to earn it. That's the kind of knowledge that deserves most attention, because only you can take care about it. It's the specific knowledge that deserves the biggest efforts from you and your colleagues. As a professional, you should know enough of the generic, industry standard knowledge, in order to be able to focus on growing the knowledge that's specific to your particular ambitions.

Specific knowledge is valuable, and cannot be found ready-made, so it's the kind of knowledge you'll have to take care of.

Knowledge is already there

Every interesting project is a learning journey to some extent, producing specific knowledge. We usually expect documentation to give us the specific knowledge we need, however the funny thing is that all this knowledge is already there: in the source code, in the configuration files, in the tests, in the behavior of the application at runtime, in memory of the various tools involved, and of course in the brain of all the people working on it.

Knowledge is already there

> In a software project most of the knowledge is present in some form somewhere in the artifacts.

The knowledge is there somewhere, but this does not mean that there is nothing to do about it. There are a number of problems with the knowledge that's already there.

Not Accessible: The knowledge stored in the source code and other artifacts is not accessible to non technical people. For example, source code is not readable by non developers.

Too Abundant: All the knowledge stored in the project artifacts is in huge amounts, which makes it not usable efficiently. For example, each logical line of code encodes knowledge, but for a given question, only one or two lines may be relevant to give the answer.

Fragmented: There is knowledge that we think of as one single piece but that is in fact spread over multiple places in the projects artifacts. For example, a class hierarchy in Java is usually spread over multiple files, one for each subclass, even if we think about the class hierarchy as a whole.

Implicit: A lot of knowledge is present but implicitly in the existing artifacts. It's 99% there, but is missing the one more 1% to make it explicit. For example when you use a design pattern like a Composite, the pattern is visible in the code, but only if you're familiar with the pattern.

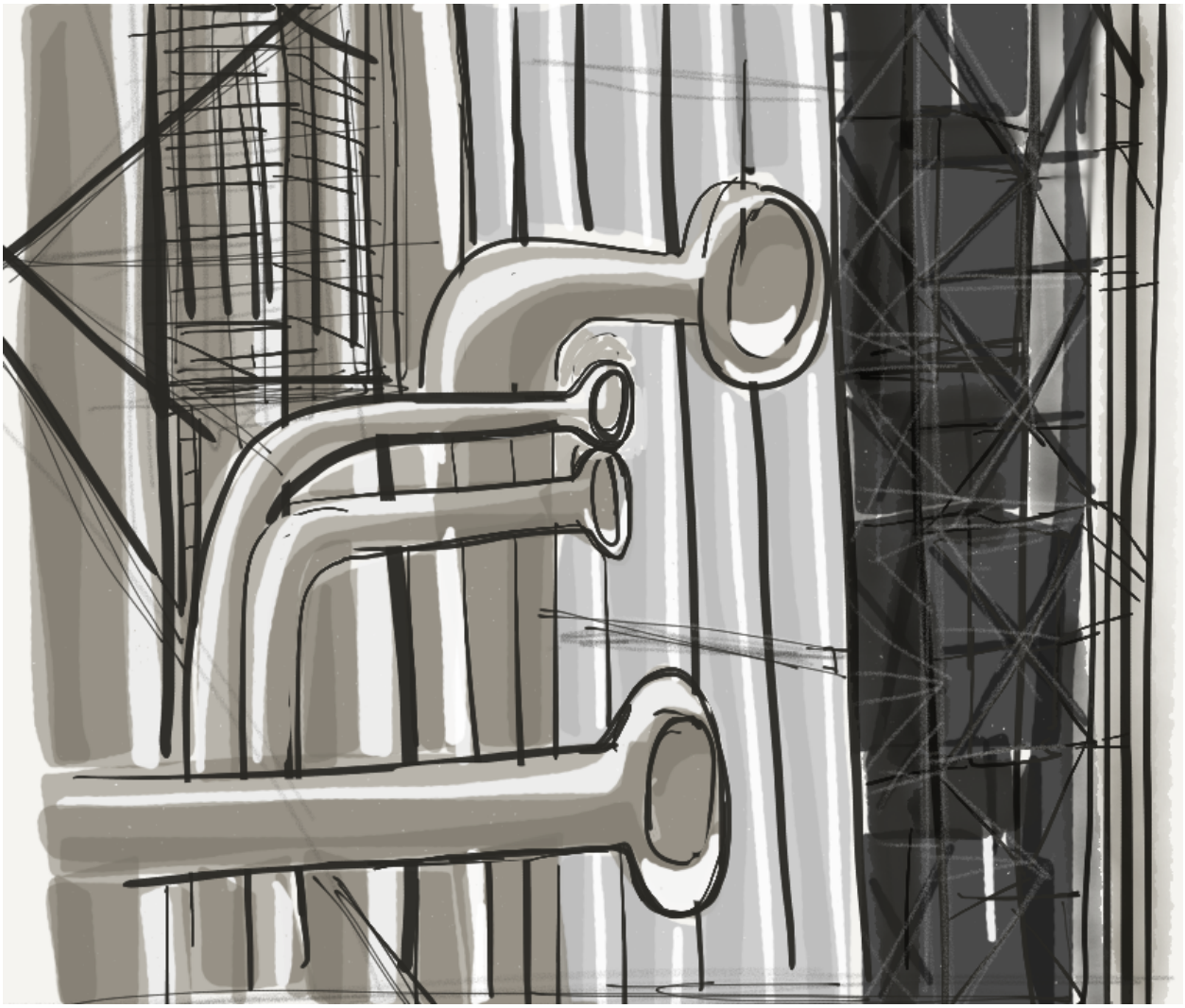
Unrecoverable: It happens that the knowledge is there but there is no way to recover it because it's excessively obfuscated. For example business logic is expressed in code but the code is so bad that nobody can understand it.

Unwritten: In the worst case, the knowledge is only in people's brain, and only its consequences are there in the system. For example, there is a general business rule but it has been programmed as a series of special cases, so the general rule is not expressed anywhere.

Internal Documentation

The best place to store documentation is on the documented thing itself

You've probably seen the pictures of the Google datacenters and of the Centre Pompidou in Paris. They both have in common a lot of color-coded pipes, with additional labels printed or riveted on the pipes themselves. On the Pompidou Center, air pipes are blue, water pipes are green. This logic of color-coding expands beyond the pipes: electricity transport is yellow, and everything about moving people is red, like the elevators and stairways in the outside.



The Centre Pompidou building is color-coded

This logic is also ubiquitous in datacenters, with even more documentation printed directly on the pipes. There are arrows to show the direction of the water flow, and labels to identify them. In the real world, such color-coding and ad hoc marking is often mandatory for fire prevention and fire fighting: water pipes for firefighters have very visible labels riveted on them indicating where they come from. Emergency exits in buildings are made very visible above the doors. In airplanes, fluorescent signs on the central corridors document where to go. In a situation of crisis, you don't have time to look for a separate manual, you need the answer in the most obvious place: right where you are, on the thing itself.

Internal vs. External documentation

Persistent documentation comes in two flavors: external or internal.

External documentation is when the knowledge is expressed in a form that has nothing to do with the chosen implementation technologies of the project. This is the case of the traditional forms of documentation, with separate MS Office documents on shared folders, or wikis with their own database.

The advantages of an external documentation is that it can use whatever format and tool that is most convenient for the audience and for the writers. The drawback is that it's extremely hard, if not impossible, to ensure that an external documentation is up-to-date with respect to the latest version of the product. External documentation can also simply be lost.

In contrast, an internal documentation directly represents the knowledge by using the existing implementation technology. Using Java annotations or naming conventions on the language identifiers to declare and explain design decisions is a good example of an internal documentation.

The advantages of an internal documentation is it's always up-to-date with any version of the product, as it's part of its source code. Internal documentation cannot be lost as it's embedded within the source code itself. It's also readily available and comes to the attention of of any developer working on the code just because it's under their eyes.

Internal documentation also means you can benefit from all the tooling and all your the goodness of your fantastic IDE, like autocomplete, instant search, and seamless navigation within and between elements. The drawback is that your expressing the knowledge is limited to the possible extension mechanisms built-in the language. For example, as far as I know there's little we can do to extend the Maven XML with additional knowledge about each dependency. Another big drawback is that knowledge expressed as internal documentation is not readily accessible to non developers. However we know how to workaround that limitation with automated mechanisms that extract the knowledge and turn it into the kind of documents accessible to the right audience.

> If you're familiar with the book Domain-Specific Languages by Martin Fowler and Rebecca Parsons, you'll recognize the similar concept of an internal vs external DSL. An external DSL is independent from the chosen implementation technology. For example the syntax of regular expressions has nothing to do with the programming language

chosen for the project. In contrast, an internal DSL uses the regular chosen technology, like the Java Programming Language, in a way that makes it look like another language in disguise. This style is often called a Fluent style, and is common in mocking libraries.

Examples

Examples of internal documentation

- Self-documenting code and Clean Code practices, including
- Class and method naming, using Composed Methods
- Types
- Annotations that add knowledge to elements of the programming language
- Javadoc comments on public interfaces, classes and main methods
- Folder organization, modules and submodules decomposition and naming

It's not always easy to tell whether it's internal or external, as it's sometime relative to your perspective. Javadoc is a standard part of the Java Programming Language, so it's internal. But from the Java implementors perspective it's another syntax embedded within the Java syntax, so it would be external. Regular code comments are just in the middle grey area. They're formally part of the language, but do not provide anything more than free text. You're on your own to write them with your writing talent, and the compiler will not help check for typos beside the default spell-checking based on the English dictionary.

We'll take the point of view of the developer. From the perspective of the developer, every standard technology used to build the product can be considered as a host for internal documentation. Whenever we add documentation within the their artifacts, we benefit from the our standard toolset, with the advantage of being in the source control, close to the corresponding implementation so that it can evolve together with it.

- Feature files
- Markdown files and images next to the code with a naming convention or linked to from the code or feature files
- Tools manifests: dependency management manifest, automated deployment manifest, infrastructure description manifest etc.

Examples of external documentation

- README and LICENSE files
- Checkstyle configuration
- Any HTML, MS Office document about the project

Choosing between external or internal documentation

In this book I'm definitely in favor of internal documentation, coupled with just enough automation for the cases where it's necessary to publish more traditional documents out of it. I'd suggest choosing internal documentation by default, and at least for all knowledge that's at risk of changing regularly.

Even for stable knowledge I'd go for internal documentation first, and I would only chose to do external documentation when there's a clear value added, for example for a documentation that must be maximally attractive, perhaps for marketing reasons. In that case I'd do hand-crafted slides, diagrams with careful manual layout, and appealing pictures. The point of using external would be to be able to add a human feel to the final document, so I'd use Apple Keynote or MS Powerpoint, select or create beautiful quality pictures, and beta test the effectiveness of the documents on a panel of colleagues to make sure it's well received.

Note that appeal and humor are two things that are hard to automate or to encode into formal, but it's not impossible either (See Google Annotation Galery).

In situ Documentation

Internal documentation is also an *in situ documentation*

> In situ: situated in the original, natural, or existing place or position
>
> in situ. Dictionary.com. Dictionary.com Unabridged. Random House, Inc. http://dictionary.reference.com/brow situ (accessed: August 10, 2015). Based on the Random House Dictionary, (C) Random House, Inc. 2015.

This implies that the documentation is not only using the same implementation technology, but that it's also directly mixed into the source code, within the artifact that build the product. In Big Data space, "in situ data means bringing the computation to where data is located, rather than the other way". That's the same with in situ documentation, where any additional knowledge is directly added within the source code that is most related.

This is convenient for the developers. Like in designing user interfaces, where the term in situ means that a particular user action can be performed without going to another window, consuming and editing the documentation can be performed without going to another file or to another tool.

Machine-readable documentation

Good documentation focuses on high-level knowledge like the design decisions on top of the code, and the rationale behind these decisions. We usually consider this kind of knowledge to be only of

interest to people, but even tools can take advantage of them. Because internal documentation is expressed using implementation technologies, it's most of the time parseable by tools. This opens new opportunities for tools to assist the developers in their daily tasks. In particular it enables automated processing of the knowledge, for curation, consolidation, format conversion, automated publishing or reconciliation.

Accuracy Mechanism

When it comes to documentation, the main evil is that it's not accurate, usually because of obsolescence. Documentation that is not 100% accurate all the time cannot be trusted. As soon as we know it can be misleading from time time, it looses its credibility. It may still be a bit useful, but it will take more time to find out what's right and what's wrong in it. And when it comes to creating documentation, it's hard to dedicate time on it when we know it won't be accurate for long, it's a big motivation killer.

But updating documentation is one of the most ungrateful task ever. Almost everything is more interesting and rewarding than that. This is why we can't have nice documentation.

But in fact we can have a nice documentation, if we take the concern seriously and decide to tackle it with a well-chosen mechanism to enforce accuracy at all times.

You need to think about how you address accuracy of your documentation

Documentation by Design

As we've seen before, the authoritative knowledge, the one we can trust, is already somewhere, usually in the form of source code. In this perspective, the poison of documentation is to have duplicated knowledge, because it would multiply the cost of updating it in the case of change. This applies to source code, of course, and this applies to every other artifact too. We usually call "design" the discipline of making sure that change remain cheap at any point in time. We need design for the code, and we need the same design skills for everything about documentation.

A good approach for documentation is a matter of design. It takes design skills to design a documentation that is always accurate, without slowing down the software development work.

Accuracy Mechanism for a reliable documentation for fast-changing projects

With knowledge that can change at any time, there are a number of approaches to keep documentation accurate. They are listed below, ordered from the most to the least desirable

Single Sourcing

The knowledge is kept in a single source, that is authoritative. And that's it. This knowledge is only accessible to the people who can read the files. For example source code is a natural documentation of itself for developers, and with good code there's no need for anything else. For example a manifest to configure the list of every dependencies for a dependency management tool like Maven or NuGet is a natural authoritative documentation for the list of dependencies. As long as this knowledge is only of interest for developers, it's just fine as it is, there's no need for a publishing mechanism to make the knowledge accessible to other audiences.

Single Sourcing is the approach to favor whenever possible.

Single Sourcing with a Publishing Mechanism

The knowledge is kept in a single source, that is authoritative. It's made available under various forms as a published and versioned document thanks to an automated publishing mechanism. Anytime there's a change it's updated there and only there.

As an example, source code and configuration files are often the natural authoritative homes for a lot of knowledge. When necessary, the knowledge from this single source of truth is extracted and published in another form, but it remains clear that there is only one place that is authoritative. The publishing mechanism should also be automated to be ran frequently, and without introducing manual errors in the process.

Javadoc even without the additional comments is a good example of that approach: the reference documentation is the source code itself as parsed by the Javadoc Doclet, and it's published automatically as a website for everyone to browse the structure of interfaces, classes and methods, including the class hierarchies, in a convenient and always accurate manner.

Redundant Sources with a Propagation Mechanism

The knowledge may be duplicated in various places, but there's a reliable tooling that automatically propagates any change in one place to every other places. Automated refactorings in your IDE are the best examples of that approach. The class names, interfaces names and method names are repeated everywhere in the code, yet it's easy to rename them becase the IDE knows how to reliably chase every reference and update it correctly. This is far superior and safer than the good old *Find and Replace* that has the risk to replace random strings by mistake.

In a similar fashion documentation toolchains like *AsciiDoc* offer buit-in mechanisms to declare attributes once that you can then embed everywhere in the text; thanks to the built-in include and substitutions features you can rename and make changes in one place while propagating the change to many places at no cost.

Redundant Sources with a Reconciliation Mechanism

The knowledge is declared in two sources, so one source may change without the other. As it's a bad thing, there's a need for a mechanism to detect whenever the two sources don't match. The reconciliation mechanism should be automated and ran frequently to ensure permanent consistency.

BDD with automation tools like Cucumber is an example of this approach, with the code and the scenarios as the two sources of knowledge, both describing the same business behavior. Whenever a test running the scenarios fails, it's a signal that the scenarios and the code are no longer in sync.

Human Dedication (anti-pattern)

In this case there is no mechanism, and it's an anti-pattern. The knowledge may be duplicated in various places, and people in the team supposedly make sure everything remains consistent at all times through a lot dedication and hard grunt work. In practice it does not work, this is not recommended an approach

There are a few cases when there is no need for an accuracy mechanism:

Single-Use Knowledge

Sometime accuracy is just not a concern when the knowledge recorded at some place will be disposed right after use, within hours or a few days. This kind of transient knowledge does not age, does not evolve, hence there's not consistency concern about it, as long as it's actually disposed immediately after use, and assuming it's used for a short period of time. For example, conversations between pair in pair-programming and the code during each baby steps in TDD don't matter once the task is done

Account from the Past

An account of past events like a **Blog Post** does not have issues of accuracy, because it is clear for the reader that there is no promise of being accurate forever. The point is solely to describe a situation as it happened, with the thinking at the time, and the related emotions perhaps.

Such knowledge that is accurate at a point in time and that's recorded in the context of this point in time does not raise the same issues as obsolete documentation. The knowledge in the blog post does get outdated over time, but this is not a problem as it's clearly in the context of a blog post with a date and a story that's clearly in the past. This is a smart way to archive episodes of work and the big idea behind a story in a persistent fashion, without pretending its Evergreen. In the context of a blog post, it won't mislead anyone as it's clear it's an account of a past reflection, in the system as it was as this past time. As a story anchored in the past, it's always an accurate story, even if you can't trust the particular code or examples that may be quoted. It's like reading a book on History, and there's a lot precious lessons to be learnt regardless of in what context they happened.

The worst that may happen to an account from the past is to get irrelevant, when the concerns long ago are probably not the concerns you have now. Still, you can learn valuable lessons from this kind of letters from the past, even if the particular details are no longer relevant.

The Documentation Checklist

> Every minute crafting documents is a minute lost to other things. Is this adding value? Is this most important? – @dynamoben on Twitter

Imagine your boss, or a customer, asked for "more documentation". From that requirement, there are a number of important questions to be asked to decide how to go further. The goal behind these questions is to make sure you're gonna use your time as efficiently as possible, in the long run.

The ordering of the questions is indicative, usually you will skip or re-arrange the questions at will. This checklist is actually primarily meant to explain the thought process, and once understood you can make the process your own.

Questioning the need for documentation at all

Documentation is not an end in itself, it's a mean for a purpose that must be identified. We won't be able to make something useful unless we understand the goal. So the first question is:

> Why do we need this documentation?

It no answer comes easily, then we're definitely not ready to start investing extra effort in additional documentation. Let's put the topic on hold until we know better. No wasting time for ill-defined objectives.

Then the next question immediately follows:

> Who's the intended audience?

If the answer is unclear or sounds like "everyone", then we're not ready to start doing anything at this stage. Efficient documentation must target an identified audience. In fact even a documentation about things "that everyone should know" has to target an audience, for example "non technical people with only a superficial knowledge of the business domain".

With that in mind, and still determined to avoid wasting our time, we're ready for **The First Question of Documentation**:

The First Question of Documentation

Do we really need this documentation?

Someone may be tempted to create extra documentation on a topic that is only of interest for himself or herself, or only relevant for the time they're working on it. Perhaps it does not make that much sense to even add a paragraph to the wiki.

Need for documentation because lack of trust

Ii may happen that the answer sounds something like "I need documentation because I'm afraid you don't work as much as I'd like, so I need to see deliverables to make sure you work hard enough".

In that case however the main issue is not a matter of documentation. As @mattwynne @sebrose said at the BDD eXchange conference: "Need for detail might indicate lack of trust". Documentation is just a symptom. The root issue is lack of trust, and this serious enough that you should stop reading this book and try to find ways to improve that. No amount of documentation alone can fix lack of trust in the first place. However, since delivering value often is a good way to build trust, sensible documentation has a side role in a remediation. For example, making the work more visible may help build trust, and is a form of documentation.

Just-In-Time Documentation, or Cheap Option on Future Knowledge

If we need documentation, it does not mean that we need it now. That's our next question:

The Other First Question of Documentation

Do we really need this documentation **now**?

Creating documentation is a cost, for an uncertain benefit in the future. The benefit is uncertain when we cannot be sure someone will have the need for it in the future.

One thing we've learnt in the past years in software development is that we're notoriously bad at anticipating the future. Usually we can just bet, and our bets are often wrong.

As a consequence, we have a number of strategies available:

- **Just-In-Time**: add documentation only when really needed
- **Cheap Upfront**: add a little documentation now, at a very low cost
- **Expensive Upfront**: add documentation now, even if it takes time to create

Just-In-Time: Decide that the cost of documenting know is not worth the uncertainty that it'll be useful in the future, and differ the documentation until it becomes really necessary. Typically we'll wait for someone to ask the question to initiate the documentation effort. On a big project with lots of stakeholders we may even decide to wait for the second or third requests before deciding it's worth investing time and effort in creating documentation.

Note that this assumes that we'll still have the knowledge available somewhere in the team when the time has come to share it. It also assumes that the effort of documenting in the future will not be too high compared to what it would be right now.

Cheap Upfront: Decide that the cost of documenting right now is so cheap that it's not worth differing it for later, even if it's never actually used. This is especially relevant when the knowledge is fresh in mind and we run the risk that it'll be much harder later to remember all the stakes and important details. And of course it only makes sense if we have cheap ways to document the knowledge, as we'll see later.

Expensive Upfront: Decide that it's worth to bet on the future need for this knowledge by creating the documentation right now, even if it's not cheap. There's the risk it can be a waste, but we're happy to take this risk, hopefully for some substantiated reason (guidelines or compliance requirement, high confidence from more than one person that it's necessary etc.).

It's important to keep in mind that any effort around documentation right now also has an impact on the quality of the work, because it put the focus on the how it's done and why and acts like a review. This means that even if it's never used in the future it can be useful at least once, right now, for the sake of thinking clearly about the decisions and their rationale.)

Questioning the need for traditional documentation

Assuming that there's a genuine need for additional documentation, for an identified purpose and for an identified audience, we're now ready for **The Second Question of Documentation**.

The Second Question of Documentation

Could we just share knowledge through conversations or working together?

Documentation should never be the default choice, as it's too wasteful unless absolutely necessary. When we say that we need additional documentation, we mean that there's a need for knowledge transfer from some people to other people. Most of the time, this is best done by simply talking, asking and answering questions instead of written documents.

Working collectively, with frequent conversations, is a particularly effective form of documentation. Pair-programming, Cross-programming, the 3 Amigos, or Mob-programming totally change the game with respect to documentation, as knowledge transfer between people is done continuously and as the same time the knowledge is created or applied on a task.

Conversations and working collectively are the preferred form of documentation, to be preferred as a default choice, but it's not totally enough though.

Sometime there's a genuine need to have formalized knowledge.

Challenging the need for Formalized Documentation

Does it have to be persistent? Does it have to be shared to a large audience? Is it critical knowledge?

If the answer is three times no, conversations and working collectively should be enough, no need for more formal documentation.

You realize, of course, that if you ask the question to a manager you're more likely to be answered "yes", just because it's a safer choice. You can't be wrong by doing more, right? It's a bit like the priority on tasks, it's common for many people to put the high priority flag on everything, making it irrelevant. But what seems to be the safe choice carries a higher cost, which can in turn endanger the project. Therefore the safe choice is to really consider this triple question in a balanced way, with not too many "yes" for each "no".

In the case knowledge must be shared to a large audience, there are several options:

- Plenary meeting with the full audience attending
- Lecture-style Conference talk in front of a large audience
- Podcast or Video, like a recorder conference talk or a recorded interview
- Artifacts that are self-documented, that need no additional documentation
- Written document

In the case knowledge must be kept persistent for the long term, there are several options:

- Podcast or Video, like a recorder conference talk or a recorded interview
- Artifacts that are self-documented, that need no additional documentation
- Written document

Finally, knowledge that is critical has a little less options:

- Artifacts that are self-documented, that need no additional documentation
- Written document

The point is that, even with particularly important knowledge, written documentation does not have to the default choice.

Minimizing the extra work now

At this point, we assume that we have a legitimate need to keep some knowledge in a formal form. However we've seen before that most knowledge is already there, somewhere, in some form. So the next series of question is the following.

Knowledge is already there

Where's the knowledge right now?

If knowledge is only in the head of the people, then it needs to be encoded somewhere, as text, code, metadata etc.

If the knowledge is already represented somewhere, the idea is to use it or reuse it as much as possible. We call that knowledge exploitation, along with knowledge augmentation when necessary.

We'll use the knowledge that's in the source code, in the configuration files, in the tests, in the behavior of the application at runtime, and perhaps in memory of the various tools involved.

In this process described in the next chapters, we'll ask the following questions:

- Is the knowledge exploitable, or obfuscated or unrecoverable?
- Is the knowledge too abundant?
- Is the knowledge accessible for the intended audience?
- Is the knowledge in one single place or fragmented?
- What's missing to make it 100% explicit?

When the knowledge is not fully there or too implicit to be used, then the game becomes finding a way to add this knowledge directly into the source of the product.

Minimizing the extra work later

It's not enough to create a documentation once, we have to consider how to keep it accurate over time.

The most important remaining question is:

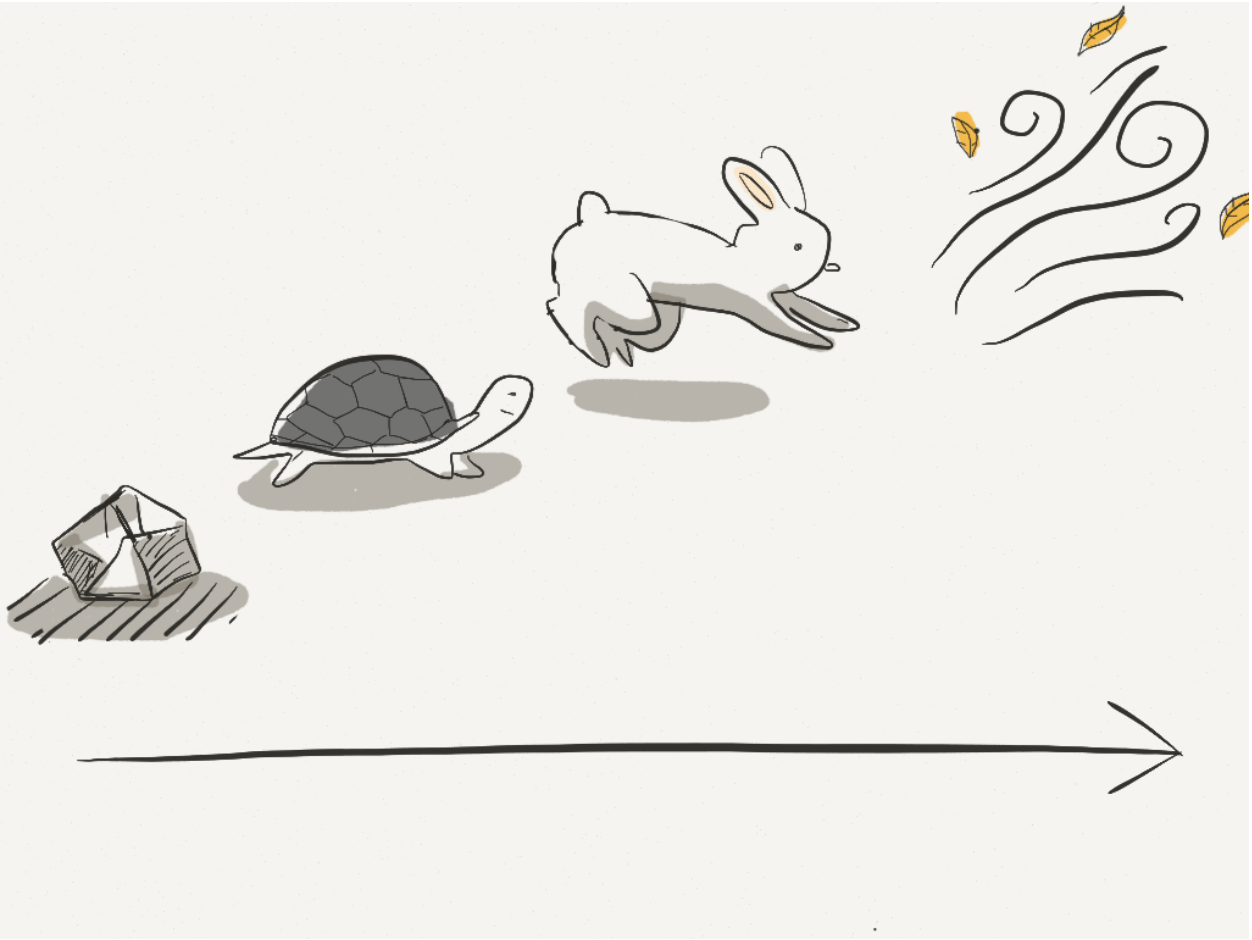
The Knowledge Stability Question

How stable is this knowledge?

Stable knowledge is easy, because we can ignore the question of its maintenance. On the other end of the spectrum, living knowledge is challenging. It can change often or at any time, and we don't want to update multiple artifacts and documents each time.

The rate of change is the crucial criterion. Knowledge which is stable over years can be taken care of with any traditional form, like writing text manually, and print it onto paper. Knowledge that is stable over years can even survive some amount of duplication, since the pain of updating every copy will never be experienced.

In contrast, knowledge which changes or may change every hour or more often just cannot afford such forms of documentation. The key concern to keep in mind of the cost of evolution and of maintenance of the documentation. Changing the source code and then having to update other documents manually is not an option.



The rate of change of the knowledge is the key criterion

In this process described in the next chapters, we'll ask the following questions:

- If it changes, what does change at the same time?
- If there's a redundant knowledge, how to make the redundant sources in sync?

Living Documentation - The Very Short Version

If you only want to spend 1 minute on what Living Documentation is all about, please remember the following big ideas:

> Favor conversations and working together over every kind of document
>
> Most knowledge is already there: it just wants to break free
>
> 99% of the knowledge is there already. Just needs to augment it with the extra 1%: context, intent, and rationale.

Pay attention to the frequency of change to choose the living documentation technique

Thinking about documentation is a way to draw attention to the quality or lack of thereof of the system being built

Is that clear enough? If so, congratulation, you've understood the key message.

Documentation Reboot

This book as a whole could actually be named *Documentation 2.0*, *Living Documentation*, *Continuous Documentation* or *No Documentation*. The key driver is to reconsider the way we do documentation, starting from the purpose. From there the universe of applicable solutions is near infinite. This book describes examples in various categories of approaches, and I expect the readers to go far beyond. Let's go through these categories now.

Approaches to better documentation

There are many ways to consider the topic of documentation in its wide definition. These approaches cover a full spectrum that can be seen as an clockwise cycle that follows a progression on several aspects.

This cycle goes crescendo in intensity, from *avoiding documentation* to *documentation to the max*, and beyond, where we question the need for documentation again and loop the cycle to less documentation again. We could say it also goes from rather lightweight approaches to more heavyweight ones.

This cycle primarily goes crescendo with respect to the rate of change (volatility) of the knowledge, from stable knowledge to knowledge that changes all the time, continuously.

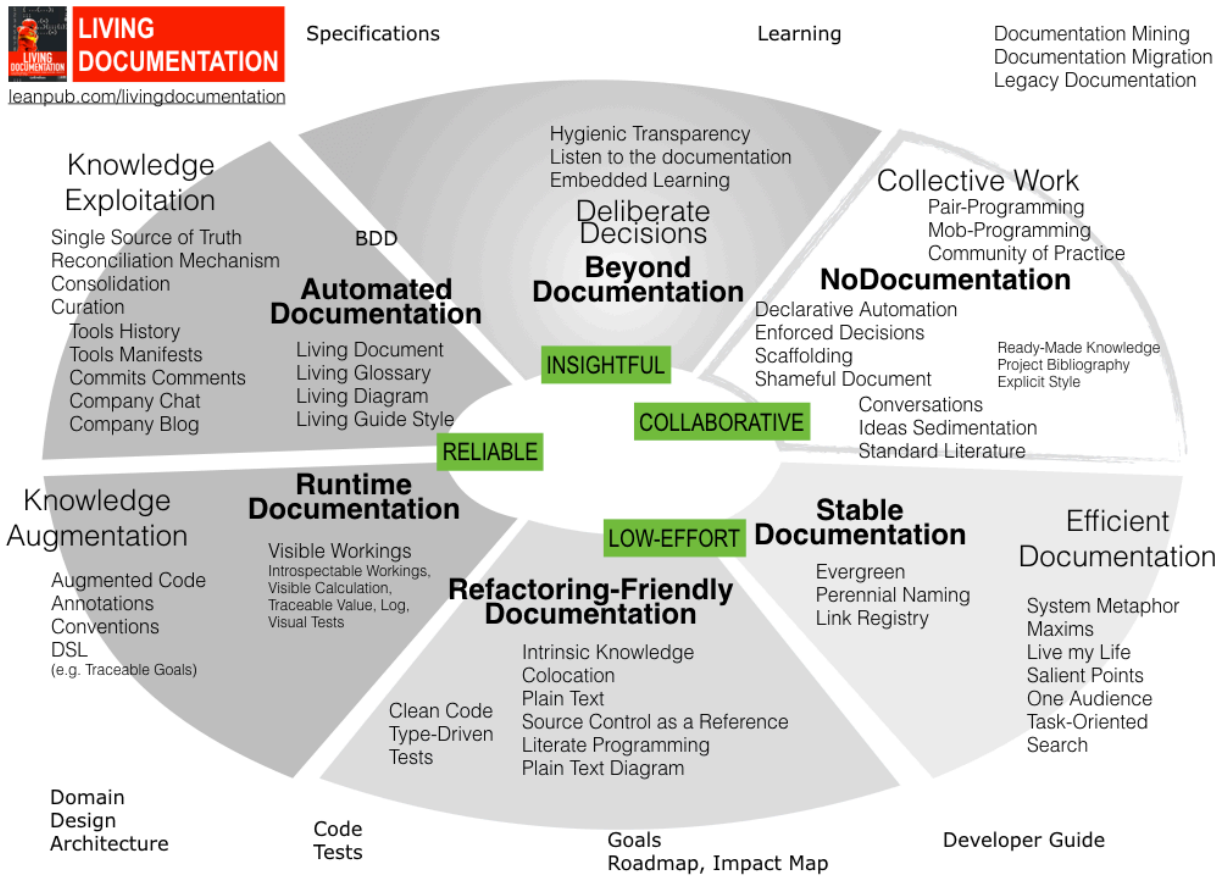
At 1 o'clock on this cycle we start with *No Documentation*, that also includes *Less Documentation*. This is about how to avoid or replace documentation with alternatives more appropriate or offering more benefits.

Then we move to *Stable Documentation*, for the lucky cases where it's possible to have stable knowledge that needs no maintenance.

Then comes the area of knowledge that changes often, but with the ability to have the documented knowledge follow the changes as they happen: *Refactoring-Friendly Knowledge*.

Further on the cycle we find *Automated Documentation*, for more heavyweight documentation that evolves in real-time with the changes in the software. *Runtime Documentation* is another flavor of such automation, but embedded in the working software.

Finally we reach the *Beyond Documentation* area, the opportunity to question everything and recognize that the topic of documentation can have benefits well beyond just transferring and storing knowledge. This is where we reach enlightenment and reconsider every other approach and technique in a more critical way.



The cycle of Living Documentation

No Documentation

The best documentation is often no documentation, because the knowledge is not worth any particular effort beyond doing the work. Collaboration with conversation or collective work are key here. Sometimes we can do even better and improve the underlying situation rather than workaround with documentation. Example include automation or fixing the root issues.

Stable Documentation

Not all knowledge changes all the time. When it's stable enough documentation becomes much simpler, and much more useful at the same time. Sometime it just takes one step forward to go from a changing piece of knowledge to a more stable one, an opportunity we want to exploit.

Refactor-Friendly Documentation

Code, tests, plain text, and a mix of all that have particular opportunities to evolve continuously in sync thanks to the refactoring abilities of the modern IDE and tools. This makes it possible to have accurate documentation for little to no cost.

Automated Documentation

This is the most geek area, with its specific tooling to produce documentation automatically in a living fashion, following the changes in the software construction.

Runtime Documentation

A particular flavor of Living Documentation involves every approach that operates at runtime, when the software is running. This is in contrast with other approaches that work at build time.

Beyond Documentation

Beyond the approaches on how to do better documentation, there's the even more important topic of "why and what for we do documentation". This is more meta, but this is where the biggest benefits are hidden. However, much like the *agile values* this is more abstract and this is better appreciated by people with some past experience.

These categories structure the main chapters of this book, in the reverse order. This reverse ordering follows a progression from more technical and rather 'easy to grasp', to more abstract and people-oriented considerations. However this also means the chapters progress from the less important to the more important.

Across these categories of approaches, there are some core principles that guide on how to do documentation efficiently.

Core Principles of Living Documentation

A Living Documentation is a set of principles and techniques for high-quality documentation at a low cost. It revolves around 4 principles that we'll keep in mind at all times:

- **Reliable** by making sure all documentation is accurate and in sync with the software being delivered, at any point in time.
- **Low-Effort** by minimizing the amount of work to be done on documentation even in case of changes, deletions or additions. It only requires a minimal additional effort, and only once.
- **Collaborative**: it promotes conversations and knowledge sharing between everyone involved.
- **Insightful**: By drawing attention to each aspect of the work, it offers opportunities for feedback and encourages deeper thinking. It helps reflect over the on-going work and guides towards better decisions

A Living Documentation also brings the fun back for developers and other team members. They can focus on doing a better job, and at the same time they get the Living Documentation out of this work.

Principles of a living documentation

The term “Living Documentation” first became popular in the book “Specifications by Examples” by Gojko Adzic. In this particular context it described a key benefit of teams doing BDD, where their scenarios created for specifications and testing were also very useful as a documentation of the business behaviors. Thanks to the test automation, this documentation is always up-to-date, as long as the tests are all passing.

It is possible to get the same benefits of a Living Documentation for all aspects of a software development project: business behaviors of course, but also business domains, project vision and business drivers, design and architecture, legacy strategies, coding guidelines, deployment and infrastructure.

Reliable

To be useful a documentation has to be trustful, in other words it has to be 100% reliable. Since humans are never that reliable we need discipline and tooling to help.

There are basically two ways to achieve reliable documentation:

- single source of truth: each element of knowledge is declared in exactly one single place (code, tests, runtime…). If we need it somewhere else, then we will link to it instead of making a copy. For example store the discount rate of 5% in a resource file and refer to it from the code and for documentation purposes. Don't copy the value 5% anywhere else.
- reconciliation mechanism: we accept that some elements of knowledge are declared in two different places. We acknowledge the risk that they are not consistent with each other (tests, validations). In BDD, the code and the scenarios both describe the behavior, so they are redundant. Thanks to a framework like Cucumber or Specflow, the scenarios become tests that act as a reconciliation mechanism: if a part of the code or a part of the scenario changes independently, the test fails so we know we have reconcile the code and he scenarios.

Low-Effort

- Simplicity: nothing to declare, it's just obvious.
- Standard over Custom solutions: standards are supposed to be known, and if that's not the case it is enough to just refer to the standard as an external reference like Wikipedia
- Perennial knowledge: there is always stuff that does not change or that changes very infrequently. As such it does not cost much to maintain.
- Refactoring-proof knowledge: stuff that don't require human effort when there is a change. This can be because of refactoring tools that automatically propagate linked changes, or because knowledge intrinsic to something is collocated with the thing itself, then changing and moving with it.

Collaborative

- Conversations over Documentation: nothing beats interactive, face-to/face conversations to exchange knowledge efficiently. Don't feel bad about not keeping a record of every discussion.
- Knowledge Decantation: even though we usually favor conversations, knowledge that is useful over a long period of time, for many people and that is important enough is worth some little effort to declare it somewhere persistent
- Accessible Knowledge: in a living documentation approach, knowledge is often declared within technical artifacts in a source control system. This makes it difficult for non-technical people to access it. Therefore, provide tools to make this knowledge accessible to all audiences without any manual effort.

- Collective Ownership: it's not because all the knowledge is in the source control system that developers own it. The developers don't own the documentation, they just own the technical responsibility to deal with it.

Insightful

- Deliberate design: If you don't know clearly what you're doing, it shows immediately when you're about to do living documentation. This kind of pressure encourages to clarify your decisions so that what you do becomes easy to explain.
- Embedded Learning: You want to write code that is so good that newcomers can learn the business domain by reading it and by running its tests.
- Emotional Feedback: A Living Documentation often leads to some surprise: "I did not expect the implementation to be that messy", "I thought I was shaved correctly but the mirror tells otherwise."

In the following chapters we'll describe a set of principles and patterns to implement a successful Living Documentation.

A Gateway Drug to DDD

Get closer to Domain-Driven Design by investing on Living Documentation

Living Documentation is a practical way to guide a team of a set of teams in their adoptions of the DDD practices. It helps make these practice more concrete with some attention on the resulting artifacts. Of course the way we work with the DDD mindset is much more important than the resulting artifacts. Still, the artifacts can at least help visualize before what DDD is about, and then they can help make visible any clear mis-practice, as a guidance on how well it's done or not.

Domain-Driven Design in a nutshell

Domain-Driven Design is an approach to tackle complexity in the heart of software development. It primarily advocates a sharp focus on the particular business domain being considered. It promotes writing code which expresses the domain knowledge literally, with no translation between the domain analysis and the executable code. As such, it calls for **modeling directly in code written in a programming language**, in contrast with a lot of literature on modeling. All this is only possible if there is the possibility of frequent and close conversations with domain experts, with everyone using the same **Ubiquitous Language**, the language of the business domain.

Domain-Driven Design calls for focusing the efforts on the **Core Domain**, the one business area with the potential to make a difference against the competition. As such, DDD encourages developers to not just to deliver code, but to contribute as partners with the business, in a constructive two-way relationship where the developers grow a deep understanding of the business and gain insights on its important stakes.

Domain-Driven Design is deeply rooted in Kent Beck's Extreme Programming, with the leitmotiv "Embracing Change". It also built on top of the pattern literature, most notably Martin Fowler's Analysis Patterns, and Rebecca Wirfs-Brock Responsibility-Driven Design, the book which coined the habit of naming practices in an "xDD" fashion.

The DDD book also includes numerous patterns to apply DDD successfully. Probably the single most important one is the notion of **Bounded Context**. A bounded context defines an area of the system where the language can be kept precise and without ambiguity. Bounded Contexts are a major contribution to system design, to simplify and partition large complicated systems into several smaller and simpler sub-systems (one by Bounded Context) without much downside. As splitting systems and work between teams efficiently is quite hard, the notion of Bounded Contexts is a powerful design tool in the tool belt.

Since the book was launched in 2003, most examples were proposed for application in object-oriented programming languages, but it has become clear since then that DDD applies just as well, with

functional programming languages. In fact I often make the claim that DDD advocates a functional programming style of code even in object oriented programming languages.

Living Documentation and Domain-Driven Design

This book is on Domain-Driven Design on several aspects:

- It promotes the use of DDD in your project, in particular through the chosen examples
- It shows how documentation can support the adoption of DDD and how it can act as a feedback mechanism to improve your practice
- It is in itself an application of DDD on the subject of documentation and knowledge management, in the way this topic is approached
- In particular, many of the practices of Living Documentation are actually directly DDD patterns from Eric Evans' book.
- The point of writing this book is to actually draw attention to design, or lack of thereof, through documentation practices which make it visible when the team sucks at design.

Does this make this book a book on DDD? I would think it does. As a fan of DDD, I would definitely love it to be.

Living Documentation is all about making each decision explicit, with not only in the consequences in code, but also with the rationale, context and the associated business stakes expressed, or perhaps we should say *modeled*, using all the expressiveness of the code as a documentation media.

What makes a project interesting is that it addresses a problem we have no standard solution for. The project has to discover how to solve the problem through **Continuous Learning**, with a lot of **Knowledge Crunching** while exploring the domain. As a consequence, the resulting code will change all the time, from small changes to major breakthrough.

> "Try, Try Again" requires a Change-Friendly Documentation

However, at all time it is important to keep the precious knowledge that took so much effort to learn. Once the knowledge is there, we turn it into a valuable and deliverable software by writing and refactoring source code and other technical artifacts. But we need to find ways to keep the knowledge through this process.

DDD advocates "Modeling with code" as the fundamental solution. The idea is that code itself is a representation of the knowledge. Only when the code is not enough do we need something else. Tactical patterns leverage on that idea that code is the primary medium, and they guide the developers on how to do that in practice using their ordinary programming language.

When Living Documentation is an application of DDD

Living Documentation not only supports DDD, it is also in itself an example of applying the DDD approach on the domain of managing knowledge throughout its lifecycle. And in many cases, Living Documentation is directly an applied case of DDD under a slightly different name.

A story of mutual roots between BDD, DDD, XP and Living Documentation

The word Living Documentation was introduced by Gojko Adzic in the book Specification by Example, which is a book on Behavior-Driven Development (BDD). BDD is an approach of collaboration between everyone involved in software development which was proposed by Dan North. Dan introduced BDD by combining Test-Driven Development (TDD) with the Ubiquitous Language of Domain-Driven Design. As a consequence, even the term of "Living Documentation" already has roots in Domain-Driven Design!

For example, Living Documentation strongly adheres to the following tenets of DDD

- **Code is the model** (and vice-versa), so we want to have as much as the knowledge of the model in the code, which is by definition the documentation
- **Tactical techniques** to make the code express all the knowledge: we want to exploit the programming languages to the maximum of what they can express, to express even knowledge is not executed at runtime
- **Evolving the knowledge all the time**, with the DDD whirlpool: the knowledge crunching is primarily a matter of collaboration between business domain experts and the development team. Through this process, some of the most important knowledge becomes embodied into the code and perhaps into some other artifacts. Because all the knowledge evolves or may evolve at any time, any documented knowledge much embrace change without impediment like the cost of maintenance
- **Making it clear what's important from what's not**, in other words a focus on curation: "focus on the core domain", "highlighting the core concepts" are from the DDD Blue Book, but there's much more we can do with curation to help keep the knowledge under control despite our limited memory and cognition capabilities
- **attention to details**: Even though it is not written as such, many DDD patterns emphasize that attention to details is important in the DDD approach. Decisions should be deliberate and not arbitrary, and guided by concrete feedback. An approach of documentation like Living Documentation has to encourage that, by making it easier to document what's deliberate, and by giving insightful feedbacks through its very process
- **Strategic design & large-scale structures**: DDD offers techniques to deal with evolving knowledge at the strategic and large-scale scales, which are opportunities for smarter documentation too.

It is hard to mention all the correspondances between the ideas of Living Documentation and their counterpart of Domain-Driven Design without re-writing parts of both books. But some examples are necessary to make the point.

Living Documentation pattern	**DDD pattern (from the book contents or from later contributions)**	**Notes**
Ready-Made Knowledge; Acknowledge Bibliography	Draw on Established Formalisms, When You Can; Read the Book; Applying Analysis Patterns	Clearly declare all the ready-made knowledge used with references to the sources
Evergreen Document	Domain Vision Statement	Higher-level knowledge is a great example of stable knowledge that can be written in an Evergreen document
Code as Documentation	Model-Driven Design; Intention-Revealing Interfaces; Declarative Design; The Building Blocks of a Model-Driven Design (to enable expressive code)	DDD is about modeling in plain code, with the purpose of having all the domain knowledge embodied in the code and its test
Living Glossary	Ubiquitous Language	When the code literally follows the Ubiquitous Language it becomes the single reference for the glossary of the domain
Listen to the documentation	Hands-On Modelers	Modeling in code with a living documentation extracted from it gives a fast feedback on the quality of the design, in a Hands-on fashion
Change-Friendly Documentation	Refactoring Toward Deeper Insight; Try, Try Again; Refactoring Toward Deeper Insight	"Embracing Change" is a permanent leitmotiv from Extreme Programming, DDD and Living Documentations
Curation	Highlighted Core; The Flagged Core; Segregated Core; Abstract Core	Segregating what is particularly important from the rest is a key driver in DDD, to best allocate effort and cognitive attention

Living Documentation exploits all that to go beyond traditional documentation and its limitations.

It elaborates on the DDD techniques and advices for knowledge about the business domain but also for the knowledge about the design, and even about the infrastructure and delivery process, which are technical domains too with respect to the project stakeholders. The ideas from Domain-Driven Design are essential to guide developers on how to invest in knowledge in a tactical and strategic way, dealing with change in the short term and in the long term as well. As such, as you are going the Living Documentation route you are learning Domain-Driven Design too.

A principled approach

To better organize what is Living Documentation, its values and principles, here's the The Spine Model[6] for it. It starts with stating the need we acknowledge and that we decide to address. Then we clarify the main values that we want to optimize for. A list of principles follows, and is there to help change the current situation.

By keeping the needs, the main values and the principles in mind, in this order of importance, we can then apply practices and use tools to get the work done in an effective fashion.

Need

Evolve software continuously, collectively and over the long run.

We want to deliver software quickly now, and at least as quickly in the future. We need to collaborate as a team, and when necessary with even more people who can't always meet at the same time or at the same place.

We want to take the best possible decisions based on the most relevant knowledge, in order to make the work on the software sustainable in the long run.

Values

We optimize for the following values:

1. Deliberate Thinking
2. Continuous Knowledge Sharing
3. Fruitful Collaboration
4. Honest Feedback
5. Fun

Principles

We leverage the following principles to change the way we work:

1. Conversations over Documents

[6] http://spine.wiki/explanation/introduction/

2. Most Knowledge is Already There
3. Internal Documentation over External Documentation
4. Automate for the long term
5. Enforced Accuracy
6. Single Source of Truth
7. Working Collectively
8. Documentation is also at Runtime
9. Documentation is Communication
10. Segregation by Pace of Change
11. Make Documentation Unnecessary
12. A Whole Activity (carefully done, with multiple benefits)

Practices

We have the following practices available to deliver value:

- Living Diagram
- Living Glossary
- Declarative Automation
- Enforced Guidelines
- Small-Scale Model
- and many others that are the focus of this book.

Tools

To get the work done, we use tools. Most of them are primarily mental tools, but tools that we can download also help of course!

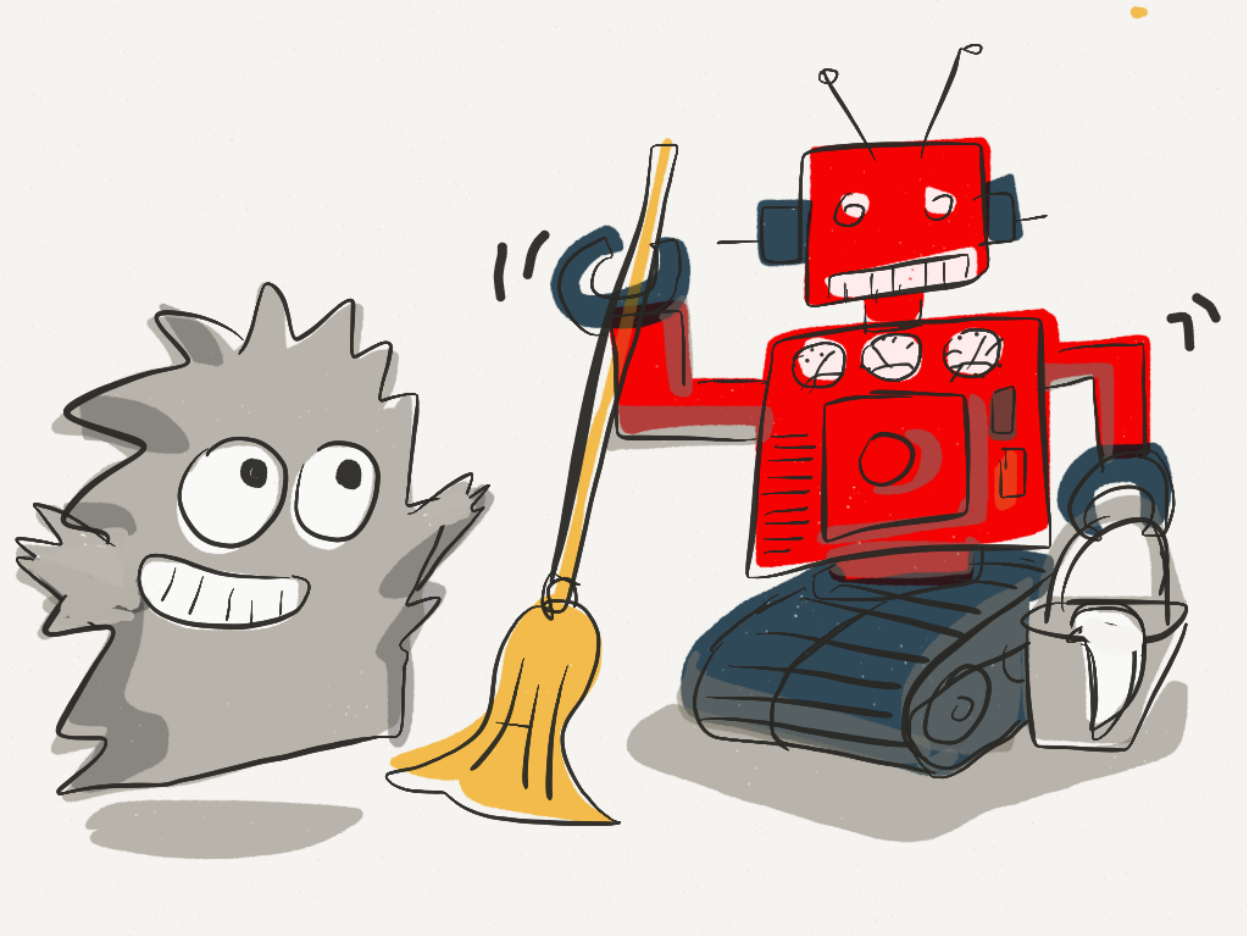
There are many tools of interest for a Living Documentation, and they evolve quickly. This list of tools starts with your regular programming languages, and extends to automation tools on top of your practice of BDD, like Cucumber or SpecFlow, and includes rendering engines for Markdown or AsciiDoc, and automatic layout engine for diagrams like Graphviz.

Fun

Fun is important for sustainable practices. If it's not fun, you'll not want to do it so often and the practice will progressively disappear. For practices to last, they'd better be fun. It's particularly important on so boring a topic like documentation.

Therefore: Chose practices that help satisfy the needs according to the principles, while being as fun as possible. If it's fun, do more of it, and if it's totally not fun, look for alternatives, like solving the problem in another way or through automation.

This assumes that working with people is fun, because there's no good way around that. For example, if coding is fun, we'll try to document as much as possible in code. That's the idea behind many suggestions in this book. If copying information from one place to another is a chore, then it's a candidate for automation, or for finding a way for not having to move data at all. Fixing the process or automating a part of it are more fun, so we're back to something that we feel like doing. That's lucky.



Fun starts with automating the chores

A rant that is not fun

One more thing: there's nothing wrong with having fun at work, as long as we're professional in our work. This means doing our best to solve the problems that matter, delivering value, reducing risk. With that in mind, we're free to chose the practices and tools that make our life more fun. After 15 years in programming I'm now confident it's always possible to do professional work while having fun. The idea that work should be boring and unpleasant because it's work, or because we're paid for it to compensate for this very unpleasantness, is just stupid. We're paid some money to deliver value worth even more money. Delivering value is fun, and behaving professionally is pleasant too. And fun is essential for working efficiently as a team, in a pleasant atmosphere.

Part 2 Living Documentation exemplified by Behavior-Driven Development

A key example of Living Documentation: BDD

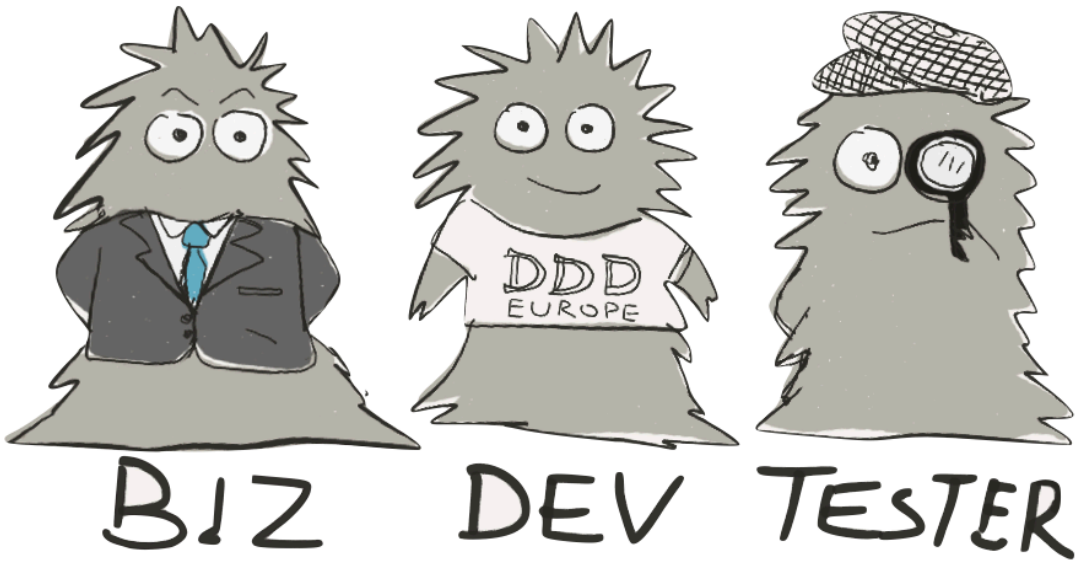
> What about documenting the business behavior? (because as you know, business people never change their mind.)

Behavior-Driven Development (BDD) is the first example of a Living Documentation. In the book *Specification by Example*, Gojko Adzic explains that when interviewing many teams doing BDD, one of the biggest benefit they mention is having this Living Documentation always up-to-date that explains what the application is doing.

Before going any further, let's quickly precise what BDD is, and what it's not.

BDD is all about conversations

If you think BDD is about testing, forget all you know about it. BDD is about sharing knowledge efficiently. This means that you can do BDD without any tool. Before anything else, BDD promotes deep conversations between the 3 amigos (or more) and the use of concrete scenarios to detect misunderstandings and ambiguities early. These scenarios must use the language of the business domain.



The 3 amigos

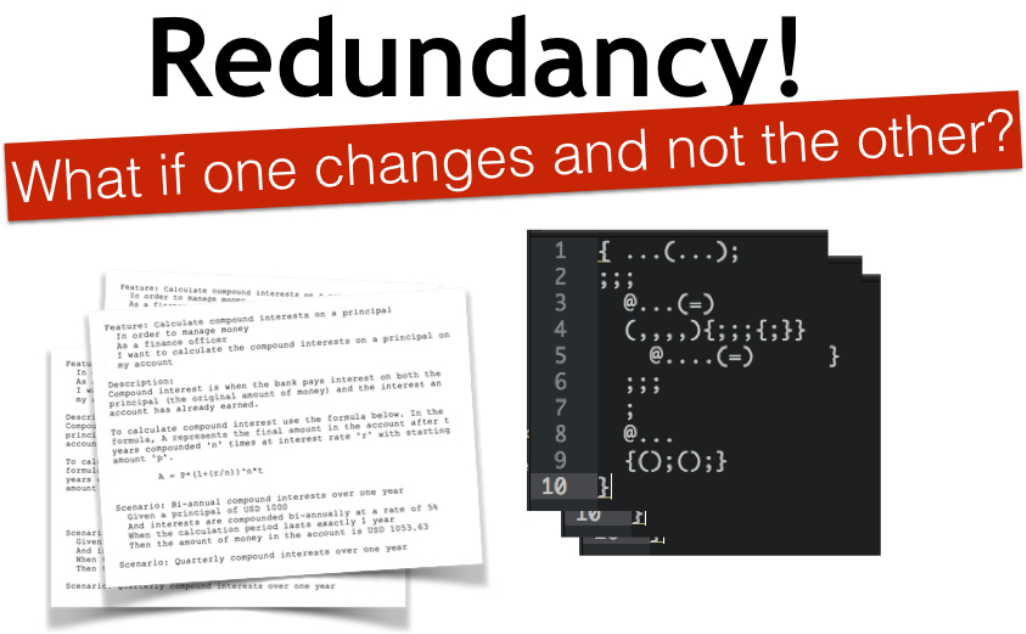
BDD with just conversations and no automation is already BDD, and there's already a lot of value of doing just that. However with additional effort to setup automation, you can reach even more benefits.

BDD with automation is all about Living Documentation

When using a tool like Cucumber, BDD still advocates the use of a domain language between every stakeholder involved and in particular between the 3 Amigos, a focus on the higher-level purpose, and the frequent use of concrete examples, also known as scenarios. These scenarios then become tests in the tool, and become a living documentation at the same time.

Redundancy + Reconciliation

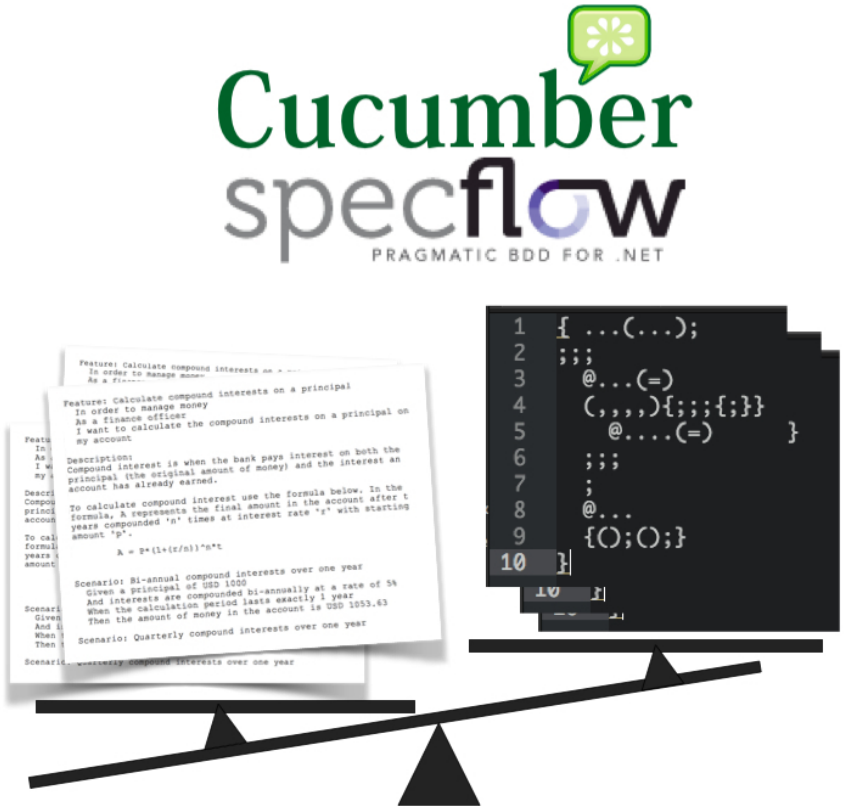
BDD scenarios describe the behavior of the application, but the source code of the application also describes this behavior: they are redundant with each other.



Scenarios and code each describe the same behavior

On one hand hand this redundancy is good news: the scenarios expressed in pure domain language, if done properly, are accessible to non-technical audiences like business people who could never read code. However this redundancy is also a problem: if some scenarios or parts of the code evolve independently, then we have two problems: what should we trust, the scenarios or the code? But we also have a bigger problem: how do we even know that the scenarios and the code are not in sync?

This is where a reconciliation mechanism is needed. In the case of BDD, we use tests and tools like Cucumber or SpecFlow for that.



Tools check regularly that the scenarios and the code describe the same behavior

These tools parse the scenarios in plain text and use some glue code provided by the developers to drive the actual code. The amounts, dates and other values in the Given and When sections of the scenarios are extracted and passed as parameters when calling the actual code. The values extracted from the Then sections of the scenarios on the other hand are used for the assertions, to check the result from the code matches what's expected in the scenario.

In essence, the tools take scenarios and turn them into automated tests. The nice thing is that these tests are also a way to detect when the scenarios or the code are no longer in sync. This is an example of a reconciliation mechanism, a mean to check redundant sets of information always match.

Anatomy of the scenarios in file

When using a tool like Cucumber or Specflow to automate the scenarios into tests, you create files called Feature files. These files are plain text files, and are stored in the source control just like code. Usually they are stored nearby the tests, or as Maven test resources. This means that they are versioned like the code too, and are easy to diff.

Let's have a closer look at a a feature file.

Intent

The file must start with a narrative that describes the intent of all the scenarios in the file. It usually follows the template **In order to**... **As a**... **I want**.... Starting with "In order to..." helps focus on the most important thing: the value we're looking for.

Here's an example of a narrative, for an application about detection of potential frauds in the context of fleet management for parcel delivery:

```
Feature: Fuel Card Transactions anomalies
 In order to detect potential fuel card abnormal behavior by drivers
 As a fleet manager
 I want to automatically detect anomalies in all fuel card transactions
```

Note that the tools just consider the narrative as text, they don't do anything with it except including it into the reports, because they acknowledge it's important.

Scenarios

The rest of the feature file usually lists all the scenarios that are relevant for the corresponding feature. Each scenario has a title, and almost always follows the **Given**... **When**... **Then**... template.

Here's an example of one out of the many concrete scenarios for our application on detection of potential frauds, in the context of fleet management for parcel delivery:

```
Scenario: Fuel transaction with more fuel than the vehicle tank can hold
 Given that the tank size of the vehicle 23 is 48L
 When a transaction is reported for 52L on the fuel card associated with vehicle 23
 Then an anomaly "The fuel transaction of 52L exceeds the tank size of 48L" is repor\
ted
```

Within one feature file there are between 3 and 15 scenarios, describing the happy path, its variants, and the most important situations.

There are a number of other ways to describe scenarios, like the outline format, and to factor out common assumptions between scenarios with background scenarios. However this is not the point of this book, and other books or online resources do a great job at explaining that.

Specification details

There are many cases where scenarios alone are enough to describe the expected behavior, but in some rich business domains like finance there are definitely not enough. We also need abstract rules and formula.

Rather than putting all this additional knowledge in a Word document or in a wiki, you can also directly embedded it directly within the related feature file, between the intent and the list of scenarios. Here's an example, still from the same feature file as before:

```
Feature: Fuel Card Transactions anomalies
In order to detect potential fuel card abnormal behavior by drivers
As a fleet manager
I want to automatically detect anomalies in all fuel card transactions

Description:
The monitoring detects the following anomalies:
* Fuel leakage: whenever capacity > 1 + tolerance,
where capacity = transaction fuel quantity / vehicle tank size
* Transaction too far from the vehicle: whenever distance to vehicle > threshold,
where distance to vehicle = geo-distance (vehicle coordinates, gas station coordinat\
es),
and where the vehicle coordinates are provided by the GPS Tracking by (vehicle, time\
stamp),
and where the gas station coordinates are provided by geocoding its post address.

Scenario: Fuel transaction with no anomaly
  When a transaction is reported on the fuel card
  ...// more scenarios here
```

These specification details are just comments as free text though, the tools completely ignore it. However the point of putting it there is to have co-located with the corresponding scenarios. Whenever you change the scenarios or the details, you are more likely to update the specification details because they are so close, as we say "out of sight, out of mind". But there is not guarantee to do so.

Tags

The last significant ingredient in feature files are tags. Each scenario can have tags, like the following:

```
@acceptance-criteria @specs @wip @fixedincome @interests
Scenario: Bi-annual compound interests over one year
  Given a principal of USD 1000
  ...//
```

Tags are documentation. Some tags describe project management knowledge, like *@wip* that stands for Work In Progress, signaling that this scenario is currently being developed. Other similar tags may even name who's involved in the development: *@bob*, *@team-red*, or mention the sprint: *@sprint-23*, or its goal: *@learn-about-reporting-needs*. These tags are temporary and are deleted once the tasks are all done.

Some tags describe how important the scenario is, like *@acceptance-criteria*, meaning this scenario is part of the few user acceptance criteria. Other similar tags may help curation of scenarios: *@happy-path*, *@nominal*, *@variant*, *@negative*, *@exception*, *@core* etc.

Lastly, some tags also describe categories and concepts from the business domain. For example here the tags *@fixedincome* and *@interests* describe that this scenario is relevant with respect to Fixed Income and Interest financial areas.

Tags should be documented too.

One folder, one chapter

When the number of feature files grows, it's necessary to organize them into folders. This organization is also a way to convey knowledge, we want the folders to tell a story too.

When the business domain is the most important thing, I'd recommend to organize the folders by functional areas, to show the overall business picture:

- accounting
- reporting rules
- discounts
- special offers, etc.

If you have any additional content as text and pictures, you can also include it in the same folders, so that it stays as close as possible to the corresponding scenarios.

In the book "Specification by Example", Gojko Adzic lists 3 ways to organize stories into folders:

- Organize by functional areas
- Organize along UI navigation routes (When documenting user interfaces)
- Organize along business processes (When end-to-end use case traceability is required)

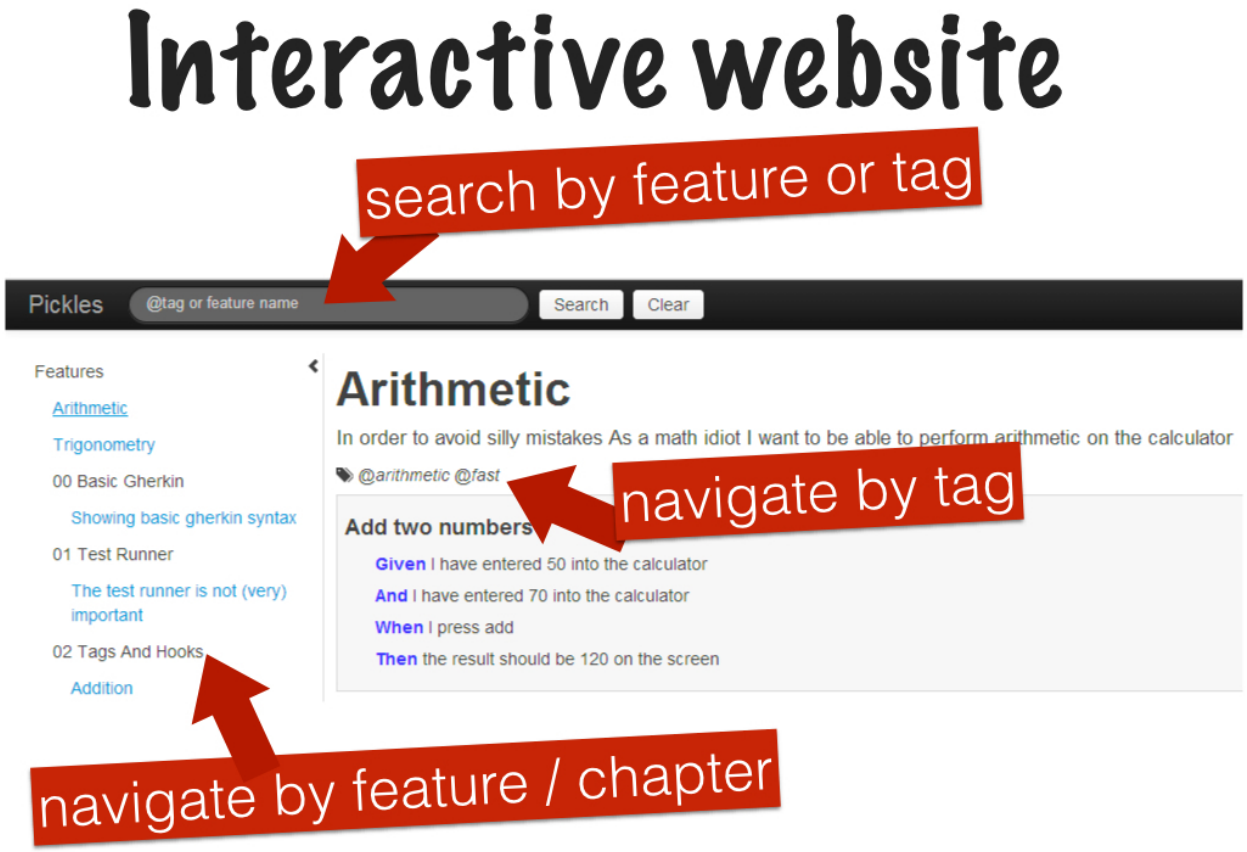
With this approach, the folders literally represent the chapters of your business documentation.

Another example of a full feature from another application is included at the end of this section.

Interactive Living Documentation

There's more than just automation as tests, as the scenarios also form the basis of a living documentation. Even better, this documentation is typically interactive, as a generated interactive website.

For example, using Pickle for Specflow, a specific one-page website is generated during each build. It displays all scenarios, together with the test results and their statistics. This is quite powerful, much more so than every paper documentation you've ever seen.



Generated interactive documentation website, with Pickles

There's a built-in search engine, allowing instant access to any scenario by keyword or by tag. This is the second powerful effect of tags, they make search more efficient and accurate.

The website shows a navigation pane that is organized by chapter, provided that your folders represent functional chapters.

Boring paper document

The interactive website is convenient for the team, for fast access to the business behavior knowledge. However there are cases where you have to provide a boring paper document (a "BPD" as some call it) e.g for mandatory Compliance requirements.

There are tools for that too. One was developed by my Arolla colleague Arnauld Loyer (@aloyer) and it's called Tzatziki, because it's a Cucumber sauce. It exports a beautiful PDF document out of the feature files. It goes a bit further than what's covered in this section. For example it also includes markdown files and images that are stored alongside the feature files into the document. This helps create nice explanations at the beginning of each functional area chapter.

This is a world in motion. If the tool you need in your context is missing, you should create it on top or as a derivation of existing tools. The sky's the limit.

BDD is a great example of a living documentation: it's not an additional work to be done, it's part of doing the work properly. It's always in sync, thanks to the tools that act as reconciliation mechanisms. And if the feature files in the source code are not enough, the generated website illustrates how a documentation can be useful, interactive, searchable and well-organized.

Feature File: another Example

Here's a full example of a fictitious yet realistic feature file in the business domain of finance.

To keep this example short, it only contains one outline scenario, along its corresponding data table. This illustrates another style of using Cucumber, Specflow and equivalent tools. The scenario is evaluated for each line of the table.

```
Feature: Calculate compound interests on a principal
  In order to manage the company money
  As a finance officer
  I want to calculate the compound interests on a principal on my account

Description:
Compound interest is when the bank pays interest on both the principal (the original\
 amount of money) and the interest an account has already earned.

To calculate compound interest use the formula below. In the formula, A represents t\
he final amount in the account after t years compounded 'n' times at interest rate '\
r' with starting amount 'p'.

  A = P*(1+(r/n))^n*t


```

```
Scenario: Bi-annual compound interests over one year
  Given a principal of USD 1000
  And interests are compounded bi-annually at a rate of 5%
  When the calculation period lasts exactly 1 year
  Then the amount of money in the account is USD 1053.63

Scenario: Quarterly compound interests over one year
//... outline scenario

Examples:

| convention  | rate  | time | amount    | remarks                    |
|-------------------------------------------------------------------|
| LINEAR      | 0.05  |  2   | 0.100000  | (1+rt)-1                   |
| COMPOUND    | 0.05  |  2   | 0.102500  | (1+r)^t-1                  |
| DISCOUNT    | 0.05  |  2   | -0.100000 | (1 - rt)-1                 |
| CONTINUOUS  | 0.05  |  2   | 0.105171  | (e^rt)-1 (not used often)  |
| NONE        | 0.05  |  2   | 0         | 0                          |
|-------------------------------------------------------------------|
```

How does it work?

With the support of tools, all the business scenarios become automated tests and a living documentation at the same time.

The scenarios are just plain text in the “feature files”. To bridge the gap between the text in the scenarios and the actual production code, you create a little set of “steps”. Each step is triggered on a particular text sentence, matched by regular expressions, and calls the production code. The text sentence may have parameters that are parsed and used to call the production code in many different ways.

```
For example:
Given the VAT rate is 9.90%
When I but a book at a ex-VAT price of EUR 25
Then I have to pay an inc-VAT price of EUR 2.49
```

To automate this scenario you need a step definite for each line. For example:

The sentence:

```
"When I but a book at a ex-VAT price of EUR <exVATPrice>"
```

Triggers the glue code:

```
Book(number exVATPrice)
Service = LookupOrderService();
Service.sendOrder(exVATPrice);
```

The result of this plumbery is that the scenarios become automated tests. These tests are driven by the scenarios and the values they declare. If you change the rounding mode of the price in the scenario without changing the code the test will fail. If you change the rounding mode of the price in the code without changing the scenario the test will fail too: this is a reconciliation mechanism to signal inconsistencies between both sides of the redundancy.

A canonical case of Living Documentation in every aspect

BDD has shown that it was possible to have an accurate documentation, always in sync with the code, by doing the specification work more carefully. BDD is a canonical case of Living Documentation (the word itself comes from there), where every principle of Living Documentation are already there:

- **Conversations over Documentation**: the primary tool of BDD is talking between people, making sure that each role out of the 3 amigos (or more) is present.
- **Targeted Audience**: All this work is targeted for an audience that include business people, hence the focus on clear, non technical language when discussing business requirements.
- **Idea Sedimentation**: Conversations are often enough, not everything deserves to be written down. Only the most important scenarios, the **key scenarios** will be written for archiving or automation.
- **Plain Text Documents**: because plain text is hyper convenient for managing stuff that changes, and to live along the source code in source control.
- **Reconciliation Mechanism**: because the business behaviors are described both in text scenarios and in implementation code, tools like Cucumber or Specflow make sure both remain always in-sync, or at least they show when then don't. This is necessary whenever there is duplication of knowledge.
- **Accessible Published Snapshot**: Not everyone has or wants access to the source control in order to read the scenarios. Tools like Pickles or Tzatziki offer a solution, by exporting a snapshot of all the scenarios at a current point in time, as an interactive website or as a PDF document that can be printed.

Now that we've seen BDD as the canonical case of Living Documentation, we're ready to move on to other contexts where we can apply Living Documentation too. Living Documentation is not restricted to the description of business behaviors as in BDD, it can help our life in many other aspects of our software development projects, and perhaps even outside of software development.

Going further: Getting the best of your living documentation

Feature files describing business scenarios are a great place to gather rich domain knowledge in an efficient way.

Most tools to support team doing BDD understand the Gherkin syntax. They expect feature files following a fixed format:

```
Feature: Name of the feature

In order to... As a... I want...

Scenario: name of the first scenario
 Given...
 When...
 Then...

Scenario: name of the second scenario
  ...
```

Over time, teams in rich domains like finance or insurance realized they needed more documentation than just the intent at the top and the concrete scenarios at the bottom. As a result, they started putting additional description of their business case in the middle area, ignored by the tools. Tools like Pickles which generate the living documentation out of the feature files adapted to this use and started to support Markdown formatting for what became called "the description area":

```
Feature: Investment Present Value

In order to calculate the breakeven point of the investment opportunity
As an investment manager
I want to calculate the present value of future cash amounts


Description
===========
```

```
10
11  We need to find the present value *PV* of the given future cash amount *FV*. The for\
12  mula for that can be expressed as:
13
14  - Using the negative exponent notation:
15
16          PV = FV * (1 + i)^(-n)
17
18  - Or in the equivalent form:
19
20          PV = FV * (1 / (1 + i)^n)
21
22  Example
23  -------
24
25           PV?                                FV = $100
26           |                  |                 |
27           ---------------------------------------> t (years)
28           0                  1                 2
29
30            For example, n = 2, i = 8%
31
32
33  Scenario: Present Value of a single cash amount
34   Given a future cash amount of 100$ in 2 years
35   And an interest rate of 8%
36   When we calculate its present value
37   Then its present value is $85.73
```

This will be rendered in the living documentation website as a pretty document:

Feature: Investment Present Value

In order to calculate the breakeven point of the investment opportunity As an investment manager I want to calculate the present value of future cash amounts

Description

We need to find the present value *PV* of the given future cash amount *FV*. The formula for that can be expressed as:

- Using the negative exponent notation:

```
    PV = FV * (1 + i)^(-n)
```

- Or in the equivalent form:

```
    PV = FV * (1 / (1 + i)^n)
```

Example

```
    PV?                                FV = $100
    |                  |               |
    ------------------------------------> t (years)
    0                  1               2

     For example, n = 2, i = 8%
```

Scenario: Present Value of a single cash amount

```
    Given a future cash amount of 100$ in 2 years
    And an interest rate of 8%
    When we calculate its present value
    Then its present value is $85.73
```

No guarantee of correctness

This opens a lot of potential to gather every documentation in the same place, directly within the source control. Note that this kind of description is not really living, it is just co-located with the scenarios which; if we change the scenarios, we're just more likely to also update the description on top, but there is no guarantee.

The best strategy would be to put knowledge that does not change very often in the description section, and to keep the volatile parts within the concrete scenarios. One way to do that is to clarify that the description uses example numbers, not the numbers necessarily used for the configuration of the business process at any point in time.

Tools like Pickle[7], Relish[8] or the tool created by my Arolla colleague Arnauld Loyer Tzatziki[9] now understand Markdown descriptions and even plain Markdown files located next to the feature files. This makes it easy to have an integrated and consistent approach for the domain documentation. And Tzatziki can export a PDF from all this knowledge, as expected by the regulators in finance.

Property-Based Testing and BDD

Requirements often come naturally as properties: "The sum of all amounts paid and received must be zero at all times", or "nobody can ever be a lawyer and a judge at once". When doing BDD or TDD, we must clarify these general properties into specific concrete examples, which will help find out issues and build code incrementally.

It's a good idea to keep track of the general properties for their documentation value. We usually do that as plain text comment in the feature file as described before. But it happens that the technique of Property-Based Testing is precisely about exercising these properties against randomly generated samples. This is performed with a property-based testing framework that runs the same test over and over with inputs generated from generators of samples. The canonical framework is QuickCheck in Haskell, and there are now similar tools in most programming languages.

Integrating property-based testing into your feature files eventually makes the general properties executable too. In practice it's a matter of adding special scenarios describing the general property, invoking the property-based testing framework underneath:

```
1 Scenario: The sum of all cash amounts exchanged must be zero for derivatives
2  Given any derivative financial instrument
3  And a random date during its life time
4  When we generate the related cash flows on this date for the payer and the receiver
5  Then the sum of the cash flows of the payer and the receiver is exactly zero
```

Such scenarios typically use sentences like "given ANY shopping cart…". This wording is a code smell for regular scenarios, but it's ok for property-oriented scenarios on top of property-based testing tooling, supplementing the regular concrete scenarios.

Manual glossary

The idea glossary is a living one, extracted directly from your code. However in many cases this approach is not possible, but you'd still want a glossary.

[7] http://www.picklesdoc.com/
[8] http://www.relishapp.com/
[9] https://github.com/Arnauld/tzatziki

It's possible to do a glossary manually as a Markdown file and to co-locate it with the other feature files. This way it will be included in the living documentation website too. You could even do it as dummy empty *.feature* file.

Linking to non-functional knowledge

Not all the knowledge shall be described in the same place. In particular we don't want to mix domain knowledge with UI-specific or legacy-specific knowledge. This knowledge is important and should be stored elsewhere. And when it's related to the domain language, then we should use links to represent the relationship and make it easy to find it.

As described in this book, you can use different approaches to linking: you may link directly to an URL, with a risk of having a broken link whenever it changes.

```
https://en.wikipedia.org/wiki/Present_value
```

You may go through a link registry that you maintain to manage links and to replace broken links with working ones.

```
go/search?q=present+value
```

You may also use bookmarked searches to link to places that include the related content.

```
https://en.wikipedia.org/w/index.php?search=present+value
```

This way you can have a resilient way to link to related content, at the expense of letting the reader select the most relevant results each time.

Part 4 Automated Documentation

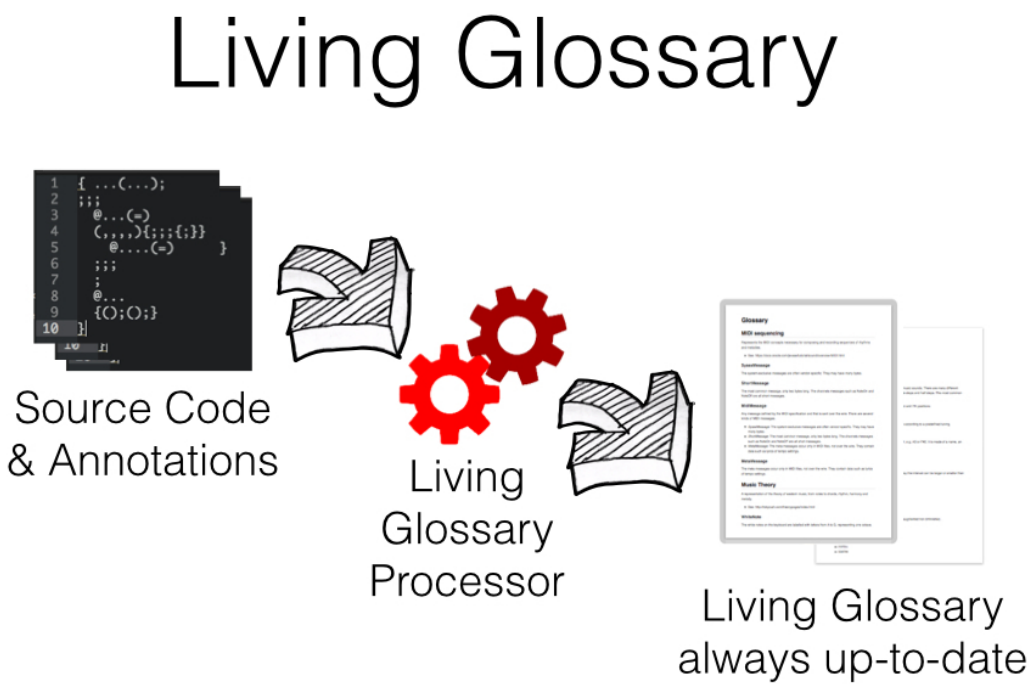
Living Glossary

How to share the Ubiquitous Language of the domain to everyone involved in a project?

The usual answer is to provide a complete glossary of every term that belongs to the Ubiquitous Language, together with a description that explains what you need to know about it. However the Ubiquitous Language is an evolving creature, so this glossary needs to be maintained, and there is the risk it becomes outdated compared to the source code.

In a domain model, the code represents the business domain, as closely as possible to the way the domain experts think and talk about it. In a domain model, great code just literally tells the domain business: each class name, each method name, each enum constant name and each interface name is part of the Ubiquitous Language of the domain*. But not everyone can read code, and there are almost always some code that is less related to the domain model.

Therefore: Extract the glossary of the Ubiquitous Language from the source code. Consider the source code as the Single Source of Truth, and take great care of the naming of each class, interface and public method whenever they represent domain concepts. Add the description of the domain concept directly into the source code, as structured comments that can be extracted by a tool. When extracting the glossary, find a way to filter out code that is not expressing the domain.



Overview of a Living Glossary

A successful Living Glossary requires the code to be declarative. The more the code looks like a DSL of the business domain, the better the glossary. Indeed for developers there will be no need for a Living Glossary, because the glossary is the code itself. A Living Glossary is a matter of convenience, especially useful for non developers who don't have access to the source core in an IDE. It brings additional convenience in being all on a single page.

A Living Glossary is also a feedback mechanism. If your glossary does not look good, or if it's hard to make it work, this is a signal that suggests you have something to improve in the code.

How it works

In many languages documentation can also be embedded directly within the code as structured comments, and it is good practice to write a description of what a class, interface or important method is about. Tools like Javadoc can then extract the comments and report a reference documentation of the code. The good thing with Javadoc is that you can create your own Doclet (documentation generator) based on the provided Doclet, and this does not represent a large effort. Using a custom Doclet, you can export custom documentation in whatever format.

Annotations in Java and attributes in C# are great to augment code. For example you can annotate classes and interfaces with custom domain stereotypes (@DomainService, @DomainEvent, @BusinessPolicy etc.), or on the other hand domain-irrelevant stereotypes (@AbstractFactory, @Adapter etc.). This makes it easy to filter out classes that do not contribute to expressing the domain language. Of course you need to create this small library of annotations to augment your code.

If done well, these annotations also express the intention of the developer who wrote the code. They are part of a Deliberate Practice.

In the past we have used the complete approach above to extract a reference business documentation that we directly sent to our customer abroad. A custom Doclet was exporting an Excel spreadsheet with one tab for each category of business domain concepts. The categories were simply based on the custom annotations added to the code.

An example please!

Ok, here's a brief example. The following code base represents a cat in all its state. Yes, I know it's a bit oversimplified:

```
module com.acme.catstate

// The activity the cat is doing. There are several activities.
@CoreConcept
interface CatActivity

  // How the cat changes its activity in response to an event
  @CoreBehavior
  @StateMachine
  CatState nextState(Event)

// The cat is sleeping with its two eyes closed
class Sleeping -|> CatActivity

// The cat is eating, or very close to the dish
class Eating -|> CatActivity

// The cat is actively chasing, eyes wide open
class Chasing -|> CatActivity

@CoreConcept
class Event // stuff happening that matters to the cat
  void apply(Object)

class Timestamp // technical boilerplate
```

This is just plain source code that describes the domain of the daily life of a cat. However it is augmented with annotations that highlight what's important in the domain.

A processor that builds a living glossary out of this code will print a glossary like the following:

```
Glossary
--------

CatActivity: The activity the cat is doing. There are several activities.
- Sleeping: The cat is sleeping with its two eyes closed
- Eating: The cat is eating, or very close to the dish
- Chasing: The cat is actively chasing, eyes wide open

nextState: How the cat changes its activity in response to an event

Event: Stuff happening that matters to the cat
```

Notice how the Timestamp class and the Event method have been ignored, because we they don't matter for the glossary. The classes that implement each particular activity have been presented together with the interface they implement, because that's the way we think about that particular construction.

By the way this is the State design pattern, and here it is genuinely part of the business domain.

Building the glossary out of the code is not an end to itself; from this first generated glossary we notice that the entry "nextState" is not so clear as we'd expect. This is more visible in the glossary than in the code. So we go back to the code and rename the method as "nextActivity()".

As soon as we rebuild the project, the glossary is updated, hence its name of Living Glossary:

```
Glossary
--------

CatActivity: The activity the cat is doing. There are several activities.
- Sleeping: The cat is sleeping with its two eyes closed
- Eating: The cat is eating, or very close to the dish
- Chasing: The cat is actively chasing, eyes wide open

nextActivity: How the cat changes its activity in response to an event

Event: Stuff happening that matters to the cat
```

Practical Implementation

Basically this technique needs a parser for your programming language, and the parser must not ignore the comments. In Java, there are many options like Antlr, JavaCC, Java Annotation processing API's and several other Open Source tools. However the simplest one is to go with a custom Doclet. That's the approach described here.

Even if you don't care about Java, you can still read on; what's important is largely language-agnostic.

In simple projects that cover only one domain, one single glossary is enough. The Doclet is given the root of the Javadoc metamodel and from this root it scans all programming elements like classes, interfaces and enums.

For each class the main question is: "does this matter to the business? Should it be included in the glossary?"

Using Java annotations answer a big part of this question. Each class with a "business meaningfull" annotation is a strong candidate for the glossary.

It is preferable to avoid strong coupling between the code that processes annotations and the annotations themselves. To avoid that, annotations can be recognized just by their prefix: "org.livingdocumentation.*", or by their unqualified name: "BusinessPolicy". Another approach is to check annotations that are themselves annotated by a meta-annotation like @LivingDocumentation. Again this meta-annotation can be recognized by simple name only to avoid direct coupling.

For each class to be included it then drills down the members of the class and prints what's of interest for the glossary, in a way that is appropriate for the glossary.

Information Curation

This selective showing and hiding and presentation concerns is not a detail. If it weren't for that the standard Javadoc would be enough. At the core of your Living Glossary there is all the editorial decisions on what to show, what to hide, and how to present the information in the most appropriate way. It's hard to do that outside if a context. I won't tell how to do it step by step. All I can do is give some examples:

Example of selective curation:

- An enum and its constants

- A bean and its direct non-transient fields
- An interface, its direct methods, and its main subclasses that are not technical, not abstract
- A value object and its methods that are "closed under operation", i.e. its methods that only accept as argument and return type the type itself, or primitives.

For a relevant glossary, a lot of details from the code usually have to be hidden:

- Ignore all methods from the super-object: toString(), equals() etc.
- Ignore all transient fields: they are there just for optimization purposes, they seldom mean anything for the business
- Ignore all constant fields, except perhaps the *public static final* of the type itself, if they represent important concepts of the business.
- Marker Interface don't need to list their subclasses, and the same may apply to interfaces with only one method.

The selective filtering depends to a large extent to the style of the code. If constants are *usually* used to hide technical literals then they should probably be mostly hidden, but if they are *usually* used in the public API then they may be of interest for the glossary.

Depending on the style of code, we will adjust the filtering so that it does most of the work by default, even if it goes too far in some cases. To supplement or derogate that default filtering we will use an override mechanism, for example by using annotations.

As a an example the selective filtering may ignore every method by default; we will have to define an annotation to distinguish the methods that should appear in the glossary. However I would never use an annotation named @Glossary, because it would be noise in the context of the code. A class or method is not meant to belong to a glossary or not, it is meant to represent a concept of the domain, or not. But a method can represent a core concept of the domain, and be annotated as such with a @CoreConcept annotation, that can be used to include the method in the glossary.

For more on Curation, please refer to the chapter on Knowledge Curation. For more on the proper usage of annotations to add meaning to the code, please refer to the chapter on Augmented Code.

Glossary by Bounded Context

Remember that a Ubiquitous Language can be defined with no ambiguity only within a given Bounded Context (See DDD Strategic Design).

If our source code spans several bounded contexts, then we need to segregate the glossary by Bounded Context. In order to do that, the bounded contexts must be explicitly declared.

Let's use annotations again to declare the bounded contexts, but this time the annotations will be on modules. In Java they will be package annotations, using the pseudo class *package-info.java*.

```
package-info.java

// Cats have many fascinating activities, and the way they switch from one to anothe\
r can be simulated by Markov chains.
@BoundedContext(name = "Cat Activity")
package com.acme.lolcat.domain
```

This is the first bounded context in our application, and we have another bounded context, again on cats, this time from a different perspective:

```
package-info.java

// Cats moods are always a mystery. Yet we can observe cats with a webcam and use im\
age processing to detect moods and classify them into mood families.
@BoundedContext(name = "Cat Mood")
package com.acme.catmood.domain
```

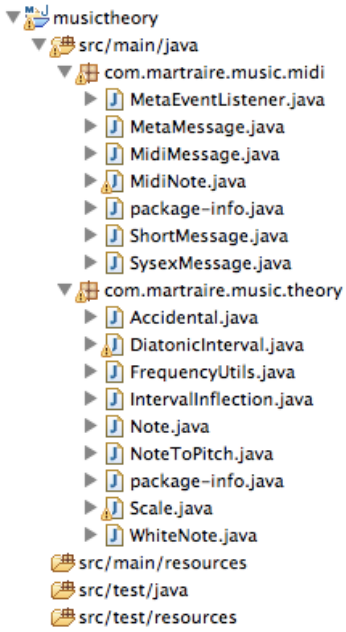
With several bounded contexts the processing is a bit more complicated, because there will be one glossary for each bounded context. We first need to inventory all the bounded contexts, then assign each element of the code to the corresponding glossary. If the code is well-structured, then the bounded context are clearly defined at the root of modules, so a class obviously belongs to a bounded context if it belongs to the module.

The processing becomes: 1. Scan all packages detect each context. 1. Create a glossary for each context. 1. Scan all classes; for each class find out the context it belongs to. This can simply be done from the qualified class name: ‘com.acme.catmood.domain.funny.Laughing’ that starts with the module qualified name: ‘com.acme.catmood.domain’. 1. Apply all the selective filtering and curation process described above for building a nice and relevant glossary, for each glossary. 1. This process can be enhanced to meet your taste. A glossary may be sorted by entry name, or sorted by decreasing importance of concepts.

Case Study

Let’s have a close look at a sample project on the domain of music theory and MIDI.

Here is what we see when we open the project in an IDE:



Tree view of the code base

There are two modules, each of one single package. Each module defines a Bounded Context. Here is the first one, that focuses on Western music theory:

```
/**
 * A representation of the theory of western music, from notes to chords, rhythm, harmony and melody.
 */
@BoundedContext(name = "Music Theory", link = "http://tobyrush.com/theorypages/index.html")
package com.martraire.music.theory;

import org.livingdocumentation.annotation.BoundedContext;
```

Declaration of the first bounded context as a package annotation

And here is the second bounded context, that focuses on MIDI:

```
/**
 * Represents the MIDI concepts necessary for composing and recording sequences of rhythms and melodies.
 */
@BoundedContext(name = "MIDI sequencing", domain = "MIDI", link = "https://docs.oracle.com/javase/tutorial/sound/
package com.martraire.music.midi;

import org.livingdocumentation.annotation.BoundedContext;
```

Declaration of the second bounded context as a package annotation

Inside the second context, here is an example of a simple value object with its Javadoc comment and its annotation:

```java
package com.martraire.music.midi;

import org.livingdocumentation.annotation.ValueObject;

/**
 * Any message defined by the MIDI specification and that is sent over the wire.
 * There are several kinds of MIDI messages.
 */
@ValueObject
public interface MidiMessage {

}
```

A value object with its annotation

And within the first context, here is an example of an enum, that is a value object as well, with its Javadoc comments, the Javadoc comments on its constants and the annotation:

```java
package com.martraire.music.theory;

import org.livingdocumentation.annotation.ValueObject;

/**
 * The accidentals alter the note by raising or lowering it by one or two half
 * steps.
 */
@ValueObject
public enum Accidental {

    /** (##) Lowered two half-steps */
    DOUBLE_SHARP("##"),
    /** (#) Lowered one half-step */
    SHARP("#"),
    /** No alteration */
    NATURAL(""),
    /** (b) Raised one half-step */
    FLAT("b"),
    /** (bb) Raised two half-steps */
    DOUBLE_FLAT("bb");

    private final String symbol;

    private Accidental(String symbol) {
        this.symbol = symbol;
    }

    public int halfSteps() {
```

An enum with its annotation

Note that there are other methods, but they will be ignored for the glossary.

Start with something, adjust manually

Now let's create the living glossary processor. We just create a custom Doclet that creates a text file and prints the glossary title in Markdown:

```
public class AnnotationDoclet extends Doclet {

  //...

        // doclet entry point
        public static boolean start(RootDoc root) {
                try {
                        writer = new PrintWriter("glossary.txt");
                        writer.println("# " + "Glossary");
                        process(root);
                        writer.close();
                } catch (FileNotFoundException e) {
                        //...
                }
                return true;
        }
```

What's left to implement is the method process(). It enumerates all classes from the doclet root, and for each class checks if it is meaningful for the business:

```
        public void process() {
                final ClassDoc[] classes = root.classes();
                for (ClassDoc clss : classes) {
                        if (isBusinessMeaningful(clss)) {
                                process(clss);
                        }
                }
        }
```

How do we check if a class is meaningful for the business? Here we do it only by annotation. We consider that all annotations from org.livingdocumentation.* mark the code as meaningful for the glossary. This is a gross simplification, but here it's enough.

```
        protected boolean isBusinessMeaningful(ProgramElementDoc doc) {
                final AnnotationDesc[] annotations = doc.annotations();
                for (AnnotationDesc annotation : annotations) {
                        if (isBusinessMeaningful(annotation.annotationType())) {
                                return true;
                        }
                }
                return false;
        }

        boolean isBusinessMeaningful(final AnnotationTypeDoc annotationType) {
                return annotationType.qualifiedTypeName().startsWith("org.livingdocumentation.anno\
tation.");
        }
```

If a class is meaningful, then we must print it in the glossary:

```
        protected void process(ClassDoc clss) {
                writer.println("");
                writer.println("## *" + clss.simpleTypeName() + "*");
                writer.println(clss.commentText());
                writer.println("");
                if (clss.isEnum()) {
                        for (FieldDoc field : clss.enumConstants()) {
                                printEnumConstant(field);
                        }
                        writer.println("");
                        for (MethodDoc method : clss.methods(false)) {
                                printMethod(method);
                        }
                } else if (clss.isInterface()) {
                        for (ClassDoc subClass : subclasses(clss)) {
                                printSubClass(subClass);
                        }
                } else {
                        for (FieldDoc field : clss.fields(false)) {
                                printField(field);
                        }
                        for (MethodDoc method : clss.methods(false)) {
                                printMethod(method);
                        }
                }
        }
```

Alright, this method is too big, but I want to show it all on one page.

The rest follows. Basically it's all about knowing the Doclet metamodel:

```
        private void printMethod(MethodDoc m) {
                if (!m.isPublic() || !hasComment(m)) {
                        return;
                }
                final String signature = m.name()  + m.flatSignature() + ": " + m.returnType().sim\
pleTypeName();
                writer.println("- " + signature + " " + m.commentText());
        }

        private boolean hasComment(ProgramElementDoc doc) {
                return doc.commentText().trim().length() > 0;
        }
```

You get the idea. The point is to have something working as soon as possible, to get the feedback on the glossary generator (Doclet) itself, and on the code itself as well. Then it's all about iterating: change the code of the glossary generator to improve the rendering of the glossary and to improve the relevance of its selective filtering; change the actual code of the project so that it is more expressive, add annotations, create new annotations if needed, so that the code itself tells the whole business domain knowledge. This cycle of iterations should not absorb a lot of time, however it never really finishes, it does not have an end state, it's a living process. There is always something to improve, in the glossary generator or in the code of the project.

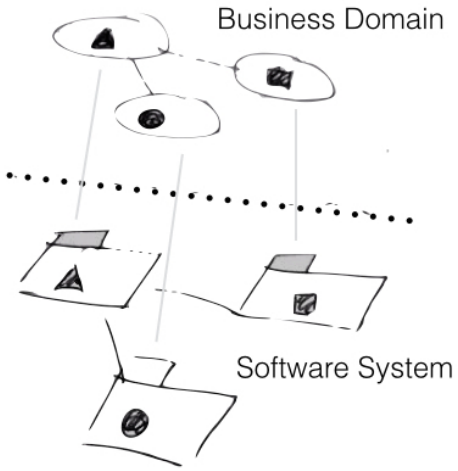
A living glossary is not a goal in itself. It's above all a process that help the team reflect over its code, to improve its quality along the road.

Case Study: Business Overview as a Living Diagram

The idea

We work for an online shop that was launched a few years ago. The software system for this online shop is a complete e-commerce system made of several components. This system has to deal with everything necessary for selling on-line, from the catalogue and navigation to the shopping cart, the shipping and some basic customer relationship management.

We're lucky because the founding technical team had good design skills. As a result, the components match the business domains in a one-to-one fashion, in other words the software architecture is well aligned with the business it supports.



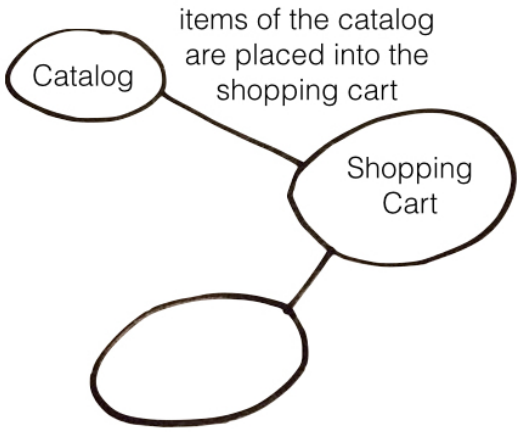
Software Components match one-to-one the business domains

Because of its success, our online shop is growing quickly. As a result there are an increasing number of new needs to support, which in turn means there are more features to add to the components. Because of this growth we'll probably have to add new components, redo some components, and split or merge existing components into new components that are easier to maintain, evolve and test.

We also need to hire new people in the development teams. As part of the necessary knowledge transmission for the new joiners we want some documentation, starting with an overview of the main business areas, or domains, supported by the system.

We could do that manually, and it would take a couple of hours in PowerPoint or in some dedicated diagramming tool. But we want to trust our documentation, and we know we'll likely forget to update the manually created document whenever the system changes. And we know it will change.

Fortunately after we've read the book on Living Documentation we decided to automatically generate the desired diagrams from the source code. We don't want to spend time on a manual layout, a layout based on the relationships between the domains will be perfectly fine. Something like this:

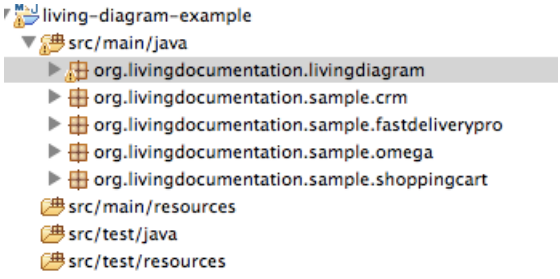


Expected style of diagram

Practical Implementation

The existing source code

Our system is made of components that are simply Java packages:



Overview of the components as Java packages

The naming of these packages is a bit inconsistent, because historically the components were named after the development project code, as it is often the case. For example the code taking care of the shipping features is named "Fast Delivery Pro", because that's how the marketing team used to

name the automated shipping initiative 2 years ago. Now this name is not used anymore, except as a package name.

Similarly, "Omega" is actually the component taking care of the catalog and the current navigation features.

We have a naming problem, that is also a documentation problem: the code does not tell the business. For some reasons we can't rename the packages right now, though we hope to do it next year. Yet even with the right names, the packages won't tell the relationships between them.

Augmenting the code

As a result, we need extra information in order to make a useful diagram. As we've seen before, one great way to add knowledge to the code is to use annotations. At a minimum, we want to add the following knowledge to the code, to fix the naming:

```
@BusinessDomain("Shipping")
org.livingdocumentation.sample.fastdeliverypro

@BusinessDomain("Catalog & Navigation")
org.livingdocumentation.sample.omega
```

For that purpose we introduce a custom annotation with just a name:

Introducing a custom annotation to declare a business domain

```
@Target({ ElementType.PACKAGE })
@Retention(RetentionPolicy.RUNTIME)
public @interface BusinessDomain {
        String value(); // the domain name
}
```

Now we'd like to express the relationships between the domains. Basically:

- The items of the catalog are placed into the shopping cart, before they are ordered.
- Then the items in orders must be shipped,
- And these items are also analyzed statistically to inform the customer relationship management.

We then extend the annotation with a list of related domains. However as soon as we refer to the same name several times, text names raise a little problem: if we are to change one name, then we must change it everywhere it is mentioned.

To remedy that we want to factor out each name into one single place to be referenced. One possibility is to use enumerated types instead of text. We then make references to the constants

of the enumerated type. If we rename one constant, we'll have nothing special to do to update its references everywhere.

And since we also want to tell the story for each link, we add a text description for the link as well.

```
public @interface BusinessDomain {
        Domain value();
        String link() default "";
        Domain[] related() default {};
}

// The enumerated type that declares in one single place each domain
public enum Domain {
  CATALOG("Catalog & Navigation"),
  SHOPPING("Shopping Cart"),
  SHIPPING("Shipping"), CRM("CRM");

        private String value;

        private Domain(String value) {
                this.value = value;
        }

        public String getFullName() {
                return value;
        }
}
```

Now it's just a matter of using the annotations on each package to explicitly add all the knowledge that was missing from the code.

```
@BusinessDomain(value = Domain.CRM, link = "Past orders are used in statistical anal\
ysis for customer relationship management", related = { Domain.SHOPPING }))
org.livingdocumentation.sample.crm

@BusinessDomain(value = Domain.SHIPPING, link = "Items in orders are shipped to the \
shipping address", related = { Domain.SHOPPING })
org.livingdocumentation.sample.fastdeliverypro

//etc.
```

Generation of the Living Diagram

Because we need a fully automatic layout that works like magic in all cases, we decide to use the tool Graphviz for the layout and rendering of the diagram. This tool expects a text file with a *.dot* extention that conforms to the dot syntax. We need to create this plain text file before running Graphviz to render it into a regular image file.

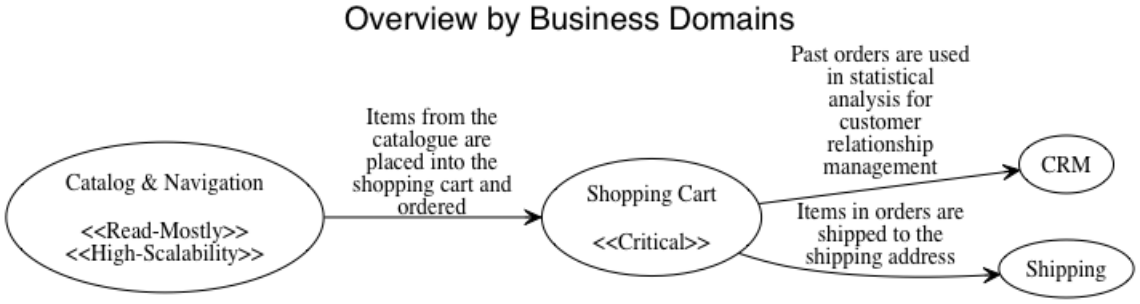
Basically the generation process is made of the following steps:

1. Scan the source code, or the class files, to collect the annotated packages and their annotation information.
2. For each annotated package, add an entry into the dot file:
    - To add a node that represents the module itself
    - To add a link to each related node
3. Save the dot file
4. Run Graphviz *dot* in command-line by passing it the .dot filename and the desired options to generate an image file.
5. We're done! The image is ready on disk.

The code to do that can fit inside one single class of less than 170 lines of code. Because we're in Java, most of this code size is about dealing with files, and the hardest part of it is about scanning the Java source code.

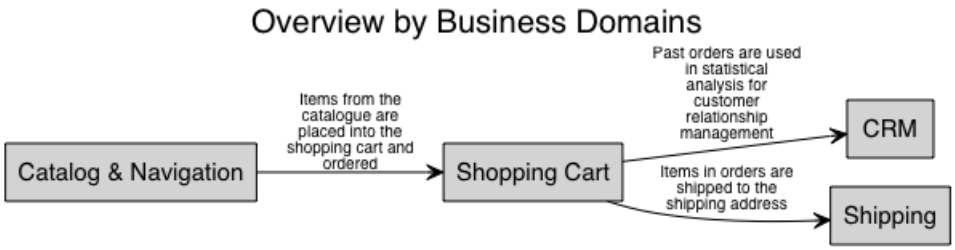
You will find the complete code in addendum.

After running Graphviz we get the following Living Diagram:



Actual diagram generated from the source code

And after adding some additional style information:



Actual diagram generated from the source code, with style

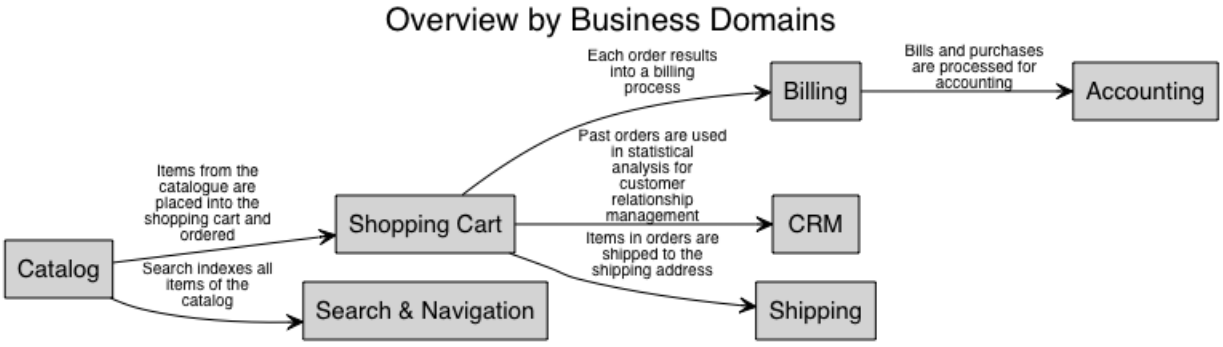
Some time later

The business has grown and the supporting software system had to grow as well. Several new components have appeared, some brand new, some as a result of spliting former components.

For example now we have dedicated components for the following business domains:

- Search & Navigation
- Billing
- Accounting

Each new component has its own package, and had to declare its knowledge in its package annotation, like any well-behaved component. Then, without any additional effort, our living diagram will automatically adapt to the new, more complicated overview diagram:



Actual diagram generated from the source code, with style

Adding other information

Now we'd like to enrich the diagram with concerns like Quality Attributes.

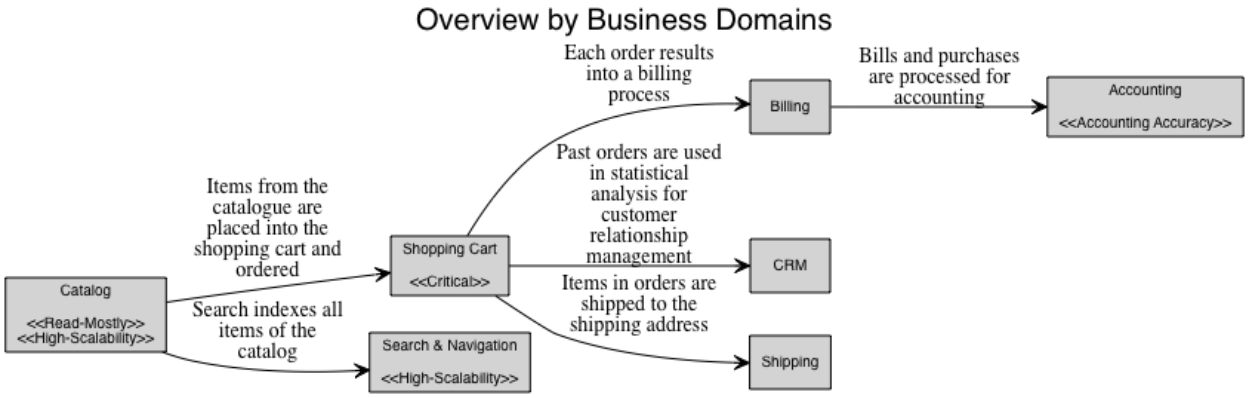
As usual, since this knowledge is missing from the code we need to add it by augmenting the code. Again we'll use package annotations for that:

```
@Concern({ HIGH_SCALABILITY })
@BusinessDomain(value = SEARCH, link = "Search indexes all items of the catalog", upstream = { CATALOG })
package org.livingdocumentation.sample.search;

import static org.livingdocumentation.livingdiagram.Domain.CATALOG;
```

Package annotations in package-info.java

We can now enhance the Living Diagram processor to extract the *@Concern* information as well in order to include them into the diagram. Once done we get the following diagram, obviously a little less clear:



Actual diagram generated from the source code, with additional quality attributes

This is just an exemple of what's possible with a Living Diagram. The only limit is your imagination and the time required to try many ideas that don't always work. However it's worth the try to play with these ideas from time to time, or whenever there's a frustration about the documentation, or about the design. A Living Documentation makes your code, its design and its architecture transparent for everyone to see. If you don't like what you see, then fix it in the source code.

How does this Living Diagram fit with the patterns of Living Documentation?

This diagram is a **Living Document**, refreshed whenever the system changes, automatically. If we were to add or delete a new module, the diagram would adjust as quickly as the next build. It is also an example of a **Plain-Text Diagram**; if we just want to change a word in a sentence, we can do it simply by editing the source code. No need to fire PowerPoint or a diagram editor.

It's a **Story-Telling Diagram**, that tells a story from on node to the next through links that display a brief description.

This diagram is an example of **Augmented Code** by using annotations to augment each main module with the knowledge of its corresponding business domain. This is also a case of **Consolidation** of information spread across many packages.

Finally the knowledge added to the source code can be used for an **Enforced Architecture**. Writing a verifier is similar to writing a living diagram generator, except that the relationships between nodes are used as a dependency whitelist to detect unexpected dependencies, instead of generating a diagram.

Part 8 No Documentation

‘#NoDocumentation’ is a manifesto for exploring alternatives to traditional forms of documentation.

> We acknowledge the purpose of documentation, but we disagree with the way it’s usually done. #NoDocumentation is about exploring better alternatives for transferring knowledge between people and across time.

No Documentation!

Documentation is only a mean, not an end. It’s only a tool, not a product.

Here are some ideas under the #NoDocumentation tag:

- Conversations over Documentation
- Continuous Training
- All forms of collective work: Pair-Programming, Mob Programming, Specification Workshops, Communities of Practice
- Declarative Automation
- Shameful Documentation
- Enforced Decisions
- Make It Easy to Do the Right Thing
- Making mistakes impossible

- Scaffolding
- On-demand Documentation
- Throw-Away Documentation

‘#NoDocumentation’: Make documentation redundant!

> We embrace documentation, but not hundreds of pages of never-maintained and rarely-used tomes. #agilemanifesto – @sgranese on Twitter

Part 9 Beyond Documentation

Hygienic Transparency

Transparency leads to higher hygiene, because the dirt cannot hide.

Internal quality is the quality of the code, design and more generally of all the process from the nebulous needs to working software that delights people. Internal quality is not meant to satisfy ego or to be pride of it, by definition it is meant to be economic beyond the short term. It is desirable to save money and time sustainably, week after week, year after year.

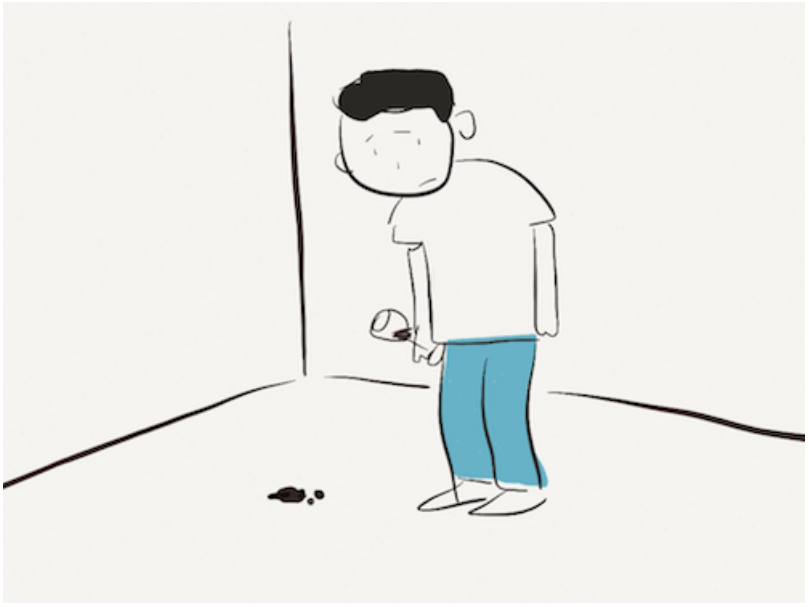
The problem with internal quality is that it's internal, hence you can't see it from the outside. That's why so many software systems are awful in the inside, provided you have developers eyes. Non developers like managers and customers can hardly appreciate how bad the code is inside. The only hints for them are the frequency of defects and the feeling that it gets slower and slower to deliver new features.

Everything that improves the transparency of how the software is made helps improve its internal quality. Once people can see the ugliness inside, there's a pressure to fix it.

Therefore: Make the internal quality of the software as visible as possible to anyone, developers and non-developers alike. Use living documents, living diagrams, code metrics and other means to expose the internal quality in a way that everyone can appreciate, without any particular skill.

Use all this material to trigger conversations and as a support to explain how things are, why they are this way, and to suggest improvements. Make sure the living documents and other techniques look better when the code gets better.

Note that the techniques that help make the software more transparent can't prove the internal quality is good, however they can highlight when it is bad, and that's enough to be useful.



In a house with everything painted in white, dirt is immediately visible

Le Corbusier and The Law of Ripolin

Le Corbusier in his book intitled "The Decorative Art of Today" explains his fascination for the Ripolin, a brand of paint famous for his white paint. In the chapter titled: "A coat of Whitewash: The Law of Ripolin", he imagines every citizen being required to replace everything with a plain coat of ripolin white paint:

"*His home* is made clean. There are no more dirty, dark corners. Everything is shown as it is. Then comes *inner* cleanness [...] once you have put ripolin on your walls you will be the master of *your own house*."

Good documentation should have a similar effect on the inner cleanness of the code, its design and basically any other aspect that becomes visible for everyone to see its dirty facets.

Diagnosis tools

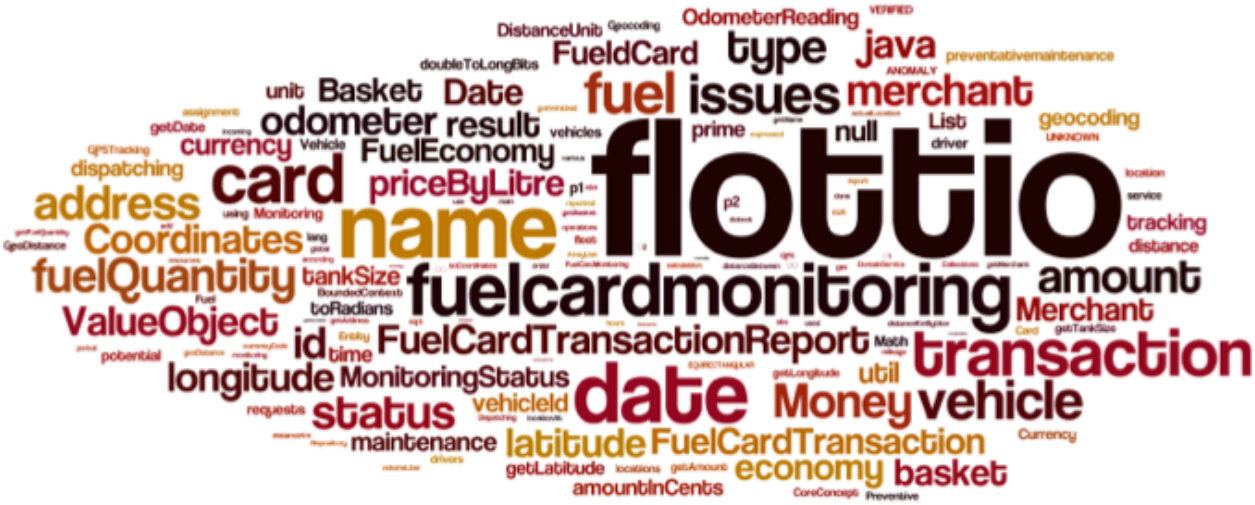
The border is very thin between typical documentation media like diagrams and glossaries, and diagnosis tools like metrics or word clouds.

Word Clouds

Word Clouds are very simple diagrams that only show words with more frequent words shown in a bigger font than less frequent words. One way to quickly assert what your application is really talking about is to generate a **word cloud** out of the source code.

What does this word cloud really tell you about your code? If technical words dominate, then it's factual that the code does not really talk about the business domain.

With this word cloud, either your business domain is on string manipulation, or it it not visible in the sourc code



In this word cloud, we clearly see the language of Fuel Cards and fleet management (Flottio is the package name, it could be filtered too)

Creating a word cloud out of the source code is not hard as you don't even have to parse the source code, you can simply consider it as plain text and filter the programming language keywords and punctuation. Something like:

```
// From the root folder of the source code, walk recursively through all *.java file\
s (respectively *.cs files in C#)

// For each file read as a string, split by the language separators (you may conside\
r to split by CamelCase too):

SEPARATORS = ";:.,?!<><=+-^&|*/\"\t\r\n {}[]()"

// Ignore numbers and tokens starting with '@', or that are keywords and stopwords f\
or the programming language:

KEYWORDS = { "abstract", "continue", "for", "new", "switch", "assert", "default", "i\
f", "package", "synchronized", "boolean", "do", "goto", "private", "this", "break", \
"double", "implements", "protected", "throw", "byte", "else", "import", "public", "t\
hrows", "case", "enum", "instanceof", "return", "transient", "catch", "extends", "in\
t", "", "short", "try", "char", "final", "interface", "static", "void", "class", "fi\
nally", "long", "strictfp", "volatile", "const", "float", "native", "super", "while"\
 }

STOPWORDS = { "the", "it","is", "to", "with", "what's", "by", "or", "and", "both", "\
be", "of", "in", "obj", "string", "hashcode", "equals", "other", "tostring", "false"\
, "true", "object", "annotations" }
```

At this point you could just print every token that was not filtered, and copy-paste the console into an online Word Cloud generator like Wordle.com.

You may as well count the occurrences yourself, using a bag (e.g. a Multiset from Guava):

```
bag.add(token)
```

And you could render the word cloud within an html page with the *d3.layout.cloud.js* library, by dumping the word data into the page.

Another similar low-tech idea to visualize the design of the code out of the source code is the "signature survey"[10] proposed by Ward Cunningham. The idea is to filter out everything but the language punctuation (coma, quote and brackets), as pure string manipulation on the source code files.

For example, contrast this signature survey, with 3 big classes:

[10] http://c2.com/doc/SignatureSurvey/

```
BillMain.java ;;;;;;;;;;;;;;{;;{"";;"";;{"";"";}{;;{;;}};;;;{{;;;{;;}{;;};;;}}{;}"";}\
{;}{;;"";"";;;"";"";;;"";"";";;";;;"";"";;;"";"";;;};;{;{"";{;}"";{;}"";{;}"";{;}"";;;;\
;;}"";{;}"";{;}"";{;};"";;;;;}"";;;;;}};}{;;;;;"";"";;;}"";"";;"";{;}}"";"";;"\
";{;}};{;}"";{;}{;};;;;;}{;;;;;;}{;;;;;;}{;;;;;;}{;"";{;}{;}{;};;};}{;"";;}{;"""";}{"";\
};;{;}{"";};;}{{;};";";;;{"";{"";;};}}{{;;;}}{;};}{;{;}";";;;{"";{"";;};}}{;;{"";{"";}"\
";{;}{;}}}};{;;;}{"";;;;;;;;}}{;{;}{;};}{;"";{;};{;{"";};}{{"";};};};}{;;;;;;;;;{{"\
";};;}{{"";};;;};}{;;;;;;;;;{;;;{"";}{{"";};}{{"";};;};};}{;;"";{;}{;};}{;;{"";{"";}{"\
";};}{;}{;{;}{{;}{;}}};}}
CallsImporter.java ;;;;;;;{;;{{"";};{;;"";;;{;}{;;{;};};{;"";{;;};;{;;{;};}{;}{;};}}\
{;}{{{;}{;}}}}{"";{;}"";{;}"";{;}"";{;}"";{;}"";{;}"";{;}"";{;}"";{;}"";{;}"";{;}"";{;}"";{;\
}"";{;}"";{;}"";{;}"";{;}"";{;}"";{;}"";{;}"";{;}"";{;}"";{;}"";{;}"";{;}"";{;}"";{;}"";{;}"";{\
;}"";{;};}}
UserContract.java ;;{;;;;{;}{;}{;}{;}{;}{;}{;}{;}{;}{;}{;}}
```

with this one, doing the exact same thing, but with more smaller classes:

```
AllContracts.java ;;;;;{;{;}{{;}}{"""";}}
BillingService.java ;;;;;;;{;{"";}{;;;;;}{"";;}{;;"";}{;}{;}{;}{"";;;;;}{"";;}{;;{{;;}}\
;}{;}}
BillPlusMain.java ;;;;;;{{;"";"";"";"";"";"";}}
Config.java ;;;;;;;{;{;{"";;}{;}{{{;}{;}}}}{;}{;;}{"";}{"";}{"";}{"";}{"";;}{;";";{;\
};}}
Contract.java ;;;;{;;;;{;;;;}{;}{;}{;}{;}{;}{"""""""""";}}
ContractCode.java ;{"""""""""""""""""""""""";;{;}{;}}
ImportUserConsumption.java ;;;;;{;;{;;}{{;}{;}}{;;}}{;"";;;;{;};}{{;}{;}}{"";;;{;{\
;}};}}
OptionCode.java ;{""""""";;{;}{;}}
Payment.java ;;;{;;;{;;;{"";}}{;}{;}{;}{;}{{;}"";}{;}{;;;;;}{;}{"""""";}}
PaymentScheduling.java ;;;;{{{;;;}}{{;;;}}{{;;;}};{;;;;}{;;{;};;}{;;;;;;;;}{;}}
PaymentSequence.java ;;;;;;{;;{;}{;;}{;}{;}{;;;}{"";}}
UserConsumption.java ;{;;{;;}{;}{;}}
UserConsumptionTracking.java ;{{;}{;}}
```

Which one do you prefer to work on?

It's probably possible to imagine similarly low-tech yet useful plain text visualization approaches. Let me know if you have any idea.

A Positive Pressure to clean the inside.

One huge issues in our field of software development is that internal quality is not visible at all for the people who manage budgets and made the biggest decisions, like saying yes or no to

developers, contracting to another company or offshoring. This does not help make good informed decisions. Instead it promotes decisions from people who are more convincing and seductive in their arguments.

Developers can become more convincing too when they can show the internal quality of the code in a way non-technical people can apprehend emotionally. A word cloud, or a dependency diagram that is a total mess, is easy to interpret even by non-developers. Once they understand by themselves the problem shown visually, it becomes easier to talk about remediation.

Developers opinions are often suspect to managers. In contrast, they appreciate the output of tools, because tools are neutral and objective, or at least they believe that. Tools are definitely not objective, but they do present actual facts, even if the presentation always carry an editorial bias.

As such, the ideas behind living documentation are not just to document for the team, they are also part of the toolbox to convince. Once everybody sees the disturbing reality, the mess, the cyclic dependencies, the unbearable coupling, the obscurity of the code, it becomes harder to tolerate that much longer.

LOL

To respect the Acyclic Dependencies Principle : have only one package!

Living documentation suggests to make the internal problems of the code visible to everyone, as a positive pressure to encourage cleaning the internal quality.