# 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 2017-03-28

Esse é um livro Leanpub. A Leanpub dá poderes aos autores e editores a partir do processo de Publicação Lean. Publicação Lean é a ação de publicar um ebook em desenvolvimento com ferramentas leves e muitas iterações para conseguir feedbacks dos leitores, pivotar até que você tenha o livro ideal e então conseguir tração.

© 2014 - 2017 Cyrille Martraire

# 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:

https://twitter.com/search?q=#LivingDocumentation

# Contents

**Note to reviewers** . . . . . **i**

**A simple question** . . . . . **ii**
Are you happy with the documentation you create? . . . . . ii

**Preface** . . . . . **iii**

**About this book** . . . . . **iv**

**Acknowledgements** . . . . . **v**

**How to read this book?** . . . . . **vii**

## Part 1 Reconsidering Documentation . . . . . 1

**A tale from the land of Living Documentation** . . . . . **2**
Why this feature? . . . . . 2
Tomorrow you won't need this sketch anymore . . . . . 2
Sorry, we don't have marketing documents! . . . . . 3
You keep using this word, but this is not what it means . . . . . 3
Show me the big picture, and you'll see what's wrong there . . . . . 4
The future of Living Documentation is now . . . . . 4

**The problem with traditional documentation** . . . . . **5**
Documentation is not cool . . . . . 5
The flaws of documentation . . . . . 6
The agile manifesto and documentation . . . . . 12
It's time for Documentation 2.0 . . . . . 13

**It's all about knowledge** . . . . . **15**

**Knowledge origination** . . . . . **16**
How does that knowledge evolve? . . . . . 16

**The reasons why knowledge is necessary** . . . . . **17**
And now for a pedantic word: Stigmergy . . . . . 18

**Software Programming as Theory Building and Passing** . . . . . **20**

**Documentation is about transferring knowledge** . . . . . **22**
Choosing the right media . . . . . 23

**Specific vs. Generic Knowledge** . . . . . **24**
Generic Knowledge . . . . . 24
Learn generic knowledge . . . . . 24
Focus on Specific Knowledge . . . . . 24

**Knowledge is already there** . . . . . **26**

**Internal Documentation** . . . . . **27**
Internal vs. External documentation . . . . . 28
Examples . . . . . 29
Choosing between external or internal documentation . . . . . 30
In situ Documentation . . . . . 30
Machine-readable documentation . . . . . 30

**Accuracy Mechanism** . . . . . **32**
Documentation by Design . . . . . 32
Accuracy Mechanism for a reliable documentation for fast-changing projects . . . . . 32

**The Documentation Checklist** . . . . . **35**
Questioning the need for documentation at all . . . . . 35
Need for documentation because lack of trust . . . . . 36
Just-In-Time Documentation, or Cheap Option on Future Knowledge . . . . . 36
Questioning the need for traditional documentation . . . . . 37
Minimizing the extra work now . . . . . 38
Minimizing the extra work later . . . . . 39
Living Documentation - The Very Short Version . . . . . 40

**Documentation Reboot** . . . . . **42**
Approaches to better documentation . . . . . 42
No Documentation . . . . . 43
Stable Documentation . . . . . 43
Refactor-Friendly Documentation . . . . . 44
Automated Documentation . . . . . 44
Runtime Documentation . . . . . 44
Beyond Documentation . . . . . 44

**Core Principles of Living Documentation** . . . . . **45**
Reliable . . . . . 47
Low-Effort . . . . . 47
Collaborative . . . . . 47

Insightful . . . 48

**A Gateway Drug to DDD** . . . **49**
Domain-Driven Design in a nutshell . . . 49
Living Documentation and Domain-Driven Design . . . 50
When Living Documentation is an application of DDD . . . 51

**A principled approach** . . . **54**
Need . . . 54
Values . . . 54
Principles . . . 54
Practices . . . 55
Tools . . . 55

**Fun** . . . **56**

# Part 2 Living Documentation exemplified by Behavior-Driven Development . . . 58

**A key example of Living Documentation: BDD** . . . **59**
BDD is all about conversations . . . 59
BDD with automation is all about Living Documentation . . . 60
Redundancy + Reconciliation . . . 60
Anatomy of the scenarios in file . . . 62
Interactive Living Documentation . . . 66
Feature File: another Example . . . 67
How does it work? . . . 68
A canonical case of Living Documentation in every aspect . . . 69
Going further: Getting the best of your living documentation . . . 70

**Description** . . . **72**
Example . . . 72
Scenario: Present Value of a single cash amount . . . 72
No guarantee of correctness . . . 72
Property-Based Testing and BDD . . . 73
Manual glossary . . . 73
Linking to non-functional knowledge . . . 74

# Part 3 Knowledge Exploitation & Augmentation . . . 75

**Knowledge Exploitation** . . . **76**
Identify authoritative knowledge . . . 76

Where is this knowledge now? . . . 76

**Single Sourcing with a Publishing Mechanism (aka Single Source Publishing) . . . 78**
Some examples of producing a published document . . . 79
Published Snapshot with a version number . . . 81
Remarks . . . 82

**Reconciliation Mechanism (aka Verification Mechanism) . . . 83**
Consistency Tests . . . 84
Reconciliation on the test assumptions . . . 85
Published Contracts . . . 86

**Information Consolidation . . . 88**
How it works . . . 89
Implementation remarks . . . 90

**Ready-Made Documentation . . . 92**
The power of a standard vocabulary . . . 93
Link to standard knowledge . . . 95
Searching for the reference . . . 95
More than just vocabulary . . . 95

**Ready-made knowledge in conversation to speed up knowledge transfer . . . 97**

**Tools History . . . 102**

**Augmented Code . . . 104**
When programming languages are not enough . . . 104

**Documentation by Annotations . . . 106**
Annotations are more than tags . . . 107
A good place to describe the rationale behind the decisions . . . 108
Annotations for learning on the job . . . 109

**Documentation by Convention . . . 113**
Legacy with conventions has opportunities for Living Documentation . . . 113
Document the Conventions . . . 114
Discipline-Driven . . . 115
Conventions have limitations . . . 115

**Sidecar files . . . 116**

**Metadata Database . . . 117**

**Designing Custom Annotations . . . 118**
Stereotypical Properties . . . 118

Stereotypes and tactical patterns . . . 119
Meaningful annotations package names . . . 120
Hijacking standard annotations . . . 121
Standard Annotation @Aspect and Aspect-Oriented Programming . . . 122
Annotation by default or unless necessary . . . 123

**Module-Wide Knowledge . . . 124**
Many kinds of modules . . . 124
In practice . . . 125

**Intrinsic Knowledge Augmentation . . . 126**

**Inspiring Exemplars . . . 128**

**Machine Accessible Documentation . . . 132**

**Literate programming . . . 133**
Other similar approaches . . . 134

**Record Your Rationale . . . 135**
What's in a rationale? . . . 135
Make it explicit . . . 136
Beyond Documentation: Motivated Design . . . 137
Don't document speculation . . . 137
Skills as pre-documented rationales . . . 138
Recording the rationale as an enabler for change . . . 138

**Commit Messages as Comprehensive Documentation . . . 140**
Commit Guidelines . . . 141
Scope of the change . . . 143
Machine-Accessible information . . . 143

**Dynamic Curation . . . 146**
Examples . . . 147
Editorial curation . . . 148
Low-maintenance dynamic curation . . . 148
One corpus of knowledge for multiple uses . . . 149
Scenario Digest . . . 149

**Highlighted Core (Eric Evans) . . . 152**

**Guided Tour, Sightseeing Map . . . 154**
A sightseeing map . . . 155
An example in Java . . . 156
A Guided Tour Example . . . 158

Living Guided Tour . . . 159
The implementation . . . 161
Related . . . 161

# Part 4 Automated Documentation . . . 163

**Living Document . . . 164**
Anatomy of a living document. . . . 164
Presentation Rules . . . 164

**Living Glossary . . . 165**
How it works . . . 166
An example please! . . . 167
Practical Implementation . . . 169
Information Curation . . . 169
Glossary by Bounded Context . . . 170
Case Study . . . 171

**Living Diagram . . . 177**
Diagrams help conversations . . . 177
Editorial Perspective . . . 178
Living Diagram to keep you honest . . . 181
The quest for the perfect diagram . . . 181
Rendering . . . 183
Visualization guidelines: . . . 186

**Hexagonal Architecture Living Diagram . . . 187**
The idea . . . 187
The architecture is already documented . . . 188
The architecture is already in the code . . . 188
Creating the living diagram . . . 189
Possible Evolutions . . . 192

**Case Study: Business Overview as a Living Diagram . . . 193**
The idea . . . 193
Practical Implementation . . . 194
How does this Living Diagram fit with the patterns of Living Documentation? . . . 199

**Living Services Diagram . . . 201**
A matter of Augmented Code, but at runtime . . . 201
Discover the architecture . . . 203
The magic to make this work . . . 203
Going further . . . 203

**Context Diagram** . . . . . **204**
Hyperlinks to the corresponding source code location . . . . . 206
Augmented Code & Knowledge Consolidation . . . . . 206
Limitations & Benefits . . . . . 208

**Domain-Specific Diagrams (aka Visible Tests)** . . . . . **209**
Domain-Specific Notations . . . . . 209
Generating custom domain-specific diagrams to get visual feedback . . . . . 212
A complement to Gherkin scenarios? . . . . . 213
Pattern-Oriented Living Diagram . . . . . 213

# Part 5 Runtime Documentation . . . . . 215

**Visible Workings: Working Software as its own Documentation** . . . . . **216**
Working software as documentation . . . . . 216
Visible Workings . . . . . 216

**Introspectable Workings** . . . . . **218**
Introspecting by reflection . . . . . 219
Introspecting without reflection . . . . . 221

# Part 6 Refactoring-Friendly Documentation . . . . . 223

**Plain-Text Artifacts** . . . . . **224**

**Source Control is the Reference** . . . . . **226**

**Plain-Text Diagram** . . . . . **227**
An example . . . . . 227
Diagram as Code . . . . . 230

**Code is Documentation** . . . . . **232**
Text Layout . . . . . 232
Coding Conventions . . . . . 235

**Integrated Documentation** . . . . . **237**
Type Hierarchy . . . . . 237
Code search . . . . . 237
Semantics derived from the actual usage . . . . . 238

**Naming As The Primary Documentation** . . . . . **239**
Naming: browsing a thesaurus . . . . . 239
Composed methods, you need to name them. . . . . 239

**Type-Driven Documentation** . . . . . **240**
Types and Associations . . . . . 242
Types over Comments . . . . . 242

**Composed Method (Kent Beck)** . . . . . **245**

**Fluent Style** . . . . . **248**
An Internal DSL . . . . . 248
Implementing a Fluent Interface . . . . . 249
Fluent Tests . . . . . 250
Test DSL aka DSTL . . . . . 250
Fluent All the Things! Not. . . . . 251

**Case Study: An example of refactoring the code, guided by comments** . . . . . **252**

**Living Documentation with Event Sourcing tests** . . . . . **255**
A concrete example in code . . . . . 256
How does it work? . . . . . 257
Living Diagrams from Event Sourcing scenarios . . . . . 257

# Part 7 Stable Documentation . . . . . 260

**Evergreen Document** . . . . . **261**
Design Vs Requirements . . . . . 261
Examples . . . . . 262
A lot of knowledge is less stable than it looks. . . . . 262
Case Study: a README file . . . . . 262
Evergreen README . . . . . 265
In closing . . . . . 266

**Don't Mix Strategy Documentation with the documentation of its implementation** . . . **267**

**Vision Statement** . . . . . **269**
Domain Vision Statement . . . . . 270
Goals . . . . . 271
Impact Mapping . . . . . 271
Traceability to the goals . . . . . 272
Stable Knowledge can also be code . . . . . 273

**Perennial Naming** . . . . . **274**
Organizing artifacts along stable axes . . . . . 274
Stability-Oriented . . . . . 275

**Knowledge Network** . . . . . **276**

Linkable Knowledge . . . 276
Volatile To Stable . . . 276
Link Registry . . . 277
Bookmarked Search . . . 277
Contract Test as a Reconciliation Mechanism . . . 279

**Acknowledge your influences . . . 280**
Project Bibliography . . . 280
Declare your Style . . . 280

**Domain Immersion . . . 282**
Investigation Wall . . . 282
Domain Training . . . 283
Live-my-Life Sessions . . . 283
Shadow User . . . 283
A long-term investment . . . 283

# Part 8 No Documentation . . . 285

**Conversations Over Formal Documentation . . . 287**
Wiio's laws . . . 289
The Rule of Three Interpretations . . . 290
Obstacles to conversations . . . 290
Knowledge Transfer sessions . . . 292
Interactive Documentation . . . 292

**Working Collectively . . . 294**
Pair-Programming . . . 294
Cross-Programming . . . 295
Mob-programming . . . 295
Event Storming as an on-boarding process . . . 296
Continuous Documentation . . . 297
Truck Factor . . . 297

**Coffee Machine Communication . . . 298**
Ideas Sedimentation . . . 300
Throw-Away Documentation . . . 302

**On-Demand Documentation . . . 303**
Just-In-Time Documentation . . . 303
Provoking Just-In-Time Learning Early . . . 304
Astonishment Report . . . 305
Some Upfront Documentation? . . . 305

**Declarative Automation** . . . . . **308**
Declarative Dependency Management . . . . . 309
Declarative Configuration Management . . . . . 311
Declarative Automated Deployment . . . . . 314
Machines Documentation . . . . . 317
Remarks on automation in general . . . . . 317

**Enforced Guidelines** . . . . . **319**
Some examples of rules . . . . . 320
Evolving the guidelines . . . . . 321
Enforcement or Encouragement . . . . . 322
Declarative Guidelines . . . . . 322
A matter of tools . . . . . 323
Guidelines, or Design Documentation? . . . . . 324
Warranty Sticker Void If Tampered . . . . . 324
Trust-First . . . . . 325

**Shameful Documentation** . . . . . **326**
An example . . . . . 327
The Troubleshooting Guide . . . . . 327
Don't tolerate the documented pain . . . . . 328
Shameful code documentation . . . . . 329

**Don't document, influence or constraint behavior instead!** . . . . . **330**
Make It Easy to Do the Right Thing . . . . . 330
Making Mistakes Impossible - Error-Proof API . . . . . 331

**Design principles for documentation avoidance** . . . . . **333**
Replaceability-First . . . . . 333
Consistency-First . . . . . 334

**Zero documentation & Gamification** . . . . . **335**
Continuous Training . . . . . 335

# Part 9 Beyond Documentation . . . . . 337

**Documentation-Driven Development** . . . . . **338**
Documentation to keep you honest . . . . . 339
The apparent contradiction between Documentation-Driven and "NoDocumentation" . . . . . 339

**Abusing Living Documentation** . . . . . **341**
Living Documentation with Moderation . . . . . 341
Procrastinate on Living Documentation . . . . . 342

**Listen To The Documentation** . . . . . **343**
What happened to the Language of the Domain? . . . . . 344
Programming by coincidence Design . . . . . 344
Deliberate Decisions . . . . . 346
Deliberate decision does not mean upfront decision . . . . . 348
Documentation is a form of code review . . . . . 348

**Biodegradable Documentation** . . . . . **349**

**Hygienic Transparency** . . . . . **352**
Diagnosis tools . . . . . 353
A Positive Pressure to clean the inside. . . . . 356

**Living Documentation Going Wild** . . . . . **358**

**BREAKING!!! Live Interview: Mrs Reporter Porter interviewing Mr Living Doc Doctor!** . **359**

# Part 10 Living Design & Architecture Documentation . . . . . 362

**Living Design & Architecture** . . . . . **363**
The design skills pre-requisite . . . . . 363
Documenting errors, or avoiding errors? . . . . . 365
Naming (again) . . . . . 365
Design Annotations . . . . . 365
Enforced Design Decisions . . . . . 367
Coding against a framework . . . . . 367
Exemplarity in the case of design . . . . . 368

**System Metaphor (XP, DDD)** . . . . . **369**
Explaining a system by talking about another system . . . . . 369
With or without prior knowledge . . . . . 369

**Architecture Landscape** . . . . . **372**
Architecture and documentation . . . . . 372
Everybody should know about the problem . . . . . 372
Quality Attributes . . . . . 375
Stake-Driven Architecture Documentation . . . . . 375
Explicit Assumptions . . . . . 377
Architecture Steering . . . . . 378
Decision Log . . . . . 378
Journal or Blog as a Brain Dump . . . . . 383
Fractal Polyglot Architecture . . . . . 384

Documentation Landscape . . . 384
Architecture Diagrams & Notations . . . 386
Architecture Codex . . . 387
Transparent Architecture . . . 391
Test-Driven Architecture . . . 393

**Small-Scale Simulation . . . 397**
The desirable properties of a small-scale simulation . . . 398
Techniques to simplify the system . . . 398
Building the small-scale simulation is half the fun . . . 399

# Part 11 Efficient Documentation . . . 401

**Efficient Documentation . . . 402**
Focus on Differences . . . 402
Only tell what's unknown . . . 403
Search-Friendly Documentation . . . 404

**Concrete Examples, Together, Now . . . 406**
In practice . . . 407
Fast Media and prior preparation . . . 407
Together, Now . . . 408

**StackOverflow Documentation . . . 409**

**Affordable & Attractive . . . 410**
Specs Digest . . . 410
Easter Eggs & Fun Anecdotes . . . 410
Promoting News . . . 411

**Unorthodox Media . . . 412**
Maxims . . . 412
Posters & Domestic Ads . . . 413
Meme-based posters . . . 414
Information Radiators . . . 415
Very Short Story, Humor, Cheap media, and implicit message . . . 416
Digital Native! . . . 416
Goodies . . . 417
Comics . . . 418
Infodecks . . . 419
Visualizations & Animations . . . 419
Lego blocks . . . 420
Furniture . . . 420

3D printed stuff . . . 420

# Part 12 Introducing Living Documentation . . . 421

**Introducing Living Documentation . . . 422**
Undercover Experiments . . . 422
Official Ambition . . . 422
New Things Are difficult for two reasons: they have to work, and they have to be accepted. 423
Starting gentle . . . 423
Going big and visible . . . 424
A Tale of introducing Living Documentation to a team member . . . 425
Common objections . . . 428
Migrating legacy documentation into a Living Documentation . . . 429
Marginal documentation . . . 430

**Introducing Living Documentation by example . . . 431**
README and Ready-Made Documentation . . . 431
Business Behavior . . . 432
Visible Workings and Single Source of Truth . . . 432
Integrated Documentation for developers, Living Glossary for other stakeholders . . . 433
Living Diagram to show the design intent . . . 433
Contact information, and Guided Tour . . . 433
Micro-services Big Picture . . . 434

**Selling Living Documentation to management . . . 435**
Start with an actual problem . . . 435
A Living Documentation initiative . . . 436
Contrasting current situation to the promise of a better world . . . 438
A Strategy must match people aspirations . . . 438

**Documentation for Compliance requirements . . . 440**
The ITIL example . . . 442

**Documenting Legacy Applications . . . 444**
Documentation Bankruptcy . . . 444
Legacy application as a complementary documentation . . . 444
Archeology . . . 446
Bubble Context (Evans) . . . 446
Superimposed Structure . . . 448
Highlighted Structure . . . 450
External Annotations . . . 451
Biodegradable Transformation . . . 452
Maxims . . . 453

Enforced Legacy Rules . . . . . 454

**Summing it up: the curator preparing an art exhibition . . . . . 456**
Selecting and organizing existing knowledge . . . . . 456
Selecting and organizing existing knowledge . . . . . 456
Adding what's missing when needed . . . . . 457
Accessible for people who can't attend, and for posterity . . . . . 458

**Closing . . . . . 459**

# 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. (...)

[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

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

> 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 r\
eported
```

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 coord\
inates),
and where the vehicle coordinates are provided by the GPS Tracking by (vehicle, \
timestamp),
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 orig\
inal amount of money) and the interest an account has already earned.

To calculate compound interest use the formula below. In the formula, A represen\
ts the final amount in the account after t years compounded 'n' times at interes\
t 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
===========

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
```

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:

```
Scenario: The sum of all cash amounts exchanged must be zero for derivatives
 Given any derivative financial instrument
 And a random date during its life time
 When we generate the related cash flows on this date for the payer and the rece\
iver
 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 3 Knowledge Exploitation & Augmentation

# Knowledge Exploitation

## Identify authoritative knowledge

Most of the knowledge is already there. For a given project or system, a lot of knowledge is already there and it's everywhere: in the source code of the software, in the various configuration files, in the source code of the tests; in the behavior of the application at runtime, in various random files and as data within the various tools around; and of course in the brain of all the people involved.

Traditional documentation attempts to gather knowledge into convenient documents, in paper form or online. By doing so these documents basically duplicate knowledge that was already present elsewhere. That's obviously a problem when the other document is the authority. It's the one that evolves all the time and that can be trusted.

Because knowledge is already there in many places, all we need is to setup mechanisms to extract the knowledge from where it's located, to bring it where it's needed, when it's needed. And because we're lazy and don't have much time for that, such mechanisms must be lightweight, reliable and low-effort.

I's important to find where's the authoritative knowledge is. When knowledge is repeated at different places we need to know where is the knowledge that we can trust. When decisions change, where does the knowledge reflects the changes most accurately?

**Therefore: identify all the places where authoritative knowledge is located. For a given need, setup mechanisms like automation or process to extract this knowledge and transform it into a adequate form. Make sure this mechanism remains simple and does not become a distraction.**

Knowledge about how the software works is in the source code. In the ideal case, it's easy to read and there is no need for any other documentation. When it's not the case, perhaps because the source code is naturally obfuscated, we just need to make this knowledge more accessible.

## Where is this knowledge now?

Imagine a colleague or manager asking: "Gimme a documentation on stuff X!" The first step to answer that is to ask yourself or the team: "Where is this knowledge now?"

The answer is often obvious: the knowledge is in the code, in the functional tests, or in the document on project goals.

Sometime it's less obvious: the knowledge is in people's brain, or in their head but tacit. It may even be between people, and you'll need collective workshops to elucidate it.

Often at the beginning of projects, the knowledge is genuinely missing. One of the first motivation of the work will be to learn as quickly as possible, in a Deliberate Discovery fashion as Dan North says. Spikes, proofs and concepts and timeboxed work are well-suited for that.

Some knowledge is only tangible during the evaluation of the working software, at runtime.

Once we've found the authoritative knowledge: How can we harness this knowledge to become a living documentation?

When the knowledge is there but in a form that is not accessible or not convenient for the target audience and for the desired purpose, it must be extracted from its **Single Source of Truth** into a more accessible form. This process should be automated to publish a clearly versioned document, with a link to find the latest version.

Sometime the knowledge can't be extracted. For example, the business behavior can't simply be extracted as English business sentences from the code, so we write these sentences by hand as functional scenarios or tests. By doing so we introduce a redundancy in the knowledge, so we need a **Reconciliation Mechanism** to easily detect inconsistencies.

When the knowledge is spread over many places, we need a way to do a **Consolidation** of all the knowledge into one aggregated form. And when there is an excess of knowledge, a careful selection process, i.e. a **Curation** process is essential.

# Single Sourcing with a Publishing Mechanism (aka Single Source Publishing)

When the authoritative source of knowledge is source code in a programming language or a configuration file of a tool in a formal syntax, it's often necessary to make this knowledge accessible to audiences that can't read it. The standard way to do that is to provide a document in a format everyone understand, like plain English in a PDF document, or as a MS Office document, spreadsheet or slidedeck. However if you directly create such a document and include all the relevant knowledge in a copy-pasted fashion, you will have a bad time when it changes. And on an active and healthy project, you should expect it will change a lot.

Pragmatic Programmers make it clear in their Tip 68 that "DRY also for documentation." As an example of duplication they mention that a DB schema in a specification document is redundant with the DB schema file in a formal language like SQL. One has to be produced out of the other. For example, the specification document, that's really useful as a documentation indeed, could be produced by a tool that can convert the SQL or DDL file into a plain text and diagram form.

**From authoritative knowledge to published documents**

**Therefore: Keep each piece of knowledge in exactly one place, where it's authoritative. When it must be made available to audiences who can't access it directly, publish a document out of this single source of knowledge. Don't include the elements of knowledge into the document to be published by copy-pasting, but use automated mechanisms to automatically create a published document straight from the single authoritative source of knowledge.**

## Some examples of producing a published document

There are many tools available to produce documents out of source code and other technical artifacts. Here are some examples:

- Github takes the README.md file, that is a single source of knowledge about the goals of an overall project, and turns it into a web page rendered to be pretty
- Javadoc extracts the structure and all the public or private API of the code and publishes as a website as a reference documentation. You can easily create a custom tool based on the

standard Javadoc Doclet in order to generate your own specific report, glossary or diagram, as described later in the book.

- Tools like Maven have a built-in way (e.g. 'maven site') to produce a consistent documentation, usually as a website, by putting together a number of tool reports and rendered artifacts. For example it collects test reports, static analysis tools reports, Javadoc output folders, and any markdown documents to organize all that into a standard website. Every markdown document can be rendered in the process.
- Leanpub, the publishing platform I use to write this book, is a canonical example of a single sourcing with a publication mechanism: every chapter is written as a separate markdown file, images are kept outside, the code can be in its own source files, and even the table of content is in its own file. In other words, the content is stored in the way it's most convenient to work with. Whenever I ask for a preview, Leanpub's publishing toolchain collate all files according to the table of content, renders it through various tools for markdown rendering, typesetting and code highlighting in order to produce a good quality book.

In its simplest form, you can follow this pattern with any templating mechanism and a bit of custom code. For example you could produce a PDF out of the resource file that lists every currency supported by the program.

**One single source, possibly multiple documents**

## Published Snapshot with a version number

Any document published from a single source of trust is a snapshot: it must therefore be considered as strictly immutable and should never be edited. To avoid the risk of having someone edit a published document, favor documents formats that prevent edition, or at least that make it not too easy. For example, prefer PDF over MS Office documents, that are too easy to change. Whatever the format, consider using the locking flags to prevent edits. It's not about making it impossible to edit by a hacker, but just to make it just hard enough so that it becomes easier to change the authoritative source and have it published again.

Any published document must clearly identify its version, and should also include a link to the location where the latest version can be found.

If you produce a lot of paper documents to be printed, you may consider putting on each of them a barcode with the link to the folder that always has the latest version. This way, even a printed document can easily direct the readers to the latest version.

## Remarks

Only write by hand which that could not be extracted from an already existing project artifact, and store it in its own file that has its own lifecycle. Ideally it will change much less frequently than the knowledge to extract from other places. On the other way round, if some information is missing for the document to publish, by all means try to add it into the artifact it is most related with, perhaps using annotations, tags or naming convention, or make it a new collaborative artifact on its own.

# Reconciliation Mechanism (aka Verification Mechanism)

Duplication of knowledge about a software is a bad thing because it implies recurring work to update all the places that are redundant to each other, and it also implies a risk of getting into an inconsistent state when an update is forgotten.

However if you have to accept redundancy you can relieve the pain thanks to a Verification Mechanism, for example an automated test that checks that both copies are always in sync. This does not remove the cost of making changes more than one place, but at leasts it ensures you won't forget one change somewhere.

One reconciliation mechanism everybody's familiar with is checking the bill in the restaurant. You know what you ate, which may be still visible by looking at the number of dishes, and you check each line on the bill to check there's no discrepancy.

**Checking the restaurant bill is a reconciliation mechanism**

**Therefore: When you want or have to accommodate a redundancy in the knowledge stored at various places, make sure all the redundant knowledge is kept consistent thanks to a Reconciliation Mechanism. Use automation to make sure everything remains in sync, and that any discrepancy is detected immediately with an alert prompting to fix it.**

**Automated mechanism to verify the redundant knowledge is in sync**

## Consistency Tests

A well-known example is BDD, where the scenarios are the documentation of the behavior. Whenever scenario and code disagree, it shows immediately because the test automation fails.

This mechanism is made possible thanks to tools that parse the scenario in natural domain language to drive their implementation code. The code is driven through a little layer of glue code that you write specifically for that purpose, usually called "Steps Definitions". These are adapters between the parsed scenario and the actual code being driven.

Imagine testing the following scenario:

- Given party BARNABA is marked as bankrupt
- And the trade 42 is against BARNABA
- When the risk alerting calculation is ran
- Then an alert: Trade against the bankrupt party BARNABA is triggered

The tool parses these lines of text, and recognizes the sentence "Given party BARNABA is marked as bankrupt" as one it has a step definition for:

```
Given("^party (.*) is marked as bankrupt$")
public void partyMarkedAsBankrupt(string party){
  bankruptParties.put(party);
}
```

The tools does the same for each line. Typically sentences starting with *When* trigger actual computation, and sentences starting with *Then* check assertions:

```
Given("^Then an alert: (/*) is triggered$")
public void anAlertIsTriggered(string expectedMessage){
  assertEquals(expectedMessage, actualMessage);
}
```

You realize, of course, that for this all this mechanism to work, the sentences need to actually drive the code with parameters, and the assertions must check against the expectations from the sentences as precisely as possible.

As a counter example, it would not make less sense to code the step without extracting parameter from the sentence, or we would run again a risk of being inconsistent after a few changes:

```
Given("^Then an alert: Trade against the bankrupt party BARNABA is triggered$")
public void anAlertIsTriggered(){
  assertEquals("Trade against the bankrupt party ENRON", actualMessage);
}
```

The scenario would pass even if the code had changed its behavior.

## Reconciliation on the test assumptions

Usually we use the given (or their equivalent "Arrange" phase in plain xUnit code) to create mocks or to inject data into the test database.

When testing legacy systems, it happens that you have to deal with a number of bad news:

- it's too hard to mock the database so you have to test in an end-to-end fashion
- you can't re-create or populate a database just for you tests so you have to work on a real shared database that can change anytime if someone else decides to.

In this case it's still possible to use the exact same declaration of an assumption as a *When* sentence or an Arrange phase in xUnit, but with an implementation that checks that the assumption still holds true instead of injecting the value into a mock:

```
Given("^party (.*) is marked as bankrupt$")
public void partyMarkedAsBankrupt(string party){
  assertTrue(bankruptParties.isBankruptParty(party)); // calls the DB
}
```

This not an assertion of the test, it's just a pre-requisite for the scenario (or test) to even have a chance to pass. If this assumption already fails, then the scenario "does not even fail".

I often call this kind of "tests before the tests" *Canary Tests*. They tell something's wrong even outside of the test focus, so that we know we don't have to waste time investigating in the wrong place.

## Published Contracts

Another flavor of Reconciliation Mechanism that I have first seen used by my Arolla colleague Arnauld Loyer can be used on purpose to respect contracts against third parties like external services that call your services. If your services exposes a resource with a parameter *CreditDefaultType*, with two possible values *FAILURE_TO_PAY* and *RESTRUCTURING*, you can't rename them as you wish once published.

So you may use tests, with a deliberate redundancy with respect to these elements of the contract, to enforce that they don't change. You can refactor and rename as you wish, but whenever you break the contract, the reconciliation tests will alert you with a test failure.

This is an example of an enforced documentation: ideally you would make the test the reference documentation of the contract (some tools in the API sphere enable to do that) in a readable form. Here you definitely don't want to update the test through automated refactoring, you want it out of reach of the refactoring so that it stays unchanged to represent the external consumer services.

The most naive implementation for this approach would be something like that, assuming that the internal representation of the CreditDefaultType is a Java enum named *CREDIT_DEFAULT_TYPE*:

```
@Test
public void enforceContract_CreditDefaultType
  final String[] contract = {"FAILURE_TO_PAY", "RESTRUCTURING"};

  for(String type : contract){
    assertEquals(type, CREDIT_DEFAULT_TYPE.valueOf(type).toString());
  }
}
```

Since we want to make sure that the contract for the external calling code is respected, we define this contract *again* as an array of strings, like it's being used from the outside. And since we want to check

the contract is being honed with incoming and outcoming values, we make sure the contractual string is recognized as an input with the *valueOf()*, and that it's the one being sent as an output with the *toString()*.

This example is only to explain the idea, in practice it's bad practice to use a loop inside a test as the test reporting will not tell precisely at which loop the problem was in case there's an exception. We would use a Parameterized test instead, putting the collection of values that are part of the contract as the source of parameters, but this is not the focus of the discussion.

With this approach, when a new joiner to the team decides to rename a constant of the enum, the test immediately fails to signal that it's not possible to do that, in effect acting like a defensive documentation. It's a defense against misconduct, and at the same time when there's misconduct by ignorance, it's the opportunity for the violator to learn on the spot. It's another flavor of embedded learning.

# Information Consolidation

Sometime the knowledge is spread over many place: a type hierarchy with an interface and 5 implementing classes is actually declared in 6 different files. The content of a package or module is actually stored in many files. The full list of dependencies of the project is actually defined partially in its Maven manifest (POM), and also in its parent manifest.

This means that there's a need to collect and aggregate many little bits of knowledge in order to get a full picture.

**From fragmented authoritative knowledge to unified knowledge**

For example, the big picture of a system is basically the union of the black-box view of each of its part. We say that the overall knowledge is derived by a consolidation mechanism.

Even if the knowledge is split in many little parts, it's still desirable to consider all these little bits as

the authoritative single sources of truth. The derived consolidated knowledge is therefore a special case of published document extracted from many places.

**From fragmented authoritative knowledge to unified knowledge**

**Therefore: Design a simple mechanism to automatically consolidate all the disparate information from many places. This mechanism must be ran as often as necessary to ensure that the information about the whole are up-to-ate with respect to the parts. Avoid any storage of the consolidated information, except for technical concerns like caching.**

## How it works

Basically a consolidation is like a SQL Group By. You take many things with some properties in common, and you find a way to turn this plural into an equivalent singular. In practice it's done by scanning every element within a given perimeter, while growing the result.

For example, to reconstitute the full picture of a class hierarchy within the limits of one project from its individual elements, it's necessary to scan just every class and every interface of the project. The

scanning process keeps a growing dictionary of every hierarchy under construction so far in the process, for example with a mapping *top of hierarchy -> list of subclasses*. Every time it encounters a class that extends another or that extends an interface, it adds it to the dictionary.

When the scan is done, the dictionary contains a list of all type hierarchies of the project. Of course it's possible to reduce the process only to a subset of these hierarchies of interest for a particular documentation need, like restrict the scan only to classes and interfaces that belong to a published API.

As another example, if we want to create a blackbox living diagram of a system made of smaller components that each have their own set of inputs and outputs, we want to do the following:

**The blackbox view of the whole system can be derived by a consolidation of the blackbox view of its components**

In its simplest form, the consolidation can just collect the union of every input and output from each component. In a more sophisticated approach, it can try to remove every input and output that match to each other internally. It's up to you to decide how you want it to happen for a particular need.

# Implementation remarks

As usual, the first idea must be to reuse a tool that already can do the desired consolidation. Some parsers for Java code can provide type hierarchies for example. If what you want is not there, you can add it, for example by writing another visitor on the programming language AST. Some more powerful tools even provide a their own language to query the code base very efficiently. In this idea, you may want to load the AST into a graph database if you have to do very complex queries. But if you begin to do that, I'm afraid you're becoming a software vendor of documentation tools.

If the derived knowledge is kept in cache for performance issues, make sure it does not become a source of truth, and that it can always be properly dropped then rebuilt from scratch from all the sources of truth.

For most systems it is possible to scan all parts in sequence in a batch processing fashion. This is typically done during a build, and produces the consolidation ready for publication on the project website or as a report.

For large systems like an information system, it is not practical to run calculations scanning all parts in sequence. In this case the consolidation process may be done incrementally. For example the build of each part can contribute a partial update by pushing data to an overall consolidation state somewhere in a shared place like a shared database. This consolidation state is derived information, it is less trusted than the information from each build. If anything goes wrong, drop it and let it grow again from the contributions of each build.

# Ready-Made Documentation

## Software Craftsmanship Apprenticeship Patterns book

Study the Classics - reference them when you use their knowledge in a decision (patterns, algorithms, principles and theorems)

Not all knowledge is specific to your context, a lot of knowledge is generic and shared with many other people in many other companies in the industry. Think about all the knowledge on programming languages, developers tools, software patterns and practices; most of that is industry standard, as we say.

As the State-of-the-Art is making progress, more and more of what we do everyday gets codified by talented practitioners into patterns, techniques and practices. And all that knowledge is properly documented into books, blog posts or conference talks and workshops around the world. That's Ready-Made Documentation that's readily available, for the price of a book.

Here are some random examples: - Test-Driven Development, a design technique by Kent Beck - The 23 Design Patterns from the Gang of Four - Analysis Patterns, Patterns of Enterprise Application Architecture and everything written by Martin Fowler - Domain-Driven Design by Eric Evans - Everything on the C2 wiki - Every book from Jerry Weinberg - Continuous Delivery patterns by Jez Humber and Dave Farley - All the Clean Code literature - Git workflow strategies - and thousands of other great content in the literature

We're pretty much in a situation where we could safely say:

> "If you can think about it, somebody has already written about it."

Patterns, standard names, standard practices exist, even if you don't know them yet. The literature is still growing and so huge that you cannot know it all, or you would spend so much time reading you would not have any time left to create any software.

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.

Examples: essential reading lists by business domain, 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.

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.

**Therefore: Consider that most knowledge is already documented somewhere in the industry literature. Do your homework and look for the standard source of knowledge, on the web or by asking other knowledgeable people. Don't try to document again something that's been already well-written by someone else, link to it instead. And don't try to be original, instead adopt the standard practices and the standard vocabulary as much as possible.**

In most cases, being conformist by deliberately adopting industry standards is a win. What you're doing is almost certainly already covered somewhere. If you're unlucky it will be only in a blog or two. If you're lucky it's industry standard without you knowing. Either way, you want to find where it is covered, for several reasons:

- You can refer to them instead of doing yourself the writing
- They may suggest improvements or alternatives you haven't considered
- They may describe the situation in a more deeper way than you did, giving you external insights
- The description validates that your approach makes sense. If you cannot find any account, beware.
- Most importantly, this will tell you how the rest of the planet talks about this situation.

## The power of a standard vocabulary

> He who controls vocabulary controls thought – Wittgenstein

Talking the same words as everybody on the planet is a fantastic advantage. You can now talk with shorter sentences. You could spend several sentence trying to describe the design of a text editor:

> Inline editing is done thanks to a an interface with several subclasses. The text editor delegates the actual processing to the interface, without having to care which subclass is actually doing the job. Depending on whether the inline editing is on or off, an instance of a different subclass is used.

However if you're familiar with ready-made documented knowledge like design patterns, then:

> Code written according to a consistent and shared pattern language can be described more concisely. "Inline editing is implemented as a State of the Controller" –if you know the vocabulary, you know what you'll find when you look at the code. – Kent Beck https://www.facebook.com/notes/kent-beck/entropy-as-understood-by-a-programmer-part-1-program-structure/695263730506494

Each mature industry has its own rich jargon because it's efficient to communicate. Every part in a car has its specific name depending on its role in the vehicle: s shaft is not just shaft, it's a *camshaft*, or a *crankshaft*. There's a *piston* in a *cylinder*, *pushrods*, and a *timing chain*. Domain-Driven Design advocates to carefully grow such a Ubiquitous Language of the domain.

Our industry makes progress each time its standard vocabulary grows, for example whenever Martin Fowler coins another name for a patterns that we do without thinking about it. This process is in fact a process of growing our own Ubiquitous Language for our industry.

In the book *Software Craftsmanship Apprenticeship Patterns*, Dave Hoover and Adewale Oshineye advocate "Study The Classics".

As a result, if you know what you're doing and you know how it's called in the industry, just insert a reference to the industry standard and you have achieved extensive documentation at low cost.

Patterns and pattern languages are particularly effective ways to pack ready-made knowledge in a reusable documentation. Patterns really are *canned documentation*. They create a standard vocabulary one can use, and refer to for complete reference.

> Design patterns are communication tools for experienced programmers. Not training wheels or scaffolding for beginners. – @nycplayer on Twitter

> Patterns matter. But when I started leaning about design patterns, I was trying to use them whenever I could. Its so common that some even call that *patternitis*. Then I became reasonable and learnt when not to use them.
>
> Many article have expressed harsh criticism about having the code full of patterns; however I think they miss the point: you should learn as many patterns as you can. The point is not to learn patterns in order to use them, though it can be useful, but the point is to know many patterns in order to know the standard name of what you're already doing. In this view 100% of the code could, and perhaps should, be described by the means of patterns.

Knowing the standard vocabulary also opens the door to even more knowledge: you can find books and buy training on the right topic you're interested in. You can also pinpoint people with this knowledge to hire them.

It's not so much about finding a solution. Even when you have a perfect solution it's worth finding how it's called in the industry. This way you can just refer to the work of other people who describe the solution in a well-written, peer-reviewed and time-reviewed fashion.

## Link to standard knowledge

Generic knowledge is already documented in the industry literature, in books or online. When you use it, link to the authoritative source, with an Internet link or a bibliographic reference. It has been written well once, refer to it rather than redoing a poorer job of documenting it again.

Of course the biggest problem is to identify the standard name for a piece of knowledge.

## Searching for the reference

Search engines like Google, or community websites like C2 or Stack Overflow are your friends there. You have to start by trying to guess how other talk about a topic. Then you can quickly scan the first results from the search engine to find out a more accurate vocabulary which in turn helps make narrower queries. Through this exploration you'll quickly learn a lot and get to see how much of the topic is already codified, and under what names.

Don't hesitate to ask around for suggestions, in your team or on a forum. Their experience and seniority of other people helps as they had more time to index more pointers to standard knowledge they've encountered, even superficially, over the years.

From a given term, you can also browse Wikipedia and all the various links at the end of the articles. Keep an eye on the bottom "related" links as well, until you recognize your situation. Wikipedia is a fantastic tool to map a standard vocabulary to your own mental universe.

## More than just vocabulary

A shared standard vocabulary is the key advantage to communicate effectively, in conversations and in writing. That said, once you've found a standard description, it often comes with refinements and alternatives you haven't considered. This is useful too. Ready-made documentation is in fact reusable thinking, which is a great help. It's a bit like having the author, usually a seasoned practitioner, close to you to think together. Whenever you read on a standard solution, process, pattern or approach, the secret sauce is suggested by @michelesliger on Twitter:

> YOU STILL HAVE TO THINK. BUT YOU DONT HAVE TO DO IT ALONE.

When we say that "we create an Adapter on top of the legacy subsystem", this sentence implies a lot of things in a few words, because there's more than a name in the idea of the Adapter pattern.

For example, an important consequence in this pattern is that the adaptee, the legacy subsystem in our example, should not know about the Adapter, only the Adapter should know about the legacy subsystem.

When we say that this package represents the Presentation Layer whereas this other package represents the Domain Layer, we also imply that only the former can depend on the latter, never the other way round.

It's the norm in mathematics to reuse theorems and shared abstract structures from the literature to go further without re-inventing and having to prove the same results again and again. It's not just about the vocabulary.

# Ready-made knowledge in conversation to speed up knowledge transfer

Here's a little conversation with my friend Jean-Baptiste Dusseaut ("JB" in short, @BodySplash on Twitter) to illustrate how a common culture and vocabulary helps sharing knowledge efficiently.

- Hello JB, I heard you launched a new startup, Jamshake, what is it about?

**Hello JB, I heard you launched a new startup, Jamshake, what is it about?**

- It's a social and collaborative tool for musicians. We provide both a lightweight social network to find other musicians and cool projects, and an in-browser Digital Audio Workstation, the Jamstudio, to collaborate in real time with other musicians. It's a kind of Google Doc for music.

**It's a social and collaborative tool for musicians**

- Sounds really cool! On the technical side, how's your system organized in a nutshell?
- I know you're familiar with Software Craftsmanship and design, and DDD in particular, so you won't be surprised to hear our system is made of several sub-systems, one by Bounded Context indeed.

**I know you're familiar with Software Craftsmanship and design**

- Oh yes, makes perfect sense! Each one is a microservice then?
- Yes and no. They begin as modules very strictly decoupled in terms of dependencies, i.e. ready to be extracted into their own process at runtime. But we keep them within the same process until we really need the separate processes, usually to scale with the increasing load.
- Yes! I call that the "microservice-ready" style of code. You don't have to pay the full cost of too many physical services upfront, as long as you have the option to do it at anytime. But it takes discipline.
- Yes and that's easy when you're just 1 or 2 developers like we are at the moment. In practice we regularly use these options because of the increasing load.
- Increasing load: that's good problems to have when you're a startup growing and looking for financing!
- yes absolutely.
- I'm curious to have an overview of the full system. Perhaps Bounded Context by Bounded Context?
- Sure. There are around 7 Bounded Contexts at the moment: Acquisition (registration of new users), Arrangement, Audio Rendering (mixing, limiter and compression kinds of processing), Stems Management, and Reporting. They all rely on Spring Boot instances on top of separate

Postgres databases, except the Stems management built with Node.js on top of an S3 storage. Each Bounded Context is paying attention to their domain model, except Registration that CRUD-y based on Hibernate. It's a survivor from the early version of the system!

**Picturing the full system in mind while listening to JB talking**

- Alright, I now have a clear picture in mind of what it's like. Thanks a lot JB!

## Gut feeling and spontaneity

Since when knowing something consciously should be considered less desirable than just gut-feeling?

This is however one of the various kinds of criticisms about patterns out there, and this one is clearly explained in an interesting post from Steve:

*The use of patterns is like the use of literary device. There are (probably) an infinite number of ways in which the same general thought can be expressed, but I doubt you will find a single quality writer who started off a chapter thinking, "I'm introducing a character here so it's best to paint a picture of the character. That calls for simile. Yeah, simile will do it. I think I'll also use some ironic juxtaposition." This type of writing feels forced. I've read code where the application of design patterns also felt forced.*

Steve has a point here. I must admit that gut feeling, if trained properly on examples of good quality, may have advantages over a conscious quest for perfection, perhaps because our brain is more powerful that we can ever be conscious of. And yes, very often, we pretend that what we did was intentional and deliberate whereas we're just explaining a posteriori a decision that was actually based on gut feeling.

Then Francois from the ORM *Propel* raised the topic: should developers know design-patterns? He discusses in a blog post the reasons to mention or not the various patterns used at the heart of the Propel ORM in the documentation of the engine.

ORM engines are rather sophisticated pieces of software, and they make big (and deliberate) use of patterns, in particular the Fowler PoEAA patterns:

*Propel, like other ORMs, implements a lot of common Design Patterns. Active Record, Unit Of Work, Identity Map, Lazy Load, Foreign Key Mapping, Concrete Table Inheritance, Query Object, to name a few, all appear in Propel. The very idea of an Object-Relational Mapping is indeed a Design Pattern itself.*

If you know the patterns, you can understand Propel quickly; if you don't, then you'll need to go through much more explanations to reach the same level of expertise. And next time you'll encounter another ORM you'll have to redo this discovery effort. Of course at some point you'll recognize the patterns, and you just won't know their names. You'd just be half-conscious of the patterns.

# Tools History

As we've seen before, a lot of knowledge is already there, and some of it is hidden in the history of the tools you already use. Source control systems are an obvious example. They know about every commit, when they were done, by who, what were the changes, and remember each commit comment. Other tools like Jira or even your email also know a lot about your project.

However this knowledge is not always readily accessible, and is not used as much as it could. For example if there's no screen to conveniently retrieve the the most commonly asked question on the chat, you may never know it.

Sometime you have to re-enter the same knowledge in another form in another tool. For example a commit to fix a bug with a comment that states it fixed the bug, however in many companies you have to also go to the work tracker to declare you've fixed the bug. You also have to declare the time spent on the task, only to enter it again into the time tracking tool later in a aggregates form. This is a waste of time. Consider integrating the tools together.

Note that better integration between tools also helps simplify the human tasks which reduces the need for manual documentation of the tasks. However when the integration fails, then you do need documentation. Ideally the integration component should provide this documentation. For example an integration script should be readable and as declarative as possible.

**Therefore, exploit the knowledge stored in the tools. Decide what tool is the unique authority for each bit of knowledge. Search for plugins that can provide integration with other tools, or specific reports for your documentation purposes. Learn how to use the command-line interface of your tools to use them programmatically to extract knowledge or integrate them with other tools. Discover the API's provided by the tools, including the email or chat integration.**

As a last resort, find out how to query their internal database, but beware that it may change at any time without prior notice as it's usually not part of the official API.

Some examples of tools and their knowledge:

- Source control (e.g. Git with the blame command): who changed what, when, **commit comments and Pull Requests discussions**
- Internal Chat system (Slack etc.): questions, launch build, release, mentions of words, activity, moods, who, when
- User Directory: **mailing lists**: teams, team members, team managers, i.e. who to contact for support, who to contact for escalation…
- Console history: most recently or commonly used commands or sequences of commands
- **Services registry**: the list of **every running service**, their address, plus any **additional tags**

- Configuration server: environment configuration details…
- Company Service Catalog: services governance information: who to contact, time last updated…
- Project registry, even as a spreadsheet file on a shared drive: project names, codes, leader, sponsor identification, budget code…
- Sonar components, their grouping into logical units, metrics and their trends, at various levels of details, all this across multiple repositories and multiple technologies
- Tracker history, release manager history: who, when, current versions, what changes in each version…
- Email server, to archive for auditing (e.g. by forwarding to the archiving address): manual reports, manual decisions like go-live decisions, most knowledgeable collaborators (implicitly)

# Augmented Code

> The source code might have code that is never executed, variable and procedure names that are lies, and is in general a poor way to learn the programmer's intention. To me, design is as much the decisions and the reasons as the results of the decisions. Sometimes code makes that clear, but usually it doesn't.
>
> – RalphJohnson http://c2.com/cgi/wiki?WhatIsSoftwareDesign

Software is built from its source code. Does this mean that the source code tells everything there is to know over the lifecycle of the application?

Sure, the source code tells a lot, and it has to. The source code describes how to build the software, for the compiler to do it. Clean Code goes further, it aims at making knowledge as clear as possible for the other developers working on it.

Still, code is often not enough.

## When programming languages are not enough

Most programming languages have no predefined way to declare the key decisions, to record the rationale and to explain the choice made against the considered alternatives. We need a way to add the knowledge behind these decisions into the code. But programming languages can never tell everything. They focus on their key paradigm, and then rely on other mechanisms to express the rest: naming, comments, libraries etc.

> Consider the infamous metaphor of a construction of a bridge. A bridge is built from its technical drawings. However if we are to replace its wood timbers by new ones in a new material like steel, the original technical drawings won't be enough. They will tell the dimensions chosen for the wooden timbers, but they won't tell where the dimensions come from. They won't tell about the calculations of resistance of materials, of fatigue of materials, of resistance against strong waters and extreme wind forces. They won't tell what was considered "extreme" at the time. Perhaps it should be reconsidered now to accommodate more extreme conditions in the light of recent events, like a tsunami that were thought unlikely at the time of construction but that we now know actually happens.

When it comes to documenting design decisions and their rationale, programming languages can't help much beyond simple standards decisions like the typical visibility of members, or inheritance.

When a language does not support a design practice, workarounds like naming conventions do the job. Some languages with no way to express private methods prefix them with an underscore. Languages without objects adopt a convention of having a first function parameter called 'this', and so on. Yet even with the best programming language, a lot of what's in the developer's head still cannot be fully expressed by the sole language.

It's possible to add knowledge as code comments. But comments lack structure, unless you hijack structured comments like Javadoc. Also refactoring do not apply to comments as well as they apply to the code.

**Therefore: Augment your programming language so that the code can tell the full story, in a structured way. Define your own way to declare the intentions and the reasoning behind each key decision. Declare the higher-level design intentions, the goals and the rationales.**

**Don't rely on plain comments for that. Use strong naming conventions, or the extension mechanisms of the language, like Java annotations and .Net attributes, the more structured the better. Don't hesitate to write a little code solely for this documentation purpose. Create your DSL or reuse one if needed. Rely on conventions when suitable.**

**Keep the augmented knowledge as close as possible to the code it is related to. Ideally they should be collocated to be totally refactoring-proof. Make the compiler check for any error. Rely on the autocompletion of the IDE. Make sure the augmented knowledge is easily searchable in your editor or IDE, and that it is easily parseable by tools to extract living documents out of the whole augmented code.**

With Augmented Code, even after all documentation has been lost, the code still contains a lot of valuable hints for the future maintainers.

One important consideration when adding knowledge related to the code is how it evolves when the code changes. Code will change, because that's the way it is. As a consequence it's essential that the additional knowledge either remains accurate, or changes at the same time as the code, with no or as little manual maintenance as possible. What happens when a class or package is renamed? What happens when a class is deleted? The extra knowledge we want to add should be refactoring-proof.

Augmented Code is great to make decisions explicit in the code, and to add the rationale behind the decisions.

Because it is structured, Augmented Code is also easy to search and to navigate in the IDE, without plugins. This means that it also works the other way: from a chosen rationale you can find all the code that is related to it. That's quite valuable for traceability or impact analysis.

Augmented Code in practice can be done with several approaches:

1. by Annotation
2. by Convention
3. with Sidecar files
4. with a Metadata database
5. with a DSL

# Documentation by Annotations

This is my favorite method to augment code in languages like Java or C#. Annotations do not impose any constraint on naming, code structure, which means they work in most codebases. And because they are almost as structured as the programming language itself, it's possible to rely on the compiler to prevent errors, and to rely on the IDE for autocompletion, navigation and search.

The main strength of annotations is that they are Refactoring-friendly: they are robust to renaming of the element they are attached to, they move with it when it moves, and they get deleted when it's deleted. This means no extra effort to maintain them, event the code changes a lot.

**Augmented Code = Code + Annotations**

**Explain the design and its purpose using structured annotations. Create, grow and maintain a catalogue of predefined annotations, then simply include these annotations to enrich the semantic of the classes, methods and modules.**

You can then create little tools that can exploit the additional information in the annotations, for example to enforce constraints, or to extract knowledge into another format.

Once you have the annotations and you know them, it becomes faster to declare a design decision: just add the annotation. They are like bookmarks for the thinking that happened.

Annotations can represent class stereotypes like Values, Entities, Domain Services, Domain Events. They can represent active patterns collaborators, like a Composite or an Adapter. They can declare styles of coding and 'default preference unless stated otherwise'.

It's important that your annotations correspond to standard techniques with standard names as much as possible. If you need your own custom ones then they must documented in a place that everybody knows.

Putting an annotation to declare your decisions in terms of standard knowledge and standard practices encourages Deliberate Practice. You have to know what you're doing, and you have to know how it's called in the industry literature. In the case of using standard design patterns this has been shown to reduce time required to complete a task some studies.

The annotations are also searchable in your IDE, and this is handy. From an annotation, it's easy to find every class where it's used, which gives a new way to navigate the design.

Structured annotations are a powerful tool, however they are probably not enough to completely replace all other forms of documentation to describe all design decisions and their intentions. You still need conversations between everyone involved. There is also knowledge and insights that are best explained through clear writing with a sense of nuance, something that's hard to do with annotations.

You may also find desirable to keep track of a few emotional aspects, and other media like plain text are better for that.

Lastly, the knowledge declared via annotations is machine-readable, which opens opportunities for tools to exploit this knowledge to help the team. Living Diagrams and Living Glossary for example rely on such possibilities. Imagine what you could do, or what you could have tools do for your indeed, once tools can understand your design intents!

## Annotations are more than tags

Annotations in Java and attributes in .Net are genuine citizens of their programming language. They have a name and a module name (package or namespace). They also hold parameters, and can themselves be annotated by other annotations. And being classes, they also benefit from the structured comments syntax used by documentation generators like Javadoc.

All this means you can convey a lot of knowledge through simple annotations.

As a technical example, annotations use meta-annotations to describe where they can be applied. For example here the annotation Adapter can be applied to types and packages:

```
1 @Target({ ElementType.TYPE, ElementType.PACKAGE })
2 public @interface Adapter {
3 }
```

A more useful example involves annotations with parameters. If we were to annotate an instance of a builder pattern, we could describe the type that the builder produces as a parameter of the annotation:

```
public @interface Builder {
        Class[] products() default {};
}

@Builder(products = {Supa.class, Dupa.class})
public class SupaDupaBuilder {
  //...
}
```

Often the declared return types and implemented interfaces can already tell a lot of similar information, however they miss the precise semantics that additional annotations convey. In fact, more precise annotations like that open the door to more automation because it gives tools a way to interpret the source code with a higher-level semantics.

Just like the Semantic Web aims at transforming the unstructured data into a web of data, a code base with annotations that clarify the semantics of the source code becomes a web of data that machines can begin to interpret.

## A good place to describe the rationale behind the decisions

One of the most important piece of information worth recording for the future generations is the rationale behind each decision. What may seem a stupid choice years later was probably not that stupid when it was decided.

Most importantly, when the rationale is referring to a context at some point in time, and now the context is different, you are in a better position to reconsider the decision now.

For example, an expensive database was chosen long ago because it was one of the few to be able to cache data fully in memory. Reading this rationale now you may consider NoSQL datastores for the same purpose.

As another example, your application has layers talking to each other through XML everywhere, which makes your life cumbersome, and with performances issues. The rationale for this decisions explains that this architecture was meant to be distributed physically between layers in order to scale. However after many years, it has become clear that this will never happen, so you know now that you could remove all this extra complexity. Without a clear rationale, you would always wonder if you missed something and perhaps you would never dare reconsidering the whole thing.

## Annotations for learning on the job

At a minimum, annotations should themselves be documented. If you have an annotation named "Adapter", then its comments should explain what an Adapter is. My favorite way to do that is to link to a clear online definition like the corresponding Wikipedia page: http://en.wikipedia.org/wiki/Adapter_-pattern, along with a brief description text within the comment itself:

```
/**
 * The adapter pattern is a software design pattern that allows the interface of
 * an existing class to be used from another interface.
 *
 * The adapter contains an instance of the class it wraps, and delegates calls
 * to the instance of the wrapped object.
 *
 * Reference: See <a href="http://en.wikipedia.org/wiki/Adapter_pattern">Adapter\
 pattern</a>
 */
public @interface Adapter {
}
```

This is more important than it seems. From now on, every class with this annotation is only a tooltip away from a complete documentation of its design role. Let's take the example of a random adapter class in a project, here it's an adapter on top of the RabbitMQ middleware:

```
@Adapter
public class RabbitMQAdapter {
        //...
}
```

When this class is open in any IDE, then when the mouse hovers over it, the tooltip displays its documentation:

**Overview of a Living Glossary**

The brief description does its best to provide an explanation, that is probably more useful for developers who already know and just need to be refreshed. For others who don't know, the link is one click-away to redirect them to the place where they can start to learn. They will probably ask questions in the process, but at least there's an easy entry point to the learning. In this case, the annotations not only describe that the class is an instance of the Adapter pattern, it also acts as a gateway drug to learn more about the Adapter pattern.

It's possible to elaborate a lot on this simple idea. The annotation could also link to the book or books that best explains the topic. They could link to the company e-learning program.

As an alternative to the links in comments, every annotation from the same book could have a meta-tag representing the book.

For example, both the Adapter and the Decorator are from the Gang of Four book "Design Patterns", so the information about the book can be factored in an annotation specifically about the book:

```
/**
* Book: <a href="http://books.google.fr/books/about/Design_Patterns.html?id=6oHu\
KQe3TjQC">Google Book</a>
*/
@Target(ElementType.ANNOTATION_TYPE)
public @interface GoF {
}

@GoF
public @interface Adapter {
}

@GoF
public @interface Decorator {
}
```

This is only an example, and of course it is not limited to documenting design patterns! Feel free to elaborate your own scheme for organizing your knowledge based on these ideas.

## If your programming language does not have annotations

You can use structured tags within comments:

```
/** @Adapter */
```

It's a good idea in this case to conform to a common style of structured documentation from the language. It may give you some tool support, like autocompletion and code highlighting. The

XDoclet library did that with great success in the early Java days, hijacking the Javadoc tags in order to use them as annotations.

You may also use the good old Marker Interface pattern: implementing an interface with no method just to mark the class. For example, to mark a class as Serializable, you would implement the `Serializable` interface:

```
public class MyDto implements Serializable {
            ...
        }
```

Note that this is quite an intrusive way to tag a class, and it pollutes the type hierarchy.

---

## When annotations go too far...

Google Annotations Gallery[10] is a retired little open-source project from 2010 which proposed a collection of neat annotations to augment your code with you design decision, intentions, honest feelings, and even shame.

Discovering stupid code? Leave a @LOL or @Facepalm or @WTF annotation:

```
@Facepalm
if(found == true){...}
```

Or all of them with an explanation:

```
@LOL @Facepalm @WTF("just use Collections.reverse()")
 <T> void invertOrdering(List<T> list) {...
```

You can also use Remark annotations to preemptively qualify your own miserable code:

```
@Hack public String  unescapePseudoEscapedCommasAndSemicolons(String url) {
```

Or justify it:

[10] https://code.google.com/p/gag/

```
@BossMadeMeDoIt
String extractSQLRequestFromFormParameter(String params){...}
```

You may warn your team members with the `@CantTouchThis` annotation.

Stumble across code that somehow works beyond all reason? Life's short. Mark it with @Magic and move on:

```
@Magic
 public static int negate(int n) {
   return new Byte((byte) 0xFF).hashCode() / (int) (short) '\uFFFF' * ~0 * Char\
acter.digit ('0', 0) * n * (Integer.MAX_VALUE * 2 + 1) / (Byte.MIN_VALUE >> 7) *\
 (~1 | 1);
  }
```

And when you've done a good job of design, let the world know your brilliance with the Literary Annotations:

```
@Metaphor public interface Life extends Box { }
```

Or

```
@Oxymoron public interface DisassemblerFactory { Disassembler createDisassemble\
r(); }
```

---

# Documentation by Convention

Using plain conventions to document your decisions is convenient. For example in Java every identifier starting with an upper case letter is a class, whereas identifiers starting with a lower case letter are variable names.

There are conventions for every situation, in all technologies, and you can always add your own conventions on top of any technical environment, be it code, XML, Json, assembly or SQL. Even old projects with old technologies rely on conventions to communicate knowledge, describe their structure and help navigation.

Here are some examples of documentation by convention:

- Packages names by layer: everything in a package named *.domain.* represents domain logic, whereas everything in a package named *.infra.* represents infrastructure code.
- Packages names by technical class stereotype: *.ejb.* , *.entity.* , *.pojo.* , *.dto.*
- Commit comments with conventions like "[FIX] issue-12345 free text", where the square brackets categorize the type of commit out of FIX, REFACTOR, FEATURE or CLEAN, and the issue-xxx references the ticket id in the bug tracker.
- The full Ruby on Rails style of *Convention over Configuration*

## Legacy with conventions has opportunities for Living Documentation

Whenever you have an existing code base that follows conventions, you have an opportunity to go the Living Documentation route by exploiting all these conventions, without even having to touch the source code to add anything, in contrast to Documentation by Annotation.

For example imagine an existing application that follows a Layered Design. If you're lucky its package names represent the layers directly through naming conventions:

```
/record-store-catalog/gui
/record-store-catalog/businesslogic
/record-store-catalog/dataaccesslayer
/record-store-catalog/db-schema
```

Your documentation is already there in the naming of the Java packages (namespaces or sub-projects in C#)

# Document the Conventions

If everyone in the team is familiar with this style, then it's enough. If you simply adopt conventions published by another company, then it's an example of READY-MADE DOCUMENTATION and you just have to create a reference to the external documentation of the sets of conventions in the README file. In practice though I'd always recommend to document the conventions in the README file. Here's an example of documenting conventions as in a real code base:

```
README.txt

This application follows a Layered Architecture, as described here (link).
Each layer has its own package, with the following naming conventions:


/gui/*
/businesslogic/*
/dataaccesslayer/*
/db-schema/*

The GUI layer contains all the code about the graphical user interface. All code\
 responsible for display or data entry must be there.

The business logic layer contains all the domain-specific logic and behavior. Th\
is is where the domain model is. Business logic should only be there and nowhere\
 else.

The data access layer contains all the DAO (Data Access Objects) responsible to \
interact with the database. Any change of storage technology should only impact \
this layer and no other layer (in theory at least :)

The DB Schema contains all the SQL scripts to setup, delete or update the databa\
se schema.

Important Rule: Each layer can only depend on the layers below. No layer can dep\
end on the layer or layers above, this is forbidden! Note that no layer should d\
epend on the DB Schema layer, this is a pseudo layer ;)
```

Some conventions carry a cost, especially when they add noise to the naming. For example putting prefixes or suffixes on identifiers: VATCalculationService, DispatchingManager or DispatchingDTO is a standard practice, but it's not Clean Code. The names in your code do not belong to the business domain language anymore!

When every interface in a package is a service, then adding the *Service* prefix adds no information, just noise. Every class is in a /dto/ package may not need the *DTO* suffix, or it's redundant information.

## Discipline-Driven

Documentation by Convention only works to the extent that everyone has enough discipline to adhere to the conventions consistently. The compiler does not care about your conventions and won't help much to enforce them.

One typo, and you're already not following the convention! You can of course tweak the compiler, your IDE parser or use static analysis tools to detect some violations of conventions. Sometime it's a lot of work, but other times it's surprisingly easy so you may give it a try.

If you rely on Documentation by Convention to help produce Living Documents like Living Diagrams, then it will encourage and reward following the conventions: if you break the convention, then your Living Documents will fail, which is nice.

## Conventions have limitations

Conventions work well to categorize sections of code, but they quickly show their limits when you try to enrich them with additional knowledge like rationale, alternatives etc. In contrast, annotations are a better fit to include such additional knowledge.

Conventions are often little more than free text meant for humans. However you may still get some tool support for your conventions:

- You can configure your IDE with templates for each convention; you type a few characters and it will print the full name properly that adheres to the convention; for a commit comment with a more complicated convention, it will print a placeholder that you can just fill in.
- You can have your Living Document generators interpret the conventions to perform their work
- You can enforce rules like dependencies between layers based on the naming conventions, for example using JDepend or your own tool built on top of any code parser.

Compared to annotations, conventions also have the advantage to not disrupt old habits. If your team and managers are very conservative you may prefer going the Documentation by Convention route rather than the Documentation by Annotation route. But I guess you understood that my preference goes to Documentation by Annotation.

# Sidecar files

(aka buddy files, companion files or connected files)

Sidecar files are files which store metadata that cannot be supported by the source file format. For each source file there is typically one associated sidecar file with the same name but a different file extension.

For example, some web browsers save a web page as a pair of an html file and a sidecar folder of the same name but with a *_folder* prefix. Another example is when a digital camera have the ability to record a piece of audio at the time of taking a picture, then the associated audio is stored as a sidecar file with the same name as the .jpg file, but with a .wav extension.

Sidecar files are like *external annotations*. They can be used to add any kind of information, like a classification tag or a free text comment, without having to touch the original source file on the file system.

The main problem with this approach is that when the file manager are not aware of the relationship between the source file and its sidecar file, then cannot prevent the user from renaming or moving only one of the files without the other, thereby breaking the relationship.

For this reason, I don't recommend this approach unless there is no other choice.

> Old source control systems like cvs used a lot of sidecar files

*Reference:* Side-Car Files[11]

[11] https://en.wikipedia.org/wiki/Sidecar_file

# Metadata Database

A metadata databases is a database which stores metadata referencing other source files or components. A famous example is the iTunes database, which contains a lot of metadata associated to each song and that don't fit within the audio files themselves, like playlists and recent listening history. The metadata may not fit within the file because the file format does not have a place to store them, or because it would not be a good idea to change the file at all.

Another reason could be that the metadata reference the files but are not really intrinsic to it, hence they should be stored somewhere else. For example, a photo should not store that its part of an album, we'd rather have the album being stored somewhere else. In a similar fashion, the url of the thumbnail of a photo is a metadata only of interest for the photo application, and it would be intrusive to corrupt the photo file to add that kind of metadata into its own structure, assuming it was even possible.

The main problem with this approach, just like with sidecar files, is that it's easy for the metadata database and the corresponding files to get out of sync if one of these files is renamed, moved or deleted without updating the database.

A metadata database should be the last resort choice when it's not possible to touch the files at all, therefore all metadata have to be stored elsewhere.

However it is also a convenient approach when the management of the metadata is done in bulk across all files at the same time, by different people than the people managing the files themselves.

For examples, if hundreds of photos are managed by a photographer but the metadata database is a plain spreadsheet managed by a librarian, then it is easy for the librarian to quickly add all the metadata in a column thanks to the ability to copy-paste, interpolate and calculate data that a modern spreadsheet application can offer, independently of the photographer and without risk of corrupting the photo file by mistake.

Common examples of MetadaDB are the various key-value stores embedded in your Discovery registries, Deployments / Configuration / Provisioning tools, Service Catalogues, Bookmarking registries etc. Whenever you can reference something and add tags, you have a de-facto MetadaDB!

# Designing Custom Annotations

*Evolve your vocabulary of tags and annotations from standard literature, with additions and extensions to be more specific to your own context*

Off the shelf literature is essential to learn quickly based on the experience of others, and to share a common vocabulary whatever the company, department or continent you are in. However the problem with such literature is that in order to be shared with everyone it has to give up what is specific to each particular context.

You should use this standard body of knowledge, and you can also extend it to make it even more expressive.

For example we more or less all agree on a standard circle of 6 colors, but in your own visual charter you certainly use custom variants of these colors, which are specific to you. Your "light blue" is certainly a "blue", but it is up to you to define what "light" means.

## Stereotypical Properties

When we design code, we think in terms of working behavior but also in terms of properties, desirable or not. Here are some examples of desirable properties:

- NotNull, for a parameter which cannot be null. Life is so easier when you use it almost always!
- Positive, for a parameter which has to be positive
- Immutable, for a class (or at least observed as immutable). This is not the place to elaborate on all the benefits of immutable objects, but immutable objects are good, eat them!
- Identity by value, where equality is defined as the equality of data
- Pure, a.k.a. Side-Effect Free, for a function, or by extension for every function of a class. This is a good idea to design as much as your code in a pure way.
- Idempotent: for a function which has the same effect when called more than once. This single property is a life-saver for distributed systems.
- Associative, for a function + such as (a + b) + c = a + (b + c). This property is useful when doing map-reduce kind of things.

Whenever we think about these properties, we naturally want to make it clear into the code. We do that with the type system whenever possible. For example it is possible to express the possibility of having no result with the *Option* or *Optional* built-in the language, or provided by a standard libraries. Using a Scala *case class* is in itself a shorthand for *(Immutable, Identity by value).* When it is not possible we express the properties with comments, or with custom annotations, along with automated tests and property-based testing.

## Stereotypes and tactical patterns

In a language like Java or C#, everything is a class, but not every class is of the same kind or has the same purpose. Please note that in FP languages, everything is a function, but not every function has the same purpose either. Domain-Driven Design proposes some fundamental categories of classes, like *Value Object*, *Entity*, *Domain Service* and *Domain Event*. It also suggest borrowing from other patterns like design patterns, with the *Strategy* and the *Composite* patterns mentioned as examples, the point being that some (not all) design patterns are also domain patterns.

What is interesting is that these categories of classes are compressed ways to express a lot of information. For example when I say that the class "FueldCardTransaction" is a Value Object, this means that its identity is only defined by its values, that it is immutable, should be without any side-effect, and should be transferable. It is therefore natural to declare explicitly these patterns as a simple way to do documentation.

Therefore we would introduce a set of custom annotations in our project:

- @ValueObject
- @Entity, or @DomainEntity to prevent any ambiguity with the annotations of similar names from all the technical frameworks
- @DomainService
- @DomainEvent

And we can declare the consequence explicitly by using the properties

Each category of class comes with pre-defined properties. For example a value object should have Identity by value, be Immutable and Side-Effect Free. We can easily make that explicit in our little system of annotations, by putting annotations on annotations:

```
1 @Immutable
2 @SideEffectFree
3 @IdentityByValue
4 public @interface ValueObject {
5 ...
```

When we mark a class as being a value object, we indirectly mark it with the meta annotations as well. This is also a convenient way to group properties which go together into bundles, to declare them all with only one equivalent declaration. Of course the bundle should have a clear name and meaning, not just a random bunch of properties together.

This approach enables additional enforcement of the design and architecture if you wish. For example @DomainEntity, @DomainService and @DomainEvent imply being part of a Domain-Model and perhaps related restrictions on the allowed dependencies as a result, which can all be enforced with static analysis.

As described in the Module-Wide Knowledge section, annotations in Java can be put on packages, so that a declaration in one place collectively marks every element of the package. I like to use that in a "unless specified otherwise" fashion. For example we could define a custom annotation named @FunctionalFirst, meant to be put on whole packages, which would mean @Immutable and @SideEffectFree by default for every type, unless stated otherwise explicitly on a particular type.

There are many other catalogues of patterns and stereotypes of interest to express efficiently a lot of design and modeling knowledge. It is ready-made knowledge and vocabulary about your job as a developer, doing design, modeling and solving infrastructure problems. But you can go further and extend the standard categories into finer grain categories.

For example, it is possible to refine the kind of *Value Object*. Martin Fowler wrote about the *Quantity* pattern, the *NullObject* pattern, the *SpecialCase* pattern, the *Range* pattern, and they are all specialized cases of Value Objects. It goes even further, with the Money pattern being itself a special case of the Quantity pattern. You can use all these patterns, choosing the most specific as possible. For example I would chose *Range* over just *Value Object* if it applies. It is common knowledge that a range is a Value Object.

Again we would make explicit that a Range is a special case of a Value Object with an annotation on the annotation:

```
1 @ValueObject
2 public @interface Range {
3 ...
```

You can also create your own variants. In a past project we had a lot of value objects, but they were more than that. They were also instances of the Policy pattern, the domain pattern equivalent of the Strategy pattern. More importantly, in the business domain of finance, we would usually call them standard market "conventions". So we created our own @Convention annotation, and make it clear it is at the same time a value object and a policy.

```
1 @ValueObject
2 @Policy
3 public @interface Convention {
4 ...
```

## Meaningful annotations package names

When you create a custom annotation, you have to chose its package name. You can chose it to have a meaning. I like to encode the reference place of the idea in the package name. When the annotation is drawn from a book, I would use the book name, or its most used abbreviation: *com.acme.annotation.gof* for the book "Gang of Four", *com.acme.annotation.poeaa* for the book "Patterns of Enterprise Application Architecture", and *com.acme.annotation.ddd* for the book Domain-Driven Design. When it is standard knowledge with no one golden book, I would name after the field: *com.acme.annotation.algebra*.

# Hijacking standard annotations

Many frameworks in Java-land use annotations as a form of configuration, e.g. JPA and Spring are offer the infamous choice between XML and annotations. Even though I advocate annotations for documentation purposes, I am not a big fan of using annotations as an alternative to writing code. I prefer the approach found in some .Net projects like Fluent Nhibernate where we use plain code to define the object-to-relational mappings.

However in Java at the time of writing, we still have to use annotations, unless we prefer XML, which I do not. When we use annotations to drive the behavior of frameworks, the annotations are code indeed, and since most of them relate to infrastructure concerns like persistence or web-service, they often have the annoying habit to pollute domain classes with non-domain noise.

Putting aside this little rant, you probably wonder whether these standard annotations have any documentation value. Since they are code, at the minimum the annotations document what they are doing, just like well-designed code. They tell the *WHAT*.

Let's consider a few examples of particular documentation interest.

- **Stereotyping Annotations (Spring)** This set of annotations includes `@Service`, `@Repository` and `@Controller`. They are used to stereotype classes, in addition to declaring them for registration into the dependency injection mechanism. In fact they alias the `@Component` annotation with more meaning, which is a nice way to hijack these necessary noisy annotations for something more meaningful for humans, not just for Spring.
- **Creating Custom Stereotypes (Spring)**: The approach below also supports your own custom annotations, provided you annotate them with the `@Component` meta annotation.
- **@Transactional (Spring)**: This `@Transactional` annotation is used to declare the transactional boundaries and rules, typically on a service. If you have an Hexagonal Architecture, then the transactional services should be your Application Services, in their own thin layer on top of the domain model. You could thus decide that this Spring annotation in itself also means `@ApplicationService` in the DDD sense. Because most Spring annotations are also meta annotations, you could actually define your own `@ApplicationService` annotation, mark it as `@Transactional` in order to express your intent in a way that Spring can recognize to do its magic too.
- **@Inheritance (JPA)**: This `@Inheritance` annotation and its friends directly documents the design decisions on how to do the mapping between a class hierarchy and a corresponding database schema. This directly relates to the corresponding patterns from Martin Fowler's book "Patterns of Enterprise Application Architecture". For example `@Inheritance(strategy=JOINED)` corresponds to the Single Table Inheritance[12] pattern, under another name unfortunately.
- **RESTful Web Service (JAX-RS Annotations)**: This set of annotations is clearly declarative: `@Path` identifies the URI path, `@GET` declares the GET request method, and `@Produces`

[12] http://martinfowler.com/eaaCatalog/singleTableInheritance.html

defines the media type as a parameter. The resulting code is self-documented to a large extent. Furthermore, tools like Swagger can exploit these annotations to generate a living documentation of the API.

It is possible to rely on the standard annotations for their particular documentation value, but this is almost always limited to technical concerns, where the annotation is just like particularly declarative code: it tells the **WHAT**, not the **WHY**. But as we mentioned, it is sometime possible to extend the standard mechanism to convey additional meaning, while still playing nice with the frameworks you depend on.

## Standard Annotation @Aspect and Aspect-Oriented Programming

The Spring Pet Clinic demonstrates Aspect-Oriented Programming by showing how to setup a simple Aspect that monitors call count and call invocation time *for every repository*.

What's interesting here is how the requirement "to monitor every repository" is described *literally* in the aspect declaration. This is possible because the code was augmented with the meaningful '@Repository' stereotype beforehand.

Spring AOP with @Aspect defined around annotated classes:

```
1  @Aspect
2  public class CallMonitoringAspect {
3    ...
4    @Around("within(@org.springframework.stereotype.Repository *)")
5    public Object invoke(ProceedingJoinPoint joinPoint) throws Throwable {
6      ...
7    }
8  ...
9  }
```

See spring-petclinic-gemfire on Github[13] for an example of this.

This illustrates perfectly how augmenting the code with explicit design decisions makes it possible to talk to tools the way we think, literally.

[13] https://github.com/bijoych/spring-petclinic-gemfire/blob/master/src/main/java/org/springframework/samples/petclinic/util/CallMonitoringAspect.java

## Annotation by default or unless necessary

When designing custom annotations to express properties, there's the choice of creating the annotation for the case when the property is met or when it's not:

- @Immutable or @Mutable
- @NonNull or @Nullable
- @SideEffectFree or @SideEffect

You may create both and let everyone decide which one to choose, but it may end up inconsistent, in which case no annotation means nothing at all.

You may decide on the alternative that you want to promote, so that having the annotation in many places becomes a marketing campaign: for example @NonNull everywhere will encourage making everything non-null. No annotation then suggests it's nullable.

Or on the other hand you may consider that annotations are noise, hence the fewer annotations the better. In this case the default and preferred choice should need to annotation. You'd only add an annotations to declare a deviation from the default: (default is Immutable) Oh, this class is exceptionally @Mutable!

# Module-Wide Knowledge

*Knowledge that spans a number of artifacts that have something in common is best factored out in one place.*

In a software project, a module contains a set of artifacts (essentially packages, classes and nested modules) that can be manipulated together. On each module we can define properties that apply to all the elements it contains. Design properties and quality attributes requirements, e.g. being read-only, serializable, stateless etc. often apply on a whole module, not just on distinct element within.

We can also define the primary programming paradigm at the module level: Object-oriented, functional, even procedural or reporting style.

A module is also ideal to declare architecture constraints. For example we would have distinct areas for code written from scratch with high quality standard, and for legacy code with more tolerant standards. In each module we can define preferences of style like Checkstyle configuration, metrics thresholds, unit test coverage, and allowed or forbidden imports accordingly.

**Therefore: When there is additional knowledge that spans a number of artifacts equally within a module, put this knowledge at the module level directly, with the meaning that it applies to all the contained elements. This approach can also be applied to all elements satisfying a given predicate, as long as you can find a home for this declaration, like the pointcuts in aspect-oriented programming.**

## Many kinds of modules

Packages are the most obvious modules in Java and in other languages. But a package x.y.z actually defines more than one module: the module of its direct members x.y.z.*, and the module that also include every artefact included in its subpackages: x.y.z.**

Similarly, a class also represents a "module" for its member fields, methods and nested classes: x.y.z.A#*, and x.y.z.$*.

Eclipse "working sets" also define another kind of logical grouping similar to modules, as simple collections of classes and other resources. Tools like Ant also define filesets using lists of files and regular expressions: {x.y.z.A, x.y.z.B, x.y.*.A}. Like modules, working sets and filesets are usually named for easy reference.

Source folders, e.g. src/main/java or src/test/java obviously define coarse-grain grouping of elements too. Maven modules define bigger modules, at the scale of sub-projects. Pointcuts of Aspect Oriented Programming also defines logical groupings elements across various "real" modules.

Inheritance and implementation implicitly define modules too, such as "every subclass of a class or implementation of an interface": x.y.z.A+, and if it includes every member of every nested member: x.y.z.A++.

Stereotypes implicitly define the set of their occurrences i.e. the pattern ValueObject implicitly defines the logical set of every class that is a ValueObject.

Collaboration patterns such as *Model-View-Controller* and *Knowledge Level* also imply logical groupings such as the Model part of the MVC, or each level of the Knowledge Level pattern: KnowledgeLevel or OperationalLevel.

Design patterns also define logical groupings by the role played within the pattern, e.g. "Every abstract role in the Abstract Factory pattern": @AbstractFactory.Abstract.*.

There are many other modules or quasi-modules implied by concepts like layers, domains, bounded contexts and aggregate roots.

The problem with large modules is their huge number of items, which often necessitates aggressive filtering, which may even require ranking to only consider the N-most important elements out of many more.

# In practice

All the techniques to augment the code with additional knowledge apply for module-wide knowledge: annotations, naming conventions, sidecar files, metadata database, or DSL.

A common way to add documentation to a Java package is by using a special class named *package-info.java* as a location for the Javadoc and any annotation about the package. Note that this special pseudo-class with a magic name is also itself an example of a sidecar file.

In C# modules are often projects, which can have AssemblyInfoDescriptions:

```
AssemblyInfoDescription("package comment")
```

In most programming languages, package or namespace naming convention can also be used to declare a design decision. For example *something.domain*, can be used to mark the package or namespace as a domain model.

# Intrinsic Knowledge Augmentation

*Only annotate elements with knowledge that is intrinsic to them*

This section is more abstract than usual. The concept is important but subtle. If abstract non-sense is definitely not your thing, you can safely skip it and perhaps come back to it later.

It is important to make the distinction between what things really are for themselves, as opposed to what they are for something else or for a purpose. A car is red, is a coupé, or has an hybrid engine. These properties are really *intrinsic* to the car, they are part of its identity. In contrast, the owner of the car, its location at a point in time, or its role in a company fleet are *extrinsic* to the car. This extrinsic knowledge is not really about the car in itself, but about a relationship between the car and something else. As a consequence it can change for many reasons other than the car itself. Thinking about intrinsic versus extrinsic knowledge has many benefits, for design and for documentation.

If only intrinsic knowledge is attached to an element, then:

- If you were to delete the element, the attached knowledge would go away with it without regret and without modification anywhere else. For example, when the car is recycled, its serial number is crunched at the same time and it is ok.
- Any change that it is not intrinsically about the element would not modify the element or its artifacts at all. For example, selling the car should not modify its user manual.

> I've first learnt about this notion of intrinsic versus extrinsic in the GoF book "Design Patterns", in the introduction of the Lightweight pattern. The chapter considers a glyph used in a word processor. Each letter in the text is printed on the screen as a glyph, the rendered image of a character. A glyph has a size and style attributes like italics or bold. A glyph also has a (x, y) position on the page. The core idea behind the Lightweight pattern is to exploit the difference between intrinsic properties of the glyph: its size, style, versus the extrinsic properties like its position on the page, in order to reuse the same instance of a glyph many times on the page.
>
> This explanation did have a big influence on the way I design since then. Since we don not talk about it often, it is a secret ingredient to improve the long term relevance of design decisions.

**Therefore: Only annotate elements with knowledge that is intrinsic to them. Conversely, consider attaching all intrinsic knowledge to the element itself. Avoid attaching knowledge**

**that is extrinsic, as it will change often and for reasons unrelated to the element. A focus on intrinsic knowledge will reduce the maintenance efforts of the documentation over time.**

You may think of this as a matter of more or less judicious coupling. The key question which we ask once again is: "How would my declared knowledge would have to evolve when I change the element?" The best approach is the one such as you have less work when it changes.

The common use of annotations by popular frameworks regularly does not consider that. For example you have a class which exists in itself and that can be used independently, but then you put annotations on it to declare how it is supposed to be mapped to the database or to declare that it is the default implementation for some interface. If you consider this class to really represent a domain responsibility, then this DB mapping is an unrelated concern; having it attached only makes the class more likely to change for DB reasons too.

Imagine you have a CatalogDAO interface, with two implementations: MongoDBCatalogDAO and PostgresCatalogDAO. Marking the MongoDBCatalogDAO class as the *default implementation of the CatalogDAO interface* is another example of an extrinsic concern forced on the class. The alternative would be to annotate each DAO with an intrinsic attribute like @MongoDB or @Postgres, and separately make the selection *indirectly* via this intermediate attribute.

For example we would mark all MongoDB*DAO implementation with the @MongoDB annotation, and all Postgres*DAO with the @Postgres annotation. This is intrinsic knowledge with respect to the DAO. Separately we would decide to inject every implementation for the technology chosen for a particular deployment. If we deploy with Postgres we would want to inject every @Postgres implementation. This decision to inject one technology is knowledge too, that is deto the DAO.

# Inspiring Exemplars

*The best documentation on how to write code is often just the code which is already there*

When I'm coaching teams on TDD, I pair-program randomly with developers on code-bases I have never seen before. What surprises me is that the developers pairing with me also behave as if they had never seen the code base before: for a new task, they go looking for an example of something similar already there, and then they copy-paste it into a new case. The default heuristics I've seen was "I'll find a service written by Fred", where Fred is the Team Lead who is well respected by the rest of the team. The problem is when Fred is not so good in every aspect of his code, as the flaws in his code are replicated across the whole code base.

As a consequence, a good way to improve the code quality is just to improve the examples of code that people imitate. We need exemplary code, serving as a desirable model to imitate, or at least to inspire the influenceable developers

Sam Newman writes about that in his book on building services:

> If you have a set of standards or best practices you would like to encourage, then having exemplars that you can point people to is useful. The idea is that people can't go far wrong just by imitating some of the better parts of your system.

You can point your colleagues to the exemplars during conversations, for example during pair-programming or mob-programming: "Let's look at the class ShoppingCartResource, it's the most well-designed and it's exactly in the style of code we favor as a team".

Conversations are perfect for that, but some additional documentation can have benefits too, when you are not present to point people to the exemplars, or when people are working on their own.

**Good example of code here!**

Therefore: *Highlight directly in the actual production code the places which are particularly good exemplars of a style or of a best practice you would like to encourage. Point your colleagues to these exemplars, and advertise how to find them on their own. Take care of the exemplars so that they remain exemplary, for everyone to imitate in a way that will improve the overall codebase.*

Annotations are of course a perfect fit for that: you can create a custom `@Exemplar` annotation to put on the few classes or methods which are the most exemplary. Of course exemplars are only useful if there is only a handful of them.

As usual, decisions on what code is exemplary or not are best taken collectively by the team. Make it a team exercise to find a consensus on the few exemplars to highlight with a special annotation.

Exemplars should be actual code used in production, not tutorial code, as Sam Newman says in his book on Building Microservices: "Ideally, these should be real world services you have that get things right, rather than isolated services that are just implemented to be perfect examples. By ensuring your exemplars are actually being used, you ensure that all the principles you have actually make sense."

In practice, an exemplar is hardly perfect in all aspects, it can be a very good example of design, but the code style may be a bit weak. Or other way round. My preferred solution would be to fix the weak aspect first. However if it's not possible or desirable, at least clarify why the exemplar is good, and what aspect of it should not be considered exemplary. Here are a few examples of exemplars:

- `@Exemplar("A very good example of a REST resource with content negotiation and the use of URI-templates")` (on a class)
- `@Exemplar("The best example of integrating Angular and Web Components")` (on a js file)
- `@Exemplar("A nicely designed example of CQRS")` (on a package or a key class of this part of design)
- `@Exemplar(pros = "Excellent naming of the code", cons = "too much mutable state, we recommend immutable state")` (on a particular class)

Basically, marking the exemplars directly in the code enables asking your IDE "what code is a good example of writing a REST resource?". In an *Integrated Documentation* fashion, finding exemplars is only a matter of searching for all references of the `@Exemplar` annotation in your IDE. You can just scroll the short list of results to decide which code will be your inspiration for your task.

Of course there are caveats in the approach suggested before :

- **Software development is not supposed to be so much copy-pasting** as thinking and solving problems. Highlight exemplars is not a free license to copy-paste code.
- **Copy-pasting requires refactoring**. As similar code accumulates, it must be refactored.
- **Marking the exemplars in the code is not meant to prevent asking colleagues** for exemplary code. Asking questions is good as it leads to conversations, and conversations are

key for improving the code and the skills. Don't reply "RTFM" (Read The F**ing Manual) when asked for exemplars. Instead, why not go through the suggested exemplars in the IDE together, reviewing which one would be best for the task? Always take conversations as opportunities to improve something mutually.

# Machine Accessible Documentation

*Documentation that is machine-accessible opens new opportunities for tools to help at design level*

You code at design level, not just code level, but your tools cannot help you much at design level. They cannot help because they have no idea what you are doing on the design perspective from the code alone. If you make your design explicit, for example using annotations attached to the code, then tools can begin to manipulate the code at the design level too, and help you more.

Design knowledge that can make the code more explicit is worth adding. An annotations attached to the language element is often enough, e.g. you can declare the layers on each top-level package, in the corresponding package-info.java file:

```
@Layer(LayerType.INFRASTRUCTURE)
package com.example.infrastructure;
```

By putting the annotation @Layer on the package com.example.infrastructure, you declare a particular instance of the pattern Layer, where the layer is the package itself.

As usual there are many options to design a custom annotation, like this declaring an id (this may be useful to reference it later):

```
@Layer(id = "repositories")
package com.example.domain;
```

With this design intent made explicit in the code itself, tools like a dependency checker could now automatically derive forbidden dependencies between layers, to detect when they are violated.

You could do that with tools like JDpend, but you'd have to declare each package-to-package dependency restriction. This it tedious and does not directly describes the layering, just the consequence of the layering.

Declaring every forbidden or acceptable package-to-package dependency is tedious, but now imagine doing it between classes: it's prohibitive! However, if classes are tagged, e.g. as @ValueObject, @Entity or @DomainService, now dependency checkers can enforce our favorite dependency restrictions. For example I like to enforce the following rules:

```
Value Objects should never depend on anything other than other Value Objects.
Entities should never have any Service instance as member field.
```

Once the classes are augmented with these stereotypes explicitly, we could now tell the tools what we want more literally and more concisely.

# Literate programming

> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do. – Donald Knuth literateprogramming.com[14]

It is hard not to mention Literate Programming in a book on Living Documentation. Literate programming is an approach to programming introduced by Donald Knuth. Wikipedia defines this approach as follows:

> A literate program is an explanation of the program logic in a natural language, such as English, interspersed with snippets of macros and traditional source code.

A specific tool processes the program and produces both a document for humans, and compilable source code that becomes the executable program.

This is from 1984, and although it never really became widely popular, it had a profound and widespread influence in the industry, even if the idea had been often distorted.

Literate Programming introduced several important ideas:

- **Documentation interleaved with the code**, in the same artifacts, with code inserted within the prose of the documentation. This should not be confused with documentation generation, where the documentation is extracted from comments inserted into the source code.
- **Documentation following the flow of thoughts of the programmer**, as opposed to being constrained by the compiler-imposed order. A good documentation should follow the order of the human logic.
- A programming paradigm to **encourage programmers to think deliberately** at each decision they are making. Literate programming goes well beyond documentation: it is meant to be a tool to force programmers to think deliberately, as they have to explicitly state their thoughts behind the program.

Also keep in mind that literate Programming is not a way to do documentation but a way to write programs.

Literate Programming is well alive today, with tools available for all good programming languages like Haskell, Clojure and F#. The focus now is on writing prose in Markdown, with snippets of

[14] http://www.literateprogramming.com

programming language inserted. In Clojure you would use Marginalia[15], in CoffeeScript you would use Docco[16], while in F# you would use Tomas Petricek's FSharp.Formatting[17].

## Other similar approaches

Traditionally, documentation of a software program involves a mix of code and prose, which can be combined in several ways:

- **Code in prose**: the original Literate Programming as proposed by Donald Knuth. The primary document is prose following the human logic of the programmer. The author-programmer has full controls of the narration.
- **Prose in code** this is the documentation generation approach offered by most programming languages, like Javadoc and its equivalents.
- **Separate code and prose**, **merged into one document by a tool**. Tools then perform the merge in order to publish a document, e.g. a tutorial.
- **Code and prose as the same thing**. In this approach, the programming language is so clear it can be read as prose itself. Unfortunately this Holy Grail is never reached, but some programming languages get closer than others. I've seen some F# code by Scott Waschlin which can be impressively close to this ideal.

Some tools, like Dexy[18], give the choice of how you prefer to organize the code and the prose with each other.

[15] https://github.com/gdeer81/marginalia
[16] http://jashkenas.github.io/docco/
[17] https://github.com/tpetricek/FSharp.Formatting
[18] http://www.dexy.it/

# Record Your Rationale

In the book "97 Things Every Software Architect Should Know", Timothy High says: "As explained in the axiom "Architectural Tradeoffs", the definition of a software architecture is all about choosing the right tradeoffs between various quality attributes, cost, time, and other factors." Replace the word architecture with design, or even with code, and the sentence still holds.

There are tradeoff everywhere in software, whenever a decision is being made. If you believe you're not doing any tradeoff, it just means the tradeoff is out of sight.

Decisions belong to stories. Humans love stories, and remember them better. Decisions should remember their context. The context of past decisions is necessary to re-evaluate them in the new context. Past decisions are learning tools to learn from the thinking of the predecessors. Many decisions are also more compact to describe than their consequences, hence they are easier to transfer from one brain to another than all the details that result from the decision. If you tell me your intent and the context shortly, and provided I'm a skilled professional, I may come up with the same many decisions that the ones you've made.

**Therefore: Record the rationale of all important decisions in some form of persistent documentation. Include the context and the main alternatives. And "Listen to the documentation": if you find it hard to formalize the rationale and the alternatives, then it may be that the decision was not as deliberate as it should have been. You may be programming by coincidence!**

## What's in a rationale?

Any decision happens in a context, and is one of the considered answers to a problem. Therefore a rationale is not only the reason behind the chosen decision, but also

- The **context** at the time: main stakes and concerns, for example the current volume: "Only 1000 end users using the application once a week" or the current priority: "Priority is exploring the market-product fit as quickly as possible", or an assumption: "This is not expected to change", or a people consideration: or "The development teams don't want to learn Javascript"
- The **problem** or **requirement** behind the choice: "The page must load in less than 800ms to not lose visitors", or "Decommission the VB6 module"
- The decision itself of the **chosen solution**, with the main reason or reasons: "The Ubiquitous Language is expressed with English words only, as it's simpler and every current stakeholder prefers it that way", or "This facade exposes the legacy system through a pretty API, because there is no good reason to rewrite the legacy but we still want to consume it with the same convenience as if it was brand new"

- The **main alternatives** that were considered seriously, and perhaps why they were not selected, or why they would be selected if the context was different: "Buying an off-the-shelves solution would be a better choice if the needs were more standard","A graph structure would be more powerful but is harder to map with the Excel spreadsheets of the users", "A NoSQL datastore would be a better choice, if we didn't have all this investment with our current Oracle DB"

  Generally speaking design rationale is very much about discarded options, so *not in the code* – @CarloPescio on Twitter in a conversation on self-documenting code

**What were they thinking?**

## Make it explicit

- **Ad hoc document**: Explicit document about the requirements, including all quality attributes. This needs to evolve slowly but still at least once a year; only done for the main attributes which span large areas of the system, not for more local decisions
- **Annotations** They can be standalone or with a reference to the requirements; they can evolve with most the refactoring, but not strictly always, so you may still need some maintenance in the infrequent case when the rationale itself changes

- **Blog post**: A blog post takes more time to write, and the best writing style the better; it also has to be searched and scanned when a question arises on a past decisions. However in return you have a human account of the reasoning and the human context behind a decision, perhaps even with the politics and personal agenda mentioned between the lines, which is more valuable.

## Beyond Documentation: Motivated Design

Recording the rationale is not just for the future generations or your future self, it is also useful right now, at the time of doing it. As usual, listen to what's hard as a signal that something could be improved. It if it hard to come up with rationale, or its context, perhaps the decision was not thought about seriously enough, and this should be an alert.

If it is hard to come up with 2 or 3 credible alternatives, then perhaps the first solution that fit was chosen without the diligence to explore possibly simpler or better solutions. Your current decision may not be optimal, and it may have consequences as lost opportunities in the future. Of course one rationale can always be "First solution that fit was chosen to go to market as quickly as possible", but at least it is deliberate and we understand the consequence, like being well aware that we will reconsider it next time.

With or without decision, structure just is. In the worst case, in the absence of deliberate design decisions, and with a complete lack of skills, there will be just a random structure. It's just a soap of details, and the only way out is guessing the intentions behind that. This is typical in legacy code, which we discuss in details in a next chapter.

## Don't document speculation

In the book "Building Microservices", Sam Newman advices against documenting solutions to speculative needs. He's got a shot at the traditional architecture documentation "Diagram after diagram, page after page of documentation, created with a view to inform the construction of the perfect system, without taking into account the fundamentally unknowable future. Utterly devoid of any understanding as to how hard it will be to implement, or whether or not it will actually work, let alone having any ability to change as we learn more."

Rationale are decisions taken on actual needs, proven to be necessary. In incremental approaches, as in emerging design, we grow the solution slice by slice, where each slice is driven by the most important need at each instant. We often work in a just-in-time fashion, precisely because it is an antidote to speculation: we build it just when it becomes necessary to be built.

Overall, only document what has been built, in response to actual needs.

## Skills as pre-documented rationales

There are many smaller decisions where the thinking process is already solved and documented in the literature. For example, the Single Responsibility Principle mandates to split a class which does two things into two classes which do one thing each. There is no need to document each occurrence of that happening, but you may document once, in a single place, all these principles you consistently follow, as explained in the **Acknowledge your Influences** pattern in this book.

## Recording the rationale as an enabler for change

**Without the why, they will make the same mistake again**

It's easier to dare making changes when you know all the reasons behind the past decisions, so that you can respect then or reject them deliberately. The best way to know them in a reliable way is to have them recorded, otherwise the reasoning will be forgotten. Without the explicit rationale behind the past decisions, one can just wonder if a change may have unexpected impacts with respect to a concern we don't have in mind. If you're prudent, you'll never be sure enough to decide to change,

and the status quo dominates, even though the opportunity to improve is there in front of your eyes. If you're not prudent, you may actually cause harm inadvertently because of a forgotten concern that we cannot see as it was not recorded.

# Commit Messages as Comprehensive Documentation

*Careful commit messages make each line of code well-documented*

When committing files into source control, it is good practice to add a meaningful comment, the commit message. This is often neglected, in which case you end up wasting time opening the files to discover what the change was about. However when done carefully commit messages become very valuable for several purposes, as yet another HIGH-YIELD ACTIVITY:

- **THINK** You have to think about the work done. Is it one single change or a mix of more than one that should be split? Is it clear? Is it really done? Are there new tests that should have been added or modified along the changes?
- **EXPLAIN** The commit message must make the intention explicit. It is a *feature*, or a *fix*, and the reason should be written, even briefly, as in RECORD THE RATIONALE. This will save time for all readers.
- **REPORT** The commit messages can later be used for various kinds of reporting, published like a *changelog* or integrated in the developer toolchain.

The big idea with commit messages is that on any given line of code, asking the source control for its history gives you a detailed list of reasons and, hopefully, of rationales explaining why this line of code is what it is. As Mislav Marohnić writes in his blog post Every line of code is always documented[19], "a project's history is its most valuable documentation."

Looking at the history of a given line of code tells you who did the change, when, and what other files were changed together, like tests. This helps pinpoint to the new test cases that were added, acting as a built-in mechanism for *code to test* traceability. In the history you would also find the commit message explaining the change and the reasons for the change.

To get the best of commit messages, it may be a good idea to agree on a standard set of commit guidelines if the current quality of the messages is not good satisfactory. Using a standard structure and standard keywords have several benefits. It is more formal, and therefore more concise. With a formal syntax we can write:

```
fix(ui): change the color of the submit button to green
```

which is shorter to write and to read than the equivalent full English sentence:

[19] http://mislav.uniqpath.com/2014/02/hidden-documentation/

> "This is a fix on the UI area, to change the color of the submit button to green

In many cases and depending on the writing skills of the committer, the structure message may be less ambiguous. More importantly, the structure message enforces that the required information like the "type of commit" and "location of the change" will not be forgotten. And a formal syntax turns the messages into machine-accessible knowledge, for even more goodness!

**Therefore: Take care of the commit messages. Agree on a set of commit guidelines, with a semi-formal syntax and a standard dictionary of keywords. Work collectively, or use peer pressure, code reviews or enforcement tools to ensure the guidelines are respected. Design the guidelines so that tools can use them to help you more.**

Commit messages are comprehensive documentation for each line of code. It is available in command line, or on the graphical interface on top of your source control, like shown below.

**The blame view on Github shows every contribution for each line, here shown for the famous Junit project**

# Commit Guidelines

A good example of such guidelines is the Angular commit guidelines[20], which has strict rules over how the commits messages must be formatted. From the Angular website

> We have very precise rules over how our git commit messages can be formatted. This leads to more readable messages that are easy to follow when looking through the

[20] https://github.com/angular/angular.js/blob/master/CONTRIBUTING.md#commit

project history. But also, we use the git commit messages to generate the AngularJS change log.

In this particular set of guidelines, the commit message must be structured as a header section, an optional body section and an optional footer section, each separated by a blank line.

```
<type>(<scope>): <subject>

<body>

<footer>
```

The type must be one of the following types:

- feat: A new feature
- fix: A bug fix
- docs: Documentation only changes
- style: Changes that do not affect the meaning of the code (white-space, formatting, missing semi-colons, etc)
- refactor: A code change that neither fixes a bug nor adds a feature
- perf: A code change that improves performance
- test: Adding missing tests
- chore: Changes to the build process or auxiliary tools and libraries such as documentation generation

All breaking changes must be declared in the footer, starting with the word *BREAKING CHANGE*, followed by a space and the detailed explanation of the change and of the migration aspects.

If the commit is related to issues in a tracker, the issue should be referenced in the footer as well, with the identifier of the issue in the tracker.

Here is an example of a feature related to the scope "trade feeding":

```
feat(tradeFeeding): Support trade feeding for negative-coupon bonds

Some bonds have negative coupon rates, e.g. -0.21 percents. Change the validatio\
n to not reject trades on bonds with negative coupons.

Closes #8125
```

## Scope of the change

The little syntax before is semi-formal, with a combination of keywords and free text. The first keyword denotes the type of change (feature, fix etc.) out of a small list. The second keyword denotes the scope of the change in the system or application, and is specific to your own context.

These scope can cover various aspects of your system:

- Environment: prod, uat, dev
- Technology: rabbitmq, soap, json, puppet, build, jms
- Feature: pricing, authentication, monitoring, customer, shoppingcart, shipping, reporting
- Product: books, dvd, vod, jewel, toy
- Integration: twitter, facebook, g+
- Action: create, amend, revoke, dispute

Your commit guideline could require on mandatory main scope, but you could add some more, like:

```
feat(pricing, vod): increase the rate on prime time
...
```

Of course these lists of scopes has to be defined by yourself, ideally as a whole team, including the 3+ amigos, and everyone involved in the devops close collaboration. Every change which could be committed to the source control should be covered in at least one of the scopes.

Going further, a smart list of scopes opens the door to reasoning about impacts.

## Machine-Accessible information

A semi-formal syntax for the commit messages also has the benefit of making it possible for machines to make use of them to automate more chores, like generating the *changelog*[21] document. Let's have a closer look again at Angular.js which is a neat example in this area.

Under Angular.js conventions, the change log is made of three optional sections for each version, where each section is only shown when it is not empty:

- new features
- bug fixes
- breaking changes

Below is an excerpt from an Angular.js changelog (links have been removed here):

[21] http://keepachangelog.com

# 0.13.5 (2015-08-04)

### Bug Fixes

- **file-list**: Ensure autowatchDelay is working. (655599a), closes #1520
- **file-list**: use lodash find() (3bd15a7), closes #1533

## Features

- **web-server**: Allow running on https (1696c78)

The change log is in the Markdown format, which enables links for convenient navigation between commits, versions and ticketing systems. For example, each version in the changelog links to the corresponding *compare* view in Github, showing the differences between this version and the previous one. Each commit message also links to their particular commits, and even links to the corresponding issue(s) when applicable.

Thanks to this kind of structured commit guidelines, it is possible to extract and filter the commits through command line magic, as shown in the example below borrowed from the Angular.js documentation:

```
List of all subjects (first lines in commit message) since last release:
>> git log <last tag> HEAD --pretty=format:%s

New features in this release
>> git log <last release> HEAD --grep feature
```

The changelog shown above can be generated by a script when doing a release. There are many open-source projects to do that, and you can create, like the conventional-changelog[22] project. This changelog automation script relies strongly on your chosen commit guidelines, and already supports several of them (atom, angular, jquery etc.). It is smart enough to filter out the commits and their revert when it happens.

[22] https://github.com/ajoslin/conventional-changelog

**LOL**

The Queen's speech is like the release notes for a minor new version of the UK! (from Matt Russell on Twitter)

This automation is convenient, even if a human should review and edit the generated changelog skeleton before actual release to the public.

# Dynamic Curation

*Too much information is as useless as no information.*

It's not because all the works of art are already there in the collection that there is nothing to be done to make an exhibition out of it.

In art exhibitions, the curator is as important as the director in a movie. In contemporary art, the curator selects and often interprets works of art. For example the curator searches for the prior work and places which were an inspiration for the artist, and he or she proposes a narrative or a structured analysis that links the selected works together in a way that transcends each individual piece. When a work that is essential for the exhibition is not in the collection, it will be borrowed from another museum or from a private collection, or sometime even commissioned to the living artist. In addition to selecting works, the curator[23] is responsible for writing labels, catalog essays, and oversees the scenography of the exhibition which helps convey the chosen messages.

When it comes to documentation, we need to become our own curators, working on all the knowledge that is already there to turn it into something meaningful and useful.

Curators select works or art based on many objective criteria like the artist name, date and place of creation or the private collectors who first bought them. They also rely on more subjective criteria like the relationships to Art Movements or to majors events in the History like wars or popular scandals. The curator needs the metadata about each painting, sculpture or video performance. When these metadata are missing, the curator has to create them, sometime by doing researches.

Curation is something that you already do, perhaps without being aware of it, for example when asked to demo the application to a customer or to a top manager. You have to chose just a few use-cases and screens to show in order to convey a message like "everything is under control" or "buy our product because it will help you do your job". If you have no underlying message, it's likely that your demo with be an unconvincing mess.

But in contrast to art exhibitions, in software development what we need is more like a living exhibition with content that adjusts according to the latest changes. As the knowledge evolves over time, we need to automated the curation on the most important topics.

**Therefore: Adopt the mindset of a curator, to tell a meaningful story out of all the available knowledge in the source code and artifacts. Don't select a fixed list of elements. Instead, rely on tags and other metadata in each artifact to dynamically select the cohesive subset of knowledge that is of interest for the long term. Augment the code when the necessary metadata are missing, and add the missing pieces of knowledge when they are needed for the story.**

[23] https://en.wikipedia.org/wiki/Curator

Curation is the act of selecting relevant pieces out of a large collection in order to create a consistent narrative that tells a story. It's like a remix or a mashup.

Curation is key for knowledge work like software development. Our source code is full of knowledge about many facets of the development and of various importance. On anything bigger than a toy application, extracting knowledge from the source artifacts immediately overflows our regular cognitive capabilities with too many details, becoming totally meaningless hence useless. Too much information is as useless as no information at all.

The solution is to aggressively filter the signal from the noise for a particular communication intent. What would be the noise in a particular perspective would be the signal in another perspective. For example the method names are an unnecessary detail in an architecture diagram whereas they may be important in a close-up diagram about how two classes interact with one being an Adapter to the other.

Curation as its core is the selection of pieces of knowledge to include or to ignore according to a chosen editorial perspective. It's a matter of scope. Dynamic curation goes one step further, with the ability to do the selection continuously on an ever-changing set of artifacts.

## Examples

A Twitter search is an example of an automated dynamic curation, and it is a resource in itself that you can follow just like you would follow any Twitter handle. People on Twitter are also doing curation indeed, but a manual form of curation, when they retweet content they have (more or less) carefully selected according to their own editorial perspective (if any).

A Google search is another example of a simple automated curation.

Selecting an up-to-date subset of artifact based on a criteria is something we do everyday when using an IDE:

- Show every type with a name that ends with "DAO"
- Show every method that calls this method
- Show every class that references this class
- Show every class that references this annotation
- Show every type that is a subtype of this interface

When a tag is missing to help select the pieces, it's the right time to introduce it with annotations, naming conventions or any other mean. When a piece of knowledge is missing in order to show a complete picture, then it's the right time to add it too, in a just in time fashion.

# Editorial curation

Curation is an editorial act. Deciding on an editorial perspective is the essential step. There should be one and only one message at a time. A good message is a statement with a verb, like "No dependency is allowed from the domain model layer to the other layers", rather than just "Dependencies between layer" where there is no message and it's up to the reader to guess what is to understand. At the minimum, a dynamic curation should be named with an expressive name reflecting the intended message.

# Low-maintenance dynamic curation

Selecting subsets of knowledge can be hazardous if done in a rigid way. For example a direct reference to a list of classes, tests or scenarios will rapidly becomes obsolete and will require maintenance. It is a form of copy-and-paste, making change more expensive and with the risk to forget to update it. This is not a good practice and it should avoided at all cost.

Avoid direct reference to artifacts by their name or URL.

Instead, find mechanisms to select pieces of knowledge based on criteria that are stable over time, so that the selection will remain up-to-date without any manual action:

Select artifacts indirectly, based on stable criteria.

You can describe the artifacts of interest in a stable way by using one of the stable selection criteria described below.

- **Folder organization**: "everything in the folder named 'Return Policy'"
- **Naming conventions**: "every test with 'Nominal' in its name"
- **Tags or Annotations**: "every scenario tagged as 'WorkInProgress'"
- **Links Registry** that you have control on: "the URL registered under this shortlink". The registry may need some maintenance from time to time, but at least it is in a central place
- **Tool Output**: "every file that has been processed by the compiler, as visible in its log"

By using stable criteria, the work is done by tools which automatically extract the latest content that meets the criteria to insert it into the published output. Because it is fully automated, this can be ran as often as possible, perhaps continuously on each build.

# One corpus of knowledge for multiple uses

Everything can be curated. Code, configuration, tests, business behavior scenarios, datasets, tools data etc. This can be considered as a huge corpus, accessible via automated means for analysis and curated extractions.

Provided the content of the knowledge corpus is adequately tagged, it is possible to extract by curation out of it a business view of a glossary (see Living Glossary), a technical view of the architecture (see Living Diagram), and any other perspective one can imagine:

- Audience-specific content, like business readable content only vs. technical details
- Tasks-specific content, like "how to add one more currency"
- Purpose-specific content, like Overview content vs. Reference section

Curation is only possible to the extent that metadata about the source knowledge are available to enable relevant selection of material of interest.

A good example of dynamic curation is **Scenario Digest**, where the corpus of business scenarios is curated under various dimensions in order to publish reports tailored for particular audiences and purposes.

# Scenario Digest

Curation is not just code, it's also about tests and scenarios.

When a team makes use of BDD together with an automated tool like Cucumber, a large number of scenarios are written in feature files. Not all scenario is equally interesting for everyone and for every purpose, so we need a way to do a dynamic curation of the scenarios, and for that we need to have the scenarios marked with a nicely designed system of tags.

Remember:

> Tags are documentation.

Each scenario can have tags like the following:

```
@acceptancecriteria @specs @returnpolicy @nominalcase @keyexample
Scenario: Full reimbursement for return within 30 days
  ...

@acceptancecriteria @specs @returnpolicy @nominalcase
Scenario: No reimbursement for return beyond 30 days
  ...

@specs @returnpolicy @controversial
Scenario: No reimbursement for return with no proof of purchase
  ...

@specs @returnpolicy @wip @negativecase
Scenario: Error for unknown return
  ...
```

Note that almost all the tags are totally stable and intrinsic to the scenario they relate to. I say almost, as *@controversial* and *wip* (work in progress) are actually not meant to last for too long, but they are convenient for a few days or weeks for easy reporting.

Thanks to all these tags, it becomes easy to extract only a subset of scenarios, by title only or complete with their step by step description.

- When meeting the business experts with very limited time, perhaps we could only focus on the @keyexample and on the @controversial stuff.

```
@keyexample or @controversial Scenarios:
- Full reimbursement for return within 30 days
- No reimbursement for return with no proof of purchase
```

- When reporting to the sponsor about the progress, the @wip and @pending scenarios are probably more interesting for this audience, along with the proportion of @acceptancecriteria passing green.

```
@wip, @pending or @controversial Scenarios:
- Error for unknown return
```

- When on-boarding a new team member, just going through the @nominalcase scenarios of each @specs section may be enough.

```
@nominalcase Scenarios:
- Full reimbursement for return within 30 days
- No reimbursement for return beyond 30 days
```

- For the compliance officers, they want everything that is not @wip. However, even in that case, they may want to have the big document show a summary of the @acceptancecriteria first, and the rest of the scenarios in addendum, if they find it more convenient.

# Highlighted Core (Eric Evans)

*Some elements of the domain are more important than others*

In the book Domain-Driven Design, Eric Evans explains that when a domain grows to a large number of elements, it becomes difficult to understand, even if only a small subset of them are really important. A simple way to guide developers focus on these particular subset is to highlight it in the code repository itself.

**Flag each element of the CORE DOMAIN within the primary repository of the model, without particularly trying to elucidate its role. Make it effortless for a developer to know what is in or out of the CORE.**

Using annotations to flag the core concepts directly into the code is a natural approach, one which evolves well over time. Code elements like classes or interfaces get renamed, move from one module to another, and sometime end up deleted.

```
/**
 * A fuel card with its type, id, holder name
 */
@ValueObject
@CoreConcept
public class FueldCard {
  private final String id;
  private final String name;
  ...
```

This is a perfect simple example of curation by annotations. It is an internal documentation integrated into the search capabilities of your IDE. You can see the list of all core concepts just by searching every reference of the annotation in the project, always up to date.

**The highlighted core is available instantly and at any time in your IDE through a search on all references of the @CoreConcept annotation**

And of course, tools can also scan the source and use the highlighted core as a convenient and relevant way to improve their curation. For example, a tool to generate a diagram may show everything when there are less than 7 elements, and focus only on the highlighted core when there are many more elements. A living glossary typically uses that to highlight the most important elements in the glossary too, by showing them first, or by printing them in a bold font.

# Guided Tour, Sightseeing Map

*It is easier to quickly discover the best of a new place with a guided tour or a sightseeing map.*

In a city you have never been before, you can explore randomly, hoping to bump on something interesting. This is something I love to do for an after-noon within a longer stay, to have a feel of the place. However if I have only one day and I want to quickly enjoy the best of the city, I buy a guided tour with a theme. For example I have excellent souvenirs of a guided tour of the old sky-scrappers in Chicago, where the guide knew how to have us go inside the historical lobbies to enjoy the low light that was typical of early light bulbs. One year later I enjoyed the architecture boat tour of Chicago, from the river, which is another way to really grasp the city. In Berlin, we booked a tour dedicated to Berlin's street art which was eye opening. Nothing fancy, but the same street art you see everyday without noticing much takes another dimension when put in a context and with just one extra hint from the guide.

But guided tours start at a fixed hours on a few days a week only, take two hours, and are expensive. If you happen to pass in a city on the wrong day, you are out of luck. But you can still get a tourist map, or printed guided tours. And of course, there is also an app for that! On your favorite app store, there are plenty of tourist guides, with guided tours and sightseeing maps, classified by themes like Attractions, Eat, Drink, Dance, Concerts etc. In Chicago, the Society of Architecture offers free architecture tours on leaflets too. And the internet is full of resources to help plan a visit:

- *Top 20 list of the must-see highlights*
- *Itineraries to help you plan your visit*
- *101 things to do in London

> Sometime this goes a bit too far, as in this guided tour called *101 things to do in London: unusual and quirky experiences*, for example: *Stop for coffee in a public loo*: "Don't worry, these beautifully converted old Victorian toilets were given a good scrub down before the plates of cakes were laid out. Opened in 2013, Attendant has a small bank of tables where the porcelain urinals once provided relief to gents about town."
>
> –Timeout London[24]

The same goes with a code base you are not familiar with. The best way to discover it is with a human, in other words a colleague. But if for some reasons you need to provide a standalone alternative, you can take inspiration from the tourism industry and provide itineraries of guided

[24] http://www.timeout.com/london/things-to-do/101-things-to-do-in-london-unusual-and-unique

tours and sightseeing maps. This tourism metaphor comes from Simon Brown, who writes on his blog "Coding the Architecture" and wrote the book "Architecture for Software Developers".

One important thing to realize is that all the tourism guidance in a city is highly curated: only a very small subset of all the possible content of the city is presented, for various reasons ranging from the historical importance of a landmark to more lucrative reasons.

But one important difference between a code base and a city is that a code base can change more frequently than most cities. As a result, the guidance must be done in such a way that the work to keep it up-to-date is minimized, for example using automation.

**Therefore: Provide curated guides of the code base, each with a big theme. Augment the code to be visited with extra metadata about the Guided Tour or the Sightseeing Map, and setup an automated mechanism to publish as often as desired an updated guide from these metadata.**

If the code base does not change much, a guided tour or a sightseeing map can be as simple as a bookmark with a list of the selected places of interest, and for each of them a short description and a link to its location in the code. If the code is on a platform like Github, it is easy to link to any line of code directly. This bookmark can be done in HTML, markdown, JSON, a dedicated bookmark format or any other form you like.

If the code base changes frequently or may change frequently, which is usually the case, a manually managed bookmark will require too much work to keep it up-to-date, so you would choose dynamic curation instead: place tags on the selected locations in the code, and rely on the search features of the IDE to instantly display the bookmarks. If needed you can add metadata to the tags that will enable the reconstruction of the complete guided tour simply by scanning the code base.

A sightseeing map or a guided tour based on tags in the code is a perfect example of the Augmented Code approach.

You may be worrying that adding tags about Sightseeing Maps or Guided Tours into the code pollutes the code, and you are right. These tags are not really about the tagged element intrinsically, but about how it it used. I usually prefer to avoid that. Use this approach sparingly.

> Consider your code base as the beautiful wilderness in the mountains where you go hiking. This is a protected area, and yet you have read and white hiking trails signs painted on directly on the stones and on the trees. This is a small pollution of the natural environment, but we all accept it since it's very useful at the expense of a limited degradation of the landscape.

## A sightseeing map

To get started with this approach,, you first create your custom annotation or attribute, and then you put it on the few most important places that you want to emphasize. To be effective, keep the number of place of interest low, ideally 5 to 7 and no more than 10.

It may well be that one of the most difficult decision here is to name the annotation or attribute. Here are some naming suggestions:

- KeyLandmark, Landmark
- MustSee
- SightSeeingSite
- CoreConcept, or CoreProcess
- PlaceOfInterest, PointOfInterest, or POI
- TopAttraction
- VIPCode
- KeyAlgorithm, KeyCalculation

For the approach to be useful you also need to make sure everybody knows about the tags, and knows how to search them.

## An example in C#

Let's create our custom attribute. Here we decide to put it into its own assembly to be shared by the other Visual Studio projects (which also means we don't want anything specific to any particular project there).

```
public class KeyLandmarkAttribute: Attribute
{
}
```

You can now immediately use it to tag your code:

```
public class Foo
{
  [KeyLandmark("The main steps of enriching the Customer Purchase from the initi\
al order to a ready-to-confirm purchase")]
  public void Enrich(CustomerPurchase cp)
  {
    //... interesting stuff here
  }
}
```

# An example in Java

As usual, Java and C# are very similar:

```java
package acme.documentation.annotations;

/**
 * Marks this place in the code as a point of interest worth listing on a sights\
eeing map.
 */
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface PointOfInterest {

        String description() default "";
}
```

And now we can use it:

```java
@PointOfInterest("Key calculation")
private double pricing(ExoticDerivative ...){
  ...
```

An alternative naming could be:

```java
@SightSeeingSite("This is our secret sauce")
public SupplyChainAllocation optimize(Inventory ...){
  ...
```

In C# we would use custom Attribute:

```csharp
public class CoreConceptAttribute : Attribute

[CoreConcept("The main steps of enriching the Customer Purchase from the initial\
 order to the ready to ship Shipment Request")]
```

The wording is up to you, and you can use one generic annotation with a generic name like "PointOfInterest", and add a parameter "Key calculation" to precise what it is about.

Alternatively you could decide to create one annotation for each kind of point of interest:

```java
@KeyCalculation()
private double pricing(ExoticDerivative ...){
  ...
```

# A Guided Tour Example

In the example below, the idea is to take a newcomer by the hand along the complete chain of processing of an incoming transaction, from the event listener on a message queue down to storing the out-coming report to the database. Note that even though strictly separates the domain logic and the infrastructure logic, this guided tour spans both business logic elements with elements of the underlying infrastructure, in order to give a complete picture of a complete execution path.

This guided tour currently has 6 steps. Each of these steps is anchored on a code element that can be a class, a method, a field or a package.

Here we use a custom annotation *@GuidedTour* with some parameters:

- The name of the guided tour: this is optional if there is only one tour, or if you prefer one annotation by guided tour, like @QuickDevTour
- A description of the step *in the context of this tour*. This is in contrast to the Javadoc comment on the element which describes the element out of any context.
- A rank, with a number or anything comparable, in order to order the steps when presenting them to the visitor.

Here's an example of Guided Tour:

```
/**
 * Listens to incoming fuel card transactions from the external system of the Fu\
el Card Provider
 */
@GuidedTour(name = "Quick Developer Tour", description = "The MQ listener which \
triggers a full chain of processing", rank = 1)
public class FuelCardTxListener {
```

It then goes through other steps, until the last one:

```
@GuidedTour(name = "Quick Developer Tour", description = "The DAO to store the r\
esulting fuel card reports after processing", rank = 7)
public class ReportDAO {

  public void save(FuelCardTransactionReport report){
  ...
```

Note that the numbering is not consecutive, it goes from 1 to 7 but there are only 6 steps. In the good old *BASIC* line numbering style we would number 10, 20, 30 etc. to make it easier to add another step in between when we want to.

In the case of simple selection of points of interest only for an audience of developers, we could stop there and rely on the IDE to present the tour as a whole, by doing a search on our custom annotation:

```
Search results for 'flottio.annotations.GuidedTour' 6 References in project:

flottio.fuelcardmonitoring.domain - (src/main/java/l...)
- FuelCardMonitoring
 - monitor(FuelCardTransaction, Vehicle)
- FuelCardTransaction
- FuelCardTransactionReport

flottio.fuelcardmonitoring.infra - (src/main/java/l...)
- FuelCardTxListener
- ReportDAO
```

The recap is all there, but it is not pretty, and there is no ordering. This could be enough for a small list of the main landmarks that you can explore in any order as you wish, so do not discount the value of the integrated approach, as it is much simpler and may be more convenient than more sophisticated mechanisms.

But in our case here this is not enough for a guided tour that is meant to be visited in order from start to finish.

So the next step is to create a living document out of it, a living guided tour.

## Living Guided Tour

Going further, we create a little mechanism to scan the code base to extract the information about each step of the guided tour, which then produces a synthetic report of the guided tour in the form of a ready to follow itinerary.

### 1. FuelCardTxListener

*The MQ listener which triggers a full chain of processing*

> Listens to incoming fuel card transactions from the external system of the Fuel Card Provider

### 2. FuelCardTransaction

*The incoming fuel card transaction*

> A transaction between a card and a merchant as reported by the fuel card provider

### 3. FuelCardMonitoring

*The service which takes care of all the fuel card monitoring*

> Monitoring of fuel card use to help improve fuel efficiency and detect fuel leakages and potential driver misbehaviors.

## 4. monitor(transaction, vehicle)

*The method which does all the potential fraud detection for an incoming fuel card transaction*

```
public FuelCardTransactionReport monitor(FuelCardTransaction transaction, Vehicl\
e vehicle) {
  List<String> issues = new ArrayList<String>();

  verifyFuelQuantity(transaction, vehicle, issues);
  verifyVehicleLocation(transaction, vehicle, issues);

  MonitoringStatus status = issues.isEmpty() ? VERIFIED : ANOMALY;
  return new FuelCardTransactionReport(transaction, status, issues);
}
```

## 5. FuelCardTransactionReport

*The report for an incoming fuel card transaction*

> The fuel card monitoring report for one transaction, with a status and any potential issue found.

## 6. ReportDAO

*The DAO to store the resulting fuel card reports after processing*

Note that in this guided tour, each title is actually a link to the corresponding line of code on Github. When the point of interest is a method, we have decided to include its block of code verbatim into the guided tour document, for convenience. In a similar fashion, when the point of interest is a class we could include an outline of the non-static fields and the public methods if we find it convenient and relevant to the focus of the guided tour.

This living guided tour document is generated in markdown, for convenience. Then 'Maven site' (or sbt or whatever other similar too) can do the rendering to a web page or any other format. An alternative that we have done here, is to use a Js library to render the markdown in the browser, which requires no additional toolchain.

An alternative to using Strings in the Guided Tour annotations would be to use enums; enums take care of naming, descriptions and ordering at the same time. However this moves the descriptions of each step of the Guided Tour from the annotated code to the enum class:

```
public enum PaymentJourneySteps {
  REST_ENDPOINT("The single page app call this endpoint with the id of the shopp\
ing cart"),
  AUTH_FILTER("The call is being authenticated"),
  AUDIT_TRAIL("The call is audit-trailed in case of dispute and to comply to reg\
ulation"),
  PAYMENT_SERVICE("Now we enter the actual service to perform the job"),
  REDIRECT("The response from the payment is sent by through a redirect");

  private final String description;
}
```

This enum is then used as value in the annotation

```
@PaymentJourney(PaymentJourneySteps.PAYMENT_SERVICE)
public class PaymentService...
```

# The implementation

In Java we use a Doclet-like library called QDox to do the grunt work here, as we want to be able to access the Javadoc comments. If you don't need the Javadoc, then any parser and even pain reflection could work.

QDox scans every Java file in *src/main/java*, and from the collection of parsed elements we can do the filtering by annotation. When a Java element (class, method, package…) has our custom *GuidedTour* annotation, it is included in the guided tour. We extract the parameters of the annotation, extract the name, Javadoc comment, line of code, and other information like the code itself when necessary. We turn all that into fragments of markdown for each step, stored in a map sorted by the step rank criteria. This way when the scan is done, we can render the whole document by concatenating each fragment in the rank ordering.

That said, the devil is in the details, and this kind of code can quickly grow hairy depending on how demanding you are with respect to the end result. Scanning code and traversing the Java or C# metamodel is not always nice. In the worst case you could even end up with a Visitor pattern. I expect that more mainstream adoption of these practices will lead to new small libraries which will take care of most of the grunt work for common use-cases, exactly like for Living Diagrams and Living Glossaries.

# Related

A Guided Tour is reminiscent of Literate Programming, but in reverse. Instead of having *Prose with Code*, we have *Code with Prose*. For a sightseeing map you only have to select the points of interest,

and perhaps group them by big themes. For a guided tour, you need to devise a linear ordering of the code elements. In Literate Programming you also tell a linear story which progresses through the code to end up with a document explaining the reasoning and the corresponding software at the same time.

A Guided Tour or Sightseeing Map is not just a documentation concern, but also a way to encourage continuous reflection on your own work as you do it. In this perspective, it would be a good idea to document a guided tour as soon as you are building the early walking skeleton of the application. This way, you will benefit from thoughtful effect of doing the documentation at the same time of doing the work.

See also SMALL-SCALE MODEL for similar ideas.

# Part 4 Automated Documentation

# Living Document

*A document that is evolving at the same pace than the system it describes*

**A living document is a document that is evolving at the same pace than the system it describes. It's prohibitively time-consuming to do manually, hence it's usually achieved through automation.**

As the names suggest Living documentation relies a lot on living documents, whenever Code as Documentation, Evergreen Documents and Tools History are not enough.

A living document works like a reporting tool that produces a new report after each change. A change is usually a code change, but could be just a key decision done during a conversation.

In this chapter we'll present a few key examples of Living Documents, like Living Glossary and Living Diagram.

## Anatomy of a living document.

A Living Document typically works in 4 main steps:

1. Select a range of data stored somewhere, for example source code in source control
2. Filter the data according to the objective of the document
3. For each piece of data that made it out through the filter, extract the subset of its content that is of interest for the document. It can be seen as a projection, and it's totally specific to the purpose of the diagram
4. Convert the data, and their relationships into the target format to produce the document. For a visual document it can the API of the rendering library. For a text document it can be a list of text snippets, or the library to produce a PDF.

If the rendering is very complex, the last step of converting into another model may be done twice, by creating an intermediate model that is then used to drive the final rendering library.

The hard part in each step is the interplay between the Editorial Perspective and the Presentation Rules. What data to select or ignore? What information to add from another source? What layout?

## Presentation Rules

There are rules for a good document, such as showing or listing no more than 7+/-2 items at a time. There are also rules for choosing a particular layout, a list or a table or chart so that it is congruent with the structure of the problem. This is not a book on that topic, however some awareness on these presentation rules will help you make your documents more efficient.

# 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 an\
other 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 us\
e image 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:

```
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:

```
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.\
annotation.");
        }
```

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()\
.simpleTypeName();
                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.

# Living Diagram

*A diagram that you can generate again on any change so that it's always up-to-date.*

> Automation should make it easier to change code safely, not harder. If it's getting harder, delete some. And never automate stuff in flux. – Liz Kheogh on Twitter

Some problems are difficult to explain with words, but are much easier to explain with a picture. This is why we frequently use diagrams in software development for static structures, sequences of actions and hierarchies of elements.

Most of the time we only need diagrams for the time of a conversation. Quick sketches on the napkin are perfect for that purpose. Once the idea has been explained or the decision taken you don't need the diagram any more.

But there are diagrams you'd like to keep, because they explain important parts of the design that everybody should know. Most teams create diagrams and keep them as separate documents: slides, Visio or CASE tools documents.

The problem, of course, is that the diagram will become outdated. The code of the system changes, and nobody has the time or remembers to update the diagram. As a consequence it's very common to have diagrams that are a bit wrong. People get used to that and don't trust diagrams too much. They become increasingly useless until someone has the courage to delete them. From this point it will require a lot of skills to look at the system as it is and try to recognize how it was designed and why. It becomes a matter of reverse-engineering.

This is all frustrating, but the worst part is that important knowledge is lost in the process, knowledge that was there at the beginning.

**Therefore: Whenever a diagram will be useful for the long term, for example it has already been used several times, setup a mechanism to automatically generate the diagram from the source code without any manual effort. Have your Continuous Integration trigger it on each build, or on a special build that is ran on-demand at the click of a button. Don't re-create or update the diagram manually each time.**

## Diagrams help conversations

Conversations and diagrams are not incompatible:

> Unexpected side effect of having a living diagram of the system: it makes development more tangible. You can point to things in discussions. Rinat Abdullin on Twitter @abdullin

Being able at all times to refer to the latest version of a diagram reflecting the current state of the software is a catalyzer of discussions.

## Editorial Perspective

When we create and maintain diagrams manually, and given the time it takes, it's tempting to put as much as possible onto the same diagram to save effort, even if this is detrimental to its users. However once the diagrams are generated, there is no reason any more to make your diagrams more complicated. Creating another diagram is not so much effort.

The Editorial Perspective is based on the intent of the considered document. Of course this assumes that each document has a clearly identified purpose, for an identified audience, which should be the case.

Diagram real estate is in limited supply, and so is the time and cognitive resources of its audience.

A document whose purpose is to clarify the external actors of the system for a non-technical audience should hide everything except the system as a black box and each actor with its non-technical name and the business relationship with the system. It should not show anything about JBoss, http or Json. It should not show components or service name. The Editorial Perspective is what makes a document relevant or not. A document that tries to show different things at the same time requires more work from its audience and does not convey a clear message.

**1 diagram, 1 story**

**Therefore: Remember each diagram should have one and only one purpose. Resist the**

**temptation to add extra information to an existing diagram. Instead, create another diagram that focus on the extra information while removing other information that is less valuable for this new different purpose. Filter superfluous information aggressively; only the essential elements deserve to make it onto the diagram.**

> 1 Diagram, 1 Purpose

A related anti-pattern is showing what's convenient rather than showing what's relevant to an identified purpose.

Remember the reverse-engineering / round-trip tools of the end of the 90's? It was magic at the beginning, until you end up with diagrams like this (or worse):

**How useful is that diagram?**

Too much information is like no information at all, it's equally useless. It takes a lot of serious filtering for your diagrams to be useful! But if you clearly know the point of the diagram, you're already half-way.

A challenge for living diagrams is the filtering and extraction of only the relevant data out of the mass of available data. On any real-world codebase, a living diagram without file rerun is close to useless, it's just a mess of boxes and wires that don't help understand.

Useful diagrams tell one thing. They have a clear focus. Dependencies. Hierarchy. Workflow. A particular decomposition of modules. A particular collaboration between classes, as in a design

pattern. You name it, but you only chose one. After all, since they're generated, it's easy to create one diagram for each aspect to explain, no need to try to mix them. Deciding the focus of the desired diagram is the most important decision, it's an editorial decision.

Once the focus is chosen, the filtering step will only select the elements that really contribute to the focus, and ignore the rest. Ideally there should be maximum 7-9 elements at this stage. Then for each element the extraction step will extract only the minimal subset of data that are really relevant for the focus. Resist the temptation to show everything. If you've ever tried UML tools with magic round-trip mechanisms, you've probably seen what a death by over-complex diagrams means when you let the reverse-engineer your codebase.

## Living Diagram to keep you honest

It's important to keep the code for the living diagram itself into source control in order to run it again and again, so that when the code changes it's easy to generate the updated diagram. Going further this generator can become a plugin in the build tool to generate the latest version of the diagrams during each build.

When a living diagram is part of the build, it becomes another way to look at the current state of the code. You may have a look at it during code review, or during design meetings, or just randomly to see if everything is like expected. The biggest benefit from this kind of diagrams is that they show the code as it is, so that you can be surprised. They keep you honest about the quality of the design. As discussed with Rinat Abdullin on Twitter, if you have to code a new module on your own, using an auto-generated diagram can be your first development feedback. And if you work with colleagues, another benefit of having a living diagram of the system is that "it makes development more tangible. You can point to things in discussions."

## The quest for the perfect diagram

From the traditional, manually-crafted diagram to the perfect Living Diagram, there is a scale of flavors, each with a different level of automatic adaptation to change, and and a different level of effort. The lower in the scale, the less effort to produce one diagram, but the more effort it will require to update the diagram in reaction to changes.

- **Napkin Sketch**: Throw-away, using pen and paper
- **Proprietary Diagram**: Look nice, but take a lot of time to create and maintain
- **Plain Text Diagrams**: Plain text is easier to maintain, source control and diff-friendly, but you still have to maintain it
- **Code-Driven Diagram**: You may author your diagram using code as an alternative to plain text. It could even be refactoring-proof when a class is renamed (the diagram is renamed too) or removed (the compiler tells you there is something wrong)

- **Living Diagram**: The diagram is totally created out of the code base itself.

We'll now go through each flavor of diagram for more details.

Napkin Sketch: perfect for the instant, but disposable. No need for anything beyond a pen and a random piece of paper: the back of a letter, the napkin, whatever.

Proprietary Diagram: not the preferred option unless you want to do the layout manually, if you need more complete UML support, if you really want all the extra features the tools offer, or if you have to use them by law. Time consuming, only editable by people with the tool installed, hard to diff, large files, takes time to adjust the layout and every graphical possibility.

Plain Text Diagrams: malleable, easy to change, easy to diff. Some IDE can propagate refactorings like renaming to class names in text, which may help reduce maintenance, but it can constraint the text too.

ASCII diagrams are a kind of plain text diagrams.

Code-Driven Diagram: more refactoring-proof, they are programmatic diagrams, with dedicated code and/or application code, eg driven by a DSL that includes references to code.

If you need a diagram only once and then you can throw it away immediately after use, choose the Napkin Sketch. On the other hand, if the knowledge is important enough to over a period of time, choose another flavor of diagram in the spectrum. Chose one you feel comfortable with. For simple diagrams which won't change much except for a few additions, deletions and renaming refactorings, I would recommend Plain Text Diagram or Code-Driven Diagrams.

If you need a beautiful diagram to convince or sell, then a generated diagram is probably not a good fit. Generated diagrams seldom look particularly attractive. As soon as the diagram becomes a stake in itself, it becomes worth doing it well, using the right tools to make it shiny. You may try commercial Proprietary CASE tools, but you will probably resort to graphical design tools, or even call a graphic designer to do the job.

**LOL**

I know this diagramming tool is not friendly and you hate it, but you must use it, we have already bought an unlimited enterprise license and there's a support team of 4 people to help!

## Rendering

I won't detail every possible way to create a diagram using a programming language, and it could be the topic for many other books, for various technologies and each context.

Remember a diagram should tell a story. One story. It should hide everything that does not matter for the story. As a result, most of the work for a living diagram is in ignoring everything that is not central to the story. The story must be the sole focus of the diagram.

The generation of a living diagram depends on what kind of diagram you need to create, but it typically involves 4 steps:

1. Scan the source code
2. Filter the relevant parts out of the huge number of elements
3. Extract the relevant information from each part, including the few meaningful relationships if relevant for the focus of the diagram
4. Render the information using a layout that matches the focus of the diagram

Let's take a simple example. We have a code base with many classes, some of which related to the concept of Order. We'd like to see a diagram that focuses only on the Order-related classes, and how they depend to each other.

Our code base looks like this:

```
...
Order
OrderPredicates
...
SimpleOrder
CompositeOrder
OrderFactory
Orders
OrderId
PlaceOrder
CancelOrder
... // many other classes
```

First we need a way to **scan** the code. We can use reflection or dynamically loading code for that. Starting from a package we can then enumerate all its elements.

There are many classes in the domain model of this application so we need a way to **filter** the elements we're interested in. Here we're interested in every class or interface related to the concept of Order. For the sake of simplicity we'll do the filtering on all elements that contain "Order" in their name.

Now we need to decide the focus of the diagram. In our case we'd like to show dependencies between the classes, perhaps to highlight those that may be undesired. This means that we during the scan of all the classes and interfaces we will **extract** only their name and the aggregated dependencies between them. For example we'll collect all fields types, enum constant, methods parameters types and return types, and super types that constitute the dependencies of a class.

We typically do that using a simple parser for the Java language, and with a visitor that walks through all declarations: imports, superclass, implemented interfaces, fields, methods, method parameters, method return, and exceptions, collecting all dependencies found into one Set. We may decide to ignore some of them based on our Editorial Perspective.

The last step is to **render** the diagram itself, using a specialized library. If we chose Graphviz, then it's about converting our model of classes with dependencies into the Graphviz text language. Once it's done we run the tool and we got the diagram.

In the current example, for each class with a name containing "Order", we would have its name and its list of dependencies. It is already a graph, that we can map to any graph rendering library like Graphviz.

Since we want to tell a story, here we can use the links as sentences:

<actor A> “does something to” <Actor B>

To do that we would have to keep some text to qualify the relationship between a class and each of its dependencies. “SimpleOrder is an implementation of Order”, “CompositeOrder groups together a number of Orders” etc.

There are many tools available for rendering, but not so many are able to do a smart layout of an arbitrary graph. Graphviz is probably the best, but it’s a native tool. Fortunately it now also exist as a Javascript library, easy to include into a web page to render your diagram in your browser. And this Javascript library has also become a pure Java library[25]! I used to use my old little Java wrapper[26] on top of Graphviz dot, but `graphviz-java` now sounds like a better alternative.

## A word on tooling

Here are some tools or technology that can help render a living diagram: Pandoc, D3.js, neo4j, AsciiDoc, PlantUML, Ditaa, Dexy and many other not well-known tools on Github or even on Sourceforge. Creating a Plain Svg file is an option too, but you have to do the layout yourself. It may a good approach if you can use it as a template too, like you would do dynamic html pages with a template. Simon Brown’s Structurizr is another tool as well.

To scan the source code you need parsers. Some can only parse the metamodel, while other have access to the code comments. For example in Java, Javadoc standard Doclet or the alternative tools QDox give access to the structured comments. On the other hand, the excellent [Google Guava] ClassPath(http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/reflect/ClassPath.html) only gives access to the programming language’s metamodel, which is enough in many cases.

Diagrams types by layout complexity

- Tables: perhaps not really a diagram, but there is a strict layout anyway
- Pins on a fixed background: like the markers on Google Map, it takes a way to map a (x, y) location for each element to pin on the background
- Diagram Template: Use a template (svg, dot) that is evaluated with the actual content extract from the source code
- Simple One-Dimension Flow (left-to-right, top-down), these are simple layouts you could even program yourself
- Pipeline, sequence diagram, in-out ecosystem black box
- Tree structure (left-to-right, top-down, radial). A tree structure is more complicated but it is still doable by yourself if you really want to.
- Inheritance tree, layers

[25] https://github.com/nidi3/graphviz-java
[26] https://github.com/cyriux/dot-diagram

- Containment (auto layout using the cluster feature of Graphviz)
- Rich Layout (vertical + horizontal layout + containment)

Of course if you want to be more creative you could also try to turn your diagram into a piece of art, doing a Photo collage, or even turn it into something animated or interactive.

## Visualization guidelines:

There are rules for a good document: showing or listing no more than 7+/-2 items is an important one, choosing a layout or list style or table or chart that is congruent with the structure of the problem etc.

> Why do so many engineers think complicated system diagrams are impressive? What's truly impressive are simple solutions to hard problems. – @nathanmarz

> The ultimate rule of thumb: if there is at least one line crossing another in a diagram, the system is too complicated – @pavlobaron

To get the most from your diagrams, consider making everything meaningful:

- Make the left-right and top-down axes meaningful: Manual vs Automatic, API vs SPI, Do Nothing vs Do Something, Single vs Plural, same intent beyond variety, orthogonal stuff, causality left-to-right, dependencies top-to-bottom…
- Make the Layout meaningful too: proximity, boundaries
- Make the elements attributes meaningful: size, color, texture, fill, border color…

# Hexagonal Architecture Living Diagram

## The idea

The Hexagonal Hexagonal Architecture is an evolution of the Layered Architecture, and goes further with respect to dependencies constraints.

This architecture basically only has two layers: an inside, and an outside. And there's a rule: dependencies must go from the outside to the inside, and never the other way round.

The inside is the domain model, clean and free from any technical corruption. The outside is the rest, and in particular all the infrastructure required to make the software work in relation with the rest of the world. The domain is in the center, with sometime a small application layer around (in red in the picture below). Around the domain model there are adapters to integrate the domain model and the ports that connect to the rest of the world: databases, middleware, servlets or REST resources etc.

**Hexagonal Architecture in a nutshell**

Now we have to create a documentation for it, perhaps because the boss asked for it, or because we'd like to explain this nice architecture to our colleague. How do we do that?

# The architecture is already documented

The first thing to realize is that this architecture is already documented in many places in the industry literature, starting with the website of Alistair Cockburn, who first described this pattern.

**Hexagonal Architecture diagram on Alistair Cockburn's website**

This architecture pattern is also described in many books, like Growing Object-Oriented Software, Guided by Tests (GOOS), Implementing DDD (IDDD), or the to-be-published book Clean Architecture by Uncle Bob. It is also known in the .Net circles as the Onion Architecture by Jeffrey Palermo.

This means that there is no need to explain much about Hexagonal Architecture yourself. You can just link to an external reference, where it is well explained. Why trying to rewrite what's been already written by a better writer?

> That's Ready-Made Architecture Documentation

# The architecture is already in the code

So the architecture itself is already documented in the literature, but what about its particular implementation in our custom project?

Because we're serious about our craft, the Hexagonal Architecture is already there in the code: the domain model is in its own package (read namespace or project in .Net), and the infrastructure is in one or several other packages, clearly segregated from the domain model.

With some experience about this pattern, you can recognize it just by looking at the packages and their content. Such a clean and strict segregation never happens by pure chance, it demonstrates a clear design intent. If you can recognize the Hexagonal Architecture just by looking at the code, we're done then!

Not really. Not everyone knows about Hexagonal Architecture, and architecture is by definition something everybody should be aware of. We need to make the architecture explicit in some way. It's 99% already there, we need to add the missing 1% to make it fully visible to everyone. We need to do some Knowledge Augmentation, using annotations or naming conventions. Both work well here.

The naming convention in fact is already there:

- Every class, interface, enum is in a package under the root package *.domain.*
- Every infrastructure code is under *.infra.*

We need this convention documented, and stable of course.

We could also use annotations instead. This would enable adding more information, like a rationale:

```
@HexagonalArchitecture.DomainModel(rationale = "Testability + Independence from \
the frameworks and technologies", alternatives = "DDD.Conformist")
package flottio.fuelcardmonitoring.domain;

import flottio.annotations.hexagonalarchitecture.HexagonalArchitecture;
```

# Creating the living diagram

## Knowing what we want

What we want is a diagram with an hexagon (or any shape indeed) at the center, representing the domain model with its most important elements inside. Outside and around this shape we expect to have every significant element of the infrastructure, with arrows showing how they depend with the domain elements inside. Something like this:

**A quick sketch of the kind of diagram we'd like to generate**

We want a layout that flows from left-to-right, from the calls to the API to the domain, and then to the service providers and their implementations in the infrastructure.

## Where's the knowledge right now?

As we've seen, the biggest part of the knowledge about the Hexagonal Architecture is in the naming convention of the packages. The other part of the knowledge is simply the list of every class, interface, enum contained in these packages, along with their relationships.

A convenient convention when drawing Hexagonal Architecture is to have every element consuming the domain model on the left, while every element providing services to the domain model are more on the right. How do we extract that information from the source code?

In our current application we have several opportunities for simplification: every class that calls the domain model does so through its member fields, and every service provider integrates with the domain model by implementing one of its interfaces. This is a common situation, but not a rule, e.g. a caller may be get its response through a callback. In other cases you may have to declare explicitly who's on the API side and who's on the SPI (service provider) side if you care about that in the diagram layout.

## Filtering out irrelevant details

Even in small projects, the source code contains a lot of information, so we always need to carefully decide what to keep out of the diagram:

- Every primitive

- Every class that act as a primitive (like the most basic Value Objects)
- Every class that is not related to other classes mentioned in the diagram

We're going to include classes in the following fashion:

- Include all classes and interfaces within the domain model (appart from quasi-primitives like units of measurement etc.). Being in the domain model is a matter of naming convention, or being in a package annotated as such.
- Include their mutual relationships that make sense. We may want to fold type hierarchies into their super type to save diagram real estate.
- Include infrastructure classes that have a relationship with elements already included in the domain model
- For each infrastructure class, include their relationship with respect to the domain classes, and between infrastructure too. In order to have a directed diagram in the API to SPI direction from the left to the right, we make sure that *call* and *implement* relationships are in opposite directions: *A Calls B* and *A implements B* must be in opposite direction. If you don't understand this now, no problem, this is my fault, and you will understand it clearly as soon as you try to make it work!

All this is just one example that works fine in one context. It is by no mean a universal solution for this kind of diagram, you should expect to try various alternatives, and you may have to filter more aggressively if your diagram gets too big. For example you may decide to only show the core concepts, based on additional annotations.

## Scanning the source code

For a living diagram like this one, all you need is a way to iterate through all the classes, and the ability to introspect them. Any standard parser can do that, and you can even do that without any parser just by using reflection. Because the focus is on the Hexagonal Architecture and nothing else, it's all about segregation of elements and highlighting the dependencies between them.

In this example of diagram, I just use standard Java reflection, with the help of the Google Guava *com.google.common.reflect.ClassPath* to scan the full classpath conveniently. I use my own utility library DotDiagram as a convenience wrapper on top of the Graphviz Dot syntax to create the .dot files. Then it's up to Graphviz Dot to do the auto-magical layout and rendering.

Here's an example of a diagram we get after rendering:

**Hexagonal Architecture Living Diagram generated from the source code**

## One month later

We are not happy with a name 'Trend' of the domain interface, and we decide to rename it into 'SentencesAuditing'. No need to update the diagram by hand, a new, up-to-date diagram is generated on the next build with the new name.

# Possible Evolutions

The Hexagonal Architecture constraints dependencies: they can only go from the outside to the inside, and never the other way round. However our living diagram shows all dependencies, even those that violate the rule. This is very useful to make the violations visible.

It's possible to go even further and to highlight all violations in a different color, e.g. with big visible red arrows when they are in the wrong direction. This illustrates that the line is very thin between a living diagram and static analysis to enforce guidelines.

---

You may have mentioned that it's impossible to talk seriously about a living diagram without talking deeply about the purpose of the diagram, in other words, without talking about design.

This is no coincidence. Useful diagrams must be relevant, and to be relevant when you're supposed to describe a design intent you must really understand this design intent. This suggests that doing design documentation well converges with doing design well.

# 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 \
analysis 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.

# Living Services Diagram

Distributed tracing based on the Google Dapper paper[27] is becoming a vital ingredient of a microservices architecture. It's "the new debugger for distributed services", a key tool for monitoring, typically to solve response time issues.

But it's also a fantastic ready-made Living Diagram tool to discover the living architecture of your overall system with all its services on a given day.

For example, Zipkin UI and Zipkin Dependencies provide a services dependency diagram out-of-the-box:

**Zipkin dependencies diagram on screen**

This view is nothing more than the aggregation of every distributed trace over some period, for example for a whole day.

## A matter of Augmented Code, but at runtime

For distributed tracing to work, you need to augment the system through 'instrumentation'. Every service or component must use a `Tracer` that conforms to the Trade Identifiers to declare the

[27] http://research.google.com/pubs/pub36356.html

reception of a request, and the sending of the response, along with annotations, and additional "baggage" as a key-value store.

The Trace Identifiers involve a context made of 3 identifiers which enable to build the call tree as an offline process:

- Trace ID // the correlation ID of a complete call tree
- Span ID // the correlation ID of this single client - server call
- Parent ID // the parent call we're in

The span name can be specified, for example with Spring Cloud Sleuth it's done with an annotation:

```
@SpanName("calculateTax")
```

Some of the core annotations used to define the start and stop of a client - server request are:

- cs - client start
- sr - server receive
- ss - server send
- cr - client receive

The annotations may be extended to classify your services or to perform filtering. However the tools may not naturally support your own annotations.

The baggage, or "binary annotation" goes beyond to capture key runtime information:

```
responsecode = 500
cache.somekey = HIT
sql.query = "select * from users"
featureflag.someflag = FALSE
http.uri =        /api/v1/login
readwrite = READONLY
mvc.controller.class = Application
test = FALSE
```

Here, all the tagging with metadata and other live data happens at realtime. But you recognize this is a similar approach to Augmented Code. You need to inject some knowledge for tools to help more!

## Discover the architecture

The ability to inspect the distributed system in realtime isn't just for the front end developers. by aggregating the traces over time into a runtime service dependency graph "goes a long way to help architects and management to understand accurately how things work, negating much of the need for higher level documentation" as Mick Semb Wever write on his blog[28].

## The magic to make this work

By sampling, some request get instrumented as they go through each node of the system. The instrumentation generate span traces that are collected and stored into some central (logically) datastore. Individual traces can then be searched and displayed. A daily cron triggers then post-processes all the traces into aggregates representing the "dependencies" between services. The aggregation is something like this (simplified):

```
select distinct span
from zipkin_spans
where span.start_timestamp between start and end
and span.annotation in ('ca', 'cs', 'sr', 'sa')
group by span
```

The UI then displays all the dependencies using some sort of automated nodes layout.

## Going further

All the above-mentioned is just the beginnging. By getting creative on the tags and through test robots stimulating the system on predefined scenarios, a distributed infrastructure like Zipkin has a lot of potential for Living Architecture Diagrams:

- Create "controlled" traces from a test robot driving one or more service(s), with a specific tag to flag the corresponding traces
- Display different diagrams for the "Cache = HIT" and the "cache = MISS" scenarios
- Display distinct diagrams for the "Write part" vs the "Read part" of an overall conversation across the system.

Trying something in this area? Please let me know!

[28] http://thelastpickle.com/blog/2015/12/07/using-zipkin-for-full-stack-tracing-including-cassandra.html

# Context Diagram

*Showing how a system is integrated to surrounding actors*

No system is an island, every system is part of a bigger eco-system with other actors, typically people and other systems. From a developer point of view, the integration to other systems is sometime considered obvious knowledge not worth documenting, especially in the early years of a system. But after a while the system has grown and is now deeply integrated with many other actors, and even people in the team no longer know about this ecosystem. To reconstitute the whole picture you'll now have to review all the code manually and interview knowledgeable people, where "knowledgeable" is also synonymous with "busy".

This context knowledge is essential to reason about impacts to or from other actors when considering changes in this system or in another external system.

As such it deserves to be made clearly visible, up-to-date at any time.

Basically an context diagram is about a recap of all actors using the system (API side) or used by the system (Service Providers side):

```
  Actors using     * --> System --> *   Actors used
  using the system                     by the system
```

The context can be expressed as a simple list:

API (actors using the system)

- Fuelo Card API
- Fleet Management Teams
- Support & Monitoring

SPI (actors providing services to the system)

- Google Geocoding
- GPS Tracking from Garmin
- Legacy Vehicle Assignment

But a visual layout has its advantages too:

**A generated context diagram with 3 actors on the left and 3 actors on the right**

You can create such diagram by hand each time you need it. It will be tailored for your matter at hand. Or you could generate it.

The above diagram was generated from the sample *Flottio* fleet management used throughout this book.

> The name 'context diagram' is borrowed from Simon Brown C4 Model[a], a lightweight approach to architecture diagrams which is becoming increasingly popular among developers.
>
> [a] http://www.codingthearchitecture.com/2014/08/24/c4_model_poster.html

This diagram tells the story of the system through its links to external actors, with some brief descriptions on some of them.

This diagram is a **Living Document**, refreshed whenever the system changes, automatically. As any living diagram, it is generated by scanning the augmented source code and calling a graph layout engine like *GraphViz*. 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 **Refactoring-proof Diagram**; if we just want to rename a module in the code, the diagram will be renamed too without extra effort. No need to fire PowerPoint or a diagram editor each time.

## Hyperlinks to the corresponding source code location

As opposed to a manually created diagram, a it can feature hyperlinks to the accurate locations in the code base. Click on any external actor on the diagram to jump to the corresponding URL in the source code repository online.

Note that even without a link, the wording in the diagram can be used verbatim to perform a search in the code base. Since the wording came from the code, it would be easy to find the corresponding location.

## Augmented Code & Knowledge Consolidation

The problem, of course, is to identify automatically the external actors, their name, description and direction of use (using or being used). Unfortunately I haven't found a miracle solution for that.

To generate this diagram, the code had to be augmented with some annotations to declare the *external actor*. This is an example of **Augmented Code** and is also a case of **Consolidation** of information spread across multiple packages and sub-packages.

For example this package `flottio.fuelcardmonitoring.legacy` which takes care of the integration with the legacy system for vehicle assignments to drivers, a provider of services for the system under consideration:

```
/**
 * Vehicle Management is a legacy system which manages which drivers is associat\
ed to a vehicle for a period of time.
 */
@ExternalActor(
   name = "Legacy Vehicle Assignment System",
   type = SYSTEM,
   direction = ExternalActor.Direction.SPI)
package flottio.fuelcardmonitoring.legacy;

import static flottio.annotations.ExternalActor.ActorType.SYSTEM;
import flottio.annotations.ExternalActor;
```

Another example is the class listening to the incoming message bus, which basically uses the system to check if fuel card transactions have anomalies:

```java
package flottio.fuelcardmonitoring.infra;
   // more imports...

/**
 * Listens to incoming fuel card transactions from the external system of the Fu\
el Card Provider
 */
@ExternalActor(
   name = "Fuelo Fuel Card Provider",
   type = SYSTEM,
   direction = Direction.API)
public class FuelCardTxListener {
   //...
```

We don't have to use annotations. We could also add sidecar files in the same folder than the annotation code, with the same content than the annotation inside, as Yaml, Json or as a .ini file:

```ini
; external-port.ini
; this sidecar file is in the integration code folder
name=Fuelo Fuel Card Provider
type=SYSTEM
direction=API
```

Some time later we want to add information to the context diagram, so we add this information to the code itself, in the Javadoc of the integration code, and then the diagram gets updated:

**A generated context diagram with 3 actors on the left and 3 actors on the right**

## Limitations & Benefits

Because of the need for some code augmentation with annotations or sidecar files, there remains the risk of not knowing about some external actors.

If in your project you can enumerate the few only possible ways perform integration, you may try to detect them all and add them to the diagram unless silenced explicitly through code augmentation.

In any case, integration through the database will be hard to detect and document. You may believe the database is a private detail of your system, but if another system queries or writes into it directly, it will be hard to find out without a conversation with the culprits.

On the other hand, this diagram shows every possible integration, but it cannot tell if they are active in production or not. If the code base is a toolkit for a product line, it will show all the potential integrations, not the one actually used in practice in a particular instance.

Another drawback of a generated diagram like the above compared to a ad hoc manual diagram is that they are not tailored for the particular matter of the day. But they're also much faster to produce.

Still, you may want to tweak the diagram generator, for example to focus on a subset of the context.

# Domain-Specific Diagrams (aka Visible Tests)

*Tests that produce a visual output for human review in a domain-specific notation*

Good tests check the code against predefined assertions all the time. They are silent unless something goes unexpected: a failed assertion, or an error.

However I've found that tests could sometime also be used to produce visible output like diagrams, in various domain-specific notations.

When starting in exploration mode, for example during a spike, when the problems are not clear and you're not sure how to solve them, it's hard to define accurate assertions at this stage. However a visible output gives a fast feedback on whether it works as expected or not.

Later, as the tests turn into non-regression tools, we add actual assertions, but we may still decide to keep some of the visible outputs, as a way to show what's happening.

## Domain-Specific Notations

Many business domains have grown their own specific notations over time. Domain experts use it naturally, usually with pen and paper.

For example in supply chain we tend to draw trees from the upstream producers to the distributors downstream:

**supply-chain tree**

In stock exchange, we often have to draw order books when it comes to decide how the matching happens:

| Bid Volume | Price | Ask Volume |
|---|---|---|
| | 105.00 | 40 |
| | 104.50 | 30 |
| | 104.00 | 20 |
| | 103.50 | 10 |
| 10 | 103.00 | |
| 20 | 102.50 | |
| 30 | 102.00 | 50 |
| 40 | 101.50 | |
| 50 | 101.00 | |

**order book for matching orders**

In finance, financial instruments pay and receive cash flows (amounts of money) over a timeline, which we draw using vertical arrows on a timeline:

**Cash flows over a timeline**

# Generating custom domain-specific diagrams to get visual feedback

I used to create simple tests with no assertions at the beginning that simply generate basic, ugly Svg files like the following:

**generating an Svg file**

Compare that to a spreadsheet table:

```
EUR13469  20/06/2010
EUR13161  20/09/2010
EUR12715  20/12/2010
EUR12280  20/03/2011
EUR12247  20/06/2011
EUR11939  20/09/2011
EUR11507  20/12/2011
EUR11205  20/03/2012
EUR11021  20/06/2012
EUR8266   20/09/2012
EUR5450   20/12/2012
EUR2695   20/03/2013
```

It's much easier to check the evolution of the amounts paid over time visually.

Of course you could also dump a .csv file and graph it in your favorite spreadsheet application. Or you could even generate an .xls file with the graph inside programmatically (in Java you could use Apache POI for example to do that).

Here is a fairly more complicated example of generated diagram, showing how the cash flows are conditioned by market factors:

**generating an Svg file**

As you can see, I'm not an expert of Svg and it was a quick graphing to get visual feedback during the initial spike of a bigger project. Nowadays you would use a modern js library to produce beautiful diagrams!

# A complement to Gherkin scenarios?

I haven't tried it yet, but I'd love to have some key scenarios in Cucumber or Specflow produce such domain-specific diagrams in addition to the test results for their assertions. This sounds quite feasible, so if you happen to try it, please let us know!

# Pattern-Oriented Living Diagram

## The challenges with automated generation of design documentation

Producing a documentation of the design of a software project manually requires a lot of work, and becomes obsolete very quickly after the next change or refactoring. Manually drawing meaningful UML diagrams is very time consuming, and even choosing what to display takes a lot of time.

As Domain-Driven Design states, the code is itself the model, but code lacks the ability to clearly express larger-than-class structures and collaborations. This is why some additional carefully

selected design documentation is useful to show the bigger picture. And it can be generated from the code, as long as the code is augmented with the design intentions.

## Generating design documentation

The use of patterns to help with the process of generating a design documentation is promising. Patterns naturally lie "on top" of the language elements. They address a particular problem within a context, discuss a solution and have a clear intent. They involve a collaboration of several elements from the language, such as several classes and their protocol, or just relations between fields and methods within a class. Each pattern is a chunk of design knowledge. When comes the time to automate the description of the design, it sounds natural to chunk the automation process by pattern as well.

In past projects I've declared some of the patterns used in the code (using annotations), and created little tools derive from them partial macro structures of the software design around these patterns. Since each pattern come with a context, this context helps for selecting what to show or hide, and how to show it.

From the patterns declared in the code, the tool can then generate a better-than-generic design documentation (e.g. diagram) informed by the knowledge chunked pattern by pattern.

# Part 5 Runtime Documentation

# Visible Workings: Working Software as its own Documentation

## Working software as documentation

The Agile Manifesto says "Working Software over Comprehensive Documentation". What if the working software was itself a kind of documentation?

It is already quite common to design the User Experience so that the users can have successful interactions with an application without ever having to open the user manual. However, it's less common to design the software so that its developers can understand it without even having to open the source code.

It is possible to learn the business domain to some extent just by using a related and well-designed application. The software is by itself a piece of documentation about itself and its business domain. This is why every developer on an application should at least know how to use their application for most standard use-cases, even if it is a complicated application dealing with complicated financial instruments.

Anything that can answer a question can be considered documentation. If you can answer questions by using the application, then the application is part of the documentation.

## Visible Workings

A more radical perspective on the software as a documentation is to rely on the software itself to explain how it works inside, something Brian Marick calls **Visible Workings**, i.e. make the internal mechanisms visible from the outside.

There are many ways to achieve that, and they all have in common to rely on the software itself to output the desired form of documentation.

As an example, many application perform calculations like payrolls or bank statements and other forms of data crunching. It is often necessary to describe how the processing is done, for external audiences like business people or compliance auditors.

You may think of Visible Workings approaches like an 'export' or 'reporting' feature, but on the way it works internally. You want to be able to ask the software "How do you compute that?" or

"What's the formula for this result?", and it just tells you, at runtime. There should be no need to ask a developer to get the answer.

It's the kind of feature that is not often requested by the customer, but it's a valid answer to a need for more documentation. It's also worth noting that the development team has full latitude to decide to add features that make its own life easier, since the team is obviously one of the key stakeholders of any project. The key is to spend just enough time for the expected benefit. Visible Workings techniques are obviously very useful for the development teams.

This pattern comes in various forms:

- Introspectable Workings
- Visible Calculation
- Queryable Object Log

# Introspectable Workings

At runtime the code often takes the form of an object tree. This is the tree of objets that you create by using the new operators, factories or builders, or Dependency Injection frameworks like Spring or Unity.

Often, the exact nature of the object tree may vary according to the configuration or even on a by-request basis.

**Object tree at runtime may vary by configuration or even request-by-request**

How do you know what the object tree really looks like at runtime for a given request?

The regular way is to look at the source code and trying to imagine how it will wire the object tree. But we would still like to check if our understanding is correct.

**Introspect trees of objects at runtime in order to display the actual arrangement of objects, their actual objects types, and their actual structure**.

In languages like Java or C# this can be done through reflection, or through methods on each member of the structure to be introspected. The simplest form of this idea is just to rely on the toString() methods of each element to tell about itself, and about its own members with some indentation scheme. When using DI containers, you may as well try to ask the container to tell what it constructed.

**Introspecting the object tree, from its root**

Let's take the example of a little search engine of Hip-Hop beats loops. It's made of an engine, at the root, that itself queries a reverse index for fast search queries. For indexing purposes, it also browses a repository of links contributed by users of the service, using a loop analyzer to extract the significant features of each beats loop to put into the reverse index. The analyzer itself makes use of a waveform processor.

Each of the engine, reverse index, links repository and loop analyzers are all abstract interfaces with more than one implementation each. The exact wiring of the object tree is determined at runtime, and changes according to configuration by environment.

# Introspecting by reflection

> If it's an object, we can traverse it - Arnold Schwarzenegger

Introspecting a tree of objects is nothing but a trivial recursive traversal. From the given (root) instance, we get its class and enumerate each declared field, because that's how classes store their injected collaborators here. For each collaborator, we carry on the traversal through a recursive call.

As usual, we probably need to filter uninteresting elements that we don't want to include in the traversal, classes likes Strings or other low-level stuff. Here the filtering is just based on the qualified name of the classes. If passed an instance of a class that has nothing to do with our business logic, just ignore it.

```
final String prefix = "org.livingdocumentation.visibleworkings.";

public void introspectByReflection(final Object instance, int depth) throws Ille\
galAccessException {
    final Class<?> clazz = instance.getClass();
    if (!clazz.getName().startsWith(prefix)) {
      return;
    }
    // System.out.println(indent(depth) + instance);
    for (Field field : clazz.getDeclaredFields()) {
      field.setAccessible(true);// necessary
      final Object member = field.get(instance);
      introspectByReflection(member, depth + 1);
    }
  }
```

With this code, if we just print each element with the proper indentation, the console displays:

```
SingleThreadedSearchEngine
..InMemoryLinkContributions
..MockLoopAnalyzer
....WaveformEnergyProcessor
..MockReverseIndex
```

Our Engine is a SingleThreaded one, and it uses an InMemory repository of contributed links, together with a mock of a loop analyzer, and another mock of a reverse index.

With the same code, we can instead build a dot diagram with each element and the proper relations between them:

**Introspecting the object tree, in practice**

This diagram shows the same information here, but each relationship could show additional information.

# Introspecting without reflection

To introspect an object tree without using reflection, all the objects in the tree must exhibit an accessible way to enumerate their collaborators. You could do that with public fields, but I'd not recommend that. Instead they can expose a public method that returns the list of their members.

In the simplest case, every element would implement some interface like Introspectable, becoming an instance of the Composite pattern:

```
interface Introspectable {
  Collection<?> members();
}
```

Thus the traversal of the tree is again nothing but the recursive traversal of the composite:

```
private void traverseComposite(Object instance, int depth) {
  final String name = instance.getClass().getName();
  // Add this node to the diagram
  digraph.addNode(name).setLabel(instance.toString());
  if (!(instance instanceof Introspectable)) {
      return;
  }
  final Introspectable introspectable = (Introspectable) instance;
  for (Object member : introspectable.members()) {
    traverseComposite(member, depth + 1);
    // Add the relationship from this node to its member to the diagram
    digraph.addAssociation(name, member.getClass().getName());
  }
}
```

Obviously this second approach produces exactly the same output than by reflection.

Which approach to chose? If all the object are created by the team and there aren't too many of them, i'd go the composite flavor, as long as it doesn't pollute too much the classes. In this case introspection has to be considered as another responsibility of the code, by design.

In all other cases, the approach of introspection by reflection is the best or only choice.

This approach helps make the inner workings visible. In the case of a workflow, decision-tree or decision table that is built on the fly for each given business request, Introspectable Workings is a way to make the particular structure that was built visible for users and developers alike.

Sometimes however you don't even need any introspection at all. When the procesing is driven by a configuration, hardcoded, from a file or from a database, displaying the workings may be simplified a lot, as this is just a way to display the configuration in a nice way. At a minimum, every workflow or processing that is driven by a configuration should be able to display the configuration that is used for a particular processing.

# Part 6 Refactoring-Friendly Documentation

# Plain-Text Artifacts

Nothing beats plain text when it comes to collaboration through documents. Plain text formats are ideal for making changes, comparing changes, merging changes, version control. They are also rather small and don't require proprietary tools.

> Keep Knowledge in Plain Text

from "Pragmatic Programmers"

> We believe the best format for storing knowledge persistently is plain text. pragprog2[29]

Plain text has many advantages over other binary or proprietary formats: - No need for special tools to read and edit, you can work in your favorite text editor - Readable by humans, and should also be understandable by humans - Easy to search and most OS knows how to index plain text files - Works well with source control, easy to diff and merge in case of conflict

**Therefore: Agree as a team to collaborate on plain-text files as much as possible, stored on the source control together with the source code. Treat these collaborative artifacts as the authoritative single sources of knowledge. At build time, and if necessary, generate every other document from them through automated means.**

Tool-specific files formats require having the tool installed to read and write them. Unfortunately it's not uncommon that you cannot open a file written in version 8 of a tool when you have only version 7 of the tool installed. Over time it becomes increasingly difficult to have the right tool in the right version for everyone involved, therefore tool-specific files become progressively inaccessible and unmaintained. If the files contained important knowledge, this knowledge becomes inaccessible hence lost, and that is sad. In contrast, information stored in plain text never gets lost. Also plain text is always editable, even if corrupted.

Open Plain Text formats should be preferred whenever possible, e.g. .csv over xls, rtf or .html over .doc, otherwise the usual big PPT files must go to another dedicated wiki where they can be safely forgotten and become instantly deprecated.

Plain text should also be not just readable by humans, but also understandable by humans.

> Dave Thomas: I can give you 128 bit cipher key as ASCII, and you can read it, but it may not make sense to you. Andy Hunt: So it is readable, but not understandable. To be understandable, a plain text file can be self-descriptive thanks to meta-data. CSV files

[29] http://www.artima.com/intv/plainP.html

> with headers that describe the meaning of each column are an example. Well-named XML tags are another example. Self-descriptive means you can read and understand the content of a file without a user manual.

This approach is nothing really new (think about LateX…), and many of the tools we need for it already exist (Markdown renderers, diagram auto-layout tools), web site generators (Maven), website generator to organize and display Gherkin scenarios (Pickles).

# Source Control is the Reference

Source code has to be in source control. Period. What's the latest version of the code? It's easy, it's the code in the Mainline.

For non source code there's more opportunity for ambiguity. Knowledge in text documents, slides or spreadsheets may be stored anywhere in many different places. "Where is the latest version of the project goals? - Mmmh, I don't remember, have a look in the shared drive, otherwise in the wiki or the intranet."

We don't want to spend time chasing the latest decisions, or discussing what version of document to trust. No ambiguity requires simple rules.

Therefore: **Any change should always be materialized as a commit**.

No exception, or it would defeat the whole thing. Every piece of knowledge that is important for the project should be committed to the source control. Favor plain text documents whenever possible.

When this pattern is not possible, a variant is to keep links to the right documents and to external repositories in the source control. To reduce link maintenance, consider using a Link Registry.

In this ultimate approach, every changes requires a commit, with a commit comment. The commit can also trigger a build. The build can produce the latest version of all living documents, publish them where appropriate, send events to news feeds or company chat.

# Plain-Text Diagram

Most diagrams are short-lived. They are useful for a particular discussion, to help reason on a specific design decision, but once the idea has been communicated, or once the decision has been taken, they immediately lose most of their interest. That's why **Napkin Sketches** are the default way to go about diagrams. I use the word *napkin sketches* to actually refer to any kind of low-tech visual and tangible techniques, it could be *Whiteboarding*, *CRC Cards* or *Event Storming*. They're all great tools to communicate, reason and try things in a visual fashion.

However it does happen that some diagrams remain of interest for the longer term, in which case you want to persist the initial napkin sketch, set of cards or stickers, or the initial whiteboard into something better suited for the posterity. The first idea is to simply take a picture of the outcome and to store in the wiki, or directly in the source control, co-located with the related artifacts.

This works fine if the picture describes stable knowledge, but if it describes decisions that evolve regularly, then you'll have a misleading picture after a while. You could try to do a **Living Diagram**, but this may too hard or too much work to do compared to the expected benefits.

This is when you need a Plain-Text Diagram.

**Therefore: Take your initial napkin sketch, set of CRC cards and turn it into plain text, and use a text-to-diagram tool to render it into a visual diagram automatically. Then on every change, maintain the plain text description of the diagram, and keep it in source control within the related code base.**

An important idea of a plain text diagram is that we favor the content over the formatting. You want to focus on the content in plain text, and let the tools take care of the formatting, layout and rendering, as much as possible.

## An example

Let's take the example of the fuel card fraud detection algorithm. We started with a napkin sketch when thinking about the problem, listing every related responsibility needed, and how they interoperate to solve the problem:

**The initial napkin sketch**

After a few days we agree we need to keep as part of our documentation, and we need to make it more easy to read, and to maintain as we expect it to change from time to time.

This diagram should tell one story. It should hide everything that does not matter for the story. To be story-oriented we use links as sentences:

```
1   <actor A> "does something to" <Actor B>
```

So basically we look at the napkin sketch and literally describe it using sentences in this format:

```
FueldCardTransaction received into FuelCardMonitoring
FuelCardMonitoring queries Geocoding
FuelCardMonitoring queries GPSTracking
Geocoding converts address into AddressCoordinates
GPSTracking provides VehicleCoordinates
VehicleCoordinates  between GeoDistance
AddressCoordinates  between GeoDistance
GeoDistance compared to DistanceThreshold
DistanceThreshold delivers AnomalyReport
```

Then we can use a tool like Diagrammr (http://www.diagrammr.com apparently no longer working at the time of editing this chapter) it's easy to turn this set of sentences into a nice diagram.

The default layout of the rendered diagram is an Activity-like diagram:

**The diagram rendered from the text**

But the same text sentences can also be rendered as a Sequence diagram instead:

**Another layout to tell the same story in a different way**

A tool like that is in fact only a wrapper on top of an automatic layout tool like Graphviz. Each sentence describes a relationship between two nodes. The first word of the sentence represents the start node, while the last word of the sentence represents the target node. This is really rustic an approach.

It's not difficult to create your own flavor of this approach, using perhaps different conventions to interpret the text sentences. However the point is to keep it really rustic. If you don't pay attention to the simplicity of the syntax, you may end up with a syntax so complicated you have to look at its syntax sheet sheet all the time, which would defeat the purpose of simplicity.

When there are changes that require to update the diagram, it's easy to make them in the text. Renaming can be done with a find-and-replace. Perhaps your IDE can even have its refactoring automation reach the plain-text files, in which case you're less at risk of forgetting to update the diagram.

# Diagram as Code

An alternative flavor of a plain-text diagram is to use code in a programming language as the way to declare the nodes and their relationships. There are benefits:

- Auto-completion
- Checks from the compiler or interpreter to catch invalid syntax
- Can move along with any automated refactoring to remain in sync with all changes
- Can programmatically generate many dynamic diagrams from data sources

There are some drawbacks too:

- The code itself is less readable by non-developers than plain-text
- Identifiers names cannot contain whitespace

Here is an example of a diagram generated from my little library DotDiagram[30] which is a just a wrapper on top of Graphviz:

[30] https://github.com/LivingDocumentation/dot-diagram

```
final DotGraph graph = new DotGraph("MyDiagram");
final Digraph digraph = graph.getDigraph();

// Add the nodes
digraph.addNode("Car").setLabel("My Car").setComment("This is a BMW");
digraph.addNode("Wheel").setLabel("Its wheels").setComment("The wheels of my car\
");

// Add the associations between the nodes
digraph.addAssociation("Car", "Wheel").setLabel("4*").setComment("There are 4 wh\
eels").setOptions(ASSOCIATION_EDGE_STYLE);

// Render everything
final String actual = graph.render().trim();
```

This diagram expressed as code should render like this:

**Rendering MyCar —> 4* Wheel**

Of course the biggest benefits by far are the ability to generate diagram from any source of data. This technique is a key ingredient for any Living Diagram.

# Code is Documentation

> "Programs are meant to be read by humans and only incidentally for computers to execute." – H. Abelson and G. Sussman (in "Structure and Interpretation of Computer Programs")

Code is literally documentation. Code is written for machines of courses, but that's the easy part. The hard part is that code is also written for human beings to understand it for its maintenance and evolution.

> That, yes, but more. The source code is also the ONLY document in whatever collection you may have that is guaranteed to reflect reality exactly. As such, it is the only design document that is known to be true. The thoughts, dreams, and fantasies of the designer are only real insofar as they are in the code. The pictures in the reams of UML are only veridical insofar as they are in the code. The source code is the design in a way that no other document can claim. One other thought: maybe gloss isn't in the writing, maybe it's in the reading.
>
> – RonJeffries

It takes a lot of skills and techniques to improve the ability of the code to be quickly and clearly understood by people. It's a core topic in the Software Craftsmanship community, with many books, articles and conference talks on the topic, and this book is not meant to replace all that. Instead we'll focus on a few practices and techniques that are especially relevant, typical, or original with respect to the idea of code being itself documentation. As Chris Epstein once said during a talk, "be kind to your future self." Learn how to make the code easy to understand.

Many books have been written on writing code that is easy to read. Of particular importance are Clean Code by Robert Cecil Martin (Uncle Bob), and Implementation Patterns, by Kent Beck.

In this later book, Kent Beck advocates to ask yourself: "What do I want to say to someone when they read this code? […] Not just, "What will the computer do with this code?" but, "How can I communicate what I am thinking to people?"

## Text Layout

We usually think of code as a linear medium, however code is itself is a graphical arrangement of characters in the 2-dimensional space in the text editor. The 2D layout of the code can be used to express meaning.

The most common example are the guidelines on the ordering of the members of a class:

1. class declaration
2. fields
3. constructors and methods

With this ordering, even the class is declared as plain text, there is a visual aspect implied by the layering of the blocks of text on the page. This is not that far from how a class is visually represented in UML:

**UML visual notation for a class**

The main difference between the code layout and the visual notation is the absence of the border lines around the blocks of text in the code.

Let's have a look at other cases of code layout.

## Tabular Code Layout

Take the example of a socket considered as a state machine. This state machine can be fully described through its *state transition table*, which can be expressed literally as code. In this case, the layout really matters, the vertical alignment of the current states, the transitions and the next states is in itself meaningful.

```
public static enum State {
    CREATED, OPEN, CLOSE;
}

public static enum Action {
    CREATE, CONNECT, CLOSE;
}

public Object[][] socketStableTransitionTable() {
    final Object[][] stableTransitionTable = {
    //state,          action,          next state
    { State.CLOSE,    Action.CREATE,   State.CREATED },
    { State.CREATED,  Action.CONNECT,  State.OPEN },
    { State.OPEN,     Action.CLOSE,    State.CLOSE }};
    return stableTransitionTable;
}

@Test
public void testSocketStateMachine() {
```

**The transition table of a socket as a state machine with its expressive code layout**

This is easy to do with code, except that the automatic code formatting of the IDE may often break this alignment. Putting empty comments sections /**/ at the beginning of lines can prevent the formatter from re-ordering the lines, but it's hard to preserve the whitespaces. Of course this all depends on your IDE and its capabilities to auto-format in a smarter way.

## Arrange - Act - Assert

Unit tests also offer examples of how the 2D graphical layout of the code can be used to express meaning. The convention *Arrange - Act - Assert* advocates organizing the code in three different sections, each below the previous one, as shown in the example below:

```
@Test
public void aaa_distance_4km2_between_centre_pompidou_and_Eiffel_Tower() throws Exception {
    // Arrange
    final Coordinates centrePompidou = new Coordinates(48.8608333, 2.3516667);
    final Coordinates eiffelTower = new Coordinates(48.858222, 2.2945);

    // Act
    final double distance = GeoDistance.EQUIRECTANGULAR.distanceBetween(centrePompidou, eiffelTower);

    // Assert
    assertEquals(4190, distance, 10);
}
```

**The convention Arrange - Act - Assert in a unit test**

Once you're familiar with this convention, the vertical layout makes it graphically obvious what each section is doing, just by looking at the composition of text versus whitespace.

Another convention in unit tests is about considering that a unit test is basically about matching a given expression on the left with another expression on the right. In this approach the horizontal

layout is meaningful: we want the full assertion on one single line, with the two expressions on both sides of the assertion, as shown in the example below:

```
@Test
public void distance_4km2_between_centre_pompidou_and_Eiffel_Tower() throws Exception {
    assertEquals(4190, GeoDistance.EQUIRECTANGULAR.distanceBetween(CENTRE_POMPIDOU, EIFFEL_TOWER), 10);
}
```

**A test is about matching the expressions on the left with the expression on the right**

There is much more to say about every possible way to organize your code graphically, but this is not the point of this book, appart from drawing your attention on this universe of possibilities.

# Coding Conventions

Programming has always been relying on conventions to convey additional meaning to the code. The programming language syntax does a lot of the job, for example In C# and Java it's easy to recognize a method *play()* from a variable *play* because methods have parentheses. But this is not enough to tell the difference between class identifiers and variable identifiers.

As a result we rely on naming conventions to quickly distinguish between a class name and a variable name, just by its particular use of lower and upper case. Such conventions are so ubiquitous that they can be considered mandatory.

For example in Java, class names must be in mixed case with the first letter of each internal word capitalized, e.g. *StringBuilder*. This convention is sometime named *CamelCase*. Instance variables follow the same convention except that they must start with a lowercase first letter, e.g. *myStringBuilder*. Constants on the other hand should be all uppercase with words separated by underscores ("_"), e.g. *DAYS_IN_WEEK*. Once familiar with this convention, we don't even think about it any more, and we instantly recognize Classes, variables and CONSTANTS from their case.

Note that the standard Java and C# notation are redundant with the coloring and syntax highlighting of your IDE. Instance variables are in blue color, whereas static variable are underlined etc. So in theory we should not even need the naming convention any longer.

The Hungarian notation is an extreme example of using naming convention to store information, and is definitely not a convention I would recommend.

Hungarian notation is an identifier naming convention in computer programming, in which the name of a variable or function indicates its type or intended use. (Wikipedia)

The idea is to encode the type into a short prefix, e.g. (examples from Wikipedia): - lAccountNum : variable is a long integer ("l") - arru8NumberList : variable is an array of unsigned 8-bit integers ("arru8")

The visible drawback of this notation is that it makes identifiers ugly, as if they were obfuscated.

> A convention is more than just a matter of convenience, it's also a social construct, a social contract between all developers in a community. Once familiar with a convention, we feel at home with it, and feel disturbed whenever we encounter a different convention. Familiarity of a notation also makes it almost invisible, even if it's very cryptic to everyone else.

The Hungarian notation originated in languages without a type system, so you had to use such a notation to remember the type of each variable. However, unless you're still coding in BCPL it's very unlikely you need such notation. It impedes code readability too much, for almost no benefit.

It's unfortunate that C# has kept the convention of prefixing every interface with 'I', as this is reminiscent of Hungarian notation, and has no benefit. From a design perspective we should not even know whether a type is an interface or a class, it does not matter from a caller point of view. In fact we would often start with a class, and later generalize into an interface when really needed, and it should not change the code much. However it's part of the standard convention, so it should be followed, unless all developers involved in the application agree not to.

In language with not built-in support for namespaces, it's common practice to prefix all types with a module-specific prefix:

- *ACMEParser_Controller* // module ACMEParser
- *ACMEParser_Tokenizer* // module ACMEParser
- *ACMECompiler_Optimizer* // module ACMECompiler

This is usually a bad practice, as it pollutes the class names with information that could be factored out in their package (Java) or namespace (C#):

*acme.parser - Controller - Tokenizer*

*acme.compiler - Optimizer*

As we've seen, coding conventions try to extend the syntax of the programming language to support features and semantics that are missing. When you have no type, you have to manage the type by hand with some help from the naming convention. On other hand, if you have types they can help a lot for documentation.

# Integrated Documentation

This documentation is even more integrated into your coding thanks to the autocompletion. This is sometimes called “intellisense” for its ability to guess what you need from the context. As you write code, the IDE shows what’s available.

Write the name of a class and press the dot key. Instantly the IDE Showa a list of every method of the class. In fact it’s not every method, it’s filtered to only show what you can really access in the context of your code under the cursor. It won’t show the private method if you’re not within he class for example.

This is a form of documentation that is task-oriented and highly curated for your context.

## Type Hierarchy

A class hierarchy diagram is a classic element of a reference documentation. Because they usually use the UML notation they take a lot of screen real estate. In contrast, your IDE can display a custom type hierarchy diagram on the fly from any selected class. The diagram is interactive: you select whether to display the hierarchy above or below the selected type; you can expand or fold branches of the hierarchy. And because it’s not using UML it’s quite compact, so you can see a lot in a fraction of the screen.

Imagine you’re looking for a concurrent list with a fixed length but you can’t remember its name. Select the standard ‘List’ super type and ask the IDE for its type hierarchy.

The IDE displays every type that is a list. Now you can examine each type by their name, have a look at their Javadoc by mouse over, and select the one you want. Look ma, no documentation!

Indeed, this is documentation. Just different. Again, this is a form of documentation that is task-oriented and interactively curated for your context.

## Code search

It would be unfair to talk about the IDE without mentioning their searching capabilities.

When you’re looking after a class but don’t remember its name, you can just type stems that belong to its name and the interval search engine will display a list of every type that contains the stems. The same works with just initials of each stem. For example You can type “bis” as a shortcut for “BufferedInputStream”.

(An example please)

## Semantics derived from the actual usage

A colleague at Arolla, Mathieu Pauly, once told me about the idea that meaning comes from the associations between things. One way to learn what a class means is by looking at its relationships with all other classes that we already know.

Superficially this is something that you probably do. Imagine that you need to find every transactional service within the codebase. If the services use annotations like '@Transactional' then it's easy. Select the annotation anywhere, and ask the IDE to find all usage.

Otherwise, let's suppose that transactions are done through the standard Java class 'Transaction' and it's method 'Commit()'. You can ask the IDE to retrieve the call stack for this method. Every class calling directly or indirectly this transaction stuff should be a transactional service. The IDE is an assessment tool.

It's far from perfect. You have to translate your actual goal into what the IDE offers. Still, all these capabilities replace advantageously a lot of documentation that would otherwise be necessary. The IDE is a great integrated documentation tool.

# Naming As The Primary Documentation

*Searching for just the right words is a valuable use of time while designing.*

(@KentBeck & @WardCunningham, http://t.co/PjQfDzRZcX)

One of the most important tools is just naming. Despite its unattractiveness, naming should never be overlooked. More often than not, the names left by the original authors are the only element of documentation available to retrieve their knowledge. Good naming is extremely important. Good naming is immensely important.

But good naming is hard. Names as social convention: need agreement and shared connotations. Thesaurus, conversations, active listening & overheard, names voting to the rescue!

## Naming: browsing a thesaurus

Also good names are not just useful when reading them, they are also useful when you're searching for something. Good naming makes sure all names are searchable. For an example of naming that fails on the Search-able aspect, the programming language named "Go" is quite unlucky, especially considering that it originates from the "Search Company" known as Google.

## Composed methods, you need to name them.

Names don't live in isolation. In object-oriented programming languages, the set of class names form a language, and the words have various relationships to each other, gaining expressivity as a whole. In the paper "A Laboratory For Teaching Object-Oriented Thinking" (1989), @KentBeck and @WardCunningham write:

> The class name of an object creates a vocabulary for discussing a design. Indeed, many people have remarked that object design has more in common with language design than with procedural program design. We urge learners (and spend considerable time ourselves while designing) to find just the right set of words to describe our objects, a set that is internally consistent and evocative in the context of the larger design environment.

For more on naming and practical advices, I suggest reading the chapter on naming written by Tim Ottinger in Robert C. Martin's book "Clean Code".

# Type-Driven Documentation

Types are powerful vehicles to store and convey knowledge, for developers, and for tools to assist too. With a type system you need no Hungarian notation, the type system knows which type is there. That's part of your documentation, regardless of being compile time (Java, C#) or runtime (Javascript, Clojure).

In a Java or C# IDE you can see the type of everything by putting the mouse over it, and a tooltip will tell you about its type.

Primitives are types, but types really shine when you use custom types instead of primitives. For example using:

```
int amount = 12000; // EUR
```

Does not tell the whole story that this quantity is supposed to represent an amount of money, and you need a comment to tell the currency. But if you create a custom type Money, for example as a class, it becomes explicit: now you know it's an amount of money, and the currency is part of the code:

```
Money amount = Money.inEUR(12000);
```

There are many advantages to create types for every concept, but documentation is a very important one. This is not a random integer anymore, it's an amount of money, the type system knows that and can tell you.

We can also check the Money type to know more about it. For example here its class Javadoc comment description:

```
/**
* A quantity of money in Euro (EUR), for accounting purpose, i.e. with an accura\
cy of exactly 1 cent. Not suited for amounts over 1 trillion EUR.
*/
class Money {
...
}
```

That's valuable information, and it's best located in the code itself rather than in random document somewhere else.

**Therefore: Treat your types as an essential part of your documentation. Type everything, and name the types carefully. **

**Therefore: Use types whenever possible, the stronger the better. Avoid bare primitives and bare collections. Promote them into first-class types. Name your types carefully according to the Ubiquitous Language, and add just enough documentation on the types themselves.**

## From primitives to type

In the following example, the code switches on a String, i.e. not a type:

```
validate(String status)
  if (status == "ON")
  ...
  if (status == "OFF")
  ...
  else
    // some error message
```

This kind of code is shameful. Because a String can be anything, we need an additional *else* clause to catch any unexpected value. All this code describes the expected behavior, but if this behavior was done by the type system, e.g. by using a typed enum, there would simply be no code to write at all:

```
switch (Status status){
  case: ON ...
  case: OFF ...
}
```

## Documented Types and Integrated Documentation

A type is a perfect place to put documentation about a concept, in a Javadoc section or its C# equivalent. Such documentation will evolve through the life of the type: it's created when the type is created, and if the type is deleted its documentation will go away with it. If the type is renamed or moved, its documentation remains attached to it, so there is no maintenance.

The only risk is that if you change the definition of the type without changing its documentation you may still end up with misleading documentation. However this risk is as low as possible since the documentation is co-located with the type declaration.

An obvious benefit of using types with their documentation is that this gives you an **Integrated Documentation**, directly within your IDE. When you mouseover a type name, the IDE shows a small popup with the related documentation, anywhere in the code. When you're using the auto-completion, a brief excerpt of the documentation is shown in front of each auto-completion proposal.

## Types and Associations

Associations in code are expressed as member fields to types. The code and its types can tell a lot, but sometimes we need something more. Let's consider a few examples.

When the associations are 1-to-1, and the member fields are well-named, we need nothing more:

```
// nothing to say
private final Location from;
private final Location to;
```

There is no need to tell much when types can **also** express meaning themselves. In the example below, the annotation is redundant with the declared type. It is common knowledge that a `Set` enforces unicity.

```
@Unicity
private final Set<Currency> currencies;
```

Similarly the code below does not need the additional ordering declaration, it is implied by the concrete type. But is it really the case from the caller point of view?

```
@Ordered
Collection<Item> items = new TreeSet<Item>();
```

We could refactor into a new declared type to make the documentation redundant:

```
SortedSet<Item> items = new TreeSet<Item>();
```

But doing that exposes a lot of methods we may not want to expose. Perhaps we would only like to expose `Iterable<Item>`. If that's the case, perhaps the ordering is an internal detail.

We see here is that we prefer **Types over Annotations** as well!

## Types over Comments

Comments can lie and they often do. So does naming, to a less extent. But types don't lie, or the program would not even compile.

A method name may pretend to be:

```
GetCustomerByCity()
```

But regardless of its name, if the signature and its types is actually:

```
List<Prospect> function(ZipCode)
```

You get a much more accurate picture of it really is. And it could even be improved: `List<Prospect>` could be a type in itself, something like `Prospects` or `ProspectionPortfolio`.

With just primitives you're in your own to decide if you can trust the naming or not. What does the Boolean "ignoreOrFail" mean? Enums add accuracy: IGNORE, FAIL

Optional<Customer> expresses the possible absence of result with total accuracy. In languages that support them, monads signal the presence of side-effects with total accuracy. In these examples the information is accurate because the compiler enforces it.

Generics: Map<User, Preference> tells a lot, whatever the variable name.

In case you're still not convinced, here's a study on the topic: Types names help more than documentation[31]

## A Touch of Type-Driven Development

When using types, even if we didn't name the variables we could still know a lot about them thanks to their type. Consider the following variable declaration:

```
FuelCardTransactionReport x = ...
```

The type name tell it all. The variable name will only be useful if there's more than one instance of the same time in the scope.

The same goes for functions and methods. Any function that takes a ShoppingCart as argument and returns a Money has probably something to do with pricing or tax calculation, even without knowing its name. By just looking at the function signature you can have a reasonably good understanding of what the function can do.

In reverse, if you're trying to find the code doing the pricing of the shopping cart, you have two options:

- guess how the class or method is named and perform a search from this guess
- guess its signature in terms of type and perform a search by signature

In Haskell there's a documentation tool called *Hoogle* that will show every function with a given signature. In Java using Eclipse (Kepler), you can also search by method signature. In the search menu, select the Java Search tab, select the radio buttons *Search For*: Method, *Limit To*: Declarations, then type in the Search string:

[31] http://www.slideshare.net/mobile/devnology/what-do-we-really-know-about-the-differences-between-static-and-dynamic-types

```
*(int, int) int
```

**Search by method signature in Eclipse**

You'll get a lot of search results of methods that take two integers as parameters and return another integer, for example:

```
com.sun.tools.javac.util.ArrayUtils.calculateNewLength(int, int) int
com.google.common.math.IntMath.mean(int, int) int
com.google.common.primitives.Ints.compare(int, int) int
org.apache.commons.lang3.RandomUtils.nextInt(int, int) int
org.joda.time.chrono.BasicChronology.getDaysInYearMonth(int, int) int
...
```

It does not just work for primitives like integers, but for any type. For example if we're looking after a method to calculate the distance between two Coordinates (Latitude, Longitude) objects, we would search for the following signature, using the fully qualified type names:

```
*(flottio.domain.Coordinates, flottio.domain.Coordinates) double
```

Which would find the service we were looking for, without knowing its name:

```
GeoDistance.distanceBetween(Coordinates, Coordinates) double
```

You may have heard about Type-Driven Development, or Type-First Development (TFD). These approaches develop around similar ideas around types.

http://techblog.realestate.com.au/the-abject-failure-of-weak-typing/

# Composed Method (Kent Beck)

Clear code does not happen by chance, you have to make it emerge through continuous refactoring, using all your design skills. For example it could be a good idea to follow the 4 Rules of Simple Design expressed by Kent Beck.

http://martinfowler.com/bliki/BeckDesignRules.html https://leanpub.com/4rulesofsimpledesign

Among all the design skills, the Composed Method pattern is particularly relevant for documentation purposes.

> What's this block of code doing?
>
> - It's squishing the fibbly-bar.
> - So should we extract it into a squishFibblyBar function?
>
> [...] clear code, like clear writing, is hard to do. Often you can only tell how to make it clear when someone else looks at it, or you come back to it at a later date.
>
> Ward Cunningham explained it like this. Whenever you have to figure out what code is doing, you are building some understanding in your head. Once you've built it, you should move that understanding into the code so nobody has to build it from scratch in their head again."
>
> Martin Fowler http://martinfowler.com/articles/workflowsOfRefactoring/#comprehension

Composed Method is an essential technique to write clear code. It's about dividing the code into a number of small methods that each perform one task. Because each method is named, method names are the primary documentation.

A common refactoring is to replace a block of code that requires a comment into a composed method named after the comment.

Consider the following example:

```java
public Position equivalentPosition(Trade... trades) {
        // if trades list has no trade
        if (trades.length == 0) {
                // return position of quantity zero
                return new Position(0);
        }
        // return quantity of first trade
        return new Position(trades[0].getQuantity());
}
```

Here the comments suggest that we can do better, like simplifying the code, or extracting methods into composed methods. We'll extract little cohesive blocks of code into their own composed method, like shown below.

```java
public Position equivalentPosition(Trade... trades) {
        if (hasNo(trades)) {
                return positionZero();
        }
        return new Position(quantityOfFirst(trades));
}

//----

private boolean hasNo(Trade[] trades) {
        return trades.length == 0;
}

private Position positionZero() {
        return new Position(0);
}

private static double quantityOfFirst(Trade[] trades) {
        return trades[0].getQuantity();
}
```

Notice that the first method now describes the overall processing, whereas the other 3 methods underneath describe low-level parts of the code. This is another way to make the code clear by organizing the methods into different levels of abstraction.

Here the first method is one level of abstraction above the 3 other methods. Usually we can just read the code in the higher level of abstraction to understand what it does without having to deal with all the code in the lower-levels of abstraction. This makes it more efficient to read and navigate unknown code.

By the way, the code above also illustrates how the layout of text is meaningful: we can graphically see the two levels of abstraction one on top of the other, just through the ordering of the methods.

# Fluent Style

One of the most obvious way to make the code more readable is to make it mimic natural language, a style that is called **Fluent Interface**. Let's take this example taken from a software application to calculate mobile phone billing:

```
Pricing.of(PHONE_CALLS).is(inEuro().withFee(12.50).atRate(0.35));
```

This reads just like English: "The Pricing of phone calls is in Euro, with a fee of 12.50, at a rate of 0.35".

It can grow bigger while remaining readable as a quasi English sentence:

```
Pricing.of(PHONE_CALLS)
  .is(inEuro().withFee(12.50).atRate(0.35))
  .and(TEXT_MESSAGE)
    .are(inEuro().atRate(0.10).withIncluded(30));
```

## An Internal DSL

As seen before, this technique usually relies a lot on method chaining, among other tricks. A Fluent Interface is an example of an Internal DSL, a domain-specific language built on the programming language itself. The advantage is that you get the power of expression without giving up all the good things around your programming language: compiler checking, auto-completion, automated refactoring features etc.

Creating a nice Fluent Interface takes some time and effort, so I would not recommend to make it the default programming style in all situations. It's especially interesting for your Published Interface, the API you expose to all your users, and for anything about configuration, and for testing so that the tests become a living documentation readable by anyone.

A famous example of a Fluent Interface in .Net is the LINQ syntax. It's implemented through Extension Methods, and it manages to mimic SQL quite closely (sample from Wikipedia):

```
var translations = new Dictionary<string, string>
                   {
                       {"cat", "chat"},
                       {"dog", "chien"},
                       {"fish", "poisson"},
                       {"bird", "oiseau"}
                   };

// Find translations for English words containing the letter "a",
// sorted by length and displayed in uppercase
IEnumerable<string> query = translations
        .Where   (t => t.Key.Contains("a"))
        .OrderBy (t => t.Value.Length)
        .Select  (t => t.Value.ToUpper());
```

Here's another example of a Fluent Interface for data validation:

```
using FluentValidation;

public class CustomerValidator: AbstractValidator<Customer> {
  public CustomerValidator() {
    RuleFor(customer => customer.Surname).NotEmpty();
    RuleFor(customer => customer.Forename).NotEmpty().WithMessage("Please specif\
y a first name");
    RuleFor(customer => customer.Discount).NotEqual(0).When(customer => customer\
.HasDiscount);
    RuleFor(customer => customer.Address).Length(20, 250);
    RuleFor(customer =>
  }
```

https://github.com/JeremySkinner/FluentValidation

# Implementing a Fluent Interface

Like in the first step of writing tests in TDD: start by dreaming. Write examples of using the ideal fluent interface, by imagining it's there and perfect, even though you haven't started to built it yet.

Then take a subset of it and start to make it work. You'll probably find difficulties that will force to re-consider alternative ways to express the same behavior.

http://martinfowler.com/bliki/FluentInterface.html http://martinfowler.com/books/dsl.html

# Fluent Tests

A Fluent style is particularly popular for testing. JMock, AssertJ, JGiven and NFluent are well-known libraries to help write tests in a fluent style. When tests are easy to read, they become the documentation of the behaviors of the software.

NFluent is a test assertion library in C# created by Thomas Pierrain. Using NFluent, you can write your test assertions in a fluent way, like this:

```
int? one = 1;
Check.That(one).HasAValue().Which.IsPositive().And.IsEqualTo(1);
```

Through method chaining and many other tricks, in particular around the C# generics, the library allows for a very readable style of tests.

```
var heroes = "Batman and Robin";
Check.That(heroes).Not.Contains("Joker").And.StartsWith("Bat").And.Contains("Rob\
in");
```

An equivalent in Java is AssertJ.

http://n-fluent.net/ http://joel-costigliola.github.io/assertj/

# Test DSL aka DSTL

You can create your own Domain Specific Test Language to enable writing pretty scenarios in plain code. This involves Test Data Builders, obviously.

## Example 2: With builders

Using builders it is not very difficult to create an internal DSL for creating test data. Nat Pryce calls that a Test Data Builder (see ref). (Also see Dave Farley post)

We could extend the previous example with the use of a test data builder to create objects on the given section.

See my github repo. Test data builders can be nested. Les build a bundled travel. We define a bundled travel as a travel that groups flights, accommodation and additional services into one basket so that it's more convenient to buy. We have a test data builder to create each element independently.

```
aFlight().from("CDG").to("JFK").withReturn().inClass(COACH).build();

anHotelRoom("Radisson Blue")
  .from("12/11/2014").forNights(3)
  .withBreakfast(CONTINENTAL).build();
```

We have another test data builder to create the bundle from each product.

```
aBundledTravel("Blue Week-End in NY")
  .with(aFlight().from("CDG").to("JFK")
  .withReturn().inClass(COACH).build())
.with(
   anHotelRoom("Radisson Blue")
   .from("12/11/2014").forNights(3)
   .withBreakfast(CONTINENTAL).build()).build();
```

It is possible that test data builders are so useful that you decide to use them not just for tests. It happened to me to move them into the production code, making sure they are no longer "test" data builders but just regular companion builders with nothing test-specific in them.

See Martin Fowler book on DSL for more on DSL.

## Fluent All the Things! Not.

Fluent is not an end to itself, and coding with a Fluent style is not always the right thing to do:

- It's more complicated to create the API, hence it's not always worth to spend the extra effort
- A Fluent API is sometimes harder to use when writing the code, because of non-idiomatic use of the programming language. In particular it can be confusing to know when to use method chaining or nested functions, or object scoping.
- The methods used as part of a Fluent style have names that are not meaningful on their own, like *Not(), And(), That(), With() or Is().*

# Case Study: An example of refactoring the code, guided by comments

Let's start with this random class, taken from a legacy C# application:

```
public class Position : IEquatable<Position>
{
    //could be just DealId
    private IEnumerable<Position> origin;

    // Position properties to be definied ...
    private double Quantity { get; set; }
    private double Price { get; set; }

    // MAGMA properties to dispatch a job
    public int Id { get; set; }
    public string ContractType { get; set; }
    public string CreationDate { get; set; }
    public string ModificationVersionDate { get; set; }
    public bool FlagDeleted { get; set; }
    public string IndexPayOffTypeCode { get; set; }
    public string IndexPayOffTypeLabel { get; set; }
    public string ScopeKey { get; set; }
    // end MAGMA properties to dispatch a job

    #region constructors
...
```

Notice that most comments delimitate sections. For example the last comment basically tells in plain English: from here to there, this is a sub-section that is used only by the application MAGMA". Unfortunately plain English is code for people, and it takes developers, like you to deal with it each time and time again.

We can do better than these free text comments to describe sections: we can turn them into formal sections represented by distinct classes. This way we turn the fuzzy knowledge in plain English into a strict knowledge expressed in the programming language instead.

Let's do that for the last section:

```
public class MagmaProperties
{
    public int Id { get; set; }
    public string ContractType { get; set; }
    public string CreationDate { get; set; }
    public string ModificationVersionDate { get; set; }
    public bool FlagDeleted { get; set; }
    public string IndexPayOffTypeCode { get; set; }
    public string IndexPayOffTypeLabel { get; set; }
    public string ScopeKey { get; set; }
}
```

We could apply this approach once or twice again here, on the subsets of the fields. For example, Creation Date and Modification Version Date probably go together as a Versioning sub-section that could become a generic shared class:

```
public class AuditTrail
{
    public string CreationDate { get; set; }
    public string ModificationVersionDate { get; set; }
}
```

Doing that opens opportunities to think deeper about what we're doing. For example while naming it AuditTrail it becomes now obvious that it should probably be immutable to prevent mutating the history.

*IndexPayoffTypeCode* & *IndexPayoffTypeLabel* also probably go together, as suggested by their similar naming:

```
IndexPayoffTypeCode
IndexPayoffTypeLabel
```

The prefix of the name acts like a poor-man module name or namespace. Again this would be well-expressed as an actual class:

```
public class IndexPayoffType
{
    public string Code { get; set; }
    public string Label { get; set; }
}
```

We could go on and on, improving the code and its design, purely guided by comments, and naming. Use the formal syntax of your languages instead of fragile and ambiguous text comments.

Comments, sloppy naming and other shameful signals suggest opportunities for improving code. If you see recognize that and you don't know the alternative techniques, this also means you need some external help on Clean Code, Object-Orientated Design or Functional Programming style.

# Living Documentation with Event Sourcing tests

Event Sourcing is a way to capture all changes to an application state as a sequence of events. In this approach, every change to the state of an application (an *aggregate* in DDD terminology) is represented by events which are persisted. The state at a current point in time can be built by applying all past events.

When a user or another part of the system wants to change the state, it sends a command to the corresponding state holder (the "aggregate") through a command handler. The command can be accepted or rejected. In either case, one or more events are sent for everyone interested to know.

Events are named as verbs in the past tense, using nothing but domain vocabulary. Commands are named with imperative verbs, also from the domain language.

We can represent all this in the following way:

```
Given past events
when we handle a command
Then new events are emitted
```

In this approach, each test is a scenario of the expected business behavior, and there is not much to do to make it a business-readable scenario in fluent English. Back to typical BDD goodness, without Cucumber!

**Therefore: You need no "BDD framework" when you're doing Event Sourcing. In this approach, and if the commands and events are named properly after the domain language, then the tests are naturally business-readable scenarios. If you want an additional reporting for non-developers, pretty print the events and the command through simple text transformations in your Event Sourcing testing framework.**

There are many benefits of using Event Sourcing, and one of them is that you get a very decent automated tests and living documentation almost for free. This was initially proposed by Greg Young in various talks[32] and Greg has made his related *Simple.Testing* framework available on Github[33]. This idea was later elaborated by (Jeremie Chassaing](thinkb4coding).

[32] http://skillsmatter.com/podcast/design-architecture/talk-from-greg-young

[33] https://github.com/gregoryyoung/Simple.Testing

# A concrete example in code

Let's take the example of making (and eating) batches of cookies, a neat example taken from Brian Donahue on the CQRS mailing list[34] discussing Greg's approach:

> Given: Batch Created with 20 Cookies
>
> When: Eat Cookies: Amount = 10
>
> Then: Cookies Eaten: AmountEaten = 10, AmountRemaining: 10

For illustration purpose, I've done a similar framework in Java[35] in its simplest possible form.

In this approach, and using this framework, the scenario is written literally in code, through the direct use of Domain Events and Commands which form the Event Sourcing API:

```
1 @Test
2 public void eatHalfOfTheCookies() {
3   scenario("Eat 10 of the 20 cookies of the batch")
4     .Given(new BatchCreatedWithCookiesEvent(20))
5     .When(new EatCookiesCommand(10))
6     .Then(new CookiesEatenEvent(10, 10));
7 }
```

This is a test, as the 'then' clause is an assertion. If no 'CookiesEatenEvent' event is emitted, then this test fails. But it's more than just a test, it's also a part of the living documentation, since running the test also describes the corresponding business behavior in a way that is quite readable, even for non developers:

```
1 Eat 10 of the 20 cookies of the batch
2         Given Batch Created With 20 Cookies
3         When Eat Cookies 10 cookies
4         Then 10 Cookies Eaten and 10 remaining cookies
```

Here the framework just invokes and prints the 'toString()' method of each event and command involved in the test aka scenario. This is as simple as that.

As a result this is not as beautiful and "natural language" as text scenarios written by hand in a tool like Cucumber or Specflow, but still it is not bad.

Of course there can be more than one event in the prior history of the aggregate, and more than one event emitted as a result of applying the command:

[34] https://groups.google.com/forum/#!topic/dddcqrs/JArlssrEXIY

[35] https://github.com/cyriux/jSimpleTesting

```
@Test
public void notEnoughCookiesLeft() {
  scenario("Eat only 12 of the 15 cookies requested")
    .Given(
      new BatchCreatedWithCookiesEvent(20),
       new CookiesEatenEvent(8, 12))
    .When(new EatCookiesCommand(15))
    .Then(
       new CookiesEatenEvent(12, 0),
       new CookiesWereMissingEvent(3));
}
```

This second scenario would print as the following text:

```
Eat only 12 of the 15 cookies requested
        Given Batch Created With 20 Cookies
        And 8 Cookies Eaten and 12 remaining cookies
        When Eat Cookies 15 cookies
        Then 12 Cookies Eaten and 0 remaining cookies
        And 3 Cookies were missing (no more cookies)
```

## How does it work?

This little framework is just a builder producing test cases using Method Chaining[36] between its 3 methods Given(Event…), When(Command) and Then(Event…). Each method just stores the events and command passed as parameters. Calling the Then() method at the end runs the full test and prints its text scenario by calling the toString() method of each event and command, prefixed by the keyword 'Given, When or Then'. When a keyword is repeated, it is aliased by 'And'.

The method 'scenario(title)' instantiates the 'SimpleTest' class of the framework the way you want it to print and log. From there you can elaborate to go further than just tests. For example, we may also use the knowledge from these tests to document the possible behaviors as living diagrams.

## Living Diagrams from Event Sourcing scenarios

In the example shown above the test checks the behavior and prints a description of the business behavior in plain text readable by anyone.

There are several tests, each with different incoming events, commands and outcoming events. The union of all these tests represents the use-cases for the considered aggregate. This is often enough.

[36] https://en.wikipedia.org/wiki/Method_chaining

If you do want to turn that into diagrams, the Event Sourcing-based testing framework can collect all these inputs and outputs across the test suite in order to print a diagram of the incoming commands and the outcoming events.

Each test collects commands and events. When the test suite has completed, it's time to print the diagram in the following fashion:

```
add the aggregate as the central node of the diagram
add each command as a node
add each event as a node

add a link from each command to the aggregate
add a link from the aggregate to each command
```

Once rendered with Graphviz in the browser, we get something like this:

**The generated living diagram of commands, aggregate and events for the Cookies Inventory aggregate**

It is up to you to find this kind of diagram useful or not, or to make your own based on this approach. This example illustrates how automated tests are also a data source to be mined for valuable knowledge that can then be turned into a living document or a living diagram.

Note that the same content could also be rendered as a table:

| Cookies Inventory Commands |
|---|
| BakeCookiesCommand |
| EatCookiesCommand |

| **Cookies Inventory Events** |
| --- |
| BatchCreatedWithCookiesEvent |
| CookiesEatenEvent |
| CookiesWereMissingEvent |

You may also want not to mix scenarios together, or to improve with additional information mixed into the same picture. You may remove the noise of the 'Event' or 'Command' suffixes. Please customize this idea for in your particular context.

# Part 7 Stable Documentation

# Evergreen Document

> An evergreen document is a document written in a way that is relevant to a specific audience over a long period of time. This relevance comes from a universal acceptance or application of document contents. (source: Wikipedia)

An Evergreen Document does not change, and yet it remains useful, relevant and accurate. Obviously not every kind of documents have this privilege.

Evergreen Documents tend to:

- Be rather short, with not much details
- Focus on high-level knowledge, à la "big picture"
- Focus on goals and intentions rather than implementation decisions
- Focus more on the business concepts than on the technical ones

These characteristics are key for the stability of the document.

Note that it's not because the knowledge is stable that it is necessarily useful and worth documenting.

**Traditional means of communication like written documents sent by email are appropriate for knowledge that seldom changes. When this is the case and that the knowledge is useful, don't bother with Living Documentation. Just write the knowledge down in whatever kind of document, like text document, PDF document, content management system, hand-crafted diagram, slides or spreadsheet.**

**You don't have to spend a lot of time on creating Evergreen Documents, but if you do, at least it will benefit the readers for a long time.**

## Design Vs Requirements

> If you can't change a decision, it's a requirement to you. If you can, it's your design."
>
> *Alistair Cockburn* https://twitter.com/TotherAlistair/status/606892091432701952

If you can't change a decision, then this decision has already lost one reason to change. Hence high-level requirements may be stable enough for Evergreen Documents to be well-suited.

Of course this is not usually true in the details of the expected behavior. Low-level requirements like business behavior may change frequently, in which case practices like BDD are more appropriate to deal with the changes efficiently, since conversations are efficient for fast-changing knowledge, together with some automation when it fits.

# Examples

## Company or Project Vision

The best companies have a vision to change the world. This high-level vision is stable and is part of the company identity. Startups at an early stage tend to pivot regularly (see Lean Startup), but the vision often remains the same.

In large corporations change happens all the time everywhere, however the traditional approach to management is to consider most decisions and knowledge as certain, predictable and stable. Within a department, team or project, everything around can often be considered stable.

A Project Vision, when expressed in a few sentences, like an elevator pitch, a project vision can be quite stable. And if it ever changes, the project will probably be stopped or totally reconsidered.

Why does your project exist? Who's sponsoring it? What are the business drivers? What are the expected benefits, and what are the success criteria?

If you're into #NoEstimates, you'll take extra care to keep the vision high-level enough not to prematurely constraint the execution of the project.

For example, the project vision "Create a library to report sales to the regulator" already presumes the solution. Once stated this way, the team has already lost better opportunities, for example extending 2 existing services so that they together can deliver the report. This example of vision is also more fragile to changes. If a new CTO comes and decides that everything now has to be services, no library allowed, you'll have to update the vision of the project.

A better vision for the project would simply be "Report the sales to the regulator", or even better: "Extend reporting to meet MIFID II regulation". It does not matter whatever you do to achieve the goal, every option remains open. Paraphrasing Woody Zuill, "be prepared for the wonderful".

# A lot of knowledge is less stable than it looks.

There's a limit with Evergreen Documents: even if the knowledge itself does not change much, as a document they are also made of a graphical style, with a company logo and company-specific footers. These elements of style change too.

Another limit is that we'd like all documents, including Evergreen Documents, to reside on the same source control as the source code. This encourages lightweight formats of documents, like text and HTML over MS Office documents or other binary proprietary formats. Keeping knowledge in plain text is also the preferred way to go for stable knowledge.

# Case Study: a README file

As an example, let's consider this README file from our fleet management system:

```
# Project Phenix
(Fuel Card Integration)

Project Manager: Andrea Willeave

## Syncs daily
Transaction data from the pump is automatically sent to Fleetio. No more manual \
entry of fuel receipts or downloading and importing fuel transactions across sys\
tems.

## Fuel Card Transaction Monitoring
Transaction data from the pump are verified automatically against various rules \
to detect potential frauds: gas leakage, transactions too far from the vehicle e\
tc.

*The class responsible for that is called FuelCardMonitoring. Anomalies are dete\
cted if the vehicle is further than 300m away from the gas station, of if the tr\
ansaction quantity exceeds the vehicle tank size by more than 5%*

## Odometer readings
When drivers enter mileage at the pump, Fleetio uses that information to trigger\
 service reminders. This time-saving approach helps you stay on top of maintenan\
ce and keeps your vehicles performing their best.

*This module is to be launched in February 2015. Please contact us for more deta\
ils.*

## Smart fuel management
...
```

There are many issues in this file which will require updating the file regularly:

- The project name “Phenix” will change many times for political or marketing reasons
- The name of the project manager will also likely change, in average every 2 years
- The class name will be renamed, split or merged with another at some point if the team is doing refactoring, which we expect to be the case. Each time, this document will need to be updated
- Close to the class name, there are concrete parameters which will change anytime: “300m” will become “500m”, and “5%” can become “3%”
- The launch date is likely to change as it’s already in the past by the way…

How to fix that?

We'll start by changing the title to be a stable name, by reference to the core business of the module. It may not be stable forever either, but at least it is more stable than a name which is driven by internal company politics, from:

```
# Project Phenix
(Fuel Card Integration)

Project Manager: Andrea Willeave
```

to this title, along with a short introduction line:

```
# Fuel Card Integration

Here are the main features of this module:
```

We also got rid of the project manager name in this file. This is not the right place for that piece of information. Instead it could be in a Team section of the wiki, or in the Team section of your project manifest (Maven POM file for example). Note that we could replace the project manager name by a link to the page with this information.

We should also remove the launch date from this file. Instead we could link to the corporate calendar, news portal, dedicated forum or internal social network, or to the Twitter or FaceBook page where the launch will be announced.

The class name has nothing to do here. If we really want to bridge from this file to the code, we may instead link to a search on the source control, something like "link to the classes tagged as @EntryPoint".

Finally, the detailed parameters values are not necessary here. If we really need them, we can either look at the code or configuration, or check the scenarios which describe the expected behavior and which are used by Cucumber or Specflow.

To sum it up:

```
~~# Project Phenix~~
~~(Fuel Card Integration)~~

# Fuel Card Integration

Here are the main features of this module:

~~Project Manager: Andrea Willeave~~

Find who's in the team here // link to the wiki
```

```markdown

## Syncs daily
Transaction data from the pump is automatically sent to Fleetio. No more manual \
entry of fuel receipts or downloading and importing fuel transactions across sys\
tems.

## Fuel Card Transaction Monitoring
Transaction data from the pump are verified automatically against various rules \
to detect potential frauds: gas leakage, transactions too far from the vehicle e\
tc.

~~*The class responsible for that is called FuelCardMonitoring.*~~

**The corresponding code is on the company Github // link to the source code repos\
itory, but not to a concrete class name**

~~*Anomalies are detected if the vehicle is further than 300m away from the gas st\
ation, of if the transaction quantity exceeds the vehicle tank size by more than\
 5%*~~

**For more details on the business rules of the fraud detection, please check the \
business scenerios here // link to the living documentation generated from the C\
ucumber feature files.**

## Odometer readings
When drivers enter mileage at the pump, Fleetio uses that information to trigger\
 service reminders. This time-saving approach helps you stay on top of maintenan\
ce and keeps your vehicles performing their best.

~~*This module is to be launched in February 2015. Please contact us for more deta\
ils.*~~

**For news and announcements on this product, please check our Facebook page // li\
nk to the FB page**

## Smart fuel management
...
```

# Evergreen README

The README file at the root of a code repository has become the norm.

> We have projects with short, badly written, or entirely missing documentation… There must be some middle ground between reams of technical specifications and no specifications at all. And in fact there is. That middle ground is the humble Readme.'
>
> from http://tom.preston-werner.com/2010/08/23/readme-driven-development.html

For a given project `Blabla`, the README file can be safely evergreen if it focuses on answering the following key questions:

- What is Blabla?
- How does Blabla work?
- Who uses Blabla?
- What is Blabla's goal?
- How can your organization benefit from using Blabla?
- How to get started with Blabla. But beware: keep it so simple that it should not change often. In particular, don't embed the version number, instead refer to the place where you can find the most recent version number.
- Licensing information for Blabla (could also be detailed in the LICENCE.txt side car file)

This level of key information is at the same time essential, and at the same time quite stable over time.

Beware instructions on how to develop, use, test, help, contact information except permanent mailing lists.

Also beware when using an online source code repository like Github: avoid linking from the README to pages on the wiki: the README is versioned whereas the wiki is not, so links will break, in particular when cloning or forking.

# In closing

Even in the most fast-changing projects there is still some room for Evergreen Documents, but not for all knowledge. Paying attention to how volatile pieces of knowledge are is a good strategy to reduce your workload over time, by avoiding having to manually update stuff that changes regularly.

All the examples presented here are not rules, but just examples to illustrate the approach. Feel free to judge how often things really change in your own environment. For example, if there is no politics, it may be that arbitrary project names are most stable than naming after the domain language.

Still, projects which can deliver any change in hours will prefer more dynamic forms of documentation over Evergreen Documents. They will rely more on conversations, working collectively, and living documents instead.

# Don’t Mix Strategy Documentation with the documentation of its implementation

*Strategy and its implementation don’t evolve at the same pace*

In the page 80 of their book “Agile Testing: A Practical Guide for Testers and Agile Teams Book”, Lisa Crispin and Janet Gregory recommend not to mix the documentation of a strategy with the documentation of its implementation, taking the example of the test strategy:

> If your organization wants documentation about your overall test approach to projects, consider taking this information and putting it in a static document that doesn’t change much over time. There is a lot of information that is not project specific and can be extracted into a Test Strategy or Test Approach document.
>
> This document can then be used as a reference and needs to be updated only if processes change. A test strategy document can be used to give new employees a high-level understanding of how your test processes work.
>
> I have had success with this approach at several organizations. Processes that were common to all projects were captured into one document. Using this format answered most compliance requirements. Some of the topics that were covered were:
>
> - Testing Practices
> - Story Testing
> - Solution Verification Testing
> - User Acceptance Testing
> - Exploratory Testing
> - Load and Performance Testing
> - Test Automation
> - Test Results
> - Defect Tracking Process
> - Test Tools
> - Test Environments

Therefore: **Don’t mix documentation of a strategy and documentation of its implementation. Make the strategy documentation a pure Evergreen Document. Use another Living Documentation approach for the implementation, considering the implementation will change more frequently**.

The strategy should be documented as an Evergreen Document, stable and even shared between multiple projects. Omit every detail that could change or that would be project-specific from the strategy document. All theses details that change more frequently and that differ from project to project must be kept separately, probably using the techniques proposed in this book which are more suited for knowledge that changes often: declarative automation, BDD etc.

# Vision Statement

*Sharing the vision above the business goals*

Probably the single most important piece of knowledge everybody in the project should absolutely know is the vision of the project or of the product.

> A vision is a picture of the world as it will be when you're done working on it. - mccarthys

With a clear vision, the efforts of each team member can really converge into making the vision come true. A vision is a dream indeed, but a dream that is also call to action for the team who decides to make it real.

A vision often originates in a particular person, who tries to share it with others people using various means:

- a talk, lecture-style, perhaps with great visuals, like a TED talk
- repeating the pitch of the vision often and to everyone
- telling stories who illustrates or exemplify the vision
- writing down a vision statement on paper

All that is a matter of sharing knowledge, in other words it's documentation. A brilliant talk recorded in video may be the best documentation of the vision.

A vision has to be simple enough, and as a result it can be pitched in a few sentences. For example, the vision - or more precisely the "mission", of Fake Grimlock is "TO DESTROY SUCK ON INTERNET, REPLACE WITH AWESOME".

Startup love vision statements, but they sometime lack depth, by just extrapolating on existing successful startups: "It's like Google+, but for oenologists" (from the pitch generator nonstartr.com[37]. Instead, following the advices of Guy Kawasaki, good startups should decide to make the world a better place. For example, Change.org is a for-profit company, a certified B Corporation with a social mission stating: "On Change.org, people everywhere are starting campaigns, mobilising supporters, and working with decision makers to drive solutions."

The perfect companion to a vision statement is a couple of stories that illustrate it and make it more real.

[37] http://www.nonstartr.com

> When a manager comes to me, I don't ask him, 'What's the problem?' I say, 'Tell me the story.' That way I find out what the problem really is. Grocery store chain owner Avram Goldberg, quoted in The Clock of the Long Now, p. 129.

A vision statement is usually on the stable end of the spectrum, at least compared to other project artifacts like source code and configuration data. But it is true that a company pivoting could change its vision several times.

Once the vision is set, it can be split into high-level goals.

# Domain Vision Statement

A particular kind of vision statement focuses on the business domain the product is about. The purpose of this document is to describe the value of the future system to be built, before it actually exists. This description may span several sub-domains, since at the beginning we don't know yet how the domain should be split into sub-parts. The point of the domain vision statement is to focus on the critical aspects of the domain.

In the words of Eric Evans:

> Write a short (∼1 page) description of the CORE DOMAIN and the value it will bring, the "value proposition". Ignore those aspects that do not distinguish this domain from others. Show how the domain model serves and balances diverse interests. Keep it narrow. Write this statement early and revise it as you gain new insights.

Most technical aspects and infrastructure or UI details are not part of the domain vision statement.

Here is an example of Domain Vision Statement for fuel card monitoring in the fleet management business:

> Fuel Card Monitoring of every incoming fuel card transaction helps detect potential abnormal behavior by drivers.
>
> By looking for abuse patterns and by cross-checking facts from various sources, it reports anomalies that are therefore investigated by the fleet management team.
>
> For example, a client using Fuel Card Monitoring with the GPS fleet-tracking features is able to bust an employee for padding hours, falsifying timesheets, and stealing fuel, or buying non-fuel goods with the fuel card.
>
> Each fuel card transaction is verified against vehicle characteristics and its location history, taking into account which driver was assigned to the vehicle at the time and the address of the merchant of the transaction. Fuel Economy can also be calculated, in order to detect engines in need of a repair.

A domain vision statement is useful as a summary of the main concepts of the domain and how they are related in order to deliver value to the users. It can be seen as a proxy for the actual software that is not yet built.

# Goals

The vision is the single most important piece of knowledge everybody should know and keep in mind at all times. From that vision, many decisions will be made to converge to a solution and its implementation.

A vision alone is often not enough for people to start working, and we may have to precise intermediate goals, e.g. to share work between different teams, or to explore early what could be done and their alternatives.

Goals can be described as a tree of goals and sub-goals, with the vision at the root. Goals are lower-level than the vision, but they are high-level compared to all the details that describe how a system is built. As such, they are on the stable side, and the higher-level the more stable.

Goals are also long-term, must be known by most people, and are critical because they drive many further decisions. As a consequence they must be documented in a persistent fashion. Since they are also on the rather stable end of the frequency of change spectrum, traditional forms of documentation fit for documenting goals:

- MS Word documents
- Slide decks
- Paper document

This does not mean that it's easy to make a good documentation of the goals. It's still all too easy to waste a lot of time into a document that will not be read because it's too long or too boring.

Remember there is a danger in deciding goals prematurely, which is the risk to over-constrain the project too early, at a time we know very little about it. This may impede the project execution badly.

This is why Woody Zuill advices on his blog[38] to "Keep your requirements at a very high & general level until just before use", as if they were perishable goods. We do not want to reject opportunities early because of premature sub-goals.

# Impact Mapping

A great technique to explore goals and organize high-level knowledge about a project or a business initiative is Impact Mapping[39], proposed by Gojko Adzic. It advocates working on the goals through interactive workshops and keeping the alternative goals together on the map, to keep options open

[38] http://zuill.us/WoodyZuill/2011/09/30/requirements-hunting-and-gathering/

[39] http://www.impactmapping.org/

during the execution of the project. This collaborative technique remains simple and lightweight, and visualizes assumptions and targets.

A key point is that it shows options and alternate paths to reach the goal. As such it does not constraint the execution as much as other traditional linear "roadmaps".

An impact map itself is rather stable, however it's recommended to reconsider it at low frequency, typically twice a year. On the other hand, tracking the project execution on the map obviously changes often if you release often, and should be not be done by modifying the map each time.

Let's take as an example the result of an Impact Mapping session for a company in the music industry, presented as a tree-like mind-map:

```
Reduce processing cost of song royalties
  IT Department
    100x volumes
    50% Cheaper processing
  Sales Department
    Hourly stats feeds
  Billing Department
    Online real-time reporting (2s or less)
```

Impact Mapping suggests classifying the goals by main stakeholders: IT department, Sales Department and Billing Department in the example above. It also requires the goals to be quantified in the impact maps, with quantitative figures of success, called the "performance targets".

There are other similar techniques like the EVO method Gilb[40] to explore requirements in various ways.

With or without Impact Mapping, a tree of goals is ideally created with sticker notes on a wall. If you want to keep a clean representation for later you can then use any mind-mapping applications like Mindmup, MindNode, Mindjet MindManager, Curio, or MindMeister to record and show a cleaner layout of the map.

These applications can read and write mind maps in various forms, including *indented text*, at least as an "import" option. As a fan of plain text artifacts, I like indented text best!

# Traceability to the goals

How do we know that goals have be reached? Conversely, how do we know why a given module of code has been built in the first place?

Most goals were about new business behaviors. In this case, acceptance tests provide the link between the code implementing the behavior and the corresponding goal. Whatever If the goals are described

[40] http://gilb.com

somewhere, the tests can have an explicit reference to the goal name or identifier, making the link explicit. Acceptance tests usually describe functional behavior, but they can just as well describe other quality attributes like response times, compatibility with a given piece of software or hardware, or even fault-tolerance requirements.

Some goals have related performance targets that cannot be measured at compile time. In this case the associated performance targets can become thresholds into your monitoring tools, and they could have clear labels that link to the goals documented elsewhere too. After all, monitoring is just continuous testing.

Sometimes the performance targets of the goals are described using *fuzzy terms* like "rush hours" or "nominal load". As such they are not quantifiable and therefore not testable or monitorable. However *carefully curated data sets that are also carefully named* can help describe more precisely the phrasing of the expected performances. For example a recorded file of market data activity during a highly volatile market episode can accurately describe what is meant by "highly-volatile markets", a situation of "market crash", "quiet period" or "opening rush". These files can also be used as a basis for acceptance testing.

## Stable Knowledge can also be code

The below idea is fresh and experimental. Try at your own risks!

An Evergreen Document does not have to be English prose or Markdown, it can also be structured as code. Code is an attractive media:

- The compiler enforces every reference from code to code, if a link is broken the compiler throws errors
- Code is easy to refactor with good tool support. You can rename one site and have all the references to it updated safely by the tool.
- Code is formal and easy to parse by tools to process it. You can turn it into various kinds of diagrams, filter items out, enforce stuff, check inconsistencies, or export into a format suitable for another tool.

It's not difficult to turn a tree of goals into code as an internal DSL (Domain-Specific Language). A tree is easy to encode in any programming language. If your goals are documented as a tree expressed in the programming language of the project, you can reference items of the impact map directly from the code, for traceability purposes.

For example, you can add annotations to declare what impact your module is targeting: @ImplementedImpact(MyImpacts.REDUCE__PROCESSING_COST).

# Perennial Naming

Naming is one of the most powerful tools available to transfer knowledge. Unfortunately, many kinds of names change frequently, like marketing brands and product names, project code names or teams names. When this happens, it costs maintenance work: somebody has to chase every place where the old name is used and update them.

Not all names are equal in how often they change. For example, it's common for marketing names, legal names and company organization names to change every 1-to-3 years. These names are volatile.

Choosing names judiciously so that they don't change often is important to reduce the amount of maintenance work in all kinds of artifacts. This is important in the code, and in all other documents.

**Therefore: Use stable names over volatile names in all documentation that you maintain. Name classes, interfaces, methods, code comments and every document after stable names. Avoid references to volatile names in all documents.**

## Organizing artifacts along stable axes

At the macro-level, how do you organize your documentation?

There are many different ways to organize the documents:

- by application name (CarPremiumPro, BestSocksOnline)
- by business process (Sell Car in Retail, sell socks online)
- by target clients (individual car buyers, urban middle-class men; B2B or B2C)
- by team name (team B2B, team Ninjas)
- by team purpose (Software Delivery Paris, R&D London)
- by project name (MarketShareConquest, GoFastWeb)

For each other these organization mode, the important question is: how does it evolve over time? If you think back about your past work experiences, which ones remained unchanged, and which ones were changing from time to time or even several times a year?

Projects start and end. They are cancelled, and sometimes resuscitated under a new name. Applications last longer, but in turn they end up being decommissioned and replaced by another that provides similar business benefits.

## Stability-Oriented

Names describing business benefits are more stable, often over decades. Business is changing, but from a high-level perspective it's still about selling, purchasing, preventing losses and reporting for example. If you open an old book about doing business in your domain, you'll recognize that although the typical way of doing business has evolved since then, most words in the book are still valid and still mean the same thing. Business domain vocabulary is on the stable end of the spectrum.

On the other end of the spectrum, everything about the organization, legal stuff and marketing is volatile: company name, subsidies, brands and trademarks change all the time. Avoid using them in more than one place. Prefer stable names instead.

Look at the company org chart now and compare with the one 2 or 3 years ago: how is it different? New executives often change the org structure. In some companies the top management switches every 3 years. Departments are split and merged, and renamed. It is a game of perpetual business and politics-driven refactoring that changes the org structure without changing the underlying business operations much.

Do you want to spend time changing words everywhere in your code and in your documents because of those changes? I certainly don't want that, therefore I chose to go for stable names whenever I can, with a preference for business domain names.

> I noticed that arbitrary code names, like "SuperOne" that don't describe anything are more volatile than common names that describe what they do. Even if you just work with a company for a 2 or 3 years you will see some of these names changing. But arbitrary names are more attractive, perhaps that's because we change them often to match the current fashion. On the other hand, common words that describe the things, like "AccountingValuation", are dull, but they are less likely to be renamed, hence more stable. More importantly, in the later case, the name itself is an element of documentation. Without anything else, you may know what this component does.

# Knowledge Network

Information is more valuable when it is connected. Relationships convey additional information, and also bring structure.

On a particular topic, or on a project, all information is related to another in some way. On the internet, links between resources adds a lot of value: who's the author? Where to find more? What does this definition mean? Who's quoted here? In a book or paper, the bibliography tells you the context. Was the author aware of this publication? If it's cited in the bibliography then you can guess it was the case.

That's the same with your documentation.

**Therefore: Link knowledge to other related knowledge. Qualify the relationship. Define a clear resource identification scheme. It can be URL, or a citation scheme. Decide on a mechanism to avoid broken links.**

It's important to qualify the link with some meta-data: source, reference on the topic, review, criticism, author, is part of, implements, is composed of, etc.

> Beware the direction of the links. Just like in design, links should go from the less stable to the more stable.

## Linkable Knowledge

A great way to link to some piece of knowledge is to make it accessible through an URL.

Expose them as a web resource accessible through a link. Whenever necessary, refer to them through a link. Use a link registry to ensure the permanence of the links.

Many tools expose their knowledge through links: issue trackers, static analysis tools, planning tools, blogging platforms, social code repositories like Github. If you want to link to a particular version of something, use permalinks (portmanteau of permanent link). If on the other hand you prefer to link to the most recent version of something, link to the front page, or index, or folder, that will usually show the latest version first.

## Volatile To Stable

*When you refer something, make sure the direction of the reference is from the more volatile to to the more stable elements.*

It's way more convenient to couple the volatile to the stable than the over way round. A reference to something stable is not that expensive as there won't be many impacts from the dependency as it does not change often. On the other way round, a reference to a volatile dependency means you'll have to make changes all the time, whenever the dependency changes. This sentence can be read in terms of code, and in terms of documentation just as well.

For an example in code, most programming language propose to couple the implementation to the contract or interface they implement, and not the other way round.

This illustrates the advice *Couple the Specific to the Generic, and the concrete to the more abstract, not the other way round*. This is implied by the fact that generic stuff is usually more stable than more specific stuff. Being common for many cases and shared across many people, it should be more stable, unless you are in pure hell.

In the universe of representing knowledge that we call documentation, prefer references the following ways, not the other way round:

- From the artifacts (code, tests, configuration, resources) to the project goals, constraints and requirements
- From the goals to the project vision

# Link Registry

All links need maintenance, because the web is a living thing, and so does your company internal web. When a link is broken, the last thing your want is to go through every document with the broken link to replace it with another one.

Therefore: **Don't directly include direct links in multiple places in your artifacts**. **Instead use a link registry under that control**.

This link registry gives you intermediate URL's as alias on the actual links. When a link is broken you just need to update the link registry in one single place to redirect to another link.

An internal URL shortener works perfectly as a link registry. Some of these shortener allow to choose your own pretty short link; not only the links become more manageable, they also get shorter and prettier.

I've seen companies install their own on-premise link registry. This is necessary for companies that care a lot of confidentiality of all their knowledge. You can find many URL shorteners that you can install on-premise, some open-source and some with commercial licenses.

# Bookmarked Search

Another way to link in a way that is more robust to change is to link to a bookmarked search instead of linking to a direct resource.

Imagine you want to link to the class 'ScenarioOutline' in a repository. You could link through a direct link, for example in Github you would use a link like this:

```
https://github.com/Arnauld/tzatziki/blob/4d99eeb094bc1d0900d763010b0fea495a5788d\
d/tzatziki-core/src/main/java/tzatziki/analysis/step/ScenarioOutline.java
```

The problem is that this class can move into another package, or its package can be renamed. The class itself could be renamed too, even though it is not so likely, as this concept has been known like this for a long time now. But any of these changes would turn the link into a broken link. That's bad.

We can make the link more robust but using a bookmarked search instead of the direct link. For example we would search for a Java class in this particular repository, with 'ScenarioOutline' in its name.

Using the Github advanced search[41], you would create the following search:

```
ScenarioOutline in:path extension:Java repo:Arnauld/tzatziki
```

Where each options is useful to have a more relevant search:

- ScenarioOutline // search for this term
- in:path // search term must appear in the path name
- extension:Java // the file extension must be Java
- repo:Arnauld/tzatziki // only search in this one repo

The result page of this search will show more than one result, but the one we're looking for is easy to grab in the list (here it is the second result in the list):

```
.../analysis/exec/model/ScenarioOutlineExec.java
.../analysis/step/ScenarioOutline.java
.../pdf/emitter/ScenarioOutlineEmitter.java
.../analysis/exec/gson/ScenarioOutlineExecSerializer.java
.../pdf/model/ScenarioOutlineWithResolved.java
```

Bookmarked advanced search is not just useful for more robust links. It is an important tool for living documentation in general. It offers the power of an IDE for everyone with a browser. By creating curated bookmarked searches, you create guided tours for navigating code and for quickly discovering everything related to a concept, like shown here around the concept of *ScenarioOutline*.

[41] https://help.github.com/articles/searching-code/

# Contract Test as a Reconciliation Mechanism

If you have a link to a direct resource then you need a way to detect when the link is broken.

You can use a Broken Link Checker[42] on your overall documentation for that.

You may also use low-tech Contract Tests that will fail when there is a change breaking the link. This way you know when you have to fix the link or the code so that they get back in sync. This is another example of a reconciliation mechanism.

You create a unit test to compare the code, which could change at any time, against hardcoded laterals which represent the external contract, like the link. When the test fails, you know you have to update the doc, or revert the change in some rarer cases.

For example if the qualified class name is used directly in a link, then the contract test would look like this:

```
@Test
public void checkLinks() {
  assertEquals(
      "flottio.fuelcardmonitoring.domain.FuelCardMonitoring",
      FuelCardMonitoring.class.getName());
}
```

Whenever we refactor, the check against the hardcoded literal would fail to signal we need to make a fix.

[42] https://www.google.fr/search?q=broken+link+checker

# Acknowledge your influences

## Project Bibliography

Good books care about their bibliography. For the reader, it's a way to learn more, but it's also a way to check the influences of the author. When a word has different meanings, looking at the bibliography helps find out how to interpret it.

> Read the Book! – Eric Evans, in Domain-Driven Design

A project bibliography provides a context for the readers. It reveals the influences of the team at the time of building the software.

The project bibliography is composed of links to books, articles and blogs either crafted by hand or extracted from your annotations and comments, or using a mix of both.

## Declare your Style

Like painters that belong to specific painting schools like the Surrealists or the Cubists, there are various schools of thoughts in software development.

Some painters can switch between styles from one work to another; similarly, developers can create a module in a very functional-programming style, with everything pure and immutable, and another using semantic technologies and graph-oriented stores.

To provide context for the readers, it is useful to declare the style and the main paradigm if any that you chose for some area of code, typically for a module or for one project.

This overall statement show some similarity to a resume, but for the team or teams: - Modeling Paradigms (DDD) - Authors we follow - Books we've read, blogs we often go to - Languages and frameworks we're familiar with - Any kind of inspiration that matters, e.g. "Stripe as an inspiration for developer-friendliness" - Typical kinds of projects we've mostly done so far (web, server, embedded, crud…)

To be refactoring-proof, this information should reside within the module or project itself. It can be done with annotations @Style(Styles.FP) on packages (Java), attributes on the AssemblyInfo (.Net) or using a style.txt file with a key-value syntax at the root of the module or project.

Style is also useful for tools; for example the declared style can be linked to specific rulesets for static analysis.

Declaring your style also helps enforce consistency within area of the code base.

**LOL**

Coined Gierke's law yesterday: from the structure of a software system you can derive the book the architect read most recently… From Oliver Gierke @olivergierke[a] on Twitter

[a] https://twitter.com/olivergierke

# Domain Immersion

When working on a new business domain, there is a lot to learn quickly on this new domain.

Traditionally, the project itself is the main way to learn. Task after task, each work part brings new vocabulary and new concepts that are learnt on the job, because this is necessary to do the job.

This presents a number of weaknesses.

There is not enough time to deliver a task and to study seriously a part of the business domain more in-depth. Learning remain superficial.

Many tasks can be done with only superficial understanding of the underlying business. It may appear to work by coincidence, while really being a time bomb for next business requirements.

Even if you decide to dedicate 2h out of the task to learn, the domain experts may not be available at that time, and maybe not before next week.

Whenever the lack of domain knowledge is the bottleneck, it's an attractive proposition to invest some time early on to learn the domain. One of the best way to do that is by immersion.

**Therefore: Invest time early to immerse the team into domain. Visit the place where the business actually takes place. Take pictures. Get copies of the documents being used. Listen carefully to the conversations of the business people. When possible, ask questions. Make sketches of what you see and take plenty of notes.**

Domain Immersion is also an effective practice for new joiners to quickly discover what the domain is about. As such, it is an alternative form of knowledge transfer, directly from the field, which also means it is a genuine form of documentation.

Sometime it is not possible, or prohibitively expensive to go to the field, in which case we need cheaper alternatives for this precious knowledge, like an Investigation Wall or simply trainings.

## Investigation Wall

You may even create a wall of findings, much like investigation walls in criminal investigation movies, where the detectives cover the walls with lot a of pictures, notes, maps with pins to fully immerse with the dossier.

Similarly you can dedicate a space on the wall with pictures, notes, sketches and sample business documents to keep a feel of the actual business domain while you work for it.

## Domain Training

Once there, the next step would be to register the team, or part of the team, to specialized trainings about the business domain.

In one of my past projects we've decided to invest in domain knowledge early, when the pressure was not so strong: twice a week, we dedicated 30 mn after lunch for a mini-training session. A business analysis or a product manager that was identified with a particular area of expertise joined the team as the domain expert to explain all we needed to know on one concept at a time: one session on bond coupons, another about standard financial options, another on a new regulation etc. It was considered useful by the team, all the developers enjoyed it.

## Live-my-Life Sessions

Going even further, you may try "live my life" sessions. For a period of time from half a day to 2 days, one or two developers stay close to someone doing business operations, to see what's really like to work in the business, using the software tools they have. It may be in the back, trying not to interfere and just watching passively. However it's best to have the ability to ask questions at any time, or during some predefined pauses.

The experiment may be more involved, like being an assistant of the business person. Some companies go further and have employes completely switch their role for one day. As a developer, doing the job of an accountant for one day can be one of the best way to get to appreciate their stakes, and therefore to improve their software. It can also do wonders for the User Experience.

## Shadow User

A variant of this idea is to watch the behavior of the users as a "shadow user". You login as another real user, in a readonly fashion, and you see their screen in realtime. This is very valuable to watch how they actually use the software to achieve their business objectives.

This is obviously not feasible in many cases, mainly for privacy reasons, or because the installed software is not accesible. You also need this feature of "shadow user" in order to do that.

## A long-term investment

All this can be seen as an *investment*, because the business domain is usually quite stable. The details of doing the business do change all the time, but the business still uses the same old concepts.

I realized that in 2007 when I opened a book on Finance written in 1992. The book was still relevant in all its content, except the examples were no longer realistic: interest rates in 1992 were often

around 12-15% in some currencies, whereas 15 years later they were closer to 2%. And at the time of writing this book, they now are around 0.2%!

Even books written well before the advent of computers would remain interesting.

Another direct way to look at that an an investment is that all this contextual knowledge will inform many decisions every day, every minute, to make them better. And all the domain-specific words learnt as an investment will make discussions during meetings more efficient. You won't spend the first part of each meeting on clarifying the vocabulary any more.

# 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

# Conversations Over Formal Documentation

> Having conversations, Is more important then documenting conversations, Is more important then automating conversations – Liz Kheogh

Written documentation is often the default choice when it comes about documentation, to the point that the word "Documentation" has become a close synonymous to "written document".

However it's unlucky. When we need documentation, we mean that there's a need for knowledge transfer from some people to other people. The bad news is that not all media are equal when it comes to their efficiency of transferring knowledge.

Alistair Cockburn analyzed three dozen projects, over the course of two decades. He reported[43] on his findings in books and articles, with a famous diagram illustrating the effectiveness of different modes of communication.

[43] http://alistair.cockburn.us/ASD+book+extract%3A+%22Communicating,+cooperating+teams%22

This diagram recaps his observation that people working and talking together at the whiteboard is the most effective mode of communication, whereas paper is the least effective.

Most of the time, effective sharing of knowledge is best done by simply talking, asking and answering questions instead of written documents.

**Therefore: Favor conversations between everybody involved over written documents. Conversations are interactive, fast, convey feelings and have a high bandwidth as opposed to all written artifacts.**

> A phone call can save twenty emails. A face to face chat can save twenty phone calls – @geoffcwatts on Twitter

Conversations are:

- **High-Bandwidth**: conversations offer a high bandwidth compared to writing plus reading, as more knowledge can be efficiently communicated over a given period of time
- **Interactive**: both sides of the conversation have the opportunity to ask for clarification and orient the topic on what's most useful for them, whenever they want to
- **Just In Time**: both sides of the conversation only talk about what's of interest for them

These key properties of conversations make them the most effective form of communication for sharing knowledge.

In contrast, written documentation is not just wasteful because it takes time to write, but also because it takes time to locate where the relevant parts are, and then it's unlikely the content will fit the expectations. Even worse, it's likely that the content will be misunderstood.

## Wiio's laws

Wiio's laws are humoristically formulated serious observations about how human communication usually fails except by accident, by Professor Osmo Antero Wiio.

> Communication usually fails, except by accident.
>
> If communication can fail, it will.
>
> - If communication cannot fail, it still most usually fails.
> - If communication seems to succeed in the intended way, there's a misunderstanding.
> - If you are content with your message, communication certainly fails.
> - If a message can be interpreted in several ways, it will be interpreted in a manner that maximizes the damage.
>
> From Wikipedia[44]

Human communication works best through interactive dialogues, with the opportunity for the receiver of information to react, disagree, rephrase or ask for more explanation. This feedback mechanism is essential to fix the curse of one-way human communication highlighted by Professor Wiio.

Alistair Cockburn has similar findings:

> To make communications as effective as possible, it is essential to improve the likelihood that the receiver can jump the communication gaps that are always present. The sender needs to touch into the highest level of shared experience with the receiver. The two people should provide constant feedback to each other in this process so that they can detect the extent to which they miss their intention.

A face-to-face, interactive and spontaneous form of documentation is the best way to improve on the fate of miscommunication highlighted by Professor Wiio. If all your stakeholders are happy with talking with the team for all questions and feedback, then change nothing. You don't need written documentation.

[44] https://en.wikipedia.org/wiki/Wiio%27s_laws

Agile documentation goal is to "help people interact"

- knowing who to contact
- knowing how to work on the project, guidelines, style and inspirations
- sharing the same vocabulary
- sharing the same mental model, metaphors
- sharing the same goal

## The Rule of Three Interpretations

Jerry Weinberg writes in his book "Quality Software Management Vol 2 (Chapter 6)":

> Whenever I'm aware that I'm making an interpretation, I have another choice: I can allow myself to know that more than one interpretation is possible. A good check on premature interpretation is the **Rule of Three Interpretations**:
>
> If I can't think of at least three different interpretations of what I received, I haven't thought enough about what it might mean.
>
> This rule slows down the Interpretation step and gives me, the receiver, a chance to engage my brain before using my mouth. Even after I have thought of three possible interpretations, however, I should always be aware of one more possibility: that my list still may not include your intending meaning.

## Obstacles to conversations

There would be no need for this pattern if people were having conversations so easily in the workplace. Unfortunately this is not the case often enough.

Years of working together by handing documents over the wall have trained many people at not having conversations, except in meetings where conversations become an art of negotiation. Corporate environments with politics and information retention have also trained colleagues at not sharing too much knowledge too early in order to remain in the game and to keep power, including blocking power.

People from different teams, departments, or assigned on different projects, in different locations, tend to have much less conversations than close neighbors in the same team and project. They tend to use colder (not interactive) and less effective modes of communication like email or phone calls instead of face-to-face communication. It's important to note that hierarchical distance – not having the same management, is at least as great an impediment to having conversations than geographic distance.

Separation of people by functions in separate teams, like the Dev, QA and BA teams, is also a great way to make conversations less likely.

The idea of ownership of activities is another conversation-killer:

- Product "Manager"
- Product "Owner"
- Scrum "Master"

> I have no idea why people aren't collaborating!
>
> – Melissa Perri (@lissijean) on Twitter

Old clichés also reduce the likelihood that people even imagine meeting and talking together:

> "I'm a tester, I must wait for the development to be finished to start testing"
>
> "I'm a BA, so I must solve the problem by myself before handing it to the developers to implement"
>
> "I'm a developer, my job is to execute what's been specified beforehand, and my job is not to test it once it's done."

I've heard that some Business Analysts have a hard time imagining not producing documents of a large enough size, for fear that their work is not visible otherwise. Simply talking to help the project may not be enough to justify their role. Here we see how perverse this system has become, producing waste (large early documents) not for their value per se but to make the work visible to managers. Fear of losing your job or individual incentives feed this kind of counter-productive behaviors.

> Defect tracking systems certainly don't promote communication between programmers and testers. They can make it easy to avoid talking directly to each other. – Agile Testing, Lisa Crispin and Janet Gregory

To improve on that, make sure that everybody knows that the only goal is to deliver value. Make the work environment safe for everyone. Even with much less documents, there's still a role for traditional BA and QA team members, it's just transforming into a continuous contribution to a collective adventure that we call a project or a product.

Make sure it's perfectly ok to just have conversations often, and spend less time writing stuff. Promote collective working over separate job posts. Have everyone, even from different teams, sit close to each other most of the time, around the same table if possible, so that spontaneous communication happens without obstacle.

# Knowledge Transfer sessions

Some teams plan *KT* (Knowledge Transfer sessions), often in addition to brief documents, to make sure the knowledge is actually shared and well understood. According to the Wiio's laws, that is a great idea! A typical example of KT is to exchange the knowledge on the deployment before a release, when the Ops are in another silo of the organization. One way to share knowledge in this case is to perform a dry run of the deployment, based on the deployment document and all the automated deployment manifests. This way, any issue, question or mistake can be spotted quickly during the session, and all this during regular working hours.

Of course an alternative is to directly work collectively between the developers and the Ops people to prepare, configure and document all the deployment process. KT can be a step in this direction for traditional companies, just like code reviews are a step towards pair-programming.

# Interactive Documentation

Written documents don't have the opportunity for interaction. As Korpela[45] comments on the Wiio's laws, whenever a written document "such as a book or a Web page or a newspaper article, miraculously works, it's because the author participated in dialogues elsewhere.".

It takes more work than just typing text for a written document to be useful. Georges Dinwiddie advises in his blog[46] to "Document questions the reader may have" and to "Get it reviewed by multiple people". As such, the written documentation is like a record of an interactive conversation that worked, which makes it more likely to work again.

But we can also push the limits of written words on paper thanks to the available technologies all around us. We can create documentation that is interactive to some extent.

As an example, Gojko Adzic turned a checklist of test heuristics into an additional menu in the browser, as a small assistant called BugMagnet[47]:

**BugMagnet**

[45] http://www.cs.tut.fi/~jkorpela/wiio.html
[46] http://blog.gdinwiddie.com/2010/08/06/the-use-of-documentation/
[47] https://github.com/gojko/bugmagnet

Clicking on the item “Names / NULL” in the menu directly fills the edit field in the browser with the string “NULL”. This could have remained a plain checklist to input yourself manually into the forms, but Gojko made the extra step to make it a little more interactive. Note the suggestive effect of navigating the menu, it calls for being used, at least more than a printed checklist.

Therefore: **Whenever possible prefer documentation that is interactive over static written words**. **Use hypermedia to the content navigable through links**. **Turn the documentation into tools like checkers**, **tasks assistants or search engines**.

You already know several examples of interactive documentation, it is around us already: - Hypermedia documentation with navigable links, as generated by Javadoc and equivalent systems in other languages - Tools like Pickles which turn the Cucumber or Specflow reports into an interactive website, or Fitnesse which had always been interactive from the start. - Tools like Swagger which document your web API into an interactive website, with built-in capability to directly send the requests and to show the responses - Your IDE which offers a lot of documentation features with a keystroke or a mouse click: Call Stack, Search for type or reference, Type Hierarchy, Find occurrences, Find in the programming language Abstract Syntax Tree…

As described in Declarative Automation, promoting documentation into an automated form that is also readable allows for interactive discovery: you can execute and tinker the automation code (scripts and tests) to understand the topic more more in depth as you change it and see the effects.

# Working Collectively

Conversations are good. When creating software, we need to have conversations, and we need to program code. It's often a great idea to do all that at once, continuously, together with one or more colleagues.

There are many good reasons for working collectively, like improving the quality of the software for its users and for its maintainers, thanks to the continuous review and the continuous discussions on the design.

But working collectively, with frequent conversations, is a particularly effective form of documentation too. Pair-programming, Cross-programming, Mob-programming and the 3 Amigos 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.

## Pair-Programming

Pair-Programming is a key technique from Extreme Programming. If code reviews are good, why not do them all the time?

> Pair-Programming is an agile software development technique in which two programmers work as a pair together on one workstation. One, the driver, writes code while the other, the observer, pointer or navigator, reviews each line of code as it is typed in.
>
> https://en.wikipedia.org/wiki/Pair_programming

Or in a more evocative form:

> OH: "Mob programming. It's like 'pair programming meets RAID6'" From Phil Calcado @olivergierke[48] on Twitter

In Pair-programming, the driver writing code talks his or her mind out loud for the observer to follow what's happening, who's in turn replies with acknowledgement, remarks, corrections or any other kind of feedback. The observer, also known as the *navigator*, talks to the driver to guide the work in progress, suggesting possible next steps and expressing the strategy for solving the task.

Working in pair is not something you are comfortable and good at immediately, it's something you learn through practice, on the job or in coding dojos or code retreats. There are various styles of

[48] https://twitter.com/pcalcado

pair-programming, like the ping-pong pairing: one of the pair writes a failing test then passes the keyboard for the other to make it pass and refactor.

For sharing the knowledge as much as possible in order to have true Collective Ownership, in pair-programming it's common to change the partners in the pairs on a given task regularly. Depending on the teams, this pairs rotation can happen as frequently as every hour, or every day, or just once a week. Some teams don't have a fixed frequency but require that any task cannot be finished by the pair who started it.

> Pair programming: the best way to do less email, attend fewer meetings, AND write less documentation! – @sarahmei on Twitter

## Cross-Programming

> Cross-Programming is a variant of Pair-Programming where the observer is not a developer but a business expert. Whenever the programming task requires a deep understanding of the business domain, it's a form of collaboration that is highly efficient but also very effective as all decisions taken by the pair in front of the computer are more relevant to the business.
>
> https://speakerdeck.com/fakih/cross-programming-forging-the-future-of-programming

## Mob-programming

Mob-programming is a recent addition to the bestiary of collective forms of programming, and has quickly gained popularity. If extreme programming turned the code review knob to the max (10), mob-programming goes even further, turning it to 11.

> Mob programming is a software development approach where the whole team works on the same thing, at the same time, in the same space, and at the same computer. This is similar to pair programming where two people sit at the same computer and collaborate on the same code at the same time. With Mob Programming the collaboration is extended to everyone on the team, while still using a single computer for writing the code and inputting it into the code base.
>
> "All the brilliant people working at the same time, in the same space, at the same computer, on the same thing" – Woody Zuill
>
> mobprogramming.org

In Mob-programming there is no question of pair rotation as everybody's always present on any task so everybody knows about every task. That's literally collective ownership, in the same place at the same time.

If a team of 5 people doing mob-programming full time, knowledge sharing is no longer an issue, it's done continuously, every second. Whenever someone has to attend a meeting outside, the rest of the team keeps on working, almost unaffected.

> ## The 3 Amigos (or more)
>
> a Product Owner, a Developer, and a Tester sit down to talk about something that the system under development should do. The Product Owner describes the user story. The Developer and Tester ask questions (and make suggestions) until they think they can answer the basic question, "How will I know that this story has been accomplished?""
>
> No matter how or when it's done, these three amigos (to borrow a term from my friends at Nationwide) must agree on this basic criteria or things will go wrong. http://blog.gdinwiddie.com/2009/06/17/if-you-dont-automate-acceptance-tests/

The concept of the 3 Amigos working together during *Specification Workshops* is central to the BDD approach. In contrast with Pair-programming, Cross-programming and Mob-programming, they are not working on code but on concrete scenarios describing the expected business behavior of the software to build. Still, everyone involved owns the scenarios, and it does not matter who writes them down on paper or in a the test automation tool like Cucumber, as it's on behalf of every other. We use the term "3 amigos", but in practice they may be more than three whenever another perspective is key for the success of the work. There may be a need for a UX expert, an Ops etc.

# Event Storming as an on-boarding process

Alberto says that some teams find it valuable to run a new Event Storming session whenever a new member joins the team, as an fast on-boarding mechanism.

I can testify it works very well for that. As a consultant spending just a few days always in new teams with new domains, I need to learn as much as possible of the new domain in a short period of time. Recently I've used short Event Storming sessions for that, even if the team had dome it several times before. It is really impressive how much you can learn in just 2 hours with this kind of workshop.

It happened recently that a business domain expert in the Event Storming told us he had already created well-crafted diagrams on the domain. Once we were mostly done with posting the events on the wall and organizing them, he drew the diagram on the whiteboard.

What was interesting is that his diagram was in many ways more complete than our wall of events. Still, the interactive workshop form meant that we were all much more engaged with our wall of stickers than is typically the case when just looking at a static diagram. The session became a game of comparing the diagram and the events wall to better understand both, and a lot of new insights appeared in this process!

## Continuous Documentation

Collective forms of work optimize for continuous documentation. Because face-to-face interactive conversations are the most efficient form of communication, Pair-programming, Cross-programming, the 3 Amigos or Mob-programming organize the work precisely to maximize the opportunities for effective conversations. Documentation happens at the very time the knowledge is necessary. Everyone who must know about it is present. They can immediately ask questions to clarify a point.

When the task is done, they remember some of the key parts of knowledge, and can forget the rest. If someone goes on vacations, the knowledge is safe in his or her colleagues mind, so it does not impede the work in progress.

## Truck Factor

Working collectively is very good to improve the Truck Factor of a project.

> ### Truck Factor
>
> "The number of people on your team who have to be hit with a truck before the project is in serious trouble"

The Truck Factor is a measurement of the concentration of information in individual team members. a truck factor of one means that only one person knows critical parts of the system, and if that person is not available it would be hard to recover the knowledge.

When several team members collaborate on every part of a project, knowledge is naturally replicated in more people. When they leave, or go on vacations or just leave for a meeting, the work can carry on without them.

A small truck factor usually means someone is a hero on the project, with a lot of knowledge not shared with other team mates. This is definitely a problem for the resilience of the project that the management should be aware of. Introducing collective forms of programming is a nice answer to mitigate that risk. Moving the hero to another team nearby is another way to deal with that.

Conversations and working collectively represent the ideal form of documentation for most knowledge. However it's not enough for knowledge that is essential in the long term, when all team members are gone or have forgotten knowledge from the remote past. It's not enough for knowledge that is of interest to a large number of people, and it's not enough for knowledge that's too critical to be left as spoken words.

# Coffee Machine Communication

*Not all exchange of knowledge has to be planned and managed. Spontaneous discussions in a relaxed environment often work better and must be encouraged.*

Random discussions at the coffee machine or at the water fountain are invaluable. The best exchange of knowledge is spontaneous. You meet a colleague or two and start talking. Follows something like a content negotiation to find topics each of you is interested in. It may be on a non professional topic. In this case this is just bond-making, which is also invaluable. When it is on a professional topic, then nothing can beat this kind of communication.

You've chosen this topic because all of you have an interest in it. You have questions about your current tasks, and the other people are happy to help with answers or stories from their own experience.

I believe that this kind of communication is the best way to exchange knowledge there is. Topic is chosen freely from shared interests. It's interactive with questions and answers and a lot of spontaneous storytelling. It takes as long as required. I've already missed meetings because the discussion at the coffee machine was way more essential to a project than the meeting I was supposed to attend.

Open Space Technology used for meet ups and un-conferences replicates just that idea setting for larger groups. The rules of the two feet states that everyone is free to move where the topic is most interesting. The other principles say that "The people who are there are the right persons" and that "Whenever it starts it's the right time".

For all this to work there must be no hierarchy pressure around the coffee machine. Everybody is free to chat with the CEO without being specially formal or shy.

**Therefore: Don't discount the value of random discussions at the coffee machine, water fountain or in the relaxation area. Create opportunities for everyone to meet and talk at random, in a relaxed setting. Decree that the rank in the hierarchy must be ignored within all relaxed areas.**

Google and other web startups propose fantastic facilities to encourage people to meet and talk.

> Just ask Jeff Dean, the famed Googler who often is referred to as the Chuck Norris of the Internet. As the 20th Googler, Dean has a laundry list of impressive achievements, including spearheading the design and implementation of the advertising serving system. Dean pushed limits by achieving great heights in the unfamiliar domain of deep learning, but he couldn't have done it without proactively getting a collective total of 20,000 cappuccinos with his colleagues.
>
> "I didn't know much about neural networks, but I did know a lot about distributed systems, and I

> just went up to people in the kitchen or wherever and talked to them," Dean told Slate. "You find you can learn really quickly and solve a lot of big problems just by talking to other experts and working together." source[a]
>
> [a] http://techcrunch.com/2015/09/11/legendary-productivity-and-the-fear-of-modern-programming/

La Gaité Lyrique, a place for cultures in the digital age in Paris, has offices and meeting rooms, however the staff often prefers to host meetings in the foyers that are open to the public. They even serve beer there, but I haven't seen people from the staff drink beer during the day.

**The informal foyer where most meetings take place**

I've spent countless hours in their foyers writing this book. I've seen benefits that we miss in traditional work environments with closed meeting rooms.

**The atmosphere**: because it's mixed with people from the outside, many working, other having fun around a tea or even beer, the atmosphere is quite relax. This is more pleasant. I believe it also encourages thinking more creatively. You also have the choice of low sofas and lounge chairs, or

dining tables with kitchen chairs. On a rather tense topic I'd go for the lounge setting each time! To work on a diagram, I'd chose the dining table.

**Impromptu discussions**: for example, the General Director had a meeting with two people from the staff. They didn't book a space. Once done with the discussion, he then looked around to see who was there, then went on to have very brief side discussions with colleagues that were attending another meeting in the foyer.

Thinking in retrospect of all the frustration of planning meetings with busy people, in boring meeting rooms in the company I was working for, I was jealous.

Being there with the staff also means I had the opportunity to ask questions to the director himself in an impromptu fashion. No appointment. No secretary to filter access. Wow.

The director definitely encourages informal meetings. Spending leisure time at the foyer instead of working is not a problem since everyone owns their responsibilities, regardless of how, when, where or how long they work. Impromptu meetings can be totally improvised, à la coffee machine, or just planned in an informal space, like in the coffee machine area.

All this is not suited for every case, of course. There is no guarantee you'll find the people you want to talk to around the coffee machine, unless you planned the meeting. There's also no flip chart, no whiteboard, and unfortunately no tele-conference system. And there is no privacy.

# Ideas Sedimentation

A lot of knowledge is only important at the moment it's created. You debate design options, try one, find out it's not right, try another. After some time it's obvious it was the right choice, and the choice is visible in the code. It's already there. No need to do anything more.

You discuss options around the coffee machine. You simulate how they perform mentally. Everybody agrees on the best option. Then a pair goes back to their computer to implement it. The knowledge exchanged and created during the discussion was important *at that particular time*. But the day after it's already nothing more than a mere detail.

Once in a while, some of this knowledge remains important, even after a while. It gets reinforced, until it's worth recording to be shared to larger audience and for the future.

**Therefore: Favor quick, fast, cheap interactive means of knowledge exchange like conversations, sketching and sticker notes by default. Only promote the fraction of knowledge which proved repeatedly useful, critical or that everybody should know.**

- Start with impromptu conversations, later turn the key bits into something permanent: Augmented code, Evergreen document, or anything durable.

  The sedimentation metaphor relates to ideas flowing like sand in the water flowing quickly in the water streams. The sand particles move away quickly, but some of them

become sediment at the bottom of the river, where they accumulate slowly. A similar process is at work in a wine decanter.

**Particles of the wine decanting**

- Start from Napkin sketch to document a design aspect , later if it proves essential turn it into something maintainable like a plain text diagram or a living diagram, or a Visible Test.
- Start with bullet points to document the quality attributes, later when it hasn't changed much turn them into executable scenarios

Memory is the residue of thought." - simple but profound realization that is so important to my work. I intend to honor it more fully. Tim Ottinger on Twitter.

**Conversations to Traces**

## Throw-Away Documentation

*Documentation that's only useful for a limited period of time, before it can be deleted.*

You need a specific diagram while you're designing around a problem. Once you're done with the problem, the diagram immediately loses most of its value because now nobody cares any more about the focus of this diagram. And for the next problem, you'd likely need another completely different diagram with another focus.

**Therefore: Don't hesitate to throw away documentation that is specific to a particular problem.**

When it's worthwhile to archive a diagram, turn it into a blog post, telling the story with the diagram as an illustration.

One important set of transient documentation is everything about planning, like the User Stories and everything about estimation, tracking etc. User Story is only useful just before development. A burn down chart is only useful during an iteration. You may want to keep the stats to check later how hard it is to plan and estimate, but that is something different. Throw the User Stories stickies away after the iteration.

# On-Demand Documentation

The best documentation is the one that you really need and that suits actual purposes. The best way to achieve that is to create the documentation on-demand, in response to actual needs.

The need we have right now is a proven need from a real person. It's not a speculation of *something* that *someone* could find useful in *some future*. The need we have right now is precise and has a purpose, and it can be expressed as a question. The documentation to be created will just have to answer the question. This is a simple algorithm to decide *when* to create documentation about *what* topic.

## Just-In-Time Documentation

Documentation is best introduced just-in-time. The need for documentation is a precious feedback, a "Knowledge Gap" signal that should trigger some documentation action in response. The most important bit of documentation may be the one that is missing. Listen to knowledge frustrations to decide when to fill the gap.

> The idea of Just-In-Time Documentation are inspired from the *Pull System* of Lean. A pull system is a production or service process which is designed to deliver goods or services as they are required by the customer or, within the production process, when required by the next step.

Still you may not invest time in some documentation action on each question. There's a need for some threshold:

- Some follow the "Rule of 2": Once you have to answer the same question twice, start the documentation about it.
- Open-Source project sometime rely on the community votes to decide what to spend time on, including for the documentation.
- Commercial products sometime rely on website analytics to decide what to spend time on, including for the documentation.
- Peter Hilton on Documentation Avoidance[49] has his own take on this process, a bit similar to the Rule of 2:

1. Don't write the docs.

[49] https://www.slideshare.net/pirhilton/documentation-avoidance

2. Bluff, “it’s on the wiki”.
3. Still don’t write the docs.
4. If they come back, feign ignorance.
5. Then write it.
6. Later, casually mention that you “found it”.

In practice, you can keep it low-tech: every time you’re asked for information that you don’t have any documentation already available for, log the request as a sticky note on a wall.

Whenever you have repeated requests for a similar kind of information you can decide as a team to invest some minimal work to create it. It’s a rustic voting mechanism on the wall.

Start manual and informal; observe and discuss the stickers during the team ceremonies; throw away or promote into a clean automated documentation as a result.

Start by explaining interactively, using whatever existing and improvised support: browsing the source code, searching and visualizing in the IDE, sketching on paper or whiteboard, or even in PowerPoint or Keynote as a quick drawing pad (it’s sometime easier to use a tool when you need a lot of “copy-paste-change a little” kinds of sketches). Then immediately refactor the key parts of the explanation into a little section of documentation. You know what parts the explanations are key mainly from the interactions with the colleague. If it was difficult to understand, or surprising, or if it was recognized as a “Aha! Moment” by your counterparty, then it’s probably worth keeping for other people later.

Peter Hilton has another fantastic trick to write doc, which he calls “Reverse Just-In-Time Doc”:

> Instead of writing documentation in advance, you can trick other people into writing JIT documentation by asking questions in a chat room (and then pasting their answers into the docs)

## Provoking Just-In-Time Learning Early

Fixing bugs or making small evolutions, from the code to production, is a great way to learn quickly about an application and its complete development process. That’s why many companies include bug fixing and minor evolutions tasks as part of the immediate on-boarding process for newcomers. This creates needs for knowledge, which in itself triggers the need to find sources of knowledge: people, artifacts, whatever.

In some startups there’s a policy that you have to deliver something into production by yourself, with some guidance, within the first two days on the job. It forces you to discover quickly the full process and all colleagues involved, if any. It’s a also mark of trust, you are trusted enough to be allowed to deliver something immediately, for real. It’s also a mark of confidence in the process, in particular its tests and deployment automation strategy. You not only learn the code, you also learn

that you can trust the delivery approach, and you also learn that the typical timeframe of a change is that long.

It's also a great way to get fresh feedback on the process. If the installation all the pre-requisite workstation setup takes two days or more, there's no way you can deliver something in two days. If someone has to help often during the local developer setup, then you need better documentation at the minimum, or preferably better automation of this process. The same goes for the full delivery pipeline, and any other matter.

If you have a weird in-house or proprietary stuff that new joiners have to learn, newcomers will tell you that there is a standard alternative that you could switch to.

## Astonishment Report

Astonishment Report is a simple yet effective tool to learn both about what should be documented and about what could be improved.

**Ask every newcomer to report all their surprises during their very first days on the job. Even if they come from the same company or from a similar background they may bring fresh perspectives. Suggest they keep a notebook to take notes immediately as they notice an astonishment, or they will forget most of them. It's paramount to preserve the candor, so keep the observation period short, like two days, or a week. Beware, two days may be long enough to get accustomed enough so that weird stuff is no longer that weird. Improve based on the remarks.

## Some Upfront Documentation?

Sometime an on-demand documentation approach can supplemented by some upfront documentation. The danger is to create speculative documentation that may never be useful. The benefit is that obviously essential knowledge is available to help people without waiting for the Rule of 2 to trigger.

Imagine yourself as a beginner on the project, knowing nothing. If you remember how it was like when you joined, it's simpler. Then create the ideal documentation that you would have loved to find.

In the words of Willow Brugh on Twitter:

> Be the adult you wish you had around when you were a child. Write the documentation you wish you had when you started on this project. @willowbl00

However there's this *Curse of Knowledge* that will make this approach mostly ineffective. You simply can't imagine any more how it's like not knowing something once you know it.

> The curse of knowledge is a cognitive bias that occurs when an individual, communicating with other individuals, unknowingly assumes that the others have the background to understand.
>
> Curse of knowledge on Wikipedia[a]
>
> [a] https://en.wikipedia.org/wiki/Curse_of_knowledge

It's extremely hard to guess in advance what information will be useful for other people we don't know yet, trying to do tasks we can't predict.

Still, there are some heuristics to help decide when a piece of knowledge should be documented right now:

- Everybody agrees it should be documented
- It's a hot topic (controversial)
- It's been discussed for a long time, e.g. during the sprint planning ceremony
- There has been a profound or shocking misunderstanding by some people involved
- It's important and there's no way it can be guessed or inferred from the code
- It should be refactored to avoid the need for documentation, but it's not practical to do now

Andy Schneider has really nice words on improving the documentation everyday, with a focus on empathy:

> **Make the value you've added permanent** – Andy Schneider
>
> - Comment code that you are working on so the next person doesn't have to go through the same pain [...]

This maxim does not tell you precisely when or when not to do something documentation-related. It's still up to your judgement. But it reminds the point that it's all about protecting *value for other people*.

## Knowledge Backlog

Other techniques to stimulate on-demand documentation is to define the content of the documentation with the help of a skills matrix or through a knowledge backlog.

Let each team member add what each piece of knowledge they'd like to have on sticky notes on a wall. Then have everyone decide by consensus or dot-voting what should be documented first. This could become your knowledge backlog. Every few weeks or every iteration you take one or two

item and decide how to address it: shall we pair-program? Shall I augment the code to make this structure visible in the code itself? Shall you document your specific knowledge of this area as an Evergreen document on the Wiki?

This session can be done within your retrospective.

However beware of backlogs when they grow. Please don't make it into an electronic tracker, stickers at the bottom of your whiteboard are enough, and the lack of room will remind keeping the backlog small.

## Skills matrix

An alternative is to create a skills matrix with predefined areas, and ask each team member to declare their level of proficiency for each area. One limitation is that the matrix will reflect the views of the person creating it, and will ignore the skills areas ignored or neglected by this person.

You could do use a skills matrix as a chart with many quadrants, as described by Jim Heidema in his blog post[50]

> This is a chart that can be posted in the room to identify the skills needed and the people on the team. On the left column you list all the team members. Along the top you list all the various skills you need on the team. Then each person reviews their row, looking at each skill, and then identifies how many quadrants of each circle they can fill in, based on the range below the chart. The range is from no skills through to teach all skills in a given column.
>
> 0: no skill 1: basic knowledge 2: perform basic tasks 3: perform all tasks (expert) 4: teach all tasks

Whenever the skills matrix reveals a lack of skills, it calls for planning a training or improving the documentation in some form.

[50] http://www.agileadvice.com/2013/06/18/agilemanagement/leaving-your-title-at-the-scrum-team-room-door-and-pick-up-new-skills/

# Declarative Automation

*Every time you automate, you should take the opportunity to make it a form of documentation.*

Software development increasingly makes use of automation in all its aspects. Over the last decades, popular tools have changed the way we work, replacing repetitive manual tasks with automated processes. Continuous Integration tools automate the building of the software from its source, and they automate the tests executions, even on remote target machines.

Tools like Maven, NuGet or Gradle automate the burden of retrieving all the required dependencies. Tools like Ansible, Chef or Puppet declare and automate the configuration of all the IT infrastructures.

There's something interesting in this trend: you have to describe what you want in order to automate it. You declare the process, then the tool interprets it to do it so that you don't have to. The good news is that when you declare the process, you actually document it, not just for the machine, but also for humans as you have to maintain it too.

**Therefore: Whenever you automate a process, take the opportunity to make it the primary form of documentation for this process. Favor tools with a declarative style of configuration over tools that rely on prescriptive style of scripts. Make sure the declarative configuration is meant primarily for human audience, not only for the tool.**

The goal is to have the declarative configuration as the single source of truth for the process. This is a great example of a documentation that is both a documentation for humans and a *Documentation for Machines*.

How did we do before all the new automation tools? In the worst case, the process was done manually by someone with tacit knowledge of how to do it. When he or she was away, there was no way we could do it at all. Manual process, tacit knowledge and no documentation at all.

When we were a little more lucky, there was a MS Word document describing the process in a mix of text and command lines. However the few times you tried to use it, you could hardly succeed without asking questions to the author: some parts were missing, and other were obsolete, with wrong indications. Manual process, but deceiving documentation this time.

When we were lucky, there was a script to automate the process. However when it was throwing errors you again had to ask the author for help to fix it, as the script code is quite obscure. And there was a separate MS Word document, rather incomplete and obsolete, pretending to describe the same process to please the management. Automated process, but still no useful documentation.

But now we know better, and the keyword to fix all that are *Declarative*, and *Automation*.

## Declarative style

For an artifact to be considered as a documentation itself, it must be expressive and easy to understand by people. it should also explain the intentions and the high-level decisions, not just the details of how to make it happen.

Imperative scripts that prescribe step by step what to do fail at that for any non-trivial automation. It's all about the 'how', and the interesting decisions and reflexions on too only have comments to be expressed.

On the other hand, declarative tools are more successful at supporting a nice documentation, thanks to two things:

- They already know how to do a lot of typical low-level things, which have been codified well once by dedicated developers into reusable ready-made modules. This is an abstraction layer.
- They offer a declarative domain-specific language on top, that is at the same time more concise and rather expressive. This DSL is standard and is itself well-documented, which makes it more accessible than your in-house scripting language. This DSL usually describes the desired state in a stateless and idempotent fashion; by moving the current state out of the picture, the explanations become really simpler.

## Automation

Automation is essential to force the declared knowledge to be honest.

With the modern approaches to automation, you tend to run the process very often, even continuously, like dozens of times per hour. This is a nice pressure to keep it reliable and always up-to-date. You have to be smart to reduce its maintenance. Automation you rely upon therefore acts like a **Reconciliation Mechanism** that makes it obvious when the declared process becomes wrong.

Together, all that is a (R)evolution. At last you can have knowledge that is up-to-date and that really explains what we want, the way you would talk about it. Tools are getting closer to the way we think, and that's changing the game in many aspects and in particular with respect to documentation.

In the following sections, we will have a look at various examples of declarative automation for software projects.

# Declarative Dependency Management

In the build automation landscape, *dependency managers*, aka *package managers*, are tools that play a key role as part of the build process. They reliably download libraries, including their transitive dependencies, resolve a large part of the conflicts, and support your dependency management strategy even across many modules.

Before that automation, dependency management was a chore done manually. You would manually download the libraries in some version into a /lib folder, later stored into the source control system. If the dependency has dependencies, you had to look at their website and download them all too. And you had to redo all that whenever you had to switch to a new version of a dependency. It was not fun.

Popular dependency managers are available for most programming languages: Maven and Apache Ivy (Java), Gradle (Groovy and JVM), NuGet (.Net), RubyGems (Ruby), sbt (Scala), npm (Node.js), leiningen (Clojure), Bower (web), and many other.

To do their job of automating, these tools need you to declare all the direct dependencies you expect. You usually do that in a simple text file often called a *manifest*. This manifest is the Bill of Materials that dictates what to retrieve in order to build your application.

When using Maven, the declaration is done in an XML manifest called pom.xml

```
<dependency>
   <groupId>com.google.guava</groupId>
   <artifactId>guava</artifactId>
   <version>18.0</version>
</dependency>
```

In leiningen, the declaration is of course done in Clojure:

```
[com.google.guava/guava "19.0-rc1"]
```

Whatever the syntax, the declaration of the expected dependencies always consists in a tuple of the three values: group-id, artifact-id, requested version.

In some of the tools the requested version can be not only a version number like *18.0*, but also a range *[15.0,18.0)* (meaning: from version 15.0 to version 18.0 excluded), or a special keyword like *LATEST*, *RELEASE*, *SNAPHOT*, *ALPHA*, *BETA*. We can see with these concepts of range and keywords that the tools have learnt to work at the same level of abstraction we think at as developers. The syntax to express the necessary dependencies is declarative, and it's a good thing.

As we're in a case of Declarative Automation, the declaration of the requested dependencies is also the single source of truth for the documentation of the dependencies. The knowledge is already there, in your dependency manifest.

As a consequence, there is no need to list these dependencies again in another document or in a Wiki, or it would just mean taking the risk to forget to update it and it would then be misleading.

But as usual, there's one thing missing so far in the declaration of the dependencies: we'd like to declare not just what we request to the tool, but also the corresponding rationale. We need to record the rationale so that future newcomers can quickly grasp the reason behind each dependency included. Adding one more dependency should never be done too easily, so it's good to always be able to justify them with a convincing reason.

One way to do that is just with comments next to each dependency entry in the file:

```
<dependencies>
  <!-- Rationale: A very lightweight alternative to JDBC, with no magic -->
  <dependency>
          <groupId>org.jdbi</groupId>
          <artifactId>jdbi</artifactId>
          <version>2.63</version>
  </dependency>
<dependencies/>
```

We could be tempted to add a description, but we don't even have to since it's already included in the pom of the dependency itself. In IDE like Eclipse it's very easy to navigate to the pom of the dependency by pressing Ctrl (or Cmd when using Mac OS X). As your mouse hovers over the dependency element in your pom, it turns into a link to directly jump to the pom of the dependency.

```
<dependencies>
    <dependency>
        <groupId>com.google.guava</groupId>
        <artifactId>guava</artifactId>
        <version>RELEASE</version>
    </dependency>
    <dependency>
```

**Navigating the Maven dependencies in Eclipse POM Editor**

That's integrated documentation mixed with declarative automation. Pure awesomemess!

Is that knowledge accessible? It depends on the audience. For developers, looking at the manifest and using the IDE is the most accessible way, hence there's no need to do anything more. One issue may be that when using ranges or keyword for the versions you don't know the exact version being retrieved at a given point in time just by looking at the manifest. However developers know how to query the dependency manager to have this information on-demand.

For example in Maven they would run: *mvn dependency:tree -Dverbose*

For non developers you would want to extract and publish the interesting content into an Excel document or to the Wiki. But are non-developers really that interested in that kind of knowledge?

# Declarative Configuration Management

Configuration Management is much more complex than dependency management. It's about resources like applications, daemons, files etc. each with many attributes and with all their dependencies. However some tools have taken a declarative approach similar to the dependency managers and their manifests.

The most popular tools to manage configuration are *Ansible*, *Puppet*, *CfEngine*, *Chef* and *Salt*. However among them some are imperative (Chef) while others are declarative (Puppet, Ansible).

**LOL**

> Sorry this is taking so long, I lost my bash history and therefore have no idea how we fixed this last time. From @honest status page[a] on Twitter
>
> [a] https://twitter.com/honest_update

For example, Ansible states:

> Ansible's philosophy is that playbooks (whether for server provisioning, server orchestration or application deployment) should be declarative. This means that writing a playbook does not require any knowledge of the current state of the server, only its desirable state.

Puppet have a similar philosophy. Here's an excerpt of a Puppet manifest for managing NTP:

```
# Some comment if necessary...
service { 'ntp':
  name      => $service_name,
  ensure    => running,
  enable    => true,
  subscribe => File['ntp.conf'],
}

file { 'ntp.conf':
  path    => '/etc/ntp.conf',
  ensure  => file,
```

```
  require => Package['ntp'],
  source  => "puppet:///modules/ntp/ntp.conf",
}
```

Puppetlabs emphasize that Puppet manifests are self-documented and proof of compliance even for many regulatory bodies:

> **Self-documentation**
>
> Puppet manifests are so simple, anyone can read and understand them, including people outside your IT and engineering departments.
>
> **Auditability**
>
> Whether it's an external or internal audit, it's great to have proof that you pass. And you can easily validate to your own executives that compliance requirements have been met. Puppetlabs Blog[51]

A declarative language like used in these tools allows to communicate the expected desired state, not only to the tool, but to the other humans on your team or even to external auditors.

Again, what's often missing to make these manifests a complete and useful documentation for humans is the rationale for each decision. This can be done with comments, If we consider that a Puppet manifest as-is is accessible to all the interested audience, then it would make sense to document the rationales and other high-level informations into the manifest, for example as comments.

Because the knowledge about the configuration is declared in a formal way for the tools, it also becomes possible to generate Living Diagram when it can help reasoning. For example Puppet includes a *graph* option that generates a .dot file of a diagram showing all the dependencies. This is useful when you experience an issue in the dependencies or if you want to have a more visual view of what's in the manifests.

Here's an example of a generated diagram from Puppet:

[51] https://puppetlabs.com/blog/puppets-declarative-language-modeling-instead-of-scripting

**A diagram of the resources dependencies, generated by Puppet (John Arundel)**

This kind of diagram can also be handy to refactor the manifests to make them cleaner, simpler and more modular. As John Arundel write in his blog[52]:

> As you develop Puppet manifests, from time to time you need to refactor them to make them cleaner, simpler, smaller and more modular, and looking at a diagram can be very helpful with this process. For one thing, it can help make it clear that some refactoring is needed.

# Declarative Automated Deployment

Similar to configuration management, there are a number of tools that can automate your deployment, including the necessary company workflows, rollback procedures, and deploying only what needs to be changed. Some of these tools include Jenkins with custom or standard plugins, Octopus Deploy (.Net).

Here's an example of a deployment workflow from the Octopus website:

- Redirect load balancer to a "down for maintenance" site
- Remove web servers from load balancer

[52] http://bitfieldconsulting.com/puppet-dependency-graphs

- Stop application servers
- Backup and upgrade the database
- Start application servers
- Add web servers back to load balancer

In a tool like this, the deployment and release workflow is typically setup by clicking on its UI, and persisted in a database behind. Still, the workflow is described in a declarative manner, that everyone can understand when looking at the tool screens. Whenever you want to know how it's done, you just have to look it up in the tool.

Because it's declarative and because the tool knows about the basics of deployment, we can describe complex workflows in a concise way, closer to the way we think about it. For example, we can apply standard patterns of Continuous Delivery like Canary Releases and Blue-Green Deployment. Octopus Deploy manages that with a concept they call *Lifecycle*, an abstraction useful to easily take care of this kind of strategies.

Thanks to tools like that, not only they automate the work itself and reduce the likelihood of errors, but they also provide a ready-made documentation for the standard patterns you could, or should, be using. This means this is another documentaton you don't have to write by yourself!

Imagine that you decide to adopt Blue-Green deployment for your application. You can configure the tool to take care of it, and here is all you have to do now:

- Declare in a stable document like the README file that you have decided to do Blue-Green Deployments
- Link to an authoritative literature on the topic, e.g. the pattern on Martin Fowler website[53]
- Configure the tool and the lifecycle to support the pattern, and
- Link to the page on the tool website[54] that describes how the pattern is taken care of specifically in the tool.

By the way, below is that description of the pattern in the context of the tool:

> Staging: when blue is active, green becomes the staging environment for the next deployment
>
> Rollback: we deploy to blue and make it active. Then a problem is discovered. Since green still runs the old code, we can roll back easily
>
> Disaster recovery: after deploying to blue and we're satisfied that it is stable, we can deploy the new release to green too. This gives us a standby environment ready in case of disaster

[53] http://martinfowler.com/bliki/BlueGreenDeployment.html
[54] http://docs.octopusdeploy.com/display/OD/Blue-green+deployments

For an automation to be a case of declarative automation that provides documentation, the important thing is that the configuration of the tool has to be genuinely declarative, be it in text or on a screen and a database. It also to be at an abstraction level close to what matters for everyone involved. In particular, it cannot be obscure imperative steps with a lot of conditionals based on low-levels details like the absence of a file or the state of an OS process.

## Scaffolding Over Step by Step guide

Whenever you join a new team or a new project, you need to setup your work environment, and you need some documentation for that. At least that's how many company still do. There's a Newcomers page on the wiki with a long list of steps to go through in order to start working on the application.

I've seen many times that this list is never totally up-to-date. Links are broken. Essential information is missing because it was obvious in the mind of the author. And these issues exist even when newcomers join regularly.

Some teams have taken a step further, by providing an installer for newcomers. You run the installer, it prompts for some specific questions, and you're done! They don't always work quite well when they're custom in-house scripts, but the idea is there: why document in text what could be automated, and documented, as a tool?

This approach is often called Scaffolding, and is not just for newcomers, but also to start an application quickly. Ruby On Rails is probably the most popular tool with this approach.

Many tools can be used to do scaffolding. You can do scaffolding with custom scripts, Maven archetypes, Spring Roo, JHipster and many others. Configuration management tools may sometimes also be used to setup a working setup for new team members, or to setup templates of applications that can be modified later. Standard tools

If the resulting automation is rock-solid, documentation of what it does is less of an issue, but in general we would favor standard tools over in-house scripts, and tools that are themselves well-documented, maintained and with a declarative configuration that can be considered itself as the documentation.

The scaffolding has to be really easy to use without a user guide. - asking simple questions, step by step - with sensible default values - with very good example of answers

Here is an example of an Open-Source tool for scaffolding called JHipster[55]. It works with a command-line wizard, and here are some of the questions prompted when creating a new application from scratch:

- What is the base name of your application?
- Do you want to use Java 8?
- Which *type* of authentication would you like to use?

[55] http://drissamri.be/blog/technology/starting-modern-java-project-with-jhipster/

- Which *type* of database would you like to use?
- Which *production* database would you like to use?
- Do you want to use Hibernate 2nd level cache?
- Do you want to use clustered HTTP sessions?
- Do you want to use WebSockets?
- Would you like to use Maven or Gradle?
- Would you like to use Grunt or Gulp.js for building the frontend?
- Would you like to use the Compass CSS Authoring Framework?
- Would you like to enable translation support with Angular Translate?

For each question, there is a clear narrative explaining the possible answers and the consequences to help make the decision. This is an inline, tailored help. The resulting code is the consequence of all the decisions. If you've chosen MySQL as the database, then you have a MySQL database setup.

It would be interesting to record the responses to all the questions of the wizard into a file (they're only kept as logs or in the console) to provide a high-level technical overview of the application. It may be included into the README file for example.

A particular, degenerate, example of a wizard is to design helpful exceptions that precisely tell you what how and where to fix the problem when thrown.

# Machines Documentation

Before the Cloud, we had to know our machines one by one, so there was often an Excel spreadsheet somewhere with a list of machines and their main attributes. And it was often obsolete too.

Now that the machines are moving somewhere in the cloud, we can no longer affort to do that, as it changes much too frequently, sometime many times a day. But since the Cloud itself is automated, very accurate documentation now comes for free, through the Cloud API.

This is very similar to declarative automation. You declare what you want: "I want a Linux server with Apache", and then you can query your current inventory of machines available and all their attributes. Many of these attributes are tags and meta-data that add a higher level of information to the picture: it's not just a "2.6GHz Intel Xeon E5", but it's a "High-CPU machine.", in the

# Remarks on automation in general

People are good for novel stuff. Machines are good for repeated stuff.

Automation provides benefits at a cost. It is not an end in itself, but a mean to save time and to improve reliability on repeated tasks. But there is always a point where the cost exceeds the benefits. Invest in automation as long as the cost is low compared to the recurring benefits. Decide what to keep manual.

> Don’t do the same thing twice. If it’s déjà-vu, then it’s time to automate. – Woody Zuill in a conversation

On the other hand, if a task is new or different each time, wait until you see enough repetition somewhere in the task before thinking about automation.

# Enforced Guidelines

*The best documentation does not even have to be read, if it can alert you at the right time with the right piece of knowledge*

Making information available is not enough. Nobody can read and remember all the possible knowledge ahead of time. And there is a lot of knowledge that you'd need without having any way to figure out that you need it.

> You don't even know that you don't know something that you should know.

Consider the code guidelines.

Many companies and teams spend time writing guidelines. Still, the resulting documents are seldom read and often ignored.

How do you document all the decisions that have been made and that everybody should conform to when doing their work? Examples of these decisions include the main architectural decisions, coding guidelines and other decisions about style and team preferences.

The common approach is to spend time to write these decisions into a guidelines, style book or any other name. The problem is that theses decisions quickly add up to more pages than you expected, and even a 12 pages-long document full of "you shall do this" and "you shall not do that" is far from an exciting read. As a consequence, most of these documents are like legal documents: they are so boring most team members never read them, not even once. They pretend to have read them on arrival, but in fact they hardly went further than the second or third page.

Even when they've actually read them, the format as a list of rules is not memorable, and unless you like all the rules you probably won't remember most of them. In practice these guides are essentially useful as a reference in case of doubt, and not much more.

However without guidelines the code is at the mercy of everybody's own style, preferences or lack of skills. A consistent set of shared guidelines is essential to really work in a collective ownership fashion.

So how do people learn about all the decisions and style they have to conform to for real? They learn all that essentially by reading other people's code, through code reviews, and through the feedback from static analysis tools who catch when they violate a rule.

Reading code works well when the code is exemplary, which is not always the case. Of course code review and static analysis help improve that. Code review works well as long as the reviewers have all the decisions and style preferences in mind and all agree with it. Static analysis works well for

every rule or decision that doesn't need nuance or contextual interpretation. And because the static analysis tools must be configured to be useful, once configured they are themselves naturally the reference documentation about the all the guidelines.

**Therefore: Use a mechanism to enforce the decisions that have been made into guidelines. Use tools to detect violation and provide instant feedback on the violation as visible alerts. Don't waste time writing guideline documents that nobody reads. Instead, make the enforcement mechanism self-descriptive enough so that it can be used as the reference documentation of the guidelines.**

Code analysis tools help maintain a high level of quality anywhere in the code, which in turn helps the code to be exemplary. And it also helps as a reference when the programmers have an hesitation about one rule during a code review or while pair-programming, a form on continuous code review.

The point of enforced guidelines is to accept that documentation does not even have to be read to be useful. The best documentation brings you the right piece of knowledge at the right time, i.e. when you need it. Enforcing rules, properties, decisions through tools (or code reviews) is a way to teach the team members with the knowledge they need precisely at the moment they ignore it.

Enforced Guidelines: persistent knowledge made interactive again

## Some examples of rules

Cosmetic rules help for code consistency and when merging code

- Curly brackets must not be omitted
- Field names must not use hungarian notation

Rules on metrics help discourage overly-complicated code

- AvoidDeepInheritanceTreeRule (max = 5)
- AvoidComplexMethodsRule (max = 13)
- Line should not be too long (max = 120 chars)

Rules are a way to encourage or enforce better code

- DoNotDestroyStackTraceRule
- Exceptions should be public

Some rules can directly avoid bugs

- ImplementEqualsAndGetHashCodeInPairRule
- Test for NaN correctly

Even some architectural decisions can be made as rules

- DomainModelElementsMustNotDependOnInfrastructure
- ValueObjectMustNotDependOnServices

**The Guidelines Enforcement force**

## Evolving the guidelines

Guidelines have a purpose, for example to help working together, reduce issues like bugs or errors when merging code, and preserve quality attributes like performance and maintainability. With that in mind, there's no such a thing as the ideal and definitive set of guidelines. Instead, start with something, use it and evolve it to make it as relevant as possible.

The best guidelines don't come from above. The best guidelines grow from the team or teams doing their work and talking to each other to agree on shared guidelines that are useful. Don't hesitate to change the guidelines when necessary. Of course you may not want to change the length of lines of code everyday.

### Sample guidelines for a greenfield project

- Unit Test Coverage >80%
- Complexity by method <5
- LOC by method < 25
- Inheritance depth < 5
- Args number < 5
- Member data fields number < 5
- All base checkstyle rules

## Enforcement or Encouragement

On a greenfield project, you typically start with a lot of enforced guidelines in a strict fashion, and every new line of code that violates them will have its commit rejected.

Other the other hand, on a legacy project you usually can't do that because the existing code would already contain thousands of violations even on a small module. Instead you chose to only enforce the few most important guidelines, and you put every other guidelines as *warnings*.

Another approach is to have stricter rules only for new lines of code.

Some teams start with some guidelines, and once they are comfortable with them they add more rules and make the existing guidelines stricter in order to progress.

When your company requires every application to follow a minimum set of guidelines, each team or application can still decide to make it stricter, but no weaker. Tools like Sonar provide inheritance between sets of guidelines, called 'quality profiles', to help do that. You can define a profile that extends the company profile, and add more rules or make the existing rules stricter to meet your own taste.

## Declarative Guidelines

Because sets of guidelines, 'quality profiles', can be named, their names are also part of the documentation on guidelines. You can simply refer new joiners to the build configuration, where they will find the name of the set of guidelines. From there they can look it up on the tool and find out that it extends the company sets of guidelines. They can browse the rules by categories, severity, check their parameters as they wish, in an interactive fashion. There's even a search engine.

Each given rule has a key, a title and a brief description of what it is and why. With the key or the title you can look up its more complete documentation on the tool or directly on the web.

For example if we lookup *ImplementEqualsAndGetHashCodeInPairRule* on the web, we immediately find its reference documentation, from the Gendarme plugin for .Net:

> This rule checks for types that either override the Equals(object) method without overriding GetHashCode() or override GetHashCode without overriding Equals. In order to work correctly types should always override these together.

This reference documentation usually include several code samples, a bad example, and a good example to illustrate the point of the rule.

This is great, because that documentation that's already there. Why write that again when it has been already done well by someone else?

# A matter of tools

Compilers, Code coverage, static code analysis tools, bug detectors, duplication detectors, dependency checkers are common examples of ways to setup Enforced Guidelines in practice.

Sonar is a popular tool that itself rely on many plugins to actually do its job. When the configuration of the tools is often not meant to be a documentation, with verbose XML and rules identifiers, tools like Sonar (see SonarQube) can make the configurations of coding rules more accessible in a convenient UI, to the point of becoming the reference about guidelines.

Even when the plugins are actually configured via an XML file, Sonar displays the list of coding rules nicely on screen, and you can modify them there, along with the reference description in prose. This can also be exported in a spreadsheet format. If you really want to spend time documenting coding guidelines manually, just tell the overall intentions, priorities and preferences, and let the tools tell the details!

Other guidelines may be enforced by access control. You decided that this legacy component is frozen from now on, nobody has the right to commit on it? Simply revoke the write grants to everyone. But this in itself does not explain why. So expect questions, and the knowledge transfer will happen as a conversation.

Most automated means are not 100% relevant at any time, so sometime the enforcement will be violated anyway. This is not necessarily a disaster, as long as the enforcement maintains enough continuous awareness about the guidelines.

If an element of the guideline is not enforceable, then perhaps it is not really an element of a guideline; otherwise you can add it to a short checklist for manual code review or during pair-programming. But this is not **Enforced Guidelines** any longer.

However if you have new rules, you may consider extending the existing tools with a new rule or new plugin. Compilers often have extension points where you can hook your own additional rules. Tools like Sonar are extensible with custom plugins, and checkers are extensible with new rules, sometime by XML, sometime only with code.

## Guidelines, or Design Documentation?

Imagine your set of guidelines for the Domain Model is like the following:

- functional-first (immutable & side-effect-free by default)
- null-free
- no framework pollution
- no SQL
- no direct use of a logging framework
- no import of any infrastructure technology

At the time of writing, existing static analysis tools and plugins likely don't support all that out of the box, so you can't do Enforced Guidelines unless you create your own tooling. However, these guidelines are design decisions that can be documented in the code itself, for example using annotations as seen in a previous chapter.

In fact, such design declarations expressed as annotations in turn make it possible to enforce them with analysis tools. Once you declare that your code should be all immutable in a given package, it becomes possible to check the main violations using a parser. (see Patternity project on Github)

Immutability and null-free expectations can to some extent be enforced programmatically. This is far from perfect, but this is enough for any new joiner to learn the style after a few commits.

## Warranty Sticker Void If Tampered

*Knowledge that you can't avoid, right when you need time*

Hamcrest[56] is a popular Open Source project providing matchers to write beautiful unit test. It provides a lot of matchers out of the box, but you can also extend it with your own custom matchers. Usually when you do that you should read the Developer's Guide, but not everyone does that.

That's why Hamcrest uses naming in a creative way to make it very unlikely to break a design decision by ignorance:

[56] http://hamcrest.org

```
/**
* This method simply acts a friendly reminder not to implement Matcher directly \
and
* instead extend BaseMatcher. It's easy to ignore JavaDoc, but a bit harder to i\
gnore
* compile errors .
*
* @see Matcher for reasons why.
* @see BaseMatcher
* @deprecated to make
*/
@Deprecated void _dont_implement_Matcher___instead_extend_BaseMatcher_();
```

Hamcrest Matcher method `don't implement Matcher ___ instead extend the Base Matcher` is an impossible to miss useless documentation method. You can still break it deliberately, but the point is that you're aware of doing that. That's a kind of warranty sticker "void if tampered". This is an original way to do Unavoidable Documentation.

The funny things is that in this example of Enforced Guidelines, the enforcement is done by the potential violator him/herself.

Some more similar examples:

- Documentation by Exception: You decide turn a legacy component from `READ-WRITE` to `READ-only`. You can document that with text or annotations, but how to make sure nobody will add `WRITE` behavior? One way is to keep the WRITE methods on all the Data Access Objects but have them throw exceptions: `IllegalAccessException("The component is now READ-ONLY")`
- You create a module that nobody should import except one particular project, and you have no way to do that within the package manager itself. You can implement a very simple `license` mechanism: When you import the module it throws exceptions complaining it's missing a license text file or license `ENV` variable. The license can be the verbatim text: "I should not import this module" acting as a disclaimer. You can hack it, but this means you accept the disclaimer!

# Trust-First

Enforcing guidelines as automated rules or through access restrictions may express a lack of trust to the teams. This actually depends a lot on your company culture. If your culture really is a culture of trust, autonomy and responsibility between everyone then introducing Enforced Guidelines should be decided by consensus after discussions between everyone involved. In the worst case it could send the wrong signal and undermine trust, which would be a greater loss than the benefits you're after.

# Shameful Documentation

> Just because it is documented, it doesn't make it less stupid. – Dalija Prasnikar on Twitter

Documentation where its existence is in itself revealing issues that should be fixed rather than documented

Documentation, when up-to-date and accurate, is often considered a good thing. However there are a number of cases where this is quite the opposite: the existence of the documentation in itself demonstrates the presence of a problem. The infamous Troubleshooting Guide is probably the best example in this category. Someone decided to take the time to document the known troubles, usage traps and other anomalies of behavior, and this effort demonstrates that the issues are important enough to be worth documenting. However, this also means that these issues are not fixed, perhaps not even planed to be fixed.

This is a kind of shameful documentation, documentation you should be ashamed of. This documentation, by its sole existence, should be seen as a confession of something to be fixed. Going further, the time spent creating the documentation should have been allocated fixing the troubles instead.

**Therefore: Recognize the situations when documentation is a poor substitute for actually fixing a problem. Whenever possible, decide against adding more documentation and allocate time to fix the problem instead.**

Of course there are many reasons for teams to add documentation instead of fixing the issues:

- Budget: there is money allocated for documentation but no more money for working on the code
- Laziness: it may seem easier to add a quick documentation on troubleshooting rather than actually tackling the root issue
- Lack of time: documenting the issue is faster than fixing it
- Cost: it may be genuinely difficult to address some issues. For example somes issues would require releasing a new version of the application to dozens of clients, which makes it prohibitively expensive.
- Missing knowledge: sometime the team knows about the issues but missing knowledge and skills on where and how to fix the issues.

If there is no time available to fix now, then the right place to document the issue is the defect tracker. However in the mindset of Shameful Documentation, a defect tracker is also in itself a demonstration of a deeper issue: defects should not accumulate, they should be prevented earlier

or fixed immediately as much as possible. And are defects that can remain for a long time without being fixed really defects?

If a feature is implemented so badly that it requires a manual with many pages of warnings and workaround instructions, or a lot of assistance from the support team, you may consider removing it until it is implemented correctly; chances are that almost nobody manages to use it anyway, or that using it is so expensive that it's not worth it.

## An example

In a past mission at a customer, I discovered a 16-pages-documentation on How to run and test a application. This is a guide for the all users, including end -user. We'll call this application Icare to protect the innocents. This is not a new project, it's used several times every day by dozens of people in the company. This document is full of screenshots highlighted in red color bubbles to show how to proceed, which is not so unintuitive. However most of the 16 pages describe where to "pay attention".

> "Pay attention…[this may not work properly] Please note that…[there is a bug here] etc."

This document is indeed full of warnings! "Pay attention, Icare is launched from another directory!". "Take really good care to not launch these tasks anytime because it will kill everything on the corresponding environment!".

Pay attention, we're not professional.

Almost half of everything written is about a trap waiting to bite you. "Pay attention to the name of the trigger, sometimes, it's not correctly named, so check in the trigger". Remember this is a document for the end users.

And it gets even better: "After an export in XML, you should make a test of re-import to be sure that it works well". Okay, we can see that a developer had the time to write this document instead of fixing the code.

And again: "Pay attention, partitions Icare_env1 and Icare_env2 are inversed between UAT and PROD!!!" Ah this time you mean everyone knows and it's been like that for years, but it's not in anyone plan to fix that? Or maybe the process is so heavyweight you'd first have to find a sponsor to pay for the fix first.

## The Troubleshooting Guide

And finally you see that at the end of the documentation, the infamous "known problems" section:

```
1 Known problems

1.1 Icare Job does not start

It often happens. First of all, try to launch ift directly from Icare (so launch\
 the application manually from the correct directory [c:/icare/uat1/bin for UAT,\
 c:/icare/prod/bin for PROD]).


If you are not able to launch it manually, it's because configuration of the job\
 is not correct (missing or incorrect parameter date or calculation date, etc.).\
 If it runs well, there is a problem when launching Icare in command line, so yo\
u need to check the log (to find where it logs, check the icarius_mngt.exe.log4n\
et).

In the past, there was also a problem for the first execution. It requires to ha\
ve made a manual connection to the environment with the good login (IcariusId). \
When a first connection was established, the batch mode was correctly working.
```

By the way notice the unconsistent naming of the application Icarius or Icare.

Shameful documentation does not always means bugs, it may instead suggest opportunities for a better Ops-friendliness:

```
"you have to check the caches are up otherwise they will hit the DB and degrade \
performance results"
[...]
"Very important : As we are not able to guarantee the synchronization of the two\
 environments for the duration of jobs, we cannot launch different type of jobs"\
.
```

Once you begin to listen carefully to the documentation, it becomes a source of suggestions. What about a way to automatically monitor the caches, or even better, a mechanism to ensure they are always preloaded before operations?

What about adding a safety mechanism so that if you do the error you're warned and you avoid the issue?

## Don't tolerate the documented pain

Don't tolerate the documented pain. Say no. Writing the documentation is wasting time, and reading it is wasted time too, and it will not even prevent anyone from falling into the trap completely, which will waste time again and again.

The Troubleshooting Guide is not the only example of Shameful Documentation. Any document that is getting too big becomes a case of a Shameful Documentation in itself. A Developer Guide with 100 pages, or a thick user manual reveal issues of code quality and user-friendlyness respectively. You need a big user guide when the application is not intuitive to use, but addressing the real issue instead would probably be a better investment if you care about the users.

## Shameful code documentation

The same applies for software design. If it takes a lot of pages and many diagrams to explain the design of an application, then it also demonstrates that the design is poor.

*See the dedicated chapter for more about that.*

Finally, Shameful Documentation also applies to code. Every time the developer feel the urge to add a comment like:

```
// beware this tricky case
...
// should never happen
...
// FIXME: remove this hack!
```

it should instantly trigger a reaction to remove the comment and immediately fix the questionable code instead.

There are many other examples of code that suggests improvements. They are discussed later.

# Don’t document, influence or constraint behavior instead!

## Make It Easy to Do the Right Thing

Enforcing guidelines is not the only approach to bring the right piece of knowledge at the right time to the developers; an interesting alternative is to make it easy to do the right thing in the first place.

For example, you could decide that “from now on, developers MUST create more modular code, as new small services that MUST be deployed individually”. You could print that on the guidelines document, hope everyone will read it and follow this decision.

Or you could invest into changing the environment:

- **Providing good self-service CI/CD tools**: By making it easy to setup a new build and deployment pipeline, you make it more likely that developers will create new separate modules rather than putting all new code into the same big ball of mud that we know how to build and deploy.
- **Providing a good Micro-Services Chassis** (from Chris Richardson website[57]) encourage modularity by making it easy to bootstrap a new micro-service without spending time wiring together all the necessary libraries and frameworks.

In his book “Building Microservice”, Sam Newman write on making it easy to do the right thing, with what he calls a *Tailored Service Templates*:

> Wouldn’t it be great if you could make it really easy for all developers to follow most of the guidelines you have with very little work? What if, out of the box, the developers had most of the code in place to implement the core attributes that each service needs?
>
> (…)
>
> For example, you might want to mandate the use of circuit breakers. In that case, you might integrate a circuit breaker library like Hystrix. Or you might have a practice that all your metrics need to be sent to a central Graphite server, so perhaps pull in an open source library like Dropwizard’s Metrics and configure it so that, out of the box, response times and error rates are pushed automatically to a known location.

[57] http://microservices.io/patterns/microservice-chassis.html

The most famous tech companies embrace this approach with open-source libraries that you too can use. In the words of Sam Newman:

> Netflix, for example, is especially concerned with aspects like fault tolerance, to ensure that the outage of one part of its system cannot take everything down. To handle this, a large amount of work has been done to ensure that there are client libraries on the JVM to provide teams with the tools they need to keep their services well behaved.

The environment is also passing information. It's implicit, passive and we don't often pay attention to that. You can make it deliberate and decide what message to pass by designing the path of least resistance in the environment to be the one that you favor.

More generally, it's about making behavior not just *easier* but also more *rewarding*. By showing the commit history as a nice pixel art diagram, Github makes it rewarding to commit often. Developers pride is powerful!

*A major point of Living Documentation in general as advocated in this book is to offer simple ways to document, to encourage doing it more.*

# Making Mistakes Impossible - Error-Proof API

*Design your API in a way that makes it impossible to misuse. This reduces the need for documentation, since there's nothing to warn the user of.*

Michael L Perry listed many common API traps in a blog post[58]:

- You must set a property before calling this method
- You must check a condition before calling this method
- You must call this method after setting properties
- You cannot change this property after calling a method
- This step must occur before that step

These traps should not be documented; instead they should be refactored to be removed! *Otherwise the documentation will be a great case of Shameful Comment.*

There are endless ways to make an API impossible to misuse:

- Using types to only expose methods you can actually call, in any order
- Using enums to enumerate every valid possible choice
- Detecting invalid properties as early as possible (e.g. catching invalid inputs directly in the constructor) well before it is actually used then repair whenever possible, such as replacing nulls with null objects in the constructors or setters

[58] http://qedcode.com/practice/provable-apis.html

- It’s not just about errors, but also about any harmful naive usage. For example if a class is likely to be used as the key in a hashmap, it should not make the hashmap slow or inconsistent. You could use internal caches to memoize the results of any slow computations of hashcode() and toString().

A common objection is that experienced developers don’t get caught hence no need to be so defensive; however even good developers have more important things to focus on that avoiding the traps of your API.

In the wording of Don Norman, these advises on how to guide the use of something would all be called “affordances”, from his famous book “The Psychology of Everyday Things” (http://www.jnd.org/dn.mss/affor and.html)

# Design principles for documentation avoidance

During QCon 2015, Dan North talked about a model where code is either so old and well established that everybody knows how to deal with it, or that's so young that the people doing it are still there so they know well about it. Problems happen when you're in in the grey zone between these two modes.

This thinking emphasizes the central role of knowledge sharing and knowledge preservation as a key ingredient of successful teams. But Dan also goes further and suggests alternative ways to deal with this issue.

## Replaceability-First

You don't need much documentation for components you can replace easily. Sure, you need to know *what* the components were doing, but you don't have to know *how* they were doing it.

In this mindset you could give up maintenance. If you have to change something, you could just rebuild it all. For this approach to work every part has to be reasonably small, and as independent as possible from every other component. This shifts the attention on the contracts that are between components.

**Therefore: Favor a design that makes it easy to replace a part within the whole. Make sure that everybody knows exactly what the part does. Otherwise, you need documentation for the behavior, for example the working software you can easily play with, self-documented contracts of the inputs and outputs, or automated and readable test.**

When the team does not care enough about design, the components just grow and get hairy. They quickly get coupled to everything. As a result you can never really replace them totally. Making the code easy to replace is still an act of design, it does not happen out of pure luck or without skills and care. It takes discipline. One obvious way is to limit the size of the component, for example up to one page on the screen. Another way is to decide strict restrictions on what components can call each other and how they should not share data storage. For more on all these ideas please check books on micro-services.

Even with an approach that favors Replaceability, design skills remain necessary. For example the Open-Close Principle is indeed a case of making the implementation replaceable easily, along with it's good friend the Liskov Substitution Principle. The other SOLID principles also help. They are usually discussed at the class and interface level. Yet, they also apply at the bigger granularity of components or services. But to be really replaceable at low cost they have to be small, hence the idea of "micro"-services.

# Consistency-First

Consistency in the code base is when code that you've never seen looks familiar so that you can deal with it easily (Dan North).

In practice consistency is hard to maintain beyond bounded areas: consistency is more natural within one component, within one programming language, and even within one layer. You often don't follow the same programming style for GUI than for server-side domain logic.

For a given area of the code base with a consistent style of code, once you know the style there's nothing more to say for all elements in the area. Consistency makes everything standard. Once you know the standard there is nothing else to tell.

This all depends on the surrounding culture: for example, in a JEE-heavy company, there is no need to tell why you decided to use EJB, but you'd need to explain when you decide not to use them. In another company with better taste, that would be the opposite.

If you decide as a team that no method is allowed to return null within your domain model, then this decision only has to be documented in one place, for example in the root folder of the domain model source control. Then there's no need to talk about that any more on each method.

**Therefore: Agree as a team on concrete guidelines to apply within chosen bounded areas. Document them briefly in one place.**

There has to be exceptions to the rule. Not every class will be consistent. However as long as the number of exceptions is low, it's still cheaper to document the derogations explicitly than to document everything on every class.

Here's an example of the guidelines that a team decided for a Domain Model:

- No abbreviation in naming of public signatures
- Business-readable names in all public interfaces and their methods
- Null-free: no null allowed as return type or as method parameter
- All classes immutable by default
- All methods side-effect-free by default
- No SQL
- No import of frameworks at all, including javax.*
- No import of infrastructure (middleware etc.)

**Enforced Guidelines** are a way to document the guidelines in a way that is effective even if nobody reads them.

# Zero documentation & Gamification

**An approach to force better naming and better practice in general to share knowledge without additional prose**

I've heard of a team who decided to forbid documentation: they're proudly doing Zero Documentation. And it isn't that stupid.

Once you understand that most of the time written documentation prose or diagram is a poor substitute for expressing the knowledge better within the work product itself in the first place, it makes sense to minimize it. And because it sounds radical and a bit insane, it's stimulating and becomes a game. This makes it more likely to stick in team members mind, driving their behavior for the better, hopefully.

I haven't tried it myself but what my colleague told me about it is that it's usually driving virtuous behavior in practice.

Because we don't all share the same definition of the word "documentation", a game of Zero Documentation must clarify its rules. The above-mentioned team refuses comments in the code and on methods, all forms of written prose, external documents and traditional office documents. They happily embrace tests, Gherkin scenarios (Cucumber / Specflow), favor simple code, and enjoy working collectively as their primary mean of sharing knowledge. They're happy with all this.

I think augmenting the code with annotations, a README file kept simple and generating living documents still fits within the rules of the game. You decide when to put the cursor!

## Continuous Training

As general knowledge becomes more widespread, the less you need to document in average. Investing in continuous training is therefore a way to lower the need for documentation.

Learning standard skills also makes it more likely to use more ready-made knowledge at the expense of original solutions. This is good the quality of the solution and because it alleviates the need for specific documentation.

More consistency of skills and shared culture also helps faster decision-making. It's not about removing all diversity in the team, since diversity is an essential ingredient. Still, we don't need all diversity in every detail, and there's a lot that we can make more consistent without loosing much.

Investing in continuous training can be done with:

- Coding Dojos on code kata, e.g at lunch time every Friday

- Short Training sessions during the day
- Interactive Mini-Trainings, e.g. half an hour twice a week right after lunch
- Time for deliberate practice, e.g. 20% time policy.

# Part 9 Beyond Documentation

# Documentation-Driven Development

*Start by explaining your goal or end result, e.g. how your system will be used, in order to drive the construction and to help notice early potential inconsistencies*

It is often a good idea to start with a focus on the end result we are aiming for. By focusing on the end, we first focus on where the value is, to make sure it's really there. Then we can derive what's really necessary to achieve the goal, no less and no more, avoiding unnecessary work.

Chris Matts in a talk at the BDD eXchange conference had a great example on the most typical British goal of *having a cup of tea*. Starting with this goal, we can then derive the need for hot water, a clean cup, and a tea bag, and so on.

> Here's a secret about documentation. It's not just useful to read. It's the act of writing that pushes for quality in the same way as tests @giorgiosironi

Some developer find that starting with a piece of documentation helps do that, like Dave Balmer in his blog I want control of my documentation[59]:

> I can start by documenting only that which is important. That satisfies the "write this down before I forget" part of documentation, and frees me up to improve it in later drafts.

Test-Driven Development and its close cousin BDD exploit that effect by focusing on the desired behavior first, as a test, or a scenario or example written before starting the coding. So if you're practicing TDD or BDD you're already doing a form of documentation-driven development too.

When uncertainty is very high, at the very inception of an idea, writing the README file *as if the project was done* helps clarify the purpose and flesh out your expectations. Once materialized on paper, ideas become objects of deeper scrutiny, they can be criticized, reviewed, shared with other people early.

If you are alone, just let the time pass for a few days before going back to these notes: when you see them again with a fresh look, you can review your own work in a more objective fashion, thanks to the documentation from yourself to your future self.

[59] https://davebalmer.wordpress.com/2011/03/29/source-code-documentation-javadoc-vs-markdown/

## Documentation to keep you honest

Continuous improvement starts with honest retrospectives on how well we performed. When we work based on assumptions, it is easy at the end to forget about our past assumptions and either blame the environment in case of failure, or congratulate yourself for the success. Opportunities of improvement are in looking back at our assumptions in retrospect, to learn from them. Next time "I will not assume that", or "I will first check the assumption before investing more time".

Therefore: **Document early what you assume and the experiments you try in order to have reliable and honest data when it is time for retrospective**.

This is a way to be a little more data-driven. And there are tools for that! For example growth.founders.as offers a tool to declare your assumptions and to describe your experiments.

## The apparent contradiction between Documentation-Driven and "NoDocumentation"

At this stage you may confused at the apparent contradiction between being *Documentation-Driven* avoiding documentation as expressed in the earlier part of the book and in the *No Documentation* section in particular.

The contradiction is a matter of ambiguity of words indeed. When talking about documentation-driven development, even though we use the word *documentation*, we don't mean it as a way to share knowledge between people. Instead, it's just a cheap media to explore the requirements at the very inception of a project, before we move on to more expensive material like tests and source code.

The fundamental idea is that it can be desirable to use a different material at different levels of unknown and uncertainty: at the very start, conversations are usually the best material; at the early stages, conversations, notes and sketches on paper, low-fi mockups, README file with intentions and scenarios, code exploration in a REPL, writing code in a spike without tests, perhaps using a scripting or dynamic language may be the idea material to learn and explore. A bit later, when things start to stabilize, another programming language with tests and even with TDD becomes the material of choice. In this view, documentation early is just a material to get started.

Appart from this case, documentation must not drive the development, it must capture and help present the ideas which happened and what has been developed that the system itself and its code cannot explain themselves. Making the code as self-documenting as possible is the goal. Whenever we fail despite our best attempts, then we have to resort to some documentation effort, and we keep it to the minimum.

**Explore vs Capture & Present**

There's no contradiction. We're just talking different meanings of the word *documentation*.

# Abusing Living Documentation

So you' now a fan of Living Documentation and now you're generating diagrams during each build. You like the idea so much that you spend your time figuring out what new kind of diagram could be generated. You want to generate all the things!

You pretend to apply Domain-Driven Design but you actually spend your time on exciting tools that generate diagrams, if not code or bytecode. Because we all know that DDD is primarily about tools, right? Oh, yes, you remember some folks used to do that seriously, and they called it MDA. Ouch!

You prefer working on the diagram generator rather than fixing bugs in the production code. Of course, it's way more fun than boring production issues!

Is all that really a good thing?

## Living Documentation with Moderation

It's easy to abuse Living Documentation, and it can backfire. If you spend too much time on tools to generate glossaries, reports and diagrams, instead of doing the work to be done, it's not professional, and the management may decide to stop and forbid any documentation-improving initiative. You don't want that.

**Therefore: Keep your efforts in automating living documentation reasonable compared to the actual delivery work. Remember they are just a mean to an end and not an end in itself. The actual goal of living documentation is to help deliver more and with a better quality, not just producing documentation or having fun. Ideally, every effort in improving your living documentation should yield short-term demonstrable benefits in delivery, quality or user satisfaction.**

As the author of the book on living documentation, I don't want this topic to have a bad press because of people abusing it. Please don't say that this book is asking you to put in place every example described into your own project, because it's not true. All the examples are, well, examples, not requirements.

This is true that the point of this book is to excite the inner geek inside you to try the living documentation ideas. Still I would never advise you to do it all without a good reason for each of the ideas in the book.

# Procrastinate on Living Documentation

**Oops I've done it again**.

As developers, we are always tempted to make things more complicated than they need to be. This is true for production code, and it also true for your living documentation tools.

When the everyday work looks so boring, making it technically more complicated is a great way to have fun. However, you know, it's not professional. If you consider yourself as a software craftsman or craftswoman, you know you should not be doing that. However we all fall for it from time to time, even without being aware of it.

**Therefore: If really you need a space where you can have fun and make things needlessly over-complicated, then by all mean do it in the code of your living documentation tools, not in your production code. Your life and the life of your colleagues will be better as a result**.

This is not an advice to gold-plate your living documentation tools. This is just a preference between two abuses that are both not professional. If you're lucky enough to have some slack time for you to have fun with code, then this advice is for you!

# Listen To The Documentation

So you've discovered Living Documentation and you want to try it. You try to create a living diagram but you find it hard to generate out of the current source code?

This is a signal.

You try do generate a living glossary but you find it hard, almost impossible to achieve?

This is a signal, again.

Nat Pryce and Steve Freeman say about tests: "if you find it hard to write the tests, it's a signal that your design has issues". Similarly if you find it hard to generate living documents out of your code, then it's a signal that your code has design issues.

**Listen to your documentation!**

## What happened to the Language of the Domain?

If you're into DDD and you find it hard to generate a living glossary of the business domain language, then it's probably because this language is not expressed clearly enough in the code. For example, the language is perhaps:

- **translated** into other words, like technical words, synonyms, or worst, legacy database names
- **mixed with technical** concerns in a way that is impossible to recover, like business logic mixed with data persistence logic or presentation concerns,
- **completely lost**, with code doing business stuff without any reference to the corresponding business language.

Whatever the answer, the signal that the living documentation is hard to is highlighting that you're probably doing DDD, and domain modeling in general, wrong. The design should be aligned as much as possible with the business domain and its language, literally word by word.

So instead of trying to make a complicated tool to generate a living glossary, take this as an opportunity to re-design the code so that it better expresses the domain language. Of course, it's up to you to decide whether it's reasonable to do so, and when and how to do so. And in this case, you don't even have to invite a consultant on DDD to find out by yourself that you need to improve your practice of DDD!

## Programming by coincidence Design

In order to generate a design diagram, first you have to know what particular design decision you expect the diagram to explain. But can you tell what your design is like? The most common difficulty when trying to generate living diagram is simply that you often don't know clearly enough what your design is like, and why it's that way.

> "We don't know what we're doing, and we don't know what we've done"
>
> *Fred Brookes*

This suggests that you may be programming by coincidence[60]. You know how to make it work but you don't really know why, and you haven't really considered alternatives. This design is a bit arbitrary, not deliberate.

> If there is no choice to be made, you're not doing design. – Carlo Pescio in Design, Structure, and Decisions[61]

[60] PragmaticProgrammers

[61] http://www.carlopescio.com/2010/11/design-structure-and-decisions.html

> I love the essays of this guy. In fact I don't like the writing style much, but I like the way he writes about his mind musing on hard and deep matters of software development. Some crazy ideas, some stretched metaphors, but a lot of insights to spark my imagination, envisioning future breakthroughs in our field.

Building software is about continuous decision-making. Big decisions usually get a lot of attention, with dedicated meetings and several written documents, while other "less important" decisions are somewhat neglected. The problem is that many of these decisions end up being arbitrary rather than well-thought about, and their accumulated effect (perhaps even a compounding effect) is what will make the source code hostile to work with.

Why does this function return null instead of an empty list? Why are they not even consistent? Why are most of the DAO, but not all, in this package? Such neglected decisions sometime get close to better solutions but miss them, for lack of properly thinking about the matter. Why do you have the same method signature in 5 different classes, but without a common interface to unify them? All these examples represent lost opportunities for a better design.

> Whenever you find out something unexpected in the code and its design, consider asking yourself the question: "what would it take to come back to the standard situations in the literature?"

We want to encourage deliberate thinking. Documenting decisions as they are taken is one way to encourage a deeper thinking because trying to explain a decision often reveals its weaknesses.

> *If you can't explain it simply, you don't understand it well enough.*

Sometime it's frustrating when working with a team at a customer site to observe decisions to be taken without anyone being clear on the reasoning. "Just make it work right now" seems to be the motto. In one instance I had to take notes about one such situation:

> We've been discussing for one hour over the semantics of the messages between a legacy app and a new event-sourcing-based app. Is it event or command? As usual, the discussion doesn't lead to a clear conclusion, and yet the unclear choice works. Had we decided to document the semantics of all integration interactions clearly, we would have had to decide, and to turn it into a tag or something written and visible. Then we would have to conform to it, or to question it explicitly when it's no longer relevant.
>
> Instead, we're going to live with the continuous confusion. Each contributor will interpret as he or she wishes. And it will bite us.

Now I can tell that coming back one year later, the team has matured and now this discussion would be probably converge to a sound reasoning.

## Deliberate Decisions

It is very hard to document random decisions. It is like attempting to describe noise: there is at the same time too many low-levels details, and almost nothing to tell at a higher-level. In contrast, when the decisions are deliberate, they are clear and conscious, so documentation is basically about putting that into words.

If the decision is standard enough, it's READY-MADE KNOWLEDGE which has been already discussed in a book under a standard name, for example as a pattern. Documenting in this case is only a mark in the code that refers to the standard name, along with some brief reasons, motivation, context and main forces that led to the decision.

*If a decision is deliberate, it's already half-documented.*

Being deliberate in the way we do our work is a big recurring theme in Agile circles. In Software Craftsmanship we encourage Deliberate Practice to improve our craft. We dedicate time to practice katas and coding dojos to achieve that goal of getting better at our craft. In the BDD community, Dan North explains that projects should be seen as learning initiatives, a mindset he calls Deliberate Discovery. Together with Chris Matts they claim we should do whatever it takes to learn as quickly as possible as early as possible. Both cases illustrate how being deliberate is about doing extra effort to do a better work in one aspect, in a conscious way.

Deliberate Design is about thinking clearly about each design decision. What is the goal? What are the options? What do we know for sure, and what do we suspect? What does the literature say on this kind of situation?

There is more than that, as the better the design, the less there is to document. Better design is simpler, and "simpler" actually means fewer but more powerful decisions that solve more of the problem:

- **Symmetries**: the same code or interface takes care of all other symmetric cases
- **Higher-level concepts**: the same code deals with all special cases at once
- **Consistency**: some decision is repeated everywhere without exception
- **Replaceability and encapsulation**: local decisions within a boundary do not matter, as they can be re-considered or redone later even if their knowledge was lost

As such the mere quantity of specific knowledge to document a piece of software is an indicator of the maturity of the design. Software that can be described in 10 sentences has a better design that one that needs 100 sentences to be described.

## Engineering is a deliberate practice

In French engineering schools and other "Grandes Ecoles", from mechanical engineering to electronic engineering or even industrial design, it's primarily important for students to demonstrate that all decisions they made are substantiated. Arbitrary decisions are just not acceptable.

During the final exams the most important aspects of the evaluation are about framing the work precisely, and then at each steps of designing the solution each decision has to be justified against enough alternatives and chosen according to explicit criteria: budget, weight, feasibility or any other constraint.

In software development we seldom are that deliberate in every detail, still we should keep in mind to be as deliberate as possible. It does not matter that every decision is recorded in writing, but making more conscious decisions will often improve the decisions.

If you know what you're doing, how it's called in the literature and why you've made this decision, all it takes for a complete documentation is to add that information in the code in one line: a link to the literature, and some text to explain the rationale. Once you've got the thinking right, the writing takes care of itself.

You have to realize, of course, that thinking takes time. It looks slow and may be confused with slack hence in many companies people don't have much opportunities to think: "we don't have time for that!". However alternatives to thinking only give the illusion of speed, at the expense of accuracy. As Wyatt Earp says: "Fast is fine, but accuracy is everything." Accuracy comes from rigorous thinking. Thinking with more than one brain, as in pair programming or mob programming, also improves accuracy and helps make your design more deliberate. With more brains, it's more likely that someone knows the standard solutions from the literature for any situation.

Documentation also helps in that aspect. You know the saying: "you don't really understand something until you can explain it to someone else." Having to clarify your thoughts for documentation purposes is virtuous because, well, you have to clarify your thoughts. Having to justify decisions in a persistent form is another incentive to think with more rigor.

Deliberate Design works particularly well when doing TDD. TDD is a very deliberate practice with rules. Starting with naive code that just works, the design emerges from successive refactorings, but it's the developers who are driving the refactorings, and they have to think before applying each refactoring. "Do we really need to make that more complex?" "Is it worth adding an interface now?" "Shall we introduce a pattern to replace these two IF statements?". It's all about tradeoffs, which requires clear thinking.

Living Documentation is a pretext to encourage attention to virtuous practices, and in particular design. Living Documentation makes bad design visible to anyone. It's also rewarding when you improve the design and your design documentation comes almost for free as a result.

### A Confession from the author

Deliberate Design is my secret goal for writing this book! People don't pay enough attention to design, and I strongly regret that. Living Documentation is a Trojan Horse, or a gateway drug to get more people addicted to better design.

## Deliberate decision does not mean upfront decision

We want emerging design, where the natural decision emerges by listening to the working code and its flaws. For example, noticing duplication triggers a refactoring to something better. It is at this point that we have to make a conscious, deliberate decision: what is the "better" we want to refactor to? Deliberate means that we understand the troubles, we can imagine the benefits we are looking for, and we have found more than one way to improve. Making a decision means that we will chose one way out of all the possible others. This will be our deliberate decision.

## Documentation is a form of code review

Documentation makes the product and the development process more transparent. As such, they are also useful feedback tools, to adjust and correct over the complete life cycle of the application. Decisions with no rationale have nowhere to hide. With living diagrams and the other ideas of living documentation, the neglected design areas become visible to everyone, making it harder to ignore them. This puts a positive pressure to put more care on every aspect of code quality.

Living documentation generated from the actual source code, especially diagrams, also work great as debug tools to detect mistakes like unexpected cycles in the dependencies or an excessive coupling with too many arrows on the diagram. You were expecting some design structure, but when trying to render it as a diagram you have to admit the code does not exhibit much structure after all. You were expecting the code to tell the business domain, but when trying to make it into a glossary it appears that the business is mangled in the middle of the processing and there is no easy way to get it out.

It becomes interesting to compare the top-down documentation you may have done before building the code with the actual bottom-up documentation generated from the sources, to spot inconsistencies, or even better, to realize once again how it is hard to speculate about how the code will be before it is actually developed.

Indeed, even before doing living diagrams, just trying to document by hand on paper already reveals design issues. Maxime Saglan, a Lead Developer from one of our customers, reacted when reading an early version of this book: "That's totally what happened when I started to have the team do sketching workshops around Simon Brown's C4 model, on the legacy system."

# Biodegradable Documentation

Biodegradable Documentation

**Therefore: Consider what it would take for documentation to become unnecessary. This becomes the direction you should move towards.**

It becomes clear at this point that Living Documentation is not an end in itself, but a mean to an end. Trying to setup a Living Documentation reveals issues about the design, or any other aspect. This becomes an opportunity to improve the root cause, which is above all good for the project and the product, and that in turn makes it possible to have a living documentation, or helps improve the living documentation. Doing that repeatedly leads to a stream of simplifications and standardizations. By this logic, everything becomes so simple and so standard that you don't need documentation any longer. And that would be perfect.

It does not matter that you actually reach this point or not, but it has to be the goal. The goal of a living documentation initiative is to achieve the level of quality where documentation is mostly unnecessary. The goal of living documentation is not to end up with a lot of beautiful generated diagrams and documents. Instead, they should be considered as workaround solutions or intermediate steps towards better solutions that need less documentation.

**Living Documentation long term goal is for documentation to become unnecessary**

My Arolla colleague Khaled Souf once told me of a former experience at a bank:

> In that bank, I joined a team who took pride in conforming to every standard. I mean market standards, not in-house standards. The result was that I was able to be productive as soon as the first day! Since I knew the technologies and their standard usage, I was immediately familiar on all the project perimeter. No need for documentation, no surprise, no need for any specific customization.
>
> Make no mistake, this was taking a real and continuous effort indeed. Find out the standards, find out the way to solve specific issues while still conforming to standards. This was a deliberate approach, and the benefits were real, for everyone but especially for new joiners!

In the book *Software Craftsmanship Apprenticeship Patterns*, Dave Hoover and Adewale Oshineye advocate: "Create feedback loops". A living documentation with generated diagrams, glossary, word cloud or any other media is such a feedback look to help criticize what you're doing and to check

against your own mental model. This feedback loop becomes particularly useful when your mental model and the content of the generated documents don't match.

# 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 \
files (respectively *.cs files in C#)

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

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

// Ignore numbers and tokens starting with '@', or that are keywords and stopwor\
ds for the programming language:

KEYWORDS = { "abstract", "continue", "for", "new", "switch", "assert", "default"\
, "if", "package", "synchronized", "boolean", "do", "goto", "private", "this", "\
break", "double", "implements", "protected", "throw", "byte", "else", "import", \
"public", "throws", "case", "enum", "instanceof", "return", "transient", "catch"\
, "extends", "int", "", "short", "try", "char", "final", "interface", "static", \
"void", "class", "finally", "long", "strictfp", "volatile", "const", "float", "n\
ative", "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"[62] 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:

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

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

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.

# Living Documentation Going Wild

Living documentation is not a free license to rehash old ideas from the 90's. In particular, beware of the following, which is not Living Documentation:

- **MDA and everything code generation**: No, code is not a dirty detail to replace or generate, it is the reference and the preferred media whenever we can. Extend your language, or chose a better programming language, instead of generating code from diagrams
- **Documenting everything, even automatically**: Documenting has a cost, which must be weighted against the benefits. The ideal case is when the code is so self-descriptive it needs nothing else, but even that is not an absolute. Perfection and the quest for purity is often a form of procrastination to be avoided.
- **UML Obsession**: Some basic UML is fine, but it is not an end in itself. Chose the simplest notation that the intended audience will really understand with as few explanation as possible. Don't obsess on generic notations, problem-specific or domain-specific notations are often more expressive.
- **Design Patterns Everywhere**: Patterns are good to know, and they help documenting the design through the vocabulary they bring. Don't abuse patterns. It's called "patternitis". Simplicity always first. Perhaps two IF statements are better than a Strategy pattern here!
- **Analysis Paralysis**: Spending 15 mn all the team together on the whiteboard before each important design decision is time well spent. Spending hours or days is waste. Start with something, anything really. Then it becomes obvious to everyone what's ok and what's not so ok. Now iterate, and take some brief notes Living Documentation-style!
- **Living Documentation Masterpiece**: Again that's a form of procrastination, when the production work is so boring you escape and work on something more fun instead. Keep in mind Living Documentation is just a mean to help deliver production code, not the other way round.
- **Documentation before building**: Documentation should reflect what's actually built, more than prescribe what will be built. If your project is interesting, then nothing can beat starting the code. Detailed design specs are waste. Start coding and reflect along the way, collectively, in a just-enough, just-in-time fashion.

# BREAKING!!! Live Interview: Mrs Reporter Porter interviewing Mr Living Doc Doctor!

**Living Doc Doc interviewed by Reporter Porter**

**What is a good documentation?**

The best documentation is when you never have to use the word. When everything is so obvious you understand it immediately, or he naming is so good it's instantly clear. Or when it's so integrated in your workflow and into you daily tools that you don't even think about it as really documentation. One striking example is when a tools reminds you of something you forgot or didn't know, right when you need it. We usually don't call that documentation, but the end purpose of bringing the right piece of knowledge at the right time is what matters.

**Why is that not popular?**

I think it's popular, but nobody noticed. Remember all the focus on UML in the early 2000's? Now projects are bigger, and we don't use them. Instead, every IDE offers instant, integrated and highly contextual Type Hierarchy trees, outline, smooth hypermedia navigation between classes… this is

more useful than hundreds of static UML diagrams. Still we take it for granted and still feel bad about the "lack of proper documentation" (lol)

And there's new technologies as well…

**How new techs changed the picture?**

Most people still haven't realized all the potential of newer tools and practices when it comes to transfer knowledge.

Consul, Zipkin offer live recap of what's actually there, even as living diagrams. They offer tag mechanism to customize and convey intents.

Monitoring of key SLA metrics with thresholds get close to documenting the SLA.

Puppet, Ansible and Docker files allow for a declarative style for describing what you expect. Imagine all the Word documents they advantageously replace!

**So you need not doing anything special now?**

Almost. But not totally. All the new techs & practices is fantastic to document the WHAT and the HOW, but the weak point mostly everywhere remains the rationale, the WHY, that often gets forgotten. That's why you should still find a way to record the rationale of the main decisions. Immutable Append-Only Log, Augmenting Code with tags, and a few Evergreen Documents for the overall vision can be invaluable to complete the whole picture.

**And what about the code?**

Code should be self-documented as much as possible. Your tests and business-readable scenarios are an important part of this recorded knowledge. But sometime you have to add extra code just to record your design decision and intention right inside the corresponding code: custom annotations for documentation and naming conventions are your tools of choice here.

**Ok, but these days systems are made of dozens of services, how do I do with such fragmented systems?**

You just apply the same techniques, but at a different level. For example, annotations become tags in your service registry and in your distributed tracing system. Naming conventions of packages and modules become naming conventions of services and their endpoints. Similar thinking, similar design skills, different implementation!

**Do we really need documentation? After all, we've been living with little or no documentation for years and we're still alive!**

Of course we can live without express documentation. Anyone can take an unknown system and make it work, for some definition of "work". But just "making it work" is a very low bar. And how much time does it take? Documentation accelerates delivery because it considerably shortens the time to rebuild your mental model of the system to work on. But the other effect of documentation is that trying to record the knowledge about the system is a great way to learn about what's not right about the system. Paying attention to documentation is an investment for later, obviously, but less obviously there's also a return for right now!

**Thank you very much!**

No, thank YOU very much!

# Part 10 Living Design & Architecture Documentation

# Living Design & Architecture

The best definition of design I know is from Alistair Cockburn:

> If it's your decision to make, then it's 'design'; if it's not, then it is a requirement to you.
> From Alistair Cockburn[63]

Note that we call architecture is design too.

Design is not a phase, it's about the all the concerns we decide to pay attention to while coding. We would consider the ease of evolution of the code, its readability, how it can be understood without help.

Software design is about deciding carefully between all the possible ways to code the same behavior. Or in Jeremie Chassaing's words: "it's picking one in gazillions of possibilities with good reasons." Hence the frequent objection during design discussions: "but it's the same thing at the end!" Yes it is. If you just care about making it work, design is irrelevant. Design is when you care about concerns beyond just making it work.

If you practice TDD, design is what you do in the third step "Refactoring", after creating a failing test (specs) and making it work with just enough code.

Design is about trying things only to change them later when we know better. Being serious about design means continuous changes of the design and the code which embodies the design.

The code as you see it is a consequence of all the past design decisions. You can recognize many of these past decisions just by looking at the code base, but you'll miss many as well.

Living Documentation in the case of design is precisely about finding practices that help explain more design decisions accurately, without slowing down the continuous flow of changes that we make when we design.

We've covered in previous chapters several examples of living documentation applied on design concerns: living diagrams on Hexagonal Architecture, on micro-services, Ready-Made Knowledge with design patterns, Code as Documentation. This section elaborates a bit on top of that.

## The design skills pre-requisite

Even if you start a Living Documentation journey with the goal of solving the documentation problem, you'll quickly discover that your actual problem is that the design is poor or arbitrary:

[63] http://alistair.cockburn.us/A+requirement+is+a+relationship+to+a+decision

"design by coincidence". To solve the documentation problems you have to solve the design problems. This is all good news indeed!

Through the focus on documentation you end up with a concrete visible criteria for everyone to see the big mess that is the current state of the design. This becomes a positive pressure to improve the design, which has benefits well exceeding the sole benefits of documentation per se. But as we've mentioned before, there's even more good news for you:

**Good design skills also make good living documentation skills**.

Focus on living documentation, and focus on design skills. Practice both together and everything will get better!

Conversely, with the right skills, you can recognize - reverse-engineer indeed, many of the past decisions just by noticing the happy coincidences in the code base: "it cannot happen by chance, so it must have been designed".

With the right skills, the design tells out loud what it is and how it is expected to be extended. Just like this multiplug, that is visibly ready for extension by plugging additional plugs to it:

**Visibly ready for extension**

## Documenting errors, or avoiding errors?

An important part of documentation is to describe the error cases. But there's a better way: we could avoid the error cases altogether with better design and better coding practices.

Let's take the example of a function that calculates an inverse. If the divisor is zero then there is no result. This is often a case of error management, but an alternative is to make the function a Total Function.

A total function is a function that works for all possible parameters values. In our case, all it takes to make our function total is to extend the number type with the NotANumber special value. Now the function can just return NotANumber when there's a division by zero, instead of going the error management road.

## Naming (again)

Naming style does not have to be uniform. For example in my taste I always go for business domain names within a domain model or domain layer: `Account`, `ContactBook`, `Trend`. But on the infrastructure layer or adapters (in the Hexagonal Architecture sense) I'd go for prefixes and suffixes to qualify technologies and patterns being used in the corresponding implementing sub-classes: `MongoDBAccount`, `PostgresEmailsDAOAdapter`, `RabbitMQEventPublisher`.

I'd say that the name here tells all you need to know. Any additional detail can just be put in the class structured comment.

## Design Annotations

Any design information that can make the code more explicit is worth adding. If you follow the Layers pattern, you can document it by putting a custom annotation @Layer on the package at the root of each layer: `com.example.infrastructure/package-info.java`

```
1  @Layer(LayerType.INFRASTRUCTURE)
2  package com.example.infrastructure;
```

Stereotypes-like patterns represent intrinsic roles or properties that qualify a language element like a method:

```
1  @Idempotent
2  void store(Customer customer);
```

or

```
@SideEffect(SideEffect.WRITE, "Database”)
void save(Event e){...}
```

Specific risks or concerns can also be denoted directly on the corresponding class, method or field:

```
@Sensitive("Risk of Fraud")
public final class CreditCard {...
```

Design patterns in general are good candidates for design annotations. You place the annotation on an elements that participates actively to the pattern. You can check that by wondering “if I removed the pattern, should I keep this element?” If the answer is no then you can safely declare the pattern on it (the class or method is only there to realize the pattern). It is often the element in the role having the name of the pattern itself, like the `Adapter`, or the `Command`.

Sometime you need values in your annotations. For example if you want to declare an occurrence of the DDD repository pattern that is manipulating a particular aggregate, you would do it like this:

```
@Repository(aggregateRoot = Customer.class)
public interface AllCustomers {...
```

You can create your own patterns catalogue with the patterns you use most commonly. It would include patterns from the GoF, DDD, Fowler (Analysis Patterns and PoEAA), EIP, some PLoP & POSA patterns, and several well-knowns and/or trivial basic patterns and idioms, plus all your custom in-house patterns.

In addition you may create custom annotations to classify some important sources of knowledge, like Business Rules, policies etc.

```
@BusinessRule
Public Date shiftDate(Date date, BusinessCalendar calendar){...}
```

Here are some more examples to illustrate:

- @Policy to highlight each company policy
- @BusinessConvention
- @KeyConcept, @Core to emphasize what’s important
- @Adapter, @Composite to denote the use of a pattern
- @Command or @Query to clarify the semantics of write or read on a module or
- @CorrelationID, or AggregateID on a field

# Enforced Design Decisions

Thanks to the augmentation of the code with design knowledge, using annotations, naming conventions, tags in your service registry or any other mechanism, you can delegate conformity checks to tools. You can check dependencies according to the declared patterns and stereotypes knowledge. I like to raise an anomaly if a class annotated as a `Value Object` has field-level dependencies to classes annotated as `Entity` or `Service`. That's my taste, and I can ask tools to check it for me.

```
if (type.isInvolvedIn(VALUE_OBJECT)) {
  if (dependency.isInvolvedIn(ENTITY) ||
     dependency.isInvolvedIn(SERVICE)) {
      ... raise an anomaly
}
```

You may also create custom rules in your static analysis tool. For example, using the SonarQube built-in Architectural Constraint template, you could create the rules:

- "Persistence layer cannot depend on web code": forbid access to **.web. from **.dao.** classes
- "Hexagonal Architecture": forbid access to **.domain.** to **.infra.**

The name of the rule and its definition as access restrictions clearly documents and protects the design decision at the same time, all in one place.

# Coding against a framework

The code not written needs no documentation. When you use an off-the-shelf framework like Spring Boot (a lightweight microservice framework) or Axon-Framework (a framework for Event-Sourced applications), a lot of code is already written, and your code has to conform to what's expected by the framework. Choosing such frameworks may be a good idea for teams of limited maturity where the framework will constrain the design to follow some structure. This may sound like a bad thing, but it's a good thing from a knowledge transfer perspective: no room for surprise, once you're familiar with the framework you can understand most of the code. In addition, such frameworks are well-documented, and their use of annotations also documents your code:

```
  @CommandHandler
  void placePurchase(PlacePurchaseCommand c){...}
```

> But if you write an app 'without a framework', you end up with an under-specified, undocumented, informal framework. – from Hacker News: https://news.ycombinator.com/item?id=10839081

# Exemplarity in the case of design

If you advertise some kind of segregation of concerns in the code, you should follow the same kind of segregation in the documentation. This way the very documentation structure brings the point home.

For example in functional programming we pay a lot of attention in segregating the side-effects from the pure (side-effect-free) code. When the documentation, whatever its form, follows the same structure:

1. Pure domain code (XL)
2. Side-Effects (S)
3. Control (S)

The segregation and strong preference for pure code becomes obvious through the structure, ordering and relative size of each section: most of the program is pure, making it the largest section by far. It's also where's the interesting things about the domain are represented, so this section comes first.

The two other sections explain the rest, the necessary evil for the program to be useful. The code for that is kept simple and minimal, therefore these sections are small.

# System Metaphor (XP, DDD)

## Explaining a system by talking about another system

If you happen to do trainings, you must know how hard it is to explain something to an audience you sit know. The key question is "What do they know already?", because you'll build on that.

That's where metaphors take their power: by leveraging on things most people are already familiar with, we can explain new stuff more efficiently.

> "It's like X" "it's structured like X"

When I explain monoids and how they compose, I usually take the metaphor to the tangible world, using real glasses of beers that I can stack. Or chairs that can be stacked. Or anything stackable indeed. This brings the point of monoid-esque composability, and it's fun, which is also very good for learning.

Suggestive metaphors we're all familiar with: an Assembly Line, a water Pipeline, Lego Building Blocks, a Train on its Rails, or a Bill of Materials.

The system Metaphor was part of Extreme Programming (XP) to unify an architecture and provide naming conventions.

"A simple shared story of how the system works, a metaphor". From the C2 Wiki[64]

The famous eXtreme Programming project C3 "was built as a production line" while the other famous XP project VCAPS "was structured like a bill of materials". The chosen metaphor acts as a system of names, relations and roles working together towards a shared purpose. When using the metaphor, you invoke all the prior knowledge of the audience to be reused in the context of the system being considered. You know that an assembly line is typically linear, with multiple machines in line along the conveyor belt that is moving parts from one machine to the next. You also know that any defect upstream will result in defects downstream.

## With or without prior knowledge

Last time we built a rich cash flow calculation engine able to recreate cash from any complicated financial instrument, we used the metaphor of a modular synthesizer. Granted, not everyone is that familiar with this one, but that team was. The few who didn't were happy to learn about it.

[64] http://c2.com/cgi/wiki?SystemMetaphor

By the way that's interesting: a metaphor remains a little useful even when you don't know it just as a redundancy mechanism. Imagine you're trying to mentally picture the cash flow engine as an Interpreter pattern, and you're not fully sure you got it right; now I'll explain what's a modular synth, and it should help.

A modular synth is a kind of synth made of independent modules, or "building blocks*" that we connect to each other using short Jack cords. Some modules produce simple sounds (Oscillators), some alter a sound timbre (Filters), other alter the sound intensity (Amplifier), other combines different sounds together (Mixer), other add effects. Some modules don't produce sound but modulate other module's action: for example a module continuously ramping up and down (so called LFO, for Low-Frequency Oscillator) can be wired ("patched") to the sound producer (Oscillator) to modulate its pitch, resulting in a "vibrato" effect. The patching combinations between every connector are near infinite, for a large variety of sounds to be produced.

- Notice the metaphor in the metaphor here?

Physics and mathematics regularly borrow from other fields. For example the "Simulated annealing" is a notorious method to solve optimization problems:

> The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects – From https://en.wikipedia.org/wiki/Simulated_annealing

A good metaphor is a model with some generative power: if I know that stopping a production line is very expensive, I can wonder how that would translate into our software system. Perhaps it does, and just like on a production line, we should perform strict validation of the raw materials in inputs to protect the line. But the metaphor may not stand on this aspect, and that's it.

The more common culture the more inspiration available to use as a metaphor. Once you know what a Sales Funnel is, you can talk about it to explain key aspects of an e-commerce system, with its successive business stages from visitor to inquiries, to proposals, to new customer. And it's called a funnel because the volume at each stage decreases significantly.

**A sales funnel**

This knowledge comes in handy when doing architecture, as it informs scalability reasoning: the upstream stage like the Catalogue needs more scalability than the downstream stage like the Payment.

# Architecture Landscape

## Architecture and documentation

There is a close relationship between architecture and documentation.

How do you document the architecture of your system? The short answer is that whatever what you call architecture, doing architecture is in itself an act of documentation between every people involved in the project.

Many definitions of architecture have been proposed over the past decades, but the following two are my favorite:

1. Things that everybody should know
2. Things that are hard to change

The first definition in itself admits that architecture is totally a matter of shared knowledge, hence a matter of documentation. The second definition is more precise, but it seems likely that the things that are hard to change should also be known by everyone.

## Everybody should know about the problem

Architecture always starts with really understanding all the objectives and constraints of the problem to solve. You won't build the same point-of-sale system for a brand with 50 hot dog stands in the street as you would for a 1500 high-end pret-à-porter shops around the world, even if they have the same high-level basic features.

**Architecture is about things that everybody should know**

The high-level goals, and the main constraints are "things that everybody should know", and as such they are always part of the architecture.

Therefore: **Whatever your definition of architecture, make sure it is considered as much a documentation challenge as a technical one. Focus the discussions and their written records on the problems to solve, not just on the solutions. Make sure the essential knowledge about the problem is well described in a persistent form, and ensure it is in everybody's mind**.

You may ask random questions from time to time to check everyone involved knows about the essential business knowledge. I regularly like to do that, since it it's not the case we'll probably waste a lot of time during every discussion.

Keep in mind that written form is almost never enough, not everyone will read it. You'd better complement with random discussions and roadshows to present it to every team during official work time.

## An example of a problem brief

Here is an example inspired from a real-world project on a legacy system for one of my customer.

The brief is not in the wiki, but in a single Markdown text file at the root of the source code repository of the new component. It may even be in the README file.

# Vision Statement

Date: 01/06/2015

*Delight the users with great UI's and new features delivered frequently*

## Description

The INSURANCE 2020 program aims at revamping the legacy software supporting the insurance claim management processes, with two main goals in target:

1. User Experience (UX) & user-friendliness UI's
2. Continuous Delivery: reduced time to market and reduced cost of change

## Stakeholders

The primary stakeholders are the Insurance Adjusters.

Other stakeholders are:

- Actuaries
- Management

IT-related stakeholders are:

- Development team
- Central Architecture Group
- Support and Ops teams

## Business Domain

The business domain focuses on the claim management part, and in particular the Claim Adjustment phase. This starts at the early mention of a claim to start every investigation necessary to plan, witness the damages, contact the police officers, lawyer in order to propose a monetary amount to give to the policy holder.

The main business capabilities include for example:

- Take note of a claim without much information about it
- Enrich the claim whenever more information is available: parties involved, checks, evidences, photographs…

- Prepare the claim with one or more possible settlement offer(s) (each made of one or more monetary amount(s))
- Manage the claim team and the related workflows
- Report the current state of one or all pending claims
- Help users see their tasks to do at any time

# Quality Attributes

In software, quality attributes shape the solution. The technical solution to a given business problem will be radically different for millions of concurrent users as opposed to 100 concurrent users, if it's real-time as opposed to daily, or if each minute of downtime cost $500k to the company.

As a consequence, everybody should be aware of the most challenging quality attributes. They should also understand that other quality attributes are which are not challenging are opportunities to keep the architecture simple. Pretending that your design should support millions of concurrent users when you really have only thousands is a dangerous misuse of the sponsor's money and time.

**Therefore: At the start of a project, clarify the main quality attributes in writing. It can be as simple as a list of bullet points. Make it clear how to interpret the quality attributes, for example using maxims as guidance.**

An example of describing the main quality attributes.

> The system shall respond to user requests with a response time within 1 second for 98% of the transactions. The system shall support 3000 concurrent users.

It should come with some internal guidance on how to interpret the quality attribute, for example:

> Overquality is NOT quality

> Design for ∼10X growth, but plan to rewrite before ∼100X – Jeff Dean (Google)[65]

These quality attributes can then be turned into executable scenarios against the system, expressing the quality attributes literally in plain English sentences (see Test-Driven Architecture).

# Stake-Driven Architecture Documentation

We've seen before two definitions of architecture: "Things that everybody should know", and "Things that are hard to change. There are many other perspectives on architecture. Some developers

[65] http://static.googleusercontent.com/media/research.google.com/en//people/jeff/WSDM09-keynote.pdf

consider architecture as being all about the large-scale system with its infrastructure, expensive middleware, distributed components and databases replication. In fact this is normal for different people working on different systems to focus on different aspects of software and call it "architecture": they call architecture the aspects of the software which are most at stake in their context.

This diversity of perspectives is made obvious when doing an Architectural Kata. During this workshop format proposed by Ted Neward, groups of people are tasked in creating an architecture for a given business problem. Each group has 30mn and a big piece of paper with markers to prepare and present a proposal. The rules clearly emphasizes that the group should be able to justify any decision taken. The workshop ends with each group presenting its architecture to everyone else, as if they were defending the proposal in front of a client. Other attendees are invited to ask questions to challenge the proposal, as a skeptical client would do.

This workshop is very interesting to think about architecture. It is in itself a communication exercise. It is not just about the decisions taken, it is also about expressing them in a convincing way. Almost invariably, this kata reveals how different people think very differently about the same problem.

> You may be tempted to use this kata on real business cases, as a form of competitive engineering, with different groups proposing different views which are later compared. However the risk is that on a real case would be to have "winners" and "losers" groups at the end. You should practice it several times as a pure kata first, without real stakes. You will get a lot of value and thinking out of it, and you will also learn how to avoid the "winner vs. loser" effect.

What I learnt from this kata is that different business problems call for a focus on different areas. The main aspect of point of sale system in the street is to be lightweight, cheap in case it is stolen, and easy to use while making hog dogs in a hurry in the middle of a little crowd. In contrast, a mobile app meant to sell itself on an app store has to be primarily visually attractive, whereas an enterprise system meant to serve millions of transactions by second should before all focus on performance as its main stake. And there are systems where the main stake is on their deeper understanding of the business domain.

These key stakes of the system are what should be primarily recorded for everyone to know.

Therefore: **Identify early the main stake of the project: business domain challenge, technical concern, quality of the user-experience, integration with other systems… You may ask the question: "What would most easily make the project a failure?". Make sure your documentation efforts primarily cover this main stake.**

To caricature, don't spend too much of your time documenting the server technology stack when the main stake of the whole project is on the UX.

# Explicit Assumptions

When knowledge is incomplete, like it usually is at the beginning of any interesting project, we make assumptions. Assumptions make it possible to move on, but at the expense of potentially being shown wrong later on. It is a matter of documentation to make it cheaper to rewind the tape when you reconsider an assumption. A simple way to do that is to explicitly mark decisions with the assumptions they depend upon. This way, when an assumption is reconsidered, it is possible to find all its consequences, to reconsider them in turn. For this to work efficiently, it should all be done as an Internal Documentation, in place within the decisions, i.e. usually in the source code itself.

## Brief to explain

A good architecture is simple and looks obvious. It is also easy to describe in just a few sentences. A few key decisions, sharp and opinionated, which guide every other decision is a good architecture.

If architecture is "what everyone should know", then this puts an upper bound on its complexity. Anything complex to explain will not be understood by most.

> I've seen a good example of a good architecture in Fred George's talk at Oredev 2013 on micro-services architecture. Fred Georges manages to explain the key ideas of this architecture in minutes. It sounds as if it was simplified, and it probably is, deliberately. There is a lot of value in a caricatural architecture, which can be quickly understood by everyone. On the other hand, optimizing every detail is harmful if it makes the whole impossible to explain quickly.

**Therefore: Try to express the architecture out loud in less than 2 minutes as a test of its quality. If you succeed, then write it down immediately. If it takes much longer and too many sentences to explain the architecture, then it can probably be improved a lot.**

## Evolving continuously - Change-Friendly Documentation

The best architecture is an evolving creature, since it is hard to get it right at first try, and then it has to adapt to the changing context.

A good architecture is easy to explain succinctly and minimizes the number of decisions that are hard to change. But whatever is left hard to change or that everybody should know has to be documented. It has to be persistent over time and made accessible to everyone, by definition.

This means that whatever makes the architecture or its documentation hard or expensive to change must be avoided. Your team should learn how to make reversible decisions, or to defer irreversible decisions. And if you fear changing the architecture because you have a lot of static documentation about it that would have to be redone, then your documentation is harming you and you should reconsider how you do it.

**Pay attention to how much words and diagrams are needed to explain the architecture, the less being probably the better.**

Keep it all evolving, removing any process or artifact which would impede continuous change.

# Architecture Steering

> Architecture should not be defined but rather discovered, refined, evolved, and explained. #theFirstMisconceptionAboutArchitecture – @mittie

The old-fashioned idea of architecture as something to perform *before* doing the implementation doesn't fit well with modern projects. Change is expected everywhere and at anytime, in the code and in the architecture, whatever you call architecture.

Software architecture is about making sure that the major quality attributes of the overall system are met (e.g. conceptual integrity, performance, maintainability, security, fault-tolerance…) and communicating the most important decisions to everyone involved.

Documentation in any form is therefore an integral part of what architecture really is. But we don't want old-fashioned architecture practices to slow down our projects. We want fast documentation that can help communicate knowledge to everyone, and that can also help reason and make sure the quality attributes are satisfied.

Note that the quality attributes requirements usually don't change that frequently, but the decisions in the code do.

**Therefore: Regularly visualize the architecture as the software changes. Compare the architecture as-implemented to your architecture as-intended. If they differ you may want to adjust one or the other. With automated support of Living Diagrams or other Living Documents, this comparison can be done as often as during each build.**

All this assumes you have some vision of what your intended architecture should be. But if you don't have one then you can gradually reverse engineer it from your architecture as-implemented.

There are tools available that can help with architecture visualization and checking, and you can also create your own living diagram generators totally dedicated to your own specific context.

# Decision Log

> Technology is about tradeoffs and choices @simonbrown from Twitter

Why does the project use this particular heavyweight technology? Hopefully it was chosen because of some requirements, following some evaluation. Who remembers that? Now that the works has changed could you switch to something simpler?

What do you talk about during meetings with the stakeholders? From inception meetings to sprint planning meetings and other impromptu meetings, a lot of concepts, thinking and decisions are taken. What happens to all this knowledge?

Sometime it only survives in the mind of the attendees. Sometime it is quickly written as minutes of the meeting and sent by email. Sometime a snapshot of the whiteboard is taken and shared. Some put everything into the tracker tool, or in their wiki.

One common problem in all these cases is that this knowledge lacks structure in the way it is organized

**Therefore: Maintain a Decision Log of the most important architectural decisions. It can be as simple as structured text files in a folder at the root of the code repository. Keep the Decision Log versioned with the code base. For each important decision, record the decision, its rationale (why it was taken), the main alternatives considered, and the main consequences, if any. Never update the entries in the decision log; instead, add a new entry that supersedes the previous one, with a reference to it.**

Michael Nygard calls this decision log an Architectural Decision Record[66], or ADR in short. Nat Pryce created ADR Tool[67] to support them from the command line.

The structuring assumptions that shape the solution are part of the decision log, as part of the rationale for an important decision. For example, if you assume that articles published in the last 24 hours represent over 80% of the visits on your website, then it will show in the rationale for the decision to partition *recent news* vs "archived news" as two distinct sub-systems, each with a different local architecture.

In practice it's not always that easy to record the rationale of major architecture decisions, for example when the decisions are done for the wrong reasons. The managements insisted to include this technology. The developers insisted to try this new library for Résumé-driven development reasons. It's hard to make that explicit in writing for everyone to see!

[66] http://thinkrelevance.com/blog/2011/11/15/documenting-architecture-decisions

[67] https://github.com/npryce/adr-tools

**Decision log recording a not so good rationale…**

*You can find good examples of ADR online in the Arachne-framework Repository of Architecture Decision Records*[68]

## An example of structured Decision Log

In the example below, the decision log is maintained as a one single Markdown file at the root of the new Claim Management repository, after the vision statement, the description of the business domain and of the main stakeholders.

> Because we want this new micro-service to become the blueprint for many other micro-services to create on top of the existing legacy system, we may end up with other decision logs looking quite similar to each other. When this becomes an issue, we may decide to turn the recurring decisions into a style, document it on one place (e.g. in its own empty repository) and reference it in each service which conforms to this style.
>
> # The main decisions

[68] https://github.com/arachne-framework/architecture

To improve the overall User Experience, it has been decided:

- A UX approach, with a focus on beautiful and user-friendly screens, responsive across mobile devices, consistent between them regardless of the actual application behind, and with fast perceived response times. The focus is also on making sure that common tasks can be fulfilled efficiently, with few clicks and few page navigation.

The context of the existing legacy software makes it hard to achieve the vision stated above. This is why a large part of the program is to revamp the legacy, by decommissioning it as much as possible. To mitigate the risks of this decommissioning, the following decisions have been made:

- A Progressive approach, with frequent delivery: no Big Bang. New modules and legacy modules will co-exist, with a progressive migration to new code.
- A Domain-Driven Design approach to help partition the legacy in a way which makes sense at the business domain level, to better understand the domain mechanisms, and to be easier to evolve when the business rules change.

Another challenge is that many business rules are tacit in the mind of senior Adjusters, and need to be formalized. On top of that, with claims taking months to complete, these rules may change during the life of a pending claim. As a consequence, the following decision has been made:

- A Business Process Modeling approach to formalize tacit domain business rules in one place which can be easily audited and changed.

# Consequences

## Risks

One risk is the lack of expertise in the selected approaches. To mitigate this risk, external Experts have been involved:

- UX experts (from the internal UX Center)
- BPMN expert (Not found yet)
- DDD expert (from Arolla)

Another risk comes from the legacy context, in particular:

- Cost of testing: the lack of automated tests of all kinds makes each release expensive (manual testing) and/or dangerous (not enough testing)
- User-Perceived Performance: the legacy is slow, which makes it not suited for the expected response time perceived by end-users.

To reduce the cost of testing, and to not impede the users during all the changes in the legacy, test automation will be key (Unit Tests, Integration Tests, Non Regression Tests) in order to protect the system against regressions or defects

On the issue of user-perceived performance, the design will have to find workarounds to improve the perceived performance even though the legacy code behind may remain slow.

# Technical Decisions

## new Claim Management as Single Source of Truth until the claim is accepted by the customer

Accepted on 01/12/2015

### Context

We want avoid confusion arising from unclear authority of data, which consumes developer time to fix failing reconciliations. This requires that only source of truth (aka Golden Source) can exist at any point in time for a given piece of domain data.

### Decision

We decide that Claim Management is the only source of truth (aka Golden Source) for Claim on claim inception and until the claim is accepted by the customer, at which time it is **pushed** to the legacy claim mainframe. From the moment it is pushed, the only source of truth is the legacy claim mainframe (LCM).

### Consequences

Given the legacy background, it is unfortunately necessary for some time to have a different Golden Source across the life of a claim. Still, at any point in the life of the claim, the authoritative data are clearly in one single source. This should be re-considered to move to one constant single source whenever possible.

Because of that discrepancy, before the push: commands to create or update a claim are sent to Claim Management, with events sent around and in particular to LCM to sync the LCM data (*Legacy claim mainframe as a Read Model*). After the push: remote calls to LCM are used to update the claim in LCM, with events sent back to Claim Management to sync it (*Claim Management as a Read Model*).

See CQRS, Read Models and Persistence on InfoQ[69]

## CQRS & Event Sourcing

Accepted on 01/06/2015

### Context

In the claim adjustment domain audit is paramount: we need to be able to tell what happened in an accurate fashion.

[69] https://www.infoq.com/news/2015/10/cqrs-read-models-persistence

We want to exploit the asymmetry between write and read actions to the Claim Management models, in particular to speed up read accesses.

We also want to keep track of the user intents by being more task-oriented.

### Decision

We follow the CQRS[70] approach combined with Event Sourcing[71]

### Consequences

We chose AxonFramework[72] to structure the developments with its ready-made interfaces, annotations and boilerplate code already written.

## Value-First

Accepted on 01/06/2015

### Context

We want to avoid bugs that arise from mutability.

We also want reduce the amount of boilerplate code necessary in Java to create value objects.

### Decision

We favor value objects[73] whenever possible. They are immutable, with a valued constructor. They may come with a builder when needed.

### Consequences

We chose Lombok framework[74] to help generate the boilerplate code for value objects and their builders in Java.

# Journal or Blog as a Brain Dump

An alternative to a formal Decision Log is to dump your brain by telling the full story of what happened, what you learnt and how the team came up with the decisions, tradeoff or a particular implementation detail.

In the book *Software Craftsmanship Apprenticeship Patterns*, Dave Hoover and Adewale Oshineye advocate “Record what you learn” and “Share what you learn”. A blog from the team members is a nice complement to any other kind of documentation. Though it is more personal, it tells stories that are more compelling that most documentation. It tells important bits of the adventure, with the feelings of the people that were part of it.

[70] http://martinfowler.com/bliki/CQRS.html
[71] http://martinfowler.com/eaaDev/EventSourcing.html
[72] http://www.axonframework.org
[73] http://martinfowler.com/bliki/ValueObject.html
[74] https://projectlombok.org

Dan North[75] seems to agree on Twitter, talking to Liz Kheogh[76] and Jeff Sussna[77]:

> I like having a product and/or team blog. Journalling decisions and conversations as you have them documents history. It also shows how decisions got made, and lets you see changing tastes or learnings over time.

# Fractal Polyglot Architecture

Considering a large system, we should give up on the idea of one single uniform architecture everywhere. A system is made of several sub-systems, and each should have its own architecture, plus the overall architecture of how they're inter-related.

Therefore: **Consider your system as several smaller sub-systems, or "modules". They may be physical units, as components or services, or just logical modules at compile time. Document the architecture independently for each module, and describe the over-arching architecture between the modules as one system-level architecture too.**

Typically you would document the architecture of each module with Internal Documentation, using a combination of package naming conventions, annotations in the source code, and a little plain text. You would document the overall architecture with more Evergreen Documents in plain text, and perhaps some specific DSL if you have one that fits. However the documentation of the overall architecture could also use some generated documents built by consolidation of the knowledge extracted from each module.

# Documentation Landscape

Ready-made architecture document templates may help, if you like them:

- Arc42
- IBM/Rational RUP
- Company-specific set of templates, as a Documentation landscape

Some templates try to plan for every possible architectural documentation need. I totally hate having to laboriously fill large templates.

[75] https://twitter.com/tastapod
[76] https://twitter.com/lunivore
[77] https://twitter.com/jeffsussna

**LOL**

I've spent one week working on a Software Architecture Document, friendly called SAD. No acronym would be more appropriate. From @weppos[a] on Twitter

[a] https://twitter.com/weppos

Another positive side of templates is as extensive checklists. For example, the ARC 42 "Concepts" section is a nice checklist to find out what you may've forgotten to consider, as shown below (the list is shortened from the original template[78])

- Ergonomics
- Transaction Processing
- Session Handling
- Security
- Safety
- Communication and Integration
- Plausibility and Validity Checks
- Exception/Error Handling
- System Management and Administration
- Logging, Tracing
- Configurability
- Parallelization and Threading
- Internationalization
- Migration

[78] http://www.arc42.org/

- Scaling, Clustering
- High Availability
- (…)

How many of these aspects do you neglect in your current project? How many of them do you neglect to document?

Draw inspiration from all these established formalisms to derive your own documentation landscape, on a *module by module* basis. According to Stake-Driven Architecture Documentation, focus each documentation landscape on what matters most for the stakes of this sub-system.

On a module with a rich business domain, you would focus primarily on the domain model and its behaviors as key scenarios. On a module more CRUDdy, there may be very little to say, as everything is probably standard and obvious. On a legacy system, the testability and migration may be the most challenging aspects, which would probably deserve the documentation.

Your documentation landscape can be a plain text file with predefined bullets and tables, but it can take the form of a small annotations library, to directly mark the source code elements with their architectural contribution and rationale. It could be a specific DSL. In practice you would mix all these ideas according to what works best. You may even use a Wiki, or even proprietary tools which would instantly solve all your problems…

A typical documentation landscape for a system would have to at least describe the following points:

1. The overall purpose of the system, its context, users and stakeholders
2. The overall required quality attributes
3. The key business behaviors, and business rules and business vocabulary
4. The overall principles, architecture, technical style and any opinionated decision or parti

This does not mean at all that you need to create documents with all that. Living documentation is all about reducing the need for manually written documents, thanks to alternatives which are cheaper and remain up-to-date.

For example, we could use plain text Evergreen Documents for the first point, system-level acceptance tests for the point 2, a BDD approach with automation for point 3, and a mix of a README, a Codex and custom annotations in the source code for point 4.

# Architecture Diagrams & Notations

Many authors have proposed formalisms to describe software architecture since a long time. A number of standards is available, like IEEE 1471 “Recommended Practice for Architecture Description of Software-Intensive Systems”, ISO/IEC/IEEE 42010 “Systems and software engineering — Architecture description”, while the Philippe Kruchten’s “4+1 model” has gained recognition

in the enterprise world. However all these approaches are not precisely *ligthtweigh* and require some learning curve to be understood. The provide a set of "views" to describe different aspects of the software system, with a logical view, a physical view etc. Overall, these approaches are not particularly popular among developers.

Simon Brown acknowledged this need and consequently proposed the C4 Model[79], a lightweight approach to architecture diagrams which is becoming increasingly popular among developers. This approach draws in particular from the work of Eoin Woods and Nick Rozanski in their book Software Systems Architecture[80], and has the benefit of being usable without prior training. It suggests 4 simple types of diagrams to describe a software architecture:

- **System Context Diagram**: A starting point for diagramming and documenting a software system, allowing you to step back and look at the big picture
- **Container Diagram**: To illustrate the high-level technology choices, showing web application, desktop application, mobile app, database, file system
- **Components Diagram**: A way to zoom into a container, by decomposing in a way that makes sense to you: services, subsystems, layers, workflows etc.
- **Classes Diagrams**: (Optional) To illustrate any specific implementation detail with one or more UML class diagram(s).

My favorite is the Context Diagram, both simple, obvious but so often neglected.

# Architecture Codex

When describing a solution to people, probably the most critical part is to share the thinking and reasoning which led to the solution.

Rebecca Wirfs-Brock was at the ITAKE un-conference in Bucharest in 2012, and during her talk and the later conversations we had about it, she gave the example of EcmaScript, where the thinking process is clearly documented:

Here are from my notes on the topic some of the rationale for decisions in EcmaScript:

- "Invoke similarities with other existing folklore"
- "Usually we want to learn and understand as little as possible to do the job."
- "Recipe for making change: figure out how similar change has been done before."

Later I have been doing department-wide architecture in a bank, and I introduced this idea of a Codex of principles guiding all the architecture-sensitive decisions. The Codex was built from the accumulation of concrete cases of decision-making, by trying to elucidate formally the reasoning

[79] http://www.codingthearchitecture.com/2014/08/24/c4_model_poster.html
[80] http://www.viewpoints-and-perspectives.info

behind the decision. Often, the principle was already in the head of other senior architects, but it was tacit and nobody else knew about it.

**The all-mighty Codex**

Some of these principles were like the following:

- “Know your golden source” (aka Single Source of Truth)
- “Don’t feed the monster”: improving the legacy only helps last it longer
- Increase the Straight-Through Processing Automation Ratio
- Customer convenience first
- API-First!
- “Manual process is just a special case of an electronic process”

This codex proved useful for everybody involved in architecture. The goal was to publish it for everyone, even if it was incomplete and not always easy to understand. At least it was useful to provoke questions and reactions. It was never formally published as far as I know, however its content leaked on many occasions and has been used several times for more consistent decision-making.

One recent consulting gigs I found it helpful again to express the value referential of the team as a list of preferences, like:

- “Code over XML”
- “Templating engine is ok but keep the logic out”

Of course it is a good idea to adopt standard principles already documented in the literature too, as a kind of READY MADE DOCUMENTATION. For example:

- “Keep your middleware dumb, and keep the smarts in the endpoints.” by Sam Newman

It is very important to keep this codex as a working document, never finished. Whenever we hit a contraction in its principles, then it’s time to fix it or evolve it. This did happen, and this should not be seen as a failure, but as an opportunity for collective decision-making to be even more relevant. After all, architecture is a kind of a consensus thing, isn’t it?

An architecture codex can be just a text file in the source control, a set of slides, and it can even be expressed in code. The following is an example of using a simple enum to materialize the principles of the codex:

```java
/**
 * The list of all principles the team agrees on.
 */
public enum Codex {

        /** We have no clue to explain this decision */
        NO_CLUE("Nobody"),

        /** There must be only one authoritative place for each piece of data. */
        SINGLE_GOLDEN_SOURCE("Team"),

        /** Keep your middleware dumb, and keep the smarts in the endpoints. */
        DUMP_MIDDLEWARE("Sam Newman");

        private final String author;

        private Codex(String author) {
                this.author = author;
        }
}
```

Sam Newman in his book "Building Microservices" mentions his colleague Evan Bottcher created a big poster on the wall displaying the key principles visibly on the wall, organized in three columns from left to right:

## Principles on a poster on the wall

- Strategic Goals, like "Enable Scalable Business", "Support Entry into new Markets"
- Architectural Principles, like "Consistent Interfaces and Data Flow", "No Silver Bullet".
- Design & Delivery Practices, like "Standard REST/http", "Encapsulate Legacy", or "Minimal Customization of COTS/SAAS".

That's a nice way to sum up the system vision, principles and practices in one place!

# Transparent Architecture

When architecture documentation becomes embedded within the software artifacts themselves in each source code repository, with Living Diagrams and Living Documents generated out of them automatically, everyone gets access to all the architecture knowledge by themselves. Contrast that

with companies where the architecture knowledge remains in tools and slide-decks only known by the official architects, and not up-to-date.

One consequence is that this enables decentralizing the architecture and the decision making dependent on architecture knowledge. I call that "transparent architecture": if everyone can see the quality of the architecture by themselves, then they can take decisions accordingly, by themselves, without necessarily asking the people in an architect role.

**With access to the whole picture, each team can directly take decisions consistent with respect to the whole system**

For example, in a microservices architecture, a transparent architecture will make use of Living System Diagrams generated out of the working system at runtime. This knowledge is already there in the distributed tracing infrastructure (e.g. Zipkin). You may have to augment it a bit with custom annotations and binary annotations added in your instrumentation.

You may as well rely on your Service Registry (e.g. Consul, Eureka…) and its tags to produce Living Documents. Dependencies between services can also be derived from the Consumer-Driven Contracts, if you apply this practice. And if you care about the physical infrastructure, it can be made visible through custom Living Diagrams generated with Graphviz from data you get from your Cloud through its programmatic API. Note that more "virtuous" practices also makes living documentation easier!

**Living Diagram of a Cloud infrastructure generated from cron, python, boto, pydot, graphviz - from James Lewis slides**

*All this is fine, but we can go even further with Test-Driven Architecture.*

# Test-Driven Architecture

Test-Driven Development brings a mindset which is not just for writing code "in the small". It's a discipline to first describe what we want, before we implement it, at which point we make it clean to help our work in the longer term.

We can try to follow this same process at the architecture scale. The challenges we face are the larger scale of everything, which may not fit in our heads, and the longer feedback loops, which means we may forget what we were after when can eventually get the feedback.

Ideally, we would start by defining the desired quality attributes as tests. They will not pass for weeks or months, until they eventually pass, at which point they become the only really sincere documentation of the current quality attributes.

For example, consider a performance quality attribute:

> "10k requests over 5mn with less than 0.1% error and response time within 100ms at 99.5 percentile"

First write it down in the bullet list of quality attributes, for example in the Markdown file.

Then implement this criteria as literally as possible as a Gatlin or JMeter test on a realistic environment (perhaps even on production). It's not every likely that it passes right away.

Now the team can work on it, among other things depending on the priorities. It may takes a few sprints to make it pass.

The first time I mentioned that at a Socrates conference in Germany, the comment was:

> We already do that indeed, as test scripts for proofs of concepts. Except we throw them away after.

Perhaps it does't takes that much more effort to turn experiments you're already doing on a one-off basis into maintainable assets that can assert you still meet the requirements and that can document them at the same time.

## Literal quality attributes as scenarios

The test should describe the quality attribute as as declaratively as possible. One way to do that is to dress the criterion as a special Cucumber scenario:

> @QualityAttribute @SlowTest @ProductionLikeEnv @Pending
>
> Scenario: Number of requests at peak time
>
> Given the system is deployed on a production-like environment
>
> When it receives a load of 10k requests over 5mn Then the error rate is less than 0.1%
>
> And the response time is below 100ms at 99.5 percentile

Note the custom tags:

- @QualityAttribute to classify it as a Quality Attribute requirement
- @SlowTest to launch it only as part of the nightly slow tests run
- @ProductionLikeEnv to flag that this test is only relevant on a production-like environment for the metrics to be meaningful
- @Pending to signal that this scenario is not passing yet

With this approach, as soon as the scenario is written, it can become the Single Source of Truth for the corresponding Quality Attribute. Moreover, the Scenario Tests Reports becomes the Table of Content for these "Non-Functional Requirements" too.

Note that the quality attributes scenarios are useful even if they are never actually implemented as true tests.

You may describe all the quality attributes this way:

- **Persistence**: "Given a purchase has been written, when we shutdown then restart the service, then the purchase we can read all the purchase data". Is it going to far documenting the obvious?

- **Security**: "When we run *standard* penetration testing suites, then zero flaw is detected". Note that here the trick is the word "Standard" which refers to a more complete description somewhere outside of the scenario. This external link is part of your documentation too, even if you didn't write it yourself.

When the Quality Attribute can be checked at compile time, it will probably be part of your quality dashboard, for example in Sonar. In this case you can turn this tool into your table of content of these quality attributes. And you may use something like the *Build Breaker* plugin to fail the build in case of too many violations.

(This is another way of implementing Enforced Guidelines).

## Quality Attributes at runtime in production - Netflix Simian Army

There are quality attributes which are too difficult to test outside of their natural habitat. This calls for a more monitoring-oriented approach. Netflix introduced the Chaos Monkey to assert the fault-tolerance at the service-level. Later they introduced the Chaos Gorilla at the data center level:

> Chaos Gorilla is similar to Chaos Monkey, but simulates an outage of an entire Amazon availability zone. We want to verify that our services automatically re-balance to the functional availability zones without user-visible impact or manual intervention. (from the Netflix techblog[81])

The mere description of these two Chaos engines, along with their configuration parameters in terms of outage frequencies, is in itself a documentation of the fault-tolerance requirements.

Some cloud providers or container orchestration tools support automatic rollback if some metrics are degraded following a deployment. This configuration de facto documents what's considered "normal" metrics: CPU / memory usage, conversion rate etc.

## Other quality attributes

Some quality attributes cannot be tested automatically: financials expectations, user satisfaction, etc. They often reside within spreadsheets on shared drives. Alternatives exist online, to encourage sincere declarations of the objectives, before comparing them against the actual achievements.

> To keep track of your expectations before doing experiments on the product, its successes and failures: http://growth.founders.as #startup #hypotheses – @fchabanois on Twitter

This kind of tools encourage working in a TDD-ish fashion for startup objectives.

[81] http://techblog.netflix.com/2011/07/netflix-simian-army.html

## From fragmented knowledge to usable documentation

This approach can end up with many fragmented and heterogenous sources of truths about all the quality attributes. They need to be curated and consolidated into one or two Living Table of Content.

**Therefore: Dress your quality attribute tests as Cucumber scenarios, into a separate "Quality Attributes" folder (hence into a separate chapter in the corresponding Living Documentation). Use tags to classify them more precisely. Decide on one existing tool to host the main table of content as the single entry point for all the quality attributes documentation, with references to any other tool.**

For example you may decide that Cucumber is the main table of contents. You'll add pseudo scenarios to link to the Sonar configuration and to the *permalinks* to the configuration of each static analysis tools. (Permalinks are links which are permanent). You'll also mention the Chaos Monkey as a scenario and a link to its configuration on some Git repository.

Or you decide on your build tool as the main table of content. By adding custom steps in the build pipeline (e.g. Jenkins or Visual Studio) you can pinpoint to Cucumber reports, Sonar reports, and to the Chaos Monkey configuration.

These tools can at the same time be a table of content and fail the build in case one of the quality attribute is not met anymore. This helps keep the documentation sincere. If you just use a wiki as the main table of content, you no longer have that enforcement.

# Small-Scale Simulation

Large and complex software applications or system of applications are a challenge to documentation. judging by the size of the source code and configuration they're made of, the amount of knowledge that is necessary to describe them is so huge it is useless as-is. And at the same time the critical higher-level design decisions and all the thinking that went into building it are often implicit.

If it was smaller, it would be easier to understand. Just by reading the handful of classes, running the few tests, exploring what happens when playing with the code in a REPL and watching the dynamic behavior at runtime, we would quickly get a sound understanding of its purpose and of how it works. Even if the thinking that led to it was lost, we would be able to recover it from observing the small-scale system in action. This would perhaps be tacit knowledge, but it would still be much better than nothing.

**Therefore: Create a small-scale replica of your software system, as a a stripped-down reimplementation of just the key bits of code with some tests, for the sole purpose of documentation. Through aggressive curation, select a small subset of features and code that focus only on the one or two aspects that matter and that fits on your brain as a whole. Simplify every other distracting aspect, even if it makes it slower and with a limited set of features. Make sure this small-scale replica works realistically, producing accurate, if not perfect, results, but not necessarily in all cases.**

The advantage of small-scale simulation is that it becomes human-scale, it fits in your head. Note that here we're talking small-scale primarily about reduced complexity, not just reduced size.

I've tried this idea several times.

When developing an exchange system for financial products, the core of the matching engine was growing larger and more complex because of various optimizations, with other concerns like timing, scheduling and permissions management blurring the overall picture. We've created a smaller version of the core of the matching engine, with just the minimal set of basic and naive classes that can make a matching work in its most interesting aspects. In this case, the smaller-case system was not a replica but was built mostly from the same elements as the actual system, because the design was flexible enough to accommodate that.

In a very large legacy system with a few applications and many batches running in the background several times a day, the overall behavior of the system was quite nebulous. We've created a Java small-scale, simplified emulation of the most important batch so that we could better understand it and explore its interactions with our new code.

In two startups with rich domain, we paired a few days to create a small-scale model, working for just one very simplified case. However, it was the opportunity to quickly get to explore and map the domain, discover the main issues and stakes, grow a vocabulary, and agree on a shared vision

of the whole system. From that small-scale system, later discussions had a concrete reference code to ground the discussions, it's really a communication tool you can point at during conversations.

At a big company where everything takes ages, creating a Small-Scale Model under the name of a "Proof of Concept" is a great alternative to never-ending studies delivering nothing but slides and illusions. The focus on working code helps converge and makes it harder to elude the tough questions. You probably already build Proofs of Concepts at the beginning. But do you keep them for their explanatory power for later?

## The desirable properties of a small-scale simulation

A small-scale simulation must be:

- **Small enough to fit in the brain** of a normal person, or of a developer. This is the most important property, and it implies that the simulation will not account for everything of the original system
- **Tinker-able and inviting for interactive exploration**. The code should be easy to run partially, by just taking a class of function and being able to do something with it without having to rebuild the full simulation.
- **Executable to exhibit the Dynamic behavior** at runtime. The simulation must predict results through its execution, and we must be able to observe it easily, even during the computation if possible, in debug mode, with traces or just by running its phases independently.

A small-scale software project that is executable and works in a realistic fashion is a valuable for reasoning on the system. You can reason on its static structure, just by observing it in the code. You can also tinker with it by creating one more test case, or by interacting with it in a REPL.

This approach is also useful also as a cheaper proxy to impractical legacy or external systems; instead of running a complex batch that depends on the state of the database and that has numerous side-effects everywhere, you can run its emulation and have a grasp of its effect in relation with what you're doing.

## Techniques to simplify the system

The key thing is to simplify aggressively, with an exclusive focus on the one or two aspects that matter. Just like every documentation, it should explain one thing well, rather than explain 10 things badly (there's already the real system for that). Note that you can still decide to build more than one small-scale simulation, one for each important point to explain.

All this means that the small-scale system will lose a lot of details, and will *NOT* show a lot of otherwise valuable knowledge. This is harder to do that it looks, because when you're attached to

a system you've built you'd like to tell about all its interesting facets, but you have to refrain from doing so and learn to focus (something I have a hard time doing when writing this book).

Interestingly, the techniques to build a small-scale simulation are the techniques you probably already use to create convenient tests.

Concretely, we can simplify a system in many ways, always by deciding to ignore one or many distracting concerns:

- **Curation**. Give up the idea that it has to be Feature-Complete. Get rid of all the member data that are not central to the current focus. Ignore side-stories and secondary stuff like special cases that don't intersect the current focus
- **Mocks**, **stubs and spies**. Give up performing all the computations. Instead, use the usual test companions to totally get rid of all the non-centrally relevant sub-parts. Use in-memory collections instead of middleware, and simulate third parties.
- **Approximation**. Give up on strict accuracy and settle only on realistic accuracy, that looks good enough, like the right value without the digits, or 1% correct.
- **More Convenient Units**. Give up the ability to really put in production the simulation with the actual data. Instead if the dates are only used to decide if something happens before or after a given data, you may replace the dates that are cumbersome to manipulate by hand with plain integers.
- **Brute Force calculation**. Give up the optimizations that are not central to your current focus. Instead, make it work using the algorithm that's the simplest to grasp, the one with the most explanatory power.
- **Batch vs**. **Event-Driven**. Turn the original event-driven approach into a batch mode, or the other way round, if it's simpler to code and understand, assuming it's not central to the current focus.

# Building the small-scale simulation is half the fun

You learn a lot by creating a small-scale simulation. You have to clarify your thoughts, and nothing's more demanding that simple, working code to force that.

From a design perspective, cutting through the details to focus on the essentials gives a lot insights to improve the design of the original system in turn. For example if you can replace dates with integers in the simulation, perhaps the original functions don't really need to operate on dates, but on just anything comparable.

If the simulation can work without all these distracting aspects, this also means the original design should follow the Single Responsibility Principle and therefore separate all the concerns. By the way, you know when you've reached that state when you can create your small-scale simulation by reusing the same code of the original system, just by assembling a a more naive subset of its elements.

This idea used in the context of starting a project is known under various names in the literature: Alistair Cockburn talks about a Walking Skeleton[82], Pragmatic Programmers Dave Thomas and Andy Hunt talk about Tracer Bullets, while similar ideas have been documented since 1975 and apparently even since the 50's.

It's also similar in many aspects to the pattern *Breakable Toys* described in the book *Software Craftsmanship Apprenticeship Patterns* by Dave Hoover and Adewale Oshineye. A small-scale simulation can be used to try things much faster than the actual system. This comes handy to perhaps try two or three competing approaches quickly to decide on the best based on actual facts rather than on opinions.

Such a tinkerable system is very valuable so that new joiners can build their own mental model about it. If as Peter Naur claims it's very hard to express a theory using codified rules like text, having the ability to form your own theories about a system by just playing with it without risk can help. That's how kids learn about all the laws of physics indeed.

[82] http://alistair.cockburn.us/Walking+skeleton

# Part 11 Efficient Documentation

# Efficient Documentation

> The most common myth of communication is that it happened – @ixhd from Twitter

There are many little techniques to communicate more efficiently. The overal goal is to get the message through with less words, faster, more accurately, without wasting people's time.

This is about all communication, not just documentation. It's as useful to describe requirements or in training on anything.

## Focus on Differences

When describing a specific thing, e.g. a dog, focus on its differences versus the generic thing, e.g. a mammal. The generic thing must be well-known, or well-described preliminary. This enables to describe a rich thing with just a few points, one for each significant difference.

> The precise keyword here is *Salience*, which means "most noticeable or important". We primarily want to describe the salient points out of the mass of information.

### How is Your lemon?

During his training on BDD, Joe Rainsberger sometimes tells a great story about lemons. I don't remember the exact story but I do remember the key insight, so here's my own totally distorted account of the story:

> The trainer asked everyone to describe how a lemon's like. The group described the typical lemon-shape, yellow color, acid taste and grained texture of a lemon. The trainer then gave them a real lemon, one for each attendee, and asked them to study carefully their lemon for a few minutes.
>
> As an attendee, he analyzed his own lemon. One end of the lemon was bent in a weird way. There was a variation of color somewhere in the middle. The lemon was kind of small compared to the average lemon.
>
> He then asked everyone put their lemon back together into the basket, and then asked them to recognize it among all lemons. This was surprisingly easy! Each attendee realized they got to learn their own lemon very intimately. "It's my lemon!", they all said! They even had felt a bit of attachment to their lemon.

By looking carefully at a specific lemon *in contrast to the generic concept* of a lemon that everybody knows, you can describe it very effectively. It's at the same time precise with lots of details, and efficient because you don't have to describe the generic thing.

I've seen colleagues use that technique to describe concepts from the business domain. For example, during a presentation to new joiners on financial asset classes, the trainer was only mentioning the 5-7 bullet points that were distinctive to a particular asset class like commodities, in contrast to a generic well-known asset class like equities.

> In the Power market (electricity), a specificity is that the prices are very seasonal during the day and during the year, in contrast to company stocks. In the Oil market, geography matters, you don't ship oil anywhere.

# Only tell what's unknown

There is no point explaining something to people who already know it. The key is to identify what your audience knows. During conversations it is possible to assert what's the people you're talking to know or don't know, from their questions, body language, and by asking them directly. In written form this is more difficult, but not impossible. There are several ways to do that.

## Segment by known audience

For each audience, get feedback from the most frequently asked questions and by talking to the support team to know better what's well-known and what needs to be explained more. Then focus on what's unknown for each audience.

## Flexible content

Organize the written content so that it can be skimmed, skipped, and read partially. Clearly mark optional sections. Make the titles informative enough so that readers can decide if this is what they are after.

For example, Martin Fowler suggests writing Duplex Books[83]. The idea is to split the content into two big parts: the first part is a narrative designed to be read "cover to cover", while the second part is a reference material, not meant to be read cover to cover. You read the first part to get an overall understanding of the topic, and you keep the rest for when you actually get to need it.

[83] http://www.martinfowler.com/bliki/DuplexBook.html

## Low-Fidelity Content

Too often a diagram that was meant to brainstorm, explore, propose ideas is misunderstood as a piece of specification. This results in premature feedback on details like "I'd prefer another color", even though the whole thing will change a lot in the next hours or days. This situation is especially true for everything done on a computer, since it is quick and easy to create nice-looking documents, pictures and diagrams using the proper piece of software.

**Therefore: When the knowledge is still being shaped, make it clear in the documents thanks to low-fidelity content like wireframe and sketchy rendering.**

As @kearnsey said at #agile2014 (reported by @OlafLewitz):

> Use low fidelity representation for output as long as you want people feel invited to add their input.

## Visual Facilitation

"I'm talking about *that*" when finger pointing a box on a digram on the whiteboard or on a screen is much more concise and precise than "I'm talking about this thing that takes care of filtering the duplicate entries upstream of the realtime secondary calculation engine". As Rinat Abdulin said on Twitter on a conversation we had about living diagrams, "Stuff 'you can point to' during discussions helps communicate faster and with better accuracy. Having conversations supported by visual media is a powerful technique.

During meetings or an open-space session, the visual notes on the flip-chart not just report on what has been said, they also *influence* the further discussions just by being in front of everyone's eyes.

This influence is even stronger if the scribe having the marker on the whiteboard is skilled in visual facilitation. He or she rearranges the way the information is organized, sorting concepts, using a meaningful layout, noting links, side-remarks and drawing little decorations about the connotations involved in the discussions.

**Therefore: Don't discount the importance of visual supports during discussions. Invest in some visual facilitation skills, and learn to exploit how this can help shape the dynamics of the work.**

Visual notes are redundant with what was said and therefore help if you did not catch a word or an idea immediately. They help as a way to catch up after a quick day dreaming, for everyone to remain involved in the session. When done well, visual facilitation is also an opportunity to make people smile thanks to some visual humor.

# Search-Friendly Documentation

Making information available is not enough. You have to know where to find what you need when you need it, and it's a matter of being easily searchable.

Being easily searchable is first of all a matter of using distinctive words.

> For example, “GO” as a name for a programming language, from a company like Google who is into search, is totally not search friendly. To make it search-friendly again it has to be actually named “golang”.

Then the piece of knowledge should mention clearly the user needs it addresses, since this is the most likely question that will be searched for. To help on this, keywords should be added, including words that don’t really occur in the actual content but that are likely to be used when searching for it. Use the words from actual users, found from the analytics of failed searches etc.

Remember to mention synonyms, antonyms, and faux-sens and common confusions, for improved discoverability by search.

All this is usually considered only for written text in a traditional document, however this applies just as well in the code, considered as text too. If you use annotations, you may try to add a `keywords = {"insurance policy", "home insurance", "cover"}` to ease full text search on the code.

# Concrete Examples, Together, Now

*Make sure to have every attendee agree on concrete examples when discussing specifications.*

This probably sounds familiar:

> Now that we're in agreement on this change, we can stop this meeting. You will work on test cases, detailed design and screen mockup that we'll discuss next week. In the meantime, feel free to ask if you have any questions.

The lost opportunity here is that everyone involved will most likely waste time after the meeting. The collective agreement during the meeting is often an illusion. As the saying goes: "The devil hides in the details". It's only when starting to create the mockup for the new screen that the issues will really start to jump out of the box. It's only when trying to code the *abstract* requirement that the misinterpretation will happen, and it will be only detected days or weeks later.

A better approach is to reply with this unorthodox proposition:

> Why about creating a concrete example together, like right now, during this meeting?

A similar key question I use often is to say:

> I believe we're all in agreement on what needs to be done. But to be 100% sure, just in case, we should take a few minutes to draft a concrete example all together right now.

It may sound like a waste of time right now. "We don't have time for the low-level details here" is sometime the objection. And it's true that it looks slow to observe your colleague slowly performing the collage of buttons and panels on a screenshot in MS PowerPoint on the big video screen. However at the same time you're saving much more time in decision-making, because everybody is there to confirm, adjust, or raise an alarm.

**Therefore: Whenever there's a workshop on specifications, make sure to have every attendee agree on concrete examples during the meetings, during the session right now. Resist the temptation to save time by doing it offline. Acknowledge that decision-making is often the main bottleneck, in contrast to drafting concrete examples. Some of the resulting examples will be an important part of your documentation.**

It does not matter if the examples are scenarios expressed in text, data tables, flip-chart sketching, or visual screen mockups in a tool projected on a big screen, or whatever else. What matters is that everyone involved understands the examples so that they can immediately notice there's something wrong in them. For that purpose it's also essential for examples to be concrete. Don't settle on abstract agreement. Everybody agrees that "the discount is 50%"", but how do we do when the price is $1.55? How do we take care of the rounding? You need the concrete examples to notice that.

# In practice

There are many common objections you'll hear the first time you'll try to request creating concrete examples during the meeting. Concrete looks verbose and slow, whereas abstract looks concise and fast. They are in the very short term, but in the longer term in the context of specifications, it's rather the opposite: concrete is faster.

In fact you're painfully aware of that, so you may be the first to suggest to do the examples offline: "I don't want to waste your time, tell me how to do it, and I'll do it later on my own".

Instead, consider the following sentence: "Sorry you'll have to wait for 3mn while I fire the tool, but then we know for sure we're in agreement on the solution. This way we can avoid a ping-pong of emails and further meetings in the coming days and weeks."

So when it comes to specifications, where communication is particularly fallible, keep in mind these Do & Don't:

- **DON'T** "We can stop there to save time, I'll go on alone then we'll have another meeting to discuss the results"
- **DO** "Let's go as far and as quickly as we can together, so that we know quickly what are the troubles and where we may disagree"

# Fast Media and prior preparation

Of course it helps to chose fast media to create the consensus on concrete examples: - Flip-chart or whiteboard and markers, writing carefully so that everybody can read - Big pages of paper on the table, with pens - People talking and a spokeperson taking verbatim notes, restating the notes regularly to other attendees - A simple text editor on a big screen - A screen mockup tool that you know well so that you're fast with it - MS Powerpoint to do a collage of pre-selected screenshots, again if you're comfortable with the solution.

It helps to come with some prior preparation, with some ready-made materials. I've seen a colleague who was carrying a big slip full of screenshots from all the important screens, printed diagrams of the main workflows of the application in every meeting, just to improve communication during discussions, even spontaneous ones. I've done a similar approach but electronically, keeping a default Powerpoint full of screenshots and other stuff, just in case it would be needed during specification workshops to answer a question or be reused as part of a screen collage.

The same idea can be generalized to other aspects of the specifications, like its quality attributes. When discussing performances, latency, fault-tolerance requirements, it would be a good idea to not only define the expectations, but also go the extra mile and collectively agree on the acceptance criteria. This acceptance criteria should then become a test that will not only document but also enforce that the quality attributes are actually met.

# Together, Now

The power of "Together, Now" suggests going the extra mile after an agreement until all attendees agree to a solution proven by concrete examples: UI mockup, interaction workflow, impact map, scenarios of the expected business behavior, as text or sketches with accurate numbers on it, etc.

Productive specification meetings that really produce concrete examples are valuable. They rely on face-to-face conversations for effective communication, while producing quality documentation as an outcome.

The canonical example is of course the **Specification Workshops** where the 3+ amigos define the key scenarios (Gojko Adzic).

There are many similar examples of interactive collaborative creation of concrete results in the literature on agile software development:

- **Mob-programming**: "all the brilliant minds together, on the same task, on the same machine"
- **CRC Cards**, a technique for instant, interactive and collaborative modeling with CRC cards on a table (Ward Cunningham and Kent Beck)
- **Modeling with stickers** on the wall, as in Model Storming (Ambler) and Event Storming (Alberto Brandolini)
- **Codeanalysis**: modeling directly in code in a programming language during the meeting with the domain expert (Greg Young)

# StackOverflow Documentation

*Don't write all the documentation, let people on SO do it for you.*

Several times I heard colleagues or even candidates tell that SO is by far the best place to go for documentation, and my experience tends to corroborate this. Official documentation pages are often boring and seldom task-oriented. The funny thing is that people answering on SO often had to use the official documentation pages to build their own knowledge, together with trial and errors or even by having to read the source code sometime.

People answer questions very quickly on SO. It's another form of *living* documentation: contribute a question, then people all over the world will quickly contribute answers, making the documentation a really living thing.

**Therefore: When the topic is popular enough, let SO provide a good task-oriented documentation on top of your reference documentation you provided. Let your teams post questions on SO, and let them answer other people's questions as well.**

This requires your project to be published online, usually with its source code. It especially requires the project to be successful with enough demand to attract contributors.

Or you can keep it internal and closed source and use an equivalent on-premises Stack Overflow clones[84]. However a domestic Stack Overflow clone may probably miss the scale to work as efficiently as the true worldwide site.

One downside with Stack Overflow is that if your product is crap it will show. However you can't prevent that happening on the web, unless you make the product better of course. You may also dedicate many employees to answer questions in a positive way to improve a bit the user experience too.

[84] http://meta.stackexchange.com/questions/2267/stack-exchange-clones

# Affordable & Attractive

> We can make information available, but we cannot make people care for it. Maybe journalism as a solution? My Arolla colleague Romeu Moura

> Documentation should be attractive for the same reason flowers are attractive: for self-preservation. (paraphrasing Romeu Moura again)

## Specs Digest

*Small is Beautiful*

I've seen a project where the team decided to curate all the accumulation of design and specifications documents into a much shorter (about 10 pages-long) "Specs Digest" document. This was mostly done by copy-pasting the best parts out of various existing documents, updated, fixed and supplemented with the obviously missing bits in the process. This digest was a highly-valued document in the team.

The digest is strongly organized into sections, each typically half-page long, with clear titles recaped in a table of contents. The structure is meant so that you can skip any section safely to jump directly to the part of interest.

The content mostly focuses on everything not obvious: business calculations (dates, eligibility, financial and risk calculations), principles and rules. But it may also describes key semi-technical aspects like the versioning scheme between multiple related concepts.

Note that if you already have a Living Documentation based on scenarios in a Cucumber-ish kind of tool, you should move all this content into the feature files themselves, or in side-car "preambule" markdown files in the same folders.

## Easter Eggs & Fun Anecdotes

*Having fun is the best way to learn*

Turn any kind of documentation into something more engaging by hiding anecdotes from the project, its sponsors and team members into the document, as a way to encourage and reading. Add simple illustrations.

As Peter Hilton mentions in his talk on Documentation Avoidance[85]:

> Use humour. There's no rule that says that jokes aren't allowed. Insufficiently serious documentation is probably not your biggest problem. Staying awake might be.

[85] https://www.slideshare.net/pirhilton/documentation-avoidance

# Promoting News

Adding knowledge somewhere is not enough for its audience to notice and use it. Provide ways to promote the documentation, especially when it changes:

- "Recent Changes" page, the bare minimum
- Changes feed (Swagger…) pushed to Slack
- Slackbot custom replies, to remind where's the doc is in response to keywords
- Release notes: do you really read them?
- Journalism by a real person: when you're really serious about knowledge sharing, hire a real professional journalist on-site!

This paragraph is too short for this very important topic. Unfortunately it's so hard I don't have miracle solutions here…

# Unorthodox Media

The corporate world tends to be unoriginal. When it comes to documentation, the traditional media remain the Mighty Email, MS Office with the boring mandatory templates, SharePoint, and all the various Enterprise tools notorious for their outstanding User Experience.

**But life does not have to be so dull. Shake up your team or department by using unexpected Unorthodox Media for your communication and your documentation purposes!**

Below are various ideas to use as inspirations to spice up your communication in general, and which can be useful to share knowledge and objectives.

## Maxims

When your current initiative is to improve the code quality:

> Fix a bug? Add a test.

Or

> Fight Legacy code Write unit tests

Don't directly copy and paste these maxims. Create yours that will stick in your culture. The only way to know if a maxim stick is to tell it out loud on different occasions, to see if you resonates and if anyone mentions it later.

> You may read the book *Made to Stick: Why Some Ideas Survive and Others Die* by Chip Heath and Dan Heath on this topic.

Good maxims are useful and amusing at the same time:

> If in doubt, do it the way Erlang does it. (from by @BeRewt on Twitter)

Once you have a maxim, your job is to repeat it as often as possible (without becoming a spammer of course).

Repetition also works inside a maxim. For example "Mutable state kills. Kill mutable state!" has internal repetition which can help make it more memorable.

A maxim has to remain trivially simple, because complicated stuff does not scale over larger audiences. You want to broadcast your maxims. Therefore, be ready to trade nuance for stickiness. You can only tell one or two key messages. Make sure these are the most important messages. Take care of the other less important messages in a different way.

**Pro Tip** Statements that rhyme are more believable than those that don't. This is the rhyme-as-reason effect[86] (or Eaton-Rosen phenomenon)

## Posters & Domestic Ads

Consider your communication like a marketing campaign. You can use the same tools, internally.

Once you have maxims, you can turn them into illustrated posters.

The first thing you can do is start with an image search. For example consider this maxim:

> The only way to go fast is to go well!

Searching on Google Image shows a ready-made image meme with this exact text and with Uncle Bob picture. This is no surprise considering that he like to repeat this maxim. And by the way this maxim also exhibits internal repetition and symmetry around the word 'go', which makes it more sticky.

[86] https://en.wikipedia.org/wiki/Rhyme-as-reason_effect

**Robert C. Martin: The only way to go fast is to go well!**

## Meme-based posters

Now consider another you don't have your maxim yet but you'd like everyone to remember to close the door of the bathroom after use. Let's create a motivational poster for that!

That's easy with all the available free online *meme generators*. From a given idea you can browse the most common memes until you find one that fits the message best. Here we found a "Mr T" meme (this example is a real one I've seen at a customer site. The poster was awesome on the bathroom door.)

**Mr T. Picture: Are you awesome? Close the door once you're done.**

One drawback of memes is that they tend to become annoying when used too frequently.

**Pro Tip** Display Cute Kitten along or between your messages. Everybody loves cute kitten!

## Information Radiators

Posters don't necessarily have to be printed and pinned on the walls or windows to be visible. Some companies have TV sets on the walls or in the lifts with a carousel of slides for internal communication. This is a nice place for your posters.

The downside is that you probably have to go through an acceptance process, and you may be rejected.

Still, you can insert your posters as banners into your build walls, screen savers, or pair-programming blocker screen!

## Very Short Story, Humor, Cheap media, and implicit message

You may have seen this very cheap yet quite efficient poster already:

Storytelling is very powerful, even when this short. It takes training or pure luck to author this kind of gem. Fortunately you can reuse (steal or hijack) many existing such gems for your own purposes. Twitter is a great source of very short, and often funny, stories to plagiarize. But keep in mind that having everyone doing it does not mean it's legal.

## Digital Native!

Maxims can be so short they can fit within a hashtag. This is a popular practice on Twitter, with hashtags like "BlackLivesMatter". Our software industry also loves hashtags as a way to name new practices: #NoEstimates[87], #BeyondBudgeting[88], or #NoProject[89].

Note that hashtags are not just for Twitter or Facebook. You can use them IRL (In Real Life), and even verbally, which sounds deliciously awkward.

[87] https://twitter.com/search?q=%23NoEstimates
[88] https://twitter.com/search?q=%23BeyondBudgeting
[89] https://twitter.com/search?q=%23noprojects

**Pro Tip** Use wifi password as a maxim (you have to carefully type it manually! For example, if like in my company Arolla you'd like to encourage environment-friendly behavior, you could rename your wifi network or wifi code as "ReduceReuseRecycle".

## Goodies

Goodies are not always green, but sometime they are useful. Goodies are a traditional way to repeat a message, and it does not have to be your brand, it can be a maxim too.

The conference DDD Europe recently did a great job at that with 3 different T-shirt designs with 3 different maxims, for example:

- MAKE THE IMPLICIT EXPLICIT
- THROW AWAY THE MODEL

**MAKE THE IMPLICIT EXPLICIT blue t-shirt**

Most typical goodies are T-shirts, card decks, cheat sheets, large takeaway posters, mugs, pens, postcards, stickers, sweets, relaxation widgets…

## Comics

Comics are compelling ways to tell a story, for example a story of frustrated users now, with their dream of a better software. This can be used to document and explain the rationale for a new project.

Stories of users doing their job and sharing their most important stakes are also great to explain hence document in an accessible way the fundamental business stakes of a business activity.

I've used child-ish comics in very corporate environments to explain a process for the development team. I've used less child-ish comics to help explain a governance process to senior management too, in a real big serious bank. It worked and was appreciated.

There are several online comics generators which can help create basic comics from libraries of characters, settings and effects. This makes it possible to anyone to create a comics, even without any drawing skill.

# Infodecks

Infodecks are slides used as documents to be read on screen rather than projected in front of an audience. As Martin Fowler writes, they are "more approachable and easier to communicate information than a traditional prose text"

Infodecks offer many advantages:

- You can use spatial layout to help explanation
- They discourage long prose that people don't read
- It's easy to include diagrams as primary elements in the communication

The important thing is not to confuse infodecks with slide decks meant to be projected to a large audience. When projected, there should very little text, using a very big font size, along with many illustrations.

"Infodecks are an interesting form to me, if only because it seems nobody takes them seriously. [...] A colorful, diagram-heavy approach that uses lots of virtual pages is an effective form of document - especially as tablets become more prevalent." – Martin Fowler bliki[90]

# Visualizations & Animations

A bit more difficult to produce, animations and animated visualizations are perfect to explain temporal behaviors.

A great example is the beautiful visualization of Distributed Consensus in Raft[91] http://thesecretlivesofdata.com/raft/ which shows how the nodes elect their leader in the face of various events.

Another personal favorite is the apparently crazy idea of showing how sort algorithms work[92] using sound along with crude display:

[90] http://martinfowler.com/bliki/Infodeck.html

[91]

[92] http://m.youtube.com/watch?v=kPRA0W1kECg

# Lego blocks

Lego block have become popular among Agile circles over the past years, so now we can use legos during meetings, as a planning tool, or even to represent a software architecture physically in 3D. Other system of avatars or construction blocks can be used as well as a mediation tool during conversations. The problem with these constructions is that usually nobody can understand what they meant after a few days.

# Furniture

Even your furniture can tell stories. Fred Georges explained in one of his talks how the tables expressed literally the internal organization of a startup: each table represents one project team. No more room on the table means the team has reached its maximal size. Otherwise you're welcome to join the team if you feel like to! It's a direct proposition!

Furthermore, you can tell from the huge iMac screens where the designers are, whereas Linux machines more likely suggest developers are working there.

# 3D printed stuff

3D printed models are now easy to produce. This means you could project a particular view of your application and print it in a solid material. This helps everyone use their visual and world-sensing strength to grasp visually and by touching the elements. 3D and removable layers are useful to represent several dimensions of the problem stacked on each other, and well aligned.

# Part 12 Introducing Living Documentation

# Introducing Living Documentation

It starts with someone willing to improve the current state of affairs in either the documentation or the way software is done. Since you are reading this book you are probably this person. You may want to start Living Documentation because you're afraid to lose knowledge, or because you'd like to go faster with the relevant knowledge more readily available. You may also want to start it as a pretext to show flaws in the way the team is making software, e.g. in the lack of deliberate design, and you expect the documentation to make it visible and obvious to everyone.

The hardest step is to find out a compelling case of missing knowledge. Once you have a demonstrated case, and provided you can solve it with one of the Living Documentation approach, then you're on the right track.

## Undercover Experiments

If you feel alone in your interest for Living Documentation, you may want to start gentle, at your own pace, without making a lot of noise about it and most importantly, without asking for authorization. The idea is that documenting, whatever the way it is done, is part of the natural work of a professional developer.

**Introduce nuggets of Living Documentation naturally as part of the daily work. Start annotating your design decisions, intents and rationales at the time you're making them. When there is some slack time or a genuine need for documentation, turn the allotted time to create simple documentation automation like a simple living diagram or a basic glossary. Keep it simple enough to have it work in a few hours or less. Don't talk about it as a revolution, but just as a natural way to do things efficiently. Emphasize the benefits, not the theory from this book.**

Of course, when people become more interested in the approach, you can talk about "Living Documentation as a topic", and direct them to the book.

## Official Ambition

Another way to introduce Living Documentation is through an official ambition.

Going the official route usually starts from management, or at least requires that the management is a sponsor. Documentation is often a source of frustration and anxiety for the managers, therefore this topic is often promoted even more by managers than by the development team itself.

Having a sponsor is good news: you have dedicated time and perhaps even a team to implement. The counterpart is that as an official ambition, it will be highly visible, closely monitored, and there

will be pressure to deliver something visible quickly. This pressure may endanger the initiative by forcing success. But Living Documentation is a discovery journey, there's an experimental side to it and there is no clear path to success in your own context. You'll have to try things, decide that some are not applicable, adjust other to your own cases. This is better done without excessive scrutiny by higher-ups.

This is why I'd recommend to start with Undercover Experiments, and only promote the topic as an Official Ambition after you've found the sweet spots of Living Documentation in your environment.

## New Things Are difficult for two reasons: they have to work, and they have to be accepted.

A typical path I advice is to start with creating appetite, then try for quick opportunities to show benefits, and then to grow from there.

1. Start by creating awareness in the larger audience. A great way to do that is through a all-audience talk, informative and entertaining. The point is not to explain how to do things, but to show how life could be better in contrast to the current state of affairs. Nancy Duarte's book Resonate[93] is full of advices on how to that well. Listen to the feedback at the end of the session and a few days later to decide whether the appetite is there to go further. Otherwise, you may want try again some weeks or months later, or you may decide to go undercover first.
2. Spend some time with the team or an influencer team member to identify what knowledge would most deserve to be documented. From that propose quick wins to try as short items in the backlog, or as part of time dedicated for improvements. Retrospectives are also a good time to consider Living Documentation issues and propose actions. It is important to focus on real needs that many people find important.
3. Build something useful in a short period of time, and demo it like any other task. Collect feedback, improve, and collectively decide of whether to expand now or later, if needed.

## Starting gentle

As a consultant I regularly sit with teams in companies of all sizes. When they ask to create more documentation, I tend to suggest the below stone steps.

First of all, I remind that interactive and face to face knowledge transfer must be the primary mean of documentation, before anything else.

This said, we can then consider techniques to record the key bits of knowledge that have to be known by everyone, that every newcomer has to learn, and that matter in the long run.

[93] http://www.duarte.com/book/resonate

At this point they say "Let's write that stuff in our Wiki". Which is fine, as long as we understand that a Wiki is a nice place for Evergreen Documents, for knowledge that does not change often. For everything else, we can do better.

Where to start? I like to mention various ideas very quickly to scan the interest of the team members. For example I would mention briefly each of the following:

- Adding a simple README describing what the project is about
- Adding a simple Decision Log as a Markdown file in the base directory of the project, with a recap of the 3-5 main architectural and design decisions since its inception
- Tagging the code with a custom annotation or attribute to highlight the Key Landmarks or the Core Concepts. Combined with the search by reference in the IDE this becomes a simple yet effective way to provide a sightseeing map of the code base.
- On a similar note, tagging the code with Guided Tour annotation or attribute to provide a simple way to follow a request or a processing end-to-end across the various fragments of code, across the various layers or modules, in a linear fashion. Again this relies on the search by reference in the IDE.
- Turn the most important Napkin Sketch into an ASCII diagram in the Decision Log file.

This list deliberately contains only stuff that can be done and committed within a short period of time. For example we've been able to add and commit a decision log with 5 past key decisions, marking three Key Landmarks plus a Guided Tour with 5 steps within 2 hours. This includes the creation of the two custom attributes for the Key Landmarks and the Guided Tour respectively, checking the search in the IDE worked well and checking the Markdown rendering was fine in TFS.

The goal so far is more to create awareness and interest by reaching attractive results quickly. The goal is to hear "Wow, I really like that approach, I'm hooked now!" after doing that.

Another goal is that just by going through these simple steps, the team members can already experience the "Beyond Documentation effect": "Ouch, I now realize how sloppy and half-finished our structure is." That's a lot of goodness for 2 hours!

Given genuine interest from the team and some available time, we can go further and try Word Clouds, Living Glossary or Living Diagrams.

# Going big and visible

After a gentle start, you may want to go for bigger ambitions. I'm not saying it is always a bad idea, but it may be dangerous for several reasons:

- Visible ambitions usually need to exhibit symbolic progress shown by a quantity of outcomes or even KPI's. But does it mean anything "to be 40% Living Documentation"? Doing living documentation for the sake of it will eventually discredit the approach.

- The benefits can be deferred after months, which can make it hard to show the return on investment if measured over 3 months.
- As mentioned before, it may take various adjustments when applying the techniques from this book in your particular context; these adjustments may be perceived as failures in the meantime.
- What's useful for the team may not be what the management expected. If that's the case, put yourself in the management shoes: what would make you happy with respect to documentation? If you can make previously hidden knowledge accessible to non developers, it may be a good thing for everyone. The managers will be able to judge something by themselves, based on objective facts extracted from the code base. And when you setup the Living Diagram or whatever other mechanism you have an opportunity to do the curation and the presentation in a way that promotes your agenda, for example to encourage a good thing or to warn against a bad one.

In any case, remember that documentation, living or not, is not an end to itself but just a mean to accelerate delivery. This acceleration of delivery can be direct, when decisions are taken faster thanks to the knowledge readily available through the living documentation. It can also be indirect, when creating the documentation raises awareness on everything sloppy in the system, in the thinking or in the communication between the stakeholders. By fixing the root cause you improve the whole system, which in turn will accelerate delivery.

# A Tale of introducing Living Documentation to a team member

One week I met with a team member who was interested in learning more on Living Documentation. He was just curious, not convinced. But he was at least curious, which is a good start.

## Conversations first

I start with questions in a conversational style. I'm supposed to explain what Living Documentation is; instead I start by putting myself in the shoes of another team member willing to learn about the project:

"Tell me about the current projects." - "I work on 3 different projects." - "Let me take notes and sketch what we say on this flipchart."

> What's the name of the project? What's its purpose, and for who?
>
> What's the eco-system with the external systems and external actors? What are the overall Input and Outputs?
>
> What's the execution style: is it interactive, a nightly batch, a Github hook? What's the main language: Ruby, Java & Tomcat?

These are all standard questions so far. Answers come naturally. But then I ask:

> What is the Core domain in your opinion?

This comes at a surprise. He needs some time to think about it. His first moment of surprise is that the answer was not obvious, after several months on the project.

"Oh… Now that you mention that, I now realize that our core domain is probably the way we insert deep links that point to our system in the feed we provide to the external partners, so that they bring us qualified inbound web traffic. I didn't think about it this way before, and I'm not sure everyone in the team is aware of that."

"But is this deep link thing the raison d'être for the whole project?" - "Yes, absolutely." - "Do you think everyone should know about that? - "Obviously, yes". - "So we should document that somewhere?" - "Of course!"

## First Debrief

Now is the time to debrief and introduce the basic concepts of Living Documentation:

"You realize how I learnt precisely what I was interested in through conversations?".

Living Documentation is primarily about **having conversations to share knowledge**. My goal in the conversations so far was to show I could learn a lot of what matters to me, quickly and without wasting time on any other stuff. Interactive conversations and the high bandwidth of talking are hard to beat, especially with the support of the flipchart.

> It was great that you sketch, it helped me check your understanding of what's I said.

The second point I can introduce now is that some of the knowledge we talked about so far needs to be recorded in a persistent form. And the good thing to absolutely recognize is that most of this knowledge so far is **stable over time**. This is lucky, so in this case we can use **Evergreen Documents** in any form: Wiki, text etc. But we must make sure not to mix any volatile and short-lived knowledge, or we immediately loose the benefits of Evergreen documents: documents that don't need any maintenance yet remain true forever (or for a very long period of time).

There is a third point here already: the concept of "Deep Linking" we uncovered is a standard concept already documented in the online literature. As such it's **Ready-Made Knowledge**. We can link to it on the web, so there is no need to explain what it is again. We're lazy.

One last point we begin to see in this last example is that by paying attention to the documentation, even the person with the knowledge also learns and gains additional awareness in the process. That illustrates the benefits "Beyond documentation", and it's probably the biggest value added of a Living Documentation.

## Time to talk about the code

After all my questions on the context and on the problem side, I'd like now to know more about the solution side, in other words the code:

"How's the code organized in packages & dependencies?"

Then we draw the folder hierarchy on the flipchart. That's concrete, I recognize a structure very close to the Hexagonal Architecture (no, it's not a coincidence).

I know that it's hard to tell your knowledge because what you know is, well, obvious cause you know it, so everybody knows it, right? That's the curse of knowledge. So I like to provoke a little bit:

"Imagine you're all gone, after the project is delivered and there was no budget left to keep you. Then a year later the project has to resume with additional features to deliver, so a new team is formed; what risks do you see that the new team will degrade the current system?"

In this fictive situation, it's easier to answer:

"I'd say junior developers new to the project may put business logic in the REST endpoints, and that would bad". - "Sure it would be bad. Still I think there should be no need to tell that in 2016, it's supposed to be known by any professional developer nowadays."

More generally, they mentioned that they were doing everything pretty much standard, with no surprise. This means no need for documentation on all the standard stuff. Also, the code is rather clean, so it tells HOW it's done. It just doesn't tell WHY it's done this way.

Any other risk new team members could degrade the design of the system by accident?

"In fact we designed a whitelist and a blacklist mechanism to filter content that we export depending on the external partner. But we did it in such a way that the code is totally agnostic with respect to the external partner. Only their configuration is specific."

"You mean that does not necessarily show from the code without giving any hint?" - "Yes, it's likely that a newcomer would quickly add an IF statement around the next partner-specific behavior they need to support, and break the design as a result."

## Decision Log & Guided Tour

We should record that design decision. We can do that in a **Decision Log** as a plain Markdown file at the root of the project, in the source control. It's quite concise: date, decision, rationale, consequence. Three sentences can be enough.

What else? The code of the project is not bad, but still is not obvious to follow a user request through all its stage in the system.

"For that we could do a guided tour", I say. And I explain and show how to create a custom annotation `@GuidedTour` to mark each step in our guided tour. He quickly devises the best 7 steps of the guided tour and adds the annotation on each of them. It took 20 minutes to introduce the first tour.

Furthermore, through the tour I found out that a significant part of the overall behavior was a cache on calculations on web-services, in a `Read-through` fashion: that's Ready-Made Knowledge again!

We then create another custom annotation `@ReadThroughCache` to mark that knowledge, with a brief definition and a link to a standard explanation on the web.

After 2 hours and a half of talking and creating annotations to support our very first Living Documentation, it's time to get feedbacks, and it sounds encouraging:

"I like the idea of using annotations for documentation: it's lightweight and easy to add without asking the right to do it. I can start solo and locally. In contrast, other techniques like Living Diagram are more like team decisions I think. And linking to Ready-made knowledge saves time and is more accurate than if I tried to explain it myself in writing."

I concur, mentioning that it's part of an Embedded Learning approach:

"It's often the case. Simple annotations in the core also hint your team members at interesting ideas in the literature they may not know otherwise."

But he's not totally convinced that it works for everyone:

"Yes, if they realize they don't know and are curious to learn more. Some will read the links and learn by themselves, but some will probably not and will ask me instead…"

"- But I see that as a feature! This invites a discussion. That's another opportunity for learning, probably for both of you."

# Common objections

It's not because you'd like to start doing living documentation that everyone around agrees. Perhaps they don't have the need, or they don't see the benefits.

## Annotations are not meant for documentation

"I don't like to use annotations for documentation because I don't like adding code that does not execute"

- "You know, you do it already when you mark code as [Obsolete] or @Deprecated".
- "Oh, yes. Fair point. Why not then."

For the sake of simplicity, let's be caricatural and let's polarize comments vs annotation as good vs bad: "Comments are bad and should be avoided; but if the information to record is really important, then it's worth its own custom annotation"

### "We do it already"

That's a standard objection to anything. To some extent everything looks like everything. "At the end it's the same thing".

Yes, you certainly do the practices in this book to some extent, but is that really a living documentation approach? The keyword here is *deliberate*. If you happen to do some of that by chance then it's fine, but it will be even better when done deliberately. It's up to your team to decide where to put the cursor and what's your documentation strategy. Such a strategy has to be emergent and deliberate. It must fit your particular context and be accepted by everyone involved.

Your documentation strategy will mix practices you already do, push some of them further, and introduce new practices that sound promising. And you will adjust all that over time to get the most benefits with the minimum of efforts.

"We have all the knowledge that we need"

Perhaps you do have all the knowledge because you were there before the rest of the team, are you sure everyone else feels so comfortable?

> If you're having lots of technical meetings, it MAY indicate that your internal documentation could be better Mark Seeman (@ploeh) on Twitter

Perhaps you just hate documentation, and I can totally understand that. But please acknowledge what you don't know.

## Migrating legacy documentation into a Living Documentation

If you have legacy documentation, you may leverage on it. This avoids the Blank Page syndrome, and offers an opportunity to review past knowledge in a new light. You have old PowerPoint documentation? Turn it into a living one! Put the knowledge from the PPT back into the source code, whenever it fits best:

- Vision & goals could go into the README file, as Markdown
- Pseudo-code / sequence diagram could be made as plain-text diagrams or ASCII diagrams, or just replaced by a reference to a test doing the same scenario
- Description of main modules and methods could be done within the source code itself, through some class and module-level comments, annotations and naming conventions
- Comments in config items

Notice how all this knowledge escapes from shared drives and wikis to find a new home all in the source control.

> A It's also striking that the old content that was all concentrated within a few slide decks or Word documents becomes spread all over the code base when moving to Living Documentation. It may sound like a bad thing. Sometime you would prefer some overview slides kept together as one document. But for most of the practical knowledge, the best location to keep it is as close as possible to the place you'd need it.

You could perform a documentation mining on all existing written documents: emails, Word documents, reports, meeting minutes, forum posts, entries into various company tools like application catalogues… Every time a piece of knowledge *still sounds relevant after all this time*, then it's probably worth preserving.

In practice you would deprecate or remove the old content, possibly with a redirection to the new location of the similar knowledge or an explanation on how to find it from now on. A former colleague Gilles Philippart (@gphilippart) calls this migration "Strangle your documentation" by comparison with the Martin Fowler Strangler Application legacy pattern

## Marginal documentation

Your documentation endeavor does not have to be complete at first attempt. It should evolve over time. One approach that's often a good idea when willing to improve something is to focus on the marginal work:

**From now on, every new piece of work will follow a much higher standard**.

Improve your documentation marginally. By paying close attention to what you do from now on, even the parts of the legacy that still matter will be taken care of over time. And don't worry too much about the rest.

Sometime you can segregate the new addition to live in their own clean bubble context; this makes it easier to clearly set a higher standard of living documentation, which is nothing but a higher standard of everything: naming, code organization, top-level comments, clear and bold design decisions made visible in the code, plus the more "typical" Living Documentation stuff like Living Glossary and Diagrams, Enforced Guidelines etc.

# Introducing Living Documentation by example

This real-world example is about batches to export credit authorizations from one application to external systems. Members of the team stay less than 3 years in average, therefore the need for some documentation is not controversial here. The team and the managers heard about Living Documentation, they're interested so we eventually spend one hour discussing what could be done.

When considering what to do, we try to focus on everything that should be documented in order to improve the life of the development team. Then by looking at the current state of the available documentation, we can then propose actions to better manage the knowledge.

Currently, there are some documents but they are out of date and not reliable. We usually have to ask the most knowledgeable team member all the time to get the knowledge needed to perform any task.

There's a lot of potential for improvement, including some quick wins. We could introduce all the items below to start a Living Documentation journey.

## README and Ready-Made Documentation

The source code repository does not have a README file at its root. Therefore, first add a `README` file at the root of the module.

In this README file, mention clearly that this module follows the **Data Pump pattern** with a brief explanation and a link to a reference on the web. From a Living Documentation perspective, we are referencing a READY-MADE documentation.

To be more useful, we can elaborate a little bit on the data pump with a description of its main 'parameters' in the `README` file:

- **Target system & format**: Company standard XML dialect
- **Governance**: this data pump belongs to the *Spartacus Credit Approval* component and is managed as part of it
- **Rationale**: The Data Pump pattern is chosen over more standard integration through services endpoints because the target system imposes a **bulk** integration style, with lots of data to transfer between the two systems daily.

All this remains a bit abstract, so it's desirable to include in the `README` file a link to a folder containing some sample files describing the inputs and outputs of the component:

```
 Sample input and output files can be found in '/samples/' (with a link to 'targ\
et/doc/samples')
```

# Business Behavior

The core complexity of the module is the determination of eligibility. It is best described by business scenarios, already partially automated in *Cucumber JVM*.

We can reuse some of these scenarios to generate the sample files mentioned before. This way the sample files will remain up-to-date.

Having business-readable scenarios is nice, but here we need to make these scenarios accessible to non-developers. The basic *Cucumber Report* can show the scenarios as a web page online. You may consider the alternative tool *Pickles* for the living documentation to be available online to anyone in a better form and with a search engine.

# Visible Workings and Single Source of Truth

The transcoding used to generate the XML report is defined in code, and in an Excel file as well:

```
 | input field name | output field name | formatter           |
 | trade date       | TrdDate           | ukToUsDateFormatter |
```

We realize there is duplication of knowledge for no particular benefit here. Who's the authority in case of disagreement? Usually it should be the spreadsheet file, but after a while it will be the code.

We could improve that situation by deciding that the spreadsheet file is the single source of truth (aka the Golden Source). The code then parses this file and interprets it to drive its behavior. In this approach, the file is directly its own documentation.

For example in pseudo-code:

```
  For each input field declared in a generic data dictionary
    Fetch the value from the input field
    Apply the formatter to obtain the value
    Lookup the corresponding output field
    Assign the formatted value to the output field
```

You may go the other way round too, by deciding that the code is the single source of truth and you generate a file directly out of the code. This won't work if your code is mostly made of a lot of IF statements. Being able to generate a readable file from the code imposes a generic structure to the design of the code. Basically the code would embed the equivalent of the former spreadsheet file, but hardcoded as a dictionary, e.g. in a `Map` in Java.

This data structure can then be exported as a file in various formats (`xls`, `csv`, `xml`, `json`...) for non-developers audiences.

# Integrated Documentation for developers, Living Glossary for other stakeholders

Do you really need to produce Javadoc reports? It's so easy to browse the code in your IDE that you probably won't use the Javadoc reports much. The Javadoc reports are now available directly at your fingertips in your IDE. The same is true for UML class diagrams of classes and their type hierarchies. All this is already Integrated Documentation built-in your editors.

If you really need a reference to give access to the concepts to non-developers, you may introduce a Living Glossary. It scans the code in the `/domain` package to generate a markdown and HTML glossary of all the business domain concepts in the code, extracted from classes, interfaces, enum constants and perhaps some methods names and Javadoc comments. Of course, for the glossary to be good you'll probably have to review and fix many of these Javadoc comments.

# Living Diagram to show the design intent

If the internal design follows a known structure like the *Hexagonal Architecture*, we can make it visible thanks to naming conventions of the corresponding modules. This naming convention and the name of the structure must be documented in the `README` file:

```
 The design of this module follow the Hexagonal Architecture pattern (link to a \
reference on the web).

 By convention, the domain model code is in the src/*/domain*/ package, and the \
rest is all infrastructure code
```

That's Ready-Made documentation, again.

You may include a link to the domain model package, but it has to survive refactoring changes like moving the domain folder into another folder; to make the link more stable we can make it a Bookmarked Search directly based on the naming convention as a regular expression: `src/*/domain*/`

# Contact information, and Guided Tour

Who should I contact for questions? Just check the service registry, *Consul* in our case, which should have this information as required by the company architects.

A Guided Tour just for the batch is not that difficult to create with a custom annotation, but it may not be very useful for developers. The batch is built with the very standard and well-documented `Spring Batch` framework. This framework controls completely the way the processing takes place. We can safely assume that every developer knows about this framework and the way it works, or that they can learn about it from the standard documentation and tutorials. No need to create any additional custom Guided Tour for that.

## Micro-services Big Picture

How does our Data Pump module fit within the bigger system made of many micro-services? Answering this question takes more effort. One approach would be to regularly run a Journey Test (End-to-End Scenario going through a large number of components of the system) on some environment with distributed tracing enabled. Tools like *Selenium* for running the test and *Zipkin* for distributed tracing may come to mind. You could then visualize the distributed traces to produce another kind of Guided Tour revealing what happens between services during each Journey test, as a big picture of the system. As usual for living documents, the curation is key to filter what matters (e.g. what services are talking to what other services in this scenario) out of a huge number of details (all the calls between services and all the events on the messaging bus between them).

# Selling Living Documentation to management

A common question about a new approach is “how do I convince my management to try it?”. There are different answers depending on your context.

The first, and my preferred, answer is that it is up to the team to select the way to meet the expectations of other stakeholders. It is everyone’s business to require that knowledge is shared, however does the team really need approval to decide how to perform its work efficiently? Keep in mind that everyone in the team, developers, testers, business analysts are also stakeholders of the project. To better deliver to other stakeholders they have to take care of themselves first. They also need enough autonomy to try practices, and as Woody Zuill says “to amplify what works” and to stop what does not.

If your company and managers are proud to be “really agile” and to “empower their teams”, then they should trust the team, and you should not need any formal approval to try Living Documentation or any of the related recommended practices, even the most radical ones like pair-programming or mob-programming. Of course this autonomy comes with the full responsibility for the actual results.

That said, it may be the case that putting in place a living glossary or a living diagram for the first time requires from half a day to two days of work. This may be too long to do the effort without having it in a formal backlog, in which case you need to convince someone.

If there is a documentation budget or documentation tasks planned already, you may also want to reuse that time to invest on living documentation instead. This again may again require an approval.

## Start with an actual problem

As usual when introducing new approaches, don’t preach. Instead, show the benefits, and the best way to do that is on a real problem, when it is time to tackle it.

To find out a real knowledge problem, you may ask around you “Is there anything you would not feel comfortable working on alone?”, or “Is there anything that is not clear to you?”. You may not ask anything but just pay attention to the questions asked during the day, the week or the iteration. Some will hint at candidates for documentation.

One efficient way to know what’s important is to carefully take notes of everything you mention or explain to each newcomer, during the on-boarding period. If you ask the new joiners for an astonishment report, it will also contain candidates for stuff that either should be fixed, or that should be documented because they are surprising.

Once you identified a knowledge sharing issue, make sure that everyone acknowledges it is a genuine documentation problem worth tackling. Then propose a solution, inspired from this book. You don't have to use the term "Living Documentation", you can just mention that this approach has already been done in other companies, in large-ish corporations, and in small early startups too.

You may also start with something small, done on your own time, that you can show to the managers you want to convince. It may be a report, or a diagram, or a mix of a documentation plus some indicators managers are particularly interested in. Emphasize how you can save time and improve the satisfaction thanks to the approach.

Once it is done, the benefits should be enough to convince of keeping the approach. And if the benefits are not there, please tell me so that I can improve the book. Still, even in the worst case you will learn something valuable in the process, and you will probably have one example of a traditional documentation that was just a bit more expensive than usual.

# A Living Documentation initiative

If there is a lot of pressing documentation issues, you may want to go with a bigger ambition on the topic. This book is also made for that purpose of pushing the idea forward and make it a standard package that can be a reference. Show the book to the people you want to convince. Show the video of talks on the same topic, I've done some already that were well-received judging by the feedbacks in the venues and on Twitter.

Showing the benefits on a first "pilot" case within the company remains the best option to start with. Nobody will pay much attention at first, but with early successes more people will try to replicate or to even formalize the initiative for the benefit of their career.

As soon as we're talking about an identified initiative, we have to convince upper management that it is worth investing time in the teams, and perhaps some additional coaching and consulting if needed. One way to sell Living Documentation is to consider that it is a prerequisite to achieve sustainable **Continuous Delivery**, a bit like testing is a pre-requisite too.

**Going fast is not possible unless the quality is under control at all time, thus we need a testing strategy with automated testing to the max.**

**Similarly, going fast in the long term is not possible unless the quality of the design and of the understanding of the business domain is under control at all time, thus we need a living documentation strategy.**

Many of the key reasons to adopt a Living Documentation approach are already in front of your eyes, and show in your weekly time-tracking and in the current state of your knowledge management.

Overall, my feeling so far is that documentation is a concern close to managers' heart. The matter of skills and knowledge transmission between the team members is already a common source of anxiety for the management; it represents a cost in time but more importantly in defects and mistakes:

- Skills matrix creation and updates.
- Turnover rate
- Time spent on-boarding newcomers
- Anxiety related to the Truck Factor, the risk of losing key knowledge is the team is hit by a truck when going back from lunch
- Ratio of defects and incidents caused by "I didn't know that".

Lack of documentation is a hidden cost, just like the lack of tests. Every change needs a complete investigation and an assessment, sometime even a pre-study. The hidden knowledge has to be mined again each time. Alternatively, the changes are made in a way that is not in line with the previous vision of the system, which makes the application increasingly bloated, making the matter worse over time. This may show like the following:

- Time to deliver a change
- Negative trends of any code quality metrics, perhaps the most telling being the mere size of the code base. If it grows regularly, it is probably a sign that the design is too weak. There is not enough refactoring to start with, and the each change is basically an addition.

And there are also the arguments on the documentation, or lack of thereof, in itself:

- Unmet, or not updated frequently enough, compliance requirements with respect to documentation
- Time spent on writing documentation, or on updating the existing documentation
- Time lost searching the right documentation
- Time lost reading documentation that is incorrect

You may want to perform a review of the actual quality of the existing documents that pretend to be the documentation, with a focus on various indicators:

- Number of different places where documentation can be found (including the source code, the wiki, each shared drive, team members machines etc.)
- Time of last update
- Proportion of authors of the last updates who left the team
- Proportion of rationale (explaining WHY not just WHAT) in the documentation
- Proportion of pages or paragraphs or diagram that can still be trusted
- Proportion of knowledge redundant between the source and another kind of documentation
- Short survey like "Do you know where I can find knowledge on that?" on a random set of concerns

And of course you can come up with many other ideas to help realize the actual state of everything documentation. If everything is fine and under control, then the only thing that Living Documentation may improve is the long term cost, thanks to team members working more together, automation and reduction of various waste.

Otherwise, Living Documentation can make documentation feasible again, at a reasonable cost and with an identified value added.

On the value side, it is worth putting the emphasis on the biggest benefits which are not just the sharing of knowledge, but especially the side benefits in improving the software in the process, as described in "Beyond Documentation" part of the book.

## Contrasting current situation to the promise of a better world

## A Strategy must match people aspirations

Nancy Duarte, the author of the book Resonate, offers suggestions on how stimulate excitement and enthusiasm through presentations. It starts with knowing why you want to change things. If you've decided to introduce Living Documentation into your team or company, you could start with asking yourself "Why do I want to share and promote that? Why am I excited?"

Then the current situation can be used in contrast with the new practice you'd like to promote. Here are examples of common frustrations that could be contrasted with the benefits of a Living Documentation approach:

- You don't write documentation, and you feel guilty about that
- Explaining things to team members, new joiners and stakeholders outside the team takes a lot of time, on an on-going basis
- You write documentation, and you'd prefer to write code instead
- You're looking for documentation and when you find some you cannot trust it because it's out of date
- When you create diagrams you're frustrated it takes so much time
- Looking for the right document itself takes so much time for little result that you often give up and try to do the work without
- When you collaborate the agile way with lots of conversations, you feel uncomfortable because your organization expects to deliver more traceable and archived documents
- You do a lot of tedious work manually, including deployment, explaining stuff to external people, and paperwork, and you have the feeling that it could be avoided

Of course it's up to you to customize and decide which items make the most impact in your context, and to decide what part of Living Documentation remedies that frustration most.

More generally, and at the risk of being caricatural, developers usually:

- Don't like writing documentation
- Like to write code
- Love code, doing more with code is appealing
- Hate manual, repetitive tasks
- Love automation
- Are proud of beautiful code
- Love plain text and their favorite tools
- Love logical things: text-first, DRY
- Love to exhibit mastery and geek culture
- Want recognition of skills
- Empathize with real-life messy situations (like everyone)

Whereas managers usually: - Love to see things they usually don't see - Love to see things presented in ways they can feel, and understand whether it's getting better or worse - Love to see documentation they can themselves show someone else and be proud of - Love documentation to be more turnover-proof

Resonate with all that. It's critical for a documentation strategy to exhibit a vision that everybody would genuinely like to happen.

# Documentation for Compliance requirements

*Even demanding compliance requirements can be satisfied with little additional effort with a Living Documentation approach, as part of a continuous delivery cycle.*

If your domain is regulated or if your company requires a lot of documentation process for compliance reasons, like ITIL, you probably spend a lot of time on documentation tasks. This is where the ideas from Living Documentation can meet the compliance goals, reducing the burden for the teams and saving time, while improving the quality of the produced documentation and of the product at the same time.

Regulators often focus on requirements tracking and change management as a way to improve quality. For example, the U.S. Food and Drug Administration writes in its General Principles of Software Validation; Final Guidance for Industry and FDA Staff[94]:

> Seemingly insignificant changes in software code can create unexpected and very significant problems elsewhere in the software program. The software development process should be sufficiently well planned, controlled, and documented to detect and correct unexpected results from software changes.
>
> Given the high demand for software professionals and the highly mobile workforce, the software personnel who make maintenance changes to software may not have been involved in the original software development. Therefore, accurate and thorough documentation is essential.

The same FDA document also describes the importance of testing and of design and code reviews.

It may looks at first glance that agile practices are less documentation-oriented, and therefore not well-suited for demanding compliance requirements. But it is quite the opposite really. When agile practices which are part of the living documentation spectrum are applied, what you actually have is a documentation process which is more rigorous than all the traditional documentation-heavy processes.

Specification by Example (BDD) with scenarios with automation, Living diagrams and a living glossary provide extensive documentation, on each build. If you commit 5 times in an hour, you get your documentation updated 5 times per hour, and always accurate. Even paper-heavy processes do not dream about that level of performance!

[94] http://www.fda.gov/RegulatoryInformation/Guidances/ucm085281.htm

Working collectively, with colleagues in turn to ensure that at least 3 or 4 people know of each change is also an important contribution to various compliance requirements, even though the knowledge is not necessarily written outside of the source code.

You see the idea here: a development teams with a good command of the "agile development" practices and principles, including living documentation and other continuous delivery ideas, is already quite close to checking most compliance requirements, even the notoriously heavy ones like ITIL.

An important remark is that agile practices in general do not necessarily meet the *implementation details* of your company compliance guidelines, which are often full of burdensome procedures and paperwork; still, agile practices often meet or even exceed the higher-level goals aimed for by the compliance bodies, which revolve around risk mitigation and traceability. Agile or not, in the development team or in the compliance office, we all want risk mitigation, some reasonable amount of traceability, quality under control and improving everything. You don't have to follow 2000 pages of boring ITIL guidelines. You can substitute alternative practices which are more efficient, and still be able to check most checkboxes in the checklist of the high-level objectives.

Therefore: **Review the compliance documentation requirements, and for each item identify how it could be satisfied with a Living Documentation approach, typically by using lightweight declarations, knowledge augmentation and automation. Mandatory formal documents based on company templates can easily be generated from knowledge managed in a totally different fashion (e.g. from the source control, the code and the tests). When the compliance expectations are too burdensome, go back to their higher-level goal, and identify how this goal could be directly satisfied with your practices instead. Whenever there is a real gap, then it's likely an opportunity to improve your development process. Finally, make sure that your lightweight process is reviewed from time to time by the compliance team, so that they can grant your team with a permanent pre-approval stamp.**

You'll be surprised how your living documentation can meet or exceed the compliance expectations.

Paul Reeves says in a great article Agile Vs. ITIL[95]:

> Often people believe that rapid deployment / continuous deployment / daily builds etc. can't work in a an environment that is highly process oriented, where rules and process have to be followed. (Usually they just don't like someone else's rules.)
>
> Well, the process is there to ensure consistency, responsibility, accountability, communication, traceability, etc. and of course it CAN be designed to be a hinderance. It, alternatively, CAN be designed to allow quick passage of releases. People blaming process or ITIL are just being immature. They may as well blame the weather.

Paul Penny[96] also makes it clear that

[95] http://reevesresults.blogspot.fr/2011/03/agile-vs-itil.html

[96] http://ppenny.varrowblogs.com/?p=199

> ITIL is about defining, designing, delivering, measuring, and improving services that add value to the business.
>
> Because, contrary to the horribly poor implementations many folks have experienced, ITIL is NOT all about being slow and inflexible. ITIL is about defining, designing, delivering, measuring, and improving services that add value to the business. Last time I checked, this is still something that is expected from IT.

Our experience from applying the ideas of Continuous Delivery have shown indeed that it is possible to map from a lightweight, agile, low-cycle-time process inside of the development team to a more traditional, usually slower and paper-intensive process outside. In contrast to common beliefs, your agile process is probably more disciplined than the other project managed in an ITIL-by-the-book fashion: It's hard to beat a process where automation can produce extensive functional documentation, extensive test results and coverage, security and accessibility checks, design diagrams, and release notes with links to the requested features in a tool and archived emails for the release decision, on each build, several times a day!

When strict procedures are important, automation and enforced guidelines are the best way to make sure they are respected, while reducing the burden of manually applying them. Procedures are great for machines, not for people. The right tooling protects the development team and removes the manual chores at the same time. However, and it may seem like a paradox, good tooling still draws attention to the quality expectations by making very visible whenever they are not met. With this protective harness, every team member is learning the quality expectations on the job, while having the satisfaction of always doing a productive work.

# The ITIL example

Let's focus on the management of requests for change under the ITIL conceptual framework:

## Request for Change management

| Change Activity | Example of agile practice | Example of documentation media |
|---|---|---|
| Collection of change requests | User Stories or Bugs & Enhancements with description, origin, requestor, date, business priority & expected benefits | Stickers on the wall and a Tracking Tool (Jira…) |
| Study & Impacts | BDD, TDD, Test Automation | All Living Documentation artifacts |
| Prioritization | Decision, names of decision makers, target version, date | CAB report (email as PDF) |

| Change Activity | Example of agile practice | Example of documentation media |
|---|---|---|
| Follow up | not started, in progress, done), assignee | Tracking Tool (Jira…) |

Note that agile practices promote slicing the work as shortly as possible. This makes it inconvenient to manage every slice in a tracking tool when a single week contains dozens of slices, each only a few hours long. But this level of granularity does not matter that much for the management of request for change; as a consequence you may only track cohesive aggregates of slices in the tool.

## Release management

| Release Activity | Example of Agile practice | Example of documentation media |
|---|---|---|
| Content | Release notes with a link to related change(s), dates, downtime, test strategy, impacts (business, IT, infra, security) | Ticketing Tool (may be automated as a mix of pre-written documents and generated release notes) |
| Impacts | Based on the change study plus the feedbacks from the Iteration Demo | Living Documentation, archived as PDF |
| Release checks | Automated Test, including tests on SLA, deployments tests in pre-production environments, smoke tests | CI tool, deploy tool results, tests reports |
| Approval | Decision, names of the decision makers, actual delivery date, target version, rollout date, decision date, go/no-go conditions | Email saved as PDF |
| Deployment successful | Deployment and post-deployment tests | deploy tool & post-deploy tests reports |
| Continuous improvement | Retrospectives notes, with names, action plan, issues | Wiki, email, picture of the whiteboard… |

The point here is really to realize that your living documentation can meet or exceed the toughest compliance expectations, while keeping the extra compliance-specific work to a the minimum. This could be an incentive in itself to introduce a living documentation if you're in a compliance-intensive environment.

# Documenting Legacy Applications

> The universe is made of information, but it doesn't have meaning - meaning is our creation. Searches for meaning are searches in a mirror. - @KevlinHenney

## Documentation Bankruptcy

This quote illustrates the case of legacy systems: they are full of knowledge, but it is usually encrypted and we have lost the keys. Without tests, we have no clear definition of its expected behavior. Without consistent structure, we have to guess how it was designed, for what reasons, and how it is supposed to be evolved. Without a careful naming, we also have to guess and infer the meaning of variables, methods and classes, what code is responsible for what.

In short, we call systems 'legacy' when their knowledge is not readily accessible. They exemplify what we could call a "documentation bankruptcy".

Legacy applications are quite valuable, they cannot be simply unplugged. And most attempts to completely rewrite large legacy systems eventually fail. legacy systems are a problem of rich organizations, and that is a good problem to have.

Still, legacy applications raise issues when they have to evolve due to changing context, because they are usually expensive to change. This prohibitive cost of change is related to many flaws like duplication and lack of automated testing, but also directly in the lost knowledge. Any change requires a long and painful reverse-engineering of the knowledge from the code base, including a lot of guesswork, before one line of code is eventually touched at the end.

All is not lost anyway. In this chapter we'll see a few Living Documentation techniques which particularly apply for legacy systems in the context of a project to change them.

## Legacy application as a complementary documentation

*Legacy systems should not be considered blindly as a reliable documentation to rewrite new systems*

We've seen before that anything that can answer a question can be considered documentation. If you can answer questions by using the application, then the application is part of the documentation. This is of great relevance in the case of a legacy system with lost specifications. You have to use it to know how it behaves.

In the context of rewriting a part of a legacy system, this may be handy since the new system will probably inherit a significant part of the behaviors of its predecessor. For each feature that will make

it into the new system, the specifications can draw on the former system. In practice, while doing the specifications workshops, you can check how the legacy application behaved as an inspiration for the new one.

## The "rewriting with the same specs" fallacy

One common failure mode of rewriting legacy systems is to rewrite them with the "exact same specifications". It makes little sense to rewrite a system without changing anything, it is just a lot of risk and a lot of waste. Changing only the technology stack is hardly a good idea either, unless your hardware is no longer available commercially and there is no emulator.

Rewriting a piece of software is an expensive endeavor, even from a purely technical perspective, and the best way to improve the return is by taking the opportunity to reconsider the specifications at the same time. Many features are no longer useful. Many features should be adapted to new usages and context. The UI and its UX probably has to change drastically, with impacts on the underlying services. You'll also want the new application to be cheaper to deliver more frequently, so you'll want automated testing too, which come cheaper when you start from clear specifications as concrete examples, as advised by BDD.

I would strongly suggest against rewriting with the same specs. Rewrite a limited portion of the system, and consider it is a project from scratch, with the legacy system, the working application and its source code, as a bonus to draw inspiration from.

Therefore: **In the context a rewriting a part of a legacy system, consider the legacy system as a documentation to complement the discussions on the specifications, not as the given specifications. Make sure a business person like a domain expert, Business Analyst or Product Owner works closely with the team. Don't fall into the fallacy that the legacy system is in itself a sufficient description of the new system to be rebuilt. Take the opportunity of the rewriting to challenge every aspect from the legacy system: the functional scope, the business behaviors, the way it is structured into modules and so on. Regain control of the knowledge from the start, with clear specifications expressed as concrete scenarios, and a clear design.**

The ideal configuration is a Whole Team, with all skills and roles inside the team, as described earlier when talking about the 3 Amigos: business perspective, development perspective and quality perspective.

Having access to both the working legacy application and its source code is a nice bonus compared to projects starting purely from scratch. It's like having another expert in the team, even if it is an old, sometimes irrelevant, expert. After all, the legacy system is the result of a patchwork of the decisions of many different people over a long period of time. It's a fossil.

The perfect case is when the legacy system is instrumented, in which case it can also provide answers to the question "how often is this feature used?".

# Archeology

> Software source code is one of the most densely packed forms of communication we have. But, it is still a form of human communication. Refactoring gives us a very powerful tool for improving our understanding of what someone else has written – Chet Hendrickson, Software Archeology: Understanding Large Systems

When you ask questions to a legacy code base, you need a piece of paper and a pen close to your keyboard at all times to take notes and draw.

This is where you create an on-demand map of the terrain for the task at hand. While exploring the code and playing with it at runtime or with the debugger, you write down the inputs, outputs and more generally all the effects you discover. You take note of what's read or written since the side-effects are what matter ultimately. It will also be essential for mocking or estimating the impacts of a change. You sketch how each responsibility depends on its neighbors, a technique Michael Feathers calls "Effect Map" in his book *Working Effectively with Legacy Code*.

It's important to keep the process low-tech so that it does not distract from the task itself. This documentation work is dedicated for the specific task, therefore there is no need to make it clean and formal right now. However when you're done with the task, you may review the notes and sketches and select the one or two key bits of information that are general enough and that would help for many tasks. They can be promoted into a clean diagram, an additional section or an addition within an existing document. Grow your documentation by a decantation.

Of course you may find questions that the code does not answer. Perhaps the code itself is obscure or surprising. So you need help, ideally from your colleagues nearby, in which case human communication comes back in the picture. The legacy system is not just code, there are documents of all ages, slides, old blog posts, pages on the Wiki, and of course they are all wrong to some extent now.

A legacy environment also include people who were there at the beginning. The old developers may have moved to other positions now but they may answer questions, especially about the context that led to the decisions years ago.

# Bubble Context (Evans)

*Even in a legacy system, you want to work as much as possible in the ideal land where everything is nice and clean.*

If you have some amount of features to build, then you may decide to build the new features in their own new clean Bubble Context. In practice it can be a new specific module, namespace or project, which means it is then easy to document as explained in previous chapters, e.g. using Annotations, Naming Conventions and Enforced Guidelines. A Bubble Context brings the comfort

and the efficiency of writing software from scratch in a brand new project, but integrated within a bigger legacy surrounding.

**A clean Bubble Context integrated within a legacy mess**

As a Bubble Context is a project from scratch inside of a legacy project, it is also the perfect place to practice TDD, BDD and DDD on a limited functional area, to deliver a bulk of related business value.

Therefore: **If you need to make a lot of changes on a legacy system, consider creating a Bubble Context. A Bubble Context defines boundaries within the rest the of the system. Within these boundaries, you can rewrite in a different way, for example driven by tests. In this Bubble Context, you can invest in knowledge by following a Living Documentation approach. Conversely, if you really need full documentation of a part of a legacy application, consider rewriting this part as a Bubble Context, using the state-of-the-art practices for the tests, the code and the documentation.**

It is a good idea to start with high expectations for the code inside the Bubble Context. Its architecture and guidelines should be enforced using automated tools, as a set of Enforced Guidelines. For example you may want to forbid any new commit from having direct references (java import or C# using) on a deprecated component. You may require and enforce a test coverage higher than 90%, no major violation, a maximum code complexity of 2, and a maximum of 5 parameters by method.

Going further in the coding style, if you use the Bubble Context approach you can declare demanding requirements for the full bubble as a whole, e.g. using package-level annotations:

```
@BubbleContext(ProjectName = "Invest3.0")
@Immutable
@Null-Free
@Side-Effect-Free
package acme.bigsystem.investmentallocation

package acme.bigsystem.investmentallocation.domain
package acme.bigsystem.investmentallocation.infra
```

The first annotation just declares that this module (package in Java or namespace in C#) is the root of a Bubble Context corresponding to a project named "Invest3.0".

The other annotations document that the expected coding style in this module favors immutability and avoids nulls and side-effects. They can then be enforced by pair-programming or code review.

A bubble context is a perfect technique to rewrite a part of a legacy system, as in the Strangler Application (Fowler) pattern. The idea is to rebuild a consistent functional area which will progressively take over the old system.

## Superimposed Structure

Especially when creating a Bubble Context integrated within a bigger legacy application, it is hard to define the boundaries between the old and the new systems. And it is even hard to just discuss about that clearly, because it is hard to talk about a legacy system. You would expect to see a simple and clear structure, but what you actually discover is a big unstructured mess.

**Mental Expectations Vs. Actual Situation**

Even when there is a structure, it is often arbitrary and can mislead more than it helps.

**Unlucky project structure**

With legacy code you usually start with lots of effort to make it testable. These tests enable to make changes but are not enough. In order to do changes you also need to reconstruct a mental model of the legacy application in your head. This can be local within a function, or as big as the full business behavior plus the complete technical architecture.

For that purpose you read code, you interview older developers, you fix bugs to better understand the behavior. At the same time you use your brain to make sense of what you see. The result is a structure in your head that you *project* over the existing application. Since the existing application does not show this structure, it is up to you to superimpose a new clear structure onto the existing application.

Therefore: **In the context of creating a Bubble Context, adding a feature or fixing a difficult bug in a legacy system, create your own mental model of the legacy system. This model does not have to visible at all when reading the legacy code. Instead this new structure of the old system is a projected vision, an invention. Document this vision using whatever form of documentation, so that it becomes part of your language for future discussions and decisions.**

This new structure is an hallucination, a vision, that is not directly extracted from the system as it is currently built. You may see it as the description of the system *as it should have been built* as opposed to the "how it is built", in retrospect, now that we know better.

You can show the new model as a superimposed structure on top of the legacy as a plain sketch that you show to everyone involved.

It is desirable to show how this new structure relates to the current state, but this can be too hard to achieve as soon as you want some details, given that the current system may have a totally different structure.

You can invest time in making it a proper slidedeck to present to every stakeholder during a roadshow. You can also decide to make it visible within the code itself to make it more obvious and to pave the way towards further transformations.

Some examples of mental models superimposed on top of legacy systems are:

- **Business Pipeline**: this perspective of the business is similar to the standard Sales Funnel of salespeople. It focuses on the system as a pipeline of stages in the order they happen in a typical user journey: a visitor navigates the catalog (catalog stage), adds items to the shopping cart (shopping cart stage), reviews the order (order preparation stage), pays (payment stage), receives a confirmation and the product, followed by a after-sale service when things go wrong. This model assumes that the volume decreases by a large factor at each stage, which is a nice insight to design each stage technically and from an operational point of view.
- **Main Business Assets**, as in "Asset Capture" (Fowler): this perspective simply focuses on the 2 or 3 main assets of the business domain, like the Customer and the Product in the case of an e-commerce system. Each asset can be seen as a dimension which can be itself split into segments, like customer segments and product segments.

- **Domains and sub-domains**, or by Bounded Contexts (Evans). This perspective requires some maturity on both DDD and the overall business domain, but it also has the most benefits, especially in addition to the other views.
- **Levels of Responsibility: Operational, Tactical, and Strategic Levels**, from the business perspective. Eric Evans mentions that as well in his DDD book.
- A mix of these views, for example 3 dimensions: customer, product, and processing stage, each segmented in stages, customer segment and product segments. You can also mix a business pipeline from left-to-right and the Operational, Tactical, and Strategic Levels bottom-up.

Whatever the superimposed structure, once you have it it becomes simpler to talk about the system. You can propose to "rewrite everything about the payment stage, starting with products that can be downloaded as a first phase". You can decide to "rewrite the catalog part only for B2B customers". Communication becomes more efficient.

However it is up to every member of the team to interpret these sentences the way they see it. It is useful to make this superimposed structure more visible.

# Highlighted Structure

*Making a superimposed structure visible in relation with the existing source code*

The superimposed structure can be linked to the existing code. If you're lucky, the mapping between the superimposed structure and the existing structure of the code is just a large number of messy one-to-one relationships. If you're not lucky, this can just be an impossible task.

**Example of mapping between a technical structure and a business-driven structure**

You can add the intrinsic information of the superimposed structure on each element. For example, this DTO is part of the Billing domain, this one is part of the Catalog domain, etc.

In order to make the new structure visible, you can use annotation on classes, interfaces, methods and even modules or projects-level files. Some IDE also offer ways to tag files in order to group them, but this depends on the IDE and the tags are not usually stored within the files themselves.

```
module DTO
- OrderDTO @ShoppingCart
- BAddressDTO @Billing
- ProductDTO @Catalog
- ShippingCostDTO @Billing
```

This will help prepare the next step: move the classes that deal with Billing into the same Billing module. But even if you don't do that, your code now has an explicit structure showing the business domain.

```
module Billing
- BillingAddressDTO //renamed
- ShippingCostDTO
- ShippingCostConfiguration
- ShippingCost @Service
```

The end purpose of a superimposed structure should be to become the primary structure of the system, i.e. no longer "superimposed". Unfortunately in many cases this will never happen since the effort will not reach the "end state". This should not stop you from following this approach since the approach will help deliver precious business value in the meantime. Even if the legacy code is badly structured, as long as you reason about it using a better structure you already get the benefits of better decisions.

## External Annotations

*Sometime we don't want to touch a fragile system just to add some knowledge in it*

It is sometime hard to touch and commit in large code bases just to add extra annotations. You don't want to risk introducing random regressions. You don't want to touch commit history. It may be so hard to build that you don't want to build it unless absolutely necessary. Or your boss may refuse you change the code at all "just for documentation".

In that situation it is still possible to apply most techniques of Living Documentation, except that the internal means of documentation (annotations, naming conventions) has to be replaced by an external document. For example, a text file mapping package names to tags:

```
acme.phenix.core = DomainModel FleetManagement
acme.phenix.allocation = DomainModel Dispatching
acme.phenix.spring = Infrastructure Dispatching
...
```

With that it is possible to build tools which parse the source code and exploit these external annotations just like they would exploit the regular internal ones.

**Using a registry so that the code is not touched**

The issues with this approach is that it is an External kind of documentation, hence fragile to changes in the legacy system. If you ever rename a package in the legacy, you have to update the related external annotations.

# Biodegradable Transformation

*Documentation of a temporary process should disappear with it when it is done*

Many legacy ambitions involve transformations from one state to another. This transformation may take years, and may never really reach the end state. However you need to explain this transformation to all teams, and you want to show it as part of your Living Documentation.

## Example: Strangler Application

For example, if you're building a Strangler Application[97] that is expected to replace an older application over time, this strangler application will probably live in its own Bubble Context.

You could just annotate this Bubble Context as being a Strangler Application. However, that it is playing the role of a Strangler is a temporary fact and is not necessarily intrinsic to the new application; when it has successfully strangled the old one, it will just become the nominal application, and its annotation will be meaningless. The Strangler Application strategy is therefore a Biodegradable Transformation.

In the meantime every developer needs to know they should use the new strangler application instead of the one being strangled.

Therefore you'd better add a StrangledBy("new bubble context application") annotation to the strangled application to explain that a strangling is pending. When it can be safely deleted the annotation will go away with it.

Of course you could still tag the new application as a StranglerApplication, but you will have to clean this tag eventually, when you are done. And if the strangling never completes, this will another hint pointing to the unfinished initiative.

[97] http://martinfowler.com/bliki/StranglerApplication.html

**An application annotated as being strangled by another**

## Example: Bankruptcy

Some legacy applications are so fragile that they break anytime you try to change them, and it takes weeks of work to stabilize them again. When you recognize that, you may decide to declare them officially "bankrupt". This means nobody should change them, ever.

In large legacy systems with new applications strangling older ones, you don't want to perform the maintenance on two applications at the same time, so you can mark the older one as "frozen" or "bankrupt" too.

You can mark the application as "bankrupt" using a number of means:

- Annotation on the package or an attribute in the AssemblyConfig file
- BANKRUPTCY.txt file explaining what you need to know and to do (or avoid to do)
- Removing the commit privilege to everyone. If anyone tries to commit and asks why it is not possible, it is an opportunity to explain that it is bankrupt.
- As a weaker alternative, you could monitoring the commits and raise an alert when changes are met in a bankrupt application.

# Maxims

*Big changes to legacy systems are made by a number of people who share common objectives*

Once you have your legacy transformation strategy, you want to make sure everyone knows about it well. You may have created a Superimposed Structure. You may have annotated your Bubble Context in the code of the project. However of all the things you need to share with everyone, there are a few key decisions you really want everyone to keep in mind at all time.

Maxims are a powerful answer for that, and they have been for ages.

When your project is to rewrite only a portion of a large legacy system, and you don't want to rewrite more than what's absolutely useful now, that is the billing engine and nothing else:

> One work site at a time (the billing engine)

It has been one of my favorite maxims in a big legacy project. It was meant to remind everyone not to get distracted when working on the project; they had to focus on the main worksite only.

> When in Rome, do as Romans do

This was the counterpart to the single work site maxim. When you happen to walk outside of the main work site, don't innovate or change much, just do the minimum in the local style, even if you don't like it. Be conservative when working in the legacy code that will not be rewritten.

Another legacy maxim which was proposed by Gilles Philippart (him again!) was an extremely powerful one:

> Don't feed the monster! (Don't improve the legacy Big Ball of Mud, it would only make it live longer)

I've found maxims to be a valuable form of documentation. The point is to repeat them often, whenever it makes sense, ideally at least once a day. The maxim format is made to stick, and that is why you may want to give it a try next time. Maxims can also help share the conclusions of your team retrospectives, as agreed upon by the team.

# Enforced Legacy Rules

In a legacy application you've decided that some method should not be called anymore, except from one specific place. For example you decided to turn a read-write legacy application into a *Legacy Read Model*, which should not accept any request to update it, except from the listener responsible to sync this read model from the other authoritative model.

The design decision can be stated in a Decision Log:

> "This model is a Read Model. It is therefore read-only. Don't call this Save method, unless you are the listener which syncs this Read Model from the events sent from the Authoritative Write Model."a

With the following rationale:

> The legacy system has proven to be unmaintenable so we don't want to develop anymore in it. This is why we're building a replacement as another system. But because so many external systems are integrated to it we can simply remove it in one go. This is why we decide to keep this old system just as a Legacy Read Model for integration purposes.

We could also document it directly into the code itself:

- Mark the design decision with a custom annotation @LegacyReadModel with the message and the rationale
- Mark the method as @Deprecated

However being in a legacy system also means we have legacy teams around, some of them remote or in other departments, and we can never be sure they will read our documentation or email, or that they will pay attention when we mention that in our daily standup. And you know that if some developers don't respect the design decision then bad things will happen. We'll get bugs and pay the cost of extra accidental complexity due to the inconsistent data management strategies.

My colleague Igor Lovich came up with a simple way to document that decision as an Enforced Guideline. Let's express the design decision as:

> "Never call this deprecated method unless you're in the White-List of the one or two classes responsible for the sync."

This is a custom design rule that can then enforce at runtime with some additional code:

- Capture the stack trace in the method to find out who's calling it and check it's the allowed piece of code (e.g. throw an exception within a try-catch and extract its stack trace in Java)
- Check that at least one caller in the stack trace belongs to the White-List of allowed callers methods
- Make the check into a Java 'assert', if you want to fail-fast in some environments but not all of them
- Log when the check fails in a way that will trigger specific follow-up (if it gets fired then it's actually a defect)

Coming back to the last maxim mentioned before: "Don't feed the monster! (Don't improve the legacy)", this maxim can be turned into Enforced Legacy Rule too, by forbidding commits into a particular area of the codebase. Or you may raise a warning when a commit is done there. Such enforcement is simple and more effective than long explanations that people miss or ignore all too often.

In practice legacy makes everything more complicated than expected. It takes courage and some creativity to come up with relatively "not too bad" solutions!

# Summing it up: the curator preparing an art exhibition

## Selecting and organizing existing knowledge

The curator of an exhibition primarily decides on a key editorial focus. The focus usually becomes the title of the event. Sometimes the focus is trivial: “Claude Monet, the surrealism”, but even in this case there is an opiniated decision, for example the parti to exclude prior art from the artist which was not yet surrealism.

> Any documentation initiative has to deliver clearly one key message

Good exhibitions try to bring an amount of surprise to create interest: “You’ve always thought Kandinsky paintings were fully abstract, but we’ll show how the abstract shapes evolved from his prior figurative paintings”. You just come to watch the art pieces, but to also grow your culture and understand relationships between artists, art pieces and their era.

> Good documentation brings value added with new knowledge, emphasis on relationships, and by offering a different perspective of things

## Selecting and organizing existing knowledge

The curator selects art works based on the chosen editorial focus. Most of the pieces available are left in the storage room, and only the few pieces of particular interest for the particular exhibit are on display.

> Documentation is a curation activity, to decide what’s more important in a given perspective

He or she decides which to display in which room. A room may be organized around a time period, a phase in the life of the artist, or on a theme.

Art pieces may be displayed side by side in order to suggest comparisons between them. They may be displayed with an ordering which tells a story, chronological or by a succession of themes.

> Organization of knowledge is a key tool to add meaning to a plain collection of pieces of knowledge. We group elements by named folders, tags, or naming conventions.

**The curator in the museum**

## Adding what's missing when needed

The curator writes a few texts explaining the big idea of each section of the exhibition. She or he also writes small labels for each art work. These labels are then displayed on the wall directly next to the art piece.

> Documentation needs an augmentation of knowledge, using annotations, DSL or naming conventions. Some limited amount of text is useful too in some places. This knowledge is attached to the related code elements whenever possible.

When a work considered essential for the exhibition is not in the collection, it will be borrowed from another museum or from a private collection, or sometime even commissioned to the living artist. Sometimes the artist also contributes to the organizing of his or her pieces directly.

Sometime an information is missing. The curator can mandates researchers to conduct investigations, chemical analysis on the painting or by looking at written archives, to find the missing piece in the puzzle of knowledge. For example Le Louvre museum exploits research results on the style of brushing colors on the canvas in order to tell the visitors how much Raphael really participated to each of his paintings. And it reveals that the famous Master did not touch many of them indeed!

> Documentation is a feedback mechanism which helps notice that something is missing or wrong in the code, or in the related knowledge.

## Accessible for people who can't attend, and for posterity

The curator also creates a catalogue of the exhibition, which recaps all the content displayed: the explanative texts by section, the art pieces as quality pictures, their labels. The catalogue as a book usually follows a similar organization as the rooms in the exhibition venue.

Museums now offer exhibition catalogues in a full form, expensive and heavyweight, and also as a shorter form, with just a digest of the major pieces. I usually buy the shorter catalogue, which is a more attractive read by far!

> Documentation also cares about making knowledge accessible, and for making sure the important pieces are persisted for the future. We publish content as documents and on an interactive website, targeted for different audiences and different needs.

# Closing

If you've read this far, congratulations! You've no graduated on Living Documentation!

This is just the beginning of the journey. I'd love to hear from you, your feedback, and more importantly your own initiatives on the topic.

Don't hesitate to get in touch with me, for example via my Twitter handle @cyriux[98]. And if you happen to come to Paris, ping me so that we can chat!

[98] https://twitter.com/cyriux