CODEMOTION WORKSHOP

CLEAN CODE - DESIGN PRINCIPLES KATA

DEVELOP HIGH QUALITY APPLICATIONS, FASTER

CARLO BONAMICO / carlo.bonamico@gmail.com / @carlobonamico

RICCARDO LA TORRE

MODULE 00

Introduction

WHILE YOU ARE WAITING...

- download the labs from
  - https://github.com/carlobonamico/clean-code-design-principles-in-action

```
git clone https://github.com/carlobonamico/clean-code-design-principles-in-action
```

or plain "Download Zip" from browser

ABSTRACT

Most developers agree that "Software must be Well-Designed".

But what is a "Good" Design?

This workshop presents a core set of Design Principles that will help you make your application easier to implement, change and understand.

Starting from a focused but realistic specification, we will practice with a Kata where each step introduces a challenge that can be solved by applying different Principles & Patterns.

TRAINING OBJECTIVES

Through the workshop you will learn to:

- understand the basic dynamics of a complex Software system, and key Design concepts such as Cohesion, Coupling, Abstraction and Impact of Change.
- apply core Design Principles to improve the structure of your applications
- balance the pros & cons of alternative Design choices

WHO THE WORKSHOP IS DEDICATED TO?

Basically, to all developers! Independently from the language / platform you are developing on, this workshop is for you if:

- you are moving from implementing a detailed specification to designing new features and application
- you are interested in improving your Software Design approach and skills
- you want to develop more robust and maintainable applications with more productivity

TABLE OF CONTENTS

TOPICS

- Single Responsibility Principle for methods
- Single Responsibility Principle for Classes
- Single Responsibility means splitting ...
- How to start?

TOPICS

- Reviewing your Design
- Collaborating with other classes
- Generalizing the model
- Incremental development and Evolutionary Design
- How to continue by yourself: references for further learning

PREREQUISITES

- Working knowledge and practical experience in one programming language (you should be able to write/compile/test/debug by yourself a program which reads and parses input and presents output either on the command line or in a simple GUI).
- Good knowledge of Object Oriented concepts (Class, Interface, Method, Variable Scope and Visibility) and ability to "read" them from a sketch/diagram
- Basic knowledge of HTTP

HARDWARE AND SOFTWARE REQUIREMENTS

- Laptop
- Web Browser (Chrome or Firefox)
- Text Editor (Sublime, Atom, Visual Studio Code,â Š)
- IDE (Eclipse, NetBeans, Intellij, Visual Studio,..)
- Pen and Paper

KEY REFERENCES

- All Labs and links available at
  - https://github.com/carlobonamico/clean-code-design-principles-in-action
- Clean Code: the book
  - https://books.google.it/books/about/Clean_Code.html?id=hjEFCAAAQBAJ

WHAT OFTEN HAPPENS

- huge files
- deep interconnections between features
- cross-cutting mechanisms "spread" everywhere
- fragility and ripple effects
- risk of change increases
- productivity decreases over time

We need to better design the software

WHAT IS SOFTWARE DESIGN?

IDEAS...

Among all the possible working implementations,

make a "minimum energy" choice

- that can tolerate some degree of changes ("applied force")
- choosing what to separate and what to keep together

See Carlo Pescio's work http://www.physicsofsoftware.com/
https://www.youtube.com/watch?v=WPgYju3KnIY

SO

- reducing cost of change
- preventing fragility in the face of changes

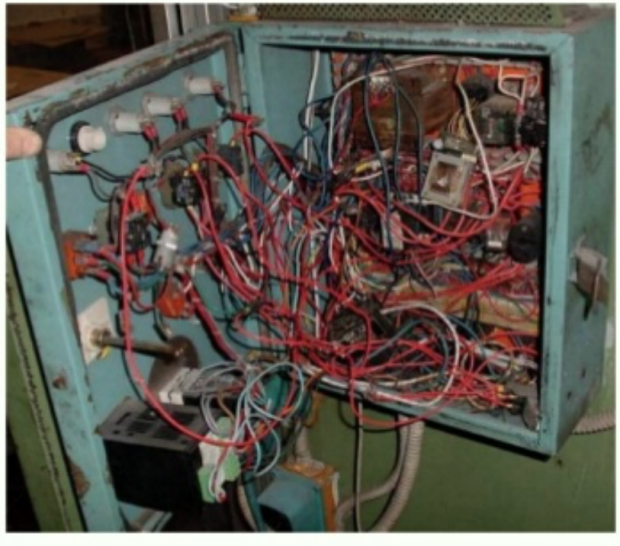
BUT ALSO

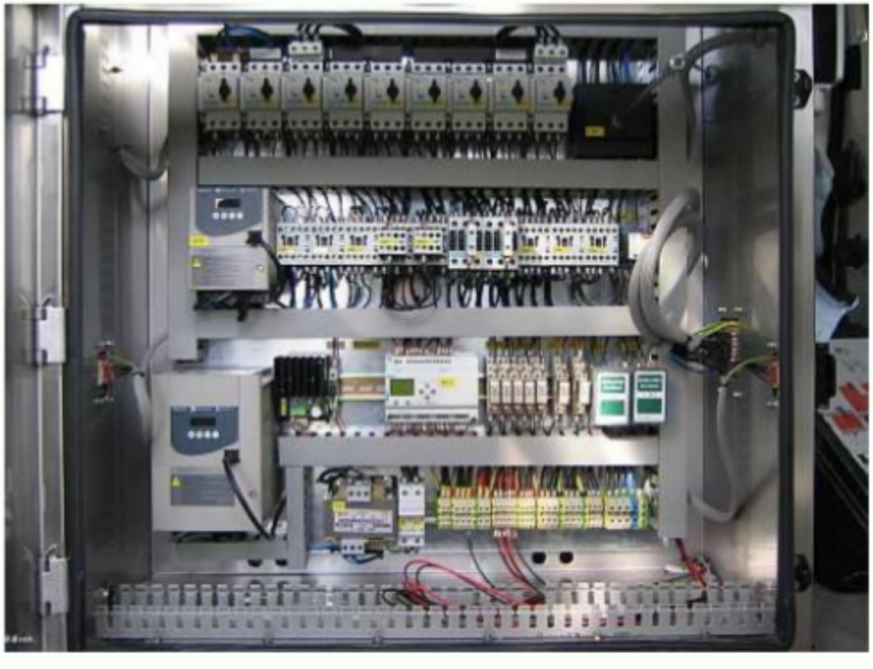
- reducing cost of development
- making possible to create complex systems
- keeping collaboration effective as team grows

https://martinfowler.com/bliki/DesignStaminaHypothesis.html

WHAT IS A "GOOD" DESIGN?

“Spaghetti Binding” vs Components





IDEAS

Good design

- is visible
- is as simple as possibile
- makes it easy to do the correct things
- makes it difficult to do the wrong things
- helps understanding
- helps reuse
- is sustainable

TOPICS

- up-front vs continuous design
- Design vs Agile
- Evolutionary vs Emergent Design
- Modeling vs Design

MAYBE IF WE DO AGILE, WE DO NOT NEED IT?

DESIGN IS VERY IMPORTANT EVEN IN AGILE

IT IS JUST DONE AT DIFFERENT TIMES

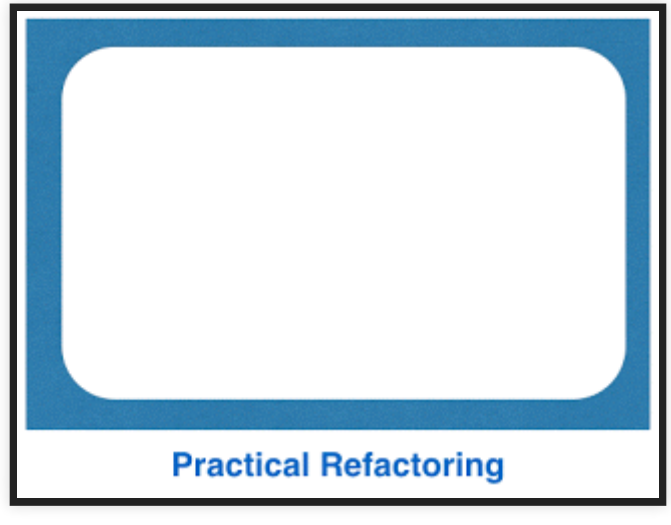
- https://martinfowler.com/articles/designDead.html

EVOLUTIONARY DESIGN

- Making things easier to change
- This does not mean that you do not have a vision
- Plan the overall Path
  - but execute a step at a time
  - take decisions **at the last responsible moment**
  - minimize unneeded complexity
  - maximize available information
- Rebecca Parsons' Book
  - https://martinfowler.com/articles/evo-arch-forward.html
  - https://files.thoughtworks.com/pdfs/Books/Building+evolutionar

EMERGENT DESIGN

- discovery from tests
- discovery from requirements
- making the invisible visibile
- continuous refactoring



DESIGN IS NOT NEEDED EVERYWHERE IN THE SAME AMOUNT/WAY

Think of a building

- external structure
- floor height
- wiring / plumbing
- room layout
- furniture layout

IN SOFTWARE

- interfacing with other system
- splitting the system in modules
- identifying compnents and their interfaces in modules
- implementing low-level details
- repetitive/standard areas vs custom/innovative areas

SO DESIGN NEEDS ARE VERY DIFFERENT FOR

- UI
- Persistence layer
- Infrastructure
- Application-specific logic

WHY IS IT IMPORTANT?

- the two bottlenecks of Software Development: learning and communication
- Managing Complexity

LEARNING IS THE BOTTLENECK

Software development is a learning process

Working code is a side effect

Alberto Brandolini

- https://www.youtube.com/watch?v=yQz9ZYU1bfA
- https://www.youtube.com/watch?v=klsksbDJOhI

LEARNING TO LEARN

- Kathy Sierra
- https://www.youtube.com/watch?v=FKTxC9pl-WM

COMMUNICATION IS THE BOTTLENECK

COMMUNICATION BETWEEN

- Users / Domain Experts and Developers
- Developers / Team members
- Future Developers / Maintainers

DESIGN IS A COMMUNICATION TOOL

COMPLEXITY

Software development tasks are increasingly challenging

We need design to make Complexity tractable

- Essential Complexity
- Accidental Complexity

ESSENTIAL COMPLEXITY

- inherently present **in the business Domain**

If you oversimplify Essential Complexity, you get

- higher coupling
- worse defect rate / maintainability

Can be however reduced through

- Abstraction
- Composition
- Changing the Context Boundaries

ACCIDENTAL COMPLEXITY

- **added** during the Analysis, Design, Implementation

  Can, and must be removed as much as possible

> *Simplicity --the art of maximizing the amount of work not done-- is essential.*

Principles behind the Agile Manifesto

IMPROVE OUR CODE

It takes a Deliberate approach and constant effort

> *To complicate is easy, to simplify is hard To complicate, just add, everyone is able to complicate Few are able to simplify*

Bruno Munari

COMPLEXITY VS DESIGN VS UNDERSTANDING

- Encapsulation, Cohesion, Decoupling, Abstraction are important because... they let us **think** about a single aspect at a time

KEY IDEAS FROM EVENT STORMING & DOMAIN DRIVEN DESIGN

TOPICS

Key ideas from Event Storming & Domain Driven Design

- Ubiquitous Language
- focus on the Domain
- "Big Picture" vs local consistency (Bounded Contexts)

DOMAIN DRIVEN DESIGN

Key ideas:

- focus on the Application Domain
- strong Domain Expert / Developer collaboration
- (Multiple) Consistent Models
  - Bounded Contexts

UBIQUITOUS LANGUAGE

- agree on a single definition / word per concept
  - split concepts whenever it is needed
- always and consistently use it in
  - code
  - documents
  - tests
  - talk

WHY UBIQUITOUS LANGUAGE

- reduces ambiguity
- enables Developer - Domain Expert collaboration
  - or at least review and validation by the Domain Expert

MODEL-DRIVEN DESIGN

We think **Model-Centered** is better

> *All models are wrong. Some are useful*

- `Model <--> Code`
- the code should reflect the model as much as possible
- very strong consistency **within a Context**

AVOID MENTAL MAPPING

- finite "thinking slots"
- Mental energy is finite
  - attention over time
  - amount of information: 7 +/- 2

KEEP THE MODEL CONSISTENT AND UP TO DATE

- continuous refinement
- continuous learning
- progressively more explicit

EVENT STORMING IDEAS

- Focus on Domain Events
  - things that happen

A great way to kickstart the Domain Model discovery

https://www.slideshare.net/ziobrando/50000-orange-stickies-later

THE TRAVEL EXPENSES KATA

TOPICS

The Travel Expenses Kata

- what is a Kata?
- overview of the Kata
- basic requirements

WHAT IS A KATA?

- Learn by repeating a known track
- but **trying to make it better** every time

Deliberate Practice means

- iterate small skills until >90% perfect
- focus on improving a specific aspect at a time

https://jamesclear.com/deliberate-practice-theory

OUR KATA

Approach

- focus on the business logic
  - limited UI (only a TXT report)
- see how new requirements impact our design

GENERAL REQUIREMENTS

Design on paper the macro-structure of an Expense Report Validator Use Case

- load a single Request from JSON
- (in a second phase, also load the Validation Rules)
- validate one or more conditions (e.g. amount limits)
- present a TXT or HTML report showing validation results

With the goal of letting the user take a decision and change the status of the request

HAVE A LOOK AT THE INPUTS...

SAMPLE VALIDATION REPORT

```
Employee: Carlo Bonamico
Month: March
Year: 2017

Expenses: 3

Date      Category  Requested Amount  Status  Allowed Amount
10 March  food                  10.5  NODOC   0
10 March  food                  35.7  PARTIAL 25
10 March  taxi                  10.2  OK      10.2
...
March     TOTAL                 66.4          35.2
Warnings
- expenses with no document
Blocking Errors
- overall total > 200 euro
```

APPROACH

- Do everything **incrementally**
- start from the basics

LAB 1- MODEL STORMING

- define on paper the main Domain Elements
  - input data (e.g. Request...)
  - output data (e.g. Report..)
  - processing steps/policies
- define their properties
- define their relationships
- define main operations

focus on the application domain

pay attention to the **Ubiquitous Language**

DISCUSSION

- which new *"implicit"* entities appeared in the model?
- what is the main operation?
- what are the main steps of this operation?

LAB 2 - IDENTIFY AN IMPLEMENTATION ROADMAP

In 5-10 minutes

- how can we split the design and implementation in
  - phases
  - sub-tasks

HOW TO START?

It's difficult!

The general vision is needed, but we must implement it incrementally

make it smaller!

MAKE IT SMALLER: ASK YOURSELF QUESTIONS

- what if instead of X Y Z we only do X?
- A & B -> A then B
- top down vs bottom up

http://agileforall.com/resources/how-to-split-a-user-story/

CONTINUOUS CHAIN

- Faster small steps beat slower bigger steps
  - concept of **One Piece Flow** in Lean
- also easier to parallelize
- The smaller the better

LAB 3 - FIND THE "MINIMUM VIABLE IMPLEMENTATION"

- choose a single case to process, from end to the end
  - End-to-end means Request loading to Report output
- keeping the input data
- a simple case
- a **representative** case

A CHAIN OF SAFE STEPS

A complex system that works evolves from simpler systems that works

John Gall

https://signalvnoise.com/posts/1414-a-complex-system-that-works-is-invariably

- you need to be able to check that everything works
- review the model frequently
- run frequently
- test frequently

WALKING SKELETON

- entire application / workflow structure
- made of empty (or logging-only) components
- incrementally filled-in / fleshed out
- also useful for testing
- *in-app mocking*

HOW TO KEEP TRACK OF WHAT YOU DO AND WHAT'S MISSING

- Write it down
- comment it with temporary comments
- code it!
  - write the API you ideally would like to have

LAB 4 - OUTLINE AN IMPLEMENTATION PLAN

- Define the main structure
- Split in sub-tastks with post-its
- Discuss the optimal order
- Introduce mock / support steps

SHARED PLAN - PHASE 1

- Parsing Json -> Minimal (always ok) Validation -> Generate Report
  - generate empty Report
  - generate report with the number of expenses header
  - generate report with the first expense and allowed amount = amount
- Validate documentation present of the first expenses
  - status in expense line

PHASE 2

- add warnings at the bottom
- Validate amount limit on single expense
- iterate validations on all expenses
- Compute amount total and allowed amount total (introduce aggregation)
- implement blocking validations ???
- implement monthly overall limit
- implement monthly limit per Category

ADVANCED FEATURES

- Implement limit per day (Optional)
- implement daily limit per Category (Optional)

INCREMENTAL COMMITS

- Each commit should start from a stable state and lead to a stable but more complete state

LINK: CONTINUOUS DELIVERY

http://continuousdelivery.com/

MORE DESIGN PRACTICE AND KATAS

- Elefant Carpaccio
  - https://docs.google.com/document/u/1/d/1TCuuu-8Mm14oxsOnlk8DqfZAA1cvtYu9WGv67Yj_sSk/pub

KEY FORCES IN SOFTWARE DESIGN

TOPICS

Key forces in Software Design

- Cohesion
- Coupling
- Impact of Change
- fragile vs robust designs

DESIGN PRINCIPLES

Basically, Common Sense applied to Software Design

Treat your code like your kitchen

C.B., about 2013

easy in the real world...

practice "seeing this way" code and abstract concepts

COHESION

Things which

- are related
- are used togehter
- change for the same reason
- change at the same time need to stay together (or at least nearby)

Think **forks and knives**

Impacts cost of changing one element

COUPLING

Couple related things, decouple unrelated things Thinks that

- are NOT related
- are NOT (necessarily) used togehter
- change for the DIFFERENT reasons
- change at DIFFERENT times need to stay separate

Think **forks and milk**, or **bread and socks**

Impacts propagation of changes from one element to the others

ENCAPSULATION

- Objects have a clear **Outside** and **Inside**
- helps minimizing the impact of change
- helps reducing coupling

IDEA

Make code easy to **delete**

- https://programmingisterrible.com/post/139222674273/write-code-that-is-easy-to-delete-not-easy-to
- https://18f.gsa.gov/2016/06/24/5-lessons-in-object-oriented-design-from-sandi-metz/

A "BAD" EXAMPLE

How does our code become unmanageable? A practical example

- fast-forward demo through the life of an (apparently) trivial function

PARSING EXPENSES FROM CSV

```
10/01/2015, 10.50
11/01/2015, 8.50
12/01/2015, 5.50
15/01/2015, 8.50
```

THE RESULTS

120 Lines, and already unmaintainable

labs/clean-code-expenses-ugly

THE EFFECTS:

- simple changes impact most of the codebase
- code-writing time -> decreases
- application-ready time -> never done
- time needed for bug fixes and new features -> increases

WHY?

- reading code vs writing code
- understanding effort
- fragility due to interdependence
- Symptoms of Rotten Design
  - http://www.objectmentor.com/resources/articles/Principles_and

WHAT CAN WE DO ABOUT THAT?

Clean Code, Design Principles and Lean to the rescue

- improving our code
- improving our design
- iterate
- practice, practice, practice and continuous / daily improvement (Kaizen)

LAB 5 - IMPLEMENT THE FIRST STEPS

paying attention to Cohesion and Coupling

- Parsing Json -> Minimal (always ok) Validation -> Generate Report
  - generate empty Report
  - generate report with the number of expenses header
  - generate report with the first expense and allowed amount = amount

SYNCHRONIZATION POINT

You should have at least classes for these concepts:

- loading and parsing the Request
- validator process
- validation result
- generating output txt from validation result

WORKS != DONE

If you identified initially 1-2 classes, take some time to split them

And name them well

DO YOU KNOW THIS MAN?

Google "Ignaz Semmelweis"

IT IS *NOT* BRAIN SURGERY

- Ignaz Semmelweis
  - http://www.npr.org/sections/health-shots/2015/01/12/375663920/the-doctor-who-championed-hand-washing-and-saved-women-s-lives
  - http://semmelweis.org/about/dr-semmelweis-biography/
- He championed washing hands before childbirth and surgery

CLEAN CODE AND BASIC DESIGN PRINCIPLES

- cannot solve all development problems...
- But can make them way more tractable!

especially if applied consistently.

Never underestimate the impact of doing something all the time

The power of compounding many small changes *in the same direction*

THE BOY SCOUT RULE

Leave the campsite a little better than you found it

Every time you touch some code, leave it a little better

SINGLE RESPONSIBILITY PRINCIPLE FOR METHODS

TOPICS

Single Responsibility Principle for methods

- separing inputs from outputs
- Primitives vs orchestrators
- if you have to do 3 things, make 4 functions
- Steps vs Flow

SINGLE RESPONSIBILITY

Each function should do 1 thing

Or even better, have a single responsibility

- and reason to change

As with all Design Principles, this is more of a **compass direction** than an absolute rule

HOW TO FIND RESPONSIBILITIES?

Ask yourself questions...

- What?
- Who?
- When?
- Why?
- Where?

And put the answer in different sub-functions

3 THINGS, 4 FUNCTIONS

If your function needs to perform a non-trivial task:

- import data, transform it and store it in the DB

Instead of

```
readData(){
  file.open();
  while(..)
    {
      line = readLine();
      obj = trasformLine(line);
      saveInDB(obj);
    }
}
```

what's better?

3 THINGS, 4 FUNCTIONS

```
importData(){
  data = readData();
  obj = transformData (data);
  saveInDB(obj);
 }
```

- a function for each step
- a function to call the steps

LAB 6 - REVIEW YOUR METHODS

- Particularly the validate() and generate() functions
- Split the methods in elementary responsibilities

SYNCHRONIZATION POINT

At this point you should have at least separate methods for

- validating a single expense
- generating the header
- generating the validated expenses table
  - generating a single line of the output
- generating the footer

PRIMITIVES, ORCHESTRATORS, LEVEL OF ABSTRACTION

- Primitives: small, focused, typically use-case independent
- Orchestrators: implement use-cases by combining primitives
- rinse and repeat over multiple levels of abstraction
- benefits:
  - more reusable
  - easier to test
  - easier to understand

STEPS VS FLOW

- Another example: have methods for each step of an algorithm
- another method to decide the flow among them

LAB 7 - ADD MORE VALIDATIONS

- Validate documentation present of the first expenses
  - status in expense line
- add warnings at the bottom (Optional)

SINGLE RESPONSIBILITY PRINCIPLE FOR CLASSES

TOPICS

Single Responsibility Principle for Classes What is a Responsibility

- reason to change
- what if ...
- looks similar vs changes for the same reason

SINGLE RESPONSIBILITY PRINCIPLE

Have you ever seen your grandmother put dirty clothes in the fridge?

Or biscuits in the vegetable box?

So, why to we do this all the time in our code?

SINGLE RESPONSIBILITY PRINCIPLE

Responsibility == **reason to change**

SRP - AGAIN

A class should do one thing

- and have a single reason to change

Consequences:

- classes should be small
- classes should be focused
- classes need to collaborate to perform complex tasks

LAB 8 - COLOR THE RESPONSIBILITIES

- Take the biggest class written up to now or any other code example
- Paste it in word / Google Docs
- Outline in different colors the various responsibilities

LAB 9 - VALIDATE ALL INDIVIDUAL AMOUNTS

- Validate amount limit on single expense
- iterate validations on all expenses

Question: where do we take the limit from?

REVIEWING YOUR CODE

Look at parts of the code and ask yourself what if that changes?

SYNCHRONIZATION POINT

At this point you should have separate classes for

- validating the amount of a single expense
- validating the documentation presence
- iterating on all expenses

LOOKS SIMILAR VS CHANGES FOR THE SAME REASON

https://web.archive.org/web/20090411030053/http://threeriversinstit
p=104

Sandy Metz https://www.sandimetz.com/blog/2016/1/20/the-wrong-abstraction

SINGLE RESPONSIBILITY MEANS SPLITTING ...

TOPICS SINGLE RESPONSIBILITY MEANS SPLITTING ...

- UI from logic
- Logic from persistence
- I/O from logic
- Sync from Async
- Intent from implementation

LAYERED ARCHITECTURE

- each layer depends only on lower layers
- ideally, with the domain layer at the bottom
  - *Hexagonal Architecture*

TOPICS

Reviewing your Design

- naming
- model out loud
- validate with examples
- what if this changes?
- the Open / Closed Principle

CONTINUOUSLY REFINED MODEL

- Knowledge crunching
- maintaining Model consistency
- continuous learning
- breaktroughs
- making implicit explicit

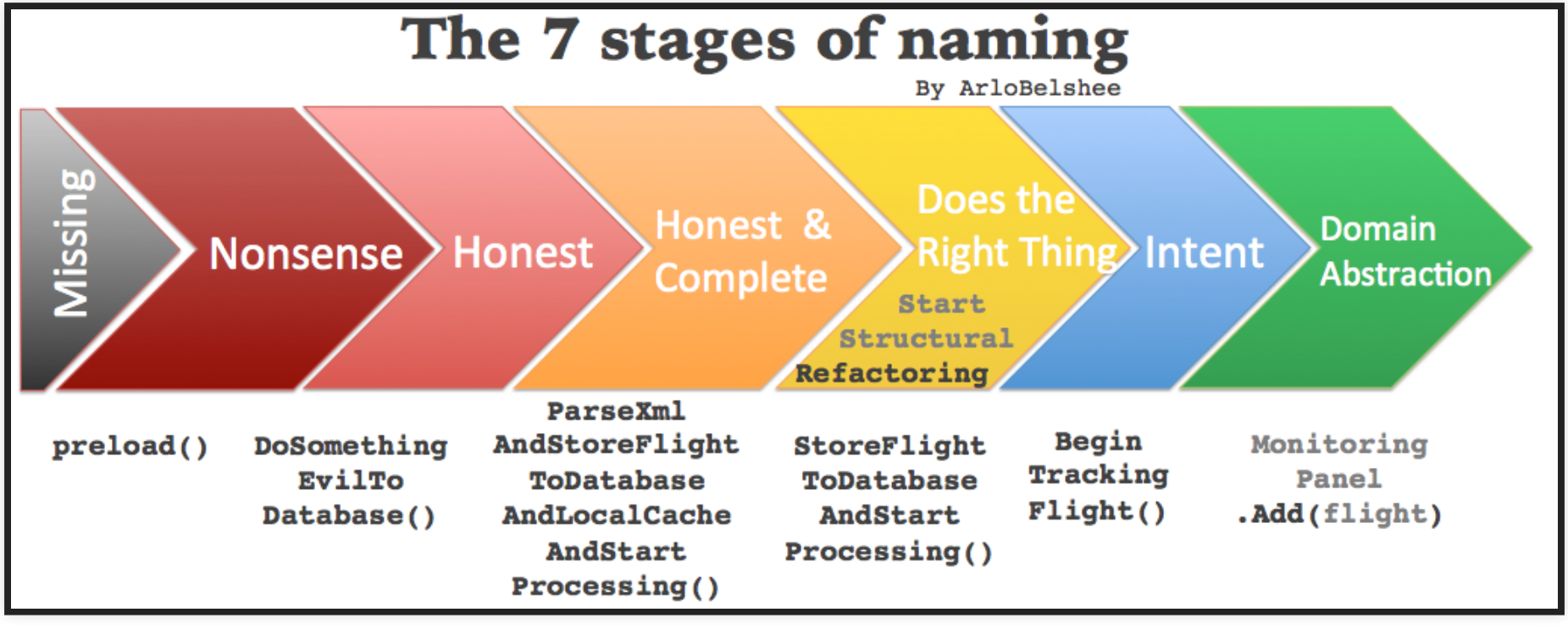
READING CODE VS WRITING CODE

> *What is written without effort is in general read without pleasure.*
>
> *Samuel Johnson*

Most code is written once, but read

- every time you need to fix a bug
- to add new features
- by other developers
  - including your future self

WHAT IS A GOOD NAME



- nonsense / honest / honest & complete / does the right thing / intent
- domain abstraction

http://llewellynfalco.blogspot.it/p/infographics.html

LAB 10 - REVIEW YOUR DESIGN

- check naming
- check if you can make implicit concepts visible

TOPICS

Collaborating with other classes

- Prefer Composition to Inheritance
- Dependency Injection

COLLABORATING CLASSES

- We need a way of making collaboration easier
- With Dependency Injection
  - separate creation of classes from linking instances
  - create A
  - create B
  - something else passes or **injects** B into A

DEPENDENCY INJECTION

Goal: using capabilities from another object should be almost as easy as calling a method in your object

- You do not (necessarily) need a framework for that...

LAB: SEPARATE CLASS INITIALIZATION AND WIRING

INHERITANCE - WITH CAUTION

- Inheritance is the strongest link between classes
- useful with caution

PREFER COMPOSITION

- combine parts
- a derived class becomes the composition of a base behaviour + additional custom behaviour

Achieve complex structures, behaviours, interactions by **composing** and **coordinating** simple elements

LAB: MONTHLY EXPENSE LIMITS

Compute amount total and allowed amount total (introduce aggregation)

- implement monthly overall limit
- delegating aggregation to dedicated class

Validate aggregate properties = Aggregate + Validate result

- Define the main structure
- Split in incremental sub-tastks
- Discuss the optimal order

INTERFACES AND CONTRACTS

- explicit vs implicit
- Decoupling changes and detecting regressions
- separate
  - clean parts from dirty code
  - Intent from Implementation

TYPES OF INTERACTION

- what is effective in the small
  - e.g. everyone talking at the same time in a room to solve a problem
  - interaction within local variables

chaos if same approach applied in the large

- anyone talking to anyone
- global interactions

LAB: GENERALIZING THE VALIDATION PROCESS

- Introduce the Validator interface
  - inputs
  - outputs
- Define a list of individual validators
- separate the validators from their application to the expenses

OCP

Open for extension, Closed for Modification

LAB: VALIDATING MONTHLY CONSTRAINTS PER CATEGORY

- implement limit per category
- Implement limit per day (Optional)

LAB: IMPACT ANALYSIS

- Revise the expenses lab in the light of these concepts
- Consider the impact of the following additional requirements
  - allow the operator to manually modify the allowed reimbursement
  - enable the `Approve` button only if there are no blocking errors

LAB: VALIDATING EXPENSE LIMITS PER DAY

- implement daily limit per Category (Optional)

WHAT WE HAVE LEARNED

Generalizing the model

- making the implicit explicit
- Levels of abstraction vs Levels of implementation
- breakthroughs
- Emergent Design

LAB: MAKING VALIDATION RULES CONFIGURABLE

Introduce the concept of Validation Rule

- implement it
- istantiate it in code

LAB

- load validation rules from file

HOW TO DO EVERYTHING INCREMENTALLY

- You can do *everything* incrementally
  - decouple release from deployment
  - branch by abstraction
  - do both
  - expand-contract

DECOUPLE RELEASE FROM DEPLOYMENT

EXPAND-CONTRACT

Refactor from A to B

- First do A
- then do A + part of B
- then do B
- then remove A

CONCLUSION

TESTABLE CODE AND GOOD DESIGN ALIGMENT

Things that make code testable

- clear interfaces
- small classes / functions
- decoupled
- composition

Things that make code well-designed, easy to evolve

- clear interfaces
- small classes / functions
- decoupled
- composition

THE PRINCIPLES

So what did we just do? Understand the principles

- the relationship between quality and productivity
- the need for a continuous chain of small, safe steps of design & implementation

QUALITY VS PRODUCTIVITY

Traditionally, Quality is seen as an alternative to raw development Speed

- this is partly true only in the short term

Four quadrants:

- high quality, high productivity -> tends to further improve
- high quality, low productivity -> tends not to improve, and go to
- low quality, low productivity -> tends to get worse
- low quality, high productivity -> tends to go to the previous one
- Productivity curves at different quality ratio

HOW TO CONTINUE BY YOURSELF: REFERENCES FOR FURTHER LEARNING

- improving our design
- practice, practice, practice and continuous / daily improvement (Kaizen)
- Principles of Package Design
- More on TDD: Test-friendly vs Well-designed

IMPROVE OUR CODE

It takes a Deliberate approach and constant effort

If I don't practice for a day, I notice

If I don't practice for two days, my orchestra notices

If I don't practice for three days, the public notice

Claudio Abbado

- practice, practice, practice and continuous / daily improvement (Kaizen)

PRINCIPLES OF PACKAGE DESIGN

The same concepts can be applied at the package level

- Java package / .Net namespace
- Java JAR / .Net assembly
- https://en.wikipedia.org/wiki/Package_principles

See Robert C. Martin

- https://vimeo.com/68236438

MORE PRACTICE AND KATAS

- http://matteo.vaccari.name/blog/tdd-resources
- https://github.com/xpmatteo/simple-design-in-action/tree/master/stage-01-hello-world/script
- http://codekata.com/
- https://www.industriallogic.com/blog/modern-agile/

LEARNING TO LEARN

- Kathy Sierra
- https://www.youtube.com/watch?v=FKTxC9pl-WM

EVENT STORMING

http://www.slideshare.net/ziobrando/event-storming-recipes
https://www.youtube.com/watch?v=veTVAN0oEkQ
http://www.slideshare.net/ziobrando/idea-stickies-green-bar-wroclaw-edition

TEAM IMPROVEMENT

About how to improve teamworking and raise technical proficiency and autonomy

The phoenix Project http://itrevolution.com/books/phoenix-project-devops-book/

Drive (sulla motivazione) https://books.google.it/books?id=E0H_DIkg0I4C

http://www.davidmarquet.com/

MODULE

References

TO LEARN MORE

- Online tutorials and video trainings:
  - https://cleancoders.com
- Full lab from my Codemotion Workshop
  - https://github.com/carlobonamico/clean-code-design-principles-in-action

HOW TO CONTINUE BY YOURSELF: REFERENCES FOR FURTHER LEARNING

- Principles of Package Design
  - http://www.objectmentor.com/resources/articles/Principles_and
- More on TDD
  - http://matteo.vaccari.name/blog/tdd-resources
- Modern Agile
  - https://www.industriallogic.com/blog/modern-agile/
- Lean, Quality vs Productivity and DevOps
  - http://itrevolution.com/books/phoenix-project-devops-book/

JAVASCRIPT

- http://humanjavascript.com/
- http://javascript.crockford.com/
- http://yuiblog.com/crockford/
- Free javascript books
- http://jsbooks.revolunet.com/

THANK YOU

- Other trainings
  - https://github.com/carlobonamico/
- My presentations
  - http://slideshare.net/carlo.bonamico
- Follow me at @carlobonamico / @nis_srl
- Contact me carlo.bonamico@nispro.it / carlo.bonamico@gmail.com andrea.passadore@nispro.it

##

THANK YOU FOR YOUR ATTENTION

CARLO BONAMICO

@CARLOBONAMICO

CARLO.BONAMICO@GMAIL.COM

HTTP://MILANO.CODEMOTIONWORLD.COM

HTTP://TRAINING.CODEMOTION.IT/