Clean Code Notes - (Intro)

- Refactor mercilessly
- Maintenance is priority
- In scrum, recommended practice is that refactor is part of Done
- Book is about clean principles

Chapter 1

- Craftmanship

Knowledge + Work DO IT!

↳ patterns
↳ principles
↳ practice

- Later == Never
- What is clean code?

Def 1 (Bjarne S. - inventor of C++) ELEGANT & EFFICIENT

- LOGIC STRAIGHTFORWARD → HARD FOR BUGS TO HIDE
- MINIMAL DEPEDENCIES → EASY MAINTENANCE
- STRATEGY FOR ERROR HANDLING
- OPTIMAL PERFORMANCE
- Do 1 thing well
- wasted cycles are inelegant

Def 2 Dave thomas (Eclipse)

- others beside original author can enhance it
- meaningful names
- tests
- One way to do one thing
- minimal dependencies
- should be literate
  ↳ reference to Knuth literate programming

Def 3 Ron Jeffries - (Ext. Prog. Intalled)

- Run all tests
- No duplication
- Minimize entities (classes, methods , functions)
- look at if object or method is doing more than 1 thing
  ↳ use Extract method refactoring it (submethods showing how is done)

Example

- Find things in a collection
  ↳ db
  ↳ []
  ↳ map

_hide implementation

Def 4 Ward Cunningham (inventor of Ext. Prog.)

- Each routine returns what we expect
- Code make looks like language was made for the problem

Uncle Bob Definition

- Easy to read
- ~~Boy~~ Girl Scout → Keep clean
- rest of the book on details . . .

Chapter 2 - Meaningful Names

1. Use intention revealing names

E.g. What is being measured and unit of measurent

int elapsedTimeInDays

2. Avoid Disinformation

accountLi~~st~~   int l = 1

hp ~ unix command

3. Meaningful Distinction

ProductInfo   ProductData

4. Searchable Names

- MAX_CLASSES_PER_STUDENT
  instead of plain 7 "

- Single letter names only used as local var
  inside short methodes

" the lenght of a name should correspond
to the size of its scope "

5. Interfaces and implementation

~~I~~ShapeFactory   encode Interface or Implem?

6. single letter var for loops is fine
(small scope)

i, j, k

7. Class Names

avoid: Manager, Processor, Data, Info
(cannot be a verb)

use: noun phrase names customer, Account

8. Method Names

use: verb deletePage()

- if constructors are overloaded, use static factory methods with names describing args.

Complex fulcrumPoint = Complex.FromRealNumber(23.0);

9. One word per concept

- fetch, retrieve, get → pick 1 and stick with it

- controller, manager, driver in the same code base

On the opposite don't use same word for 2 ≠ concepts

add (parameters and returns must match)

create new value

insert/append → put into collection

10. Use SolutionDomain Names

AccountVisitor → pattern

JobQueue

11. Consider context

firstName, state, zipcode → create class Address

→ addrState, addrZipcode

↓

State used alone would be confusing

12. Shorter Names are generally better than long ones

- as long as they are clear
- Address is fine for name class (if we do not need to distinguish bet/ MAC addr port Addr...)

Chapter 3 - Functions

- Be small
  - Blocks within if/else/while should be one line long (probably a function call)

- Do one thing.
  ↳ rule: can we extract another function from it (not merely a restatement)

  ↳ sections within fc → doing +1 thing.

- One level of abstraction per Function

  - getHtml() high level abs.
  - Parser-render() intermediate abst.
  - append() low lvl abstraction

⇒ switch statements (or if/else)

↳ large, do N things

↳ avoid or bury in low level class
(use polymorphism)

⇒ SRP → should have 1 reason
to change

⇒ OCP (open closed principel)

must change when new types
are added.

- switch of type with fn
doing similar things :
page 38

↳ use Abstract Factory

Switch: Appear only once +
used to create polymorphic objects +
hidden behind inheritance relationship

- Function arguments

  0 : niladic

  1 : monadic

  2 : dyadic

- don't mutate arg, return.

- boolean args (flags UGLY

- Functions should have no side effects -

  ↳ create temporal coupling

- Command query separation

  set → only set, not check for attr existence.

- Exceptions over Error codes

~~if (delete(page) == E_OK)~~

issue is caller must handle error immediately

use try/catch

- try/catches:

extract body to functions of their own

* DRY (don't repeat yourself)

root of evil DUPLICATION!

How to do it? tell a story.

write → improve → write tests

& keep tests passing,

Chapter 4 - Comments

Good Comments

- Legal
- Informative (better rewrite)
- Explanation of intent (why decision)
- Clarification (e.g. obscure args)
- Warning of consequence (too long test to run)
- TODOs
- Amplification (e.g. importance of something apparently inconsequential)

Bad Comments

↳ Mumbling

↳ Redundant

↳ Misleading

↳ Mandated (e.g. mandatory docs)

↳ Commenting bad code

↳ Position markers (// Actions //////////)

↳ Commented code

↳ Too much info

↳ Function headers

Chapter 6 - Objects and Data Structures

Objects

expose behavior
and hide data

⇓

easy to add
new kind of objects
without changing existent behavior

Data Structures

expose data
have no behavior

⇓

hard to add new
data structures to
existent functions
that uses it.

Law of Demeter

a method f of a class should not invoke methods on objects returned by allowed functions.

↳ class of f

↳ object created by f

↳ an arg. of f

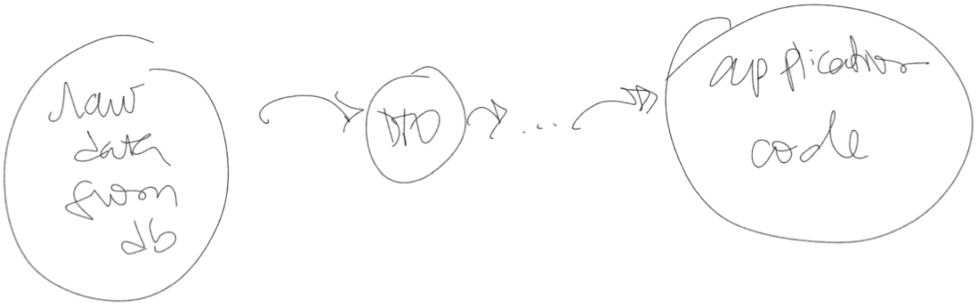
↳ object held in instance variable of C

- Hybrids = 1/2 object + 1/2 data structure

- Hiding Structure

DTOs (Data transfer objects)

- class with public values and no functions
- useful for comm with DB, parsing messages from sockets.
- special case: active records (have methods e.g. save, find) direct translation from db tb.



→ Conclusion:

choose between flexibility to add type or behavior "

Chapter 7 - Error Handling

- Write Try/catch/Finally before full implementation
- Provide context with exceptions (source & location of error)
- We can use exception classes, if there are times we want to catch one exception and allow otherone to pass
- Wrap 3rd party API code
  - ↳ easier to test
  - ↳ minimize depedencies
- Careful to not obscure logic with error handling: separate business logic and error handling.
- Special cases handling: SPECIAL CASE PATTERN (fowler)
- Don't return null from method
  - ↳ throw exception
  - ↳ return specialcase object
- Don't pass null to method
  - ↳ no good ways to deal with a null passed by accident, so the rational approach is forbid passing null by default.

Conclusion: clean code is readable, but it must also be robust.

nge interface, many provide more capability

```
public class sensors {
    private Map sensors = new HashMap();
     public Sensor getById ( id) {
            return (Sensor) sensors.get(id)
     }
```

- Do not use the Map in an interface boundary

- Learning tests / boundary tests: usage of the 3rd party - code

Conclusion: have a very few places in the code that refer to the 3.p.code.

Chapter 9: tests

- Keep tests clean, tests change as much as production code, should be easy to maintain

- Focus on readability

- Build / operate / check pattern

- single concept per test

Chapter 10 - Classes

- Should be small

- SRP (single responsability principle) class should have one reason to change

- Cohesion, methods and variables are co-dependent (each method uses each variables in ideal cohesion)

= OCP : open to extension, closed

modification