Software Guidelines for Non-Coders

Jaime Silvela

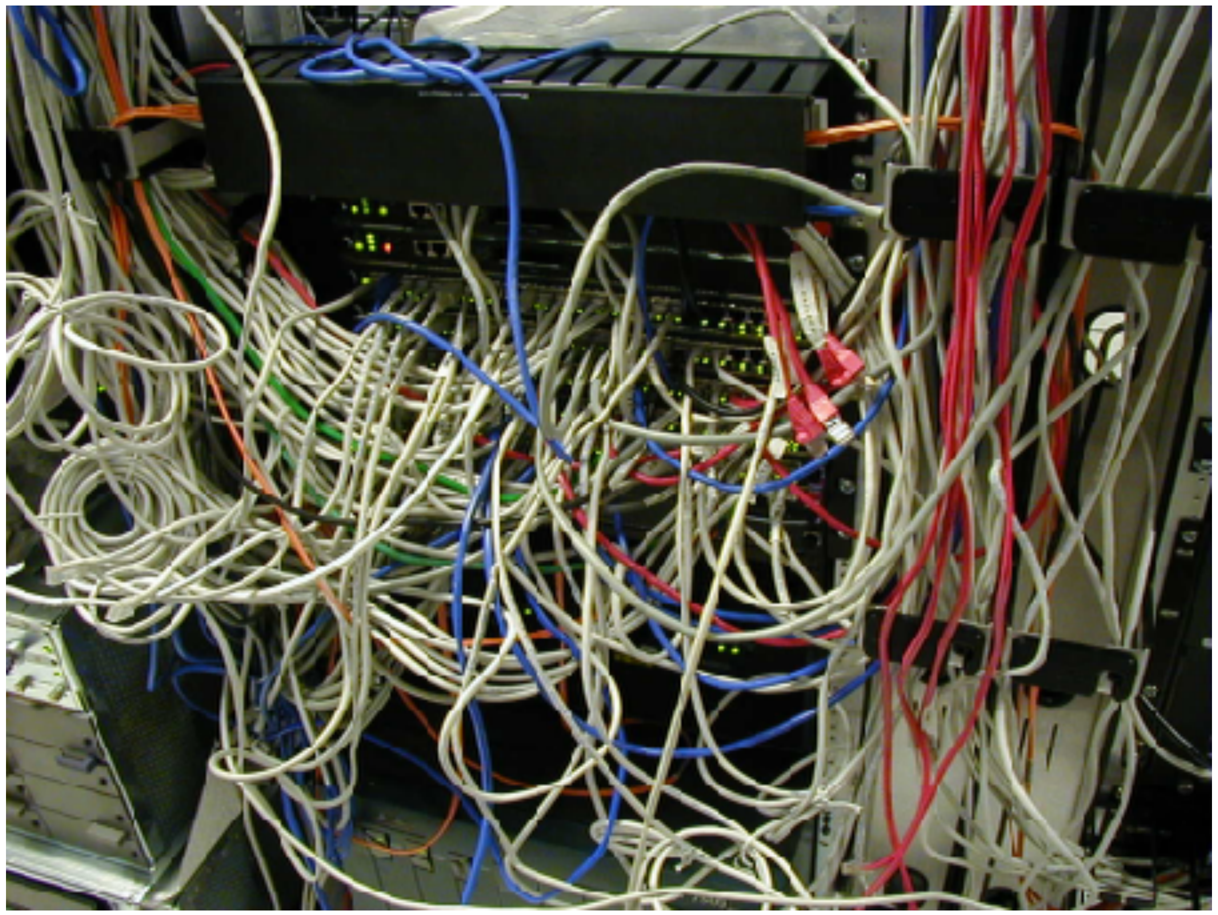
April 2017

0.- Admit you own software

- You wrote more than 10 lines of code, and some people rely on them.
- … you own software. Care for it, or it will grow chaotic.
- Be aware of rising **technical debt**.

0.- Admit you own software

Technical debt



Where we want to get

1.- Don’t be clever; be clear

“The best writing is rewriting.” — E.B. White

“Simplicity does not precede complexity, but follows it.” — Alan Perlis

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.” — Donald Knuth

1.- Don’t be clever; be clear

- It’s not about using the quickest algorithm or the fanciest technology.
- It’s about mapping the problem domain into code as clearly as possible.
- Keep clarifying and cleaning your program as it evolves.
- Give good names to things.

2.- Focus on your data structures

- Loops and conditionals are a done deal — like bouncing the ball in basketball.
- Your data structures drive your algorithms.

“Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won’t usually need your flowcharts; they’ll be obvious.”

Fred Brooks in *The Mythical Man-Month*
(Recommended book)

2.- Focus on your data structures

Example 1/3

- Design and algorithm to solve:
- Problem: given an array of numbers `nums`, and a radius `r`, cluster the numbers in order to get a (kind-of) sparse histogram with precision `r`.
- We don't want a proper histogram, because most entries in the histogram would be 0. We're likely to have fewer than 10 clusters, and want 1/100-th precision.
- Example: `nums` = [1.2, 1.3, 5, 5.1, 5.5], `r` = 0.2 ➔ 1.2 × 2, 5 × 2, 5.5 × 1

2.- Focus on your data structures

Example 2/3

People generally come up with complicated solutions. Code that backtracks, complicated program state, kernels, "matlabby" thinking…

2.- Focus on your data structures

Example 3/3

Now, define:

```
cluster {
    value float,
    count int
}
```

Restate the problem: we want algorithm `clusterNumbers`, such that

```
clusterNumbers(nums[], r) ➔ cluster[]
```

Imagine that we have a cluster array. If we are given a new number x to cluster, do we know what to do?

```
addNumToClusters(num, r, clusters[]) ➔ clusters[]
```

3.- Program with pen and paper

or marker and whiteboard
… some of the time

- De-focus from syntax and micro detail.
- Focus on high level, get a broad view.
- Iterate on design ideas in this medium.

4.- Diagram the flow of data in your program

cf. 2 and 3

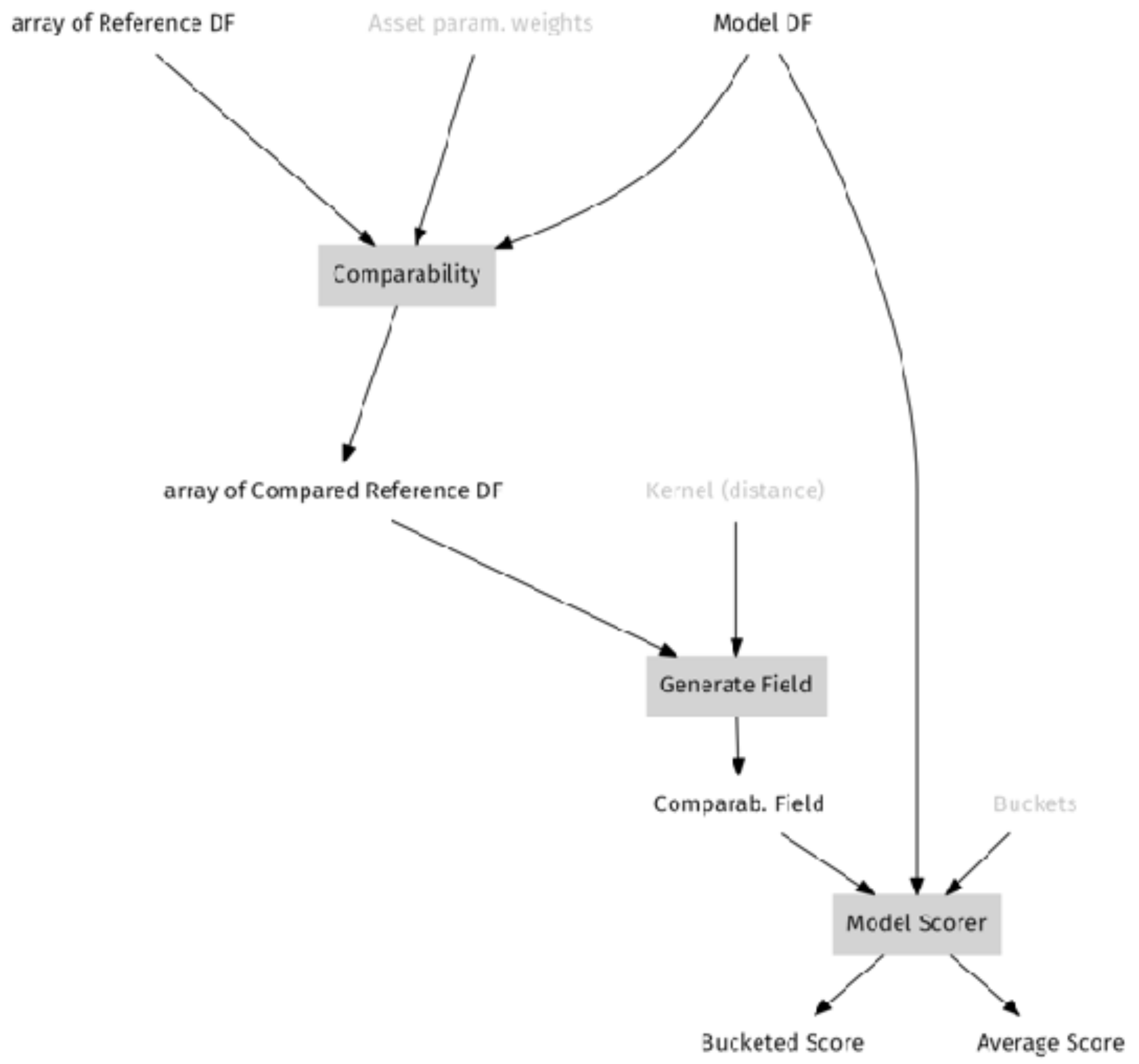
- Flow charts for loops/conditionals aren’t much use.
- Things like UML class diagrams, again, not that useful.
- Your data flow tells you what needs to happen where.

4.- Diagram the flow of data in your program

```
Model/Reference DF
 Metadata
 Asset type
 Location
 Curve: array of {
     Intensity
     Damage Ratio
 }


 Compared Ref. DF
  Reference DF
  Comparability


Comparab. Field
 array of {
      Intensity
      Damage Ratio
      Comparability
 }
```



5.- Separate interface from implementation

ie. think of modules

- Think of each of your modules as a service to be called by others.
- What is relevant for the users? This is your interface.
- Keep your interface small and clean.

"Each module is designed to hide [a design decision] from the others."

David Parnas in *On the Criteria to Be Used in Decomposing Systems into Modules*
(Recommended paper)

5.- Separate interface from implementation

Example 1/5

We're building a game with several kinds of enemy ships. We hire Giulio to write a relativistic motion enemy. We hire Nuria to write a magneto-hydrodynamic submarine enemy.

5.- Separate interface from implementation

Example 2/5

- Giulio's code:

```
func GravityAt(…) {…}
func Speed(…) {…}
…
```

- Nuria's code:

```
func MagneticFieldAt(…) {…}
func FriggingLaserBeam(…) {…}
…
```

5.- Separate interface from implementation

Example 3/5

The animation loop is very complex, and needs to know details of relativity and magneto-hydrodynamics.

```
func AnimationLoop() {
        forever {
                Sleep(deltaT)
                canvas = clearCanvas()
                for foe in AllFoes {
                        if foe is Relativistic {
                                foe.GravityAt(…)
                                …
                        } else if is MagnetoHydrodynamic {
                                foe.FriggingLaserBeam(…)
                                …
                        }
                }
        }
}
```

5.- Separate interface from implementation

Example 4/5

Let's do it another way. The animation loop requires only that enemies be able to compute their next state after some time has elapsed, and that they be able to draw themselves on a canvas.

- Giulio's new code:

```
func NextState(dt time.Interval) {…}
func DrawOn(canvas Canvas) {…}
— Hidden —
func GravityAt(…) {…}
…
```

- Nuria's new code:

```
func NextState(dt time.Interval) {…}
func DrawOn(canvas Canvas) {…}
— Hidden —
func MagneticFieldAt(…) {…}
…
```

5.- Separate interface from implementation

Example 5/5

- The new animation loop.

```
func AnimationLoop() {
    forever {
        Sleep(deltaT)
        canvas = clearCanvas()
        for foe in AllFoes {
            foe.NextState(deltaT)
            foe.DrawOn(canvas)
        }
    }
}
```

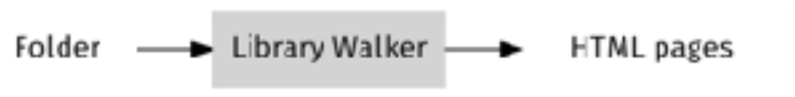
Everything is more robust. Nuria can change her implementation details if she wants, as long as she upholds the interface. Giulio can use his code for another simulation that expects the same interface.

6.- Separate data from presentation

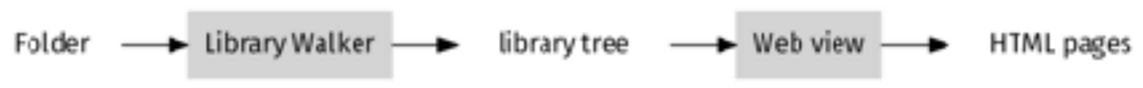
- The core of your programs is the problem domain’s data and logic.
- Your presentation layer (UI’s, PDF’s …) should be detached from domain logic.
- Do the main computations in domain logic, have “dumb” presentation code.

6.- Separate data from presentation

Conflated data and presentation



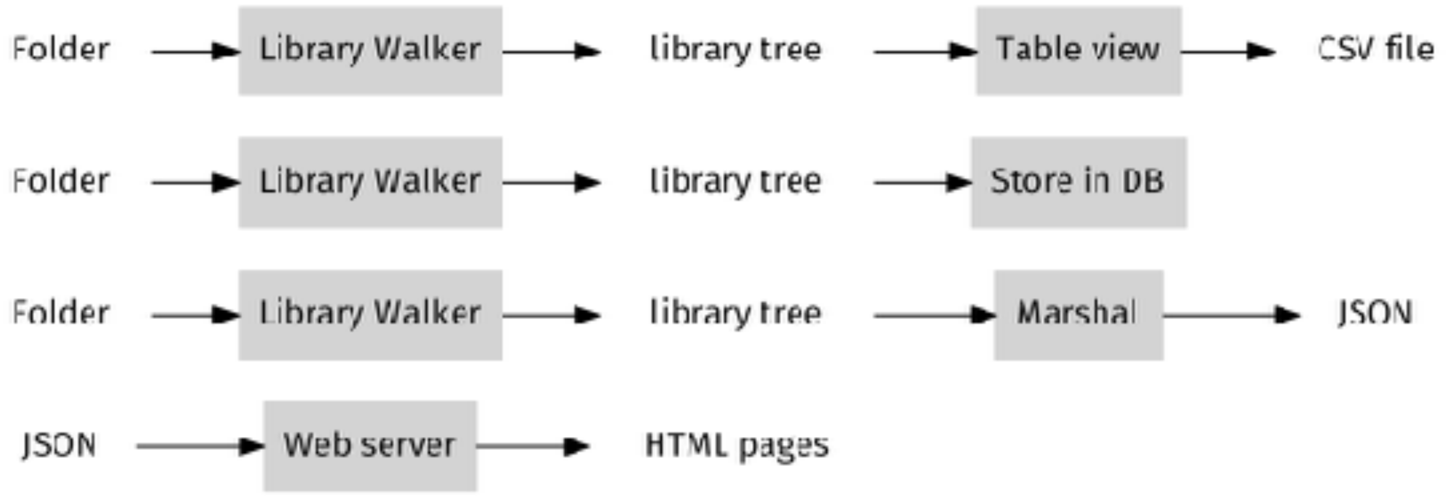
Decoupled



6.- Separate data from presentation

Decoupling opens up many possibilities.

We can even handle parts of the process at different times, in different workflows, written in different languages.



7.- Inject your functional dependencies

- Don’t connect to a DB or other service in the middle of your code.
- Every function/object that uses an external service should name it as a parameter.
- I.e. the dependency gets injected.
- You should establish service connections very visibly in one place.

7.- Inject your functional dependencies

Don’t do this.

What if we want to run our program, only connecting to a different database?

If we want to write tests, will we be querying / modifying the production database?

If we want to connect as a different user/ password, we need to find all points in our programs that connect to the DB.

```
func someStuff(…) {
       …
       dbConn = openDB("dbServer",
              "username",
              "password")
dbConn.Query("my SQL query")
       …
}

func moreStuff(…) {
       …
       dbConn = openDB("dbServer",
              "username",
              “password”)
         dbConn.Query("another query")
       …
}
```

7.- Inject your functional dependencies

Do this instead.

As a side benefit, the function signature will provide clear documentation that a function depends on the database.

```
func someStuff(dbConn) {
        …
        dbConn.Query("my SQL query")
        …
}

func otherStuff(dbConn) {
        …
        dbConn.Query("another query")
        …
}

func main() {
        …
        dbConn = openDB("dbServer",
                "username",
                "password")
        …
        someStuff(dbConn)
        …
        moreStuff(dbConn)
        …
}
```

8.- Test in code. Don’t overdo it

- Test domain logic.
- Test interfaces.
- Testing low level implementation detail is not as clearly beneficial.

James Coplien on unit testing

9.- Limit your external dependencies

- You may like an external library that makes something easier for you.
- In a few months, it may be incompatible, or vanished.
- Try to depend only on well established libraries.
- If you need only a tiny fraction of an external library, think of copying.
- You may want to store your dependencies as part of your code repository.
- Three fallacies of dependencies

10.- Use good tools

- Use version control. Even for small projects. Even for documentation.
- Use a good programming text editor. Eg: Visual Studio Code, Emacs, Atom.
- Use a font meant for coding. `0 = O? 1 = l = I?` Eg. Consolas, Source Code Pro.
- Collect notes, bugs, TODO items, in a file(s) where you look often.
- If you’re in a team, share that info. Maybe JIRA? (awful but useful)

11.- A few gotchas

- Floating point numbers are tricky. **0.2 + 0.1 ≠ 0.3**
- Text encoding: try to read/store all text as UTF-8.
- Case sensitivity; most modern stuff is case-sensitive. But …
- Case in-sensitive: SQL*, Windows file names*, Fortran, Lisp.
- Line endings: Unix/Linux and Windows use different codes for newlines.
- “Smart” programs like Excel may modify your data.