Soft Skills

Soft Skills

- Focus on Deliver results
- Fix Bugs cleanly
- Deal with Technical Debt
- Play well with others
- Know how to write

Focus on Deliver results

Focus on Deliver results

- Results
- Focus
- Deliver smaller results often

Focus on Deliver results

What are results?

A result is something that makes a positive change in how the business works

- It can be a feature, bugfix, documentation, or an answer to *anybody needing insight*

Focus on Deliver results

What are not results?

- An intermediate deliverable
- A progress report
- Anything outside the Definition of Done (i.e. tested)

Focus on Deliver results

To keep focus on what you do

- Keep promises at minimum
- Take promises seriously
- Deliver small results more often

Fix Bugs cleanly

Fix Bugs cleanly

Repeatable process

1. Understand the problem
2. Write tests that **fail**
3. Solve the problem
4. Refactor
5. Commit

Fix Bugs cleanly
Understand the problem

Agile does not mean *just start coding*

- Navigate through the app to reveal the feature
- Ask the product owner or a colleague about the details of a feature or a bugfix
- Try to explain to them what you think you understood

Fix Bugs cleanly
Write tests that fail

- Do we have to lose time to write tests that fail?
  > Top developers do not ask for permission to do their jobs

Fix Bugs cleanly
Write tests that fail

It's a To Do list of the features

- Helps us check that we previously understood the problem
- Helps us identify corner cases

Solve the problem
...as quickly as you can

Goal: make sure that the problem **can** be solved

- Code only what solves the problem
- Use tests to know when you have solved it

Once done, the work is **not** done, so we don't need to report your progress

Fix Bugs cleanly
Refactor the code

Goal: Make the code readable and maintainable

- No copy-paste code
- Use *descriptive* variable names
- Coding style should *match* the previous one
- Make changes one at a time, re running tests to make sure it works

Fix Bugs cleanly
Refactor - Comments

- When we face a piece of code with comments, that is a place you might consider refactoring
- Most of the times, good code does not need comments (but not always!)
- Use comments to explain **why**, not **what**

Fix Bugs cleanly
Commit your changes

- Take special care to write a good message
- *First line*: What the change is
- *Other lines*:
  - A link to the issue or details about conversations about it.
  - Explain why you did things, especially if solution isn't perfect.

Deal with Technical Debt

Deal with Technical Debt

Technical debt is a concept in software development that reflects the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer.

– Wikipedia

What is not TECHDEBT === SLOP

- Copy-pasted code, repeated elsewhere
- Wrongly named variables/methods
- Unused code

If it works, should you ship this code?

What is not TECHDEBT === SLOP

- copy-pasted code, repeated elsewhere
- wrongly named variables/methods
- unused code

If it works, should you ship this code?

NO

It needs to be addressed now, before it infects the system

What is not TECHDEBT === SLOP

```
class Product {
    ...
}

class Order {
    let customer: Customer
    let discounts: Float
    let product: Product

    var isFree: Bool {
        return customer.storeCredit + discounts >= product.price
    }
}
```

What is not TECHDEBT === SLOP

```
class Product {
    let price: Float

    func isFreeFor(customer: Customer, code: DiscountCode?) -> Bool {
        let discount: Float = code?.discounts ?? 0.0
        return customer.storeCredit + discounts >= price
    }
}

class Order {
    let customer: Customer
    let discounts: Float
    let product: Product

    var isFree: Bool {
        return customer.storeCredit + discounts >= product.price
    }
}
```

What is not TECHDEBT === SLOP

```
class Product {
    let price: Float

    func isFreeFor(customer: Customer, discountCode: DiscountCode?) -> Bool {
        let discount: Float = discountCode?.discounts ?? 0.0
        return isPurchaseFree(customer: customer, discounts: discounts)
    }

    func isPurchaseFree(customer: Customer, discounts: Float) -> Bool {
        return customer.storeCredit + discounts >= price
    }
}

class Order {
    let customer: Customer
    let discounts: Float
    let product: Product

    var isFree: Bool {
        return product.isPurchaseFree(customer: customer, discounts: discounts)
    }
}
```

Why is SLOP bad?

- Duplicates behaviors
- Obscures the intent
- Needs more training for future devs
  - ++ Comments
  - ++ Tests

What is TECHDEBT?

- Code to fix later (or never)
- Future-features, but not now-features
- Tradeoffs taken in a controlled way

What is TECHDEBT?

```
class DiscountCode {
    let amount: Float
}

class Order {
    var discountsApplied: Float

    func applyDiscountCode(discountCode: DiscountCode) {
        discountsApplied += discountCode.amount
    }
}
```

What is TECHDEBT?

```
class DiscountCodeInteractor {
    func isDiscountCodeValid(_ discountCode: DiscountCode) -> Promise<Bool> {
        /* code to talk to external service
         * currently just checks if the code is valid and it wasn't used before
         * should use the country of a discount code to validate */
        ...
    }
}

class DiscountCode {
    let amount: Float

    func isApplicable(shippingAddress: Address) -> Promise<Bool> {
        /* TECHDEBT: we don't have promotions yet outside ES,
         * and DiscountCodeInteractor hasn't the ability to check
         * code based on its country, so we hardcode that for now
         * should be fixed before adding discounts for other countries */
        return DiscountCodeInteractor.isDiscountCodeValid().map({ isCodeValid in
            isCodeValid && shippingAddress.country.code == "ES"
        })

    }
}
```

Why is TECHDEBT good?

No one knows what will be the shape of the code in the future

- when (if ever) will the feature be needed
- what constraints will a feature meet

Why is TECHDEBT good?

No one knows what will be the shape of the code in the future

- When (if ever) will the feature be needed
- What constraints will a feature meet

...it allows to focus on what's important *now*
...it allows to ship one feature at a time (and only this)

Play well with others

Play well with others

We need a way to communicate with actual human beings
to do that, we need to:

- Empathize with others
- Adapt Information
- Abstract Information

Play well with others

Empathize with others

Understand people different from you:

- Managers
- Marketing
- Customers/App users

Play well with others

Empathize with others

- They don't know how to do what you do.

Play well with others

Empathize with others

- They don't know how to do what you do.
- They don't have the vocabulary to know what to ask.

Play well with others

Empathize with others

- They don't know how to do what you do.
- They don't have the vocabulary to know what to ask.
- They don't know what they don't know, so they make *wrong assumptions*.

Play well with others

Empathize with others

- They don't know how to do what you do.
- They don't have the vocabulary to know what to ask.
- They don't know what they don't know, so they make *wrong assumptions*.

... but they are **not** stupid

Play well with others
Empathize with others

They work on a totally different set of priorities

- We need to to understand them.
- We need to make ourselves understood.
- We have to push back when asked to solve the *wrong problem*.

...so we need to **adapt** terms and **abstract** concepts

Play well with others

Adapt Terms

- Avoid technical jargon
- Use longer descriptive sentences instead of jargon
- Do not talk down
- Listen carefully to others, ask if you're not 100% sure

Play well with others

Abstract Concepts

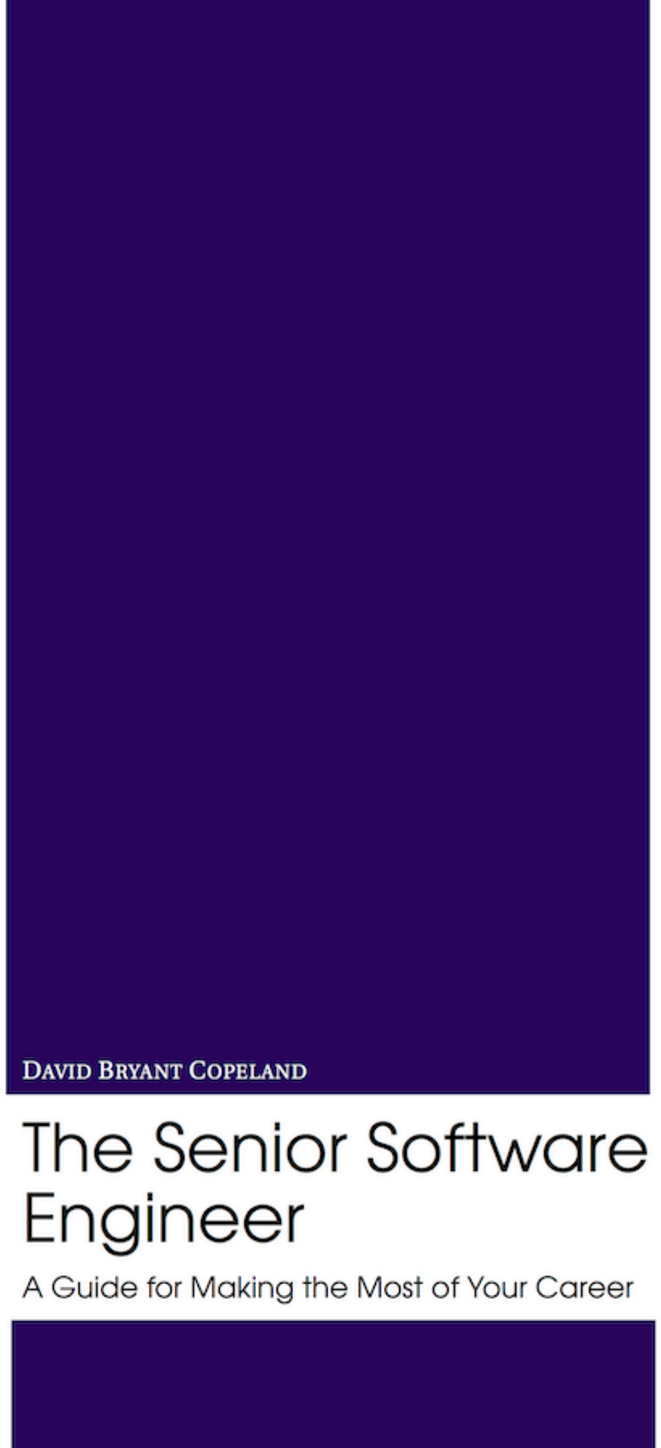
- Avoid technical details
- Explain things using analogies
- Use diagrams or sketches
- Repeat often the context of the discussion
- Be prepared to justify your position if challenged

Know how to write

We ultimately code not
for machines,
But for people to use,
Understand and make it
grow

Invest time in your team:

- Document throughly your Pull Requests
  - PRs with more than 300 lines of code
  - One line fixes for complex issues
- Explain the causes of your patch
- Make the documentation easy to index/search
  - Use Github



For more, please read "The Senior Software Engineer" by David Copeland