Secure Software Design

Andey Robins

Spring 23 - Week 8

Best Practice

Outline

- Clean Code
- Clean Coders
- Tools of the Trade
- Solid Workflows
- Refactoring Kata

Preface

Much of this may seem self-evident. However, most developers struggle to maintain these principles all the time. Just like one security oversight can pollute the security of an entire application, it only takes one location of unclean code to pollute the code base.

Future programming assignments will assume you're turning in clean code, potentially incurring deductions if the code doesn't follow clean principles. Now is a great time to start building the habit of refactoring your code before you submit it. This is a habit that will serve you well going into a software engineering career.

Scouts Rule

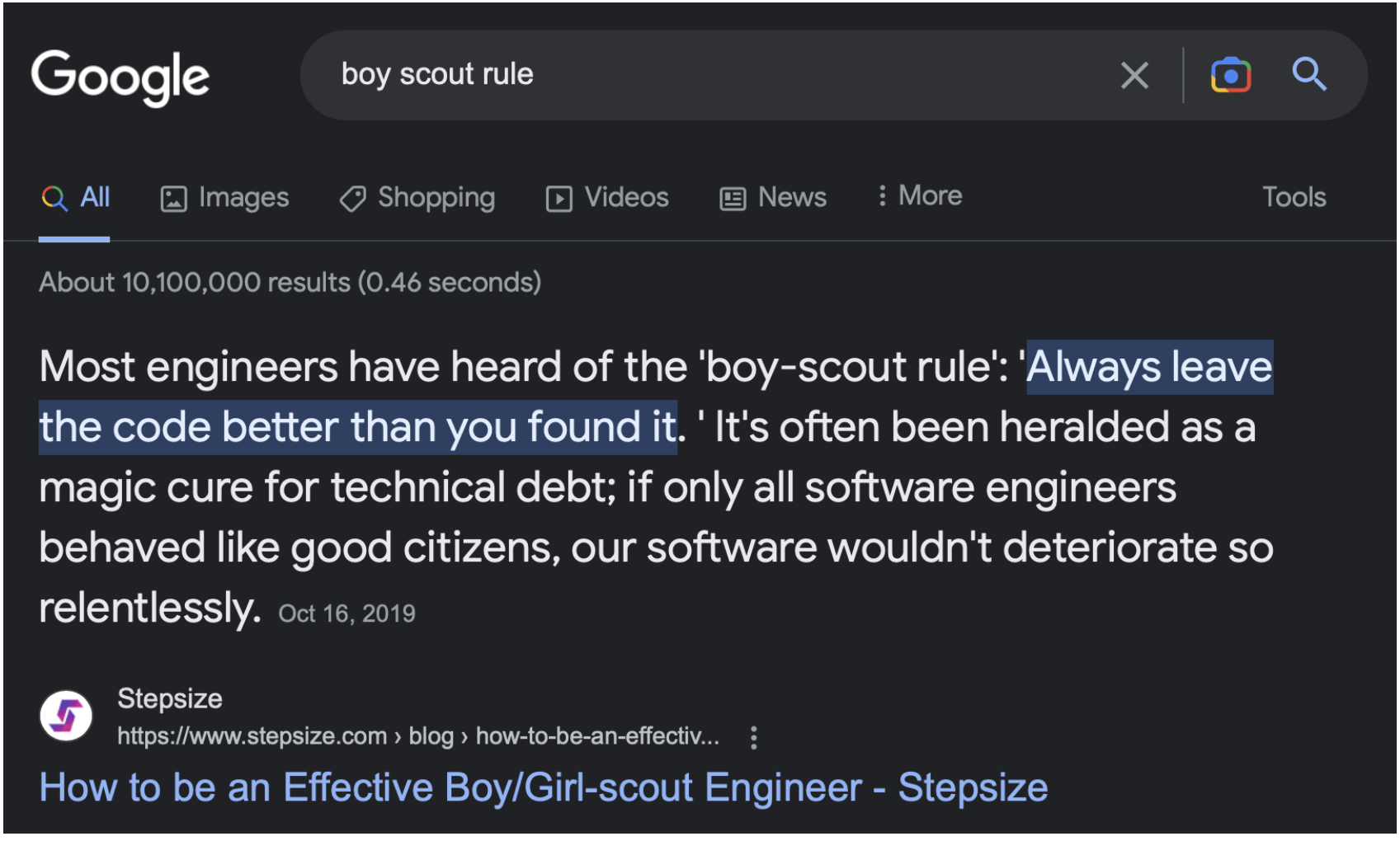

Figure 1: The Scouts Rule of Programing

Clean Code

Naming

1. Use intention revealing names
2. Avoid disinformation
3. Avoid encodings
4. Use one word per concept

Intentional Names

What is this code supposed to do?

```
const getThem: ((inList: Array<int[]>) =>
    Array<int[]>) = () => {

    list1: Array<int[]> = new Array<int[]>();
    for (x in inList) {
        if (x[0] === 4) {
            list1.push(x);
        }
    }
    return list1;
}
```

Is this better?

```
const getFlaggedCells: ((gameBoard: Array<int[]>) =>
    Array<int[]>) = () => {

    const FLAG = 4;
    flaggedCells: Array<int[]> = new Array<int[]>();
    for (cell in gameBoard) {
        if (cell[0] === FLAG) {
            flaggedCells.push(cell);
        }
    }
    return flaggedCells;
}
```

What can we still improve?

```
type Cell = int[];

interface Cell {
    isFlagged: () => bool,
}

const getFlaggedCells: ((gameBoard: Array<Cell>) =>
    Array<Cell>) = () => {

    flaggedCells: Array<Cell> = new Array<Cell>();
    for (cell in gameBoard) {
        if (cell.isFlagged()) {
            flaggedCells.push(cell);
        }
    }
    return flaggedCells;
}
```

Best can be language dependant.

```
type Cell = int[];

interface Cell {
    isFlagged: () => bool,
}

const getFlaggedCells: ((gameBoard: Array<Cell>) =>
    Array<Cell>) = () => {

    return [...gameBoard.filter(
        (cell) => cell.isFlagged()
    )];
}
```

Disinformation

Programmers must avoid leaving false clues that obscure the meaning of code. We should avoid words whose entrenched meanings vary from our intended meaning.

```
// incorrectly informs us that this is a list
accountList: AccountGroup = new AccountGroup();

// unclear what it holds
bunchOfAccounts: AccountGroup = new AccountGroup();

// both alright
accountGroup: AccountGroup = new AccountGroup();
accounts: AccountGroup = new AccountGroup();
```

Encodings

Encoding type or scope information into names simply adds an extra burden to deciphering.

Hungarian notation is called out as an example as well as the convention of `mVarName` for local scoping.

Instead of:

```
usernamesList := ["arobins", "mborowczak", "eseidel"]
for i, nameString := range usernamesList {
    login(nameString);
}
```

Do This:

```
usernames := ["arobins", "mborowczak", "eseidel"]
for i, name := range usernames {
    login(name);
}
```

Instead of:

```
class JobManager {
    jobs: []Job

    runJobs() {
        for (mJob in this.jobs) {
            mJob.run();
        }
    }
}
```

Do this:

```
class JobManager {
    jobs: []Job

    runJobs() {
        for (job in this.jobs) {
            job.run();
        }
    }
}
```

One Word per Concept

Modern editing environments like Eclipse and IntelliJ-provide context-sensitive clues, such as the list of methods you can call on a given object. But note that the list doesn't usually give you the comments you wrote around your function names and parameter lists. You are lucky if it gives the parameter names from function declarations. The function names have to stand alone, and they have to be consistent in order for you to pick the correct method without any additional exploration.

This is no longer true, but the principle still assists in lowering the required mental overhead to understand autocomplete suggestions.

Meaning and intention is unclear. We can refactor.

```
class RssLoader {

    fetch(url) {
        /// --- SNIP --- ///
    }

    get(url) {
        /// --- SNIP --- ///
    }
}
```

```
class RssLoader {
    // fetchChanges will get a list of updated
    // resources since the last fetch.
    fetchLatestChanges(url) {
        /// --- SNIP --- ///
    }

    // getResource will retrieve a single resource
    // that was previously fetched.
    getIndividualResource(url) {
        /// --- SNIP --- ///
    }
}
```

Functions

Functions should be:

1. Small!
2. Single Purpose
3. Have no Side Effects
4. DRY

Small Functions

Historically, we use “small” and “you can fit the entire thing on one screen” synonymously. In other words, if a function takes multiple screens to read, it becomes very difficult to reason about.

```
func Start(in io.Reader, out io.Writer) {
    scanner := bufio.NewScanner(in)
    env := object.NewEnvironment()

    for {
        fmt.Printf(PROMPT)
        scanned := scanner.Scan()
        if !scanned {
            return
        }

        line := scanner.Text()
        if line == "exit()" {
            return
        }
        lex := lexer.New(line)
        par := parser.New(lex)
```

```
            program := par.ParseProgram()
            if len(par.Errors()) != 0 {
                printParserErrors(out, par.Errors())
                continue
            }

            evaluated := evaluator.Eval(program, env)
            if evaluated != nil {
                io.WriteString(out, evaluated.Inspect())
                io.WriteString(out, "\n")
            }
        }
    }
```

This function is probably on the edge of what is “small enough,” but it can still be shortened even just by moving things around with DDD principles.

```
func Start(in io.Reader, out io.Writer) {
    scanner := bufio.NewScanner(in)
    env := object.NewEnvironment()

    for {
        Prompt(scanner, out)
        line := GetLine(scanner, out)

        lex := lexer.New(line)
        par := parser.New(lex)

        prog := MustParseProgram(lex, par, out)
        MustEvaluateProgram(prog, env, out)
    }
}
```

DRY

"*The DRY principle is a **best practice** in software development that recommends software engineers **to do something once, and only once.**" - Laura Fitzgibbons via WhatIs.com*

I believe this one is probably pretty self evident. It's how we frame teaching functions to new programmers.

Single Purpose

"***FUNCTIONS SHOULD DO ONE THING. THEY SHOULD DO IT WELL. THEY SHOULD DO IT ONLY.***"

Perhaps a little aggressively typeset, but the idea is clear. Functions should have one job. This is derived from the idea that we write functions to decompose a larger concept.

What are the Side Effects?

```
class UserValidator {
  cryptographer: Cryptographer;

  checkPassword(uname: string, pass: string): bool {
    let user: User = UserGateway.findByName(uname);
    if (user !== undefined) {
      let codedPhrase: string = user.getPhrase(pass);
      let phrase: string = this.cryptographer.decrypt(
        codedPhrase, pass
      );
      if (phrase.isValid()) {
        Session.initalize();
        return true;
      }
    }
    return false;
  }
}
```

No Side Effects

The `checkPassword` function doesn't give us any indication that it will initialize a session. In the case of the code from the previous slide, there is a side-effect of setting up a session which could be incorrectly triggered by someone just trying to check a password.

Aside: Exceptions

Clean Code contains the assertion that you should "Prefer exceptions to error codes." This is a topic we've already seen multiple opinions related to. It's clear there is no explicit consensus on whether to use exceptions or not, so make a decision with intention and build your code with those paradigms in mind.

Comments

- Comments don't make up for bad code
- Comments should explain intent
- Comments should be removed whenever possible

Comments Should Explain Intent

The one thing that isn't present in code is the intent behind why you do something in particular. *That* is the purpose of comments: not to describe the what of code but the *why.*

```
type PrefixExpression struct {
    // the prefix token, either BANG or MINUS
    Token    token.Token
    // the ascii operator, either '!' or '-'
    Operator string
    Right    Expression
}
```

Comments Should Be Removed Whenever Possible

What does this comment do?

```
protected:
    // Private Class Level Variables
    std::string mName;
    int mHealth;
    float mSpeed;
    sf::Vector2f mPosition;
    sf::Sprite mSprite;
    sf::Texture mTexture;
```

```cpp
// Print Item Details
void Item::printDetails(std::ostream &out)
{
    out << mName << std::endl;
    out << "  Cost:   " << mCost << std::endl;
    out << "  Weight: " << mWeight << std::endl;
    out << "  Damage: " << mDamage << std::endl;
}
```

```
// declare vector
std::vector<Item*> items;

//declare variables
int windowX = 800;
int windowY = 600;
int characterSize = 40;

//create character
Characters character("Character", 100, 5.0f);
character.setSpeed(1.0f);

//generate items (name, cost, weight, damage)
items = generateItems(items, "Dagger of Suffering",
                          14, 4, 7.5f);
items = generateItems(items, "Broadsword",
                          30, 12, 15.5f);
```

Comments and Bad Code

Does this comment fix our previous bad code?

```
// getThem takes in the game state and returns the
// list of flagged cells
const getThem: ((inList: Array<int[]>) =>
    Array<int[]>) = () => {

    list1: Array<int[]> = new Array<int[]>();
    for (x in inList) {
        if (x[0] === 4) {
            list1.push(x);
        }
    }
    return list1;
}
```

Comments and Bad Code

Adding comments to bad code might help explain what it's doing, but it doesn't make the code good. When modifications are needed, now you have to:

1. Confirm the code actually does what the comments say
2. Modify the functionality
3. Update the comments to reflect the changed code

Instead of trying to explain bad code, take that effort and clarify the code itself.

Why?

Because what comments can we add that make this function easier to read?

```
const getFlaggedCells: ((gameBoard: Array<Cell>) =>
    Array<Cell>) = () => {

    return [...gameBoard.filter(
        (cell) => cell.isFlagged()
    )];
}
```

Formatting

Clean Code dedicates an entire chapter to how to manually format your code well. This is largely useless information now because you should instead just be using your languages auto-formatting functionality.

- Go - gofmt
- Rust - rustfmt
- JS - prettier

This is the reason you can go read any Go code and feel pretty comfortable. They've standardized the syntax and formatting so well that everything just looks like "go."

Unit Tests

"By now everyone knows that TDD asks us to write unit tests first, before we write production code. But that rule is just the tip of the iceberg."

Rules of TDD

First Law: You may not write production code until you have written a failing unit test.

Second Law: You may not write more of a unit test than is sufficient to fail, and not compiling is failing.

Third Law: You may not write more production code than is sufficient to pass the currently failing test.

Less Strict Rules (i.e. Andey's Peronal Rules)

1. You may provide outlines of behavior which refers to functions not written.
2. Write tests for a function, validating intended behavior.
3. Write the production code for the function, ensuring tests pass before moving on.

Questions?

Clean Coders

Coding

Practicing

Testing

Estimation

Collaboration

Optimization

Premature optimization is the root of all evil.

Questions?

Tools of the Trade

Version Control

IDEs

Component Testing Tools

Solid Workflows

CI/CD

Commit Hooks

PRs and Git

Refactoring Kata

Questions?

Next Time

- Spring Break!
- Specialized Topics