Writing Maintainable Software

How to be nice to your fellow engineers

Overview

- Maintainable software and its value
- Practices that help write maintainable software
- Short exercise to apply some practices
- Comments and file structure
- Tooling
- General guidelines

Symptoms of hard to maintain software

- Hard to read
- Hard to understand
- Hard to change
- Inconsistent style and implementation

Causes

- Experimentation
- Time pressure
- Going fast
- Unfamiliarity with the codebase/library/language
- Neglect/code rot

What is maintainable software?

- Future-reader-focused
- Easy to read
- Easy to understand
- Easy to change
- Consistent style and implementation

Considerations

- Guidelines, not rules
- Lots of exceptions
- Opinions and personal preference
- Language-dependent
- Context-dependent

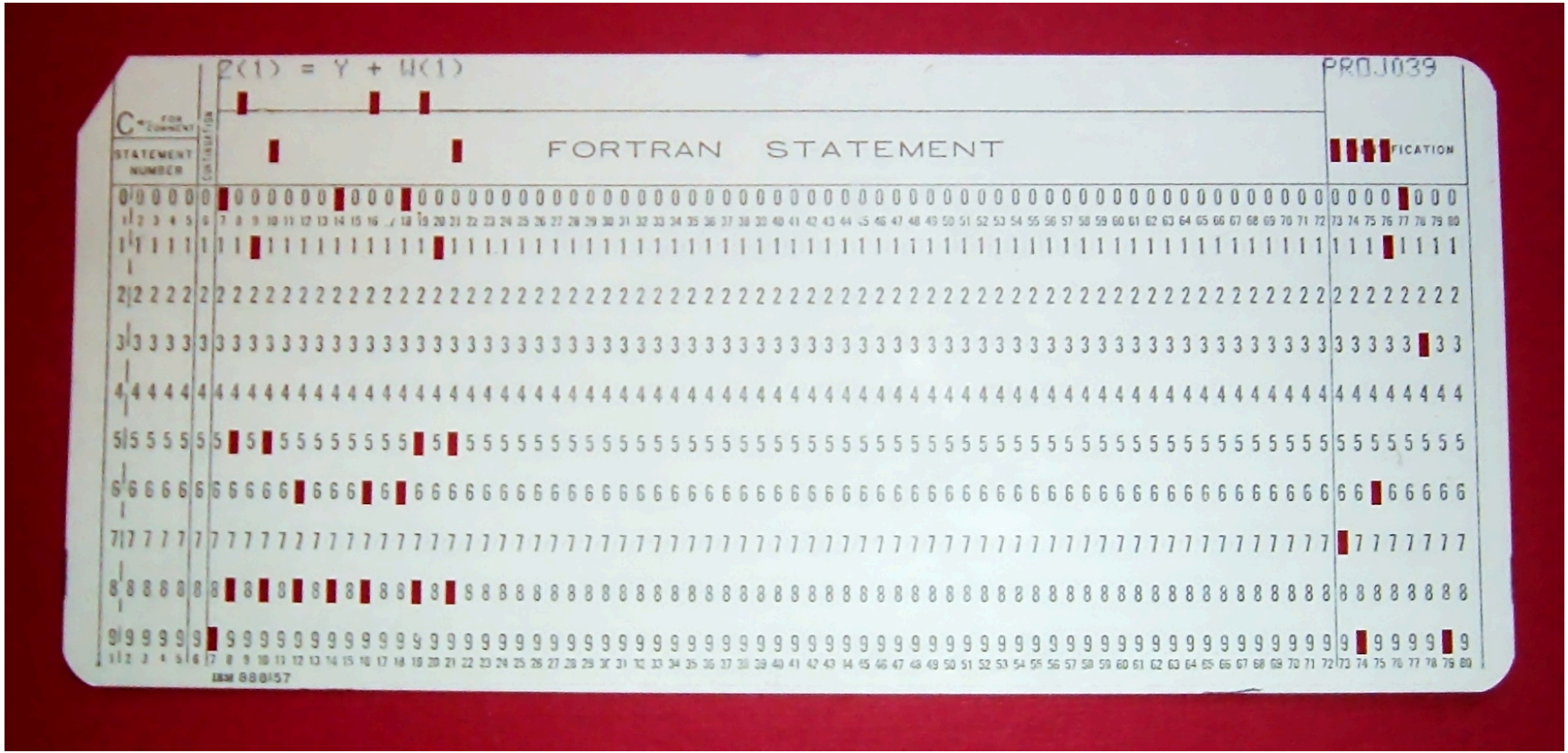
Naming

"There are only two hard things in computer science: cache invalidation and naming things."

– Phil Karlton

Naming

- Bad names can slow you down
- Good names (in combination with accurate types) can almost take away the need to actually read the code
- A bit of effort goes a long way
  - You can *always* do better than single-letter variables like `x` or `i`
- Meaningful
- Pronounceable
- Long-ish names are OK
  - Leverage your IDE's autocomplete for longer names, we don't use punchcards like in the 60s



Wikipedia - Punched card

Avoid Abbreviations

```ts
interface Addr {
  num: number;
  name: string;
};

const addrStrng = '123 Test St'; // User input

const prsAddr = (a: string): Addr => {};

const addr = prsAddr(addrStrng);

interface Address {
  streetNumber: number;
  streetName: string;
};

const addressInput = '123 Test St'; // User input

const parseAddressFromInput = (input: string): Address => {};

const parsedAddress = parseAddressFromInput(addressInput);
```

Use Names to Add Meaning

```
if (age >= 4 && age <= 18) {
  applyTaxBenefit();
}

const isOfSchoolAge = (age: number): boolean => {
  return age >= 4 && age <= 18;
};

if (isOfSchoolAge(age)) {
  applyTaxBenefit();
}
```

Functions

- Ideally small
- Minimal responsibility
  - Ideally does one thing, and does it well
- Named based on what it does

Benefits

- Easier to test
- Easier to understand how they work
- Easier to compose together
- Easier to change
- Easier to detect a bloated function
- A well-named function can often replace a comment

Nested Control Structures

```
if (
  datePicker.startDate === null ||
  datePicker.endDate === null
) {
  sendInvalidDateMessage();
} else {
  if (
    datePicker.startDate !== null &&
    datePicker.endDate !== null
  ) {
    if (datePickerStart.SelectedDate !== datePickerEnd.SelectedDate) {
      if (index1 === index2) {
        if (StartInt === EndInt) {
          if (radioButton.IsChecked === true) {
            printTime();
          } else {
            printTimeInADifferentWay();
          }
        }
      }
    }
  }
}
```

Nested Control Structures

```
const { startDate, endDate } = datePicker;

const isInvalidDate = startDate === null || endDate === null;

if (isInvalidDate) {
  sendInvalidDateMessage();
  return;
}

const startAndEndDatesAreEqual = startDate === endDate;
const someBusinessRule = index1 !== index2 || startInt !== endInt;

if (startAndEndDatesAreEqual || someBusinessRule) {
  return;
}

if (radioButton.isChecked) {
  printTimeInADifferentWay();
  return;
}

printTime();
```

Nested Control Structures

- Avoid overly-nested control structures (i.e. less indentation is better)
- Destructure values that are used more than once
- Split out and name conditional logic
- Use early `return`s to handle branches

Switch Statements

```
const getAnimalSound = (animal: string): string => {
  switch (animal) {
    case "Dog":
      return "Woof";
    case "Cat":
      return "Meow";
    case "Cow":
      return "Moo";

    default:
      return "Growl";
  }
}
```

Switch Statements

```
const soundFromAnimal: Record<string, string> = {
  Dog: "Woof",
  Cat: "Meow",
  Cow: "Moo",
};

const defaultAnimalSound = "Growl";

const getAnimalSound = (animal: string): string =>
  soundFromAnimal[animal] || defaultAnimalSound;

const allAnimals = Object.keys(soundFromAnimal);
const allSounds = [...Object.values(soundFromAnimal), defaultAnimalSound];
```

Switch Statements

- Easy to forget `break` or `default` case (TypeScript can help)
- Fall through logic can be risky, harder to understand
- Not bad maintainability for small number of cases
- Maintainability does not scale well as more cases are added

Data-driven Mapping

- Data is not coupled to mapping function
- Types are not coupled to mapping function
- Easier to add/remove/change a mapping

Long if-else-if Chains

```
const doThing = (input: string): string => {
  let output = input;

  if (input.startsWith("foo")) {
    output += "1";
  }

  else if (input.endsWith("foo")) {
    output += "2";
  }

  else if (input.startsWith("bar")) {
    output += "3";
  }

  else if (input.endsWith("bar")) {
    output += "4";
  }

  output = doMoreStuff(output);

  return output;
}
```

Long if-else-if Chains

```
function doThing(input: string): string {
  if (input.startsWith("foo")) {
    return `${input}1`;
  }

  if (input.endsWith("foo")) {
    return `${input}2`;
  }

  if (input.startsWith("bar")) {
    return `${input}3`;
  }

  if (input.endsWith("bar")) {
    return `${input}4`;
  }

  return input;
}

let output = doThing("foobar");
output = doMoreStuff(output);
```

Too Many Function Parameters

```
foo();

bar(ok, nice);

baz(maybe, its, time, to, refactor);
```

Too Many Function Parameters

- Function is doing more than one thing
- Harder to interpret at a glance
- Readers more likely to ignore long lists
- Types can help, but don’t fix the problem

Options Object Parameter

```
transform("the quick brown fox", "en", false, 3, " ");

transform("the quick brown fox", {
  locale: "en",
  delimiter: " ",
  maxLines: 3,
  truncate: false,
});
```

Exercise + Break

Comments

Comments

- Ideally none (unrealistic)
- No need to comment every line
- Can provide valuable information, but can quickly become stale
- Writing is hard!

Comments - Explain With Code

```
// Check if eligible for long service leave
if (employee.type === 'Permanent' && employee.tenure >= 7) {
  // ...
}

if (isEligibleForLongServiceLeave(employee)) {
  //...
}
```

Comments - Non-obvious Information

```
// matches hh:mm:ss
const timeRegexp = new RegExp('\\d\\d:\\d\\d:\\d\\d');
```

Comments - TODOs and Context

```
// TODO: Handle edge case where {some condition}

// TODO: Refactor to use a Set instead of an Array
// so we can deduplicate the results

// See {link to github issue} for why we need this workaround

// You would think X would work here, but it doesn't because
// {a reason that took 4 engineers and 40 coffees to figure out}
```

Reasons to Leave a Comment

- Unexpected/non-standard implementation/decision
- Reasoning for one approach over another
- Something is hard to understand at a glance and can’t be simplified (regex)
- Link to a formal specification that you’re adhering to
- Link to github issue for a bug/workaround

Commented Code

```
// const add = (a: number, b: number) => {
//   return a + b;
// }

const add = (a: number, b: number) => {
  return a + b - b + b - a + a;
}
```

Commented Code

- Fine temporarily, just don't commit it
- Bloats files
- If it's committed, just delete it, it can be recovered (that's the point of version control)
- If it's *not* committed and you're worried about losing it, commit it and then delete it

Structure

File and Line Length

- Line length limits are a thing of the past
- Code formatting tools help ensure lines don't get too long
- The contents of a file should be related to the name of the file
- Generic names like `utils.ts`, `helpers.ts`, become dumping grounds for shared functions
- Group shared code by their function, e.g. `formatters.ts`, `validators.ts`
- Large files can signal an opportunity to split things up

Proximity Implies Association

```
const thing = getThingFromSomewhere();

// ... 20 lines later ...

doSomethingWith(thing);

const thing = getThingFromSomewhere();
doSomethingWith(thing);
```

Whitespace

- Very much a personal preference
- Code formatting tools get you 90% of the way there
- IMO newlines are very underrated in their effect on readability

Separate Groups of Related Code

```
const myFunction = (input: string) => {
  const uppercaseInput = input.toUpperCase();
  const words = uppercaseInput.split(" ");
  const filteredWords = words.filter(myWordFilter);
  if (filteredWords.length === 0) {
    console.log("No words remaining :(")
    return;
  }
  console.log(`${filteredWords.length} words remaining :)`);
};

const myFunction = (input: string) => {
  const uppercaseInput = input.toUpperCase();
  const words = uppercaseInput.split(" ");
  const filteredWords = words.filter(myWordFilter);

  if (filteredWords.length === 0) {
    console.log("No words remaining :(")
    return;
  }

  console.log(`${filteredWords.length} words remaining :)`);
};
```

Separate Top-Level Stuff

```
import { foo } from 'foo';
import { bar } from 'bar';
const myFunction = () => {
  return 1;
};
const myOtherFunction = () => {
  return 2;
};

import { foo } from 'foo';
import { bar } from 'bar';

const myFunction = () => {
  return 1;
};

const myOtherFunction = () => {
  return 2;
};
```

Separate If Statements

```
const myFunction = (input: number) => {
  if (input > 0) {
    doThing();
  }
  if (input < 0) {
    doOtherThing();
  }
  if (SOME_GLOBAL_VARIABLE === true) {
    doSomeOtherThing();
  }
};

const myFunction = (input: number) => {
  if (input > 0) {
    doThing();
  }

  if (input < 0) {
    doOtherThing();
  }

  if (SOME_GLOBAL_VARIABLE === true) {
    doSomeOtherThing();
  }
};
```

Separate Final Return Statements

```
const myFunction = (input: number) => {
  if (input > 0) {
    doThing();
  }

  if (SOME_GLOBAL_VARIABLE === true) {
    doSomeOtherThing();
  }
  const output = input + 1;
  return output;
};

const myFunction = (input: number) => {
  if (input > 0) {
    doThing();
  }

  if (SOME_GLOBAL_VARIABLE === true) {
    doSomeOtherThing();
  }

  const output = input + 1;

  return output;
};
```

Separate Test Cases

```
describe("All my cool functions", () => {
  describe("myFunction", () => {
    it("should return 1 when given an input of 0", () => {
      // test stuff
    });
    it("should return 2 when given an input of 1", () => {
      // test stuff
    });
  });
  describe("myOtherFunction", () => {
    it("should return -1 when given an input of 0", () => {
      // test stuff
    });
    it("should return 0 when given an input of 1", () => {
      // test stuff
    });
  });
});
```

Separate Test Cases

```
describe("All my cool functions", () => {
  describe("myFunction", () => {
    it("should return 1 when given an input of 0", () => {
      // test stuff
    });

    it("should return 2 when given an input of 1", () => {
      // test stuff
    });
  });

  describe("myOtherFunction", () => {
    it("should return -1 when given an input of 0", () => {
      // test stuff
    });

    it("should return 0 when given an input of 1", () => {
      // test stuff
    });
  });
});
```

Tooling

Code Formatters

- Formats your code according to specific rules
- Enforce consistent style
- Consistency trumps personal preference
- In the JavaScript/TypeScript ecosystem Prettier is the commonly used code formatter

Code Formatters

```
// before
const myFunction=(input:string)=> {
  let output=intput +"123";
  output+='777';
  if(output.length>20) {console.log('big string')}
  return output;
}

// after
const myFunction = (input: string) => {
  let output = intput + '123';
  output += '777';
  if (output.length > 20) {
    console.log('big string');
  }
  return output;
}
```

Linters

- Warns/errors on specific code patterns
- Enforce certain syntax patterns
- ESLint common in the JavaScript/TypeScript ecosystem
- Identify code smells
- Prevent footguns
- Can automatically fix some issues

Linters

```
// before
const myFunction = (input: string) => {
  return `${input}123`;
}

// after
const myFunction = (input: string) => `${input}123`;
```

Git and GitHub

- Commit early, commit often
- Past PRs contain tons of context and learnings
- Keep PRs focused and small if possible
  - Split up large changes into smaller PRs
  - No rule that says you can only do 1 PR per Jira card/item of work
- Review your own PR (even before you open it!)
  - Helps catch silly mistakes/oversights
  - Helps gauge whether or not your changes should be split up into multiple PRs
- Add comments to your own PRs (try and pre-empt your reviewer's questions)
- GitHub search is super useful for finding examples/inspiration (both in SEEK repos and external repos)

Acronyms

Acronyms

- KISS (**K**eep **I**t **S**imple, **S**illy)
- DRY (**D**on’t **R**epeat **Y**ourself)
- WET (**W**rite **E**verything **T**wice)
- AHA (**A**void **H**asty **A**bstractions)
- "Prefer duplication over the wrong abstraction"
- "Optimize for change"

Final Thoughts

"Always leave the ~~campground~~ code cleaner than you found it."

– A scout that grew up to be a software developer, probably

Writing Maintainable Code is a Communication Skill

->