Refactoring with stratified design

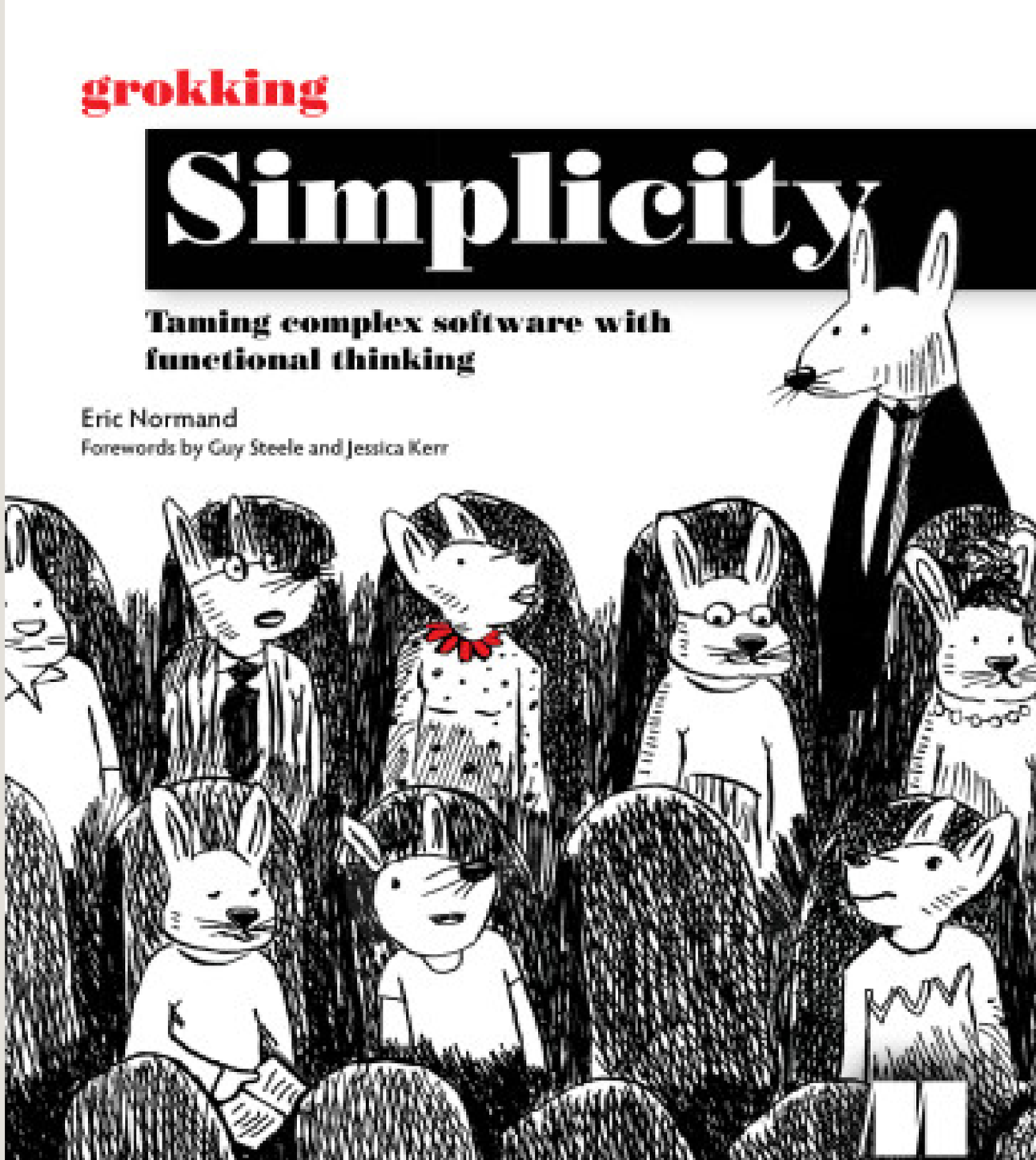
Outline

- Some background
- Stratified design as a concept
- Coding
- Recap

What’s the point?

- Readability, maintainability, testability etc
- Power of conceptualization
- Fun?

- Normand, Eric 2021: Grokking simplicity. Manning.
- *Stratified design*
- Some basic refactoring patterns



A totally contrived example

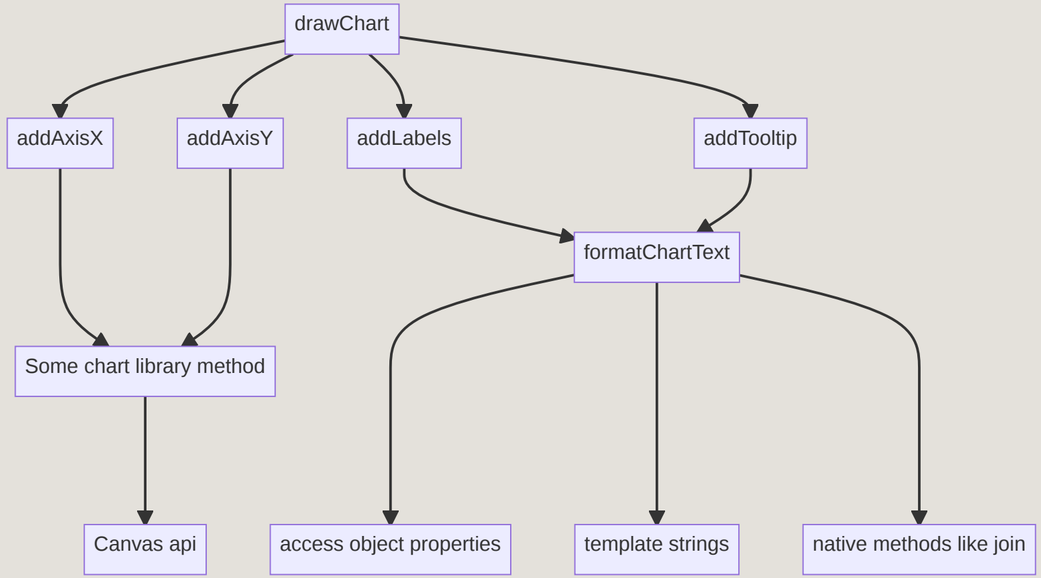
```
const users: Player = [
  { name: "Paul", id: 1, strokes: undefined, rank: undefined },
  { name: "Ricky", id: 2, strokes: undefined, rank: undefined },
];
const holes: Hole = [
  { no: 1, par: 3 },
  { no: 2, par: 3 },
  { no: 3, par: 4 },
];
const scoreCards: ScoreCard = [
  { userId: 1, strokes: [2, 2, 4] },
  { userId: 2, strokes: [2, 2, 3] },
];
```

```
import sum from "mathlib";

const getTotal = (results: number[]) => {
  return sum(results);
};

const getPlayersCard = (player: Player, scoreCards: ScoreCard
  return scoreCards.find((scoreCard) => scoreCard.playerId ==
};

const convertResultsToCsvRows = (players: Player[]): string[]
  const rows: string[] = [];
  for (let player of players) {
    const card = getPlayersCard(player, scoreCards);
    let row: ScoreRow[] = [];
    const results: number[] = [];
    for (let i = 0; i < holes.length; i++) {
      const hole = holes[i];
      const strokes = card.strokes[i];
      const par = holes[i].par;
      results.push(strokes - par);
    }
    const total = getTotal(results);
    rows.push([...results, total].join(","));
  }
  return rows;
};
```

- native language features
- generic function
- specific functions of domain X
- specific functions of domain Y

Some principles

- "Arrow length": reaching out to features on a different layer
- Level of details vs. current level of thinking
- Abstraction barriers: set of functions forming a line not to be crossed
- Maintainability, testability, reusability
- Cf. traditional concept of domains



Conceptualizing a familiar process

- How to make a language feature first-class?

```
try {
  doSomething();
} catch {
  logErrors();
}
```

```
try {
  doSomethingElse();
} catch {
  logErrors();
}
```

```
function doSomethingAndLogErrors() {
  try {
    doSomething();
  } catch {
    logErrors();
  }
}
```

```
function doAndLogErrors(f) {
  try {
    f();
  } catch {
    logErrors();
  }
}
```

- As a pattern: replace body with callback

1. identify *before*
2. identify *after*
3. identify *body*
4. Extract the whole thing as a function
5. Extract the body of the function as a callback passed as an argument

Workshop

- a) Take a look at files at `./examples/`
  - look at the code and try to figure out a logical structure
  - stratify the code into logical layers
  - should work well as a pair-programming excercise
- b) Think about a piece of code you wrote today / this week
  - Think about the strata in that code
  - sketch out a call graph with arrows of different length
  - would it make sense to move something around?
  - would that bring any benefits in terms of maintainability / readability / testability?