Manychat

2025

Refactor, Rewrite, Reengineer:

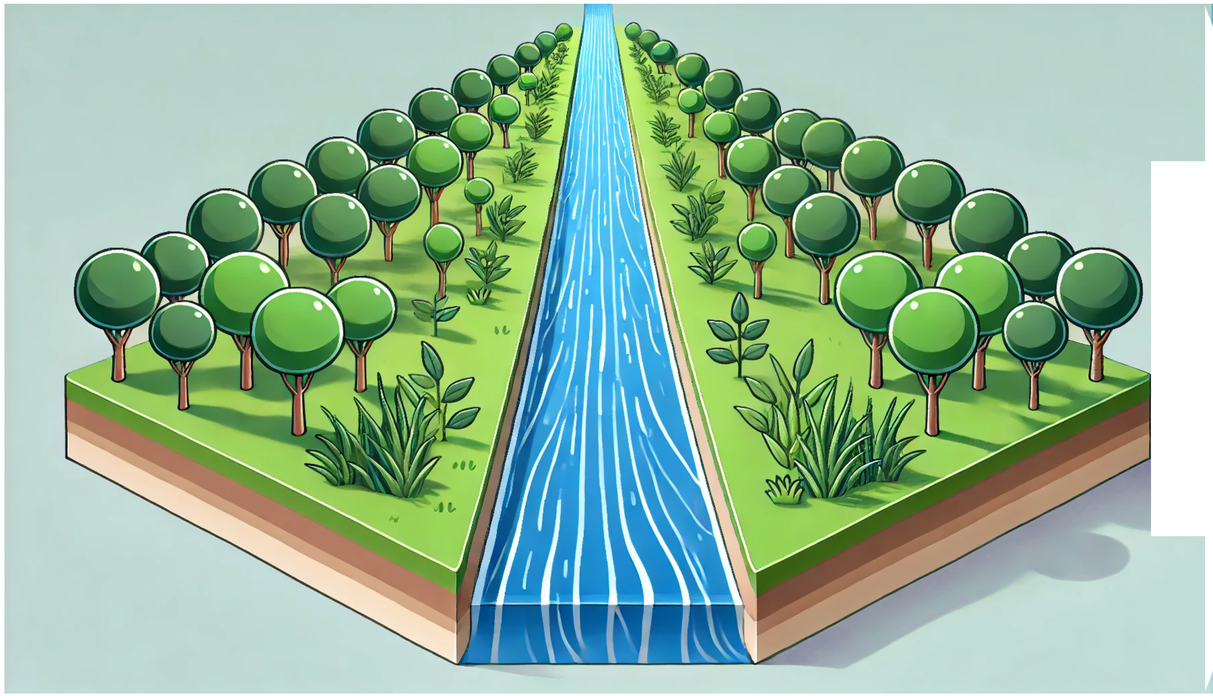
Evolving a codebase

Egor Naidikov

- 15 years of experience in software engineering
  - Frontend
  - Full-stack
  - Mobile
- Frontend engineer at Manychat

Why we need to change the code?

Evolving business requirements



INITIAL BUSINESS REQUIREMENTS

BUSINESS REQUIREMENTS OVER TIME

Evolving standards

Simple React component

2014 approach

```jsx
import React, { Component } from 'react'
import PropTypes from 'prop-types'

class InputWithButton extends Component {
  static propTypes = {
    buttonText: PropTypes.string,
    onSubmit: PropTypes.func.isRequired,
  }

  static defaultProps = {
    buttonText: 'Submit',
  }

  constructor(props) {
    super(props)
    this.state = {
      inputValue: '',
    }

    this.handleInputChange = this.handleInputChange.bind(this)
    this.handleButtonClick = this.handleButtonClick.bind(this)
  }

  handleInputChange(event) {
    this.setState({
      inputValue: event.target.value,
    })
  }

  handleButtonClick() {
    this.props.onSubmit(this.state.inputValue)
  }

  render() {
    const { buttonText } = this.props
    const { inputValue } = this.state
    const { handleInputChange, handleButtonClick } = this

    return (
      <div>
        <input type="text" value={inputValue} onChange={handleInputChange} />
        <button onClick={handleButtonClick}>{buttonText}</button>
      </div>
    )
  }
}

export default InputWithButton
```

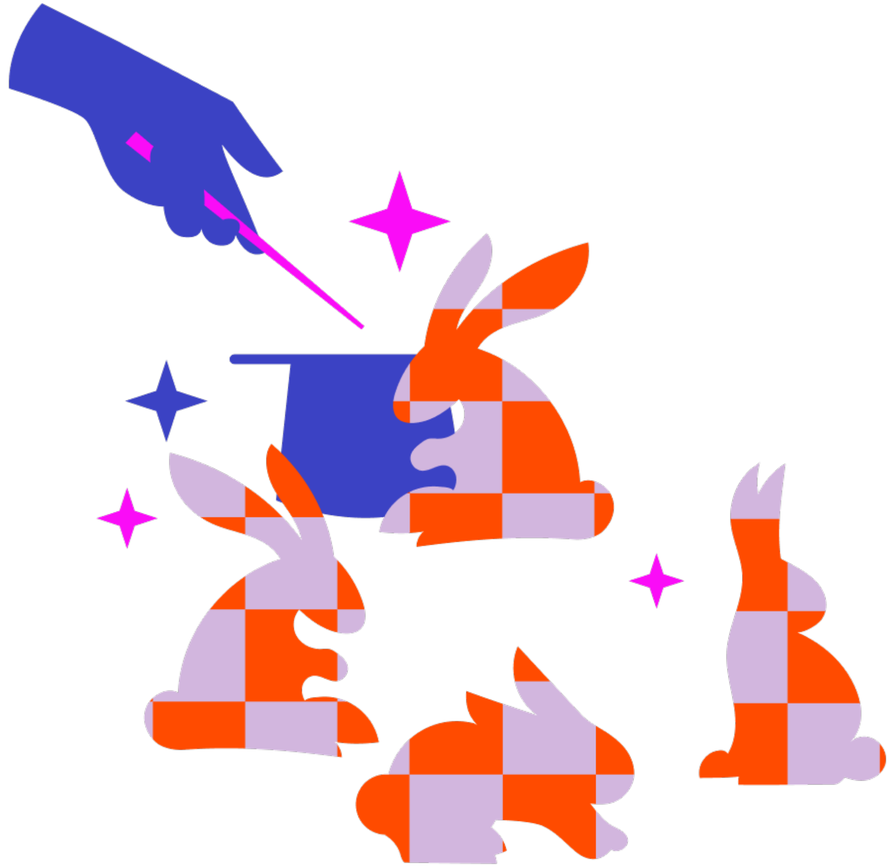
Simple React component

2024 approach

```tsx
import React, { useState } from 'react'

interface InputWithButtonProps {
  buttonText?: string
  onSubmit: (inputValue: string) => void
}

export const InputWithButton = ({ buttonText = 'Submit', onSubmit }: InputWithButtonProps) => {
  const [inputValue, setInputValue] = useState<string>('')

  const handleInputChange = (event: React.ChangeEvent<HTMLInputElement>) => {
    setInputValue(event.target.value)
  }

  const handleButtonClick = () => {
    onSubmit(inputValue)
  }

  return (
    <div>
      <input type="text" value={inputValue} onChange={handleInputChange} />
      <button onClick={handleButtonClick}>{buttonText}</button>
    </div>
  )
}
```

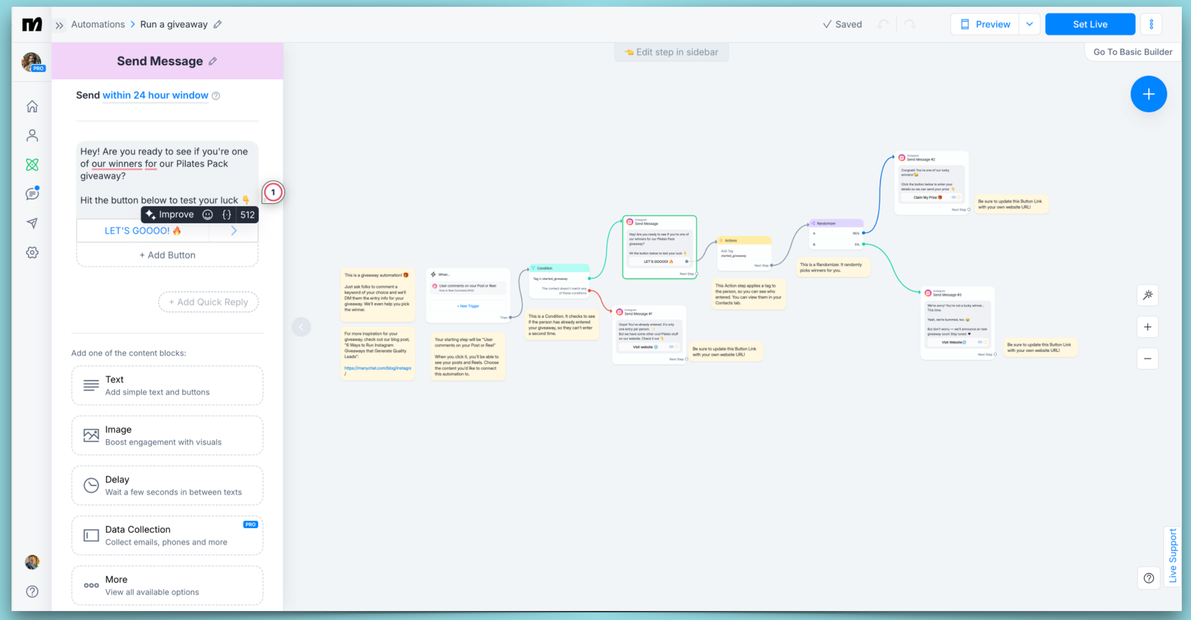
```jsx
import React, { Component } from 'react'
import PropTypes from 'prop-types'

class InputWithButton extends Component {
  static propTypes = {
    buttonText: PropTypes.string,
    onSubmit: PropTypes.func.isRequired,
  }

  static defaultProps = {
    buttonText: 'Submit',
  }

  constructor(props) {
    super(props)
    this.state = {
      inputValue: '',
    }

    this.handleInputChange = this.handleInputChange.bind(this)
    this.handleButtonClick = this.handleButtonClick.bind(this)
  }

  handleInputChange(event) {
    this.setState({
      inputValue: event.target.value,
    })
  }

  handleButtonClick() {
    this.props.onSubmit(this.state.inputValue)
  }

  render() {
    const { buttonText } = this.props
    const { inputValue } = this.state
    const { handleInputChange, handleButtonClick } = this

    return (
      <div>
        <input type="text" value={inputValue} onChange={handleInputChange} />
        <button onClick={handleButtonClick}>{buttonText}</button>
      </div>
    )
  }
}

export default InputWithButton
```

```tsx
import React, { useState } from 'react'

interface InputWithButtonProps {
  buttonText?: string
  onSubmit: (inputValue: string) => void
}

export const InputWithButton = ({ buttonText = 'Submit', onSubmit }: InputWithButtonProps) => {
  const [inputValue, setInputValue] = useState<string>('')

  const handleInputChange = (event: React.ChangeEvent<HTMLInputElement>) => {
    setInputValue(event.target.value)
  }

  const handleButtonClick = () => {
    onSubmit(inputValue)
  }

  return (
    <div>
      <input type="text" value={inputValue} onChange={handleInputChange} />
      <button onClick={handleButtonClick}>{buttonText}</button>
    </div>
  )
}
```

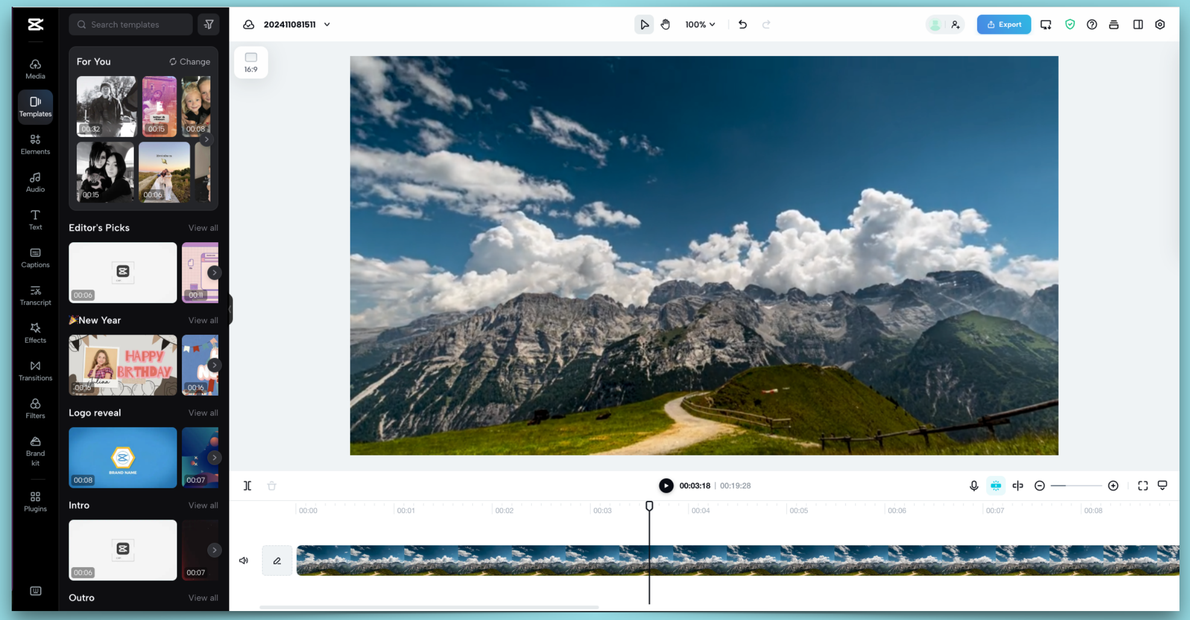


Increasing complexity

Web apps became as complex as desktop apps



Manychat



Cap Cut Web

Our codebase needs to be constantly updated to adapt for new business requirements and industry standards, improve its maintainability and scalability.

01 Refactoring

02 Reengineering
03 Wrapping up


SPACE RANGER
LIGHTYEAR
TO INFINITY AND BEYOND
©Disney/Pixar

Refactoring is the process of changing a software system in a way that does not alter the external behavior of the code yet improves its internal structure.

M. Fowler

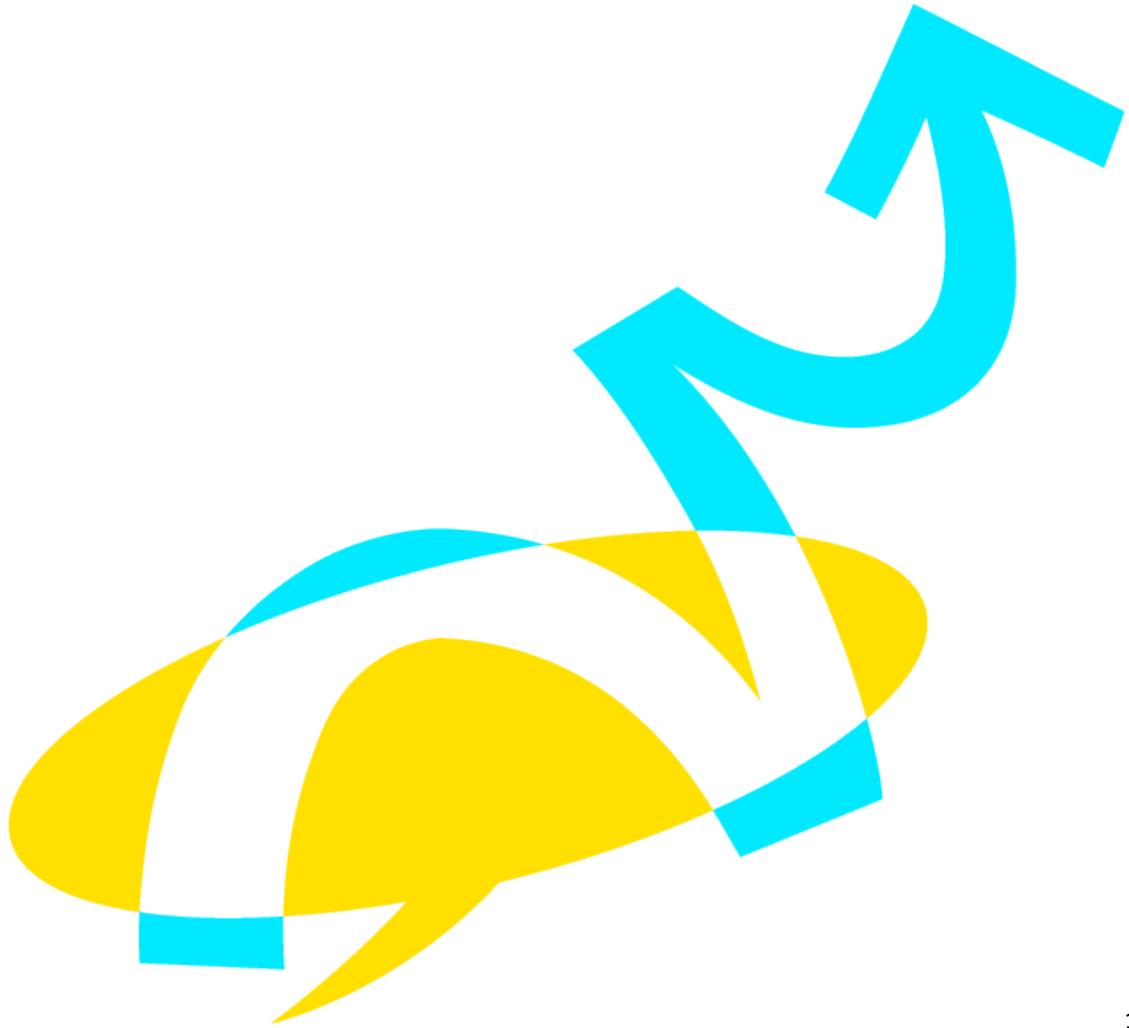
Refactoring Improving the Design of Existing Code

Refactoring is always done to make the code “easier to understand and cheaper to modify.”

M. Fowler

Refactoring Improving the
Design of Existing Code

How refactoring can go wrong



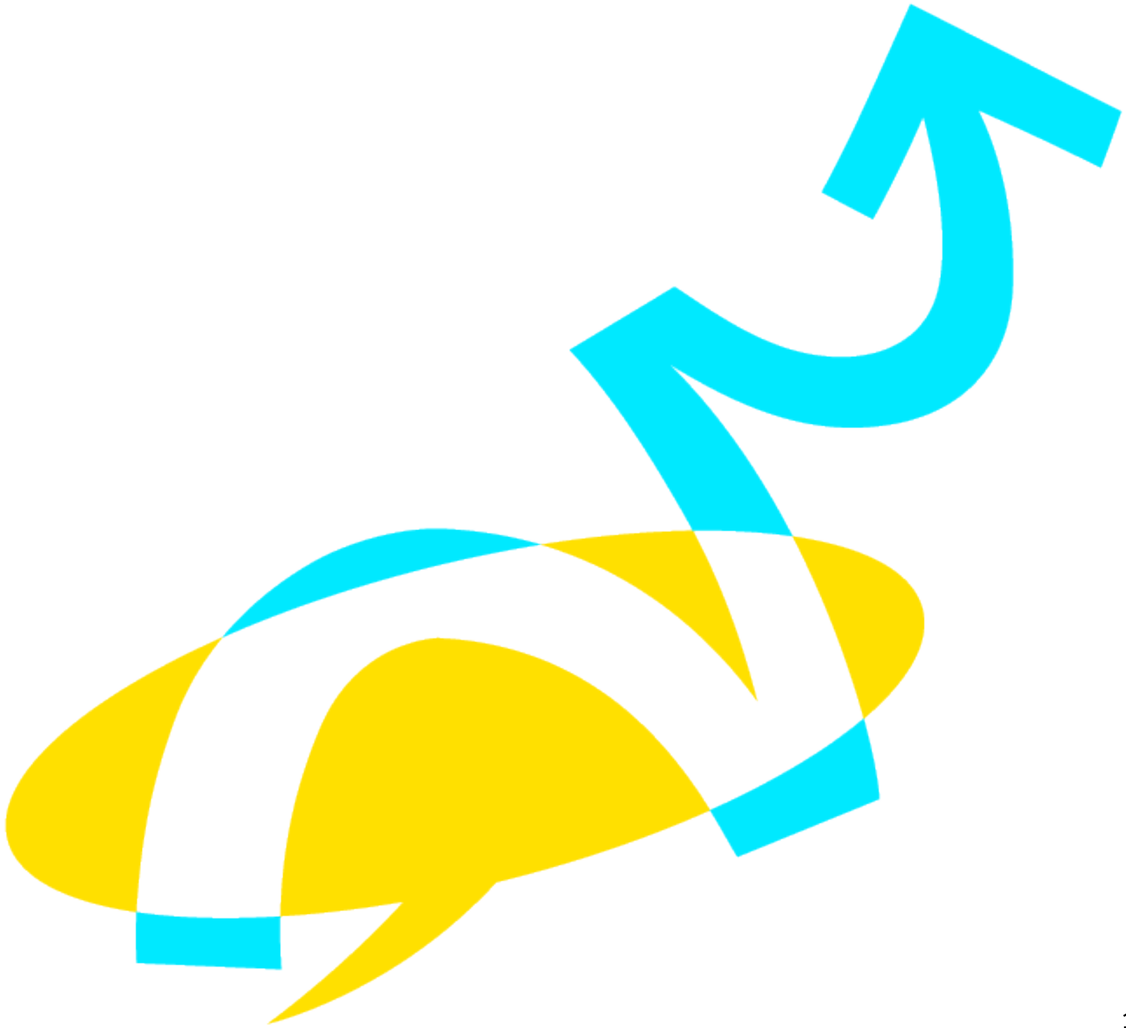
How refactoring can go wrong

Task:

Refactor WidelyUsedComponent.js

Motivation:

Prepare code to implement upcoming feature easier



Refactor WidelyUsedComponent.js

What we have:

- Legacy JS class component
- >100 usages in codebase

Initial scope:

- Cover component with tests
- Rewrite to Functional component using TypeScript
- Get rid of PropTypes and DefaultProps

```
export class WidelyUsedComponent extends React.Component {
  static propTypes = {
    label: PropTypes.string,
    value: PropTypes.string,
    onChange: PropTypes.func,
    helpText: PropTypes.string,
    submitText: PropTypes.string,
    showHelpText: PropTypes.bool,
    isReadonly: PropTypes.bool,
    isFullWidth: PropTypes.bool,
    isWithSpacer: PropTypes.bool,
    isWithoutLabel: PropTypes.bool,
    isAccent: PropTypes.bool,
    isElevated: PropTypes.bool,
    isMuted: PropTypes.bool,
    isDisabled: PropTypes.bool,
  }

  static defaultProps = {
    label: 'Label:',
    value: '',
    onChange: () => {},
    helpText: 'This is some help text.',
    submitText: 'Submit',
    showHelpText: true,
    isFullWidth: false,
    isWithSpacer: false,
    isWithoutLabel: false,
    isReadonly: false,
    isAccent: false,
    isElevated: false,
    isMuted: false,
    isDisabled: false,
  }

  constructor(props) {
    super(props)
    this.state = {
      value: this.props.value,
    }
  }

  handleChange = (event) => {
    this.setState({ value: event.target.value })
  }

  handleSubmit = () => {
    this.props.onChange(this.state.value)
  }

  handleKeyPress = (event) => {
    if (event.key === 'Enter') {
      this.handleSubmit()
    }
  }

  handleBlur = () => {
    this.setState({ value: this.props.value })
```

Refactor WidelyUsedComponent.js

```
export class WidelyUsedComponent extends React.Component {
  static propTypes = {
    label: PropTypes.string,
    value: PropTypes.string,
    onChange: PropTypes.func,
    helpText: PropTypes.string,
    submitText: PropTypes.string,
    showHelpText: PropTypes.bool,
    isReadonly: PropTypes.bool,
    isFullWidth: PropTypes.bool,
    isWithSpacer: PropTypes.bool,
    isWithoutLabel: PropTypes.bool,
    isAccent: PropTypes.bool,
    isElevated: PropTypes.bool,
    isMuted: PropTypes.bool,
    isDisabled: PropTypes.bool,
  }

  static defaultProps = {
    label: 'Label:',
    value: '',
    onChange: () => {},
    helpText: 'This is some help text.',
    submitText: 'Submit',
    showHelpText: true,
    isFullWidth: false,
    isWithSpacer: false,
    isWithoutLabel: false,
    isReadonly: false,
    isAccent: false,
    isElevated: false,
    isMuted: false,
    isDisabled: false,
  }

  constructor(props) {
    super(props)
    this.state = {
      value: this.props.value,
    }
  }

  handleChange = (event) => {
    this.setState({ value: event.target.value })
  }

  handleSubmit = () => {
    this.props.onChange(this.state.value)
  }

  handleKeyPress = (event) => {
    if (event.key === 'Enter') {
      this.handleSubmit()
    }
  }

  handleBlur = () => {
    this.setState({ value: this.props.value })
```

Refactor WidelyUsedComponent.js

Found problems:

- Lack of tests
- Minor bug in component logic
- Interface optimization
  - Unused props
  - Props could be merged

```
export class WidelyUsedComponent extends React.Component {
 static propTypes = {
    // General props
    label: PropTypes.string,
    value: PropTypes.string,
    onChange: PropTypes.func,
    helpText: PropTypes.string,
    submitText: PropTypes.string,
    isReadonly: PropTypes.bool,
    isDisabled: PropTypes.bool,

    // Style props
    isFullWidth: PropTypes.bool, // not used inside the component
    isWithSpacer: PropTypes.bool, // not used inside the component
    isWithoutLabel: PropTypes.bool, // not used inside the component
    showHelpText: PropTypes.bool, // duplicates logic of helpText

    // Theme props, could be combined into one prop
    isAccent: PropTypes.bool,
    isElevated: PropTypes.bool,
    isMuted: PropTypes.bool,
 }


  static defaultProps = {
    label: 'Label:',
    value: '',
    onChange: () => {},
    helpText: 'This is some help text.',
    submitText: 'Submit',
    showHelpText: true,
    isFullWidth: false,
    isWithSpacer: false,
    isWithoutLabel: false,
```

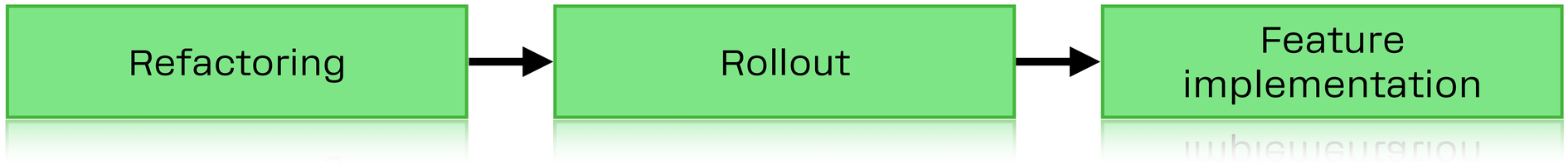
Refactor WidelyUsedComponent.js

Actual scope:

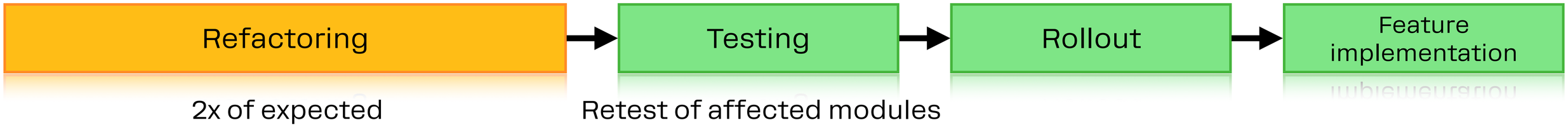
- Covered component with tests
- Rewritten to Functional component using TypeScript
- Fixed bug in logic
- Updated component interface
- Get rid of PropTypes and DefaultProps
- Updated all usages (>100)

```tsx
enum Theme {
  DEFAULT = 'default',
  ACCENT = 'accent',
  ELECATED = 'elevated',
  MUTED = 'muted',
}

interface WidelyUsedComponentProps {
  label?: string
  value?: string
  onChange?: (value: string) => void
  helpText?: string
  submitText?: string
  isReadonly?: boolean
  isDisabled?: boolean
  theme?: Theme
}

export const WidelyUsedComponent = ({
  label = 'Label:',
  value = '',
  onChange = () => {},
  helpText = 'This is some help text.',
  submitText = 'Submit',
  isReadonly = false,
  isDisabled = false,
  theme,
}: WidelyUsedComponentProps) => {
  const [inputValue, setInputValue] = useState(value)

  const handleChange = (event: React.ChangeEvent<HTMLInputElement>) => {
    setInputValue(event.target.value)
  }

  const handleSubmit = () => {
    onChange(inputValue)
  }

  const handleKeyPress = (event: React.KeyboardEvent<HTMLInputElement>) => {
    if (event.key === 'Enter') {
      handleSubmit()
    }
  }

  const handleBlur = () => {
    setInputValue(value)
  }

  const wrapperClassName = theme ? theme : Theme.DEFAULT
```

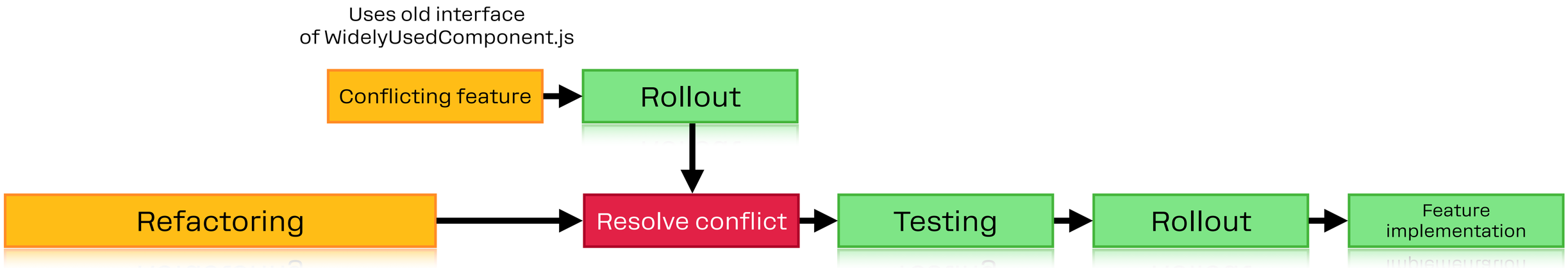
Expexted lifecycle



Actual lifecycle

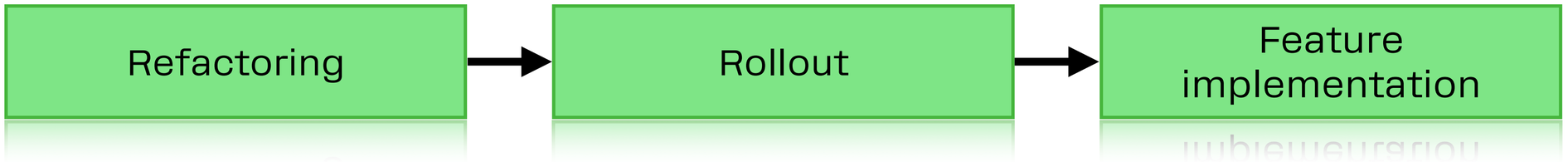


Worst case scenario lifecycle

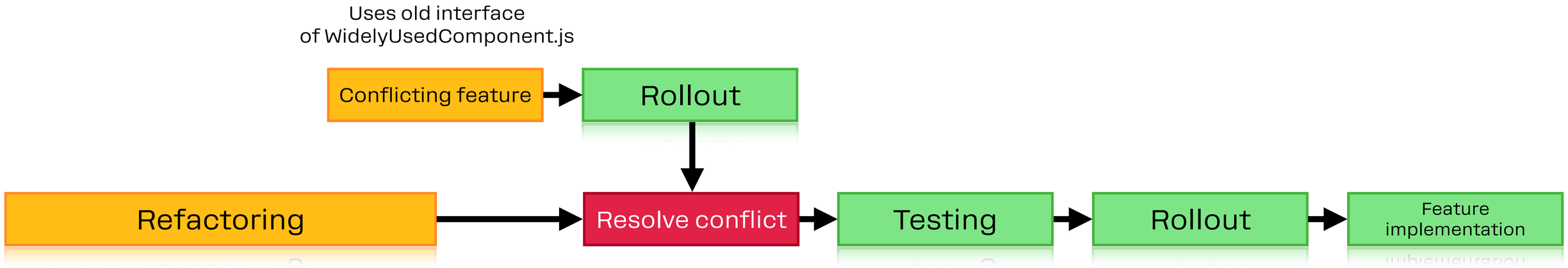


Lifecycle in comparison

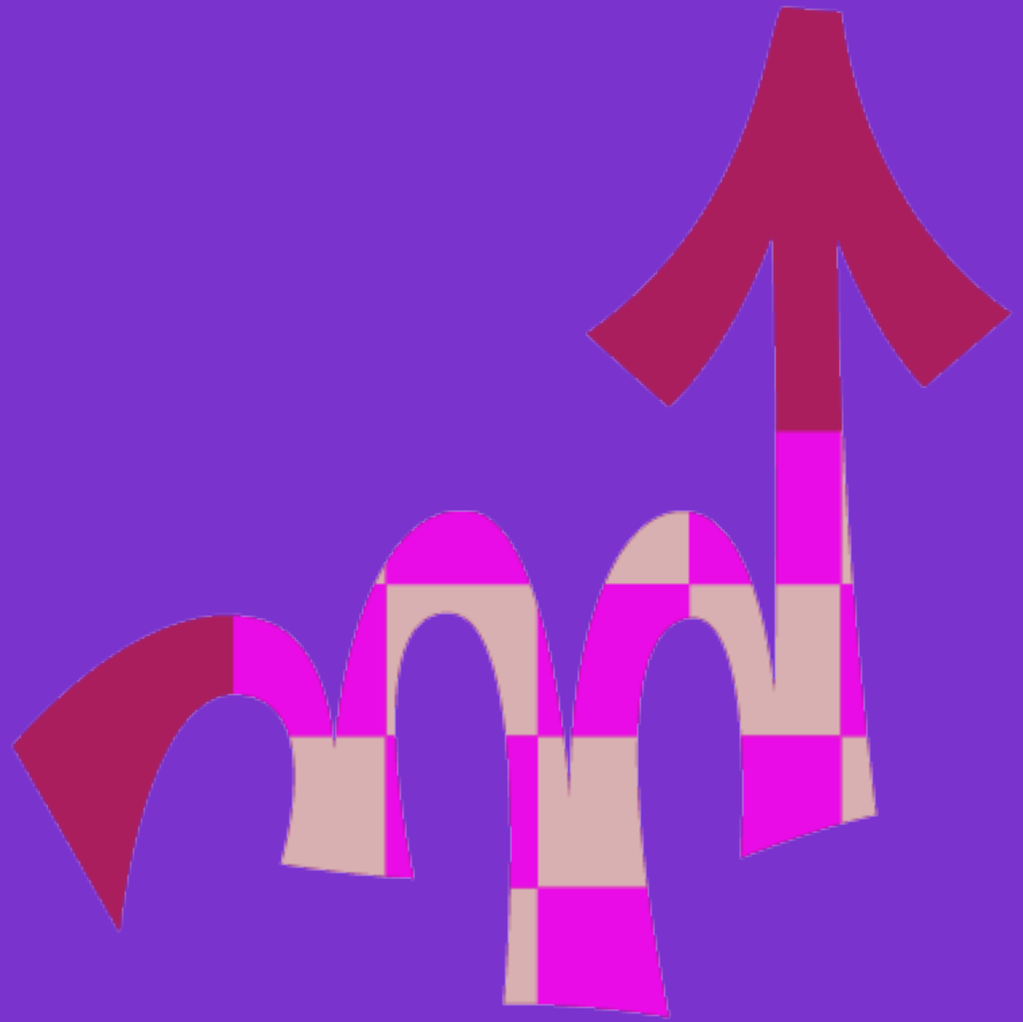
Expected:



Actual, worst case scenario:



Insights



Insight 1: Misunderstanding "Refactoring" can lead to scope creep

Insight 2: Separate concerns for better project management

Insight 3: Communication is key in refactoring projects

Insight 4: Keep refactoring small and focused

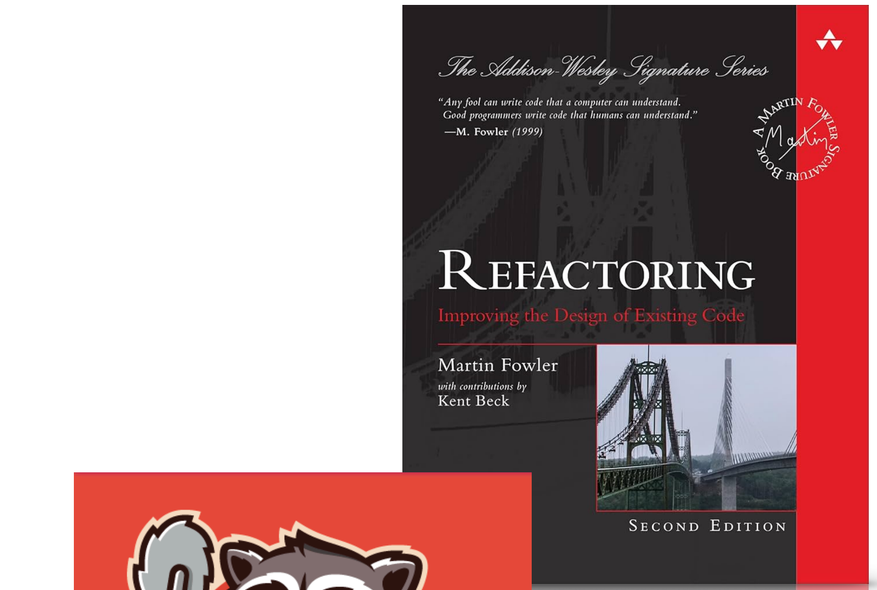
If someone says their code was broken for a couple of days while they are refactoring, you can be pretty sure they were not refactoring.

M. Fowler

Refactoring Improving the Design of Existing Code

Scope of refactoring:

- Renaming variables or functions for clarity
- Eliminating code smells
- Replacing magic numbers with named constants
- Simplifying complex conditional logic
- Removing unused or redundant code
- Consolidating duplicate code segments
- Breaking down large modules into smaller, manageable units
- Dead code removal

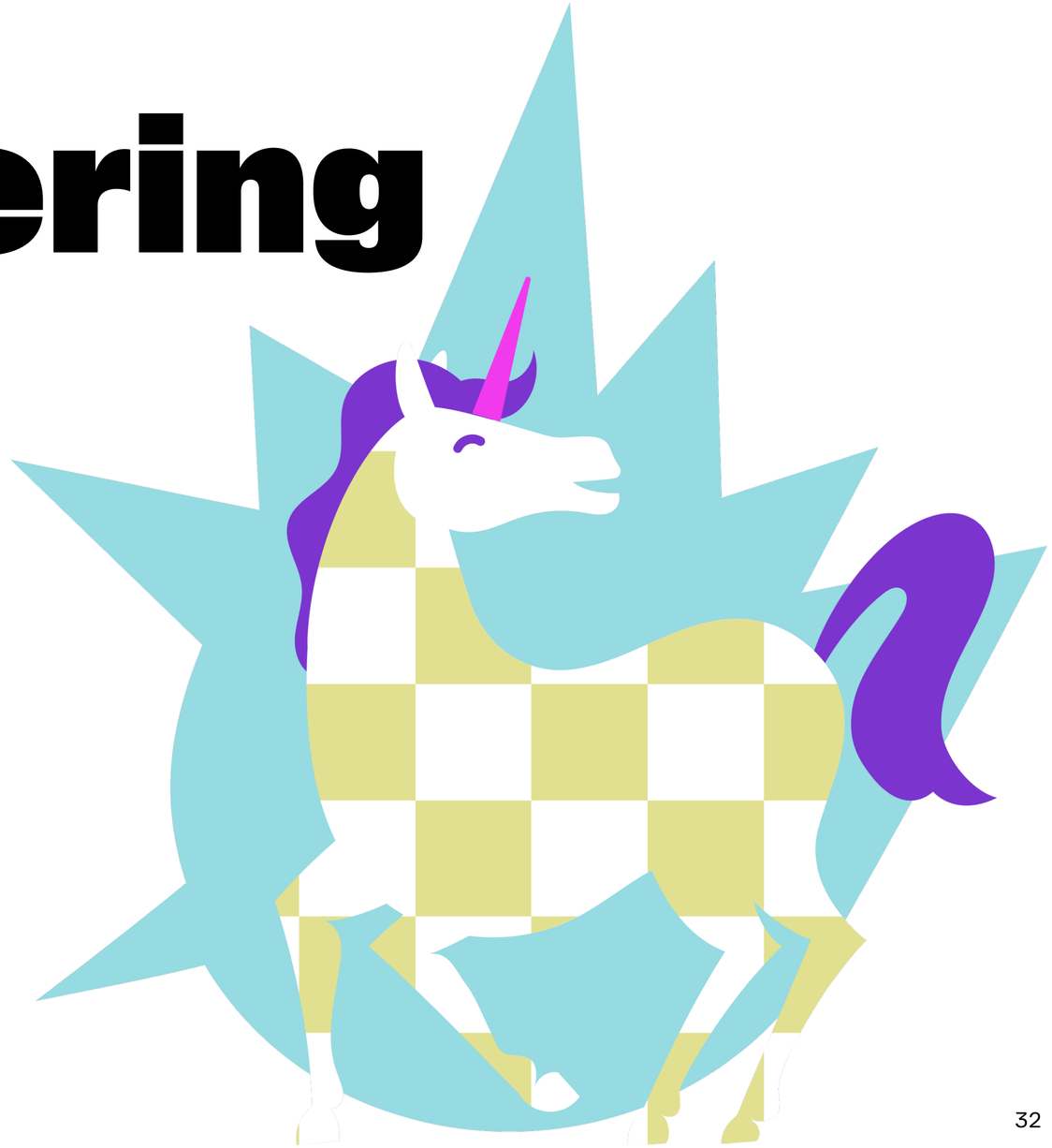


https://refactoring.guru

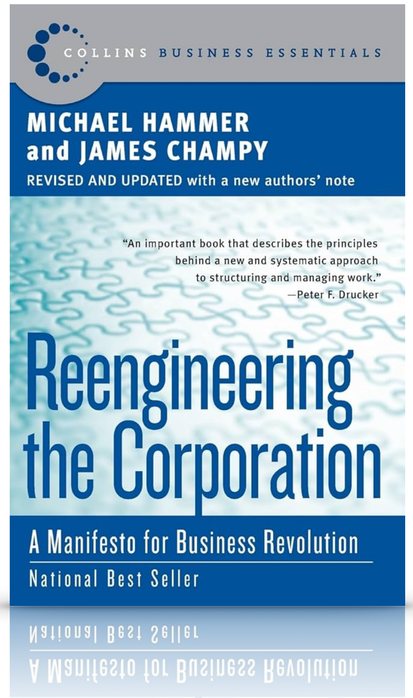
01 Refactoring

02 Reengineering

03 Wrapping up



Reengineering the Corporation: A Manifesto for Business Revolution



1993

Reengineering is the examination and alteration of a system to reconstitute it in a new form.

E.J. Chikofsky, J.H. Cross

Reverse engineering and design recovery: a taxonomy

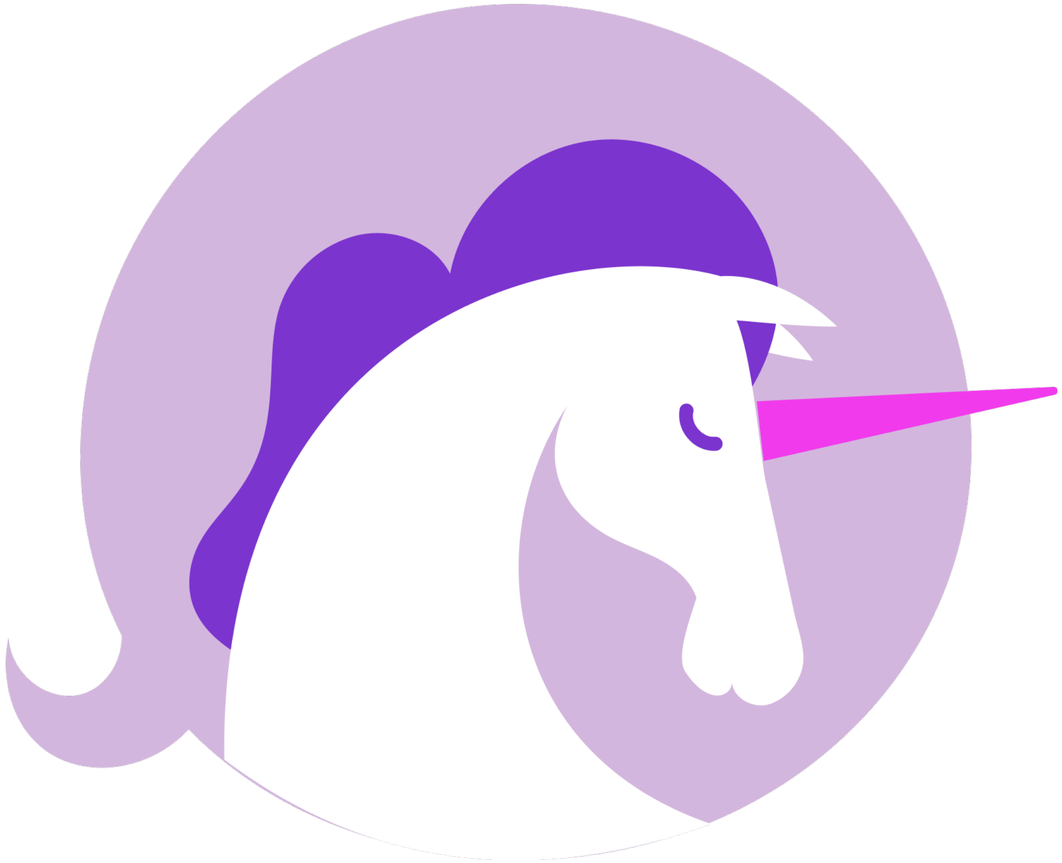
IEEE Software (1990)

Reengineering is not a larger refactoring

In comparison

	Refactoring	Reengineering
Goal	Improve code structure without changing external behavior	Enhance maintainability, scalability, and performance while preserving core functionality
Approach	Small, gradual code-level improvements	Modernization of existing system or its components which may involve major architectural changes

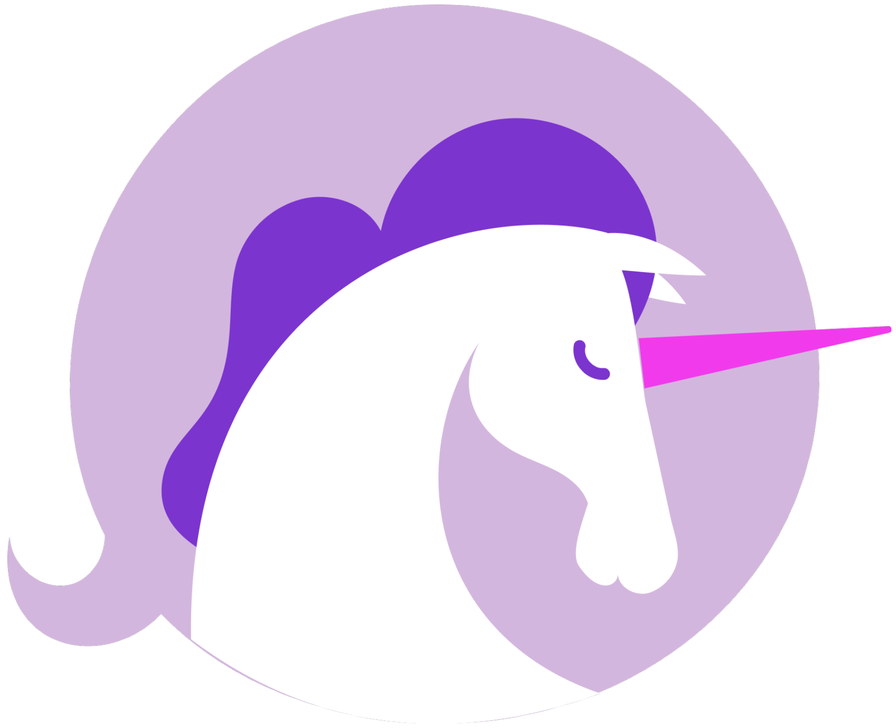
Example: Update React 17 to 18



Example: Update React 17 to 18

Project info:

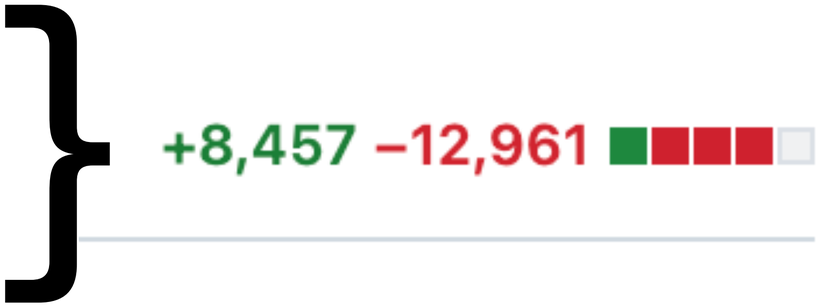
- React + Redux + RTK
- @testing-library/react, Playwright
- 382 034 lines of JS
- 133 666 lines of TS



Example: Update React 17 to 18

Scope of work:

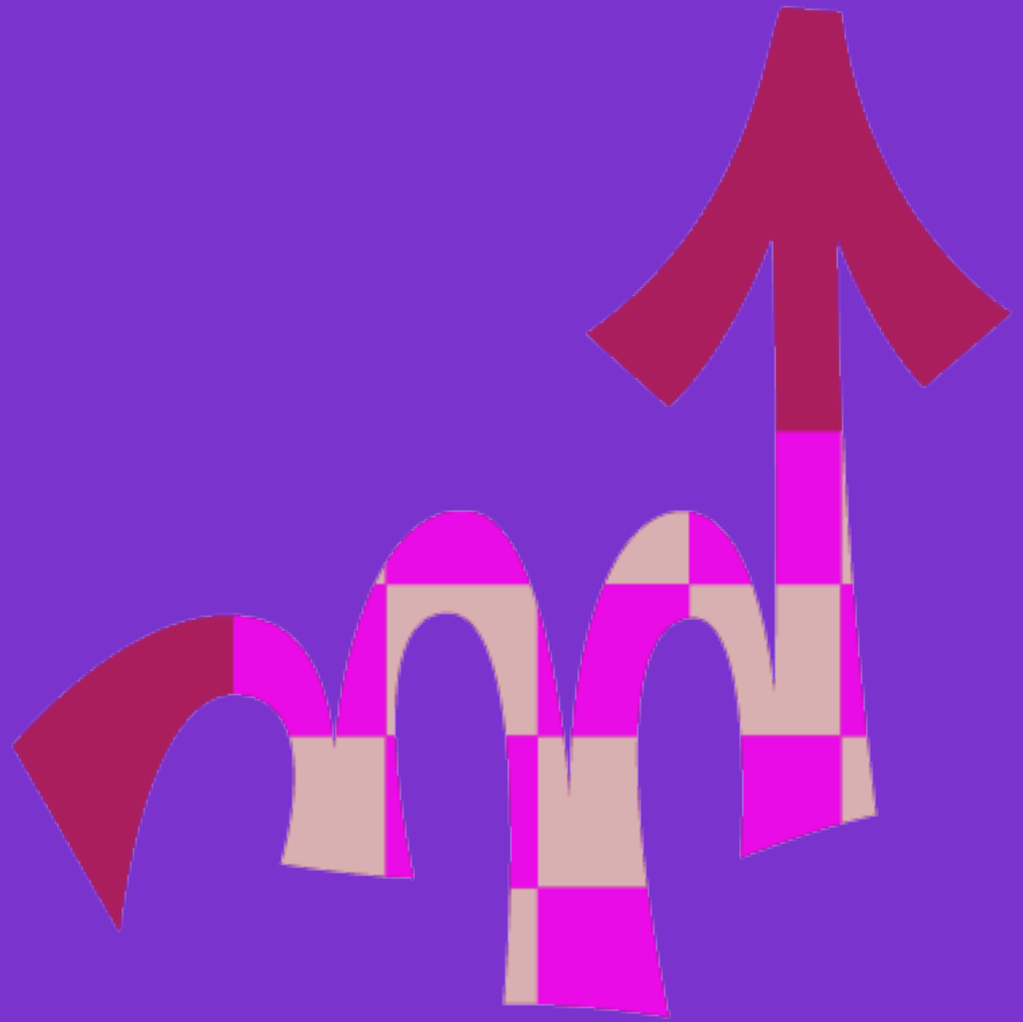
- Update vendor dependencies:
  - Update if possible
  - Replace unupdatable deps with alternatives
  - Write own implentation for unmaintained dependencies
- Refactoring + codemods (FC, DefaultProps, etc.)
- Migration all tests to a new @testing-library/react
- **Update React library** 🚀
- Fix all broken features
- Fix deprecation warnings



Reengineering steps

1. Motivation and goal setting
2. Reverse engineering
3. High level vision, goal finalizing, planning, risk assessment
4. Restructuring & Forward engineering
5. Post reengineering

Insights



Insight 1: Reengineering is hard

Insight 2: Record your motivation of reengineering

Insight 3: Reverse-engineering is a key to success

The better you understand the current system, the more predictable the results will be

Insight 3.1: Usually impossible to do a 100% reverse-engineering

Insight 3.2:

Leave artifacts of reverse-engineering: diagrams, dependency graphs, code structure documentation, use cases etc.

Insight 4: Knowledge sharing

Insight 4: Knowledge sharing

- Documentation
- Master classes
- Pair programming
- Automatic tools like linters

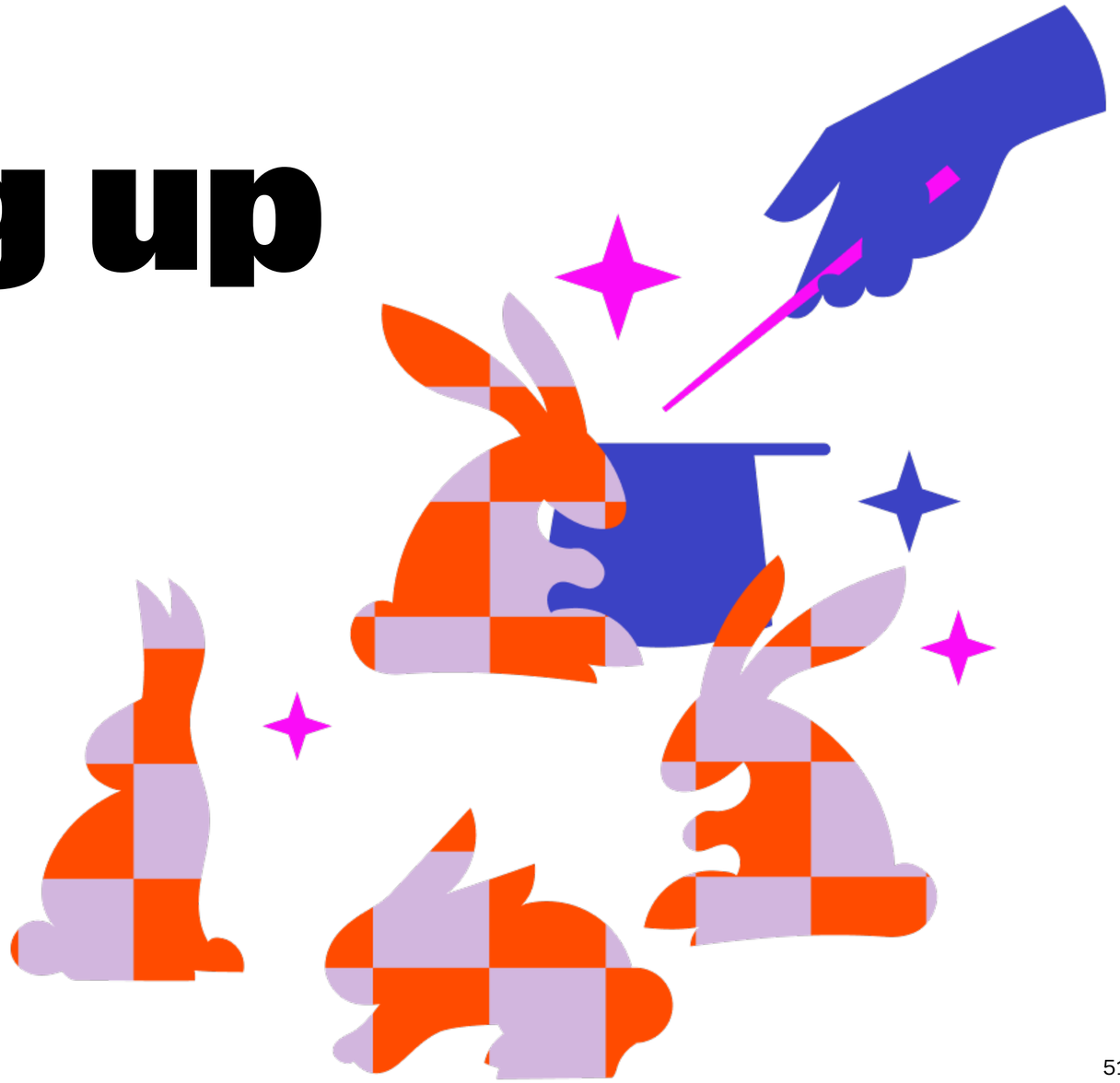
Insight 5:
Pay attention to a feedback loop

Reengineering steps

1. Motivation and goal setting
2. Reverse engineering
3. High level vision, goal finalizing, planning, risk assessment
4. Restructuring & Forward engineering
5. Post reengineering

01 Refactoring
02 Reengineering

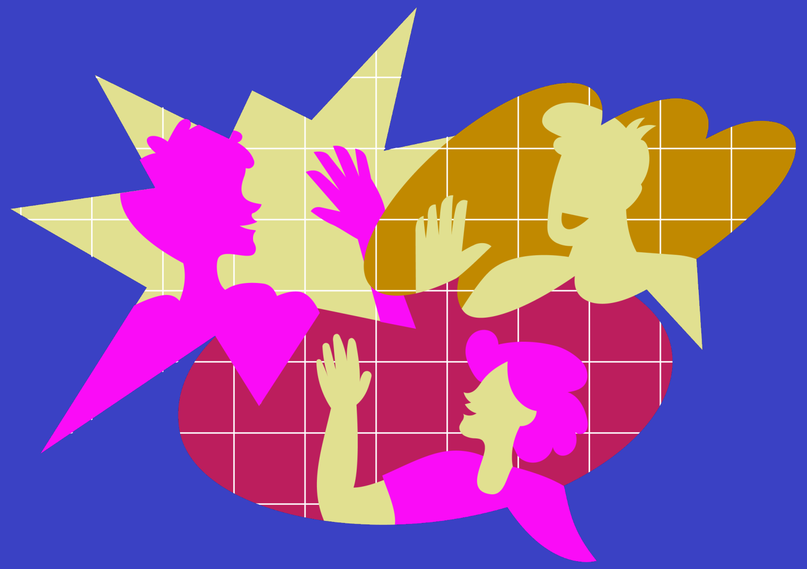
03 Wrapping up



Approaches comparison

	Refactoring	Reengineering	Rewriting
Goal	Improve code structure without changing external behavior	Enhance maintainability, scalability, and performance while preserving core functionality	Create a new system or subsystem
Approach	Small, gradual code-level improvements	Modernization of existing system or its components which may involve major architectural changes	Complete overhaul, developing from scratch
Risk level	Lower risk, as changes are incremental	Moderate risk, requires careful planning and execution	Higher risk, as it involves replacing entire system
Stakeholders	Developers	Developers + Business	Entire organization / Affected departments
Complexity	⭐	⭐⭐⭐	⭐⭐

Thank you



LinkedIn :
Egor Naidikov

Email :
enaidikov@gmail.com

Resources