Андрій Овчаров

Як рефакторити хаотичний JavaScript-код: покроковий гайд

Чи стикалися ви з JavaScript-кодом, який виглядає як лабіринт? Змінні з незрозумілими назвами, функції на сотні рядків, змішана логіка — усе це ускладнює підтримку та розвиток проєкту. Рефакторинг хаотичного коду може здаватися складним завданням, але з правильним підходом ви можете перетворити "спагеті-код" на читабельний і структурований. У цьому гайді я поділюся покроковим процесом рефакторингу JavaScript-коду, з практичними прикладами та порадами, які допоможуть вам зробити ваш код чистішим і ефективнішим.

Чому рефакторинг важливий?

Рефакторинг — це не просто "прибирання" коду. Це інвестиція в майбутнє вашого проєкту. Хаотичний код призводить до:

- Складнощів у підтримці: Зміни займають більше часу через заплутану логіку.
- Помилок: Незрозумілий код частіше містить баги.
- Проблем у командній роботі: Нові розробники витрачають години, щоб розібратися в коді.

Рефакторинг покращує читабельність, зменшує технічний борг і полегшує масштабування. Тож давайте розглянемо покроковий підхід до рефакторингу хаотичного JavaScript-коду.

Покроковий гайд із рефакторингу

Крок 1: Оцініть стан коду

Перш ніж почати рефакторинг, зрозумійте, що саме потрібно виправити. Задайте собі питання:

- Чи є змінні з нечіткими назвами (наприклад, x, data)?
- Чи містять функції забагато логіки?
- Чи є дублювання коду?
- Чи є тести, які допоможуть перевірити зміни?

Приклад хаотичного коду:

```
function doStuff(x, y) {
  let z = x + y;
  if (z > 0) {
    let arr = [];
    for (let i = 0; i < x; i++) {
      arr.push(i * y);
    }
    return arr;
  }
  return [];
}
```

Проблеми: незрозумілі назви (doStuff, x, y, z, arr), змішана логіка, відсутність коментарів.

Крок 2: Додайте тести (якщо їх немає)

Рефакторинг без тестів — це ризик зламати функціонал. Якщо у вас немає тестів, створіть їх, щоб переконатися, що поведінка коду не зміниться після рефакторингу. Використовуйте бібліотеки, такі як Jest або Mocha.

Приклад тесту для функції вище:

```
describe('doStuff', () => {
  it('should return an array of multiples', () => {
    expect(doStuff(3, 2)).toEqual([0, 2, 4]);
    expect(doStuff(0, 5)).toEqual([]);
    expect(doStuff(-1, 2)).toEqual([]);
  });
});
```

Тести дозволяють перевірити, що функція повертає масив чисел, кратних y, для заданої довжини x.

Крок 3: Покращте іменування

Чіткі назви змінних і функцій роблять код зрозумілим. Перейменуйте функцію та змінні, щоб вони відображали їхню мету.

Рефакторений код:

```
function generateMultiples(count, multiplier) {
  let total = count + multiplier;
  if (total > 0) {
    let multiples = [];
    for (let i = 0; i < count; i++) {
      multiples.push(i * multiplier);
    }
    return multiples;
  }
  return [];
}
```

Покращення: doStuff → generateMultiples, x → count, y → multiplier, arr → multiples.

Крок 4: Розбийте великі функції

Якщо функція виконує кілька завдань, розбийте її на менші. У нашому прикладі логіку створення масиву можна винести в окрему функцію.

Рефакторений код:

```
function generateMultiples(count, multiplier) {
  if (count + multiplier <= 0) {
    return [];
  }
  return createMultiplesArray(count, multiplier);
}

function createMultiplesArray(count, multiplier) {
  let multiples = [];
  for (let i = 0; i < count; i++) {
    multiples.push(i * multiplier);
  }
  return multiples;
}
```

Покращення: Логіка розбита на дві функції з чіткими задачами. Функція generateMultiples перевіряє умову, а createMultiplesArray створює масив.

Крок 5: Усуньте дублювання коду

Шукайте повторювані фрагменти коду та замінюйте їх функціями чи утилітами. Наприклад, якщо у вашому проєкті кілька функцій форматують дати, створіть одну утиліту.

Приклад дублювання:

```
function getUserInfo(user) {
  return `${user.firstName} ${user.lastName}`;
}

function getUserDisplay(user) {
  return `${user.firstName} ${user.lastName}`;
}
```

Рефакторений код:

```
function formatFullName(user) {
  return `${user.firstName} ${user.lastName}`;
}

function getUserInfo(user) {
  return formatFullName(user);
}

function getUserDisplay(user) {
  return formatFullName(user);
}
```

Покращення: Усунуто дублювання шляхом створення функції formatFullName.

Крок 6: Спростіть умовну логіку

Складні умовні конструкції роблять код важким для читання. Використовуйте раннє повернення (early return) або спрощені умови.

Приклад складної логіки:

```
function processData(data) {
  let result = null;
  if (data) {
    if (data.length > 0) {
      result = data.map(item => item * 2);
    } else {
      result = [];
    }
  } else {
    result = [];
  }
  return result;
}
```

Рефакторений код:

```
function processData(data) {
  if (!data || data.length === 0) {
    return [];
  }
  return data.map(item => item * 2);
}
```

Покращення: Усунуто вкладені умови за допомогою раннього повернення.

Крок 7: Використовуйте сучасні можливості JS

Скористайтеся сучасними конструкціями JavaScript, такими як стрілкові функції, деструктуризація чи методи масивів, щоб зробити код компактнішим.

Приклад застарілого коду:

```
function getUserNames(users) {
  let names = [];
  for (let i = 0; i < users.length; i++) {
    names.push(users[i].name);
  }
  return names;
}
```

Рефакторений код:

```
function getUserNames(users) {
  return users.map(({ name }) => name);
}
```

Покращення: Використано метод map і деструктуризацію для компактності та читабельності.

Крок 8: Додайте інструменти для підтримки чистоти коду

Використовуйте інструменти, такі як ESLint і Prettier, для автоматичного виявлення проблем у коді та забезпечення єдиного стилю. Наприклад, ESLint може попередити про невикористані змінні чи нечіткі назви.

Приклад конфігурації ESLint:

```
{
  "env": {
    "browser": true,
    "es2021": true
  },
  "extends": ["eslint:recommended"],
  "rules": {
    "no-unused-vars": "warn",
    "consistent-return": "error"
  }
}
```

Крок 9: Перевірте рефакторинг

Після всіх змін запустіть тести, щоб переконатися, що функціонал не зламався. Якщо тестів немає, протестуйте код вручну. Наприклад, для функції generateMultiples:

```
console.log(generateMultiples(3, 2)); // [0, 2, 4]
console.log(generateMultiples(0, 5)); // []
```

Крок 10: Документуйте зміни

Якщо ваш код використовується в команді, задокументуйте ключові зміни в коментарях або в документації проєкту. Наприклад:

```
// Generates an array of multiples based on count and multiplier
function generateMultiples(count, multiplier) {
  if (count + multiplier <= 0) {
    return [];
  }
  return createMultiplesArray(count, multiplier);
}
```

Поради для успішного рефакторингу

- Рефакторте поступово: Не намагайтеся виправити весь код одразу. Працюйте над однією функцією чи модулем.
- Використовуйте версійний контроль: Перед рефакторингом створіть гілку в Git, щоб зберегти можливість повернутися до попередньої версії.
- Залучайте команду: Обговоріть зміни з колегами, щоб переконатися, що новий код відповідає стандартам проєкту.
- Вивчайте патерни: Ознайомтеся з книгою "Refactoring" Мартіна Фаулера або принципами чистого коду Роберта Мартіна (Uncle Bob).

Реальний приклад: рефакторинг React-компонента

Розгляньмо хаотичний React-компонент:

```
function Comp(props) {
  let d = props.data;
  let x = [];
  for (let i = 0; i < d.length; i++) {
    if (d[i].active) {
      x.push(<p>{d[i].name}</p>);
    }
  }
  return <div>{x}</div>;
}
```

Рефакторений код:

```
function ActiveUserList({ users }) {
  const activeUsers = users.filter(user => user.isActive);
  return (
    <div>
      {activeUsers.map(user => (
        <p key={user.id}>{user.name}</p>
      ))}
    </div>
  );
}
```

Покращення:

- Чітка назва компонента: Comp → ActiveUserList.
- Деструктуризація пропсів: props.data → { users }.
- Використання filter і map замість циклу.
- Додано key для списку, щоб уникнути помилок React.

Висновок

Рефакторинг хаотичного JavaScript-коду — це мистецтво, яке вимагає терпіння та системного підходу. Починайте з оцінки коду, додавайте тести, покращуйте іменування, розбивайте функції та використовуйте сучасні можливості JS. Результатом буде код, який легко читати, підтримувати та масштабувати.