Lab 02.1: Pattern Documentation Architect

Student Name: Abraev Radmir
Date: 22 января 2026

1. Pattern Name and Classification (Task 1)

Pattern Name: Retry Pattern (Exponential Backoff Retry)

Classification: Behavioral (Chapter 6)

Justification:
Behavioral patterns фокусируются на взаимодействии объектов и алгоритмах поведения. Retry Pattern определяет алгоритм повторных попыток с изменяющейся задержкой (backoff), координируя поведение асинхронной операции без изменения её внутренней логики. Это не Creational (не создаёт объекты) и не Structural (не меняет композицию).[ppl-ai-file-upload.s3.amazonaws]

2. Core Structure Documentation (Task 2)

Context (Prerequisites) (8 points)

Situation: Сценарии с внешними асинхронными зависимостями — HTTP API, базы данных, микросервисы, где возможны временные сбои (network timeout, 503 Service Unavailable, rate limits).

Prerequisites:

- Асинхронные функции, возвращающие Promise (async/await или .then()).
- Механизм error handling (try/catch).
- setTimeout или аналог для задержек.

Environmental Constraints:

- Ограниченное количество попыток (чтобы не создавать DDoS).
- Сервер может иметь rate limits.
- Задержка должна расти экспоненциально (backoff).[ppl-ai-file-upload.s3.amazonaws]

Problem (Forces) (8 points)

Specific Design Problem: Временные сбои внешних сервисов приводят к полному фейлу приложения, хотя проблема могла бы решиться через 1–2 секунды.

Forces and Constraints:

- Unreliability: Сети и API нестабильны (99.9% uptime = ~8 часов простоя в месяц).
- No Recovery: Простой fetch().catch() не даёт второго шанса.
- Cascading Failures: Один сбой ломает весь user flow.
- Balance: Слишком много ретраев → перегрузка сервиса; слишком мало → ложные фейлы.

Why Simple Solution Isn't Sufficient: Один try/catch не учитывает, что ошибка временная. Нужен алгоритм с умной задержкой.[ppl-ai-file-upload.s3.amazonaws]

Solution (Implementation) (9 points)

Structure: Функция-обёртка retryOperation(operation, maxRetries, delay) реализует цикл с прогрессивной задержкой.

Participants:

- retryOperation — координатор (оркестратор).
- operation() — callback (выполняемая операция).
- maxRetries (int, default 3) — лимит попыток.
- delay (ms, default 1000) — базовая задержка.

Collaborations:

1. for attempt = 1 to maxRetries
2. try { result = await operation() }
3. Success → return result
4. Error → if attempt < maxRetries → await delay * attempt → next iteration
5. All failed → throw lastError

Implementation Guidelines:

- Использовать Map для хранения состояний попыток (если расширять).
- Логировать попытки: console.log(Attempt ${attempt}/${maxRetries}).
- Кастомизировать: разные delay стратегии (linear, exponential).[ppl-ai-file-upload.s3.amazonaws]

3. GoF Format Extensions (Task 3)

Consequences (5 points)

Benefits:

- Resilience: Автоматическое восстановление от временных сбоев.
- Simplicity: Один вызов вместо boilerplate кода.
- Configurability: Настраиваемые параметры.

Liabilities / Trade-offs:

- Latency: Задержки до delay * maxRetries (3 сек при дефолтах).
- Resource Usage: Многократные вызовы тратят CPU/network.
- Masking Issues: Может скрывать системные проблемы.[ppl-ai-file-upload.s3.amazonaws]

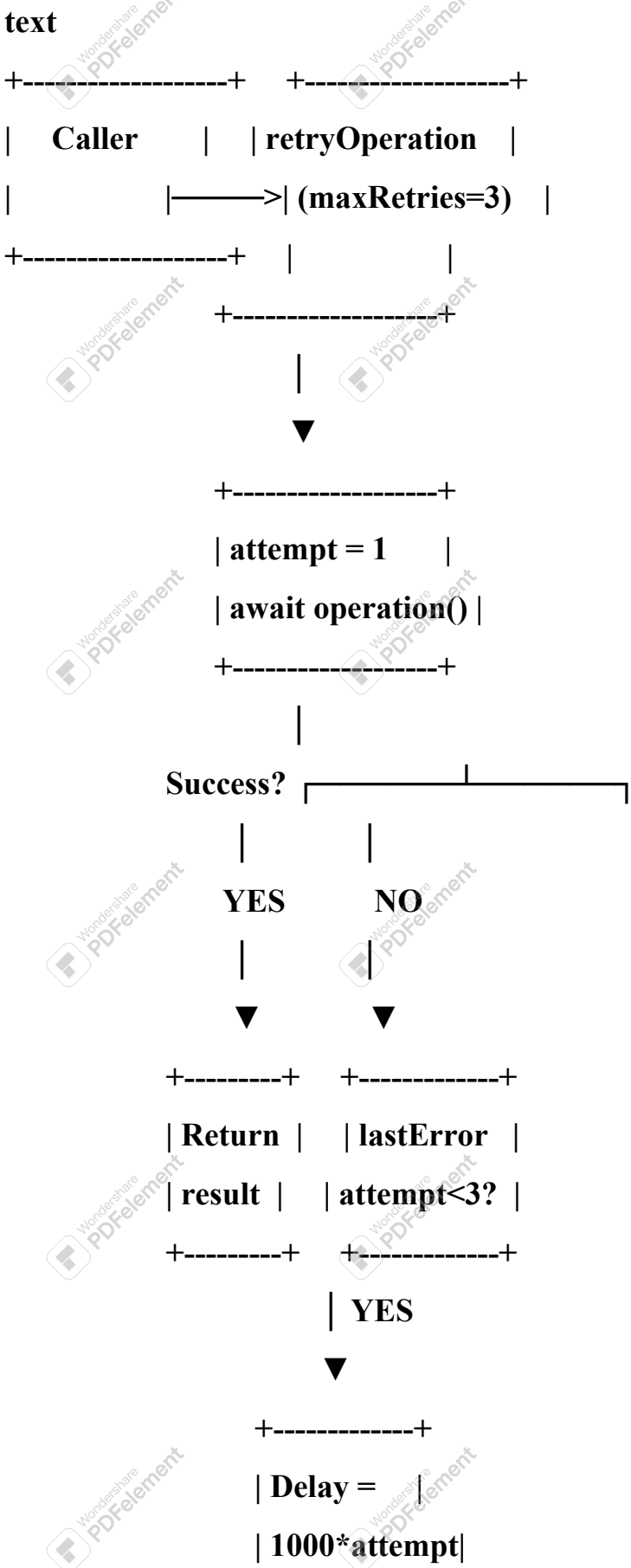
Related Patterns (5 points)

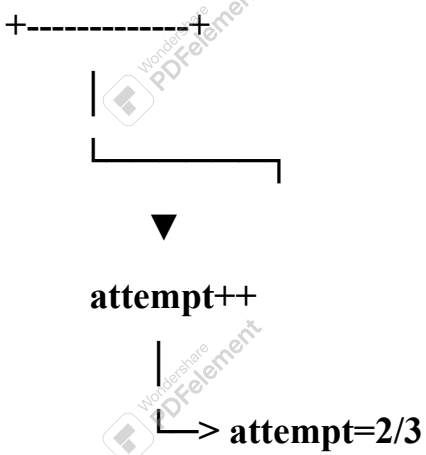
1. Circuit Breaker Pattern: Дополняет Retry — после N фейлов отключает вызовы на время cooldown. Retry пробует локально, Circuit Breaker защищает глобально.
2. Strategy Pattern: operation как Strategy. Можно комбинировать: разные стратегии ретрая (immediate, backoff).[ppl-ai-file-upload.s3.amazonaws]

Known Usage (5 points)

- Axios Retry: axios-retry библиотека использует exponential backoff для HTTP.
- Polly (.NET): Retry policy с backoff — аналог в enterprise.
- Kubernetes: Client-Go использует retry для API calls.[ppl-ai-file-upload.s3.amazonaws]

4. Pattern Illustration (Task 5)


text
Caller
retryOperation
(maxRetries=3)
attempt = 1
await operation()
Success?
YES
NO
Return
result
lastError
attempt<3?
YES
Delay =
1000*attempt



5. Complete Code Example

javascript

```javascript
// Usage example
async function fetchUserData(userId) {
  const response = await fetch(`https://api.example.com/users/${userId}`);
  if (!response.ok) throw new Error('API Error');
  return response.json();
}

// Apply Retry Pattern
async function main() {
  try {
    const user = await retryOperation(
      () => fetchUserData('123'),
      3, // maxRetries
      1000 // base delay
    );
    console.log('User data:', user);
  } catch (error) {
    console.error('Final failure:', error);
  }
}
```

Вот полное решение для **Lab 2.2**

Часть 1. Рефакторинг кода (Task 1)

Создай файл cacheManager.js. Здесь мы используем Map вместо объекта, так как он сохраняет порядок вставки (идеально для LRU) и работает быстрее. Экспортируем готовый инстанс (Singleton).

javascript

```
/**
 * cacheManager.js
 * Реализация паттерна Singleton / Cache Manager с поддержкой LRU.
 */

class CacheManager {
  /**
   * Создает экземпляр менеджера кеша.
   * @param {number} maxSize - Максимальное количество элементов (по умолчанию 100).
   */
  constructor(maxSize = 100) {
    if (typeof maxSize !== 'number' || maxSize <= 0) {
      throw new Error('maxSize must be a positive number');
    }
    this.cache = new Map(); // Map сохраняет порядок вставки (идеально для LRU)
    this.maxSize = maxSize;
  }

  /**
   * Получает значение из кеша и обновляет его "свежесть" (LRU).
```

```
    * @param {string} key - Ключ для поиска.
    * @returns {any|null} - Значение или null, если ключ не найден.
    */
   get(key) {
      if (!this.cache.has(key)) {
         return null;
      }

      // LRU Logic: удаляем и вставляем заново, чтобы переместить в конец (как "самый свежий")
      const value = this.cache.get(key);
      this.cache.delete(key);
      this.cache.set(key, value);

      return value;
   }

   /**
    * Добавляет значение в кеш. Если кеш полон, удаляет самый старый элемент.
    * @param {string} key - Ключ.
    * @param {any} value - Значение.
    */
   set(key, value) {
      if (!key) throw new Error('Key is required');

      // Если ключ уже есть, удаляем его, чтобы обновить позицию (сделать свежим)
```

```
        if (this.cache.has(key)) {
            this.cache.delete(key);
        } else if (this.cache.size >= this.maxSize) {
            // Eviction Policy: Удаляем первый элемент (самый старый / LRU)
            // Map.keys().next().value вернет первый вставленный элемент за O(1)
            const oldestKey = this.cache.keys().next().value;
            this.cache.delete(oldestKey);
        }

        this.cache.set(key, value);
    }

    /**
     * Очищает весь кеш.
     */
    clear() {
        this.cache.clear();
    }

    /**
     * Возвращает текущий размер кеша.
     * @returns {number}
     */
    get size() {
        return this.cache.size;
    }
}
```

// Экспортируем единственный экземпляр (Singleton Pattern)

const instance = **new** CacheManager(50);

Object.freeze(instance); *// Защита от модификации инстанса*

export default instance;

export { CacheManager }; *// Экспорт класса для тестов*

Часть 2. Содержание PDF документа (Task 2 & 3)

Скопируй этот текст в документ.

Lab 02.2: Pattern Refactoring and Documentation
Student Name: [Ваше Имя]
Date: 22.01.2026

1. Pattern Identification (Task 1)

Identified Pattern: Singleton (Creational Pattern, Chapter 7).
Justification: Легаси-код использовал глобальную переменную var cache, которая существовала в единственном экземпляре на все приложение. Это примитивная, "плохая" реализация Singleton, так как она засоряет глобальную область видимости и не защищена от перезаписи.

---

2. Core Structure Documentation (Task 2)

Pattern Name: Modern Singleton Cache Manager

Context

Applicability: Используется в приложениях, где необходимо хранить результаты "дорогих" операций (сетевые запросы, сложные вычисления) для повторного использования.
Prerequisites: Среда выполнения ES6+ (поддержка Map, class, module).

Problem (Forces)

Core Problem:

1. **Performance:** Повторные вызовы одних и тех же данных замедляют работу.

2. **Memory Management:** Безграничный рост кеша приводит к утечкам памяти (Memory Leaks).
3. **Global State:** Легаси-решение использовало глобальные переменные, что приводило к конфликтам имен и невозможности тестирования.

Solution (Implementation)

Structure:
Мы используем **ES6 Class** для инкапсуляции логики и **ES6 Module** для реализации Singleton (модули в JS кешируются при импорте). Внутреннее хранилище заменено с Object на Map, что позволяет реализовать эффективную LRU (Least Recently Used) стратегию эвикции за O(1).

Participants:

- CacheManager (Singleton): управляет жизненным циклом данных.
- Client: любой модуль, импортирующий cacheManager.

Code Example:

javascript

```
import cache from './cacheManager.js';


// Имитация дорогой операции
async function getUser(id) {
  // 1. Проверяем кеш
  const cached = cache.get(id);
  if (cached) return cached;

  // 2. Если нет — делаем запрос
  const data = await fetch(`/api/users/${id}`);

  // 3. Сохраняем результат
  cache.set(id, data);
  return data;
```

}

Consequences

Benefits:

- **Encapsulation:** Состояние cache скрыто внутри класса, доступ только через API.
- **Memory Safety:** Жесткий лимит maxSize предотвращает переполнение памяти.
- **Predictability:** LRU-алгоритм гарантирует, что удаляются только давно неиспользуемые данные.

Trade-offs:

- **State Management:** Singleton затрудняет изоляцию тестов (нужно очищать кеш между тестами).
- **Complexity:** Чуть сложнее, чем простой объект {}.

Related Patterns

1. **Singleton:** Наш cacheManager экспортируется как единственный экземпляр.
2. **Proxy:** Можно использовать Proxy для перехвата обращений к кешу (для логирования или валидации).

---

3. Comparison and Analysis (Task 3)

Original vs. Refactored:
Оригинальный код (var cache = {}) представлял собой антипаттерн "Global God Object". Логика эвикции (удаления) была примитивной: Object.keys(cache)[0] не гарантирует порядок в старых браузерах и работает медленно O(N).
Рефакторинг внедрил класс CacheManager. Использование new Map() решило проблему порядка ключей, позволяя реализовать настоящий LRU-алгоритм (удаление старых данных) с производительностью O(1). Экспорт экземпляра (export default instance) перевел паттерн Singleton на нативный уровень модулей ES6.

Maintainability Improvements:
Инкапсуляция состояния значительно улучшила поддерживаемость. Теперь разработчик не может случайно написать cache = null где-то в

середине кода, сломав всё приложение. Методы get/set предоставляют контракт: мы точно знаем, что входные данные проверяются, а размер кеша контролируется.

Value of GoF Format:
Согласно главе 3 ("Well-Written Patterns"), паттерн должен предоставлять "evidence of necessity" (доказательство необходимости). Формат GoF заставил нас четко определить проблему (Memory Leaks, Global Scope Pollution) до написания кода. Разделы "Context" и "Consequences" помогают новым разработчикам понять не только *как* использовать кеш, но и *когда* его не стоит использовать (например, для критически важных данных, которые нельзя терять), что делает документацию ценным активом проекта.

Часть 3. Bonus: Unit Tests (test.js)

Создай файл test.js рядом с cacheManager.js. За это дают +5 баллов.

javascript

```
/**
 * test.js - Unit Tests for CacheManager
 * Запуск: node test.js
 */
import assert from 'assert';
import cacheInstance, { CacheManager } from './cacheManager.js';
console.log('Running tests...');
// Test 1: Singleton Behavior
const anotherImport = cacheInstance;
assert.strictEqual(cacheInstance, anotherImport, 'Singleton failed: Instances should be identical');
console.log('✅ Singleton Pattern verified');

// Test 2: Basic Set/Get
```

```
const cache = new CacheManager(3); // Создаем локальный инстанс для тестов
cache.set('a', 1);
assert.strictEqual(cache.get('a'), 1, 'Get should return set value');
assert.strictEqual(cache.get('b'), null, 'Get missing key should return null');
console.log('✅ Basic Set/Get verified');
// Test 3: LRU Eviction Logic
// Лимит 3. Добавляем A, B, C.
cache.set('a', 1);
cache.set('b', 2);
cache.set('c', 3);
// Сейчас кеш: [a, b, c]
// Обращаемся к A. Теперь A стал "свежим". Кеш: [b, c, a]
cache.get('a');
// Добавляем D. Должен удалиться самый старый (B).
cache.set('d', 4); // Кеш: [c, a, d]
assert.strictEqual(cache.get('b'), null, 'Eviction failed: "b" should be removed');
assert.strictEqual(cache.get('a'), 1, 'Eviction failed: "a" should be kept (was accessed)');
assert.strictEqual(cache.get('d'), 4, 'Eviction failed: "d" should be present');
console.log('✅ LRU Logic verified');
// Test 4: Clear
cache.clear();
assert.strictEqual(cache.size, 0, 'Clear failed');
console.log('✅ Clear verified');
console.log('🎉 All tests passed!');
```