Relatório Técnico: Refatoração do Serposcope com Padrões de Projeto

Luis Fernando Batista Lima - 538134
Vitor Loula Silva - 540622

Introdução

O **Serposcope** é um *verificador* de posicionamento de código aberto, projetado para monitorar a classificação de sites em mecanismos de busca. Esta aplicação permite o acompanhamento de um número ilimitado de palavras-chave, a gestão de várias contas de usuário, o rastreamento de concorrentes e a integração com proxies e soluções de captcha. A versão disponível no repositório serphacker/serposcope é classificada como um **sistema legado**: o autor considera o código antigo e em *"modo de* sobrevivência", sem suporte ou correções de bugs, e recomenda uma reescrita completa. Tecnicamente, a aplicação original foi desenvolvida com o framework web **Ninja**, a injeção de dependência **Guice** e a biblioteca **QueryDSL** para consultas SQL; o guia de construção sugere a execução em modo "superdev" para permitir recarregamento a quente durante o desenvolvimento.

Visão geral do sistema legado

O Serposcope foi criado para fazer pesquisas automatizadas em mecanismos de busca, como o Google, extraindo o ranking das páginas e gerando relatórios para os administradores. As principais funções destacadas pelo autor incluem:

Funcionalidade (legado)	**Descrição resumida**
Código aberto / self-hosted	Usuários podem executar a ferramenta localmente (Windows, macOS ou Linux) e criar servidores próprios.
Palavras-chave ilimitadas e monitoramento de concorrentes	O software permite adicionar quantas palavras-chave e domínios desejar, inclusive concorrentes diretos.

Pesquisa local, móvel e personalizada	É possível selecionar o tipo de pesquisa (desktop ou mobile) e região/idioma específicos.
Gestão de contas de usuários	O sistema possui um painel administrativo que permite criar usuários e restringir acessos.
Suporte a proxies e captchas	Implementa integração com proxies e resolvers de captcha para reduzir bloqueios.

No repositório original, o autor informa que não haverá correções de bugs, que as *pull requests* não serão aceitas e que o projeto precisa ser reescrito. Isso indica que a base de código está desatualizada, possui dívidas técnicas e provavelmente não adota práticas modernas de design.

Arquitetura técnica

O arquivo BUILD-INSTRUCTIONS.md descreve a pilha tecnológica utilizada no projeto. O backend é desenvolvido com o framework **Ninja**, enquanto o mecanismo de dependência **Guice** é responsável pela injeção de componentes. O acesso ao banco de dados é realizado por meio do **QueryDSL**. Além disso, o sistema utiliza APIs do Google para realizar buscas e emprega *web scraping* para processar os resultados obtidos. Para a construção do projeto, é necessário utilizar o Apache Maven. Recomenda-se o uso do modo super devmode durante o desenvolvimento, pois ele recompila automaticamente o código sempre que há alterações.

Problemas identificados antes da refatoração

Analisando a *branch* main do repositório /serphacker/serposcope, observam-se problemas estruturais clássicos presentes em sistemas legados:

1. **Código monolítico e acoplamento excessivo** – A classe GoogleScraper centraliza as responsabilidades de acesso HTTP, resolução de captchas e, principalmente, a **lógica de parsing** dos resultados de pesquisa. O método que analisa a SERP (página de resultados) contém centenas de linhas de código misturadas para extrair o número de resultados, obter links orgânicos, identificar a próxima página, entre outras funções. Essa abordagem dificulta a leitura e impede a reutilização do parser em outros contextos.

2. **Baixa coesão** – A implementação não isola as variações dos algoritmos de parsing. Se fosse necessário modificar a forma de extrair o número de resultados ou ajustar os seletores HTML, seria preciso alterar a classe GoogleScraper, o que geraria impactos indesejados. A falta de interfaces dificulta a extensão para outros

buscadores ou novos algoritmos.

3. **Duplicação de código e falta de reutilização** – Diversas classes implementam operações semelhantes de acesso HTTP ou manipulação de proxies. A ausência de uma interface comum para um cliente HTTP limita a reutilização desses componentes em outros módulos. Isso também prejudica os testes, pois a classe depende diretamente de implementações concretas.

4. **Dificuldade de manutenção** – O aviso do autor de que o projeto não será mantido e que o código é considerado legado indica que a base não suporta atualizações. As dependências podem estar desatualizadas, e a falta de modularização dificulta a isolação e correção de bugs sem efeitos colaterais.

5. **Ausência de documentação de padrões** – Apesar da complexidade da aplicação, não há menção a padrões de projeto no código original. A falta de clareza na arquitetura complica a integração de novos desenvolvedores e aumenta a curva de aprendizagem.

Justificativa da escolha dos padrões de projeto

Padrão Strategy

O padrão Strategy permite criar várias versões de um algoritmo, cada uma em sua própria classe, e escolher qual usar durante a execução. Isso evita que o código principal precise saber como o algoritmo funciona. No Serposcope, a forma de interpretar os resultados da busca (SERP) muda conforme o buscador, o tipo de dispositivo ou o layout do Google. Para lidar com isso, foi criada a interface SerpParsingStrategy, com diferentes implementações. Assim, o sistema pode se adaptar a mudanças sem precisar alterar o código do GoogleScraper.

Padrão Facade

O padrão Facade cria uma interface simples para acessar sistemas complexos. Ele esconde os detalhes internos e oferece métodos fáceis de usar. No Serposcope, há várias classes que cuidam de proxies, cookies, captchas e buscas. Para facilitar o uso por outras partes do sistema, como a interface gráfica ou uma API REST, foi criado o ScraperFacade. Ele oferece métodos diretos, como scrapeGoogle, e cuida dos bastidores — como configurar o cliente HTTP, aplicar estratégias e definir o user agent.

Padrão Adapter

O padrão Adapter permite que classes com interfaces diferentes funcionem juntas. Ele envolve a classe antiga (adaptee) em uma nova que oferece os métodos esperados. No sistema legado, havia o ScrapClient, um cliente HTTP com uma API própria. Na refatoração, foi criada a interface genérica ScraperHttpClient. Para reaproveitar o ScrapClient sem reescrever tudo, foi criado o ScrapClientAdapter, que implementa a nova interface e redireciona as chamadas para o código antigo.

Padrão Builder

O padrão Builder separa a construção de um objeto complexo da sua representação, permitindo criar diferentes configurações usando o mesmo processo de construção. Ele é especialmente útil quando um objeto possui muitos parâmetros opcionais ou requer validação complexa. No Serposcope, a classe ScrapClient possui diversas configurações — como user agent, timeout, tamanho máximo de resposta, proxy, SSL inseguro, redirecionamentos e cabeçalhos customizados. Antes da refatoração, essas configurações eram definidas através de múltiplos métodos setters ou passadas diretamente no construtor. Para melhorar a legibilidade e facilitar a criação de instâncias configuradas, foi criada a classe ScrapClientConfig com um Builder interno, permitindo uma construção fluente e validação dos parâmetros antes da criação do objeto.

Padrão Factory Method

O padrão Factory Method define uma interface para criar objetos, mas permite que as subclasses decidam qual classe instanciar. Ele delega a lógica de criação de objetos para subclasses especializadas. No ScrapClient, a criação de requisições HTTP (GET e POST) envolve configurações específicas para cada tipo — como headers, referrer e entity. Originalmente, essa lógica estava espalhada nos métodos get e post do ScrapClient. Para isolar e organizar a criação de requisições, foi criada a hierarquia HttpRequestFactory (classe abstrata base) com implementações concretas GetRequestFactory e PostRequestFactory. Cada factory encapsula a lógica específica de criação e configuração do tipo de requisição correspondente.

Benefícios Esperados

Os padrões escolhidos oferecem os seguintes benefícios:

- **Extensibilidade** – É possível implementar novos algoritmos de parsing ou diferentes clientes HTTP como estratégias ou adaptadores, sem precisar modificar as classes existentes. O padrão Factory Method facilita a adição de novos tipos de requisições HTTP. Tudo isso está alinhado ao princípio "aberto/fechado".
- **Reutilização** – Componentes como o ScraperFacade, a interface ScraperHttpClient e as factories de requisição podem ser utilizados por outras partes da aplicação ou em projetos distintos.
- **Testabilidade** – A dependência de interfaces permite a criação de mocks de estratégias e clientes HTTP para testes unitários. As factories podem ser testadas isoladamente. O baixo acoplamento facilita o teste de cada componente de forma isolada.
- **Manutenibilidade** – A clara separação de responsabilidades torna o código mais compreensível e facilita a identificação de bugs. O padrão Builder melhora a legibilidade da configuração. Alterações pontuais em algoritmos não afetam outras partes do sistema.
- **Imutabilidade e Thread-Safety** – O padrão Builder permite criar objetos de configuração imutáveis, garantindo que o estado não seja alterado inadvertidamente após a construção, melhorando a segurança em ambientes multi-thread.

- **Validação Centralizada** – O Builder consolida toda a validação de parâmetros em um único ponto antes da criação do objeto, evitando estados inválidos e melhorando a robustez do sistema.

Descrição das refatorações realizadas (antes/depois)

1) Parsing da SERP (padrão Strategy)

Antes

- A classe GoogleScraper concentrava métodos privados extensos de parsing, como parseSerpLayoutRes, parseSerpLayoutMain, extractResultsNumber e hasNextPage.

- Qualquer mudança no layout do buscador exigia alterar o próprio GoogleScraper, elevando acoplamento e complexidade.

Depois

- Extração do algoritmo de parsing para a interface SerpParsingStrategy e sua implementação padrão DefaultSerpParsingStrategy.

- GoogleScraper delega à estratégia a extração de resultados, contagem e detecção de paginação.

```
public interface SerpParsingStrategy {

    void parse(String html);

    int parseResultsCount(String html);

    boolean hasNextPage(String html);

    List<Result> extractOrganicResults(String html);

}
```

GoogleScraper após a refatoração

```
public class GoogleScraper {

    private final ScraperHttpClient httpClient;
```

```
    private final SerpParsingStrategy parsingStrategy;
    private final CaptchaSolver captchaSolver;
        public GoogleScraper(ScraperHttpClient httpClient, SerpParsingStrategy parsingStrategy, CaptchaSolver solver) {
        this.httpClient = httpClient;
        this.parsingStrategy = parsingStrategy;
        this.captchaSolver = solver;
    }
    public List<Result> scrape(String keyword) {
        httpClient.get(searchUrl(keyword));
        String html = httpClient.getContentAsString();
        parsingStrategy.parse(html);
        int resultsCount = parsingStrategy.parseResultsCount(html);
        boolean next = parsingStrategy.hasNextPage(html);

        return parsingStrategy.extractOrganicResults(html);
    }

    private String searchUrl(String keyword) {
        return "...";
    }
}
```

Impacto

- Inclusão de novos buscadores/dispositivos requer apenas **novas estratégias**.
- Redução de complexidade ciclomática e maior legibilidade/manutenibilidade.

2) Operações HTTP (padrão Adapter)

Antes

- Dependência direta do ScrapClient (legado) para HTTP, cookies, rotas e proxy.
- Dificuldade para testar sem rede e para trocar de biblioteca HTTP no futuro.

Depois

- Criação da interface estável ScraperHttpClient e do adaptador ScrapClientAdapter para reaproveitar o legado sem reescrita.

```
public interface ScraperHttpClient extends Closeable {

    ScrapProxy getProxy();

    void setProxy(ScrapProxy proxy);

    void setUseragent(String useragent);

    void removeRoutes();

    void setRoute(HttpHost to, HttpHost via);

    int get(String url);

    int get(String url, String referrer);

        int post(String url, Map<String,Object> data, ScrapClient.PostType dataType,
String charset, String referrer);

    Exception getException();

    String getResponseHeader(String name);

    String getContentAsString();

    byte[] getContent();

    void clearCookies();

    void addCookie(Cookie cookie);
```

```
}

public class ScrapClientAdapter implements ScraperHttpClient {

    private final ScrapClient delegate;


    public ScrapClientAdapter(ScrapClient delegate) {

        if (delegate == null) throw new IllegalArgumentException("delegate must not be null");

        this.delegate = delegate;

    }

    @Override public ScrapProxy getProxy() { return delegate.getProxy(); }

    @Override public void setProxy(ScrapProxy proxy) { delegate.setProxy(proxy); }

                @Override public void setUseragent(String useragent) { delegate.setUseragent(useragent); }

    @Override public void removeRoutes() { delegate.removeRoutes(); }

     @Override public void setRoute(HttpHost to, HttpHost via) { delegate.setRoute(to, via); }

    @Override public int get(String url) { return delegate.get(url); }

        @Override public int get(String url, String referrer) { return delegate.get(url, referrer); }

            @Override public int post(String url, Map<String,Object> data, ScrapClient.PostType type, String charset, String referrer) {

        return delegate.post(url, data, type, charset, referrer);

    }

    @Override public Exception getException() { return delegate.getException(); }

           @Override public String getResponseHeader(String name) { return delegate.getResponseHeader(name); }
```

```
    @Override public String getContentAsString() { return delegate.getContentAsString(); }

    @Override public byte[] getContent() { return delegate.getContent(); }

    @Override public void clearCookies() { delegate.clearCookies(); }

    @Override public void addCookie(Cookie cookie) { delegate.addCookie(cookie); }

    @Override public void close() throws IOException { delegate.close(); }

}
```

Impacto

- **Desacoplamento** dos scrapers em relação ao cliente HTTP concreto.
- **Testabilidade** com mocks (sem rede).
- **Evolução**: futura troca por OkHttp/Apache HttpClient requer apenas novo adapter.

3) Orquestração e consumo (padrão Facade)

Antes

- Configuração de proxy, user-agent, solver de captcha e seleção de estratégia espalhadas nos chamadores.
- Alto custo de integração e curva de aprendizado.

Depois

- ScraperFacade encapsula composição, configuração e execução do fluxo, expondo uma API simples para camadas superiores (UI/REST/batch).

```
public class ScraperFacade {

    private final ScraperHttpClient httpClient;

    private SerpParsingStrategy parsingStrategy;

    private CaptchaSolver captchaSolver;
```

```
        public ScraperFacade(ScraperHttpClient httpClient, SerpParsingStrategy parsingStrategy, CaptchaSolver solver) {
        this.httpClient = httpClient;
        this.parsingStrategy = parsingStrategy;
        this.captchaSolver = solver;
    }

    public List<Result> scrapeGoogle(String keyword) {
            GoogleScraper scraper = new GoogleScraper(httpClient, parsingStrategy, captchaSolver);
        return scraper.scrape(keyword);
    }
}
```

Impacto

- **API de alto nível** (ex.: scrapeGoogle) para consumidores do subsistema.
- Ponto único para políticas de proxy, rotas, cookies e user-agent.

4) Configuração do cliente HTTP (padrão Builder)

Antes

- Construção do ScrapClient através de construtores simples seguidos de múltiplas chamadas a métodos setters (setUseragent, setTimeout, setProxy, setInsecureSSL, etc.).
- Impossibilidade de validar a configuração completa antes da criação do objeto.
- Código verboso e propenso a erros, especialmente quando múltiplas configurações precisavam ser aplicadas.
- Falta de imutabilidade na configuração, permitindo alterações indesejadas após a construção.

Depois

- Criação da classe ScrapClientConfig com padrão Builder interno, permitindo construção fluente e validação centralizada.
- ScrapClient agora aceita um objeto de configuração imutável no construtor.

```
public class ScrapClientConfig {
        private final String userAgent;
        private final Integer timeoutMS;
        private final int maxResponseLength;
        private final ScrapProxy proxy;
        private final boolean insecureSSL;
        private final int maxRedirect;
        private final List\<Header\> requestHeaders;
        private ScrapClientConfig(Builder builder) {
            this.userAgent \= builder.userAgent;
            this.timeoutMS \= builder.timeoutMS;
            this.maxResponseLength \= builder.maxResponseLength;
            this.proxy \= builder.proxy;
            this.insecureSSL \= builder.insecureSSL;
            this.maxRedirect \= builder.maxRedirect;
                        this.requestHeaders \= Collections.unmodifiableList(new
        ArrayList\<\>(builder.requestHeaders));
        }
        public static Builder builder() {
            return new Builder();
        }
        public static class Builder {
            private String userAgent \= ScrapClient.DEFAULT\_USER\_AGENT;
            private Integer timeoutMS \= ScrapClient.DEFAULT\_TIMEOUT\_MS;
```

```
                                private int maxResponseLength \=
ScrapClient.DEFAULT\_MAX\_RESPONSE\_LENGTH;
    private ScrapProxy proxy \= null;
    private boolean insecureSSL \= false;
    private int maxRedirect \= 0;
    private List\<Header\> requestHeaders \= new ArrayList\<\>();
    public Builder userAgent(String userAgent) {
        this.userAgent \= userAgent;
        return this;
    }
    public Builder timeout(Integer timeoutMS) {
        this.timeoutMS \= timeoutMS;
        return this;
    }
    public Builder proxy(ScrapProxy proxy) {
        this.proxy \= proxy;
        return this;
    }
    public Builder followRedirects() {
        this.maxRedirect \= 10;
        return this;
    }
    public ScrapClientConfig build() {
        validate();
        return new ScrapClientConfig(this);
```

```
        }

        private void validate() {

            if (timeoutMS \!= null && timeoutMS \< 0\) {

                    throw new IllegalArgumentException("Timeout não pode ser
    negativo");

            }

            if (maxResponseLength \< 1024\) {

                throw new IllegalArgumentException("maxResponseLength deve ser
    \>= 1024 bytes");

            }

        }

    }

}
```

Uso do Builder

```
ScrapClientConfig config = ScrapClientConfig.builder()

        .userAgent("CustomBot/1.0")

        .timeout(5000)

        .proxy(myProxy)

        .followRedirects()

        .insecureSSL(true)

        .build();

ScrapClient client = new ScrapClient(config);
```

Impacto

- **Legibilidade:** Código de configuração auto-documentado e fluente.
- **Validação:** Todos os parâmetros são validados antes da construção, evitando estados inválidos.

- **Imutabilidade:** ScrapClientConfig é imutável após a construção, garantindo thread-safety.
- **Flexibilidade:** Fácil adicionar novos parâmetros de configuração sem quebrar código existente.
- **Valores padrão:** Builder fornece valores padrão sensatos para todos os parâmetros.

5) Criação de requisições HTTP (padrão Factory Method)

Antes

- Lógica de criação e configuração de requisições HTTP (GET/POST) misturada nos métodos get e post do ScrapClient.
- Dificuldade para adicionar novos tipos de requisição (PUT, DELETE, PATCH).
- Código duplicado para configuração de headers comuns (user-agent, referrer).
- Violação do princípio da responsabilidade única — ScrapClient responsável tanto pela execução quanto pela criação de requisições.

Depois

- Extração da lógica de criação para hierarquia HttpRequestFactory com implementações especializadas.
- Template Method no método create() define o fluxo de criação e configuração.
- Factory Methods abstratos (createRequest) delegam a criação específica para subclasses.

```
public abstract class HttpRequestFactory {

    protected final String url;

    protected final String userAgent;

    public HttpRequestFactory(String url, String userAgent) {

       this.url \= url;

       this.userAgent \= userAgent;

    }

    public final HttpRequestBase create() {

       HttpRequestBase request \= createRequest();

       configureRequest(request);

       return request;

    }
```

```
    protected abstract HttpRequestBase createRequest();

    protected void configureRequest(HttpRequestBase request) {

       if (request.getFirstHeader("user-agent") \== null) {

          request.setHeader("User-Agent", userAgent);

       }

    }

}
```

Uso das Factories no ScrapClient

```
public int get(String url, String referrer) {

    HttpRequestFactory factory \= new GetRequestFactory(url, useragent, referrer);

    return request(factory.create());

    }

    public int post(String url, Map<String, Object> data, PostType dataType, String charset, String referrer) {

    // ... lógica para criar HttpEntity ...

    HttpRequestFactory factory \= new PostRequestFactory(url, useragent, entity, referrer);

    return request(factory.create());

}
```

Impacto

- **Separação de responsabilidades:** ScrapClient delega a criação de requisições para factories especializadas.
- **Extensibilidade:** Novos tipos de requisição (PUT, DELETE, PATCH) podem ser adicionados criando novas factories sem modificar ScrapClient.
- **Reutilização:** Configuração comum de headers centralizada na classe base HttpRequestFactory.
- **Testabilidade:** Factories podem ser testadas isoladamente, facilitando testes unitários.

- **Princípio aberto/fechado:** Sistema aberto para extensão (novas factories) mas fechado para modificação (ScrapClient).

Conclusão

Este relatório documentou a refatoração do Serposcope através da aplicação de cinco padrões de projeto clássicos, transformando um sistema legado monolítico em uma arquitetura modular, extensível e manutenível.

Resumo dos Padrões Implementados

Padrão	Objetivo	Componentes Principais
Strategy	Isolar algoritmos de parsing da SERP	SerpParsingStrategy, DefaultSerpParsingStrategy
Facade	Simplificar interface do subsistema de scraping	ScraperFacade
Adapter	Integrar cliente HTTP legado com nova interface	ScraperHttpClient, ScrapClientAdapter
Builder	Construir configurações complexas de forma fluente	ScrapClientConfig, ScrapClientConfig.Builder
Factory Method	Delegar criação de requisições HTTP	HttpRequestFactory, GetRequestFactory, PostRequestFactory