Este documento apresenta de forma objetiva os padrões de projeto aplicados no desenvolvimento do sistema estratégico de Fórmula 1. O projeto utiliza quatro design patterns: **Singleton, Builder, Adapter e Strategy.**

Padrões Criacionais

Singleton:

Garante que exista apenas uma instância de uma classe durante toda a execução do programa. Ele é aplicado sempre que o sistema precisa de um único ponto centralizado de acesso a um recurso compartilhado.

Uso no projeto:

O padrão Singleton foi utilizado na classe **EquipeF1**, responsável por armazenar o nome da equipe inserido pelo usuário.

Justificativa:

- Evita criação desnecessária de múltiplas instâncias.
- Centraliza o estado da equipe em um único objeto.
- Simplifica o fluxo principal da aplicação.

```java
package br.f1.equipe;

public class EquipeF1 {  6 usages
    private static EquipeF1 instancia;  3 usages
    private String nome;  2 usages

    private EquipeF1(String nome) { this.nome = nome; }

    public static EquipeF1 getInstancia(String nome) {  1 usage
        if (instancia == null) instancia = new EquipeF1(nome);
        return instancia;
    }

    public String getNome() { return nome; }
}
```

Builder

É recomendado quando um objeto precisa ser construído por etapas, especialmente quando possui vários atributos ou quando o construtor tradicional ficaria longo e difícil de manter.

Uso no projeto:

O padrão foi aplicado na classe CarroF1, que possui diversos atributos (motor, número, piloto). O Builder possibilita criar carros de forma clara e fluida, sem necessidade de construtores com muitos parâmetros.

Justificativa:

- Torna a criação do objeto mais legível.
- Facilita a manutenção e expansão do modelo do carro.
- Evita combinações inválidas e melhora a segurança da construção.

```java
package br.f1.carro;

public class CarroF1 {  5 usages
    private String motor;  2 usages
    private int numero;  2 usages
    private String piloto;  2 usages

    private CarroF1(Builder builder) {  1 usage
        this.motor = builder.motor;
        this.numero = builder.numero;
        this.piloto = builder.piloto;
    }

    public static class Builder {  5 usages
        private String motor;  2 usages
        private int numero;  2 usages
        private String piloto;  2 usages

        public Builder motor(String motor) {  no usages
            this.motor = motor;
            return this;
        }

        public Builder numero(int numero) {  no usages
            this.numero = numero;
            return this;
        }

        public Builder piloto(String piloto) {  no usages
            this.piloto = piloto;
            return this;
        }

        public CarroF1 build() { return new CarroF1( builder: this); }
    }

    public String getInfo() { return "Carro N° " + numero + " | Motor: " + motor + " | Piloto: " + piloto; }
}
```

Padrão Estrutural

Adapter

Permite que classes com interfaces incompatíveis possam trabalhar juntas. Ele funciona como um tradutor entre duas interfaces.

Uso no projeto:

Foi utilizado na classe TelemetriaAdapter, permitindo a integração com o módulo externo SistemaTelemetriaExterno, cuja interface não é diretamente compatível com a interface esperada pelo sistema (Telemetria).

Justificativa:

- Permite integração com sistemas externos sem modificar o código original.
- Garante flexibilidade e fácil substituição do sistema externo futuramente.
- Isola dependências externas, facilitando manutenção e testes.

```java
package br.f1.telemetria;

public interface Telemetria {  2 usages  1 implementation
    String lerDados();  1 usage  1 implementation
}
```

```java
package br.f1.telemetria;

public class SistemaTelemetriaExterno {  3 usages
    public String getData() { return "Velocidade: 320km/h | Temperatura: 850°C"; }
}
```

```java
package br.f1.telemetria;

public class TelemetriaAdapter implements Telemetria {  1 usage
    private SistemaTelemetriaExterno externo;  2 usages

    public TelemetriaAdapter(SistemaTelemetriaExterno externo) { this.externo = externo; }

    public String lerDados() { return externo.getData(); }
}
```

Padrão Comportamental

Strategy

Encapsula diferentes comportamentos ou algoritmos, permitindo que eles sejam trocados em tempo de execução sem modificar as classes que os utilizam.

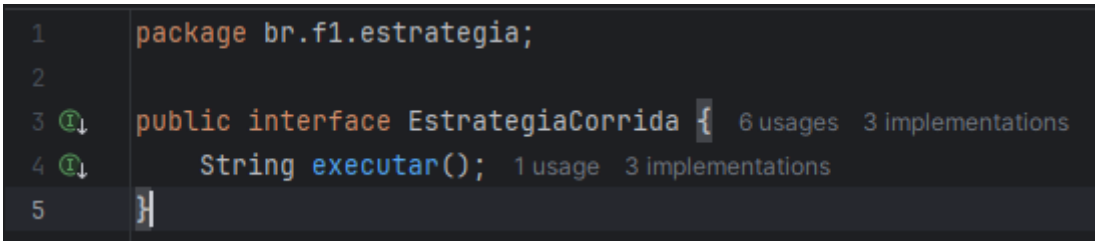
Uso no projeto:

O padrão foi aplicado para gerenciar as estratégias de corrida do piloto. As classes EstrategiaAtaque, EstrategiaEconomia e EstrategiaChuva implementam comportamentos

distintos. O piloto pode trocar livremente a estratégia em tempo real chamando apenas um método.

Justificativa:

- Elimina condicionais extensas no código.
- Facilita a adição de novas estratégias sem alterar o código existente.
- Permite que o comportamento do piloto seja alterado dinamicamente, simulando cenários reais de corrida.



A aplicação desses padrões melhora a arquitetura geral, facilita manutenção, reduz acoplamento e deixa o projeto preparado para evoluções futuras.