Padrão de Projeto: *Template Method*

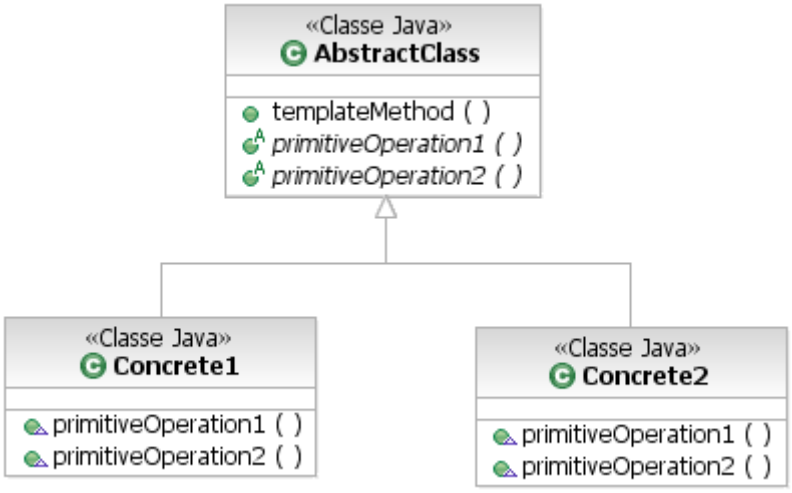
Aplicação:

Define os passos de um algoritmo e permite que a implementação de um ou mais desses passos seja fornecida por subclasses. O *Template Method* protege o algoritmo e fornece métodos abstratos para que as subclasses possam implementá-los.

Benefícios:

Permite reutilizar código sem perder o controle dos nossos algoritmos.

Diagrama de classes:



No diagrama de classe acima temos a classe AbstractClass contendo o método templateMethod() que possui o algoritmo e que define os métodos primitiveOperation1() e primitiveOperation2() que são abstratos. As classes concretas Concrete1 e Concrete2 implementam os métodos abstratos que serão chamados quando templateMethod() precisar delas. Vale salientar que o método templateMethod() é final, ou seja, ele não pode ser sobrescrito, seu algoritmo não pode ser mexido. Já os métodos primitiveOperation1() e primitiveOperation2() podem ser sobrescritos. Além disso, ainda poderíamos ter um método concreto ou ainda um método final que não poderia ser sobrescrito e seria utilizado no algoritmo do templateMethod(). Isso ficará mais claro no exemplo de implementação abaixo.

Exemplo de Implementação em Java:

```java
public abstract class Treinos {

	final void treinoDiario() {
		preparoFisico();
		jogoTreino();
		treinoTatico();
	}

	abstract void preparoFisico();
	abstract void jogoTreino();

	final void treinoTatico() {
		System.out.println("Treino Tatico");
	}

}
```

```java
class TreinoNoMeioDaTemporada extends Treinos {

	void preparoFisico() {
		System.out.println("Preparo Fisico Intenso.");
	}

	void jogoTreino() {
		System.out.println("Jogo Treino com Equipe Reserva.");
	}
}
```

```java
class TreinoNoInicioDaTemporada extends Treinos {

	void preparoFisico() {
		System.out.println("Preparo Fisico Leve.");
	}

	void jogoTreino() {
		System.out.println("Jogo Treino com Equipe Junior.");
	}

}
```

```java
public class App {

    public static void main(String[] args) {

        System.out.println("\n\tTreino no início da Temporada:\n");
        Treinos t1 = new TreinoNoInicioDaTemporada();
        t1.treinoDiario();

        System.out.println("\n\tTreino no meio da Temporada:\n");
        Treinos t2 = new TreinoNoMeioDaTemporada();
        t2.treinoDiario();

        System.out.println("\n\tTreino no final da Temporada:\n");
        Treinos t3 = new TreinoNoFinalDaTemporada();
        t3.treinoDiario();
    }

}
```

Saída no Console em Java:

```
        Treino no início da Temporada:

Preparo Fisico Leve.
Jogo Treino com Equipe Junior.
Treino Tatico

        Treino no meio da Temporada:

Preparo Fisico Intenso.
Jogo Treino com Equipe Reserva.
Treino Tatico

        Treino no final da Temporada:

Preparo Fisico Moderado.
Jogo Treino com Equipe Oficial.
Treino Tatico
```

No exemplo acima temos a classe principal Treinos que tem o método treinoDiario() que tem como principal finalidade ser um algoritmo a ser seguido por todas as outras classes que estenderem essa classe. Este método é final e não pode ser alterado, todos os treinos devem seguir estas etapas. No entanto, as classes TreinoNoMeioDaTemporada e TreinoNoInicioDaTemporada implementam os métodos abstratos preparoFisico() e jogotreino(). Ambos estes métodos podem ser alterados dependendo se o treino está sendo feito no inicio de uma temporada ou no meio dela. Assim cada método é implementado por sua classe especifica, mas sempre seguindo o algoritmo. O método treinoTatico() é definido pelo algoritmo e pela classe base, este método é concreto e final, portanto não pode ser alterado e é seguido por todos os treinamentos. Nada impede também de termos métodos concretos que podem ser sobrescritos nas subclasses ou que simplesmente podem ser usados da sua classe base.

Utilização do Template Method nas API's Java

O Padrão *Template Method* também é utilizado nas API's do Java, como na API Swing onde a classe JFrame define o método paint() como abstrato para ser implementado nas subclasses que estendem JFrame. A substituição do conteúdo de paint() permite que você se conecte ao algoritmo de JFrame para exibir sua área da tela e incorporar a sua própria saída gráfica ao JFrame. Os Applets também utilizam o padrão *Template Method* através dos métodos init(), start(), stop(), destroy(), e outros.

Considerações:

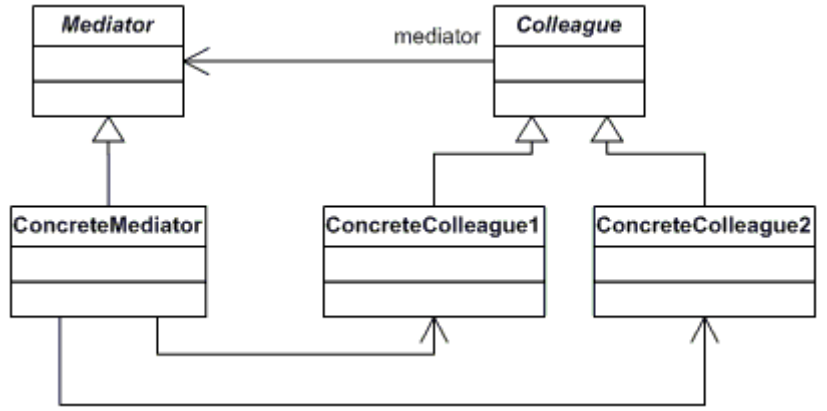
O Padrão *Template Method* nos permite reutilizar código sem perder o controle do nossos algoritmos. No Template Method são definidos passos de um algoritmo, permitindo que alguns desses passos sejam implementados por subclasses. Na classe abstrata temos métodos abstratos, concretos e finais. O *Template Method* é bastante utilizado inclusive na API Java, como pudemos constatar.

Padrão de Projeto: *Mediator*

Aplicação:

Permite criar um objeto que age como mediador, controlando a interação entre um conjunto de objetos. Diminui o acoplamento entre os objetos. É aplicado quando existe um grande número de objetos que se comunicam entre si de maneira bem definida, mas de forma complexa.

Diagrama de classes:



Considerações:

Hierarquia de subclasses é limitada apenas a classe Mediator. Substitui o relacionamento de objetos de * para muitos para um para *. Abstração da interação entre os objetos. Centralização do comportamento.

Exemplo de Implementação em Java:

```java
public abstract class Colleague {
    protected Mediator mediator;

    public Colleague(Mediator m) {
        mediator = m;
    }

    public void enviarMensagem(String mensagem) {
        mediator.enviar(mensagem, this);
    }

    public abstract void receberMensagem(String mensagem);
}
```

```java
public class IOSColleague extends Colleague {

    public IOSColleague(Mediator m) {
        super(m);
    }

    @Override
    public void receberMensagem(String mensagem) {
        System.out.println("iOs recebeu: " + mensagem);
    }
}
```

```java
public class AndroidColleague extends Colleague {

    public AndroidColleague(Mediator m) {
        super(m);
    }

    @Override
    public void receberMensagem(String mensagem) {
        System.out.println("Android recebeu: " + mensagem);
    }
}
```

```java
public class MensagemMediator implements Mediator {

    protected ArrayList<Colleague> contatos;

    public MensagemMediator() {
        contatos = new ArrayList<Colleague>();
    }

    public void adicionarColleague(Colleague colleague) {
        contatos.add(colleague);
    }

    @Override
    public void enviar(String mensagem, Colleague colleague) {
        for (Colleague contato : contatos) {
            if (contato != colleague) {
                definirProtocolo(contato);
                contato.receberMensagem(mensagem);
            }
        }
    }

    private void definirProtocolo(Colleague contato) {
        if (contato instanceof IOSColleague) {
            System.out.println("Protocolo iOS");
        } else if (contato instanceof AndroidColleague) {
            System.out.println("Protocolo Android");
        } else if (contato instanceof SymbianColleague) {
            System.out.println("Protocolo Symbian");
        }
    }

}
```

```java
public interface Mediator {

        void enviar(String mensagem, Colleague colleague);

}
```

Tools:

- http://hilite.me/

Bibliografia:

- http://www.devmedia.com.br/padrao-de-projeto-template-method-em-java/26656
- http://www.devmedia.com.br/padrao-mediator-curso-padroes-de-projeto-com-c-26/27407
- https://sourcemaking.com/design_patterns/mediator
- https://sourcemaking.com/design_patterns/template_method

Bibliografia Complementar:

- Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra. Head First Design Patterns. O'Reilly Media, 2004.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 2010.