Engenharia de Software III

Aula 9
Padrões de Projeto

l.bertholdo@ifsp.edu.br

Conteúdo

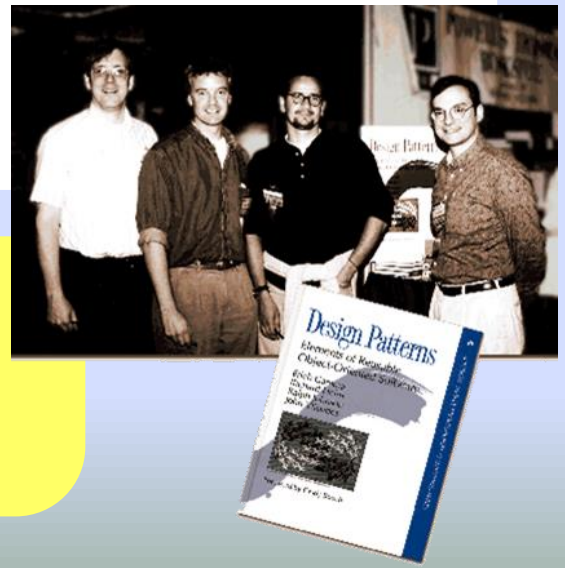
- Padrões de Projeto (*Design Patterns*)
- Objetivos de *Design*
- *Design* Orientado a Objetos
- Padrões de Projeto GoF
  - Observer
  - Singleton
  - Factory

l.bertholdo@ifsp.edu.br

Padrões de Projeto

- Um **padrão de projeto** (*design pattern*) descreve uma solução, baseada na experiência e em boas práticas, previamente testada e aprovada para um problema comum de implementação, de modo que esta solução pode ser **reutilizada** em diferentes projetos.

Imagine que você participa de um projeto de software, no qual surge uma dúvida sobre como resolver um problema relacionado ao design do sistema. É pouco provável que você tenha sido o primeiro a passar por isso e que não existem soluções já desenvolvidas, testadas e utilizadas no mercado com o mesmo cenário do seu projeto. É justamente dessas soluções já existentes que se tratam os padrões de projeto. Eles evitam que tenhamos de "reinventar a roda", economizando tempo e gastos.



*Os padrões de projeto mais conhecidos foram originalmente descritos pela **Gangue dos Quatro (Gang of Four, ou GoF)**, em 1995, no livro **Design Patterns**. Há também os **padrões Java EE** desenvolvidos pela Sun Microsystem, os quais foram influenciados pelos padrões da **GoF**.*

l.bertholdo@ifsp.edu.br

Padrões de Projeto

- Os elementos básicos de um padrão de projeto são:

1. **Contexto:** situação na qual ocorre o problema de *design* a ser tratado.
2. **Problema:** descrição do problema para o qual o padrão se destina.
3. **Solução:** descrição da solução incluindo suas partes, seus relacionamentos e responsabilidades.
4. **Consequências:** vantagens e desvantagens de se utilizar o padrão.
5. **Padrões associados:** padrões similares, ou usados para compor o padrão.

*Com relação à **notação**, os padrões de projeto são representados por meio de **diagramas de classe** e **de sequência** da UML.*

l.bertholdo@ifsp.edu.br

Padrões de Projeto

- Uma dúvida relacionada aos padrões de projeto é **quando devemos utilizá-los**. Muitos desenvolvedores, após aprenderem alguns deles, tendem a querer a resolver todos os problemas utilizando padrões.

- Porém, padrões de projeto não representam uma solução para todos os problemas de *design*. Por isso, é preciso encontrar um meio-termo, para não acabar usando mais padrões do que seria necessário.

- Em muitos casos, também é preciso adaptar os padrões ao nosso contexto, ou, até mesmo, utilizar mais de um padrão em conjunto para resolver um problema individual.

*Os padrões são uma ótima ideia, mas, para usá-los efetivamente, é preciso ter **experiência de desenvolvimento de software**. É preciso reconhecer as situações em que cada padrão pode ser aplicado. Programadores inexperientes, mesmo que tenham lido um livro de padrões, terão dificuldade para decidir se devem usar um padrão ou desenvolver uma solução customizada.*

l.bertholdo@ifsp.edu.br

Objetivos de Design

- Os padrões de projeto possuem **objetivos de *design***, que são as capacidades requeridas pelos **requisitos não funcionais** do sistema:

Adaptabilidade: *atender a novos propósitos dentro do domínio do sistema, sem grandes alterações no sistema.*

Extensibilidade: *adicionar novas funcionalidades ou estender funcionalidades existentes sem impactar o restante do sistema.*

Manutenibilidade: *diminuir o esforço exigido para localizar e reparar erros, ou para adaptar o sistema a um novo ambiente.*

Reusabilidade: *reutilizar componentes do sistema em outras aplicações, ou em outros módulos do mesmo sistema.*

Performance: *garantir um bom desempenho em requisitos como: velocidade, taxa de transferência de dados e tempo de resposta.*

Escalabilidade: *manter boa performance e qualidade no sistema mesmo com o aumento do número de usuários.*

Confiabilidade: *garantir a exatidão das informações, a recuperação do sistema e dados em caso de falhas, e que poucas falhas ocorram durante o uso.*

Usabilidade: *assegurar a facilidade de uso do sistema, de modo que o usuário final possa utilizar suas funcionalidades de forma natural e intuitiva.*

Desenvolvimento eficiente: *empregar um processo eficiente de desenvolvimento com reutilização de código ou de frameworks de mercado.*

l.bertholdo@ifsp.edu.br

Design Orientado a Objetos

- O ***design* orientado a objetos** é baseado em vários conceitos da orientação a objetos, entre eles: encapsulamento, acoplamento, coesão, herança, composição e polimorfismo.

- Além de utilizar amplamente esses **conceitos**, os padrões de projeto também aplicam alguns **princípios** do *design* orientado a objetos:
  - Programação para **interfaces**.
  - Favorecimento da **composição** em vez da **herança**, já que a herança leva a um acomplamento maior entre as classes.
  - Princípio **aberto-fechado**, em que entidades do sistema, como as classes, devem estar abertas para **extensão** e fechadas para **alteração**.
  - *Design* preparado para mudanças, já que os requisitos frequentemente mudam.

l.bertholdo@ifsp.edu.br

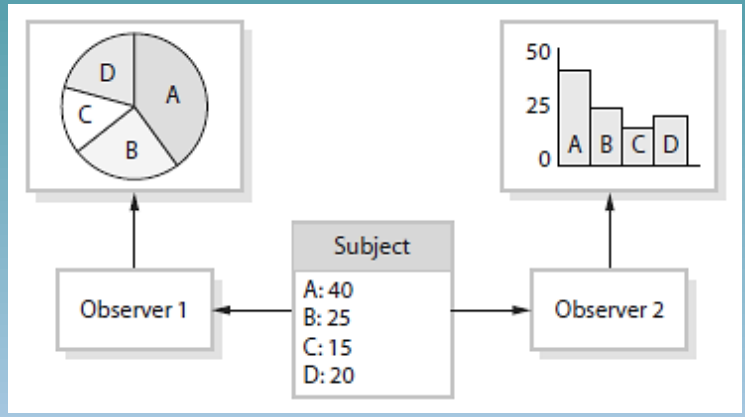
Padrões de Projeto GoF

- O livro **Design Patterns** da GoF apresenta 23 padrões de projeto, os quais são divididos nos grupos a seguir. Segundo os autores:

> *"Um padrão de projeto nomeia, abstrai e identifica os principais aspectos de uma estrutura comum que a tornam útil à criação de um design orientado a objetos reutilizável. O padrão de projeto identifica as classes e instâncias participantes, seus papéis e colaborações e a distribuição de responsabilidades."*

Grupos	Padrões
Padrões de comportamento: descrevem possíveis formas de interação entre os objetos.	Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor
Padrões de criação: descrevem padrões relacionados à criação de objetos.	Abstract Factory, Builder, Factory Method, Prototype, Singleton
Padrões estruturais: descrevem como os objetos podem ser compostos.	Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy

Observer



- O padrão **Observer** permite notificar vários objetos que o estado de algum outro objeto mudou.

- Temos o **objeto observado (*subject*)** e os **objetos observadores (*observers*)**, sendo que o ***subject*** é responsável por notificar os ***observers*** sobre a ocorrência de mudanças e eventos em algum outro objeto.

- Esse padrão também pode ser usado quando é preciso apresentar o estado do objeto modificado de **diferentes formas**.

Exemplo: *Imagine um sistema para uma empresa de telefonia celular que deseja enviar mensagens de promoções aos seus assinantes. Neste cenário, teremos um* ***subject*** *(objeto* ***CentralSubject****) que notificará seus* ***observers*** *(objetos* ***Assinante****), sempre que tais mensagens tiverem que ser enviadas.*

l.bertholdo@ifsp.edu.br

Observer

*Interface **Subject**: define os métodos de **adição** e **exclusão** de observadores, além do método que **notifica** os observadores de algum evento (no nosso caso, uma nova mensagem para todos os assinantes). Essa interface será implementada pela classe concreta **CentralSubject**.*

```
package interfacePackage;

import observer.Mensagem;

public interface Subject {
    public void addObserver(Observer o);

    public void removeObserver(Observer o);

    public void notifyObservers(Mensagem mensagem);
}
```

*Interface **Observer**: define o método responsável pela configuração da **mensagem** de cada objeto **Assinante**. Essa interface será implementada pela classe concreta **Assinante**.*

```
package interfacePackage;

import observer.Mensagem;

public interface Observer {
    public void updateMensagem(Mensagem mensagem);
}
```

l.bertholdo@ifsp.edu.br

Observer

```java
package observer;

import java.util.ArrayList;
import interfacePackage.Observer;
import interfacePackage.Subject;

public class CentralSubject implements Subject {
    private ArrayList<Observer> observers = new ArrayList<Observer>();

    @Override
    public void addObserver(Observer o) { observers.add(o); }

    public void removeObserver(Observer o) { observers.remove(o); }

    public void notifyObservers(Mensagem mensagem) {
        for (Observer o : observers)
            o.updateMensagem(mensagem);
    }

    public ArrayList<Observer> getObservers() { return observers; }
}
```

Aceita qualquer tipo que implemente a interface **Observer**.

Notifica os observadores (objetos **Assinante**) sobre a nova mensagem.

```java
package observer;

import interfacePackage.Observer;

public class Assinante implements Observer {
    private String nome = "", mensagem = "";

    public Assinante(String nome) { this.nome = nome; }

    public String getNome() { return nome; }

    public void setMensagem(Mensagem mensagem) { this.mensagem = mensagem.getMensagem(); }

    public String getMensagem() { return mensagem; }

    @Override
    public void updateMensagem(Mensagem mensagem) {
        this.mensagem = nome + ", você tem uma nova mensagem: " + mensagem.getMensagem();
    }
}
```

Configura a mensagem de cada objeto **Assinante**, a qual é customizada com o **nome** de cada assinante.

do@ifsp.edu.br

Observer

*A classe **Mensagem** encapsula a mensagem principal e representa o **objeto que muda de estado**, no caso, em virtude da definição de uma nova mensagem para os assinantes.*

```
package observer;

public class Mensagem {
    String mensagem = "";

    public Mensagem(String mensagem) { this.mensagem = mensagem; }

    public void setMensagem(String mensagem) { this.mensagem = mensagem; }

    public String getMensagem() { return mensagem; }
}
```

```
package observer;

public class ObserverMain {
    public static void main(String[] args) {
        Mensagem mensagem = new Mensagem("Temos um novo plano para você!");

        CentralSubject cs = new CentralSubject();

        Assinante a = new Assinante("João");
        cs.addObserver(a);

        Assinante b = new Assinante("Maria");
        cs.addObserver(b);

        cs.notifyObservers(mensagem);

        System.out.println("Mensagem assinante A:\n" + a.getMensagem());
        System.out.println("Mensagem assinante B:\n" + b.getMensagem());
    }
}
```

*O resultado mostra que todos os objetos **Assinante** (observadores) foram notificados sobre a nova mensagem pelo objeto **CentralSubject** (observado), e que o estado do objeto **Mensagem** foi apresentado de diferentes formas para cada objeto observador.*

Resultado da execução do código

```
Mensagem assinante A:
João, você tem uma nova mensagem: Temos um novo plano para você!
Mensagem assinante B:
Maria, você tem uma nova mensagem: Temos um novo plano para você!
```

Singleton

- O padrão **Singleton** garante que uma classe possa ser instanciada apenas uma única vez durante a execução do sistema, ou seja, ele assegura que exista um **único objeto** da classe na memória, de modo que este objeto é usado por todos os clientes dessa classe.

- Para implementar esse padrão, a classe deve ter um construtor **privado**, para impedir sua instanciação direta por outras classes.

- Também deve ter um **método estático** para retornar o objeto *singleton* da classe, o qual impedirá a criação de mais de um objeto dela.

Exemplo: *Imagine um sistema em que um perfil de usuários tem privilégios para realizar um determinado tipo de processamento de dados. No entanto, este processamento não pode ser disparado novamente enquanto não for finalizado. Desse modo, quando outro usuário do mesmo perfil tentar disparar um segundo processamento, o sistema deve mostrar que já existe um processamento em andamento e que é preciso aguardar seu término.*

l.bertholdo@ifsp.edu.br

Singleton

*A classe **Singleton** possui o atributo **statusProcess**, que indica se existe um processamento em andamento ou não. O método **processaDados()** realiza o processamento apenas quando não houver nenhum processamento ativo.*

Construtor privado, para impedir sua instanciação direta por outras classes.

O atributo ***instance*** também precisa ser estático, já que métodos estáticos só podem manipular atributos estáticos da classe.

```java
package singleton;

public class Singleton {
    private String statusProcess;
    private static Singleton instance = null;

    private Singleton() { statusProcess = "Processamento inativo."; }

    public static Singleton getInstance() {
        // Só deixa criar uma instância da classe, se ela estiver nula.
        if (instance == null)
            instance = new Singleton();
        return instance;
    }

    public void setStatusProcess(String flagProcessamentoOn) {
        this.statusProcess = flagProcessamentoOn;
    }

    public String getStatusProcess() { return statusProcess; }

    public void processaDados() { /* Realiza processamento */ }
}
```

Como o construtor é privado, não é possível criar uma instância por meio do dele. Logo, este método precisa ser **estático**, de modo que outras classes possam chamá-lo sem uma instância: ***Singleton.getInstance()***.

***Atributos estáticos** não têm vínculos com as instâncias de sua classe. Por isso, seus dados não se alteram conforme a instância que o invoca.*

l.bertholdo@ifsp.edu.br

Singleton

*No resultado, a 1ª instância (**s1**) mostra que o processamento está inicialmente **inativo**. Em seguida, essa instância **ativa** o processamento e exibe seu novo status. Ao criar a 2ª instância (**s2**) e verificar se existe algum processamento em andamento, a aplicação mostra que há um processamento **ativo** e solicita aguardar o término deste.*

```
package singleton;

public class SingletonMain {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        System.out.println(s1.getStatusProcess());

        s1.setStatusProcess("Processamento ativo.");
        System.out.println(s1.getStatusProcess());

        Singleton s2 = Singleton.getInstance();
        if (s2.getStatusProcess().equals("Processamento inativo."))
            s2.processaDados();
        else {
            System.out.print(s2.getStatusProcess());
            System.out.println(" Aguarde o término do processamento.");
        }
    }
}
```

*Note que o valor do atributo **statusProcess** é o mesmo ("Processamento ativo"), para qualquer instância (**s1** ou **s2**) que chame o método **getStatusProcess()**. Isso ocorre porque ambas as instâncias referenciam o mesmo e único objeto **Singleton** existente na memória, o objeto estático **instance**.*

Resultado da execução do código

```
Processamento inativo.
Processamento ativo.
Processamento ativo. Aguarde o término do processamento.
```

i.bertholdo@ifsp.edu.br

Factory

- O padrão **Factory** especifica uma classe especializada em criar objetos para outro objeto cliente, sem que este precise saber as características ou a lógica de implementação do objeto a ser criado.
- Nesse padrão, o cliente simplesmente informa o tipo de objeto desejado ao **objeto fábrica** que, por sua vez, retorna uma referência para o **objeto concreto** solicitado.
- Esse padrão também permite adicionar subclasses ao sistema sem impactar na estrutura atual do código, exceto o da própria fábrica.
- Por exemplo, em uma aplicação para hotéis, podemos adicionar novas classes concretas, relacionadas a tipos de quartos, sem alterar o código que utiliza estas classes.

l.bertholdo@ifsp.edu.br

Factory

Exemplo: *Imagine um sistema para uma clínica veterinária que atende mamíferos e aves. Seja qual for o tipo de animal, ele terá* ***informações cadastrais*** *e um* ***histórico médico****. Os dados de um paciente mamífero e de um paciente ave têm suas respectivas particularidades. Por isso, haverá classes concretas específicas para cada tipo de paciente (****Mamifero*** *e* ***Ave****), as quais implementarão a interface* ***Paciente****.*

```java
package interfacePackage;

public interface Paciente {
    public String getDadosDoPaciente();
}
```

```java
package clinica;

import interfacePackage.Paciente;

public class Mamifero implements Paciente {
    private FichaCadastral fc = null;
    private HistoricoMedico hm = null;

    public Mamifero(String nome, String dataUltimaConsulta) {
        fc = new FichaCadastral(nome);
        hm = new HistoricoMedico(dataUltimaConsulta);
    }

    public String getDadosDoPaciente() {
        return "Paciente mamífero:\nNome: " + fc.getNome() +
               "\nData da última consulta: " + hm.getDataUltimaConsulta();
    }
}
```

```java
package clinica;

import interfacePackage.Paciente;

public class Ave implements Paciente {
    private FichaCadastral fc = null;
    private HistoricoMedico hm = null;

    public Ave(String nome, String dataUltimaConsulta) {
        fc = new FichaCadastral(nome);
        hm = new HistoricoMedico(dataUltimaConsulta);
    }

    public String getDadosDoPaciente() {
        return "Paciente ave:\nNome: " + fc.getNome() +
               "\nData da última consulta: " + hm.getDataUltimaConsulta();
    }
}
```

Embora essas classes de exemplo tenham códigos muito parecidos, em uma aplicação real elas teriam diversas ***particularidades****, de modo que haveria mais diferenças do que semelhanças.*

l.bertholdo@ifsp.edu.br

Factory

```java
package clinica;

public class FichaCadastral {
    String nome = "";

    public FichaCadastral(String nome) {
        this.nome = nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

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

```java
package clinica;

public class HistoricoMedico {
    String dataUltimaConsulta = "";

    public HistoricoMedico(String dataUltimaConsulta) {
        this.dataUltimaConsulta = dataUltimaConsulta;
    }

    public void setDataUltimaConsulta(String dataUltimaConsulta) {
        this.dataUltimaConsulta = dataUltimaConsulta;
    }

    public String getDataUltimaConsulta() {
        return dataUltimaConsulta;
    }
}
```

Factory

O sistema solicita um paciente à fábrica **PacienteFactory**, informando o tipo e os dados do paciente desejado, e esta retorna o paciente solicitado (**Mamifero** ou **Ave**).

```java
package clinica;

import interfacePackage.Paciente;
import factory.PacienteFactory;

public class FactoryMain {
    public static void main(String[] args) {
        PacienteFactory pf = new PacienteFactory();
        Paciente p = null;

        p = pf.getPaciente("mamífero", "Pitoca", "10/02/2022");
        System.out.println(p.getDadosDoPaciente());

        p = pf.getPaciente("ave", "Chico", "18/07/2021");
        System.out.println(p.getDadosDoPaciente());
    }
}
```

```java
package factory;

import clinica.Ave;
import clinica.Mamifero;
import interfacePackage.Paciente;

public class PacienteFactory {
    public Paciente getPaciente(String tipo, String nome, String dataUltimaConsulta) {
        Paciente p = null;
        if (tipo.equals("mamífero"))
            p = new Mamifero(nome, dataUltimaConsulta);
        else if (tipo.equals("ave"))
            p = new Ave(nome, dataUltimaConsulta);
        return p;
    }
}
```

Em seguida, chama um método genérico que mostra os dados do paciente, independentemente se ele é mamífero ou ave.

Resultado da execução do código

```
Paciente mamífero:
Nome: Pitoca
Data da última consulta: 10/02/2022
Paciente ave:
Nome: Chico
Data da última consulta: 18/07/2021
```

Pelo resultado, percebemos que a fábrica retorna objetos diferentes conforme o tipo solicitado a ela. Construindo o sistema dessa forma, torna-se simples adicionar quaisquer outros tipos de animal, bastando implementar novas classes concretas para esses outros tipos.

Referências

- ENGHOLM JÚNIOR, Hélio. **Análise e Design:** orientado a objetos. São Paulo: Novatec Editora, 2013.

- GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Padrões de projeto:** soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2007.

- PRESSMAN, Roger S.; MAXIM, Bruce R. **Engenharia de software:** uma abordagem profissional. 8. ed. Porto Alegre: AMGH, 2016.

- SOMMERVILLE, Ian. **Engenharia de software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.