

Aula 1: Introdução aos padrões de projeto

Apresentação

Nesta aula, demonstraremos o histórico do desenvolvimento de padrões de projeto de software, decodificando o que eles seriam e os principais problemas que resolvem.

Esses padrões evoluem ao longo do tempo quando incorporam as boas práticas de programadores e analistas experientes, que compartilham tal conhecimento para a comunidade dos desenvolvedores de sistemas. Arrolaremos ainda conceitos básicos de programação e análise orientada a objeto pré-requisito para auxiliar na compreensão do conteúdo desta aula.

Objetivos

- Identificar a importância dos padrões de projetos para o desenvolvimento de software de melhor qualidade;
- Descrever as características de um padrão de projeto de software;
- Relacionar os eventos históricos sobre a evolução do conceito de padrões de projeto.

Primeiras palavras

Em engenharia de software, um padrão de projeto (do inglês *design pattern*) é uma solução genérica que se aplica a um tipo de problema que ocorre frequentemente em uma situação específica no desenho (*design*) de software. Ele não é um conjunto de instruções que pode ser diretamente transformado em programa de computador. Trata-se, na verdade, de uma descrição ou de um modelo¹ (*template*) genérico sobre a forma de resolver um problema, podendo ser usado em muitos projetos diferentes.

Padrões de projeto

Padrões são as melhores práticas formalizadas que uma equipe de projeto de desenvolvimento de software pode usar para resolver problemas comuns baseados na reutilização de ideias (não de um código).



 (Fonte: Shutterstock)

Exemplo

Padrões de projeto orientados a objeto normalmente mostram relacionamentos e interações entre classes ou objetos sem especificá-los na aplicação final.

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

O conceito básico sobre padrões de projeto (*design patterns*) alicerça-se em:

Organização das ideias de desenvolvedores de software experientes;

Sistematização e disponibilização dessas ideias em um grande banco de ideias acessível para programadores menos experientes utilizarem.

Mas o que são padrões de software?



A descrição de um problema que ocorre com frequência.




A base de uma solução para ele com um nome.



Modelos baseados na reutilização de ideias (não de um código).

As características obrigatórias que devem ser atendidas por um padrão de projeto contêm basicamente quatro elementos. Leia o texto abaixo.

 Elementos

 Clique no botão acima.

Elementos

1. Nome do padrão

- Resume em uma ou duas palavras o problema, as soluções e as consequências do uso do padrão;
- Deve ser facilmente lembrado, indicando o conteúdo do padrão.

2. Problema a ser resolvido

- Descreve em qual situação ou em quais condições deve-se aplicar o padrão. Explica o problema, seu contexto, os sintomas e as condições.

3. Solução dada pelo padrão

Ela deve apresentar:

- Elementos que constituem seu design;
- Relacionamentos;
- Responsabilidades;
- Colaboradores.

Dica: A solução deve ser genérica para poder ser aplicada em muitas situações diferentes.

4. Consequências

- Analisa vantagens, desvantagens e compromissos decorrentes da aplicação do padrão.

Exemplo: Alguns aspectos que podem ser analisados: prazo de implantação, custo, reusabilidade, portabilidade e extensibilidade do código, além do desempenho do sistema. (**Reusabilidade:** Adapta uma classe existente em um sistema para outro a fim de executar um mesmo conjunto de ações. **Portabilidade:** É a capacidade de o software ser compilado ou executado em diferentes arquiteturas de hardware ou de software. **Extensibilidade:** É a capacidade que o sistema tem de crescer pela adição de novos componentes.)

Em termos de orientação a objetos, padrões de projeto identificam:

- Classes;
- Instâncias;
- Papéis;
- Colaborações;
- Distribuição de responsabilidades.

Trata-se, portanto, de descrições de classes e objetos que se comunicam. Eles são desenvolvidos para solucionar um problema comum em um contexto específico. (SORROCHE; LOPES, 2003)

Principais vantagens

Utilizar padrões em um projeto oferece as seguintes vantagens:

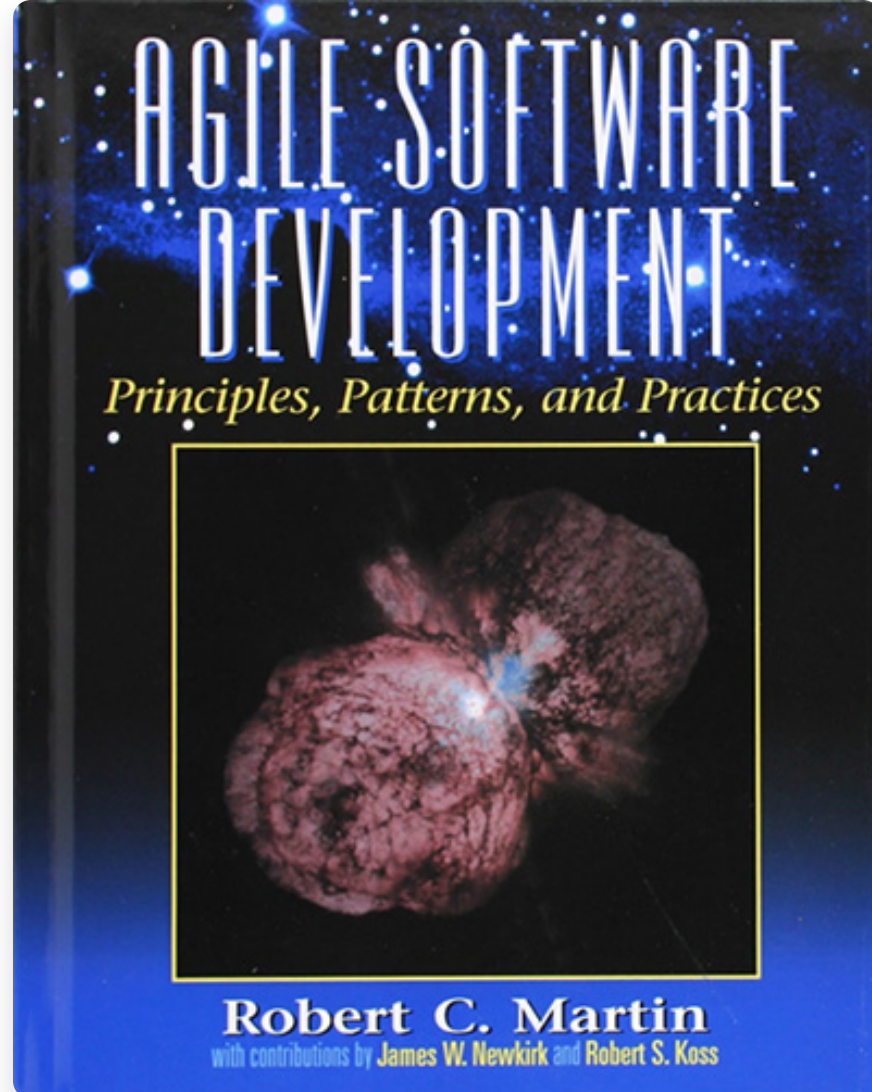
- **Foram testados:** Refletem a experiência e o conhecimento dos desenvolvedores que utilizaram tais padrões com sucesso em seu trabalho;
- **São reutilizáveis:** Fornecem uma solução pronta que pode ser adaptada para diferentes problemas quando necessário;
- **São expressivos:** Formam um vocabulário comum para expressar grandes soluções sucintamente;
- **Facilitam o aprendizado:** Reduzem o tempo de aprendizado de uma determinada biblioteca de classes. Isso é fundamental para o aprendizado dos desenvolvedores novatos;


- **Diminuem retrabalho:** Quanto mais cedo forem usados, menor será o retrabalho em etapas mais avançadas do projeto.

Fonte: (ALUR; CRUPI, MALKS, 2002)

Princípios de design de software

Tais princípios representam um conjunto de diretrizes que nos ajudam a evitar o desenvolvimento de um software com design ruim. Robert Martin os reuniu no livro *Agile software development: principles, patterns and practices*. Veremos a seguir suas principais características.



 Clique nos botões para ver as informações.

Sete características de um design ruim



Sete características de um design ruim

De acordo com Robert Martin (2002), elas devem ser evitadas:

- 1. Rigidez:** Há dificuldade em realizar mudanças no software porque cada mudança afeta muitas outras partes do sistema.
- 2. Fragilidade:** Alterações nos programas provocam falhas inesperadas em outras partes do sistema.
- 3. Imobilidade:** Reutilizar o software em outro sistema é difícil por ele não poder ser desvinculado do atual.
- 4. Viscosidade:** Existem duas formas de viscosidade: a do software e a do ambiente de desenvolvimento. No primeiro caso, ela ocorrerá quando as manutenções realizadas nele não preservarem o design, cuja viscosidade é alta. No segundo, ela será alta quando esse ambiente for lento e ineficiente.
- 5. Complexidade desnecessária:** O design ruim ocorrerá quando o desenvolvedor inserir elementos que atualmente não são úteis no software para tratar possíveis mudanças futuras no sistema. No começo, isso pode parecer uma boa ideia, embora, muitas vezes, seu efeito seja o oposto: o design fica cheio de instruções que nunca serão usadas. Alguns desses preparativos podem compensar, mas isso não ocorre na maioria das vezes, tornando o software complexo e difícil de entender.
- 6. Repetição desnecessária:** Deve-se evitar a duplicação de código-fonte em várias partes do sistema. Quando ela ocorrer, mesmo em formas ligeiramente diferentes, os desenvolvedores estarão perdendo a oportunidade de criar uma classe abstrata para representar esse comportamento e tornar o sistema mais fácil de ser compreendido e mantido.
- 7. Opacidade:** Representa a tendência de a compreensão de um código-fonte ser difícil para quem não o desenvolveu por ele não ter sido escrito de forma clara e expressiva.

Cinco princípios de design

Eles devem ser utilizados no desenvolvimento de software orientado a objetos:

1. Princípio da responsabilidade única: Este princípio foi introduzido por Tom DeMarco (1979) em seu livro *Structured analysis and systems specification*: “Uma classe deve ter apenas um motivo para mudar”. Se houver duas razões para mudá-la, é porque ela está mal definida, pois uma classe deve ter apenas uma função. Deveremos, se for o caso, dividir as duas funções em duas classes. Robert Martin (2002) reinterpretou o conceito e definiu a responsabilidade como uma razão para mudar.

2. Princípio do Aberto para expansão e Fechado para modificação: Entidades de software, como classes, módulos e funções, devem estar abertas para extensão, mas fechadas para modificações. Este princípio pretende garantir que, quando precisarmos alterar o comportamento de uma classe ou de um conjunto delas, não precisemos alterá-la, e sim estendê-la. Ele pode ser implementado pelos padrões *Template Method* e *Strategy*, que serão apresentados em futuras aulas deste curso.

3. Princípio de segregação de interface: Os clientes não devem ser forçados a depender de interfaces que não usem. Este princípio alerta sobre os cuidados que devemos ter ao escrevermos nossas interfaces. Devemos adicionar apenas métodos que deveriam estar lá. Se incluirmos os que não deveriam, as classes que a implementarem terão de implementar esses métodos também. Exemplo: se criarmos uma interface chamada **Cliente** e adicionarmos um método de **Cônjuge**, todos os clientes terão de implementá-lo. E se um cliente for solteiro?

4. Princípio da inversão da dependência: Módulos de alto nível não devem depender daqueles de baixo nível. Ambos, na verdade, devem precisar de abstrações, que, por sua vez, não devem necessitar de detalhes. Estes, por outro lado, devem depender de abstrações. Exemplo: O padrão *Abstract Factory* – que será apresentado na próxima aula – implementa este princípio.

5. Princípio da substituição de Liskov² : Tipos derivados devem ser completamente substituíveis por seus tipos base.

As novas classes derivadas devem ser capazes de substituir as classes base sem qualquer alteração no código.

Evolução do conceito de padrões de software

Os conceitos sobre padrões de projeto têm evoluído desde 1977, quando o arquiteto Christopher Alexander estabeleceu as características que um padrão deve ter e seu formato de descrição no contexto da arquitetura de prédios e cidades:

Cada padrão descreve um problema que ocorre repetidamente de novo e de novo em nosso ambiente, e então descreve a parte central da solução para aquele problema de uma forma que você pode usar esta solução um milhão de vezes sem nunca a implementar duas vezes da mesma forma.

- ALEXANDER, 1977

1987

O primeiro artigo científico sobre padrões de projeto para as suas interfaces é publicado, adaptando as ideias de Alexander para a engenharia de software



1994

Surge o padrão GoF (em inglês, gang of four; em português, gangue dos quatro) no primeiro livro sobre o assunto: Design patterns: elements of reusable object-oriented software. A obra dos autores Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides defende a ideia de que os padrões de projeto simplificam a reutilização de projetos e arquiteturas bem-sucedidas;



2005

Vem à tona o padrão GRASP³ (em inglês, general responsibility assignment software patterns or principles), cuja visão sobre os padrões de projeto se expande com a ideia de que eles não devem apenas expressar novas ideias de projeto, mas também codificar conhecimento, idiomas e princípios existentes que foram testados e sejam verdadeiros.



Pensadores

Christopher Alexander (1977)

Evolução das ideias sobre padrões de projeto

A ideia de padrões para arquitetura de prédios surge em 1977 com a publicação do livro *Pattern language: towns, buildings, construction*.

Ward Cunnhingham e Kent Beck (1987)	Em 1987, é publicado o primeiro artigo sobre padrões para projeto de interfaces software na conferência OOPSLA (<i>Object-oriented programming, systems, languages & applications</i>).
Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (1994)	Em 1994, é lançado o livro <i>Design patterns: elements of reusable object-oriented software</i> (em português, padrões de projeto: soluções reutilizáveis de software orientado a objetos).
Craig Larman (2005)	Em 2005, é publicado o livro <i>Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development</i> contendo o padrão GRASP.

 Quadro 1: Histórico dos principais pensadores sobre padrões de projetos.

Atualmente, padrões de projeto vêm sendo propostos para projetos de:



Desenvolvimento de jogos de computador (gamefied software).



Software para computação em nuvem (cloud software).




Aplicações de big data.



Inteligência artificial.



 (Fonte: Shutterstock)

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

Principais problemas tratados pelos padrões de software

A utilização desses padrões em projetos de desenvolvimento de software aumenta a qualidade dos documentos produzidos, facilita a manutenção e reutilização dos programas e torna mais fácil entender os produtos que o projeto estiver produzindo. Já a falta de padronização deixa os desenvolvedores livres para utilizar o próprio método de codificação, tornando mais difícil a integração de módulos do sistema, além de comprometer aspectos relacionados com a qualidade e a confiabilidade do sistema.

Os padrões de projeto são codificados no formato de um par problema/solução, que pode ser aplicado em novos contextos acompanhados de conselhos sobre como tais padrões devem ser aplicados. Em geral, eles têm nomes sugestivos, o que, de certa forma, ajuda a fixar na nossa memória os seus conceitos.

Padrões que tratavam do relacionamento entre as classes e os objetos de um sistema orientado a objeto tiveram de resolver inicialmente os seguintes problemas:



- Como tornar um sistema orientado a objeto independente em relação à forma como seus objetos são criados, compostos e representados?
- Como flexibilizar a composição de objetos dinamicamente, durante a execução do programa, para formar estruturas maiores?
- Como melhorar a comunicação entre classes e objetos, evitando que um objeto fique sobrecarregado com muitos métodos e distribuindo a responsabilidade da execução de tarefas para melhorar a cooperação entre os objetos?

Perguntas

1. Padrões classificados por propósito

Os padrões GoF surgiram para tentar resolver tais problemas. Estes padrões foram classificados em três grupos:

- **Criação:** Tem cinco padrões que abstraem o processo de criação de classes e objetos, deixando o sistema independente em relação à maneira como seus objetos são criados, compostos e representados;
- **Estrutura:** Composto por sete padrões que tratam a forma como as classes e os objetos podem ser compostos para formar estruturas maiores;
- **Comportamento:** Possui 11 padrões que tratam da atribuição de responsabilidades entre objetos por meio de padrões de comunicação entre si.

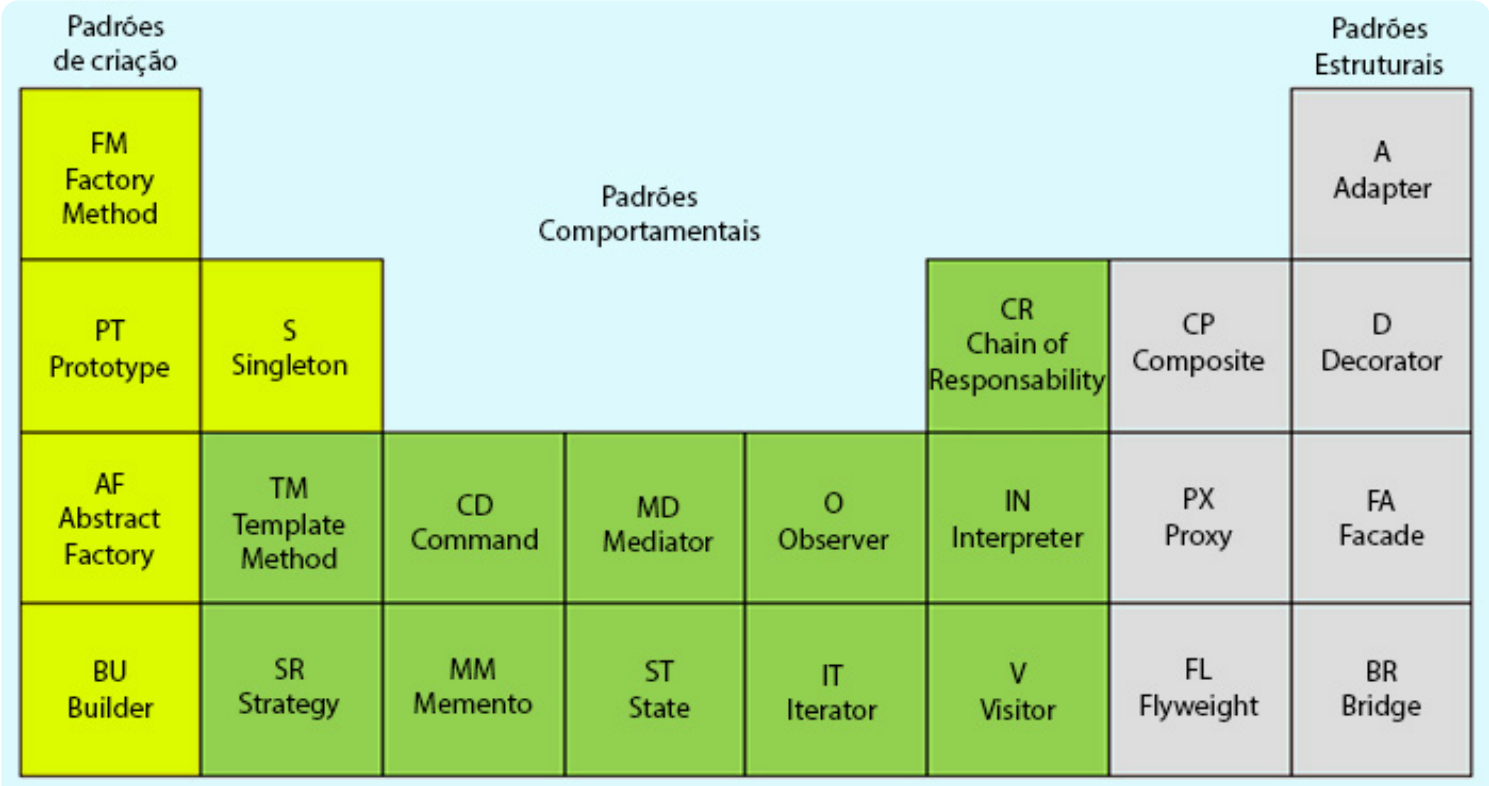


Figura: Padrões GoF classificados por propósito.

2. Padrões classificados por escopo

Os padrões de projeto de software podem também ser classificados por escopo ao se referirem a classes ou objetos.

- **Classe:** Um padrão de criação dela negocia relacionamentos entre classes e subclasses usando a herança para variar a classe a ser instanciada. Essas relações são estáticas porque são feitas em tempo de compilação do programa: utilizam a hierarquia para compor ou variar os objetos, mantendo a flexibilidade do sistema;
- **Objeto:** Um padrão de criação dele delegará a instanciação para outro objeto. Esse relacionamento entre objetos é dinâmico, pois eles podem se alterar em tempo de execução do programa.

3. Padrões classificados por intenção

Classificação proposta por Steven John Metsker. Para ele, os padrões de projeto também foram agrupados em cinco grupos segundo a sua intenção (o problema a ser solucionado):

- **Interface:** Oferecer uma interface;
- **Responsabilidade:** Atribuir uma responsabilidade;
- **Construção:** Realizar a construção de classes ou objetos;
- **Operação:** Controlar as formas de operação;
- **Extensão:** Implementar uma extensão para a aplicação.

Intenção	Padrões
1. Interface	Adapter, Facade, Composite, Bridge
2. Responsabilidade	Singleton, Observer, Mediator, Proxy, Chain of Responsibility, Flyweight.
3. Construção	Builder, Factory Method, Abstract Factory, Prototype, Memento.
4. Operações	Template Method, State, Strategy, Command Interpreter.
5. Extensões	Decorator, Iterator, Visitor.

 Quadro 2: Classificação de padrões por intenção.

Surgiram posteriormente novos desafios que poderiam ser tratados por padrões de projeto. Abordaremos alguns deles nas próximas aulas.

GRASP

É o conjunto de práticas para atribuição de responsabilidades a classes e objetos em projetos de desenvolvimento de software que utilizem a análise e a programação orientada a objeto. Neste tipo de projeto, os casos de uso e diagrama de classes, nas fases de análise e projeto do sistema, são criados para distribuir tarefas ou responsabilidades entre múltiplos objetos participantes do sistema. No entanto, surgem os seguintes problemas:




- Como melhorar o desempenho do código?
- Como garantir melhor interface entre as classes e os objetos do projeto?
- Como evitar a sobrecarga de objetos no processo de atribuição de responsabilidade para os objetos?

Exemplo

Pensem na construção de uma casa. Utilizando os princípios da programação estruturada, identificaríamos as tarefas necessárias para tal e as executaríamos sequencialmente. Utilizando os princípios da programação estruturada, deve-se contratar uma equipe de objetos e atribuir-lhes responsabilidades. Dessa forma, a equipe constitui-se dos seguintes profissionais: pedreiro (cuja responsabilidade é assentar os tijolos); azulejista (cuida dos azulejos); encanador (parte hidráulica); eletricitista (parte elétrica); marceneiro (a montagem do telhado) etc. Quando ela estiver formada, será criada, portanto, uma dinâmica de trabalho em que cada trabalhador sabe quais são as suas responsabilidades, passando a interagir entre si para que a casa seja montada.

Um programa orientado a objetos funciona da mesma maneira: criamos um conjunto de objetos, atribuímos-lhes responsabilidades e, depois, os colocamos para interagir.



 Clique no botão acima.

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

GRASP

1. Tipos de responsabilidade

a) Responsabilidade de saber

A única tarefa dos objetos é informar, quando forem solicitados, aquilo pelo qual são responsáveis. Eles sabem:

- Guardar o conhecimento consigo;
- Quem guarda o conhecimento;
- Como derivar um conhecimento de outros conhecimentos.

b) Responsabilidade de fazer

Os objetos têm de executar alguma tarefa. Eles terão de:

- Fazer alguma coisa sozinho;
- Criar objetos e delegar-lhes responsabilidades;
- Controlar e coordenar tarefas em objetos existentes.

2. Padrões

A maneira como atribuímos responsabilidades entre objetos pode fazer toda a diferença entre um design bem ou mal feito. Este é o escopo dos padrões GRASP, que são divididos em dois grupos: básicos e avançados.

a) Básicos

- Information expert (especialista na informação);
- Creator (criador);
- High cohesion (alta coesão);
- Low coupling (baixo acoplamento);
- Controller (controlador).

b) Avançados

- Polymorphism (polimorfismo);
- Pure fabrication (invenção pura);
- Indirection (indireção);
- Protected variations (variações protegidas).

Palavras finais

Há um princípio que se aplica a quase todos os padrões de projeto de software:

Identifique os aspectos do seu aplicativo que variam e separe-os daquele que permanecer igual.

Deve-se, portanto, encapsular as partes do seu sistema que variam para ser possível alterá-las sem afetar as outras que não o fazem.

Atividade

1 - Os padrões de projeto de software contribuem para a melhoria da qualidade do software porque:

- a) Apresentam um modelo genérico de como resolver problemas de modelagem do software, aumentando a qualidade dos documentos de software produzidos.
 - b) Fornecem exemplos de códigos de programação que podem ser reutilizados.
 - c) Fornecem padrões para relacionamento entre os atributos do sistema.
 - d) Servem como modelo para testar os softwares desenvolvidos.
 - e) Apresentam as boas práticas de programação estruturada que podem ser reutilizadas na programação orientada a objetos.
-

2 - São características de padrões de projetos de software todas as sentenças a seguir, exceto:

- a) Nome do padrão
 - b) Autor do padrão
 - c) Problema a ser resolvido
 - d) Solução dada pelo padrão
 - e) Consequências
-

3 - São vantagens da utilização de padrões de projetos de software as afirmativas a seguir, exceto:

- a) Os padrões já foram testados e trazem a experiência e o conhecimento de desenvolvedores experientes.
 - b) Os padrões podem ser adaptados para diferentes problemas.
 - c) Usar padrões pode aumentar o tempo de aprendizado das classes do sistema pelos desenvolvedores novatos.
 - d) Padrões podem diminuir o retrabalho durante as etapas do projeto de desenvolvimento de software.
 - e) Padrões contêm um conjunto de expressões padronizadas para expressar soluções complexas de forma resumida.
-

4 - Quando surgiram as primeiras ideias sobre o uso de padrões de projeto?

- a) Em 1977, com a publicação do livro *A pattern language*, por Christopher Alexander.
 - b) Em 1987, com o primeiro artigo sobre padrões para projeto de interfaces de software.
 - c) Em 1994, com a publicação do Livro *Design patterns* pela gangue dos quatro.
 - d) Em 2005, com a publicação do livro *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*, que contém o padrão GRASP.
 - e) Em 1947, com a publicação do livro *Design patterns*, por Craig Larman.
-

5 - Identifique qual opção apresenta problemas que podem ser tratados pelos padrões de projeto de software.

- a) Problema de comunicação entre as classes e os objetos do sistema.
- b) Problema de criação de objetos e classes dinamicamente em tempo de execução do programa.
- c) Problema de independência do sistema para criar e representar classes e objetos.
- d) Problema de velocidade de execução do sistema.
- e) Problema de definição de tipos de dados (atributos) a serem utilizados no sistema.

Notas

Modelo¹

Um modelo é uma simplificação da realidade que abstrai certas características importantes para simplificar o entendimento do problema.

Princípio da substituição de Liskov²

Este princípio é apenas uma extensão do princípio do Aberto para expansão e Fechado para modificação em termos de comportamento. Isso significa que devemos garantir que novas classes derivadas estendam as classes base sem alterar seu comportamento.

GRASP¹
Referências

Falaremos deste padrão de maneira mais detalhada no final desta aula.

ALEXANDER, C. A **Pattern language**: towns, buildings, construction. New York: Oxford University Press, 1977.

ALUR, D.; CRUPI, J.; MALKS, D. **Core j2ee patterns**: as melhores práticas e estratégias de design. Rio de Janeiro: Campus, 2002.

DEMARCO, T. **Structured analysis and systems specification**. New Jersey: Prentice-Hall, 1979.

FRIEMAN, E. **Use a cabeça!** Padrões de projeto. 2. ed. Rio de Janeiro: Elsevier, 2007.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns**: elements of reusable object-oriented software. New York: Addison-Wesley Professional, 1994.

GUDWIN, R. R. **Componentes, frameworks e design patterns**. 2010.

LARMAN, C. **Applying UML and patterns**: an introduction to object-oriented analysis and design and iterative development. 2. ed. New Jersey: Prentice-Hall, 2001.

LEÃO, L. **Padrões de projeto de software**. Rio de Janeiro: SESES, 2018.

MARTIN, R. C. **Agile software development**: principles, patterns and practices. New Jersey: Prentice-Hall, 2002.

SORROCHE, R.; LOPES, M. C. **Uso de design patterns e J2EE**: um estudo de caso. Blumenau/SC: Universidade de Blumenau, 2003.

Próxima aula

- Apresentação dos padrões Abstract Factory, Builder e Factory Method.

Explore mais

Veja este vídeo: [Design patterns](#).

Leia este texto: [Aprendizagem de design patterns utilizando mapas conceituais](#).