



Introdução aos Design Patterns

Fernando J. Barros

Dept.º Eng.º Informática, FCTUC

1

Design Patterns

- O design de software orientado a objectos é uma actividade complexa.
- O design de software reutilizável pode ser ainda mais complexo
- É necessário descobrir os objectos pertinentes, organizá-los em classes com a granularidade adequada, definir o interface dessas classes e a sua organização hierárquica, e estabelecer as relações entre as várias classes.
- O design deve ser específico para o problema que se pretende resolver, mas ao mesmo tempo suficientemente genérico para que, no futuro, possa ser utilizado em problemas semelhantes.

2

Design Patterns

- O objectivo principal é evitar tornar a fazer o mesmo projecto ou pelo menos minimizar o esforço na realização de um novo projecto.
- A experiência tem mostrado que um design flexível e reutilizável é difícil ou mesmo impossível de conseguir à primeira tentativa.
- Antes de um design ser terminado os programadores experientes tentam reutilizá-lo por diversas vezes, modificando-o de cada vez que o usam.

3

Design Patterns

- Programadores experientes obtêm designs de qualidade.
- Programadores experientes não resolvem os problemas a partir do zero. Pelo contrário, reutilizam soluções que usaram no passado. Quando encontram uma boa solução usam-na repetidas vezes. É esta experiência que faz deles peritos.
- É possível encontrar padrões de classes e de comunicação entre objectos em muitos sistemas orientados a objectos.

4

Design Patterns

- Estes padrões resolvem problemas de design específicos e tornam os designs orientados a objectos mais flexíveis, elegantes e em última análise reutilizáveis.
- Um designer familiar com estes padrões pode aplica-los de imediato em problemas de design sem ter que os redescobrir.
- Novelistas e dramaturgos raramente criam as suas obras a partir do zero. Ao contrário, seguem padrões tais como “A História de Aventuras”, ou “O Romance Policial”.

5

Design Patterns

- Designers de software orientado a objectos seguem padrões tais como “representar estados com objectos”, “decorar objectos para que certos aspectos possam ser facilmente alterados”.
- Quantas vezes tivemos a impressão de termos já resolvido um problema mas não nos recordarmos como o resolvemos?
- Design Patterns têm por objectivo catalogar a experiência acumulada no design de software orientado a objectos.

6

O que é um Design Pattern?

- Christopher Alexander (Arquitecto): “Cada pattern descreve um problema que ocorre repetidas vezes, e por isso descreve o núcleo de uma solução para esse problema, de forma que essa solução possa ser utilizada repetidas vezes”.

7

O que é um Design Pattern?

- Em geral um Design Pattern tem 4 elementos principais:
 - O nome é uma forma de identificar o problema de design.
 - O problema descreve quando aplicar o design pattern.
 - A solução descreve os elementos que constituem o design, o seu relacionamento, responsabilidade e colaboração.
 - As consequências são os resultados e as desvantagens da aplicação do design pattern.

8

O que é um Design Pattern?

- Design patterns são descrições de objectos e classes interrelacionados que são adaptados para resolver um problema de design num contexto particular.

9

O que não é um Design Pattern?

- Design Patterns não têm a ver com o design de listas ou tabelas de hash.
- Também não são designs de aplicações ou subsistemas completos.

10



MVC Design Pattern

- A tríade de classes MVC (Model/View/Controller) é usada para construir interfaces (Smalltalk).
- MVC consiste em três tipos de objectos. O Modelo é o objecto aplicação, a View é representação gráfica, o Controller define a forma como o interface reage à entrada de dados.
- Antes da criação da pattern MVC a era habitual agrupar-se a funcionalidade destes três objectos num único.

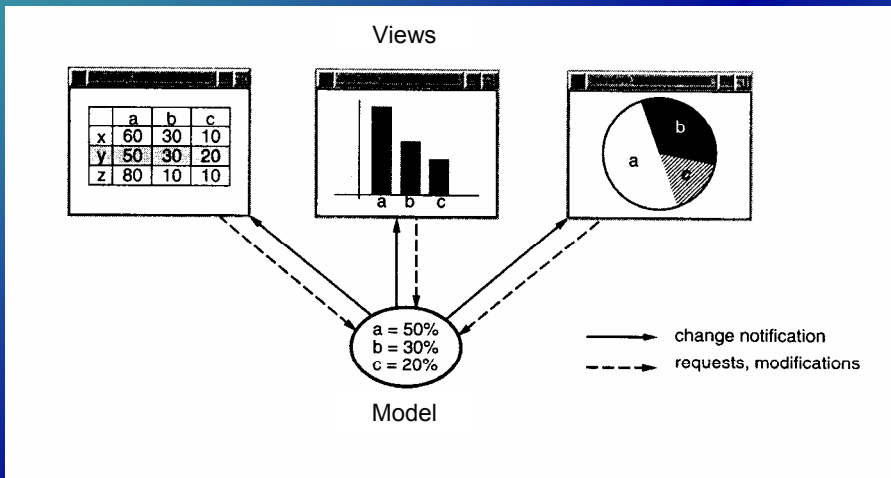
11

MVC Design Pattern

- MVC faz a separação entre views e modelos estabelecendo um protocolo subscrição/notificação. Uma view deve assegurar que a sua aparência reflecte o estado do modelo. Quando o estado do modelo se altera todas as views que estão dependentes são notificadas.
- Esta abordagem permite que se associe um número variável de views a um modelo para se conseguir diferentes representações gráficas. É também possível associar novas views a um modelo sem o alterar.

12

MVC Design Pattern



13

Descrição de Design Patterns

- Nome e classificação
- Intenção: o que faz o design pattern
- Motivação: cenário que ilustra o problema de design
- Aplicabilidade
- Estrutura
- Participantes
- Colaborações
- Consequências

14

Catálogo de Design Patterns

- Abstract Factory
 - Interface para criar famílias de objectos relacionados sem a especificação da sua classe concreta.
- Adapter
 - Converte o interface de uma classe num outro interface que o cliente espera. Adapter permite a comunicação de classes com interfaces incompatíveis.

15

Catálogo de Design Patterns

- Bridge
 - Separa uma abstracção da sua implementação de forma a que ambas possam variar de forma independente.
- Builder
 - Separa a construção de um objecto complexo da sua representação de forma a que o mesmo processo de construção possa ser utilizado para criar objectos diferentes.

16

Catálogo de Design Patterns

- Chain of Responsibility
 - Evita o acoplamento entre o objecto que envia uma mensagem ao objecto que a recebe, fazendo com que vários objectos tenham a oportunidade de responder à mensagem. Estabelece uma cadeia de objectos e faz passar a mensagem ao longo desta cadeia até que um dos objectos seja capaz de responder à mensagem.
- Command
 - Encapsula uma mensagem sob a forma de objecto, permitindo a parametrização de clientes com diferentes mensagens.

17

Catálogo de Design Patterns

- Composite
 - Agrupa objectos em estruturas para representar hierarquias parte/todo. A pattern composite permite tratar de uma forma uniforme objectos simples e compostos.
- Decorator
 - Permite adicionar de uma forma dinâmica responsabilidades a um objecto.
- Façade
 - Fornece um interface uniforme a um conjunto de interfaces num subsistema. Esta pattern define um interface de alto nível que permite que os subsistemas sejam mais simples de utilizar.

18

Catálogo de Design Patterns

- Factory Method
 - Define um interface para a criação de um objecto, mas deixa às subclasses a tarefa de definir que classes instanciar.
- Flyweight
 - Permite utilizar de uma forma eficiente um grande número de objectos de pequenas dimensões.
- Interpreter
 - Define uma representação para a gramática de uma linguagem e um interprete que é capaz de reconhecer instruções nessa linguagem.

19

Catálogo de Design Patterns

- Iterator
 - Permite o acesso aos elementos de uma agregação de um forma sequencial sem expôr a representação.
- Mediator
 - Define um objecto que encapsula a forma como diverso objectos interagem. Esta SDP promove o fraco acopolamento entre evitando que estes se referenciem explicitamente.
- Memento
 - Sem violar o encapsulamento, captura e torna visível o estado interno de um objecto.

20

Catálogo de Design Patterns

- Observer
 - Define uma dependência de 1 para vários entre objectos, de forma a que quando um objecto é modificado todos os objectos dependentes são notificados (MVC).
- Prototype
 - Permite criar objecto através da cópia de objectos protótipo.
- Proxy
 - Fornece uma forma de controlar o acesso a outro objecto.

21

Catálogo de Design Patterns

- Singleton
 - Assegura que uma classe tem apenas uma instância.
- State
 - Permite que um objecto modifique o seu comportamento quando o seu estado muda. Em aparência o objecto parece mudar de classe.
- Template Method
 - Define a estrutura de um algoritmo, deixando para as subclasses a definição de alguns passos.

22

Catálogo de Design Patterns

- Visitor
 - Representa uma operação que pode ser executada sobre os elementos de uma estrutura. Esta SDP permite a definição de novas operações sem que seja necessário modificar as classes dos elementos.

23

Organização do Catálogo de DPs

- Objectivo
 - Criacional: Criação de objectos
 - Estrutural: Composição de objectos ou classes
 - Comportamental: Interação e divisão de responsabilidades entre objectos e classes
- Âmbito
 - Classe
 - Objecto

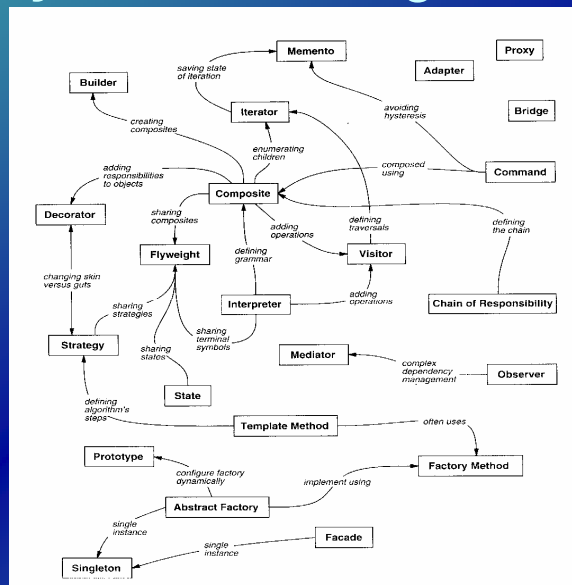
24

Organização do Catálogo de DPs

		Objetivo		
		Criacional	Estrutural	Comportamental
Âmbito	Classe	Factory Method	Adapter(Cl.)	Interpreter Template Method
	Objecto	Abstract Factory Builder Prototype Singleton	Adapter(Obj.) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

25

Relação entre Design Patterns

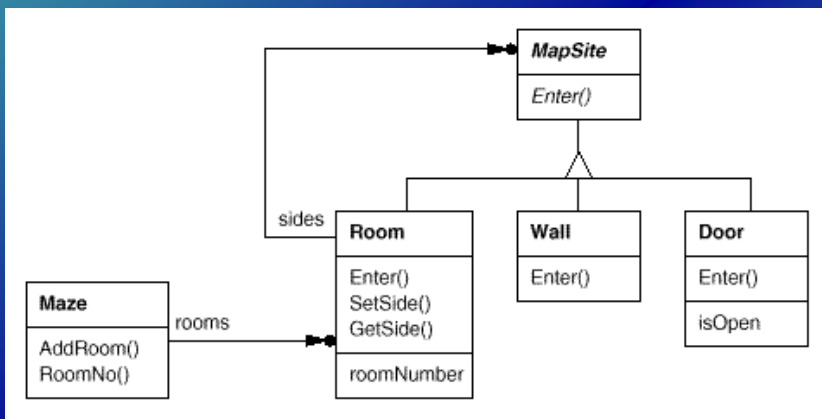


26

Aplicação

27

Labirinto



28

Labirinto

```
public enum Direction {
    INVALID = -1,
    NORTH,
    EAST,
    SOUTH,
    WEST
}

public interface MapSite {
    void Enter();
}
```

29

Labirinto

```
public class Room: MapSite {
    protected MapSite[] sides = new MapSite[4];
    protected int roomNumber;
    public void Enter() {...}
    public int Number { get { return roomNumber; } }
    public Room(int roomNumber) {
        this.roomNumber = roomNumber;
    }
    public MapSite GetSide(Direction direction) {
        return sides[(int)direction];
    }
    public void SetSide(Direction direction, MapSite side) {
        if (direction == Direction.INVALID) return;
        sides[(int)direction] = side;
    }
}
```

30

Labirinto

```
public class Door: MapSite {
    protected Room room1;
    protected Room room2;
    protected bool isOpen;
    public Door(Room r1, Room r2) {
        room1 = r1;
        room2 = r2;
    }
    public void Enter() {}
    public Room OtherSideFrom(Room room) {
        if(room == room2) return room1;
        if(room == room1) return room2;
        return null;
    }
}
```

31

Labirinto

```
public class Wall: MapSite {
    public Wall() {}
    public void Enter() {...}
}

public class Maze {
    protected Room[] rooms;
    public Maze(int n) {
        rooms = new Room[n];
    }
    public void AddRoom(Room room) {
        rooms[room.Number] = room;
    }
    public Room RoomNo(int n) {
        return rooms[n];
    }
}
```

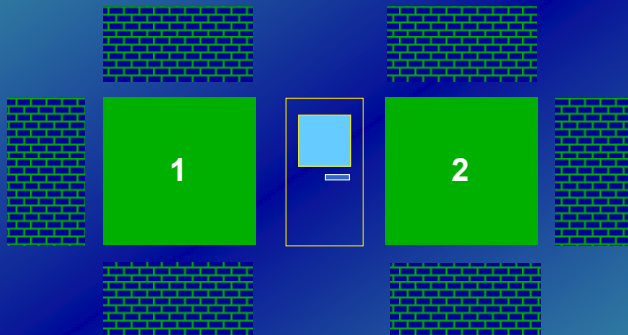
32

Labirinto

```
class BFMazeGame {
    public Maze CreateMaze() {
        Maze maze = new Maze();
        Room r1 = new Room(1);
        Room r2 = new Room(2);
        Door door = new Door(r1,r2);
        maze.AddRoom(r1);
        maze.AddRoom(r2);
        r1.SetSide(Direction.NORTH, new Wall());
        r1.SetSide(Direction.EAST, door);
        r1.SetSide(Direction.SOUTH, new Wall());
        r1.SetSide(Direction.WEST, new Wall());
        r2.SetSide(Direction.NORTH, new Wall());
        r2.SetSide(Direction.EAST, new Wall());
        r2.SetSide(Direction.SOUTH, new Wall());
        r2.SetSide(Direction.WEST, door);
        return maze;
    }
}
```

33

Labirinto



34

Labirinto

- Como criar novos labirintos ?

```
Room CreateRoom(string kind) {  
    if(kind == "Standard")  
        return new StandardRoom();  
    if(kind == "Enchanted")  
        return new EnchantedRoom();  
    ...  
}
```

35

Builder

36

Builder

- Objectivo
 - Separar a construção de um objecto complexo da sua representação de forma a que o mesmo processo de construção permita criar diferentes representações.

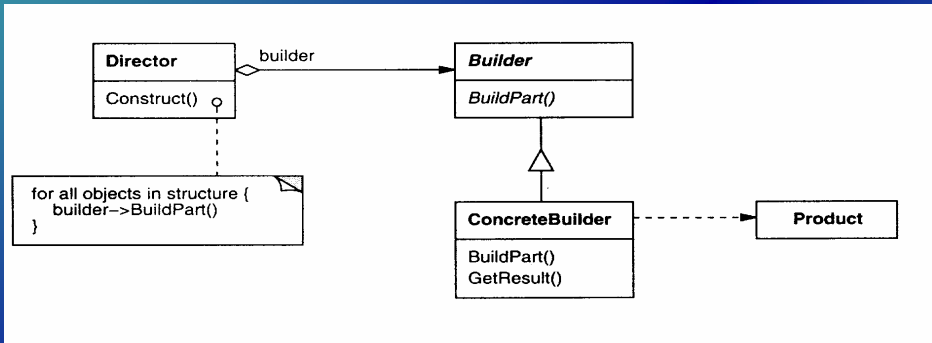
37

Builder

- Aplicabilidade
 - O algoritmo que cria um objecto complexo deverá ser independente dos componentes que compõe o objecto e da forma de interligação.
 - O processo de construção deverá permitir diferentes representações para o objecto que é construído.

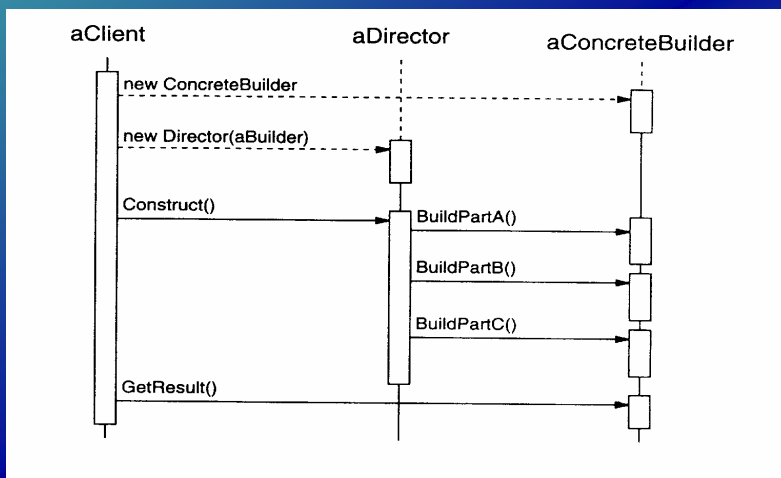
38

Builder



39

Builder



40

Builder

```
abstract public class MazeBuilder {
    protected int sizeX, sizeY;
    public bool validRoomNumber(int n) { return n > 0 && n < sizeX * sizeY; }
    public virtual void buildMaze(int x, int y) { sizeX = x; sizeY = y; }
    abstract public void buildRoom(int room);
    abstract public void buildDoor(int roomFrom, int roomTo);
}
```

41

Builder

```
public class MazeGameDirector {
    public void constructMaze(MazeBuilder builder) {
        builder.buildMaze(5,5);
        builder.buildRoom(1);
        builder.buildRoom(2);
        builder.buildDoor(1,2);
    }
    public void constructComplexMaze(MazeBuilder builder) {
        builder.buildMaze(10,10);
        builder.buildRoom(1);
        ...
        builder.buildRoom(100);
        ...
    }
}
```

42

Builder

```
public class StandardMazeBuilder: MazeBuilder {
    protected Maze currentMaze;
    protected Direction commonWall(Room r1, Room r2) {...}
    public StandardMazeBuilder() { currentMaze = null; }
    override public void buildMaze(int x, int y) {
        base.buildMaze(x,y);
        currentMaze = new Maze(x * y);
    }
}
```

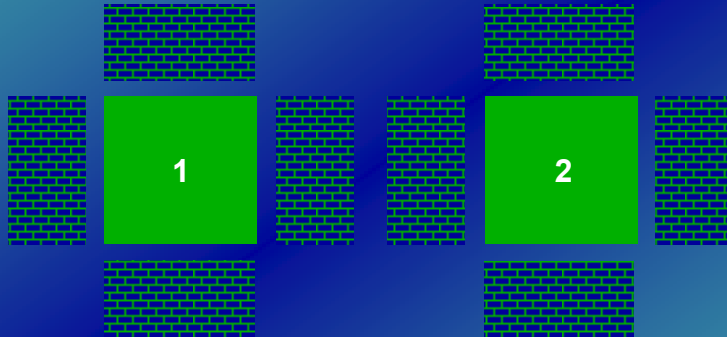
43

Builder

```
public class StandardMazeBuilder: MazeBuilder {
    override public void buildRoom(int n) {
        if (!validRoomNumber(n)) return;
        if (currentMaze.RoomNo(n) != null) return;
        Room room = new Room(n);
        room.SetSide(Direction.NORTH, new Wall());
        room.SetSide(Direction.SOUTH, new Wall());
        room.SetSide(Direction.EAST, new Wall());
        room.SetSide(Direction.WEST, new Wall());
        currentMaze.AddRoom(room);
    }
}
```

44

Labirinto



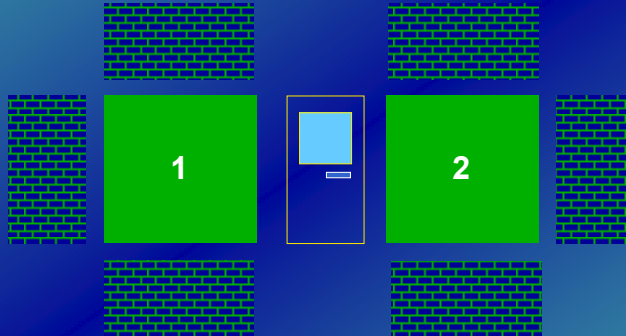
45

Builder

```
public class StandardMazeBuilder: MazeBuilder {  
    override public void buildDoor(int n1, int n2) {  
        Room r1 = currentMaze.RoomNo(n1);  
        Room r2 = currentMaze.RoomNo(n2);  
        Door d = new Door(r1, r2);  
        r1.SetSide(commonWall(r1, r2), d);  
        r2.SetSide(commonWall(r2, r1), d);  
    }  
    override public Maze getMaze() { return currentMaze; }  
}
```

46

Labirinto



47

Builder

```
public static void Main(string[] args) {  
    StandardMazeBuilder builder = new StandardMazeBuilder();  
    MazeGameDirector game = new MazeGame();  
    game.constructMaze(builder);  
    Maze maze = builder.getMaze();  
}
```

48

Builder

```
public class CountingMazeBuilder: MazeBuilder {
    protected int doors, rooms;
    public CountingMazeBuilder() { doors = rooms = 0; }
    override public void buildRoom(int n) { rooms++; }
    override public void buildDoor(int n1, int n2) {
        if(validRoomNumber(n1) && validRoomNumber(n2))
            doors++;
    }
    public void getCounts(out int r, out int d) {
        r = rooms;
        d = doors;
    }
}
```

49

Builder

```
public static void Main(string[] args) {
    CountingMazeBuilder builder = new CountingMazeBuilder();
    MazeGameDirector game = new MazeGame();
    game.constructMaze(builder);
    int x, y;
    builder.getCounts(out x, out y);
    Console.WriteLine("x = {0} y= {1}", x, y);
    Maze maze = builder.getMaze();
}
```

50

Builder

- O cliente não necessita de conhecer os detalhes internos dos produtos.
- As operações do Builder facilitam a criação de novos produtos.
- A extensão do software com a introdução de novos Builders é simplificada.

51

Design Patterns na Lic.^a Eng.^a Informática

- *Reutilização de Software em Simulação (5º Ano).*
 - Utilização de Design Patterns no desenvolvimento de simuladores (C++ → C#).

52

Bibliografia

- Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.
- Sherman Alpert, Kyle Brown, Bobby Woolf, *The Design Patterns Smalltalk Companion*, Addison Wesley, 1998.

53

Conclusões

- As aplicações podem tornar-se mais fáceis de documentar e manter devido à standardização das soluções usadas.
- Cada design pattern facilita a adaptação, em circunstâncias específicas, das aplicações a novas situações.
- Design Patterns ⇒ Programas mais fáceis de manter e estender.

54



Questões/Comentários