



Pós-Graduação em Computação Distribuída e Ubíqua

INF612 - Aspectos Avançados em Engenharia de Software
Padrões de Projeto e Idiomas de Programação

[Holub on Patterns], [Head First Design Patterns]
[Design Patterns Explained]

Sandro S. Andrade
sandroandrade@ifba.edu.br

Contexto e Motivação



- Uma rápida história:
 - Você é um professor com a responsabilidade de informar aos estudantes, ao final da sua aula, o local onde ocorrerá a aula seguinte
- Uma primeira implementação:
 - 1) Obtenha a lista de estudantes da classe
 - 2) Para cada estudante, faça o seguinte:
 - 1) Encontre a próxima aula que ele assistirá
 - 2) Localize a sala onde ocorrerá esta aula
 - 3) Encontre um caminho da sua sala para a sala localizada
 - 4) Informe ao estudante o caminho a ser seguido

Contexto e Motivação



- Uma rápida história:
 - Você é um professor com a responsabilidade de informar aos estudantes, ao final da sua aula, o local onde ocorrerá a aula seguinte
- Uma primeira implementação:
 - 1) Obtenha a lista de estudantes da classe
 - 2) Para cada estudante, faça o seguinte:
 - 1) Encontre a próxima aula que ele assistirá
 - 2) Localize a sala onde ocorrerá esta aula
 - 3) Encontre um caminho da sua sala para a sala localizada
 - 4) Informe ao estudante o caminho a ser seguido

Programa de Controle

Contexto e Motivação



- Uma rápida história:
 - Você é um professor com a responsabilidade de informar aos estudantes, ao final da sua aula, o local onde ocorrerá a aula seguinte
- Uma outra implementação:
 - 1) Publique no mural os caminhos da sua sala para todas as outras
 - 2) Informe para toda a turma: “O mural contém os caminhos para as outras salas. Utilizem-o e dirijam-se para as suas próximas aulas”

Contexto e Motivação



- O que mudou entre as soluções ?
 - Na primeira, você presta atenção a uma série de detalhes e é responsável por tudo
 - Na segunda, instruções básicas são fornecidas e espera-se que cada pessoa saiba fazer o seu trabalho

Contexto e Motivação



- O que mudou entre as soluções ?
 - Na primeira, você presta atenção a uma série de detalhes e é responsável por tudo
 - Na segunda, instruções básicas são fornecidas e espera-se que cada pessoa saiba fazer o seu trabalho

DESVIO DE RESPONSABILIDADES

Contexto e Motivação



- O que mudou entre as soluções ?
 - Na primeira, você presta atenção a uma série de detalhes e é responsável por tudo
 - Na segunda, instruções básicas são fornecidas e espera-se que cada pessoa saiba fazer o seu trabalho

DESVIO DE RESPONSABILIDADES

ESTUDANTES RESPONSÁVEIS PELO SEU PRÓPRIO COMPORTAMENTO

Contexto e Motivação



- Porque esta reorganização de responsabilidades é importante ?
 - Suponha que alunos bolsistas tenham de preencher um formulário de avaliação da aula antes de se dirigir à aula seguinte
 - Na primeira solução o programa de controle deve ser modificado de modo a distinguir os tipos de alunos e dar instruções específicas a cada tipo
 - Na segunda solução o comportamento dos alunos bolsistas seria modificado e o programa de controle continuaria dizendo “Vá para a sua próxima aula”

Contexto e Motivação



- Porque esta reorganização de responsabilidades é importante ?
 - Suponha que alunos bolsistas tenham de preencher um formulário de avaliação da aula antes de se dirigir à aula seguinte
 - Na primeira solução o programa de controle deve ser modificado de modo a distinguir os tipos de alunos e dar instruções específicas a cada tipo
 - Na segunda solução o comportamento dos alunos bolsistas seria modificado e o programa de controle continuaria dizendo “Vá para a sua próxima aula”

Contexto e Motivação



- Porque esta reorganização de responsabilidades é importante ?
 - Suponha que alunos bolsistas tenham de preencher um formulário de avaliação da aula antes de se dirigir à aula seguinte
 - Na primeira solução o programa de controle foi modificado de modo a distinguir os tipos de instruções específicas a cada tipo
 - Na segunda solução o comportamento dos alunos bolsistas seria modificado e o programa de controle continuaria dizendo “Vá para a sua próxima aula”

DESIGN FOR CHANGE

Quero saber mais !

Contexto e Motivação



- Para que isso funcione é necessário que:
 - Os alunos conheçam o seu tipo (bolsista / não bolsista) e sejam responsáveis pelo seu próprio comportamento
 - O programa de controle se comunique com diferentes tipos de alunos de maneira uniforme
 - O programa de controle não dependa dos passos particulares que estudantes executam para se dirigir de uma sala à outra

Contexto e Motivação



- Para que isso funcione é necessário que:
 - Os alunos conheçam o seu tipo (bolsista / não bolsista) e sejam responsáveis pelo seu próprio comportamento
 - O programa de controle se comunique com diferentes tipos de alunos de maneira uniforme
 - O programa de controle não dependa dos passos particulares que estudantes executam para se dirigir de uma sala à outra

Objeto como um
conjunto de
responsabilidades

Contexto e Motivação



- Para que isso funcione é necessário que:
 - Os alunos conheçam o seu tipo (bolsista / não bolsista) e sejam responsáveis pelo seu próprio comportamento
 - O programa de controle se comunique com diferentes tipos de alunos de maneira uniforme
 - O programa de controle não dependa dos passos particulares que estudantes executam para se dirigir de uma sala à outra

Interfaces como
um mecanismo
para representar
conceitos

Objeto como um
conjunto de
responsabilidades

Contexto e Motivação



que isso funcione é necessário que:

- Os alunos conheçam o seu tipo (bolsista / não bolsista) e sejam responsáveis pelo seu próprio comportamento
- O programa de controle se comunique com diferentes tipos de alunos de maneira uniforme
- O programa de controle não dependa dos passos particulares que estudantes executam para se dirigir de uma sala à outra

Objeto como um
conjunto de
responsabilidades

Herança como
conceitos

Contexto e Motivação



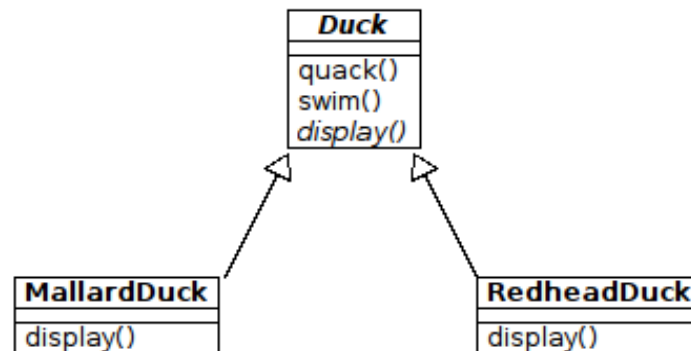
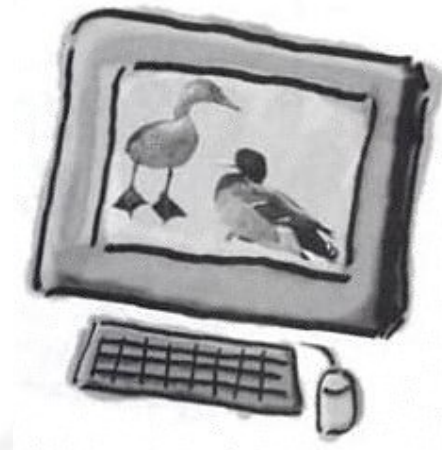
- Perspectivas no processo de desenvolvimento:

PERSPECTIVA	DESCRIÇÃO
CONCEITUAL	Qual a minha responsabilidade ?
ESPECIFICAÇÃO	Como eu sou utilizado ?
IMPLEMENTAÇÃO	

Contexto e Motivação



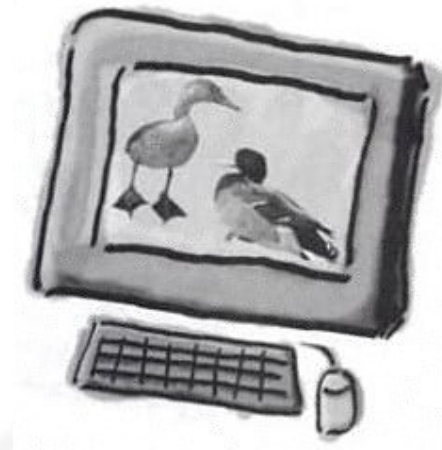
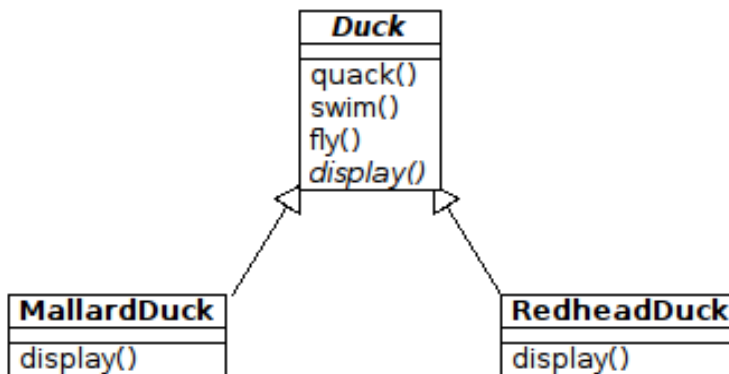
- Porque precisamos de padrões de projeto ?
 - Exemplo: jogo de patos



Contexto e Motivação



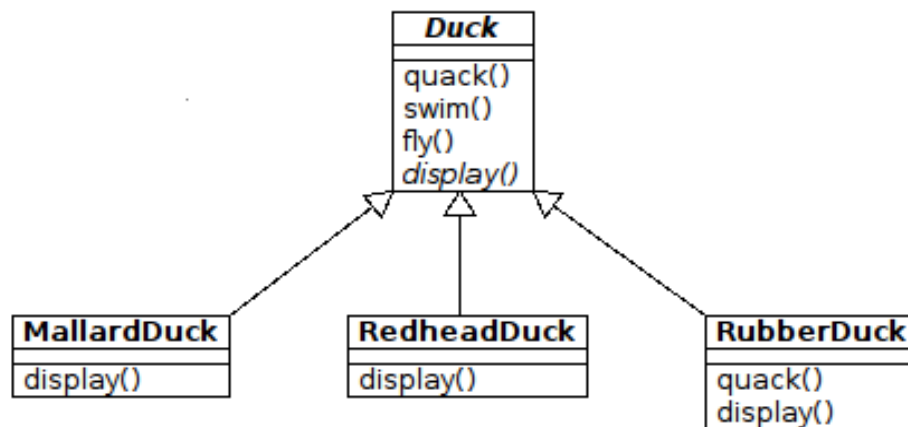
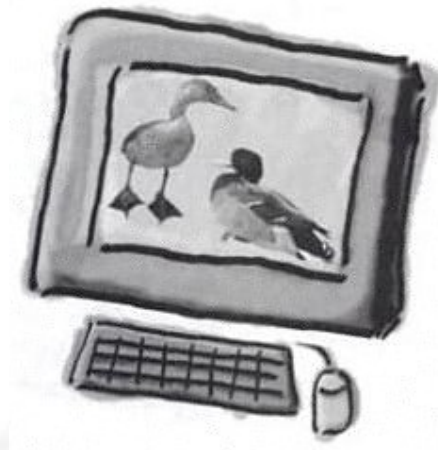
- Porque precisamos de padrões de projeto ?
 - 1a mudança: patos agora podem voar



Contexto e Motivação



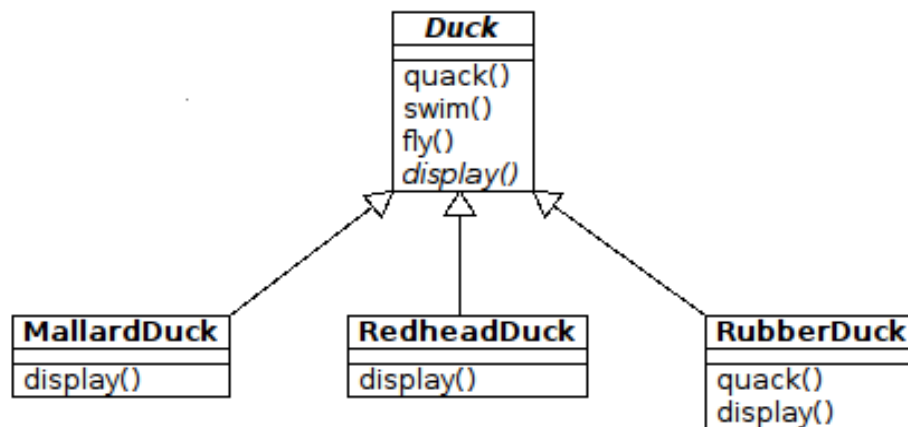
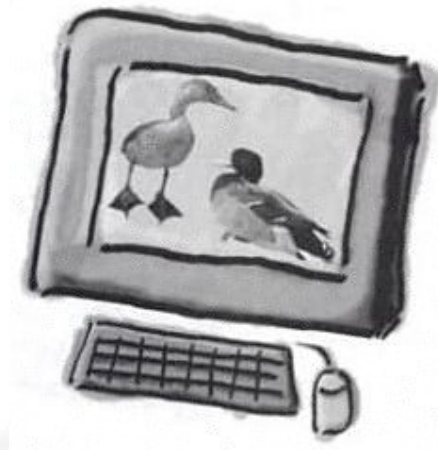
- Porque precisamos de padrões de projeto ?
 - 2a mudança: suportar patos de borracha



Contexto e Motivação



- Porque precisamos de padrões de projeto ?
 - 2a mudança: suportar patos de borracha

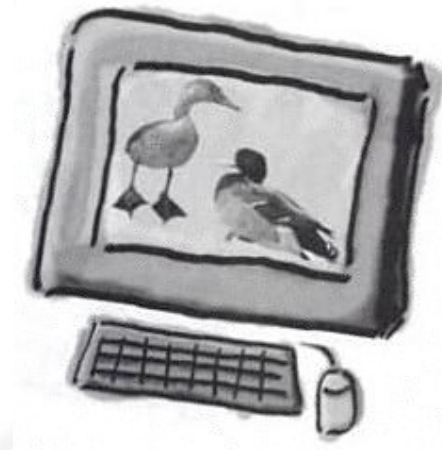
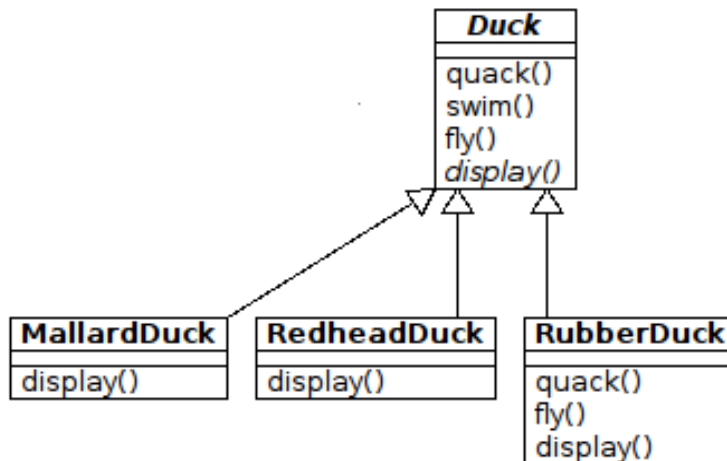


Hmm, patos de borracha não devem voar. O que fazer ?

Contexto e Motivação



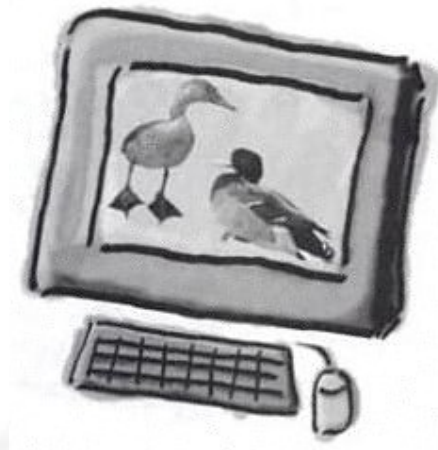
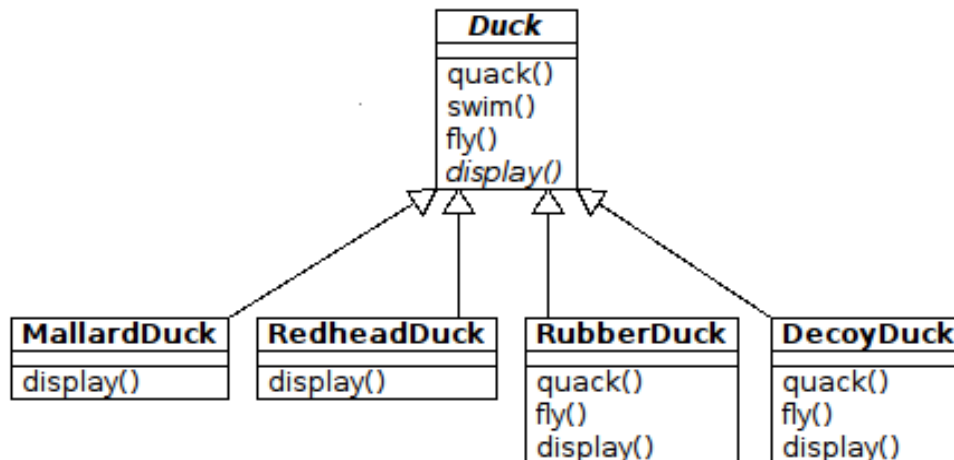
- Porque precisamos de padrões de projeto ?
 - 1a tentativa: sobreposições vazias ?



Contexto e Motivação



- Porque precisamos de padrões de projeto ?
 - 1a tentativa: sobreposições vazias ?

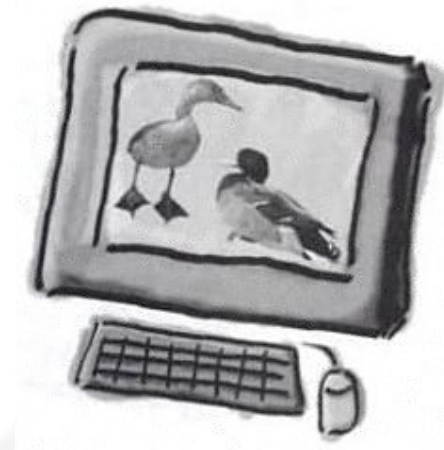
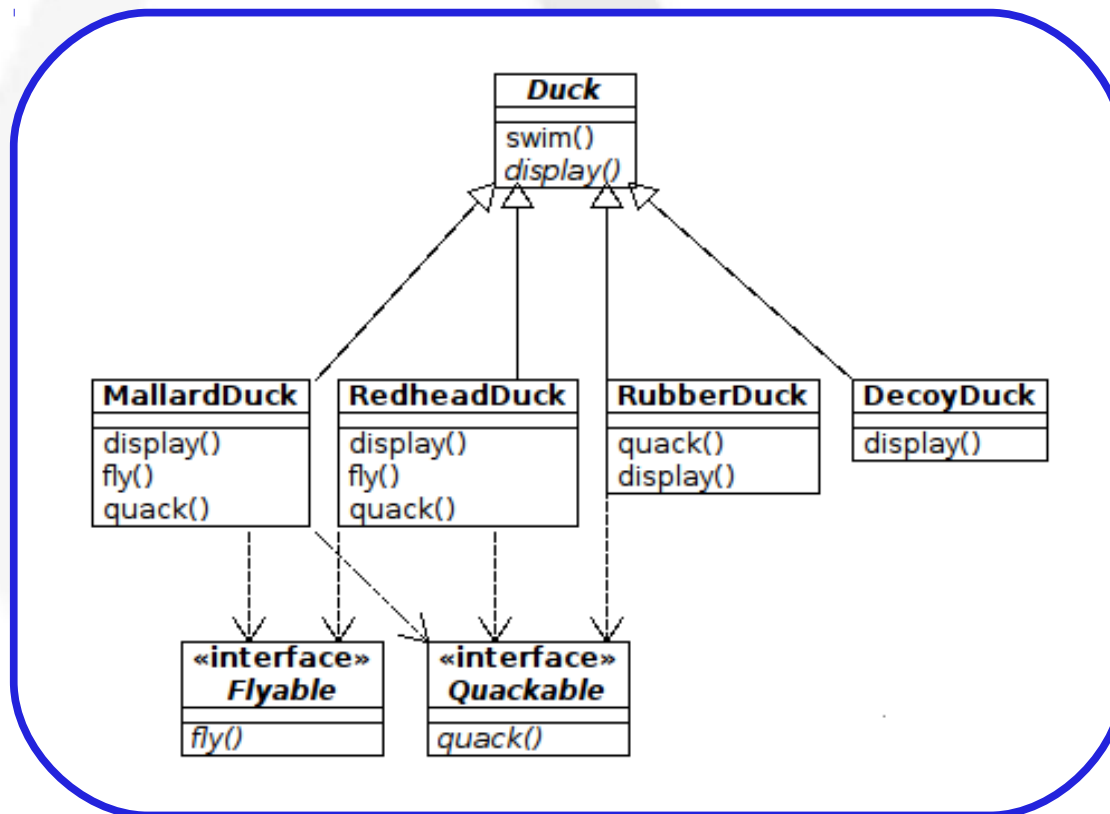


E se tivéssemos
que suportar
patos decorativos
?

Contexto e Motivação



- Porque precisamos de padrões de projeto ?
 - 2a tentativa: interfaces ?



Padrões de Projeto



Padrões de Projeto são descrições de objetos e classes que se comunicam e que são configurados para resolver um problema genérico de projeto OO em um contexto particular

- **Características:**
 - Promovem reuso de boas soluções
 - Estabelecem uma terminologia comum
 - Mantêm a discussão no âmbito de projeto, não de implementação
 - São descobertos e não inventados

Princípios de Projeto OO



- 1 Encontre o que varia e o encapsule
- 2 Programe pensando em interfaces
- 3 Prefira agregação a herança
- 4 Princípio do Aberto-Fechado (*Open-Closed Principle*)
- 5 Princípio da Substituição de Bárbara Liskov
- 6 Peça por ajuda, não por informação
- 7 *One Rule, One Place*

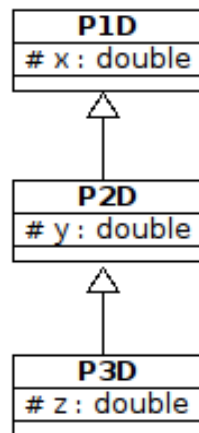
Princípios de Projeto OO



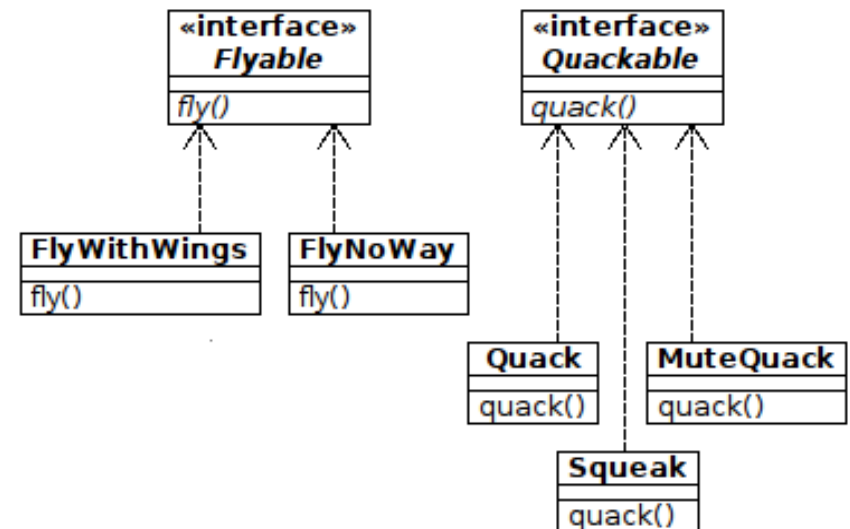
1

ENCONTRE O QUE VARIA E O ENCAPSULE

HERANÇA DE IMPLEMENTAÇÃO (sub-classing)



HERANÇA DE INTERFACE (sub-typing)



Princípios de Projeto OO



1

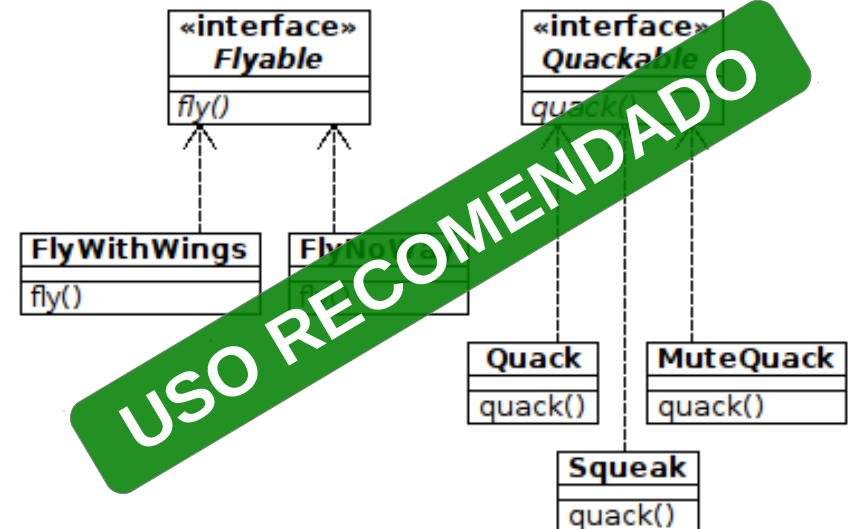
ENCONTRE O QUE VARIA E O ENCAPSULE

HERANÇA DE IMPLEMENTAÇÃO
(sub-classing)



USE COM CUIDADO

HERANÇA DE INTERFACE
(sub-typing)



USO RECOMENDADO

Princípios de Projeto OO



1

ENCONTRE O QUE SE ADEQUA À SITUAÇÃO

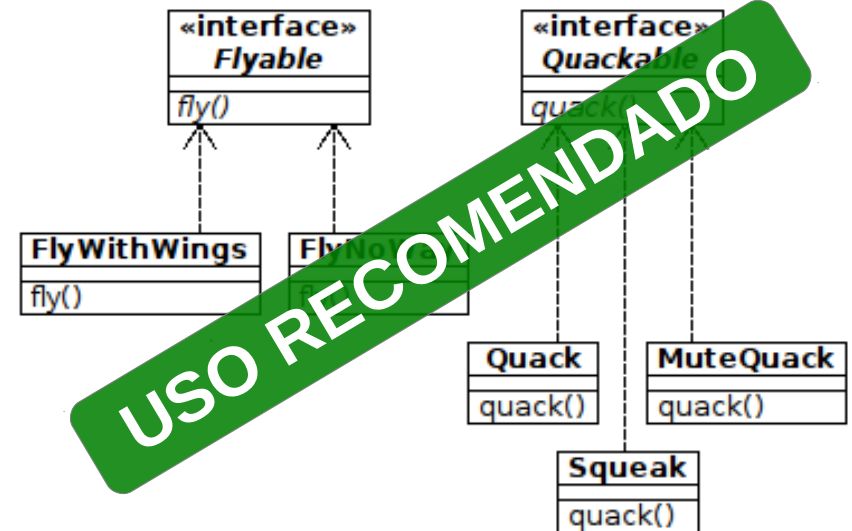
Quero saber mais !

HERANÇA DE IMPLEMENTAÇÃO
(sub-classing)



USE COM CUIDADO

HERANÇA DE INTERFACE
(sub-typing)



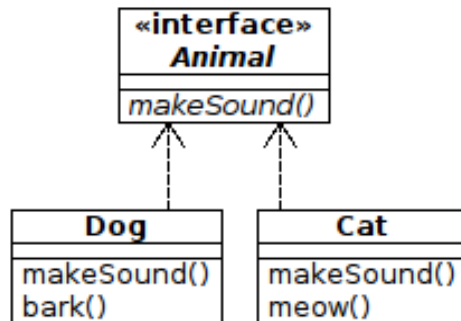
USO RECOMENDADO

Princípios de Projeto OO



2

PROGRAME PENSANDO EM INTERFACES



PROGRAMAÇÃO DIRIGIDA A IMPLEMENTAÇÕES

```
Dog dog = new Dog();
dog.bark();
```

PROGRAMAÇÃO DIRIGIDA A INTERFACES

```
Animal animal = new Dog();
animal.makeSound();
```

OU (AINDA MELHOR)

```
Animal animal = createAnimal();
animal.makeSound();
```

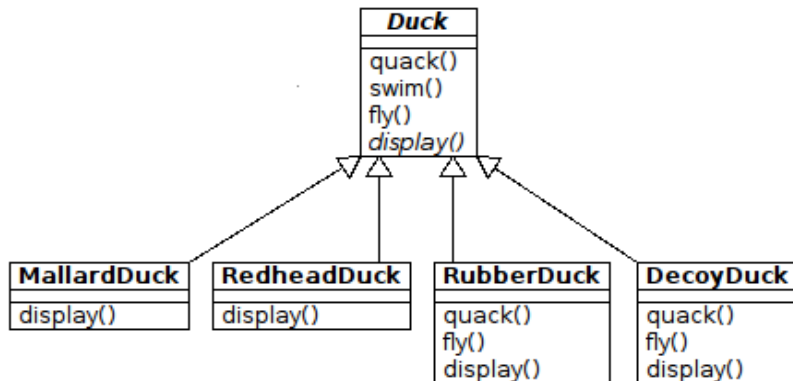
Princípios de Projeto OO



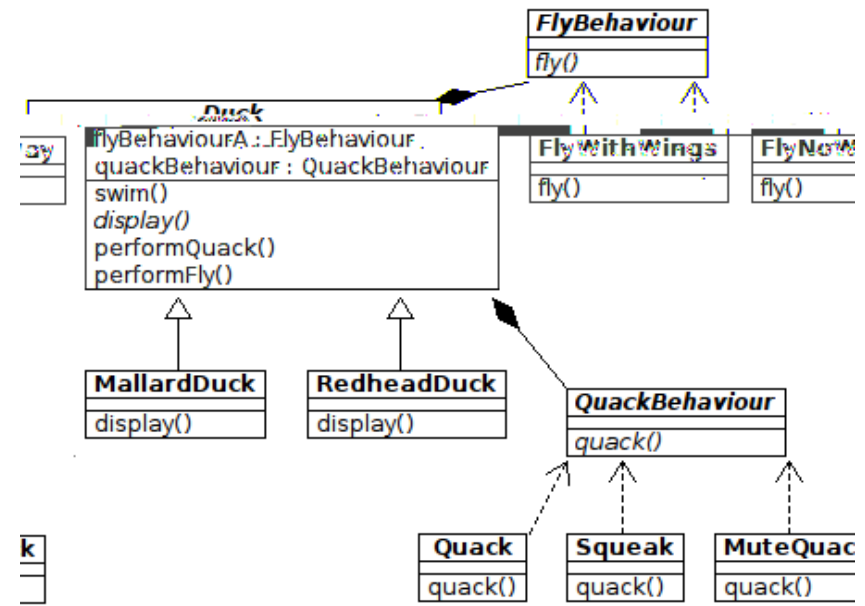
3

PREFIRA AGREGAÇÃO A HERANÇA

COM HERANÇA



COM AGREGAÇÃO

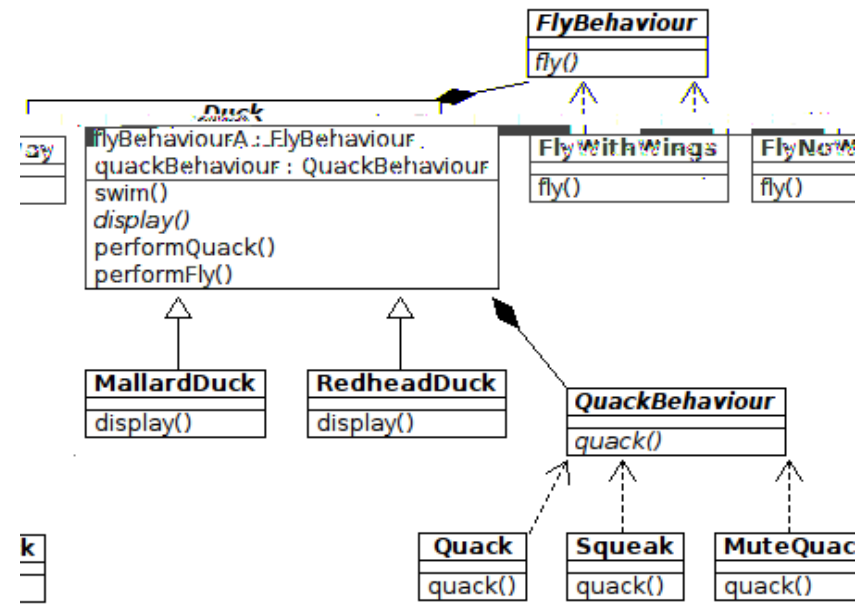


Princípios de Projeto OO



4 PRINCÍPIO DO ABERTO-FECHADO: módulos (classes) devem ser fechados para modificação e abertos para extensão

- Novas funcionalidades são criadas com a introdução de novas classes
- Evita a introdução de novos *bugs*
- Fundamenta uma arquitetura de *plugins*



Princípios de Projeto OO



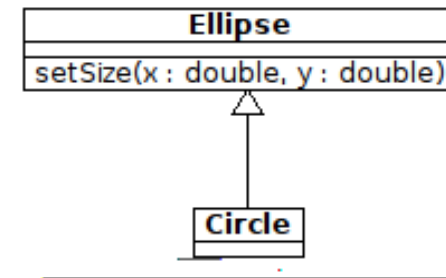
5

PRINCÍPIO DA SUBSTITUIÇÃO DE BÁRBARA LISKOV

Seja uma propriedade definida para objetos do tipo .
Então deve ser verdade para objetos do tipo , onde
é um sub-tipo de

- Define o conceito de *substitutability*
- Formaliza a metodologia de *design by contract*

VIOLAÇÕES DO PRINCÍPIO



Princípios de Projeto OO



5

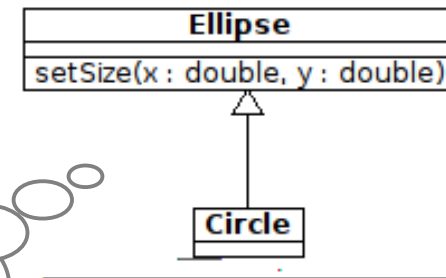
PRINCÍPIO DA SUBSTITUIÇÃO DE BÁRBARA LISKOV

Seja uma propriedade definida para objetos do tipo .
Então deve ser verdade para objetos do tipo , onde
é um sub-tipo de

Quero saber mais !

- De acordo com o conceito de *substitutability*
- Formaliza a metodologia de *design by contract*

VIOLAÇÕES DO PRINCÍPIO



Quero saber mais !

Princípios de Projeto OO



6

PEÇA POR AJUDA, NÃO POR INFORMAÇÃO

- “A manutenibilidade é inversamente proporcional à quantidade de dados que trafega entre os objetos”

[James Gosling]

- Em poucos casos *gets* e *sets* são justificados

COM GETS E SETS

```
Money a, b;  
a.setValue(a.getValue() +  
           b.getValue());
```

SEM GETS E SETS

```
Money a, b;  
a.increaseBy(b);
```

Princípios de Projeto OO



6

PEÇA POR AJUDA, NÃO POR INFORMAÇÃO

- “A manutenibilidade é inversamente proporcional à quantidade de dados que trafega entre

[James Gosling]

- Em poucos *sets* são justificados

Quero saber mais !

COM GETS E SETS

```
Money a, b;  
a.setValue(a.getValue() +  
           b.getValue());
```

SEM GETS E SETS

```
Money a, b;  
a.increaseBy(b);
```

Princípios de Projeto OO



7

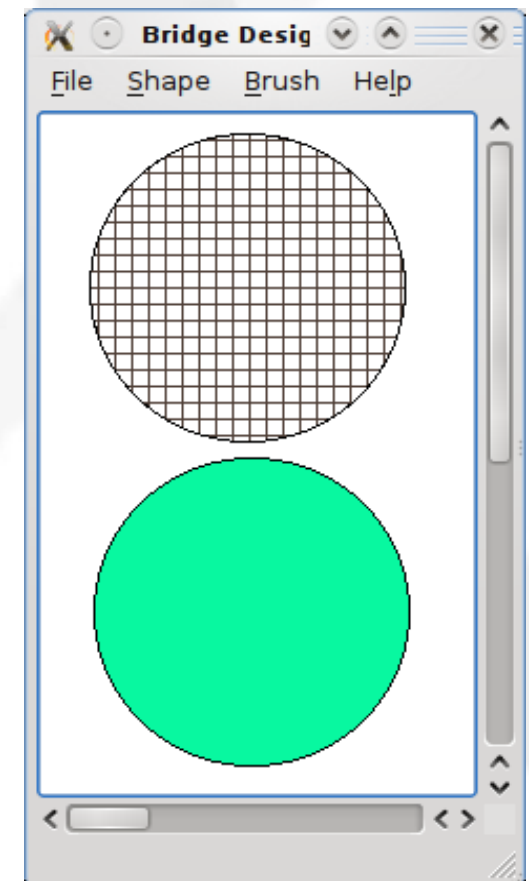
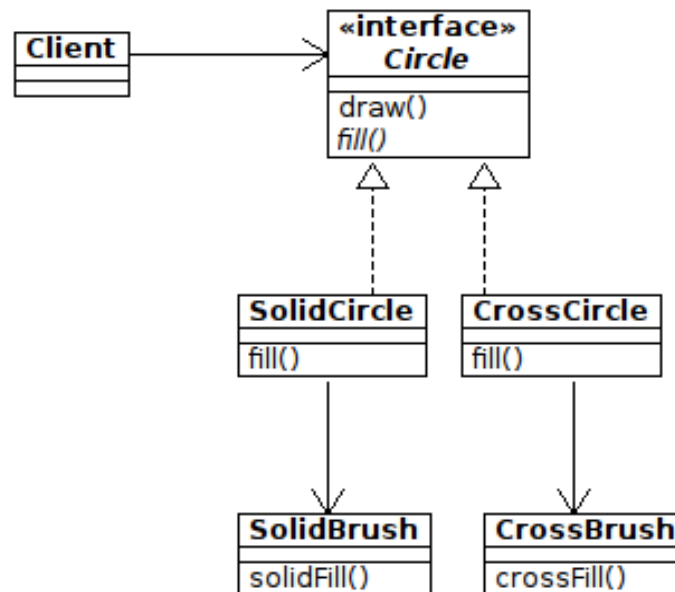
ONE RULE, ONE PLACE

- Cada regra de negócio deve ser implementada em somente um lugar, sem código redundante

Estudo de Caso: Bridge

- Contexto: você deve implementar um sistema que desenhe círculos com dois tipos diferentes de preenchimento - sólido e cruzado

1ª SOLUÇÃO

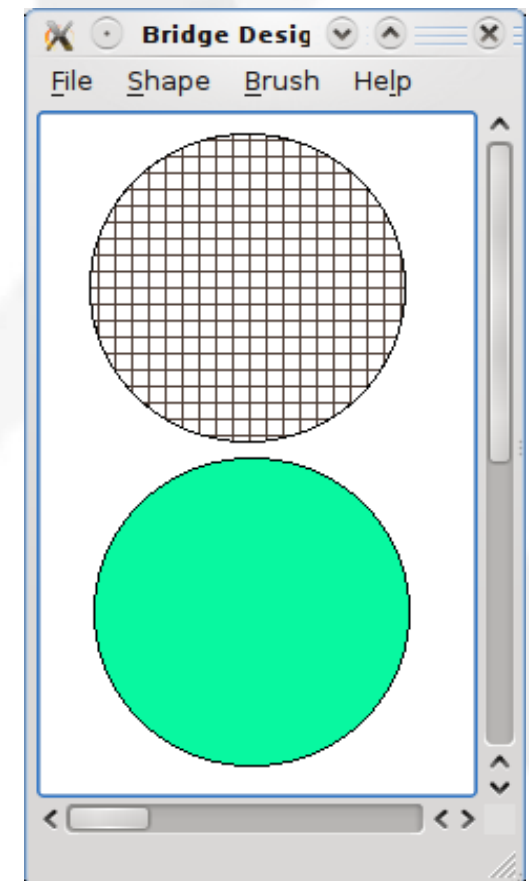
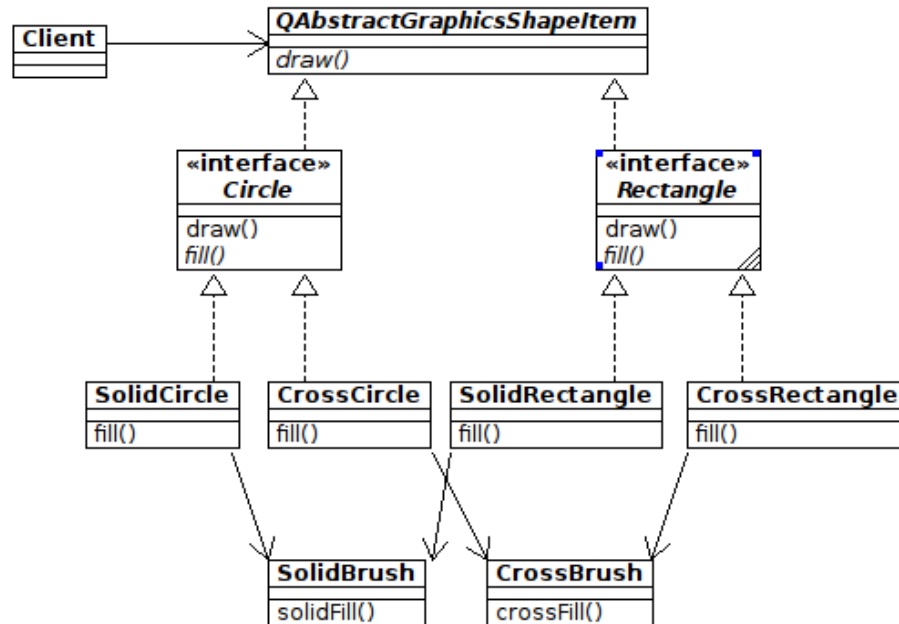


Estudo de Caso: Bridge



- 1a mudança: o sistema deve agora também desenhar retângulos com as duas possibilidades de preenchimento

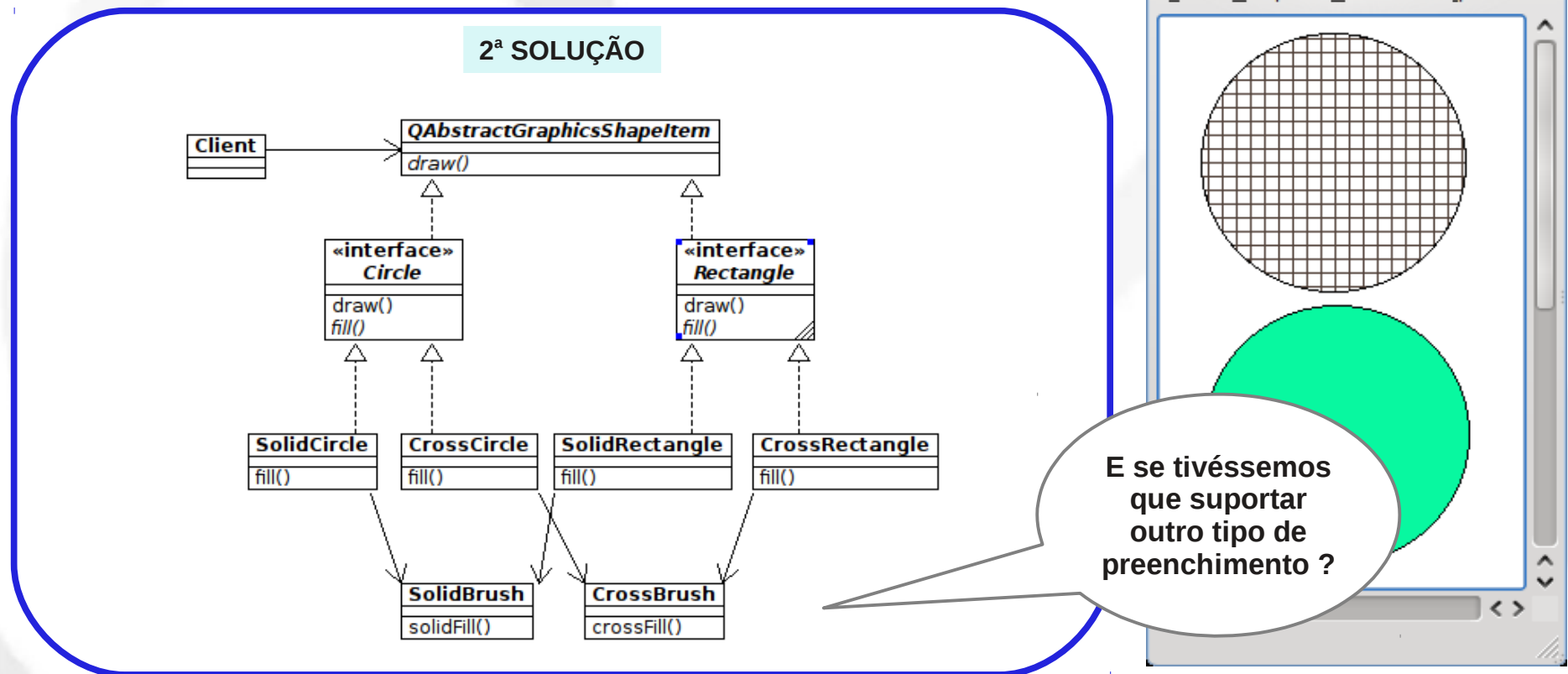
2ª SOLUÇÃO



Estudo de Caso: Bridge



- 1a mudança: o sistema deve agora também desenhar retângulos com as duas possibilidades de preenchimento



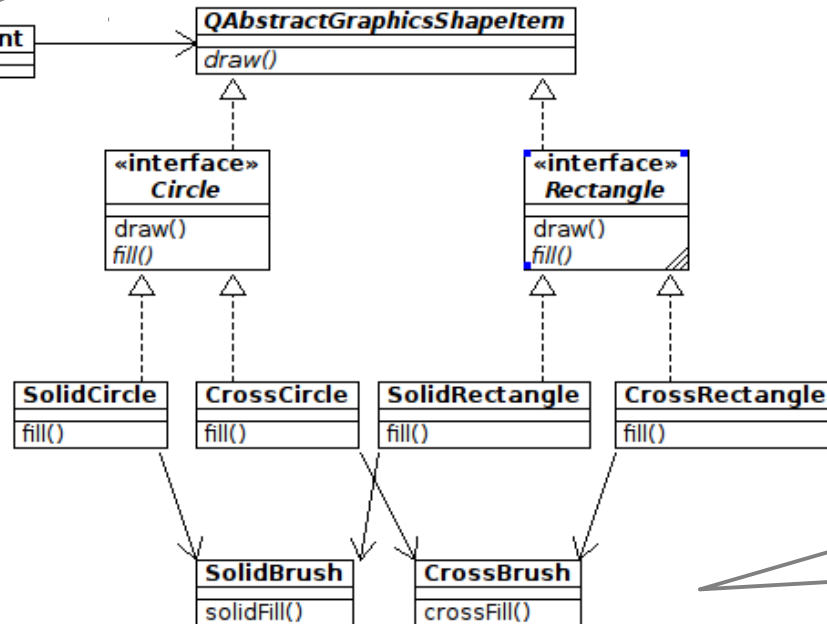
Estudo de Caso: Bridge



- 1a mudança: o sistema deve agora também desenhar retângulos com as duas possibilidades de preenchimento

E se tivéssemos que suportar outra primitiva gráfica ?

2ª SOLUÇÃO

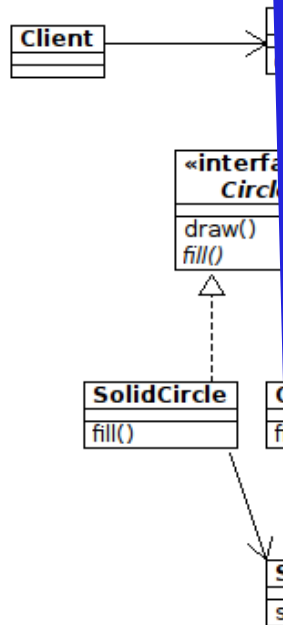


E se tivéssemos que suportar outro tipo de preenchimento ?



Estudo de

- 1a mudança: o
pode
retângulos co
ixe v7



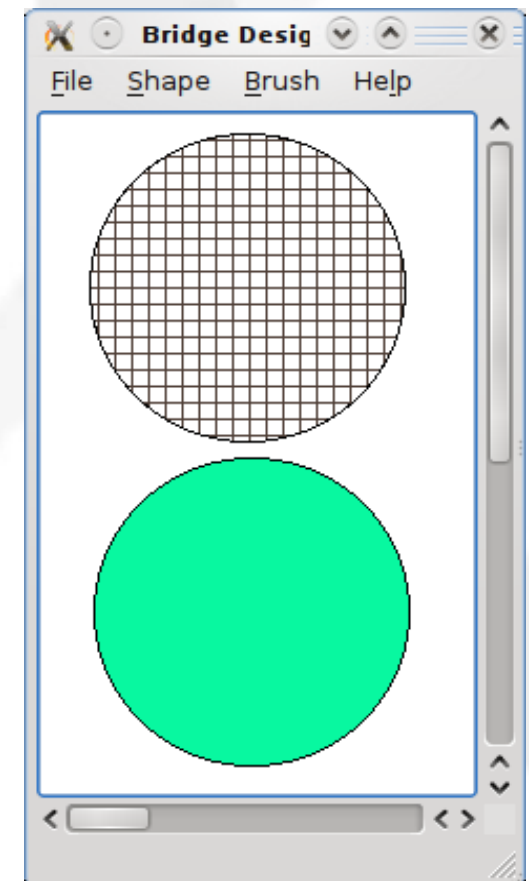
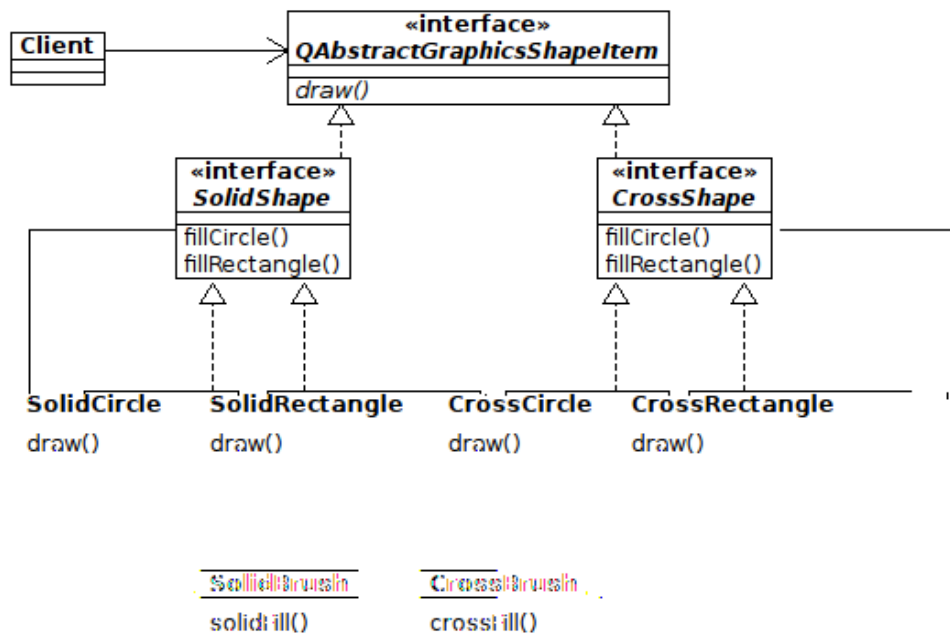
Área



Estudo de Caso: Bridge

- 1a mudança: o sistema deve agora também desenhar retângulos com as duas possibilidades de preenchimento

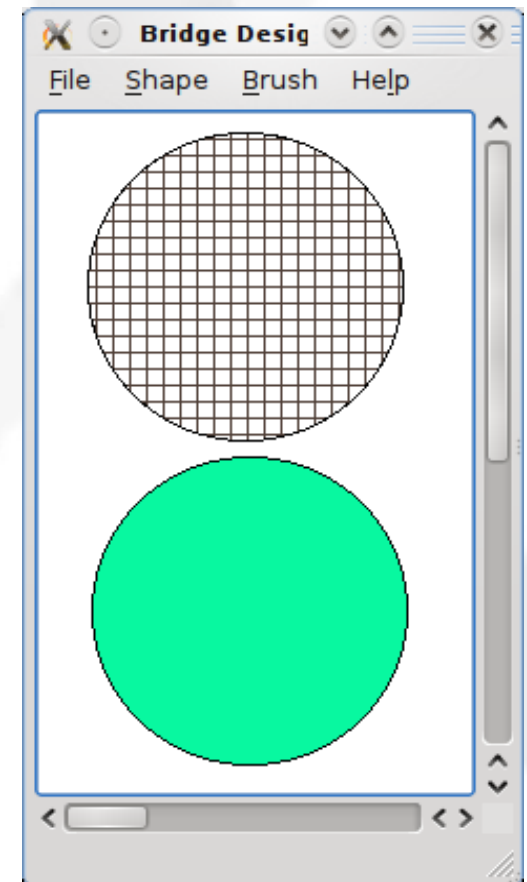
3ª SOLUÇÃO



Estudo de Caso: Bridge



- 1a mudança: o sistema deve agora também desenhar retângulos com as duas possibilidades de preenchimento



Estudo de Caso: Bridge

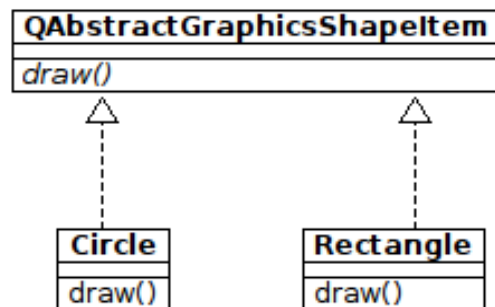


- O que há de errado com estes projetos ?
 - Violam o princípio ❶ “Encontre o que varia e o encapsule” e o princípio ❸ “Prefira agregação a herança”
- Quais aspectos variam ?
 - As primitivas gráficas: círculo, retângulo etc
 - As formas de preenchimento: sólido, cruzado etc
- Deve-se conceber tais aspectos como conceitos que podem possuir diferentes implementações
- Heranças de implementação acoplam eternamente a classe-filha com a classe-pai

Estudo de Caso: Bridge



- Um projeto melhorado:

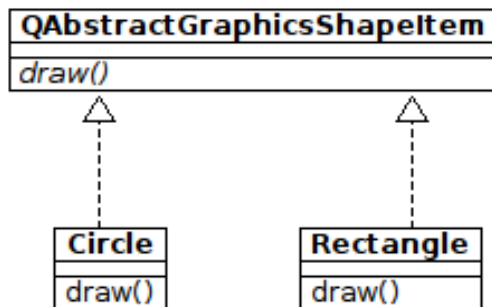


Estudo de Caso: Bridge



- Um projeto melhorado:

1º conceito
que varia



Estudo de Caso: Bridge



- Um projeto melhorado:



Estudo de Caso: Bridge



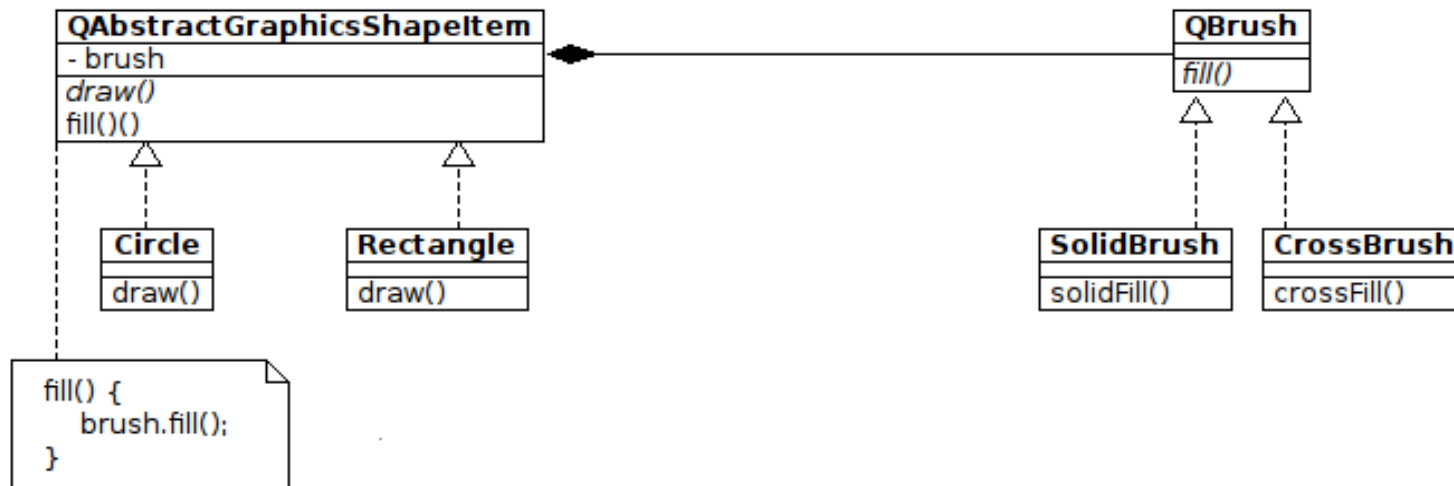
- Um projeto melhorado:



Estudo de Caso: Bridge



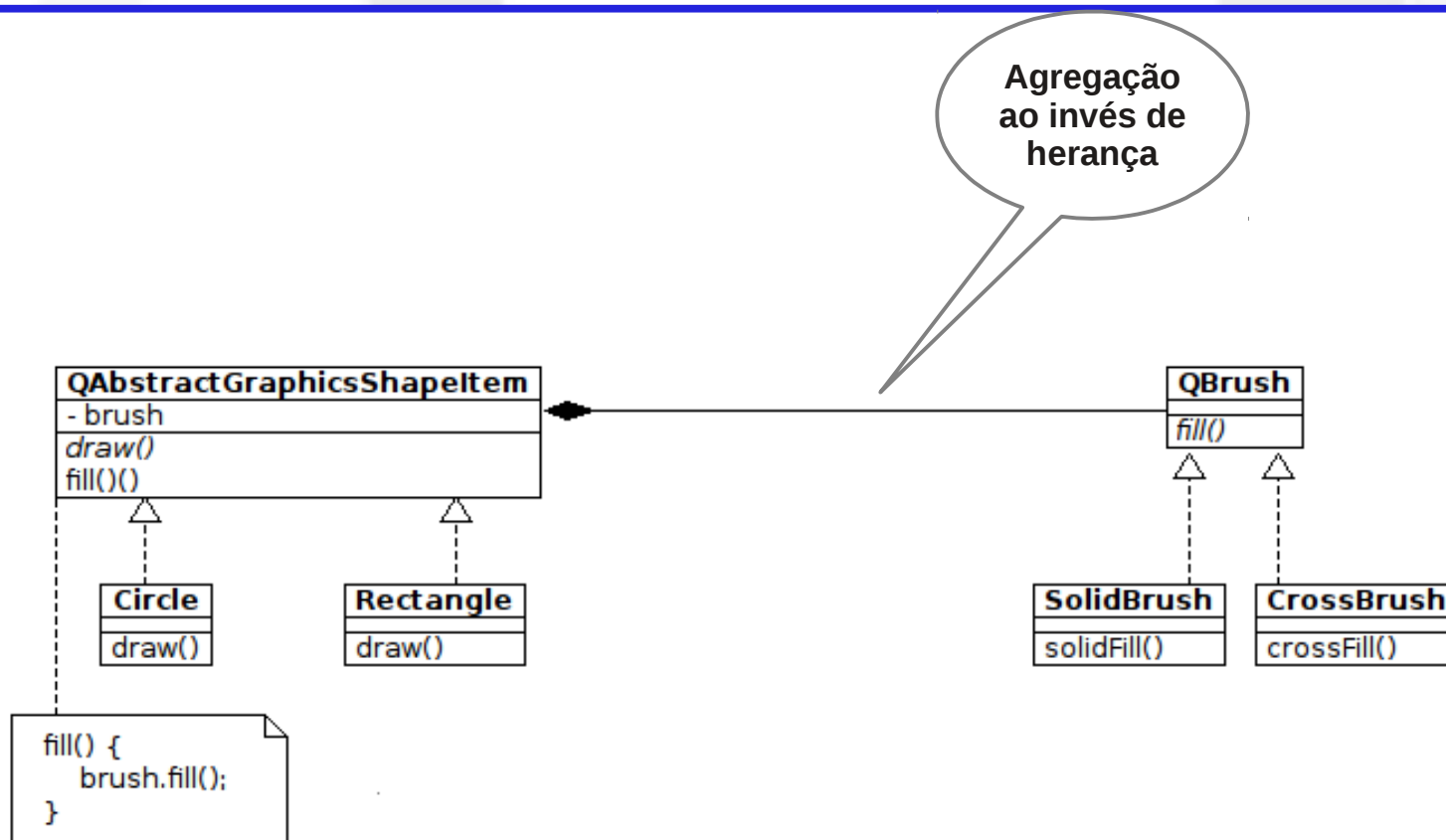
- Um projeto melhorado:



Estudo de Caso: Bridge



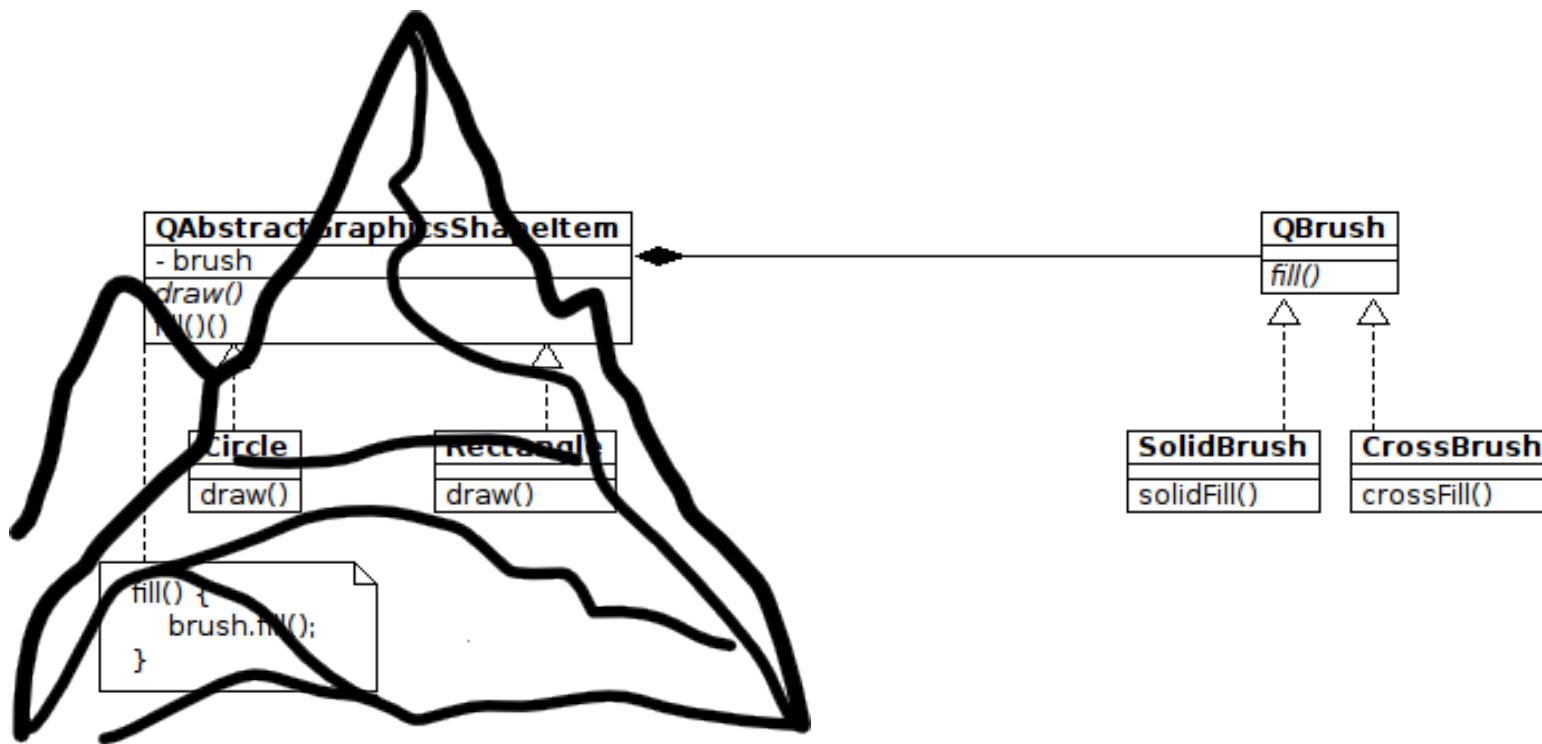
- Um projeto melhorado:



Estudo de Caso: Bridge



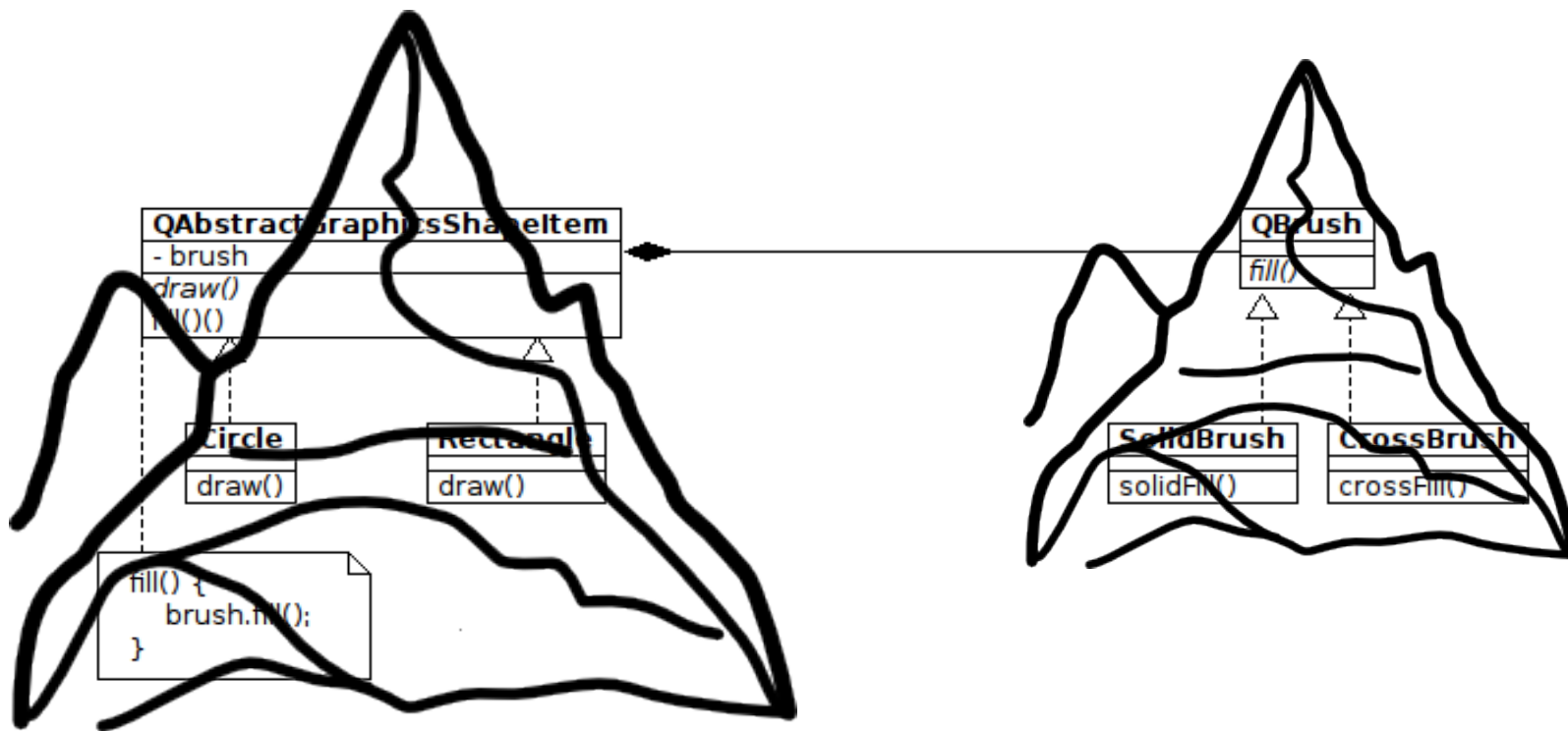
- Um projeto melhorado:



Estudo de Caso: Bridge



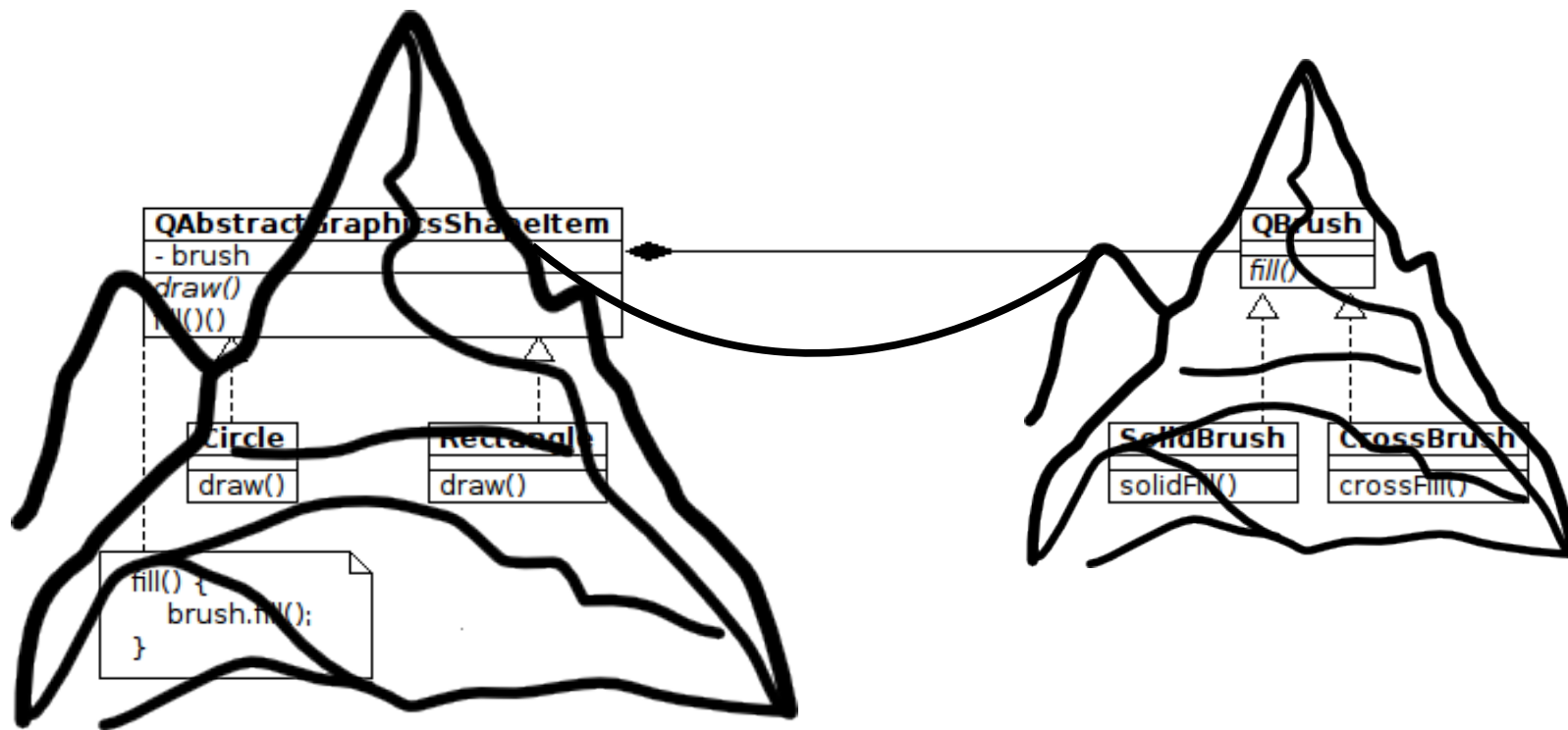
- Um projeto melhorado:



Estudo de Caso: Bridge



- Um projeto melhorado:



Estudo de Caso: Bridge



BRIDGE

Desacopla a abstração da sua implementação, de modo que os dois possam variar de forma independente

Padrões de Projeto

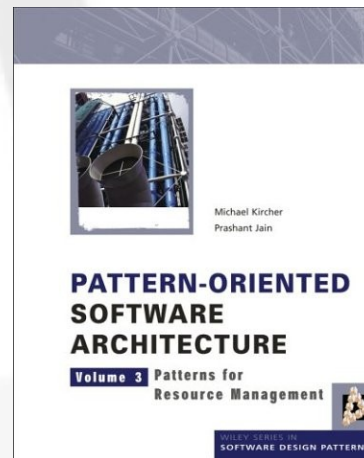
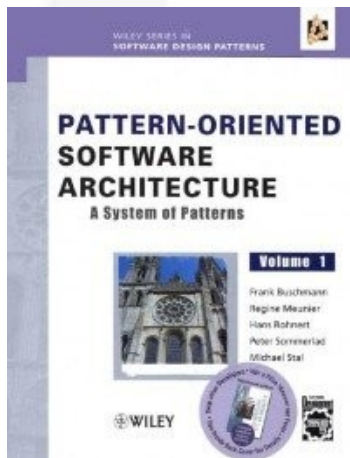


- Muitos outros padrões existem e são amplamente utilizados:
 - *Creational Patterns*: desacoplam o código que cria as instâncias de classes do código que efetivamente utiliza estes objetos. Ex: *Abstract Factory*, *Builder* e *Prototype*
 - *Structural Patterns*: formam estruturas maiores através da composição de objetos, sem comprometer a flexibilidade da solução. Ex: *Adapter*, *Bridge*, *Decorator* e *Flyweight*
 - *Behavioral Patterns*: descrevem padrões de comunicação entre objetos, permitindo a variação de algoritmos e de responsabilidades. Ex: *Strategy*, *Observer* e *Iterator*

Padrões de Projeto



- Mas não é só isso:
 - Outros padrões estão disponíveis para solucionar problemas de concorrência, computação distribuída, tempo-real e aspectos específicos de domínio



Padrões de Projeto



- Padrões são aplicados em diversos níveis de abstração:

PADRÕES E ESTILOS ARQUITETURAIS

PADRÕES DE PROJETO

IDIOMAS DE PROGRAMAÇÃO

Padrões de Projeto



- Padrões são aplicados em diversos níveis de abstração:

PADRÕES E ESTILOS ARQUITETURAIS

PADRÕES DE PROJETO

IDIOMAS DE PROGRAMAÇÃO

Boas práticas para desenvolvimento em uma determinada linguagem

Padrões de Projeto



- Padrões são aplicados em diversos níveis de abstração:

PADRÕES E ESTILOS ARQUITETURAIS

PADRÕES DE PROJETO

IDIOMAS DE PROGRAMAÇÃO

Soluções flexíveis
para problemas de
projeto OO
recorrentes

Boas práticas para
desenvolvimento em
uma determinada
linguagem

Padrões de Projeto



- Padrões são aplicados em diversos níveis de abstração:

PADRÕES E ESTILOS ARQUITETURAIS

Soluções recorrentes
para modularização
em alto nível

PADRÕES DE PROJETO

IDIOMAS DE PROGRAMAÇÃO

Soluções flexíveis
para problemas de
projeto OO
recorrentes

Boas práticas para
desenvolvimento em
uma determinada
linguagem

Idiomas de Programação

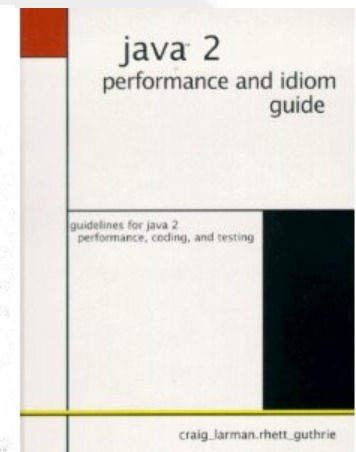
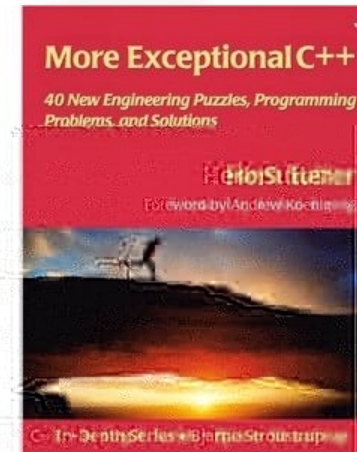
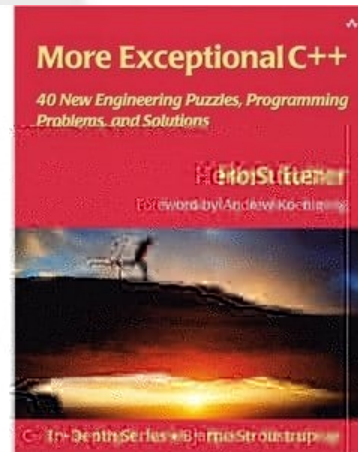
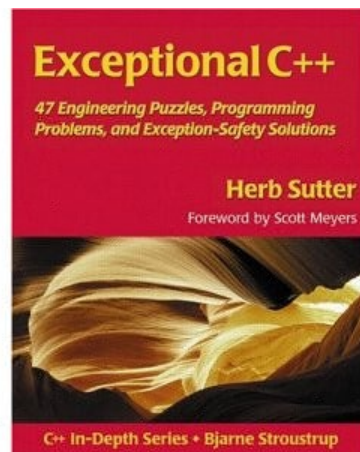
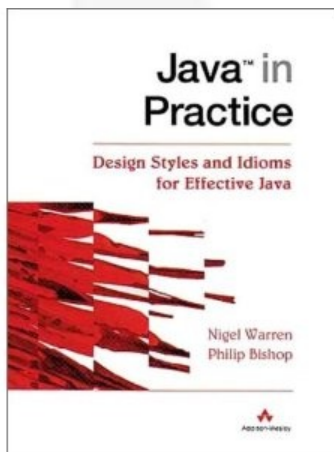


- Um Idioma de Programação é um meio para expressar uma construção recorrente em uma ou mais linguagens de programação
- Exemplos de idiomas simples:
 - Incremento de um contador:
 - BASIC: $i = i + 1$
 - C: $i += 1$ ou $++i$ ou $i++$
 - Pascal: `Inc(i)`
 - *Swap* de variáveis:
 - Maioria das linguagens: $\text{temp} = a; a = b; b = \text{temp};$
 - Perl: $(\$a, \$b) = (\$b, \$a);$

Idiomas de Programação



- Os idiomas são importantes para obter fluência na linguagem de programação e desenvolver código mais robusto e de melhor qualidade
- Geralmente os benefícios obtidos com o uso de idiomas não são solicitados durante a compilação, sob a forma de erros



Idiomas de Programação



- Exemplo: operadores de cópia no C++

SEM APLICAR O IDIOMA

```
void operador= (Matrix m)
{
    if (this == &m)
        return *this;

    // Implementação da atribuição
}
```

APLICANDO O IDIOMA

```
Matrix &operador= (const Matrix &m)
{
    if (this == &m)
        return *this;

    // Implementação da atribuição
    Return *this;
}
```

Idiomas de Programação



- Exemplo: *Pointer to Implementation* (pimpl)

VERSÃO 1.0 DA CLASSE BOOK

```
class Book
{
public:
    void print();
private:
    std::string m_Contents;
};
```

VERSÃO 1.1 DA CLASSE BOOK

```
class Book
{
public:
    void print();
private:
    std::string m_Contents;
    std::string m_Title;
};
```

Idiomas de Programação



- Exemplo: *Pointer to Implementation* (pimpl)
 - As modificações realizadas da versão 1.0 para a 1.1 quebram a compatibilidade binária e irá requerer que todas as aplicações que usam Book sejam recompiladas
 - Isso não pode ser feito a menos que uma nova versão *major* seja lançada, por exemplo, 2.0.

Idiomas de Programação



- Exemplo: *Pointer to Implementation* (pimpl)
- Solução:

VERSÃO 1.1 DA CLASSE BOOK com PIMPL (arquivo .h)

```
class Book
{
public:
    void print();
private:
    class BookImpl* m_p;
};
```

VERSÃO 1.1 DA CLASSE BOOK com PIMPL (arquivo .cpp)

```
class BookImpl
{
Public:
    void print();
private:
    std::string m_Contents;
    std::string m_Title;
};
```

```
Book::Book()
{
    m_p = new BookImpl();
}
void Book::print()
{
    m_p->print();
}
```

Idiomas de Programação



- Mais idiomas do C++ [http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms]
 - Adapter Template
 - Address Of
 - Algebraic Hierarchy
 - Attach by Initialization
 - Attorney-Client
 - Barton-Nackman trick
 - Base-from-Member
 - Boost mutant
 - Calling Virtuals During Initialization
 - Capability Query
 - Checked delete
 - Clear-and-minimize
 - Coercion by Member Template
 - Compile Time Control Structures
 - Computational Constructor
 - Concrete Data Type
 - Const auto_ptr
 - Construct On First Use
 - Construction Tracker
 - Copy-and-swap
 - Copy-on-write
 - Counted Body (intrusive reference counting)
 - Curiously Recurring Template Pattern
 - Detached Counted Body (non-intrusive reference counting)
 - Empty Base Optimization
 - Emulated Exception
 - enable-if
 - Envelope Letter
 - Erase-Remove
 - Exemplar

Idiomas de Programação



- 4 anti-idíomas do Java e como resolvê-los

<http://www.javaworld.com/javaworld/jw-07-2008/jw-07-harmful-idioms.html>

1) Variáveis locais, argumentos e atributos: quem é quem ?

SEM APLICAR O IDIOMA

```
public boolean equals (Object arg) {  
    if (! (arg instanceof Range)) return false;  
    Range other = (Range) arg;  
    return start.equals(other.start) && end.equals(other.end);  
}
```

APLICANDO O IDIOMA

```
public boolean equals (Object aOther) {  
    if (! (aOther instanceof Range)) return false;  
    Range other = (Range) aOther;  
    return fStart.equals(other.fStart) && fEnd.equals(other.fEnd);  
}
```

Idiomas de Programação



- 4 anti-idiomias do Java e como resolvê-los

<http://www.javaworld.com/javaworld/jw-07-2008/jw-07-harmful-idioms.html>

2) Package by layer: impedindo o uso do escopo *package-private*

SEM APLICAR O IDIOMA – package by layer - (todas as classes públicas)

com.blah.action
com.blah.dao
com.blah.model
com.blah.util

APLICANDO O IDIOMA – package by feature

com.blah.painting
com.blah.buyer
com.blah.seller
com.blah.auction
com.blah.webmaster
com.blah.useraccess

Idiomas de Programação



- 4 anti-idíomas do Java e como resolvê-los

<http://www.javaworld.com/javaworld/jw-07-2008/jw-07-harmful-idioms.html>

2) Package by layer: impedindo o uso do escopo *package-private*

- Vantagens do *package by feature*:
 - Pacotes com maior coesão e modularidade. O acoplamento é minimizado
 - Maior auto-documentação de código: “*Holy Grail of legibility*”
[Code Complete: A Practical Handbook of Software Construction]
 - Ainda há separação por camadas, através de classes separadas dentro de cada funcionalidade
 - Itens relacionados estão sempre no mesmo local
 - Itens são *package-private* por *default*, como deveriam ser
 - Para remover uma funcionalidade remove-se simplesmente um diretório
 - Haverá menos itens por package e a evolução será mais natural

Idiomas de Programação



- 4 anti-idíomas do Java e como resolvê-los

<http://www.javaworld.com/javaworld/jw-07-2008/jw-07-harmful-idioms.html>

3) JavaBeans: porque utilizá-los quando temos os *immutables*?

- Vantagens dos objetos *immutables*:
 - São simples, *thread-safe* e não requerem sincronização
 - Podem ser compartilhados e não precisam ser *deep-copied*
 - Nunca precisam de um *copy constructor* ou método *clone()*
 - Constituem bons *building blocks* para outras classes e boas chaves para *Maps*
 - Possuem *failure atomicity*

Idiomas de Programação



- 4 anti-idíomas do Java e como resolvê-los

<http://www.javaworld.com/javaworld/jw-07-2008/jw-07-harmful-idioms.html>

3) JavaBeans: porque utilizá-los quando temos os *immutable*?

- Provavelmente *ResultsSets* não seria implementados como JavaBeans:
 - Construtores sem parâmetros não fazem sentido
 - Integração com *events* e *listeners* também não
 - Possuiriam um mecanismo de validação de dados
 - Este mecanismo reportaria erros para o usuário

Idiomas de Programação



- 4 anti-idíomas do Java e como resolvê-los

<http://www.javaworld.com/javaworld/jw-07-2008/jw-07-harmful-idioms.html>

4) Atributos privados: porque colocá-los antes ?

```
public class OilWell implements EnergySource {  
    private Long id;  
    private String name;  
    private String location;  
    private Date discoveryDate;  
    private Long totalReserves;  
    private Long productionToDate;  
  
    public Long getId() {  
        return id;  
    }  
  
    public void setId(Long id) {  
        this.id = id;  
    }  
}
```

Idiomas de Programação



- Mais idiomas do Java:
 - <http://c2.com/cgi/wiki?Javaldioms>
 - <http://www.cs.wright.edu/~tkprasad/courses/ceg860/paper/paper.html>

Conclusões



- Estudar padrões de projeto é uma excelente oportunidade para efetivamente entender os princípios fundamentais da orientação a objetos
- Existe uma solução de compromisso entre o uso de padrões de projeto e simplicidade (*paralysis by analysis*)
- Catálogos de padrões para domínios específicos estão disponíveis (J2EE, *real-time*, sistemas distribuídos, etc)
- Boas bibliotecas, tais como o Qt 4 ou as APIs do Java fazem uso intensivo de padrões e merecem ser estudadas
- Um padrão de projeto geralmente aparece em conjunto com (ou como parte de) outro padrão



Pós-Graduação em Computação Distribuída e Ubíqua

INF612 - Aspectos Avançados em Engenharia de Software
Padrões de Projeto e Idiomas de Programação

Sandro S. Andrade
sandroandrade@ifba.edu.br