Padrões de Projeto
Curso 6170 – Aulas 12 a 14

1. Padrões de projeto

Um padrão de projeto é:

. uma solução padrão para um problema comum de programação
. uma técnica capaz de tornar o código mais flexível ao fazer com que o código satisfaça certos critérios
. um projeto ou uma estrutura de implementação que satisfaz com sucesso um propósito específico
. um idioma de programação em alto nível
. uma maneira mais prática de se descrever certos aspectos da organização de um programa
. conexões entre componentes de programas
. a forma de um diagrama de objeto ou de um modelo de objeto

1.1 Exemplos

Aqui estão alguns exemplos de padrões de projeto os quais você já deve ter visto. Para cada padrão de projeto, esta lista apresenta o problema que se está tentando resolver, a solução fornecida pelo padrão de projeto, e quaisquer desvantagens associadas com o padrão de projeto. Um projetista de software deve balancear as vantagens e desvantagens quando decidir qual padrão utilizar. Projetos balanceados entre flexibilidade e performance são comuns, como você irá descobrir, na ciência da computação (e em outros campos).

Encapsulamento (escondendo os dados)

Problema: campos expostos podem ser manipulados diretamente a partir de código externo, levando a violações da invariante de representação ou a dependências indesejáveis que impedem a alteração da implementação.

Solução: esconda alguns componentes, permitindo apenas acessos estilizados ao objeto.

Desvantagens: a interface pode não (eficientemente) fornecer todas as operações desejadas. O acesso indireto pode reduzir a performance.

Derivação de classes (herança)

Problema: abstrações similares possuem membros similares (campos e métodos). Esta repetição é tediosa, propensa a erros, e uma dor de cabeça durante a manutenção.

Solução: derive membros padrão de uma super classe; em tempo de execução, selecione a implementação correta através de resoluções a respeito de qual implementação deve ser executada.

Desvantagens: o código de uma classe acaba ficando distribuído em várias, reduzindo o potencial de compreensão. A utilização de resoluções em tempo de execução introduz *overhead* (processamento extra).

Iteração

Problema: clientes que desejam acessar todos os membros de uma coleção devem realizar uma varredura especializada para cada tipo de coleção de dados na qual a iteração irá ocorrer. Isto introduz dependências indesejáveis que impedem a extensão do código para outras coleções.

Solução: implementações, realizadas com conhecimento da representação da coleção, realizam varreduras e registram o progresso da iteração. Os resultados são comunicados aos clientes através de uma interface padrão.

Desvantagens: a ordem da iteração é fixada pela implementação e não está sob o controle do cliente.

Exceções

Problema: problemas que ocorrem em uma parte do código, normalmente, devem ser manipulados em outro lugar. O código não deve ser poluído com rotinas de manipulação de erro, e nem de valores de retorno para identificação de erros.

Solução: introduzir estruturas de linguagem para que se jogue e intercepte exceções.

Desvantagens: desta forma o código ainda pode estar cheio de rotinas de manipulação de erro. Pode ser difícil saber onde uma exceção será manipulada. Os programadores podem ser tentados a utilizar exceções para o controle do fluxo normal de execução, o que é confuso e normalmente ineficiente.

Estes padrões de projeto são tão importantes que eles estão embutidos no Java. Outros padrões de projeto são tão importantes que estão embutidos em outras linguagens. Alguns padrões de projeto podem nunca ser construídos dentro de uma linguagem, mas ainda assim são úteis.

1.2 Quando (não) utilizar padrões de projeto

A primeira regra da utilização de padrões de projeto é a mesma que a primeira regra da utilização de otimizações: protelar. Assim como não se deve realizar a otimização prematuramente, não utilize padrões de projeto prematuramente. O melhor a se fazer é implementar alguma coisa primeiro e se certificar de que funciona, utilize então o padrão de projeto para melhorar as fraquezas; isto é verdade, especialmente, se você ainda não identificou todos os detalhes do projeto (se você compreender totalmente o domínio e o problema, pode fazer sentido utilizar padrões desde o de início, assim como faz sentido utilizar algoritmos mais eficientes desde o princípio em algumas aplicações).

Padrões de projeto podem aumentar ou diminuir a capacidade de compreensão de um projeto ou uma implementação. Eles podem diminuir a capacidade de compreensão ao adicionar acessos indiretos ou ao aumentar a quantidade de código. Eles podem aumentar a capacidade de compreensão ao melhorar a modularidade, separando melhor os conceitos, e simplificando a descrição. Uma vez que você aprenda o vocabulário dos padrões de projeto, você será capaz de interagir mais precisamente e rapidamente e com outras pessoas que também utilizam este vocabulário. Por exemplo, é melhor dizer "esta é uma instância do padrão V*isitor*" do que "este é um código que varre a estrutura e realiza chamadas de retorno, sendo que alguns métodos devem estar presentes, para serem chamados em uma ordem particular e de uma determinada forma".

A maioria das pessoas utiliza padrões de projeto quando percebem um problema com seu projeto - alguma coisa que deveria ser fácil, mas não é - ou com sua implementação - como a performance. Examine um código ou projeto dessa natureza. Quais são seus problemas, e quais são seus comprometimentos. O que você gostaria de realizar que agora está sendo difícil? Então, cheque uma referência de padrão do projeto. Procure por padrões que correspondam aos problemas com os quais você está envolvido.

A referência mais utilizada quando o assunto é padrões de projeto é o livro "gang of four":
Design Patterns: Elements of Reusable Object-Oriented Software por Erich Gamma, Richard Helm, Ralph Johnson, e John Vlissides, Addison-Wesley, 1995.

Os padrões de projeto estão muito populares, e novos livros continuam a aparecer.

1.3 Por que você deve se preocupar?

Se você é um projetista talentoso e um programador, ou se você tem um monte de tempo para ganhar experiência, você pode encontrar ou inventar muitos padrões de projeto por conta própria. No entanto, esta não é uma maneira eficiente de utilizar o seu tempo. Um padrão de projeto representa o trabalho de uma outra pessoa que encontrou o mesmo problema, tentou muitas soluções possíveis, selecionou e descreveu uma das melhores. Você deve se aproveitar deste fato.

Padrões de projeto podem parecer abstratos à primeira vista, ou você pode não estar convencido de que eles abordam um problema significante. Você começará a apreciá-los ao passo que você constrói e modifica grandes sistemas, provavelmente no seu trabalho final de curso: o projeto Gizmoball.

2. Padrões de criação

2.1 Padrões Factory (fábricas)

Suponha que você está escrevendo uma classe para representar uma corrida de bicicletas. Uma corrida consiste de muitas bicicletas (entre outros objetos, talvez).

```
class Race {
        Race createRace() {
                Frame frame1 = new Frame();
                Wheel frontWhee11 = new Wheel();
                Wheel rearWhee11 = new Wheel();
                Bicycle bike1 = new Bicycle(frame1, frontWhee11, rearWhee11);
                Frame frame2 = new Frame();
                Wheel frontWhee12 = new Wheel();
                Wheel rearWhee12 = new Wheel();
                Bicycle bike2 = new Bicycle(frame2, frontWhee12, rearWhee12);
                ...
        }
        ...
}
```

Você pode especializar a classe ***Race*** para outras corridas de bicicleta:

```
// corrida francesa
class TourDeFrance extends Race {
        Race createRace() {
```

```
                Frame frame1 = new RacingFrame();
                Wheel frontWhee11 = new Whee1700c();
                Wheel rearWhee11 = new Whee1700c();
                Bicycle bike1 = new Bicycle(frame1, frontWhee11, rearWhee11);
                Frame frame2 = new RacingFrame();
                Wheel frontWhee12 = new Whee1700c();
                Wheel rearWhee12 = new Whee1700c();
                Bicycle bike2 = new Bicycle(frame2, frontWhee12, rearWhee12);
                ...
        }
        ...
}


//corrida na terra
class Cyclocross extends Race {
        Race createRace() {
                Frame frame1 = new MountainFrame();
                Wheel frontWhee11 = new Whee127in();
                Wheel rearWhee11 = new Whee127in();
                Bicycle bike1 = new Bicycle(frame1, frontWhee11, rearWhee11);
                Frame frame2 = new MountainFrame();
                Wheel frontWhee12 = new Whee127in();
                Wheel rearWhee12 = new Whee127in();
                Bicycle bike2 = new Bicycle(frame2, frontWhee12, rearWhee12);
                ...
        }
        ...
}
```

Nas subclasses, *createRace* retorna um objeto ***Race*** porque o compilador Java certifica-se que métodos sobrepostos tenham valores de retorno idênticos.

Para economia de espaço, os fragmentos de código acima omitem muitos outros métodos relacionados a corridas de bicicleta, alguns dos quais aparecem em todas as classes, enquanto outros apenas em algumas classes.

A repetição do código é tediosa e, em particular, nós não fomos capazes de reutilizar o método *Race.createRace*. (Podemos observar a abstração da criação de um único objeto ***Bicycle*** através de

uma função; utilizaremos esta abstração sem mais discussões, pois ela é óbvia, pelo menos depois de se realizar o curso 6001). Deve haver uma maneira melhor. O padrão de projeto Factory fornece uma resposta.

2.1.1 O Padrão Método Factory

Um método Factory é um método que fabrica objetos de um tipo particular:

```
class Race {
        Frame createFrame() { return new Frame(); }
        Wheel createWheel() { return new Wheel(); }
        Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
                return new Bicycle(frame, front, rear);
        }
        // retorna uma bicicleta completa sem necessitar de nenhum argumento
        Bicycle completeBicycle() {
                Frame frame = createFrame();
                Wheel frontWheel = createWheel();
                Wheel rearWheel = createWheel();
                return createBicycle(frame, frontWheel, rearWheel);
        }
        Race createRace() {
                Bicycle bike1 = completeBicycle();
                Bicycle bike2 = completeBicycle();
                ...
        }
 }
```

Agora subclasses podem reutilizar *createRace* e até mesmo *completeBicycle* sem qualquer alteração:

```
//corrida francesa
class TourDeFrance extends Race {
        Frame createFrame() { return new RacingFrame(); }
        Wheel createWheel() { return new Whee1700c(); }
        Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
                return new RacingBicycle(frame, front, rear);
         }
}
```

```
//corrida na terra
class Cyclocross extends Race {
        Frame createFrame() { return new MountainFrame(); }
        Wheel createWheel() { return new Whee126inch(); }
        Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
                return new RacingBicycle(frame, front, rear);
        }
 }
```

Os métodos de criação são denominados métodos Factory.

2.1.2 O Padrão Objeto Factory

Se existem muitos objetos para construir, a inclusão dos métodos Factory em cada classe pode inflar o código tornando-o difícil de se alterar. Subclasses irmãs não podem facilmente compartilhar o mesmo método Factory.

Um objeto Factory é um objeto que encapsula métodos Factory.

```
class BicycleFactory {
        Frame createFrame() { return new Frame(); }
        Wheel createWheel() { return new Wheel(); }
        Bicycle createBicycle(Frame frame, Wheel front, Wheel rear){
                return new Bicycle(frame, front, rear);
        }

        // retorna uma bicicleta completa sem necessitar de nenhum argumento
        Bicycle completeBicycle() {
                Frame frame = createFrame();
                Wheel frontWheel = createWheel();
                Wheel rearWheel = createWheel();
                return createBicycle(frame, frontWheel, rearWheel);
        }
}

class RacingBicycleFactory {
        Frame createFrame() { return new RacingFrame(); }
        Wheel createWheel() { return new Whee1700c(); }
        Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
```

```
            return new RacingBicycle(frame, front, rear);
        }
    }

    class MountainBicycleFactory {
        Frame createFrame() { return new MountainFrame(); }
        Wheel createWheel() { return new Whee126inch(); }
        Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
            return new RacingBicycle(frame, front, rear);
        }
    }
```

Os métodos de ***Race*** utilizam os objetos Factory.

```
    class Race {
        BicycleFactory bfactory;
        //construtor
        Race () {
            bfactory = new BicycleFactory();
        }
        Race createRace() {
            Bicycle bike1 = bfactory.completeBicycle();
            Bicycle bike2 = bfactory.completeBicycle();
            ...
        }
    }

    class TourDeFrance extends Race {
        //construtor
        TourDeFrance() {
            bfactory = new RacingBicycleFactory();
        }
    }
    class Cyclocross extends Race {
        //construtor
        Cyclocross () {
            bfactory = new MountainBicycleFactory();
        }
     }
```

Nesta versão do código, o código de cada tipo de bicicleta ainda é codificado em cada variedade de corrida. Há um método mais flexível que requer uma alteração na forma pela qual os clientes invocam o construtor.

```
class Race {
        BicycleFactory bfactory;
        //construtor
         Race(BicycleFactory bfactory) {
                this.bfactory = bfactory;
        }
        Race createRace() {
                Bicycle bike1 = bfactory.completeBicycle();
                Bicycle bike2 = bfactory.completeBicycle();
                ...
         }
}

class TourDeFrance extends Race {
        //construtor
        TourDeFrance(BicycleFactory bfactory) {
                this.bfactory = bfactory;
        }
}
class Cyclocross extends Race {
        //construtor
        Cyclocross(BicycleFactory bfactory) {
                this.bfactory = bfactory;
        }
}
```

Este é o mecanismo mais flexível de todos. Agora um cliente pode controlar tanto a variedade da corrida quanto a variedade da bicicleta utilizada na corrida, por exemplo, por meio de uma chamada do tipo

```
new TourDeFrance(new TricycleFactory())
```

Uma razão pela qual métodos Factory são necessários é a primeira fraqueza dos construtores Java: construtores Java sempre retornam um objeto do tipo especificado. Eles nunca podem retornar um objeto de um subtipo, mesmo que exista uma coerência de tipos (seja com relação ao mecanismo de subtipagem do Java como com relação ao verdadeiro comportamento da prática de subtipagem, como será descrito na aula 15).

2.1.3 O Padrão Prototype (protótipo)

O padrão Prototype fornece uma outra maneira de se construir objetos de tipos arbitrários. Ao invés de passar um objeto ***BicycleFactory***, um objeto ***Bicycle*** é recebido como argumento. Seu método *clone()* é invocado para criar novos objetos ***Bicycle***; estamos construindo cópias do objeto fornecido.

```
class Bicycle {
        Object clone() { ... }
}
class Frame {
        Object clone() { ... }
}
class Wheel {
        Object clone() { ... }
}
class RacingBicycle {
        Object clone() { ... }
}
class RacingFrame {
        Object clone() { ... }
}
class Whee1700c {
        Object clone() { ... }
}
class MountainBicycle {
        Object clone() { ... }
}
class MountainFrame {
        Object clone() { ... }
}
class Whee126inch {
        Object clone() { ... }
```

```
}
class Race {
        Bicycle bproto;
        //construtor
         Race(Bicycle bproto) {
                this.bproto = bproto;
        }
        Race createRace() {
                Bicycle bike1 = (Bicycle) bproto.clone();
                Bicycle bike2 = (Bicycle) bproto.clone();
        }
}
class TourDeFrance extends Race {
        //construtor
        TourDeFrance(Bicycle bproto) {
                this.bproto = bproto;
        }
}
class Cyclocross extends Race {
        //construtor
        Cyclocross(Bicycle bproto) {
                this.bproto = bproto;
        }
}
```

Efetivamente, cada objeto é, ele próprio, um Factory especializado em construir objetos iguais a si mesmo. Prototypes são utilizados freqüentemente em linguagens dinamicamente tipadas como Smalltalk, e menos freqüentemente utilizadas em linguagens estaticamente tipadas como C++ e Java.

No entanto, esta técnica tem um custo: o código para se criar objetos de uma classe particular deve estar em algum lugar. Métodos Factory colocam o código em métodos do cliente; objetos Factory colocam o código em métodos de um objeto Factory; e Prototypes colocam o código em métodos de clonagem.

2.2 O Padrão Sharing (compartilhamento)

Muitos outros padrões de projeto estão relacionados com a criação de objetos no sentido de que eles afetam os construtores (necessitam da utilização de Factories) e no sentido de que estão relacionados à estruturação ao determinarem padrões de compartilhamento entre vários objetos.

2.2.1 O Padrão Singleton

O padrão Singleton garante que um único objeto de uma classe particular irá existir. Você pode desejar utilizar este padrão na sua classe ***Gym*** do projeto Gym Manager, pois os métodos deste padrão são mais adequados ao gerenciamento de uma única localidade.

Um programa que instancia múltiplas cópias, provavelmente, possui um erro, mas a utilização do padrão Singleton faz com que tais erros sejam inofensivos.

```
class Gym {
        private static Gym theGym;
        //construtor
        private Gym() { ... }
        //método Factory
        public static getGym() {
                if (theGym == null) {
                        theGym = new Gym();
                }
                return theGym;
        }
}
```

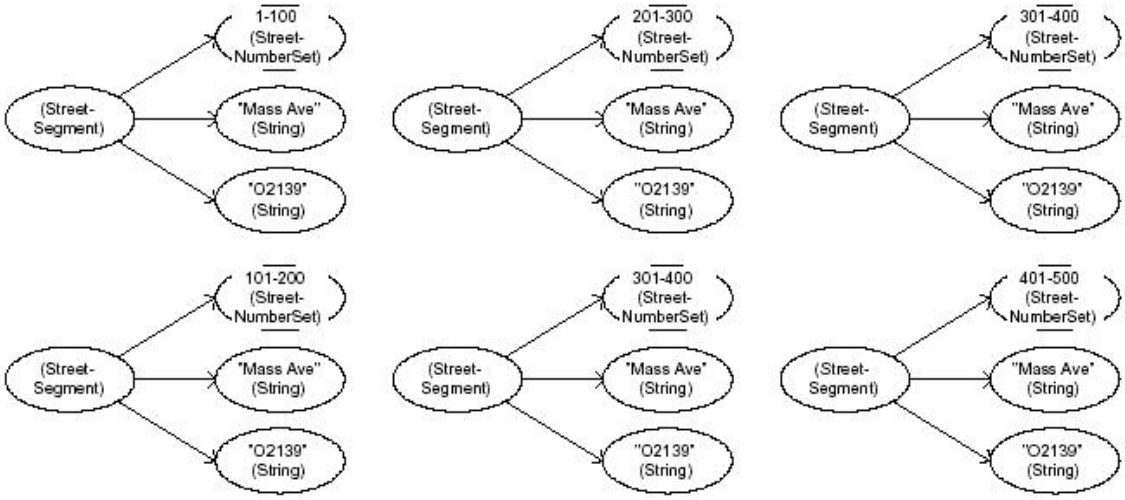
O padrão Singleton também é útil para objetos grandes e caros que não devem ser instanciados múltiplas vezes.

A razão pela qual deve-se utilizar um método Factory, ao invés de um construtor, é a segunda fraqueza dos construtores do Java: construtores Java sempre retornam um novo objeto, nunca um objeto já existente.

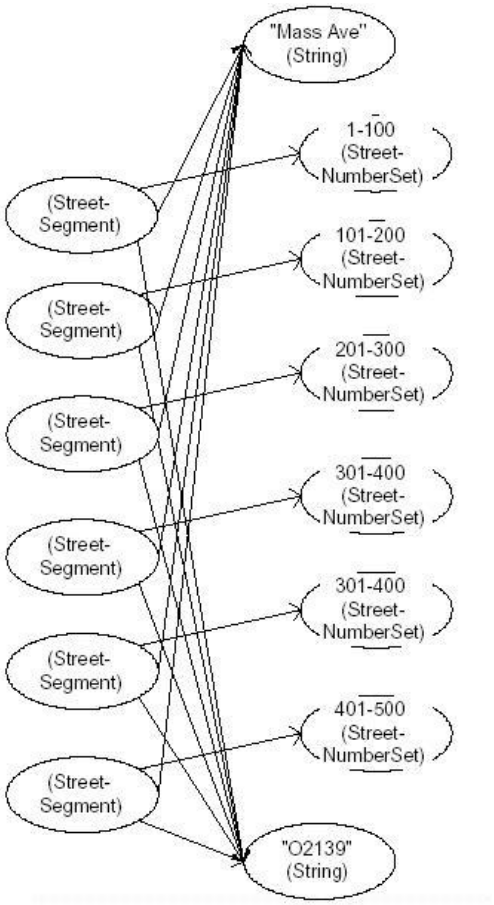
2.2.2 Padrão Interning

O padrão de projeto Interning reutiliza objetos existentes ao invés de criar novos. Se um cliente faz uma requisição por um objeto que é igual a um objeto já existente, então, o objeto já existente é retornado. Esta técnica só funciona para objetos imutáveis.

Como um exemplo, a classe ***MapQuick*** pode representar uma determinada rua através de muitos objetos da classe ***StreetSegments***. Os objetos ***StreetSegments*** irão possuir o mesmo nome de rua e mesmo zipcode (ou CEP). Aqui apresentamos um possível diagrama de objeto (snapshot) para uma parte da rua.



Esta representação está correta (por exemplo, todos os pares de nomes de rua são considerados iguais através do método equals), mas requer uma desnecessária perda de espaço. Uma melhor configuração do sistema seria:



A diferença na utilização de espaço é substancial - é suficiente dizer que é improvável que você possa ler, mesmo que uma pequena base de dados, em um objeto ***MapQuick*** na ausência deste compartilhamento.

O padrão Interning possibilita que objetos que sejam imutáveis sejam reutilizados - ao invés de se criar um novo objeto, uma representação canônica é reutilizada. A representação canônica é a instância do objeto a qual deve-se garantir que será a única instância deste objeto. Pode-se criar uma instância do objeto e utilizá-la apenas para se recuperar a representação canônica, esta representação não canônica, então, será descartada garantindo-se que a representação canônica seja a única.

O padrão Interning requer uma tabela de todos os objetos que já foram criados; se esta tabela contiver qualquer objeto que seja igual a um objeto desejado, este objeto já existente é retornado. Por razões de performance, uma tabela hash, que mapeia conteúdo para objetos, é utilizada (já que a igualdade depende apenas do conteúdo).

Aqui é apresentado um fragmento de código que realiza a operação de Interning em strings que denominam nomes de segmentos:

```
HashMap segnames = new HashMap();
canonicalName(String n) {
        if (segnames.containsKey(n)) {
                return segnames.get(n);
        } else {
                segnames.put(n, n);
                return n;
        }
}
```

As strings são um caso especial, pois a melhor representação para uma seqüência de caracteres (o conteúdo) é a própria string; e terminamos com uma tabela que mapeia strings para strings. Esta estratégia é correta em geral: o código constrói uma representação não canônica, esta representação não canônica é mapeada para a representação canônica, e esta representação é retornada. No entanto, dependendo do quanto de trabalho é realizado pelo construtor, pode ser mais eficiente não construir a representação não canônica caso não seja necessário, caso em que a tabela pode realizar o mapeamento de conteúdo (e não de objeto) para representação canônica. Por exemplo, se estivéssemos realizando a operação de Interning sobre objetos de uma classe denominada ***GeoPoints***, iríamos indexar a tabela utilizando latitude e longitude, ao invés de um objeto ***GeoPoints***.

O código do exemplo mais acima utiliza o mapeamento das strings para as próprias strings, mas não pode utilizar um objeto ***Set*** no lugar de um objeto ***Map***. A razão disto é que a classe ***Set*** não possui uma operação *get*, apenas uma operação *contains*. A operação *contains* utiliza *equals* para realizar comparações. Portanto, mesmo que *myset.contains(mystring)*, isto não significa que *mystring* seja um membro, idêntico, de *myset*, e não há maneira conveniente de se acessar o elemento de myset que corresponde (*equals*) a *mystring*.

A noção de se ter uma única versão de uma determinada string é tão importante que tal conceito está embutido no Java; *String.intern* retorna à versão canônica de uma string.

O texto de Liskov discute o padrão Interning na seção 15.2.1, mas denomina a técnica por 'flyweight' (ou 'peso-mosca'), que é um termo diferente da terminologia padrão da área.

2.2.3 Flyweight

O padrão Flyweight é uma generalização do padrão Interning. (O texto de Liskov na seção 15.2.1, intitulado "Flyweight" discute o padrão Interning que é um caso especial de Flyweight). O padrão Interning é aplicável apenas quando um objeto é completamente imutável. A forma mais geral do padrão Flyweight pode ser usada quando a maior parte (não necessariamente o todo) do objeto é imutável.

Considere o caso do raio (*spoke*) da roda de uma bicicleta.

```
class Wheel {
        ...
        FullSpoke[] spokes;
        ...
}
//Mais adiante iremos definir uma versão simplificada
//desta classe, denominada "Spoke"
class FullSpoke {
        int length;
        int diameter;
        boolean tapered;
        Metal material;
        float weight;
        float threading;
        boolean crimped;
        int location; //local no qual o raio se encaixa no cubo e no aro da roda
}
```

Tipicamente, existem de 32 a 36 raios/roda (até 48 em uma bicicleta do tipo Tandem). No entanto, existem apenas três diferentes variedades de raio por bicicleta: um tipo para a roda da frente e dois tipos para a roda de trás (pois o cubo da roda de trás não é centrado, de forma que raios de diferentes comprimentos são necessários). Preferiríamos alocar apenas três diferentes objetos ***Spoke*** (ou ***FullSpoke***) ao invés de um por raio da bicicleta. Não é aceitável se ter um único objeto ***Spoke*** na classe ***Wheel*** ao invés de um array, não apenas por causa da falta de simetria da roda de trás, mas também porque eu poderia substituir um raio (depois de uma quebra, por exemplo) por outro que tem o mesmo comprimento, mas é diferente em outras características. O padrão Interning não pode ser utilizado, pois os

objetos não são idênticos: eles diferem em seu campo *location*. Em um rali de bicicletas, com 10000 bicicletas, é possível que existam apenas algumas centenas de variedades de raios, mas milhões de instâncias destes raios; seria desastroso alocar milhões de objetos ***Spoke***. Os objetos ***Spoke*** poderiam ser compartilhados entre as diferentes bicicletas (dois amigos com bicicletas idênticas poderiam compartilhar o mesmo raio aro 22 na roda da frente), ainda assim não teríamos um compartilhamento representativo e, em qualquer evento, é mais provável que existam raios semelhantes em uma bicicleta do que entre várias bicicletas.

O primeiro passo para a utilização do padrão Flyweight é separar os estados intrínsecos dos estados extrínsecos. Os estados intrínsecos são mantidos no objeto; os estados extrínsecos são mantidos fora do objeto. Para que o padrão Interning seja possível, os estados intrínsecos devem ser tanto imutáveis quanto similares entre vários objetos.

Criemos uma classe ***Spoke*** menos dependente da propriedade *location* para armazenar os estados intrínsecos:

```
class Spoke {
        int length;
        int diameter;
        boolean tapered;
        Metal material;
        float weight;
        float threading;
        boolean crimped;
}
```

Para se adicionar os estados extrínsecos, não é possível fazer da seguinte forma

```
class InstalledSpokeFull extends Spoke {
        int location;
}
```

pois esta é apenas uma forma simplificada da classe ***FullSpoke***; a classe ***InstalledSpokeFull*** consome a mesma quantidade de memória que ***FullSpoke*** pois tem os mesmos campos. Uma outra possibilidade é

```
class InstalledSpokeWrapper {
        Spoke s;
        int location;
}
```

Este é um exemplo de um wrapper (que iremos aprender a respeito em breve) que economiza uma boa quantidade de espaço, pois os objetos ***Spoke*** podem ser compartilhados por meio de objetos ***InstalledSpokeWrapper***. No entanto, há uma solução que consome ainda menos memória.

Perceba que a propriedade *location* de um dado raio é igual ao valor do índice do objeto ***Spoke*** que representa este raio no array *Wheel.spokes*:

```
class Wheel {
        ...
        Spoke[] spokes;
        ...
}
```

Não há necessidade nenhuma de se armazenar a informação *location* (extrínseca) na classe acima. No entanto, algumas partes do código cliente (em ***Wheel***) devem ser alteradas, pois os métodos de ***FullSpoke*** que utilizavam o campo *location* devem ter acesso a esta informação.

Dada esta versão utilizando a classe ***FullSpoke***:

```
class FullSpoke {
        // tencione o raio girando o parafuso o número
        // especificado de vezes (turns)
        void tighten(int turns){
                ... location...
        }
```

```
}

class Wheel {
        FullSpoke[] spokes;
        //o método deveria ter o nome "true",
        //mas este nome de identificador não é bom
        void align() {
                while (a roda está desalinhada) {
                        ... spokes[i].tighten(numturns) ...
                }
        }
}
```

A versão correspondente utilizando o padrão Flyweight é:

```
class Spoke {
        void tighten(int turns, int location) {
                ... location...
        }
}

class Wheel {
        FullSpoke[] spokes;

        void align() {
                while (a roda está desalinhada) {
                        ... spokes[i].tighten(numturns, i) ...
                }
        }
}
```

Uma referência para uma classe ***Spoke*** desenvolvida com o padrão Interning é muito menos onerosa para o sistema se comparada com uma classe ***Spoke*** desenvolvida sem a utilização desse padrão, pode-se dizer que esta nova versão da classe é mais leve podendo ser considerada peso-mosca (flyweight) em contraposição à sua versão mais pesada; o mesmo

princípio pode ser aplicado à classe ***InstalledSpokeWrapper***, embora seu consumo extra de memória seja no mínimo três vezes maior (e possivelmente mais do que isso).

A mesma técnica funciona se a classe ***FullSpoke*** contiver um campo 'roda' (*wheel*) que se refere à roda na qual o raio sendo representado pela classe está instalado; os métodos da classe ***Wheel*** podem, facilmente, passar o próprio objeto ***Wheel*** para os método da classe ***Spoke***.

Se a classe ***FullSpoke*** também contiver um campo booleano denominado '*broken*' (quebrado) que indicará se o raio está quebrado ou não, como este fato pode ser representado? Trata-se de uma outra informação extrínseca, que não aparece no programa explicitamente, da mesma forma que o fazem as propriedades *location* e *wheel*. Esta informação deve estar explicitamente armazenada na classe ***Wheel***, provavelmente como um array booleano, paralelo ao array de objetos ***Spoke***. Isto é um tanto inadequado - o código está começando a ficar feio - mas é aceitável se a necessidade de economia de espaço for crítica. Se houver muitos campos semelhantes ao campo *broken*, no entanto, o projeto deve ser a reconsiderado.

Lembre-se que o padrão Flyweight deve ser utilizado apenas após que uma análise do sistema determinar que a economia de espaço de memória é crítica para a performance, isto é, a memória é *bottleneck* do programa. Ao introduzir-se tais construtores em um programa, estamos complicando seu código e aumentando a possibilidade de ocorrência de erros. O padrão Interning deve ser colocado em prática apenas em circunstâncias com recursos limitados.