OD 2002

Uma nova abordagem para modelagem de requisitos

Separação Multi-Dimensional de Interesses

Helder da Rocha (helder@acm.org)

argonavis.com.br

Objetivos

1. Discutir as limitações existentes no paradigma orientado a objetos para modelar eficientemente um sistema de software
2. Descrever problemas típicos e possíveis soluções com técnicas OO tradicionais e extensões como AOP e SOP*
3. Apresentar Multi-Dimensional Separation of Concerns (MDSC) - Separação Multi-Dimensional de Interesses - como uma solução para lidar com interesses que surgem durante o ciclo de vida da aplicação
4. Apresentar a ferramenta Hyper/J como forma de utilizar MDSC em Java

* Aspect (AOP) e Subject (SOP) Oriented Programming

Separação de interesses

- A separação de interesses é objetivo essencial do processo de decomposição de um problema
  - Decomposição deve continuar até que cada unidade do problema possa ser compreendida e construída
  - Cada unidade deve lidar com apenas um interesse
- Separação de interesses eficiente promove código de melhor qualidade
  - Maior modularidade
  - Facilita atribuição de responsabilidades entre módulos
  - Promove o reuso
  - Facilita a evolução do software
  - Viabiliza análise do problema dentro de domínios específicos

Tipos ou "dimensões" de interesse

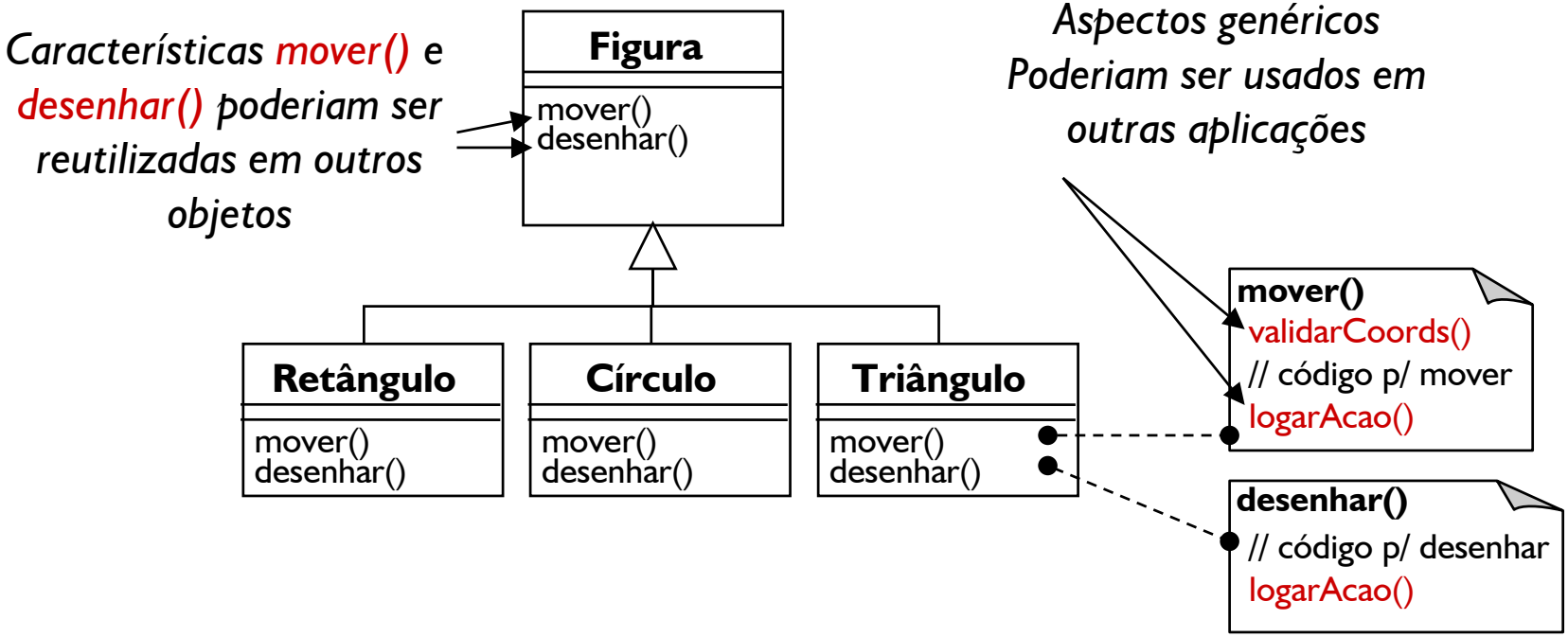
- Interesses podem ser classificados em tipos, ou "dimensões"
  - Definição de tipos de interesse permite a definição de estruturas (módulos) para encapsular interesses
- Exemplos de tipos de interesse são
  - Procedimento, método
  - Classe, objeto
  - Aspecto, estilo, recurso, artefato, caso de uso, etc.
- Linguagens contemporâneas oferecem estruturas que representam apenas uma dimensão de interesse
  - Facilita a representação de interesses diretamente mapeáveis a essas estruturas, mas
  - Causa a fragmentação de outros interesses

Exemplo: tipos de interesse

- Considere os seguintes requerimentos
  - Um software contém diferentes figuras
  - Toda figura sabe se desenhar
  - Toda ação deve ser gravada em log
  - Coordenadas podem ser movidas
  - Movimentos devem ser validos
- Pode-se identificar vários tipos de interesse
  - Classe: Figura, Retângulo, Círculo, Triângulo
  - Características (recursos): desenhar(), mover()
  - Aspectos: logarAcao(), validarCoords()
  - Procedimentos: logarAcao(), validarCoords(), desenhar(), mover()

Exemplo: tipos de interesse

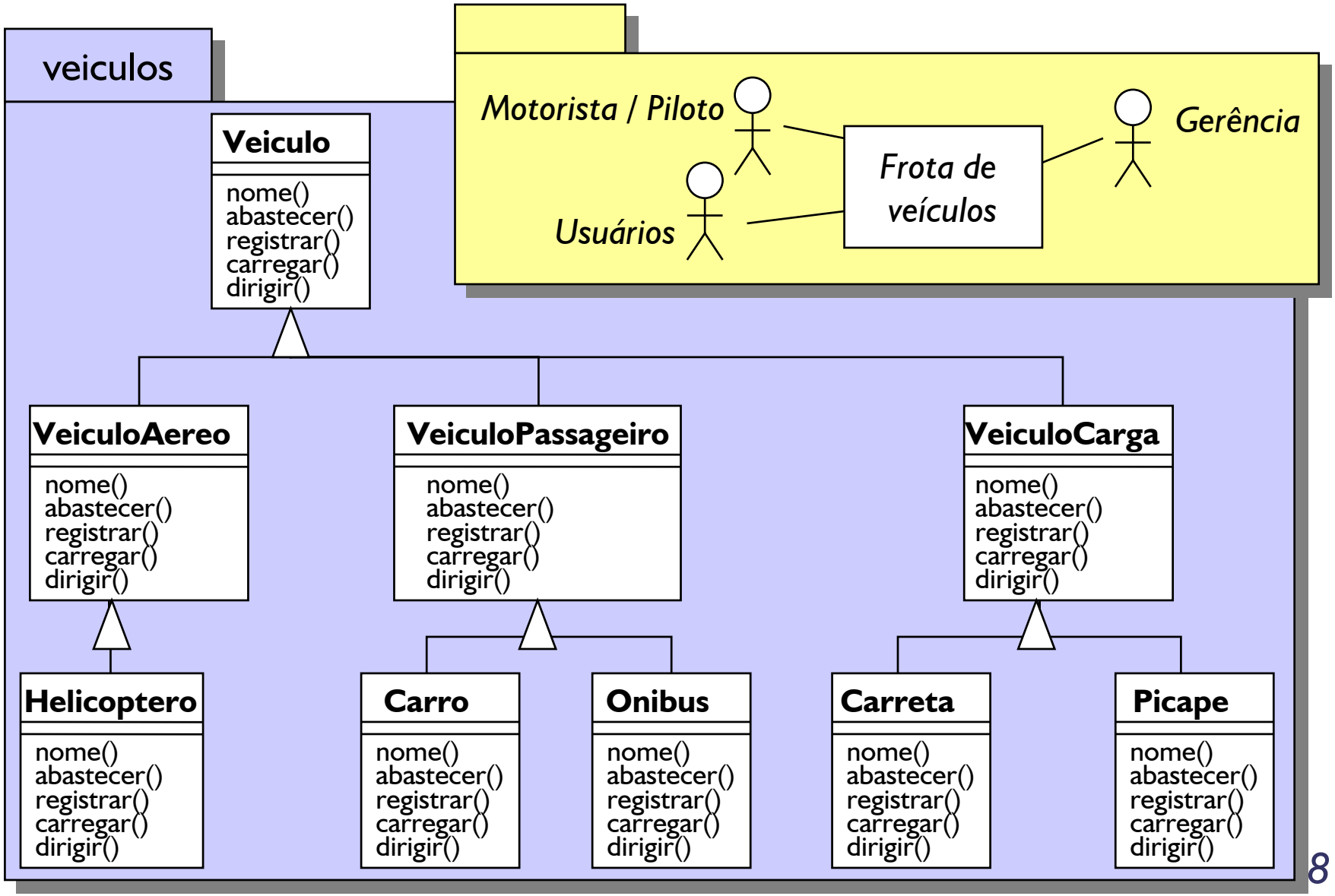
- *Usando uma linguagem orientada a objetos, pode-se representar de forma eficiente e modular as classes e procedimentos (que são sempre encapsulados em métodos)*
- *Outros interesses são implementados como partes de classes e partes ou como composição de procedimentos*



As soluções

- *Em A&DOO, interesses que podem ser representados como classes e métodos são identificados e encapsulados*
  - *Outros interesses precisam ser representados em termos de relacionamentos entre classes e objetos*
- *Em OO, maior separação pode ser alcançada com*
  - *Uso adequado de herança, associação, agregação e composição*
  - *Uso adequado de interfaces com cuidadosa divisão de responsabilidades*
  - *Uso adequado de design patterns*
- *Requer planejamento prévio ou refactoring invasivo*

Aplicação exemplo



Exemplos de código Java

```
package veiculos;
public class VeiculoPassageiro extends Veiculo {
  public void abastecer() {
    super.abastecer() + gasolina;
  }
  public void carregar() {
    while (capacidade >= passageiros) {
      assento[++i].ocupar();
    }
  }
}
```

```
package veiculos;
public class VeiculoCarga
       extends Veiculo {
  public void abastecer() {
    super.abastecer()
                 + diesel;
  }
  ...
}
```

```
package veiculos;
public class Carreta
              extends VeiculoCarga {
  public void abastecer() {...}
  public void carregar() {
    if (cargaMaxima >= carga) {
      aceitar(carga);
    }
  }
  ...
}
```

Problema

- Novos requerimentos decididos quando projeto já estava concluído
  - 1. Registrar abastecimento de todos os veículos
  - 2. Incluir seguro para veículos de passageiro e carga
- (1) é um aspecto
  - Requer inclusão de código adicional ao código existente em cada método abastecer() - pode-se incluir uma instrução antes e/ou depois da execução do método
- (2) é um novo recurso
  - Pode ser implementado como um novo método segurar() nas classes ou superclasses indicadas
- Ambos requerem alterações invasivas no código
  - Código extra espalhado e misturado com código atual

Scattering e Tangling

```
public class VeiculoPassageiro
                extends Veiculo {
  public void abastecer() {
    super.abastecer() + gasolina;
    Logger.log(nome() +
      " abastecido com gasolina");
  } ...
}
```

```
public class VeiculoCarga
                extends Veiculo {
  public void abastecer() {
    super.abastecer() + diesel;
    Logger.log(nome() +
      " abastecido com diesel");
  } ...
}
```

```
public class VeiculoAereo
                extends Veiculo {
  public void abastecer() {
    // código
    Logger.log(nome() +
      " abastecido com " +
      x + " litros");
    // código
  }  ...
}
```

Scattering: *acrescentar* ***um*** *requerimento causa espalhamento do código em várias classes*

Tangling: *código para implementar o requerimento se mistura com lógica do código existente*

Inclusão de um novo recurso

- *Também sujeito a scattering e tangling*
  - *Recurso espalhado por várias classes (o recurso consiste de métodos contidos em várias classes)*
  - *Recurso misturado com outros recursos (não é representado por uma entidade separada)*
  - *Difícil de acrescentar, remover ou manter um recurso*

```
public class VeiculoPassageiro
             extends Veiculo {
  public void abastecer() {...}
  public void carregar() {...}
  public Apolice segurar() {
    Apolice a =
            new Apolice(...);
    // preenche apolice basica
    return a;
  }
  ...
}
```

```
public class VeiculoCarga
             extends Veiculo {
  public void abastecer() {...}
  public Apolice segurar() {
    Apolice a = new Apolice();
    // preenche apolice p/
    // segurar carro e carga
    return a;
  }
  public void carregar() {...}
  ...
}
```

Conclusões

- Decomposição OO deveria simplificar um problema, mas pode complicar!
  - Não é possível representar **um** aspecto como **um** componente do modelo
  - **Um** recurso não pode ser manipulado como **um** conceito independente das classes onde se aplica
- Atuais linguagens OO difucultam a representação de
  - Interesses que atravessam a estrutura de classes (suas partes estão espalhadas por várias classes)
  - Interesses independentes do domínio (que poderiam ser reutilizados em outras aplicações)
- Herança e patterns podem ajudar mas não resolvem o problema completamente

Como reduzir tangling e scattering em OO

1. Criar subclasse que implemente recursos, sobreponha métodos, etc.
   Problema: classes cliente têm que ser alteradas para que saibam como criar a nova subclasse:

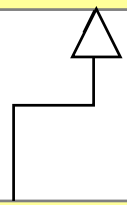
```
VeiculoCarga v =
      // new VeiculoCarga();
      new VeiculoCarga2();
```

2. Design patterns como *factory method* resolvem o problema

```
VeiculoCarga v =
    VeiculoCarga.create();
```

   mas interface precisa ser planejados com antecedência

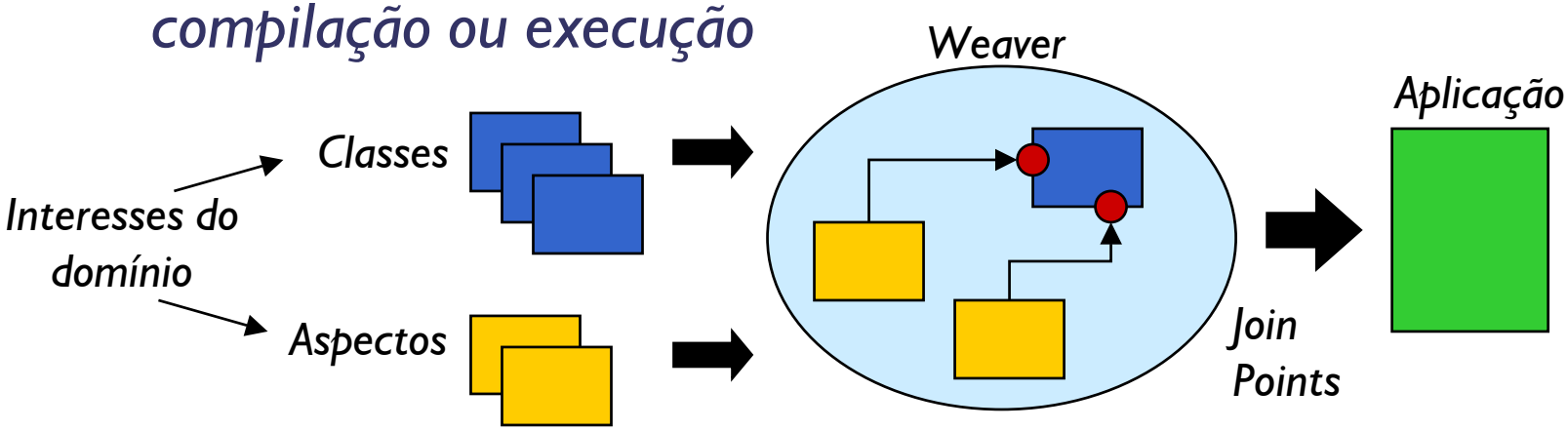
```
package veiculos;
public class VeiculoCarga
        extends Veiculo {
  public void abastecer() {
    super.abastecer()
                  + diesel;
  }
}
```



```
package veiculos.versao2;
public class VeiculoCarga2
  extends veiculos.VeiculoCarga {
  public void abastecer() {
    super.abastecer() + diesel;
    Logger.log(nome() +
    " abastecido com diesel");
  }
  public Apolice segurar() {...}
}
```

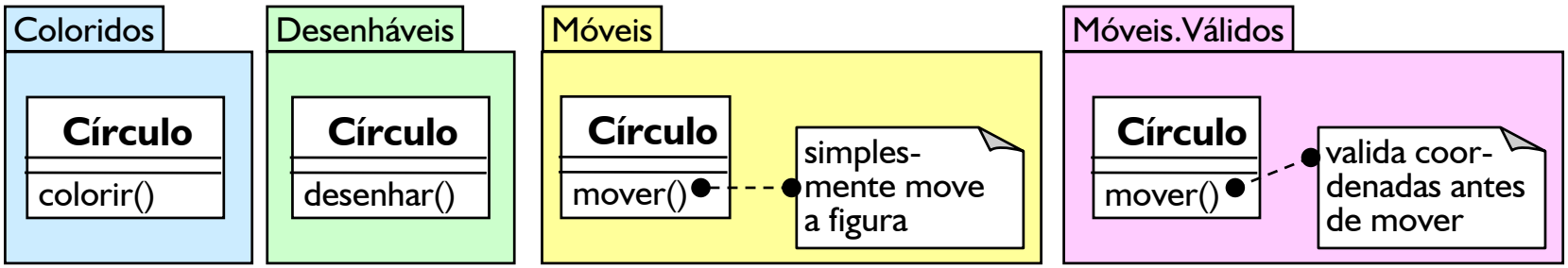
Aspect-Oriented Programming [5]

- AOP divide interesses em dois grupos
  - Componentes: quando o interesse é encapsulável em unidades OO (classe, objeto, método ou procedimento).
  - Aspectos: interesses que não podem ser decompostos como componentes mas podem representar propriedades que interferem no seu funcionamento ou estrutura.
- Aspectos são interesses ortogonais (crosscutting)
  - São costurados ao código principal em tempo de compilação ou execução

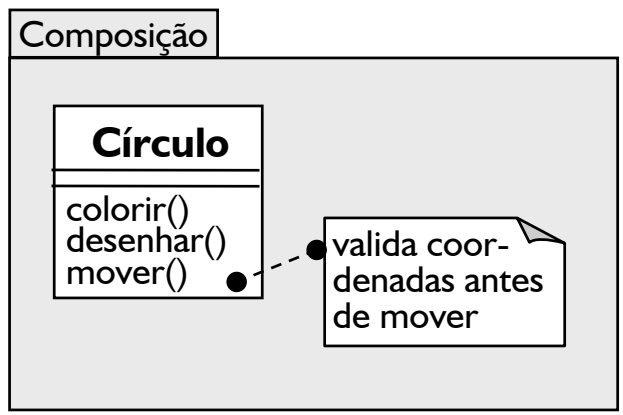


Subject-Oriented Programming [2]

- *Sujeitos (subjects) são abstrações diferentes de um mesmo conceito dentro de um domínio específico*



- *Para implementar a aplicação, sujeitos são compostos gerando um objeto*
  - *Regras definem forma como os sujeitos serão compostos*
- *Pode-se trabalhar com conceitos simples (domain-specific)*

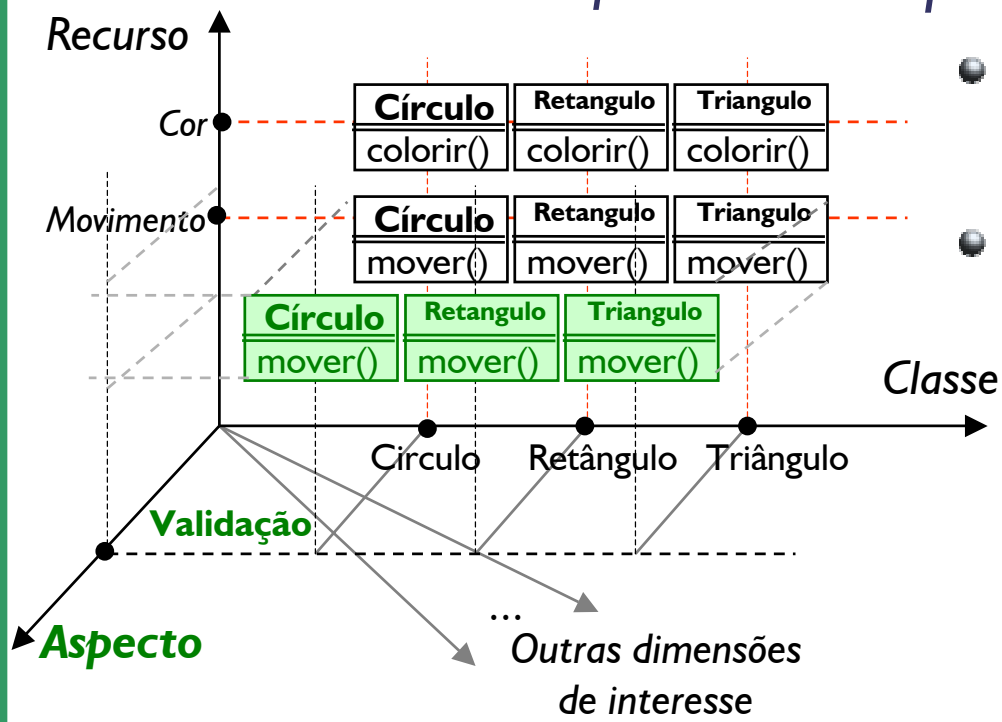


Multi-Dimensional Separation of Concerns [1]

- Generalização do SOP
  - Possui regras de integração mais precisas que SOP
- Permite a identificação e encapsulamento de todos os tipos de interesse existentes em um sistema
  - Interesses podem ser identificados a qualquer momento durante o ciclo de vida do sistema
  - Permite a co-existência de múltiplas decomposições, relacionamentos entre módulos, representação de interesses que se sobrepõem
  - Módulos são similares a sujeitos em SOP. Podem representar estruturas completas (hierarquias de classes) ou componentes individuais (classes, aspectos, etc.)

HyperSpaces

- *Representação de unidades de software em múltiplas dimensões de interesse*
  - *Eixos representam dimensões de interesse*
  - *Pontos nos eixos representam interesses*
  - *Unidades de software são representadas no espaço*



- *Hyperslices*
  - *Encapsulam interesses*
- *Hypermodule*
  - *Conjunto de hyperslices e relacionamentos de integração (informam como hyperslices se relacionam entre si)*

Hyper/J

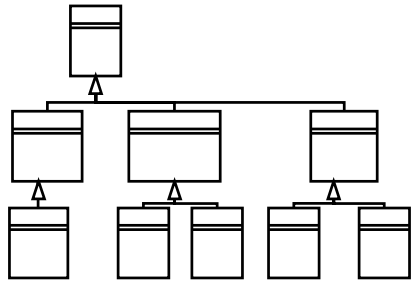
- Ferramenta open-source da IBM
- Ambiente que implementa HyperSpaces em Java
  - Com Hyper/J é possível representar diferentes dimensões de interesse para código Java
  - Unidades suportadas são pacotes Java, interfaces, classes, métodos, atributos e construtores
  - Gera arquivos .class para todos os hypermodules criados
- Hyper/J não interfere no código existente
  - Não é preciso recompilar nada
  - Regras de composição e relacionamentos ficam em arquivos separados

Solução usando Hyper/J - passo-a-passo

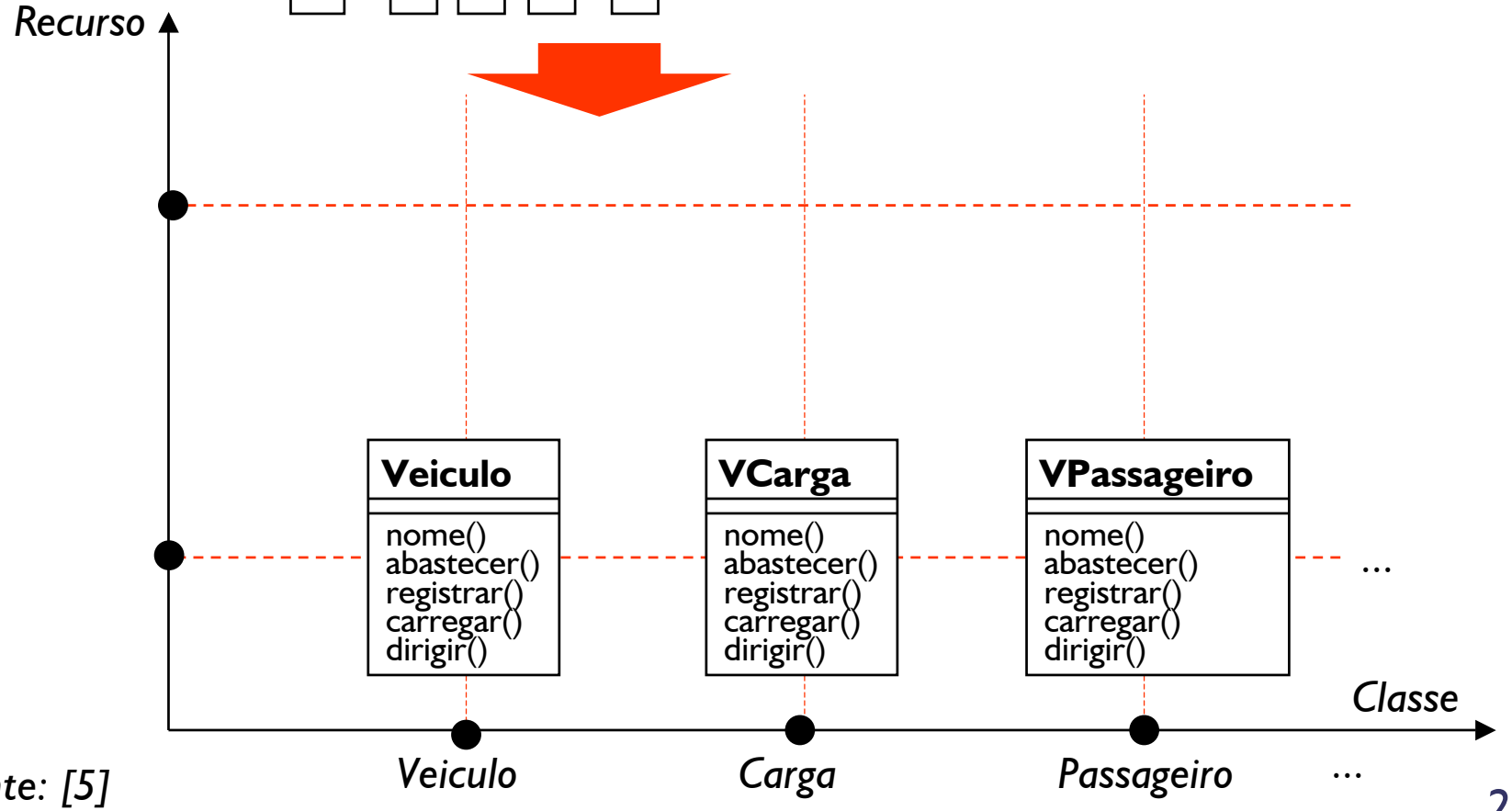
1. Insira código existente no hiperespaço (hyperspace) identificando os interesses importantes
2. Implemente novos recursos em Java
3. Insira novo código no hiperespaço encapsulado como um novo interesse
4. Crie um hipermódulo e acrescente a ele os interesses desejados
5. Indique os relacionamentos de composição entre os interesses no hipermódulo
6. Use relacionamentos para produzir o software composto

Fonte: [5]

1. Incluir código existente no hiperespaço



Identifique interesses de classe



Fonte: [5]

2. Implementar módulo de interesse em Java

- *Recurso segurar() implementado em classe separada*

Classes originais (intocadas)

```
package veiculo;
public class VeiculoPassageiro
              extends Veiculo {
  public void abastecer() {...}
  public void carregar() {...}
  ...
}
```

```
package veiculo;
public class VeiculoCarga
              extends Veiculo {
  public void abastecer() {...}
  public void carregar() {...}
  ...
}
```

Classes novas

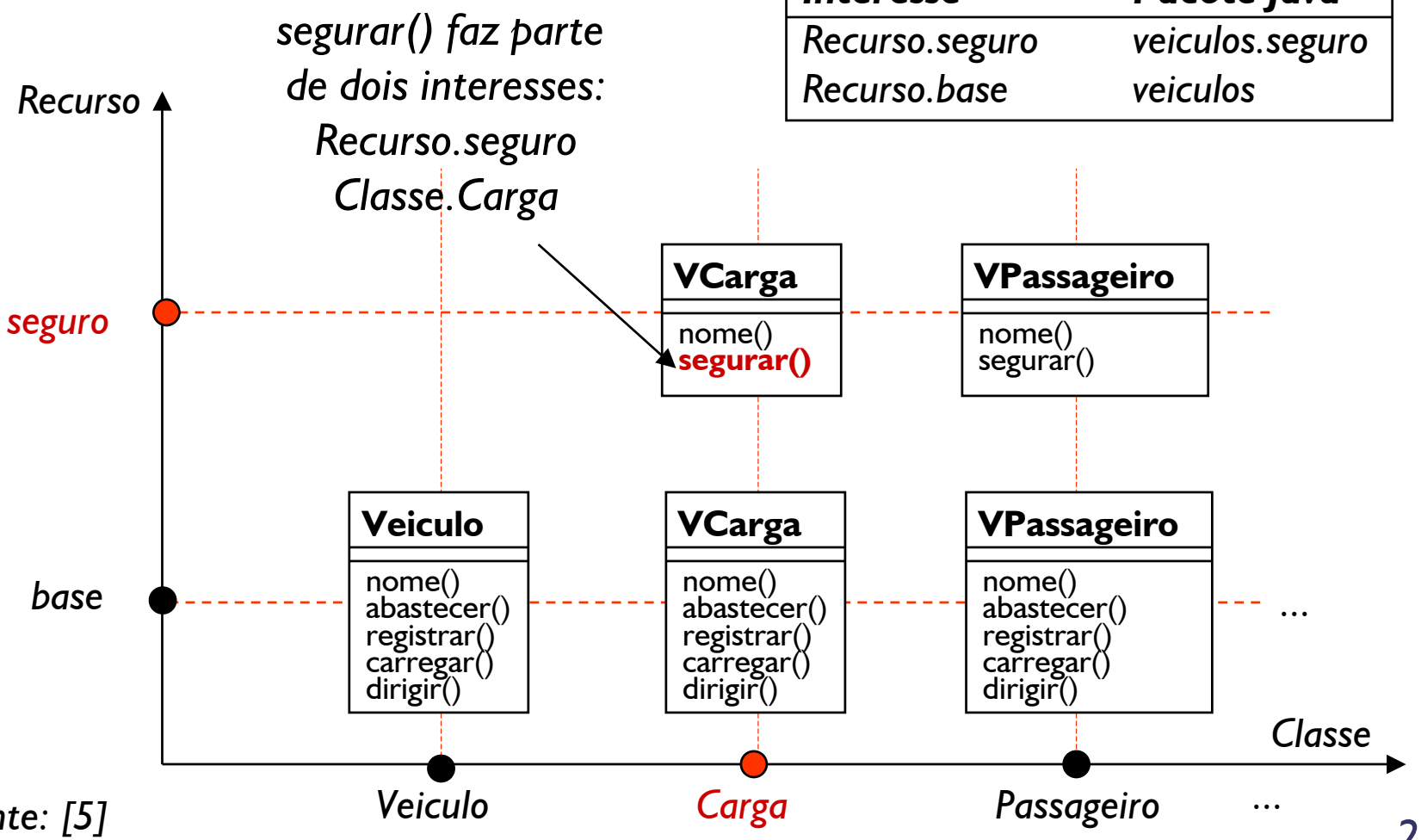
```
package veiculos.seguro;
public abstract class
VeiculoPassageiro extends Veiculo{
  public Apolice segurar() {
    String nome = nome();
    Apolice a = new Apolice();
    // preenche apolice basica
    return a;
  }
  public abstract String nome();
}
```

```
package veiculos.seguro;
public abstract class
VeiculoCarga extends Veiculo {
  public Apolice segurar() {
    String nome = nome();
    Apolice a = new Apolice();
    // apolice p/ carro e carga
    return a;
  }
  public abstract String nome();
}
```

3. Inserir unidades no hiperespaço

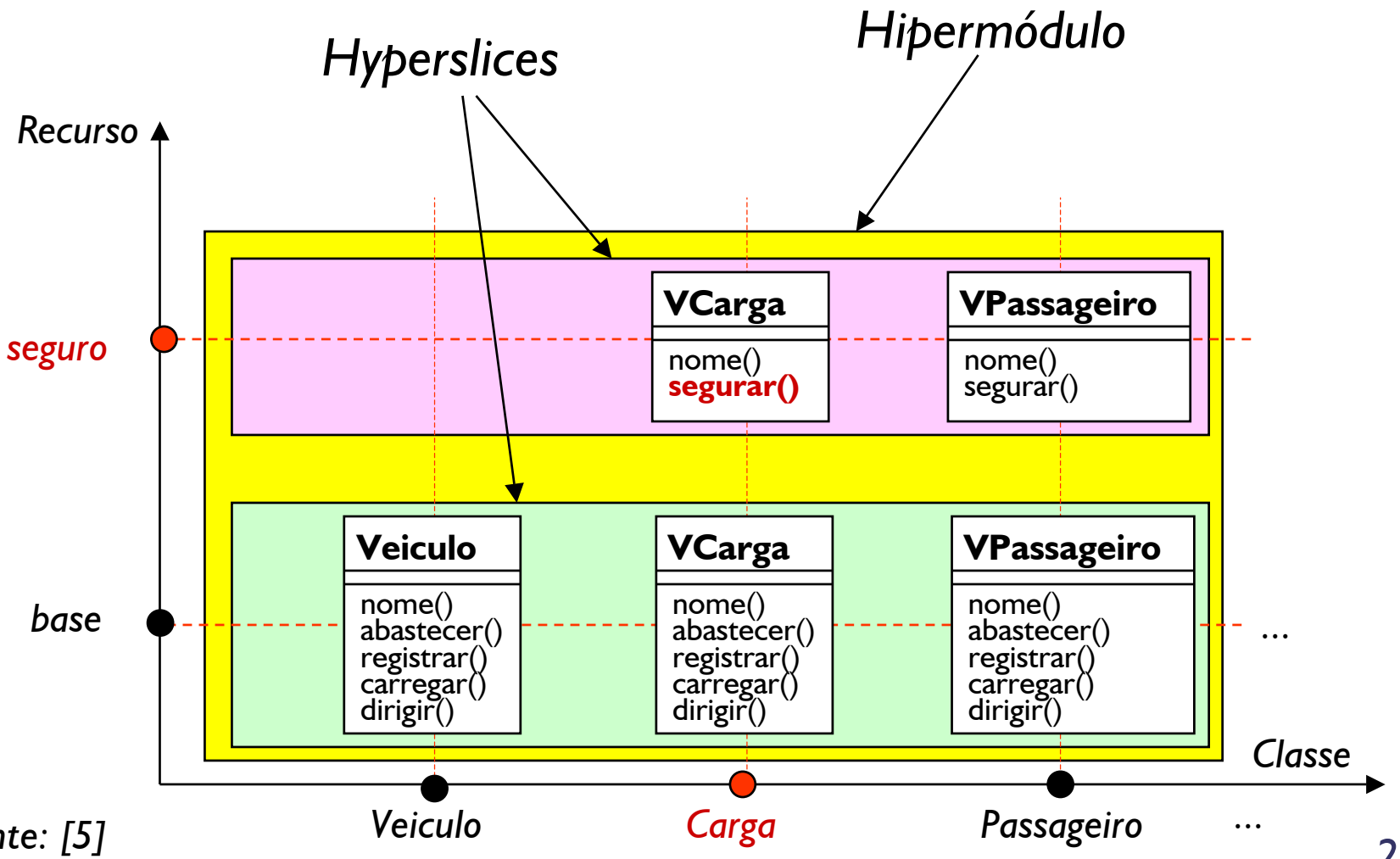
Mapeamento de interesses

Interesse	*Pacote Java*
Recurso.seguro	*veiculos.seguro*
Recurso.base	*veiculos*



Fonte: [5]

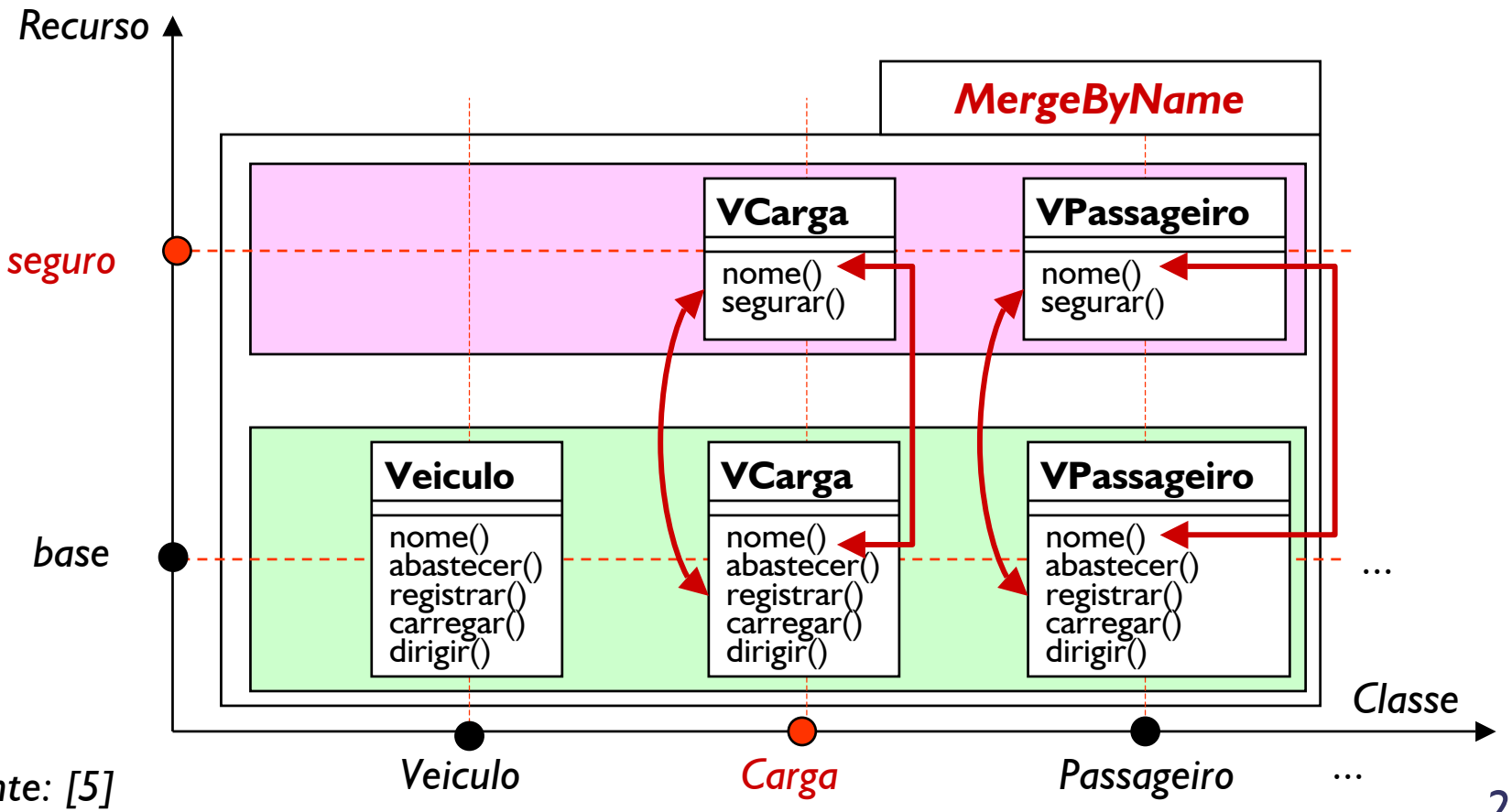
4. Criar hipermódulo



Fonte: [5]

5. Criar relacionamentos de composição

Relacionamento do tipo MergeByName



Fonte: [5]

Arquivo de especificação dos hipermódulos

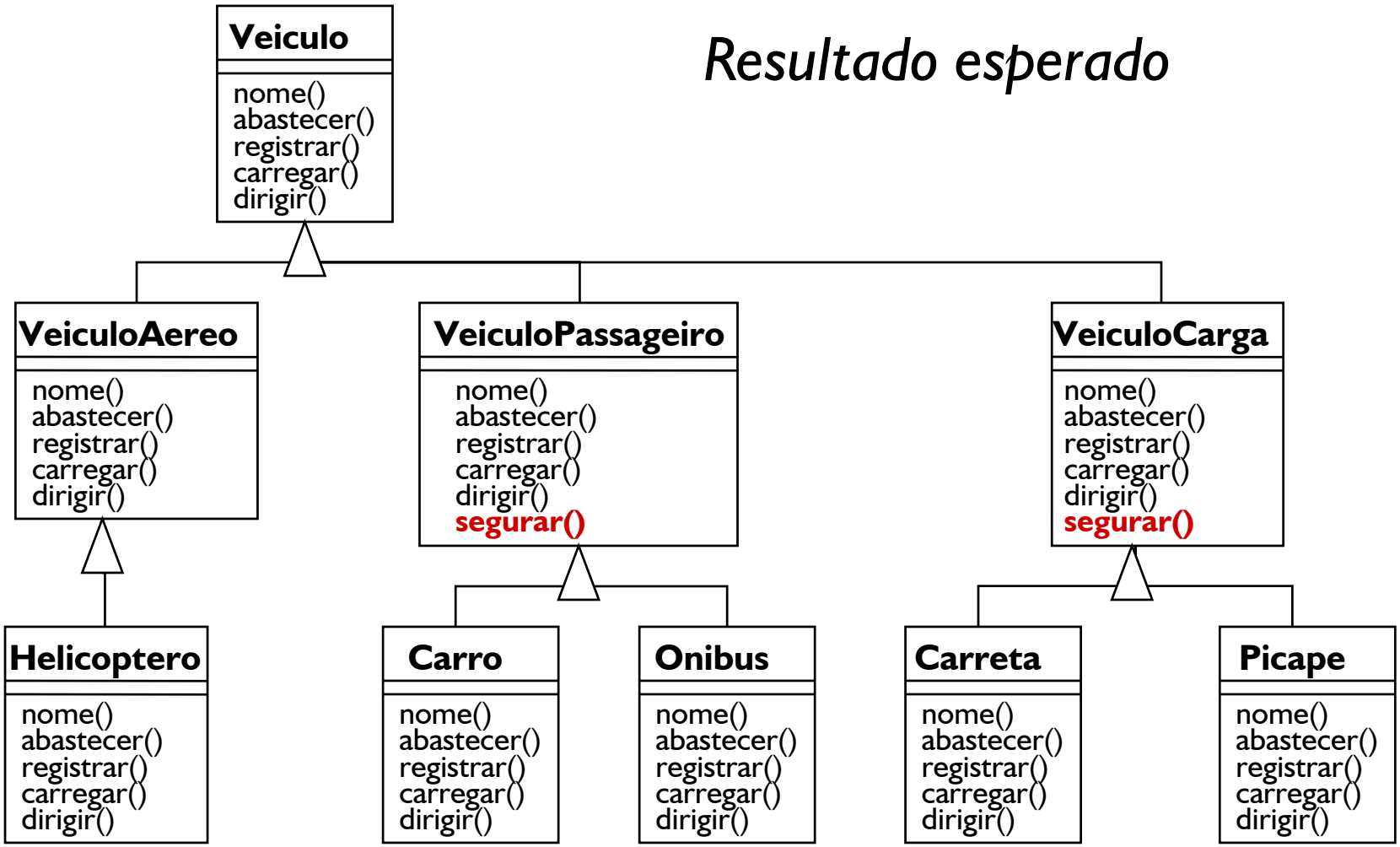
seguro.hjc

```
-hyperspace                    Arquivos Java que serão incluídos
  hyperspace Veiculos
    composable class veiculos.*;
    composable class veiculos.seguro.*;

-concerns                      Mapeamento de interesses
  package veiculos          : Recurso.veiculos
  package veiculos.seguro   : Recurso.seguro

-hypermodules                  Relacionamentos de composição
hypermodule veiculosSeguros
  hyperslices:
    Recurso.veiculos,
    Recurso.seguro;
  relationships:
    mergeByName;
end hypermodule;
```

Essas informações podem ser incluidas em arquivos separados.

6. Usar Hyper/J para produzir código

Resultado esperado

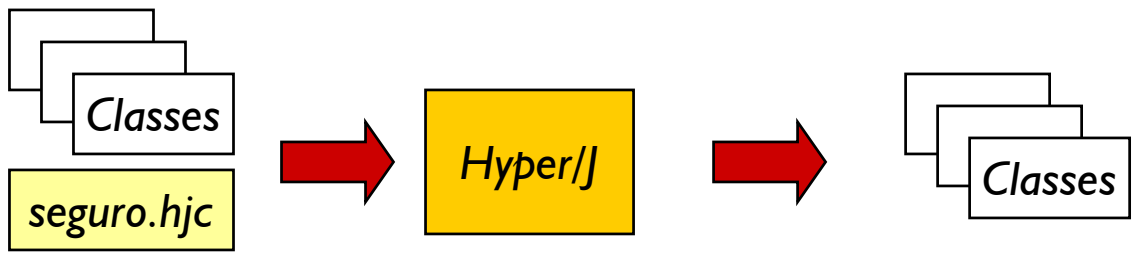


Como usar Hyper/J

- Inclua no seu CLASSPATH
  - Caminho das classes base compiladas
  - $JAVA_HOME/lib/tools.jar
  - $JAVA_HOME/jre/lib/rt.jar, jaws.jar e i18n.jar
  - $HYPERJ_HOME/bin/hyperj.jar
- Execute

```
> java com.ibm.hyperj.hyperj seguro.hjc
```

- Resultados
  - Classes geradas estão em subdiretório veiculosSeguros, com a mesma hierarquia de pacotes anterior.
  - Classes são 100% Java e podem ser usadas em qualquer lugar (sem a necessidade do hyperj.jar)



Conclusões

- *Linguagens que encapsulam apenas interesses de tipos dominantes em suas estruturas, fazem com que outros interesses tenham que ser fragmentados (scattered) e misturados (tangled) no meio do código existente.*
- ***AOP** (Programação Orientada a Aspectos) e **MDSC** (Separação Multi-Dimensional de Interesses) são estratégias que visam simplificar a modelagem e implementação de requerimentos em sistemas de software*
  - ***AOP** permite a captura e encapsulamento de interesses que atravessam estruturas dominantes.*
  - ***MDSC** permite o encapsulamento de quaisquer interesses em estruturas que são reutilizadas através de composição.*

Conclusões

- *HyperSpaces for Java, ou Hyper/J é uma ferramenta open-source da IBM que permite o desenvolvimento usando MDSC em Java*
- *Hyper/J possibilita*
  - *Separação de interesses em hyperslices, que podem ser definidos arbitrariamente*
  - *Composição de hyperslices em hipermódulos, que podem ser usados para gerar código 100% Java*
  - *Definição de regras de composição entre unidades*
  - *Introdução de novos recursos e interesses sem que seja necessário ter acesso ou modificar o código-fonte original*
- *Hyper/J pode ser usado tanto em projetos novos como em projetos existentes*
- *Onde conseguir: http://research.ibm.com/hyperspace*

Referências

[1] Tarr, Ossher, Harrison & Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. ICSE'99. Artigo apresentando MDSC.

[2] Harrison & Ossher. Subject-Oriented Programming - A Critique of Pure Objects. OOPSLA'93. Artigo sobre Subject-Oriented Programing (SOP), de onde evoluiu o MSDC.

[3] Tarr & Ossher. Hyper/J User and Installation Manual. IBM Research. http://www.research.ibm.com/hyperspace. Manual do Hyper/J. O site contém ainda implementação para download, textos, artigos e palestras introdutórias sobre MDSC, HyperSpaces e Hyper/J.

[4] Tarr, Ossher & Sutton Jr. Hyper/J: Multi-Dimensional Separation of Concerns for Java. IBM Research. Apresentação sobre Hyper/J e MDSC.

[5] Kiczales et al. An Overview of AspectJ. ECOOP 2001. http://www.aspectj.org. Artigo introdutório sobre AspectJ.

Obrigado!

helder@acm.org

argonavis.com.br