SUBJECT-ORIENTED PROGRAMMING

(Programação Orientada a Sujeitos)

Emeline B. Regis
Gustavo F. Tondello
Ronnie F. de Brito

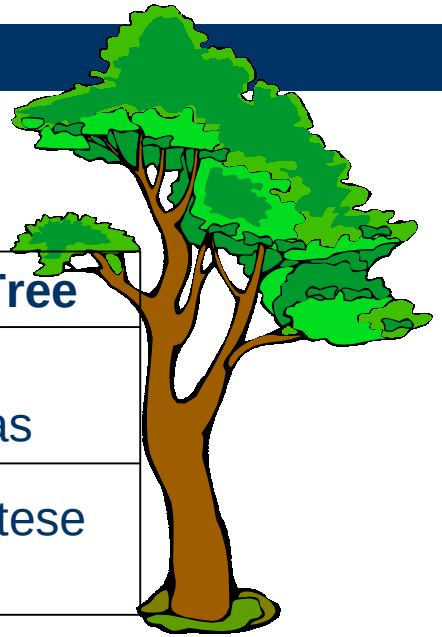
Introdução

- Orientação a Objetos: insuficiente para a construção de grandes conjuntos de aplicações que manipulam os mesmos objetos.
- Por quê?

Introdução

- Cada aplicação tem uma visão diferente do objeto
- Cada um dos atributos ou métodos que fazem parte de um objeto não devem ser vistos por todas as aplicações.
- Vamos ver um exemplo.

Introdução



Nature.Tree

Altura
QtdFolhas

Fotossíntese
Crescer

Bird.Tree

QtdComida
TemNinho

Pousar

Assessor.Tree

ValorDoTerreno

EstimarValor
CalcularTaxa

Woodsman.Tree

PreçoDeVenda
TempoP/Cortar

CalcularLucro

Introdução

- Utilizando Orientação a Objetos, temos que escolher uma das duas saídas:
  - Não encapsular os atributos e métodos das diferentes aplicações em uma única classe. Ou seja, cada aplicação terá sua própria classe que representará o seu objeto.
  - O designer do objeto deverá conhecer todas as possíveis aplicações do mesmo para que possa modelar uma abstração completa que possa atender bem a todas

Introdução

- Nenhuma das duas alternativas parece uma boa solução para gerenciar esta complexidade.
- A idéia da Orientação a Sujeitos foi criada por uma equipe do T.J. Watson Research Center, da IBM, em 1993, e surgiu como uma proposta de modelo mais poderoso que a Orientação a Objetos para o desenvolvimento de conjuntos de aplicações.

Objetivos

- Facilitar o desenvolvimento e a evolução de conjuntos de aplicações cooperativas, ou seja, aplicações que compartilham os mesmos objetos e agem em conjunto na execução de operações

Objetivos

- Possibilidade de desenvolver aplicações separadamente e depois compô-las (juntá-las)
- As aplicações separadas não devem depender umas das outras
- As aplicações compostas podem ter um nível de interação alto ou baixo

Objetivos

- Possibilidade de desenvolvimento de novas aplicações, sem que seja necessário alterar as já existentes, e aproveitando os objetos persistentes já existentes
- Aplicações não esperadas devem ser suportadas
- Dentro de cada aplicação, deve ser possível utilizar as vantagens da herança, do encapsulamento e do polimorfismo

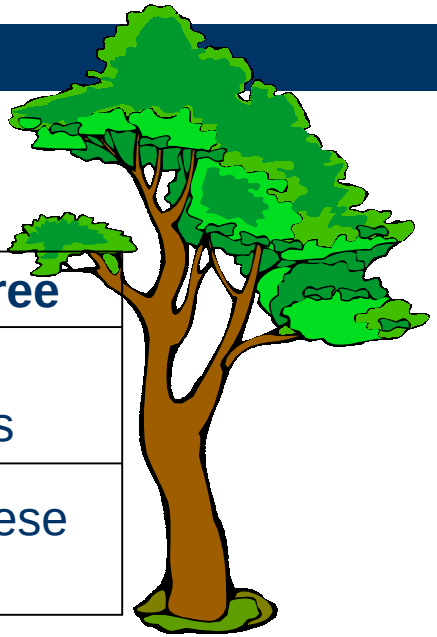
Conceitos

- Sujeito *(Subject)*: percepção do mundo sob algum ponto de vista. Um Sujeito define também estados e comportamentos associados, de acordo com o ponto de vista. Exemplo: **Bird.Tree**, **Assessor.Tree**.
- Ativação de sujeito (*Subject Activation*): uma instância de um sujeito, criada em tempo de execução, com seu estado definido.

Conceitos

- Composição (*Composition*): um conjunto de sujeitos combinados, formando um grupo cooperativo. Exemplo: podemos combinar o sujeito **Bird.Tree** com o sujeito **Woodsman.Tree** para formar uma Composição **Tree**.
- Regra de Composição (*Composition rule*): regra que especifica como os sujeitos serão compostos, por exemplo, como serão tratados comportamentos iguais de sujeitos diferentes e como compartilhar estados.

Exemplo



Nature.Tree

Altura
QtdFolhas

Fotossíntese
Crescer

Bird.Tree

QtdComida
TemNinho

Pousar

Assessor.Tree

ValorDoTerreno

EstimarValor
CalcularTaxa

Woodsman.Tree

PreçoDeVenda
TempoP/Cortar

CalcularLucro

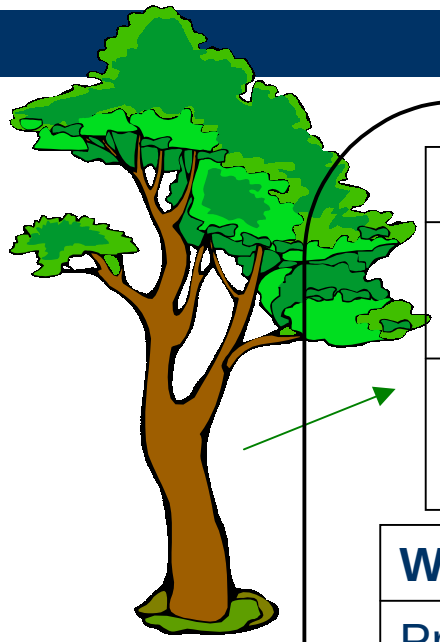
Exemplo

- Utilizando Orientação a Sujeitos, cada uma das visões sobre a árvore seria um Sujeito.
- A junção destes criaria uma Composição, que seria a representação completa da árvore.
- A Regra de Composição irá especificar como os estados e comportamentos de cada um dos Sujeitos separados deverão interagir quando estiverem juntos na Composição.

Exemplo

- Exemplos:
  - A invocação do comportamento **Cortar** da visão do Lenhador irá alterar o estado **Altura** da visão do Pássaro.
  - A construção de um **Ninho** pelo Pássaro pode alterar a decisão do Lenhador de **Cortar** a Árvore ou não.
  - O Assessor pode pedir ao Lenhador que corte a Árvore como parte do pagamento de uma taxa.
  - O Lenhador e o Pássaro podem compartilhar a mesma noção de **Altura** da Árvore.

Exemplo



TREE

Nature.Tree

Altura
QtdFolhas

Fotossíntese
Crescer

Bird.Tree

QtdComida
Altura
TemNinho

Pousar
FazerNinho

Woodsman.Tree

PreçoDeVenda
TempoP/Cortar
Altura

CalcularLucro
Cortar
TemNinho

Assessor.Tree

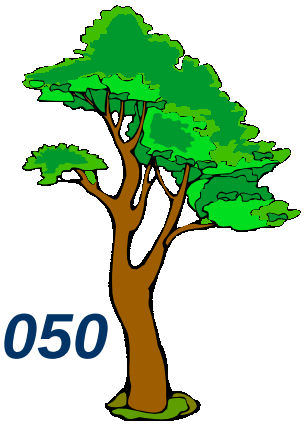
ValorDoTerreno

EstimarValor
CalcularTaxa

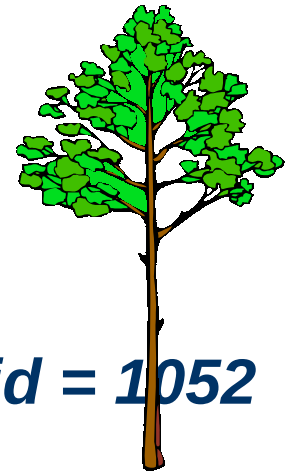
Conceitos

- Identificador de Objeto (*Object-identifier [oid]*): a identificação única utilizada por um objeto para ser reconhecido pelos diferentes Sujeitos.
- Exemplo: Existe um objeto **Tree**. Sua identificação é, por exemplo, “árvore do jardim”. Esta árvore é vista pelo pássaro através do sujeito **Bird.Tree** e pelo lenhador através do sujeito **Woodsman.Tree**. Ambos os sujeitos a conhecem, entretanto, como “árvore do jardim”.

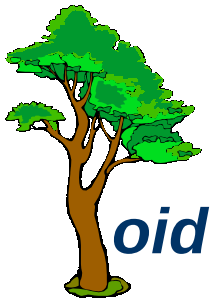
Exemplo



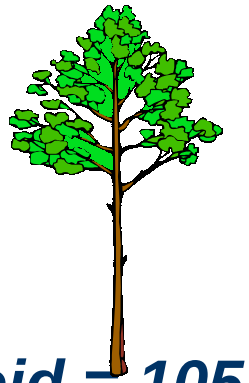
oid = 1050



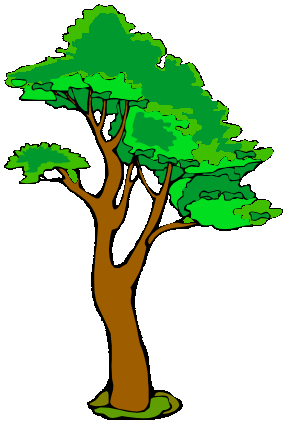
oid = 1052



oid = 1049



oid = 1051



oid = 1055

Ferramentas

- A equipe de pesquisa da IBM que criou o conceito de Programação Orientada a Sujeitos está trabalhando em três ferramentas que permitem a programação utilizando esta abordagem.
- Para C++, foi desenvolvido um suporte, que funciona sobre o compilador IBM VA C++ 4. Ainda não está sendo distribuído comercialmente, mas está disponível para um “pequeno número de usuários sérios”, que devem contactar o autor por e-mail.

Ferramentas

- Para Java, está em fase final de desenvolvimento o Hyper/J, que irá suportar Separação Multi-Dimensional de Interesses, o que inclui a Programação Orientada a Sujeitos.
- Para Smalltalk existe apenas um protótipo, descrito em artigo, sem previsão para desenvolvimento.

Como funciona

- Vamos ver, rapidamente, como funciona o suporte à SOP para C++ desenvolvido pela IBM.
- Cada Sujeito é escrito como um conjunto de Classes em um Namespace, definindo somente o que lhe interessa em sua abstração.
- Além disso, são criados dois outros arquivos, um define as relações entre namespaces e sujeitos, e o outro define as regras de composição.

Como funciona

- No momento da compilação, os Sujeitos serão compilados primeiro, gerando seus Labels.
- Depois, as Regras de Composição serão aplicadas, criando uma Composição que irá funcionar em tempo de execução.
- Todas as chamadas a um Sujeito serão mapeadas para sua Composição. Ou seja, a chamada a um método de um Sujeito irá na verdade chamar a operação na Composição.

Definição do Sujeito

- No namespace que define um Sujeito, devem ser descritas todas as Classes, atributos e métodos que este sujeito tem acesso, ou seja, a visão completa que este tem do mundo. Os métodos podem ser apenas declarados, mas não implementados, em um Sujeito. Neste caso, sua implementação será criada na Composição.

Arquivo de especificação de Sujeito

- No arquivo de especificação de Sujeito são definidos que código pertence a qual sujeito. Isto é necessário porque nem tudo que está em um namespace deve virar sujeito. Além disso, as declarações fora de um namespace também podem virar um sujeito (não podem existir dois ou mais sujeitos sem namespace definido).

Arquivo de especificação de Sujeito

- Este arquivo tem as seguintes partes:
- Nome do sujeito, entre colchetes: `[ExampleSubject]`
- Escopo: define se um Sujeito está definido em um namespace, ou não. Neste caso, o escopo será definido como global, e todas as declarações de classes fora de namespaces serão consideradas como pertencentes a este sujeito.
  `scope=global ou scope=namespace`

Arquivo de especificação de Sujeito

- Quando o escopo é um namespace, o padrão é que este tenha o mesmo nome do Sujeito. Caso isto não ocorra, o nome do namespace pode ser explicitamente declarado na forma:
  `namespace=nome`
- Devem ser definidas quais classes em um Sujeito podem ser compostas:
  `composableclasses=x`
  onde x seria uma lista de classes separada por espaços ou * para todas.

Arquivo de especificação de Sujeito

- Importação: um Sujeito pode importar declarações de outros, na forma `imports=x`
  onde `x` seria o nome de um Sujeito.
- Também poderia ser utilizado `global` como nome do Sujeito, para importar declarações globais. (por enquanto, só esta opção está sendo suportada pelo compilador)

Arquivo de especificação de Sujeito

- Exemplo:

```
[originalPayroll]
scope=global
composableClasses=employee
sales_person manager regular_emp
sales_mgr
[affirmativeAction]
imports=global
composableClasses=employee
```

Subject Labels

- A Composição dos Sujeitos não opera diretamente sobre o código em C++, e sim sobre uma descrição abstrata do programa, chamada de Subject Label (Rótulo do Sujeito).
- Estes labels são gerados no momento da compilação, antes da composição.

Subject Labels

- O Label de um Sujeito irá realizar a descrição do mesmo em termos de Operações, Classes e Mapeamentos, na forma:

```
Subject
    Operations
    Classes
        Instance-Variables
    Mapping
        (Class, Operation)
        Realizations
```

Subject Labels

- **Operações:** as declarações de métodos (assinaturas) são dissociadas de suas classes, e trazidas nesta seção como operações que podem ser executadas pelo Sujeito.
- **Classes:** nesta seção são listadas as Classes que pertencem ao Sujeito, juntamente com as declarações de seus atributos (sem os métodos).

Subject Labels

- **Mapeamentos:** Nesta seção, as Operações que o Sujeito executa são associadas com as suas Classes. Para cada uma, é definido um conjunto de Realizações chamado de Realization Poset.
- A criação deste conjunto serve para permitir que, em uma Composição, a chamada de uma Operação resulte na execução de não um só, mas de diversos métodos, que se encontram em Sujeitos diferentes.

Subject Labels

- Exemplo:

```
Subject: PAYROLL
   Operations: Print()
   Classes: Employee
with Instance Variables: _emplName;
   Mapping:
        Class Employee, Operation
Print() implemented by:
        Realization poset with
realization(s):
             &Employee::Print()
```

Arquivo de Regras de Composição

- Este arquivo irá definir as Regras de Composição que dizem como a Composição deve ocorrer.
- Tipicamente, as regras serão do tipo: *"Componha todas as classes, seus atributos e métodos, quando tiverem nomes iguais"*.
- Também podem haver exceções, do tipo: *"No entanto, componha o método* `Employee.Print` *do Sujeito* Payroll *com o método* `Employee.OutputEmployeeId` *do Sujeito* Personnel".

Arquivo de Regras de Composição

- Lembrando-se que os nomes de Classes e Métodos utilizados nas Regras referem-se aos nomes gerados no Label.
- Geralmente, a sintaxe para escrever uma Regra de Composição é a seguinte: `NomeDaRegra (Resultado, <entrada1, entrada2, ...>)`
- Onde Resultado será o Sujeito Composto que será criado a partir da Composição das entradas.

Arquivo de Regras de Composição

- Portanto, este arquivo será apenas uma lista das regras a serem utilizadas no momento da Composição.
- Exemplo:

```
Equate(SALARY_REPORT, <PAYROLL,
PERSONNEL>);
Correspond(operation
SALARY_REPORT.Print,
<PAYROLL.Print,
PERSONNEL.OutputEmployeeId>,
<PAYROLL.Print>);
```

Regras de Composição

- **Regras de Correspondência:** especificam correspondência entre elementos, mas não como devem ser combinados.
  - `Correspond`: explicitamente indica correspondência entre dois elementos
  - `MatchByName` e `DontMatch`: especifica correspondência (ou não) implícita entre elementos através do nome.
  - `Equate`: especifica correspondência explícita entre elementos e correspondência por nome dentro do escopo do resultado.

Regras de Composição

- **Regras de Combinação:** especificam como combinar os elementos cuja correspondência já foi definida.
  - `Join`: o resultado será a união das funcionalidades.
  - `Replace`: utilizado para substituir elementos de um Sujeito pelos de outro.
  - `ConstrainOrdering`: especifica restrições de ordem dentro de um Realization Poset (ou seja, qual método executar primeiro).

Regras de Composição

- **Regras de Correspondência/Combinação:** definem ao mesmo tempo Correspondência e Combinação entre os elementos.
  - `Merge` (ou `ByNameMerge`): declara correspondência por Nome e combinação por junção (Join).
  - `Override`: declara correspondência por Nome e combinação por substituição (Replace).
  - `NoncorrespondingMerge`: sem correspondência por nome e junção (Join) dentro do escopo.

Exemplo

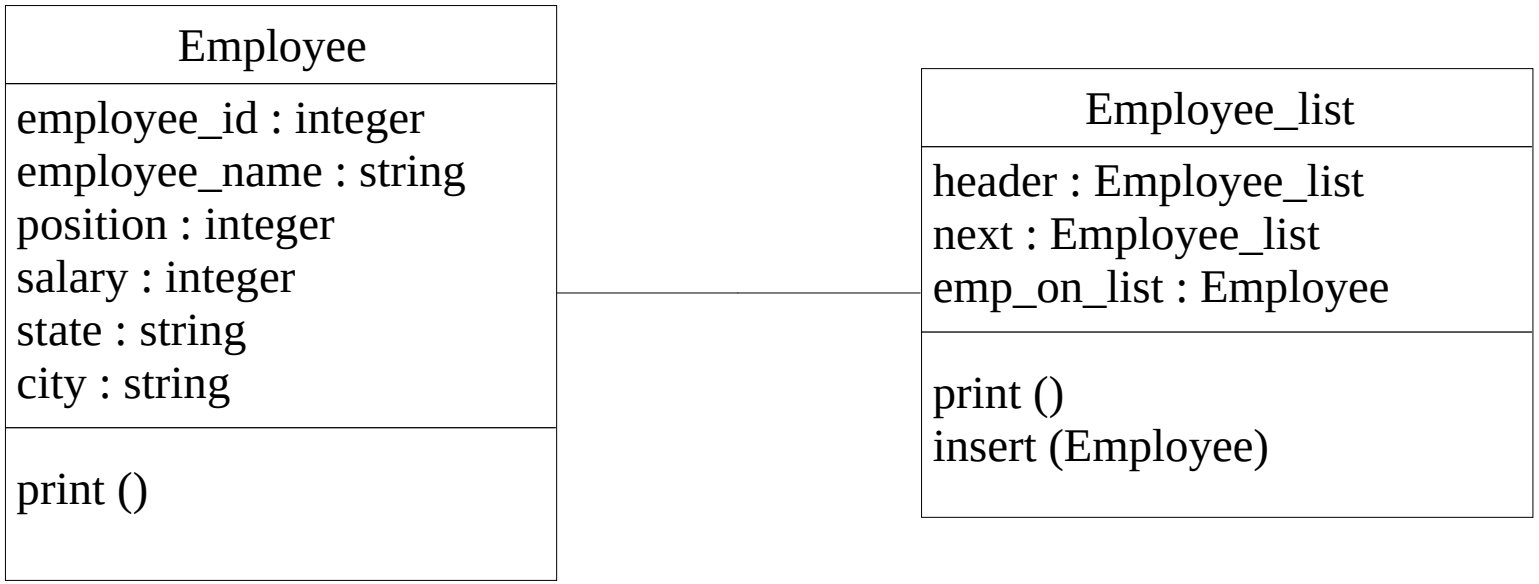
- Iremos mostrar um Exemplo, descrito no site da IBM, de como duas equipes podem desenvolver funções complementares para os mesmos Objetos, de uma maneira independente.
- Neste exemplo, cada equipe desenvolve uma aplicação completa por si só, mas que também pode ser combinada numa aplicação agregada.

Exemplo

- A primeira equipe irá desenvolver um aplicativo de Folha de Pagamento, que irá trabalhar sobre Empregados.
- A segunda equipe irá desenvolver um aplicativo que será um Localizador de Empregados.
- Para simplificar, iremos assumir que as duas equipes compartilharam suas especificações durante o projeto e evitaram criar diferenças nas definições de atributos e operações.

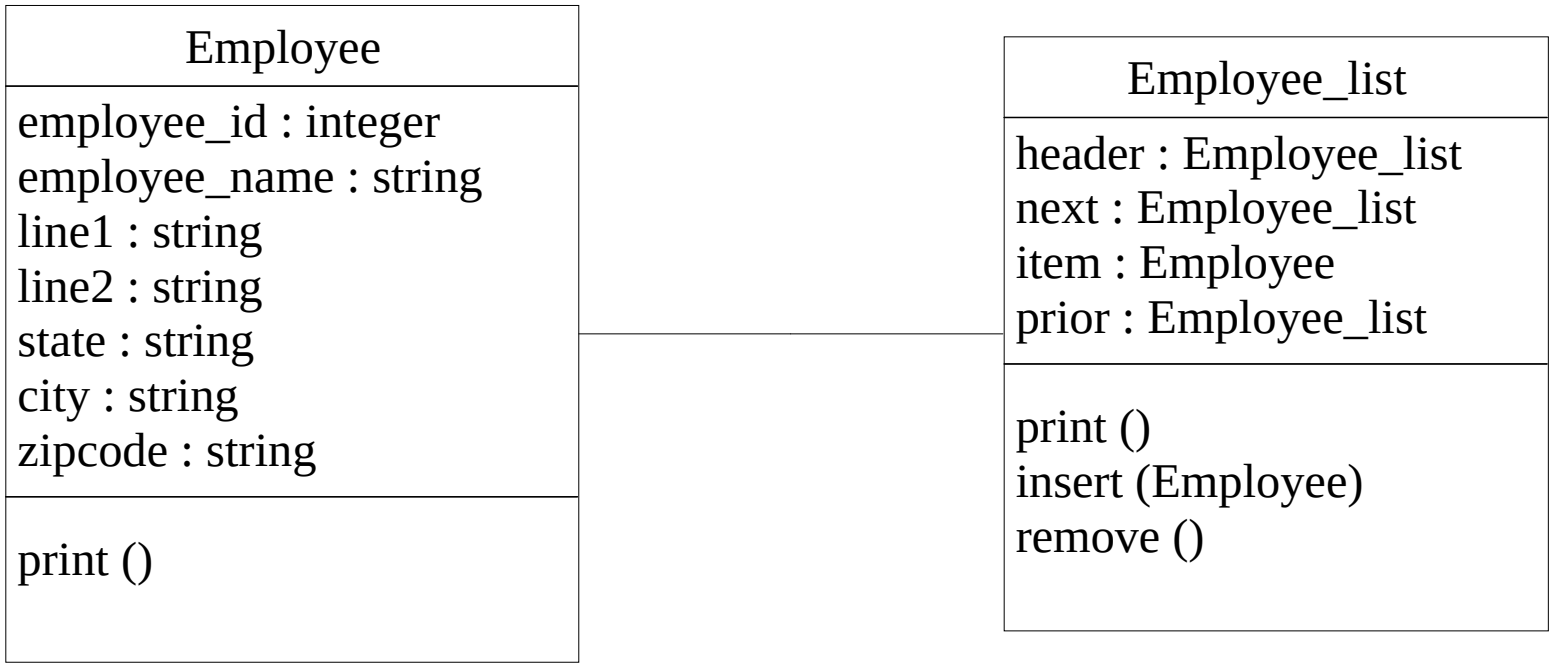
Exemplo

- A primeira equipe desenvolve seu aplicativo *Payroll* com as seguintes Classes:



Exemplo

- A segunda equipe desenvolve *EmployeeLocator* com as Classes:



Exemplo

- Quem compõe as aplicações neste exemplo não são as equipes que desenvolveram as aplicações, e sim uma terceira Equipe que decide combiná-las.
- Analisando as partes, a Equipe percebe que uma simples regra ByName e Merge irá trazer alguns resultados indesejáveis para a aplicação composta.

Exemplo

- Primeiro, como existe um método `Insert` em cada Sujeito, a invocação da operação `Insert` na Composição iria resultar em execução deste método nos dois Sujeitos. Ou seja, o Empregado seria inserido na lista duas vezes.
- O mesmo aconteceria com o método `Print`. A lista seria percorrida duas vezes, imprimindo empregados. Como cada Empregado existe em dobro, seria impresso quatro vezes.

Exemplo

- A melhor solução para esta situação seria utilizar uma regra Override, mantendo apenas uma das implementações.
- Neste caso, como a implementação de Employee_list da aplicação *EmployeeLocator* é mais completa, ela será mantida, sobrepondo (overriding) a implementação de *Payroll*.

Exemplo

- O arquivo de definição de Sujeitos (`composed.sub`) irá apenas definir os dois Sujeitos e declarar todas as suas Classes como *Composable* (podem ser compostas):

```
[payroll]
imports=global
composableClasses=*
[locator]
imports=global
composableClasses=*
```

Exemplo

- O arquivo de Regras de Composição (`composed.rul`) irá trazer a correspondência por nome e a combinação Override:

```
ByNameMerge(composed, <locator, payroll>);
Override(realizationset
composed.insert.employee_list,
<locator.insert.employee_list,
payroll.insert.employee_list>);
Override(realizationset
composed.print.employee_list ,
<locator.print.employee_list,
payroll.print.employee_list >);
```

Exemplo

- A implementação descrita permite conseguir os seguintes objetivos:
  - Nenhuma das duas Equipes teve que mudar o código por causa da outra. Além disso, durante o trabalho, nenhuma equipe se preocupou com as conseqüências que suas implementações teriam para a outra.
  - Qualquer uma das aplicações pode ser distribuída sozinha, ou como parte da aplicação combinada.
  - Nenhuma das equipes teve custo adicional tendo que prever as extensões da outra.

Conclusão - Vantagens

- Possibilidade de desenvolvimento separado de aplicações que irão operar em conjunto sobre Objetos comuns.
- Permite composição de aplicações sem que seja necessário alterar o projeto ou o código de nenhuma delas.
- Facilita a criação de abstrações sobre um domínio, já que cada aplicação pode construir separadamente a sua, que depois poderá ser composta, se necessário.

Conclusão - Dificuldades

- Necessidade de ferramentas que suportem esta abordagem.
- Tecnicamente mais complexa que OO.
- Documentação: como documentar o projeto de uma aplicação orientada a Sujeitos? Com UML não há esta possibilidade...
- Como saber que Sujeitos compor, que métodos e atributos são correspondentes?
  - Métodos com mesmo nome ou parâmetros podem fazer coisas diferentes, por exemplo.

Conclusão - Dificuldades

- A perspectiva é que a SOP venha a permitir, ainda, identificação de correspondências através de interfaces. Mas a realidade por enquanto prevê somente correspondência por nomes ou explicitamente declarada.
- Portanto, a Equipe que irá realizar a Composição deverá conhecer cada detalhe de cada um dos Sujeitos para saber como corresponder e combinar cada um dos atributos e métodos.
  - Como, se não existe ainda padrão de documentação? Analisando código?

Conclusão - Dificuldades

- Como combinar atributos que significam a mesma coisa, mas têm tipo diferente (ex.: id: string e id: integer)?
  - Para este problema existe a possibilidade de criação de um Glue Subject que irá mapear um atributo para o outro. No site da IBM há um exemplo desta implementação.
- Como mapear métodos com assinaturas diferentes?

Conclusão

- A Programação Orientada a Sujeitos traz muitas vantagens com relação a Orientação a Objetos no desenvolvimento de conjuntos de aplicações.
- No entanto, suas promessas de possibilidade de desenvolvimento de aplicações completamente separadas nos pareceram um tanto maiores do que o que realmente se obtém na aplicação desta abordagem.

Conclusão

- Na prática, percebemos que o ideal é que haja um prévio acordo entre as partes que irão desenvolver os Sujeitos, seguindo alguns padrões comuns de nomenclatura, interfaces, semânticas, etc.
- Caso isto não ocorra, ainda assim a Composição é possível, mas o esforço poupado no desenvolvimento das partes pode resultar em um excesso de esforço na compreensão das mesmas para que seja possível criar as Regras de Composição para combinar Sujeitos muito diferentes.

Novas áreas

- A equipe de pesquisas da IBM tem prosseguido em seus trabalhos, e já tem em mente duas novas abordagens, que têm o propósito de estender ainda mais a Orientação a Sujeitos.

Novas áreas

- **Message Central:** um conjunto de ferramentas para auxiliar na integração de componentes através da correspondência entre interfaces, mesmo entre domínios diferentes, como IDLs de CORBA ou definições em XML.

Novas áreas

- **Separação Multi-Dimensional de Interesses** *(Multi-Dimensional Separation of Concerns)***:** uma nova abordagem que irá permitir a separação de interesses (classes, sujeitos, persistência, domínios, etc.) em múltiplo níveis, com integração automática entre eles através de regras de Composição. Isto será conseguido através da implementação de Hyperspaces. A ferramente Hyper/J está sendo desenvolvida para suportar esta abordagem.

Maiores Informações

- Subject-Oriented Programming
  - William Harrison and Harold Ossher, **Subject-Oriented Programming - A Critique of Pure Objects**, Proceedings of 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications, September 1993.
    - Disponível no site da ACM (www.acm.org)
  - www.research.ibm.com/sop

Maiores Informações

- Message Central:
  - www.research.ibm.com/messagecentral
- Multi-Dimensional Separation of Concerns:
  - www.research.ibm.com/hyperspace