
**Departamento**
**Engenharia**
**Informática**

**Modelação**
**Engenharia de Software**
**Sistemas Distribuídos**

# Livro de Receitas

*Framework de aplicações web*

15-04-2009 Versão 1.3.3

# Índice

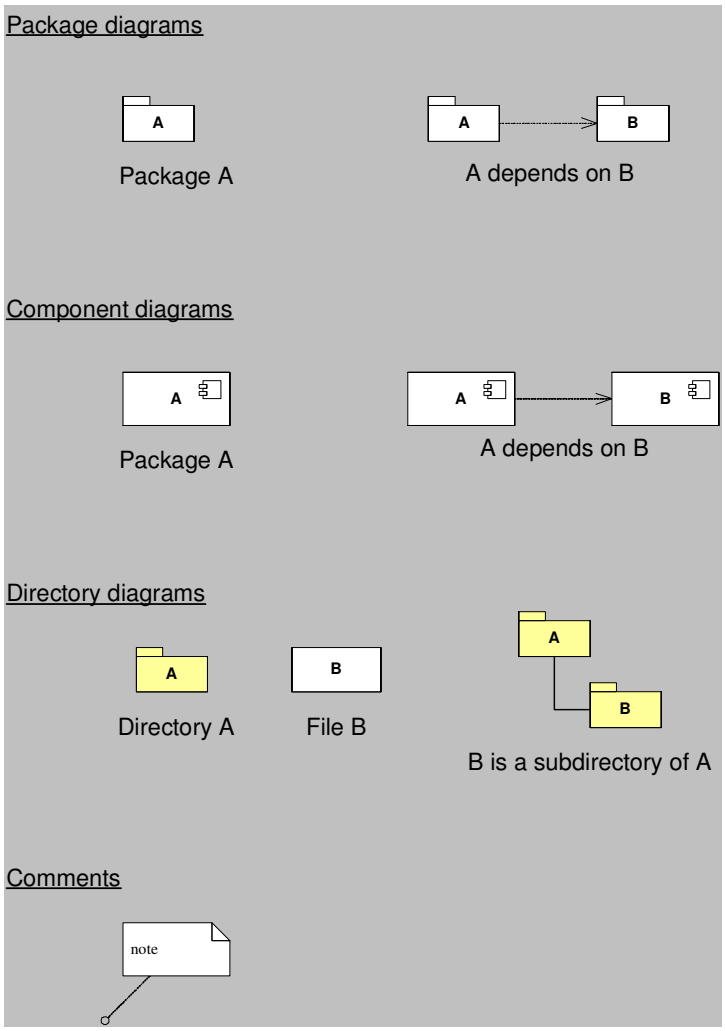
# Nota prévia

Este documento visa guiar a equipa de desenvolvimento no uso da *framework* utilizada no projecto conjunto de Engenharia de Software e Sistemas Distribuídos. Embora a *framework* seja de uso genérico, o livro de receitas que se segue é direccionado tendo em conta as especificidades do projecto a desenvolver.
As *receitas* aqui descritas assumem conhecimento das *frameworks* externas utilizadas (Hibernate, Stripes e JUnit) bem como da API JAXB.

## *Convenções*

| | |
|---|---|
| `$` | Indica a execução de um comando na consola. |
| `Enunciado.pdf` | Indica texto (por exemplo: nomes de classes, ficheiros, caminhos, comandos do sistema operativo) que deve ser introduzido exactamente como apresentado. |
| *`filename`*`.txt` | Indica um elemento variável que é necessário substituir por um valor concreto. |

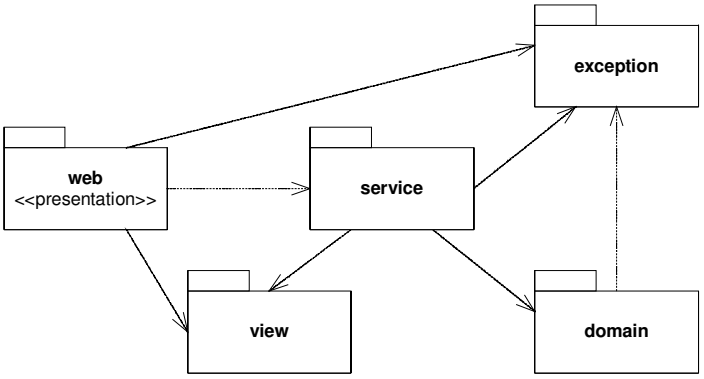
## *Legenda*



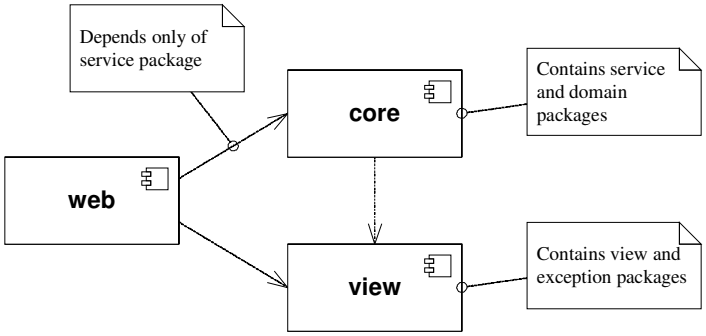
# Breve introdução à framework

A framework é uma biblioteca Java que visa facilitar a construção de aplicações Java em camadas. Cada camada tem uma responsabilidade atribuída. A camada central de uma aplicação é o **domínio**.

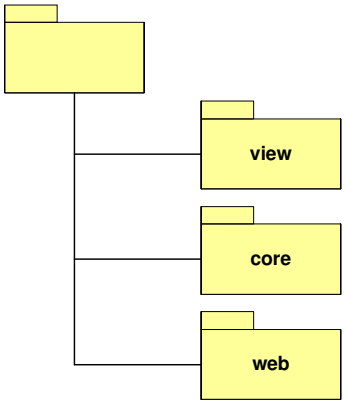
Uma aplicação da framework deverá ter as seguintes camadas (arrumadas em **pacotes**), com as dependências indicadas:



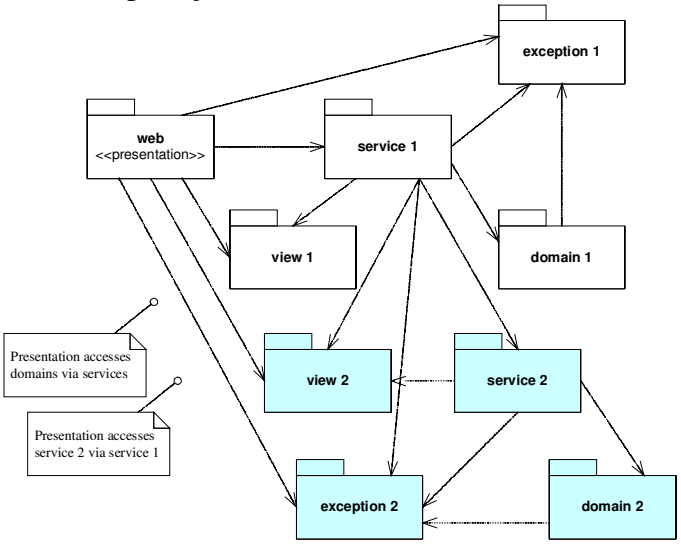

O desenvolvimento de uma aplicação da framework arruma os pacotes em **componentes** que são compilados e distribuídos individualmente:



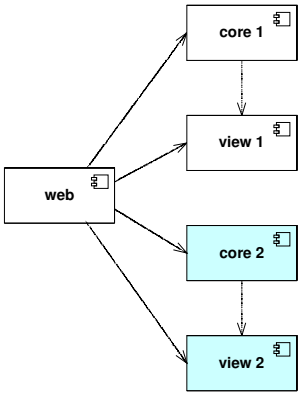

Para facilitar a compilação e distribuição, a cada componente corresponde um **directório** dentro de um sub-projecto:



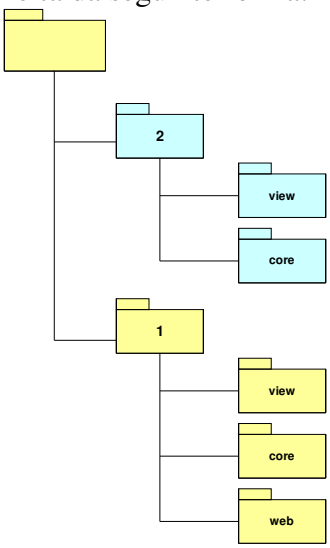
As figuras apresentadas acima mostram uma aplicação da framework com 1 domínio. No entanto, é possível ter uma aplicação Web com **2 domínios**. Neste caso, os **pacotes** são:



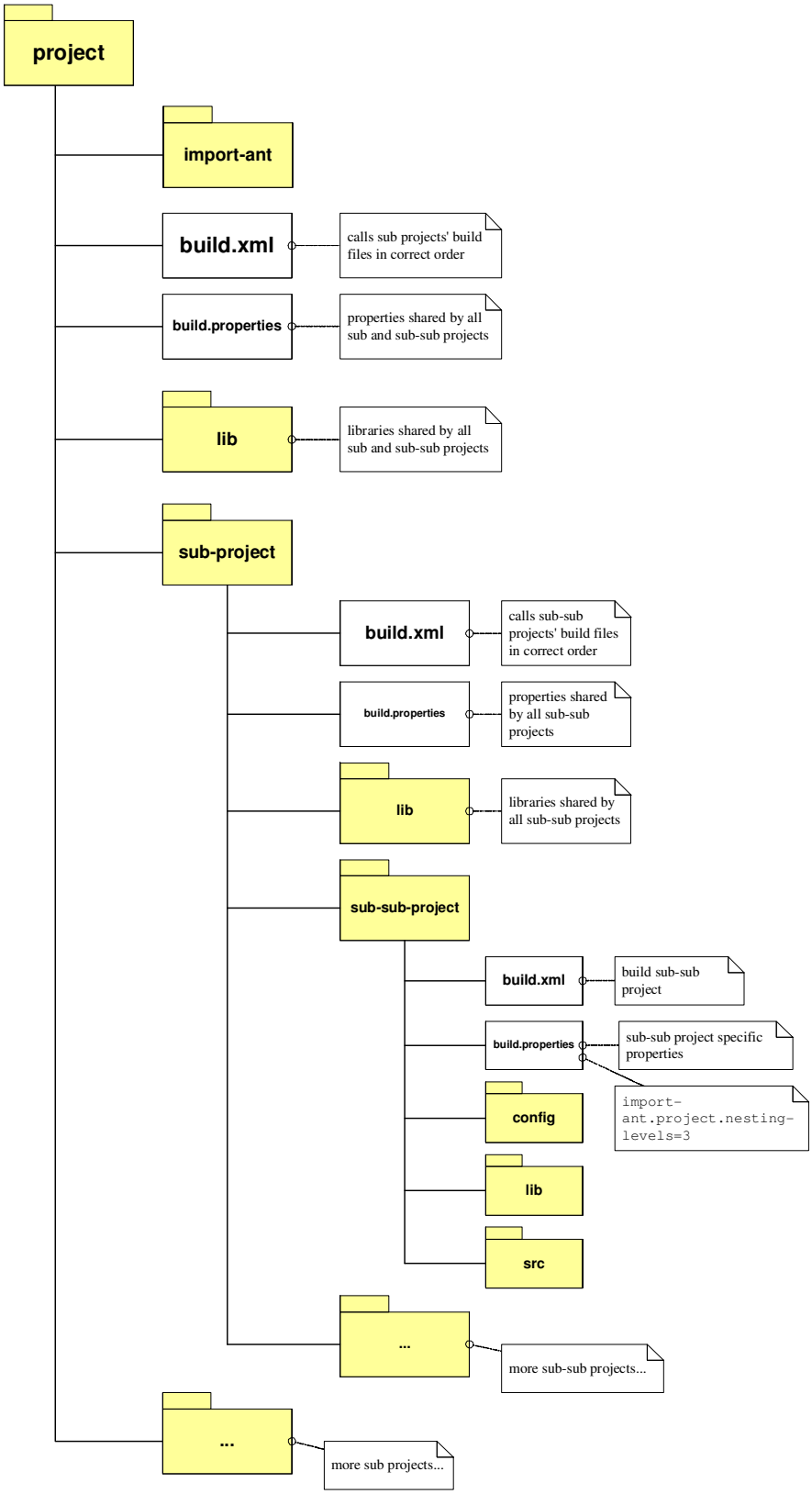

Os **componentes** são os seguintes:



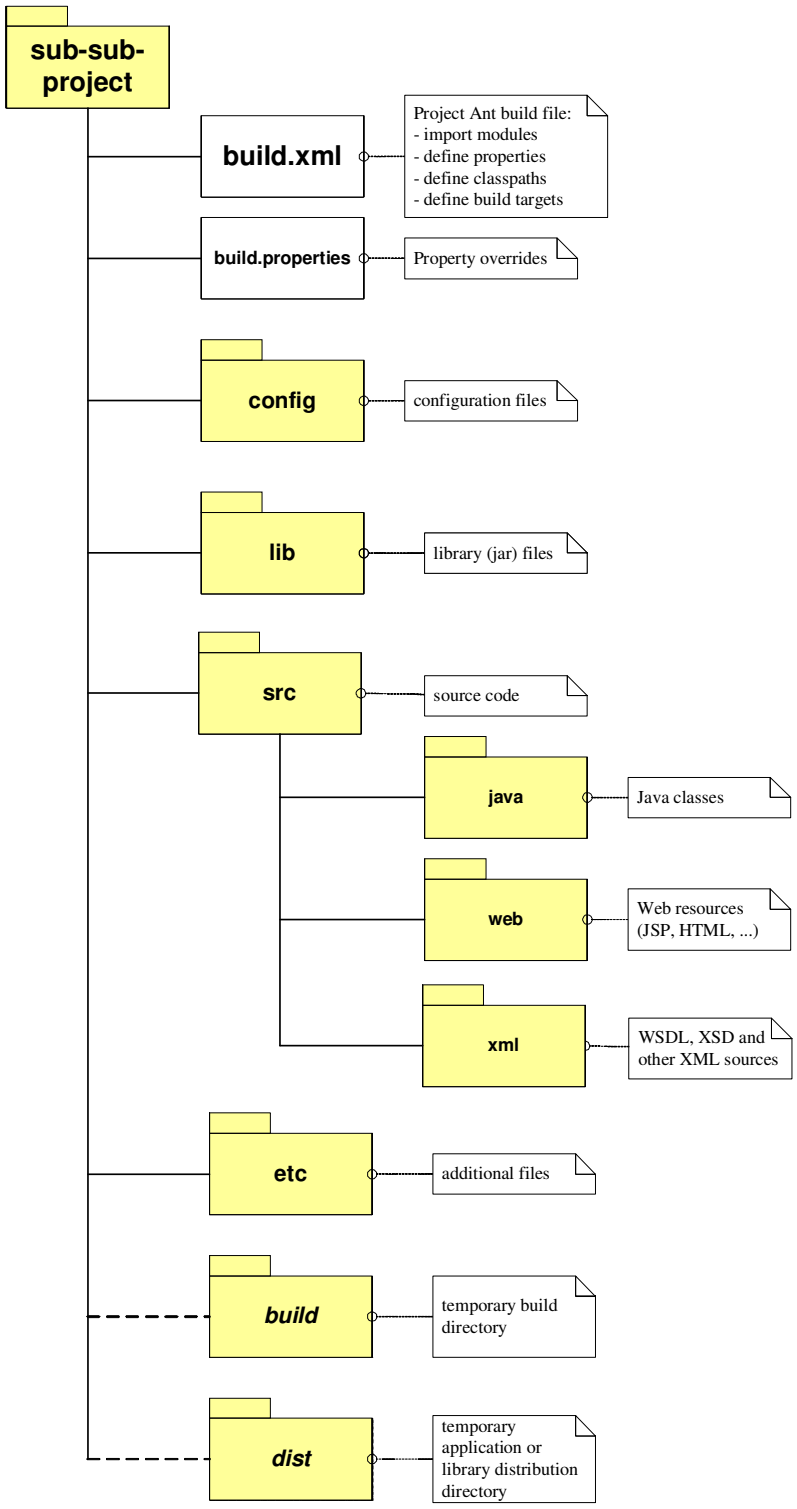

E a arrumação em **directórios** é feita da seguinte forma:



A **estrutura de directórios global** de um projecto da framework é a seguinte:



Cada sub-sub-projecto (designado **componente** na framework) terá uma arrumação semelhante à apresentada de seguida:



# Preparação

## *Iniciar o desenvolvimento de um projecto de software*

### *Estratégia*

Um projecto de software necessita de uma estrutura de directórios pré-definida, que deve ser criada antes de começar o desenvolvimento. A estrutura de directórios integra um directório com o ImportAnt, um directório de bibliotecas necessárias ao projecto e um ficheiro com a definição de propriedades do projecto.

### *Passos básicos*

1. Criar o directório do projecto
   `$ mkdir` *`projecto`*
2. Criar o directório onde serão guardadas as bibliotecas comuns
   `$ mkdir` *`projecto`*`/lib`
3. Descarregar a versão mais recente do import-ant (disponível em http://import-ant.sourceforge.net) e gravar localmente com o nome `ImportAnt-`*`versão`*`.zip`
4. Descomprimir para o directório do projecto
   `$ unzip ImportAnt-`*`versão`*`.zip -d` *`projecto`*
5. Descarregar as bibliotecas comuns (disponíveis em http://disciplinas.ist.utl.pt/leic-sod/2008-2009/software/STEP/)
6. Descomprimir o conteúdo para o directório do projecto
   `$ unzip STEP-lib.zip -d` *`projecto`*
7. Criar um novo ficheiro `build.properties` no directório do projecto e definir as propriedades deste projecto:

```
### definições do ambiente
# usar o servidor aplicacional Tomcat
web-app-env=tomcat.xml
# usar ambiente JWSDP para JAX-B
jax-b-env=jwsdp.xml
# usar ambiente JWSDP para JAX-WS
jax-ws-env=jwsdp.xml
# usar ambiente JWSDP para WS-Registry
ws-registry-env=jwsdp.xml

# credenciais do servidor de desenvolvimento
tomcat.username=username_de_administrador_do_tomcat
tomcat.password=password_de_administrador_do_tomcat
ws-registry.username=username_de_ws-registry
ws-registry.password=password_de_ws-registry
```

8. Iniciar componentes necessários.

## *Variantes*

### *Iniciar sub-projecto*

Um projecto de software pode ser decomposto em sub-projectos. Nesta situação, os sub-projectos definem-se como um projecto, mas dentro do directório do projecto "pai". Propriedades definidas para o projecto "pai" são válidas para os seus sub-projectos.

1. Criar o directório do sub-projecto
   `$ mkdir projecto/sub-projecto`
2. Criar (se necessário) o directório onde serão guardadas as bibliotecas utilizadas
   `$ mkdir projecto/sub-projecto/lib`
3. Criar um novo ficheiro `build.properties` no directório do sub-projecto e definir as propriedades deste sub-projecto. Por exemplo, as configurações de acesso a uma base dados podem fazer sentido neste ponto:

```
database.host=máquina_onde_está_o_servidor_de_BD
database.name=nome_da_base_de_dados_a_utilizar
database.username=username_da_base_de_dados
database.password=password_da_base_de_dados
```

4. Iniciar componentes necessários.

## *Ver*

Iniciar o desenvolvimento de um componente de software

## *Iniciar o desenvolvimento de um componente de software*

### *Estratégia*

Um componente tem uma estrutura de directórios pré-definida, que deve ser criada antes de começar o desenvolvimento. A estrutura de directórios integra um directório com código fonte, um directório de bibliotecas necessárias apenas ao componente (opcional) e um directório de configurações. Adicionalmente, no directório raíz de um componente deve existir ainda um ficheiro com propriedades do componente e um ficheiro de build para automatizar as configurações do componente.

### *Passos básicos*

1. Criar o directório do componente
   `$ mkdir projecto/componente`
2. Criar o directório onde será colocado o código fonte
   `$ mkdir projecto/componente/src`
3. Criar o directório onde será colocado o código fonte Java
   `$ mkdir projecto/componente/src/java`
4. Criar o directório onde será colocado o código fonte XML
   `$ mkdir projecto/componente/src/xml`
5. Criar o directório onde deverão ser colocadas as bibliotecas necessárias apenas a este componente
   `$ mkdir projecto/componente/lib`
6. Criar o directório onde serão guardadas as configurações
   `$ mkdir projecto/componente/config`
7. Criar um novo ficheiro `build.properties` no directório do componente. Caso não seja um componente do projecto de topo, é necessário definir o nível de aninhamento do componente em relação ao topo da hierarquia de projecto (o projecto de topo tem nível de aninhamento 1):

```
import-ant.project.nesting-levels=nível_de_aninhamento
```

8. Criar o ficheiro `build.xml` responsável por definir o processo de construção do componente.

### *Ver*

Definir um novo modelo de domínio

# Desenvolvimento de um modelo de domínio

## *Definir um novo modelo de domínio*

### *Estratégia*

Usando a framework, pretende-se definir o modelo de domínio de uma aplicação orientada a objectos. Todas as classes do modelo de domínio devem herdar da classe `DomainObject` da framework, que define um conjunto de características comuns a todas as aplicações que utilizam a framework. Para facilitar a posterior extensão do modelo de domínio da aplicação, é vantajoso definir uma sub-classe de `DomainObject` que defina características adicionais comuns a todos os objectos de domínio da aplicação.
O pacote domínio é armazenado no componente `core`.

### *Passos básicos*

1. Criar, no pacote *`dominio`*`.domain`, a classe *`Dominio`*`DomainObject`, que herda de `step.framework.DomainObject`.
2. Definir, na nova classe, os atributos e métodos comuns a todos os conceitos do novo domínio.

### *Ver*

Introduzir um novo conceito no modelo de domínio

## *Introduzir um novo conceito no modelo de domínio*

### *Estratégia*

Numa aplicação orientada a objectos, o modelo de domínio representa o conjunto de conceitos que a aplicação necessita de manipular. Introduzir um novo conceito passa por adicionar uma classe ao modelo de domínio e estabelecer relações com outras classes já existentes nesse modelo.

### *Passos básicos*

1. Criar, no pacote `dominio.domain`, uma classe que represente o conceito (deve herdar de `DominioDomainObject`).
2. Definir, na nova classe, os atributos (e, se necessário, os seus métodos acessores `getAtributo`/`setAtributo` e se necessário `resetAtributo`) e métodos que caracterizam o conceito.
3. Definir um construtor que recebe como argumentos todos os parâmetros que caracterizam uma instância da classe e os guarda internamente como atributos.
4. Acrescentar a (pelo menos) uma classe de domínio existente uma associação navegável para esta classe (inclui definir atributo e métodos acessores adequados).
5. Definir o método `init`, sem argumentos, que servirá para estabelecer associações de outros objectos (guardados como atributos no construtor) para este objecto.

### *Variantes*

#### *Definir objecto raiz.*

Numa aplicação orientada a objectos, deve ser possível navegar por todo o modelo de domínio a partir de um objecto (dito *raiz*) e das suas relações. Este conceito deve ser o primeiro a ser definido, visto todos os restantes conceitos estarem directa ou indirectamente relacionados com ele.

1. Aplicar os passos básicos de criação de um novo conceito de domínio (excepto o 4).
2. A classe criada deve implementar o padrão *singleton*, definindo um método estático `getInstance` que devolve a única instância da classe (criando-a e inicializando-a se necessário).

### *Ver*

Definir uma nova excepção de domínio
Relacionar conceitos de domínio
Permitir apagar um objecto de domínio
Introduzir persistência para um conceito de domínio

## *Permitir apagar um objecto de domínio*

### *Estratégia*

Apagar um objecto em Java resume-se a eliminar todas as referências para ele do grafo de objectos. Para centralizar este processo, e sempre que possível, este deve ser concretizado num método da própria classe. O processo passa sempre por navegar para todos os objectos com que o objecto a eliminar se relaciona e eliminar a associação inversa.

### *Passos básicos*

1. Identificar a classe que se pretende poder eliminar.
2. Definir, na classe identificada, o método `destroy`, sem argumentos, que remove associações (bidireccionais) de outros objectos para este.

### *Variantes*

#### *Permitir apagar um objecto com associações unidireccionais navegáveis para si.*

Nas situações em que há uma ou mais associações unidireccionais e navegáveis para o objecto a eliminar, é necessário que cada um dos objectos origem da associação elimine a sua associação com o objecto antes de poder ser invocado o método de eliminação do objecto.

### *Ver*

Introduzir um novo conceito no modelo de domínio
Relacionar conceitos de domínio
Introduzir persistência para um conceito de domínio

## *Definir uma nova excepção de domínio*

### *Estratégia*

Numa aplicação orientada a objectos, o comportamento anómalo é identificado através do mecanismo de excepções. Sempre que seja necessário representar um comportamento anómalo é necessário definir uma nova classe que represente a semântica dessa excepção. Deve ser possível identificar inequivocamente a semântica associada ao uso de uma excepção de domínio.

### *Passos básicos*

1. Criar, no pacote `dominio.exception`, uma classe que represente a nova excepção (deve ser uma subclasse da excepção raiz deste domínio).
2. Redefinir, na nova classe, todos os construtores existentes (sem argumentos, que recebe apenas a excepção que causou esta, que recebe uma mensagem associada, que recebe uma causa e uma mensagem).

### *Variantes*

#### *Definir excepção raiz.*

Todas as excepções de domínio devem fazer parte da mesma hierarquia, de forma a poderem ser tratadas de forma uniforme em camadas acima. Assim deve ser criada uma excepção raíz para cada domínio, da qual todas as excepções desse domínio devem herdar, directa ou indirectamente.

1. Criar, no pacote `dominio.exception` uma classe `domínioException` que represente a nova excepção (deve herdar de `DomainException`).
2. Redefinir, na nova classe, todos os construtores existentes (sem argumentos, que recebe apenas a excepção que causou esta, que recebe uma mensagem associada, que recebe uma causa e uma mensagem).

## *Relacionar conceitos de domínio*

### *Estratégia*

Quando é necessário relacionar dois conceitos de domínio é necessário identificar qual dos conceitos terá a responsabilidade de gerir a relação. O conceito responsável deve definir métodos que permitam alterá-la e navegá-la.

### *Passos básicos*

1. Identificar a classe responsável pela associação.
2. Definir, na classe responsável, um atributo privado que guardará a referência para o objecto relacionado.
3. Definir, na mesma classe, métodos acessores (`get`*`Atributo`*/`set`*`Atributo`* e se necessário `reset`*`Atributo`*) que permitem gerir a relação.

```
public class Responsavel extends DominioDomainObject {
   ...
   private ClasseRelacionada atributo;
   ...
   public ClasseRelacionada getAtributo() {...}
   public void setAtributo (ClasseRelacionada valor) {...}
   public void resetAtributo () {...}
}
```

4. Actualizar, caso necessário, o construtor e o método `init` da classe responsável pela associação.
5. Actualizar, caso se deseje poder apagar a classe alvo da associação, o seu método `destroy` (apenas para associações bidireccionais).

## *Variantes*

### *Relação para-muitos*

Numa relação para-muitos, a uma instância de um conceito estão associadas várias instâncias de outro conceito. Para a representar usa-se uma colecção, tendo o cuidado de definir métodos que a manipulem, nunca permitindo obter uma referência directa.

1. Identificar o tipo de colecção que melhor se adapta à semântica da relação (*mapa*, *conjunto*, *lista*, *vector*, *fila*, etc).
2. Definir, na classe que se deve associar com os vários objectos, um atributo privado que guardará a colecção de objectos relaccionados.
3. Definir, na mesma classe, métodos modificadores que permitam adicionar e remover objectos à associação (`add`*`Elemento`*, `remove`*`Elemento`*), e um método acessor que permita obter uma colecção não modificável (`get`*`Elementos`*) com os objectos alvo da associação.

```
public class Responsavel extends DominioDomainObject {
   ...
   private Coleccao<ClasseRelacionada> elementos;

   ...
   public Coleccao<ClasseRelacionada> getElementos() {
         return Collections.unmodifiableColeccao(elementos);
   }
   public void addElemento (ClasseRelacionada valor)
     throws NomeException {...}
   public void removeElemento (ClasseRelacionada valor)
     throws NomeException {...}
}
```

4. Actualizar, caso necessário, o construtor e o método `init` da classe responsável pela associação.
5. Caso se deseje poder apagar a classe elemento, actualizar o seu método `destroy` para que se remova da colecção (apenas para associações bidireccionais).
6. Caso se deseje poder apagar a classe que guarda a colecção, actualizar o seu método `destroy` para que os elementos da colecção se dissociem da classe que a guarda (apenas para associações bidireccionais).

## *Ver*

Introduzir persistência para um conceito de domínio
Permitir apagar um objecto de domínio

## *Introduzir persistência para um conceito de domínio*

### *Estratégia*

Para que um conceito de domínio possa ser guardado de forma persistente, é necessário acrescentar à classe que o representa, e as classes com que se relaciona, meta-informação que descreva a correspondência entre o modelo orientado a objectos utilizado na aplicação e o modelo relacional utilizado pelo sistema de gestão de bases de dados.

### *Passos básicos*

1. Anotar a classe com `@Entity`.
2. Caso necessário, definir um nome **único** para a tabela que guardará a informação das instâncias da classe (importante para classes com o mesmo nome, mas em pacotes distintos).
3. Alterar o método `init` para invocar o método `save`, herdado de `DomainObject` e responsável por marcar o objecto como persistente no final da unidade de trabalho corrente.
4. Alterar o método `destroy` (caso exista) para invocar o método `delete`, herdado de `DomainObject` e responsável por marcar o objecto como transiente.
5. Caso necessário, anotar atributos da classe de acordo com a sua semântica.
6. Introduzir persistência para classes de domínio navegáveis a partir desta classe.
7. Adicionar, no ficheiro *`projecto/componente`*`/build.xml`, a descrição da nova classe a persistir (no alvo `-replace-hibernate-custom-tokens`)
   `<mapping class="`*`nome.qualificado.da.classe`*`" />`

### *Variantes*

#### *Hierarquia de classes*

Numa situação em que se quer tornar persistente uma hierarquia de classes é necessário definir ainda qual a estratégia a aplicar para a representar no modelo relacional.

1. Aplicar os passos básicos de introdução de persistência para um conceito de domínio.
2. Identificar a estratégia a utilizar: `TABLE_PER_CLASS`, `SINGLE_TABLE` ou `JOINED`.
3. Anotar a super-classe com `@Inheritance(strategy = InheritanceType.`*`estrategia`*`)`.
4. Introduzir persistência para as sub-classes.

### *Classes abstractas*

Uma classe abstracta não é instanciável, pelo que não é possível fazer a correspondência entre uma instância e um registo numa base de dados relacional. No entanto tipicamente define atributos que caracterizam as sub-classes (não abstractas) que se pretendem guardar persistentemente.

1. Anotar a classe com `@MappedSuperclass`.
2. Aplicar os passos básicos de introdução de persistência para um conceito de domínio (excepto o 1 e o 7).

***Ver***

Permitir apagar um objecto de domínio

## *Obter objecto de domínio*

### *Estratégia*

Quando os objectos do modelo de domínio são mantidos em suporte persistente, torna-se necessária a leitura dos objectos a partir desse suporte. A utilização do Hibernate, em conjunto com a existência de um objecto raíz (a partir do qual se pode navegar pelo grafo de objectos) permite simplificar este processo. A única leitura explícita de objectos é a do objecto raíz. Todos os outros objectos são automaticamente lidos e instanciados quando as associações são navegadas.

### *Passos básicos*

1. Identificar a classe que define o objecto raíz.
2. Invocar, na classe `DomainObject`, o método estático `loadSingleton`, passando-lhe como argumento a classe identificada no ponto anterior.

```
...
RootDomainObject root;
root = DomainObject.loadSingleton(RootDomainObject.class);
...
```

### *Ver*

Introduzir um novo conceito no modelo de domínio

# Desenvolvimento da camada fina de serviços

## *Criar um novo serviço*

### *Estratégia*

Um serviço permite disponibilizar funcionalidade do domínio às camadas superiores da aplicação. Introduz como valor acrescentado a possibilidade de satisfazer requisitos não funcionais transversais ao domínio (ex.: *logs*, transacções) e relacionar múltiplas invocações ao domínio para disponibilizar uma funcionalidade que não faz parte do modelo original.

### *Passos básicos*

1. Criar, no pacote *dominio*.service, uma classe que concretiza o serviço (deve herdar de *Dominio*LocalService) e parametrizada com uma subclasse *Dominio*View (no exemplo, *Vista*) ou Void, caso não seja necessário retorno.
2. Definir, na nova classe, os atributos privados necessários para guardar os parâmetros de invocação do serviço.
3. Definir, na nova classe, o constructor que recebe os argumentos do serviço e os guarda nos atributos privados definidos em 2.
4. Redefinir o método *protected* action, sem argumentos, que retorna o tipo parametrizado e pode lançar excepções do tipo *Dominio*Exception. Concretizar este método com as invocações do domínio necessárias.

```
public class ConcreteService extends DominioLocalService<Vista> {
      //atributos
      ...

      public ConcreteService(...) {...}

      protected Vista action() {...}
}
```

5. Verificar que o serviço criado se comporta como esperado através de testes automáticos.

## *Variantes*

### *Criar um serviço raiz.*

Usando a framework, pretende-se definir uma nova camada de serviço. Todos os serviços devem herdar da classe `LocalService` da framework, que define um conjunto de características comuns a todos os serviços das aplicações que utilizam a framework. Para facilitar a posterior extensão do comportamento comum a todos os serviços da aplicação, é vantajoso definir uma sub-classe de `LocalService` que defina características adicionais comuns a todos os serviços da aplicação.

1. Criar, no pacote *`dominio`*`.service`, a classe abstracta *`Dominio`*`Service`, que herda de `step.framework.LocalService`.
2. Definir, na nova classe, os atributos e métodos comuns a todos os serviços desta aplicação.
3. Não redefinir o método `action`.

```
public abstract class DominioService<V> extends LocalService<V> {
      // atributos e métodos comuns
      ...

      public DominioService(...) {...}

}
```

## *Ver*

Invocar um serviço existente
Definir uma nova vista
Testar um serviço

## *Invocar um serviço existente*

### *Estratégia*

Os serviços de uma aplicação definem a funcionalidade disponibilizada para as camadas superiores da aplicação, responsáveis pela sua invocação. Ao invocar um serviço devem ser tratadas as excepções de domínio que possam ser lançadas, bem como as excepções de serviço.

### *Passos básicos*

1. Instanciar o serviço, passando como argumentos do construtor os parâmetros do serviço.

```
ConcreteService service = new ConcreteService(...);
```

2. Executar o método execute() e processar o seu retorno, caso necessário.

```
Vista vista = service.execute();
```

3. Efectuar tratamento de excepções para as excepções que possam ser lançadas.

```
...
Vista vista;
ConcreteService service = new ConcreteService(...);
try {
     vista = service.execute();
} catch (DominioException ex) {
     ...
} catch (ServiceException ex) {
     ...
}
...
```

## *Definir uma nova vista*

### *Estratégia*

Entre camadas de uma aplicação deve haver uma interface bem definida que isole a camada superior dos detalhes de implementação das camadas inferiores. Uma vista é uma representação dos objectos de domínio, devolvida pelos serviços e é utilizada para disponibilizar informação às camadas superiores.
Um forma prática de criar as vistas (e de garantir que são apenas contentores de dados) é efectuar a sua descrição em XML Schema Definition (XSD) e usar a Java API for XML data Binding (JAX-B) para gerar classes Java.
As vistas são definidas no componente `view`.

### *Passos básicos*

1. Criar o ficheiro XSD em `src/xml`. O nome deverá ser *`dominio`*`-view.xsd`
2. Na secção `xsd:annotations`, criar `jaxb:annotations`, nomeadamente `xjc:serializable` e `xjc:superClass`, para indicar que o código a gerar deve implementar a interface `java.io.Serializable` e estender a classe *`Dominio`*`Vista`.
3. Para cada classe de vista:
   1. Definir um `xsd:complexType`. O nome do tipo deverá ser o nome simples da classe, por exemplo, `Cliente`.
   2. Dentro do tipo complexo definir a sequência dos elementos. Cada elemento deverá ter um tipo simples (`xsd:string`, `xsd:int`, `xsd:dateTime`, etc) ou um tipo complexo já definido entretanto.
   3. Definir um `xsd:element` para cada tipo complexo, cujo nome é igual excepto que começa por letras minúsculas (convenção), neste caso, `cliente`.
4. No ficheiro `build.xml` do componente `view`, acrescentar a importação do módulo `${jax-b-env}` e `jax-b.xml`; acrescentar às *classpaths*, `jax-b.jars.path` e a dependência `build-jax-b` ao target build. Para gerar o código fazer, `ant build-jax-b` e consultar o código gerado na directório `build/jax-b/xjc`

### *Variantes*

#### *Listas*

Para definir listas, indicar no `xsd:element` desejado `minOccurs="0"` e `maxOccurs="unbounded"`.

## *Testar um serviço*

### *Estratégia*

Quando se introduz um novo serviço, a classe criada deve ser testada, para garantir que se comporta como previsto. Para o fazer usa-se a *framework* JUnit, definem-se classes de teste para cada serviço a testar, e executa-se regularmente toda a bateria de testes.

### *Passos básicos*

1. Identificar classe de serviço a testar.
2. Para o serviço identificado devem ser identificadas os casos de teste relevantes:
   1. Comportamento correcto (se houver várias combinações relevantes de parâmetros de entrada, devem ser definidos vários casos de teste).
   2. Situações que provocam o lançamento de excepções específicas: parâmetros inválidos, estado interno que impede a operação, etc.
3. Criar[1] uma subclasse de `step.framework.service.ServiceTest` por cada conjunto de casos de teste que partilhe o mesmo estado inicial. O nome da classe de teste deve seguir o formato *`ServiçoEstado`*`Test`, em que *`Serviço`* é o nome do serviço a testar e *`Estado`* descreve o estado inicial dos casos de teste a verificar (*`Estado`* apenas é necessário se o mesmo serviço tiver casos de teste que necessitem de múltiplos estados iniciais).
4. Para cada classe de teste criada, definir o estado inicial da base de dados. É necessário criar[1] o ficheiro *`ServiçoEstado`*`Test.xml`, que conterá a descrição do conteúdo da base de dados em formato XML.
5. Para cada classe de teste criada, criar métodos de teste (anotados com `@Test`) para os casos de teste associados (o estado inicial é o definido no ficheiro criado em 4).
6. Correr toda a bateria de testes, corrigindo erros e falhas detectadas, até todos os testes (deste serviço e de todos os outros serviços) executarem sem erros ou falhas.

---

[1] O directório de código fonte para testes é *`componente`*`/tests/`. Tal como para o directório de código fonte *`componente`*`/src`, o código java deve ser colocado num sub-directório `java`, e ficheiros XML num sub-directório `xml`.

## *Variantes*

### *Definir estado inicial comum aos testes de vários serviços*

É frequente classes de teste de serviços diferentes partilharem o mesmo estado inicial. Se estas classes herdarem de uma classe comum, é possível definir o estado inicial para todas uma única vez.

1. Criar a classe *Estado*`ServiceTest`, que herda de `step.framework.service.ServiceTest`.
2. Definir o estado inicial da base de dados. É necessário criar[1] o ficheiro que conterá a descrição do conteúdo da base de dados em formato XML. Este ficheiro pode ser obtido a partir do estado actual da base de dados através do alvo `dbunit-export-data`.
3. Redefinir, na nova classe, o método `getSetupDataSetName`, para devolver o nome do ficheiro com o estado inicial a utilizar definido no ponto anterior.

### *Utilizar o mesmo estado inicial em testes de serviços diferentes*

Quando se pretende partilhar o estado inicial por testes de vários serviços, pode usar-se uma classe de teste raíz que define o estado inicial, em vez de definir o estado inicial em cada classe de teste.

1. Aplicar os passos básicos 1 e 2 de teste a um serviço.
2. Criar uma subclasse da classe de teste que define o estado inicial comum aos testes de vários serviços. O nome da classe de teste deve seguir o formato *Serviço**Estado*`Test`, em que *Serviço* é o nome do serviço a testar e *Estado* descreve o estado inicial dos casos de teste a verificar (*Estado* apenas é necessário se o mesmo serviço tiver casos de teste que necessitem de múltiplos estados iniciais).
3. Aplicar agora os passos básicos 5 e 6 de teste a um serviço

### *Definir estado esperado por caso de teste*

Uma vez que o sistema assenta em suporte persistente, torna-se relevante verificar o estado do sistema após a execução de cada caso de teste, para garantir que o comportamento do serviço é o esperado ao nível da base de dados.

1. Para cada caso de teste, definir o estado esperado da base de dados. É necessário criar o ficheiro *Serviço**Estado*`Test-result.xml`, semelhante ao que define o estado inicial, mas aplicando as modificações resultantes da execução do caso de teste. Colunas omitidas não serão comparadas.
2. Para cada caso de teste, invocar o método `assertDatabase`, passando como argumento o nome do ficheiro definido no passo 1.

## *Ver*

Invocar um serviço existente
Executar bateria de testes

## *Executar bateria de testes*

### *Estratégia*

Uma bateria de testes só é verdadeiramente útil se for executada regularmente. Para tal deve haver uma forma de automatizar o processo de executar testes.

### *Passos básicos*

1. Acrescentar, na zona de *imports* do ficheiro `build.xml` do componente, a linha:
   ```
   <import file="${import-ant}/dbunit.xml" />
   ```
2. Acrescentar, na zona de alvos do ficheiro `build.xml` do componente, o alvo:
   ```
   <target name="run-tests" depends="compile,config"
           description="Runs tests for service classes">
       <antcall target="build-database">
           <param name="hibernate.auto" value="create" />
       </antcall>
       <antcall target="-run-tests">
           <param name="junit.batchtest.include" value="*.java" />
       </antcall>
   </target>
   ```
3. Executar o alvo criado para correr todos os testes criados
   ```
   $ ant run-tests
   ```

### *Ver*

Testar um serviço

# Desenvolvimento da camada de apresentação web

## *Disponibilizar funcionalidade através de um navegador WWW*

### *Estratégia*

A interacção com o utilizador, numa aplicação web, faz-se através de um navegador WWW. Vamos utilizar o *Stripes* para facilitar o processo de desenvolvimento, pelo que disponibilizar uma funcionalidade implica definir uma *vista* e um *controlador* (o *modelo* é a camada fina de serviços).

### *Passos básicos*

1. Definir a interacção que se pretende, incluindo:
   - que dados são necessários à execução da funcionalidade a disponibilizar;
   - como devem ser recolhidos os dados;
   - como apresentar os resultados ao utilizador.
2. Definir uma acção que processará o pedido, validando os dados, invocando um ou mais serviços e reencaminhando o fluxo de execução para uma vista que apresente os resultados da operação.
3. Desenhar as páginas JSP necessárias para a recolha dos dados necessários (ou adaptar páginas já existentes).
4. Desenhar uma vista (página JSP) para apresentação de resultados, caso necessário.

1.

### *Ver*

Processar um pedido

## *Processar um pedido*

### *Estratégia*

Sempre que um pedido chega ao servidor web necessita de ser validado, processado e apresentado um resultado. Na framework *Stripes* isto equivale a definir o *controlador*.

### *Passos básicos*

1. Criar, no pacote *dominio*.web, a classe *Nome*Action que herda da acção raíz do componente.
2. Definir, na nova classe, atributos para guardar os parâmetros do pedido (com nome *parâmetro*), bem como os métodos acessor (get*Parâmetro*) e modificador (set*Parâmetro*) para manipulação desses atributos.
3. Anotar o atributos utilizados para guardar parâmetros do pedido com as validações necessárias.
4. Definir, na nova classe, o método acessor (get*Parâmetro*) para o(s) resultado(s) da operação.
5. Definir, na mesma classe, o método que vai ser invocado para processar o pedido (não recebe parâmetros e deve devolver um reencaminhamento).
6. Determinar o reencaminhamento a fazer[2].

```
public Resolution operacao() {
   ...
   return new ForwardResolution("/proximo.action");
}
```

[2] O *Stripes*, caso não encontre uma classe associada ao nome pedido (*Proximo*Action), tenta procurar um JSP com o nome pedido e extensão .jsp, neste caso *proximo*.jsp) e redirecciona automaticamente para ele. Se os reencaminhamentos forem sempre feitos para uma acção, este comportamento permite relegar para mais tarde a decisão de continuar o processamento do pedido antes de apresentar o JSP.

## *Variantes*

### *Definir uma acção raíz*

Cada aplicação web deve definir uma acção raíz que uniformiza e simplifica o acesso aos atributos guardados na sessão

1. Criar, no pacote *dominio*.web, a classe *Dominio*Context que herda de ActionBeanContext.
2. Definir, na nova classe, métodos acessor get*Atributo* e modificador (set*Atributo*) para os atributos a guardar em sessão (a concretização destes métodos deve ler o atributo a partir da sessão e guardá-lo em sessão, respectivamente):

```
public class DominioContext extends ActionBeanContext {
    public TipoDoAtributo getAtributo() {
        return (TipoDoAtributo) getRequest().getSession()
                                .getAttribute("chaveDoAtributo");
    }
    public void setAtributo(TipoDoAtributo valor) {
        getRequest().getSession()
                    .setAttribute("chaveDoAtributo", valor);
    }
    ...
}
```

3. Criar, no pacote *dominio*.web, a classe *Dominio*Action que implementa ActionBean.
4. Definir, nesta nova classe, o atributo context e métodos acessor e modificador getContext/setContext para lhe aceder.

```
public class DominioAction implements ActionBean {
    private dominioContext context;

    public DominioContext getContext() { return context; }
    public void setContext(ActionBeanContext context) {
        this.context = context;
    }
}
```

### *Processar múltiplas operações numa única classe*

O esforço de criar uma nova operação que partilha parte dos parâmetros de entrada de outra operação, e cujo resultado é do mesmo tipo pode ser reduzido, definindo numa acção já existente a nova operação

1. Identificar a classe que contém a operação semelhante.
2. Definir, na classe identificada, os métodos acessores para os parâmetros do pedido que não são comuns às restantes operações .
3. Aplicar os passos básicos 5 e 6.