

EAPLI

Princípios de Design OO: Factories & Repositories

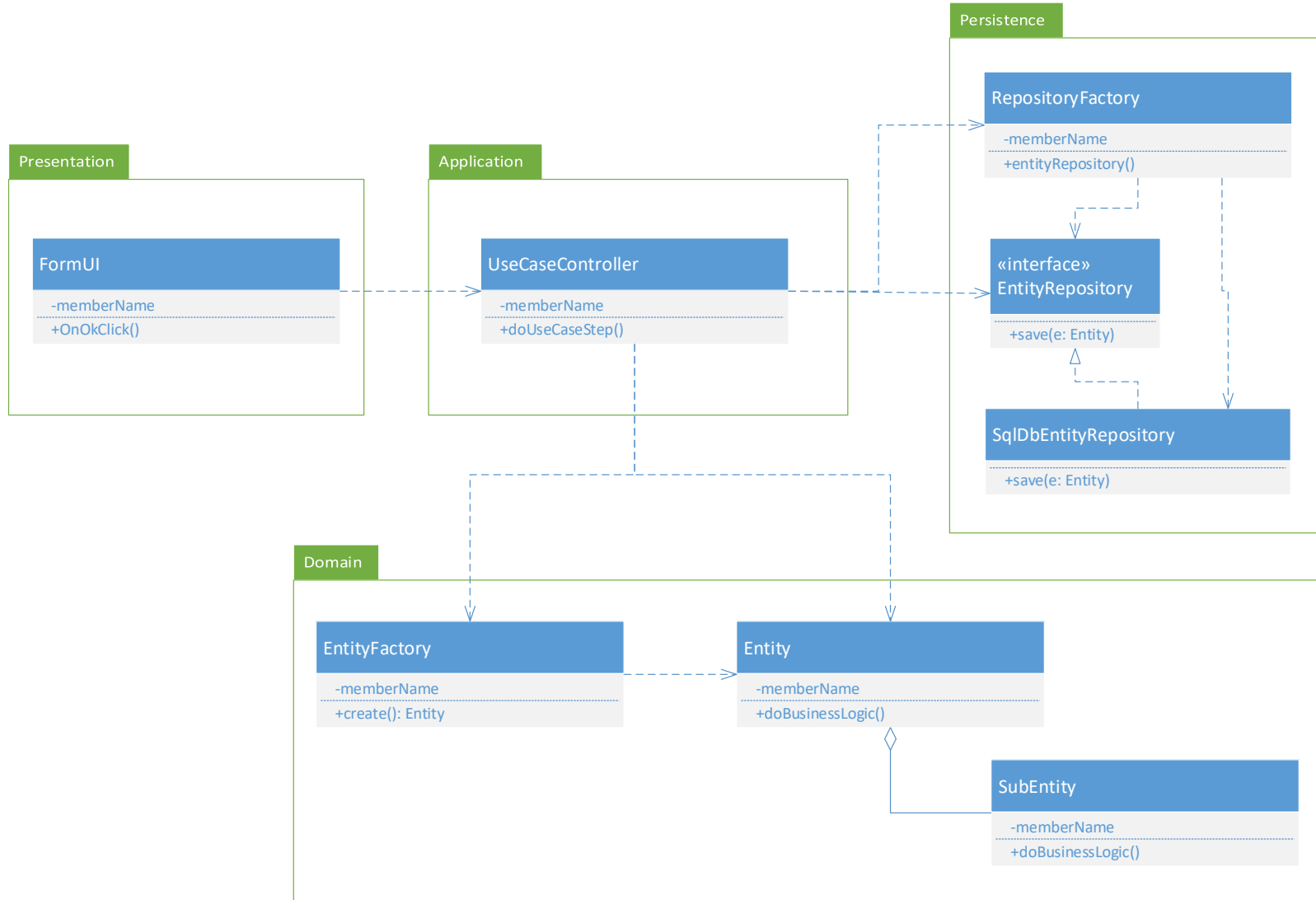
Paulo Gandra de Sousa
pag@isep.ipp.pt

Questões comuns

- A que classe atribuir uma dada responsabilidade?
- Como organizar as responsabilidades do sistema?
- Quem deve ter responsabilidade de coordenar a interação de caso de uso?
- Quem deve ter a responsabilidade de representar e implementar a lógica de negócio?
- Como gerir o ciclo de vida de um objeto?
 - Criar um objeto?
 - persistir objetos?
- Como proteger o código para modificação?

Topic	Principles and patterns
Which class should a responsibility be assigned to?	Information Expert Tell, don't ask Single Responsibility Principle Interface Segregation Principle Intention Revealing Interfaces
How to organize the system's responsibilities?	
How to model the domain?	Persistence Ignorance Entity Value Object Domain Service Aggregate Domain Event Observer
How to handle an object's lifecycle?	
How to prepare the code for modification?	

Sterotypical architecture



Como gerir o ciclo de vida de um objeto?

Quem deve ter a responsabilidade de criar objetos?

Quem deve ter a responsabilidade de persistir objetos?

Object's lifecycle

1. An object is created
2. The object is used
3. It must be persisted for later use
4. Later, the object is reconstructed from persistence
5. It is used (provably changed)
6. It is stored back again for later use
7. ...

Separate use from construction

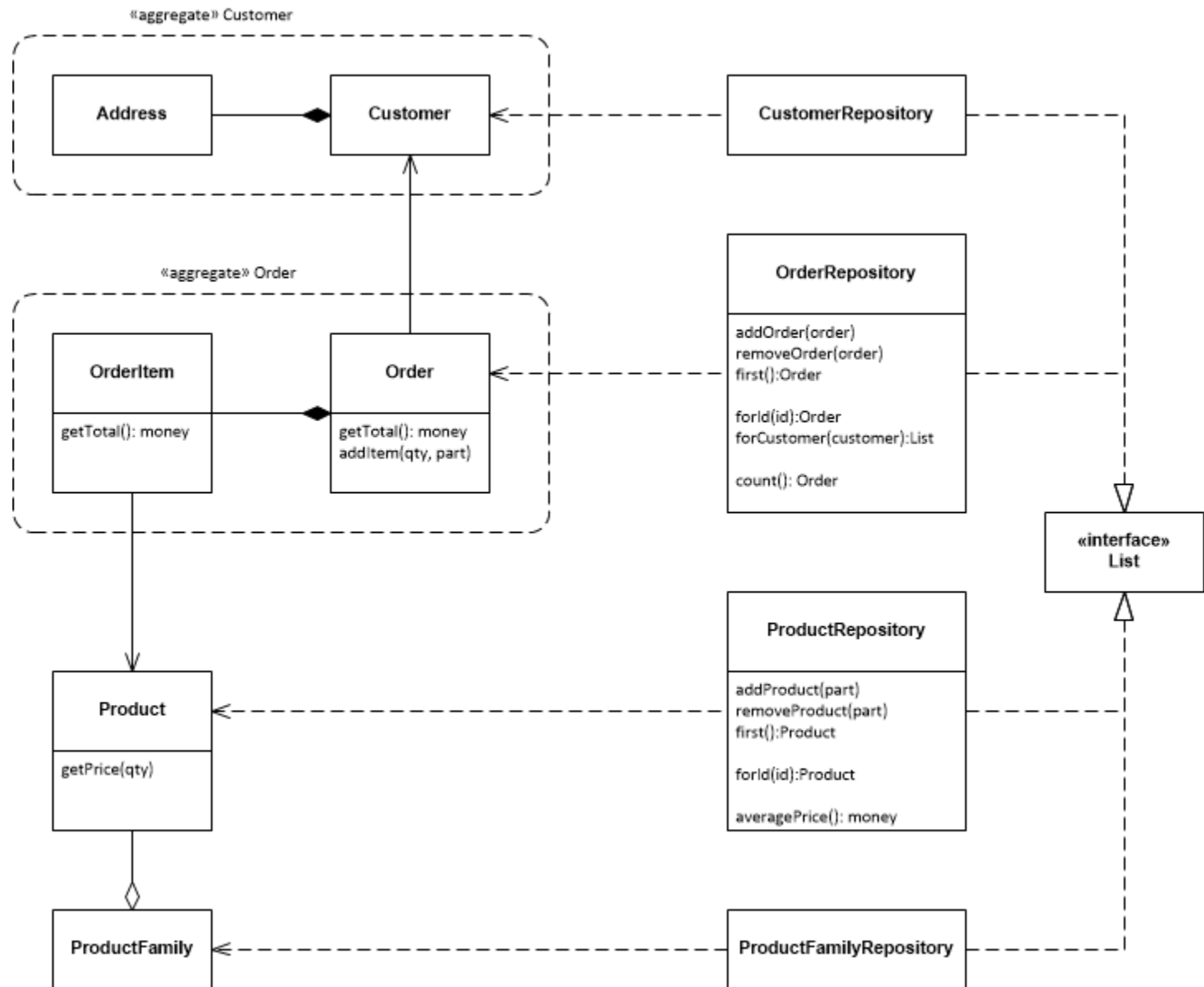
- Criar um objeto pode ser complexo
- “criar” é uma responsabilidade diferente da de “usar”
- Tipicamente um objeto é criado num sitio e usado em muitos outros

Quem deve ter a responsabilidade de persistir e reconstruir objetos a partir da persistência?

Repository


- Problem:
 - How to hide the details of persisting and reconstructing an object while keeping the domain language?
- Solution:
 - Query access to aggregates expressed in the ubiquitous language
 - Abstract the persistence of an object in a Repository class that behaves as a list

Repository example



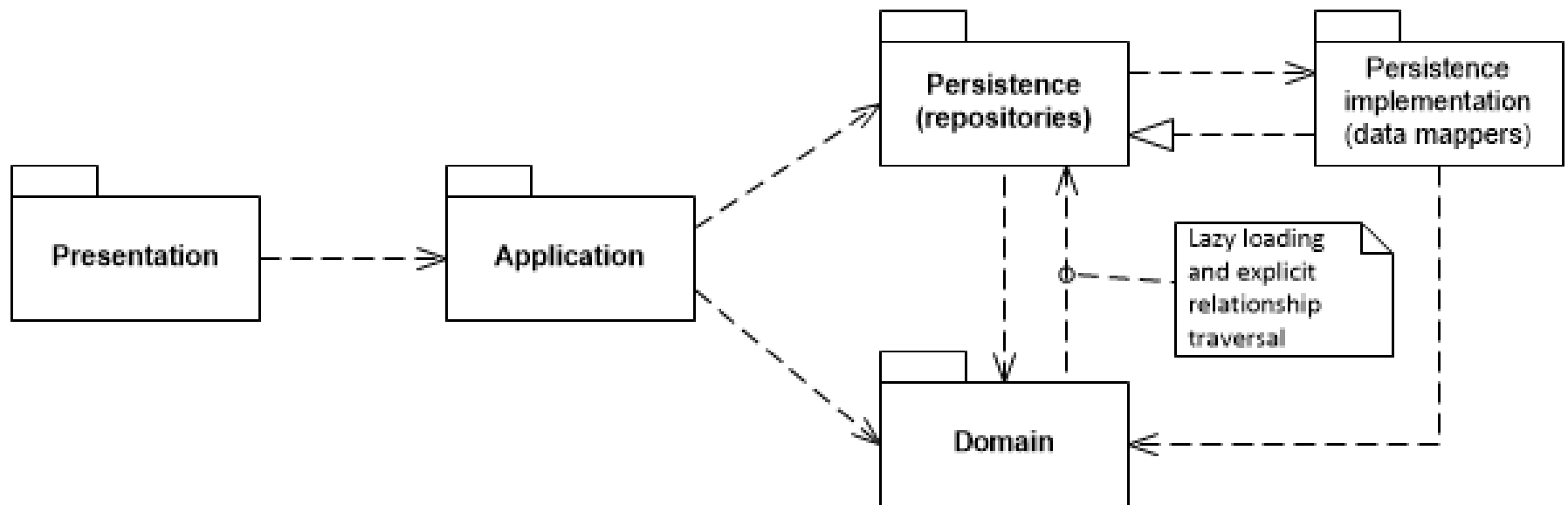
Repository's methods

- findById
- save
 - Eventually add, update
- delete
- all, page, iterator
- Specialized finders, e.g.,
 - ProductRepository.findByIdByFamily
 - ExpenseRepository.findByIdByMonth
- Aggregation functions, e.g.,
 - ProductRepository.getAverageRetailPrice
 - ExpenseRepository.getTotalExpenditure



Careful as we are actually performing business logic at the data layer

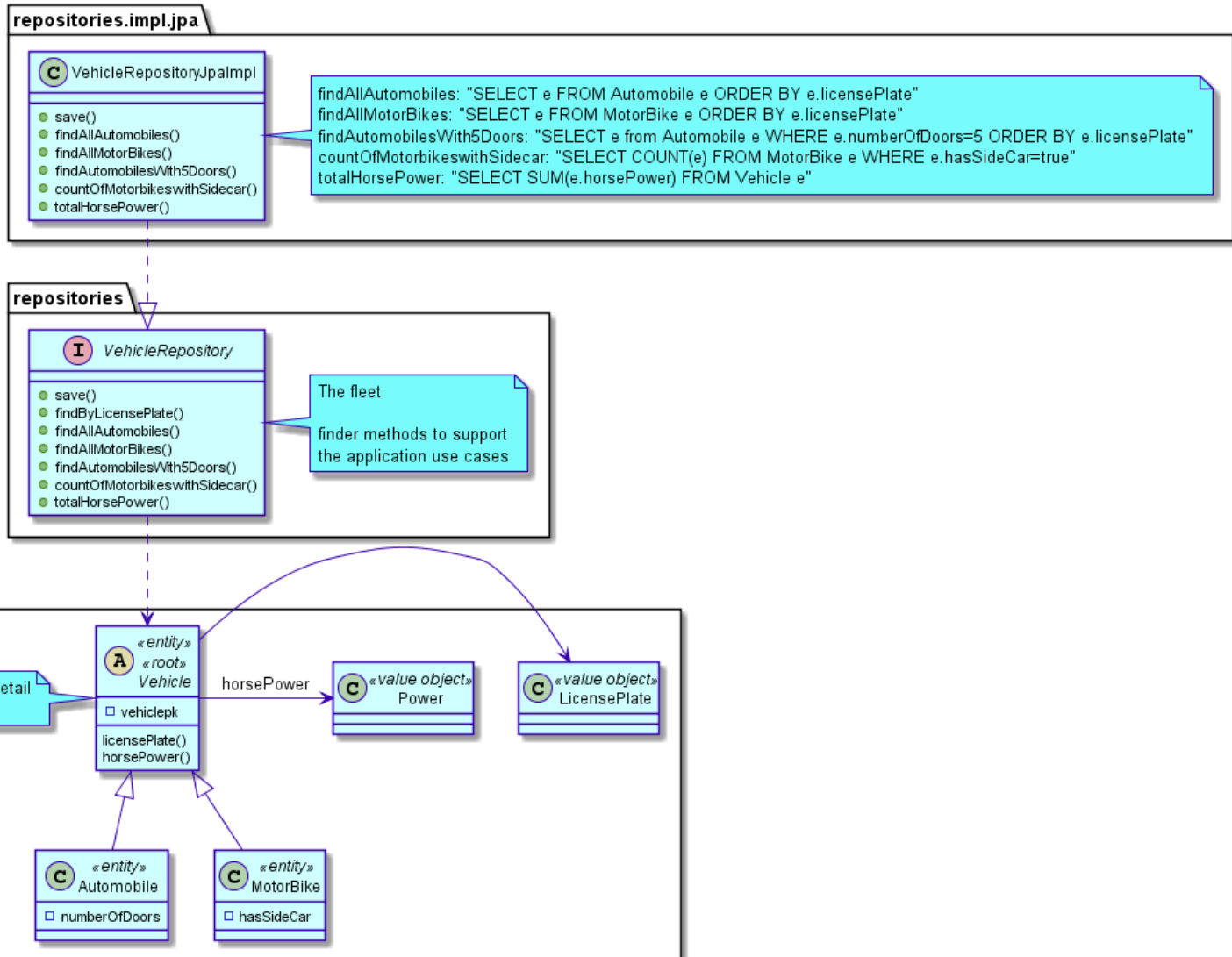
Repositories and layers



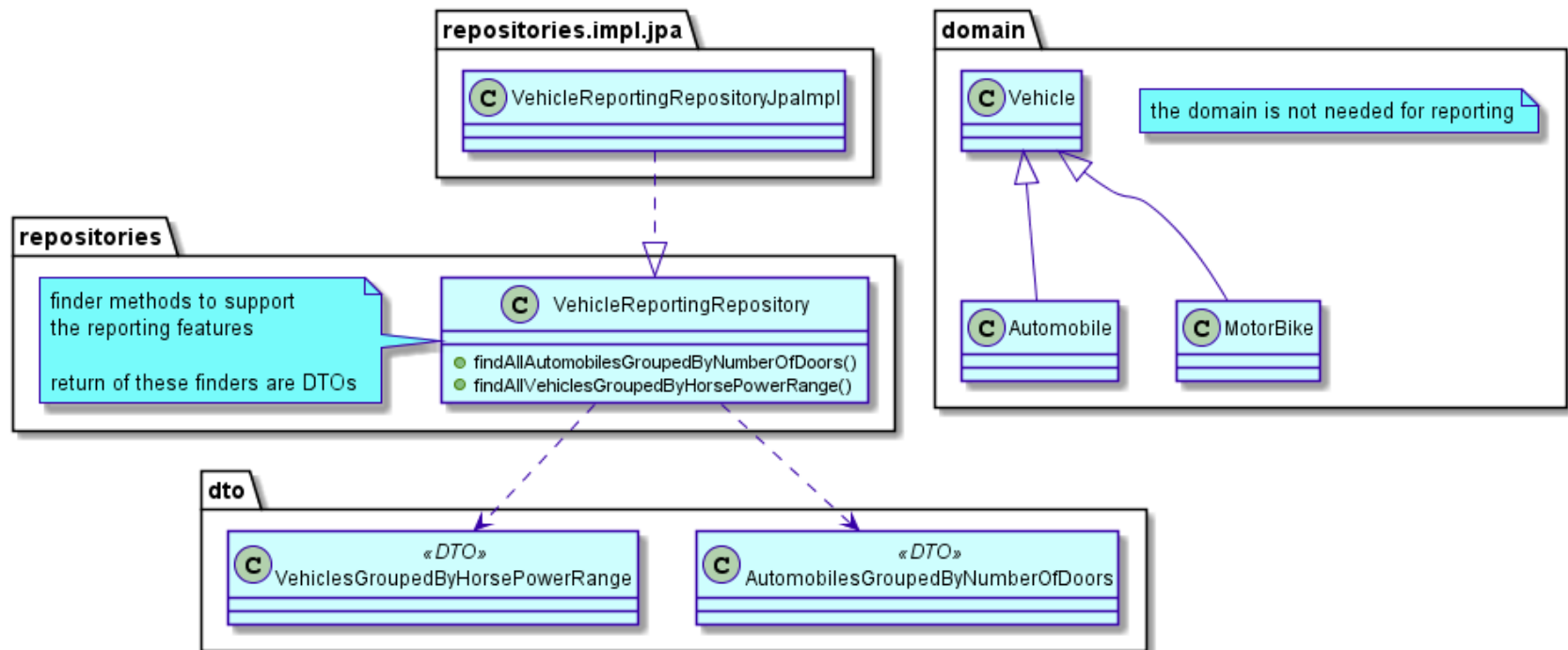
Repository checklist

- Separate interface from implementation
- Do not expose implementation details, e.g., database primary key
- Filtering and sorting is done in the implementation (e.g., JPQL)
- One repository per aggregate

Repository exemple (inheritance)



Reporting example





Quem deve ter a responsabilidade de criar novos objetos?

Factory

When creation of an entire, internally consistent aggregate, or a large value object, becomes complicated or reveals too much of the internal structure, factories provide encapsulation.

Domain in vs. out

- Domain factories
 - Create domain entities, aggregates and value objects
- Other types of factories
 - E.g., Persistence

Constructors vs. Factories

- Sometimes a constructor is enough
- Does not allow manipulation of different interfaces

Creator

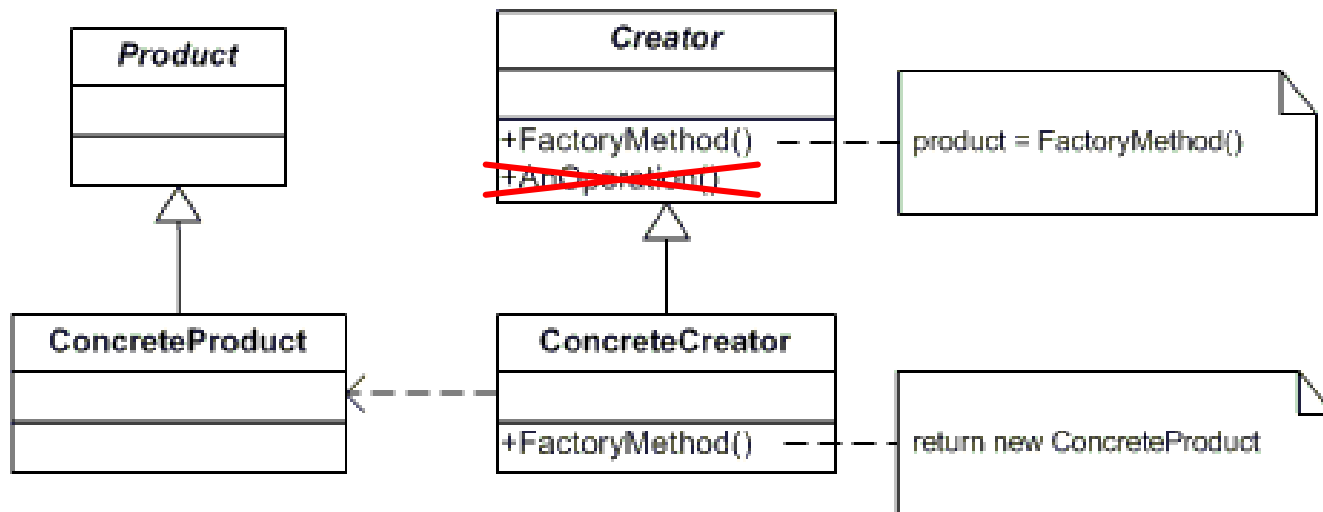
- Problema:
 - Quem deve ser responsável pela criação de objetos de uma classe?
- Solução:
 - Atribuir à classe B a responsabilidade de criar instâncias da classe A nas seguintes condições:
 - B contém ou agrega objetos da classe A
 - B regista instâncias da classe A
 - B possui os dados usados para inicializar A
 - B está diretamente relacionado com A
 - Se mais de uma condição se aplicar, escolhe-se a “classe B que contém ou agrega objetos da classe A”

Factory Method

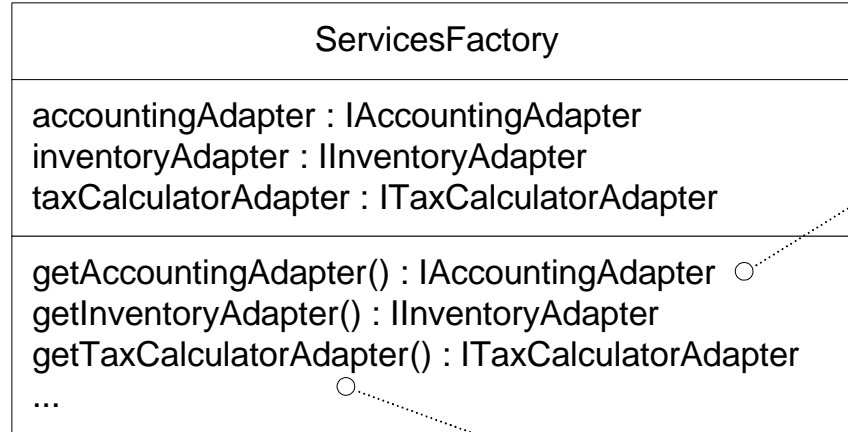
- Problema:
 - Como se simplifica a manipulação de diferentes implementações da mesma interface
- Solução:
 - Esconder a criação num método.
 - O método devolve um tipo que é mais geral que o seu tipo de facto.

Simple Factory

- A class with factory methods only



Factory pode criar objectos diferentes a partir dum Ficheiro



note that the factory methods return objects typed to an interface rather than a class, so that the factory can return any implementation of the interface

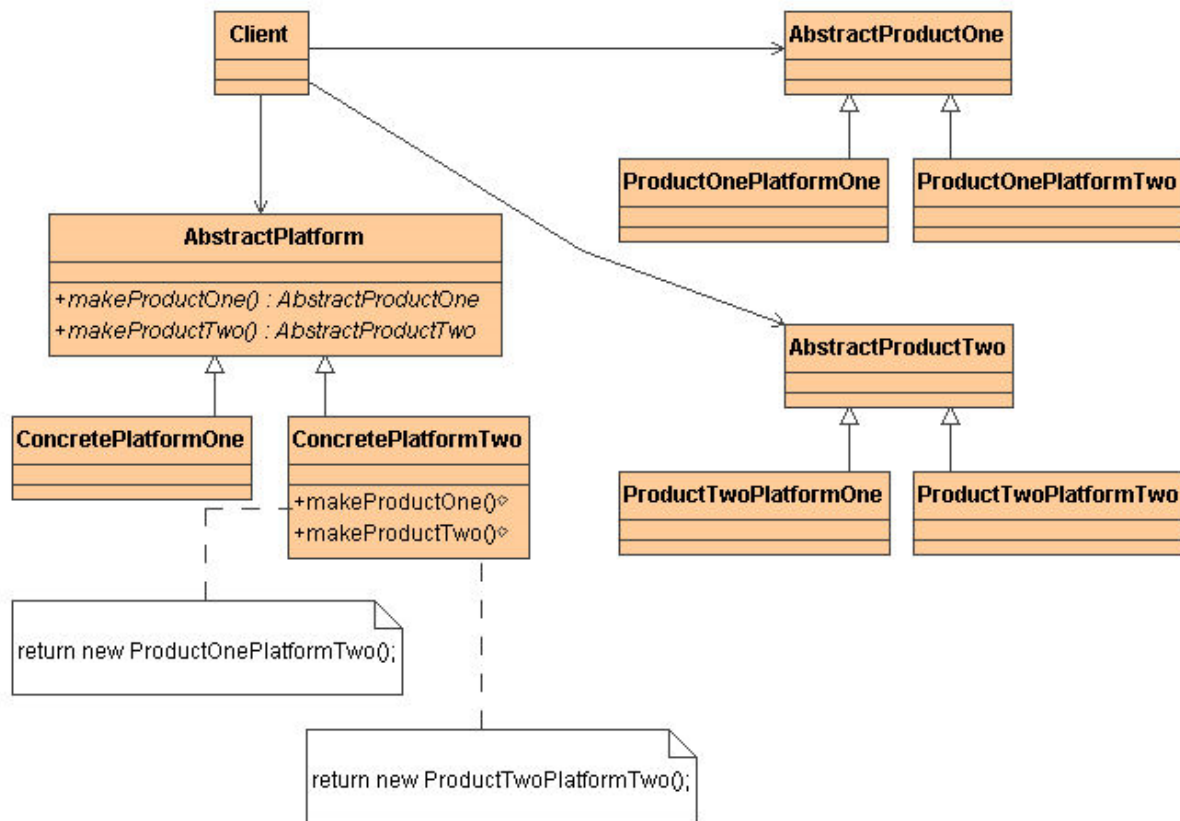
```
if ( taxCalculatorAdapter == null )
{
    // a reflective or data-driven approach to finding the right class: read it from an
    // external property

    String className = System.getProperty( "taxcalculator.class.name" );
    taxCalculatorAdapter = (ITaxCalculatorAdapter) Class.forName( className ).newInstance();
}
return taxCalculatorAdapter;
```

Figure 26.5

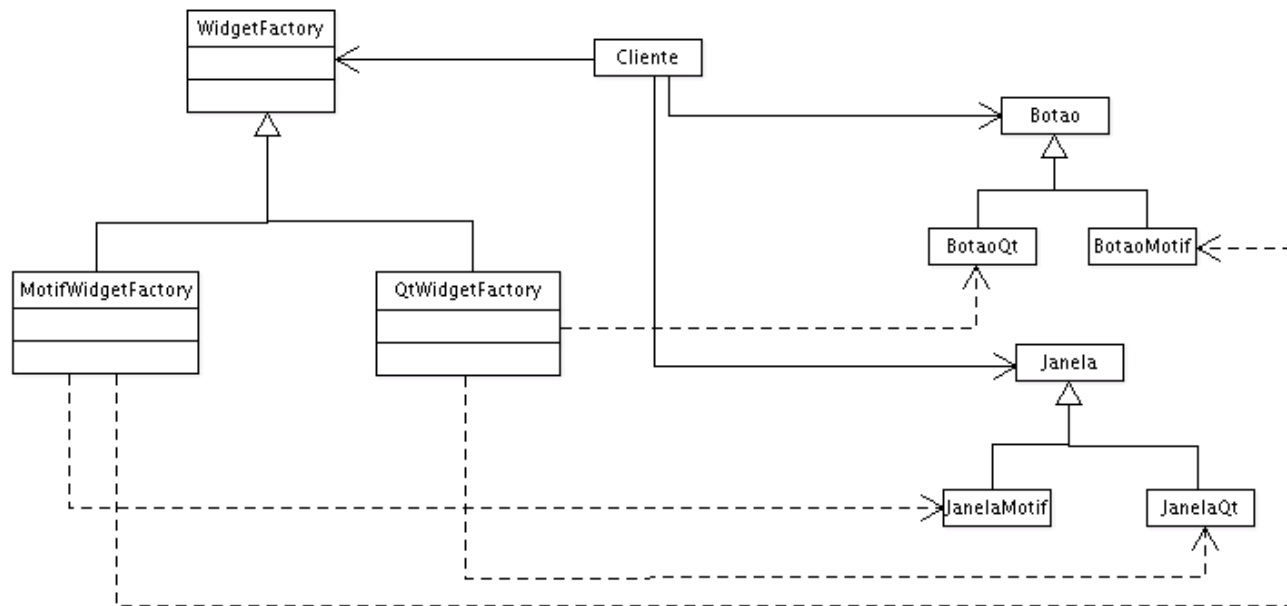
Abstract Factory

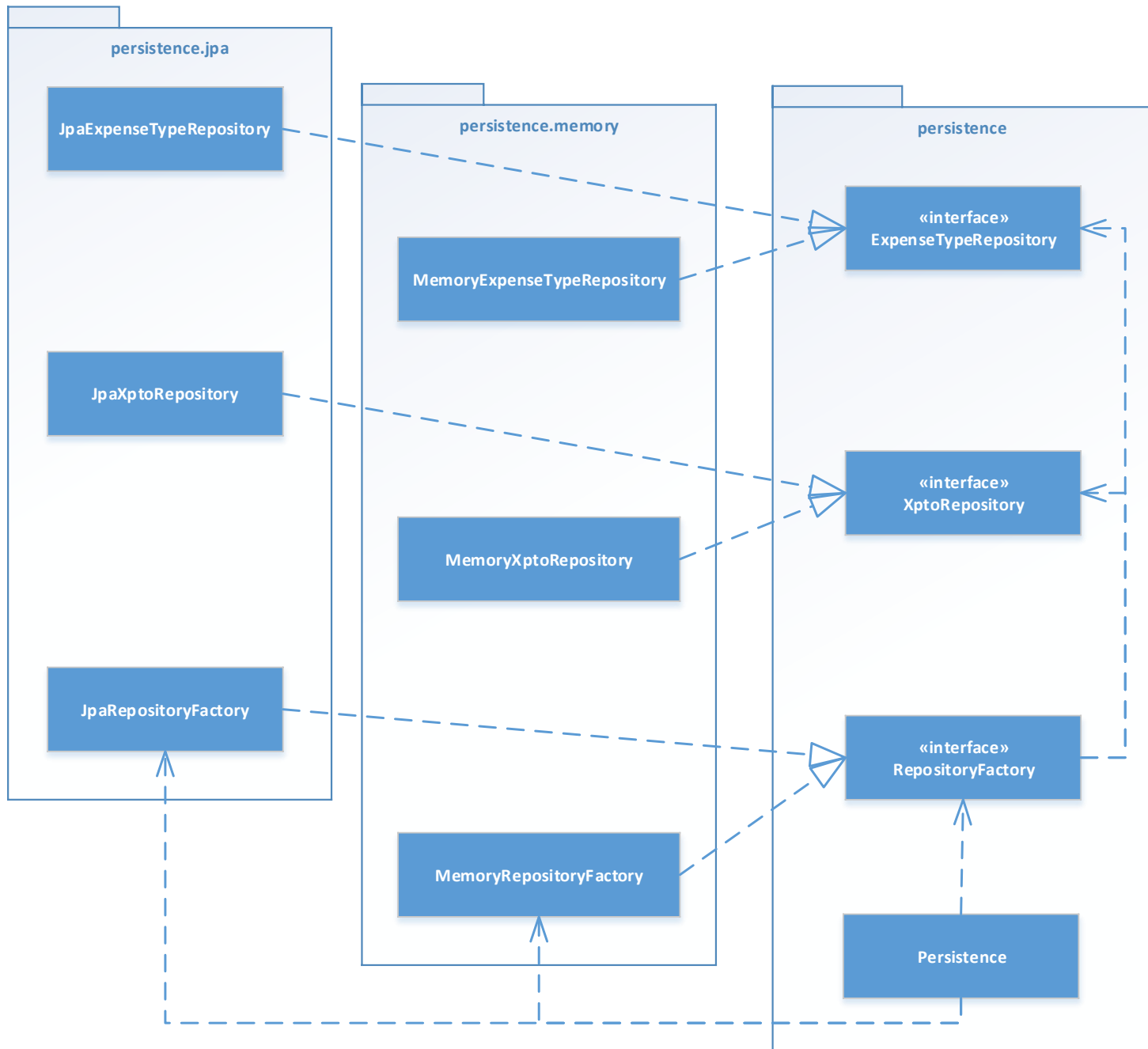
- Disponibiliza uma interface para a criação de famílias de objectos relacionados ou dependentes, sem especificar as suas classes concretas



Abstract Factory (exemplo)

- Os objectos das interfaces gráficas são séries de produtos, que fazem sentido em conjunto (i.e. não faz sentido usar ao mesmo tempo uns controles de um estilo e outros de outros)
- Assim, uma factory deve ser capaz de construir todos os objectos do mesmo estilo/série, em vez da sua criação estar distribuída por várias fábricas





Builder

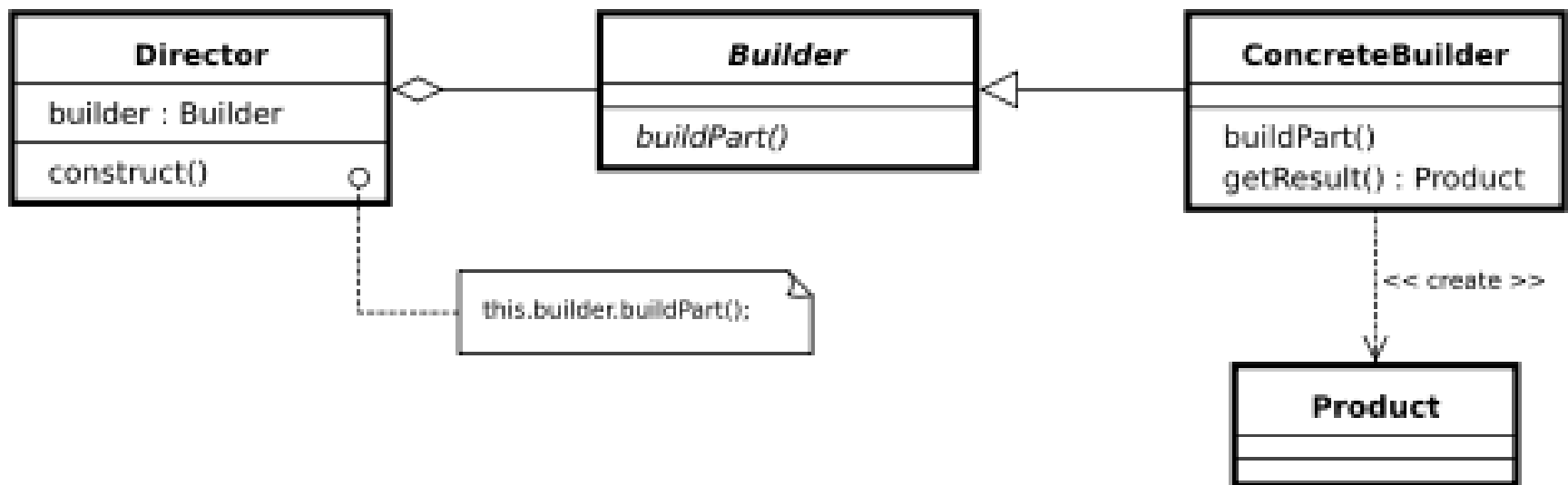
■ Problem

- How to avoid different constructors for different situations and allow to create an object step-by-step?
- How to create an object composed of different parts that can be combined in multiple ways?

■ Solution

- Create a builder object that constructs the desired object by providing a more fluent interface to its parts.

Builder



Example

```
interface Car {  
    ...  
}
```

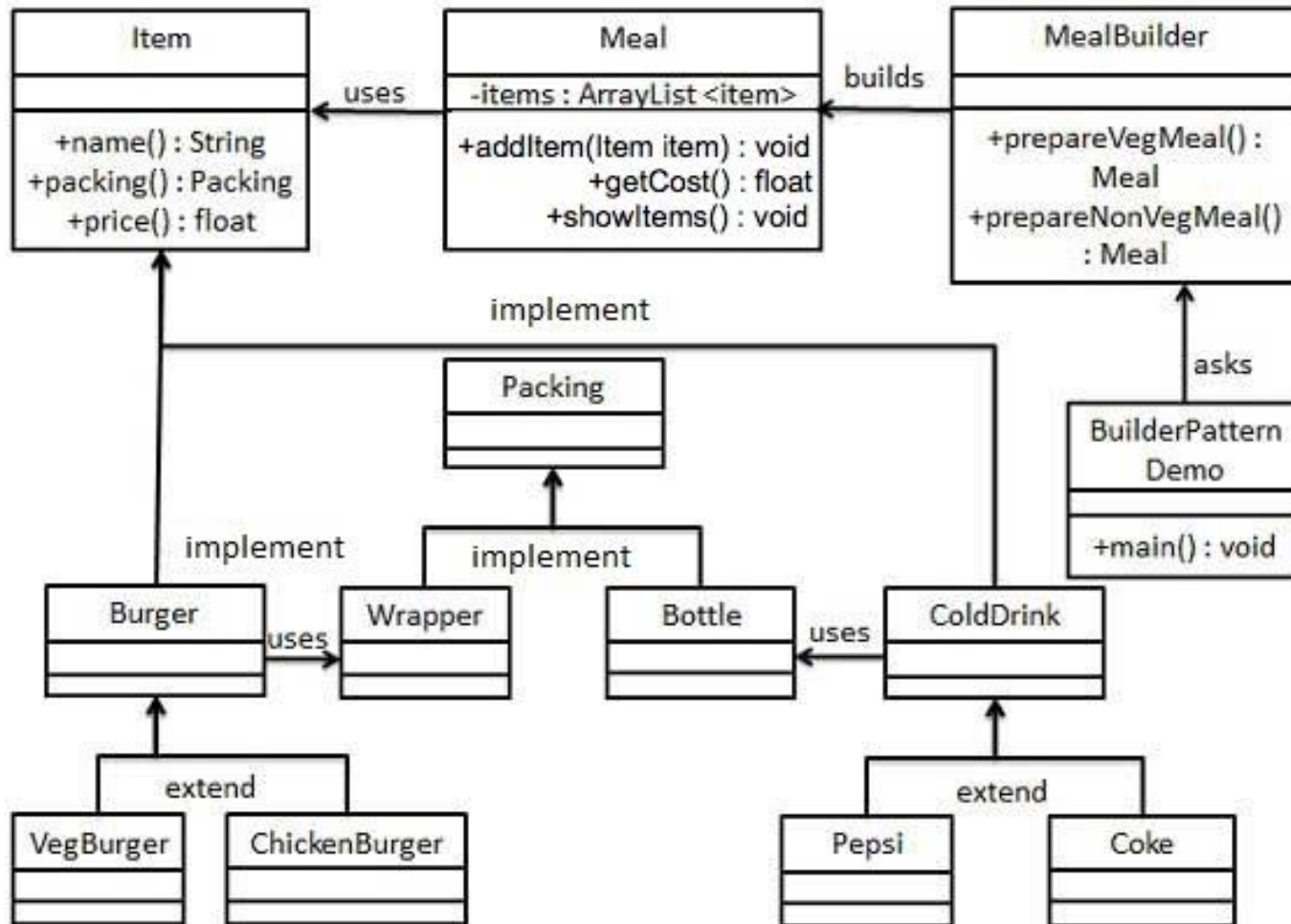
```
interface CarBuilder {  
    void buildChassis(ChassisType c);  
    void buildFrame(FrameType f);  
    void buildEngine(EngineType e);  
    Car getCar();  
}
```

```
class SuvBuilder implements CarBuilder { ... }
```

```
class SedanBuilder implements CarBuilder { ... }
```

```
class TruckBuilder implements CarBuilder { ... }
```

Another example



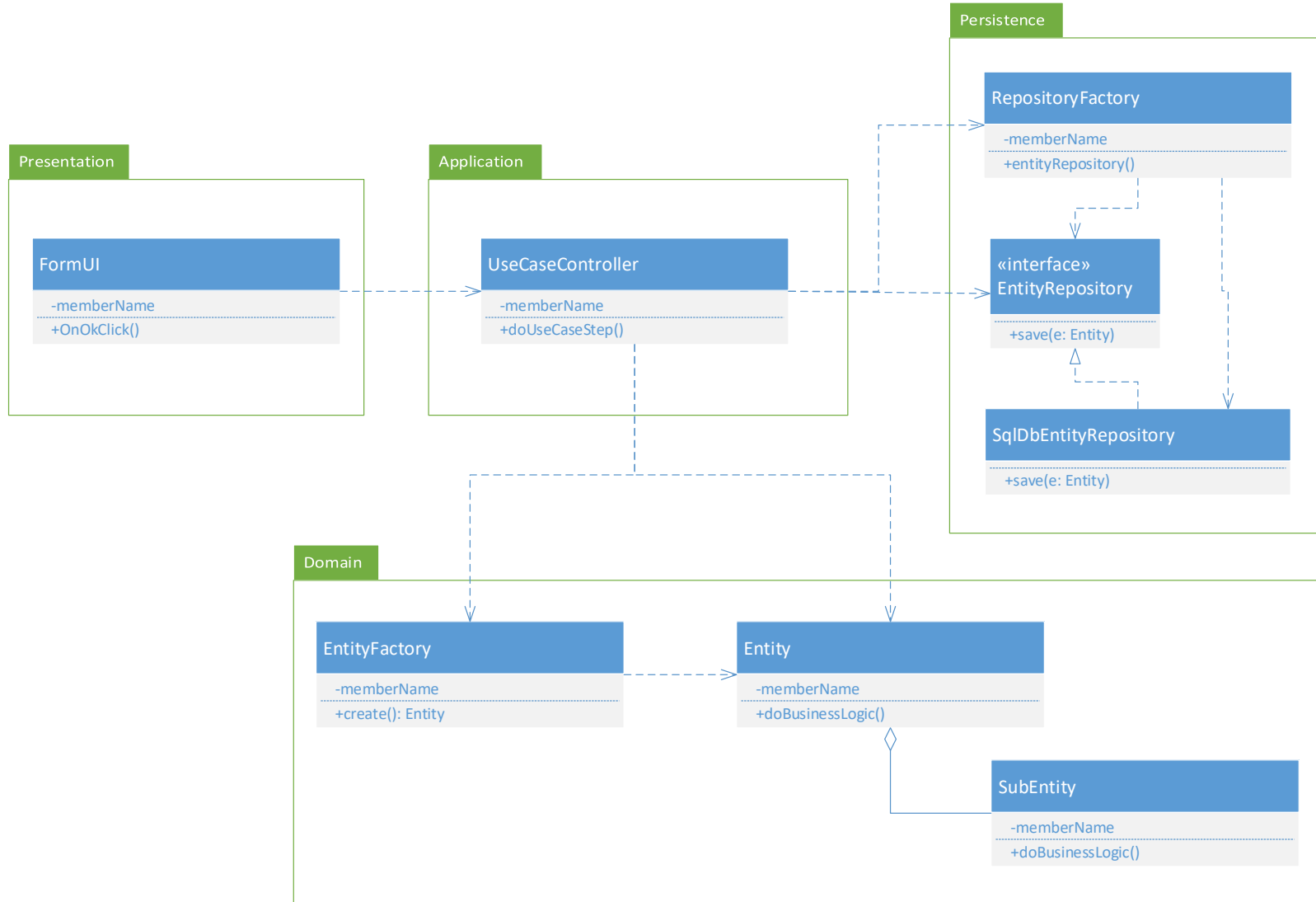
Source: http://www.tutorialspoint.com/design_pattern/builder_pattern.htm

Builders might hold inconsistent objects

- An object must always be consistent
- Constructing a complex object in one single shot might not be practical
- An object might allow a builder to construct it incrementally as long as the builder does not expose the inconsistent object to the outside world

Topic	Principles and patterns
Which class should a responsibility be assigned to?	Information Expert Tell, don't ask Single Responsibility Principle Interface Segregation Principle Intention Revealing Interfaces
How to organize the system's responsibilities?	
How to model the domain?	Persistence Ignorance Entity Value Object Domain Service Aggregate Domain Event Observer
How to handle an object's lifecycle?	Factories <ul style="list-style-type: none"> - Factory method - Simple factory - Abstract factory Repositories
How to prepare the code for modification?	

Sterotypical architecture



Bibliografia

- Domain Driven Design. Eric Evans
- Applying UML and Patterns; Craig Larman; (2nd ed.); 2002.
- Why getters and setters are Evil. Allan Holub.
<http://www.javaworld.com/article/2073723/core-java/why-getter-and-setter-methods-are-evil.html>
- Design Principles and Design Patterns. Robert Martin.
http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf
- Design Patterns-Elements of Reusable Object-oriented Software, Gamma et al. (Gang of Four)