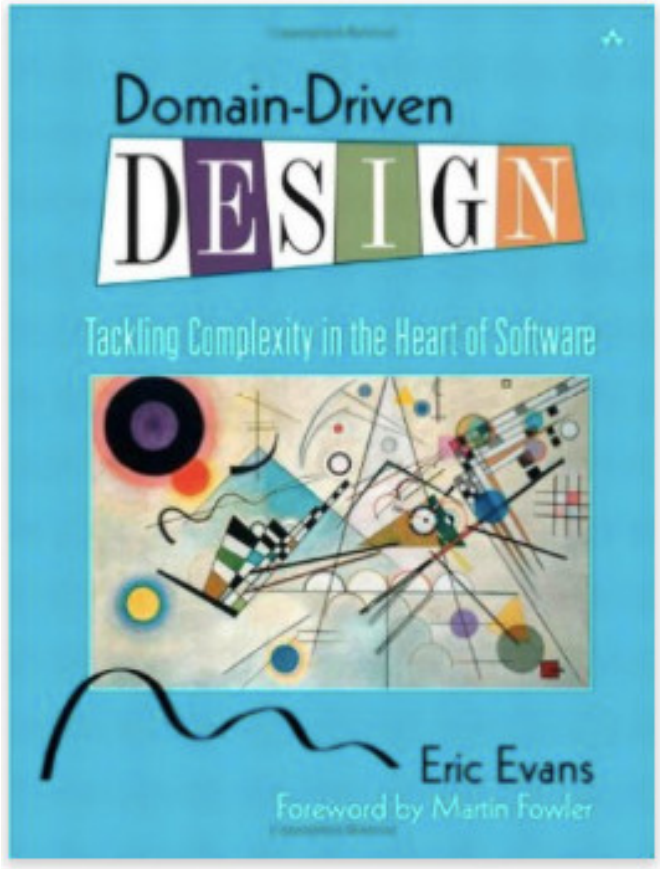
EDOM - Engenharia de Domínio
Mestrado em Engenharia Informática
Lecture 05.1
Life Cycle of Domain Objects

Alexandre Bragança atb@isep.ipp.pt

Dep. de Engenharia Informática – ISEP

2017/2018

Acknowledgement



"Domain-Driven Design: Tackling Complexity in the Heart of Software", Eric Evans, Addison Wesley, 2003

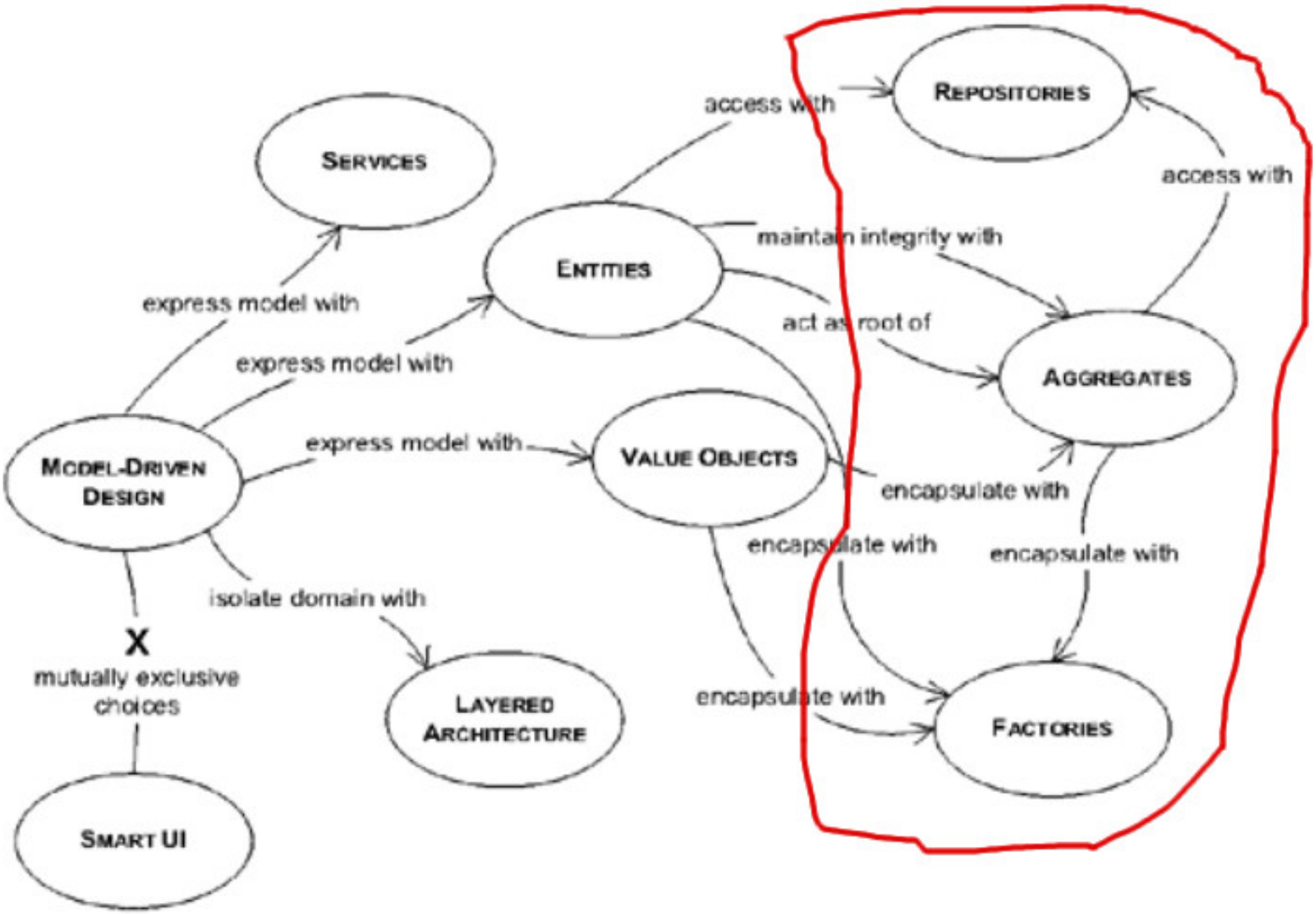
This lecture is based on the contents of this book.

There is a reference document explaining the principles of the book that you can find at: `https://domainlanguage.com/ddd/reference/PatternSummariesUnderCreativeCommons.doc`)

There is a reference document containing some updates that you can find at: `https://domainlanguage.com/ddd/reference/DDD_Reference_2015-03.pdf`)

We will be focusing on the section "Building Blocks of a Model-Driven Design".

Life Cycle of Domain Objects

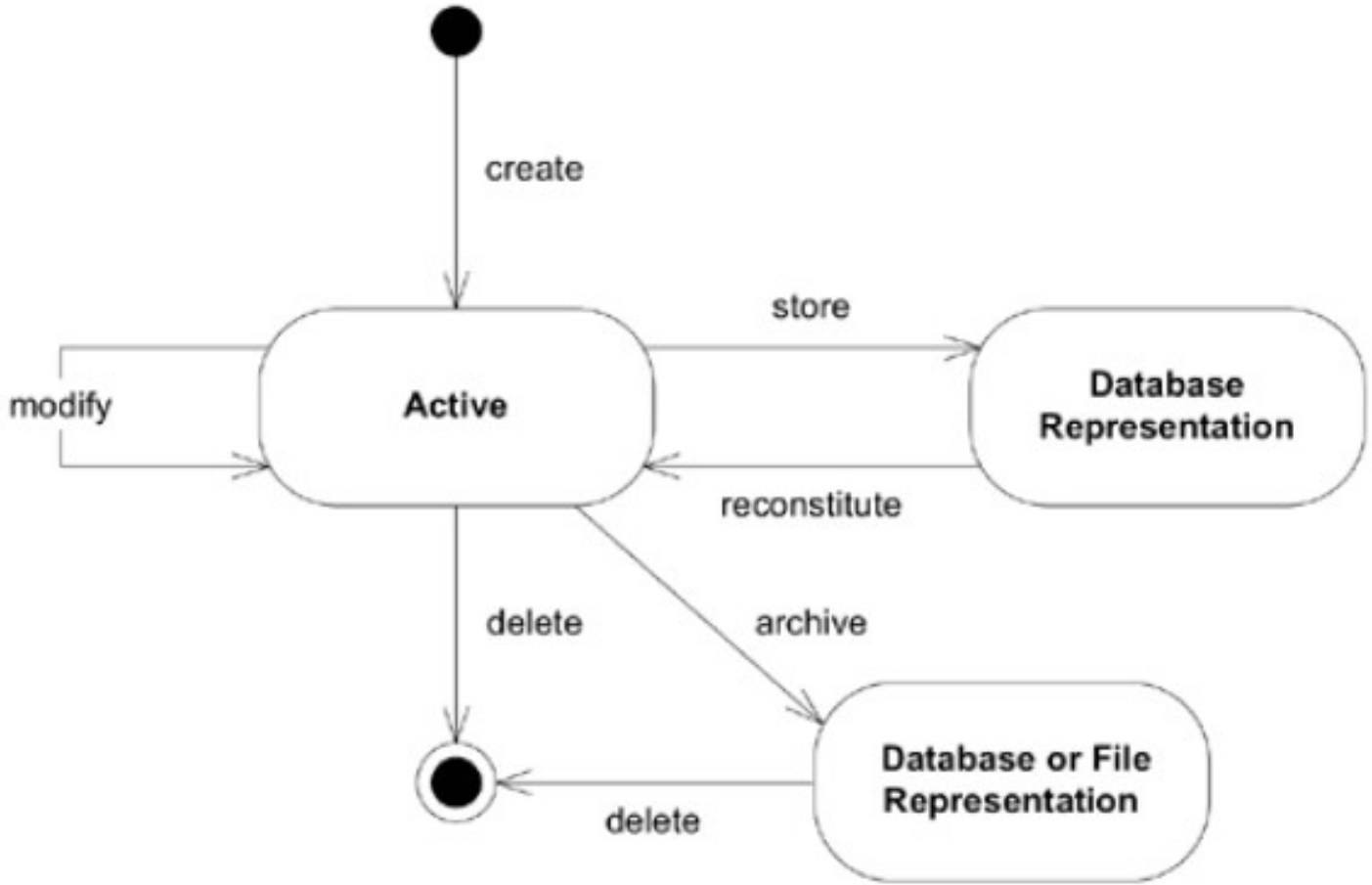


- We will discuss: **Modules**, **Aggregates**, **Factories** and **Repositories**.

Modules

- **Cognitive overload is the primary motivation for modularity**. It is difficult to deal with several concepts at the same time. We package concepts into Modules.
  - We can look at the detail within a Module without being overwhelmed by the whole...
  - ...or we can look at relationships between Modules in views that exclude the interior detail.
- There should be **low coupling between Modules and high cohesion within them**.
- If our model tells a story, each Module is a chapter in that story. The name of the module conveys its meaning.

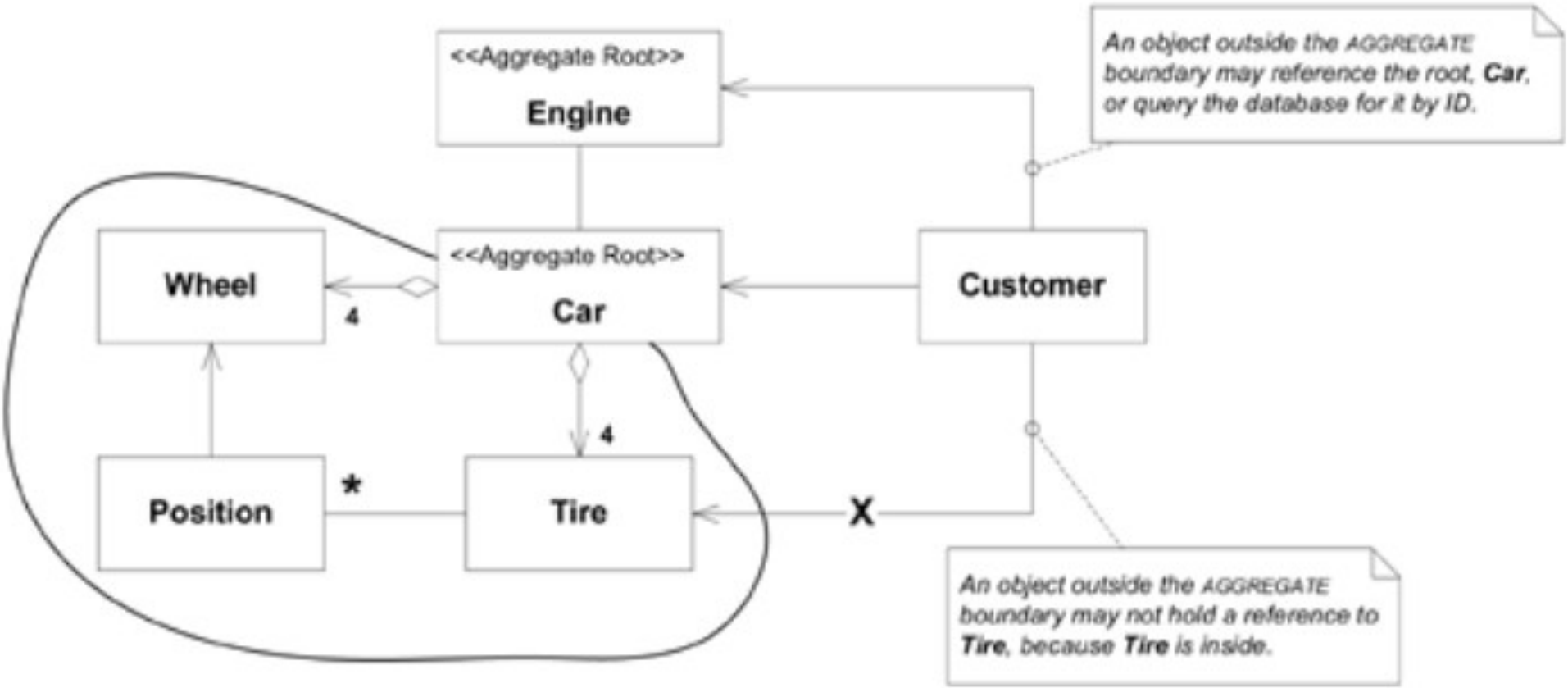
The Life Cycle of a Domain Object



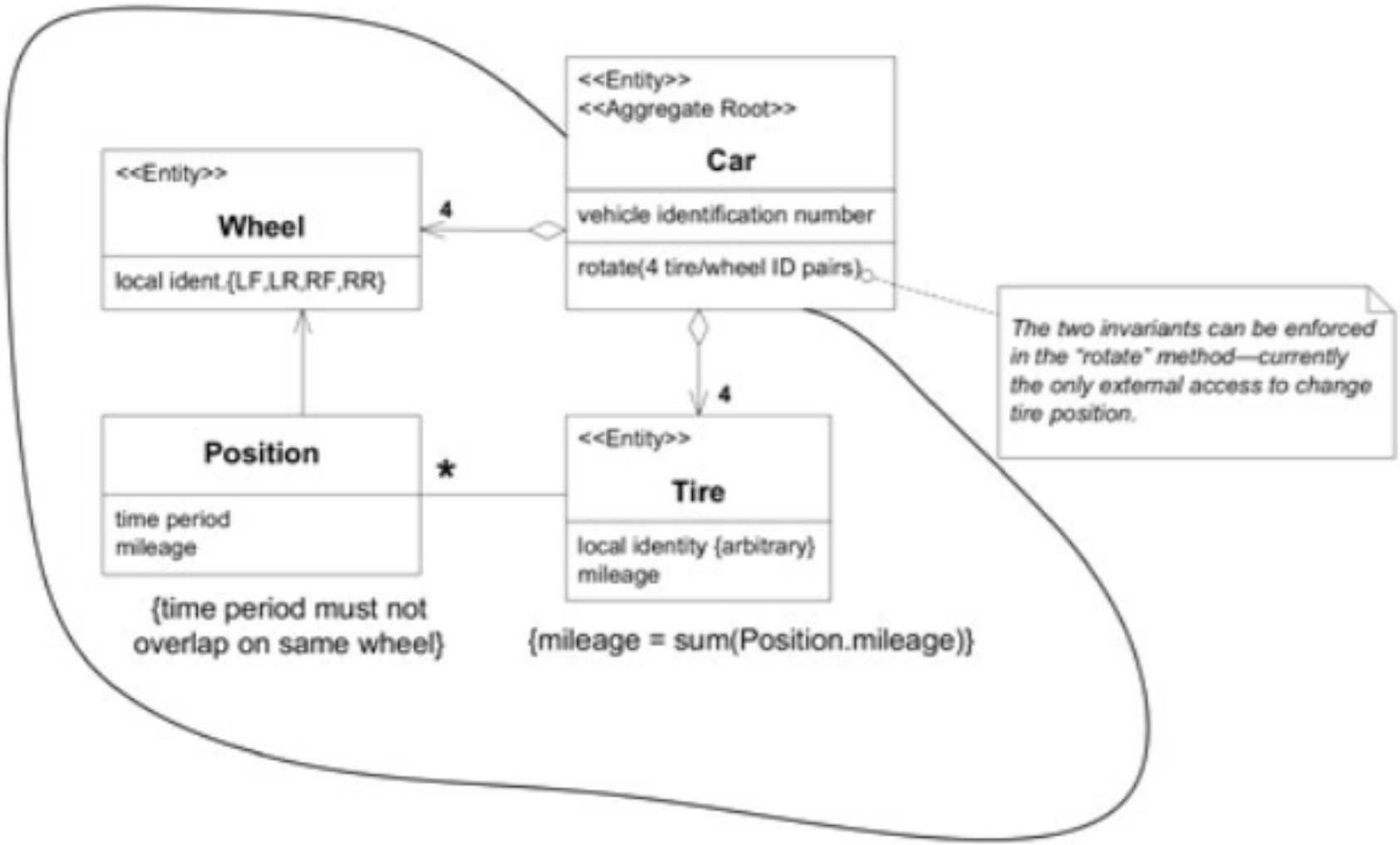
Aggregates

- An **agregate** is a **cluster of associated objects** that we treat as a **unit for the purpose of data changes**.
- Each Aggregate has a **root** and a **boundary**.
- The Boundary defines what is inside the Aggregate.
- The root is a simple, specific Entity contained in the Aggregate.
- The root is the only member of the Aggregate that outside objects are allowed to hold references to, although objects within the boundary may hold references to each other.
- **Entities other than the root have local identity**, but that identity needs to be distinguishable only within the Aggregate, because no outside object can ever see it out of the context of the root Entity.

Aggregates: Example



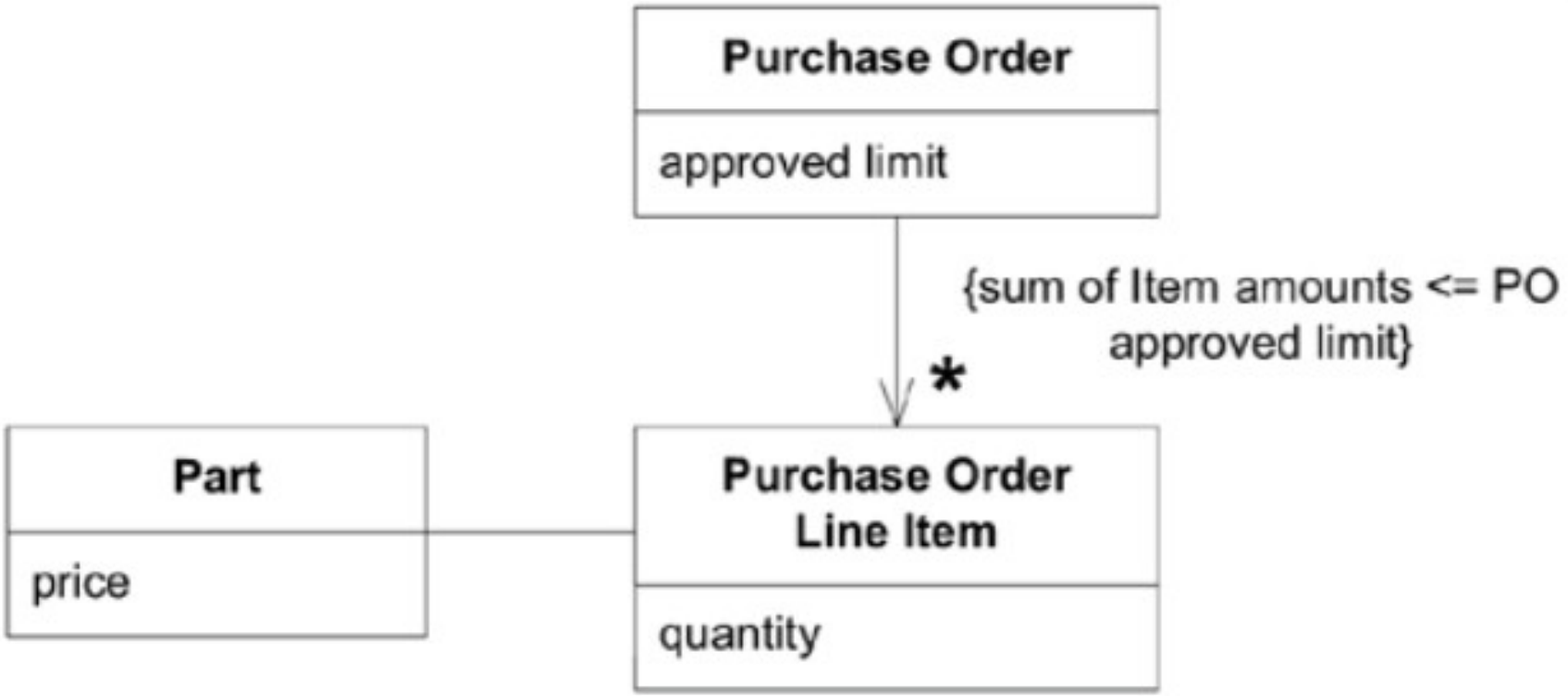
Aggregates: Invariants Example



Aggregates Invariants Rules

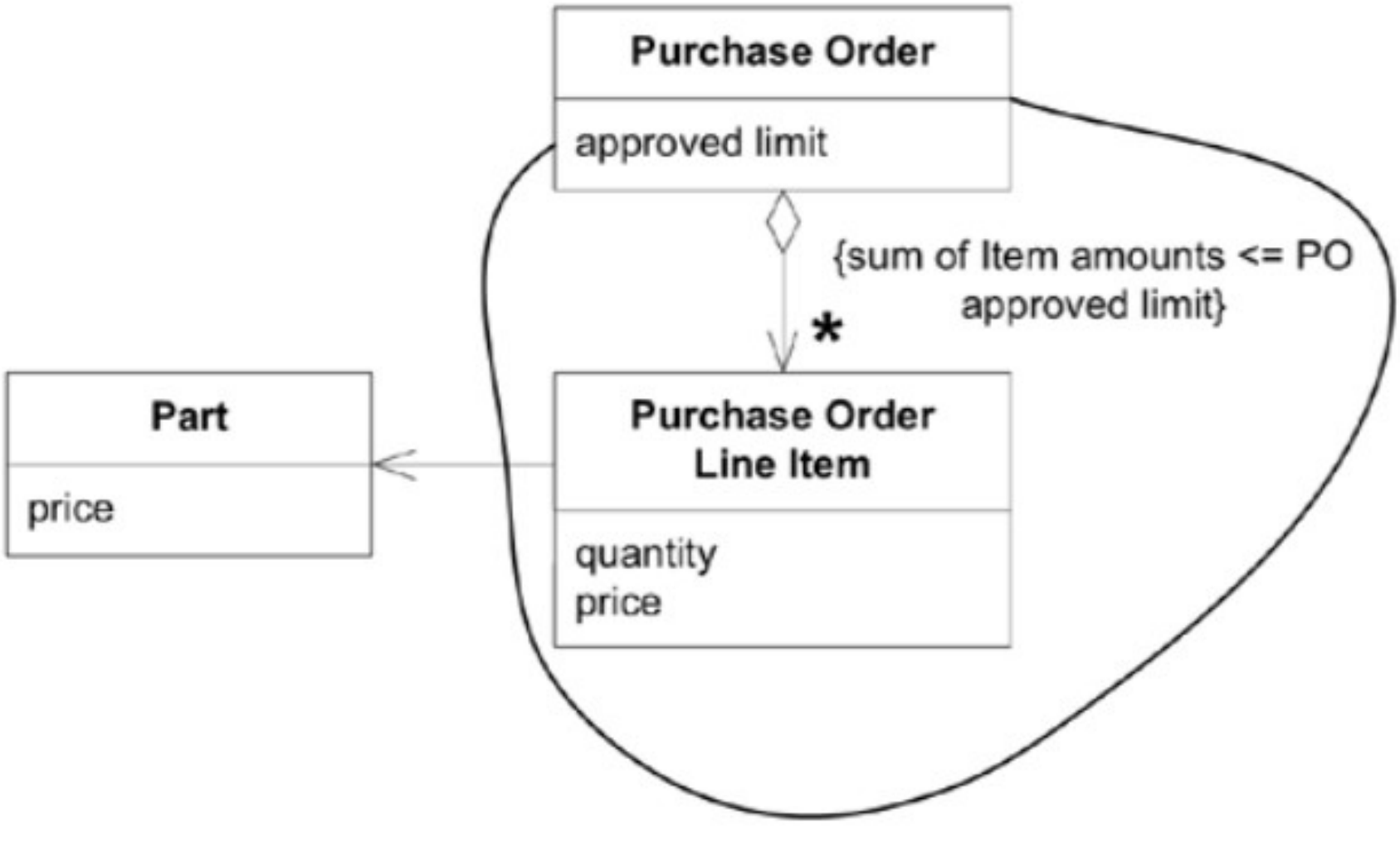
- The **root ENTITY** has **global identity** and is ultimately **responsible for checking invariants**.
  - Root ENTITIES have global identity. ENTITIES inside the boundary have local identity, unique only within the AGGREGATE.
- **Nothing outside the AGGREGATE** boundary **can hold a reference to anything inside**, except to the root ENTITY.
  - The root ENTITY can hand references to the internal ENTITIES to other objects, but those objects can use them only transiently, and they may not hold on to the reference.
  - The root may hand a copy of a VALUE OBJECT to another object, and it doesn't matter what happens to it, because it's just a VALUE and no longer will have any association with the AGGREGATE.
- As a corollary to the previous rule, **only AGGREGATE roots can be obtained directly with database queries**. All other objects must be found by traversal of associations.
- Objects within the AGGREGATE can hold references to other AGGREGATE roots.
- A delete operation must remove everything within the AGGREGATE boundary at once.
- When a **change** to any object within the AGGREGATE boundary **is committed**, **all invariants of the whole AGGREGATE must be satisfied**.

Aggregates: Purchase Order Integrity Example



- This model has a problem...
- If we have multiple updates to Order Line Item...
- Or if the price of the Part changes...

Aggregates: Purchase Order Integrity Example



- We define the aggregate and its invariants.
- The price is copied into the Order Line Item.

Aggregates: Summary

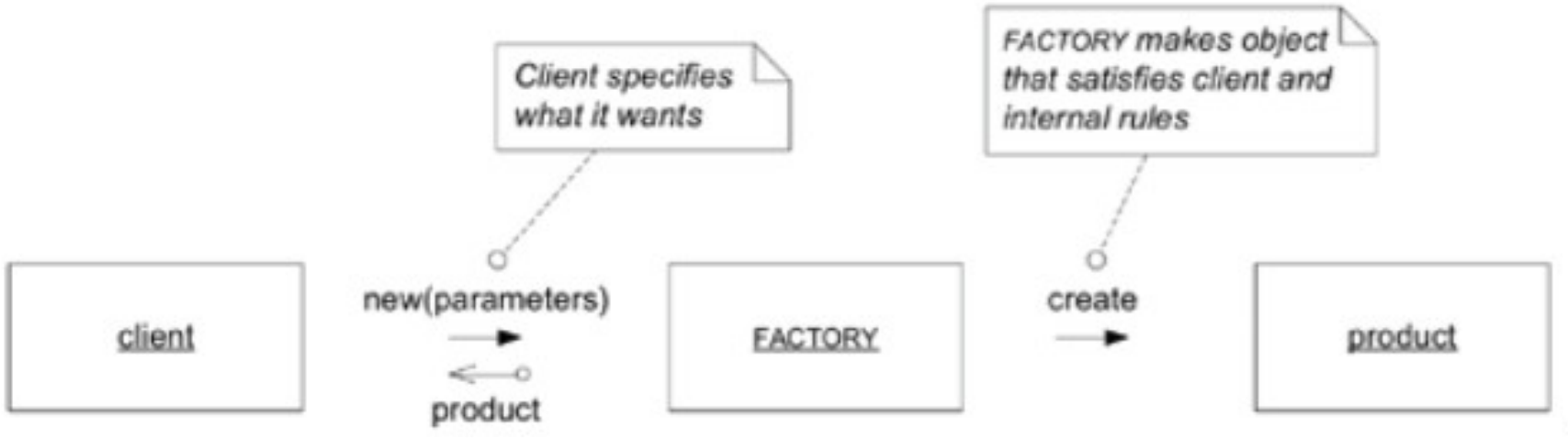
- **AGGREGATES mark off the scope** within which **invariants have to be maintained** at every stage of the life cycle.
- The following patterns, **FACTORIES** and **REPOSITORIES**, operate on AGGREGATES, encapsulating the complexity of specific life cycle transitions. . . .

Factories

When creation of an object, or an entire AGGREGATE, becomes complicated or reveals too much of the internal structure, FACTORIES provide encapsulation.

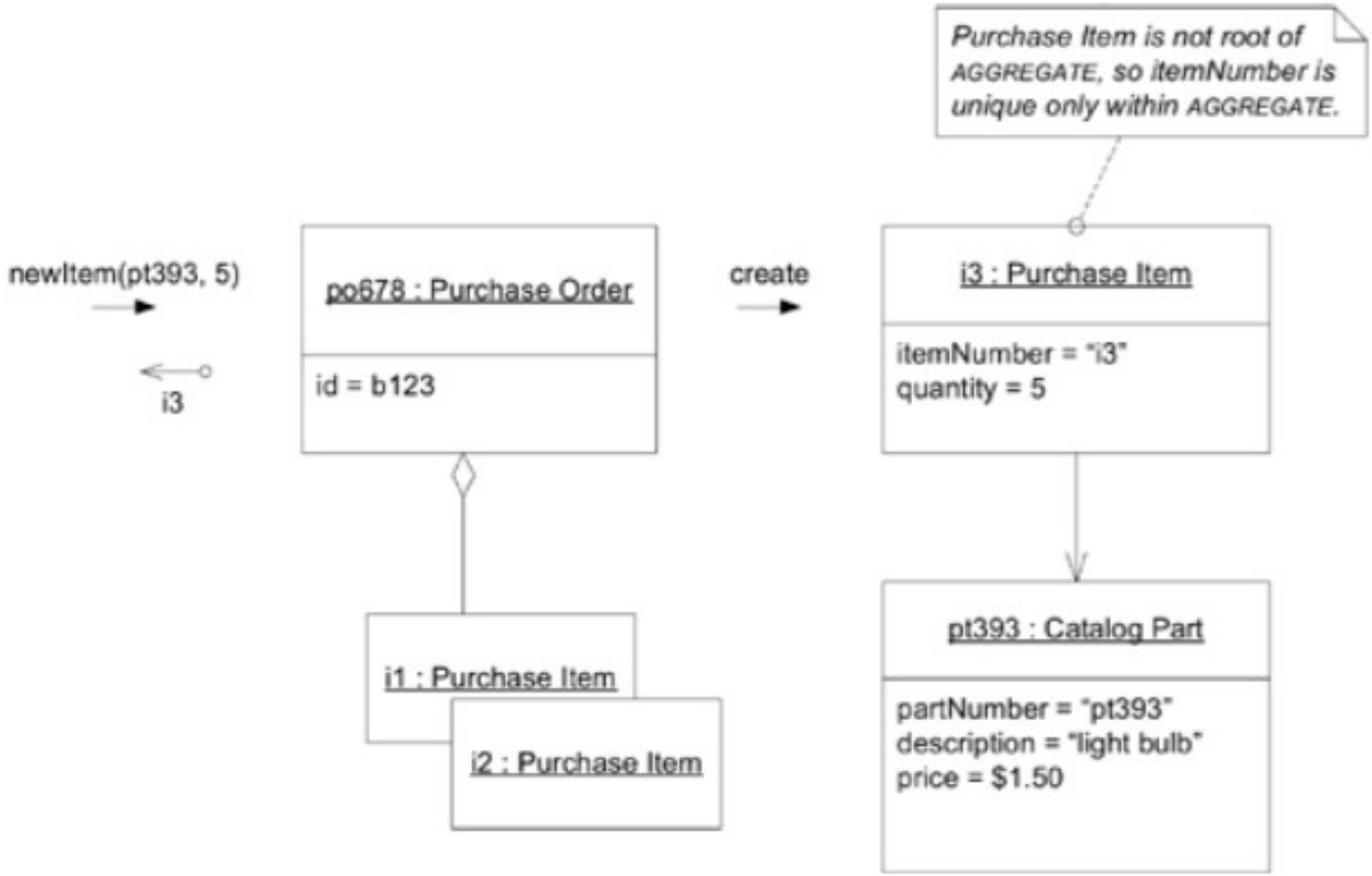
- Creation of an object can be a major operation in itself
- Complex assembly operations do not fit the responsibility of the created objects.
- Combining such responsibilities can produce ungainly designs that are hard to understand.
- Making the client direct construction muddies the design of the client, breaches encapsulation of the assembled object or AGGREGATE, and overly couples the client to the implementation of the created object.
- **Complex object creation is a responsibility of the domain layer**, yet **that task does not belong to the objects that express the model**.
- To solve this problem, we have to **add constructs to the domain design** that are not ENTITIES, VALUE OBJECTS, or SERVICES.

Basic Interactions with a Factory



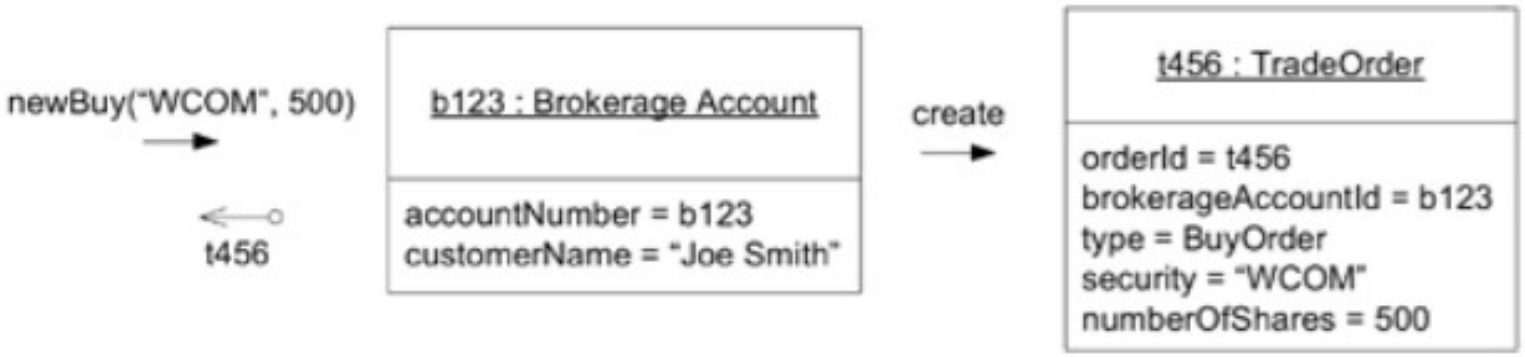
- **Shift the responsibility for creating instances** of complex objects and AGGREGATES **to a separate object**, which may itself **have no responsibility in the domain model but is still part of the domain design**.
- Provide an interface that encapsulates all complex assembly and that does not require the client to reference the concrete classes of the objects being instantiated.
- Create entire AGGREGATES as a piece, enforcing their invariants.

Choosing Factories and Their Sites: Factory Method (1/2)



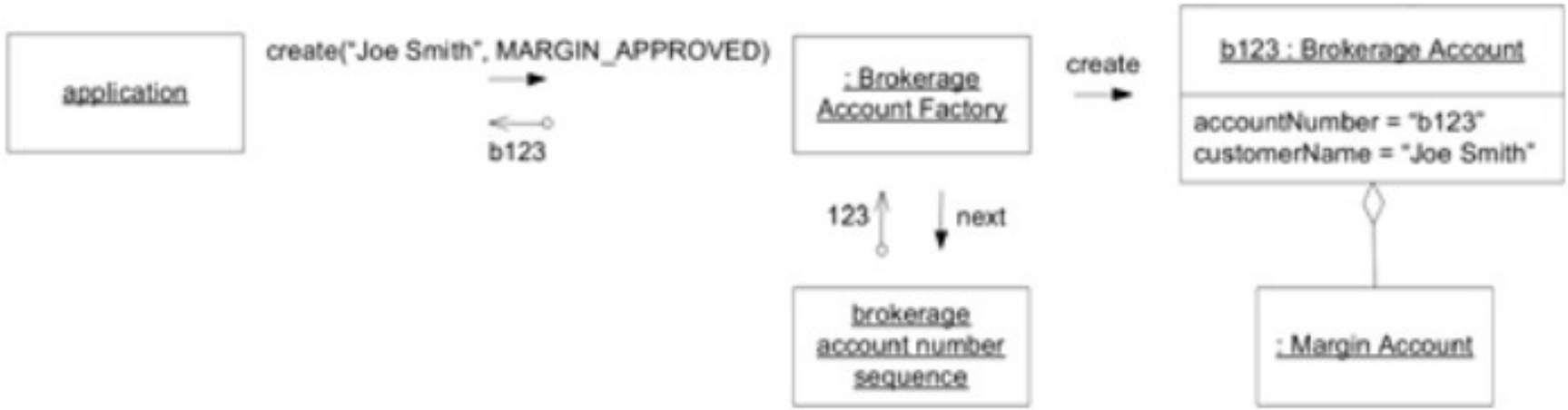
- You should **place the Factory where you want the control to be**.
- For example, if you need to add elements inside a preexisting Aggregate, you might create a **Factory Method on the root of the Aggregate**.

Choosing Factories and Their Sites: Factory Method (2/2)



- Here we see an example where **a Factory Method spawns an Entity that is not part of the same Aggregate**.

Choosing Factories and Their Sites: Standalone Factory



- Here we see an example of **a standalone Factory that builds the entire Aggregate**.
- In this case there is no other Entity that could act as a Factory for the Aggregate.

Factories: Designing the Interface

- **Each operation must be atomic**. You have to pass everything needed to create a complete product in a single interaction with the Factory. In case of failure you can throw and exception or return null.
- **The Factory will be coupled to its arguments**. You should use simple arguments when possible in order to avoid high dependencies to the parameters.
- **Use the abstract type of the arguments, not their concrete classes**. The Factory is coupled to the concrete class of the products; it does not need to be coupled to the concrete parameters also.

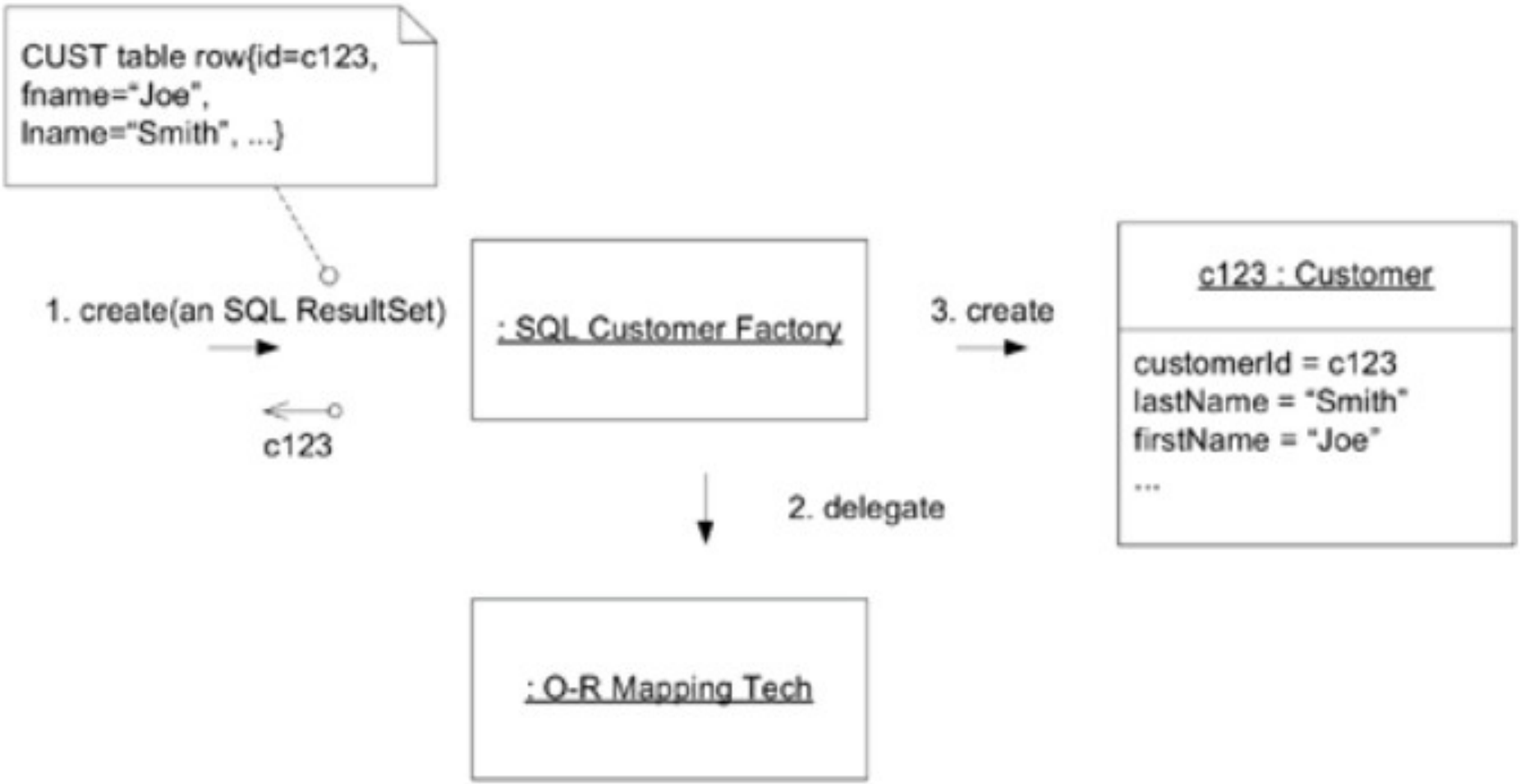
Note on Invariant Logic: A Factory can delegate invariant checking to the Product, for instance with Factory Methods attached to other domain objects. However, standalone Factories can enforce Aggregate rules that span many objects

Factories: Reconstituting Stored Objects

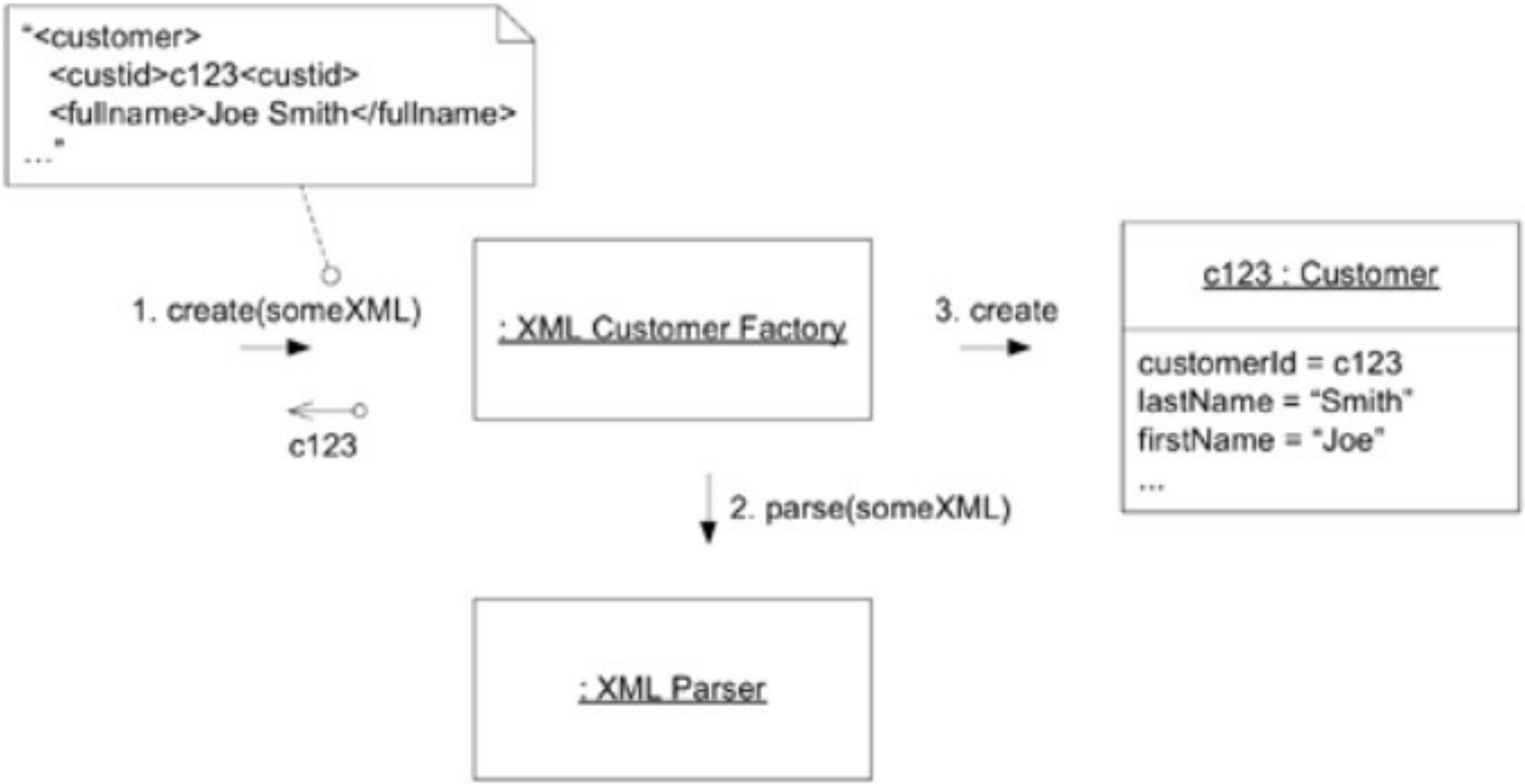
A Factory used for reconstitution is very similar to one used for creation, with two major differences:

- **An Entity Factory used for reconstitution does not assign a new tracking ID**. So, identifying attributes must be part of the input parameters in a Factory reconstituting a stored object.
- **A Factory reconstituting an object will handle violation of an invariant differently**.
  - During creation of a new object, a Factory should simply balk when an invariant isn't met, but a more flexible response may be necessary in reconstitution.
  - If an object already exists somewhere in the system (such as in the database), this fact cannot be ignored.
  - Yet we also can't ignore the rule violation. There has to be some strategy for repairing such inconsistencies, which can make reconstitution more challenging than the creation of new objects.

Factories: Example Reconstituting Stored Objects from a Database



Factories: Example Reconstituting Stored Objects from XML

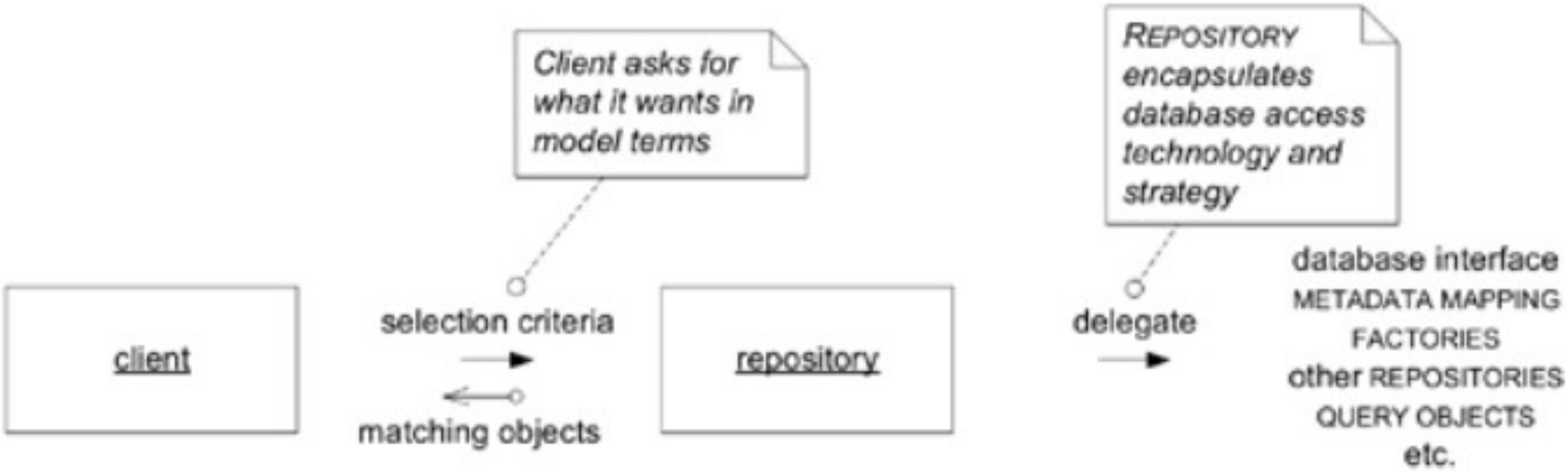


Repositories

Associations allow us to find an object based on its relationship to another. But we must have a starting point for a traversal to an Entity or Value in the middle of its life cycle.

- A subset of persistent objects must be globally accessible through a search based on object attributes.
- **Such access is needed for the roots of Aggregates that are not convenient to reach by traversal.**
- They are usually Entities, sometimes Value Objects with complex internal structure, and sometimes enumerated Values.
- Providing access to other objects muddies important distinctions. Free database queries can actually breach the encapsulation of domain objects and Aggregate. Exposure of technical infrastructure and database access mechanisms complicates the client and obscures the Model-Driven Design.

Repository: Example



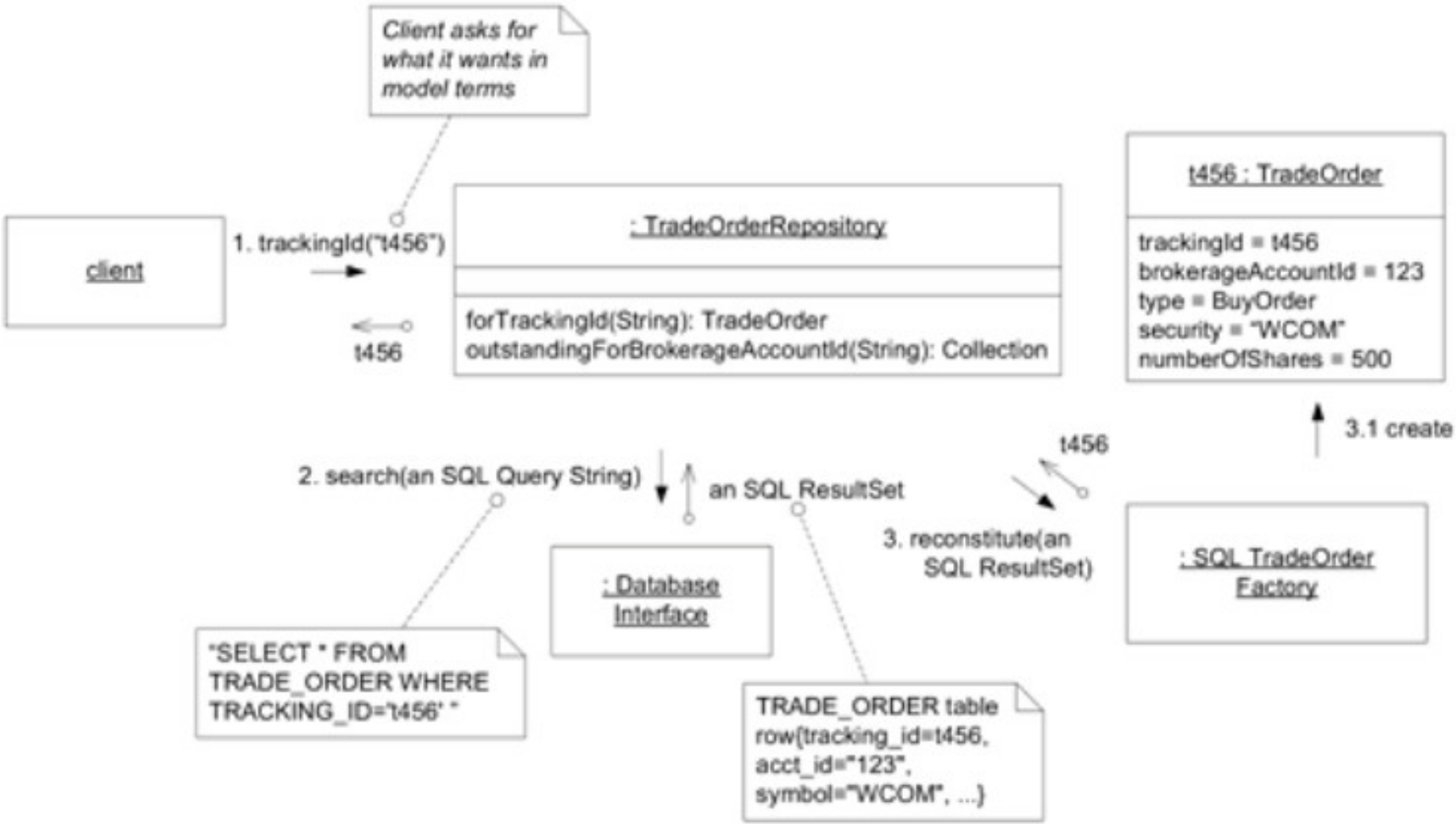
Repositories Advantages

- They present clients with a simple model for obtaining persistent objects and managing their life cycle.
- They decouple application and domain design from persistence technology, multiple database strategies, or even multiple data sources.
- They communicate design decisions about object access.
- They allow easy substitution of a dummy implementation, for use in testing (typically using an in-memory collection).

Querying a Repository

This examples shows an Specification-based query.

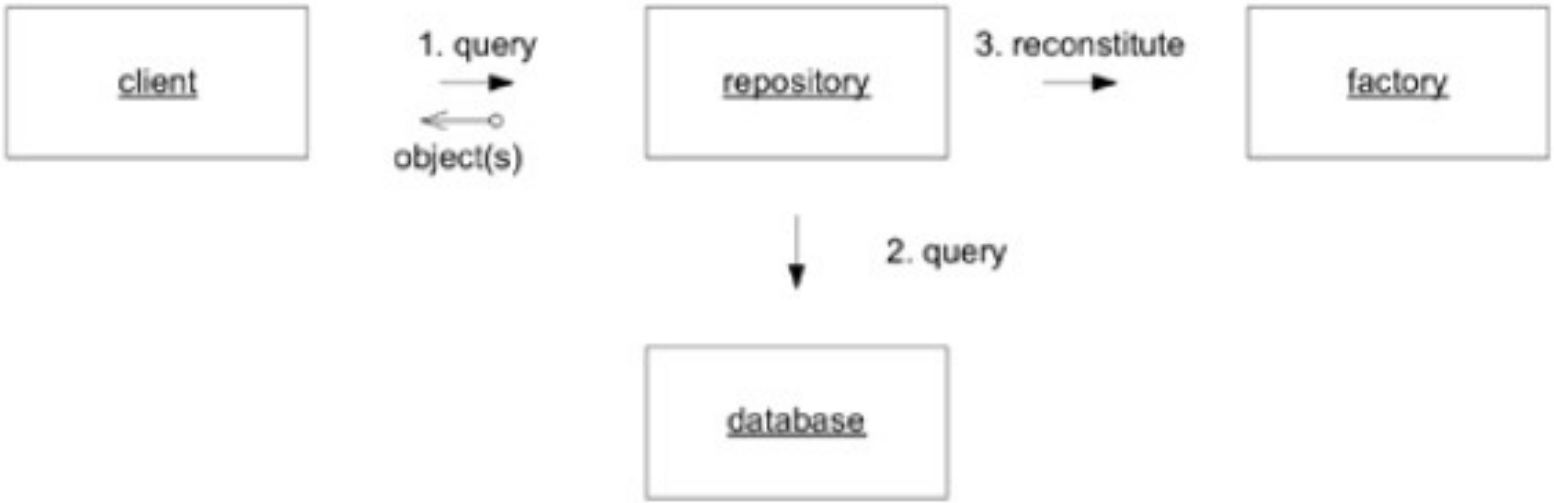
Implementing a Repository: Example



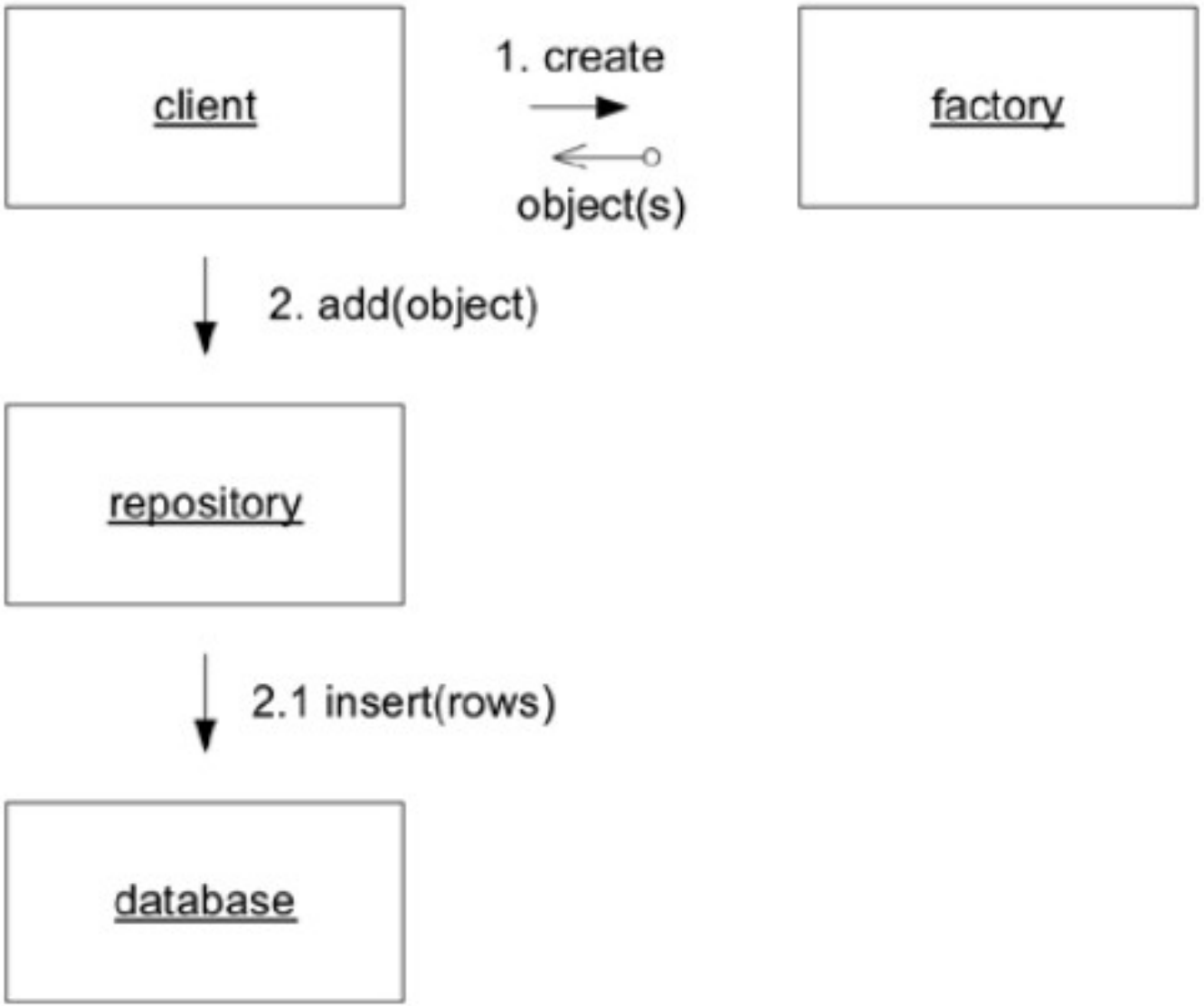
Guidelines for Implementing a Repository

- **Abstract the type**. Take advantage of the decoupling from the client. You have more freedom to change the implementation of a Repository than you would if the client were calling the mechanisms directly.
- **Leave transaction control to the client**. Although the Repository will insert into and delete from the database, it will ordinarily not commit anything. It is tempting to commit after saving, for example, but the client presumably has the context to correctly initiate and commit units of work. Transaction management will be simpler if the Repository keeps its hands off.

Repository and Factories: Reconstitute Objects



Repository and Factories: Store Objects

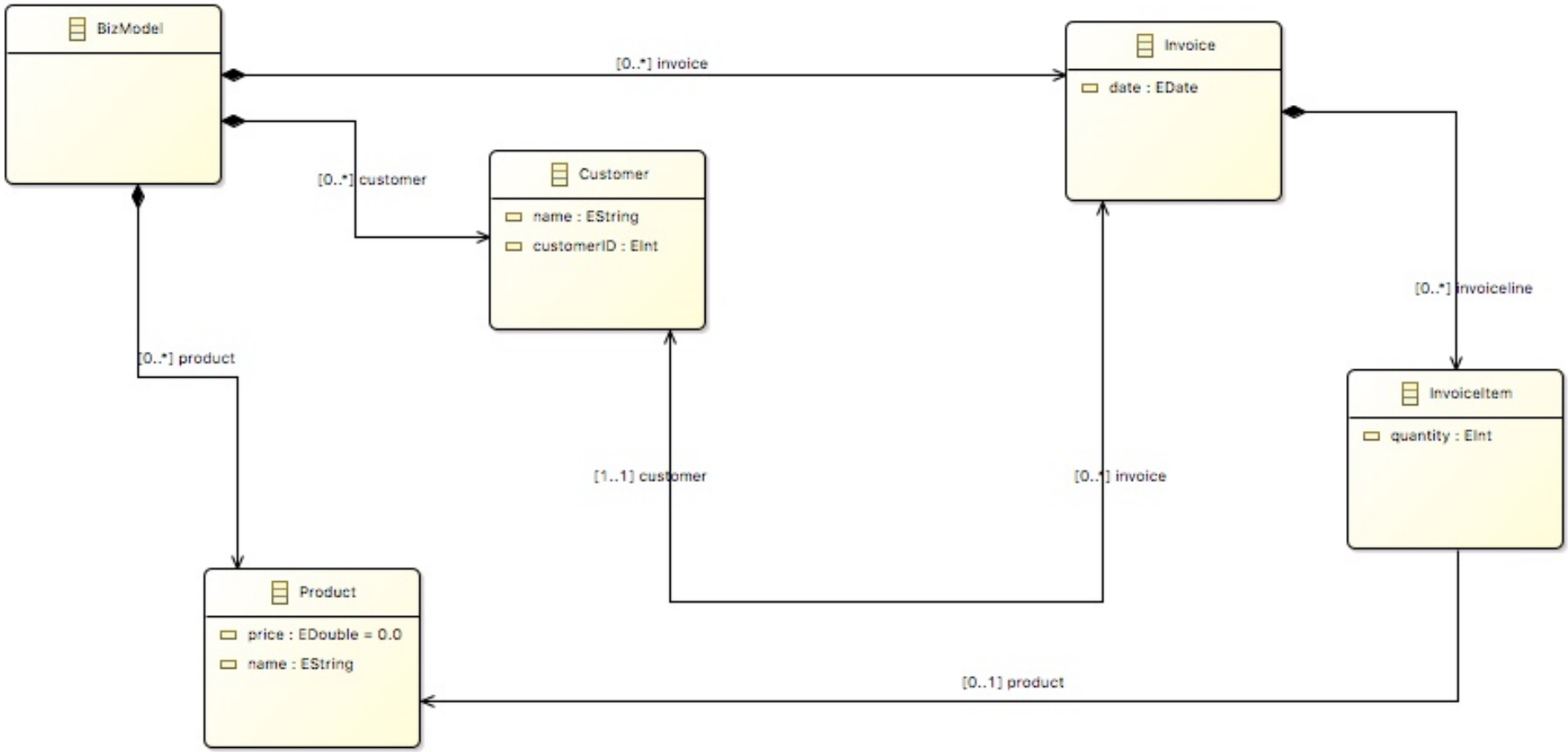


EMF Example....

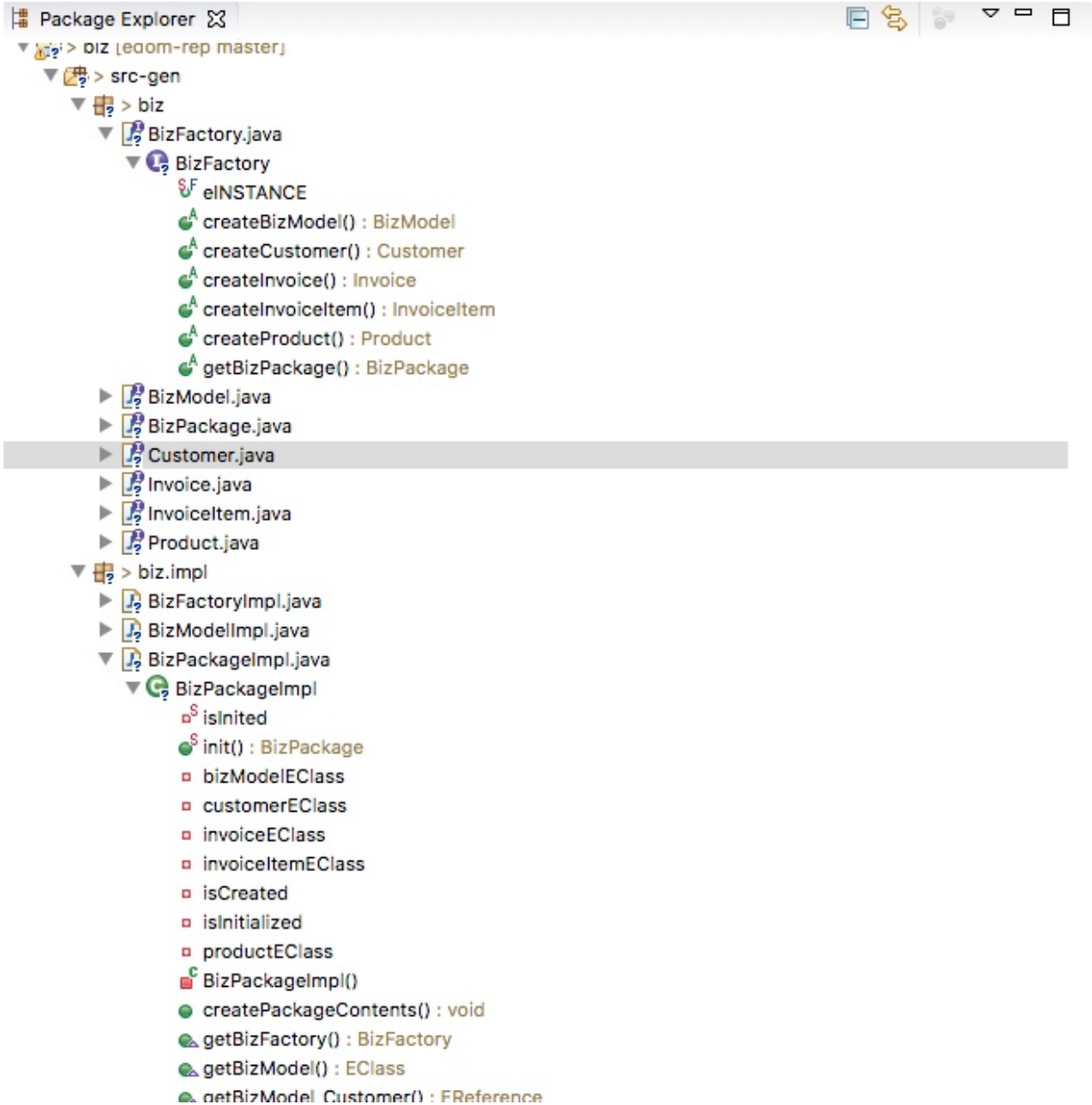
Using programmatically:

- **EMF Factories:** Creating instances of metamodel EClasses.
- **EMF Resource Sets:** Persistence instances (i.e., loading and saving models).

EMF Example: Consider the following metamodel



EMF Example: The generated code...includes a Factory and a Package



EMF Example: Accessing the model programmatically

- If you create a EMF project with the previous metamodel and generate the model code...
- You can then use code as the one presented in the next slides to manipulate the model...

EMF Example: The **biz.Run** class (1/4)

```
1  package biz;
2
3  import java.io.File;
4  import java.io.IOException;
5  import java.util.Collections;
6
7  import org.eclipse.emf.common.util.URI;
8  import org.eclipse.emf.ecore.EObject;
9  import org.eclipse.emf.ecore.resource.Resource;
10 import org.eclipse.emf.ecore.resource.ResourceSet;
11 import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
12 import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl;
13
14 public class Run {
15
16    private static void writeModel() {
17       //-------------------------------------------------------
18       // Programmatically editing the model instance...
19       BizFactory factory = BizFactory.eINSTANCE;
20
21       BizModel model = factory.createBizModel();
22
23       // Customer 1
24       Customer customer = factory.createCustomer();
25       customer.setName("Alexandre Braganca");
26       model.getCustomer().add(customer);
27
28       // Customer 2
29       Customer customer2 = factory.createCustomer();
30       customer2.setName("Joao Mendes");
31       model.getCustomer().add(customer2);
```

EMF Example: The **biz.Run** class (2/4)

```
32
33          //-------------------------------------------------------
34          // Saving the model...
35          // Create a resource set.
36          ResourceSet resourceSet = new ResourceSetImpl();
37
38          // Register the default resource factory -- only needed for stand-alone, i.e, running outside
                eclipse!
39          resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap()
40              .put(Resource.Factory.Registry.DEFAULT_EXTENSION, new XMIResourceFactoryImpl());
41
42          // Get the URI of the model file.
43          URI fileURI = URI.createFileURI(new File("instances/mybiz.xmi").getAbsolutePath());
44
45          // Create a resource for this file.
46          Resource resource = resourceSet.createResource(fileURI);
47
48          // Add the book and writer objects to the contents.
49          resource.getContents().add(model);
50
51          // Save the contents of the resource to the file system.
52          try
53          {
54             resource.save(Collections.EMPTY_MAP);
55          }
56          catch (IOException e) {}
57      }
```

EMF Example: The **biz.Run** class (3/4)

```
59      private static void readModel() {
60         //-------------------------------------------------------
61         // Loading the model...
62         // Create a resource set.
63         ResourceSet resourceSet2 = new ResourceSetImpl();
64
65         // Register the default resource factory -- only needed for stand-alone!
66         resourceSet2.getResourceFactoryRegistry().getExtensionToFactoryMap()
67            .put(Resource.Factory.Registry.DEFAULT_EXTENSION, new XMIResourceFactoryImpl());
68
69         // Register the package -- only needed for stand-alone!
70         BizPackage libraryPackage = BizPackage.eINSTANCE;
71
72         // Get the URI of the model file.
73         URI fileURI2 = URI.createFileURI(new File("instances/mybiz.xmi").getAbsolutePath());
74
75         // Demand load the resource for this file.
76         Resource resource2 = resourceSet2.getResource(fileURI2, true);
77
78         // Print the contents of the resource to System.out.
79         try {
80            resource2.save(System.out, Collections.EMPTY_MAP);
81         }
82         catch (IOException e) {}
```

EMF Example: The **biz.Run** class (4/4)

```
84          // Accessing the elements in the Model...
85          // resource2.
86          for (EObject obj:resource2.getContents()) {
87             if (BizModel.class.isInstance(obj)) {
88                BizModel bizModel=(BizModel)obj;
89
90                System.out.println("Model found:");
91                for (Customer c:bizModel.getCustomer()) {
92                   System.out.println(" >Customer: "+c.getName());
93                }
94             }
95          }
96       }
97
98       public static void main(String[] args) {
99          writeModel();
100
101         readModel();
102      }
103
104   }
```

Further readings

For EMF

- https://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.emf.doc%2Freferences%2Foverview%2FEMF.html

For Domain Driven Design

- https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/microservice-domain-model