7 Elaboration Iteration 3 Intermediate Topics

庞雄文

Tel：18620638848

Wechat: augepang

QQ: 443121909

Contents

- Part 5: Elaboration Iteration 3 Intermediate Topics
  - UML Activity Diagrams and Modeling
  - UML State Machine Diagrams and Modeling
  - Relating Use Cases
  - Architectural Analysis and Refinement
  - More Object Design with GOF Patterns
  - Designing a Persistence Framework with Patterns

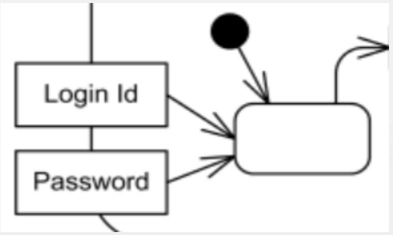
From iterator 2 to iterator 3

- Iterator 3
  - **NextPos**
    - Provide failover to local services when the remote services cannot be accessed. For example, if the remote product database can't be accessed, use a local version with cached data.
    - Provide support for POS device handling, such as the cash drawer and coin dispenser.
    - Handle credit payment authorization.
    - Support for persistent objects.
  - Monogame
    - implement a basic, key scenario of the Play Monopoly Game use case
    - When a player lands on a Lot, Railroad or Utility square, the following logic applies…

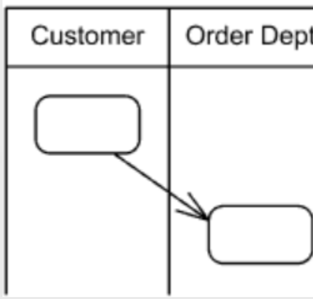
UML Activity Diagrams and Modeling

- **activity diagram**
  - shows sequential and parallel activities in a process.
  - useful for modeling business processes, workflows, data flows, and complex algorithms
    - show both control flow and data flow

- Elements

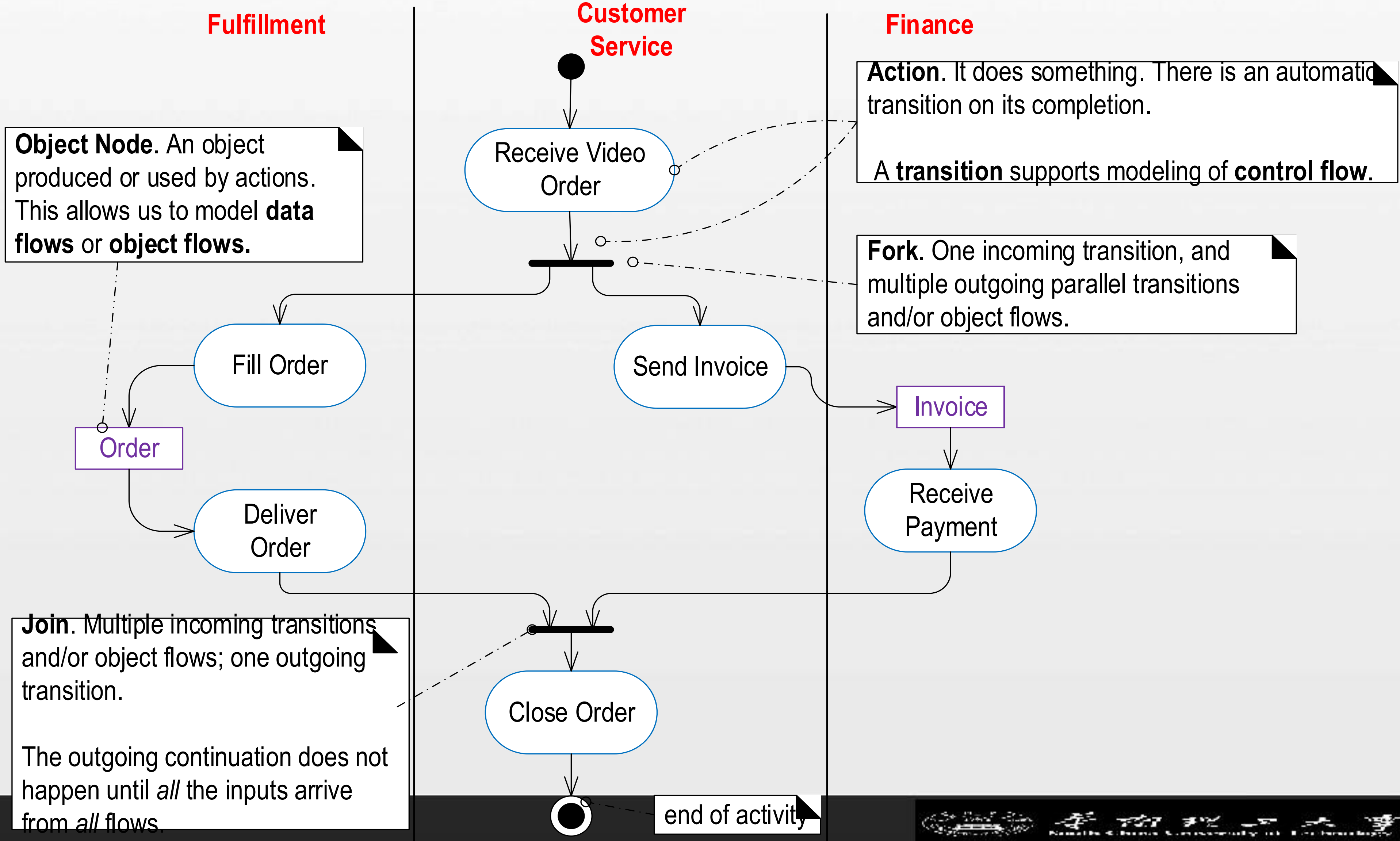


  - **Activity**
    - parameterized **behavior** represented as coordinated flow of **actions**.
    - May **be** **action, object, control**
    - could have pre- and post-condition constraints
  - Partition
    - **activity group** for actions that have some common characteristic
    - organizational units or business actors in a business model.
  - Start node and end node ● ◉



UML Activity Diagrams and Modeling

- activity diagram example for business model



Action. It does something. There is an automatic transition on its completion.

A **transition** supports modeling of **control flow**.

Object Node. An object produced or used by actions. This allows us to model **data flows** or **object flows.**

Fork. One incoming transition, and multiple outgoing parallel transitions and/or object flows.

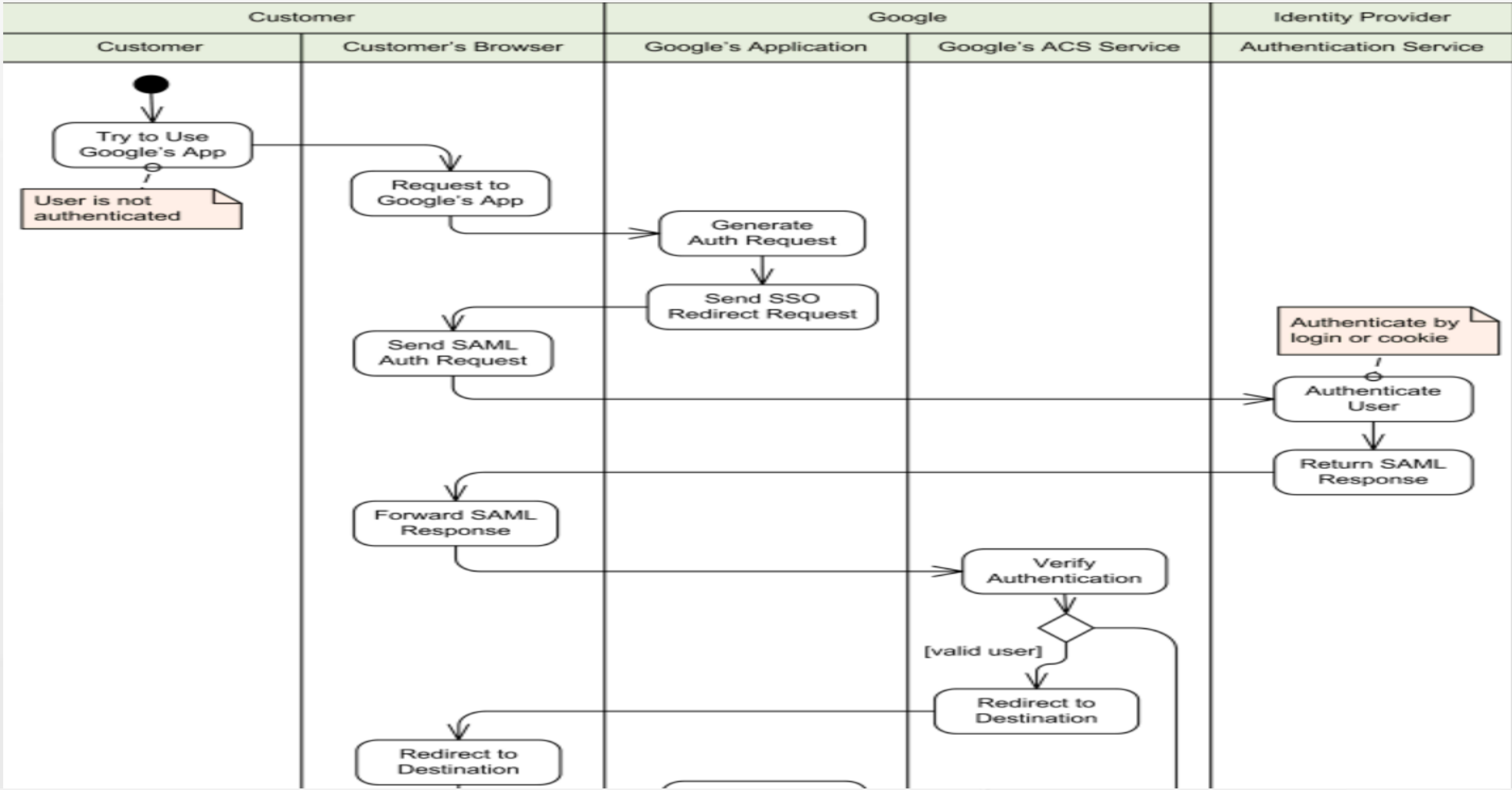
Join. Multiple incoming transitions and/or object flows; one outgoing transition.

The outgoing continuation does not happen until *all* the inputs arrive from *all* flows.

end of activity

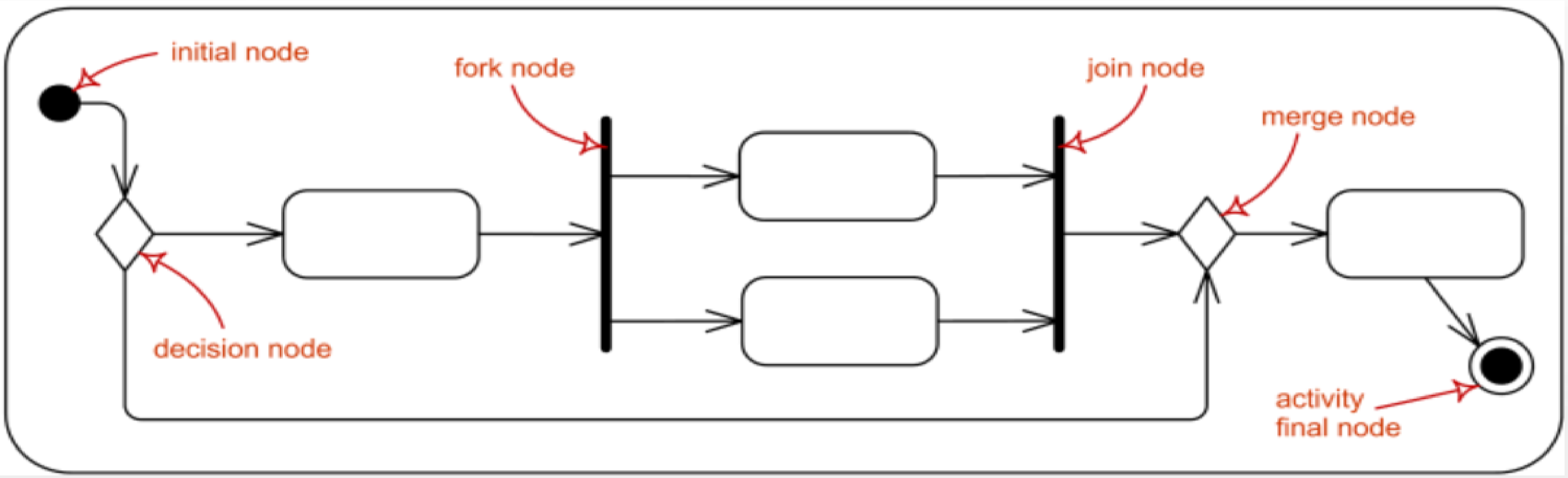
UML Activity Diagrams and Modeling

- **activity diagram example for Algorithm Modeling**
  - **Single Sign-On** (SSO) to Google Apps.

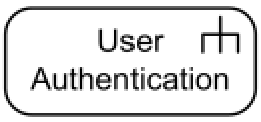


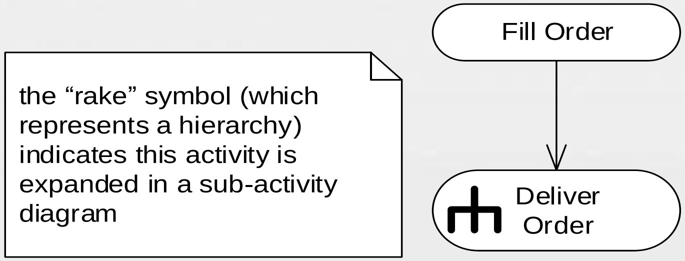
UML Activity Diagrams and Modeling

- Control node



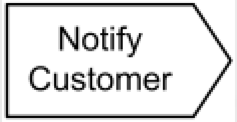
- Special Actions
  - Call activity action



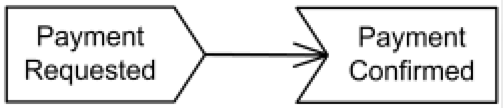


the “rake” symbol (which represents a hierarchy) indicates this activity is expanded in a sub-activity diagram

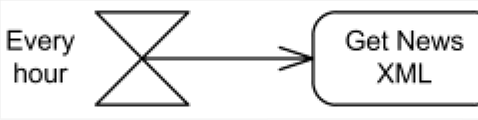
  - Send signal action



  - Accept Signal Action

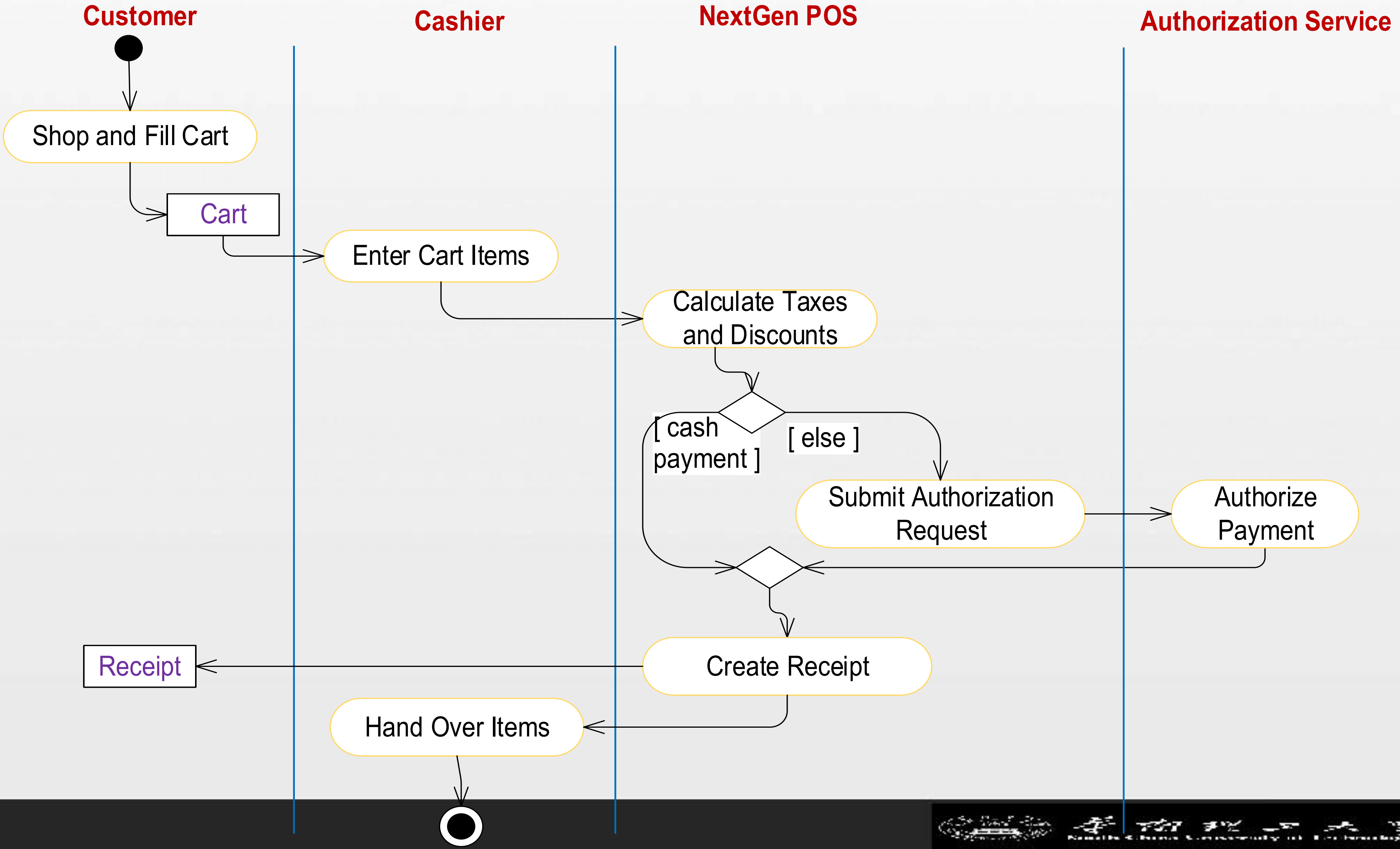


  - Wait Time Action



UML Activity Diagrams and Modeling

- **Example: NextGen Activity Diagram**



Customer

Cashier

NextGen POS

Authorization Service

Shop and Fill Cart

Cart

Enter Cart Items

Calculate Taxes and Discounts

[cash payment]

[else]

Submit Authorization Request

Authorize Payment

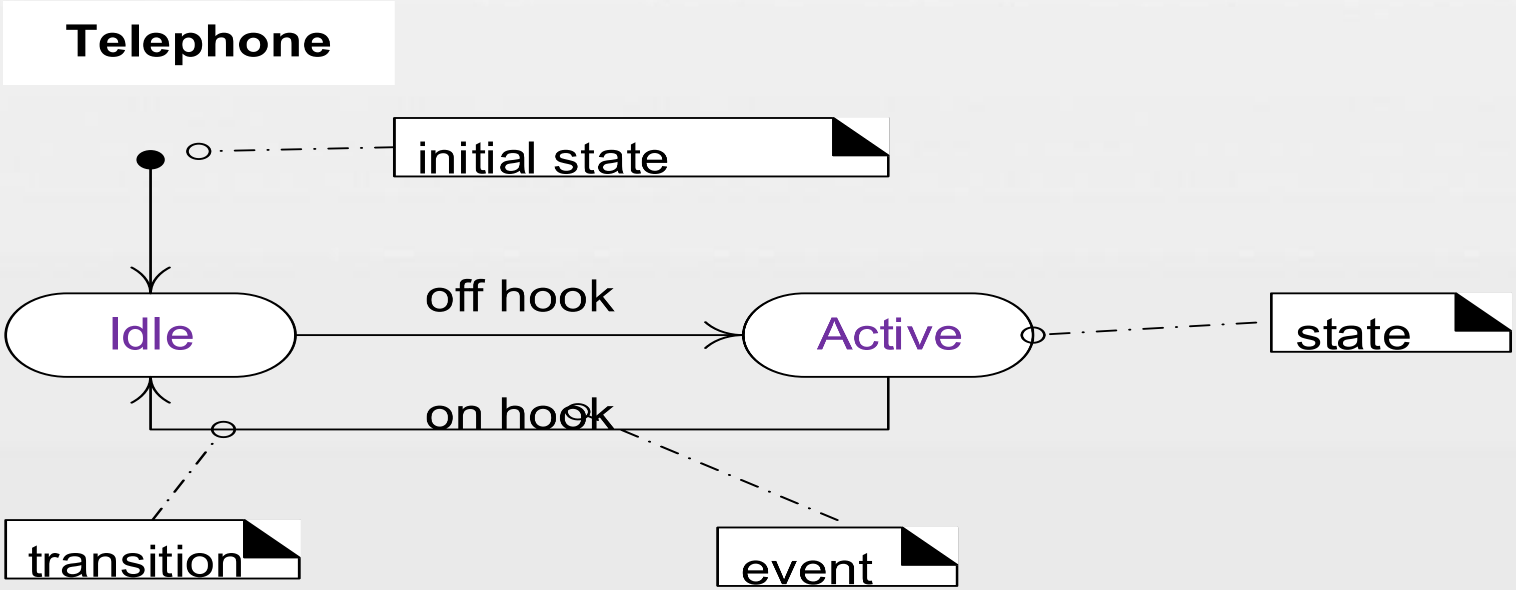
Create Receipt

Receipt

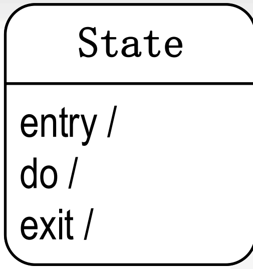
Hand Over Items

State Machine Diagrams and Modeling

- **state machine diagram**
  - **behavior diagram** which shows discrete behavior of a part of designed system through finite state transitions
  - shows the **lifecycle of an object**: what events it experiences, its transitions, and the states it is in between these events
  - Only for core object, key object and complex object
- Example

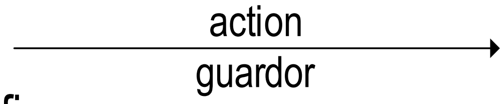


State Machine Diagrams and Modeling

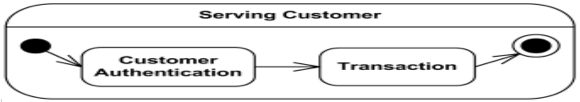


Elements of State Machine Diagrams

- **State**
  - the condition of an object at a moment in time the time between event
    - entry (behavior performed upon entry to the state)
    - do (ongoing behavior, performed as long as the element is in the state)
    - exit (behavior performed upon exit from the state)
- **Event,** a significant or noteworthy occurrence
- **Transition** →
  - a relationship between two states that indicates that when an event occurs, the object moves from the prior state to the subsequent state
- **Transition Actions and Guards**



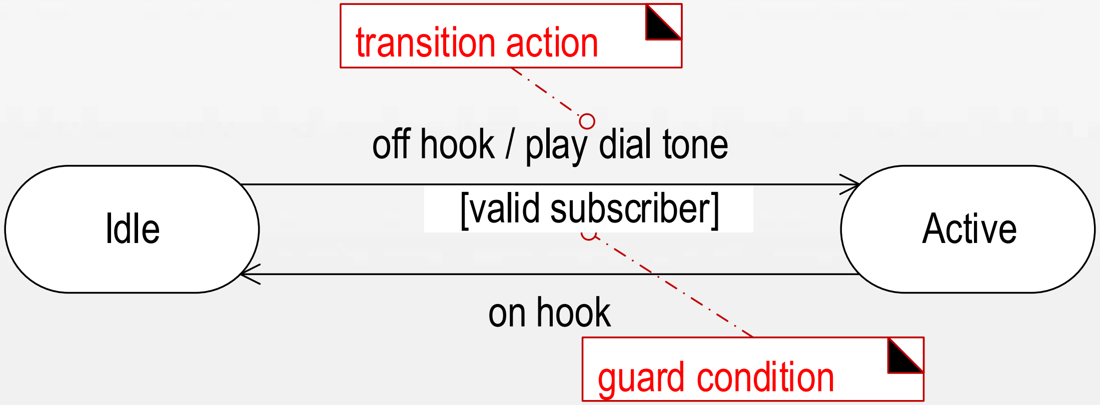
  - A transition can cause an action to fire
  - a conditional guarder Boolean test. The transition only occurs if the test passes
- Nest**ed States**

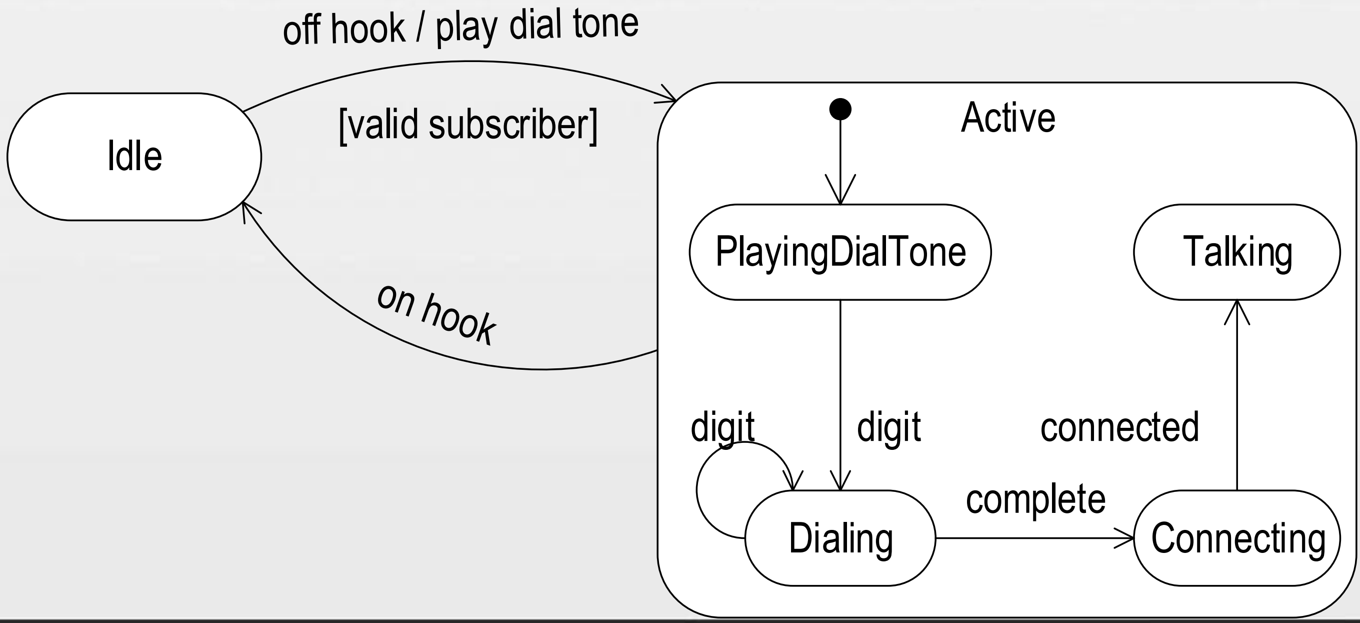


  - state allows nesting to contain sub-states; a sub-state inherits the transitions of its super-state

State Machine Diagrams and Modeling

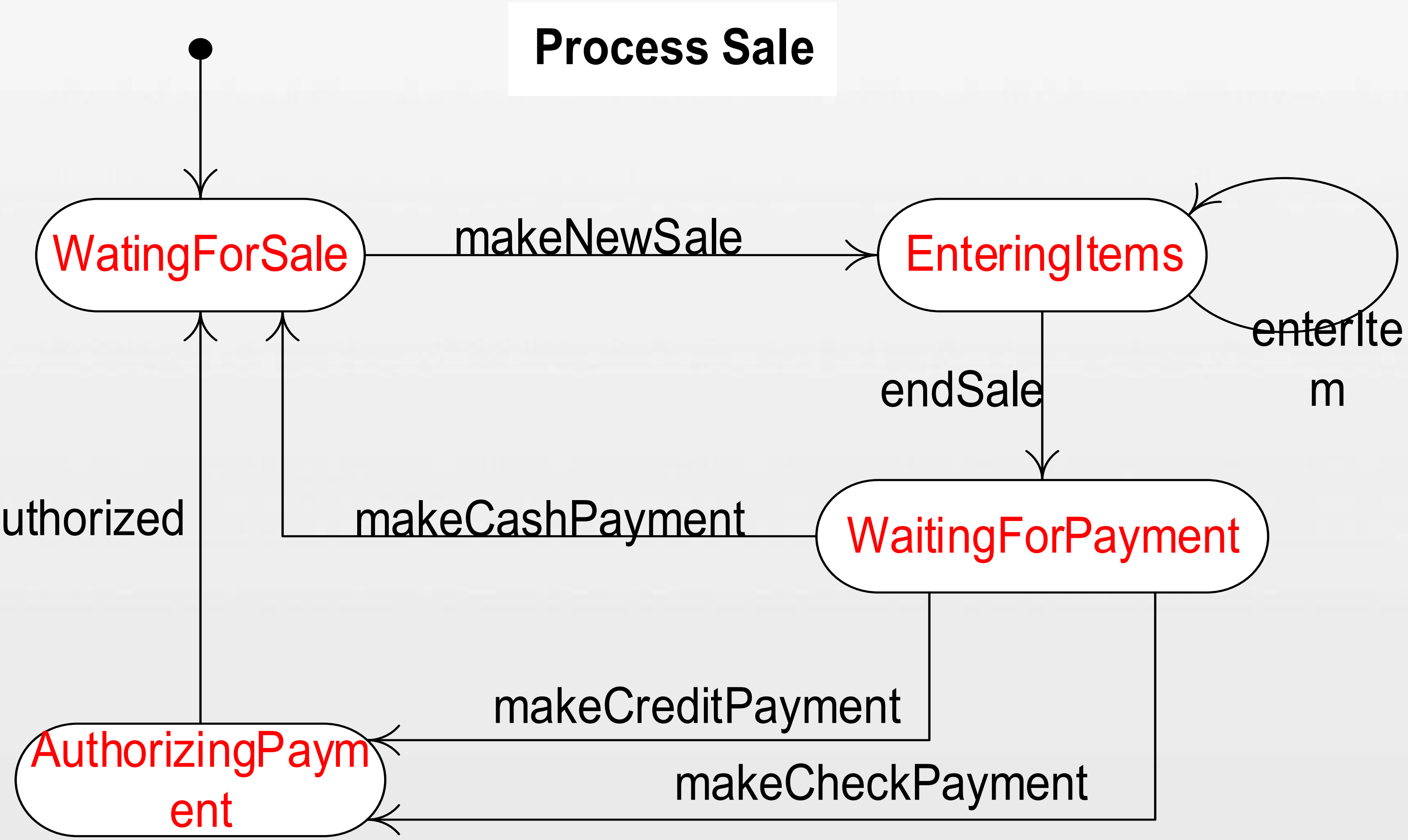
- **Elements of State Machine Diagrams**





State Machine Diagrams and Modeling

- Example: NextGen Use Case State Machine Diagram



Process Sale

WatingForSale

makeNewSale

EnteringItems

enterItem

endSale

authorized

makeCashPayment

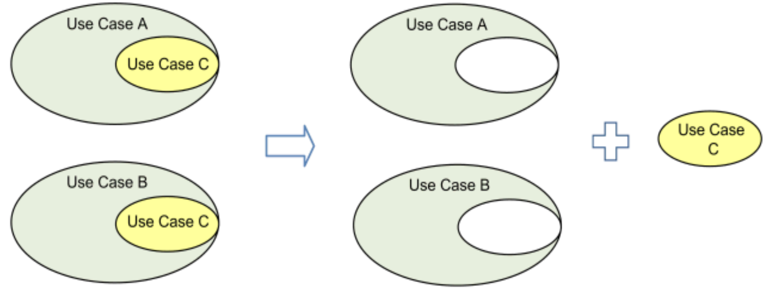
WaitingForPayment

makeCreditPayment

AuthorizingPayment

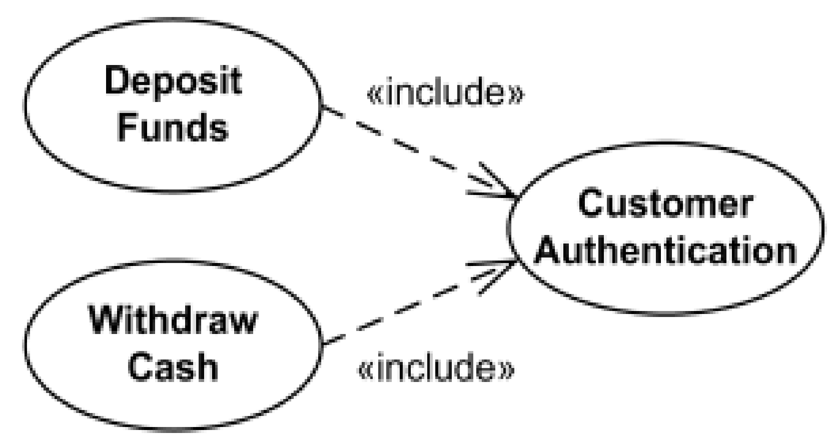
makeCheckPayment

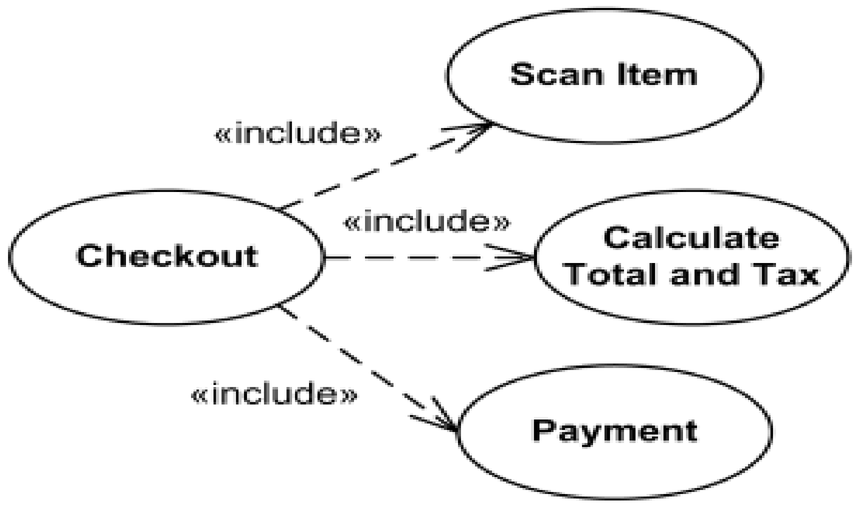
Relating Use Cases



The include Relationship

- When two or more use cases have some **common behavior**, this common part could be extracted into a separate use case to be included back by the use cases with the UML **include** relationship.
- **A use case is very complex and long, and separating it into subunits aids comprehension**.
- dashed arrow with an open arrowhead from the **including (base) use case** to the **included (abstract) use case**
- including use case becomes incomplete by itself and requires included use cases to be complete

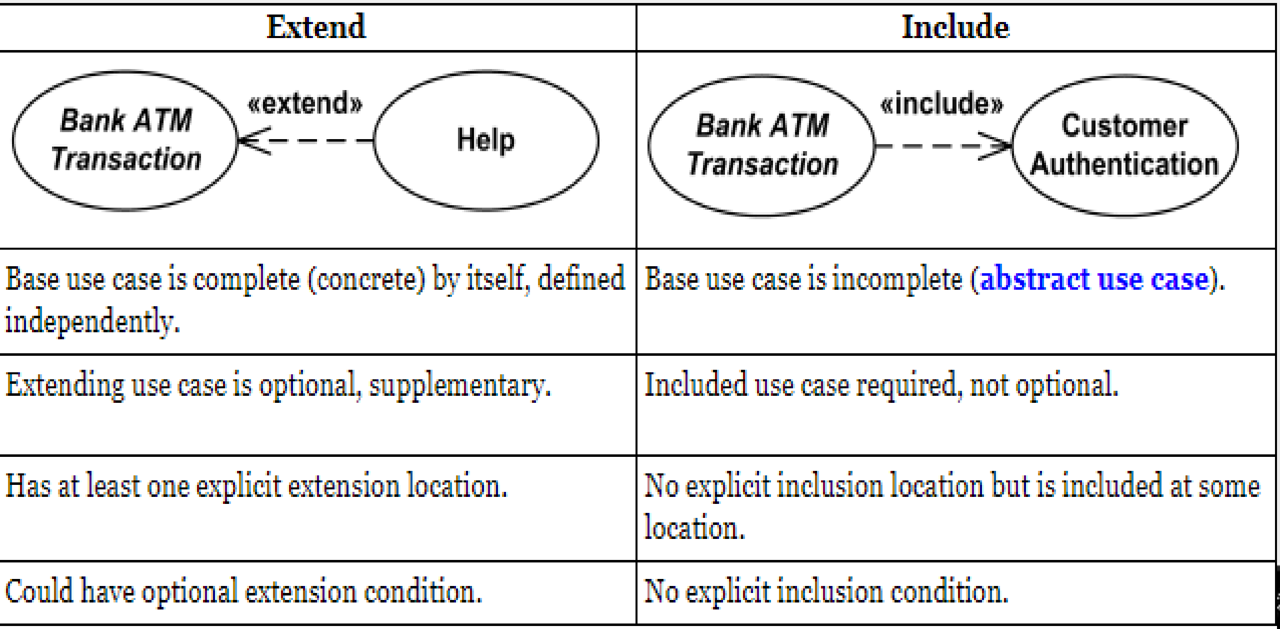




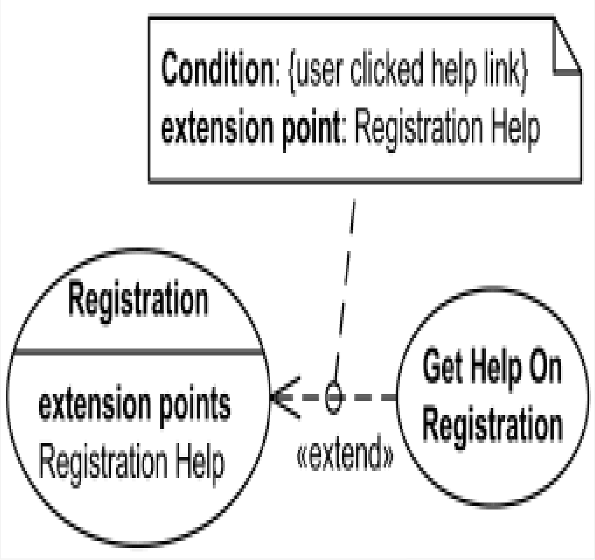
Relating Use Cases

■ The extend Relationship

- Define how and when the behavior in usually supplementary (optional) **extending(additional) use case** can be inserted into the **behavior** defined in the **extended(base) use case**.
  - **extension points**
  - **Conditions**
- an open arrowhead directed from the **extending(additional) use case** to the **extended (base) use case**

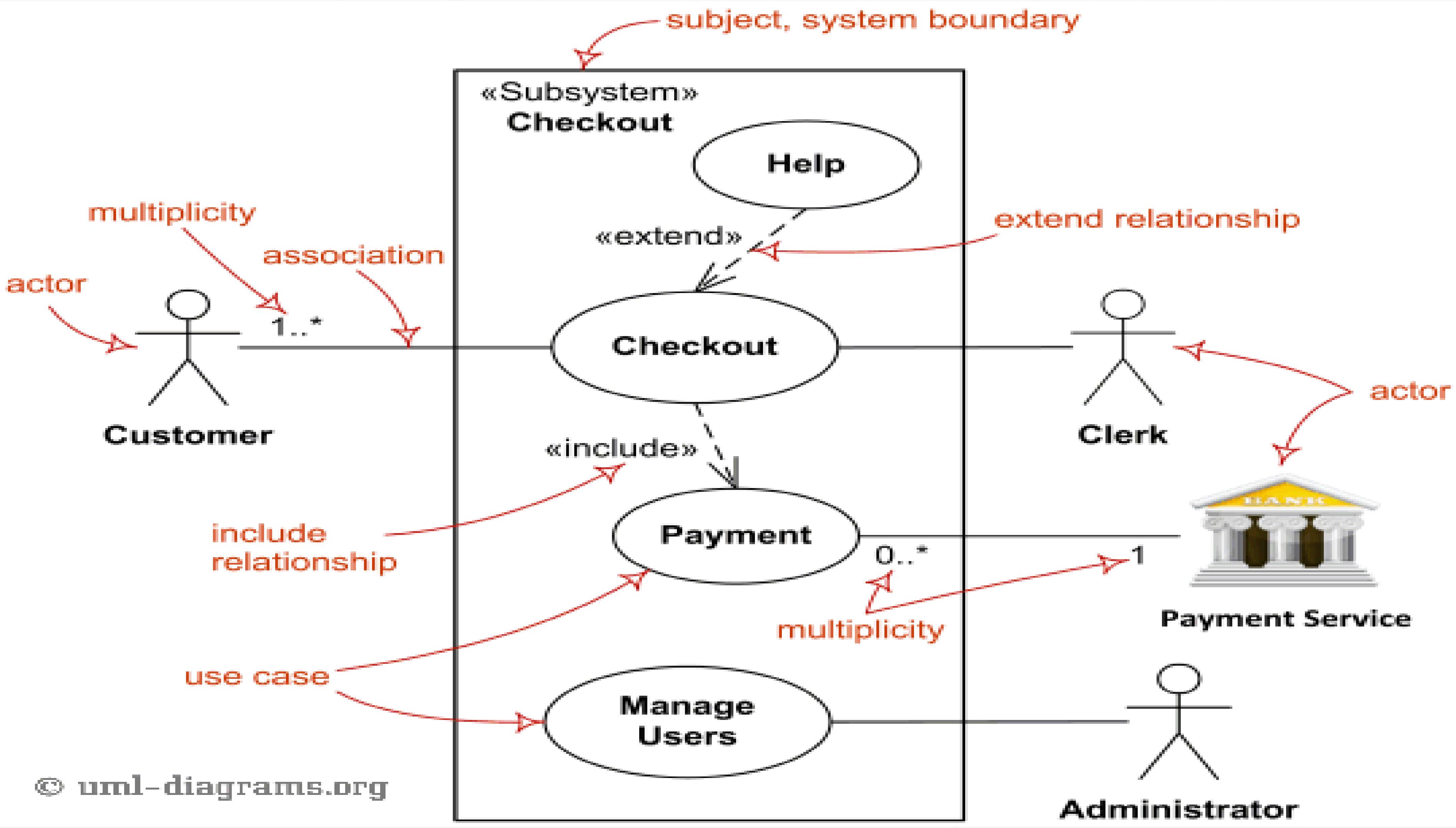


Extend	Include
Bank ATM Transaction ←«extend»-- Help	Bank ATM Transaction --«include»→ Customer Authentication
Base use case is complete (concrete) by itself, defined independently.	Base use case is incomplete (**abstract use case**).
Extending use case is optional, supplementary.	Included use case required, not optional.
Has at least one explicit extension location.	No explicit inclusion location but is included at some location.
Could have optional extension condition.	No explicit inclusion condition.



Relating Use Cases

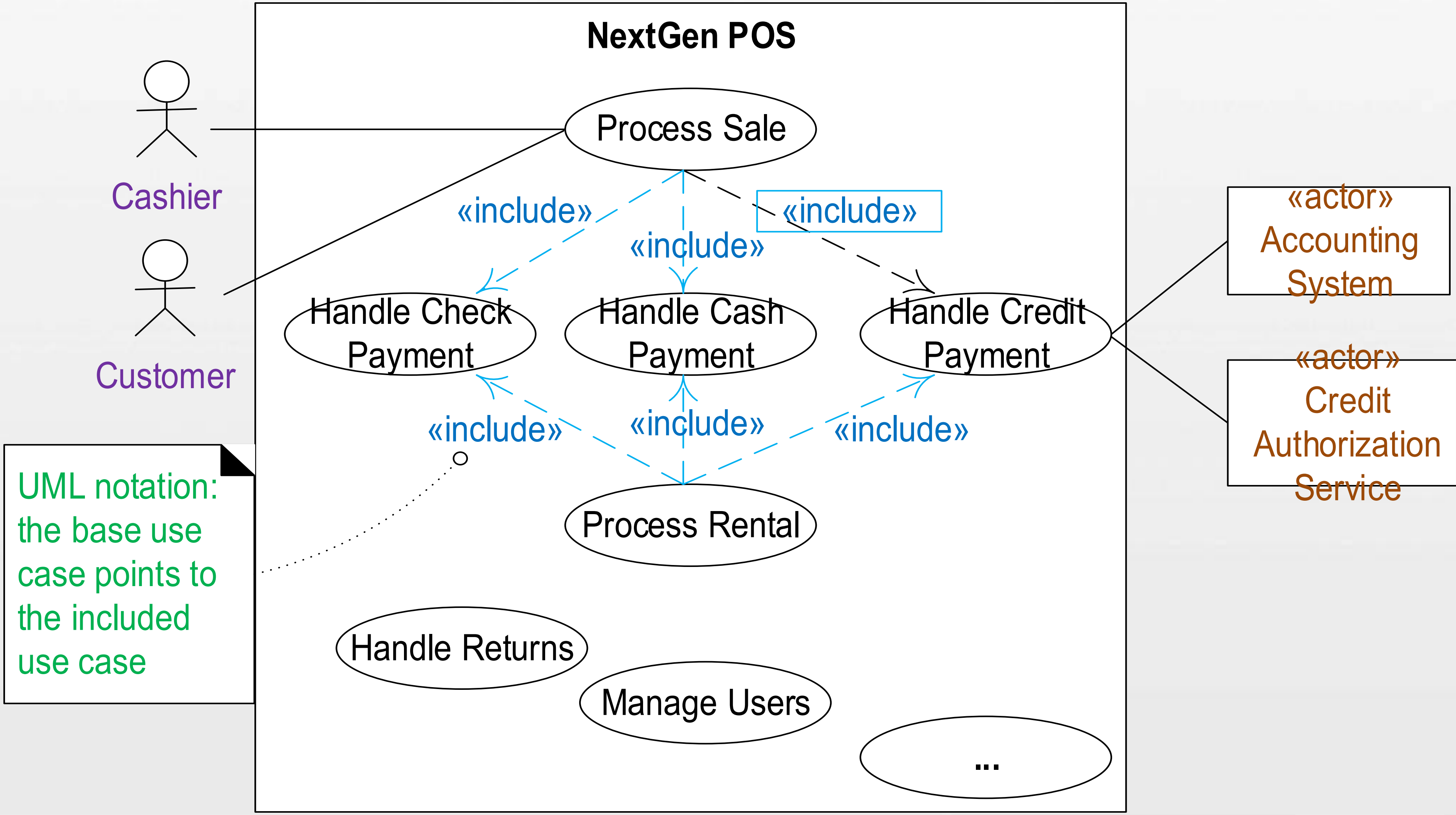
- **Example**



© uml-diagrams.org

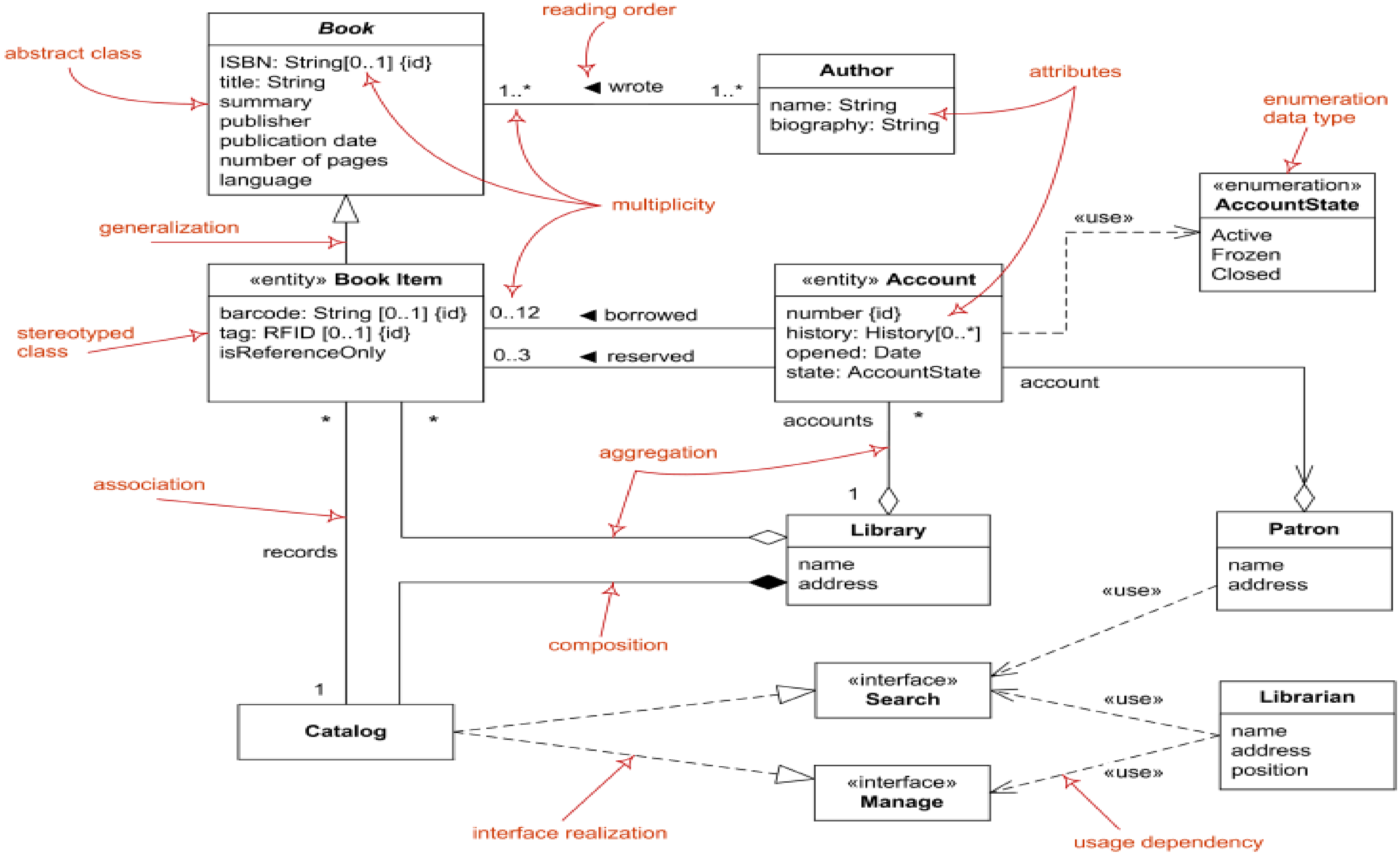
Relating Use Cases

- NextGen POS Use case Diagram



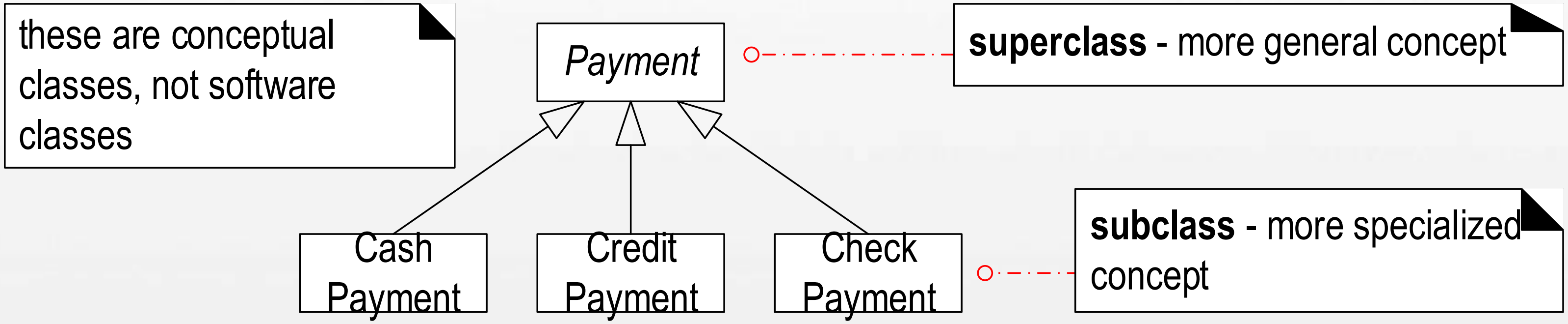
UML notation: the base use case points to the included use case

Domain Model Refinement



Domain Model Refinement

Generalization



these are conceptual classes, not software classes

Payment

superclass - more general concept

Cash Payment

Credit Payment

Check Payment

subclass - more specialized concept

When to Define a Conceptual Subclass?

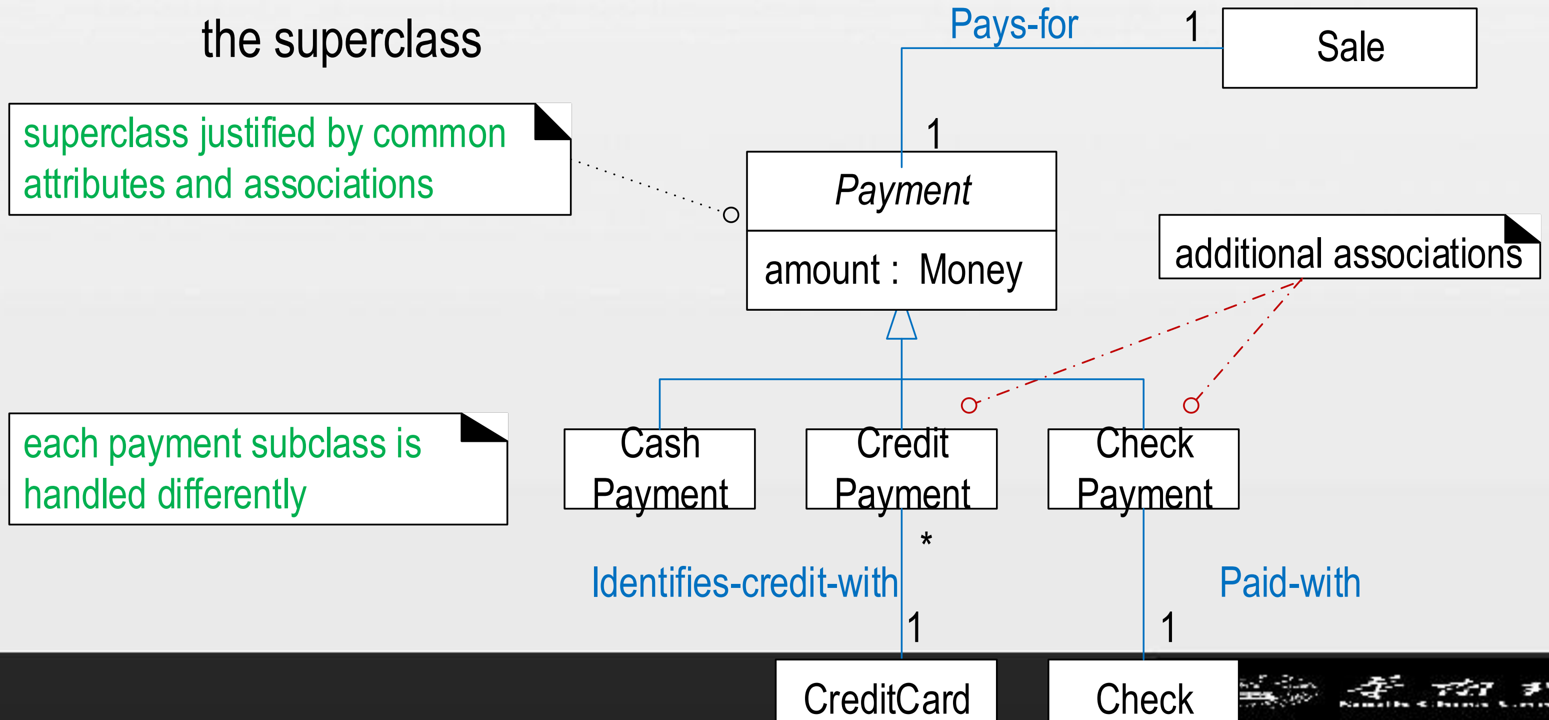
- The subclass has additional attributes of interest.
- The subclass has additional associations of interest.
  - CreditPayment is associated with a CreditCard.
- The subclass concept is operated on, handled, reacted to, or manipulated differently than the superclass or other subclasses
  - CreditPayment is handled differently than others in how it is authorized.
- The subclass concept represents an animate thing (for example, animal, robot) that behaves differently than the superclass or other subclasses

Domain Model Refinement

Generalization

When to Define a Conceptual Superclass?

- The potential conceptual subclasses represent variations of a similar concept.
- The subclasses will conform to the 100% and Is-a rules.
- All subclasses have the same attribute that can be factored out and expressed in the superclass.
- All subclasses have the same association that can be factored out and related to the superclass



superclass justified by common attributes and associations

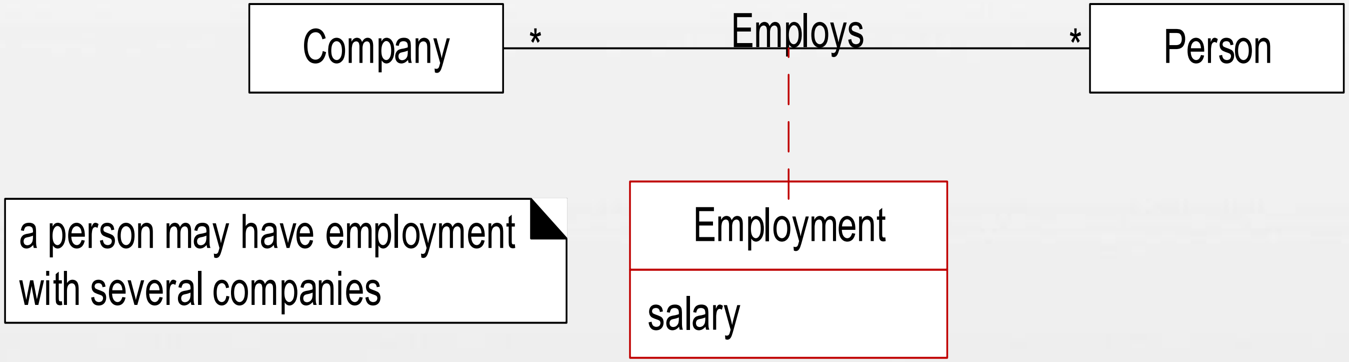
additional associations

each payment subclass is handled differently

Domain Model Refinement

Association Classes

- An attribute is related to an association.
- Instances of the association class have a lifetime dependency on the association.
- many-to-many association between two concepts and information associated with the association itself.



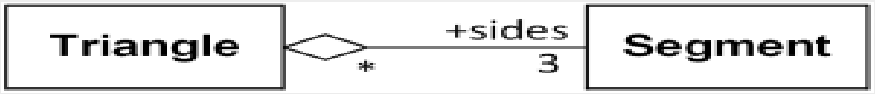
Company * Employs * Person

a person may have employment with several companies

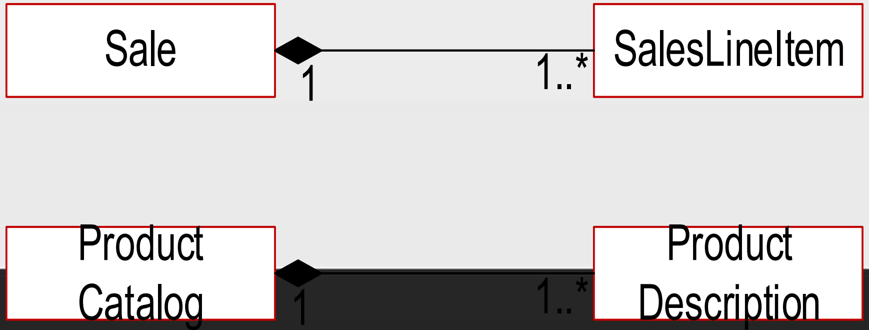
Employment

salary

Aggregation and Composition-- whole-part relationships

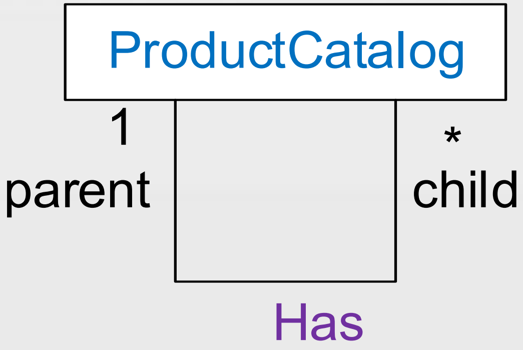


Triangle * +sides 3 Segment



Sale 1 1..* SalesLineItem

Product Catalog 1 1..* Product Description



ProductCatalog

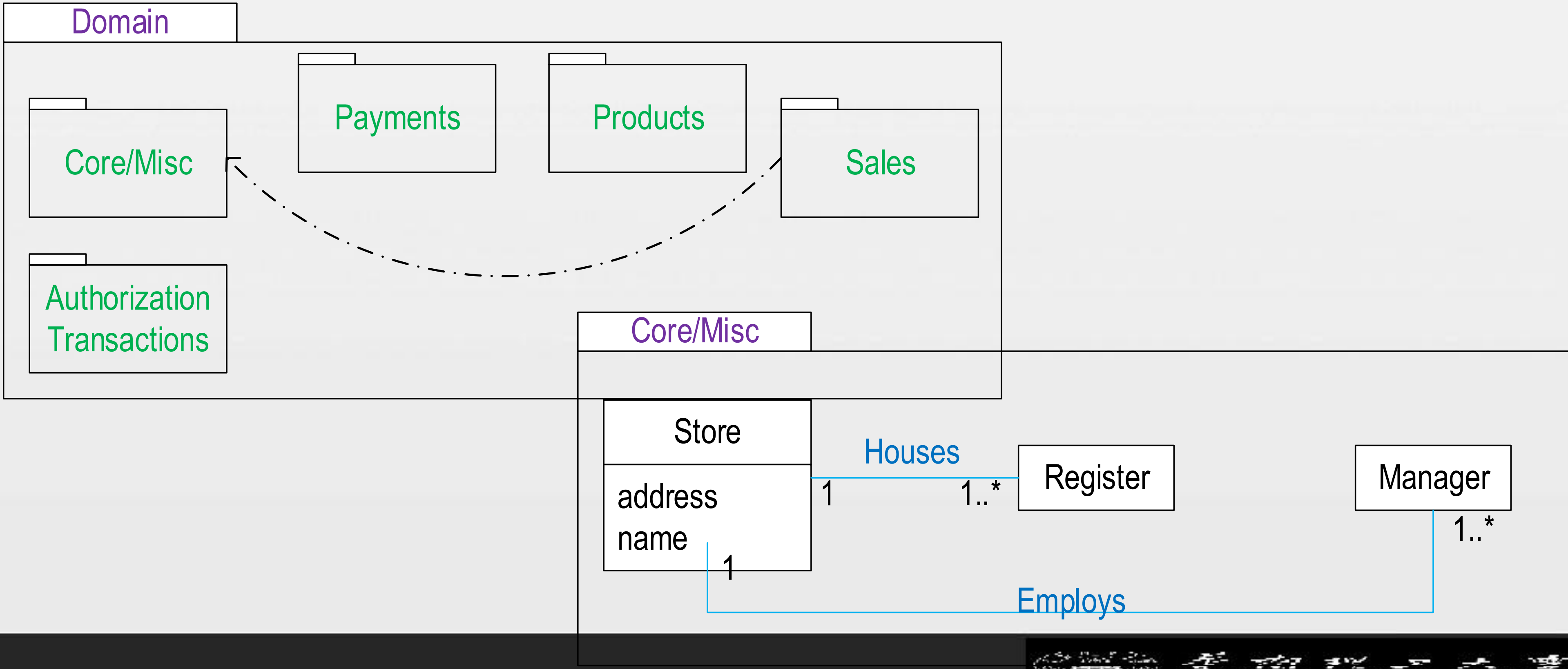
1 parent

* child

Has

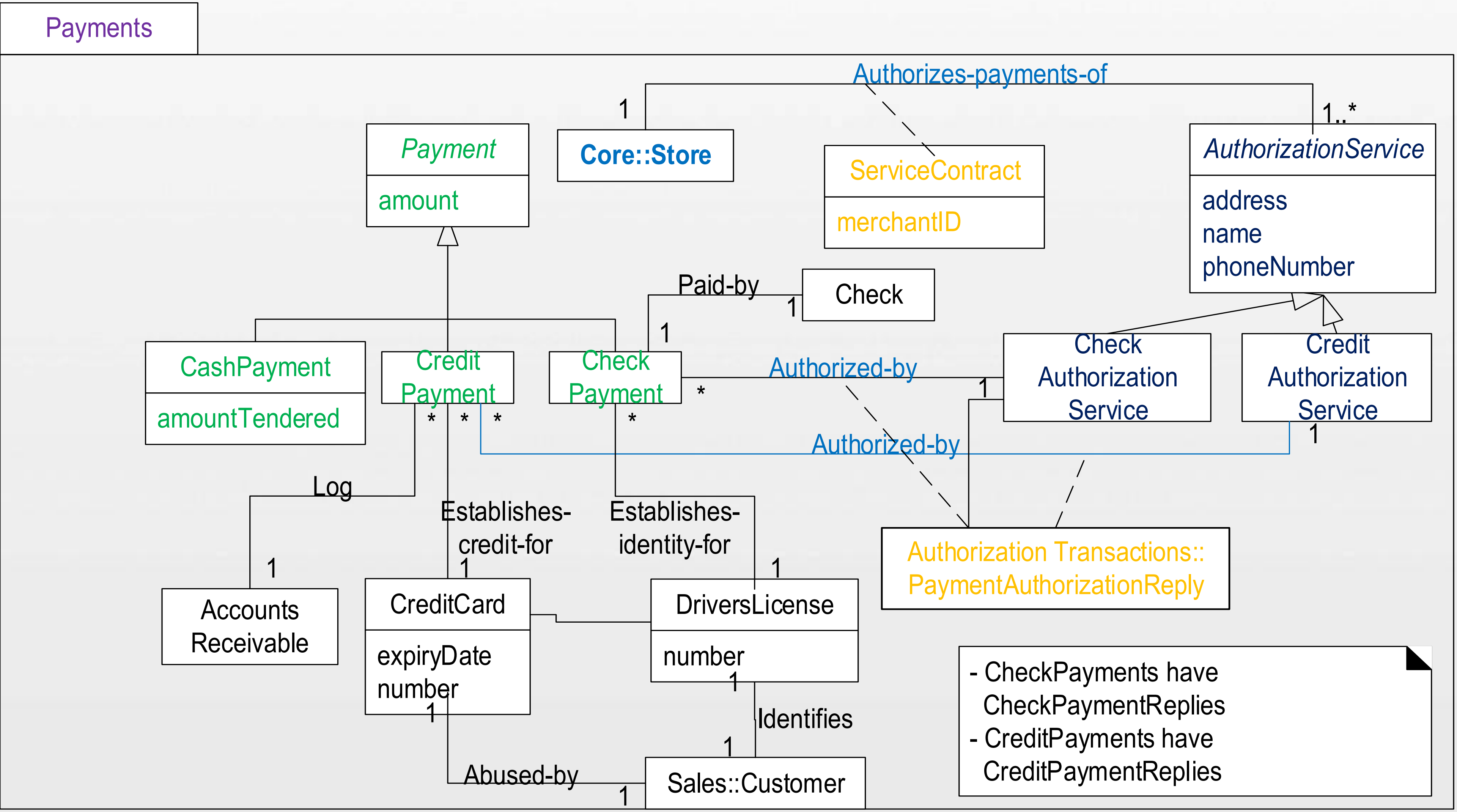
Domain Model Refinement

- **Using Packages to Organize the Domain Model**
  - **UML Package. a tabbed folder**
    - Ownership and References
    - Package Dependencies
  - **POS Domain Model Packages**



Domain Model Refinement

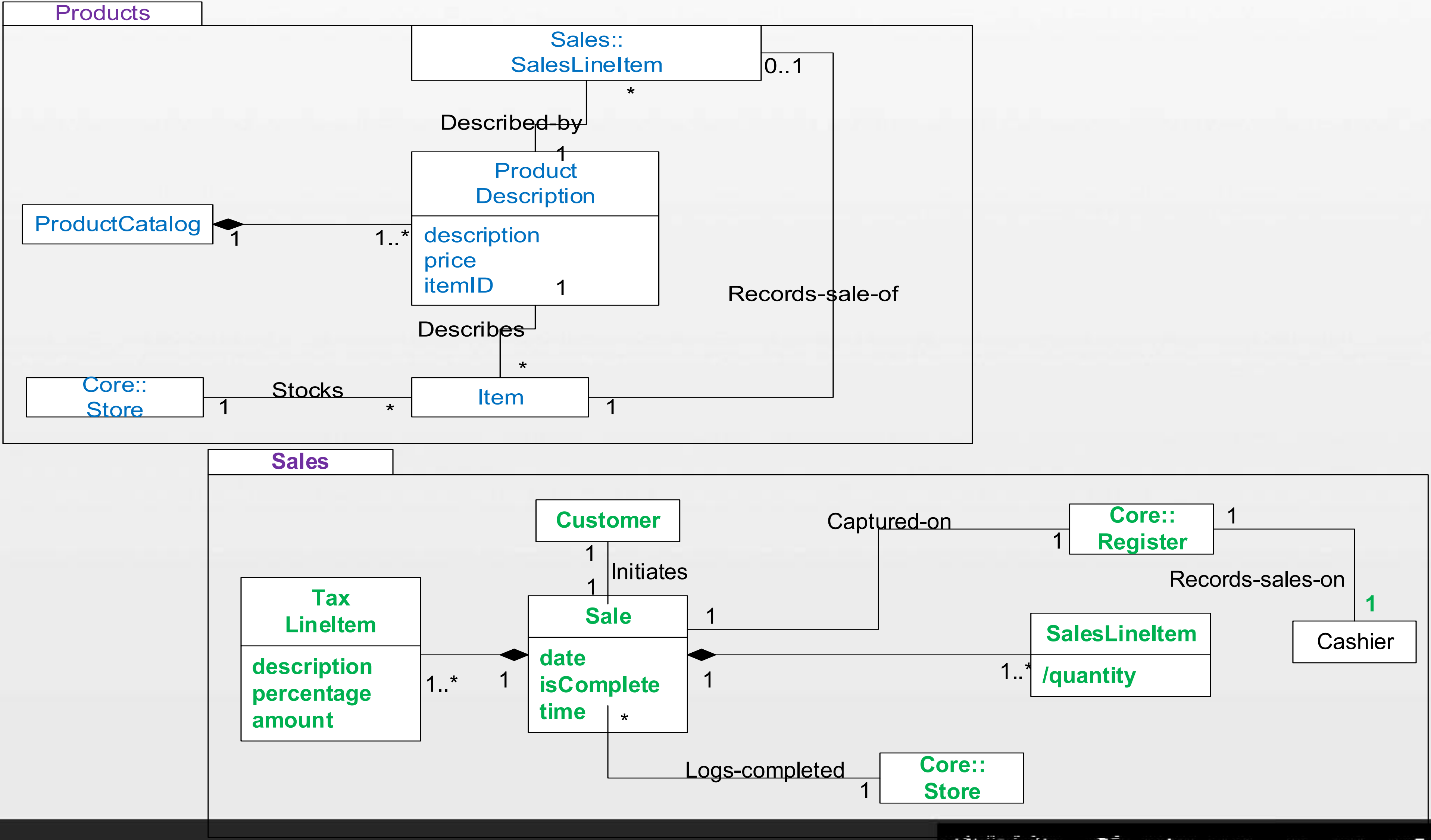
- Using Packages to Organize the Domain Model



- CheckPayments have CheckPaymentReplies
- CreditPayments have CreditPaymentReplies

Domain Model Refinement

- Using Packages to Organize the Domain Model



Products

Sales::
SalesLineItem

0..1

*

Described-by

1

Product
Description

description
price
itemID

1

ProductCatalog

1

1..*

Records-sale-of

Describes

*

Core::
Store

Stocks

1

*

Item

1

Sales

Customer

1

Initiates

1

Captured-on

Core::
Register

1

1

Records-sales-on

Tax
LineItem

description
percentage
amount

Sale

date
isComplete
time

1

1..*

1

1

*

SalesLineItem

/quantity

1..*

1

Cashier

Logs-completed

Core::
Store

1

Architectual Analysis

- **Software architectural concerns non-functional requirements (architectural factors)**
- **Common Steps in Architectural Analysis**
  - Identify and analyze the non-functional requirements that have an impact on the architecture
    - **variation point**
    - **evolution point**
  - analyze alternatives and create solutions to resolve the impact
- **Identification and Analysis of Architectural Factors**

Factor	Measures and quality scenarios	Variability (current flexibility and future evolution)	Impact of factor (and its variability) on stakeholders, architecture and other factors	Priority for Difficulty or Risk Success

- **Quality Scenarios, measurable (or at least observable) responses**

Architectural Analysis

Sample of Architectural Factor Table

- Should be recorded in Supplementary Specification
- record use-case related factors with the use case

Factor	Measures and quality scenarios	Variability (current flexibility and future evolution)	Impact of factor on stakeholders, architecture and other factors	Priority	Difficulty or Risk
Reliability and Recoverability					
Recovery from remote service failure	When a remote service fails, reestablish connectivity with it within 1 minute of its detected re-availability, under normal store load in a production environment.	current flexibility - our SME says local client-side simplified services are acceptable (and desirable) until reconnection is possible. evolution - within 2 years, some retailers may be willing to pay for full local replication of remote services (such as the tax calculator). Probability? High.	High impact on the large-scale design. Retailers really dislike it when remote services fail, as it prevents or restricts them from using a POS to make sales.	H	M
…	…	…	…		

Architectural Analysis

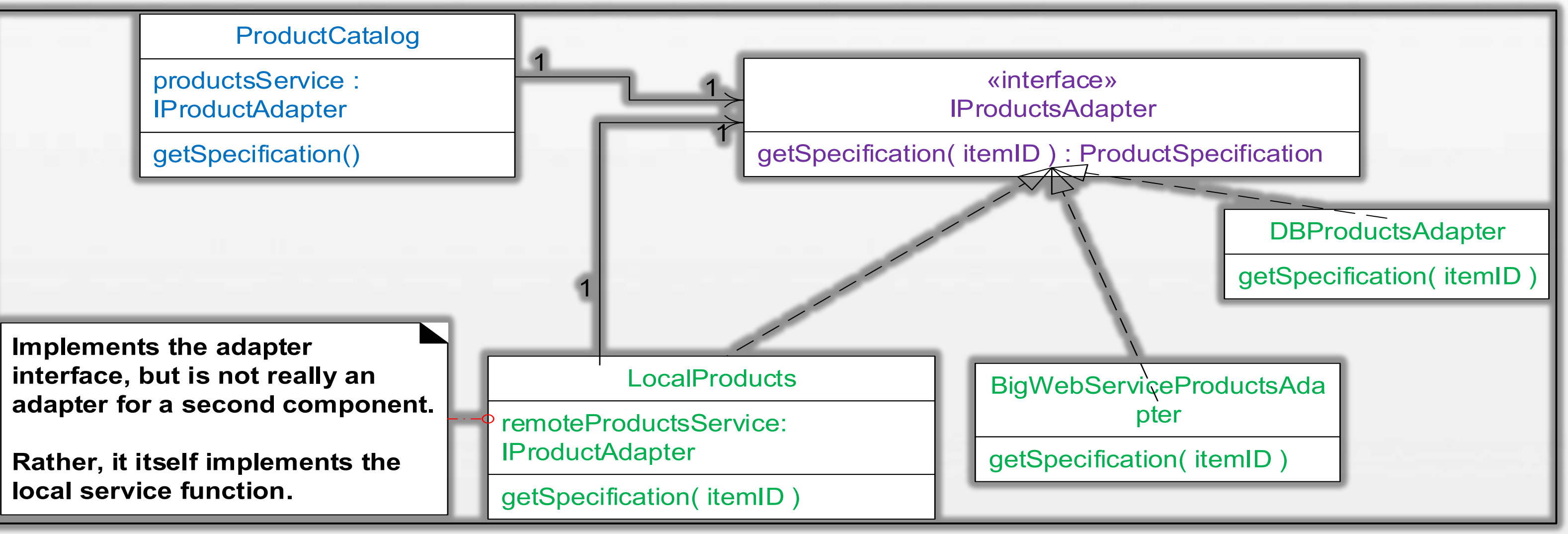
- **Resolution of Architectural Factors, example**
  - Software architectural documents (SAD) or technical memos
- **Example**
  - Issue: Reliability Recovery from Remote Service Failure
  - Solution Summary: Location transparency using service lookup, failover from remote to local, and local service partial replication.
  - Factors
    - Robust recovery from remote service failure (e.g., tax calculator, inventory)
    - Robust recovery from remote product (e.g., descriptions and prices) database failure
  - **Solution**
    - Achieve protected variation with respect to location of services using an Adapter created in a ServicesFactory. Where possible, offer local implementations of remote services, usually with simplified or constrained behavior

More Object Design with GoF Patterns

- **Objectives**
  - **Apply GoF and GRASP in the design of the use-case realizations**
    - failover to a local service when a remote service fails
    - local caching
    - support for third-party POS devices, such as different scanners
    - handling credit, debit, and check payments
- **Failover to Local Services; Performance with Local Caching**
  - **Example, Access to product information**
    - Use local cache of Product Description objects for both performance reasons and recoverability
      - in-memory Product Catalog object
      - local products service maintain a larger persistent (hard disk based) cache
    - the local cache should always be searched for a "cache hit" before attempting a remote access

More Object Design with GoF Patterns

- Failover to Local Services; Performance with Local Caching



ProductCatalog

productsService : IProductAdapter

getSpecification()

1

1

1

«interface»
IProductsAdapter

getSpecification(itemID) : ProductSpecification

DBProductsAdapter

getSpecification(itemID)

1

Implements the adapter interface, but is not really an adapter for a second component.

Rather, it itself implements the local service function.

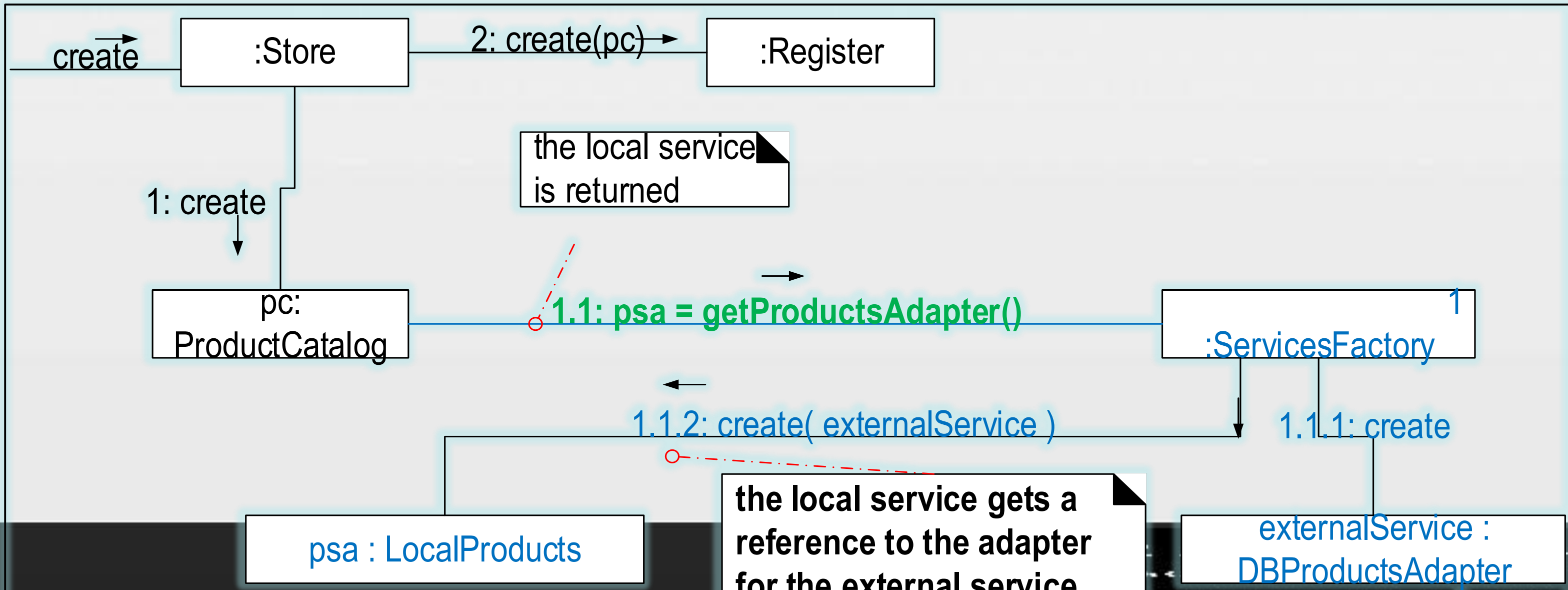
LocalProducts

remoteProductsService: IProductAdapter

getSpecification(itemID)

BigWebServiceProductsAdapter

getSpecification(itemID)



create

:Store

2: create(pc)

:Register

the local service is returned

1: create

pc: ProductCatalog

1.1: psa = getProductsAdapter()

1

:ServicesFactory

1.1.2: create(externalService)

1.1.1: create

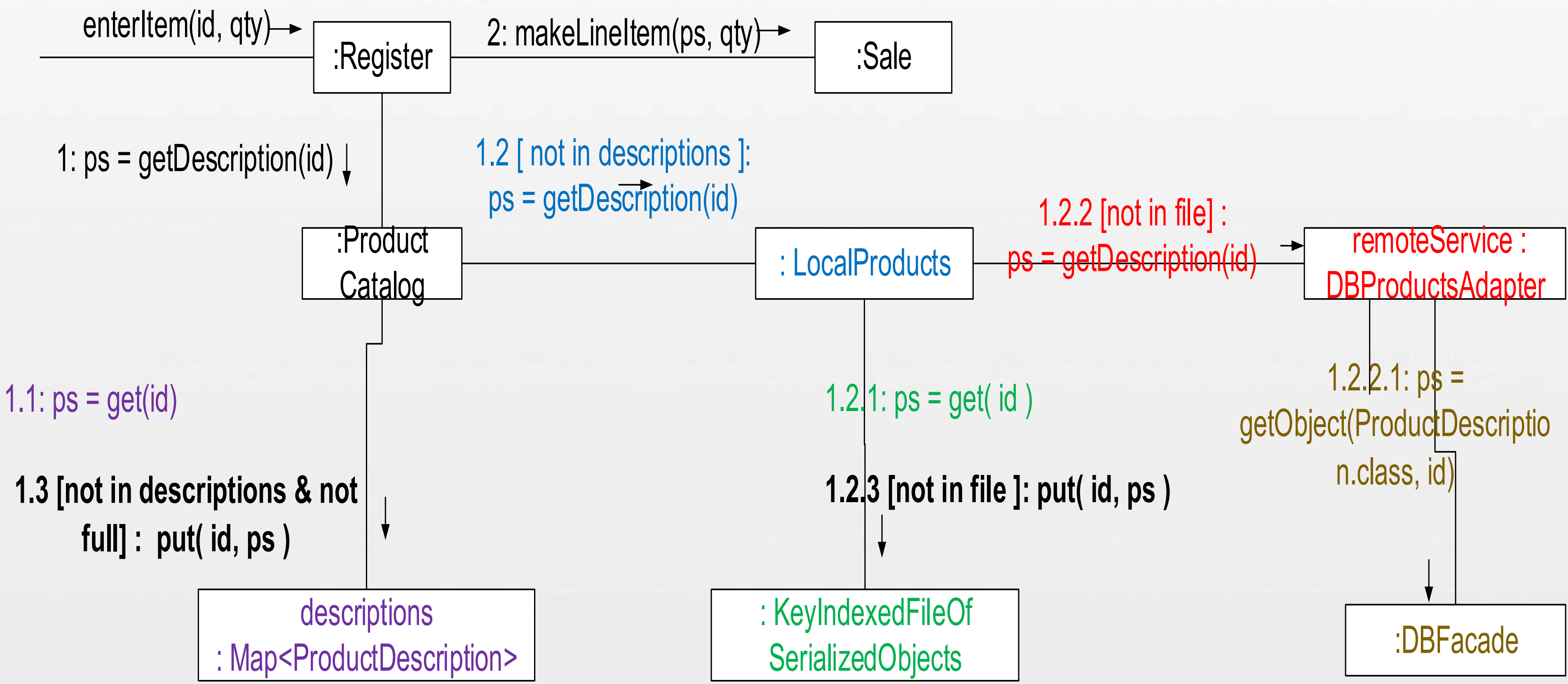
the local service gets a reference to the adapter for the external service

psa : LocalProducts

externalService : DBProductsAdapter

More Object Design with GoF Patterns

- Failover to Local Services; Performance with Local Caching



enterItem(id, qty)

:Register

2: makeLineItem(ps, qty)

:Sale

1: ps = getDescription(id)

1.2 [not in descriptions]:
ps = getDescription(id)

:Product
Catalog

: LocalProducts

1.2.2 [not in file] :
ps = getDescription(id)

remoteService :
DBProductsAdapter

1.1: ps = get(id)

1.2.1: ps = get(id)

1.2.2.1: ps =
getObject(ProductDescriptio
n.class, id)

1.3 [not in descriptions & not
full] : put(id, ps)

1.2.3 [not in file]: put(id, ps)

descriptions
: Map<ProductDescription>

: KeyIndexedFileOf
SerializedObjects

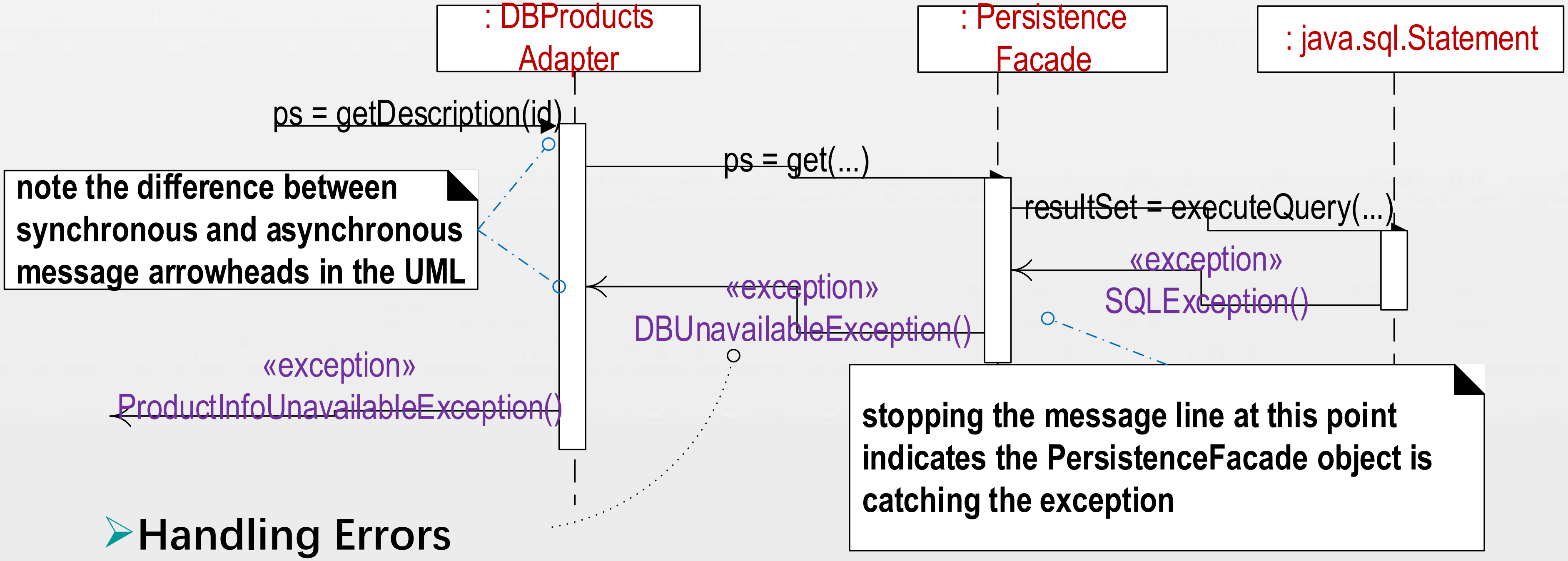
:DBFacade

- lazy initialization ? eager initialization
- Stale Cache?
  - queries every n minutes and updates its cache

More Object Design with GoF Patterns

■ Failover to Local Services; Performance with Local Caching

- What to do where there isn't a local cache hit and access to the external products service fails?
  - Throwing Exceptions



: DBProducts Adapter

: Persistence Facade

: java.sql.Statement

ps = getDescription(id)

ps = get(...)

resultSet = executeQuery(...)

«exception» SQLException()

«exception» DBUnavailableException()

«exception» ProductInfoUnavailableException()

note the difference between synchronous and asynchronous message arrowheads in the UML

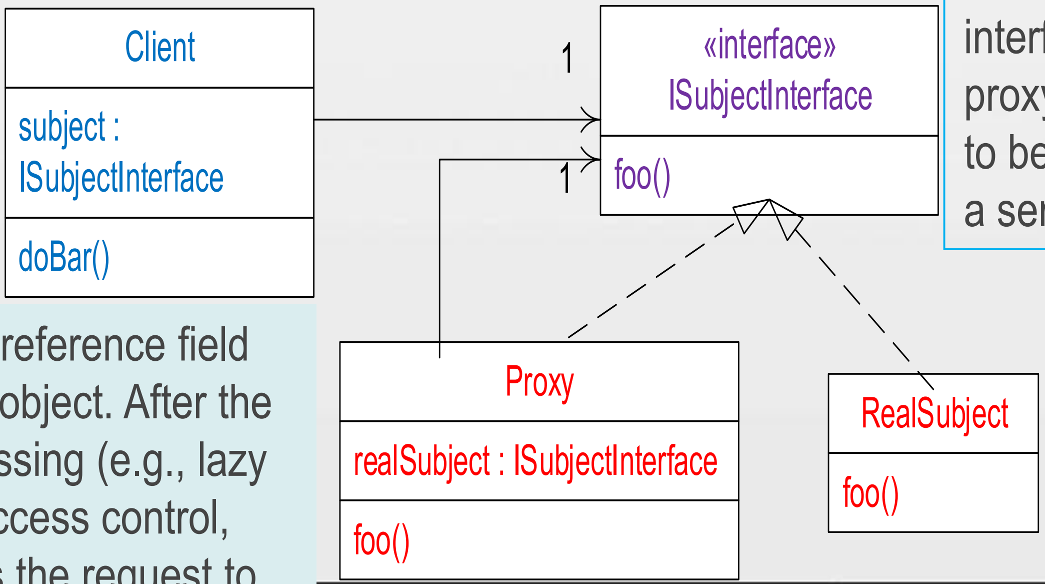
stopping the message line at this point indicates the PersistenceFacade object is catching the exception

- Handling Errors
  - Centralized Error Logging
  - Error Dialog

More Object Design with GoF Patterns

Failover to Local Services with a Proxy (GoF)

- Prefer remote service than local service
- Problem:
  - Direct access to a real subject object is not desired or possible. What to do?
- Solution
  - Add a level of indirection with a surrogate proxy object that implements the same interface as the subject object, and is responsibility for controlling or enhancing access to it.



1 the **Interface** declares the interface of the Service. The proxy must follow this interface to be able to disguise itself as a service object.

2 The **RealSubject** is a class that provides some useful business logic.

3 The **Proxy** class has a reference field that points to a service object. After the proxy finishes its processing (e.g., lazy initialization, logging, access control, caching, etc.), it passes the request to the service object.

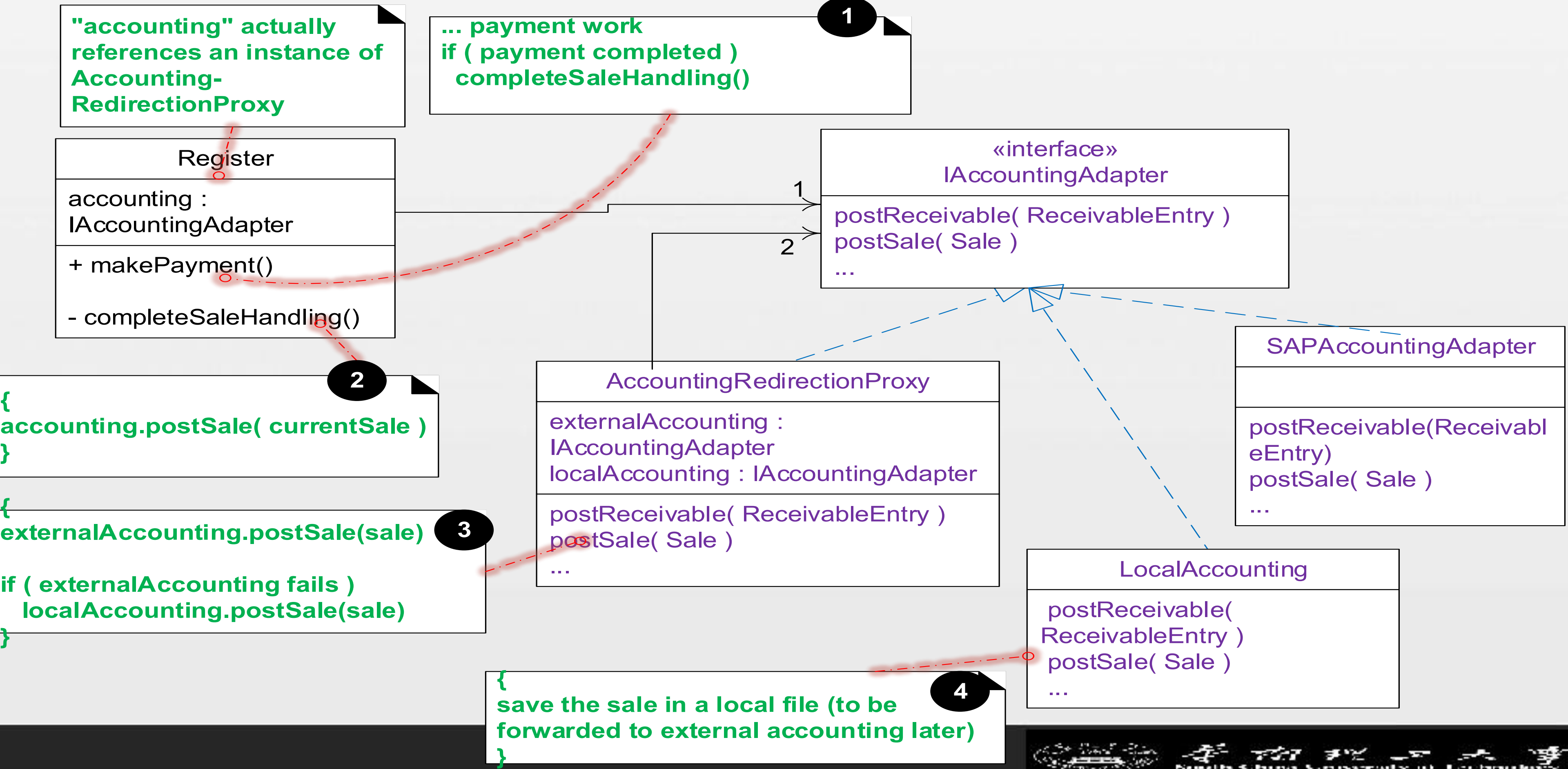
More Object Design with GoF Patterns

- **Failover to Local Services with a Proxy (GoF)**
  - **优点：**
    - 代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用；同时代理对象可以扩展目标对象的功能；
    - 代理模式能将客户端与目标对象分离，降低了系统的耦合度
  - 缺点：
    - 增加代理对象，会造成请求处理速度变慢，同时系统复杂性增加
  - **根据代理对象的目的和使用场景，代理模式可以分为**：
    - 远程代理(Remote Proxy)：为位于不同的地址空间的对象提供本地代理对象
    - 虚拟代理(Virtual Proxy)：创建资源消耗较大的对象时，可以先创建一个消耗相对较小的对象来表示，真实对象在需要时才会真正创建
    - 保护代理(Protect Proxy)：控制对一个对象的访问，给不同的用户提供不同级别的使用权限
    - 缓冲代理(Cache Proxy)：为目标操作的结果提供临时的存储空间，以便多个客户端可以共享这些结果
    - 智能引用代理(Smart Reference Proxy)：当一个对象被引用时提供额外的操作，例如记录对象调用次数等

More Object Design with GoF Patterns

Proxy Pattern summary

- a proxy is an outer object that wraps an inner object, and both implement the same interface. A client does not know that it references a proxy or the real subject(The Proxy intercepts calls in order to enhance access to the real subject)



"accounting" actually references an instance of Accounting-RedirectionProxy

1
```
... payment work
if ( payment completed )
  completeSaleHandling()
```

Register

accounting : IAccountingAdapter

+ makePayment()

- completeSaleHandling()

2
```
{
accounting.postSale( currentSale )
}
```

3
```
{
externalAccounting.postSale(sale)

if ( externalAccounting fails )
   localAccounting.postSale(sale)
}
```

«interface»
IAccountingAdapter

postReceivable(ReceivableEntry)
postSale(Sale)
...

AccountingRedirectionProxy

externalAccounting : IAccountingAdapter
localAccounting : IAccountingAdapter

postReceivable(ReceivableEntry)
postSale(Sale)
...

SAPAccountingAdapter

postReceivable(ReceivableEntry)
postSale(Sale)
...

LocalAccounting

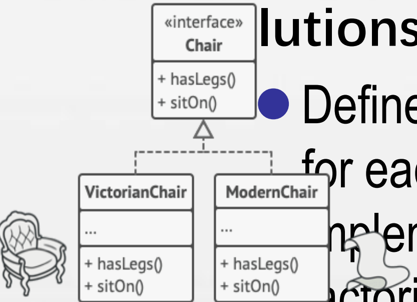
postReceivable(ReceivableEntry)
postSale(Sale)
...

4
```
{
save the sale in a local file (to be
forwarded to external accounting later)
}
```

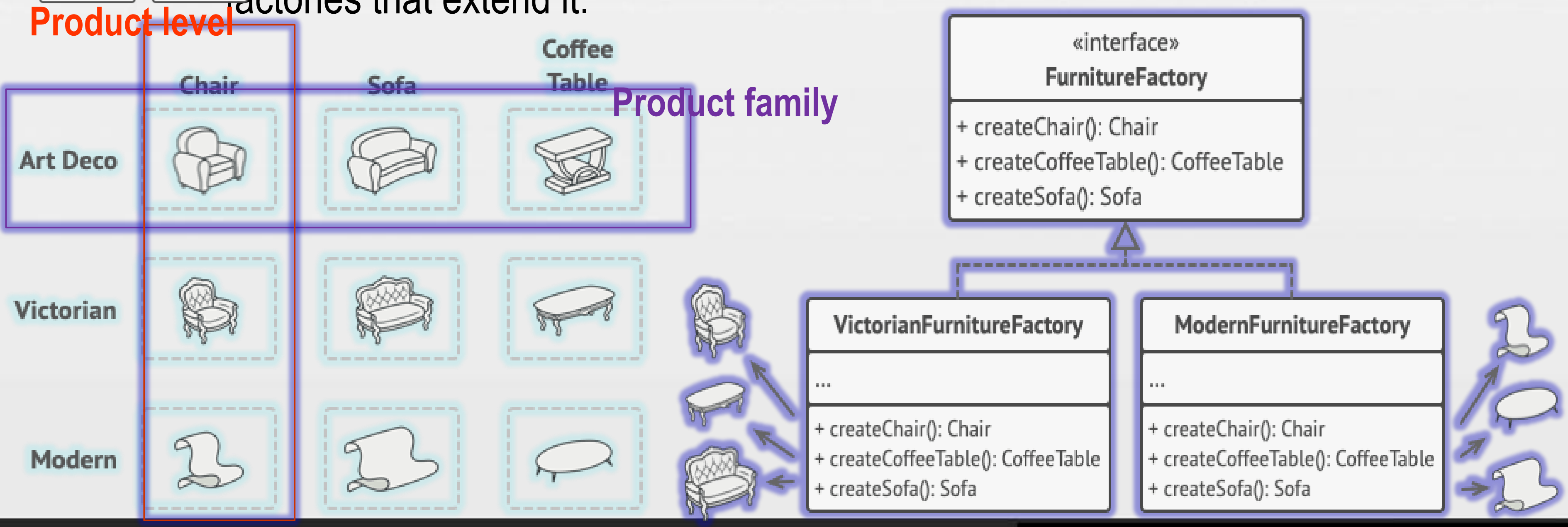
More Object Design with GoF Patterns

Abstract Factory (GoF) for Families of Related Objects

- Problem
  - How to create families of related classes that implement a common interface?



- lutions
  - Define a factory interface (the abstract factory). Define a concrete factory class for each family of things to create. Optionally, define a true abstract class that implements the factory interface and provides common services to the concrete factories that extend it.



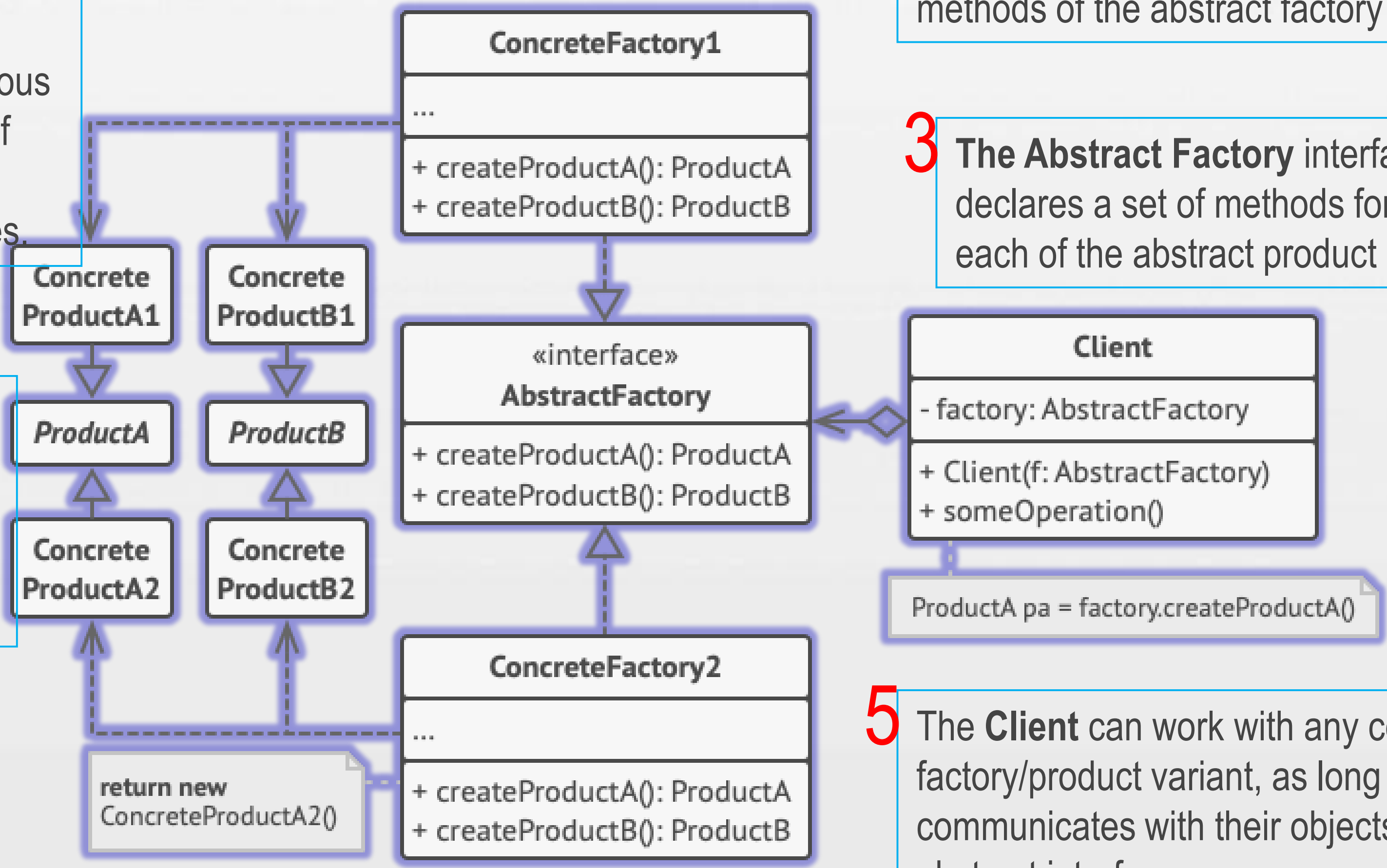
More Object Design with GoF Patterns

Abstract Factory (GoF) for Families of Related Objects

- Structure

2 **Concrete Products** are various implementations of abstract products, grouped by families.

1 **Abstract Products** declare interfaces for a set of distinct but related product



ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
Concrete ProductA1
Concrete ProductB1
ProductA
ProductB
Concrete ProductA2
Concrete ProductB2
«interface»
AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
ProductA pa = factory.createProductA()
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()

4 **Concrete Factories** implement creation methods of the abstract factory

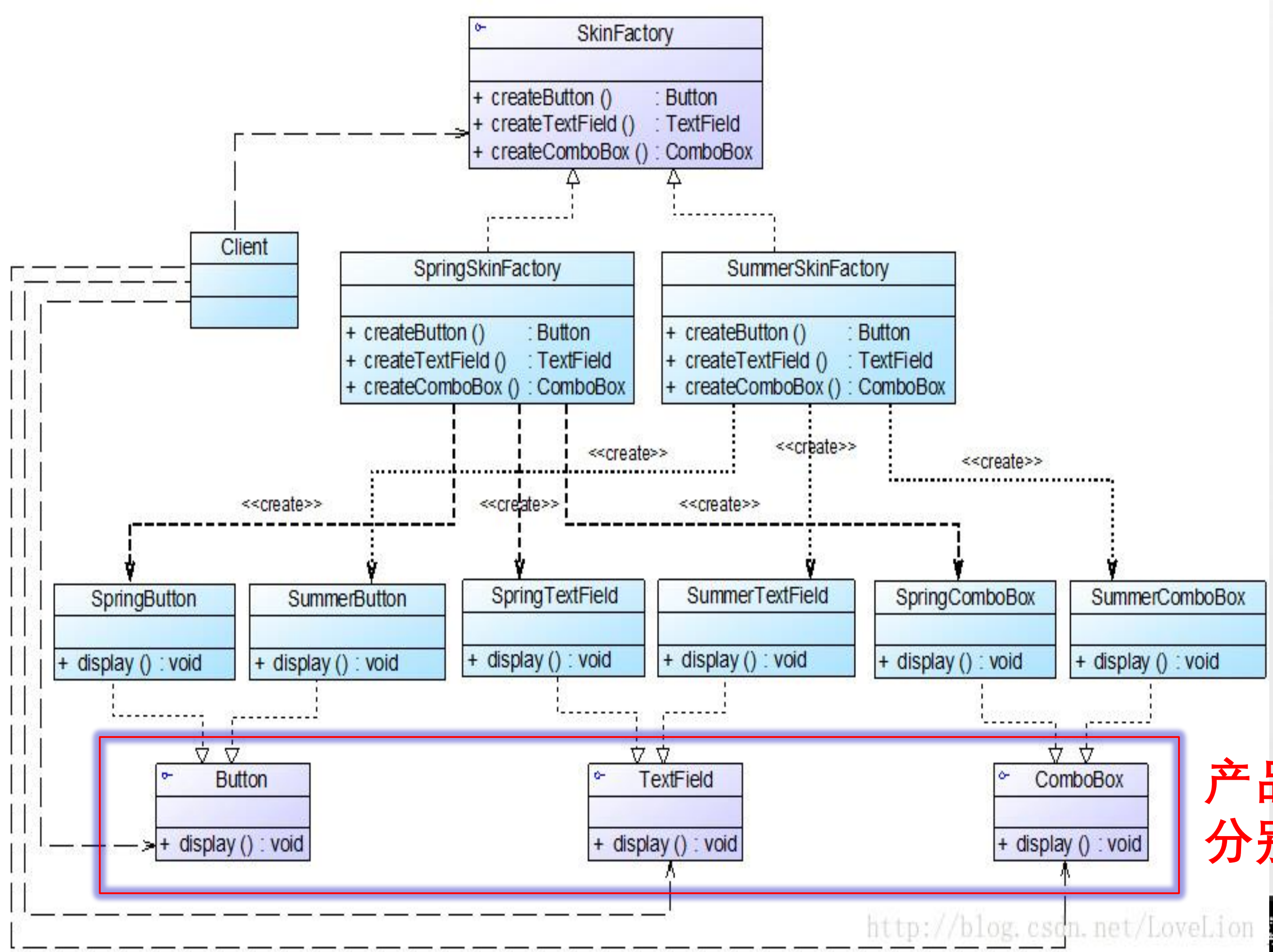
3 **The Abstract Factory** interface declares a set of methods for creating each of the abstract product

5 The **Client** can work with any concrete factory/product variant, as long as it communicates with their objects via abstract interfaces.

More Object Design with GoF Patterns

Abstract Factory (GoF) example

- 界面皮肤库，用户在使用时可以通过菜单来选择皮肤，不同的皮肤将提供视觉效果不同的按钮、文本框、组合框等界面元素

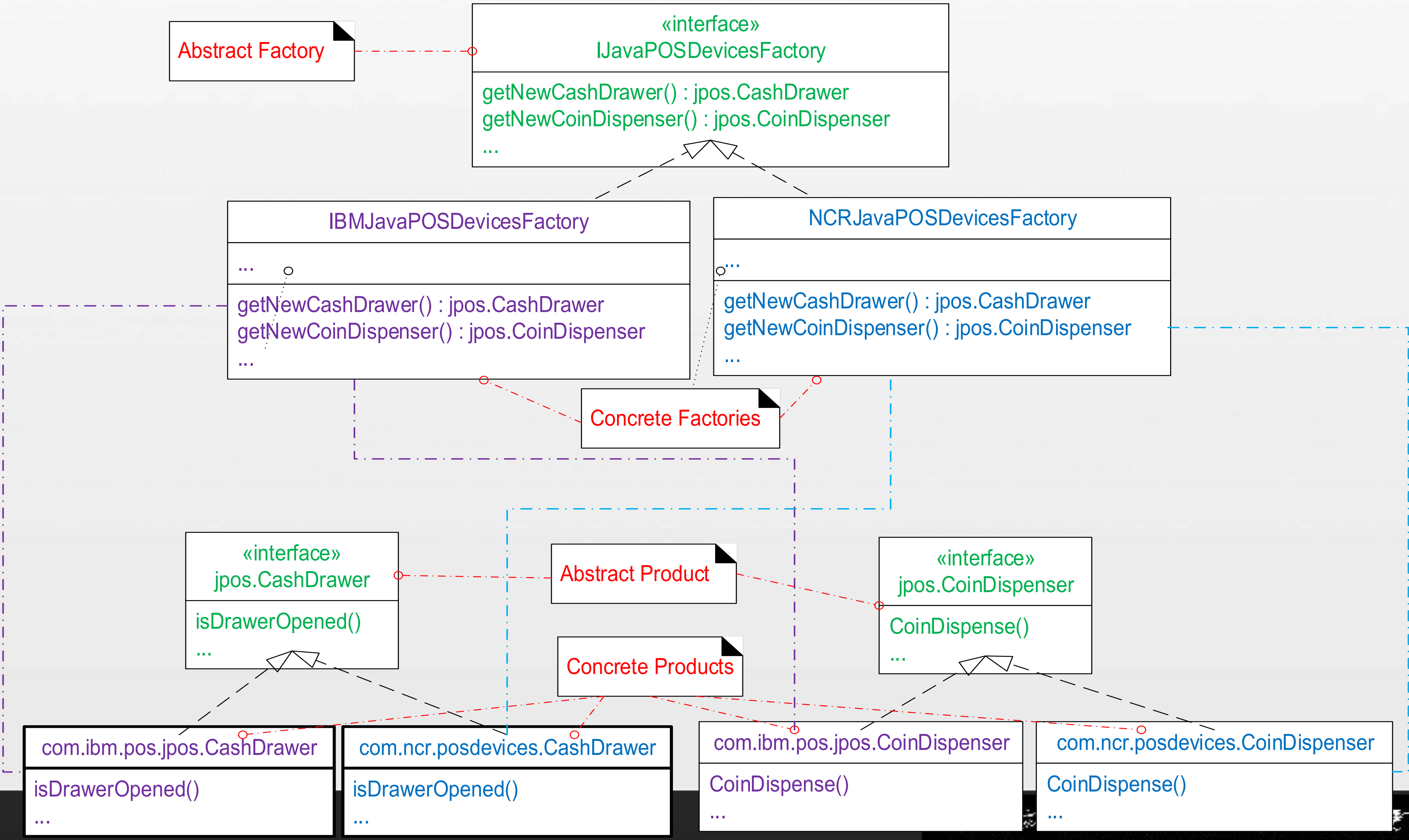


```
//抽象工厂类
SkinFactory factory;

//在一起工作的产品族
Button bt;
TextField tf;
ComboBox cb;

//根据用户的配置文件选择皮肤
factory = (SkinFactory)XMLUtil.ge
//获得该皮肤相关的产品的
bt = factory.createButton();
tf = factory.createTextField();
cb = factory.createComboBox();
```

产品族？产品层次?
分别在那里?

More Object Design with GoF Patterns

■ Abstract Factory (GoF) for Families of Related Objects


Abstract Factory
«interface»
IJavaPOSDevicesFactory
getNewCashDrawer() : jpos.CashDrawer
getNewCoinDispenser() : jpos.CoinDispenser
...
IBMJavaPOSDevicesFactory
...
getNewCashDrawer() : jpos.CashDrawer
getNewCoinDispenser() : jpos.CoinDispenser
...
NCRJavaPOSDevicesFactory
...
getNewCashDrawer() : jpos.CashDrawer
getNewCoinDispenser() : jpos.CoinDispenser
...
Concrete Factories
«interface»
jpos.CashDrawer
isDrawerOpened()
...
Abstract Product
«interface»
jpos.CoinDispenser
CoinDispense()
...
Concrete Products
com.ibm.pos.jpos.CashDrawer
isDrawerOpened()
...
com.ncr.posdevices.CashDrawer
isDrawerOpened()
...
com.ibm.pos.jpos.CoinDispenser
CoinDispense()
...
com.ncr.posdevices.CoinDispenser
CoinDispense()
...

Designing a Persistence Framework with Patterns

Introduction

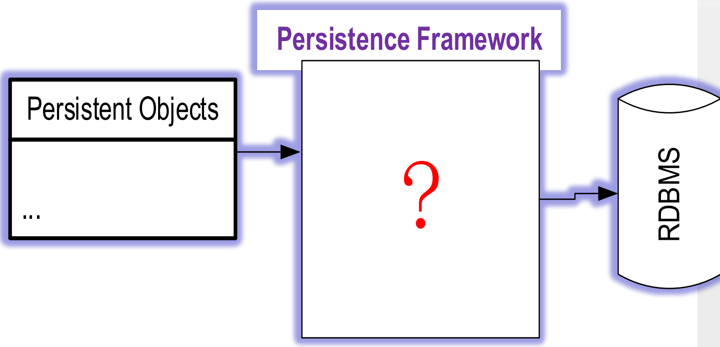
- There are many excellent free, robust, industrial-strength open source persistence frameworks, such as Hibernate and ibatis
- using persistence Framework to introduce key OO framework design principles and patterns

Persistent Objects

- require persistent storage, such as Product-Description instances

Storage Mechanisms

- File
- Database(relational, object, nosql, memory)
- Elastic Search



Persistence Framework

- a general-purpose, reusable, and extendable set of types to support persistent objects
- A persistence service (or subsystem) actually provides the service
- A persistence service worked with RDBs, is called an **O-R mapping service**

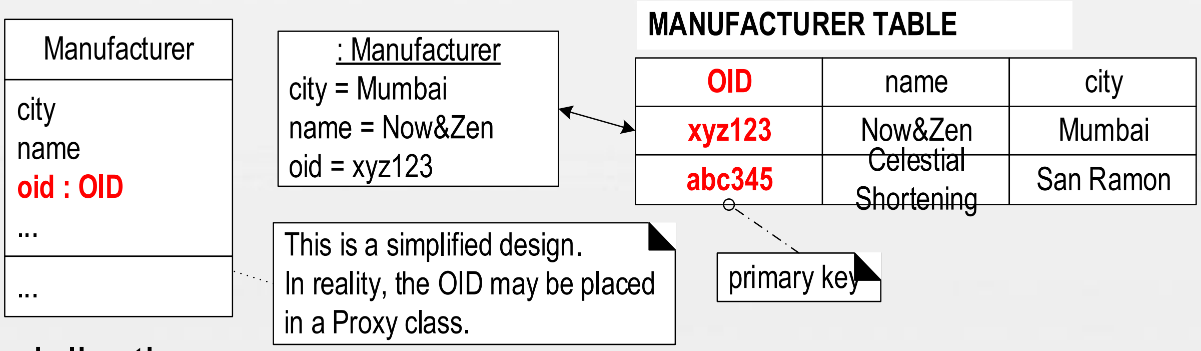
Designing a Persistence Framework with Patterns

Persistence Frameworks requirement

- store and retrieve objects in a persistent storage mechanism
- commit and rollback transactions(?)
- **support different storage mechanisms and formats, such as RDBs, records in flat files, or XML in files**

Key Ideas

- Mapping
- Object identity(OID)
- Materialization and dematerialization
- Database mapper
- Caches
- Transaction state of object….



Manufacturer
city
name
oid : OID
...
...

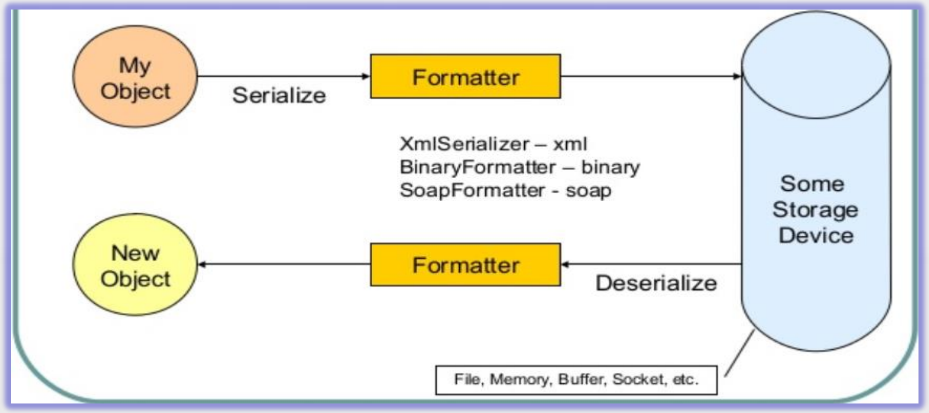
: Manufacturer
city = Mumbai
name = Now&Zen
oid = xyz123

MANUFACTURER TABLE

OID	name	city
xyz123	Now&Zen	Mumbai
abc345	Celestial Shortening	San Ramon

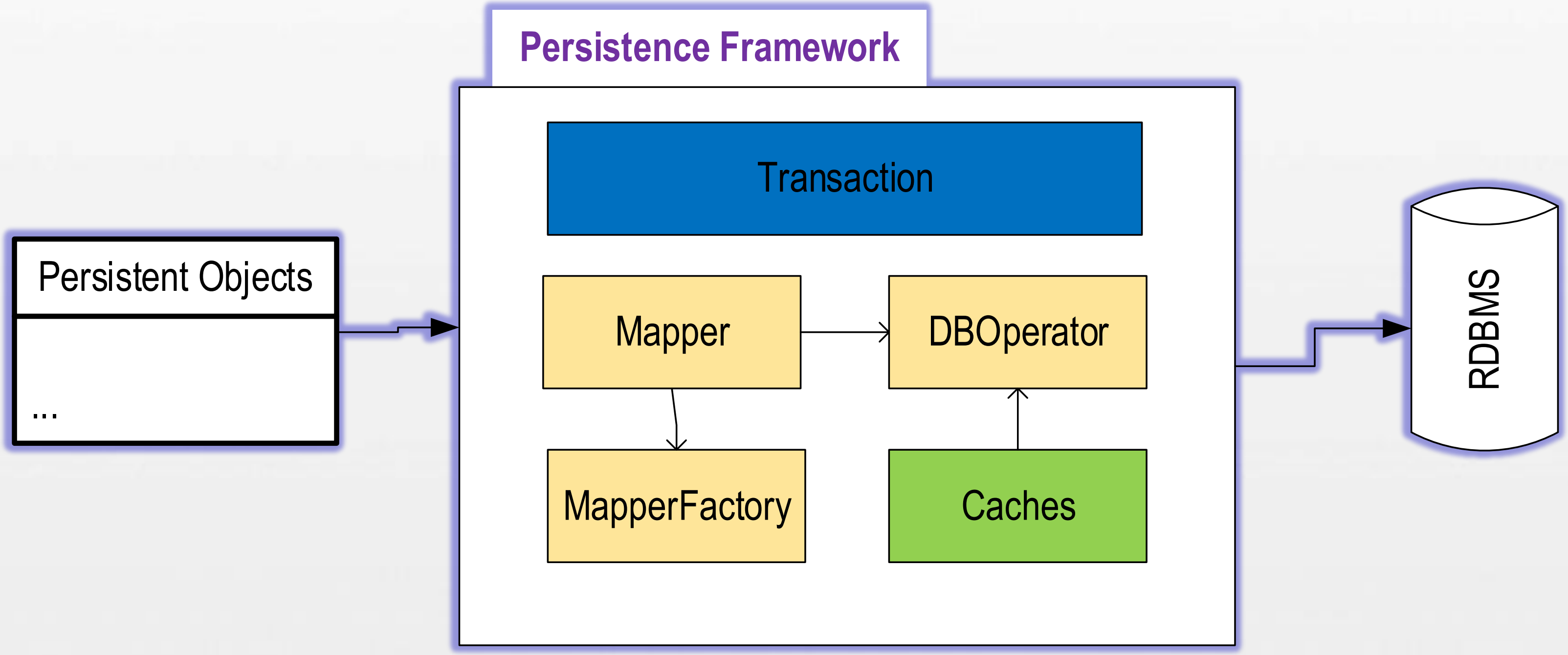
This is a simplified design.
In reality, the OID may be placed in a Proxy class.

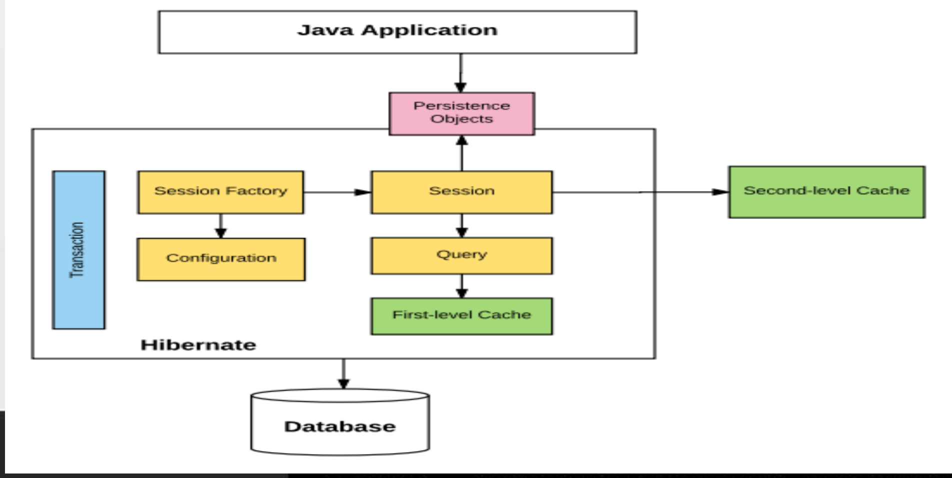
primary key



Designing a Persistence Framework with Patterns

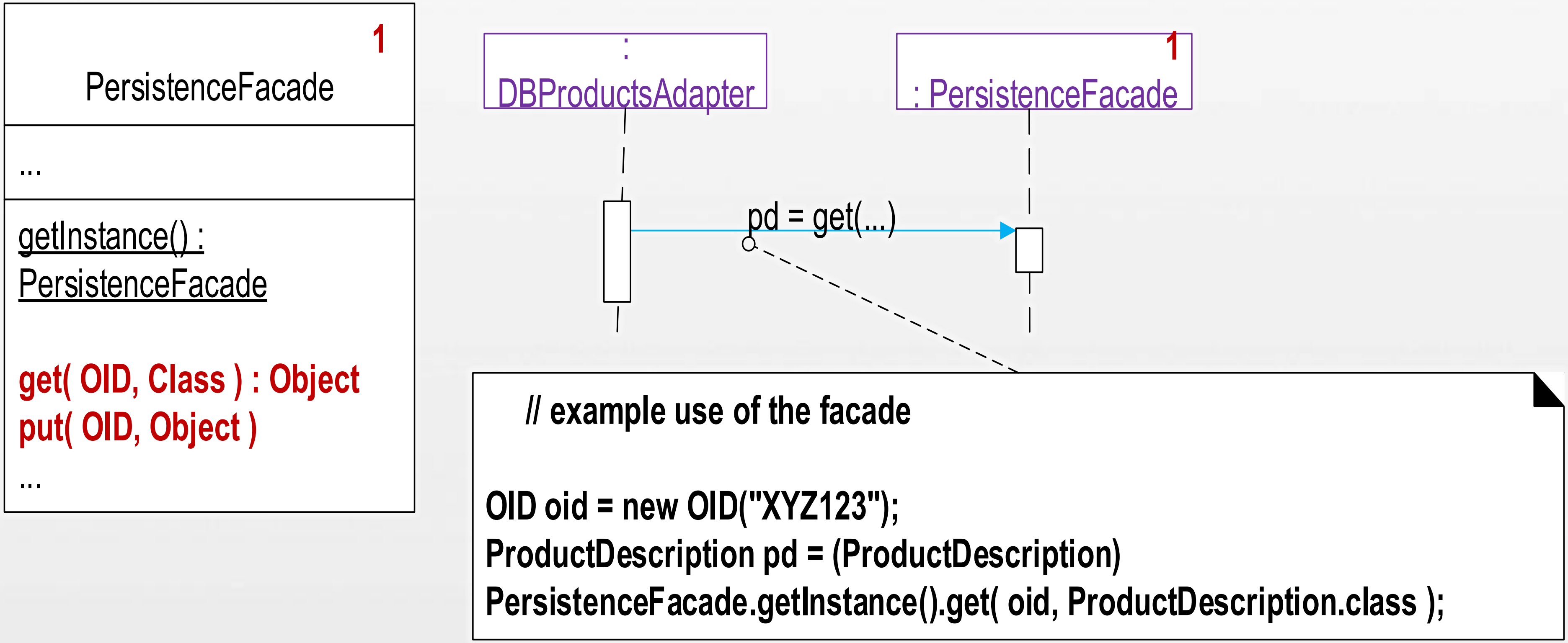
- Persistence Frameworks





Designing a Persistence Framework with Patterns

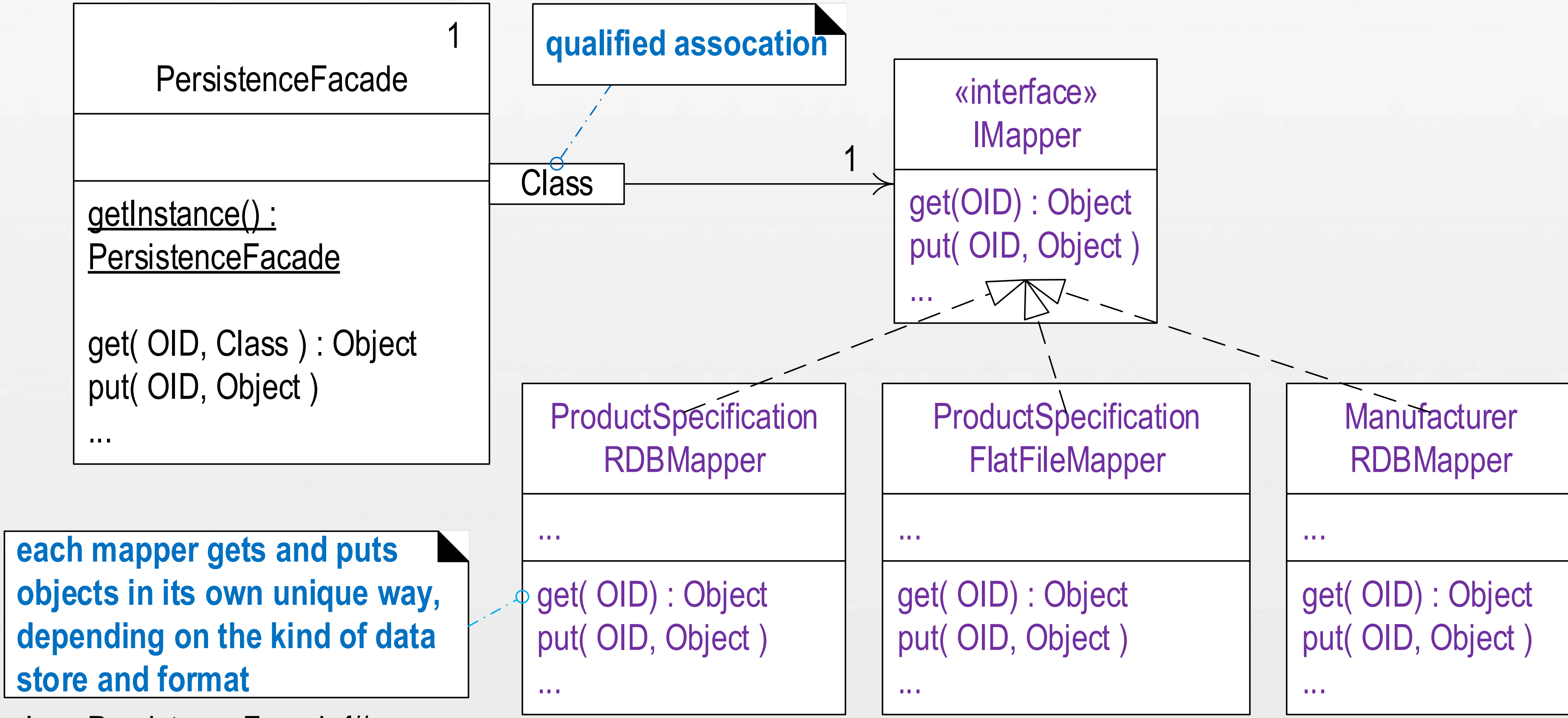
■ Accessing a Persistence Service with a Facade



```
    // example use of the facade

OID oid = new OID("XYZ123");
ProductDescription pd = (ProductDescription)
PersistenceFacade.getInstance().get( oid, ProductDescription.class );
```

Designing a Persistence Framework with Patterns

Database Mapper



1

PersistenceFacade

getInstance() : PersistenceFacade

get(OID, Class) : Object

put(OID, Object)

...

qualified assocation

Class

1

«interface»
IMapper

get(OID) : Object

put(OID, Object)

...

ProductSpecification RDBMapper

...

get(OID) : Object

put(OID, Object)

...

ProductSpecification FlatFileMapper

...

get(OID) : Object

put(OID, Object)

...

Manufacturer RDBMapper

...

get(OID) : Object

put(OID, Object)

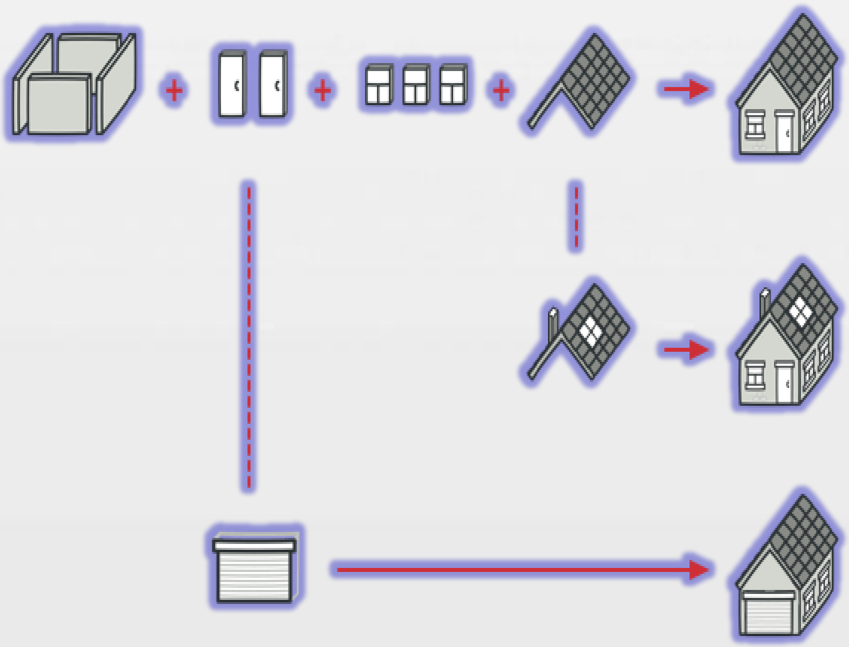
...

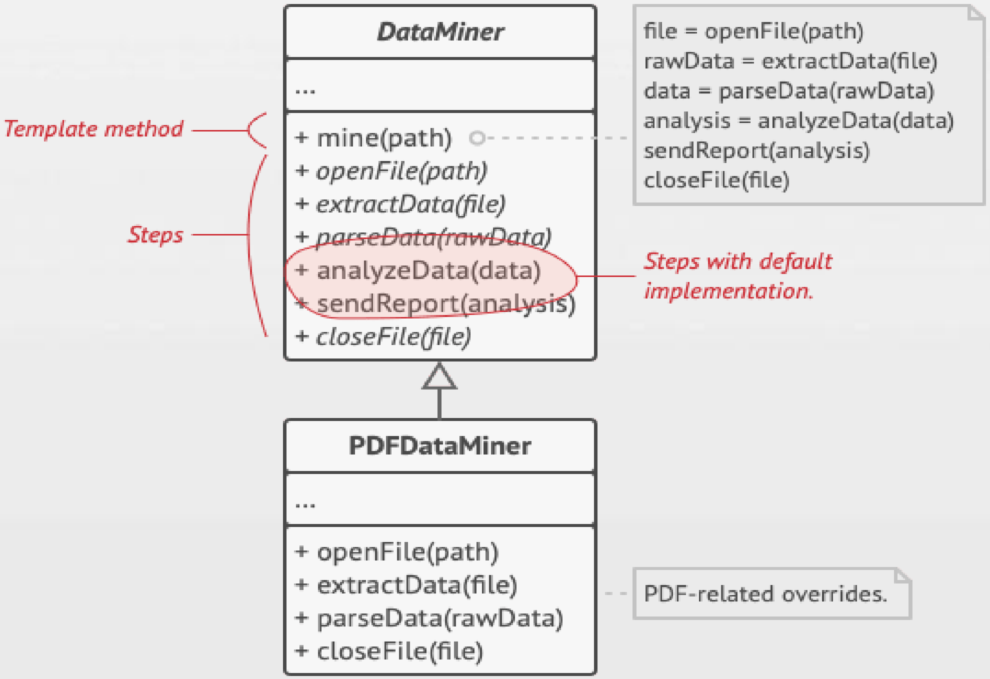
each mapper gets and puts objects in its own unique way, depending on the kind of data store and format

```
class PersistenceFacade{//...
   public Object get( OID oid, Class persistenceClass ){
            // an IMapper is keyed by the Class of the persistent object
              IMapper mapper = (IMapper) mappers.get( persistenceClass );   // delegate
              return mapper.get( oid );}//...}
```

Designing a Persistence Framework with Patterns

- Design Persistence Framework with the Template Method Pattern
  - **Template Method is the heart of framework design**
    - define a method (the Template Method) in a superclass that defines the skeleton of an algorithm, with its varying and unvarying parts
    - varying parts can be overridden in a subclass to add their own unique behavior





DataMiner

...

+ mine(path)
+ *openFile(path)*
+ *extractData(file)*
+ *parseData(rawData)*
+ analyzeData(data)
+ sendReport(analysis)
+ *closeFile(file)*

file = openFile(path)
rawData = extractData(file)
data = parseData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)

Template method

Steps

Steps with default implementation.

PDFDataMiner

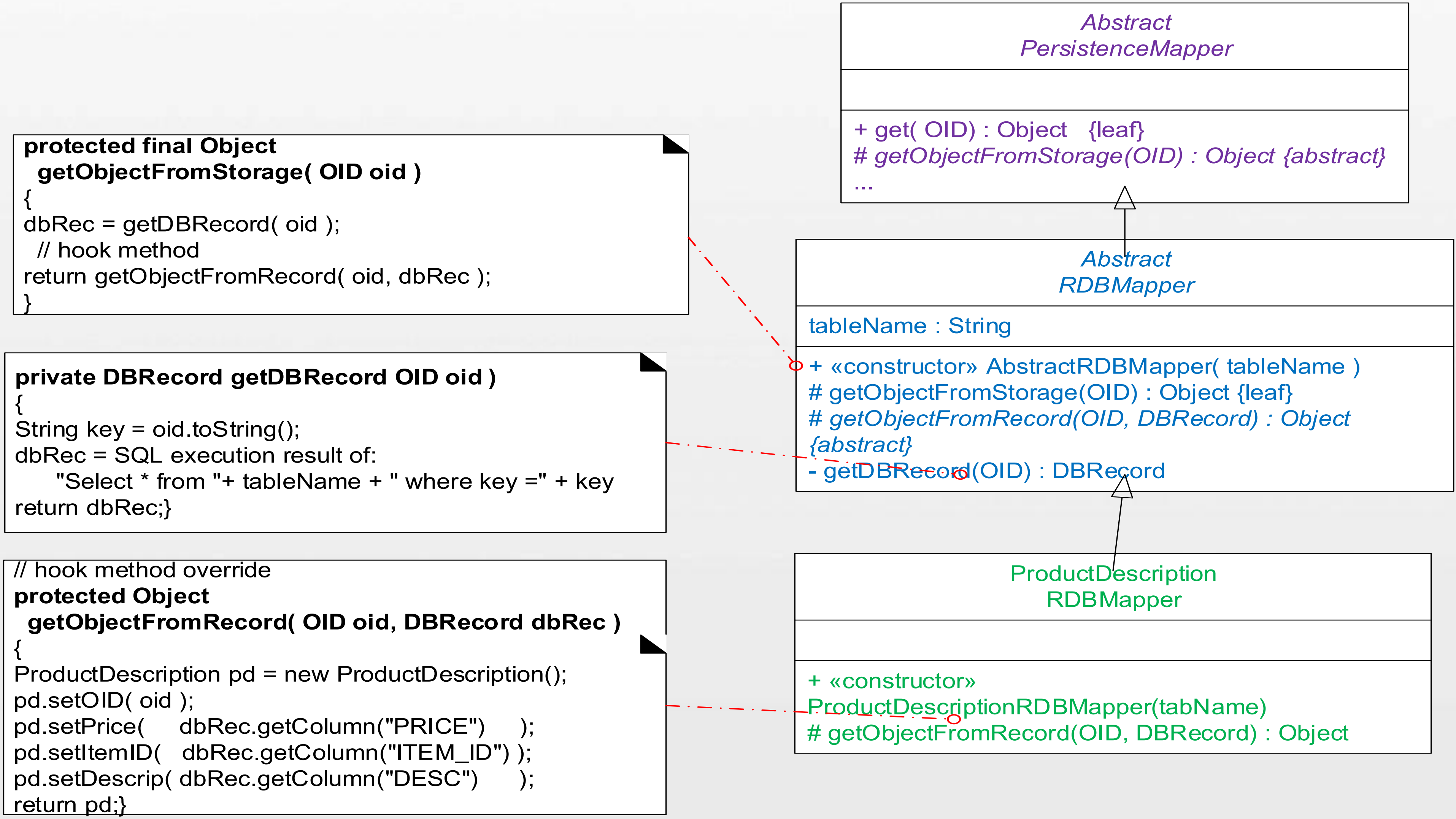
...

+ openFile(path)
+ extractData(file)
+ parseData(rawData)
+ closeFile(file)

PDF-related overrides.

Designing a Persistence Framework with Patterns

Design with the Template Method Pattern



```
protected final Object
  getObjectFromStorage( OID oid )
{
dbRec = getDBRecord( oid );
  // hook method
return getObjectFromRecord( oid, dbRec );
}
```

```
private DBRecord getDBRecord OID oid )
{
String key = oid.toString();
dbRec = SQL execution result of:
     "Select * from "+ tableName + " where key =" + key
return dbRec;}
```

```
// hook method override
protected Object
  getObjectFromRecord( OID oid, DBRecord dbRec )
{
ProductDescription pd = new ProductDescription();
pd.setOID( oid );
pd.setPrice(     dbRec.getColumn("PRICE")     );
pd.setItemID(   dbRec.getColumn("ITEM_ID") );
pd.setDescrip( dbRec.getColumn("DESC")      );
return pd;}
```

Abstract PersistenceMapper
+ get(OID) : Object   {leaf}
# getObjectFromStorage(OID) : Object {abstract}
...
Abstract RDBMapper
tableName : String
+ «constructor» AbstractRDBMapper(tableName)
# getObjectFromStorage(OID) : Object {leaf}
# getObjectFromRecord(OID, DBRecord) : Object {abstract}
- getDBRecord(OID) : DBRecord
ProductDescription RDBMapper
+ «constructor» ProductDescriptionRDBMapper(tabName)
# getObjectFromRecord(OID, DBRecord) : Object

Designing a Persistence Framework with Patterns

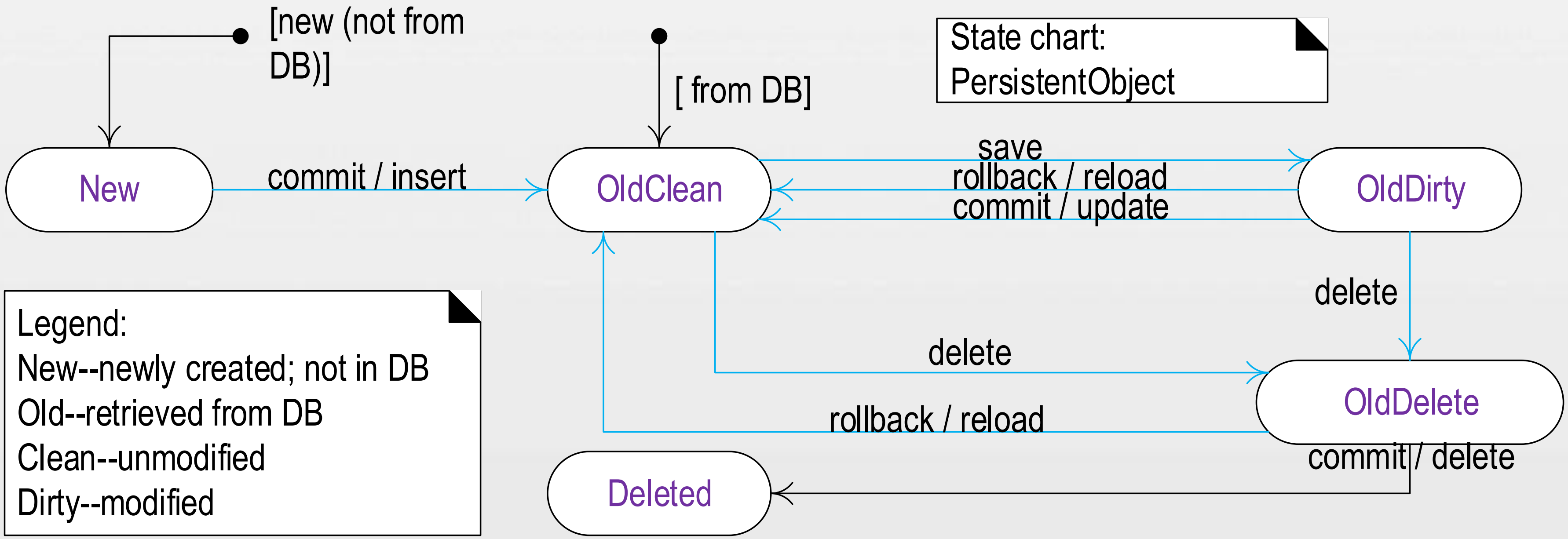
- **Persistence Frameworks**
  - Configuring Mappers with a MapperFactory
  - Cache Management
  - Consolidating and Hiding SQL Statements in One Class

Designing a Persistence Framework with Patterns

Transactional States and the State Pattern

Persistent objects

- can be inserted, deleted, or modified
- Operating on a persistent object does not cause an immediate database update, rather, an explicit commit operation must be performed
- the response to an operation depends on the transactional state of the object



[new (not from DB)]

[from DB]

State chart:
PersistentObject

New

commit / insert

OldClean

save

rollback / reload

commit / update

OldDirty

delete

delete

rollback / reload

OldDelete

commit / delete

Deleted

Legend:
New--newly created; not in DB
Old--retrieved from DB
Clean--unmodified
Dirty--modified

Designing a Persistence Framework with Patterns

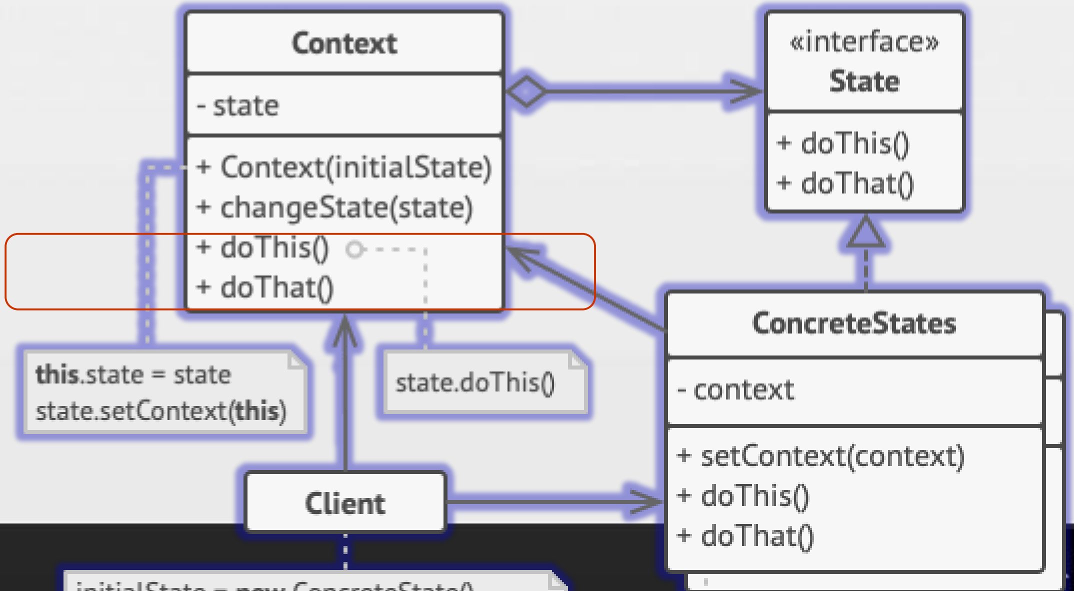
■ State Pattern

➢Context/Problem

- An object's behavior is dependent on its state, and its methods contain case logic reflecting conditional state-dependent actions. Is there an alternative to conditional logic?

➢Solutions

- Create state classes for each state, implementing a common interface. Delegate state-dependent operations from the context object to its current state object. Ensure the context object always points to a state object reflecting its current state.



Context

- state

+ Context(initialState)
+ changeState(state)
+ doThis()
+ doThat()

«interface»
State

+ doThis()
+ doThat()

ConcreteStates

- context

+ setContext(context)
+ doThis()
+ doThat()

this.state = state
state.setContext(this)

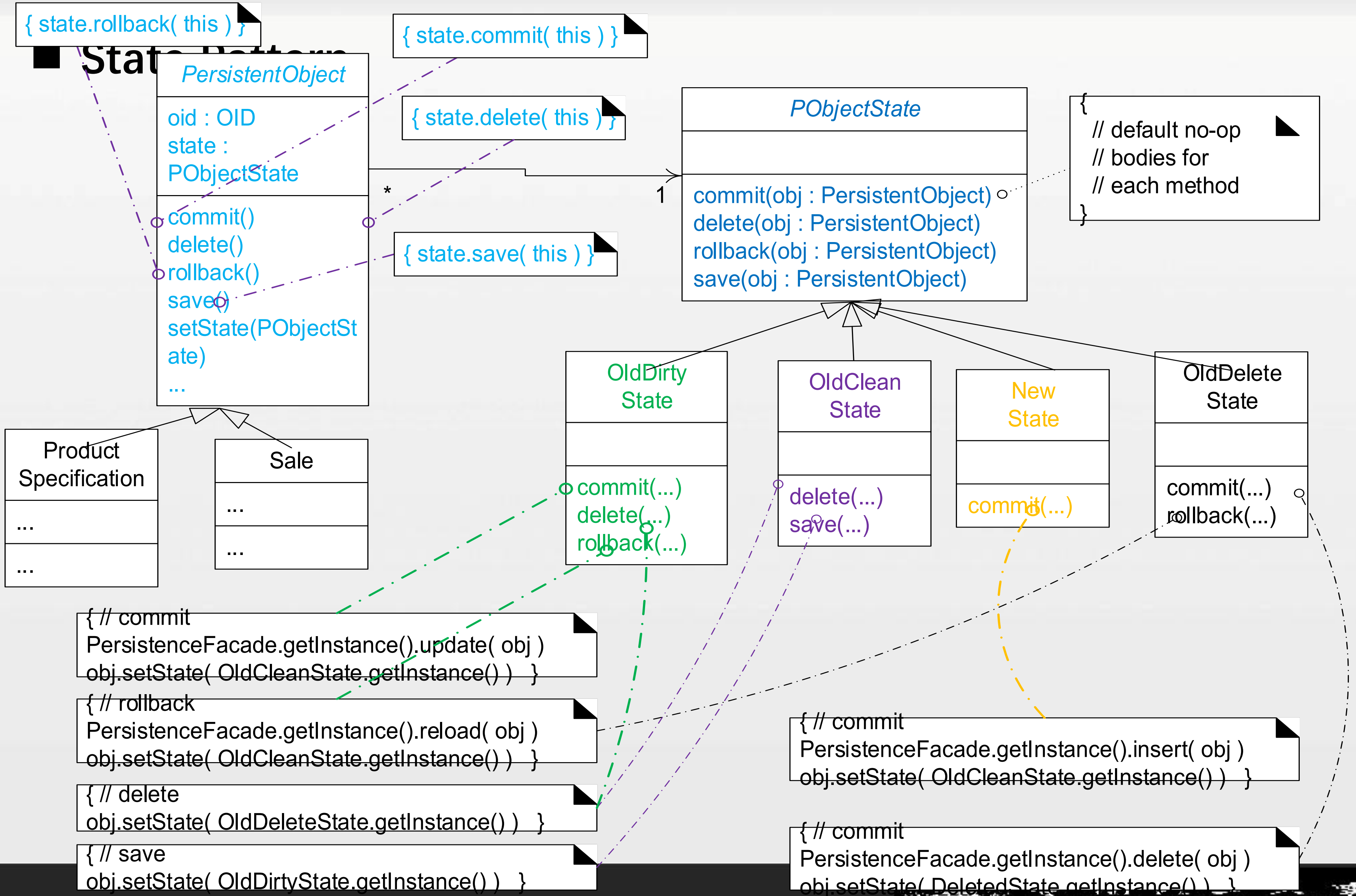
state.doThis()

Client

initialState = new ConcreteState()

Designing a Persistence Framework with Patterns

- **State Pattern**



```
{ state.rollback( this ) }
```

```
{ state.commit( this ) }
```

```
{ state.delete( this ) }
```

```
{ state.save( this ) }
```

```
{
  // default no-op
  // bodies for
  // each method
}
```

```
{ // commit
PersistenceFacade.getInstance().update( obj )
obj.setState( OldCleanState.getInstance() )   }
```

```
{ // rollback
PersistenceFacade.getInstance().reload( obj )
obj.setState( OldCleanState.getInstance() )   }
```

```
{ // delete
obj.setState( OldDeleteState.getInstance() )   }
```

```
{ // save
obj.setState( OldDirtyState.getInstance() )   }
```

```
{ // commit
PersistenceFacade.getInstance().insert( obj )
obj.setState( OldCleanState.getInstance() )   }
```

```
{ // commit
PersistenceFacade.getInstance().delete( obj )
obj.setState( DeletedState.getInstance() )   }
```

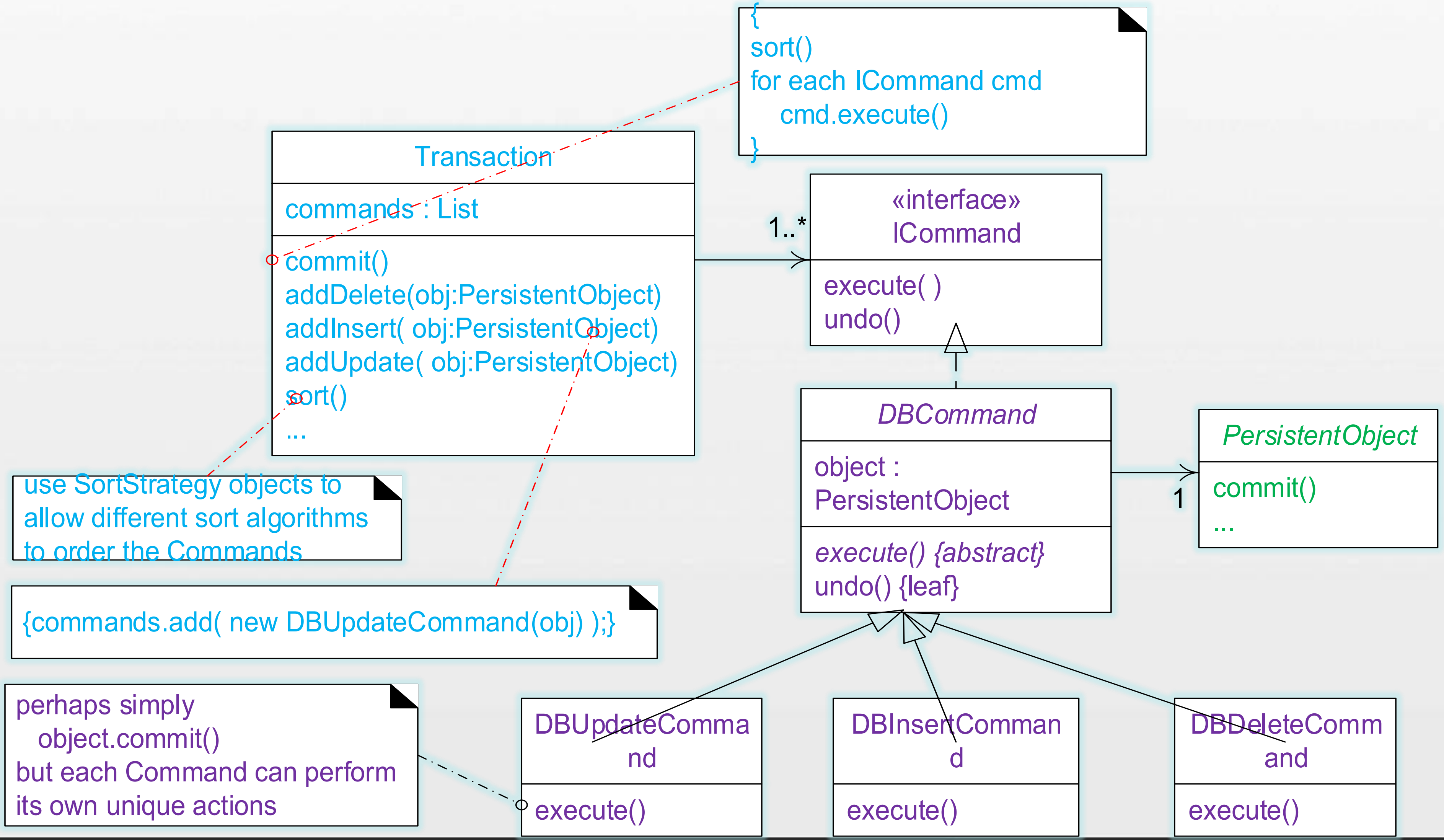
Designing a Persistence Framework with Patterns

- **Designing a Transaction with the Command Pattern**
  - Transaction is atomic, whose tasks must all complete successfully, or none must be completed
  - the order of database tasks within a transaction can influence its success (and performance).

- **Command Pattern**
  - problems
    - How to handle requests or tasks that need functions such as sorting (prioritizing), queueing, delaying, logging, or undoing?
  - Solutions
    - Make each task a class that implements a common interface.

Designing a Persistence Framework with Patterns

- **Designing a Transaction with the Command Pattern**



```
{
sort()
for each ICommand cmd
    cmd.execute()
}
```

Transaction

commands : List

commit()
addDelete(obj:PersistentObject)
addInsert(obj:PersistentObject)
addUpdate(obj:PersistentObject)
sort()
...

1..*

«interface»
ICommand

execute( )
undo()

DBCommand

object : PersistentObject

execute() {abstract}
undo() {leaf}

1

PersistentObject

commit()
...

use SortStrategy objects to allow different sort algorithms to order the Commands

{commands.add(new DBUpdateCommand(obj));}

perhaps simply
  object.commit()
but each Command can perform its own unique actions

DBUpdateCommand

execute()

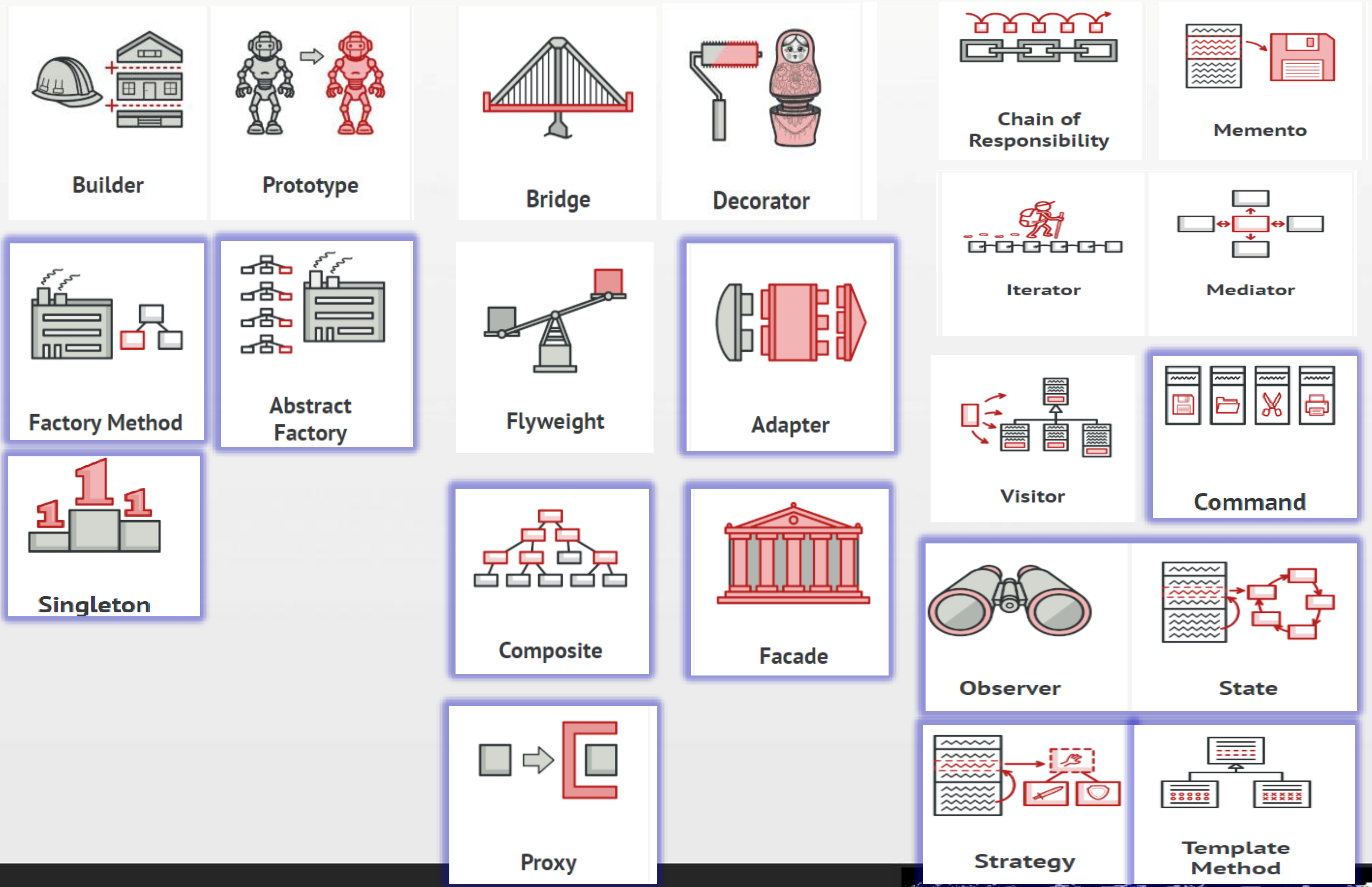
DBInsertCommand

execute()

DBDeleteCommand

execute()

Other Patterns not is this class



Builder

Prototype

Factory Method

Abstract Factory

Singleton

Bridge

Decorator

Flyweight

Adapter

Composite

Facade

Proxy

Chain of Responsibility

Memento

Iterator

Mediator

Visitor

Command

Observer

State

Strategy

Template Method

Eed of Course

The End

Thanks everyone for listening in this special summer